UNIVERSITY OF NICE - SOPHIA ANTIPOLIS

These d'Habilitation a Diriger des Recherches

presented by
Davide BALZAROTTI

Web Applications Security

Submitted in total fulfillment of the requirements
of the degree of Habilitation a Diriger des Recherches

Committee:

Reviewers:	Evangelos Markatos	- University of Crete
	Frank Piessens	- Katholieke Universiteit Leuven
	Frank Kargl	- University of Ulm
Examinators:	Marc Dacier	- Qatar Computing Research Institute
	Roberto Di Pietro	- University of Padova

Acknowledgments

Even though this document is mostly written in first person, every single "I" should be replaced by a very large "We" to pay the due respect to everyone who made this hundred-or-so pages possible.

Let me start with my two mentors, who toke me fresh after my Ph.D. and taught me how to do research and how to supervise students. The work of a professor requires good ideas, but also strong management skills, and a good number of tricks of the trade. Giovanni Vigna, at UCSB, was the first to introduce me to web security and to teach me how to do things right. Engin Kirda, at Eurecom, filled up the missing part – teaching me how to find and manage money and how to supervise other Ph.D. students. A big thank to both of them.

The second group of people that made this possible are the students I was lucky to work with and supervise in the past five years. Giancarlo Pellegrino, Jelena Isacenkova, Davide Canali, Jonas Zaddach, Mariano Graziano, Andrei Costin, and Onur Catakoglu. They are not all represented in this document, but it is their hard work that transformed some of the papers that are part of this dissertation from an idea to a real scientific study.

Third, I would like to thank everyone else I met on this journey - from the office mates in California (Wil, Vika, Marco, Fredrik, Greg, Chris, ...) to all the other colleagues I collaborated with in the past ten years (Aurelien, Andrea, Theodoor, ...). It was an honor to work with you all.

Finally, I big thank to my wife - who supported me and encouraged me to go through all of this.

Contents

1 Introduction

I obtained my Ph.D. from Politecnico di Milano in 2006, working in the area of network intrusion detection. As a postdoc researcher first, and as Assistant Professor later, I then expanded my interests and my research activities to the broader area of system security. After my graduation, I co-authored over 50 scientific articles in international journals and conferences, on binary and malware analysis, web security, reverse engineering, host-based protection mechanisms, botnet detection, embedded system security, and computer forensics. This broad range of topics reflects the fact that security is a cross-cutting aspect that applies to multiple, if not all, areas of computer science. On the one hand, this is what makes this field so challenging and interesting to work on. On the other hand, this variety of topics also makes it difficult to distill my contribution to the system security area in a single coherent document.

Therefore, I decided to focus this dissertation on a single area – which represents, in terms of publications, only a small part of my entire work. Selecting a single topic was very hard, and therefore I decided to base my decision on a purely temporal aspect: the first research project I worked on after I completed my Ph.D. was on web security, and my last paper submitted before completing this dissertation was again on web security. For the nine years between these two events, I always continued to work on this topic both on my own and by supervising other master and Ph.D. students in this area. Therefore, I believe web security, and the security of web applications in particular, successfully captures a line of research that characterized my past activity in system security.

For space reasons, I am not able to present in this document my entire contribution to the web security domain. Instead, I decided to include seven representative papers - grouped in three distinct areas - that summarize some of the problems I faced (and partially solved) after my Ph.D. graduation.

World Wide Web

The World Wide Web (or simply the Web) was initially proposed in 1990 by Tim Berners-Lee and Robert Cailliau as a distributed "web" of interconnected hypertext documents. These static documents were written using a markup language (HTML) and transferred over the network using a dedicated protocol (HTTP). The initial system also required two new pieces of software, one to serve the pages (a web server) and one to retrieve them and render their content on the screen (a web browser).

In few years, this initially simple architecture evolved far beyond imagination. In particular, the majority of web sites are nowadays complex distributed applications, with part of their code running on the server (to dynamically construct the pages content, typically based on information stored in a backend database) and part running in the user browser (to implement the user interface and fetch content on demand from the server). As a result, according to the HttpArchive[1], today almost 50% of the web pages require more than 30 separate connections to fetch all the required elements.

However, since the focus of this document is on the *security* of web applications, I will limit the historical discussion of the Web to three important aspects that, in my opinion, characterized the evolution of this technology from a security perspective.

A Platform for the Masses

In few years from its introduction, the Web rapidly evolved from a client-server system to deliver hypertext documents into a platform to run stateful, asynchronous, distributed applications. On the one hand, this transformation required more sophisticated software components (modern browsers are now comparable to an operating system kernel in terms of lines of code) and a more complex architecture, involving the interaction of dozens of languages and protocols (such as WebDav, XML, Soap, JSON, SSL, OAuth, just to name a few).

On the other hand, the Web was designed since the beginning to be simple for users to use and to deploy new content. In other words, on one side we have a very complex architecture, whose details are difficult to grasp also for experts in the field. On the other side, the same architecture is advertised as a platform for the masses – that even people with little to no experience in software design can use to quickly develop new web sites and applications, with advanced and customizable user interfaces. To make things worse, the extremely fast evolution of Web technologies - often driven by a market pushing for new features - was often affected by design flaws and poor security planning.

This combination had catastrophic security consequences. Hundred of thousands of web sites were created by *web designers*, experts in customizing the visual look-and-feel of an application, but completely unaware of the complexity and

[1] `http://httparchive.org`

risks of the technology they were using. Unfortunately, these web sites have often access to databases and to a large spectrum of sensitive client information. As a result, even simple vulnerabilities such as SQL injections became a plague that gave curios people first, and criminals later, an easy way to steal data and access company networks.

To close the loop, search engines provided a simple way for attackers to find vulnerable targets, so that the entire exploitation process could be easily automated. The combination of all these factors made the Web the target of choice not only for criminals and knowledgeable attackers, but also for a multitude of wanna-be hackers (the so called *script-kiddies*) with little technical background - that by simply using automated tools were able to wreak havoc with many Internet services.

Reachable by Design

If the Web revolution was bad from a software development point of view, it was not much better from a network administrator perspective. Internet was designed to connect networks together, so that every network would be reachable from everywhere else. However, networks were also designed with strict monitoring and access control in mind. For instance, firewalls limited the access to sensitive parts of a company's network, and allowed only a limited number of selected services to be open to the entire world. At the same time, intrusion detection systems carefully monitored those services for signs of attacks or suspicious behaviors.

The web broke this isolation. Usability had priority over security, and developers were fast to port many existing services to the new platform, allowing all sort of traffic to be tunneled through the web server on port 80 (open by default on most of the networks). Technology were re-invented, sometimes intentionally to avoid those security mechanism that made them harder to use otherwise. For instance, *web services* introduced remote procedure calls (RPCs) over the HTTP protocol, way before the security community had time to catch up and develop corresponding security tools to monitor and filter HTTP traffic.

Untrusted Mobile Code

A third aspect of the Web evolution that is important for security is the introduction of client-side code in web pages. When, in 1995, Netscape introduced in its browser the support for Java Applets and for the new Javascript language, it also introduced a method to run arbitrary and untrusted code on user machines.

This gave attackers the opportunity to run arbitrary code in their victims computers, by simply tricking users into visiting a link or by compromising an already existing web page. Naturally, the risk of running unknown code was understood since the beginning and therefore many layers of protection and sandboxing were put in place over the years to avoid this code from performing dangerous operations. Another aspect that was immediately clear was the need to isolate each website and strictly control their interaction with other pages loaded in the same

browser instance. The foundation of this security mechanism lies in the *Same-Origin Policy*, introduced to protect the content of a page from JavaScript code fetched from a different "origin" (defined as the tuple ⟨ protocol,hostname,port ⟩). However, this is only the tip of the iceberg of a very complex mix of technologies and security solutions that are part of a very active research area focusing on client-side (or Browser) security.

Since this dissertation targets instead the security of web applications (therefore more on the server-side of the architecture), it is sufficient to understand that, while many techniques exist to isolate JavaScript code and protect the end users against malicious scripts, this is often not enough. Web browsers are very complex applications and they are often extended by third-party plugins, making their attack surface very large. Therefore it is not unusual for attackers to find loopholes or vulnerabilities to bypass the browser security mechanisms and execute arbitrary code on the end host.

My Contribution in the Area of Web Application Security

My research in the area of web application security can be summarized along three main directions. First of all, it is impossible to work in this area (at least it was in 2005) without studying code injection vulnerabilities due to improper input validation. Driven by insecure libraries and untrained programmers, this has been the first cause of Web-related exploitations for almost a decade. The first part of this document is dedicated to this topic and it introduces the classic SQL injection and Cross Site Scripting (XSS) vulnerabilities but also some less-known input validation flaw such as parameter pollution. I chose three papers that look at the input validation problem from three different angles: one proposing a technique to identify and test validation routines, one looking at reports of classic vulnerabilities to better understand their evolution, and one measurement study that aimed at discovering the prevalence of a certain vulnerability on the Internet. I believe that these three activities (*measuring*, *protecting*, and *understanding*) provide a good summary of my activity in the area.

In the second part of this dissertation I introduce my work on a different class of web vulnerabilities. Together with some colleagues at the University of California - Santa Barbara, I was one of the first researcher to study automated approaches to detect *logic* errors in web applications. Difficult to define and to capture in a model, these vulnerabilities are very subtle and very difficult to find with automated scanners. The reason is that in order to detect a vulnerability in the logic of a Web application, one has to first understand and model the logic of the application itself. In other words, one has to infer what the application should do, and then find ways to subvert this intended behavior. In this second part of the dissertation I selected two papers, one from 2007 to present the beginning of my research in the area, and one in 2014 to show my current activity on the topic. Moreover, they also show a transition over the years between a white-box approach and a black-box solution.

To conclude this dissertation, Part III presents a different approach to Web application security that focuses on better understanding the other players involved in the problem – beyond normal users and developers. Again, I selected two publications to summarize my research activity. The first (Chapter 10) in which I looked at the role of web hosting providers in detecting when a website has been compromised and it now serves malicious content. The second (Chapter 11) presents instead a study of how attackers exploit web applications and, more importantly, what are their goals in this process.

After these three parts, in Chapter 12 I will summarize my contribution in the area and present some future lines of research that will hopefully keep me busy for the next few years.

List of Publications

This is the list of seven papers, sorted according to their publication date, that I selected to include in this document.

D. Balzarotti, M. Cova, V. Felmetsger, G. Vigna
"Multi-Module Vulnerability Analysis of Web-based Applications"
ACM Conference on Computer and Communication Security (ACM CCS) 2007

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna
"Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications"
29th IEEE Symposium on Security and Privacy 2008

Marco Balduzzi, Carment T. Gimenez, Davide Balzarotti, Engin Kirda "Automated discovery of parameter pollution vulnerabilities in web applications"
Network and Distributed System Security Symposium (NDSS) 2011

Theodoor Scholte, Davide Balzarotti, Engin Kirda
"Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications"
Financial Cryptography and Data Security (FC) 2011

Davide Canali, Davide Balzarotti, Aurelien Francillon
"The Role of Web Hosting Providers in Detecting Compromised Websites"
International World Wide Web Conference (WWW) 2013

Davide Canali, Davide Balzarotti
"Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web"
Network and Distributed System Security Symposium (NDSS) 2013

Giancarlo Pellegrino, Davide Balzarotti
"Toward Black-Box Detection of Logic Flaws in Web Applications"
Network and Distributed System Security (NDSS) 2014

Input Validation Vulnerabilities

2 The Evergreens

In the past decade, a large amount of effort has been spent to improve the security of web applications, and a large fraction of this work has focused on mitigating input validation vulnerabilities.

In few words, an input validation vulnerability arises when a web application uses malicious input as part of a sensitive operation, without properly verifying or sanitizing the input values prior to their use. The detection of these vulnerabilities can be performed in a black-box fashion, for instance by using a remote testing tool that probes an application and runs a number of pre-defined security tests. The detection of input validation vulnerabilities can also be performed by looking at the source code, typically by identifying all the paths between a *source* (where the user input enters the system) and a *sink* (where the application outputs data to a backend database or to the user browser).

While the class of input validation vulnerabilities contains many different types of flaws, SQL injection and Cross Site Scripting (XSS) are certainly the two most common examples. I call these vulnerabilities the evergreens, because (despite being simple in their formulation and often easy to avoid) they have been a major source of Internet attacks for well over a decade. SQL injections are to the web world what buffer overflows are to binary software. Both vulnerabilities have a very long history, but while exploiting buffer overflows is becoming more and more difficult every year (thanks to an endless number of compiler and operating system defenses), SQL injections in their simpler form are still a very severe issue today.

In this chapter I look at input validations vulnerabilities from three different angles: by looking more closely to the validation and sanitation problem, by looking for changes in the evolution of these vulnerabilities, and by showing that this class contains examples that are still largely unknown and that affect a large number of popular websites on the Internet.

Summary

Past research on vulnerability analysis was mostly focused on identifying cases in which a web application directly uses external input in critical operations. In other words, to keep things simple, most of the existing approaches tried to distinguish two cases: those execution paths in which the user input was sanitized, and those in which it did not.

However, little research had been done to analyze the correctness of the sanitization process. Unfortunately, the fact that a developer dedicated some code to filter and sanitize potentially malicious input says nothing about the quality of that code and about the fact that is both correct and complete. The first paper presented in this part introduces a novel approach to the analysis of the sanitization process. More precisely, in our solution we combined static and dynamic analysis techniques to identify faulty sanitization procedures that can be bypassed by an attacker. We implemented our approach in a tool, called Saner, that we used to test a number of real-world applications. In our experiments we were able to identify several novel vulnerabilities that stem from erroneous sanitization procedures.

In the second paper I look back at the evolution of SQL injection and XSS vulnerabilities - trying to find an answer to three important questions:

- Do attacks become more sophisticated over time?
  (short answer is "*no*")
- Do well-known and popular applications become less vulnerable over time?
  (short answer is "*partially*")
- Do the most affected applications become more secure over time?
  (short answer is "*at least regarding their initial vulnerabilities*")

Unfortunately, years of research have failed to eradicate this problem, and even to considerably increase the bar for the attackers.

I conclude this part of the dissertation with a paper that received the Distinguished Paper Award at NDSS 2011. The goal of this work was to perform a large black-box study of a new class of vulnerabilities called parameter pollution. We did not invent this class, but we show that since most of the developers were not aware of this particular risk, they failed to sanitize the user input appropriately. If the previous paper showed that developers still make mistakes also for flaws that are well understood, this paper shows how bad the overall picture is when we look at less famous vulnerabilities that did not receive as much attention.

We developed a free service to test for this class of vulnerabilities and we conducted experiments on over 5,000 popular websites. Our results show that about 30% of them contained vulnerable parameters (including several highprofile websites such as Symantec, Google, VMWare, and Microsoft) and that at least 14% of them could have been remotely exploited.

Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications

Davide Balzarotti[§], Marco Cova[§], Vika Felmetsger[§], Nenad Jovanovic[*], Engin Kirda[¶], Christopher Kruegel[§], and Giovanni Vigna[§]

[§] University of California, Santa Barbara
{balzarot,marco,rusvika,chris,vigna}@cs.ucsb.edu

[*] Secure Systems Lab
Technical University Vienna
enji@seclab.tuwien.ac.at

[¶] Institute Eurecom
France
engin.kirda@eurecom.fr

Abstract

Web applications are ubiquitous, perform mission-critical tasks, and handle sensitive user data. Unfortunately, web applications are often implemented by developers with limited security skills, and, as a result, they contain vulnerabilities. Most of these vulnerabilities stem from the lack of input validation. That is, web applications use malicious input as part of a sensitive operation, without having properly checked or sanitized the input values prior to their use.

Past research on vulnerability analysis has mostly focused on identifying cases in which a web application directly uses external input in critical operations. However, little research has been performed to analyze the correctness of the sanitization process. Thus, whenever a web application applies some sanitization routine to potentially malicious input, the vulnerability analysis assumes that the result is innocuous. Unfortunately, this might not be the case, as the sanitization process itself could be incorrect or incomplete.

In this paper, we present a novel approach to the analysis of the sanitization process. More precisely, we combine static and dynamic analysis techniques to identify faulty sanitization procedures that can be bypassed by an attacker. We implemented our approach in a tool, called Saner, and we applied it to a number of real-world applications. Our results demonstrate that we were able to identify several novel vulnerabilities that stem from erroneous sanitization procedures.

1 Introduction

Web applications have evolved from simple CGI-based gateways that provide access to back-end databases into full-fledged, complex applications. Such applications (e.g., email readers, web portals, or e-commerce front-ends) are developed using a number of different technologies and frameworks, such as ASP.NET [21] or PHP [32]. Even though these technologies provide a number of mechanisms to protect an application from attacks, the security of web applications ultimately rests in the hands of the programmers. Unfortunately, these programmers are often under time-to-market pressure and not always aware of the available protection mechanisms and their correct usage. As a result, web applications are riddled with security flaws that can be exploited to circumvent authentication, bypass authorization checks, or access sensitive user information. A report published by Symantec in March 2007 states that, out of the 2,526 vulnerabilities that were documented in the second half of 2006, 66% affected web applications [42].

One of the most common sources of vulnerabilities is the lack of proper validation of the parameters that are passed by the client to the web application. In fact, OWASP's *Top Ten Project*, which lists the top ten sources of vulnerabilities in web applications, puts unvalidated input as the number one cause of vulnerabilities in web applications [30]. Input validation is a generic security procedure, where an application ensures that the input received from an external source (e.g., a user) is valid and meaningful. For example, an application might check that the number of items to purchase, sent as part of a form submission, is actually provided as an integer value and not as a non-numeric string or a float. As another example, an application might need to ensure that the message submitted to a bulletin board does not exceed a certain length or does not contain JavaScript code. Also, a program typically has to enforce that arguments to database queries do not contain elements that alter the intended meaning of these queries, leading to SQL injection attacks.

A particular type of input validation is *sanitization*. In general, sanitization is performed to remove possibly malicious elements from the input. Section 2 introduces more examples of how sanitization is performed in web applications. At this point, it suffices to say that sanitization is performed on external input parameters *before* they are used in critical operations. The lack of sanitization can introduce vulnerabilities (e.g., cross-site scripting (XSS) [20] and SQL injection [1,39] flaws) that can be exploited by attackers. A number of past research efforts [9, 13, 17, 18, 22, 45] have focused on the problem of identifying vulnerabilities in which external input is used without any prior sanitization. These vulnerability detectors are typically based on some form of data flow analysis that tracks the flow of information from the application's inputs (called *sources*) to points in the program that represent security-relevant operations (called *sinks*). The underlying assumption of these approaches is that if a sanitization operation is performed on all paths from sources to sinks, then the application is secure.

Interestingly, there has been little research to precisely model how effective the sanitization process actually is. In fact, most approaches assume that if a regular expression or a certain, built-in sanitization function is applied to an external input, then the result is safe to use. Unfortunately, this is not always the case. For example, a regular expression could be used by a programmer to check for the occurrence of certain values in the input without any intention to perform sanitization. Also, it is possible to apply a sanitization function to the input that protects from certain malicious values, but does not offer complete protection from all attacks. For example, in [41], the authors discuss the possibility of subtle SQL injection vulnerabilities that can be exploited even when the input has been processed by a built-in PHP sanitization routine. Sanitization is particularly dangerous when custom checking routines are used. In these cases, a programmer does not rely on built-in input validation functions, but, instead, manually specifies a list of unwanted characters or a regular expression that should remove malicious content.

In this paper, we introduce a novel approach to analyze the correctness of the sanitization process. The approach combines two complementary techniques to model the sanitization process and to verify its thoroughness. More precisely, a first technique based on static analysis models how an application modifies its inputs along the paths to a sink, using precise modeling of string manipulation routines. This approach uses a conservative model of string operations, which might lead to false positives. Therefore, we devised a second technique based on dynamic analysis. This approach works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process.

We implemented our techniques in a system called Saner, a prototype that analyzes PHP applications. The choice of PHP was driven by the fact that PHP is one of the most popular languages for web application development. According to the latest Netcraft survey [27], more than 20 million sites were using PHP in June 2007. In the monthly Security Space Reports [36], PHP has constantly been rated as the most popular Apache module over the last few years. To evaluate our system, we used Saner on a set of real-world applications. The results show that the sanitization process is faulty in a number of cases and that apparently effective sanitization routines can be bypassed to exploit the applications.

The contributions of this paper are the following:

- We describe a static analysis technique that characterizes the sanitization process by modeling the way in which an application processes input values. This allows us to identify cases where the sanitization is incorrect or incomplete.
- We introduce a dynamic analysis technique that is able to reconstruct the code that is responsible for the sanitization of application inputs, and then execute this code on malicious inputs to identify faulty sanitization procedures.
- We compose the two techniques to leverage their advantages and mitigate their disadvantages.
- We implemented our approach and evaluated the system on a set of real-world applications. During our experiments, we identified a number of previously unknown vulnerabilities in the sanitization routines of the analyzed programs.

The rest of the paper is structured as follows. In Section 2, we provide an example of the type of errors in the sanitization process that we are interested in identifying. In Section 3, we present our techniques for the analysis of the sanitization process in web applications. Then, in Section 4, we describe a prototype implementation of our approach and the results of its evaluation on real-world applications. Section 5 presents related work. Finally, Section 6 concludes and outlines future work.

2 Motivation

In this section, we discuss in more detail the ways in which web applications can perform input validation. This discussion also helps to establish a more precise notion of sanitization. Then, we provide an example of a custom sanitization routine that is typically not handled by static vulnerability detectors. This demonstrates the need for an improved analysis process and serves as an underlying motivation for our work.

2.1 Input Validation and Sanitization

Web applications typically work by first reading some input from the environment (either provided directly by

a user or by another program), then processing this data, and finally outputting the results. As previously stated, the program locations where input enters the application are referred to as sources. The locations where this input is used are called sinks. Of course, sources often take data directly from potentially malicious users, and the application can make little (or no) assumptions about the values that are supplied. Unfortunately, many types of sinks cannot process arbitrary values, and security problems may arise when specially crafted input is passed to these sinks. We refer to these sinks as *sensitive* sinks.

An example of a sensitive sink is a SQL function that accesses the database. When a malicious user is able to supply unrestricted input to this function, she might be able to modify the contents of the database in unintended ways or extract private information that she is not supposed to access. This security problem is usually referred to as a SQL injection vulnerability [1]. Another example of a sensitive sink is a function that sends some data back to the user. In this case, an attacker could leverage the possibility to send arbitrary data to a user to inject malicious JavaScript code, which is later executed by the browser that consumes the output. This problem is commonly known as an XSS vulnerability [20].

To avoid security problems, an application has to ensure that all sensitive sinks receive arguments that are well-formed, according to some specification that depends on the concrete type of the sink. Because input from potentially malicious users can assume arbitrary values, the program has to properly validate this input. Therefore, the application checks the input for values that violate the specification. When such invalid values are found, a programmer has two options. The first option is to abort further processing: the application stops to handle the request and returns an error code to signal incorrect input. The second option is to transform the input value such that the altered value conforms to the input specification and no longer poses a security threat when passed to a sensitive sink. We denote the process of transforming the input to a representation that is no longer dangerous as *sanitization*. Typically, sanitization involves the removal of (meta)-characters that have a special meaning in the context of the sink, escaping these characters, or truncating the length of the input.

2.2 Static Analysis and Proper Sanitization

Static analysis tools that check the security of (web) applications often employ data flow analysis to track the use of program inputs. The goal of these systems is to identify program paths between the location where an input enters the application and a location where this input is used. Once such a program path is identified, the tool checks whether the programmer has properly sanitized the input on its way from the source to the sensitive sink. When input is properly sanitized on all paths from an input source to a sensitive sink, the application is correct and does not contain a security vulnerability. However, it is unfortunately not immediately obvious when to declare input as properly sanitized.

The first problem is that the input sanitization depends on the type of sink that consumes the input. For example, when an attacker can inject SQL commands into the output that the application sends back to a user, this application is not vulnerable. A security problem arises only when the attacker can inject that same input into a function that accesses the database. As a result, static analysis tools typically require a policy that specifies for each type of sensitive sinks (such as database access or output functions) the set of operations that constitute proper sanitization.

The second problem that makes it hard to ascertain proper sanitization is the difficulty of specifying all sanitization operations *a priori*. Fortunately, many languages provide built-in functions that sanitize input. For example, the PHP function `htmlentities` converts characters that have special meaning in HTML into their corresponding HTML entities (e.g., the character '<' is converted into '`<`'). This ensures that all characters in a string preserve their meanings when interpreted as HTML. The PHP manual states that this function is useful "in preventing user-supplied text from containing HTML markup." Applying `htmlentities` to an input string ensures that the resulting string can be safely sent back to a user. The reason is that all script tags in the input (such as "`<script>`") are converted into tokens that are no longer interpreted by a browser as the start of JavaScript code, but simply displayed as the string "`<script>`." Of course, it is easy to recognize the use of such functions as proper sanitization.

Besides the use of built-in sanitization functions, a programmer can also write *custom code* that strips dangerous characters from an input string. For example, the programmer could apply the PHP function `str_replace` to the input string and remove all occurrences of the character '<' (more precisely, to replace all occurrences of the angle bracket character with the empty string). In this case, the result would also be safe with respect to XSS and could be sent back to the user. To see the difference between standard and custom sanitization, consider the example code in Figure 1. Of course, the mere fact that the programmer applies a string replacement operation on an input value does not ensure that the result is properly sanitized.

```
1 $input = _GET['x'];
2
3 $standard = htmlentities($input);
4 $standard = 'Hello ' . $standard;
5 echo $standard;
6
7 $custom = str_replace('<', '', $input);
8 $custom = 'Hello ' . $custom;
9 echo $custom;
```

Figure 1. Standard and custom sanitization.

When analyzing the code in Figure 1, most static analysis tools would correctly flag the use of variable

`$standard` in Line 5 as safe. The reason is that they are typically equipped with a policy that specifies that all values processed by `htmlentities` can be safely echoed back to a user. The situation is more complicated when analyzing the use of the function `str_replace`, which performs custom sanitization in this case. In principle, static analyzers could interpret the application of any function that modifies an input (e.g., through string replacement) as an indication that the programmer performed sanitization. If this strategy is used, the analysis would correctly consider the application of the `str_replace` function on Line 7 as a form of sanitization. Of course, this approach suffers from two drawbacks. First of all, the programmer might have simply applied this function to alter the string based on some requirement implied by the application logic, and changing the string does not imply that the result is safe to be used by a sensitive sink. Second, the programmer could have made a mistake. Even when the input is modified with the intention to make it safe, there is no guarantee that the result is correct (and our results demonstrate that programmers do make frequent mistakes when using custom sanitization routines). Also, the opposite strategy of assuming that all custom sanitization operations are incorrect is problematic, because it causes static analysis tools to report incorrect warnings in case a programmer has correctly applied custom sanitization.

Current static analysis systems (see Section 5 for a detailed discussion of related work) typically disregard the use of custom sanitization routines. The result is that whenever a programmer makes use of custom sanitization, these tools report an error. This requires a tedious, manual inspection of the false positives. Of course, once a sanitization routine has been manually examined and annotated as safe, more powerful static analysis tools will honor this annotation and no longer report false positives. Unfortunately, programmers are often unlikely to spot the application of incorrect custom sanitization. The reason for this is that programmers *expect* the static analysis tools to report an error in association with their custom sanitization routines, and, therefore, there is little need to double-check them. This is dangerous, as we have found several instances in a number of real-world programs in which custom sanitization was used incorrectly.

To address the shortcomings of current analysis tools, we propose a technique that can handle the use of custom sanitization routines and properly track the effect of functions that manipulate and modify program input. The goal is to model the effect of sanitization routines so that we can check, for every sensitive sink, whether the input that can reach these sinks might contain malicious values. This solves two problems. First, we can reduce the number of false positives produced by current static analysis tools by taking into account correct sanitization. Second, we can identify incorrect sanitization routines and alert the programmer when she has made a mistake.

3 Approach

The goal of Saner is to analyze the use of custom sanitization routines to identify possible XSS and SQL injection vulnerabilities in web applications. In the context of our work, any function that takes as input a (string) value and that can output a modified version of this input is considered a possible sanitization routine. In particular, this includes functions that replace or remove certain characters or substrings from their input (such as the PHP functions `str_replace` or `eregi_replace`). As mentioned previously, this requires our system to model the ways in which these functions can modify the application's input. To this end, we use a combination of static and dynamic program analysis techniques.

The core of the approach consists of a static analysis component that uses data flow techniques to identify the flows of input values from sources to sensitive sinks. This component is based on the open-source web vulnerability scanner called Pixy [17, 18]. In its current form, Pixy only provides information about the presence of data flows between sources and sinks. In addition, it can determine whether built-in sanitization operations (such as `htmlentities`) are applied on all paths between a source and a sink. To achieve this, it is sufficient to assign one of two types (or labels) to each program variable: *tainted* or *untainted*. Whenever input is read from a user and stored in a variable, the variable initially receives the label *tainted*. Once a variable is sanitized, its label is set to *untainted*. Whenever a *tainted* variable is used in a sensitive sink, an error is signaled. Unfortunately, this simple approach cannot model the effect of sanitization routines, as a variable can only be *tainted* or *untainted*, and the tool cannot capture the set of values that the variable can hold. To address this problem, we have extended Pixy to derive an over-approximation of the set of (string) values that each program variable can hold. This calculation is done for every point in the program. For each sensitive sink, we can then check whether this value set contains any element that poses a security risk when used at that sink.

The static analysis component is sound with respect to the supported language features[1]. That is, whenever the static analysis component declares a sanitization operation to be correct, we are certain that there exists no vulnerability. The drawback of this approach is that the system might produce false positives (i.e., not every reported problem is an actual vulnerability). Because the number of false positives can be large (depending on the application), we augment the static analysis with an additional dynamic analysis phase.

The goal of the dynamic phase is to examine all those program paths from input sources to sensitive sinks that the static analysis has identified as suspicious. More precisely, using dynamic analysis, we attempt to confirm the existence of a potential security vulnerability (reported by

[1]Most notably, our analysis does not support the `eval` function and certain cases of aliased array elements.

the static analysis phase) by finding program inputs that can bypass the sanitization routines and reach the sensitive sink. To this end, the dynamic analysis is used to simulate the effect of the program operations on the input while it is propagated to the sensitive sink (in particular, sanitization operations are of interest). Of course, the analysis is performed by exercising the code with a large set of different input values, which contain many different ways of encoding and hiding malicious characters. In some sense, the dynamic analysis phase automates the actions of a programmer when a static analysis tool reports a warning. Similar to our dynamic phase, the programmer would first identify the operations that are applied to an input on the path from the source to the sink. Then, using a number of test cases, she would attempt to understand whether one of these inputs could lead to a security violation.

Whenever the dynamic analysis phase determines that a malicious value can reach a sensitive sink, this input is reported as a concrete example that violates the security of the application. If no such input can be found, there are two options. The first option is to assume that the static analysis phase has incorrectly flagged a correct sanitization routine as suspicious. This sacrifices soundness because the dynamic analysis might miss a true vulnerability, but it is convenient as no further manual inspection is required. The second option is to report confirmed vulnerabilities with higher confidence, and to yield the remaining warnings to the programmer.

3.1 Sanitization-Aware Static Analysis

As mentioned in Section 2, existing static analysis approaches simply "guess" whether a custom sanitization routine is effective or not. A sound analysis would regard all types of custom sanitization as ineffective, which typically leads to many false positives. An unsound analysis would assume that custom sanitization is always correct, which may result in missed vulnerabilities. Pixy, the analysis tool that we build upon, follows a sound approach. Hence, our first goal is to improve the existing static analysis so that it is able to assess the effectiveness of custom sanitization routines. As a result, whenever our improved analysis verifies that a custom sanitization is correct, it has essentially suppressed a false positive that (the original) Pixy would have reported. To achieve this goal, we present a technique that leverages *transducer-based, implicit taint propagation*. Once the static analysis is finished, our second objective is to provide the subsequent dynamic analysis phase with appropriate information that allows for further inspection of all suspicious sanitization procedures.

3.1.1 Basic String Automata

As a first step for modeling sanitization routines, we require information about the *set of values* that different program variables may hold, not only the information whether they are tainted or not. To this end, we employ an analysis that can approximate the string values that certain variables might hold at certain program points, using finite automata. Commonly, automata are used as acceptors. That is, they are applied for deciding whether string values belong to a certain language. For our purposes, we make use of another property of automata (or, equivalently, regular expressions), namely the ability to describe an arbitrary set of strings.

In our automata representation, every edge denotes a single-character transition. Note that the label $\langle . \rangle$ stands for an arbitrary character. In addition to the characters that make up a string value, automata also need to be able to encode information about the taint status of these stings. That is, we want to be able to express that certain parts of a string value are tainted, whereas other parts are untainted. This allows us to reconstruct which parts of the strings are possibly derived from (malicious) user input, and which parts stem from static strings embedded in the program source code. This property is achieved by associating taint qualifiers to the transitions of the automata. An edge can either represent a tainted character (represented by a *dotted* line) or an untainted character (represented by a *solid* line).

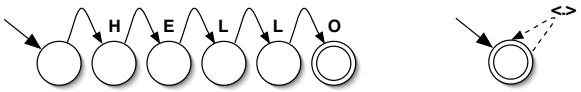


Figure 2. Automata for the string "Hello", and for an unknown, tainted string.

As example, two automata are shown in Figure 2. The left automaton represents the static string value "`Hello`", and hence, contains a series of transitions labeled with the individual characters of this string. Since the string derives from a static literal provided by the programmer, it is considered to be untainted, which is represented by solid transitions. In contrast, the automaton on the right side could represent the value set for variable `$_GET['x']`. Because this value is user-supplied and not known until runtime, the automaton describes the set of all possible strings. The dotted transition indicates that the value is tainted.

Dependence Graphs. To compute the set of string values that a variable at a certain program point can hold, we leverage Pixy's dependence analysis. Dependence analysis is a data flow analysis that computes a dependence graph for every program point (and each variable). Such dependence graphs provide reaching definitions for a particular program point. Intuitively, this means that a dependence graph provides a list of all variables (program points) that might directly influence (or reach) the current program point. As an example, consider the dependence graph in Figure 3. This graph reflects the dependencies for variable `$custom` on Line 9 in Figure 1.

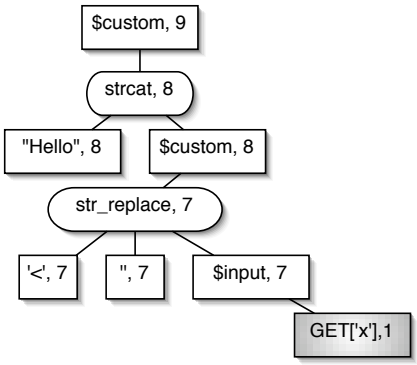


Figure 3. Example dependence graph.

```
1  decorate(Node n) {
2    decorate all successors of n;
3    if n is a string node:
4      decorate n with an automaton for this string
5    else if n is an <input> node:
6      decorate n depending on type of input
7    else if n is an operation node:
8      simulate the operation's semantics
9    else if n is a variable node:
10     decorate n with the union of n's successor
11     automata
12   else if n is a SCC node:
13     decorate n with a star automaton
14     (the taint value of its transition depends
15     on the successor nodes)
16 }
```

Figure 4. Dependence graph decoration algorithm.

Computing Automata. Assume for the moment that a dependence graph for a certain variable (at a particular program point) does not contain cycles. In this case, we can compute the automaton for this variable by applying the algorithm shown in Figure 4. This algorithm takes the root of the dependence graph as input, and recursively processes all nodes of the graph in a postorder traversal. During this traversal, each processed node is associated (*decorated*) with a separate automaton. Each of these automata describes the possible string values of the corresponding node. For computing such an automaton, the automata of all successor nodes are required as input, which explains the postorder traversal. Once all successors of a node have been successfully decorated, the way how the current node is decorated depends on the type of this node.

In the simplest case, the current node represents a string literal. Such nodes are simply decorated with an automaton that describes exactly this string. For a program ⟨`input`⟩ node (shown shaded in Figure 3), it is necessary to analyze the type of this input. If the node represents a variable whose value is taken from the user, such as `$_GET['x']`, the automaton shown on the right in Figure 2 is used for decoration. This automaton represents the set of all possible strings (we will refer to such an automaton as "star automaton" from now on). Its sole transition is tainted to reflect the fact that the user input can be malicious. If the input cannot be directly controlled by a remote user, the transition would be untainted.

If the node to be processed is an operation node (that is, a call to a built-in function), then the semantic of this operation has to be simulated. In case of a string concatenation operation (represented as `strcat`), this is simply done by concatenating the automata of the successor nodes. All other operations are divided into two groups (and our system is equipped with a list that assigns built-in functions to one of these categories). The first group contains functions that are precisely modeled. That is, our system is able to compute an automaton that describes all possible output strings, even when the input parameters to the function are not concrete string instances but automata as well. This is realized with the help of *transducers*, which are described in more detail in the following Section 3.1.2. We have developed a number of transducers for functions that manipulate strings (such as `str_replace`) as well as functions that are commonly used for input sanitization (such as `html_entities`). This is essential to be able to precisely capture the effect of sanitization routines.

The second group of operations contains functions that are not modeled. In this case, we resort to a conservative approximation and assume that each function returns the set of all possible strings (represented as the star automaton). The taint status of the automaton's transition depends on the taint status of the function's actual parameters. To be more precise, the taint status for such functions is the *least upper bound* over the taint status of their parameters. That is, if any parameter is tainted, then the return value is tainted as well. Otherwise, the return value is not tainted.

The next case in the algorithm of Figure 4 (Line 9) applies if the current node is a node representing a variable. In a dependence graph, the successor nodes of a variable node represent the values that this variable *may* possess. Different successor nodes correspond to different paths through the program. This fact can be translated into an automaton by creating the union of the successor nodes' automata. For example, if a variable `$a` depends on the two string literals "`b`" and "`c`", it means that `$a` can hold one of these two strings at runtime. An automaton that encodes this information is created by computing the union of the two automata that represent "`b`" and "`c`", respectively.

Cyclic Dependence Graphs. The previously described algorithm for transforming dependence graphs into automata is not directly applicable to graphs that contain cycles. In general, the precise modeling of cyclic string operations is a difficult problem [4, 24]. Our solution is to replace strongly connected components (SCCs) in the dependence graph with special SCC nodes, which results in a dependence graph without cycles. This explains the final Lines 12 to 15 of our decoration algorithm in Figure 4. Here, SCC nodes are treated analogously to built-in functions that are not modeled. That is, they are decorated with a star automaton, and the taint value of this automaton's

transition is given by the taint values of the SCC node's successors.

Discussion. In general, strings can be created or modified by one of the following three methods: the use of string literals (e.g., `$s = 'ab'`), the concatenation of two strings, or the use of a built-in function. The use of the first two methods, string literals and string concatenation, are always handled precisely by our algorithm. This is also true for built-in functions that are modeled by transducers. Built-in functions that are not explicitly modeled are conservatively approximated with a star automaton. As a result, our analysis either computes precise results, or a safe approximation of the actual result. The only exception is that we do not handle the manipulation of strings through indexing. For instance, it is possible to change the value of the third character of some string variable `$s` through an assignment to `$s{2}`. While this technique for modifying strings is common in C programs, it occurs rarely in PHP applications. In fact, all applications that we evaluated for this paper did not make use of index-based string modifications.

3.1.2 Precise Function Modeling

As mentioned previously, for the analysis of custom sanitization, it is necessary to introduce a precise modeling of string-modifying functions (such as `str_replace`) and replacement functions using regular expressions (`ereg_replace` and `preg_replace`). A suitable algorithm was presented in the natural language processing community by Mohri and Sproat [25]. This algorithm is based on the use of *finite state transducers*. A transducer is an automaton whose transitions are associated with output symbols. This way, it is not only able to accept (or reject) input strings, but it also produces output for each input string.

For example, when using Mohri and Sproat's algorithm to analyze the string operations on Lines 7 and 8 in Figure 1, we obtain the automaton shown in Figure 5. This automaton precisely captures the possible values of the variable `$custom`. That is, it describes the set of strings that start with the prefix "`Hello`", and end with a suffix that does not contain the '<' character. Unfortunately, the computed automaton does not distinguish between tainted and untainted transitions anymore. Instead, it simply assumes all transitions to be untainted. This is because Mohri and Sproat's algorithm is not designed to work on taint-aware automata. We will present a solution to this problem later in this section.

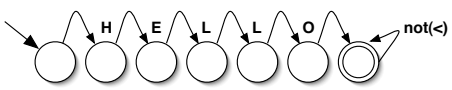


Figure 5. Automaton after replacement of '<'.

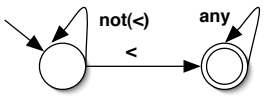


Figure 6. Example target automaton for XSS.

3.1.3 Vulnerability Detection Through Intersection

To check whether a program is vulnerable at some sensitive sink (even when sanitization routines are previously applied), it is necessary to determine whether it is possible that the input to this sensitive sink contains any malicious characters or strings. For instance, an XSS attack typically requires characters such as '<' to be present, as they are needed to construct JavaScript or HTML code. In our approach, we verify this requirement by intersecting the automaton that represents the sink's input with an automaton that encodes the set of undesired strings (the *target automaton*). If the automaton that results from this intersection is empty[2], it means that none of the undesired strings can be contained in the input, and that this sink is safe.

We are aware of the fact that certain XSS attacks do not require the attacker to inject a '<' character into the program output. For example, if an application lets a user modify HTML tags (such as CCS properties or fonts), an attacker could inject script handlers into tag attributes. In such cases, the automaton that captures such attacks will need to be more complex. This can be done without requiring any modifications to our basic technique. Moreover, by checking for the presence of the '<' character, our system already covers a significant fraction of existing XSS threats.

A simple example automaton that represents a conservative approximation of the undesired strings with respect to XSS is shown in Figure 6. This target automaton represents all strings that contain at least one '<' character. Intersecting this automaton with the automaton from Figure 5 yields an empty automaton, which means that this input cannot be used to successfully perform an attack. By doing this, we have successfully determined that the applied custom sanitization was effective. In contrast, the intersection of the target automaton with the automaton that represents the potentially dangerous value of variable `$_GET['x']` (right side of Figure 2) is non-empty, since the unknown value might contain an arbitrary number of '<' characters.

3.1.4 Implicit Taint Propagation

Unfortunately, the techniques for function modeling and vulnerability detection described above are still lacking an important ingredient for reaching a sufficient level of precision. The reason is that the algorithm of Mohri and Sproat

[2] To be precise: If the resulting automaton accepts only the empty language.

does not operate on taint-aware automata, but, instead, on traditional automata without taint qualifiers associated to their transitions. That is, the algorithm is not able to propagate taint values through the modeled functions. Without additional measures, this information loss would lead to false positives, as taint information is essential for vulnerability detection. For instance, the simple example code depicted in Figure 7 below would result in a false positive.

```
$s = "Hello\n";
$x = str_replace("\n", '<br/>', $s);
echo $x;
```

Figure 7. Code that causes a false positive.

```
$s = 'a';
$x = str_replace('a', $_GET['x'], $s);
echo $x;
```

Figure 8. Code that could cause a false negative.

In this code, all occurrences of the '\n' control character are replaced with an HTML line break. Intersecting the automaton that is computed for $x with the XSS target automaton would yield a non-empty result, since $x does contains a '<' character. As a consequence, our analysis would report a vulnerability for this example.

A possible approach to solve the problem of propagating taint values through custom sanitization functions would be to modify Mohri and Sproat's algorithm such that it becomes taint-aware. This modification would ensure that the algorithm accepts taint-aware automata as input (i.e., the arguments of the modeled sanitization function), and returns a taint-aware automaton as output. However, we propose an alternative solution that is sound, efficient, less complex, and less error-prone than a modification of the existing algorithm. Instead of explicitly keeping track of both tainted and untainted values, we concentrate our attention on the tainted parts of the automata. In this *implicit taint propagation*, strings that are statically embedded into the application by the programmer (and hence, untainted) are replaced by the empty string during the automata computation. This has the effect that only tainted strings are explicitly encoded in the automata, and that static, untainted strings can no longer lead to false positives.

If used without care, however, implicit taint propagation can cause false negatives in certain cases (i.e., vulnerabilities might be missed). Consider the (rather contrived) example in Figure 8. Here, the program replaces the character 'a' inside the string $s with a user-provided value (taken from $_GET['x']). If our analysis would propagate taint values implicitly (and replace the value of $s with the empty string), it would incorrectly deduce that the str_replace operation results in the empty string as well. Under the XSS target automaton defined above, an empty string is benign, and, therefore, the vulnerability would be missed. To ensure soundness, it is necessary to compensate the information loss due to the implicit taint propagation with a supplementary "safety net." This additional mechanism corresponds to checking whether the second parameter of str_replace (or, analogously, the replacement parameter of similar functions) is tainted. If the parameter is tainted, the result of the function invocation is conservatively approximated with the automaton that describes the set of all possible strings. This ensures that implicit taint propagation does not introduce false negatives.

3.1.5 Providing Information to Dynamic Analysis

The dynamic analysis that follows the static analysis phase is focused on the detection of routines that perform insufficient custom sanitization. Hence, it only requires information about those vulnerabilities reported by the static analysis process that actually involve the use of custom sanitization routines. For instance, consider the following example code:

```
1 $x = str_replace('<script>', '', $_GET['x']);
2 echo $x;
```

In this program, the variable $_GET['x'] is insufficiently sanitized. The programmer attempted to remove all script tags from the input. Unfortunately, this simple sanitization technique can be easily circumvented by embedding JavaScript code in HTML event handlers (such as onload), which do not require the use of script tags. When checking this code, the dynamic analysis should be informed that there is a possible vulnerability due to insufficient sanitization, and that this vulnerability involves the user-controlled variable $_GET['x'] as source and the echo statement on Line 2 as sink. That is, the dynamic analysis is provided with *source-sink pairs* that describe possible vulnerabilities due to insufficient custom sanitization. This information is extracted from the dependence graphs that static analysis uses internally by means of a simple reachability computation.

Recall that the focus of this paper is on the analysis of custom sanitization routines. As a consequence, we do not provide dynamic analysis with information about vulnerabilities that do not involve any custom sanitization. Instead, these vulnerabilities are immediately reported to the user.

3.2 Testing Sanitization Routines

The static analysis phase is conservative, and, therefore, it may generate false positives, which a developer needs to manually assess. This process, however, is tedious and error-prone. The goal of the dynamic analysis (or testing) phase is to automate this task, or at least, to automatically confirm vulnerabilities for which inputs can be found that bypass sanitization functions.

During the dynamic phase, we test the effectiveness of the sanitization routines applied along the paths between a source and the corresponding sink. This is done by directly

executing the corresponding sanitization routines, using as input a number of attack strings. Then, a decision function (typically called an " oracle" in a testing context) is used to evaluate whether the attack strings were successfully reduced to non-malicious values. If the testing process confirms that the sanitization is actually ineffective between a source and a sink, it also provides the path along which malicious input can reach the sink, as well as a sample attack string that successfully exploits the vulnerability. This information can then be leveraged by the developer to identify and fix the problem.

Ideally, one would run dynamic tests on a live installation of the application. However, it is often the case that a vulnerability may be exploited (and, therefore, discovered through testing procedures) only if the application is in a certain, well-defined state (e.g., after the administrator has logged in and the database contains specific values). Unfortunately, it is very difficult to test an application under these conditions in a completely automated fashion. Therefore, we take a different testing approach that focuses only on the sanitization process itself and abstracts away all other details of the application.

The dynamic analysis phase is composed of two different steps. First, we construct a *sanitization graph* for each pair of sources and sinks that are provided by the static analysis component. Conceptually, we model the sanitization process as the execution of a sequence of primitive operations, i.e., sanitization functions. The sanitization graph is the data structure that we use to efficiently store the sequences of sanitization operations that are applied to the input along all paths from a source to its sink. The second step of the dynamic analysis uses the sanitization graph to test the corresponding sanitization code using a number of predefined test cases.

3.2.1 Extracting the Sanitization Graph

For a given pair of source and sink nodes, the sanitization graph is a slice (subgraph) of the interprocedural data-flow graph of the application. This slice contains all nodes that correspond to sanitization instructions along the paths from the source to the sink. More precisely, we first compute the interprocedural data-flow graph of the program between the source and the sink nodes. By definition, each node of the resulting subgraph represents an operation that affects the contents of the variables used by the sink. This graph is then simplified by keeping only those nodes that correspond to statements relevant to the sanitization process. These include all calls to language-provided sanitization routines (such as `strip_tags` and `htmlspecialchars`), regular-expression-based substitution functions (`preg_replace`), string-based substitutions (e.g., `str_replace`, `strtoupper`), as well as other built-in string operations (e.g., concatenation).

As an example, consider the code snippet shown in Figure 9. The static analysis of the program identifies that a user-provided input on Line 11 (the source) can reach the

```
1  <?php
2  function sanitize($data){
3    $res = eregi_replace("<script", "", $data);
4    if(version_compare(phpversion(),"4.3.0")=="-1")
5      $res = mysql_escape_string($res);
6    else
7      $res = mysql_real_escape_string($res);
8    return $res;
9  }
10
11 $name = sanitize($_GET["username"]);
12 echo "Name: ".$name;
13 ?>
```

Figure 9. Customized sanitization function.

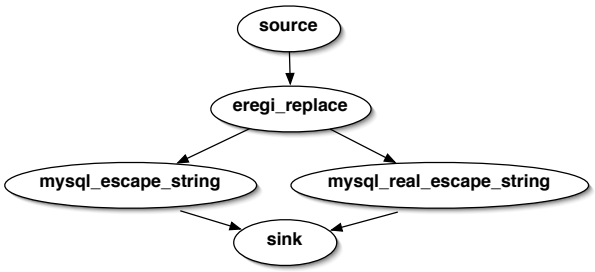


Figure 10. Sanitization graph.

`echo` statement at Line 12 (the sink) without proper sanitization. Following the data flow edges, we identify three operations that affect the content of the variable `$name` that is used by the sink: `mysql_real_escape_string` at Line 5, `mysql_escape_string` at Line 7, and the regular expression at Line 3. The nodes corresponding to these three functions are connected according to the edges in the data flow graph, to reflect the order in which the operations are performed in the original program. Figure 10 shows the resulting sanitization graph for our simple example.

3.2.2 Testing the Effectiveness of the Sanitization Routines

To analyze the effectiveness of the sanitization operations performed by the application, we use the sanitization graph to extract all possible paths P_i that lead from the source to the sink. In our experiments, we found that the sanitization graph is usually acyclic. However, in order to avoid possible paths of infinite length, we adopted the common solution of traversing each loop only once. For each path P_i, we generate a block of code C_i by concatenating the PHP instructions that correspond to each node that belongs to P_i. This operation may involve resolving values of variables to determine (or extract) parameters used in a function call. In our example, the call to `eregi_replace` is examined to fetch the constant values of the first and second parameter. The set of all the blocks C_i corresponds to all the possible combinations of sanitization operations that are applied by the program between the source and the sink.

Since the sanitization graph only considers data flow information, it is possible that our analysis generates code corresponding to infeasible paths, i.e., paths that cannot be executed at runtime. This can result from the fact that the sanitization graph does not model branch conditions, and, therefore, the dynamic analysis might consider branches that, in the original program, are infeasible. In case user input is not properly sanitized along an infeasible path, the tool might generate a false positive. This would occur if proper sanitization is performed on all other, feasible paths. However, we have not observed such a case in our experiments. Also, note that even though we remove all cycles from the sanitization graph, the number of paths can still grow exponentially. When the number of paths exceeds the capacity of our dynamic analysis to examine them all, we assume that the sanitization is incorrect. Again, this problem has not occurred during our experiments.

Depending on the type of the sink, we then select the appropriate test suite to be used in the experiment. For the prototype implementation of Saner, we implemented two different test suites: one for testing cross-site scripting attacks and one for testing SQL injection attacks. Both test suites contain a large number of test cases, each representing a particular value of user input that contains malicious data. We created these test suites using attack strings derived from both our own experience and from specialized web sites (such as [23,35]). For instance, typical input values that can be used to test the effectiveness of sanitization code for XSS are:

```
<script>alert(1);</script>
<scr<scriptipt src=http://evil.com/attack.js>
<img onmouseover="alert(1);" src=""></img>
<SCRIPT src="http://..."?<B>
```

Some of these test cases are just straightforward examples of XSS attacks, while others represent more complex cases that are likely to evade poorly-written sanitization routines. Note that we do not claim that the set of test cases that we adopt in our test runs is complete. Instead, the goal of the dynamic analysis part is to examine automatically and in detail those cases in which the static analysis phase has reported a potential vulnerability.

Finally, we invoke the PHP interpreter to evaluate the result of executing each block of code C_i on the list of malicious inputs contained in the selected test suite. The results of the executions are collected and analyzed by a special oracle function associated with the test case. This function is responsible for deciding whether the input string was successfully sanitized by the code under test. To do that, it is usually enough to check for the occurrence of particular substrings in the result. However, more advanced techniques can be used to implement oracle functions, such as analyzing the structure of a SQL query to verify the result of a SQL injection.

In our example of Figure 10, we can identify two different paths through the sanitization graph. During the execution of the first test case (the one that uses the input string `<script>alert(1);</script>`), we invoke the PHP interpreter to evaluate the two following pieces of code:

```
Code 1:
  $tmp = "<script>alert(1);</script>";
  $tmp = eregi_replace("<script", "", $tmp);
  $tmp = mysql_escape_string($tmp);

Code 2:
  $tmp = "<script>alert(1);</script>";
  $tmp = eregi_replace("<script", "", $tmp);
  $tmp = mysql_real_escape_string($tmp);
```

At the end of the two executions, the value of the `$tmp` variable is analyzed by the oracle function associated with the test case. In this simple case, the oracle is realized as a check to verify that the resulting string still contains the unescaped `script` tag. Note that the execution of this test does not spot any vulnerability, since the `eregi_replace` function does remove the opening tag from the input string. However, the following test case, which is based on the attack string `<scr<scriptipt src=http://evil.com/attack.js>`, will immediately reveal the weakness of the sanitization routine.

4 Evaluation

We implemented our approach in a prototype tool called Saner, and we evaluated this system on five popular, publicly-available PHP applications that contain custom sanitization routines.

The results of our analysis are shown in Table 1. The table shows the total number of security-sensitive sinks that are present in each analyzed program (*sinks total*). The next column (*sinks with sanitization*) shows the subset of sinks that have as input at least one value that depends on the output of a sanitization routine. That is, for each sink, there exists at least one program path such that the output of a sanitization routine flows into this sink. This number is important, as it represents those cases where incorrect sanitization could have a potential impact on the security of the program. It serves as a baseline for those sinks that need to be further analyzed for incorrect sanitization. Note that the number of sinks with sanitization is significantly smaller than the total number of sinks. The reason is that many of the sensitive sinks do not receive any tainted input. Also, in a few cases, tainted input reaches sensitive sinks directly, without any sanitization. In such cases, Pixy would report a vulnerability. However, for this paper, we are only interested in paths on which sanitization operations are performed.

The column *eliminated (basic)* shows the number of sensitive sinks with sanitization for which the basic analysis of Pixy determines that there is no security problem. This could be, for example, when the sanitization routine is not processing any malicious input, and, as a result, its output is guaranteed to be benign. The column *eliminated (advanced)* counts those cases in which the sensitive sinks do process potentially malicious input, but our improved static analysis is able to verify that the sanitization process works as intended. Finally, all sinks for which the static

Application	Sinks		Static Analysis		Dynamic Analysis		
	Total	With Sanitization	Sinks Eliminated (Basic)	Sinks Eliminated (Advanced)	Sinks Analyzed	Sinks Vulnerable	Not Vulnerable
Jetbox 2.1	311	7	5	0	2	1	1
MyEasyMarket 4.1	737	50	0	45	5	5	0
PBLGuestbook 1.32	41	5	2	0	3	3	0
PHP-Fusion 6.01	1,015	67	19	0	48	4	44
Sendcard 3.4.1	84	10	2	0	8	1	7
Totals	2,188	139	28	45	66	14	52

Table 1. Detection results.

analysis process cannot verify the correctness of the preceding sanitization routines are forwarded to the dynamic analysis (*sinks analyzed*).

The column *sinks confirmed* reports the total number of sinks that are confirmed to be vulnerable after running the dynamic analysis phase. The last column (*not vulnerable*) shows those sinks for which the dynamic analysis could not find input to bypass the sanitization routines. We manually verified that all these cases are indeed harmless and represent false positives produced by the static analysis phase. This demonstrates that the dynamic analysis phase was able to correctly produce input values to bypass the sanitization routines in all vulnerable cases. Of course, in general, dynamic analysis suffers from false negatives. Thus, to remain sound, a human analyst would have to verify all cases manually in which no malicious input is found. However, our experiments show that the dynamic analysis phase is very accurate in practice. Therefore, one could choose to trust its results. In this case, the system would only report sanitization routines that are very likely incorrect and also suggest input values that can be used to exploit the identified vulnerabilities.

The outcome of the experiments confirms our original hypothesis that sanitization mechanisms used in real-world applications are not always effective and can often be circumvented by determined attackers. To the best of our knowledge, all the ineffective sanitization routines discovered during the experiments correspond to novel, previously unknown vulnerabilities (with the exception of one of those found in PBLGuestbook, which was previously described in CVE-2006-3617). Therefore, we identified 13 novel vulnerabilities in the five applications we analyzed. We have working exploits for each vulnerability and notified the appropriate application developers.

4.1 Discussion of Sanitization Errors

A detailed analysis of the vulnerabilities detected in our experiments reveals that the sanitization process performed by a program can be ineffective for several reasons, which we classify based on the most common cases that we have encountered.

First, the code that performs the sanitization can contain programming errors. That is especially true if the sanitization is based on regular expressions, whose complex syntax can lead inexperienced developers to introduce subtle bugs in their specification. For example, MyEasyMarket attempts to sanitize the user-provided parameter `www` by using the following regular expression-based substitution:

```
ereg_replace("[^A-Za-z0-9 .-@://]","",$www);
```

Clearly, the parameter is used to store a URL and the developer intended to allow it to include the '−' (dash) character. Unfortunately, the dash character, when used inside a character class definition (marked by the square brackets), is interpreted as the character range separator. Therefore, the regular expression leaves unaffected all characters included in the range between '`.`' (dot) and '`@`' (at), which includes the open and close tag characters (< and >) and the equal symbol (=). Thus, an attacker can inject the string `<script src=http://evil.com/attack.js/>` and successfully perform a cross-site scripting attack.

Second, the sanitization process can be bug-free but insufficient. This is usually due to two reasons: the developer is not aware of all possible attack vectors (e.g., does not remove all HTML elements that can cause the execution of JavaScript code), or, even if she is, she ignores the fact that browsers accept and interpret malformed, nonstandard documents. As an example of the first problem, consider the following sanitization performed in Jetbox (slightly simplified for readability):

```
function removeEvilTags($source){
  $allowedTags = "<h1><b><i><a>";
  $source = strip_tags($source, $allowedTags);
  return preg_replace('/<(.*?)>/ie',
     "'<'.removeEvilAttributes('\\1').'>'",
     $source);
}

function removeEvilAttributes($tagSource){
   $stripAttrib = 'javascript:|onclick|".
     "ondblclick|onmousedown|onmouseup|".
     "onmouseover|onmousemove|onmouseout|".
     "onkeypress|onkeydown|onkeyup|style|".
     "onload|onchange';

   return preg_replace("/$stripAttrib/i",
     'forbidden', $tagSource);
}
```

The developer intended to only permit the use of a limited number of HTML tags cleaned of attributes that allow for the execution of JavaScript code. However, the list of insecure attributes is not complete: for example, the input string `<a onfocus="malicious code" href="url">dummy</a>` remains unaltered when it is processed by the sanitization routines, and thus, it can be used to execute malicious code. As an example of the second problem, consider the sanitization performed by PBLGuestbook:

```
preg_replace(
    "/\<SCRIPT(.*?)\>(.*?)\<\/SCRIPT(.*?)\>/i",
    "SCRIPT BLOCKED", $value);
```

Note that the specified pattern looks for a closing `script` tag. Unfortunately, most browsers accept malformed documents where an open tag is not followed by a corresponding close tag, and automatically insert the missing close tag. Therefore, an attacker can provide the input string `<script>malicious code<` to circumvent this sanitization.

Finally, the sanitization process implemented by a developer may correctly take into account all attack vectors but still be evadable. For example, consider the following sanitization:

```
str_replace("script","", $input)
```

This sanitization routine is intended to completely remove all occurrences of the string "`script`" from the user input. Unfortunately, an attacker can bypass this sanitization by providing the string `<scrscriptipt> code </scrscriptipt>` as input. The reason is that this string is transformed by the sanitization procedure into `<script>code <script>`, which invokes the embedded JavaScript code.

4.2 Discussion of Effectiveness and Efficiency

The combination of static and dynamic techniques proved to be effective. In fact, for the smaller applications, the dynamic testing phase was always able to automatically confirm all the alerts provided by the static analysis part. The advantage of using the dynamic analysis is more evident when analyzing larger applications. For example, in PHP-Fusion, the static analysis component generates a large number of alerts, which, in all benign cases, were correctly identified as false positives by the dynamic analysis phase.

Our results also indicate that current state-of-the-art vulnerability analysis tools would benefit from our approach, especially when analyzing applications that use a non-trivial amount of custom sanitization code. In fact, our approach provides a method to reduce the false positives that are generated when a tool conservatively considers all custom sanitization routines to be ineffective, and false negatives if the tool takes the opposite approach of considering all sanitization routines to be secure. The reason is that the sanitization functions are precisely modeled, and not *a priori* assumed to be either entirely correct or faulty.

Table 2 presents the runtime performance of Saner. For each application, we report the total number of lines of code (*lines of code*), the time required to perform the static analysis phase (*static analysis time*), the time required to perform the dynamic analysis phase (*dynamic analysis time*), and the total time (*total time*). Note that, even though in this prototype implementation performance was not a primary consideration, the time required to perform our analysis is in the order of a few minutes for almost all applications, and, in all cases, well under 20 minutes. Through careful engineering, the performance of our research prototype could be further enhanced. However, we believe that the current system operates well in practice and can be successfully used with large, real-world applications.

Interestingly, during the dynamic analysis phase, most of the time was spent to compute the inter-procedural data flow graph and extract the sanitization graphs. In our experiments, sanitization graphs were generally small, both in terms of number of nodes (i.e., sanitization primitives used) and of number of paths. Therefore, the time spent running the test attacks had a limited impact on the total time.

5 Related Work

The approach described in this paper is a composition of static and dynamic analysis techniques. Therefore, in the following two sections, we review the research work that is related to these two types of analysis.

5.1 Static Analysis

Type-Based Analysis. For typed programming languages, information about the taint status of variables can be propagated through the program by extending the type system of the language. For example, CQual [7] is a tool that allows one to extend the type system of the C language with user-defined qualifiers. After defining the new type system, the programmer manually introduces the additional qualifiers at a few key points in the application. CQual's qualifier inference then determines whether the program contains a type error under the extended system. This technique was used by Shankar et al. [38] for the detection of format string vulnerabilities, and by Johnson and Wagner [16] to identify user/kernel pointer bugs in the Linux kernel. Analogously, Zhang et al. [46] discovered security problems regarding the placement of authorization hooks in the Linux Security Modules framework.

JFlow [26] is an extension to the Java programming language that adds a type system for tracking information flow. In this system, the user is provided with annotations (labels) that define restrictions on the way in which the information may be used in the program, permitting the verification of information confidentiality and integrity. JFlow

Application	Lines of Code (#)	Static Analysis Time (s)	Dynamic Analysis Time (s)	Total time (s)
Jetbox 2.1	69,177	62	5	67
MyEasyMarket 4.1	2,544	202	26	228
PBLGuestbook 1.32	1,595	40	180	220
PHP-Fusion 6.01	56,339	723	386	1,109
Sendcard 3.4.1	8,504	130	38	168

Table 2. Performance results.

supports a wide range of language features (such as objects and exceptions), and is implemented in the Jif [15] tool.

Rule-Based Bug Finding. In [5], Engler et al. present meta-level compilation, a technique for the translation of simple user-defined rules (such as "never use floating point in the kernel") into extensions for the C compiler. During the compilation of a program, these extensions are able to determine whether the program violates the specified rules. An automated extraction of such program rules from a given application is described in [6]. In [2], the authors use the system to detect potentially dangerous accesses to user-supplied, unchecked values in Linux and OpenBSD.

Web Application Analysis. There exist several approaches that are focused on the detection of "taint-style" vulnerabilities (such as XSS or SQL injections), which frequently occur in web applications. Huang et al. [13] adapted parts of the techniques used in CQual to develop an intraprocedural analysis for PHP programs. In [14], the same authors present an alternative approach that is based on bounded model checking. Whaley and Lam [44] described an interprocedural, flow-insensitive alias analysis for Java applications. Their analysis is based on binary decision diagrams, and was used by Livshits and Lam [22] for the detection of taint-style vulnerabilities. As already mentioned, our approach is based on Pixy [17,18], an open source static PHP analyzer that uses taint analysis for detecting XSS vulnerabilities.

In [9], the authors applied the Java String Analyzer by Christensen et al. [4] to extract models of a program's database queries, and used these models as the basis for a runtime monitoring and protection component for SQL injection attacks. The main difference compared to our approach is that the extracted models do not contain information about the taint status of embedded variables. As a result, it is not possible to detect vulnerabilities using static analysis only. In our system, we can identify vulnerabilities using static analysis alone. In addition, we use a dynamic phase to automatically determine inputs that can exploit a vulnerability.

Xie and Aiken [45] presented an interprocedural and flow-sensitive system for the discovery of SQL injection vulnerabilities through a bottom-up analysis of basic blocks, procedures, and the whole program. In their work, the authors take into account the effect of applying one of a number of regular expressions to an input value. In principle, the authors manually specify a list of regular expressions that simply extends the list of built-in sanitization routines (such as `htmlentities`). In contrast, our technique automatically decides whether an arbitrary regular expression is suitable for sanitization, and requires no manual extensions when scanning new applications.

The system that is probably closest to ours was developed by Wassermann and Su [43]. In their paper, the authors present a static analysis technique for finding subtle SQL injection flaws. For this, they independently and concurrently developed a mechanism to determine the possible string values of variables in PHP programs. This allows them to take into account the effect of sanitization routines. The differences to our work are twofold. First, our focus is different. We attempt to verify the correctness of the sanitization process and do not limit our analysis to the detection of SQL injection vulnerabilities. Second, our system employs an additional dynamic analysis phase to find automatically input values that can exploit a vulnerability.

5.2 Dynamic Analysis

The dynamic techniques described in this paper are related to the research in the area of applying dynamic taint propagation analysis to web applications. Perl's *Taint mode* [31] is one of the best-known examples of such approaches. Similar approaches have been applied to other languages as well: Nguyen-Tuong et al. [28] propose modifications of the PHP interpreter to dynamically track tainted data in PHP programs, and Haldar et al. [8] have instrumented the Java Virtual Machine. Pietraszek and Vanden Berghe [33] present a unifying view of injection vulnerabilities and describe a general approach to the detection and prevention of injection attacks through the dynamic tracking of the flow of untrusted data inside an application. None of these approaches, however, provides a precise modeling of sanitization routines used to untaint data, and, thus, these approaches do not offer an effective protection against web attacks.

The dynamic part of our work is also related to a number of research results and tools in the areas of application security testing and fault injection [3,12,19,29,37,40]. All these systems inject malicious (or sometimes random) input into the applications to identify security problems. In our tool, we inject strings corresponding to possible XSS and SQL injection attacks to dynamically execute parts of the analyzed applications. Our work is also related to the

data flow testing of applications, such as [10, 11, 34]. In this type of testing, data flow graphs are used to identify the test case requirements for a program. In our approach, we use data flow graphs to find program statements related to the sanitization process.

6 Conclusions

Web applications perform mission-critical tasks and handle sensitive information. Even though there have been a number of research efforts to identify the use of unvalidated input in web applications, little has been done to characterize how sanitization is actually performed and how effective it is in blocking web-based attacks.

In this paper, we have presented Saner, a novel approach to the evaluation of the sanitization process in web applications. The approach relies on two complementary analysis techniques to identify faulty sanitization procedures. We implemented our approach, and by applying it to real-world applications, we identified novel vulnerabilities that stem from incorrect or incomplete sanitization. Future work will focus on the analysis of type-based validation procedures, as scripting languages often allow the programmer to interpret the values of variables in different ways, depending on the application context. This might lead to vulnerabilities that are difficult to detect.

Acknowledgements

This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368, and by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

References

[1] C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd, 2002.

[2] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, 2002.

[3] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *9th Information Security Conference (ISC)*, Samos Island, Greece, September 2006.

[4] A. Christensen, A. Møller, and M. Schwartzbach. Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*, 2003.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating System Principles (SOSP)*, 2001.

[7] J. Foster, M. Faehndrich, and A. Aiken. A Theory of Type Qualifiers. In *Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *21st Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, December 2005.

[9] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *International Conference on Automated Software Engineering (ASE)*, pages 174–183, November 2005.

[10] M. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, 1994.

[11] M. Harrold and M. Soffa. Interprocedural Data Flow Testing. In *ACM SIGSOFT 3rd Symposium on Software Testing, Analysis, and Verifications*, pages 158–167, 1989.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *11th International Conference on World Wide Web (WWW)*, 2003.

[13] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *12th International World Wide Web Conference (WWW)*, 2004.

[14] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Verifying Web Applications Using Bounded Model Checking. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

[15] Jif: Java + Information Flow. `http://www.cs.cornell.edu/jif/`, 2007.

[16] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *13th USENIX Security Symposium*, 2004.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006.

[19] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *15th International World Wide Web Conference (WWW)*, United Kingdom, May 2006.

[20] A. Klein. Cross Site Scripting Explained. Technical report, Sanctum Inc., 2002.

[21] J. Liberty and D. Hurwitz. *Programming ASP.NET*. O'REILLY, February 2002.

[22] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium*, pages 271–286, August 2005.

[23] F. Mavituna. SQL Injection Cheat Sheet, Version 1.4. `http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/`, May 2007.

[24] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *14th International Conference on World Wide Web (WWW)*, 2005.

[25] M. Mohri and R. Sproat. An Efficient Compiler for Weighted Rewrite Rules. In *34th Annual Meeting on Association for Computational Linguistics*, 1996.

[26] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages (POPL)*, 1999.

[27] Netcraft. PHP Usage Stats. `http://www.php.net/usage.php`, June 2007.

[28] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th International Information Security Conference (SEC)*, pages 372–382, May 2005.

[29] Nikto. Web Server Scanner. `http://www.cirt.net/code/nikto.shtml/`.

[30] OWASP. Top ten project. `http://www.owasp.org/`, May 2007.

[31] Perl. Perl security. `http://perldoc.perl.org/perlsec.html`.

[32] PHP: Hypertext Preprocessor. `http://www.php.net`, 2005.

[33] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, pages 372–382, 2005.

[34] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[35] RSnake. XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`, May 2007.

[36] Security Space. Apache Module Report. `http://www.securityspace.com/s_survey/ data/man.200603/apachemods.html`, April 2006.

[37] T. N. SecurityTM. Nessus Vulnerability Scanner. `http://www.nessus.org/`.

[38] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium*, 2001.

[39] K. Spett. Blind SQL Injection. Technical report, SPI Dynamics, 2003.

[40] Spike. `http://www.immunitysec.com/resources-freesoftware.shtml`.

[41] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[42] Symantec. Symantec internet security threat report, March 2007.

[43] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[44] J. Whaley and M. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[45] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, August 2006.

[46] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *11th USENIX Security Symposium*, 2002.

Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications

Theodoor Scholte[1], Davide Balzarotti[2], Engin Kirda[2,3]

[1] SAP Research, Sophia Antipolis
theodoor.scholte@sap.com

[2] Institut Eurecom, Sophia Antipolis
{balzarotti,kirda}@eurecom.fr

[3] Northeastern University, Boston
kirda@eurecom.fr

Abstract. Web applications have become important services in our daily lives. Millions of users use web applications to obtain information, perform financial transactions, have fun, socialize, and communicate. Unfortunately, web applications are also frequently targeted by attackers. Recent data from SANS institute estimates that up to 60% of Internet attacks target web applications.
In this paper, we perform an empirical analysis of a large number of web vulnerability reports with the aim of understanding how input validation flaws have evolved in the last decade. In particular, we are interested in finding out if developers are more aware of web security problems today than they used to be in the past. Our results suggest that the complexity of the attacks have not changed significantly and that many web problems are still simple in nature. Hence, despite awareness programs provided by organizations such as MITRE, SANS Institute and OWASP, application developers seem to be either not aware of these classes of vulnerabilities, or unable to implement effective countermeasures. Therefore, we believe that there is a growing need for languages and application platforms that attack the root of the problem and secure applications by design.

1 Introduction

The web has become part of everyone's daily life, and web applications now support us in many of our daily activities. Unfortunately, web applications are prone to various classes of vulnerabilities. Hence, much effort has been spent on making web applications more secure in the past decade (e.g., [4][15][28]).

Organizations such as MITRE [15], SANS Institute [4] and OWASP [28] have emphasized the importance of improving the security education and awareness among programmers, software customers, software managers and Chief Information Officers. These organizations do this by means of regularly publishing lists

with the most common programming errors. Also, the security research community has worked on tools and techniques to improve the security of web applications. These techniques include static code analysis [9, 14, 33–35], dynamic tainting [23, 24, 27], combination of dynamic tainting and static analysis [32], prevention by construction or by design [8, 13, 29, 36] and enforcement mechanisms executing within the browser [1, 7, 10, 31]. Some of these techniques have been commercialized and can be found in today's development toolsets. An example is Microsoft's FxCop [6] which can be integrated into some editions of Microsoft Visual Studio.

Although a considerable amount of effort has been spent by many different stake-holders on making web applications more secure, we lack quantitative evidence that this attention has improved the security of web applications over time. In particular, we are interested in finding out and understanding how two common classes of vulnerabilities, namely SQL injection and Cross Site Scripting, have evolved in the last decade.

We chose to focus our study on SQL Injection and Cross-Site Scripting vulnerabilities as these classes of web application vulnerabilities have the same root cause: improper sanitization of user-supplied input that result from invalid assumptions made by the developer on the input of the application. Moreover, these classes of vulnerabilities are prevalent, well-known and have been well-studied in the past decade. Thus, it is likely that there is a sufficient number of vulnerability reports available to allow an empirical analysis.

In this paper, by performing an automated analysis, we attempt to answer the following questions:

1. *Do attacks become more sophisticated over time?*
   We automatically analyzed over 2600 vulnerabilities and found out that the vast majority of them was not associated to any sophisticated attack techniques. Our results suggest that the exploits do not intend to evade any input validation, escaping or encoding defense mechanisms. Moreover, we do not observe any particular increasing trend with respect to complexity.
2. *Do well-known and popular applications become less vulnerable over time?*
   Our results show that an increasing number of applications have exactly one vulnerability. Furthermore, we observe a shift from popular applications to non-popular applications with respect to SQL Injection vulnerabilities, a trend that is, unfortunately, not true for Cross-Site Scripting.
3. *Do the most affected applications become more secure over time?*
   We studied in detail the ten most affected open source applications resulting in two top ten lists – one for Cross-Site Scripting and one for SQL Injection. In total, 197 vulnerabilities were associated with these applications. We investigated the difference between *foundational* and *non foundational* vulnerabilities and found that the first class is decreasing over time. Moreover, an average time of 4.33 years between the initial software release and the vulnerability disclosure date suggests that many of today's reported Cross-Site Scripting vulnerabilities were actually introduced into the applications many years ago.

The rest of the paper is organized as follows: The next section describes our methodology and data gathering technique. Section 3 presents an analysis of the SQL Injection and Cross-Site Scripting reports and their associated exploits. In Section 4, we present the related work and then briefly conclude the paper in Section 5.

2 Methodology

To be able to answer how Cross Site Scripting and SQL Injections have evolved over time, it is necessary to have access to significant amounts of vulnerability data. Hence, we had to collect and classify a large number of vulnerability reports. Furthermore, automated processing is needed to be able to extract the exploit descriptions from the reports. In the next sections, we explain the process we applied to collect and classify vulnerability reports and exploit descriptions.

2.1 Data Gathering

One major source of information for security vulnerabilities is the CVE dataset, which is hosted by MITRE [19]. According to MITRE's FAQ [21], CVE is not a vulnerability database but a vulnerability identification system that 'aims to provide common names for publicly known problems' such that it allows 'vulnerability databases and other capabilities to be linked together'. Each CVE entry has a unique CVE identifier, a status ('entry' or 'candidate'), a general description, and a number of references to one or more external information sources of the vulnerability. These references include a source identifier and a well-defined identifier for searching on the source's website. Vulnerability information is provided to MITRE in the form of *vulnerability submissions*. MITRE assigns a CVE identifier and a candidate status. After the CVE Editorial Board has reviewed the candidate entry, the entry may be assigned the 'Accept' status.

For our study, we used the CVE data from the National Vulnerability Database (NVD) [25] which is provided by the National Institute of Standards and Technology (NIST). In addition to CVE data, the NVD database includes the following information:

- Vulnerability type according to the Common Weakness Enumeration (CWE) classification system [20].
- The name of the affected application, version numbers, and the vendor of the application represented by Common Platform Enumeration (CPE) identifiers [18].
- The impact and severity of the vulnerability according to the Common Vulnerability Scoring System (CVSS) standard [17].

The NIST publishes the NVD database as a set of XML files, in the form: `nvdcve-2.0-year.xml`, where year is a number from 2002 until 2010. The first file, `nvdcve-2.0-2002.xml` contains CVE entries from 1998 until 2002. In order to build timelines during the analysis, we need to know the discovery date, disclosure date, or the publishing date of a CVE entry. Since CVE entries originate

from different external sources, the timing information provided in the CVE and NVD data feeds proved to be insufficient. For this reason, we fetch this information by using the disclosure date from the corresponding entry in the Open Source Vulnerability Database (OSVDB) [11].

For each candidate and accepted CVE entry, we extracted and stored the identifier, the description, the disclosure date from OSVDB, the CWE vulnerability classification, the CVSS scoring, the affected vendor/product/version information, and the references to external sources. Then, we used the references of each CVE entry to retrieve the vulnerability information originating from the various external sources. We stored this website data along with the CVE information for further analysis.

2.2 Vulnerability Classification

Since our study focuses particularly on Cross-Site Scripting and SQL Injection vulnerabilities, it is essential to classify the vulnerability reports. As mentioned in the previous section, the CVE entries in the NVD database are classified according to the Common Weakness Enumeration classification system. CWE aims to be a dictionary of software weaknesses. NVD uses only a small subset of 19 CWEs for mapping CVEs to CWEs, among those are Cross-Site Scripting (CWE-79) and SQL Injection (CWE-89).

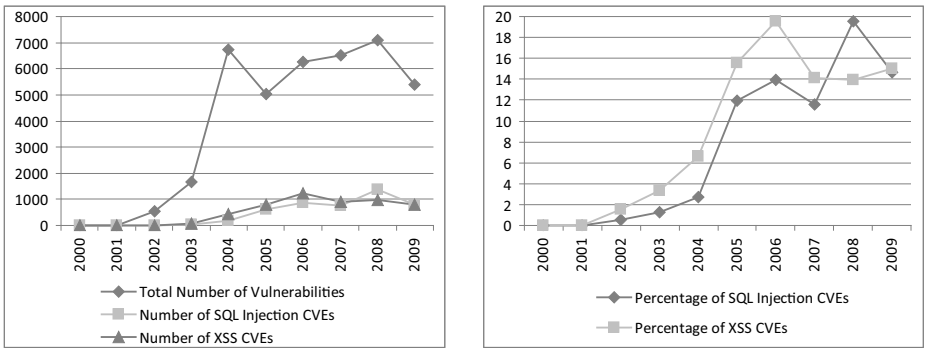
Although NVD provides a mapping between CVEs and CWEs, this mapping is not complete and many CVE entries do not have any classification at all. For this reason, we chose to perform a classification which is based on both the CWE classification and on the description of the CVE entry. In general, a CVE description is formatted according to the following pattern: {description of vulnerability} {location description of the vulnerability} *allows* {description of attacker} {impact description}. Thus, the CVE description includes the vulnerability type.

For fetching the Cross-Site Scripting related CVEs out of the CVE data, we selected the CVEs associated with CWE identifier 'CWE-79'. Then, we added the CVEs having the text 'Cross-Site Scripting' in their description by performing a case-insensitive query. Similarly, we classified the SQL Injection related CVEs by using the CWE identifier 'CWE-89' and the keyword 'SQL Injection'.

2.3 The Exploit Data Set

To acquire a general view on the security of web applications, we are not only interested in the vulnerability information, but also in the way each vulnerability can be exploited. Some external sources of CVEs that provide information concerning Cross-Site Scripting or SQL Injection-related vulnerabilities also provide exploit details. Often, this information is represented by a script or an *attack string*.

An attack string is a well-defined reference to a location in the vulnerable web application where code can be injected. The reference is often a complete URL that includes the name of the vulnerable script, the HTTP parameters, and some characters to represent the placeholders for the injected code. In addition



(a) Vulnerability trends in numbers. (b) Vulnerability trends in percentages

Fig. 1: *Cross-Site Scripting and SQL Injection vulnerabilities over time.*

of using placeholders, sometimes, real examples of SQL or Javascript code may also be used. Two examples of attack strings are:

```
http://[victim]/index.php?act=delete&dir=&file=[XSS]
http://[victim]/index.php?module=subjects&func=viewpage&pageid=[SQL]
```

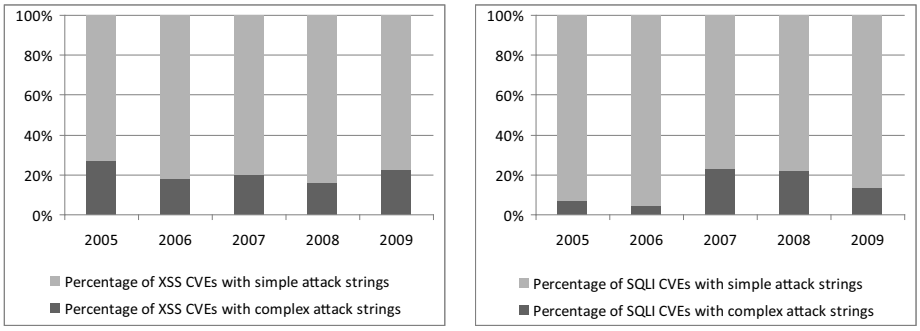
At the end of each line, note the placeholders that can be substituted with arbitrary code by the attacker.

The similar structure of attack strings allows our tool to automatically extract, store and analyze the exploit format. Hence, we extracted and stored all the attack strings associated with both Cross-Site Scripting and the SQL Injection CVEs.

3 Analysis of the Vulnerabilities Trends

The first question we wish to address in this paper is whether the number of SQL Injection and Cross-Site Scripting vulnerabilities reported in web applications has been decreasing in recent years. To answer this question, we automatically analyzed the 39,081 entries in the NVD database from 1998 to 2009. We had to exclude 1,301 CVE entries because they did not have a corresponding match in the OSVDB database and, as a consequence, did not have a disclosure date associated with them. For this reason, these CVE entries are not taken into account for the rest of our study. Of the remaining vulnerability reports, we identified a total of 5222 Cross-Site Scripting entries and 4810 SQL Injection entries.

Figure 1a shows the number of vulnerability reports over time and figure 1b shows the percentage of reported Cross-Site Scripting and SQL Injection vulnerabilities over the total CVE entries.



(a) Cross-Site Scripting (b) SQL Injection

Fig. 2: *Complexity exploits over time.*

Our first expectation based on intuition was to observe the number of reported vulnerabilities follow a classical bell shape: beginning with a slow start when the vulnerabilities are still relatively unknown, then a steep increase corresponding to the period in which the attacks are disclosed and studied, and finally a decreasing phase when the developers start adopting the required countermeasures.

In fact, the graphs show an initial phase (2002-2004) with very few reports, followed by a steep increase of Cross-Site Scripting and SQL Injection vulnerability reports in the years 2004, 2005 and 2006. Note that this trend is consistent with historical developments. Web security started increasing in importance after 2004, and the first XSS-based worm was discovered in 2005 (i.e., "Samy Worm"). Hence, web security threats such as Cross-Site Scripting and SQL Injection started receiving more focus after 2004.

Unfortunately, the number of reported vulnerabilities has not significantly decreased since 2006. In other words, the number of vulnerabilities found in 2009 is comparable with the number reported in 2006. In the rest of this section, we will formulate and verify a number of hypotheses to explain the possible reasons behind this phenomenon.

3.1 Attack Sophistication

Hypothesis 1 *Simple, easy-to-find vulnerabilities have now been replaced by complex vulnerabilities that require more sophisticated attacks.*

The first hypothesis we wish to verify is whether the overall number of vulnerabilities is not decreasing because the simple vulnerabilities discovered in the early years have now been replaced by new ones that involve more complex attack scenarios. For example, the attacker may have to carefully craft the malicious input in order to reach a subtle vulnerable functionality, or to pass certain input transformations (e.g., uppercase or character replacement). In particular, we are interested in identifying those cases in which the application developers were

aware of the threats, but implemented insufficient, easy to evade sanitization routines.

One way to determine the "complexity" of an exploit is to analyze the attack string, and to look for evidence of possible evasion techniques. As mentioned in Section 2.3, we automatically extracted the exploit code from the data provided by external vulnerability information sources. Sometimes, these external sources do not provide exploit information for every reported Cross-Site Scripting or SQL Injection vulnerability, do not provide exploit information in a parsable format, or do not provide any exploit information at all. As a consequence, not all CVE entries can be associated with an *attack string*. On the other hand, in some cases, there exist several ways of exploiting a vulnerability, and, therefore, more *attack strings* may be associated with a single vulnerability report. In our experiments, we collected attack strings for a total of 2632 distinct vulnerabilities.

To determine the exploit complexity, we looked at several characteristics that may indicate an attempt from the attacker to evade some form of input sanitization. The selection of the characteristics is inspired by so-called injection cheat sheets that are available on the Internet [16][30].

In particular, we classify a Cross-Site Scripting attack string as complex (i.e., in contrast to simple) if it contains one or more of the following characteristics:

- Different cases are used within the script tags (e.g., `ScRiPt`).
- The script-tags contains one or more spaces (e.g., `< script>`)
- The attack string contains 'landingspace-code' which is the set of attributes of HTML-tags (e.g., `onmouseover`, or `onclick`)
- The string contains encoded characters (e.g., `&#41;`)
- The string is split over multiple lines

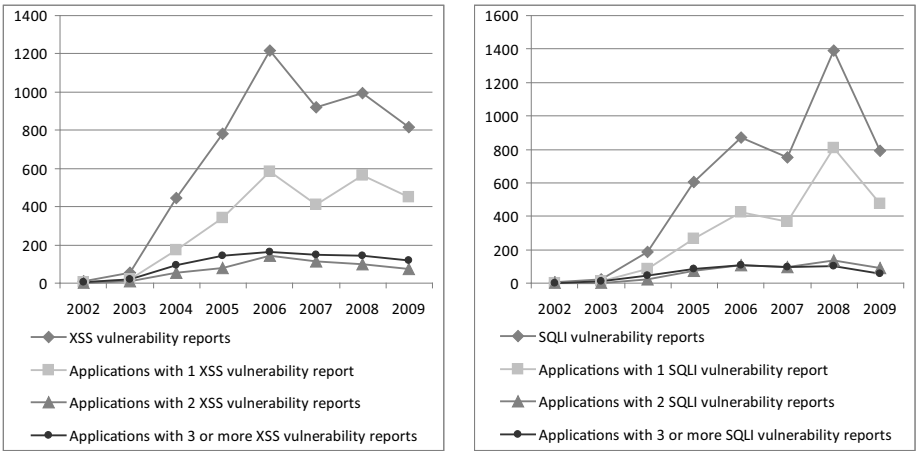
For SQL Injection attack strings, we looked at the following characteristics:

- The use of comment specifiers (e.g., `/**/`) to break a keyword
- The use of encoded single quotes (e.g., '`%27`', '`&#x27`'; '`&#39`', '`Jw==`')
- The use of encoded double quotes (e.g., '`%22`', '`&#x22;`', '`&#34`', '`Ig==`')

If none of the previous characteristics is present, we classify the exploit as "simple". Figures 2a and 2b show the percentage of CVEs having one or more complex attack strings[4]. The graphs show that the majority of the available exploits are, according to our definition, not sophisticated. In fact, in most of the cases, the attacks were performed by injecting the simplest possible string, without requiring any tricks to evade input validation.

Interestingly, while we observe a slight increase in the number of SQL Injection vulnerabilities with sophisticated attack strings, we do not observe any significant increase of Cross-Site Scripting attack strings. This may be a first indication that developers are now adopting (unfortunately insufficient) defense mechanisms to prevent SQL Injection , but that they are still failing to sanitize the user input to prevent Cross-Site Scripting vulnerabilities.

[4] The graph starts from 2005 because there were less than 100 vulnerabilities having exploit samples available before that year. Hence, results before 2005 are statistically less significant.



(a) Cross-Site Scripting affected applications.

(b) SQL Injection affected applications.

Fig. 3: *The number of affected applications over time.*

To conclude, the available empirical data suggests that an increased attack complexity *is not* the reason behind the steadily increasing number of vulnerability reports.

3.2 Application Popularity

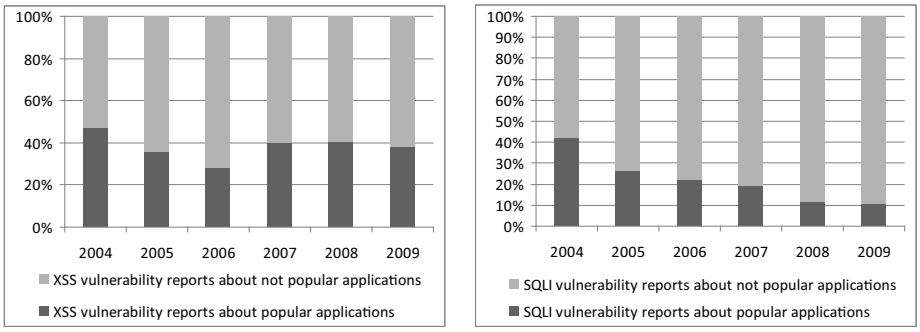
Since the complexity does not seem to explain the increasing number of reported vulnerabilities, we decided to focus on the type of applications. We started by extracting the vulnerable application's name from a total of 8854 SQL Injection and Cross-Site Scripting vulnerability reports in the NVD database that are associated to one or more CPE identifiers.

Figures 3a and 3b plot the number of applications that are affected by a certain number of vulnerabilities over time. Both graphs clearly show how the increase in the number of vulnerabilities is a direct consequence of the increasing number of vulnerable applications. In fact, the number of web applications with more than one vulnerability report over the whole time frame is quite low, and it has been slightly decreasing since 2006.

Based on this finding, we formulated our second hypothesis:

Hypothesis 2 *Popular applications are now more secure while new vulnerabilities are discovered in new, less popular, applications.*

The idea behind this hypothesis is to test whether more vulnerabilities were reported about well-known, popular applications in the past than they are today. That is, do vulnerability reports nowadays tend to concentrate on less popular, or recently developed applications?



(a) Cross-Site Scripting (b) SQL Injection

Fig. 4: *Vulnerability reports about applications and their popularity over time.*

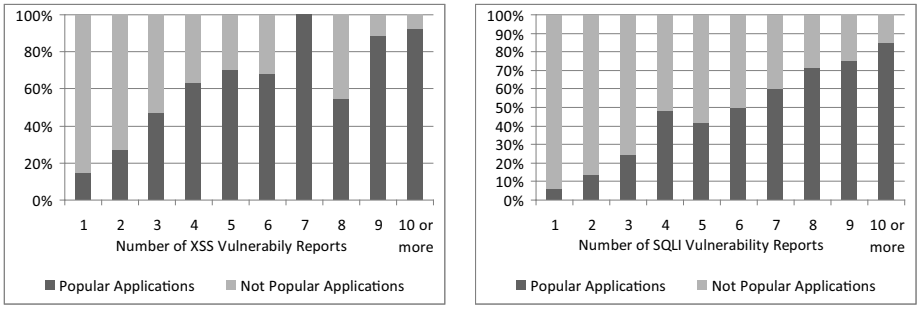
The first step consists of determining the popularity of these applications in order to be able to understand if it is true that popular products are more aware of (and therefore less vulnerable to) Cross-Site Scripting and SQL Injection attacks.

We determined the popularity of applications through the following process:

1. Using Google Search, we performed a search on the vendor and application names within the `Wikipedia` domain.
2. When one of the returned URLs contain the name of the vendor or the name of the application, we flag the application as being 'popular'. Otherwise, the application is classified as being 'unpopular'.
3. Finally, we manually double-checked the list of popular applications in order to make sure that the corresponding Wikipedia entries describe software products and not something else (e.g., when the product name also corresponds to a common English word).

After the classification, we were able to identify 676 popular and 2573 unpopular applications as being vulnerable to Cross-Site Scripting . For SQL Injection, we found 328 popular and 2693 unpopular vulnerable applications. Figure 4 shows the percentages of vulnerability reports that are associated with popular applications. The trends support the hypothesis that SQL Injection vulnerabilities are indeed moving toward less popular applications – maybe as a consequence of the fact that well-known product are more security-aware. Unfortunately, according to Figure 4a, the same hypothesis is not true for Cross-Site Scripting: in fact, the ratio of well-known applications vulnerable to Cross-Site Scripting has been relatively constant in the past six years.

Even though the empirical evidence also does not support our second hypothesis, we noticed one characteristic that is common to both types of vulnerabilities: popular applications, probably because they are analyzed in more detail, typically have a higher number of reported vulnerabilities. The results, shown



(a) Cross-Site Scripting (b) SQL Injection

Fig. 5: *Popularity of applications across the distribution of the number of vulnerability reports.*

in Figures 5a and 5b, suggest that it would be useful to investigate how these vulnerabilities have evolved in the lifetime of the applications.

3.3 Vulnerability lifetime

So far, we determined that a constant, large number of simple, easy-to-exploit vulnerabilities are still found in many web applications today. Also, we determined that that the high number of reports is driven by an increasing number of vulnerable applications, and not by a small number of popular applications. Based on these findings, we formulate our third hypothesis:

Hypothesis 3 *Even though the number of reported vulnerable applications is growing, each application is becoming more secure over time.*

This hypothesis is important, because, if true, it would mean that web applications, in particular the well-known products, are becoming more secure. To verify this hypothesis, we studied the lifetimes of Cross-Site Scripting and SQL Injection vulnerabilities in the ten most-affected open source applications according to the NIST NVD database.

By analyzing the changelogs, for each application, we extracted in which version a vulnerability was introduced and in which version the vulnerability was fixed. In order to obtain reliable insights into the vulnerabilities lifetime, we excluded the vulnerability reports that were not confirmed by the respective vendor. For our analysis, we used the CPE identifiers in the NVD database, the external vulnerability sources, the vulnerability information provided by the vendor, and we also extract information from the version control systems (CVS, or SVN) of the different products.

Table 1a and Table 1b show a total of 145 Cross-Site Scripting and 52 SQL Injection vulnerabilities in the most affected applications. The tables distinguish *foundational* and *non-foundational* vulnerabilities. Foundational vulnerabilities

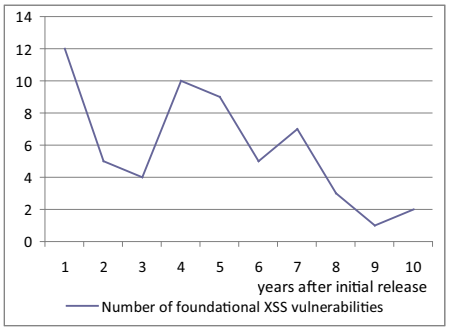
	Foundational	Non-Foundational
bugzilla	4	9
drupal	0	22
joomla	5	3
mediawiki	3	22
mybb	9	2
phorum	3	5
phpbb	4	2
phpmyadmin	14	13
squirrelmail	10	3
wordpress	6	6
Total	58	87

(a) Cross-Site Scripting

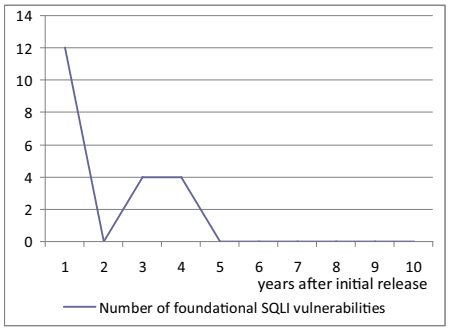
	Foundational	Non-Foundational
bugzilla	1	8
coppermine	1	3
e107	0	3
joomla	4	0
moodle	0	3
mybb	9	3
phorum	0	4
phpbb	2	1
punbb	3	3
wordpress	0	4
Total	20	32

(b) SQL Injection

Table 1: *Foundational and non-foundational vulnerabilities in the ten most affected open source web applications.*



(a) Cross-Site Scripting



(b) SQL Injection

Fig. 6: *Time elapsed between initial release and vulnerability disclosure.*

are vulnerabilities that were present in the first version of an application, while non-foundational vulnerabilities were introduced after the initial release.

We observed that 40% of the Cross-Site Scripting vulnerabilities are foundational and 60% are non-foundational. For SQL Injection, these percentages are 38% and 61%. These results suggest that most of the vulnerabilities are introduced by new functionality that is built into new versions of a web application.

Finally, we investigated how long it took to discover the *foundational* vulnerabilities. Figure 6a and Figure 6b plot the number of foundational vulnerabilities that were disclosed after a certain amount of time had elapsed after the initial release of the applications. The graphs show that most SQL Injection vulnerabilities are usually discovered in the first year after the release of the product. For Cross-Site Scripting vulnerabilities, the result is quite different. Many foundational vulnerabilities are discovered even 10 years after the code was initially released. This observation suggests that it is very problematic to find Cross-Site Scripting vulnerabilities compared to SQL Injection vulnerabilities. We believe

that this difference is caused by the fact that the attack surface for SQL Injection attacks is much smaller when compared with Cross-Site Scripting . Therefore, it is easier for developers to identify (and protect) all the sensitive entry points in the application code.

The difficulty of finding Cross-Site Scripting vulnerabilities is confirmed by the average elapsed time between the initial software release and the disclosure of foundational vulnerabilities. For SQL Injection vulnerabilities, this value is 2 years, while for Cross-Site Scripting is 4.33 years.

4 Related Work

Our work is not the first study of vulnerability trends based on CVE data. In [2], Christey et al. present an analysis of CVE data covering the period 2001 - 2006. The work is based on manual classification of CVE entries using the CWE classification system. In contrast, [22] uses an unsupervised learning technique on CVE text descriptions and introduces a classification system called *'topic model'*. While the works of Christey et al. and Neuhaus et al. focus on analysing general trends in vulnerability databases, our work specifically focuses on web application vulnerabilities, and, in particular, Cross-Site Scripting and SQL Injection. We have investigated the reasons behind the trends.

Clark et al. present in [3] a vulnerability study with a focus on the early existence of a software product. The work demonstrates that re-use of legacy code is a major contributor to the rate of vulnerability discovery and the number of vulnerabilities found. In contrast to our work, the paper does not focus on web applications, and it does not distinguish between particular types of vulnerabilities.

Another large-scale vulnerability analysis study was conducted by Frei et al. [5]. The work focuses on zero-day exploits and shows that there has been a dramatic increase in such vulnerabilities. Also, the work shows that there is a faster availability of exploits than of patches.

In [12], Li et al. present a study on how the number of software defects evolve over time. The data set of the study consists of bug reports of two Open Source software products that are stored in the Bugzilla database. The authors show that security related bugs are becoming increasingly important over time in terms of absolute numbers and relative percentages, but do not consider web applications.

Ozment et al. [26] studied how the number of security issues relate to the number of code changes in OpenBSD. The study shows that 62 percent of the vulnerabilities are *foundational*; they were introduced prior to the release of the initial version and have not been altered since. The rate at which foundational vulnerabilities are reported is decreasing, somehow suggesting that the security of the same code is increasing. In contrast to our study, Ozment el al.'s study does not consider the security of web applications.

To the best of our knowledge, we present the first vulnerability study that takes a closer, detailed look at how two popular classes of web vulnerabilities have evolved over the last decade.

5 Discussion and Conclusion

Our findings in this study show that the complexity of Cross Site Scripting and SQL Injection exploits in vulnerability reports have not been increasing. Hence, this finding suggests that the majority of vulnerabilities are not due to sanitization failure, but due to the absence of input validation. Despite awareness programs provided by MITRE [19], SANS Institute [4] and OWASP [28], application developers seem to be neither aware of these classes of vulnerabilities, nor are able to implement effective countermeasures.

Furthermore, our study suggests that a main reason why the number of web vulnerability reports have not been decreasing is because many more applications are now vulnerable to flaws such as Cross-Site Scripting and SQL Injection . In fact, we observe a trend that SQL Injection vulnerabilities occur more often in an increasing number of unpopular applications.

Finally, when analyzing the most affected applications, we observe that years after the initial release of an application, Cross-Site Scripting vulnerabilities concerning the initial release are still being reported. Note that this is in contrast to SQL Injection vulnerabilities. We believe that one of the reasons for this observation could be because SQL Injection problems may be easier to fix (e.g., by using stored procedures).

The empirical data we collected and analyzed for this paper supports the general intuition that web developers are bad at securing their applications. The traditional practice of writing applications and then testing them for security problems (e.g., static analysis, blackbox testing, etc.) does not seem be working well in practice. Hence, we believe that more research is needed in securing applications by design. That is, the developers should not be concerned about problems such as Cross Site Scripting or SQL Injection. Rather, the programming language or the platform should make sure that the problems do not occur when developers produce code (e.g., similar to solutions such as in [29] or managed languages such as C# or Java that prevent buffer overflow problems).

Acknowledgments

The research leading to these results was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) from the contract N 216917 (for the FP7-ICT-2007-1 project MASTER) and N 257007. This work has also been supported by the POLE de Competitivite SCS (France) through the MECANOS project and the French National Research Agency through the VAMPIRE project. We would also like to thank Secure Business Austria for their support.

References

1. Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 91–100, New York, NY, USA, 2010. ACM.

2. S. M. Christey and R. A. Martin. Vulnerability type distributions in cve. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007.
3. S. Clark, S. Frei, M. Blaze, and J. Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Annual Computer Security Applications Conference*, 2010.
4. Rohit Dhamankar, Mike Dausin, Marc Eisenbarth, and James King. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/, 2009.
5. Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, New York, NY, USA, 2006. ACM.
6. Microsoft Inc. Msdn code analysis team blog. http://blogs.msdn.com/b/codeanalysis/, 2010.
7. Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.
8. Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure code generation for web applications. In Fabio Massacci, Dan S. Wallach, and Nicola Zannone, editors, *ESSoS*, volume 5965 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2010.
9. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.
10. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.
11. Jake Kouns, Kelly Todd, Brian Martin, David Shettler, Steve Tornio, Craig Ingram, and Patrick McDonald. The open source vulnerability database. http://osvdb.org/, 2010.
12. Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM.
13. Benjamin Livshits and Úlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 95–104, New York, NY, USA, 2007. ACM.
14. V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.
15. Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/, 2010.
16. Ferruh Mavituna. Sql injection cheat sheet. http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/, 2009.
17. Peter Mell, Karen Scarfone, and Sasha Romanosky. A complete guide to the common vulnerability scoring system version 2.0. http://www.first.org/cvss/cvss-guide.html, 2007.
18. MITRE. Common platform enumeration (cpe). http://cpe.mitre.org/, 2010.

19. MITRE. Common vulnerabilities and exposures (cve). http://cve.mitre.org/, 2010.
20. MITRE. Common weakness enumeration (cwe). http://cwe.mitre.org/, 2010.
21. MITRE. Mitre faqs. http://cve.mitre.org/about/faqs.html, 2010.
22. Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, November 2010.
23. James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. The Internet Society, 2005.
24. Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.
25. Computer Security Division of National Institute of Standards and Technology. National vulnerability database version 2.2. http://nvd.nist.gov/, 2010.
26. Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
27. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In Alfonso Valdes and Diego Zamboni, editors, *RAID*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2005.
28. The Open Web Application Security Project. Owasp top 10 - 2010, the ten most critical web application security risks, 2010.
29. W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, pages 283–298. USENIX Association, 2009.
30. RSnake. Xss (cross site scripting) cheat sheet esp: for filter evasion. http://ha.ckers.org/xss.html, 2009.
31. K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, New York, NY, USA, 2009. ACM.
32. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *In Proceedings of 14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.
33. Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007. ACM Press New York, NY, USA.
34. Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.
35. Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
36. Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Better abstractions for secure server-side scripting. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 507–516, New York, NY, USA, 2008. ACM.

Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications

Marco Balduzzi*, Carmen Torrano Gimenez ‡, Davide Balzarotti*, and Engin Kirda* §

* Institute Eurecom, Sophia Antipolis
{balduzzi,balzarotti,kirda}@eurecom.fr

‡ Spanish National Research Council, Madrid
carmen.torrano@iec.csic.es

§ Northeastern University, Boston
ek@ccs.neu.edu

Abstract

In the last twenty years, web applications have grown from simple, static pages to complex, full-fledged dynamic applications. Typically, these applications are built using heterogeneous technologies and consist of code that runs both on the client and on the server. Even simple web applications today may accept and process hundreds of different HTTP parameters to be able to provide users with interactive services. While injection vulnerabilities such as SQL injection and cross-site scripting are well-known and have been intensively studied by the research community, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) has not received as much attention. If a web application does not properly sanitize the user input for parameter delimiters, exploiting an HPP vulnerability, an attacker can compromise the logic of the application to perform either client-side or server-side attacks.

In this paper, we present the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications. Using our prototype implementation called PAPAS (PArameter Pollution Analysis System), we conducted a large-scale analysis of more than 5,000 popular websites. Our experimental results show that about 30% of the websites that we analyzed contain vulnerable parameters and that 46.8% of the vulnerabilities we discovered (i.e., 14% of the total websites) can be exploited via HPP attacks. The fact that PAPAS was able to find vulnerabilities in many high-profile, well-known websites suggests that many developers are not aware of the HPP problem. We informed a number of major websites about the vulnerabilities we identified, and our findings were confirmed.

1 Introduction

In the last twenty years, web applications have grown from simple, static pages to complex, full-fledged dynamic applications. Typically, these applications are built using heterogeneous technologies and consist of code that runs on the client (e.g., Javascript) and code that runs on the server (e.g., Java servlets). Even simple web applications today may accept and process hundreds of different HTTP parameters to be able to provide users with rich, interactive services. As a result, dynamic web applications may contain a wide range of input validation vulnerabilities such as cross site scripting (e.g., [4, 5, 34]) and SQL injection (e.g., [15, 17]).

Unfortunately, because of their high popularity and a user base that consists of millions of Internet users, web applications have become prime targets for attackers. In fact, according to SANS [19], attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. While flaws such as SQL injection and cross-site scripting may be used by attackers to steal sensitive information from application databases and to launch authentic-looking phishing attacks on vulnerable servers, many web applications are being exploited to convert trusted websites into malicious servers serving content that contains client-side exploits. According to SANS, most website owners fail to scan their application for common flaws. In contrast, from the attacker's point of view, automated tools, designed to target specific web application vulnerabilities simplify the discovery and infection of several thousand websites.

While injection vulnerabilities such as SQL injection and cross-site scripting are well-known and have been inten-

sively studied, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) has not received as much attention. HPP was first presented in 2009 by di Paola and Carettoni at the OWASP conference [27]. HPP attacks consist of injecting encoded query string delimiters into other existing parameters. If a web application does not properly sanitize the user input, a malicious user can compromise the logic of the application to perform either client-side or server-side attacks. One consequence of HPP attacks is that the attacker can potentially override existing hard-coded HTTP parameters to modify the behavior of an application, bypass input validation checkpoints, and access and possibly exploit variables that may be out of direct reach.

In this paper, we present the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications. Our prototype implementation, that we call PArameter Pollution Analysis System (PAPAS), uses a black-box scanning technique to inject parameters into web applications and analyze the generated output to identify HPP vulnerabilities. We have designed a novel approach and a set of heuristics to determine if the injected parameters are not sanitized correctly by the web application under analysis.

To the best of our knowledge, no tools have been presented to date for the detection of HPP vulnerabilities in web applications, and no studies have been published on the topic. At the time of the writing of this paper, the most effective means of discovering HPP vulnerabilities in websites is via manual inspection. At the same time, it is unclear how common and significant a threat HPP vulnerabilities are in existing web applications.

In order to show the feasibility of our approach, we used PAPAS to conduct a large-scale analysis of more than 5,000 popular websites. Our experimental results demonstrate that there is reason for concern as about 30% of the websites that we analyzed contained vulnerable parameters. Furthermore, we verified that 14% of the websites could be exploited via client-side HPP attacks. The fact that PAPAS was able to find vulnerabilities in many high-profile, well-known websites such as Google, Paypal, Symantec, and Microsoft suggests that many developers are not aware of the HPP problem.

When we were able to obtain contact information, we informed the vulnerable websites of the vulnerabilities we discovered. In the cases where the security officers of the concerned websites wrote back to us, our findings were confirmed.

We have created an online service based on PAPAS[1] (currently in beta version) that allows website maintainers to scan their sites. As proof of ownership of a site, the website owner is given a dynamically-generated token that she can put in the document root of her website.

[1]The PAPAS service is available at: `http://papas.iseclab.org`

In summary, the paper makes the following contributions:

- We present the first automated approach for the detection of HPP vulnerabilities in web applications. Our approach consists of a component to inject parameters into web applications and a set of tests and heuristics to determine if the pages that are generated contain HPP vulnerabilities.
- We describe the architecture and implementation of the prototype of our approach that we call PAPAS (PArameter Pollution Analysis System). PAPAS is able to crawl websites and generate a list of HPP vulnerable URLs.
- We present and discuss the large-scale, real-world experiments we conducted with more than 5,000 popular websites. Our experiments show that HPP vulnerabilities are prevalent on the web and that many well-known, major websites are affected. We verified that at least 46.8% of the vulnerabilities we discovered could be exploited on the client-side. Our empirical results suggest that, just like in the early days of cross site scripting and cross site request forgery [1], many developers are not aware of the HPP problem, or that they do not take it seriously.

The paper is structured as follows: The next section give an explanation of parameter pollution attacks and provides examples. Section 3 describes our approach and presents the main components of PAPAS. Section 4 presents and discusses the evaluation of PAPAS. Section 5 lists related work, and Section 6 briefly concludes the paper.

2 HTTP Parameter Pollution Attacks

HTTP Parameter Pollution attacks (HPP) have only recently been presented and discussed [27], and have not received much attention so far. An HPP vulnerability allows an attacker to inject a parameter inside the URLs generated by a web application. The consequences of the attack depend on the application's logic, and may vary from a simple annoyance to a complete corruption of the application's behavior. Because this class of web vulnerability is not widely known and well-understood yet, in this section, we first explain and discuss the problem.

Even though injecting a new parameter can sometimes be enough to exploit an application, the attacker is usually more interested in *overriding* the value of an already existing parameter. This can be achieved by "masking" the old parameter by introducing a new one with the same name.

For this to be possible, it is necessary for the web application to "misbehave" in the presence of duplicated parameters, a problem that is often erroneously confused with the HPP vulnerability itself. However, since parameter pollution attacks often rely on duplicated parameters in practice, we decided to study the parameter duplication behavior of applications, and measure it in our experiments.

2.1 Parameter Precedence in Web Applications

During the interaction with a web application, the client often needs to provide input to the program that generates the requested web page (e.g., a PHP or a Perl script). The HTTP protocol [12] allows the user's browser to transfer information inside the URI itself (i.e., GET parameters), in the HTTP headers (e.g., in the Cookie field), or inside the request body (i.e., POST parameters). The adopted technique depends on the application and on the type and amount of data that has to be transferred.

For the sake of simplicity, in the following, we focus on GET parameters. However, note that HPP attacks can be launched against any other input vector that may contain parameters controlled by the user.

RFC 3986 [7] specifies that the query component (or query string) of a URI is the part between the "`?`" character and the end of the URI (or the character "#"). The query string is passed unmodified to the application, and consists of one or more `field=value` pairs, separated by either an ampersand or a semicolon character. For example, the URI `http://host/path/somepage.pl?name=john &age=32` invokes the `verify.pl` script, passing the values `john` for the `name` parameter and the value 32 for the `age` parameter. To avoid conflicts, any special characters (such as the question mark) inside a parameter value must be encoded in its `%FF` hexadecimal form.

This standard technique for passing parameters is straightforward and is generally well-understood by web developers. However, the way in which the query string is processed to extract the single values depends on the application, the technology, and the development language that is used.

For example, consider a web page that contains a check-box that allows the user to select one or more options in a form. In a typical implementation, all the check-box items share the same name, and, therefore, the browser will send a separate homonym parameter for each item selected by the user. To support this functionality, most of the programming languages used to develop web applications provide methods for retrieving the complete list of values associated with a certain parameter. For example, the JSP `getParameterValues` method groups all the values together, and returns them as a list of strings. For the languages that do not support this functionality, the developer has to manually parse the query string to extract each single value.

However, the problem arises when the developer expects to receive a single item and, therefore, invokes methods (such as `getParameter` in JSP) that only return a single value. In this case, if more than one parameter with the same name is present in the query string, the one that is returned can either be the first, the last, or a combination of all the values. Since there is no standard behavior in this situation, the exact result depends on the combination of the programming language that is used, and the web server that is being deployed. Table 1 shows examples of the parameter precedence adopted by different web technologies.

Note that the fact that only one value is returned is not a vulnerability per se. However, if the developer is not aware of the problem, the presence of duplicated parameters can produce an anomalous behavior in the application that can be potentially exploited by an attacker in combination with other attacks. In fact, as we explain in the next section, this is often used in conjunction with HPP vulnerabilities to override hard-coded parameter values in the application's links.

2.2 Parameter Pollution

An HTTP Parameter Pollution (HPP) attack occurs when a malicious parameter P_{inj}, preceded by an encoded query string delimiter, is injected into an existing parameter P_{host}. If P_{host} is not properly sanitized by the application and its value is later decoded and used to generate a URL A, the attacker is able to add one or more new parameters to A.

The typical client-side scenario consists of persuading a victim to visit a malicious URL that exploits the HPP vulnerability. For example, consider a web application that allows users to cast their vote on a number of different elections. The application, written in JSP, receives a single parameter, called `poll_id`, that uniquely identifies the election the user is participating in. Based on the value of the parameter, the application generates a page that includes one link for each candidate. For example, the following snippet shows an election page with two candidates where the user could cast her vote by clicking on the desired link:

```
Url: http://host/election.jsp?poll_id=4568

Link1: <a href="vote.jsp?poll_id=4568&candidate=white">
        Vote for Mr. White</a>
Link2: <a href="vote.jsp?poll_id=4568&candidate=green">
        Vote for Mrs. Green</a>
```

Suppose that Mallory, a Mrs. Green supporter, is interested in subverting the result of the online election. By analyzing the webpage, he realizes that the application does not properly sanitize the `poll_id` parameter. Hence, Mallory

Technology/Server	Tested Method	Parameter Precedence
ASP/IIS	`Request.QueryString("par")`	All (comma-delimited string)
PHP/Apache	`$_GET["par"]`	Last
JSP/Tomcat	`Request.getParameter("par")`	First
Perl(CGI)/Apache	`Param("par")`	First
Python/Apache	`getvalue("par")`	All (List)

Table 1: Parameter precedence in the presence of multiple parameters with the same name

can use the HPP vulnerability to inject another parameter of his choice. He then creates and sends to Alice the following malicious Url:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen
```

Note how Mallory "polluted" the `poll_id` parameter by injecting into it the `candidate=green` pair. By clicking on the link, Alice is redirected to the original election website where she can cast her vote for the election. However, since the `poll_id` parameter is URL-decoded and used by the application to construct the links, when Alice visits the page, the malicious `candidate` value is injected into the URLs[2]:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen

Link 1: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=white>Vote for Mr. White</a>
Link 2: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=green>Vote for Mrs. Green</a>
```

No matter which link Alice clicks on, the application (in this case the `vote.jsp` script) will receive two `candidate` parameters. Furthermore, the first parameter will *always* be set to `green`.

In the scenario we discussed, it is likely that the developer of the voting application expected to receive only one candidate name, and, therefore, relied on the provided basic Java functionality to retrieve a single parameter. As a consequence, as shown in Table 1, only the first value (i.e., `green`) is returned to the program, and the second value (i.e., the one carrying the Alice's actual vote) is discarded.

In summary, in the example we presented, since the voting application is vulnerable to HPP, it is possible for an attacker to forge a malicious link that, once visited, tampers with the content of the page, and returns only links that force a vote for Mrs. Green.

Cross-Channel Pollution HPP attacks can also be used to override parameters between different input channels. A good security practice when developing a web application is to accept parameters only from the input channel (e.g., GET, POST, or Cookies) where they are supposed to be supplied. That is, an application that receives data from a POST request should not accept the same parameters if they are provided inside the URL. In fact, if this safety rule is ignored, an attacker could exploit an HPP flaw to inject arbitrary parameter-value pairs into a channel A to override the legitimate parameters that are normally provided in another channel B. Obviously, for this to be possible, a necessary condition is that the web technology gives precedence to A with respect to B.

HPP to bypass CSRF tokens One interesting use of HPP attacks is to bypass the protection mechanism used to prevent cross-site request forgery. A cross-site request forgery (CRSF) is a confused deputy type of attack [16] that works by including a malicious link in a page (usually in an image tag) that points to a website in which the victim is supposed to be authenticated. The attacker places parameters into the link that are required to initiate an unauthorized action. When the victim visits the attack page, the target application receives the malicious request. Since the request comes from a legitimate user and includes the cookie associated with a valid session, the request is likely to be processed.

A common technique to protect web applications against CSRF attacks consists of using a secret request token (e.g., see [20, 25]). A unique token is generated by the application and inserted in all the sensitive links URLs. When the application receives a request, it verifies that it contains the valid token before authorizing the action. Hence, since the attacker cannot predict the value of the token, she cannot forge the malicious URL to initiate the action.

A parameter pollution vulnerability can be used to inject parameters inside the existing links generated by the application (that, therefore, include a valid secret token). With these injected parameters, it may be possible for the attacker to initiate a malicious action and bypass CSRF protection.

A CSRF bypassing attack using HPP was demonstrated in 2009 against Yahoo Mail [10]. The parameter injection permitted to bypass the token protections adopted by Yahoo to protect sensitive operations, allowing the attacker to

[2]URLs in the page snippets have the injected string emphasized by using a red, underlining font.

delete all the mails of a user.

The following example demonstrates a simplified version of the Yahoo attack:

```
Url:
showFolder?fid=Inbox&order=down&tt=24&pSize=25&startMid=0
%2526cmd=fmgt.emptytrash%26DEL=1%26DelFID=Inbox%26
cmd=fmgt.delete

Link:
showMessage?sort=date&order=down&startMid=0
%26cmd%3Dfmgt.emptytrash&DEL=1&DelFID=Inbox&
cmd=fmgt.delete&.rand=1076957714
```

In the example, the link to display the mail message is protected by a secret token that is stored in the `.rand` parameter. This token prevents an attacker from including the link inside another page to launch a CSRF attack. However, by exploiting an HPP vulnerability, the attacker can still inject the malicious parameters (i.e., deleting all the mails of a user and emptying the trash can) into the legitimate page. The injection string is a concatenation of the two commands, where the second command needs to be URL-encoded twice in order to force the application to clean the trash can only after the deletion of the mails.

3 Automated HPP Vulnerability Detection with PAPAS

Our PArameter Pollution Analysis System (PAPAS) to automatically detect HPP vulnerabilities in websites consists of four main components: A browser, a crawler, and two scanners.

The first component is an instrumented browser that is responsible for fetching the webpages, rendering the content, and extracting all the links and form URLs contained in the page.

The second component is a crawler that communicates with the browser through a bidirectional channel. This channel is used by the crawler to inform the browser on the URLs that need to be visited, and on the forms that need to be submitted. Furthermore, the channel is also used to retrieve the collected information from the browser.

Every time the crawler visits a page, it passes the extracted information to the two scanners so that it can be analyzed. The parameter Precedence Scanner (P-Scan) is responsible for determining how the page behaves when it receives two parameters with the same name. The Vulnerability Scanner (V-Scan), in contrast, is responsible for testing the page to determine if it is vulnerable to HPP attacks. V-Scan does this by attempting to inject a new parameter inside one of the existing ones and analyzing the output. The two scanners also communicate with the instrumented browser in order to execute the tests.

All the collected information is stored in a database that is later analyzed by a statistics component that groups together information about the analyzed pages, and generates a report for the vulnerable URLs.

The general architecture of the system is summarized in Figure 1. In the following, we describe the approach that is used to detect HPP vulnerabilities and each component in more detail.

3.1 Browser and Crawler Components

Whenever the crawler issues a command such as the visiting of a new webpage, the instrumented browser in PAPAS first waits until the target page is loaded. After the browser is finished parsing the DOM, executing the client-side scripts, and loading additional resources, a browser extension (i.e., plugin) extracts the content, the list of links, and the forms in the page.

In order to increase the depth that a website can be scanned with, the instrumented browser in PAPAS uses a number of simple heuristics to automatically fill forms (similarly to previously proposed scanning solutions such as [24]). For example, random alphanumeric values of 8 characters are inserted into `password` fields and a default e-mail address is inserted into fields with the name `email`, `e-mail`, or `mail`.

For sites where the authentication or the provided inputs fail (e.g., because of the use of CAPTCHAs), the crawler can be assisted by manually logging into the application using the browser, and then specifying a regular expression to be used to prevent the crawler from visiting the log-out page (e.g., by excluding links that include the `cmd=logout` parameter).

3.2 P-Scan: Analysis of the Parameter Precedence

The P-Scan component analyzes a page to determine the precedence of parameters if multiple occurrences of the same parameter are injected into an application. For URLs that contain several parameters, each one is analyzed until the page's precedence has been determined or all available parameters have been tested.

The algorithm we use to test the precedence of parameters starts by taking the first parameter of the URL (in the form `par1=val1`), and generates a new parameter value `val2` that is similar to the existing one. The idea is to generate a value that would be accepted as being valid by the application. For example, a parameter that represents a page number cannot be replaced with a string. Hence, a number is cloned into a consecutive number, and a string is cloned into a same-length string with the first two characters modified.

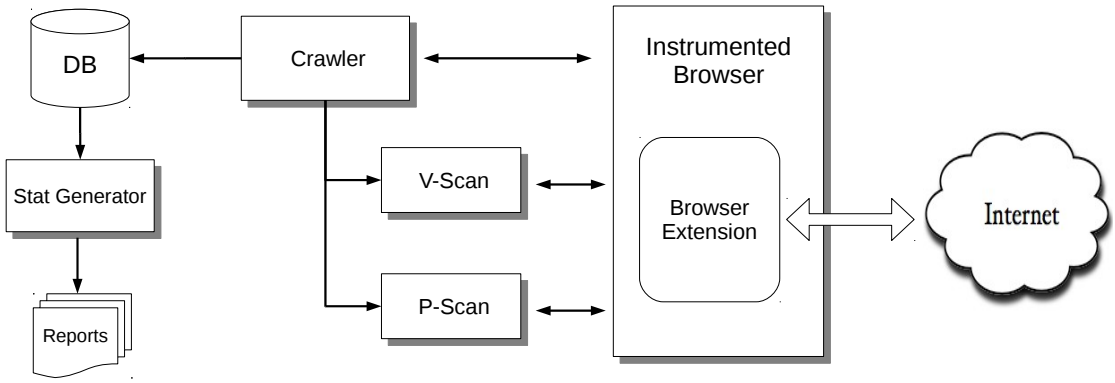


Figure 1: Architecture of PAPAS

In a second step, the scanner asks the browser to generate two new requests. The first request contains only the newly generated value `val2`. In contrast, the second request contains two copies of the parameter, one with the original value `val1`, and one with the value `val2`.

Suppose, for example, that a page accepts two parameters $par1$ and $par2$. In the first iteration, the first parameter is tested for the precedence behavior. That is, a new value new_val is generated and two requests are issued. In sum, the parameter precedence test is run on that pages that are the results of the three following requests:

```
Page0 - Original Url: application.php?
                        par1=val1&par2=val2
Page1 - Request 1:    application.php?
                        par1=new_val&par2=val2
Page2 - Request 2:    application.php?
                        par1=val1&par1=new_val&par2=val2
```

A naive approach to determine the parameter precedence would be to simply compare the three pages returned by the previous requests: If `Page1 == Page2`, then the second (last) parameter would have precedence over the first. If, however, `Page2 == Page0`, the application is giving precedence to the first parameter over the second.

Unfortunately, this straightforward approach does not work well in practice. Modern web applications are very complex, and often include dynamic content that may still vary even when the page is accessed with exactly the same parameters. Publicity banners, RSS feeds, real-time statistics, gadgets, and suggestion boxes are only a few examples of the dynamic content that can be present in a page and that may change each time the page is accessed.

The P-Scan component resolves the dynamic content problem in two stages. First, it pre-processes the page and tries to eliminate all dynamic content that does not depend on the values of the application parameters. That is, P-Scan removes HTML comments, images, embedded contents, interactive objects (e.g., Java applets), CSS stylesheets, cross-domain iFrames, and client-side scripts. It also uses regular expressions to identify and remove "timers" that are often used to report how long it takes to generate the page that is being accessed. In a similar way, all the date and time strings on the page are removed.

The last part of the sanitization step consists of removing all the URLs that reference the page itself. The problem is that as it is very common for form actions to submit data to the same page, when the parameters of a page are modified, the self-referencing URLs also change accordingly. Hence, to cope with this problem, we also eliminate these URLs.

After the pages have been stripped off their dynamic components, P-Scan compares them to determine the precedence of the parameters. Let `P0'`, `P1'`, and `P2'` be the sanitized versions of `Page0`, `Page1`, and `Page2`. The comparison procedure consists of five different tests that are applied until one of the tests succeeds:

I. Identity Test - The identity test checks whether the parameter under analysis has any impact on the content of the page. In fact, it is very common for query strings to contain many parameters that only affect the internal state, or some "invisible" logic of the application. Hence, if `P0' == P1' == P2'`, the parameter is considered to be ineffective.

II. Base Test - The base test is based on the assumption that the dynamic component stripping process is able to perfectly remove all dynamic components from the page that is under analysis. If this is the case, the second (last) parameter has precedence over the first if `P1'==P2'`. The situation is the opposite if `P2' == P0'`. Note that despite our efforts to improve the dynamic content stripping process as much as possible, in practice, it is rarely the case that the compared pages match perfectly.

III. Join Test - The join test checks the pages for indica-

tions that show that the two values of the homonym parameters are somehow combined together by the application. For example, it searches `P2'` for two values that are separated by commas, spaces, or that are contained in the same HTML tag. If there is a positive match, the algorithm concludes that the application is merging the values of the parameters.

IV. Fuzzy Test - The fuzzy test is designed to cope with pages whose dynamic components have not been perfectly sanitized. The test aims to handle identical pages that may show minor differences because of embedded dynamic parts. The test is based on confidence intervals. We compute two values, S_{21} and S_{20}, that represent how similar `P2'` is to the pages `P1'` and `P0'` respectively. The similarity algorithm we use is based on the Ratcliff/Obershelp pattern recognition algorithm, (also known as *gestalt pattern matching* [28]), and returns a number between 0 (i.e, completely different) to 1 (i.e., perfect match). The parameter precedence detection algorithm that we use in the fuzzy test works as follows:

```
if ABS(S21-S20) > DISCRIMINATION_THRESHOLD:
  if (S21 > S20) and (S21 > SIMILARITY_THRESHOLD):
    Precedence = last
  else (S20 > S21) and (S20 > SIMILARITY_THRESHOLD):
    Precedence = first
  else:
    Unknown precedence
else:
  Unknown precedence
```

To draw a conclusion, the algorithm first checks if the two similarity values are different enough (i.e., the values show a difference that is greater than a certain *discrimination threshold*). If this is the case, the closer match (if the similarity is over a minimum *similarity threshold*) determines the parameter precedence. In other words, if the page with the duplicated parameters is very similar to the original page, there is a strong probability that the web application is only using the first parameter, and ignoring the second. However, if the similarity is closer to the page with the artificially injected parameter, there is a strong probability that the application is only accepting the second parameter.

The two threshold values have been determined by running the algorithm on one hundred random webpages that failed to pass the base test, and for which we manually determined the precedence of parameters. The two experimental thresholds (set respectively to 0.05 and 0.75) were chosen to maximize the accuracy of the detection, while minimizing the error rate.

V. Error Test - The error test checks if the application crashes, or returns an "internal" error when an identical parameter is injected multiple times. Such an error usually happens when the application does not expect to receive multiple parameters with the same name. Hence, it receives an array (or a list) of parameters instead of a single value. An error occurs if the value is later used in a function that expects a well-defined type (such as a number or a string). In this test, we search the page under analysis for strings that are associated with common error messages or exceptions. In particular, we adopted all the regular expressions that the *SqlMap project* [13] uses to identify database errors in MySQL, PostgreSQL, MS SQL Server, Microsoft Access, Oracle, DB2, and SQLite.

If none of these five tests succeed, the parameter is discarded from the analysis. This could be, for example, because of content that is generated randomly on the server-side. The parameter precedence detection algorithm is then run again on the next available parameter.

3.3 V-Scan: Testing for HPP vulnerabilities

In this section, we describe how the V-Scan component tests for the presence of HTTP Parameter Pollution vulnerabilities in web applications.

For every page that V-Scan receives from the crawler, it tries to inject a URL-encoded version of an innocuous parameter into each existing parameter of the query string. Then, for each injection, the scanner verifies the presence of the parameter in links, action fields and hidden fields of forms in the answer page.

For example, in a typical scenario, V-Scan injects the pair "`%26foo%3Dbar`" into the parameter "`par1=val1`" and then checks if the "`&foo=bar`" string is included inside the URLs of links or forms in the answer page.

Note that we do not check for the presence of the vulnerable parameter itself (e.g., by looking for the string "`par1=val1&foo=bar`"). This is because web applications sometimes use a different name for the same parameter in the URL and in the page content. Therefore, the parameter "`par1`" may appear under a different name inside the page.

In more detail, V-Scan starts by extracting the list $P_{URL} = [P_{U1}, P_{U2}, \ldots P_{Un}]$ of the parameters that are present in the page URL, and the list $P_{Body} = [P_{B1}, P_{B2}, \ldots P_{Bm}]$ of the parameters that are present in links or forms contained in the page body.
It then computes the following three sets:

- $P_A = P_{URL} \cap P_{Body}$ is the set of parameters that appear unmodified in the URL and in the links or forms of the page.
- $P_B = p \mid p \in P_{URL} \wedge p \notin P_{Body}$ contains the URL parameters that do not appear in the page. Some

of these parameters may appear in the page under a different name.

- $P_C = p \mid p \notin P_{URL} \wedge p \in P_{Body}$ is the set of parameters that appear somewhere in the page, but that are not present in the URL.

First, V-Scan starts by injecting the new parameter in the P_A set. We observed that in practice, in the majority of the cases, the application copies the parameter to the page body and maintains the same name. Hence, there is a high probability that a vulnerability will be identified at this stage. However, if this test does not discover any vulnerability, then the scanner moves on to the second set (P_B). In the second test, the scanner tests for the (less likely) case in which the vulnerable parameter is renamed by the application. Finally, in the final test, V-Scan takes the parameters in the P_C group, attempts to add these to the URL, and use them as a vector to inject the malicious pair. This is because webpages usually accept a very large number of parameters, not all of which are normally specified in the URL. For example, imagine a case in which we observe that one of the links in the page contains a parameter "`language=en`". Suppose, however, that this parameter is not present in the page URL. In the final test, V-Scan would attempt to build a query string like "`par1=var1&language=en%26foo%3Dbar`".

Note that the last test V-Scan applies can be executed on pages with an empty query string (but with parameterized links/forms), while the first two require pages that already contain a query string.

In our prototype implementation, the V-Scan component encodes the attacker pair using the standard URL encoding schema[3]. Our experiments show that this is sufficient for discovering HPP flaws in many applications. However, there is room for improvement as in some cases, the attacker might need to use different types of encodings to be able to trigger a bug. For example, this was the case of the HPP attack against Yahoo (previously described in Section 2) where the attacker had to double URL-encode the "cleaning of the trash can" action.

Handling special cases In our experiments, we identified two special cases in which, even though our vulnerability scanner reported an alert, the page was not actually vulnerable to parameter pollution.

In the first case, one of the URL parameters (or part of it) is used as the *entire* target of a link. For example:

```
Url:  index.php?v1=p1&uri=apps%2Femail.jsp%3Fvar1%3Dpar1
                          %26foo%3Dbar
Link: apps/email.jsp?var1=par1&foo=bar
```

[3]URL Encoding Reference, `http://www.w3schools.com/TAGS/ref_urlencode.asp`

A parameter is used to store the URL of the target page. Hence, performing an injection in that parameter is equivalent to modifying its value to point to a different URL. Even though this technique is syntactically very similar to an HPP vulnerability, it is not a proper injection case. Therefore, we decided to consider this case as a false positive of the tool.

The second case that generates false alarms is the opposite of the first case. In some pages, the entire URL of the page becomes a parameter in one of the links. This can frequently be observed in pages that support printing or sharing functionalities. For example, imagine an application that contains a link to report a problem to the website's administrator. The link contains a parameter `page` that references the URL of the page responsible for the problem:

```
Url: search.html?session_id=jKAmSZx5%26foo%3Dbar&q=shoes

Link: service_request.html?page=search%2ehtml%3f
      session_id%3djKAmSZx5&foo=bar&q=shoes
```

Note that by changing the URL of the page, we also change the `page` parameter contained in the link. Clearly, this is not an HPP vulnerability.

Since the two previous implementation techniques are quite common in web applications, PAPAS would erroneously report these sites as being vulnerable to HPP. To eliminate such alarms and to make PAPAS suitable for large-scale analysis, we integrated heuristics into the V-Scan component to cross-check and verify that the vulnerabilities that are identified do not correspond to these two common techniques that are used in practice.

In our prototype implementation, in order to eliminate these false alarms, V-Scan checks that the parameter in which the injection is performed does not start with a scheme specifier string (e.g., `http://`). Then, it verifies that the parameter as a whole is not used as the target for a link. Furthermore, it also checks that the entire URL is not copied as a parameter inside a link. Finally, our vulnerability analysis component double-checks each vulnerability by injecting the new parameter *without* url-encoding the separator (i.e., by injecting `&foo=bar` instead of `%26foo%3Dbar`). If the result is the same, we know that the query string is simply copied inside another URL. While such input handling is possibly a dangerous design decision on the side of the developer, there is a high probability that it is intentional so we ignore it and do not report it by default. However, such checks can be deactivated anytime if the analyst would like to perform a more in-depth analysis of the website.

3.4 Implementation

The browser component of PAPAS is implemented as a Firefox extension, while the rest of the system is written in Python. The components communicate over TCP/IP sockets.

Similar to other scanners, it would have been possible to directly retrieve web pages without rendering them in a real browser. However, such techniques have the drawback that they cannot efficiently deal with dynamic content that is often found on Web pages (e.g., Javascript). By using a real browser to render the pages we visit, we are able to analyze the page as it is supposed to appear to the user after the dynamic content has been generated. Also, note that unlike detecting cross site scripting or SQL injections, the ability to deal with dynamic content is a necessary prerequisite to be able to test for HPP vulnerabilities using a black-box approach.

The browser extension has been developed using the standard technology offered by the Mozilla development environment: a mix of Javascript and XML User Interface Language (XUL). We use XPConnect to access Firefox's XPCOM components. These components are used for invoking GET and POST requests and for communicating with the scanning component.

PAPAS supports three different operational modes: *fast mode*, *extensive mode* and *assisted mode*. The fast mode aims to rapidly test a site until potential vulnerabilities are discovered. Whenever an alert is generated, the analysis continues, but the V-Scan component is not invoked to improve the scanning speed. In the extensive mode, the entire website is tested exhaustively and all potential problems and injections are logged. The assisted mode allows the scanner to be used in an interactive way. That is, the crawler pauses and specific pages can be tested for parameter precedence and HPP vulnerabilities. The assisted mode can be used by security professionals to conduct a semi-automated assessment of a web application, or to test websites that require a particular user authentication.

PAPAS is also customizable and settings such as scanning depths, numbers of injections that are performed, waiting times between requests, and page loading timeouts are all configurable by the analyst.

3.5 Limitations

Our current implementation of PAPAS has several limitations. First, PAPAS does not support the crawling of links embedded in active content such as Flash, and therefore, is not able to visit websites that rely on active content technologies to navigate among the pages.

Second, currently, PAPAS focuses only on HPP vulnerabilities that can be exploited via client-side attacks (e.g., analogous to reflected XSS attacks) where the user needs to click on a link prepared by the attacker. Some HPP vulnerabilities can also be used to exploit server-side components (when the malicious parameter value is not included in a link but it is decoded and passed to a back-end component). However, testing for server-side attacks is more difficult than testing for client-side attacks as comparing requests and answers is not sufficient (i.e., similar to the difficulty of detecting stored SQL-injection vulnerabilities via black-box scanning). We leave the detection of server-side attacks to future work.

4 Evaluation

We evaluated our detection technique by running two experiments. In the first experiment, we used PAPAS to automatically scan a list of popular websites with the aim of measuring the prevalence of HPP vulnerabilities in the wild. We then selected a limited number of vulnerable sites and, in a second experiment, performed a more in-depth analysis of the detected vulnerabilities to gain a better understanding of the possible consequences of the vulnerabilities our tool automatically identified.

4.1 HPP Prevalence in Popular Websites

In the first experiment, we collected 5,000 unique URLs from the public database of Alexa. In particular, we extracted the top ranked sites from each of the *Alexa's categories* [3]. Each website was considered only once – even if it was present in multiple distinct categories, or with different top-level domain names such as `google.com` and `google.fr`.

The aim of our experiments was to quickly scan as many websites as possible. Our basic premise was that it would be likely that the application would contain parameter injection vulnerabilities on many pages and on a large number of parameters if the developers of the site were not aware of the HPP threat and had failed to properly sanitize the user input.

To maximize the speed of the tests, we configured the crawler to start from the homepage and visit the sub-pages up to a distance of three (i.e., three clicks away from the website's entry point). For the tests, we only considered links that contained at least one parameter. In addition, we limited the analysis to 5 instances per page (i.e., a page with the same URL, but a different query string was considered a new instance). The global timeout was set to 15 minutes per site and the browser was customized to quickly load and render the pages, and run without any user interaction. Furthermore, we disabled pop-ups, image loading, and any plug-ins for active content technologies such as Flash, or

Categories	# of Tested Applications	Categories	# of Tested Applications
Internet	698	Government	132
News	599	Social Networking	117
Shopping	460	Video	114
Games	300	Financial	110
Sports	256	Organization	106
Health	235	University	91
Science	222	Others	1401
Travel	175		

Table 2: TOP15 categories of the analyzed sites

Silverlight. An external watchdog was also configured to monitor and restart the browser in case it became unresponsive.

In 13 days of experiments, we successfully scanned 5,016 websites, corresponding to a total of 149,806 unique pages. For each page, our tool generated a variable amount of queries, depending on the number of detected parameters. The websites we tested were distributed over 97 countries and hundreds of different Alexa categories. Table 2 summarizes the 15 categories containing the higher number of tested applications.

Parameter Precedence For each website, the P-Scan component tested every page to evaluate the order in which the GET parameters were considered by the application when two occurrences of the same parameter were specified. The results were then grouped together in a per-site summary, as shown in Figure 2. The first column reports the type of parameter precedence. *Last* and *First* indicate that all the analyzed pages of the application uniformly considered the last or the first specified value. *Union* indicates that the two parameters were combined together to form a single value, usually by simply concatenating the two strings with a space or a comma. In contrast, the parameter precedence is set to *inconsistent* when different pages of the website present mismatching precedences (i.e., some pages favor the first parameter's value, others favor the last). The *inconsistent* state, accounting for a total of 25% of the analyzed applications, is usually a consequence of the fact that the website has been developed using a combination of heterogeneous technologies. For example, the main implementation language of the website may be PHP, but a few Perl scripts may still be responsible for serving certain pages.

Even though the lack of a uniform behavior can be suspicious, it is neither a sign, nor a consequence of a vulnerable application. In fact, each parameter precedence behavior (even the *inconsistent* case) is perfectly safe if the application's developers are aware of the HPP threat and know how to handle a parameter's value in the proper way. Unfortunately, as shown in the rest of the section, the results of our experiments suggest that many developers are not aware of HPP.

Figure 2 shows that for 4% of the websites we analyzed, our scanner was not been able to automatically detect the parameter precedence. This is usually due to two main reasons. The first reason is that the parameters do not affect (or only minimally affect) the rendered page. Therefore, the result of the page comparison does not reach the discrimination threshold. The second reason is the opposite of the first. That is, the page shows too many differences even after the removal of the dynamic content, and the result of the comparison falls below the similarity threshold (see Section 3.2 for the full algorithm and an explanation of the threshold values).

The scanner found 238 applications that raised an SQL error when they were tested with duplicated parameters. Quite surprisingly, almost 5% of the most popular websites on the Internet failed to properly handle the user input, and returned an "internal" error page when a perfectly-legal parameter was repeated twice. Note that providing two parameters with the same name is a common practice in many applications, and most of the programming languages provide special functionalities to access multiple values. Therefore, this test was not intended to be an attack against the applications, but only a check to verify which parameter's value was given the precedence. Nevertheless, we were surprised to note error messages from the websites of many major companies, banks and government institutions, educational sites, and others popular websites.

HPP Vulnerabilities PAPAS discovered that 1499 websites (29.88% of the total we analyzed) contained at least one page vulnerable to HTTP Parameter Injection. That is, the tool was able to automatically inject an encoded parameter inside one of the existing parameters, and was then able to verify that its URL-decoded version was included in one of the URLs (links or forms) of the resulting page.

However, the fact that it is possible to inject a parameter

Parameter Precedence	WebSites	
Last	2,237	(44.60%)
First	946	(18.86%)
Union	381	(7.60%)
Inconsistent	1,251	(24.94%)
Unknown	201	(4.00%)
Total	5,016	(100.00%)
Database Errors	238	(4.74%)

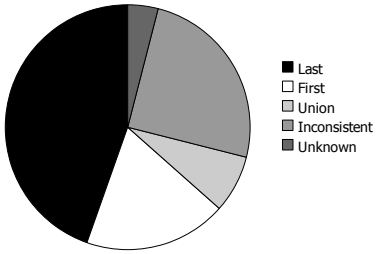


Figure 2: Precedence when the same parameter occurs multiple time

does not reveal information about the significance and the consequences of the injection. Therefore, we attempted to verify the number of *exploitable* applications (i.e., the subset of vulnerable websites in which the injected parameter could potentially be used to modify the behavior of the application).

We started by splitting the vulnerable set into two separate groups. In 872 websites (17.39%), the injection was on a link or a form's action field. In the remaining 627 cases (12.5%), the injection was on a form's hidden field.

For the first group, our tool verified if the parameter injection vulnerability could be used to override the value of one of the existing parameters in the application. This is possible only if the parameter precedence of the page is consistent with the position of the injected value. For example, if the malicious parameter is always added to the end of the URL and the first value has parameter precedence, it is impossible to override any existing parameter.

When the parameter precedence is not favorable, a vulnerable application can still be exploitable by injecting a new parameter (that differs from all the ones already present in the URL) that is accepted by the target page.

For example, consider a page `target.pl` that accepts an `action` parameter. Suppose that, on the same page, we find a page `poor.pl` vulnerable to HPP:

```
Url:  poor.pl?par1=val1%26action%3Dreset
Link: target.pl?x=y&w=z&par1=val1&action=reset
```

Since in Perl the parameter precedence is on the *first* value, it is impossible to override the `x` and `w` parameters. However, as shown in the example, the attacker can still exploit the application by injecting the `action` parameter that she knows is accepted by the `target.pl` script. Note that while the parameter overriding test was completely automated, this type of injection required a manual supervision to verify the effects of the injected parameter on the web application.

The final result was that at least 702 out of the 872 applications of the first group were exploitable. For the remaining 170 pages, we were not able, through a parameter injection, to affect the behavior of the application.

For the applications in the second group, the impact of the vulnerability is more difficult to estimate in an automated fashion. In fact, since modern browsers automatically encode all the form fields, the injected parameter will still be sent in a url-encoded form, thus making an attack ineffective.

In such a case, it may still be possible to exploit the application using a two-step attack where the malicious value is injected into the vulnerable field, it is propagated in the form submission, and it is (possibly) decoded and used in a later stage. In addition, the vulnerability could also be exploited to perform a server-side attack, as explained in Section 3.5. However, using a black-box approach, it is very difficult to automatically test the exploitability of multi-step or server-side vulnerabilities. Furthermore, server-side testing might have had ethical implications (see Section 4.3 for discussion). Therefore, we did not perform any further analysis in this direction.
To conclude, we were able to confirm that in (at least) 702 out of the 1499 vulnerable websites (i.e., 46.8%) that PAPAS identified, it would have been possible to exploit the HPP vulnerability to override one of the hard-coded parameters, or to inject another malicious parameter that would affect the behavior of the application.

Figure 3 shows the fraction of vulnerable and exploitable applications grouped by the different Alexa categories. The results are equally divided, suggesting that important financial and health institutions do not seem to be more security-aware and immune to HPP than leisure sites for sporting and gaming.

False Positives In our vulnerability detection experiments, the false positives rate was 1.12% (10 applications). All the false alarms were due to parameters that were used by the application as an entire target for one of the links. The heuristic we implemented to detect these cases (explained in Section 3.3) failed because the applications applied a transformation to the parameter before using it as a

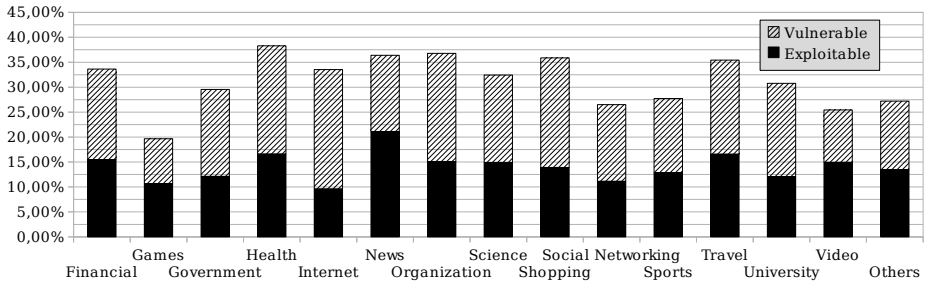


Figure 3: Vulnerability rate for category

link's URL.

Note that, to maximize efficiency, our results were obtained by crawling each website at a maximum depth of three pages. In our experiments, we observed that 11% of the vulnerable pages were directly linked from the homepage, while the remaining 89% were equally distributed between the distance of 2 and 3. This trend suggests that it is very probable that many more vulnerabilities could have been found by exploring the sites in more depth.

4.2 Examples of Discovered Vulnerabilities

Our final experiments consisted of the further analysis of some of the vulnerable websites that we identified. Our aim was to gain an insight into the real consequences of the HPP vulnerabilities we discovered.

The analysis we performed was assisted by the V-Scan component. When invoked in *extensive mode*, V-Scan was able to explore in detail the web application, enumerating all the vulnerable parameters. For some of the websites, we also registered an account and configured the scanner to test the authenticated part of the website.

HPP vulnerabilities can be abused to run a wide range of different attacks. In the rest of this section, we discuss the different classes of problems we identified in our analysis with the help of real-world examples.

The problems we identified affected many important and well-known websites such as Microsoft, Google, VMWare, About.com, Symantec, history.com, flickr, and Paypal. Since, at the time of writing, we have not yet received confirmation that all of the vulnerabilities have been fixed, we have anonymized the description of the following real-word cases.

Facebook Share Facebook, Twitter, Digg and other social networking sites offer a *share component* to easily share the content of a webpage over a user profile. Many news portals nowadays integrate these components with the intent of facilitating the distribution of their news.

By reviewing the vulnerability logs of the tested applications, we noticed that different sites allowed a parameter injection on the links referencing the share component of Facebook. In all those cases, a vulnerable parameter would allow an attacker to alter the request sent to Facebook and to trick the victim into sharing a page chosen by the attacker. For example, it was possible for an attacker to exploit these vulnerabilities to corrupt a shared link by overwriting the reference with the URL of a drive-by-download website.

In technical terms, the problem was due to the fact that it was possible to inject an extra *url-to-share* parameter that could overwrite the value of the parameter used by the application. For example:

```
Url:
<site>/shareurl.htm?PG=<default url>&zItl=<description>
                %26url-to-share%3Dhttp://www.malicious.com
Link:
http://www.facebook.com/sharer.php?
      url-to-share=<default url>&t=<description>&
      url-to-share=http://www.malicious.com
```

Even though the problem lies with the websites that use the share component, Facebook facilitated the exploitation by accepting multiple instances of the same parameter, and always considering the latest value (i.e., the one on the right).

We notified the security team of Facebook and proposed a simple solution based on the filtering of all incoming sharing requests that include duplicate parameters. The team promptly acknowledged the issue and informed us that they were willing to put in place our countermeasure.

CSRF via HPP Injection Many applications use hidden parameters to store a URL that is later used to redirect the users to an appropriate page. For example, social networks commonly use this feature to redirect new users to a page where they can look up a friend's profile.

In some of these sites, we observed that it was possible for an attacker to inject a new *redirect* parameter inside the registration or the login page so that it could override the

hard-coded parameter's value. On one social-network website, we were able to inject a custom URL that had the effect of automatically sending friend requests after the login. In another site, by injecting the malicious pair into the registration form, an attacker could perform different actions on the authenticated area.

This problem is a CSRF attack that is carried out via an HPP injection. The advantages compared to a normal CSRF is that the attack URL is injected into the real login/registration page. Moreover, the user does not have to be already logged into the target website because the action is automatically executed when the user logs into the application. However, just like in normal CSRF, this attack can be prevented by using security tokens.

Shopping Carts We discovered different HPP vulnerabilities in online shopping websites that allow the attacker to tamper with the user interaction with the shopping cart component.

For example, in several shopping websites, we were able to force the application to select a particular product to be added into the user's cart. That is, when the victim checks out and would like to pay for the merchandise, she is actually paying for a product that is different from the ones she actually selected. On an Italian shopping portal, for example, it was even possible to override the ID of the product in such a way that the browser was still showing the image and the description of the original product, even when the victim was actually buying a different one.

Financial Institutions We ran PAPAS against the authenticated and non-authenticated areas of some financial websites and the tool automatically detected several HPP vulnerabilities that were potentially exploitable. Since the links involved sensitive operations (such as increasing account limits and manipulating credit card operations), we immediately stopped our experiments and promptly informed the security departments of the involved companies. The problems were acknowledged and are currently being fixed.

Tampering with Query Results In most cases, the HPP vulnerabilities that we discovered in our experiments allow the attacker to tamper with the data provided by the vulnerable website, and to present to the victim some information chosen by the attacker.

On several popular news portals, we managed to modify the news search results to hide certain news, to show the news of a certain day with another date, or to filter the news of a specific source/author. An attacker can exploit these vulnerabilities to promote some particular news, or conceal news that can hurt his person/image, or even subvert the information by replacing an article with an older one.

Also some multimedia websites were vulnerable to HPP attacks. In several popular sites, an attacker could override the video links and make them point to a link of his choice (e.g., a drive-by download site), or alter the results of a query to inject malicious multimedia materials. In one case, we were able to automatically register a user to a specific streaming event.

Similar problems also affected several popular search engines. We noticed that it would have been possible to tamper with the results of the search functionality by adding special keywords, or by manipulating the order in which the results are shown. We also noticed that on some search engines, it was possible to replace the content of the commercial suggestion boxes with links to sites owned by the attacker.

4.3 Ethical Considerations

Crawling and automatically testing a large number of applications may be considered an ethically sensitive issue. Clearly, one question that arises is if it is ethically acceptable and justifiable to test for vulnerabilities in popular websites.

Analogous to the real-world experiments conducted by Jakobsson et al. in [21, 22], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real-world. Unfortunately, criminals do not have any second thoughts about discovering vulnerabilities in the wild. As researchers, we believe that our experiments helped many websites to improve their security. Furthermore, we were able to raise some awareness about HPP problems in the community.

Also, note that:

- PAPAS only performed client-side checks. Similar client-side vulnerability experiments have been performed before in other studies (e.g., for detecting cross site scripting, SQL injections, and CSRF in the wild [24, 29]). Furthermore, we did not perform any server-side vulnerability analysis because such experiments had the potential to cause harm.
- We only provided the applications with innocuous parameters that we knew that the applications were already accepting, and did not use any malicious code as input.
- PAPAS was not powerful enough to influence the performance of any website we investigated, and the scan activities was limited to 15 minutes to further reduce the generated traffic.
- We informed the concerned sites of any critical vulnerabilities that we discovered.

- None of the security groups of the websites that we interacted with complained to us when we informed them that we were researchers, and that we had discovered vulnerabilities on their site with a tool that we were testing. On the contrary, many people were thankful to us that we were informing them about vulnerabilities in their code, and helping them make their site more secure.

5 Related work

There are two main approaches [14] to test software applications for the presence of bugs and vulnerabilities: white-box testing and black-box testing. In white-box testing, the source code of an application is analyzed to find flaws. In contrast, in black-box testing, input is fed into a running application and the generated output is analyzed for unexpected behavior that may indicate errors. PAPAS adopts a black-box approach to scan for vulnerabilities.

When analyzing web applications for vulnerabilities, black-box testing tools (e.g., [2, 8, 24, 33]) are the most popular. Some of these tools (e.g., [2]) claim to be generic enough to identify a wide range of vulnerabilities in web applications. However, recent studies ([6, 11]) have shown that scanning solutions that claim to be generic have serious limitations, and that they are not as comprehensive in practice as they pretend to be.

Two well-known, older web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [30] and Huang et al.'s [17] vulnerability detection tool that automatically executes SQL injection attacks. Scott and Sharp's solution allows to define fine-grained policies manually in order to prevent attacks such as parameter tampering and cross-site scripting. However, it cannot prevent HPP attacks and has not been designed with this vulnerability in mind. In comparison, Huang et al.'s work solely focuses on SQL injection vulnerability detection using fault injection.

To the best of our knowledge, only one of the available black-box scanners, Cenzic Hailstorm [9], claims to support HPP detection. However, a study of its marketing material reveals that the tool only looks for behavioral differences when HTTP parameters are duplicated (i.e., not a sufficient test by itself to detect HPP). Unfortunately, we were not able to obtain more information about the inner-workings of the tool as Cenzic did not respond to our request for an evaluation version.

The injection technique we use is similar to other black-box approaches such as SecuBat [24] that aim to discover SQL injection, or reflected cross site scripting vulnerabilities. However, note that conceptually, detecting cross site scripting or SQL injection is different from detecting HPP. In fact, our approach required the development of a set of tests and heuristics to be able to deal with dynamic content that is often found on webpages today (content that is not an issue when testing for XSS or SQL injection). Hence, compared to existing work in literature, our approach for detecting HPP, and the prototype we present in this paper are unique.

With respect to white-box testing of web applications, a large number of static source code analysis tools (e.g., [23, 31, 34]) that aim to identify vulnerabilities have been proposed. These approaches typically employ taint tracking to help discover if tainted user input reaches a critical function without being validated. We believe that static code analysis would be useful and would help developers identify HPP vulnerabilities. However, to be able to use static code analysis, it is still necessary for the developers to understand the concept of HPP. Previous research has shown that the sanitization process can still be faulty if the developer does not understand a certain class of vulnerability [4].

Note that there also exists a large body of more general vulnerability detection and security assessment tools (e.g., Nikto [26], and Nessus [32]). Such tools typically rely on a repository of known vulnerabilities and test for the existence of these flaws. In comparison, our approach aims to discover previously unknown HPP vulnerabilities in the applications that are under analysis.

With respect to scanning, there also exist network-level tools such as nmap [18]. Tools like nmap can determine the availability of hosts and accessible services. However, they cannot detect higher-level application vulnerabilities.

In comparison to the work we present in this paper, to the best of our knowledge, no large-scale study has been performed to date to measure the prevalence and the significance of HPP vulnerabilities in popular websites.

6 Conclusion

Web applications are not what they used to be ten years ago. Popular web applications have now become more dynamic, interactive, complex, and often contain a large number of multimedia components. Unfortunately, as the popularity of a technology increases, it also becomes a target for criminals. As a result, most attacks today are launched against web applications.

Vulnerabilities such as cross site scripting, SQL injection, and cross site request forgery are well-known and have been intensively studied by the research community. Many solutions have been proposed, and tools have been released. However, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) that was first presented at the OWASP conference [27] in 2009 has not received as much attention. If a web application does not properly sanitize the user input for parameter delimiters, using an HPP vulnerability, an attacker can compromise the

logic of the application to perform client-side or server-side attacks.

In this paper, we present the first automated approach for the discovery of HPP vulnerabilities in web applications. Our prototype implementation called PArameter Pollution Analysis System (PAPAS) is able to crawl websites and discover HPP vulnerabilities by parameter injection. In order to determine the feasibility of our approach and to assess the prevalence of HPP vulnerabilities on the Internet today, we analyzed more than 5,000 popular websites. Our results show that about 30% of the sites we analyzed contain vulnerable parameters and that at least 14% of them can be exploited using HPP. A large number of well-known, high-profile websites such as Symantec, Google, VMWare, and Microsoft were among the sites affected by HPP vulnerabilities that we discovered. We informed the sites for which we could obtain contact information, and some of these sites wrote back to us and confirmed our findings.

We hope that this paper will help raise awareness and draw attention to the HPP problem.

Acknowledgments This work has been supported by the POLE de Competitivite SCS (France) through the MECANOS project and by the French National Research Agency through the VAMPIRE project. The work has also received support from the Secure Business Austria in Vienna.

References

[1] C. A. A-2000-02. Malicious HTML Tags Embedded in Client Web Requests, 2000. http://www.cert.org/advisories/CA-2000-02.html.

[2] Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/, 2008.

[3] I. Alexa Internet. Alexa - Top Sites by Category: Top. http://www.alexa.com/topsites/category.

[4] D. Balzarotti, M. Cova, V. Felmetsger, D. Balzarotti, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, 2008.

[5] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *19th International World Wide Web Conference. (WWW 2010)*, 2010.

[6] J. Bau, E. Burzstein, D. Gupta, and J. C. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of IEEE Security and Privacy*, May 2010.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 3986, uniform resource identifier (uri): Generic syntax, 2005. http://rfc.net/rfc3986.html.

[8] Burp Spider. Web Application Security. http://portswigger.net/spider/, 2008.

[9] Cenzic. Cenzic Hailstormr. http://www.cenzic.com/, 2010.

[10] S. di Paola and L. Carettoni. Client side Http Parameter Pollution - Yahoo! Classic Mail Video Poc, May 2009. http://blog.mindedsecurity.com/2009/05/client-side-http-parameter-pollution.html.

[11] A. Doupé, M. Cova, and G. Vigna. Why Johnny Cant Pentest: An Analysis of Black-Box Web Vulnerability Scanners. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, 2010.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. http://www.rfc.net/rfc2616.html.

[13] B. D. A. G. and M. Stampar. sqlmap. http://sqlmap.sourceforge.net.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[15] W. G. J. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.

[16] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[17] Y. Huang, S. Huang, and T. Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *12th World Wide Web Conference*, 2003.

[18] Insecure.org. NMap Network Scanner. http://www.insecure.org/nmap/, 2010.

[19] S. Institute. Top Cyber Security Risks, September 2009. http://www.sans.org/top-cyber-security-risks/summary.php.

[20] A. B. C. Jackson and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *15th ACM Conference on Computer and Communications Security*, 2007.

[21] M. Jakobsson, P. Finn, and N. Johnson. Why and How to Perform Fraud Experiments. *Security & Privacy, IEEE*, 6(2):66–68, March-April 2008.

[22] M. Jakobsson and J. Ratkiewicz. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In *15th International Conference on World Wide Web (WWW)*, 2006.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[24] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.

[25] N. J. E. Kirda and C. Kruegel. Preventing Cross Site Request Forgery Attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm), Baltimore, MD*, 2006.

[26] Nikto. Web Server Scanner. http://www.cirt.net/code/nikto.shtml, 2010.

[27] OWASP AppSec Europe 2009. *HTTP Parameter Pollution*, May 2009. http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf.

[28] J. Ratcliff and D. Metzener. Pattern matching: The gestalt approach. *Dr. Dobbs Journal*, 7:46, 1988.

[29] D. Reading. CSRF Flaws Found on Major Websites: Princeton University researchers reveal four sites with cross-site request forgery flaws and unveil tools to protect against these attacks, 2008. `http://www.darkreading.com/security/app-security/showArticle.jhtml?articleID=211201247`.

[30] D. Scott and R. Sharp. Abstracting Application-level Web Security. *11th World Wide Web Conference*, 2002.

[31] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.

[32] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2010.

[33] Web Application Attack and Audit Framework. `http://w3af.sourceforge.net/`.

[34] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.

Logic Vulnerabilities

6

From Traditional Flaws to Logic Flows

Despite the considerable effort from the security community to mitigate the problem, traditional vulnerabilities due to improper input validation are still a serious flaw. However, they are not the *only* type of vulnerability that affects web applications. Together with researchers at the University of California at Santa Barbara, I worked on the characterization of *logic errors* – a much more subtle type of vulnerability. In addition, I also proposed the first automated approaches (both white- and black-boxes) to detect these vulnerabilities [1]

To date, the class of logic vulnerabilities still lacks a formal definition. However, in practice it is typically the consequence of an insufficient validation of the business process of a web application. The resulting violations may involve both the *control plane* (i.e., the way users can and should navigate between different pages) and the *data plane* (i.e., the way information propagates from one page to another).

Attacks against the control plane exploit the fact that a web application may fail to properly enforce what are the valid sequences of actions that can be performed by the user. For example, an application may not enforce that a user is logged in as administrator to change the database settings (authentication bypass), or it may not check that all the steps in the checkout process of a shopping cart are executed in the right order. Logic errors involving the data flow of the application are caused instead by failing to enforce that the user cannot tamper with certain values that propagate between different HTTP requests. As a result, an attacker can try to reuse expired authentication tokens (replay attack), or mix together the values obtained by running several parallel sessions of the same web application.

During my postdoc at the UCSB, I worked on a novel approach to the anomaly-based detection of attacks against web applications [1]. The tool we developed was designed to analyze the internal state of a web application and learn the rela-

[1] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, Giovanni Vigna "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications" – *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*

tionships between the application's critical execution points and the application's internal state. By doing this, our solution was able to identify attacks that attempt to bring an application in an inconsistent, anomalous state. This also includes violations of the intended workflow of a web application, in which an attacker can bypass authorization mechanisms (e.g., by gaining access to restricted portions of a web application) or subvert the correct business logic of the application (e.g., by skipping a required step in the checkout sequence of operations on an e-commerce web site). Even if we did not thing about that in these terms, those represented the first examples of logic vulnerabilities and the paper also proposed an anomaly-based protection against this class of attacks.

Summary

The second part of this dissertation includes two papers that focus on logic vulnerabilities. In the first, published in 2007 at the ACM CCS conference, I studied web vulnerabilities that involve the interaction between different server-side modules. In particular, together with my co-authors, we developed a novel vulnerability analysis approach that characterizes both the *extended state* and the *intended workflow* of a web application. Using this model, we proposed an analysis technique that was able to take into account, for the first time, inter-module relationships as well as the interaction of an application's modules with back-end databases. As a result, our solution was able to identify sophisticated multi-step attacks against the application's workflow that were not addressed by previous approaches.

The second paper presented in this part was published at the NDSS conference in 2014. In this study, which I co-authored with the first Ph.D. student I supervised, I came back to the problem of logic vulnerabilities and I discussed the first automated black-box technique to detect this type of flaws. In particular, in this paper we proposed a technique that analyzes network traces in which users interact with a certain application's functionality (e.g., a shopping cart). We then applied a set of heuristics to identify behavioral patterns that are likely related to the underlying application logic. For example, sequences of operations always performed in the same order, values that are generated by the server and then re-used in the following user requests, or actions that are never performed more than once in the same session. These candidate behaviors are then verified by executing very specific test cases generated according to a number of attack patterns. The advantage of this approach is that the test case generation steps are performed offline. In other words, they do not require to probe the application or generate any additional interaction and network traffic. In our experiments we applied our prototype to seven real world E-commerce web applications, discovering ten very severe and previously unknown logic vulnerabilities.

Multi-Module Vulnerability Analysis of Web-based Applications

Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna
Computer Security Group
University of California, Santa Barbara
Santa Barbara, CA, USA
{balzarot, marco, rusvika, vigna}@cs.ucsb.edu

ABSTRACT

In recent years, web applications have become tremendously popular, and nowadays they are routinely used in security-critical environments, such as medical, financial, and military systems. As the use of web applications for critical services has increased, the number and sophistication of attacks against these applications have grown as well. Current approaches to securing web applications focus either on detecting and blocking web-based attacks using application-level firewalls, or on using vulnerability analysis techniques to identify security problems before deployment.

The vulnerability analysis of web applications is made difficult by a number of factors, such as the use of scripting languages, the structuring of the application logic into separate pages and code modules, and the interaction with back-end databases. So far, approaches to web application vulnerability analysis have focused on single application modules to identify insecure uses of information provided as input to the application. Unfortunately, these approaches are limited in scope, and, therefore, they cannot detect multi-step attacks that exploit the interaction among multiple modules of an application.

We have developed a novel vulnerability analysis approach that characterizes both the *extended state* and the *intended workflow* of a web application. By doing this, our analysis approach is able to take into account inter-module relationships as well as the interaction of an application's modules with back-end databases. As a result, our vulnerability analysis technique is able to identify sophisticated multi-step attacks against the application's workflow that were not addressed by previous approaches. We implemented our technique in a prototype tool, called MiMoSA, and tested it on several applications, identifying both known and new vulnerabilities.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Security

Keywords: Web Applications, Multi-step Attacks, Vulnerability Analysis, Static Analysis, Dynamic Analysis

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

Web applications are growing in popularity. The introduction of sophisticated mechanisms for the handling of asynchronous events in web browsers and the availability of a number of frameworks for the rapid prototyping of server-side components have fostered the development of new applications and the transition of "traditional" applications (e.g., mail readers) to web-based platforms.

While new technologies have brought in significant advantages in terms of support to the development process, improved performance, and increased interoperability, little has been done to tackle security issues. Therefore, as the complexity of web applications increases, the possibility for abuse increases as well. For example, a simple analysis of the CVE vulnerability database [4] shows that the percentage of web-based attacks rose from 25% of the total number of entries in 2000 to 61% in 2006.

This situation is made worse by the fact that web applications are usually reachable through firewalls by design, and, in addition, the server-side logic is often developed under time-to-market pressure by developers with insufficient security skills. As a result, vulnerable web applications are deployed and made available to the whole Internet, creating easily-exploitable entry points for the compromise of entire networks.

To address the security problems associated with web applications, the research community has proposed a number of solutions. A first class of solutions focuses on detecting (and possibly blocking) web-based attacks. This can be done by analyzing the requests sent to web applications [13, 2, 21, 17, 18] or, in some cases, by analyzing the data delivered by the applications to the clients [11, 8]. These solutions have the advantage that they do not require any modification to the application being protected. However, they have a significant impact on the system's performance, and, in case of false positives (i.e., wrong detections), they may block legitimate traffic.

A second class of solutions focuses on identifying flaws in the implementation of a web application *before* the application is deployed. These approaches utilize static and dynamic analysis techniques to identify vulnerabilities in web applications [7,9,14]. Most of these approaches are based on the assumption that vulnerabilities in web applications are the result of insecure data flow. Therefore, these techniques attempt to identify when data originating from outside the application (e.g., from user input) is used in security-critical operations without being first checked and sanitized.

Even though these approaches are effective at detecting suspicious uses of unsanitized data, they suffer from three main limitations. First, their scope is limited to a single web application module, such as a single PHP file or a single ASP component. Therefore, these techniques are not able to identify vulnerabilities that are caused by the interaction of multiple modules. Second, these

approaches are not able to correctly model the interactions among multiple technologies, such as the use of multiple languages in the same application, or the use of back-end databases to store persistent data. Third, and most important, these techniques do not take into account either the *intended workflow* of a web application or its *extended state*.

The intended workflow of a web application represents a model of the assumptions that the developer has made about how a user should navigate through the application. Web applications are often designed to guide the user through a specific sequence of steps. For example, an e-commerce site could be structured so that the user first logs in, then browses a catalog and chooses some goods, and eventually checks out and purchases the items. The constraints among operations (e.g., one has to select some goods before purchasing them) define the application's intended workflow.

A number of mechanisms have been devised to track the progress of a user through the intended workflow of a web application. These mechanisms provide ways to store information that survives a single client-server interaction and define the extended state of the application. For example, in a LAMP application[1] the extended state could include the request variables used in each module and, in addition, the PHP session data and the database tables, which are shared between modules. The extended state can also include information that is sent back and forth between the client and the server to keep track of a user session, such as hidden form fields and application-specific cookies. Therefore, the extended state of an application is a distributed collection of session-related information, which is accessed and modified by the modules of a web application at different times during a user session.

Unfortunately, it is possible that different modules of an application have different assumptions on how the extended state is stored and handled, leading to vulnerabilities in the application. We call these vulnerabilities *multi-module vulnerabilities* to emphasize the fact that they originate from the interaction of multiple application modules, which communicate by reading and modifying the application's extended state.

In this paper, we present a novel vulnerability analysis approach that combines several analysis techniques to identify sophisticated multi-module vulnerabilities in web applications. In our approach, we first leverage dynamic approaches to analyze block-level properties in the code of web application modules. We then use static analysis to extract properties at the module level. Finally, we use model checking techniques to identify possible paths in a web application's workflow that could lead to an insecure state.

The contributions of our approach are the following:

- We introduce a novel model of web application extended state that characterizes permanent storage and is not limited to the variables and data structures defined in a single procedure or code module.
- We present a novel approach to analyze the interaction between the application's code and back-end databases, which allows for the identification of sophisticated data-driven attacks.
- We introduce an approach to derive the intended workflow of a web application and an analysis technique to identify multi-step attacks that violate the expected inter-module workflow of a web application.

We implemented our approach in a prototype analysis tool, called MiMoSA[2], for PHP-based web applications, and we evaluated it on a number of real-world applications, finding both known and new vulnerabilities. The results show that our approach is able to identify complex vulnerabilities that state-of-the-art techniques are not able to identify.

The rest of the paper is structured as follows. In Section 2, we present some examples of the vulnerabilities that are the focus of our approach. In Section 3, we introduce the web application model that is at the basis of our analysis. Section 4 and 5 describe our approach to the identification of multi-module vulnerabilities in web applications. Then, Section 6 presents the results of applying our analysis to real-world applications. Finally, Section 7 presents related work, and Section 8 briefly concludes.

2. MULTI-MODULE ATTACKS

Multi-module attacks can be categorized into two classes: data-flow attacks and workflow attacks. Data-flow attacks exploit the insecure handling of user-provided information that is stored in the web application's state and passed from one module to another. In workflow attacks, an attacker leverages errors in how the state is handled by the application's modules in order to use the application in ways that violate its intended workflow.

Data-flow Attacks.

In multi-module data-flow attacks, the attacker uses a first module to inject some data into the web application's extended state. Then, a second module uses the attacker-provided data in an insecure way[3]. Examples of multi-module data-flow attacks include SQL injection [3] and persistent (or stored) Cross-Site Scripting attacks (XSS) [12].

A web application is vulnerable to a SQL injection attack when it uses unsanitized user data to compose queries that are later passed to a database for evaluation. The exploitation of a SQL injection vulnerability can lead to the execution of arbitrary queries with the privileges of the vulnerable application and, consequently, to the leakage of sensitive information and/or unauthorized modification of data. In a typical multi-module SQL injection scenario, the attacker uses a first module to store an attack string containing malicious SQL directives in a location that is part of the application's extended state (e.g., a session variable). Then, a second module reads the value of the same location from the extended state and uses it to build a query to the database. As a result, the malicious SQL directives are "injected" into the query.

In cross-site scripting attacks, an attacker forces a web browser to evaluate attacker-supplied code (typically JavaScript) in the context of a trusted web site. The goal of these attacks is to circumvent the *same-origin policy*, which prevents scripts or documents loaded from one site from getting or setting the properties of documents originating from different sites. In a multi-module XSS attack, a first module is leveraged to store the malicious code in a location that is part of the extended state of the application, e.g., in a field of a table in the back-end database. Then, at a later time, the malicious code is presented to a user by a different module. The user browser executes the code under the assumption that it originates from the vulnerable application rather than from the attacker, effectively circumventing the same-origin policy.

Workflow Attacks.

Most web applications have policies that restrict how they can

[1] A LAMP application is a web application based on the composition of Linux, Apache, MySQL, and PHP.

[2] MiMoSA stands for Multi-Module State Analyzer.

[3] As it will be clear later, this second module can be a second invocation of the module that performed the first step of the attack.

be navigated to ensure that their functionality and data is accessed in a well-defined and controlled way. Usually, to implement these restrictions a module stores in the web application's extended state the current navigation state, e.g., whether or not the current user has logged in or has already visited a certain page. Other modules, then, use this portion of the state information to deny or authorize access to other parts of the application.

Workflow attacks attempt to circumvent these navigation restrictions. For example, a workflow attack could try to directly access a page that is not reachable through normal navigation mechanisms, such as hyper-textual links[4]. These attacks may allow one to bypass authorization mechanisms (e.g., gaining access to restricted portions of a web application) or to subvert the correct business logic of the application (e.g., skipping a required step in the checkout sequence of operations on an e-commerce web site).

3. A FORMAL CHARACTERIZATION OF MULTI-MODULE VULNERABILITIES

In the previous sections, we described how the state of a web application can be maintained in a number of different ways. In order to abstract away from the various language- or technology-specific mechanisms, we introduce the concept of *state entity*. A state entity E is similar to a variable in a traditional programming language, in that it can be used to store parts of the application's state. Different modules can share information by accessing the same state entities. The set of all the state entities corresponds to what we defined in the introduction as the application's *extended state*.

We classify the state entities into two classes: server-side and client-side. Server-side entities model the part of the extended state that is maintained on the server. For example, a server-side entity can represent a field in a database or a PHP session variable. Client-side entities are instead used to model the part of the extended state stored in and/or generated by the user's browser. Cookies, GET and POST parameters are examples of this type of entities.

3.1 Module Views

To summarize the operations that each module performs on the application's extended state, we introduce the concept of *Module View* (or simply *view* hereinafter). Each view represents all the state-equivalent execution paths in a single module, i.e., all the paths in the control-flow graph (CFG) of a module that perform the same operations on the state entities. When an application module is executed, e.g., as a consequence of a user request, the path followed by the execution in that module is completely included in one and only one of its views. In this case, we say that the view that contains the executed path is "entered" by the user. We describe the algorithm used to summarize a module into its views in Section 4.3.

Consider, for example, the login module of an application. When a user provides correct credentials, the module may define a set of new session variables (e.g., to track that the user is authenticated and to load her preferences). On the contrary, the module may redirect unauthorized users to an error page without changing the extended state. These two different behaviors depend on the current extended state of the application, namely on the values of the request parameters and the content of the database that stores the information about the users. The view abstraction allows us to associate with each behavior a compact representation that summarizes its effect on the extended state of the application.

Formally, a view V is represented as a triple (Φ, Π, Σ) where:

- Φ is the view's pre-condition, which consists of a predicate on the values of the state entities. The program paths modeled by the view can be executed only when the view pre-condition is true (evaluated in the context of the current extended state).
- Π is the set of post-conditions of the view. These conditions model, as a sequence of write operations on state entities, the way in which the extended state is modified by the execution of the program paths represented by the view. Each write operation has the following form:

  $$write(E_L, E_R, \Psi).$$

  This operation copies the content of the left entity E_L (which can also be a constant value) to the right entity E_R. The set Ψ contains the sanitization operations applied to the left entity before its value is transferred to the right entity. If the sanitization set is empty, no sanitization is applied.
- Σ is the set of *sinks* contained inside the view. Each sink is a pair (E, Op) where E is a state entity and Op is a potentially dangerous operation (such as a SQL query or an `eval` statement) that uses the entity unsanitized. Note that the unsanitized use of an entity is not necessarily a vulnerability, since the sanitization process may take place inside one of the other views (belonging to the same module or to another module).

The extended state of an application may change as the user moves from one web page to another, clicking on links, submitting forms, following redirects, or just jumping to a new URL. In fact, when a view is entered, the extended state S is updated by applying the view's post-conditions to the extended state in which the application was before entering the view. Let $V_i = (\Phi_i, \Pi_i, \Sigma_i)$ be the view entered at step i of the user's navigation process, then:

$$S_{init} = \emptyset \qquad S_i = apply(\Pi_i, S_{i-1}).$$

In addition to the set of the entity values, the extended state also keeps track of the current sanitization state of each entity. An entity E is sanitized in the application state S_i (represented by the predicate $san(E, S_i)$) if its value is set by sanitizing write operations. In this work, we take the standard approach of assuming that sanitization operations are always effective in removing malicious content from user-provided data.

3.2 Application Paths

The presence of the pre-condition predicate in each view limits the possible paths that a user may follow inside the web application. We say that a path $P = \langle V_0, V_1, \ldots, V_n \rangle$, where V_i is a view, belongs to the set of *Navigation Paths* $\mathcal{N}$ if and only if:

$$\forall i < n, \;\; S_i \models \Phi_{i+1},$$

that is, if and only if the state at each intermediate step satisfies the pre-condition of the following step.

Since at the beginning of the execution the application state is empty, it must be $\emptyset \models \Phi_0$. In order for this to happen, the pre-condition Φ_0 must be empty or it must contain only predicates on client-side entities. This is justified by the fact that pre-conditions containing only client-side entities (for example, those requesting a particular value for a certain GET parameter) can always be satisfied if the user provides the right value. We define the set of

[4]This attack is sometimes referred to as "forceful browsing."

Application Entry Points η as the subset of views that can be used as starting points in a navigation path:

$$V_i \in \eta \qquad \text{iff} \qquad \emptyset \models \Phi_i.$$

The subset of navigation paths allowed by the application design is called the *Intended Path* set, $\mathcal{I} \subseteq \mathcal{N}$. These paths represent the workflow of the web application, expressed either through the use of explicit links provided by the application or through other common user navigation behaviors. We say that a navigation path $\langle V_0, \ldots, V_n \rangle$ belongs to the intended path set of the application if and only if:

$$\forall i < n \left(V_{i+1} \in \eta \vee \exists Link(V_i, V_{i+1}) \vee V_{i-1} = V_{i+1} \vee V_i = V_{i+1} \right).$$

In other words, at each step of the path the next view satisfies one of the following: it is an application entry point, is reachable through a link, is the same as the previous view (which corresponds to the user pressing the *back* button in her browser), or is the same as the current view (which corresponds to the use of the *refresh* button).

Given the previous definition, we can now provide a formal characterization of the two classes of vulnerabilities we introduce in this paper. A violation of the intended workflow of the application occurs when:

$$\exists p \in \mathcal{N} \mid p \notin \mathcal{I},$$

that is, when there exists a valid navigation path that is not an intended path.

A multi-module data-flow vulnerability is defined as:

$$\exists p = \langle V_0, \ldots, V_n \rangle \in \mathcal{N} \ , \ \exists E_x \in \Sigma_n \mid \neg \ san(E_x, S_{n-1}),$$

that is, there is a path in the application such that some portion of the application's extended state is used in a security-critical operation without being properly sanitized.

4. INTRA-MODULE ANALYSIS

The analysis performed by MiMoSA consists of two phases: an intra-module phase, which examines each module of the application in isolation, followed by an inter-module phase, where the application is considered as a whole.

The goal of the intra-module analysis is to summarize each application module into a set of views, by determining its pre-condition, post-conditions, and sinks. From each module, we also extract the list of all outgoing links and we associate them with the views they belong to. This information is then used by the inter-module analysis to reconstruct the intended workflow of the application.

The main steps of the intra-module phase are shown in Figure 1. Note that these steps are obviously language-dependent. Even though in this paper we focus on applications written in the PHP language, our approach can be easily extended to extract views from modules written in other programming languages.

To better illustrate our technique, we will refer to a simple web application whose code is presented in Figure 2. The application is written in PHP and consists of three modules: `index.php`, which is the application entry point, `create.php`, which allows new users to create an account, and `answer.php`, which provides some information that should be accessible only to registered users. The application state is maintained using both a relational database, which contains the users' accounts, and a PHP session variable, i.e., `_SESSION["loggedin"]`.

Even though the application is very simple, it contains representative examples of the security problems that our approach is able to identify. In particular, the application contains two vulnerabilities. The first vulnerability is caused by the fact that the `index.php` module uses usernames retrieved from the database as part of its output page. Usernames are strings arbitrarily chosen by users during the registration process implemented by the `create.php` module. Since these strings are never sanitized in any module, the application is vulnerable to XSS attacks. The second vulnerability is contained in the `answer.php` module. The module incorrectly checks the value of the `loggedin` variable instead of `_SESSION["loggedin"]` in order to verify the user status. However, if the PHP `register_globals` option is activated and the `_SESSION["loggedin"]` variable has not been defined (i.e., the user is not logged in), an attacker can include a `loggedin` parameter in her GET or POST request, effectively shadowing the session variable with a value of her choosing. This could be leveraged to bypass the registration mechanism and access the restricted `answer.php` module without being previously authenticated, thus violating the intended workflow of the application.

As it is clear from the examples above, these vulnerabilities are carried out in multiple steps and involve multiple modules. The ultimate goal of our analysis is to detect these multi-module vulnerabilities. However, in order to analyze the interactions between modules, it is first necessary to analyze the properties of each module. This analysis is the focus of the rest of this section.

4.1 Control-Flow and Data-Flow Graphs Extraction

The first step of the intra-module analysis is the extraction of the control-flow and data-flow graphs from each module of the application. Our implementation leverages Pixy [9], a static analysis tool for detecting intra-module vulnerabilities in PHP applications. We adopted Pixy's PHP parser, control-flow graph derivation component, and alias analysis component. In addition, we extended Pixy with a data-flow component that computes the def-use chains for a module using a standard algorithm [1]. The resulting tool provides all the information needed for the following steps of the analysis. The main limitation of Pixy, besides being limited to intra-module analysis only, is the lack of support for object-oriented code. Where needed, we manually pre-processed input modules to work around this problem.

4.2 Database Analysis

Databases are often used by web applications to store data permanently. This data is usually accessible by every module of the application. Therefore, it is important to characterize module-database interactions as they could be leveraged to perform a multi-module attack.

The goal of the database analysis is to translate the interaction between an application module and the back-end database into a set of variable assignments. By doing this, the following steps of the analysis (e.g., the view extraction process) can handle database operations and assignments to variables in a uniform way.

For example, consider the following SQL query that writes the content of the variable `uname` to the column `username` in the database table `users`:

```
UPDATE users SET username=$uname WHERE...
```

As a result of the database analysis, a new assignment is added after the call to the function that executes the query. In our example, MiMoSA generates the following assignment node:

```
$DB_dbname_users_username = $uname;
```

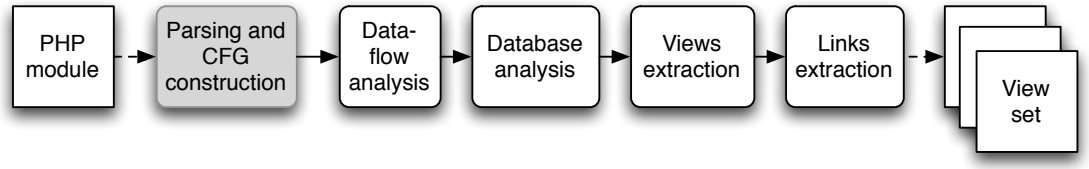


Figure 1: The main steps of the intra-module analysis. The parts in gray are implemented by Pixy.

Note that `DB_dbname_users_username` is a new variable created by our analysis to model the part of the database modified by the `UPDATE` operation.

The PHP language provides a number of internal functions to connect to different types of relational databases. In our prototype implementation, we focused on the MySQL library because of its popularity. However, if the target application uses a different database, our technique can be easily adapted to address a different set of primitives. In PHP, access to the MySQL database is usually performed by first calling the `mysql_query` function to execute a query, and then by using one of the `mysql_fetch` functions to access the results of the query in an iterative fashion.

The main challenge in the database analysis is to properly reconstruct the values that a query can assume at runtime, so that we can determine the tables and columns that are modified by the operation. To achieve this, we traverse the control-flow graph of the module, looking for calls to the `mysql_query` function. Since, in general, static analysis cannot provide the value that the query will assume at runtime, we apply a dynamic analysis technique to the block of PHP code that precedes the function call to derive the names and fields of the tables involved in the query. The analysis extracts the largest deterministic path $\widetilde{P}$ that precedes the `mysql_query` call. A deterministic path is a sequence of nodes in the control-flow graph that only contains branch instructions whose conditional expressions can be statically determined. We then remove from $\widetilde{P}$ any input/output related operation, and we replace any undefined variable in $\widetilde{P}$ with a placeholder.

The resulting code is passed to the PHP interpreter in order to dynamically determine the value that the query string can assume along the path $\widetilde{P}$. If the resulting query performs an *UPDATE* or an *INSERT* operation, it is immediately parsed to extract the assignment nodes as shown before. Queries that contain a *SELECT* statement are instead analyzed only when the analysis finds that the corresponding `mysql_fetch` function is used to assign the result values to one or more PHP variables.

Consider for instance the `mysql_fetch_assoc` call at line 16 of `index.php` of our sample application. Following the data-flow edges we reach the corresponding query string at line 12. The dynamic analysis along the deterministic path reconstructs the query `"SELECT * FROM users"`. The database analyzer then checks the database schema to resolve the `"*"` symbol to the corresponding list of column names and it finally generates the resulting assignments nodes:

```
$row["username"] = $DB_dbname_users_username;
$row["password"] = $DB_dbname_users_password;
```

Once these assignments are introduced to the module, the following analysis steps are able to treat the application state stored in a back-end database and the state stored in program variables in a uniform way.

4.3 Views Extraction

The goal of this step is to summarize a module into a set of views. This is a key step in our intra-module analysis, because it produces the module meta-information necessary to perform the inter-module vulnerability analysis.

To extract a module's views, we first perform *state analysis* to determine all statements in the control-flow graph that are state-related, i.e., that either contain state entities or are control- or data-dependent on state-related statements. We consider state entities of a PHP application the variables used to refer to request parameters (`_GET, _POST, _REQUEST`), cookies (`_COOKIE`), session variables (`_SESSION`), and the database variables generated by the database analysis step. This allows us to exclude from further analysis statements that do not depend on or modify the application state. Therefore, in the rest of the analysis we consider only the subgraph of the CFG that contains state-related nodes. The algorithm we use in this step is based on the functional data-flow analysis framework of [19], as implemented in Pixy.

4.3.1 Identifying Sinks and State Entities

To identify sinks, we determine all nodes in the CFG that contain an operation relevant to our analysis. In particular, we look for two types of operations: state-related operations and sink-related operations. State-related operations are those statements that modify the server-side state. For example, we identify uses of the session mechanism, that is, assignments to the `_SESSION` array or calls to the `session_register()` function. Sink-related operations are statements where state entities are used in sensitive sinks. Our technique focuses on identifying inter-module XSS and SQL injection attacks, and, therefore, we keep track of state entities displayed to the user or used in a database query. Consider, for example, the `create.php` module in our example. The analysis identifies two relevant operations: at line 19, a database query is executed, and, at line 21, the variable `_SESSION["loggedin"]` is modified.

After the relevant operations have been identified, we derive their conditional *guards*, i.e., the conditions associated with the branches in the CFG that must be taken in order to reach the statement associated with the operation. Note that we only keep track of state-dependent conditions, as identified by the state analysis. In our example, the two operations that we identified in `create.php` are guarded by the conditional statement at line 9. The analysis also recognizes that the true branch of the conditional must be taken to trigger the operations.

Then, for each variable that occurs in a conditional guard or in a state- or sink-related statement, we reconstruct its dependency with respect to state entities. We currently model several types of dependencies. In particular, *propagation dependencies* model the assignment of one variable to another; *call dependencies* denote the fact that a variable takes its value from the result of a function call (in particular, we currently model sanitization functions); *binary dependencies* model the composition of two variables, for

```php
<html>
<head>
   <title>The answer to Life, the
   Universe, and Everything</title>
</head>

<body>

<?php
   echo "People that know the answer:";

   $sql = "SELECT * FROM users ";
   mysql_select_db("dbname");
   $res = mysql_query($sql);

   while($row =  mysql_fetch_assoc($res))
      echo $row["username"];
?>

<a href="create.php">Create User</a>

</body>
</html>
```

index.php

```php
<?php
   session_start();

   if ($loggedin != "ok") {
      header("Location: index.php");
      exit;
   }

   echo "42";
?>

<html>
<head>
   <title>The final answer is:</title>
</head>

<body>
 <a href="index.php">Homepage</a>
</body>
</html>
```

answer.php

```php
<html>
<head>
   <title>Create a new user</title>
</head>

<body>

<?php
   if (isset($_POST["user"])) {

      $user = addslashes($_POST["user"]);
      $pass = addslashes($_POST["pass"]);

      session_start();

      $sql = 'INSERT INTO users ' .
         'VALUES (\'' . $user .
         '\', \'' . $pass . '\' )';
      mysql_query($sql);

      $_SESSION["loggedin"] = "ok";

      header("Location: answer.php");
      exit;
   }
?>

<form action="create.php"
      method="POST">

   UserName:
   <input name="user" type="text"><br>
   Password:
   <input name="pass" type="password"><br>
   <input name="create" type="submit">

</form>

</body>
</html>
```

create.php

```
Table: users
+----------+-------------+
| Field    | Type        |
+----------+-------------+
| username | varchar(32) |
| password | varchar(32) |
+----------+-------------+
```

Database schema

Figure 2: Example application.

example through mathematical or string operators; *constant dependencies* denote that a variable takes a constant value; *superglobal dependencies* indicate that a variable takes a value from one of the superglobal objects in PHP, e.g., from a request or session variable. Multiple dependencies are composed together until each variable is reduced to either a constant or a state entity.

Note that an additional set of conditional guards can be discovered during the dependency reconstruction analysis: for example, a variable used in an operation might assume different values depending on some conditions. Such conditions are added to the set of conditional guards for the operation.

In our example, the variable `_SESSION["loggedin"]`, used in the state-related statement at line 21 in `create.php`, is associated with a constant dependency that models the fact that it was assigned the constant value `ok`. The conditional guard at line 9 is reduced to the composition of a call dependency (to the `isset()` function) and a superglobal dependency (to the `_POST["user"]` variable).

4.3.2 Creating the View

After all sensitive operations and their complete set of conditional guards have been identified, we translate them into pre-conditions, post-conditions, and sinks. Currently the following predicates are used in pre-conditions: *Exist(v)* is true if and only if the entity *v* is defined in the current application state. *Compare(v, u, op)*, where *v* and *u* are state entities and *op* is an operator, is true if and only if the expression *v op u* is true. MiMoSA currently supports the operators $<$, $>$, $=$, and their combinations. The *Propagate* predicate is used in post-conditions: *Propagate(v, u, San)* denotes that the value of the entity *v* is propagated to *u* applying the sanitization operations specified by the set *San*. For sinks, the following predicates are used: *InSql(v)* denotes that the state entity *v* is used in a SQL query; *Displayed(v)* indicates that *v* is displayed to the user. Conditions can be combined with the use of *and*, *or*, and *not* operators.

In addition, we introduce the special *Unknown* predicate, which is assumed to be always satisfiable, to model the cases where we cannot resolve the dependency of a program variable to a state entity. This happens, for example, when a variable takes its value from a complex series of calls to functions that we do not model.

As an example of the view creation process, consider the module `create.php` of our sample application. MiMoSA summarizes it into two views, corresponding to the two branches of the conditional statement at line 9. One view (corresponding to the false branch) has pre-condition *not Exist($_POST["user"])* and empty post-conditions and sinks. The other view (corresponding to the true branch) has pre-condition *Exist($_POST["user"])*. The assignments introduced by the database analysis step to model the SQL query at line 19 are modeled with the post-conditions *Propagate($_POST["user"], DB_dbname.users.username, {addslashes})*

and *Propagate($_POST["pass"], DB_dbname.users.password, {addslashes})*. In both cases, the analysis keeps track of the sanitization operated by the `addslashes()` function. Finally, the assignment to the session variable `_SESSION["loggedin"]` is modeled with the post-conditions *Exist($_SESSION["loggedin"])* and *Propagate("ok", $_SESSION["loggedin"], ∅)*. The complete set of views for our example application is shown in Table 1.

In a module, the number of extracted views is exponential in the number of state-related conditional statements. As a consequence, the view extraction process is slow when dealing with very complex modules. Therefore, whenever the number of views is determined to be larger than a certain threshold, MiMoSA can be configured to switch to a simplified view construction approach. In this approach, instead of generating views for all the paths in the CFG of a module, we only generate the views corresponding to a number of paths sufficient to include all the state- and sink-related operations contained in the module. As a result, all the post-conditions and sinks of the module are extracted and will be analyzed during the detection phase. However, since not all their possible combinations are considered, the simplified approach might introduce inaccuracies.

4.4 Links Extraction

The last step before starting the inter-module vulnerability analysis is to extract the links contained in the module and associate them with the views they belong to.

We parse both PHP and HTML code looking for HTML hyperlinks, form actions and inputs, source attributes of frames, and calls to the PHP function `header()`[5]. We also have a limited support for link extraction from JavaScript code. If the URL of the link is dynamic, i.e., it is generated using a block of PHP code, the link extraction routine tries to determine its runtime value by applying a dynamic analysis technique similar to the one used in the database analysis phase.

Once all the links have been extracted, we identify the set of views to which each link belongs. In order to do this, we determine the conditional branches in the CFG that must be taken in order for a link to be shown to the user and we compare these branch expressions with the pre-conditions of the extracted views. Consider, for instance, the link to `answer.php` contained in the `create.php` module of our example application. Our analysis recognizes that it is displayed only if the execution follows the true branch of the conditional statement at line 9. `<create.php>.view_0` is the only view compatible with this execution and, therefore, it is identified as the source view of the link.

To correctly model the application workflow, in addition to having the names of the modules to which one can navigate from a given view, we also need to extract the set of inputs that are submitted along the link. In particular, we need to determine which GET and POST requests parameters are submitted if a user follows the link. For example, in our sample application, if a user submits the form at line 28 of the `create.php` module, the user-provided parameters `user` and `pass` are submitted as a part of the POST request to `create.php`.

5. INTER-MODULE ANALYSIS

In the second phase of our analysis, we connect the views extracted during the intra-module analysis into a single graph. This graph models the intended workflow of the entire web application. We then use a model checking technique to identify multi-module data-flow vulnerabilities and violations of the intended workflow.

[5]The `header()` function in PHP is commonly used to set the HTTP `Location` header to redirect users to a different page.

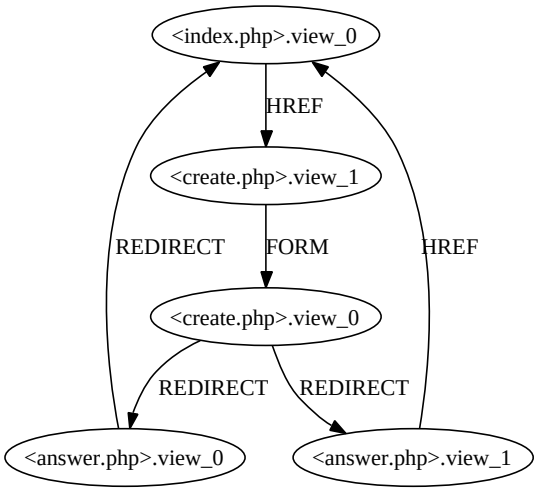


Figure 4: Intended workflow of our example application.

The main steps of the inter-module phase are shown in Figure 3. Note that since this phase is built on top of the view abstraction, it is completely independent of the programming languages in which the modules are developed.

5.1 Intended Workflow

In the first step of the inter-module phase, we use the link information extracted during the intra-module analysis to connect all the views of the application into a single graph.

We connect a source view V_i to a target view V_j if V_i contains a link `l` that references V_j's module and the parameters provided by `l` satisfy the pre-condition of V_j. In particular, we adopt the following two rules:

1. If V_j's pre-condition contains predicates over client-side state entities, we check that the extracted link satisfies these requirements. For example, if the pre-condition requires the presence of a particular GET parameter, we check that the link provides a parameter with the required name.

2. If V_j's pre-condition contains predicates over server-side state entities, we assume that these predicates are always satisfied. The rationale is that, in general, it is not possible to determine the extended state of the application considering the two views in isolation, because it depends on the path that the user has followed to reach V_i. Therefore, we conservatively assume that the state can satisfy V_j's pre-condition.

When both conditions are satisfied, we assume that there is an intended path between the two views and we connect them together. For example, the link in `<index.php>.view_0` (line 20) is connected to the view `<create.php>.view_1` but not to `<create.php>.view_0`. In fact, the pre-condition of `<create.php>.view_0` requires the existence of a POST parameter named `user` that is obviously not provided if the user clicks on the link in `index.php`. The intended workflow for our example application is given in Figure 4.

Finally, the analysis identifies the application's *entry points*. We exclude the modules that appear inside an `include` statement from this step of the analysis, because they are generally not intended to be directly accessed by the user. Of the remaining modules, we consider as entry point any view that has either an empty pre-condition or a pre-condition that contains only predicates over `GET` parameters (see Section 3).

Module	View ID	Pre-conditions	Post-conditions	Sinks
`index.php`	*view_0*	∅	∅	*Displayed(DB_dbname.-users.username)*
`create.php`	*view_0*	*Exist($_POST["user"])*	*Propagate($_POST["user"], DB_dbname.users.username, {addslashes})* *Propagate($_POST["pass"], DB_dbname.users.password, {addslashes})* *Exist($_SESSION ["loggedin"])* *Propagate("ok", $_SES-SION["loggedin"], ∅)*	∅
`create.php`	*view_1*	not *Exist($_POST["user"])*	∅	∅
`answer.php`	*view_0*	not (*Exist($loggedin)* and *Compare($loggedin, "ok", =)*)	∅	∅
`answer.php`	*view_1*	*Exist($loggedin)* and *Compare($loggedin, "ok", =)*	∅	∅

Table 1: Views generated for the example application of Figure 2.

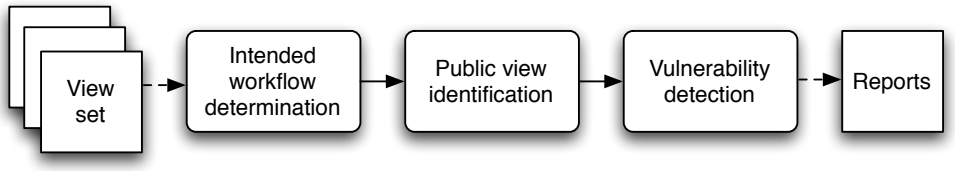


Figure 3: The main steps of the inter-module analysis.

Unfortunately, in some cases it is not possible to differentiate between an application's entry point and the developer's failure to put the necessary safety checks into a module. For example, in our experiments we tested a web application where in one of the administration pages the developer forgot to put a check to verify that the user was actually logged in as administrator. Our technique classified the views of this module as entry points since they did not have any pre-condition at all. Nevertheless, the user of our tool can easily detect these vulnerabilities by inspecting the automatically generated list of entry points.

5.2 Detecting Public Views

The *intended path* introduced in Section 3 did not model a very important concept of a web application: the existence of *publicly-accessible* pages. These pages (such as the FAQs pages) are very common in many web sites but they are rarely intended as entry points to the application. Therefore, we do not generate any security alert if it is possible to access these pages violating the intended workflow of the application.

For this reason, we adopted the following rules to detect and mark the publicly-accessible views:

- Starting from one of the application entry points, all the views that are reachable along some intended path traversing only views that have empty post-conditions are marked as *public*. This models the fact that if it is possible to reach a view through a path that does not change the extended state of the application, the access to the view is not supposed to be restricted.
- Any empty redirect view is *public*. An empty redirect view is a view that does not have any post-condition, any sink, and only contains a redirect link. This models all the views used to detect and redirect unauthenticated users that try to access a restricted page.

In the example, our algorithm marked `<create.php>.view_0`, `<create.php>.view_1`, and `<answer.php>.view_0` as public views. The first two because they are reachable without any change in the application state and the last one because it is an empty redirect.

5.3 Detection Algorithm

Our graph exploration mechanism simulates a user that moves from one view to another. At each step, we select a new view to add to the current path, we evaluate its pre-condition against the current state, and, if the pre-condition is satisfied, we update the state to reflect the effects of the view's post-conditions.

Each path is analyzed to check if it satisfies the definition we provided in Section 2 for multi-step data-flow vulnerabilities and workflow violations. In general, if the graph is correct, it is possible to find all the vulnerabilities simply by trying each possible *navigation path* in the application. Our solution is similar to a model checking approach, and, unfortunately, it suffers from the same path explosion problem. Therefore, we limit our analysis to paths that contain up to one loop and with a total length limited by a user-defined upper bound. In our experiments, in fact, we observed that most of the vulnerabilities can be exploited using a very limited number of steps (usually less than 5).

Our detection algorithm traverses the graph following the intended paths. At each step it checks if it is possible to jump to one of the views that should not be reachable from the current position. If it succeeds, it raises a workflow violation alert and it does not go any further along that path. This means that some vulnerabilities may not be discovered because they are hidden "behind" other vulnerabilities. In this case, the user should fix the discovered vulnerability and run the analysis again.

By applying MiMoSA to our sample application, we identify the two existing vulnerabilities. Figure 5 shows the reports produced by MiMoSA for the example application of Figure 2.

```
Workflow Violation:
Path:
  - index.php[view_0]
  - answer.php[view_1]
```

```
DISPLAY of unsanitized entity:
Entity: DB_dbname.users.username
Example of Exploitable Path:
   - create.php[view_0]
   - index.php[view_0]
```

Figure 5: Vulnerabilities detected in the sample application of Figure 2.

6. EVALUATION

To prove the effectiveness of our approach in detecting multi-module data-flow vulnerabilities and violations of the intended workflow of a web application, we ran our tool on five real-world web applications.

The selected applications satisfy three requirements: i) they are written in PHP and they contain multiple modules, ii) they use both session variables and database tables to maintain the application state, and iii) they do not contain object-oriented code. The list of chosen applications is shown in Table 2. The table also shows the list of known vulnerabilities for each application.

For each application we ran the intra-module analysis in order to extract the set of views corresponding to the application modules. We then ran the inter-module analysis to connect together the views and calculate the intended application workflow. Finally, we applied our detection algorithm to find anomalies in the possible navigation paths and to detect multi-module data-flow vulnerabilities.

The results of our tests are summarized in Table 3. For the intra-module phase, the table reports the number of views extracted and the time required by the analysis[6]. In the inter-module phase, we explored up to one hundred million paths, covering at least all the paths of length 3. The table reports the time required to generate the paths and the alert messages raised by our tool. The alerts are grouped according to the entities involved (for the data-flow vulnerabilities) and the modules (for the workflow violations). For both data-flow and workflow vulnerabilities, we report the number of violations detected by our tool, the number of false positives, and how many of the remaining violations correspond to exploitable vulnerabilities.

MiMoSA was able to find all the known vulnerabilities and to discover several new ones.

With regard to multi-module data-flow vulnerabilities, we had only one false positive. In fact, in the MyEasyMarket application, the PHP variable `REMOTE_ADDR` is saved in the database and later printed to the user. Even though the value of the variable is never sanitized, it is automatically set to the IP address of the client's machine by the PHP engine. Therefore, it only has a limited range of valid values (numbers and dots) that do not allow a user to mount an attack against the application.

MiMoSA also reported several violations of the intended workflow of the web applications. Even though in most of the cases they corresponded only to anomalous paths into the application (e.g., directly jumping from the login to the logout page), we were also able to confirm that some of the reported violations correspond to actual vulnerabilities that could be exploited to gain unauthorized access to a restricted page.

While the inter-module analysis is the more time consuming phase, the intra-module analysis is certainly the more fragile, since it is where the static analysis techniques that we use introduce most of the approximations. Any imperfection in this phase can result in an increasing number of both false positives and false negatives. For instance, during the construction of the intended paths, we observed that some of the views were isolated, with no connection to any other part of the application[7]. This was probably caused by an error in the view extraction, such as a missing link or a wrong pre-condition predicate.

To better test the accuracy of our intra-module analysis and evaluate its impact on the final results, we selected one of the applications in our test suite (i.e., SimpleCMS) and manually analyzed the output of each step of the view extraction phase. The results are shown in Table 4. MiMoSA achieves a high accuracy in the extraction of database operations, links, post-conditions, and sinks. Also the rate of unknown conditions, i.e., the pre-conditions that MiMoSA was not able to correctly reconstruct, is reasonable, considering that we are using a static analysis technique.

In this application, the number of generated views is, instead, considerably higher than the number of views actually present in the application code. This happens because of two main reasons. First, MiMoSA might generate views corresponding to paths that are infeasible in the program, such as the ones that traverse nodes with conflicting conditions. The presence of these views does not affect the final results since they are never entered during the detection phase. The second reason is that MiMoSA can generate duplicate views, i.e., views with different but equivalent pre-conditions. Even though this may lead to inaccuracy in the final results, in most of the cases its main effect is just to slow down the path generation phase.

7. RELATED WORK

In the introduction, we briefly mentioned some recent works in the areas of intrusion detection and application firewalls that focus on detecting and blocking web-based attacks. Since our work focuses on vulnerability analysis, and, consequently, deals with a different class of problems than the detection of attacks at runtime, we are not going to further review these works here.

There is a number of recent works in the area of vulnerability analysis of web-based applications. Most of these approaches are based on taint propagation analysis applied to application written in PHP [7, 9, 10, 22] or Java [6, 14].

The WebSSARI tool [7] is one of the first works that applies static taint propagation analysis to find security vulnerabilities in PHP. WebSSARI targets three specific types of vulnerabilities: cross-site scripting, SQL injection, and general script injection. The tool uses flow-sensitive, intra-procedural analysis based on a lattice model and typestate. When the tool determines that tainted data reaches sensitive functions, it automatically inserts *runtime guards*,

[6]All the experiments were executed on a Pentium 4 3.6GHz with 2G of RAM.

[7]These views were not taken into consideration by our path exploration algorithm since they could not provide any useful information to the user.

Application Name	PHP Files	Description	Known Vulnerabilities
Aphpkb 0.71	59	Knowledge-base management system	–
BloggIt 1.01	24	Blog engine	CVE-2006-7014
MyEasyMarket 4.1	23	On-line shop	–
Scarf 2006-09-20	18	Conference administration	CVE-2006-5909
SimpleCms	22	Content management system	BID 19386

Table 2: PHP applications used in our experiments. Vulnerabilities are referenced by their Common Vulnerabilities and Exposures ID (CVE) or their Bugtraq ID (BID).

Application	Intra-Module Analysis		Inter-Module Analysis				
	Views	Time	Time	DF Violations-(FP)	DF Vulnerabilities	WF Violations-(FP)	WF Vulnerabilities
Aphpkb	4680	31:24m	3:00h	0-(0)	0	17-(10)	-
BloggIt	339	2:12m	0:31h	14-(0)	14	3-(0)	-
MyEasyMarket	449	1:12:00h	6:36h	2-(1)	1	1-(0)	1[a]
Scarf	1721	7:30m	1:10h	3-(0)	3	3-(0)	1
SimpleCms	417	0:22m	2:50h	8-(0)	8	5-(0)	4

[a] Detected through inspection of the entry point list, as discussed in Section 5.1.

Table 3: Results of the experiments. DF: Data Flow, WF: Work Flow, FP: False Positives.

i.e., sanitization routines.

Xie and Aiken [22] use intra-block, intra-procedural, and inter-procedural taint propagation analysis to find SQL injection vulnerabilities in PHP code. This approach uses *symbolic execution* to model the effect of statements inside functions. These effects are summarized into the pre- and post-condition sets for each analyzed function. The function pre-conditions contain a derived set of memory locations that have to be sanitized before the function invocation, while the post-conditions contain the set of parameters and global variables that are sanitized inside the function. To model the effects of sanitization routines, the approach uses a programmer-provided set of possible sanitization functions, considers certain forms of casting as a sanitization process, and, in addition, keeps a database of sanitizing regular expressions, whose effects are specified by the programmer.

Pixy [9,10], which we have described in Section 4.1, specifically targets the identification of intra-module XSS vulnerabilities. This tool seems to be the most complete static PHP analyzer in terms of the PHP features modeled. To the best of our knowledge, it is the only publicly available tool for the analysis of PHP-based applications.

None of the described approaches performs inter-module analysis, that is, all the vulnerabilities identified by these approaches are local to a single application module. Unlike our approach, these techniques do not have any notion of the application's extended state, and, therefore, they are unable to capture the workflow vulnerabilities described in Section 2. By considering all inputs generated from outside of an application as being tainted, these approaches should be able to identify some types of multi-module data-flow vulnerabilities. However, because of the locality of the analysis, they are incapable of tracing the origins of multi-steps attacks, and, as a result, are subject to a much higher false positive rate.

There is also a number of works that apply dynamic analysis techniques to the analysis of web-based applications. For example, approaches that use dynamic taint propagation analysis, conceptually similar to Perl's taint mode but often with a more refined granularity, have been applied to other languages as well: Nguyen-Tuong et al. [15] propose modifications of the PHP interpreter to dynamically track tainted data in PHP programs, and Haldar et al. [5] apply a similar approach to the Java Virtual Machine.

Pietraszek and Vanden Berghe [16] present a unifying view of injection vulnerabilities and describe a general approach for detecting and preventing injection attacks. This approach is based on instrumenting the platform, such as the PHP interpreter, to track the flow of untrusted data inside the applications. A context-sensitive string evaluation is then performed at each sensitive sink to detect injection attacks.

All dynamic approaches described above either are able or, at least in theory, can be extended to detect multi-module data-flow attacks. The main difference with our approach is that we are able to detect such vulnerabilities statically, considering all the possible application's paths. Also, none of these approaches can detect workflow vulnerabilities because they do not model or take into account the application's intended workflow.

There are also several recent approaches that try to identify SQL injection attacks by building models of legitimate queries that can be performed by an application and comparing these models to the dynamically-generated queries. Whenever these queries structurally violate the static model, an attack is detected. For example, the AMNESIA tool [6] targets SQL injection attacks in Java-based applications. AMNESIA defines a SQL injection attack as the attack in which the logic or semantics of a legitimate SQL statement is changed due to malicious injection of new SQL keywords or operators. Thus, to detect such attacks, the semantics of dynamically-generated queries is checked against a derived model that represents the intended semantics of the query.

Su and Wassermann [20] propose another approach that uses the syntactic structure of the program-generated output to identify *injection attacks*, such as XSS, XPath injection, and shell injection attacks. The current implementation, called *SqlCheck* is designed to detect SQL injection attacks only. The approach works by tracking sub-strings from the user input through the program execution. The tracking is implemented by augmenting input strings with special characters, which mark the start and the end of each sub-string. Then, dynamically-generated queries are intercepted and checked

Views		Accuracy				Rate of
Extracted	Optimal	DB Operations	Links	Post-Conds	Sinks	Unknown Conditions
417	47	96%	78%	100%	100%	15%

Table 4: Accuracy of the view extraction step for SimpleCMS.

by a modified SQL parser. Using the meta-information provided by the sub-string markers, the parser is able to determine if the query's valid syntactic form is modified by the sub-string derived from user input, and, in that case, it blocks the query.

Both AMNESIA and SqlCheck can successfully detect SQL injection attacks at the time of injection; however, without a significant implementation effort, none of them can detect data-flow vulnerabilities such as persistent XSS attacks. Obviously, both approaches, as being based on the syntactic structure of legitimate output, are incapable of detecting workflow vulnerabilities/attacks.

8. CONCLUSIONS

As web applications that perform security-critical tasks become more sophisticated, there is an increasing need for techniques and tools that can address the novel security issues introduced by these applications. In particular, because of the heterogeneous nature of web applications, it is important to develop new techniques that are able to analyze the interaction among multiple application modules and different technologies.

In this paper, we presented a novel vulnerability analysis approach that takes into account the multi-module, multi-technology nature of complex web applications. Our technique is able to model both the *intended workflow* and the *extended state* of a web application in order to identify both workflow and data-flow attacks that involve multiple modules.

We developed a prototype tool, called MiMoSA, that implements our approach and we tested it on a number of real-world applications. The results show that by modeling explicitly the state and workflow of a web application, it is possible to identify complex vulnerabilities that existing state-of-the-art approaches are not able to identify.

Future work will focus on two main directions. First, we will include additional technologies so that we can cover a larger class of applications. Second, we plan to leverage the findings of the static analysis to automatically generate test drivers to reduce the number of the false positives.

Acknowledgments

This research was partially supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, and CCR-0716095.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] M. Almgren, H. Debar, and M. Dacier. A Lightweight Tool for Detecting Web Server Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 157–170, February 2000.

[3] C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd, 2002.

[4] Common Vulnerabilities and Exposures. `http://www.cve.mitre.org/`, 2006.

[5] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'05)*, pages 303–311, December 2005.

[6] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.

[8] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, pages 1–10, September 2006.

[9] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, May 2006.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 27–36, June 2006.

[11] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 330–337, April 2006.

[12] A. Klein. Cross Site Scripting Explained. Technical report, Sanctum Inc., 2002.

[13] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, October 2003.

[14] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium (USENIX'05)*, pages 271–286, August 2005.

[15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the International Information Security Conference (SEC'05)*, pages 372–382, May 2005.

[16] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 372–382, 2005.

[17] I. Ristic. ModSecurity. `http://www.modsecurity.org/`, November 2006.

[18] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *Proceedings of the International World Wide Web Conference (WWW'02)*, pages 396–407, May 2002.

[19] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In N. Jones and S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice Hall, 1981.

[20] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 372–382, January 2006.

[21] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, December 2003.

[22] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium (USENIX'06)*, pages 271–286, August 2006.

Toward Black-Box Detection of Logic Flaws in Web Applications

Giancarlo Pellegrino
EURECOM, France
SAP Product Security Research, France
giancarlo.pellegrino@eurecom.fr

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

***Abstract*—Web applications play a very important role in many critical areas, including online banking, health care, and personal communication. This, combined with the limited security training of many web developers, makes web applications one of the most common targets for attackers.**

In the past, researchers have proposed a large number of white- and black-box techniques to test web applications for the presence of several classes of vulnerabilities. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. Unfortunately, logic vulnerabilities specific to particular applications remain outside the scope of most of the existing tools and still need to be discovered by manual inspection.

In this paper we propose a novel black-box technique to detect logic vulnerabilities in web applications. Our approach is based on the automatic identification of a number of behavioral patterns starting from few network traces in which users interact with a certain application. Based on the extracted model, we then generate targeted test cases following a number of common attack scenarios.

We applied our prototype to seven real world E-commerce web applications, discovering ten very severe and previously-unknown logic vulnerabilities.

I. Introduction

Web applications play a very important role in many critical areas, and are currently trusted by billions of users to perform financial transactions, store personal information, and communicate with their friends. Unfortunately, this makes web applications one of the primary targets for attackers interested in a wide range of malicious activities.

To mitigate the existing threats, researchers have proposed a large number of techniques to automatically test web applications for the presence of several classes of vulnerabilities. Existing solutions span from black-box fuzzers and pentesting tools to static analysis systems that parse the source code of an application looking for well-defined vulnerability patterns. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. To date, more subtle vulnerabilities specific to the logic of a particular application are still discovered by manual inspection [33].

Logic vulnerabilities still lack a formal definition, but, in general, they are often the consequence of an insufficient validation of the business process of a web application. The resulting violations may involve both the control plane (i.e., the navigation between different pages) and the data plane (i.e., the data flow that links together parameters of different pages). In the first case, the root cause is the fact that the application fails to properly enforce the sequence of actions performed by the user. For example, an application may not require a user to log in as administrator to change the database settings (authentication bypass), or it may not check that all the steps in the checkout process of a shopping cart are executed in the right order. Logic errors involving the data flow of the application are caused instead by failing to enforce that the user cannot tamper with certain values that propagate between different HTTP requests. As a result, an attacker can try to replay expired authentication tokens, or mix together the values obtained by running several parallel sessions of the same web application.

Formal specifications describing the evolution of the internal state and of the expected user behavior are almost never available for web applications. This lack of documentation makes it very hard to find logic vulnerabilities. For example, while being able to add several times the same product to a shopping cart is a common feature, being able to add several times the same discount code is likely a logic vulnerability. A human can easily understand the difference between these two scenarios, but for an automated scanner without the proper application model it is very hard to tell the two behaviors apart.

Only recently the research community has started investigating automated approaches to detect logic vulnerabilities [9, 18, 21]. Unfortunately, the existing solutions have serious scalability problems that limit their applicability to small applications. Moreover, the source code of the application is often required in order to extract a proper model to guide the test case generation. As a result, to date the impact of available automated tools has been quite limited.

As an alternative approach, researchers have recently re-

NDSS '14, 23-26 February 2014, San Diego, CA, USA

http://dx.doi.org/doi-info-to-be-provided-later

sorted to manual analysis to expose several severe logic flaws in real world commercial applications [34, 35] resulting, for instance, in the ability to shop online for free. Following the step of these previous works, in this paper we show that it is possible to automatically infer an approximate model of a web application starting from a few network traces in which a user "stimulates" a certain functionality. Our goal is not to automatically reconstruct an accurate model of the application or of its protocol (several works already exist in this direction [14, 15]) but instead to empirically show that even a simple representation of the application logic is sufficient to perform automated reasoning and to generate test cases that are likely to expose the presence of logic vulnerabilities.

In this paper we propose a technique that analyzes network traces in which users interact with a certain application's functionality (e.g., a shopping cart). We then apply a set of heuristics to identify behavioral patterns that are likely related to the underlying application logic. For example, sequences of operations always performed in the same order, values that are generated by the server and then re-used in the following user requests, or actions that are never performed more than once in the same session. These candidate behaviors are then verified by executing very specific test cases generated according to a number of attack patterns. It is important to note that our approach is not a fuzzer, and both the trace analysis and the test case generation steps are performed offline. In other words, they do not require to probe the application or generate any additional interaction and network traffic.

While our approach is application-agnostic, the choice of the attack patterns reflects a particular class of logic flaws and application domain — and in our case were customized for E-commerce applications. In particular, we applied our prototype to seven large shopping cart applications adopted by millions of online stores. The prototype discovered ten previously-unknown logic flaws among which five of them allow an attacker to pay less or even shop for free.

In summary, this paper makes the following contributions:

1) We introduce a new black-box technique to test applications for logic vulnerabilities;
2) We present the implementation of a tool based on our technique and we show how the tool can be used to test several real web applications, even with a very limited knowledge and a small number of network traces;
3) We discover ten previously-unknown vulnerabilities in well-known and largely deployed web applications. Most of these vulnerabilities have a very high impact and would allow an attacker to buy online for free from hundreds of thousands of online stores.

Structure of the paper. Section II presents the black-box approach. Section III describes the experiments that we performed and Section IV shows the results. Section V discusses the limitations of our approach and Section VI presents related work on detecting logic vulnerabilities. Finally, Section VII concludes the paper.

II. Approach

The OWASP Testing Guide 3.0 [33] suggests a four-step approach to test for logic flaws in a black-box setting. First, the tester studies and understands the web application by playing with it and reading all the available documentation. Second, she prepares the information required to design the tests, including the *intended workflow* and the *data flow*. Then she proceeds with the design of the test cases, e.g., by reordering steps or skip important operations. Finally, she sets up the testing environment by creating test accounts, runs the tests, and verifies the results.

Our approach aims at automating the previous steps in a single black-box tool. First, starting from a list of network traces containing HTTP conversations, our system infers an application model and clusters resources related to the same workflow "step" (Section II-A). Second, our technique analyzes the model and extracts a set of *behavioral patterns* (Section II-B) modeling both the workflow and data flow of the application. Third, we apply a set of *attack patterns* to automatically generate test cases (Section II-C). Finally, we execute them against the web application (Section II-D), and we use an *oracle* to verify whether the logic of the application has been violated (Section II-E).

In the rest of the section we describe each phase in details using E-commerce web applications as a running example.

A. Model Inference

The technique we present is *passive* and *black-box*. We do not require any access to the application source code (both on the client- and server-side), and we do not actively crawl the application pages nor generate any traffic to probe its internal state. Instead, we take as input a list of HTTP conversations. These traces can be manually generated by the tester, or collected by logging real user activity.

For simplicity, we consider only traces that exercise a specific functionality of the web application. For example, if the web application is a shopping cart, we use traces in which users log in, add items into the cart, and check out to buy the products. Nothing prevents the tester from generating traces that also contain other functionalities, such as browsing the online catalog or posting product reviews. However, focusing only on one aspect of the business logic helps our system to find the relevant operations with a minimum number of input traces.

Web applications often involve multiple parties. For instance, E-commerce web applications typically involve the client, the store, and the payment service. However, the communication between them is normally channeled through the client and, therefore, we focus on this point to collect the traces. In addition, it is useful to collect data from different deployments of the same web application, to allow our inference method to identify parameter values hard-coded in a certain installation.

The first phase consists of building the model of the application, called *navigation graph*. This is done in two steps: resource abstraction, and resource clustering.

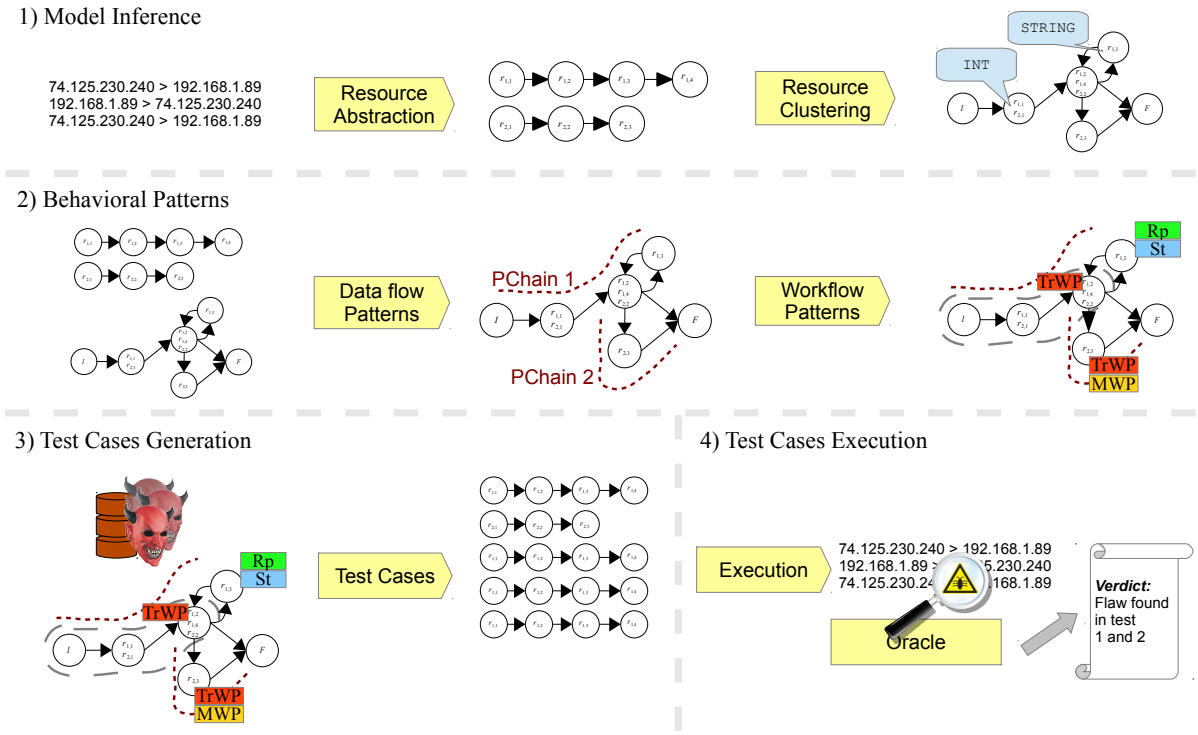


Fig. 1: Architecture of our approach.

1) *Resource Abstraction*

Input traces are sequences of pairs of HTTP requests and responses. The first step of the inference phase consists of creating a synthesis of the resources. Our approach currently supports JSON data objects [17] and HTML pages. However, it can be easily extended to other types such as SOAP messages [36].

We call *abstract HTML page* the collection of (i) its URL, (ii) the POST data, (iii) the anchors and forms contained in the HTML code and their DOM paths, (iv) the URL in the meta refresh tag, and, if any, (v) the HTTP redirection location header. We call *abstract JSON object* a collection of (i) its URL, (ii) the POST data, (iii) the pairs of value and path in the object, and (iv) the HTML links if any HTML code is contained. For example, Figure 2 shows the abstract resource of the following JSON object:

```
{'items': {
    'item1': ['price':19.9,'tax':1.6],
    'item2': [ ... ]}}
```

From each abstract resource we extract a set of elements corresponding to all possible parameters that appear in the URLs, in the POST data, and in all the links. Each element is characterized by a name, a value, a path, and an inferred syntactic type. Our approach supports the integer type, decimal type, URL type, email address type, word type (alphabetical strings e.g., "add", "remove", ...), string type, list type (comma-separated values), and *unknown* type (i.e., everything else). The type is associated to each element by inspecting the values of the element. Obvious priority rules are applied in case of ambiguity – e.g., id=20 can be both a number and a string, but being the first a subset of the second, it is considered to be a number.

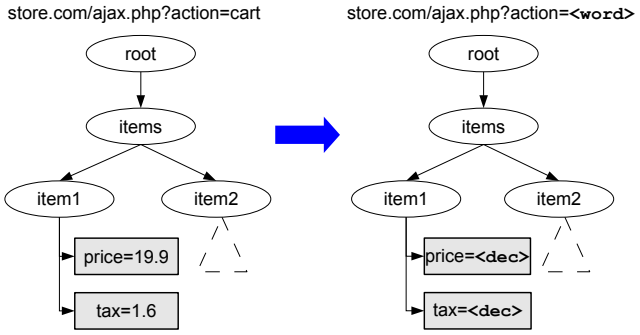


Fig. 2: Resource abstraction and syntactic type inference of a JSON data object

2) *Resource Clustering*

Modern web applications map application logic operations to different resources. For instance, the operation of displaying the shopping cart could involve an initial HTML page containing the skeleton of the web page and then use a number of asynchronous AJAX requests to populate the page with the list of items, tax, available vouchers, and so on. We cluster these resources in three phases. First, we relate asynchronous requests to the resource that originated them, i.e., synchronous resource. Then we group together resources considering both

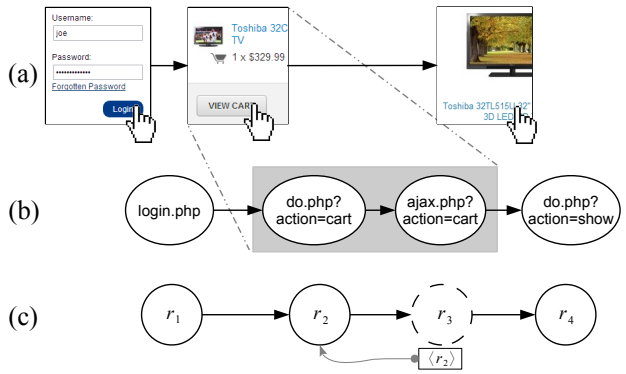


Fig. 3: (a) Application-level actions, (b) URLs requested, and (c) abstract resources with list of originators

similarity and the originators. Third, we split a cluster if a parameter of its resources encodes a command rather than carrying a value.

During the first phase, we pre-process input traces to identify AJAX requests. This can be done by checking the "X-Requested-With" HTTP request header [32] or by detecting JSON responses. After that, we associate each resource to its *originators*. Figure 3 provides an example of this first phase. In Figure 3.c we have the HTML page r_1 followed by the page r_2. Then r_2 requests r_3 by using AJAX that enriches r_2 with new HTML code, or new client-side scripts. The example then ends with r_4 that we assume to be caused by a link in r_2 or added by r_3. Figure 3.c also shows the list of originators of each resource. r_1, r_2, and r_4 have no originators, while r_3 was originated by r_2.

In the second phase, we cluster resources. In general, two resources are grouped in the same cluster if they have the same URL domain and path, the same GET/POST parameter names, and, if any, the same redirection URL. When comparing parameters we do not take into account their values, but only their syntactic types. For example, the following three URLs are equivalent:

```
store.com/do.php?action=add&id=3
store.com/do.php?action=add&id=7
store.com/do.php?action=show&id=3
```

We compare first synchronous resources as explained before, and then the asynchronous ones. Two asynchronous resources are in the same cluster if they have the same URL domain and path, GET/POST parameter names, redirection URL, and the same originators.

During the last phase, we identify the parameters that are encoding a command rather than transporting a value. For each parameter we take the pages that have the same value as that parameter. For example, the parameter `action` divides the gray cluster of Figure 4.a in two sub-groups, one for the `cart` value and one for the `show` value. We then compute the *page similarity* between pages in the same sub-group and between pages in different sub-groups. The comparison is done by looking at the DOM path of HTML forms, their action attribute (URL domain and parameter names), and the name of the nested input elements. The function is applied to sub-groups by calculating the percentage of pages that are similar.

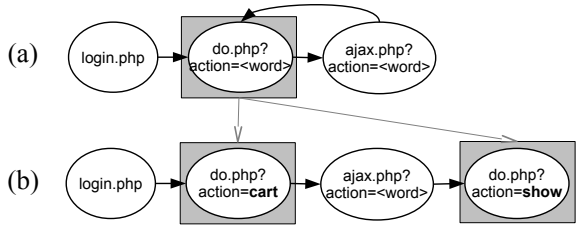


Fig. 4: (a) Clusters after comparing all the resources (b) Clusters after having identified parameters encoding a command

If the similarity inside the same sub-groups is high (more than 55%), and between different sub-groups is low (less than 45%), then we assume the parameter is used to specify an operation and we create a different node for each value. Otherwise we leave the cluster unmodified. The result of this phase is shown in Figure 4.b.

The navigation graph is a directed graph $G = (\mathcal{C} \cup \{I, F\}, E)$ where $\mathcal{C}$ is the set of clusters, I the source node, F the final node, and E the set of edges. We place the edge (u, v) if there exists one input trace π in which a resource $r' \in u$ immediately precedes a resource $r'' \in v$. Then, for each r_j at the beginning of each trace (i.e. $\pi = \langle r_j, \ldots \rangle$), we place the edge (I, u) where $r_j \in u$ and for each r_j at the end of each trace, (i.e. $\pi = \langle \ldots, r_j \rangle$) we place the edge (u, F) where $r_j \in u$. Finally, we associate to each node u the set of all the elements for every $r \in u$.

B. Behavioral Patterns

Behavioral patterns are workflow and dataflow patterns that are likely related to the logic of the application. We divide workflow patterns into *Trace Patterns*, that model what users normally do in our input traces, and *Model Patterns* that model what the navigation graph allows to be done. Finally, *Data Propagation Patterns* model how data is propagated throughout the navigation graph.

1) Trace Patterns

Trace patterns model the actions performed by the user in the input traces. In particular, we focus on three patterns:

Singleton Nodes
A node is a singleton if it is never visited more than once by any input trace. Some of the users may visit these nodes, and some may not - but no one visits them twice. For example, submitting a discount voucher can be an operation observed in some of input traces but none of them is submitting a voucher twice.

Multi-Step Operations
A Multi-Step Operation is a sequence of consecutive nodes always visited in the same order. This is very common in many functionalities in web applications. For example, payment procedures or user registrations often consist of a precise sequence of steps, and all

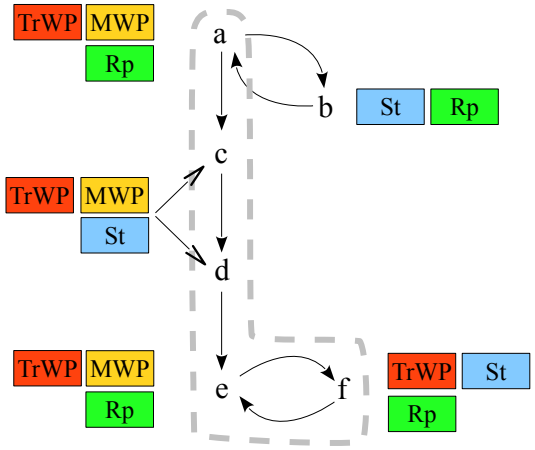


Fig. 5: Example of behavioral patterns using $\pi_1 = \langle a, b, a, c, d, e, f, e \rangle$ and $\pi_2 = \langle a, c, d, e, f, e \rangle$

traces going through those processes always execute them in the same exact order.

Trace Waypoints
We use the term *waypoint* to describe nodes that play an important role in the interaction between the user and the application. In particular, trace waypoints are those nodes that appear in all the input traces. For example, if all our traces contain a purchase, then the redirection to the payment website (e.g., PayPal) is a trace waypoint.

2) Model Patterns

Model patterns model the sequences of actions that are allowed according to the navigation graph:

Repeatable Operations
Nodes that are part of a loop in the navigation graph are associated to operations that can potentially be repeated multiple times.

Model Waypoints
Model waypoints are nodes that belong to every path in the navigation graph that goes from the source node to the final node. These nodes are not only visited in all input traces, but there is no way in the navigation graph to bypass them. By definition, every model waypoint is also a trace waypoint but not vice versa.

Figure 5 shows an example to better describe the difference between model and trace patterns. The example shows the behavioral patterns of a navigation graph extracted from two input traces $\pi_1 = \langle a, b, a, c, d, e, f, e \rangle$ and $\pi_2 = \langle a, c, d, e, f, e \rangle$. The symbols St, TrWP, Rp, and MWP stand for, respectively, singleton nodes, trace waypoints, repeatable nodes, and model waypoints. The thick dotted line delimits the multi-step operation.

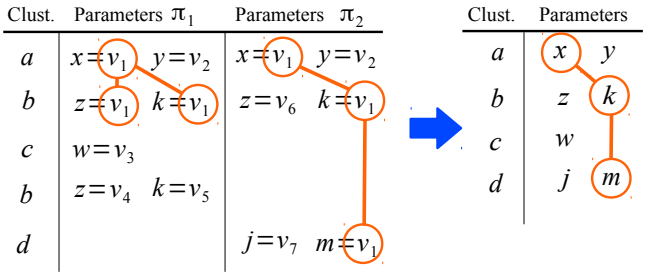


Fig. 6: Propagation Chains: from traces to the navigation graph

3) Data Propagation Patterns

A propagation chain is a set of parameters with the same value which is sent back and forth between the client and the web application during the HTTP conversation. We say that two parameters have the same value if there are some input traces in which they hold the same value, and there are no traces in which the values are different (since the user does not perform the same actions in all the traces, a certain parameter may not be present in all of them). We say that the chain is *client generated* if the initial value is chosen by the user, and *server generated* otherwise. A similar classification is used by Wang et al. [34]. However, their notion is limited to single input traces while ours is extended to traces of different lengths and to the navigation graph.

We compute propagation chains in two steps. First, we identify the propagation chain of each value within a trace. Let us consider the example in Figure 6. Here, in the input trace π_1, the parameter x has the same value of z and of k. In trace π_2, the parameter x is still equal to k, but it is now different from z. Moreover, the same value matches the parameter m. Second, by comparing the chains of traces, we remove contradictions reaching the result shown in the right side of Figure 6.

C. Test Case Generation

In this section we describe the generation of test cases. This is done by adopting a number of attack patterns that model how an attacker can use the application in an unconventional way. In particular, we focus on a set of actions an attacker could perform: repeating operations, skipping operations, subverting the order of operations, and mixing parameter values across user sessions. For each action we designed a pattern. These patterns are presented in Figure 7 and are based on the navigation graph of Figure 5. We enriched Figure 7 with numbers for showing the order in which the nodes are visited. For simplicity, we are omitting the source node I and the final node F, respectively connected to a and e.

It is important to note that, while the approach presented in this paper is generic, the choice of the attack patterns needs to reflect a particular class of logic flaws (in our case, the subversion of either the control or data-flow of the application). Other types of logic vulnerabilities, such as authentication bypass, may require the use of other patterns (e.g., randomly access administration pages) that could be added to our system

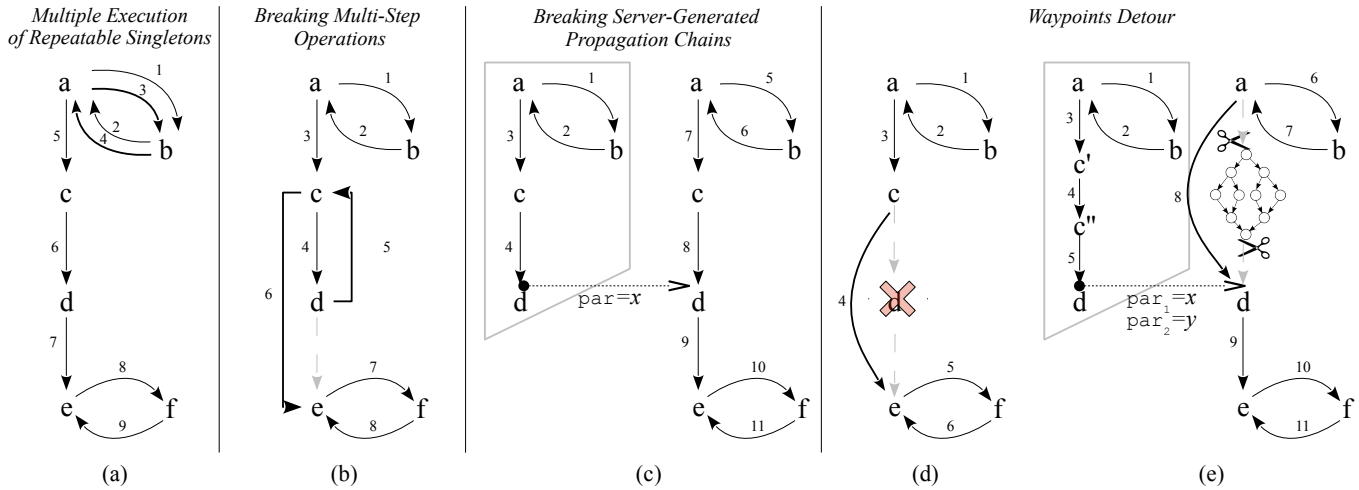


Fig. 7: Test case generation patterns

but that are outside the scope of our paper. However, the use of custom techniques to detect certain vulnerabilities is common to many other tools and approaches - e.g., a technique designed to find SQL injections cannot be used out of the box to detect other types of input sanitization vulnerabilities.

1) Multiple Execution of Repeatable Singletons

This pattern models an attacker that tries to execute an operation several times. If the model has a node that is repeatable and singleton, it means that even though there is a way to repeat an operation multiple times, this was never observed in our input traces. Therefore, the attacker tries to visit it twice.

Figure 7.a shows the steps of the test case. We select an input trace that visits b (e.g., $\langle a, b, a, c, d, e, f, e\rangle$), a repeatable and singleton node. Then we split it into two parts at the node after the singleton (e.g., $\langle a, b\rangle$ and $\langle a, c, d, e, f, e\rangle$). We call these two parts *prefix* and *suffix*. Second, we find the shortest loop from the singleton node to itself (e.g., $\langle b, a, b\rangle$). Finally, the test case is the concatenation of the prefix, the loop without the first node, and the suffix.

2) Breaking Multi-Step Operations

This pattern models an attacker that breaks multi-step operations. For example, once the payment page is reached, the attacker goes back and adds an item into the cart. In general, there are several ways of breaking the multi-step operation of Figure 5. The first approach is to use a different ordering (e.g., $\langle a, d, c, e, f\rangle$). A second approach is to interleave other steps. In this pattern, we focused on the latter approach in which we repeat a step already included in the multi-step later in the test case. For example, in the test case $\langle a, c, d, c, e, f\rangle$ in Figure 7.b we repeat c after d. In this pattern, we repeat c also after e and f, but not after a.

3) Breaking Server-Generated Propagation Chains

The goal of this attack pattern is to tamper with the data flow of the web application. An example of test case is shown in Figure 7.c. The first part of the test interacts with the application and captures the value x of a server-generated propagation chain. In the second part, we start another session and interrupt the propagation chain by replacing the value of `par` with x.

Since web applications contain many server-generated propagation chains (e.g., all the item or message IDs), this attack pattern may generate a very large number of test cases. Therefore, we focus only on two types of propagation chains: the ones containing unique values (i.e., that differ in all the input traces and are therefore related to the session) and the ones containing installation-specific values (i.e., values that are constant only within the same installation).

The test case generation is the following. First, we select the parameters belonging to the chain that appear inside an HTTP request. These parameters are called *injection points* and model the point in which an attacker can replace the value generated by the server. For example, in Figure 7.c the parameter `par` of the node d is an injection point. Second, we select two traces from different user sessions that are visiting the node of the injection point. The first is truncated at the injection point and the second is appended to the first one. With reference to Figure 7.c, the two parts are respectively at the left- and right-hand side.

4) Waypoints Detour

Waypoints are operations that are executed always by all the input traces such as payment, or providing shipping data. When these operations happen only once per input trace, they seem to indicate some sort of milestone in the execution of the business process of the web application. In the waypoint detour pattern, the attacker tries to skip these type of operations by using one of two possible techniques. If the waypoint node is not part of a propagation chain, we simply try to skip it.

Otherwise, we try to remove the part of the navigation graph between two waypoints, reconstructing the propagation chains by fetching the missing data values from another user session.

Figure 7.d shows an example of this pattern. On the left side we skip the waypoint d, while on the right side we cut the subgraph between a and d. In this second case, if the URL of node d depends on a value that appears in the segment between a and d, we prepare another user session by selecting an input trace and interrupting it at d. The generation of this part is similar to breaking propagation chains. The first user session is then $\langle a, b, a, c', c'', d\rangle$. Afterwards, we prepare the second user session that skips the sequence between a and d. In this example, there are two possibilities: skipping $\langle b, a, c', c''\rangle$ or $\langle c', c''\rangle$. Figure 7.e shows only the latter. In this case the test case is the concatenation of $\langle a, b, a, c', c'', d\rangle$ and $\langle a, b, a, d, e, f, e\rangle$. For this case we also support the variant in which the first user session is not interrupted at the node d.

D. Test Case Execution

The test cases described in Section II-C are abstract and still miss the details to be properly executed. For example, the values of some parameters cannot be determined from the model and need to be collected during the test case execution. In addition, it is important that after each test the application is reset to its initial state to avoid interferences between consecutive executions. For example, a test may leave a number of items in the shopping cart, thus affecting following experiments. In general, it is often sufficient to delete the cookies and empty the shopping cart at the end of each test.

The execution engine iterates over each node of the test case, concretizes the POST/GET parameters, and submits the HTTP request. The responses are parsed according to the propagation chains in order to extract server-generated parameters to be used in latter requests. If the execution engine is not able to properly reconstruct a chain (e.g., because the page that was supposed to generate its value returned an error) the execution engine aborts the execution and reports that the test was *not executed*; it reports *executed* otherwise.

E. Test Oracle

The approach we propose in this paper is completely independent from the business logic of the web application. Our technique can automatically identify behavioral patterns, and then generate test cases to break those patterns in a number of different ways. The system can also determine if a given test was executed correctly, but this is as far as it is possible to go with an application-agnostic approach. For example, replacing the value of a security token in a payment workflow would probably make the entire process fail. Unfortunately, without any knowledge about the underlying business logic, the test verdict could only say whether the pattern was applied successfully, but it can not draw any conclusion about the possible implications. Therefore, if we want our tool to be able to report possible violations of the application logic, we need to extract the sequence of events that occur during a test case execution and compare them with the *logic property* that we want to violate.

A simple way to express a logic property for shopping carts could be the following: *if an order is approved for a user, then the user must have completed a payment for the corresponding amount*. In this formulation two events play a central role: the fact that an order is placed, and the fact that a user has paid a certain amount. Another important aspect of this property is the time dependency between the two events. Since propositional logic can only express truth regardless of the time, in our approach, we model logic properties as Linear Temporal Logic (LTL) formulas [30, 23]. LTL adds temporal connectives like $\mathtt{O}$ (once in the past) to traditional logical operators like $\wedge$ (and), and $\implies$ (implies). This enables us to verify whether one event will eventually happen in the future or it already happened in the past.

For example, the above logic property can be written in LTL as follows:

$$ord_{placed} \wedge onStore(S) \implies \mathtt{O}(paid(U, I)) \qquad (1)$$

where ord_{placed}, $onStore(S)$, and $paid(U, I)$ are respectively the events *order placed*, *operation performed on the store* S, and *user* U *paid the price of item* I. Now, the problem of identifying a violation of the logic property is recast into the problem of checking whether the LTL formula is satisfied or not by a given test case.

In our approach, the *Test Oracle* is the component that given an execution of a test case returns *true* if a certain predefined logic property is violated, and *false* otherwise. The oracle is composed of two parts: an *events extractor* and an LTL formula checker. The extractor collects from the executed test a partially ordered set of events (events can happen in sequence or in parallel) grouped by user sessions. The second part verifies whether all sequences satisfy the provided LTL formula.

It is important to note that both the events and the LTL formula depend on the type of applications under test and on the type of vulnerabilities that we are interested to find. For example, to find authentication bypass vulnerabilities it would be interesting to observe events related to the user login and to the access of private pages. However, since in this paper we focus on the test of E-commerce applications, we are more interested in monitoring the money transfer and the value of the purchased items, as described in more details in the next section.

III. Experiments

We could use our tool to test online stores (e.g., Amazon). However, our tests require to attempt malformed operations and to complete a large number of checkout processes. This would be both unethical, since the application can malfunction as a result of our tests, and very expensive, since it requires to buy at least one product for each test case. Therefore, we opted to run our tests on seven well-known open source applications available for offline testing, as reported in Table I. The table also shows an estimation of their popularity, measured with the search results obtained by performing a

Web App.	No. of Installations
OpenCart	9,710,000
Magento	3,130,000
PrestaShop	650,000
CS-Cart	260,000
TomatoCart	119,000
osCommerce	80,500
AbanteCart	21,200
Total	13,970,700

TABLE I: Popularity index

number of *googledorks* [1]. Each Google query was built by combining both the URL structure (e.g., the path of the checkout endpoint) and some static HTML content extracted from the web pages (e.g., "powered by…" of the footer). As such, the numbers reported in the table are only a lower bound of the number of publicly-accessible installations available on the Internet. This conservative measurement shows that these seven applications are used by almost 14 million E-commerce installations. As a comparison, the two applications tested by Wang et al. [34] returned less than 40,000 hits using similar Google dorks.

A. General Setup

We installed two instances of each web application (hereinafter Store A and Store B). All installations except for AbanteCart and PrestaShop were then configured to use both the PayPal Express Checkout [3] and the PayPal Payments Standard [4] methods. In total we prepared 12 configurations[1].

All applications were configured in *sandbox* mode. In this configuration, each application performs transactions by using the PayPal sandbox payment gateway. These payments do not involve real money as they are performed between the seller and buyer testing accounts.

B. Testing Oracle

In their experiments, Wang et al. [35] used the following shopping cart property:

"The store S changes the status of an item I to "paid" with regard to a purchase being made by user U if and only if (i) S owns I; (ii) a payment is guaranteed to be transferred from an account of U to that of S in the CaaS; (iii) the payment is for the purchase of I, and is valid for only one piece of I; (iv) the amount of this payment is equal to the price of I."

However, this property is not entirely verifiable in a black-box setting. For instance, it is not possible to test the truth of the predicate "S owns I" nor to check whether the due amount has been transferred to the merchant's account. Therefore, we simplified the above invariant by removing the non-verifiable clauses. The new property that can be used for automated black-box testing becomes:

[1]When we did the experiments, AbanteCart and PrestaShop were providing, respectively, only PayPal Payments Standard and PayPal Express Checkout.

When the store S confirms the user U that an order has been placed, then in the past U paid S the amount equal to the price of I and U agreed on purchasing I from S.

We modeled the logic property using the following events extracted during each test case execution:

- ord_{placed} when the shop confirms that the order has been placed;
- $onStore(S)$ when an operation has been performed on the store S;
- $paid(U, I)$ when the user U authorizes the payment gateway to pay the price of I;
- $toStore(S)$ when the payment is meant for the store S;
- $ack(I)$, when the user acknowledges to buy I.

The logic property is then formulated as:

$$\begin{aligned} & ord_{placed} \wedge onStore(S) \implies \\ & \quad \mathtt{O}(paid(U, I) \wedge toStore(S) \ \wedge \\ & \qquad \mathtt{O}(ack(U, I) \wedge onStore(S))) \end{aligned} \tag{2}$$

C. Input Traces

To generate the input traces we created two user accounts, U_1 and U_2, each controlling a PayPal buyer testing account. For each web application we captured in total six HTTP conversations, three for each store: one with U_1 buying one item, one with U_2 buying another item, and one with U_1 buying two different items. All the input traces satisfy the logic property 2. These input traces were sufficient to stimulate the main shopping cart functionalities, but a better training could be used in the future to expose also more subtle features, or for detecting different types of logic flaws.

D. Test Case Generation

Table II shows the test cases grouped by attack pattern. The test case generation produced about 3100 test cases, an average of 262 per application. Table II shows also the test execution result. An execution failed when the test case brought the application in a state in which it was impossible to proceed (e.g., because of error pages in intermediate steps). This is a common result, since by definition our tests stress the application to expose some unexpected behavior. The number of test cases violating the LTL formula is reported in Table III. As mentioned before, there are events that are not visible to the oracle. Therefore, a violation to the LTL formula does not always correspond to a vulnerability. In fact, it is possible that further checks performed in the back end of the application would detect and block the attack. To distinguish logic vulnerabilities from other bugs (e.g., erroneously reporting to the user a failed transaction as successful) we manually inspected the balance sheets of the merchant, the list of orders, and their status. Whenever the result was not confirmed by our manual

		Test Case Generation					Test Case Execution			
Web App.		Time hh:ss	(a)	(b)	(c)	(d), (e)	Time hh:ss	Exec.	Not Exec.	Total
AbanteCart	Std	$\ll$ 00:01	9	51	21	152	04:51	74	159	233
Magento	Exp	00:02	10	82	5	246	16:23	240	103	343
	Std	00:02	14	62	7	303	14:50	210	176	386
OpenCart	Exp	00:01	10	77	3	83	02:34	140	33	173
	Std	00:01	15	38	22	60	02:08	71	64	135
osCommerce	Exp	$\ll$ 00:01	4	13	6	142	03:22	117	48	165
	Std	00:01	8	63	10	144	03:42	128	97	225
PrestaShop	Exp	$\ll$ 00:01	12	22	3	100	02:42	85	52	137
TomatoCart	Exp	00:02	9	68	10	215	04:54	238	64	302
	Std	00:02	17	32	37	138	04:36	115	109	224
CS-Cart	Exp	00:05	8	24	6	562	12:02	347	253	600
	Std	00:02	16	54	15	137	05:29	127	95	222
Total			132	586	145	2282		1892	1253	3145

TABLE II: Statistics per application on the test case generation and test case execution phases. Columns (a), (b), (c), (d), and (e) are the attack patterns in Figure 7 while columns Exec. and Not Exec. refer to the two possible outcomes of the test execution engine.

		No.	Caused by	
Web App.		of Viols.	Bugs	Vulns.
AbanteCart	Std	17	16	1
Magento	Exp	65	65	-
	Std	126	126	-
OpenCart	Exp	58	46	12
	Std	30	30	-
osCommerce	Exp	42	22	20
	Std	35	34	1
PrestaShop	Exp	-	-	-
TomatoCart	Exp	90	65	25
	Std	24	24	-
CS-Cart	Exp	313	313	-
	Std	109	108	1
Total		909	849	60
		100%	93.4%	6.6%

TABLE III: Number of test cases violating Property 2 and the root cause.

inspection, we classified it as a normal bug. The remaining cases correspond instead to anomalous behaviors associated to real software vulnerabilities, as explained in the next Section. It is important to note that over 28.9% of the test cases generated by our approach brought the application in a state that violated the LTL formula, and 1 test out of 52 exposed a previously-unknown logic vulnerability.

Test case generation does not require much resources, while the execution phase can be quite time consuming (16h for Magento). This is largely due to the lack of parallelization in our experiments, and to the fact that the PayPal sandbox is much slower than its live counterpart. The model inference – omitted from Figure II – required an average of 9m per application to build the navigation graphs that, in average, contained 34 nodes and 48 edges.

IV. Results

Table III reports the total number of violations of the security property 2. In other words, by tampering with either the workflow or the data flow according to our attack patterns, our system was able to bring the web application in a faulty state in 909 cases. All these cases corresponded to tests that were executed until the final page in which the store congratulates the customer for the successful purchase (that caused the generation of the events $ord_{placed} \wedge onStore(S)$) even though the paid amount was not correct. While these violations are all the consequences of bugs in the application code, not all of them can be exploited by an attacker.

This is an important point and a fundamental limitation of black-box approaches. Our tool can only observe the application state "from the outside", and therefore it cannot distinguish between a presentation bug (in which the information displayed on the web pages are wrong but the internal state of the application is correct) and a more serious vulnerability (in which also the internal state is compromised).

To distinguish between the two types of bugs, we manually inspected the state of the backend database: the result is the distinction summarized in Table III between harmless presentation bugs (93.4%) and real vulnerabilities (6.6%). While these results indicate that the *true positive* rate of our tool is 6.6%, also the remaining 93.4% of the violations correspond to real bugs in the application that need to be fixed by the developers. Once all the presentation issues have been solved, the alarms raised by our tool would correspond only to exploitable vulnerabilities.

A. Vulnerabilities

Table III shows that 60 of our test cases (1.9% of the total) exposed a logic vulnerability in the target applications. We discovered the following flaws:

- In osCommerce 2.3.1, CS-Cart 3.0.4, and AbanteCart 1.0.4 with PayPal Payments Standard a malicious

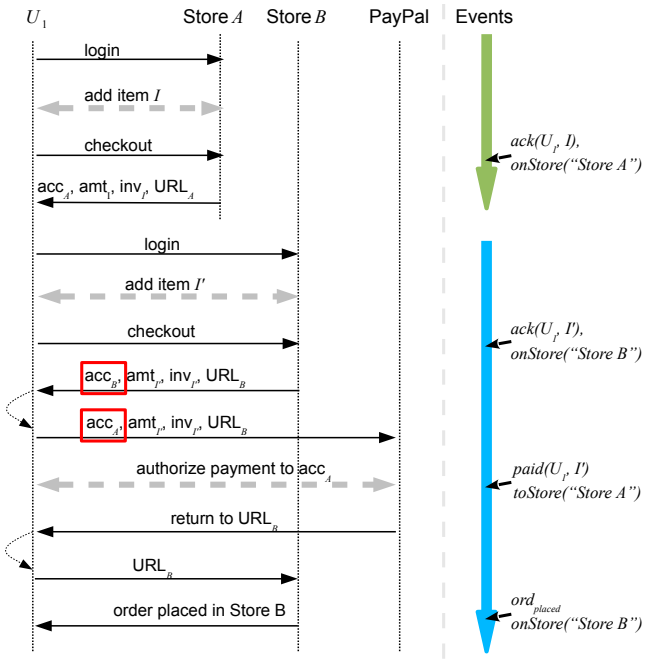


Fig. 8: Shopping for free with osCommerce 2.3.1 and AbanteCart 1.0.4

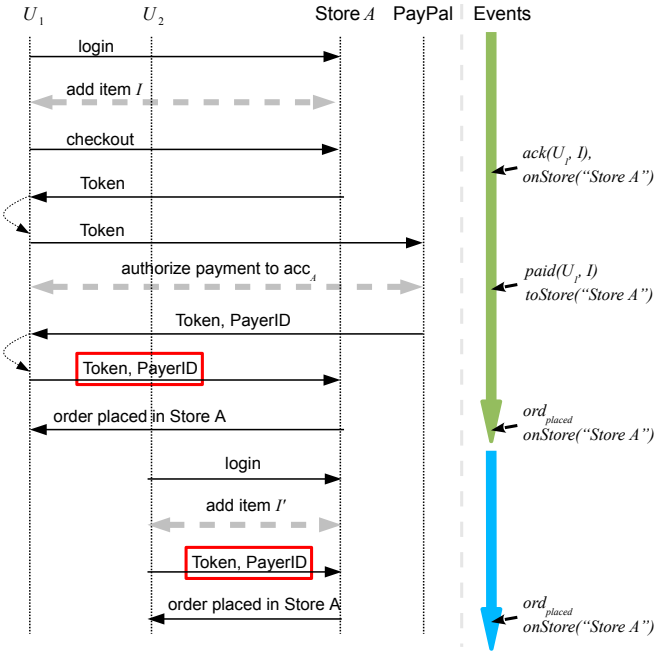


Fig. 9: Paying less with OpenCart 1.5.3.1 and TomatoCart 1.1.7

customer can shop for free (exploitable)

- In OpenCart 1.5.3.1 and TomatoCart 1.1.7 with PayPal Express Checkout a malicious customer can pay less (exploitable)
- In TomatoCart 1.1.7 with PayPal Express Checkout a malicious customer can shop for free (exploitable)
- OpenCart 1.5.3.1, TomatoCart 1.1.7, and osCommerce 2.3.1 with PayPal Express Checkout a customer can pay an amount different from what she authorized (not exploitable)
- TomatoCart 1.1.7 with PayPal Express Checkout a customer pays another customer's cart (not exploitable)

All the exploitable flaws have been already responsibly disclosed. When the developers did not answer within two weeks of our notification, we reported the vulnerabilities also to the US Cert[2]. In the following we describe each class of vulnerability we discovered in our experiments.

1) osCommerce, CS-Cart, and AbanteCart with PayPal Payments Standard - Shopping for Free

These flaws were discovered by tests that interrupted the server-generated propagation chain transporting the PayPal account of the merchant. An example is shown in Figure 8. The left-hand side of the Figure shows the message sequence chart while the right-hand side shows events grouped by user session. Each user session begins with a $login$ message. The events show how the violation was detected by the oracle. At the end of the execution, $ord_{placed} \wedge onStore(\text{“}Store\ B\text{”})$ is satisfied as all the events in it were observed. However, the left-hand side of the Formula (1) is not satisfied because none of the events in it were observed.

The manual inspection verified that (i) no payment was made to the Store B, (ii) the status of the order in the back office of Store B was "completed", and (iii) the invoice was paid. It is straightforward to turn the above test into a real attack. Indeed, when redirected to PayPal, an attacker can replace the seller PayPal account with another PayPal account under her control. In this case, the attacker can pay herself for an item she buys in an online shop.

2) OpenCart and TomatoCart with PayPal Express Checkout - Pay Less

In OpenCart and TomatoCart with PayPal Express Checkout an attacker can pay less than the value of the items. The flaw has been detected by using the waypoints detour pattern. The test case generator produced 11 test cases for OpenCart and 11 for TomatoCart in which the user U_2 skips the nodes of the redirection to PayPal for the payment and reconstructs the URL with values taken from the user session of U_2. A representative test case is shown in Figure 9. In the second user session $ord_{placed} \wedge onStore(\text{“}Store\ A\text{”})$ is satisfied. However, the other clauses of the formula are not satisfied because neither the user acknowledgment nor the payment were observed.

The manual inspection found two distinct orders in the list of orders, one for I and for I'. Both orders were in the state "paid" and ready for shipping. However, the balance sheet of the merchant contains only the transaction for I, while nothing is recorded for I'.

[2]See http://www.kb.cert.org/vuls, IDs 459446, 207540, and 583564

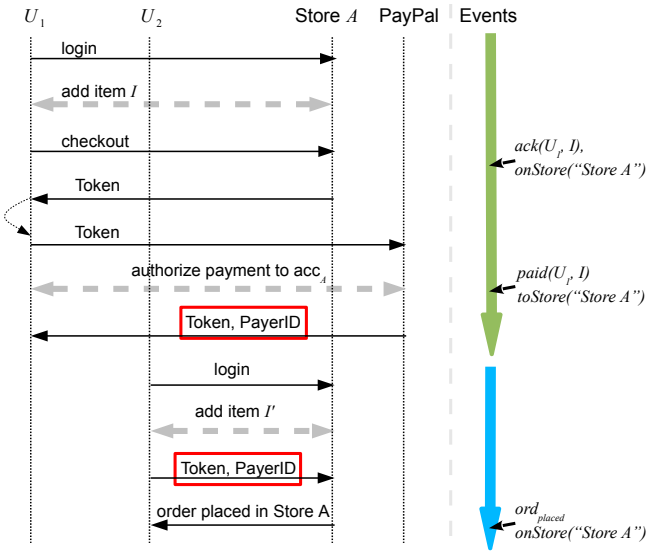


Fig. 10: Shopping for free with TomatoCart 1.1.7

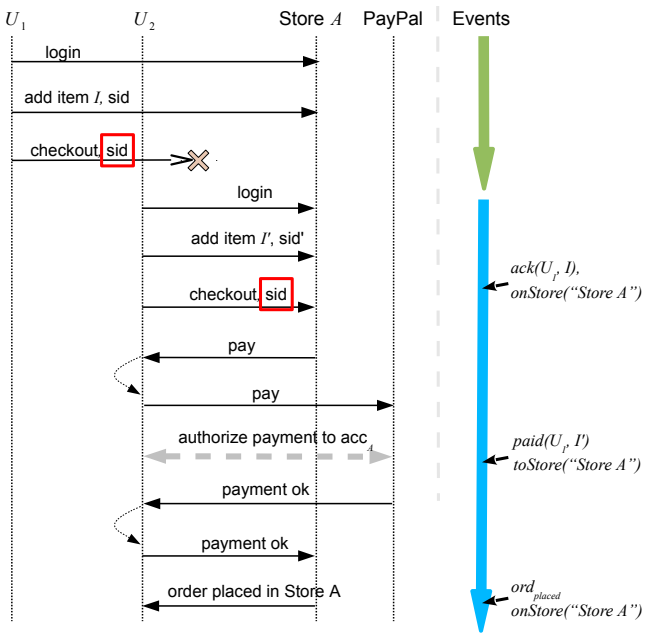


Fig. 11: Session fixation in TomatoCart 1.1.7

This test can be turned into an attack by first buying a cheap item and intercepting the redirection URL from PayPal to the store. Then the attacker can login again, add an expensive item to the cart, and replay the URL captured before to complete the transaction. Even worse, we verified that the attacker (or any other user) can reuse the same $TokenID$ and $PayerID$ to complete an arbitrary number of additional fake transactions. This process is only bounded by the timeout set by PayPal on the token.

3) TomatoCart with PayPal Express Checkout - Shopping for Free

This problem has been identified by 11 different test cases generated with the waypoint detour pattern. A representative test case is shown in Figure 10. Figure 10 shows that in the second user session $ord_{placed} \wedge onStore(\text{“}Store\ A\text{”})$ is satisfied. However, the other clauses of the formula are not satisfied because neither user acknowledgment nor the payment were observed.

The manual inspection verified that no payment for I and for I' were done. However, the list of orders contained the order for I' in a “paid” state and ready for shipping. This test case can be turned into an attack as shown before with the difference that the attacker ends the first user session after receiving $Token$ and $PayerID$ from PayPal.

4) osCommerce, OpenCart and TomatoCart with PayPal Express Checkout - Pay Less

In osCommerce the test was generated by the waypoints detour pattern, while in OpenCart and TomatoCart tests were generated by breaking server-generated propagation chains.

In osCommerce, the test is similar to the one shown in Figure 10 while for OpenCart and TomatoCart, the tests are similar to the one in Figure 8. When PayPal Express Checkout is selected, the store and PayPal are exchanging the $Token$ via redirections. Here, the pattern interrupted the chain of $Token$ when the user is redirected to PayPal for the payment. In both cases the oracle verified that the user U_2 had a confirmation and that $paid(U_2, I') \wedge toStore(A)$ is satisfied. However, the oracle could not verify $\texttt{O}(ack(U_2, I') \wedge onStore(A))$ because it observed $\texttt{O}(ack(U_2, I) \wedge onStore(A))$.

A manual inspection confirmed that only the order for I' was in the list of the orders with status “paid”, while the order for I was still “payment pending”. However, in the balance sheet of the merchant, the payment for I' was done by U_1 instead of U_2. In this case, U_1 authorized PayPal to pay for I while her credit card was charged for I'.

In order to turn this tests into a real attacks, the attacker needs to intercept the redirection URL that is carried over SSL/TLS channels. In addition, it must block the user-victim from executing the redirection. This could require the attacker to either break the SSL/TLS encryption layer or to mount a SSL/TLS MITM (Man-In-The-Middle) attack. However, in both cases the attacker will be able to capture also the payment data of the victim enabling her to shop for free in any case.

5) TomatoCart with PayPal Express Checkout - Session Fixation

Our experiments discovered a session fixation vulnerability in which U_2 could impersonate another user. The test cases were created by breaking the propagation chain of the parameter sid in two points. Figure 11 shows one of them. The events of Figure 11 did not satisfy the formula because the payment I' was of a different amount than the one the user acknowledged for I.

The parameter sid carries the same value in the cookie and breaking it causes a *session fixation* in which, in our case, U_2

results logged in as U_1. From that point on, U_2 can access the data of U_1. As a consequence, U_2 (now logged as U_1) pays the cart of U_1. However, we could not find any exploitation of this flaw. Supposing that the victim (i.e. U_2) "clicks" on an URL crafted by the attacker (U_1), then the victim could notice the fraud in three moments (i) when checking the summary of order, (ii) when providing the shipping address (it shows the attacker's one), and (iii) during the payment because the amount is different.

V. Limitations

Our approach uses attack patterns that tamper with the observed data flow and workflow. However, it does not test for other types of logic vulnerabilities such as unauthorized access to resources. Moreover, we did not consider cases in which the attacker can also play the role of a malicious store, or the cases in which the attacker can intercept and tamper with the messages between the application and the payment service. We believe that our techniques could also be effective at detecting other kinds of logic flaws, even though we have not experimentally tested this hypothesis. This could be achieved by adding input traces of privileged user (e.g., admin), by adding other behavioral patterns, or by adding new attack patterns.

Second, the test generation favors efficiency over coverage. This means that only a few values are used for each test category, to maximize the possibility to find bugs in a limited amount of time. A more thorough exploration of the attack space could be used to discover more vulnerabilities, however this could require a considerable amount of execution time. The focus of this paper is to show how an automated approach can be used to find logic vulnerabilities in many real-word applications, and not to analyze in depth a single application (a scenario that would also require more input traces to better explore the application's logic).

Finally, we modeled logic properties in LTL. The use of LTL enables us to verify events with time dependency. However, LTL do not support algebra whose terms appear at different moment of the execution. For example, our oracles cannot verify whether the payment is the sum of the items the user added into the cart at some point in the past. There are works that extend LTL with constraints on integer numbers [10], and they could be used by our oracle for checking more fine-grained properties.

VI. Related Work

A large number of solutions have been proposed to detect vulnerabilities in web applications. However, most of the previous work focus on the automated detection of well-known classes of vulnerabilities related to insufficient input validation, such as Cross-Site Scripting (XSS) [26], Cross-Site Request Forgery (CSRF) [2, 27] and SQL injection [22]. Since our goal is to find logic flaws, we will not present these solutions in this section.

a) Detection of Logic Vulnerabilities

When the source code of the application is available, tools such as MiMoSA [9], Waler [21], and Swaddler [16] can be used to discover logic vulnerabilities. MiMoSA and Waler extract a model from the source code and then use a model checker to detect a violation of invariants. Swaddler [16] detects attacks when the software is at the deployment phase of its life-cycle. It first learns the normal behavior of the application and then monitors state variables at runtime looking for deviations from the normal behavior.

When the source code is not available, the problem of extracting a model becomes more difficult. Doupé et al. [19] and Li and Xue [28] proposed two black-box testing tools. The former presents a state-aware input fuzzer to detect XSS and SQLi vulnerabilities. The tool infers a model that is used as an oracle for choosing the next URL to crawl. Both our approach and this technique infer models to improve the automatic detection of vulnerabilities. However, we use a passive learning technique tailored to generate test cases to detect logic flaws, and not an active scanning to drive an input fuzzer. The second work presents BLOCK, a tool that learns model and invariants by observing HTTP conversations and then detects authentication bypass attacks. As opposed to BLOCK our approach does not aim at intercepting attacks, but at generating security tests for detecting flaws. Both works could not be used to find this class of vulnerabilities. The former work proposes a stateful crawler with an input fuzzer that does not attempt to violate the logic of the application. The latter focuses on the detection of authentication bypass attacks by inferring session variable invariants.

An approach similar to BLOCK is InteGuard [37]. InteGuard aims at protecting multi-party web applications from exploitation of vulnerabilities in the API integration. InteGuard focuses mainly on the browser-relayed messages in which data values are exchanged between the parties through the web browser. In particular, InteGuard uses a passive model inference technique based on data-flow analysis and differential analysis to extract inter-services dataflow-related invariants. The former is used to extract the flows of data values while the latter is used to detect properties of data flows such as transaction-specific or implementation-specific values. Our approach uses similar techniques to extract these type of invariants. However, in addition to that, it extracts also invariants of the observable workflow of the application, and takes into account both intra-service invariants, e.g., idempotent operations, and inter-service invariants, e.g., multi-step operations.

Given the limited success of automated black-box techniques, manual methodologies have been recently proposed. Our work is mainly inspired by Wang et al. [34, 35], who presented an analysis of Cashier as a Service (CaaS) based web stores, and a large-scale analysis of web Single Sign-On protocols. The former work describes a black-box methodology that given a number of HTTP conversations, labels API arguments and shows with which ones an attacker could play in the attempt of violating security invariants. The latter refines the previous one by (i) considering the role played by the attacker during the protocol execution and (ii) adding semantic and syntactic labels to protocol parameters. Both techniques

had a large impact due to the severe vulnerabilities the authors were able to find in real-world applications. However, these papers propose techniques and guidelines that need to be manually applied by a security expert. Our work extends their technique in four ways. First, it infers a model from set of HTTP conversations. Second, it generalizes the notion of propagation chain of a single trace into propagation chain of an application model. Third, it infers observable characteristics of the workflow of the business function. Finally, it automatically generates and executes test cases using a number of attack patterns.

AUTHSCAN [8] is an approach similar to our work. It infers a model from implementations combining white-box and black-box techniques. AUTHSCAN focuses on the detection of flaws specific to authentication protocols (See Lowe et al.[29] for a survey of authentication property) and it requires a list of application-specific JavaScript function signatures in order to infer an accurate model of the protocol participants. On the contrary, our approach focuses on business-related web application properties and uses an application-independent model inference technique.

b) Model Inference

There is a large body of works addressing the problem of inferring a model for testing purposes. Model inference is divided in two categories: active learning and passive learning. Active learning techniques interact with the application under inference in order to explore its behavior whereas passive learning techniques build a model from a set of observations. Hossen et al. [25] proposed to apply the active learning algorithm L* [5] to infer a deterministic finite automaton and refining it with testing. Dury et al. [20] described an approach based on passive learning of web-based business applications. They used Parameterized Finite Automaton (PFA) that enriches the classic notion of finite automaton [24] with guards on transitions and parameters on states. PFA models control flow and data flow of an application. Guards are inferred using data mining algorithms like C4.5 [31]. Models are then translated into the Promela language and fed to the model checker SPIN [23] for verifying application-dependent properties. However, in the first approach the authors proposed a direction and say little on the type of flaws they aim at detecting, while in the second the authors focus on the inference part and do not cover the actual testing.

c) Model-Based Security Testing

New ideas have been proposed in order to use models for the (semi-)automatic security testing of web applications when models are available. For example, Armando et al. [7] proposed to detect logic flaws and testing web-based security protocols. The approach consists of using the SAT-based Model Checker [6] to validate a formal specification against security desiderata. If a violation occurs, it is executed against a real implementation. Büchler et al. [13] proposed an approach that assumes (*i*) a model is given (*ii*) and the model is secure. Then they propose to mutate the model by injecting vulnerabilities and to use a model checker for detecting violations. If a problem is found, then they use the counterexample returned by the model checker as an abstract test case for testing implementations. Bodei et al. [11] proposed to model Service-Oriented applications in CaSPiS (Calculus of Services with Pipelines and Sessions), a process calculus with the notion of session and pipelines [12], to perform a control flow analysis for detecting misuse of the application. The authors tested their technique on a known vulnerable version of the CyberOffice shopping cart detecting the price-modification attack. However, for all these works still remains the problem that a model of the application is often not available in practice.

VII. Conclusions

In this paper we presented a new technique for the black-box detection of logic flaws in web applications. Our approach uses a passive model inference technique that builds a navigation graph from a set of network traces. We then apply a number of heuristics to extract behavioral patterns that are likely related to the underlying application logic. These behaviors, together with a number of attack patterns, are used for generating test cases.

We developed a prototype tool and tested seven E-commerce applications. The prototype generated and executed more than 3100 test cases, 900 of which violated the expected behavior of the application. As a result, our tool detected ten previously-unknown logic vulnerabilities in the applications under test. Five of them allow an attacker to pay less or even shop for free.

Acknowledgment

This work has been partially supported by the European Union Seventh Framework Programme under grant agreement no. 257007 (project SysSec) and no. 257876 (project SPaCIoS Secure Provision and Consumption in the Internet of Services).

References

[1] "The google hacking database at hacking for charity." [Online]. Available: http://www.hackersforcharity.org/ghdb/

[2] "Requestrodeo: Client side protection against session riding," in *the OWASP Europe 2006 Conference, Report CW448, Departement Computerwetenschappen, KU Leuven, May 2006*, 2006.

[3] "Paypal express checkout integration guide," August 2012. [Online]. Available: https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_ExpressCheckout_IntegrationGuide.pdf

[4] "Paypal payments standard integration guide," June 2012. [Online]. Available: https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_WebsitePaymentsStandard_IntegrationGuide.pdf

[5] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, Nov. 1987.

[6] A. Armando, R. Carbone, and L. Compagna, "Ltl model checking for security protocols," in *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, July 2007, pp. 385–396.

[7] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti, "From model-checking to automated testing of security protocols: Bridging the gap," in *TAP*, ser. LNCS, A. D. Brucker and J. Julliand, Eds., vol. 7305. Springer, 2012.

[8] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, San Diego, California, USA, February 24-27, 2013.

[9] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007.

[10] M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi, "Smt-based verification of ltl specifications with integer constraints and its application to runtime checking of service substitutability," *CoRR*, vol. abs/1004.2873, 2010.

[11] C. Bodei, L. Brodo, and R. Bruni, "Static detection of logic flaws in service-oriented applications," in *ARSPA-WITS*, ser. LNCS, P. Degano and L. Viganò, Eds., vol. 5511. Springer, 2009.

[12] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti, "Sessions and pipelines for structured service programming," in *FMOODS*, ser. LNCS, G. Barthe and F. S. de Boer, Eds., vol. 5051. Springer, 2008.

[13] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *SERE*. IEEE, 2012.

[14] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *Proceedings of the 16th ACM Conference on Computer and Communication Security*, Chicago, IL, November 2009.

[15] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 110–125. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.14

[16] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *RAID*, ser. LNCS, C. Krügel, R. Lippmann, and A. Clark, Eds., vol. 4637. Springer, 2007.

[17] D. Crockford, "RFC4627: The application/json media type for javascript object notation (json)," July 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627

[18] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011.

[19] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the 2012 USENIX Security Symposium (USENIX 2012)*, Bellevue, WA, August 2012.

[20] A. Dury, H. H. Hallal, and A. Petrenko, "Inferring behavioural models from traces of business applications," in *Proceedings of the 2009 IEEE International Conference on Web Services*, ser. ICWS '09. Washington, DC, USA: IEEE Computer Society, 2009.

[21] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010.

[22] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[23] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[24] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[25] K. Hossen, R. Groz, and J. Richier, "Security vulnerabilities detection using model inference for applications and security protocols," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011.

[26] M. Johns, "Code injection vulnerabilities in web applications: Exemplified at cross-site scripting," Ph.D. dissertation, 2011.

[27] N. Jovanovic, E. Kirda, and C. Kruegel, "Preventing cross site request forgery attacks," in *SecureComm*. IEEE, 2006.

[28] X. Li and Y. Xue, "Block: a black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011.

[29] G. Lowe, "A hierarchy of authentication specifications," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, 1997, pp. 31–43.

[30] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE Computer Society, 1977.

[31] J. R. Quinlan, *C4.5: programs for machine learning*. San

Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[32] The jQuery Foundation, "jQuery," January 2013. [Online]. Available: http://jquery.com/

[33] The OWASP Foundation, "OWASP Testing Guide," December 2008. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[34] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services." in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.

[35] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online – security analysis of cashier-as-a-service based web stores," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011.

[36] World Wide Web Consortium, "Simple Object Access Protocol (SOAP) 1.2," April 2007. [Online]. Available: http://www.w3.org/TR/soap/

[37] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013*, San Diego, California, USA, February 24-27, 2013.

A Play with Many Actors

9

A Change of Perspective

In the previous chapters we discussed several classic and less classic vulnerabilities that affect web applications. But to fully understand why the security of the average website is so poor, it is not enough to look at the code. In fact, the security of web applications is a multi-faced problem that involves many different players. However, the vast majority of the research in this area focuses either on the *developers*, as they are the responsible to write the source code and introduce the vulnerabilities in the first place, or on the *final users*, as they are often the weakest link in the chain and the harder to secure against an attacker. The *system administrator* has also being partially explored, with a number of server-side solutions that can either prevent certain classes of vulnerabilities or mitigate the risk of exploitation.

In this final part of this document I want instead to present a parallel line of research in which I investigated the role of other players that play an important role in this ecosystem but that were often overlooked by previous researchers: the *hosting providers*, and the *attackers* themselves.

I believe it is extremely important to study these actors to better understand the general picture of web security. It is true the untrained developers and insecure ready-to-deploy web applications are the main cause of many vulnerable web sites. But this is only part of the problem. Web hosting providers offer cheap, and sometimes even free, space to host simple web applications and are the preferred choice for hundred of thousands of personal pages and little e-commerce solutions. If their clients are in large part unable to detect when their pages have been compromised, the hosting providers are in the perfect position to detect suspicious behaviors, notify their customers, and stop malicious pages that would otherwise remain unnoticed for long time.

On the other hand, studying the behavior and the motivation of the attackers allow us to better understand what they are after when they compromise a web application and how we can detect them more efficiently.

Summary

The first paper, published at the World Wide Web conference in 2013, focuses on the role of the web hosting providers. By re-creating realistic (but harmless) malicious behaviors that were clear and easily detectable signs of a compromise, we were able to estimate if (and how well) popular hosting providers all around the world could detect these signs and notify their customers. We also tested how the same providers reacted once contacted by users that wanted to report a suspicious behavior on the hosted sites.

The results were extremely negatives. While popular providers seem to invest a considerable effort to prevent criminals from registering an account with them, once a customer is registered most of the providers do absolutely nothing to detect malicious or suspicious activities. Moreover, also abuse notifications are not properly handled in most cases, thus preventing also other users to report malicious activities.

The last paper in this dissertation, proposes the use of a novel high-interaction honeypot to observe the attackers during, and *after*, they exploit a vulnerable web application. The article, published at the Network and Distributed System Security Symposium in 2013, presents a study of over 6,000 attacks – with a particular focus on the clustering and identification of the files uploaded by the attackers to the compromised machines. Based on the collected information, we were able to infer the goal of the attackers and thus understand if they were interested in installing phishing kits or in joining a botnet, in gathering information or in sending Spam. As I will explain more in Chapter 12, I believe this information is very important to design new security mechanism and fully understand how to properly address the security of web applications.

The Role of Web Hosting Providers in Detecting Compromised Websites

Davide Canali
EURECOM, France
canali@eurecom.fr

Davide Balzarotti
EURECOM, France
balzarotti@eurecom.fr

Aurélien Francillon
EURECOM, France
aurelien.francillon@eurecom.fr

ABSTRACT

Compromised websites are often used by attackers to deliver malicious content or to host phishing pages designed to steal private information from their victims. Unfortunately, most of the targeted websites are managed by users with little security background - often unable to detect this kind of threats or to afford an external professional security service.

In this paper we test the ability of web hosting providers to detect compromised websites and react to user complaints. We also test six specialized services that provide security monitoring of web pages for a small fee.

During a period of 30 days, we hosted our own vulnerable websites on 22 shared hosting providers, including 12 of the most popular ones. We repeatedly ran five different attacks against each of them. Our tests included a bot-like infection, a drive-by download, the upload of malicious files, an SQL injection stealing credit card numbers, and a phishing kit for a famous American bank. In addition, we also generated traffic from seemingly valid victims of phishing and drive-by download sites. We show that most of these attacks could have been detected by free network or file analysis tools. After 25 days, if no malicious activity was detected, we started to file abuse complaints to the providers. This allowed us to study the reaction of the web hosting providers to both real and bogus complaints.

The general picture we drew from our study is quite alarming. The vast majority of the providers, or "add-on" security monitoring services, are unable to detect the most simple signs of malicious activity on hosted websites.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive Software (e.g., viruses, worms, Trojan horses), Unauthorized access (e.g., hacking, phreaking); C.4 [**Performance of Systems**]: Measurement techniques

Keywords

Shared web hosting; web security

1. INTRODUCTION

Owning and operating a website has become a quite common activity in many parts of the world, and millions of websites are operated, every day, for both personal and professional use. People do not need anymore to be computer "gurus" in order to be able to install and run a website: a web browser, a credit card with a few dollars' balance, and some basic computer skills are usually enough to start such an activity.

Of all the possible ways to host a website, shared hosting is usually the most economical option. It consists in having a website hosted on a web server where other websites may reside and share the machine's resources. Thanks to its low price, shared hosting has become the solution of choice for hosting the majority of personal and small business websites all over the world.

Being so common, however, shared hosting websites have also high chances of being targets of web attacks, and become means for criminals to spread malware or host phishing scams. In addition, such websites are often operated by users with little or no security background, who are unlikely to be able to detect attacks or to afford professional security monitoring services.

Our work focuses on shared web hosting services, and presents a study on what shared hosting providers do in order to help their customers in detecting when their websites have been compromised. We believe this is an important commitment, given the fact that shared hosting customers are the most vulnerable to web attacks [9]. Furthermore, even a security-aware shared hosting customer would never be able to fully protect and monitor his or her account without the provider's cooperation. In fact, in a shared hosting configuration, the user has few privileges on the machine and she is not allowed to to run or install any monitoring or IDS application, nor to customize the machine's web server, its firewall, or security settings. Thus, in order to protect his or her website, a user has to fully rely on the security measures employed by the hosting provider.

In our study, we also tested the providers' reactions to abuse complaints, and the attack detection capabilities of six specialized services providing security monitoring of websites for a small fee.

In a recent survey [4], Commtouch and the StopBadware organization reported the results of a questionnaire in which 600 owners of compromised websites have been asked some questions about the attacks that targeted their websites. From this study, it emerged that, among the surveyed users, 49% of them were made aware of the compromise by a browser warning, while in fewer cases they were notified by their hosting provider (7%) or by a security organization (10%). Also, 14% of the users who took the survey said their hosting provider removed the malicious content from their website after the infection. At the end, only 12% of the customers were satisfied from the way their hosting provider handled the situation, while 28% of users who took the survey were considering to move to a new provider because of this experience.

Inspired by the StopBadware report, we decided to systematically analyze, on a wider scale and in an automated way, how web hosting companies behave with regard to the detection of compromised websites, what their reactions are in case of abuse com-

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

plaints, and how they proceed to inform a customer about his website being compromised.

To our knowledge, this is the first work studying, on a worldwide scale, the quality and reliability of security monitoring activities performed by web hosting providers to detect compromised customer websites. Unfortunately, the general picture we drew from our results is quite alarming: the vast majority of providers and "add-on" security monitoring services are unable to detect the most simple signs of malicious activity on hosted websites. It is important to note that we do not want to blame such providers for not protecting their customers, since this service is often not part of the contract for which users are paying for. However, we believe it would be in the interest of the providers and of the general public to implement simple detection mechanisms to promptly identify when a website has been compromised and it is used to perform malicious activities.

Section 2 of the paper describes the setup and deployment of the test cases we employed to carry out our study; Section 3 reports the results of our experiments, as well as some insights on how hosting providers act with regard to preventing abusive uses of their services and web attacks against their customers' websites. Section 4 explores the related work in this field. Section 5, finally, summarizes the main findings of our work, and concludes the study providing ideas for future improvements in this area of research.

2. SETUP AND DEPLOYMENT

For our study, we selected a total of 22 hosting providers, chosen among the world's top providers in 2011 and 2012 (we will refer to these as *global-1* to *global-12*), and among other regional providers operating in different countries (referred to as *regional-1* to *regional-10*). We selected the *global* providers by picking the ones appearing most frequently on lists of top shared hosting providers published on web hosting-related websites, e.g., `tophosts.com`, `webhosting.info`, and `webhostingreviews.com`. The *regional* providers were instead chosen from the "Country-wise Top hosts" list published by the `webhosting.info` website [19], with the aim of having an approximately uniform geographical distribution over every area of the world. Our final list included providers in the US, Europe, India, Russia, Algeria, Hong Kong, Argentina and Indonesia.

For our study, we limited our choice to providers that allowed international registrations, as our hosting accounts were registered using real personal data of people belonging to our research group. In fact, we noticed that some providers, probably because of regulations in their country, limit the possibility of registering a web hosting service only to national customers. This is especially true for countries such as China, Brazil, and Vietnam, whose providers often require a national ID card number upon registration.

Also, our choice was limited to providers offering shared hosting services as part of their products, allowing to host at least one domain name per account, supporting the PHP programming language, and the FTP transfer protocol.

2.1 Test Cases

We conducted our study by registering five shared hosting accounts for each of the 22 web hosting providers. Each one of the five accounts was targeting a particular class of threat, chosen among the most common types of web attacks that could be easily detected by hosting providers.

Four out of the five test cases we deployed are based on a static snapshot of a website running OsCommerce v.2.2. The application was modified so that the PHP pages always returned a static version of the site, without the need of installing a backend database. Each snapshot was modified by hand in order to include the ad-hoc code required for our experiments, and to diversify the content, the appearance, and the images shown in each page.

Our test files were deployed in the `/osco` subdirectory of every hosting account we registered, while the home page of each domain showed only an empty page with the message "Coming soon...". We did not create any link to the `/osco` subdirectory, and we excluded the possibility for web spiders to visit our test case websites by denying any robot access using the `robots.txt` file. This was done in order to avoid external visits to our test case websites, which could have interfered with our tests.

Intentionally installing and exploiting vulnerable web applications on shared hosting accounts may raise some ethical and legal concerns. For this reason, we carefully designed our tests to resemble real compromised websites - being at the same time completely harmless for both the provider and other Internet users. For example, we modified the application code to mimic an existing vulnerability but, compared to their real counterparts, our code was executed only when an additional POST parameter contained a password that we hardcoded in the application, thus allowing only us to exploit the bug.

2.1.1 SQL Injection and Data Exfiltration (SQLi)

The first test case aimed at detecting whether web hosting providers detect or block SQL injection and data exfiltration attacks against their customers' websites. The test consisted in deploying the static snapshot of OsCommerce including a page that mimics the SQL injection vulnerabilty presented in CVE-2005-4677.

Setup - The `product_info.php` page was modified to recognize our SQL injection attempts and respond by returning a list of randomly generated credit card numbers along with personal details of fictious people (name, address, email, and MD5 password hash). In order to pass the Luhn test, fake credit card numbers were generated using an online credit card test number generator [6].

Attack - The attack for this test case was run every hour, and consisted of a script mimicking a real SQL injection attack: first, the fake vulnerable page (`product_info.php`) was visited, then a sequence of GET requests were sent to the same page adding different payloads to the `products_id` GET parameter. The first request simulated somebody testing for the presence of SQL injection vulnerabilities by setting `products_id=99'`; then, five attack requests were issued to the same page by setting the following payloads for the vulnerable parameter:

```
1 : 99' UNION SELECT null,CONCAT(first_name, ...
      customers_password),1,CONCAT(cc_type, ...
      cc_expiration) FROM customers LIMIT 1,1/*
2 : 99' UNION ALL SELECT null,CONCAT(first_name, ...
      customers_password),1,CONCAT(cc_type, ...
      cc_expiration) FROM customers LIMIT 2,1/*
3 : 99' UNION S/**/ELECT null,CONCAT(first_name, ...
      customers_password),1,CONCAT(cc_type, ...
      cc_expiration) FROM customers LIMIT 3,1/*
4 : 99' UNION S/**/ELECT null,CONCAT(first_name, ...
      customers_password),1,CONCAT(cc_type, ...
      cc_expiration) FR/**/OM customers LIMIT 4,1/*
5 : 99' UNION S/**/ELECT null,CO/**/NCAT(first_name, ...
      customers_password),1,CO/**/NCAT(cc_type, ...
      cc_expiration) FR/**/OM customers LI/**/MIT 5,1/*
```

Listing 1: Payloads of fake SQL Injection requests

The purpose of these payloads was to detect whether hosting providers employ any blacklist-based approach to detect SQL injection attempts on their customers' websites. Requests in lines 1 and 2 would fail in case the providers employ simple blacklisting rules (blocking any UNION SELECT and UNION ALL SELECT) in URLs. The last three requests would fail only if provid-

ers deploy more complex rules that are able to blacklist typical SQL words even in case they are stuffed with comments, or if words like FROM, CONCAT and LIMIT are blacklisted as well.

2.1.2 Remote File Upload (Web Shell) and Code Injection Using Web Shell (SH)

The goal of this test is to understand whether providers detect the upload and usage of a standard PHP shell, automatic file modifications on the customer's account, or the presence of malicious code on the home page of the website. In the test, a fake web shell is uploaded to the hosting account, and fake commands are issued to it, resulting in some drive-by-download code being added to the home page of the e-commerce web application.

Setup - This test uses the base static snapshot of the OsCommerce v.2.2 web application, and simulates a Remote File Upload vulnerability in the file `admin/categories.php/login.php`, as the one described in [13]. Our fake attack was designed to upload a modified version of the popular c99 PHP shell (one of the most common web shells on the web), that has no harmful effects other than the ability to inject custom code in the home page of the e-commerce web application. Also in this case, the custom code injection is enabled only when certain hidden parameters are specified along with the request of the c99 shell, thus allowing only us to trigger the injection. The content to be injected in OsCommerce's index page is a snippet of a real malicious code launching a drive-by download attack, that has been disabled by wrapping it into an `if` statement with a complex condition that is always False. We submitted the index page with the injected content to the VirusTotal online virus scanning service [1], and it was detected as malicious by 13 anti virus engines.

Attack - The test case for this attack was run every hour, and consisted in a script performing the upload of the web shell, followed by a number of commands issued on the shell. The shell file, called *c99.php* as the original shell, in order to be easily identifiable from the web server logs, was uploaded to the vulnerable URL by specifying the secret parameter enabling the upload. If the upload was successful, five commands were issued to the *c99.php*, picked randomly from a list of GET and POST requests containing both Unix commands and file names, so to make the requests seem like the result of someone trying to manually explore the contents of the server. The requests simulated actions such as trying to read files (e.g., `/etc/passwd`) and execute unix commands (`who`, `uptime`, `uname`, `ls`, `ps`). Our intuition was that hosting providers would probably be alerted by requests containing some of these filenames or commands. Finally, the test used the PHP shell to inject a plaintext version of the malicious code into the home page of OsCommerce.

2.1.3 Remote File Upload of a Phishing Kit (Phish)

Similarly to the previous test, this test uses a file upload vulnerability in the OsCommerce application to upload a phishing kit to the web server. The phishing kit consists of an archive containing a static snapshot of a real Bank of America scam. The test aims at detecting whether hosting providers are able to detect the presence of a phishing kit on the customer's account. The phishing kit was installed inside a directory named `/bankofamerica.com`, thus allowing to detect any visit to the scam pages by simply looking at the requested URLs.

Setup - This copy of the application is configured with the same Remote File Upload vulnerability explained for the previous test. However, the vulnerable path for this test is `admin/banner_manager.php/login.php`. Whenever this script is issued an upload request for a file with `tar` extension, it uploads the archive and automatically unpacks its contents to the upload directory, thus allowing for an automatic installation of the phishing kit. The phishing kit we deployed is an exact copy of a real Bank of America phishing kit found in the wild, modified to remove the back end code (thus making it unable to store and send any user information).

Attack - This attack was split in two phases, which we refer to as *attacker* and *victim*. The *attacker* phase, run every 6 hours, consisted in triggering the remote file upload vulnerability and uploading the phishing kit. The *victim* phase of the attack was run four times per hour, and consisted in a script that simulated a victim falling prey of the scam. In order to look realistic, the victim requests were disguised as coming from a range of different valid User-Agent strings used by web browsers on Windows operating systems. Every simulated victim visit comprised a sequence of GET and POST requests containing the form parameters required by the phishing pages. At each victim visit, the data sent in the requests was randomly picked among a set of fake personal details we created by hand, containing names, addresses, passwords and credit card numbers of fictitious people.

2.1.4 Suspicious Network Activity: IRC Bot (Bot)

This test aims at understanding whether providers employ any network rules to detect suspicious connection attempts to possibly malicious services. For this study, we opted to deploy to our accounts a script simulating an IRC bot. The reason for this choice is that IRC bots are probably one of the most common and easily detectable bots, because IRC connections are very often made to the standard IRC port (6667) using cleartext communication.

Setup - This test uses our basic OsCommerce installation with no modifications. The executable bot client was deployed to the hosting account via FTP, thus simulating an attack in which the attacker has stolen the customer's web hosting credentials. The files to be uploaded are two IRC client binaries written in C (one compiled for 32-bit architectures, and one for 64-bit ones), and a PHP script that executes the right binary depending on the underlying OS type, and outputs its results. The IRC client, once launched, disguises itself as "syslogd" and tries to connect to a machine hosted on our premises that runs a fake IRC server on the standard IRC port. If the connection succeeds, the client and server exchange a few messages resembling real IRC commands (such as `NICK xxx, USER xxx, JOIN #channel`) and the client reports some information about the infected machine (host name, OS type, kernel version); at last, the client closes the connection.

Attack - The test case for Bot was run every hour, and started with opening a FTP connection and uploading the two binaries and the PHP file in a new directory created in the web site's root folder. If the upload succeeded, an HTTP request was issued to the PHP file launching the IRC client. The output of this request allowed us to determine whether the hosting provider was blocking the use of possibly dangerous PHP functions (IRC client execution denied - *system()* function disabled), blocking outgoing connections to certain ports (binary executed, but connection attempt failed), or allowing everything (successful connection to the server). In order to make the upload of the IRC botnet files appear even more suspicious, the FTP upload was executed using IP addresses from several different countries.

2.1.5 Known Malicious Files (AV)

This test aimed at understanding whether providers perform any scans of their disks with off-the-shelf anti virus software. The test simply consisted in deploying, via FTP, two common known malicious files to the customer's hosting account.

Test #	SQLi	SH	Phish	Bot	AV
Blocked by ModSecurity base rule set	○	○	○	○	-
Blocked by ModSecurity OWASP rule set	●	◐	○	○	-
High severity IDS alerts	5	2	2	0	0
Detectable by antiviruses	no	yes	no	no	yes

Table 1: Attacks detection using freely available state-of-the-art security scanning tools. Legend:
○ no; ◐ in part; ● yes (full); - not applicable

Setup - Websites hosting this test used a simpler structure than the previous tests, and consisted in a single static HTML page containing random sentences in English and a few images. As in test Bot, we chose to use FTP to upload the malicious files to the account, to simulate a case in which the attacker has knowledge of the customer's account credentials. The two malicious files were *c99.php*, a real c99 PHP web shell, detected on VirusTotal with a score of 25/43 (25 antivirus engines detecting it, out of 43 it was tested against), and *sb.exe*, a copy of the 2011 Ramnit worm, detected by 36 out of 42 antivirus products according to VirusTotal. In order to make sure the malicious files were not reachable by any web visitor, but only available to people having internal access to the server, they were uploaded to a directory protected by means of *.htaccess* (denying the listing of its files) and *.htpasswd* (requiring a password to access its files from the web).

Attack - The attack itself consisted simply in connecting to the hosting account's web space via FTP and uploading every time (deleting and re-uploading if already present) the protected directory and the two malicious files. Also in this case, FTP connections were issued from IP addresses in different countries.

2.2 Attack Detection Using State-of-the-Art Tools

Before deploying the tests to the shared hosting accounts, we made sure they could be detected using common state-of-the art tools, that can be easily employed by any hosting provider. In order to do so, we executed our tests against an installation of the SecurityOnion Linux distribution, which includes a preconfigured set of open source tools for monitoring suspicious network and system activity (Bro IDS, Snort, Sguil). The installation of this distribution was then equipped with the Apache2 web server and the ModSecurity plugin, along with its base recommended rule set.

We also installed the OWASP ModSecurity "Core Rule Set", a set of common security rules for Apache ModSecurity that is maintained by the OWASP foundation [14]. These are free certified rule sets providing generic protection from unknown vulnerabilities often found in web applications. We installed version 2.2.5 of the rule set on our test machine, and disabled some rule sets (base rules number 21, 23, 30) for being too generic and generating too many false alarms. We finally ran each of the five test cases toggling on and off the OWASP ModSecurity rules.

Table 1 summarizes what we were able to detect or block using this setup, during the execution of each test. Four out of the five attacks would have been blocked or detected by employing free network and host monitoring solutions like the ones mentioned above, and the remaining attack could have been easily detected by setting up a simple connection filtering rule in the firewall.

2.2.1 SQLi

The attacks of test SQLi, when run using the basic installation of ModSecurity, succeed, but generate a series of five different high severity alerts about possible web server SQL injection attempts. When the OWASP rule set is enabled, however, all the five SQL injection attempts on which the attack is built fail.

2.2.2 SH

The SH test, executed against a webserver with the basic ModSecurity rules, successfully uploads the c99 shell and injects the drive-by code in *index.php*. However, two high severity events are raised by the IDS, one of which notifying a remote code execution on OsCommerce v.2.2 (triggered by our attack to upload of the web shell). If the OWASP rules are enabled, the remote file upload succeeds but most of the commands issued to the web shell fail and raise critical alert messages, notifying the possibility of a web file injection attack. The index file modification, finally, fails and raises a message notifying the detection of multiple URL encodings in the request, as a possible sign of protocol evasion. Finally, it has to be noted that, although we removed all the existing functionalities from the original web shell, our *c99.php* contains some original PHP code to display images and UI elements, plus our custom drive-by injection code. As such, it would still be detected during a virus scan by approximately 17% of the antivirus engines on the market (its VirusTotal score is 7/42). The *index.php* containing the injected content would instead be detected by almost 30% of the antiviruses, having a VirusTotal detection score of 13/44.

2.2.3 Phish

This attack succeeded but raised two high severity events: potential remote code execution in OsCommerce v.2.2, and presence of PHP tags in the HTTP post (detected on the tar file containing the phishing kit). On the victim's side, no HTTP request is blocked when uploading personal information to the scam pages. A possible solution to stop, or at least raise alerts on the victim's requests, however, could be deploying a simple IDS/IPS rule that detects the submission of parameters containing cleartext personal details, such as credit card numbers and cvv2 codes.

2.2.4 Bot

The Bot test case was undetected by the basic and OWASP ModSecurity rule sets, as it was run via FTP. In our tests, the connection succeeded and the bot and fake IRC server completed their message exchange. A normal firewall rule blocking outgoing connections to port 6667 (IRC) would have, however, blocked the attack.

2.2.5 AV

The malware upload test (AV) was undetected by our test deployment, because no HTTP traffic was generated and no network antivirus was used. However, as explained in Section 2.1.5, we recall that the uploaded *c99.php* and *sb.exe* are common malicious files detected by VirusTotal with a detection scores of 25/42 and 36/42, respectively. Therefore, the vast majority of off-the-shelf antiviruses would have detected them during a scan of the website's root directory.

2.3 Test Scheduling and Provider Solicitation

All attacks were run without interruption on every hosting account for the first 25 days of testing. As explained in the previous section, each attack was repeated multiple times per day in order to generate more alerts and increase the probability of being detected.

If the hosting provider did not detect any suspicious activity during this time frame, the tests entered a second phase, during

which we solicited the provider to detect our attacks and take action against them. This solicitation took place as an abuse notification email for the Phish and AV tests, in which we reported the presence of malicious files on the web application. We also generated "fake" abuse notifications to study the reaction of the providers to bogus complains.

This allowed us to understand: 1) how quickly providers respond to abuse notifications, if they ever do, 2) if they actually verify the presence of malicious content or activity on the account before taking any action, and 3) what kind of actions they take in order to stop the abuse. Abuse notifications were sent to providers by email, using real (authenticated) email addresses registered on 3rd party domains, to make them look as realistic notifications from random web users.

2.3.1 Real Abuse Notifications

Starting the 25th day of testing, we started sending one abuse complaint per day to each provider on which tests Phish and AV had not been previously detected. We stopped the notification process and the real attacks on the account either when the 30 day testing period elapsed, or after the provider responded to the notification. The notification email explained that an email had been received, with a link pointing to content hosted on the provider's premises. The link pointed to the phishing kit's index page for Phish, and to the *sb.exe* file for AV test. In addition, the email mentioned that the user's antivirus raised an alert when trying to visit the URL, and suggested the web provider to check the contents of the account.

2.3.2 Fake Abuse Notifications

Apart from real abuse notifications, we also sent emails in which we complained for perfectly clean websites. To perform this test, we cleaned and re-used the account used for the SQLi and Bot tests. The website contents were replaced by a single static HTML page containing one JPG picture and a long list of news extracted from the RSS feeds of popular international news websites. Starting on the 25th day, we sent to every provider an email per day, where the user complained about the presence of offending or malicious content on these accounts. Since at the time these emails were sent the websites were absolutely clean, these fake notifications allowed us to understand whether providers actually check the veracity of the complaints they receive before taking any action. The first complaint email was from a user pretending that the website's content was offending his religious views, and kindly asking to stop the website owner from spreading such disrespectful messages. In the second scenario, the notification email was from a user claiming to have received an email with a link to the website in question. The user explained that his browser denied access to the URL, and that at a closer look the website looked like hosting a phishing scam. Also in this case, the account hosting the reported webpage was absolutely clean, hosting only the benign static HTML home page.

One may argue that, in case of these fake notifications, the provider could react by suspending or shutting down the user account by having a look at the logs of the machine on which the account was setup, and noticing past malicious activity, even though, at notification time, the website was clean. We did our best in order to avoid this from happening, by deploying our tests for fake notifications on accounts that hosted the SQLi and Bot tests. These tests could not be considered malicious (no malware nor phishing files were ever uploaded) but the mere evidence that the website was under attack. Moreover, attacks for these tests could only have been detected at a network level, since no trace was left on the disk.

3. EVALUATION

During our experiments, we evaluated the security measures put in place by web hosting providers to detect malicious activities, compromised websites, and prevent abuse of their services. We group our findings in three categories: account verification upon signup (3.1), compromise prevention and detection capabilities on live websites (3.2.2), and responses to abuse notifications (3.3).

3.1 Sign-up Restrictions and Security Measures

Even though our work was not meant to test the anti-abuse signup policies of web hosting providers, we report here some results that may contribute in understanding how much effort providers put in preventing services subscription by malicious users.

Several providers try to discourage abusers by asking to verify the information entered during the signup phase, either by calling the customers on the phone, or by requiring a scanned copy of their documents (such as government issued ID, credit card used for the purchase). Some providers also use 3rd party fraud protection services, that block purchases based on a set of heuristics. For example, we observed several cases in which the providers correlated the geographic location of the customer, the billing information, and the IP address used for the purchase.

The shared hosting accounts we used for our study were all registered using real personal information of people working in our group, and the billing information of our research institute. The sign-up process was carried out from several IP addresses, using either credit card or PayPal payments.

Anti-abuse signup policies vary widely between hosting providers. Top *global* hosting providers are more cautious with regard to signup, often blocking attempts - e.g., blocking multiple registrations from the same billing address and credit card number, verifying the customer's personal information by verification phone calls or ID and credit card checks. *Regional* providers seem to be more permissive, probably because they have less incentives in making their signup process more difficult, which could make them lose potential customers.

Among the twelve *global* providers, seven of them required us to verify our account information for at least one of the accounts we registered with them. In order to verify our account information, all these companies required a scanned version or photocopy of a government issued photo identification card (such as passport or driver's license) and the front and back of the credit card used at signup (without showing the first 12 numbers and the cvv2 code). Only one out of these seven companies claimed, on its website, to manually verify every customer's signup before allowing the purchase of its web hosting services. Indeed, this was the only provider that verified every account we registered with them.

Regional providers, instead, do not seem to be as cautious during the account signup phase. Only one out of ten blocked an account creation because of a mismatch between our billing address and the geolocation of the IP address used for registration.

Finally, three of the *regional* providers we tested had a very simple signup process, where users could register an account in one click, by filling all the required personal and payment information in one page. These providers never asked us to verify our information upon registration, and thus could possibly be a good choice for criminals wanting to perform abusive subscriptions.

Signup verification requests are either sent during registration or after a successful account registration and activation. While requiring an account verification upon signup can be effective in preventing malicious registrations, it can also make the hosting provider lose potential good customers that may not have time or patience

Provider	Verification time		
	Before payment	Before activation	After activation
global-2	25%	-	50%
global-3	25%	-	25%
global-4	33%	-	-
global-5	40%	-	-
global-6	-	33%	-
global-7	100%	-	-
global-8	50%	25%	-
regional-2	33%	-	-

Table 2: Account verification times. Values represent the percentage of verification requests on the number of accounts we registered for each provider. "Before payment" means during the registration process. "Before activation" means once the client's billing account is created, but the hosting service is not yet active. "After activation" indicates when the hosting account is active and a website has possibly already been installed.

to provide all the required information. On the other hand, requiring an account verification once the service has been purchased and set up has the drawback of temporarily suspending an account on which a website has already possibly been deployed, thus causing a service outage for a benign customer. During our experiments we encountered both situations. Table 2 shows the percentage of verification requests on the number of accounts we registered for each provider, grouped by the time at which the request was issued. Only providers that requested at least one account verification are listed.

The table shows that, in general, most of the anti-abuse systems send alerts and block a registration attempt during the customer's signup phase. This typically happens when the user enters his or her credit card details and tries to complete the hosting purchase. Others, instead, let the client sign up for the service and receive its management panel credentials, but lock the web hosting service activation until a copy of the customer's document is received by the support department. Two web hosting providers (*global-2,3*) sent verification requests when the web hosting account was already active and the customer's website deployed. This caused a temporary service disruption for the affected accounts, making their websites unavailable for several hours. Certain providers, finally, issued verification requests at different times, probably depending on the kind of alert they received from their abuse prevention system (*global-2,3,8*).

3.2 Attack and Compromise Detection

During the first phase of our experiments, we deployed our five test suites on every hosting provider and recorded whether the hosting provider took some action or contacted us to notify that malicious activity was observed on our account. As explained in Section 2.3, if no malicious activity was detected on the account during the first 25 days, we started sending abuse notifications to the hosting provider, in order to stimulate a response. The results of this second phase are summarized in Section 3.3.

To make our fake attacks look realistic, our test cases were run automatically at certain time intervals (as explained in Section 2.1), and the attacks were executed from different IP addresses belonging to several different countries. Also, in order to avoid having only "artificial" malicious requests in the web server logs of our accounts, we generated some background traffic simulating real visits to our websites. This was accomplished by developing a simple traffic generator tool, that visited every account we deployed every 10 minutes, and randomly followed links on every website up to a depth of 30. In the general case, this meant following an average of 13 links on every website, thus generating a bit less than two thousand hits per day on every active account. The machine used for traffic generation was not used for other experiments and used a different set of public IP addresses than the ones we used to run the attacks.

3.2.1 Attack Prevention

Even though our study focuses on the ability of the providers to detect compromised websites, during our experiments some of our attacks were blocked and were therefore ineffective. In some of these cases, we proceeded by manually compromising the account. For example, whenever a provider denied the possibility of running test SH, we manually uploaded the drive-by download code to the account to continue the experiment. This allowed us to test whether the provider was able to detect the menace by scanning the customer's account. For the phishing attack (Phish), since it had to be detected on a network level, we did not take such measure and thus no manual upload was performed on accounts of providers blocking the remote file upload.

Table 3 reports, for each test and provider, whether the web hosting company took any measure to prevent the attack. Such measures depend on the test case, and ranged from employing URL blacklists to blocking outgoing connections or process executions.

URL blacklisting.

Some providers employ URL blacklists in order to prevent SQL injection attempts (test SQLi) and remote file uploads (SH, Phish).

However, as shown in column *SQLi* of Table 3, none of the providers we tested were able to fully prevent our SQL injection attacks. This is probably due to the adoption of simple keyword-based blacklisting rules, that can be easily bypassed by introducing SQL comments in the middle of blacklisted keywords (such as using "SE/**/LECT" instead of "SELECT", as explained in Section 2.1.1). Two providers (*global-1, regional-2*) blocked the first four requests of our attacks, and other five providers were able to block only the first two. The remaining did not adopt any SQL-injection protection mechanism at all.

Regarding tests SH and Phish, some providers were able to prevent the attack by employing URL blacklists probably containing specific rules for the detection of common vulnerabilities on web applications, such as the ones we employed for the tests presented in Section 2.2, provided by the OWASP foundation. Regarding SH, Table 3 shows that some providers were able to only partially prevent the attack. These providers did not block the file upload itself, but employed blacklisting rules to block some requests to the web shell (these requests contained common file names, e.g., */etc/-passwd*, or common parameter sequences such as *.php?act=cmd*).

Connection and OS-level filtering.

The attack files for test Bot were first uploaded to the customer's account via FTP, then the fake IRC client was executed issuing a HTTP request to a PHP file launching an executable file using the *system()* PHP function. A total of 18 providers were able to fully stop the attack: of these, 50% did so by disabling the *system()* function in PHP, while the remaining half firewalled outgoing connections to the IRC port.

When the attack was prevented, we were expecting some form of notification regarding the suspicious activity. After all, it is not

normal that a shared hosting user has a disguised process that tries to connect to an IRC server every hour for one month.

Two hosting providers allowed the attack only at certain periods in time (*global-2* and *global-6*). This may due to temporary misconfigurations on their networks or to automatic account migrations over different machines with different configurations (for example, the account running test Bot on provider *global-6* connected to our fake IRC server from eight different hosts during the 25 days testing period).

No prevention results are shown for test AV, as this test did not run any attack and no filtering was done on the upload of malicious files via FTP.

As a final remark, we noticed that, for some tests, some providers had exactly the same behavior. This is the case, for example, of *global-1* and *regional-2* and *global-8* and *regional-3*. We thus believe that these providers employ the same protection mechanisms and web server security configurations for their shared web hosting solutions. These services are probably provided by third party companies as part of common off-the-shelf security solutions.

3.2.2 Compromise Detection

Sadly, all but one of the providers we tested did not notify their clients when their websites were compromised and were used to perpetrate obvious malicious activities.

The only hosting provider that reacted to one of the attacks was *global-4*, but that reaction happened 17 days after the beginning of test AV. The provider properly notified the presence of a malicious file (the `c99` shell) on the user's web hosting account. In addition, the provider warned the user that a service suspension would occur if no reply to the alert was received by the customer support service within 24 hours. However, the message was not mentioning the presence of the other malicious file on the account, namely, `sb.exe`. This suggests that the alert was an automated message resulting from a virus scan of the account, and that no human operator actually checked the contents of the directory in which the two malicious files were stored.

We were quite surprised by our findings, as we were expecting to have at least a few of our scenarios detected by the vast majority of web hosting providers. It emerges that, on shared hosting servers, even the most basic virus scan is not as common as one could expect. From our measurements, we are not able to tell if the hosting providers run antivirus systems on their shared hosting servers. However, if they do, they are either using outdated signature definitions, or the frequency at which they perform the scans is less than once a month.

3.3 Solicitation Reactions

As explained in detail in Section 2.3, whenever one of our test suites was not detected by the hosting provider for 25 consecutive days, we started sending daily abuse notification emails to the provider's abuse contact. The purpose of sending these messages was to understand whether web hosting providers respond and react to abuse notifications (e.g., by suspending a compromised account or notifying the customer of his or her website being compromised). To complete our test, we also sent fake abuse notifications for perfectly clean webpages, with the aim of understanding whether any providers take action without first verifying the claims. This would pose a serious menace, as it would be a very easy and effective way to conduct a Denial of Service attack against websites of other users. The following paragraphs are meant to give some insights and details on what is presented in the "Solicitation Reaction" section of Table 3.

3.3.1 Abuse Notifications

Unfortunately, 50% of both the *global* and *regional* web hosting providers never replied to any of the real abuse notifications we sent. This percentage is quite alarming, and means that if a website is hosting malicious content (such as phishing or malware), no action will be taken to stop it from spreading and reaching its victims. Moreover, phishing attacks and malware files used in dropzones usually have a short lifetime, and, as such, even a late response to a malware or phishing abuse notification would have little or no effect on the general outcome of the attack.

Seven out of the eleven providers that replied to our complaints replied either the same day or the day after the notification was sent. This is a good indicator, meaning that these companies probably care about web abuses and are able to handle these issues in a timely manner. The only provider that replied later than 5 days after the notification was *regional-5*, with an average response time of 16 days. After such a long delay any action would be basically useless, as the website may have completely changed in the meantime.

There were a variety of reactions to our abuse complaints. The most common approach was to temporary suspend the customer's account, with five companies performing at least one suspension as result of a malware or phishing abuse complaint. We consider this action a reasonable response to the abuse, causing a temporary disruption of the services the client is paying for, but blocking the immediate threat. Other providers responded to the notifications by cleaning up the account, removing the suspicious files (4 providers - note that this action seems to be more common among *regional* providers), or by forwarding the abuse notification to the customer (1 case). We considered such responses, in general, to be appropriate to stop the menaces from spreading, and at the same time avoiding to impact too much the user's services.

Provider *global-12* reacted without notifying the website's owner: in the case of AV, the account was terminated, while in the case of phishing (Phish), the directory containing the fake phishing kit was removed. Also in the case of provider *regional-6*, actions were taken without notifying the user, with the exception that, in this case, the reactions to the abuse notifications consisted in deleting all the files (including the clean ones) of the user's websites!

Controversial responses to our abuse notifications were those from providers that sent ultimatums to the user (marked with U, in the table), warning him that offending content had been found on his website, and that if no cleanup was performed within a few hours, the account would have been suspended. This was controversial because, as in the case of provider *global-6*, even though we did not take any action to respond to the provider ultimatum, the fake phishing pages were still present on our account after several days. This means that the provider did not keep to its commitment.

Finally, a few responses were partially or fully unsatisfying. The *regional-3* provider replied to the malware abuse complaint probably after scanning the customer's account using an antivirus. The reply stated that a c99 PHP shell had been found on the account, and asked the notifier if he wanted them to remove it. The malicious executable was not mentioned at all and no further action was taken, thus leaving both malicious files on the account. The case of providers *global-2, global-3* and *global-5* is quite particular. While experiments were in progress on most of the providers, and once our tests Phish and AV reached their 25th day on *global-2* and *global-5*, notifications were sent to the two providers. First, provider *global-5* replied by terminating the account (disabling both the billing and the hosting account) and giving the customer 15 days to reply and to recover his files. We replied, asking to reenable the account for recovering our files, but in the meanwhile another abuse response was received from provider *global-2*, ter-

Provider	Account verification	Attack Prevention/Detection (days)					Solicitation Reaction		
		SQLi	SH	Phish	Bot	AV	Abuse complaint	Fake abuse complaint	Avg. reply delay (days)
global-1	○	◐/ ○	●/ ○	●/ -	●/ ○	- / ○	○ N	● N	-
global-2	◐	○/ ○	○/ ○	○/ ○	◐/ ○	- / ○	○ T	- -	1
global-3	◐	- / -	○/ ○	○/ ○	●/ ○	- / ○	○ N/T	- -	-
global-4	◐	○/ ○	○/ ○	○/ ○	●/ ○	- / ●(17)	● S	◐ U	0
global-5	◐	- / -	○/ ○	○/ ○	●/ ○	- / ○	○ T	- -	0
global-6	◐	○/ ○	○/ ○	○/ ○	◐/ ○	- / ○	○ U	● O	2
global-7	●	◐/ ○	○/ ○	○/ ○	●/ ○	- / ○	○ N	● N	-
global-8	◐	◐/ ○	○/ ○	●/ -	●/ ○	- / ○	○ N	● N	-
global-9	○	○/ ○	●/ ○	●/ -	●/ ○	- / ○	○ N	● N	-
global-10	○	○/ ○	●/ ○	●/ -	●/ ○	- / ○	◐ S	● N	4
global-11	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	○ N	● N	-
global-12	○	○/ ○	○/ ○	○/ ○	○/ ○	- / ○	◐ T,C	● O	0
regional-1	○	◐/ ○	◐/ ○	○/ ○	●/ ○	- / ○	● S,C	○ S	0
regional-2	◐	◐/ ○	●/ ○	●/ -	●/ ○	- / ○	○ N	● N	-
regional-3	○	◐/ ○	○/ ○	●/ -	●/ ○	- / ○	◐ O,C	● O	0
regional-4	○	○/ ○	○/ ○	○/ ○	○/ ○	- / ○	○ N	● N	-
regional-5	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	◐ S	● O	16
regional-6	○	◐/ ○	◐/ ○	○/ ○	●/ ○	- / ○	◐ C	○ C	1
regional-7	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	○ N	◐ U	5
regional-8	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	● S,F	● O	1
regional-9	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	○ N	● N	-
regional-10	○	○/ ○	○/ ○	○/ ○	●/ ○	- / ○	○ N	○ P	0

Table 3: The results of our study. Legend:

-	**not applicable**	**N**	**no reply**	**P**	**forced password reset**
○	**no / not satisfying**	**S**	**account suspension**	**C**	**cleanup or file removal**
◐	**in part / partly satisfying**	**T**	**account termination**	**U**	**ultimatum to the user**
●	**yes (full) / satisfying**	**F**	**complaint email forwarded**	**O**	**reply but no action**

minating our account. Starting that moment, within a few hours, all the accounts we had registered on providers *global-2, global-3* and *global-5* were terminated without any explanation, even when we tried to contact the companies to ask details about the reasons of our accounts' termination. The only response we were able to get was: *"Due to certain items contained in the account information, this account was flagged for fraud. For security reasons, this flag caused the system to delete your account. At this time we ask you to seek out a new hosting company."*

Either the three companies used the same support service, provided by a third party, or they shared information between them. Indeed, the termination notifications for all the accounts on the three providers were sent by the same support representatives, and contained exactly the same text (only the email signature changed, containing the email and postal address of the appropriate company). For this reason, we expect the support center for these companies was able to link our accounts' personal information and understand they were all registered by the same group of individuals. Thus, having received complaints for two of the accounts, all the other accounts that could have been reasonably linked to them were terminated as well.

When this happened, some test cases had not been deployed yet on these providers (SQLi on *global-3, global-5*) and others had not yet reached their 25th day of execution (Phish on all, and SQLi on *global-2*), thus no fake abuse notifications were sent for them. This explains why Table 3 has missing data for such providers in columns "SQLi" and "Fake abuse complaint". This is also why in the "Abuse complaint" cell for provider *global-3*, we listed N/T: no abuse notification response was received (N), but a termination occurred anyway (T) for other reasons.

Finally, for provider *global-9*, we were not able to properly contact its abuse department: out of the four different abuse notifications we sent to its abuse email address, only the last two received an automated reply, saying that in order to report an abuse, it is necessary to click on the help link on the web hosting provider's home page and follow a series of steps (at the time we received these responses, the five-days testing period was already expired). We flagged this case as "no reply" because, although we tried to submit the complaints following the company's advice, the user interface adopted by the provider makes it very difficult, even for an experienced user, to find the right way to report a website abuse. Moreover, once a visitor is able to reach the right page for submitting a website abuse notification, he or she is required to register an account before being able to file a complaint.

3.3.2 Fake Abuse Notifications

We expected most web hosting providers to ignore our abuse notifications regarding "offending content" (see 2.3) and to check the website's contents but take no action in case of the fake phishing complaints. In Table 3, we thus marked as "satisfying" also the providers that never replied to our complaints. However, this is not always a good sign, especially when the same provided never responded to the real complaints.

Sadly, some of the reactions we observed were clearly in contrast with our expectations. Both providers marked with "U" believed either our religious complaint (*global-4*) or our phishing

one (*regional-7*), warning the website owner about the possibility for his account to be suspended if the offending content was not removed within a few days. However, contrarily to what was promised, the content of the websites was left untouched and none of these providers took any action to block the user's account after the ultimatum expired.

One provider, *regional-1*, suspended one of our clean accounts on the same day it was notified as hosting a phishing website. *regional-6*, instead, acted as in the case of real abuse complaints: all the pages on the account's web hosting directory were deleted, and the website's home page was replaced by an "under construction" page. This was already bad when associated to a real malicious content, but in case of a bogus complaint it is really an unacceptable behavior. One last provider, then, responded to the fake phishing abuse notification by sending the website owner an email stating that his website has been attacked, and as such a password reset had been forced on the account. Furthermore, the malicious files were disabled (by means of changing their access permissions) and their list was sent to the user: the list contained the benign website home page and the `jpeg` picture included in it. We were not able to figure out how the web hosting provider assumed the static HTML home page and the picture could contain malicious code.

Only four web hosting providers replied to our fake abuse notifications with messages that completely satisfied our expectations. In these cases, marked with "O" in the table, the support representative informed the notifier that upon manual inspection, the website seemed to be clean, and, in case some content seems to be offending somebody's cultural views, the issue has to be resolved in person by contacting the owner of the website. From this analysis it seems that *regional* providers are slightly more likely to perform a manual content inspection on the websites they host (at least 30% of the ones we tested), compared to *global* providers (only two out of twelve).

3.4 Re-Activation Policies

Whenever an hosting account was suspended, providers often provide the customer with the steps to follow in order to have the account re-activated. These steps usually imply changing every password of the account (billing, FTP, database passwords, etc.), writing a letter or an email stating the agreement to the provider's Terms of Service, and removing the malicious files or re-installing a clean copy of the website. Among the companies that suspended our accounts, *global* hosting providers seem to stick to strict legal requirements before allowing customers to have their accounts re-activated after a violation of the terms of service. The two hosting providers that suspended at least one of our accounts required us to send an email (*global-4*) or a scanned letter or fax (*global-10*) to their support department, stating that we have followed all the necessary steps to clean up our account and reset our login credentials, and that in future we will abide by the terms of service of the company. Regional providers appear to be more "informal" with regard to this, as often a simple email replying to the incident notification, explaining that we were running a vulnerable web application or using a weak FTP password, was sufficient to have our account re-activated. Also *regional* providers, however, in their incident notifications, advised the user to follow basic steps to secure his account (password change, website cleanup) before requesting a service re-activation. During our tests on *regional-1*, in one case, a scanned version of the customer's identification card was required in order to re-activate a suspended account.

Finally, in the case of service terminations, the providers just wanted the user to leave their company, replying to service re-activation requests with emails stating in that, given the kind of activity encountered on the account, the company was not willing anymore to provide their service to such customers.

3.5 Security Add-on Services

In our study, we also evaluated the ability of third party "add-on security providers" to detect attacks or abuses on a website. These services can be purchased separately from web hosting accounts*, and associated with a domain or website to monitor. In some cases, the subscriber has even the option to give his FTP/SFTP access credentials to the security service, to allow an in-depth scan of all the files on his or her account (also those that may not be reachable from the web). For our study, we selected four companies offering such security services, chosen among the most common and advertised on the web. We limited our choice to services that are affordable for a personal or small business use ($30/month max subscription price). We did so in order to test services that are in line with the level of web hosting we were testing. Indeed, it would not be reasonable to pay hundreds of dollars per month, or more, to protect a $10/month hosting plan.

Some of the add-on companies we evaluated are proposing several level of service, at different pricing. We thus registered every protection level available, up to the $30/month threshold we had fixed, ending up registering a total of six security add-on services (two each from the companies offering multiple levels of protection). Six additional hosting accounts were purchased, from different companies, in order to accommodate our tests for these security services. In the following, we refer to them as *sec-1* through *sec-4*. The two variants for companies offering different levels of protection are labeled with a *-basic* or *-pro* suffix, to distinguish, respectively, the cheapest version of the service from the more expensive one. Services in the *-pro* version, for both providers *sec-1* and *sec-2*, allow to scan, daily, all the files on the customer's FTP hosting account, if they are provided with his or her access credentials. We configured both services to enable this kind of scans. The other four security services, contrarily, perform only scans on publicly accessible pages of the websites they are configured to monitor. Such scans include, in most of the cases, checking for malware, malicious links, blacklisted pages, and performing reputation checks on both the website and the provider hosting its contents.

3.5.1 Evaluation of the Security Services

The security services' evaluation schedule was tighter than the normal test evaluation schedule, as we expected security add-on services to react faster to attacks and suspicious account activities, being specially designed for detecting security issues. Thus, the tests on accounts hosting the security add-on services were run for a total duration of 50 days, 10 days for each test, from SH to AV. The SQL injection test was not run on such web hosting accounts, because its attack does not generate any side effect on the hosting account and thus could not be detected by third party external security services.

We noticed that two of the companies providing the add-on security services are listed among the partners of known URL blacklisting services. We therefore used the last 10 days of testing to study reactions to the notification of suspicious URLs to such blacklists. For this, we scheduled a last test consisting in a new deployment of SH, along with the submission of its drive-by download page to a few malicious URL reporting and blacklisting services. The

*Although these services can be purchased separately, several web hosting providers offer security services from third party companies at a discounted price, if purchased in conjunction with a web hosting plan.

Provider	Attack Detection				
	SH	Phish	Bot	AV	SH-BL
sec-1-basic	○	○	○	○	○
sec-1-pro	○	○	○	◐	●
sec-2-basic	○	○	○	○	○
sec-2-pro	○	○	○	○	○
sec-3	○	○	○	○	○
sec-4	○	○	○	○	○

Table 4: Results of our evaluation of third party security services. Symbols and their meanings are the same as in Table 3.

URL blacklisting requests were sent on the same day the tests were deployed. We refer to this test as "SH-BL".

Results are shown in Table 4. One can see that detection capabilities for add-on services are comparable to those of providers. However, in this case, customers pay for a service whose only commitment should be monitoring a website in search of potential vulnerabilities or malicious content. Almost all the services we tested in this part of our study seem to completely fail this objective.

All the services were configured to send notifications to the user whenever a security issue was detected on the monitored website. None of the add-on security services detected anything anomalous during our tests SH, Phish, Bot (attacks were all successful and never blocked by the hosting provider). Test AV was not detected either, but the *sec-1-pro* service raised a warning for having detected the c99 web shell on our hosting account. However, this alert was visible only when logged on the security service's web management panel, where the c99.php file was listed as suspicious. No critical alerts were issued, nor any email was sent to the user as notification for this event. Finally, the only successful detection was performed by the *sec-1-pro* service, detecting our drive-by download page the day following our blacklisting request for its URL. As the *sec-1* security company was listed as one of the partners of the blacklisting service, we expect that our URL blacklisting request was forwarded to the security service right after our submission, thus allowing a timely detection.

4. RELATED WORK

Several works have studied the threats that affect websites all around the world as well as users visiting infected pages [15–17]. Research has been focusing also on the ways in which criminals exploit search engines in order to reach their victims, by poisoning search results for popular queries [7]. Other papers have explored how similar techniques are used in order to find vulnerable websites [12] and web servers [8]. Researchers have also studied how all these activities are combined by criminals in order to be able to conduct attack campaigns in which tens of thousands of hosts are infected [18]. Canali et al. [3] studied the behavior of actual attackers on the web, by installing vulnerable web applications in a controlled environment.

Bau et al. [2] evaluated current commercial tools for detecting vulnerabilities in web applications. Such tools mainly rely on black-box approaches, and are not able to find all possible vulnerabilities.

Recently, a web hosting provider [5] anounced an improvement of his hosting offer by adding free automated website vulnerability scanning, fixing and recovery. Such service is premsumably running as white-box approach on the network and server side. This service is related to what, in our work, we refer to as "add-on" security services. Unfortunately, this service was announced when our experiments were already completed, and it was therefore not possible to integrate it into our results.

Commtouch [4] surveyed 600 compromised websites owners and, among other things, reported on the process by which the websites owners became aware of the compromise. However, this was done with a publicly advertised pool on *detected* compromised websites and may therfore be biased.

Finally, some past work bas been focusing on studying the take-down process employed in the case of phishing websites [10, 11]. This is related to some of the findings we reported in Section 3.3, but is aimed at studying the phenomenon at a ISP and hosting provider level, rather than analyzing the providers' responses one by one and provide details on how they react to abuse notifications.

To our knowledge, this paper is the first attempt to systematically study, on a worldwide scale, how web hosting providers act with regard to the security of their customers and of their own infrastructure - focusing in particular on the detection of compromised accounts, rather than the presence of vulnerabilities.

5. LESSONS LEARNED, CONCLUSIONS

We can summarize the main findings of our experiments around the following five points:

Registration - Top providers invest a considerable effort to collect information about the users who register with them. This procedure can be an effective technique to prevent criminals from hosting their malicious pages on those providers.

Prevention - About 40% of the providers deployed some kind of security mechanism to block simple attacks, ranging from SQL injections to exploitation of common web application vulnerabilities.

Detection - Once the customer is registered, most of the providers do nothing to detect malicious activities or compromised websites - therefore providing very little help to their customers. We were surprised to discover that 21 out of the 22 tested providers did not even run an antivirus once per month (or they run them with old or insufficient signature sets) on the hosted websites. Moreover, none of them considered suspicious having multiple outgoing connection attempts towards an IRC server.

Abuse Notification - Only 36% of the providers reacted to our abuse notifications. When they promptly replied, most of the time their reaction was inappropriate or excessive. None of the *global* providers and only one of the *regional* ones were able to properly manage both the real and the fake complaints in a timely manner.

Security Services - The use of inexpensive security add-on services did not provide any additional layer of security in our experiments. Also the services that were configured to scan the content of our sites via FTP failed to discover the malicious files.

The main differences between *global* and *regional* providers appeared to be in terms of registration verification (in favor of *global* providers) and reaction to real complaints (in favor of *regional* ones).

As we already mentioned in the introduction of this paper, web hosting providers are in the position to play a key role in the security of the Web. In fact, they host millions of websites that are often poorly managed by unexperienced users, and that are likely to be compromised to spread malware and host phishing kits. Unfortunately, all the shared web hosting providers we tested in our study missed this opportunity.

6. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257007.

7. REFERENCES

[1] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

[2] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[3] D. Canali and D. Balzarotti. Behind the scenes of online attacks: an analysis of exploitation behaviors on the web. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, Feb. 2013.

[4] Commtouch and StopBadware. Compromised Websites - An Owner's Perspective. http://stopbadware.org/pdfs/compromised-websites-an-owners-perspective.pdf, February 2012.

[5] W. de Vries. Hosting provider antagonist automatically fixes vulnerabilities in customers' websites. https://www.antagonist.nl/blog/2012/11/hosting-provider-antagonist-automatically-fixes-vulnerabilities-in-customers-websites, November 2012.

[6] fyicenter.com. Credit card number generator - test data generation. http://sqa.fyicenter.com/Online_Test_Tools/Test_Credit_Card_Number_Generator.php, 2010.

[7] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. deseo: combating search-result poisoning. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[8] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 207–216, New York, NY, USA, 2011. ACM.

[9] Larry Ullman. Understand your hosting, five critical e-commerce security tips in five days. Peachpit Blog, 2011. http://www.peachpit.com/blogs/blog.aspx?uk=Understand-Your-Hosting-Five-Critical-E-Commerce-Security-Tips-in-Five-Days.

[10] T. Moore and R. Clayton. Examining the impact of website take-down on phishing. In *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, eCrime '07, pages 1–13, New York, NY, USA, 2007. ACM.

[11] T. Moore and R. Clayton. The consequence of non-cooperation in the fight against phishing. In *eCrime Researchers Summit, 2008*, pages 1 –14, oct. 2008.

[12] T. Moore and R. Clayton. Financial cryptography and data security. chapter Evil Searching: Compromise and Recompromise of Internet Hosts for Phishing, pages 256–272. Springer-Verlag, Berlin, Heidelberg, 2009.

[13] Number 7. osCommerce 'categories.php' Arbitrary File Upload Vulnerability, November 2010. http://www.securityfocus.com/bid/44995/info.

[14] OWASP foundation and TrustWave SpiderLabs. Owasp modsecurity core rule set project. https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project, 2012.

[15] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[16] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, HotBots'07, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[17] N. Provos, M. A. Rajab, and P. Mavrommatis. Cybercrime 2.0: When the cloud turns dark. *Queue*, 7(2):46–47, Feb. 2009.

[18] B. Stone-Gross, M. Cova, C. Kruegel, and G. Vigna. Peering through the iframe. In *INFOCOM, 2011 Proceedings IEEE*, pages 411 –415, april 2011.

[19] webhosting.info. Country-wise top hosts. http://www.webhosting.info/webhosts/tophosts/Country/, 2012.

Behind the Scenes of Online Attacks: an Analysis of Exploitation Behaviors on the Web

Davide Canali
EURECOM, France
canali@eurecom.fr

Davide Balzarotti
EURECOM, France
balzarotti@eurecom.fr

Abstract

Web attacks are nowadays one of the major threats on the Internet, and several studies have analyzed them, providing details on how they are performed and how they spread. However, no study seems to have sufficiently analyzed the typical behavior of an attacker after *a website has been compromised.*

This paper presents the design, implementation, and deployment of a network of 500 fully functional honeypot websites, hosting a range of different services, whose aim is to attract attackers and collect information on what they do during and after their attacks. In 100 days of experiments, our system automatically collected, normalized, and clustered over 85,000 files that were created during approximately 6,000 attacks. Labeling the clusters allowed us to draw a general picture of the attack landscape, identifying the behavior behind each action performed both during and after the exploitation of a web application.

1 Introduction

Web attacks are one of the most important sources of loss of financial and intellectual property. In the last years, such attacks have been evolving in number and sophistication, targeting governments and high profile companies, stealing valuable personal user information and causing financial losses of millions of euros. Moreover, the number of people browsing the web through computers, tablets and smartphones is constantly increasing, making web-related attacks a very appealing target for criminals.

This trend is also reflected in the topic of academic research. In fact, a quick look at the papers published in the last few years shows how a large number of them cover web-related attacks and defenses. Some of these studies focus on common vulnerabilities related to web applications, web servers, or web browsers, and on the way these components get compromised. Others dissect and analyze the internals of specific attack campaigns [13, 5, 17], or propose new protection mechanisms to mitigate existing attacks.

The result is that almost all the web infections panorama has been studied in detail: how attackers scan the web or use google dorks to find vulnerable applications, how they run automated attacks, and how they deliver malicious content to the final users. However, there is still a missing piece in the puzzle. In fact, no academic work seems to have sufficiently detailed the behavior of an average attacker *during* and *after* a website is compromised. Sometimes the attackers are only after the information stored in the service itself, for instance when the goal is to steal user credentials through a SQL injection. But in the majority of the cases, the attacker wants to maintain access to the compromised machine and include it as part of a larger malicious infrastructure (e.g., to act as a C&C server for a botnet or to deliver malicious documents to the users who visit the page).

While the recent literature often focuses on catchy topics, such as drive-by-downloads and black-hat SEO, this is just the tip of the iceberg. In fact, there is a wide variety of malicious activities performed on the Internet on a daily basis, with goals that are often different from those of the high-profile cyber criminals who attract the media and the security firms' attention.

The main reason for which no previous work was done in this direction of research is that almost all of the existing projects based on web honeypots use fake, or 'mock' applications. This means that no real attacks can be performed and thus, in the general case, that all the steps that would commonly be performed by the attacker after the exploitation will be missed.

As a result, to better understand the motivation of the various classes of attackers, antivirus companies have often relied on the information reported by their clients. For example, in a recent survey conducted by Commtouch and the StopBadware organization [7], 600 owners of compromised websites have been asked to fill a questionnaire to report what the attacker did after exploiting the website. The results are interesting, but the approach cannot be automated, it is difficult to repeat, and there is no guarantee that the

users (most of the time not experts in security) were able to successfully distinguish one class of attack from the other.

In this paper we provide, for the first time, a comprehensive and aggregate study of the behavior of attackers on the web. We focus our analysis on two separate aspects: i) the exploitation phase, in which we investigate how attacks are perfomed until the point where the application is compromised, and ii) the post-exploitation phase, in which we examine what attackers do after they take control of the application. The first part deals with methods and techniques (i.e., the *"how"*) used to attack web applications, while the second part tries to infer the reasons and goals (i.e., the *"why"*) behind such attacks.

For this reason, in this paper we do not analyze common SQL injections or cross-site scripting vulnerabilities. Instead, our honeypot is tailored to attract and monitor criminals that are interested in gaining (and maintaining) control of web applications. Our results show interesting trends on the way in which the majority of such attacks are performed in the wild. For example, we identify 4 separate phases and 13 different goals that are commonly pursued by the attackers. Within the limits of the available space, we also provide some insights into a few interesting attack scenarios that we identified during the operation of our honeypots.

The remainder of the paper is organized as follows: in Section 2 we explore the current state of the art concerning web honeypots and the detection and analysis of web attacks. Section 3 describes the architecture of the honeypot network we deployed for our study; Section 4 gives more details about the deployment of the system and the way we collected data during our experiments. Finally, Section 5 and Section 6 summarize the results of our study in term of exploitation and post-exploitation behaviors. Section 7 concludes the paper and provides ideas on future directions in the field.

2 Related Work

Honeypots are nowadays the tool of choice to detect attacks and suspicious behaviors on the Internet. They can be classified in two categories: client honeypots, which detect exploits by actively visiting websites or executing files, and server honeypots, which attract the attackers by exposing one or more vulnerable (or apparently vulnerable) services.

In this study, we are mainly interested in the second category, since our aim is to study the behavior of attackers after a web service has been compromised. Several server-side honeypots have been proposed in the past years, allowing for the deployment of honeypots for virtually any possible service. In particular, we can distinguish two main classes: high-interaction and low-interaction honeypots. The first only *simulate* services, and thus can observe incoming attacks but cannot be really exploited. These honeypots usually have limited capabilities, but are very useful to gather information about network probes and automated attack activities. Examples of these are honeyd [21], Leurre.com [20] and SGNET [16], which are able to emulate several operating systems and services. High-interaction honeypots [19], on the other hand, present to the attacker a fully functional environment that can be exploited. This kind of honeypot is much more useful to get insights into the modus operandi of attackers, but usually comes with high setup and maintenance costs. Due to the fact that they can be exploited, high-interaction honeypots are usually deployed as virtual machines, allowing their original state to be restored after a compromise.

The study of attacks against web applications is often done through the deployment of *web* honeypots. Examples of low-interaction web honeypots are the Google Hack Honeypot [3] (designed to attract attackers that use search engines to find vulnerable web applications), Glastopf [24] and the DShield Web Honeypot project [4], all based on the idea of using templates or patterns in order to mimic several vulnerable web applications. Another interesting approach for creating low interaction web honeypots has been proposed by John et al. [14]: with the aid of search engines' logs, this system is able to identify malicious queries from attackers and automatically generate and deploy honeypot pages responding to the observed search criteria. Unfortunately, the results that can be collected by low-interaction solutions are limited to visits from crawlers and automated scripts. Any manual interaction with the system will be missed, because humans can quickly realize the system is a trap and not a real functional application. Apart from this, the study presented in [14] collected some interesting insights about automated attacks. For example, the authors found that the median time for honeypot pages to be attacked after they have been crawled by a search engine spider is 12 days, and that local file disclosure vulnerabilities seem to be the most sought after by attackers, accounting to more than 40% of the malicious requests received by their heat-seeking honeypots. Other very common attack patterns were trying to access specific files (e.g., web application installation scripts), and looking for remote file inclusion vulnerabilities. A common characteristic of all these patterns is that they are very suitable for an automatic attack, as they only require to access some fixed paths or trying to inject precomputed data in URL query strings. The authors also proposed a setup that is similar to the one adopted in this paper, but they decided to not implement it due to the their concerns about the possibility for attackers to use infected honeypot machines as a stepping stone for other attacks. We explain how we deal with this aspect in Section 3.1.

If interested in studying the real behavior of attackers, one has to take a different approach based on high interac-

tion honeypots. A first attempt in this direction was done by the HIHAT toolkit [18]. Unfortunately, the evaluation of the tool did not contain any interesting finding, as it was run for few days only and the honeypot received only 8000 hits, mostly from benign crawlers. To the best of our knowledge, our study is the first large scale evaluation of the post-exploitation behavior of attackers on the web.

However, some similar work has been done on categorizing the attackers' behavior on interactive shells of high-interaction honeypots running SSH [19, 23]. Some interesting findings of these studies are that attackers seem to specialize their machines for some specific tasks (i.e., scans and SSH bruteforce attacks are run from machines that are different from the ones used for intrusion), and that many of them do not act as knowledgeable users, using very similar attack methods and sequences of commands, suggesting that most attackers are actually following cookbooks that can be found on the Internet. Also, the commands issued on these SSH honeypots highlight that the main activities performed on the systems were checking the software configuration, and trying to install malicious software, such as botnet scripts. As we describe in Section 6, we also observed similar behaviors in our study.

Finally, part of our study concerns the categorization of files uploaded to our honeypots. Several papers have been published on how to detect similarities between source code files, especially for plagiarism detection [6, 26]. Other similarity frameworks have been proposed for the detection of similarities between images and other multimedia formats, mostly for the same purpose. Unfortunately, we saw a great variety of files uploaded to our honeypots, and many of them consisted in obfuscated source code (that renders most plagiarism detection methods useless), binary data or archives. Also, many of the proposed plagiarism detection tools and algorithms are very resource-demanding, and difficult to apply to large datasets. These reasons make the plagiarism detection approaches unsuitable for our needs. The problem of classifying and fingerprinting files of any type has, however, been studied in the area of forensics. In particular, some studies based on the idea of similarity digest have been published in the last few years [15, 25]. These approaches have been proven to be reliable and fast with regard to the detection of similarities between files of any kind, being based on the byte-stream representation of data. We chose to follow this approach, and use the two tools proposed in [15, 25], for our work.

3 HoneyProxy

Our honeypot system is composed of a number of websites (500 in our experiments), each containing the installation of five among the most common - and notoriously vulnerable - content management systems, 17 pre-installed PHP web shells, and a static web site.

We mitigated the problem of managing a large number of independent installations by hosting all the web applications in our facilities, in seven isolated virtual machines running on a VMWare Server. On the hosting provider side we installed only an ad-hoc proxy script (*HoneyProxy*) in charge of forwarding all the received traffic to the right VM on our server. This allowed us to centralize the data collection while still being able to distinguish the requests from distinct hosts. A high-level overview of the system is shown in Figure 1.

The PHP proxy adds two custom headers to each request it receives from a visitor:

- *X-Forwarded-For*: this standard header, which is used in general by proxies, is set to the real IP address of the client. In case the client arrives with this header already set, the final X-Forwarded-For will list all the previous IPs seen, keeping thus track of all the proxies traversed by the client.
- *X-Server-Path*: this custom header is set by the PHP proxy in order to make it possible, for us, to understand the domain of provenance of the request when analyzing the request logs on the virtual machines. An example of such an entry is: `X-Server-Path: http://sub1.site.com/`

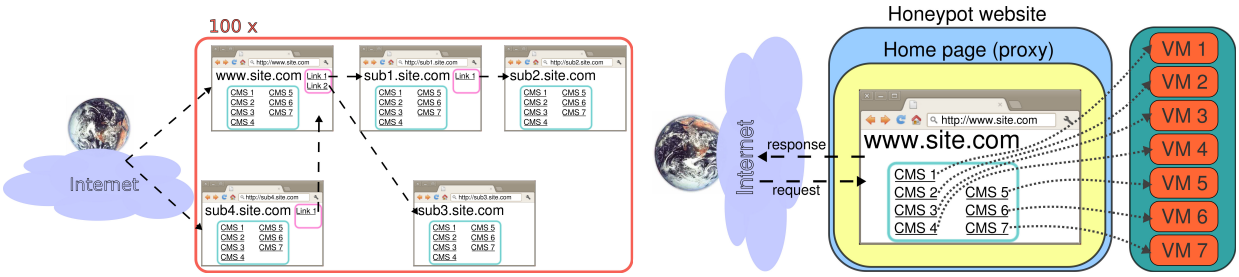
These two headers are transmitted for tracking purposes only between the hosting provider's webserver and the honeypot VM's webserver, and thus are not visible to the users of the HoneyProxy.

3.1 Containment

Each virtual machine was properly set up to contain the attackers and prevent them from causing any harm outside our honeypot. In particular, we blocked outgoing connections (which could otherwise result in attacks to external hosts), patched the source code of the vulnerable blog and forum applications to hide messages posted by spammers (that could result in advertising malicious links), and tuned the filesystem privileges to allow attackers to perpetrate their attacks, but not to take control of the machine or to modify the main source files of each application. Still, the danger of hosting malicious files uploaded by attackers exists, and we tackle this problem by restoring every virtual machine to its pristine state at regular time intervals.

In the following lines, we briefly explain the possible abuses that can be perpetrated on a honeypot machine and present our way to prevent or mitigate them.

- *Gaining high privileges on the machine*. We tackle this problem by using virtual machines with up-to-date software and security patches. In each virtual



(a) Architecture of the system - high level.

(b) Architecture of the system - detail.

Figure 1. High-level architecture of the system.

machine, the web server and all exposed services run as non privileged user. Of course, this solution does not guarantee a protection against new 0-day attacks, but we did our best to limit the attack surface, having only 3 services running on the machine (apache,sshd,mysqld), among which only the web server is exposed to the Internet. We considered the possibility of a 0-day attack against apache fairly remote, and, may it happen, a vast majority of the Internet will be exposed to it as well.

- *Using the honeypot machine as a stepping stone to launch attacks or email campaigns*. This is probably the most important concern that has to be addressed before deploying a fully functional honeypot machine. In our case, we used regular `iptables` rules to block (and log) all outgoing traffic from the virtual machines, except for already established connections. One exception to this rule is the IRC port (6667). We will explain this in more detail in sections 4 and 6.

- *Hosting and distributing illegal content(e.g., phishing pages)*. It is difficult to prevent this threat when applications have remote file upload vulnerabilities. However, it is possible to mitigate the risk of distributing illegal content by limiting the privileges of directories in which files can be uploaded and preventing the modification of all the existing HTML and PHP files. In addition, we also monitor every change on the VM file systems, and whenever a file change is detected, the system takes a snapshot of it. The virtual machine is then restored, at regular intervals, to its original snapshot, thus preventing potentially harmful content from being delivered to victims or indexed by search engines.

- *Illegally promoting goods or services (e.g., spam links)*. Another issue is raised by applications that, as part of their basic way of working, allow users to write and publish comments or posts. This is the case for any blog or forum CMS. These applications are often an easy target for spammers, as we will show in section 5.3.1, and when hosting an honeypot it is important to make sure that links and posts that are posted by bots do not reach any end user or do not get indexed by search engines. We solved this problem by modifying the source code of the blog and forum applications (namely, Wordpress and Simple Machines Forum), commenting out the snippets of code responsible of showing the content of posts. With this modification, it was still possible for attackers to post messages (and for us to collect them), but navigating the posts or comments will only show blank messages.

These countermeasures are limiting the information we can collect with our honeypot (e.g., in the case in which an attacker uploads a back-connect script that is blocked by our firewall), but we believe they are necessary to prevent our infrastructure to be misused for malicious purposes.

3.2 Data Collection and Analysis

Our analysis of the attackers' behavior is based on two sources of information: the logs of the incoming HTTP requests, and the files that are modified or generated by the attackers after they obtain access to the compromised machines.

We built some tools for the analysis of HTTP request logs, allowing us to identify known benign crawlers, known attacks on our web applications, as well as obtaining detailed statistics (number and type of requests received, User-Agent, IP address and geolocalization of every visitor, analysis of the 'Referer' header, and analysis of the inter-arrival time between requests). Our analysis tools also allow us to normalize the time of attack relatively to the timezone of the attacker, and to detect possible correlations between attacks (e.g., an automated script infecting a web application uploading a file, followed by another IP visiting the uploaded file from another IP address). We also developed a parser for the HTTP request logs of the most commonly used PHP web shells, allowing us to extract the requested commands and understand what the attacker was doing on our systems.

We employed two sources of uploaded or modified files: webserver logs and file snapshots from monitored directories. Webserver logs are the primary source of uploaded files, as every file upload processed by our honeypots is fully logged on the apache mod_security logs. File snapshots from monitored directories on the virtual machines, instead, are the primary source for files that are modified or generated on the machine, or about archives or encrypted files that are decompressed on the system. The total number of files we were able to extract from these sources was 85,567, of which 34,259 unique.

Given the high number of unique files we collected, a manual file analysis was practically infeasible. Therefore, in order to ease the analysis of the collected data, we first separate files according to their types, and then apply similarity clustering to see how many of them actually differ from each other in a substantial way. This allows us to identify common practices in the underground communities, such as redistributing the same attack or phishing scripts after changing the owner's name, the login credentials, or after inserting a backdoor.

First of all we employed the *file* Linux utility to categorize files and group them in 10 macro-categories: source code, picture, executable, data, archive, text, HTML document, link, multimedia, and other.

We then observed that many files in the same category only differ for few bytes (often whitespaces due to cut&paste) or to different text included in source code comments. Therefore, to improve the results of our comparison, we first pre-processed each file and transformed it to a normalized form. As part of the normalization process, we removed all double spaces, tabs and new line characters, we removed all comments (both C-style and bash-style), and we normalized new lines and stripped out email addresses appearing in the code. For HTML files, we used the *html2text* utility to strip out all HTML tags as well.

PHP files underwent an additional pre-processing step. We noticed that a large amount of PHP files that were uploaded to our honeypots as result of an exploitation were obfuscated. For files in this form it is very difficult, even with automated tools, to detect similarities among similar files encoded in different ways. In order to overcome this issue, we built an automatic PHP deobfuscation tool based on the *evalhook* PHP extension [10], a module that hooks every call to dynamic code evaluation functions, allowing for step-by-step deobfuscation of PHP code. We deployed our tool on a virtual machine with no network access (to avoid launching attacks or scans against remote machines, as some obfuscated scripts could start remote connections or attacks upon execution) and, for each file with at least one level of deobfuscation (i.e., nested call to *eval()*), we saved its deobfuscated code.

Our approach allowed us to deobfuscate almost all the PHP files that were obfuscated using regular built-in features of the language (e.g., gzip and base64 encoding and decoding, dynamic code evaluation using the *eval()* function). The only obfuscated PHP files we were not able to decode were those terminating with an error (often because of syntax errors) and those encoded with specialized commercial tools, such as Zend Optimizer or ionCube PHP Encoder. However, we observed only three samples encoded with these tools.

In total, we successfully deobfuscated 1,217 distinct files, accounting for 24% of the source code we collected. Interestingly, each file was normally encoded multiple times and required an average of 9 rounds of de-obfuscation to retrieve the original PHP code (with few samples that required a stunning 101 rounds).

3.2.1 Similarity Clustering. Once the normalization step was completed, we computed two similarity measures between any given couple of files in the same category, using two state-of-the-art tools for (binary data) similarity detection: *ssdeep* [15] and *sdhash* [25]. We then applied a simple agglomerative clustering algorithm to cluster all files whose similarity score was greater than 0.5 into the same group.

We discarded files for which our analysis was not able to find any similar element. For the remaining part, we performed a manual analysis to categorize each cluster according to its purpose. Since files had already been grouped by similarity, only the analysis (i.e., opening and inspecting the content) of one file per group was necessary. During this phase, we were able to define several file categories, allowing us to better understand the intentions of the attackers. Moreover, this step allowed us to gain some insights on a number of interesting attack cases, some of which are reported in the following sections as short in-depth examples.

4 System Deployment

The 500 honeyproxy have been deployed on shared hosting plans[1] chosen from eight of the most popular international web hosting providers on the Internet (from USA, France, Germany, and the Netherlands). In order for our HoneyProxy to work properly, each provider had to support the use of the cURL libraries through PHP, and allow outgoing connections to ports other than 80 and 443.

To make our honeypots reachable from web users, we purchased 100 bulk domain names on GoDaddy.com with privacy protection. The domains were equally distributed among the `.com`, `.org`, and `.net` TLDs, and assigned evenly across the hosting providers. On each hosting

[1] This is usually the most economical hosting option, and consists in having a website hosted on a web server where many other websites reside and share the machine's resources.

provider, we configured 4 additional subdomains for every domain, thus having 5 distinct websites (to preserve the anonymity of our honeypot, hereinafter we will simply call them www.site.com, sub1.site.com, sub2.site.com, sub3.site.com, sub4.site.com) Finally, we advertised the 500 domains on the home page of the authors and on the research group's website by means of transparent links, as already proposed by Müter et al. [18] for a similar purpose.

We used a modified version of the *ftp-deploy* script [11] to upload, in batch, a customized PHP proxy to each of the 500 websites in our possession. This simplified the deployment and update of the PHP proxy, and uniformed the way in which we upload files to each hosting service[2], Thanks to a combination of `.htaccess`, ModRewrite, and cURL, we were able to transparently forward the user requests to the appropriate URL on the corresponding virtual machine. Any attempt to read a non-existing resource, or to access the proxy page itself would result in a blank error page shown to the user. Not taking into account possible timing attacks or intrusions on the web hosting provider's servers, there was no way for a visitor to understand that he was talking to a proxy.

The HoneyProxy system installed on every website is composed of an index file, the PHP proxy script itself and a configuration file. The index file is the home page of the website, and it links to the vulnerable web applications and to other honeypot websites, based on the contents of the configuration file.

The linking structure is not the same for every subdomain, as can be noticed taking a closer look at Figure 1(a). Indeed, each subdomain links to at most 2 different subdomains under its same domain. We put in place this small linking graph with the aim of detecting possible malicious traffic from systems that automatically follow links and perform automated attacks or scans.

4.1 Installed Web Applications

We installed a total of 5 vulnerable CMSs on 7 distinct Virtual Machines. The Content Management Systems were chosen among the most known and vulnerable ones at the time we started our deployment. For each CMS, we chose a version with a high number of reported vulnerabilities, or at least with a critical one that would allow the attacker to take full control of the application. We also limited our choice to version no more than 5 years old in order to ensure our websites are still of interest to attackers.

Our choice was guided by the belief that attackers are always looking for low-hanging fruits. On the other hand, our honeypots will probably miss sophisticated and unconventional attacks, mostly targeted to high profile organizations or well known websites. However, these attacks are not easy to study with simple honeypot infrastructures and are therefore outside the scope of our study.

Table 1 describes the vulnerable applications installed on the 7 virtual machines, along with their publication date and the list of their known and exploitable vulnerabilities. We have installed two instances of WordPress 2.8, one with CAPTCHA protection on comments, and one without CAPTCHA protection, in order to see if there are attackers that register fake accounts by hand, or systems that are capable of automatically solve CAPTCHAs. This does not seem to be the case, since we did not receive any post on the CAPTCHA-protected blog. Therefore, we will not discuss it any further in the rest of the paper.

4.2 Data Collection

We collected 100 days of logs on our virtual machines, starting December 23rd, 2011. All the results presented in our work derive from the analysis of the logs of these 7 machines.

Overall, we collected 9.5 Gb of raw HTTP requests, consisting in approximately 11.0M GET and 1.9M POST. Our honeypots were visited by more than 73,000 different IP addresses, spanning 178 countries and presenting themselves with more than 11,000 distinct User-Agents. This is over one order of magnitude larger than what has been observed in the previous study by John et al. on low interaction web-application honeypots [14]. Moreover, we also extracted over 85,000 files that were uploaded or modified during attacks against our web sites.

There are two different ways to look at the data we collected: one is to identify and study the attacks looking at the web server logs, and the other one is to try to associate a *goal* to each of them by analyzing the uploaded and modified files. These two views are described in more detail in the next two Sections.

5 Exploitation and Post-Exploitation Behaviors

In order to better analyze the behavior of attackers lured by our honeypots, we decided to divide each attack in four different phases: discovery, reconnaissance, exploitation, and post-exploitation. The *Discovery* phase describes how attackers find their targets, e.g. by querying a search engine or by simply scanning IP addresses. The *Reconnaissance* phase contains information related to the way in which the pages were visited, for instance by using automated crawlers or by manual access through an anonymization proxy. In the *Exploitation* phase we describe the number

[2]Shared web hosting services from different providers usually come with their own custom administrative web interface and directory structure, and very few of them offer ssh access or other 'advanced' management options. Thus, the only possible way to automate the deployment of the websites was to use FTP, the only protocol supported by every provider.

VM #	CMS, version	Plugins	Description	Vulnerabilities
1	phpMyAdmin, 3.0.1.1	-	MySQL database manager	PHP code injection
2	osCommerce, 2.2-RC2a	-	Online shop	2 remote file upload, arbitrary admin password modification
3	Joomla, 1.5.0	com_graphics, tinymce	Generic/multipurpose portal	XSS, arbitrary admin password modification, remote file upload, local file inclusion
4	Wordpress, 2.8	kino, amphion lite theme	Blog (non moderated comments)	Remote file include, admin password reset
5	Simple Machines Forum (SMF), 1.1.3	-	Forum (non moderated posts)	HTML injection in posts, stored XSS, blind SQL injection, local file include (partially working)
6	PHP web shells, static site	-	Static site and 17 PHP shells (reachable through hidden links)	PHP shells allow to run any kind of commands on the host
7	Wordpress, 2.8	kino, amphion lite theme	Blog (captcha-protected comments)	Remote file include, admin password reset

Table 1. Applications installed on the honeypot virtual machines, together with a brief description and a list of their known and exploitable vulnerabilities.

and types of actual attacks performed against our web applications. Some of the attacks reach their final goal themselves (for instance by changing a page to redirect to a malicious website), while others are only uploading a second stage. In this case, the uploaded file is often a web shell that is later used by the attacker to manually log in to the compromised system and continue the attack. We refer to this later stage as the *Post-Exploitation* phase.

It is hard to present all possible combinations of behaviors. Not all phases are always present in each attack (e.g., reconnaissance and exploitation can be performed in a single monolithic step), some of the visits never lead to any actual attack, and sometimes it is just impossible to link together different actions performed by the same attacker with different IP addresses. However, by extracting the most common patterns from the data collected at each stage, we can identify the "typical attack profile" observed in our experiment. Such profile can be summarized as follows:

1. 69.8% of the attacks start with a *scout bot* visiting the page. The scout often tries to hide its User Agent or disguise as a legitimate browser or search engine crawler.

2. Few seconds after the scout has identified the page as an interesting target, a second automated system (hereinafter *exploitation bot*) visits the page and executes the real exploit. This is often a separate script that does not fake the user agent, therefore often appearing with strings such as `libwww/perl`.

3. If the vulnerability allows the attacker to upload a file, in 46% of the cases the exploitation bot uploads a web shell. Moreover, the majority of the attacks upload the same file multiple times (in average 9, and sometimes up to 30), probably to be sure that the attack was successful.

4. After an average of 3 hours and 26 minutes, the attacker logs into the machine using the previously uploaded shell. The average login time for an attacker interactive session is 5 minutes and 37 seconds.

While this represents the most common behavior extracted from our dataset, many other combinations were observed as well - some of which are described in the rest of the section. Finally, it is important to mention that the attack behavior may change depending on the application and on the vulnerability that is exploited. Therefore, we should say that the previous description summarizes the most common behavior of attacks against osCommerce 2.2 (the web application that received by far the largest number of attacks among our honeypots).

Figure 2 shows a quick summary of some of the characteristics of each phase.[3] More information and statistics are reported in the rest of the section. Then, based on the analysis of the files uploaded or modified during the exploitation and post-exploitation phases, in Section 6 we will try

[3] The picture does not count the traffic towards the open forum, because its extremely large number of connections compared with other attacks would have completely dominated the statistics.

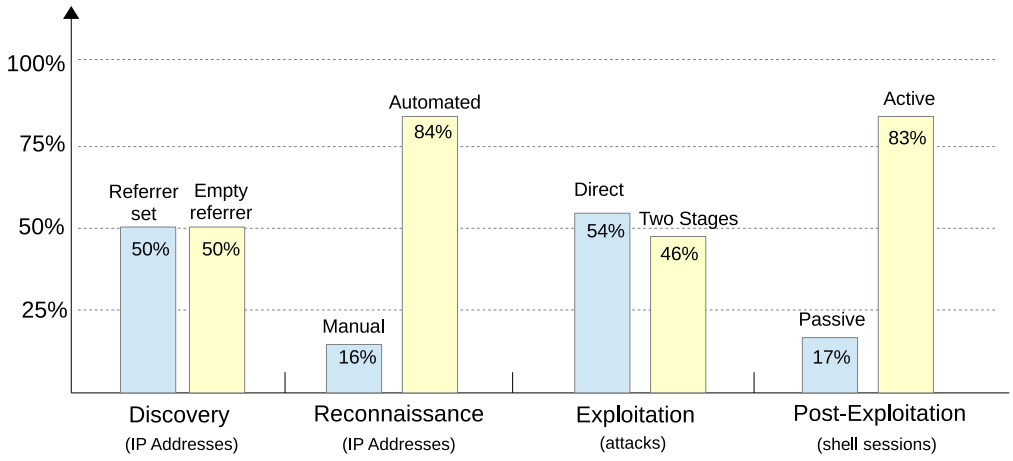


Figure 2. Overview of the four phases of an attack

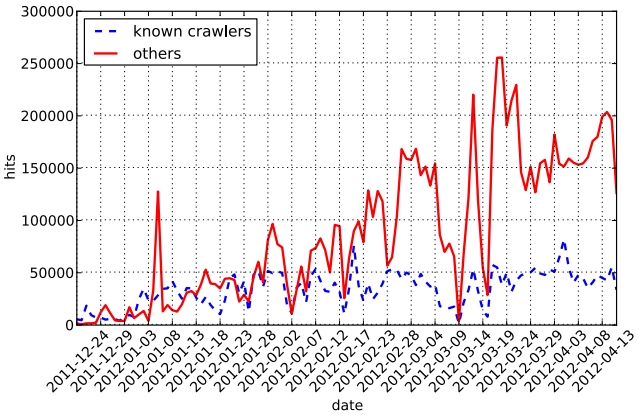


Figure 3. Volume of HTTP requests received by out honeypots during the study.

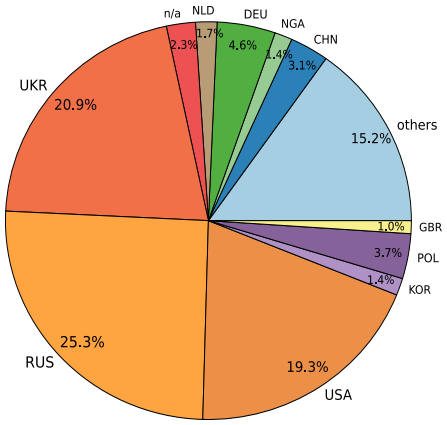


Figure 4. Amount of requests, by issuing country.

to summarize the different goals and motivations behind the attacks we observed in our experiments.

5.1 Discovery

The very first HTTP request hit our honeypot proxies only 10 minutes after the deployment, from Googlebot. The first direct request on one IP address of our virtual machines (running on port 8002) came after 1 hour and 50 minutes.

During the first few days, most of the traffic was caused by benign web crawlers. Therefore, we designed a simple solution to filter out benign crawler-generated traffic from the remaining traffic. Since HTTP headers alone are not trustable (e.g., attackers often use User Agents such as 'Googlebot' in their scripts) we collected public information available on bots [2, 1] and we combined them with information extracted from our logs and validated with WHOIS results in order to identify crawlers from known companies. By combining UserAgent strings and the IP address ranges associated to known companies, we were able to identify with certainty 14 different crawlers, originating from 1965 different IPs. Even though this is not a complete list (e.g, John et al. [14] used a more complex technique to identify 16 web crawlers), it was able to successfully filter out most of the traffic generated by benign crawlers.

Some statistics about the origin of the requests is shown in Figure 3. The amount of legitimate crawler requests is more or less stable in time, while, as time goes by and the honeypot websites get indexed by search engines and linked on hacking forums or on link farming networks, the number of requests by malicious bots or non-crawlers has an almost

linear increase.

When plotting these general statistics we also identified a number of suspicious spikes in the access patterns. In several cases, one of our web applications was visited, in few hours, by several thousands of unique IP addresses (compared with an average of 192 per day), a clear indication that a botnet was used to scan our sites.

Interestingly, we observed the first suspicious activity only 2 hours and 10 minutes after the deployment of our system, when our forum web application started receiving few automated registrations. However, the first posts on the forum appeared only four days later, on December 27th. Even more surprising was the fact that the first visit from a non-crawler coincided with the first attack: 4 hours 30 minutes after the deployment of the honeypots, a browser with Polish locale visited our osCommerce web application [4] and exploited a file upload vulnerability to upload a malicious PHP script to the honeypot. Figure 4 summarizes the visits received by our honeypot (benign crawlers excluded), grouped by their geolocalization.

5.1.1 Referer Analysis. The analysis of the Referer HTTP header (whenever available) helped us identify how visitors were able to find our honeypots on the web. Based on the results, we can distinguish two main categories of users: criminals using search engines to find vulnerable applications, and victims of phishing attacks following links posted in emails and public forums (an example of this phenomenon is discussed in Section 6.8).

A total of 66,449 visitors reached our honeypot pages with the Referer header set. The domains that appear most frequently as referrers are search engines, followed by web mails and public forums. Google is leading with 17,156 entries. Other important search engines used by the attackers to locate our websites, were Yandex (1,016), Bing (263), and Yahoo (98). A total of 7,325 visitors arrived from web mail services (4,776 from SFR, 972 from Facebook, 944 were from Yahoo!Mail, 493 from Live.com, 407 from AOL Mail, and 108 from comcast.net). Finally, 15,746 requests originated from several public web forums, partially belonging to hacking communities, and partially just targeted by spam bots.

Finally, we extracted search queries (also known as 'dorks', when used for malicious purposes) from Referer headers set by the most common web search engines. Our analysis shows that the search terms used by attackers highly depend on the application deployed on the honeypot. For example, the most common dork that was used to reach our Joomla web application contained the words *'joomla allows you'*, while the Simple Machines Forum was often reached by searching *'powered by smf'*. Our machine containing public web shells was often reached via dorks like *'inurl:c99.php'*, *'[cyber anarchy shell]'* or even *'[ftp buteforcer] [security info] [processes] [mysql] [php-code] [encoder] [backdoor] [back-connection] [home] [enumerate] [md5-lookup] [word-lists] [milw0rm it!] [search] [self-kill] [about]'*. The latter query, even though very long, was used more than 150 times to reach our machine with web shells. It was probably preferred to searching via `'intitle:'` or `'inurl:'` because script names and titles are often customized by attackers and as such searching for their textual content may return more results than searching for fixed url patterns or page titles. Some specialized search engines appear to be used as well, such as devilfinder.com, which was adopted in 141 cases to reach some of the shells on our machines. This search engine claims to show more low-ranking results than common search engines, not to store any search data, and to return up to 300 results on the same web page, making it very suitable for attackers willing to search for dorks and collect long lists of vulnerable websites.

5.2 Reconnaissance

After removing the legitimate crawlers, the largest part of the traffic received by our honeypots was from unidentified sources, many of which were responsible of sending automated HTTP requests. We found these sources to be responsible for the majority of attacks and spam messages targeting our honeypots during the study.

However, distinguishing attackers that manually visited our applications from the ones that employed automated scout bots is not easy. We applied the following three rules to flag the automated requests:

- *Inter-arrival time*. If requests from the same IP address arrive at a frequency higher than a certain threshold, we consider the traffic as originated from a possible malicious bot.

- *Request of images*. Automated systems, and especially those having to optimize their speed, almost never request images or other presentation-related content from websites. Scanning web logs for visitors that never request images or CSS content is thus an easy way of spotting possible automated scanners.

- *Subdomain visit pattern*. As described in Section 4, each web site we deployed consisted in a number of sub-domains linked together according to a predetermined pattern. If the same IP accesses them in a short time frame, following our patterns, then it is likely to be an automated crawler.

[4] Since UserAgent information can be easily spoofed, we cannot prove our assumptions about the browser and tools run by the attacker, and his or her locale, are correct.

For example, after removing the benign crawlers, a total of 9.5M hits were received by systems who did not request any image, against 1.8M from system that also requested images and presentation content. On the contrary, only 641 IP addresses (responsible for 13.4K hits) visited our websites by following our links in a precise access pattern. Among them, 60% followed a breadth first approach.

85% of the automated requests were directed to our forum web application, and were responsible for registering fake user profiles and posting spam messages. Of the remaining 1.4M requests directed to the six remaining honeypot applications, 95K were mimicking the User-Agent of known search engines, and 264K switched between multiple User-Agents over time. The remaining requests did not contain any suspicious User-Agent string, did not follow paths between domains, neither requested images. As such, we classified them as unknown (possibly benign) bots.

5.3 Exploitation

The first important activity to do in order to detect exploitation attempts was parsing the log files in search of attack traces. Luckily, knowing already the vulnerabilities affecting our web applications allowed us to quickly and reliably scan for attacks in our logs using a set of regular expressions.

Overall, we logged 444 distinct exploitation sessions. An interesting finding is that 310 of them adopted two or more different User-Agent strings, appearing in short sequence from the same IP address. As explained in the beginning of Section 5, this often happens when attackers employ a combination of scout bots and automatic attack scripts in order to speed up attacks and quickly find new targets. In particular, in two thirds (294) of the total exploitation sessions we observed, the User-Agent used for the exploitation was the one associated to the LibWWW Perl library (`libwww/perl`).

In some of these exploitation sessions, the attacker tried to disguise her tools and browser as known benign bots. Some crawler User-Agent strings that were often used during exploitation sessions were: *FreeWebMonitoring*, *Gigabot/3.0*, *gsa-crawler*, *IlTrovatore-Setaccio/1.2*, *bingbot/2.0;*, and *Googlebot/2.1*.

The most remarkable side effect of every exploitation session is the upload or modification of files on the victim machine. Quite surprisingly, we noticed that when an exploitation session uploads a file, the file is uploaded in average 9.75 times. This strange behavior can be explained by the fact that most of the exploitation tools are automated, and since the attacker does not check in real-time whether each exploit succeeded or not, uploading the same file multiple times can increase the chance for the file to be successfully uploaded at least once.

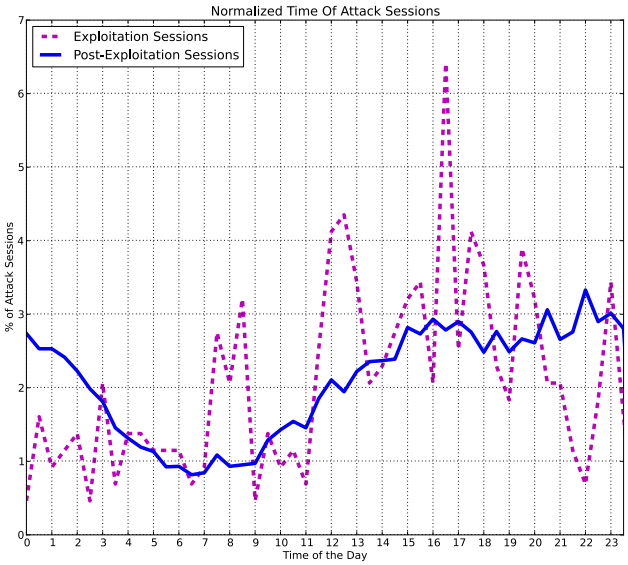


Figure 5. Normalized times distribution for attack sessions

Using the approach presented in Section 3.2, we automatically categorized the files uploaded to our honeypots as a result of exploiting vulnerable services. We then correlated information about each attack session with the categorization results for the collected files. Results of this phase show that the files uploaded during attack sessions consist, in 45.75% of the cases, in web shells, in 17.25% of the cases in phishing files (single HTML pages or complete phishing kits), in 1.75% of the cases in scripts that automatically try to download and execute files from remote URLs, and in 1.5% of the cases in scripts for local information gathering. Finally, 32.75% of the uploaded files were not categorized by our system, either because they were not similar to anything else that we observed, or because they were multimedia files and pictures (e.g., images or soundtracks for defacement pages) that were not relevant for our study.

Figure 5 shows the normalized times of the attacks received by our honeypots. The values were computed by adjusting the actual time of the attack with the timezone extracted from the IP geolocalization. As such, our normalization does not reflect the correct value in case the attacker is proxying its connection through an IP in a different part of the world. However, the graph shows a clear daylight trend for both the exploitation and post-exploitation phases. In particular, for the interactive sessions we observed fewer attacks performed between 4am and 10am, when probably also the criminals need to get some sleep. Interestingly, also the exploitation phase, that is mostly automated, shows a similar trend (even though not as clear). This could be the consequence of scans performed through botnet infected

machines, some of which are probably turned off by their users during the night.

Searching our attack logs for information about attackers reaching directly our virtual machines, without passing through the honeypot proxies, we found that a small, but still significant number of attacks were carried out directly against the ip:port of our honeypots. In particular, we found 25 of such attack sessions against our e-commerce web honeypot and 19 against our machine hosting the web shells and the static website. In both cases, the attacker may have used a previous exploit to extract the IP of our machines (stored in a osCommerce configuration file that was often downloaded by many attackers, or by inspecting the machine through an interactive shell) and use this information in the following attacks.

5.3.1 Posts. Since the 1st day of operation, our forum application received a very large amount of traffic. Most of it was from automated spamming bots that kept flooding the forum with fake registrations and spam messages. We analyzed every snapshot of the machine's database in order to extract information about the forum's posts and the URLs that were embedded in each of them. This allowed us to identify and categorize several spam and link farming campaigns, as well as finding some rogue practices such as selling forum accounts.

A total of 68,201 unique messages were posted on the forum during our study, by 15,753 users using 3,144 unique IP addresses. Daily statistics on the forum show trends that are typical of medium to high traffic message boards: an average of 604 posts per day (with a max of 3085), with an average of 232 online users during peak hours (max 403).

Even more surprising than the number of posts is the number of new users registered to the forum: 1907 per day in average, and reaching a peak of 14,400 on March 23, 2012. This phenomenon was so common that 33.8% of the IP addresses that performed actions on our forum were responsible of creating at least one fake account, but never posted any message. This finding suggests there are some incentives for criminals to perform automatic user registrations, perhaps making this task even more profitable than the spamming activity itself. Our hypothesis is that, in some cases, forum accounts can be sold in bulk to other actors in the black market. We indeed found 1,260 fake accounts that were created from an IP address and then used few days later by other, different IPs, to post messages. This does not necessarily validate our hypothesis, but shows at least that forum spamming has become a complex ecosystem and it is difficult, nowadays, to find only a single actor behind a spam or link farming campaign.

A closer look at the geolocation of IP addresses responsible for registering users and posting to the forum shows that most of them are from the United States or Eastern Europe countries (mostly Russia, Ukraine, Poland, Latvia, Romania). A total of 6687 distinct IP addresses were active on our forum (that is, posted at least one message or registered one or more accounts). Among these, 36.8% were associated to locations in the US, while 24.6% came from Eastern European countries. The country coverage drastically changes if we consider only IP addresses that posted at least one message to the forum. In this case, IPs from the United States represent, alone, 62.3% of all the IP addresses responsible for posting messages (Eastern Europe IPs in this case represent 21.2% of the total).

Finally, we performed a simple categorization on all the messages posted on the forum, based on the presence of certain keywords. This allowed us to quickly identify common spam topics and campaigns. Thanks to this method, we were able to automatically categorize 63,763 messages (93.5% of the total).

The trends we extracted from message topics show clearly that the most common category is drugs (55% of the categorized messages, and showing peaks of 2000 messages per day), followed by search engine optimization (SEO) and electronics (11%), adult content (8%), health care and home safety (6%).

All the links inserted in the forum posts underwent an in-depth analysis using two automated, state-of-the-art tools for the detection of malicious web pages, namely Google Safe Browsing [22] and Wepawet [8]. The detection results of these two tools show that, on the 221,423 URLs we extracted from the forum posts, a small but not insignificant fraction (2248, roughly 1 out of 100) consisted in malicious or possibly harmful links.

5.4 Post-Exploitation

The post-exploitation phase includes the analysis of the interaction between the attackers and the compromised machines. In our case, this is done through the web shells installed during the exploitation phase or, to increase the collected data, through the access to the public shells that we already pre-installed in our virtual machines.

The analysis of the post-exploitation phase deserves special attention since it is made of interactive sessions in which the attackers can issue arbitrary commands. However, these web shells do not have any notion of session: they just receive commands via HTTP requests and provide the responses in a state-less fashion.

During our experiments we received a total of 74,497 shell commands. These varied from simple file system navigation commands, to file inspection and editing, up to complex tasks as uploading new files or performing network scans.

To better understand what this number represents, we decided to group together individual commands in virtual "in-

teractive sessions" every time they are issued from the same IP, and the idle time between consecutive commands is less than 5 minutes.

According to this definition, we registered 232 interactive sessions as a consequence of one of the exploited services, and 8268 in our pre-installed shells[5]. The average session duration was of 5 minutes and 37 seconds, however, we registered 9 sessions lasting more than one hour each. The longest, in terms of commands issued to the system, was from a user in Saudi Arabia that sent 663 commands to the shell, including the manual editing of several files.

Interestingly, one of the most common actions performed by users during an attack is the upload of a custom shell, even if the attacker broke into the system using a shell that was already available on the website. The reason for this is that attackers know that, with a high probability, shells installed by others will contain backdoors and most likely leak information to their owner. In addition to the 17 web shells supported by our tools, we also identified the HTTP patterns associated to the most common custom shells uploaded by the attackers, so that we could parse the majority of commands issued to them.

In 83% of the cases, attackers tried to use at least one active command (uploading or editing a file, changing file permissions, creating files or directories, scanning hosts, killing a process, connecting to a database, sending emails, etc.). The remaining sessions were purely passive, with the attackers only browsing our system and downloading source and configuration files.

Finally, in 61% of the sessions the attackers uploaded a new file, and in 50% of them they tried to modify a file already on the machine (in 13% of the cases to perform a defacement). Regarding individual commands, the most commonly executed were the ones related to listing and reading files and directories, followed by editing files, uploading files, running commands on the system, listing the processes running on the system, and downloading files.

6 Attackers Goals

In this section we shift the focus from the way the attacks are performed to the motivation behind them. In other words, we try to understand what criminals do after they compromise a web application. Do they install a botnet? Do they try to gain administrator privileges on the host? Do they modify the code of the application and insert backdoors or malicious iFrames?

[5]For the pre-installed shells, we also removed sessions that contained very fast sequences of commands or that did not fetch images on the pages, because they could have been the result of crawlers visiting our public pages. Since shells uploaded by attackers were not linked from any page, we did not apply this filtering to them.

File Type	Clustered	Not Clustered	Clusters
Archive	335 (82.6%)	71 (17.4%)	159
Data	221 (62.5%)	133 (37.5%)	87
Executable	102 (82.3%)	22 (17.7%)	41
HTML doc	4341 (100.0%)	0 (0%)	822
Image	1703 (81.9%)	374 (18.1%)	811
Source code	3791 (100.0%)	0 (0%)	482
Text	886 (43.8%)	1138 (56.2%)	219
Various	118 (65.9%)	61 (34.1%)	42
Total	11,497 (86.5%)	1799 (13.5%)	2663

Table 2. Results of clustering

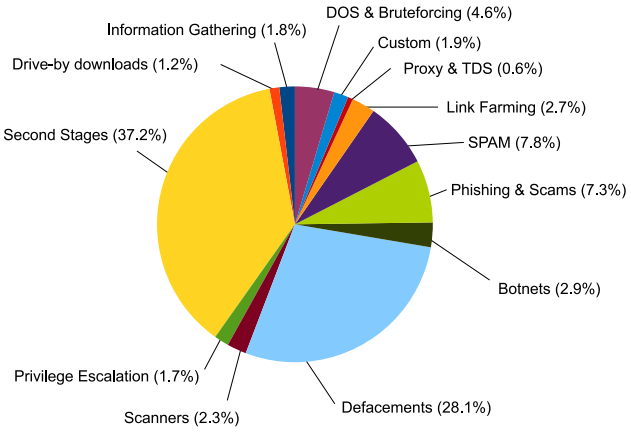


Figure 6. Attack behavior, based on unique files uploaded

To answer these questions, we analyzed the files uploaded during the exploitation phase, and the ones created or modified during the post-exploitation phase. We normalized each file content as explained in Section 3, and we clustered them together according to their similarity. Finally, we manually labeled each cluster, to identify the "purpose" of the files. The results of the clustering are summarized in table 2 and cover, in total, 86.4% of the unique files collected by our honeypots. For them, Figure 6 shows the distribution of the file categories[6]. For example, 1.7% of the unique files we observed in our experiments were used to try to escalate the privileges on the compromised machine. This is different from saying that 1.7% of the attackers tried to escalate the privileges of the machine. Unfortunately, linking the files to the attacks in which they were used is not always possible. Therefore, we computed an estimation of the attackers that performed a certain action by identifying each unique IP that uploaded a certain file during an

[6]We removed from the graph the irrelevant and damaged documents, that accounted in total for 10% of the files.

attack. Identifying an attacker only based on his or her IP address is not always correct, but still provides a reasonable approximation. Thus, if we say that a certain category has an *estimated attackers ratio* of 20%, it means that 1 attacker out of 5 uploaded at least one file of that category during his or her operation.

Only 14% of the attackers uploaded multiple files belonging at least to two separate categories. This means that most of the attacks have a precise goal, or that attackers often change their IP addresses, making it very hard for us to track them.

In the rest of the section, we briefly introduce each of the 13 categories.

6.1 Information gathering

Unique files ratio	1.8%
Estimated attackers ratio	2.2%

These files consist mainly in automated scripts for the analysis of the compromised system, and are often used as a first stage of a manual attack, in which the attacker tries to gather information on the attacked system before proceeding with other malicious actions. In general, we observed a number of attackers using scripts to search, archive, and download several system configuration files.

For example, an attack using such tools hit our honeypots on April 7, 2012. The attacker, using a normal browser and coming from a Malaysian IP address, uploaded a script called `allsoft.pl`. Once executed, the script scans the system for a list of directories containing configuration files of known CMSs (e.g., Wordpress, Joomla, WHM, phpBB, vBulletin, ...), creates a tar archive containing all the files it was able to find, and returns to the attacker a link to the created archive, that can thus be easily downloaded. The script iterates on both the users and the possible multiple home directories in the system trying to gather information from as many accounts as possible on the attacked machine.

6.2 Drive-by Downloads

Unique files ratio	1.2%
Estimated attackers ratio	1.1%

We have witnessed few attacks that aimed at creating drive-by download webpages, by inserting custom exploit code in the HTML source of the web pages of our honeypots, or by uploading documents that contain exploits for known browser vulnerabilities. This kind of activity is aimed at exploiting users visiting the website, typically to convert their machines in bots that can be later used for a large spectrum of illicit activity.

An example of such attacks was the *intu.html* web page uploaded to one of our honeypots on February 28th, 2012. When opened, the page shows *'Intuit Market. Loading your order, please wait...'*. Behind the scenes, a malicious javascript loads an iframe pointing to a document hosted at `twistedtarts.net`. This document is malicious and contains two exploits, for CVE-2010-0188 and CVE-2010-1885. Wepawet [8] reported the document as malicious on the same day this webpage was uploaded to our honeypots.

6.3 Second Stages

Unique files ratio	37.2%
Estimated attackers ratio	49.4%

This category includes downloaders (programs designed to download and execute another file), uploaders (web pages that can be used to remotely upload other files), web shells, and backdoors included in already existing documents. These are the tools of choice for attackers to perform manual web-based attacks. The reason is that such tools allow either to upload any file to the victim machine, or to issue arbitrary commands as if the attacker was logged in to one of the server's terminals. The majority of the attacks logged by our honeypot adopted a mix of web shells and custom scripts to try to hack the machine and install malicious software on it.

An example of this behavior is the attack that started at 6:50 am (GMT) on January 1st, 2012. An IP address from Englewood, Colorado, with an User-Agent set to *'blackberry8520_ver1_subvodafone'* connected directly to our honeypot virtual machine running osCommerce and exploited a file upload vulnerability, uploading several different PHP scripts, all of them launching IRC bots connecting to different IRC servers. The same person also uploaded a PHP shell, and used it to download the configuration file of the CMS installed on the machine.

The fact that the attacker was not connecting through our HoneyProxy infrastructure but directly to our IP address was unusual, and attracted our attention. Searching backwards in our logs starting the date of the attack, we found out that less than 24 hours before, an automated system with an User-Agent set to *'bingbot/2.0'* connected to one of our websites from another IP address from Englewood, Colorado, exploited a vulnerability and downloaded the osCommerce configuration file, which contains the real IP of our virtual machine hosting the e-commerce web application.

6.4 Privilege Escalation

Unique files ratio	1.7%
Estimated attackers ratio	2.2%

Privilege escalation exploits are among the oldest types of exploits in the computer security history, but are still

among the most sought after, as they allow an attacker to gain administrator privileges and thus full control of vulnerable machines. Successfully executing a privilege escalation exploit on server machines used in a shared web hosting environment would make the attacker in the position to modify the files of every website hosted on the server, possibly allowing for mass exploitations of hundreds or even thousands of websites at the same time.

An example of such kind of attack hit our honeypots on February 9, 2012. An attacker with an Hungarian IP address uploaded a file called `mempodipper.c` to our machine hosting the web shells, and used one of the shells to try to compile its source code with *gcc*. The machine had no available compiler, thus, less than 5 minutes later, the attacker uploaded a pre-compiled ELF binary named `mempodipper`, and tried to execute it through one of the shells. We found this exploit to be for a very recent vulnerability, the CVE-2012-0056, published less than 20 days before this attack. At the time of the attack, the exploit for this vulnerability, titled *Linux Local Privilege Escalation via SUID /proc/pid/mem Write* was already publicly available [27]. However, the kernel of our virtual machines was not vulnerable to it.

6.5 Scanners

Unique files ratio	2.3%
Estimated attackers ratio	2.8%

This kind of activity is performed to find other local or remote vulnerable target websites that could possibly be exploited by the attacker. For example, FTP scanning, querying search engines using 'dorks', or trying to list all the domain names being hosted on the machine belong to this category.

A concrete example is the *trdomain.php* page, uploaded to one of our honeypots on December 26th, from a Turkish IP address. It contains a local domain name scanner, that pulls the domain names configured on the machine from the local configuration files (such as named.conf), gets their PageRank from Google, as well as their document root and their owner's username, and returns a web page with a list containing all this information. The title of the page is *'Domain ve User ListeLiyici — by W£ßRooT '*; as of today, searching such title on the web still yields many results, showing that this kind of attack is very common and wide spread.

6.6 Defacements

Unique files ratio	28.1%
Estimated attackers ratio	27.7%

Attacks of this kind are among the most frequent ones on our honeypots. In this kind of attack, the attackers modify existing web pages on the honeypot, or upload new pages with the purpose of claiming their responsibility for hacking the website. Usually, but not always, the claims are accompanied by religious or politic propaganda, or by funny or shocking images. Many of the attackers performing such attacks even insert links to their personal website or Facebook page, where one can see they are mainly teenagers looking for fame and bragging in front of their friends.

One of the many defacements attacks that hit our honeypots happened around 8 pm GMT on the 6th of March. Somebody connecting from a German IP address found one of the hidden shells in our machine hosting the static website, and used it to edit one of the static html pages hosted on the machine. The code of the page was thus uploaded using copy-and-paste in a textarea provided by the web shell. The defacement page contained a short slogan from the author, an animated javascript text slowly unveiling a Portuguese quote, and a set of links to the personal Twitter pages of each member of the hacking crew, some of which had more than 1000 tweets and several hundred followers. Quickly looking at these Twitter profiles, we found out that all the members are actively posting their defacements on their profile pages. Apparently, they do so in order to build some sort of reputation. This is confirmed by the URL they posted as a personal webpage on Twitter, a web page from the `zone-h.org` website, reporting statistics about previous defacements of the crew. The statistics are quite impressive: at the time of writing the whole crew has claimed more than 41,600 defacements starting July 20, 2011, of which almost 500 are on important websites with high reputation (governative websites, universities, multinational corporations, etc.).

Thanks to attacks like this we found out that it is common practice among attackers to advertise their defacements on publicly accessible 'defacement' showcases, such as the one on the `zone-h.org` website. It seems that some of these people are really in a sort of competition in order to show off their presumed skills at hacking websites, and our honeypot domains were often reported as trophies by several groups.

6.7 Botnets

Unique files ratio	28.1%
Estimated attackers ratio	27.7%

Several attackers, after exploiting our honeypots, tried to make our servers join an IRC botnet by uploading dedicated PHP or Perl scripts.

Two of the honeypot virtual machines, and specifically those with the most severe vulnerabilities, allowing attackers to upload and run arbitrary files on the server, have been set up to allow outgoing connections to port 6667 (IRC). We

did so in order to monitor IRC botnet activity launched by an eventual attacker on our machines. We allowed connections only to port 6667, allowing thus only botnets running on the standard IRC port to connect to their management chat rooms. To avoid being tracked down by bot masters, every connection to the IRC port was tunneled through a privacy-protected VPN that anonymized our real IP address. No other outgoing connections were allowed from the machines, in order to avoid the possibility for our machines to launch attacks or scans against other hosts.

Our expectations proved to be correct, and we indeed logged several connections from our two machines to IRC command and control servers. The analysis of the packet traces showed some interesting information.

First of all, we were expecting IRC botnets to be quite rare nowadays, given the relatively high number of web-based exploit packs circulating on the black market. However, the analysis of the files that were uploaded on our honeypots showed an opposite trend, with about 200 distinct scripts launching IRC bots.

Another interesting observation is that, apparently, most of these IRC botnets are operated by young teenagers, as some IRC logs show. Some of the bot masters even put links to their Facebook or Twitter profiles in order to show off with their friends. Despite being run by youngsters, however, most of our connection logs show IRC rooms with hundreds to thousands of bots (the biggest IRC botnet we observed was comprised of 11900 bots).

While some logs showed us some of the bot masters attacking rivals on other IRC servers (which we considered a typical script-kiddie behavior), we were interested to see that these young people already deal with money and are able to use (and probably develop themselves) automated tools for searching on search engines and exploiting web vulnerabilities. We received a number of commands to perform DoS attacks, search engines scans using dorks, automatic mass exploitations, and instructions to report back usernames and passwords, as well as credit card credentials, stolen from exploited websites.

A final interesting finding, supported by the language used in the IRC logs and by an analysis of the IP addresses used for the upload of the IRC script, was that the majority of these IRC botnets were installed by users from South-Eastern asian countries (mostly Malaysia and Indonesia).

6.8 Phishing

Unique files ratio	7.3%
Estimated attackers ratio	6.3%

Phishing is one of the most dangerous activities that online criminals perform nowadays. We found proof of many attempts to install phishing pages or phishing kits on our honeypots. This kind of activity is always profit-driven; the vast majority of phishing websites are replicas of online banking websites, but we also collected few examples of online email portal phishing and even a handful of web pages mimicking ISPs and airline companies' websites.

During the 100 days of operation, our honeypots collected a total of 470 phishing-related files, 129 of which were complete phishing packages (archives often containing a full phishing website installation, including images, CSS files, and the phishing scripts themselves). Susprisingly, Nigeria seems to be a very active country for this kind of attacks, with Nigerian IP addresses responsible for approximately 45% of the phishing attacks logged by our honeypots.

An interesting case was logged by our honeypots starting on March 27th. Analyzing the Referer header of the requests received by our websites, we found 4776 requests, from 1762 different IP addresses, reaching our pages with the referer set to the mail servers of `sfr.fr`, one of the major French ISPs. Inspecting the webserver logs, we found out that all the HTTP requests having a Referer from sfr.fr requested only two png images. Both files had been uploaded to our honeypots on the 24th of March; when the first hit from SFR arrived, the virtual machines had already been cleaned up several times, but we found the original version of the pictures in our snapshots of uploaded files. Surprisingly, the pictures showed a message resembling a regular communication from SFR's customer service. All the users that hit our honeypots with a Referer from sfr.fr had thus received a phishing email containing links to the two png files, and their web client was only trying to download and show them the contents of the email.

6.9 Spamming and message flooding

Unique files ratio	7.8%
Estimated attackers ratio	9.3%

Many users still seem to use spam as a technique to make profit on the Internet. Some of the scripts we found are indeed mailers, i.e., scripts used to send out spam to a large number of recipients in an automated way. Some other scripts were email or SMS flooders, that are instead used for launching DoS attacks.

Our honeypots collected around 600 such scripts. As an example, on February 21st, a script called *a1.php* was uploaded from a Nigerian IP address. This script is a highly customizable mailer, and allows sending spam to a list of recipients in plain text or HTML format, with many options. It can also be configured to log in to a remote SMTP server in order to send spam through an authenticated account, and to disconnect and reconnect to the server after a certain threshold of sent emails is reached, probably with the purpose of avoiding bans.

6.10 Link Farming & Black Hat SEO

Unique files ratio	2.7%
Estimated attackers ratio	1.0%

Link farms are groups of web sites linking to each other, usually creating web pages with a very dense link structure, whose aim is to boost the search engine ranking of the web sites of the group. Black-hat SEO, instead, refers to using illicit or unethical techniques, such as cloaking, to boost the search engine ranking of a website, or to manipulate the way in which search engines and their spiders see and categorize a web pages. If we exclude automated posts on the forum web application, where a high percentage of posts contained links to link farming networks, this kind of behavior has not been observed very frequently on our honeypots.

An interesting attack that created a big amount of web pages on our honeypots was launched on March 19th. Somebody installed an fully functional CMS, comprising hundreds of static html pages, to one of our honeypots. All the generated pages were installed on the *images/rf/* subdirectory of our e-commerce web application, and contained russian text, along with images, CSS and JavaScript files used for presentation purposes. This page structure seems to be generated through a blog or CMS creation engine, as all the pages have a very dense link structure and point one another using absolute links (that had been customized and contained our honeypot website's domain name). We expect this to be part of an attempt to create a link farming network, or simply to be a marketing campaign for some counterfeit goods, as most of the pages we analyzed were actually advertising the sale of replica watches.

Finally, on a smaller scale, we also saw some attackers creating pages with ads or inserting links to partner sites on their uploaded pages. The reason for this is still making profit out of ads, or improving their or their partners' ranking on search engines.

6.11 Proxying and traffic redirection

Unique files ratio	0.6%
Estimated attackers ratio	0.6%

Online criminals always look for reliable ways to hide their tracks, and as time goes by, it becomes more and more difficult to rely only on open proxy networks, the TOR network, or open redirection web pages to conduct malicious activities. In fact, these services are often overloaded with (malicious) traffic and as such have very bad average performances and are very likely to be monitored by the authorities. In this scenario, the possibility of tunneling traffic on infected hosts seems idyllic, as it is quite easy to turn a webserver into a proxy, and often webservers running on hosting providers premises have high bandwidths, making them a very valuable target. We saw some attackers uploading proxy scripts or traffic redirection systems (TDS) to our honeypots, for the purpose of redirecting traffic anonymously (proxies) or redirecting users to malicious sources or affiliate websites (TDSs).

As an example, an archive of 504KB was uploaded on one of our honeypots on February 22, 2012. The archive contained a proxy tool called *VPSProxy*, publicly available at *http://wonted.ru/programms/vpsproxy/*; it is a PHP proxy fully controllable through a GUI client. Apparently, among all its features, if installed on more than one server, the tool makes it easy for the person using it to bounce between different connections. We believe tools like this can be very useful to criminals trying to hide their traces on the Internet.

6.12 Custom attacks

Unique files ratio	1.9%
Estimated attackers ratio	2.6%

This category groups all attacks that were either built on purpose for exploiting specific services, or that had no other matching category. For example, attacks in this category include programs whose aim is to scan and exploit vulnerable web services running on the server, such as the *config.php* script that was uploaded to one of our websites on April the 9th. This PHP script presents a panel for finding and attacking 9 of the most known Content Management Systems: if any of these is found on the machine, the attacker can automatically tamper with its configuration. The tool also contained other scripts to launch local and remote exploits.

6.13 DOS & Bruteforcing tools

Unique files ratio	4.6%
Estimated attackers ratio	2.9%

This category includes programs that launch Denial of Service or bruteforce attacks against specific applications and services (e.g., bruteforcing tools for FTP or web services, UDP and TCP flooding scripts).

An interesting example of this kind of behavior was the email bruteforce script that was uploaded to one of our honeypots on April 7, 2012. An IP address from Azerbaijan used a web shell to upload a file called *n.php* and a wordlist containing 1508 words, called *word.txt*. The *n.php* file, once executed, uses the cURL PHP libraries to connect to the `box.az` email portal and the uses the wordlist to bruteforce the password for a specific username that was hardcoded in the program. Our honeypots actually logged the upload of *n.php* several times, to three different domains. The attacker tried multiple times to execute the script (10

times in 16 minutes) and to edit it (4 times) as if looking for an error in the code. In reality, the script traffic was simply blocked by our firewall.

7 Conclusions

In this paper we described the implementation and deployment of a honeypot network based on a number of real, vulnerable web applications. Using the collected data, we studied the behavior of the attackers before, during, and after they compromise their targets.

The results of our study provide interesting insights on the current state of exploitation behaviors on the web. On one side, we were able to confirm known trends for certain classes of attacks, such as the prevalence of eastern European countries in comment spamming activity, and the fact that many of the scam and phishing campaigns are still operated by criminals in African countries [12]. Pharmaceutical ads appear to be the most common subject among spam and comment spamming activities, as found by other recent studies [9].

On the other hand, we were also able to observe and study a large number of manual attacks, as well as many infections aimed at turning webservers into IRC bots. This suggests that some of the threats that are often considered outdated are actually still very popular (in particular between young criminals) and are still responsible for a large fraction of the attacks against vulnerable websites.

We are currently working toward a completely automated system that can monitor the honeypot in realtime, identify and categorize each attack, and update a dashboard with the most recent trends and exploitation goals.

8 Acknowledgements

The research leading to these results was partially funded from the EU Seventh Framework Programme (FP7/2007-2013) under grant agreement n°257007.

References

[1] IP Addresses of Search Engine Spiders. `http://www.iplists.com/`.

[2] Robots IP Address Ranges. `http://chceme.info/ips/`.

[3] Google Hack Honeypot. `http://ghh.sourceforge.net/`, 2005.

[4] Dshield web honeypot project. `https://sites.google.com/site/webhoneypotsite/`, 2009.

[5] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the USENIX Security Symposium*, 2011.

[6] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker. Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on*, 50(7):1545–1551, 2004.

[7] s. Commtouch. Compromised Websites: An Owner's Perspective. `http://stopbadware.org/pdfs/compromised-websites-an-owners-perspective.pdf`, february 2012.

[8] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.

[9] Cyberoam Technologies and Commtouch. Internet Threats Trend Report October 2012. `http://www.cyberoam.com/downloads/ThreatReports/Q32012InternetThreats.pdf`, october 2012.

[10] S. Esser. evalhook. `http://www.php-security.org/downloads/evalhook-0.1.tar.gz`, may 2010.

[11] M. Hofer and S. Hofer. ftp-deploy. `http://bitgarten.ch/projects/ftp-deploy/`, 2007.

[12] Imperva Inc. Imperva's Web Application Attack Report. `http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed2.pdf`, january 2012.

[13] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. deSEO: Combating Search-Result Poisoning. In *Proceedings of the USENIX Security Symposium*, 2011.

[14] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the International World Wide Web Conference (WWW)*, 2011.

[15] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement(0):91 – 97, 2006.

[16] C. Leita and M. Dacier. Sgnet: A worldwide deployable framework to support the analysis of malware threat models. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, may 2008.

[17] T. Moore and R. Clayton. Evil searching: Compromise and recompromise of internet hosts for phishing. In *Financial Cryptography*, pages 256–272, 2009.

[18] M. Müter, F. Freiling, T. Holz, and J. Matthews. A generic toolkit for converting web applications into high-interaction honeypots, 2007.

[19] V. Nicomette, M. Kaâniche, E. Alata, and M. Herrb. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in Computer Virology*, june 2010.

[20] F. Pouget, M. Dacier, and V. H. Pham. V.h.: Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. In *In: ECCE 2005, E-Crime and Computer Conference*, pages 29–30, 2005.

[21] N. Provos. A virtual honeypot framework. In *Proceedings of the USENIX Security Symposium*, pages 1–14, 2004.

[22] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFrames Point to Us. In *Proceedings of the USENIX Security Symposium*, 2008.

[23] D. Ramsbrock, R. Berthier, and M. Cukier. Profiling attacker behavior following ssh compromises. In *in Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

[24] L. Rist, S. Vetsch, M. Koßin, and M. Mauer. Glastopf. `http://honeynet.org/files/KYT-Glastopf-Final_v1.pdf`, november 2010.

[25] V. Roussev. Data fingerprinting with similarity digests. In K.-P. Chow and S. Shenoi, editors, *Advances in Digital Forensics VI*, volume 337 of *IFIP Advances in Information and Communication Technology*, pages 207–226. Springer Boston, 2010.

[26] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 117–128. ACM, 2009.

[27] zx2c4. Linux Local Privilege Escalation via SUID /proc/pid/mem Write. `http://blog.zx2c4.com/749`, january 2012.

12 Conclusion and Future Directions

12.1 The Past

The collection of papers I presented in this document were chosen not because they were the best, because they are my favorite, or because they are published in the most prestigious venues. Instead, they were chosen because I believe they provide some sort of "coverage" of web application security – at least regarding my work in this field.

These articles cover a variety of topics, ranging from black to white-box analysis, from large Internet measurements to small code analysis experiments, and from classic to new types of web vulnerabilities. They also include research for which I was the main developer and the first author, and research that I later supervised as a professor. Despite being so different, they all fit together as little pieces of a big puzzle.

By looking at the general picture that emerges from this puzzle, I can summarize my main contributions to the field of web application security (unfortunately only partially presented in this manuscript) around the following points:

- A new approach to study input validation routines not as a binary problem (sanitized vs not sanitized) but my measuring the quality and the limitations of the code responsible to perform the sanitization.

- A new line of research that goes beyond traditional code injection and input validation vulnerabilities and tackle more subtle bugs in the logic of a web application. I approached the problem of automatically detecting logic vulnerabilities both by looking at the source code and by inferring the application model from a black box communication. I also proposed a detection

methodology, based on state invariants, to protect PHP applications against this class of attacks [1].

- A number of practical tools and services to measure the prevalence of web attacks and vulnerabilities on the Internet [2][3][4]. In particular, over the past ten years, my research resulted in many new CVEs, many discussions with developers and security teams of large organizations, and overall in hundreds of problems fixed in very popular services. This shows that my research had a tangible impact on the security of many real web applications that are used every day by millions of users.

- The use of specially designed honeypots to capture and study the behavior of the attackers on compromised web applications. Now operating for more than three years, this project allowed us to collect a large amount of data that we can use to study the evolution and motivations behind online crimes.

12.2 The Future

Forecasting the future in system security is often a futile exercise. Everything evolves very fast, starting from the technologies and protocols, all the way to the threat model and the motivations of the attackers. The challenge in the system security field is that there is always a large number of *severe* and *urgent* problems that need a solution – and no one can tell what these problems will be in five years from now. This motivate people to look only as far as the next paper, and focus their forecast only to identify emerging topics.

Despite this general "unpredictability", researchers strive to find some order in this chaos and try to identify clear directions that, abstracting from individual problems, can push the community towards a measurable improvement of the security of both system and services.

In the following, I present three ideas I currently start to investigate and that I hope will keep me busy for the next few years.

- In the first idea, I would like to put aside for a moment the existing concept of "vulnerability" and look more broadly at popular features that, if not

[1] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, Giovanni Vigna "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications" – *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*

[2]L. Bruno, M. Graziano, D. Balzarotti, A. Francillon "Through the Looking-Glass, and What Eve Found There" *USENIX Workshop on Offensive Technologies (WOOT) 2014*

[3]A. Kharraz, E. Kirda, W. Robertson, D. Balzarotti, A. Francillon "Optical Delusions: A Study of Malicious QR Codes in the Wild" *International Conference on Dependable Systems and Networks (DSN 2014*

[4]M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, C. Kruegel "A Solution for the Automated Detection of Clickjacking Attacks" *ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2010*

properly implemented, can be abused by an attacker to perform malicious actions. Examples of this kind that I discovered in the past are the friend finder functionality in online social networks [5], and the use of data compression in web protocols [6]. While these were isolated cases, I plan to work in this direction to generalize the concept and identify more examples. The goal is not only to discover problems that are often overlooked, but also to propose guidelines and security recommendations to help developers to implement security-relevant services.

- The abundance of web applications vulnerabilities is a known issue. While this is a very serious problem (one that I also discussed multiple times in this document) it is not the only reason that makes the Web vulnerable to so many unskilled attackers. The current approach and methodology used to secure web applications has been copied by what the security community developed for normal computers. But the Web works in a completely different way, starting from how the attackers reach their victims. Search engines and Google dorks allow criminals to pro-actively locate the vulnerable applications, making web vulnerabilities much easier to exploit than a vulnerability in a web browser or an office suite. This reduces the exploitation time and calls for a new approach to web security. Re-thinking the way we look at this problem requires a precise understanding of the role of all the components and actors involved in web attacks. This is part of my current research, but it will also affect my future work as I already started some long-lasting projects in this direction for the next few years.

- A third research direction I want to investigate is to study how to use the data collected by web application honeypots more effectively. Today, these systems are used to collect signatures of (mostly automated) attacks. This information can then be used as a source of intelligence or as a way to improve the detection of web application firewalls. However, this is only scratching the surface of what we can do in this area. For instance, by studying the tools and the components used by attackers on compromised machine, we can build models to detect similar signs of compromise - also used in different context and against different web applications. I believe that modeling the attackers, and not jut the attacks, can be an interesting future direction.

[5]Marco Balduzzi, Christian Platzer, Thorsten Holz, Engin Kirda, Davide Balzarotti, Christopher Kruegel "busing Social Networks for Automated User Profiling" *Symposium on Recent Advances in Intrusion Detection (RAID) 2010*

[6] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, Neeraj Suri "In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services" *24rd USENIX Security Symposium 2014*