Program Analyses of Web Applications for Detecting Application-Specific Vulnerabilities

By

Fangqi Sun

B.S. (Wuhan University) 2005
M.S. (Wuhan University) 2007

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Shyhtsun Felix Wu

Hao Chen

Committee in Charge

2013

Program Analyses of Web Applications for Detecting Application-Specific Vulnerabilities

© Fangqi Sun, 2013. All rights reserved.

Dedication

To my parents, for their constant love and support.

Contents

List of Figures vi

List of Tables vii

1 Introduction 1

1.1 Overview of Web Application Security . . . 1
1.2 Web Application Vulnerabilities . . . 2
1.2.1 XSS Worms . . . 3
1.2.2 Access Control Vulnerabilities . . . 4
1.2.3 Logic Vulnerabilities in E-commerce Applications . . . 4
1.3 Thesis Structure . . . 5

2 Client-Side Detection of XSS Worms by Monitoring Payload Propagation 7

2.1 Introduction . . . 7
2.2 Our Approach . . . 10
2.2.1 Background of XSS Worm Propagation . . . 10
2.2.2 High-level Overview . . . 11
2.2.3 Approach Details . . . 12
2.3 Implementation . . . 15
2.4 Empirical Evaluation . . . 17
2.4.1 Real-World XSS Worms . . . 17
2.4.2 Effectiveness of Our Approach for Obfuscated JavaScript Code . . . 20
2.4.3 Overhead Measurements . . . 21
2.4.4 Parameter Settings . . . 23
2.4.5 Discussion . . . 23
2.5 Related Work . . . 24
2.5.1 Worm Detection . . . 24
2.5.2 Client-Side Protection . . . 25

2.5.3 Server-Side Analysis 25
2.6 Conclusion 26

3 Static Detection of Access Control Vulnerabilities in Web Applications 27
3.1 Introduction 28
3.2 Illustrative Example 31
3.3 Approach Formulation 32
3.4 Analysis Algorithm 34
3.4.1 Vulnerability Detection 35
3.4.2 Building Sitemaps 36
3.4.3 Link Extraction 37
3.5 Implementation 41
3.5.1 Specification Rules 42
3.5.2 Sitemap Builder 42
3.5.3 Vulnerability Detector 47
3.6 Empirical Evaluation 47
3.6.1 Analysis Results 48
3.6.2 Performance Evaluation 53
3.6.3 Discussions 55
3.7 Related Work 56
3.8 Conclusion 58

4 Detecting Logic Vulnerabilities in E-commerce Applications 59
4.1 Introduction 60
4.2 Illustrative Example 64
4.3 Approach 67
4.3.1 Definitions 67
4.3.2 Automated Analysis 70
4.4 Implementation 74
4.4.1 Symbolic Execution 76
4.4.2 Path Exploration 77
4.4.3 Logic Flows 79
4.5 Empirical Evaluation 80
4.5.1 Analysis Results 82
4.5.2 Experiments on Live Websites 86
4.5.3 Performance Evaluation 87
4.5.4 Discussions 90

4.6 Related Work 91
4.7 Conclusion 93

5 Conclusion **94**
5.1 Summary 94
5.2 Future Work 95

Bibliography **98**

List of Figures

2.1 Client architecture for XSS worm detection. 11
2.2 The interface collaboration diagram. 16
2.3 Similarity results for common JavaScript obfuscators. 21

3.1 An Example of Access Control Vulnerability. Solid arrows represent explicit links, and dashed arrows represent inclusion relationship between pages. Arrows correspond to edges in sitemaps and are labeled with roles. The intended sitemap for privileged role *a* has four edges while the intended sitemap for role *b* has only one edge. 31
3.2 Algorithm for Vulnerability Detection. 35
3.3 Algorithm for Building Sitemaps. 37
3.4 Algorithm for Link Extraction. 39
3.5 System Architecture. 41
3.6 An Example of Path Exploration. 44
3.7 A Deterministic Finite Automaton Example. 45

4.1 Logic Flows in E-Commerce Web Applications. 61
4.2 Received Products from Vulnerable Websites. 62
4.3 Algorithm for Vulnerability Detection. 71
4.4 Symbolic Execution Framework. 75
4.5 Example for Path Exploration. 78

List of Tables

2.1 Statistics of XSS worms in the wild. 18
2.2 Performance overhead for top 20 websites. 22

3.1 Statistics on Evaluation Subjects. 48
3.2 Vulnerability Analysis Results. 49
3.3 Coverage and Performance Results. 53
3.4 Analysis Time. 54

4.1 Payment Modules for Cashiers. 81
4.2 Logic Vulnerability Analysis Results. 82
4.3 Performance Results. 88

Abstract

Web applications are prevalent in the modern era, regulating access to sensitive information, functionality and resources. Due to the difficulty in designing and implementing proper security checks for untrusted user inputs and actions, web applications often fall victim to various online attacks. In particular, application-specific vulnerabilities are easy to exploit and often have severe consequences. While the detection of application-independent injection vulnerabilities, such as Cross-Site Scripting (XSS) and SQL injection vulnerabilities, examines individual web pages in isolation, the detection of application-specific vulnerabilities necessitates the examination of both logic flows among web pages and user inputs of each page. The detection of application-specific vulnerabilities is more challenging because such vulnerabilities vary across different applications and designing general detection rules for them is difficult. Since manual code review is error-prone and time-consuming, it is important to develop automated detection techniques.

This dissertation presents novel, practical program analyses to detect web application vulnerabilities, especially application-specific ones. It begins by providing the first pure client-side solution to detect XSS worms which exploit XSS vulnerabilities. Unlike XSS worms, logic vulnerabilities are application-specific. Two important types of logic vulnerabilities are access control vulnerabilities in web applications and logic vulnerabilities in e-commerce applications. This dissertation formulates their respective core characteristics and introduces corresponding server-side techniques to detect them. Specifically, for access control vulnerabilities, it describes the first static analysis that infers and enforces implicit access control assumptions, and for logic vulnerabilities in e-commerce applications, it presents the first static detection of logic attacks that cause incorrect payment status. In addition, this dissertation presents practical algorithms and tools which are shown to be scalable and capable of finding unknown vulnerabilities in real-world web applications.

Acknowledgments

This dissertation would not exist without the assistance, advice and encouragement from many people. I would like to thank my family, friends and coworkers from the bottom of my heart.

I thank my doctoral advisor Prof. Zhendong Su for his invaluable advice and support. He set an excellent example of dreaming big, working hard and never giving up. Prof. Su inspired me to conduct top-notch research and taught me to view difficulties as opportunities rather than hurdles on the road. He did his best to help me and I am indebted to him for his wonderful guidance along the way.

I also thank the other members of my committee, Prof. Hao Chen and Prof. Felix Wu. The upbeat and insightful lectures of Prof. Chen introduced me to influential research in computer security. I also learned a lot from Prof. Wu's carefully prepared lectures on operating systems and enjoyed modifying a Linux kernel for the course project.

I am deeply grateful to my father for his unconditional love and support. He sparked my initial interests in science and technology in my upbringing. He has provided me with counseling and honest feedback. I would not be where I am without his encouragement.

I would like to thank also my labmates in Kemper 2249: Mehrdad Afshari, Earl Barr, Mark Gabel, Zhongxian Gu, Dara Hazeghi, Lingxiao Jiang, Taeho Kwon, Vu Minh Le, Andreas Sæbjørnsen, Martin Velez, Thanh Vo, Ke Wang and Gary Wassermann. They made working in the lab a fun experience. I appreciate their constructive and detailed feedback on research ideas, paper drafts and practice talks.

Special thanks to my partner both in life and in research, Liang Xu, for his constant love and support. It has been great to share all the joy, excitement and trying times with him.

Chapter 1

Introduction

Web applications have become increasingly pervasive and so have web-based attacks. While bringing a degree of interoperability and integration of data never before dreamed of, few web applications are immune from being compromised. The sheer size and complexity of web applications leave open a large attack surface. Additionally, the sophistication of application-specific vulnerabilities increases the difficulty of implementing perfect lines of defense. Consequently, the security of web applications is of vital importance.

1.1 Overview of Web Application Security

Web applications serve a wide range of purposes, from social networking, sensitive data management to online shopping. The advent of smart mobile devices enables users to access web applications even on the go. A recent Symantec Internet security threat report states that the number of reported web vulnerabilities increased from 4,989 in 2011 to 5,291 in 2012, incurring an average cost of $591,780 for businesses [68]. Furthermore, web vulnerabilities are widespread. A WhiteHat security statistics report published in 2013 shows that 86% of all the surveyed websites had at least one serious vulnerability during 2012, and the number of serious vulnerabilities per website was 56 on average [82].

Recent years have witnessed the rapid growth of web applications and their code complexity. Web applications have evolved from sets of simple, static web pages, to feature-rich Web 2.0 applications

which frequently integrate third-party services. Unfortunately, the more features that a modern web application possesses, the larger attack surface it often exposes to attackers. With the incorporation of client-side JavaScript code, attackers can inject malicious JavaScript code in user inputs and exploit various ways that web browsers invoke their JavaScript engines. In cases where web applications behave differently to administrators and normal users, attackers can break authentication processes and access sensitive information and functionality. In recent years, the integration of third-party services has become popular, giving attackers a new opportunity to exploit miscommunications between servers and service providers. The best practice for web security is to build an application with security in mind end-to-end. However, many web developers are unfamiliar with various attack vectors, especially application-specific ones.

Application-specific vulnerabilities in web applications are especially challenging to detect. When building an application, developers often have a clear picture of what the ideal application should be in their minds. Unfortunately, in practice, the implemented application often does more than what is intended. Put it another way, unexpected user inputs and logic flows can allow attackers to abuse implemented but insufficiently guarded application-specific functionality in dangerous ways. The uniqueness and complexity of logic flows complicate the establishment of a general line of defense against application-specific attacks.

This dissertation focuses on detecting crucial, ungrounded assumptions that web developers make about user inputs and logic flows in web applications. While existing solutions mainly focus on vulnerabilities related to untrusted user inputs, the key observation of this dissertation is that it is also important to validate critical assumptions on logic flows which are application-specific. Less traveled paths in web applications may indeed lead to fruitful attacks. Inferring and enforcing such implicit assumptions on logic flows are vital yet challenging for web application security.

1.2 Web Application Vulnerabilities

This section describes three distinct types of web application vulnerabilities: XSS worm, access control vulnerabilities in web applications and logic vulnerabilities in e-commerce web applications. While

injection vulnerabilities are the most common, their impact is not always significant. In addition, injection vulnerabilities have been extensively studied over the past few years, and there already exist clear understandings of their essence. In contrast, application-specific vulnerabilities that this dissertation focuses on often have serious impact, and their uniqueness poses significant challenges for automated detection.

1.2.1 XSS Worms

An XSS worm is a piece of self-replicating JavaScript code that exploits an XSS vulnerability to propagate itself within one web application. XSS worms often share two common characteristics. First, to avoid unwanted attention from users, they often spread silently using Asynchronous JavaScript and XML (AJAX) and only process HTTP requests and responses in the background. Second, to avoid detection, they often use encoding or obfuscation techniques.

As an example, the following code snippet shows how the Samy worm [39], a worm that affected more than one million MySpace users in less than 20 hours, avoided detection and invoked JavaScript engines. Notice that there is a newline character between strings `java` and `script` of the background URL.

```
<DIV id=mycode style="BACKGROUND: url('java
script:eval(document.all.mycode.expr)')" expr="..."></DIV>
```

The Samy worm exploited an XSS vulnerability caused by the extra newline character. The newline character helped the Samy worm avoid keyword-based detection but still invoke JavaScript engines. In other words, MySpace servers viewed the worm as a benign user input, while browsers interpreted the worm as legitimate JavaScript code. The payload of the worm, which is written in the `expr` property of a `DIV` tag, replicates itself and writes “Samy is my hero” in infected users’ profiles. The connections among MySpace users served as channels for worm propagation.

1.2.2 Access Control Vulnerabilities

Access control vulnerabilities are the culprit of privilege escalation attacks which expose sensitive information or operations. Often in violation of developers' intentions, attackers predict and directly access sensitive, hidden URLs for privileged information or functionality which are usually application-specific. The root cause of access control vulnerabilities is that developers often implicitly assume that attackers will not access hidden URLs. However, security by obscurity offers insufficient and unreliable protection.

For example, Bloomberg leaked unpublished earnings of NetApp in November, 2010 [58]. The leaked earnings report was obtained via the first URL shown below. The second and third URLs are two examples of previous report URLs.

```
http://media.netapp.com/documents/financial-fy11-q2.pdf (leaked)
http://media.netapp.com/documents/financial-q1-fy11.pdf
http://media.netapp.com/documents/financial-10-q4.pdf
```

As can be seen from the URLs, the unpublished URL is highly predictable. The administrator of NetApp assumed that unprivileged users would not access that URL. However, Bloomberg predicted the first sensitive URL based on previous URLs. The breach led to arguments between the two companies. Bloomberg claimed that the file was posted without any required password, but NetApp stated that the file was obtained from a restricted area of its website. This example illustrates the importance of explicitly protecting privileged accesses.

1.2.3 Logic Vulnerabilities in E-commerce Applications

Logic vulnerabilities in e-commerce web applications refer to missing or insufficient checks that can be exploited to cause inconsistent payment status between third-party cashiers and merchants during checkout. By using unexpected application-specific logic flows, an attacker can lead a merchant to mistakenly believe that an order has been paid correctly while the designated cashier actually has received nothing or only an insufficient payment. Once exploited, such vulnerabilities often result in serious consequences, including financial loss and merchant embarrassment.

Take for example an unknown logic vulnerability that we detected in an open source application osCommerce [1]. Suppose a user places two orders with IDs of 1001 and 1002 on a merchant's server. For either order, the designated cashier is assumed to generate a secret `MD5` value and visit a URL based on page `checkout_process.php` with the secret value only when a full payment has been made to the cashier. The secret values of orders 1001 and 1002 should be different and unpredictable. The URLs for the two orders are shown in the following.

```
http://merchant.com/checkout_process.php?orderID=1001&LKMAC=SecretMD5For1001
http://merchant.com/checkout_process.php?orderID=1002&LKMAC=SecretMD5For1002
```

The key problem for the detected logic vulnerability is that this e-commerce application does not check the request parameter values of `orderID` and `LKMAC` against trusted order ID and `MD5` value. Consequently, an attacker can skip payment and jump directly to a forged merchant URL. The attacker can use the request parameter values of order 1001 for order 1002 to avoid payment. This substitution leads the merchant to mistakenly believe that order 1002 has been paid, while the cashier actually has received nothing on behalf of the merchant for this order. Essentially, this detected vulnerability allows attackers to pay once and bypass payment for future orders.

1.3 Thesis Structure

This dissertation addresses several important vulnerabilities in web applications. It presents runtime protection against XSS worms and static analyses for application-specific vulnerabilities. The rest of this dissertation is organized as follows.

Chapter 2 An XSS worm tends to spread rapidly, sometimes even exponentially, within a web application. Based on the key observation that an XSS worm must spread from one user to another by reconstructing and propagating its payload, Chapter 2 presents the first client-side approach to detect and stop the propagation of XSS worms early on. This runtime approach identifies XSS worms by monitoring outgoing requests that send self-replicating payloads. A cross-platform Firefox extension

is built to demonstrate the effectiveness and efficiency of this approach. The evaluation shows that the browser extension detects all known real-world client-side XSS worms, produces no observed false positives and exhibits low overhead when tested on popular websites.

Chapter 3 Automated detection of access control vulnerabilities is challenging because of the lack of a general characterization and specification for access control vulnerabilities. Specifications are the basis of automated detection. Manual specification is time-consuming and often absent, while probabilistic-based inference is imprecise and computationally expensive. The key observation is that source code of an application implicitly documents intended accesses of each role. This insight enables precise extractions of access control policies from differences between per-role sitemaps. Chapter 3 presents the first role-based static analysis which detects access control vulnerabilities with minimal manual effort. This approach automatically infers privileged pages based on constructed per-role sitemaps and performs forced browsing to detect vulnerabilities. The implemented tool detects both known and new access control vulnerabilities in web applications with few false positives, demonstrating its effectiveness and scalability.

Chapter 4 Logic vulnerabilities in e-commerce web applications have serious impact due to its direct tie to financial loss. The introduction of third-party cashiers increases the difficulty of designing and implementing secure checkout processes. A chain is as strong as its weakest link, therefore any loophole in a checkout process leaves attackers an opportunity for playing tricks to avoid or reduce payment. Chapter 4 presents a key invariant of successful payments in e-commerce applications: A secure payment occurs when critical components of payment status (order ID, order total, merchant ID and currency) are tamper-proof and authentic. This chapter also shows a symbolic execution framework that integrates taint analysis to analyze critical logic flows during checkout. The implementation scales to real-world payment modules and effectively detects logic vulnerabilities, the majority of which are new.

Finally, Chapter 5 summarizes this dissertation and outlines some future research directions.

Chapter 2

Client-Side Detection of XSS Worms by Monitoring Payload Propagation

Cross-site scripting (XSS) vulnerabilities make it possible for worms to spread quickly to a broad range of users on popular websites. To date, the detection of XSS worms has been largely unexplored. This chapter proposes the first purely client-side solution to detect XSS worms. Our insight is that an XSS worm must spread from one user to another by reconstructing and propagating its payload. Our approach prevents the propagation of XSS worms by monitoring outgoing requests that send self-replicating payloads. We intercept all HTTP requests on the client side and compare them with currently embedded scripts. We have implemented a cross-platform Firefox extension that is able to detect all existing self-replicating XSS worms that propagate on the client side. Our test results show that it incurs low performance overhead and reports no false positives when tested on popular websites.

2.1 Introduction

Web applications have drawn the attention of attackers due to their ubiquity and the fact that they regulate access to sensitive user information. To provide users with a better browsing experience, a number of interactive web applications take advantage of the JavaScript language. The support for JavaScript, however, provides a fertile ground for XSS attacks. According to a recent report from

OWASP [59], XSS vulnerabilities are the most prevalent vulnerabilities in web applications. They allow attackers to easily bypass the Same Origin Policy (SOP) [54] to steal victims' private information or act on behalf of the victims.

XSS vulnerabilities exist because of inappropriately validated user inputs. Mitigating all possible XSS attacks is infeasible due to the size and complexity of modern web applications and the various ways that browsers invoke their JavaScript engines. Generally speaking, there are two types of XSS vulnerabilities. *Non-persistent XSS vulnerabilities*, also known as reflected XSS vulnerabilities, exist when user-provided data are dynamically included in pages immediately generated by web servers; *persistent XSS vulnerabilities*, also referred to as stored XSS vulnerabilities, exist when insufficiently validated user inputs are persistently stored on the server side and later displayed in dynamically generated web pages for others to read. Persistent XSS vulnerabilities allow more powerful attacks than non-persistent XSS vulnerabilities as attackers do not need to trick users into clicking specially crafted links. The emergence of *XSS worms* worsens this situation since XSS worms can raise the influence level of persistent XSS attacks in community-driven web applications. XSS worms are special cases of XSS attacks in that they replicate themselves to propagate, just like traditional worms do. Different from traditional XSS attacks, XSS worms can collect sensitive information from a greater number of users within a shorter period of time because of their self-propagating nature.

The threats that come from XSS worms are on the rise as attackers are switching their attention to major websites, especially social networking sites, to attack a broad user base [67]. Connections among different users within web applications provide channels for worm propagation. In community-driven web applications, XSS worms tend to spread rapidly — sometimes exponentially. For example, the first well-known XSS worm, the Samy worm [39], affected more than one million MySpace users in less than 20 hours in October 2005. MySpace, which had over 32 million users at that time, was forced to shut down to stop the worm from further propagation. In April 2009, during the outbreak of the StalkDaily XSS worm which hit twitter.com, users became infected when they simply viewed the infected profiles of other users. We show a list of XSS worms in Table 2.1 (Section 2.4.1). Common playgrounds for XSS worms include social networking sites, forums, blogs, and web-based email services.

At present, although much research has been done to detect either traditional worms or XSS

vulnerabilities, little research has been done to detect XSS *worms*. This is because XSS worms usually contain site-specific code which evades input filters. XSS worms can stealthily infect user accounts by sending asynchronous HTTP requests on behalf of users using the Asynchronous JavaScript and XML (AJAX) technology. Spectator [48] is the first JavaScript worm detection solution. It works by monitoring worm propagation traffic between browsers and a specific web application. However, it can only detect JavaScript worms that have propagated far enough and is unable to stop an XSS worm in its initial stage. In addition, Spectator requires server cooperation and is not easily deployable.

In this chapter, we present the first purely client-side solution to detect the propagation of self-replicating XSS worms. Clients are protected against XSS worms on all web applications. We detect worm payload propagation on the client side by performing a string-based similarity calculation. We compare outgoing requests with scripts that are embedded in the currently loaded web page. Our approach is similar in spirit to traditional worm detection techniques that are based on payload propagation monitoring [75, 76]; we have developed the first effective client-side solution to detect *XSS worms*.

This chapter makes the following contributions:

- We propose the first client-side solution to detect XSS worms. Our approach is able to detect self-replicating XSS worms in a timely manner, at the very early stage of their propagation.
- We have developed a cross-platform Firefox extension that is able to detect all existing XSS worms that propagate on the client side.
- We evaluated our extension on the top 100 most visited websites in the United States [4]. Our results demonstrate that our extension produces no false positives and incurs low performance overhead, making it practical for everyday use.

The rest of the chapter is organized as follows. Section 2.2 describes our worm detection approach in detail. Section 2.3 presents the implementation details of our Firefox extension. Section 2.4 evaluates the effectiveness of our approach and measures the performance overhead of our Firefox extension on popular websites. Finally, we survey related work (Section 2.5) and conclude (Section 2.6).

2.2 Our Approach

Section 2.2.1 describes the background of how an XSS worm usually propagates. Section 2.2.2 gives an overview of how we detect the propagation behavior of an XSS worm. We describe the details of our detection routine in Section 2.2.3.

2.2.1 Background of XSS Worm Propagation

The Document Object Model (DOM) is a cross-platform and language-independent interface for valid HTML and well-formed XML documents [73]. It closely resembles the tree-like logical structure of the document it models. Every node in a DOM tree represents an object in the corresponding document. With the DOM, programs and scripts can dynamically access and update the content, structure and style of documents.

In practice, building a flawless web application is an extremely difficult task due to the challenges of sufficiently sanitizing user-supplied data. Site-specific XSS vulnerabilities become a growing concern as attackers discover that they can compromise more users by exploiting a single vulnerability within a popular web site than by compromising numerous small websites [67]. Exploiting persistent XSS vulnerabilities, XSS worms are normally stored on the servers of vulnerable web applications. The typical infection process of an XSS worm is as follows:

1. A user, Alice, is lured into viewing a malicious web page that is dynamically generated by a compromised web application. The web page, for example, can be the profile page of Alice's trusted friend.

2. The XSS worm payload, which is embedded in the dynamically generated web page, is interpreted by a JavaScript engine on the client side. During this interpretation process, an XSS worm usually replicates its payload and injects the replicated payload into an outgoing HTTP request.

3. The crafted malicious HTTP request is sent to the web application server on Alice's behalf. By exploiting the server's trust in Alice, the XSS worm also compromises Alice's account. Later on, when Alice's friends visit her profile, their accounts will also be infected.

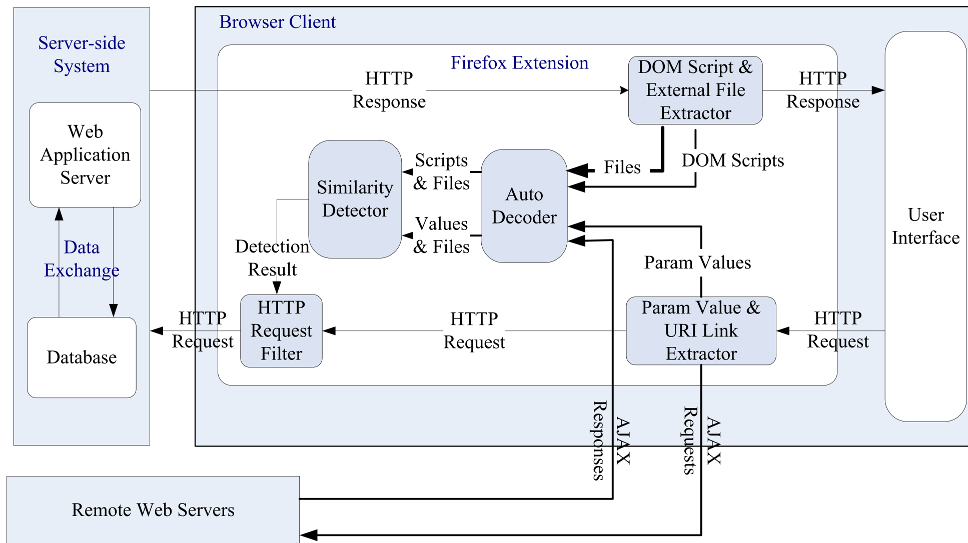

Figure 2.1: Client architecture for XSS worm detection.

2.2.2 High-level Overview

Our goal is to detect the self-replicating characteristics of XSS worms with no modification to existing web applications or browser architecture. To this end, we compare outgoing HTTP requests with scripts in the currently loaded DOM tree. Figure 2.1 shows the architecture of our client-side XSS worm detection mechanism. Our solution captures the essential self-propagating characteristics of XSS worms and protects users from infection. We choose to detect XSS worms on the client side because the propagation process of an XSS worm is normally triggered during the script interpretation process on the client side. Moreover, a client-side solution is easily deployable.

Scripts can be directly embedded in web pages or dynamically loaded from remote servers. Therefore, it is necessary to examine all external JavaScript files that are pointed to by Uniform Resource Identifier (URI) links in both outgoing HTTP requests and loaded DOM trees. The steps we take to detect an XSS worm are as follows:

1. We intercept each outgoing HTTP request that may contain the payload of an XSS worm. We extract parameter values from each intercepted request. From these parameter values, we then extract URI links which may point to malicious JavaScript files.

2. If there exist embedded URI links, we send asynchronous requests to retrieve external JavaScript files according to the extracted URI links. We do not begin our decoding process (Step 4) until we receive all responses or signals of timeout events from remote servers. We call the set of parameter values and retrieved external files set $\mathscr{P}$.

3. We extract scripts from the DOM tree of the current web page. Next, we retrieve external JavaScript files, which are dynamically loaded into the current web page, from cached HTTP responses. We call the set of extracted scripts and cached external files set $\mathscr{D}$.

4. We apply an automatic decoder on code from both set $\mathscr{P}$ and set $\mathscr{D}$. We repeat this decoding process until we find no encoded text.

5. Finally, we use a similarity detector to compare decoded code from set $\mathscr{P}$ with decoded code from set $\mathscr{D}$ in search of similar code, which indicates the potential propagation behavior of an XSS worm. If we detect suspiciously similar code, we redirect the malicious HTTP request and alert the user.

2.2.3 Approach Details

This section describes the details of our XSS worm detection algorithm: how we extract parameter values and URI links; possible locations where scripts might appear in web pages; our decoder, which can handle a number of encoding schemes; and the string similarity detection algorithm that we use.

Parameter Values and URI Links from HTTP Requests

The payload of an XSS worm could be sent in the form of plaintext or as a URI link pointing to an external file stored on a remote server. In either case, the plaintext or the URI link needs to be embedded in the parameter values of an outgoing HTTP request in order to propagate. The extracted parameter values and retrieved external files compose set $\mathscr{P}$.

As it is impossible to tell whether an HTTP request is sent by an XSS worm or a legitimate user, we intercept and process each outgoing HTTP request. We first extract parameter values, if there are any,

from the `path` property of the requested URI. We then examine the request method of the outgoing HTTP request. If the `POST` method is used, we retrieve the request body and then extract additional parameter values from it.

Since attackers may store XSS worm payloads on remote servers and propagate URI links instead of plaintext, we extract URI links from parameter values of HTTP requests using JavaScript regular expressions. We send requests to retrieve external files according to the URI links.

DOM Scripts and External Files from web pages

To enumerate possible locations where scripts may exist, we studied the source code of several XSS worms in the wild, the XSS Cheat Sheet [36], and some other documentation [73, 80]. We classify possible locations where scripts may reside into the following categories:

- *`script` elements.* A script can be defined within the `script` element of a DOM tree.
- *Event handlers.* W3C specifies eighteen intrinsic event handlers [73]. In addition, some browsers have implemented browser-specific event handlers.
- *HTML attributes.* Attackers sometimes exploit the attributes of standard DOM elements to dynamically load external files into a document.
- *Scripts specified by browser-specific attributes or tags.* Some browsers implement browser-specific attributes and tags.
- *javascript: URIs.* By declaring the JavaScript protocol, JavaScript code can be put into a place where a URI link is expected.

Based on the possible locations discussed above, we extract scripts directly embedded in the currently loaded DOM tree and retrieve cached external files pointed to by the attributes of DOM elements. Extracted scripts and cached external files compose set $\mathscr{D}$. If an XSS worm exists, its payload should be embedded in at least one of these locations in order to trigger script interpretation.

Automatic Decoder

Taking into consideration that the payload of an XSS worm may be encoded, we perform a decoding process before carrying out our similarity detection process. We decode all extracted parameter values, retrieved external JavaScript files, extracted DOM scripts, and cached external JavaScript files. In order to automate the decoding process, we use a regular expression for each encoding scheme.

It is possible that a JavaScript obfuscator applies multiple layers of encoding routines. To handle such situations, we keep track of the total match count for all regular expressions in each decoding routine. If the total match count in a decoding routine reaches zero, it means that no encoded text is found. For this case, we stop our decoding routine; otherwise, we repeat the decoding routine.

Similarity Detection

We detect both suspicious URI links and similar strings.

An XSS worm may propagate by sending a URI link pointing to itself instead of directly sending its payload as plaintext. Therefore, before comparing the elements from set $\mathscr{P}$ with the elements from set $\mathscr{D}$, we compare the URI links extracted from the parameter values of an outgoing HTTP request with the URI links embedded in the current DOM tree. If a match is found, we immediately redirect the current request and alert the user of the suspicious URI link; otherwise, taking into account the possibility that attackers may use different URI links for the same payload, we examine the contents of external files. Although we have not seen such attacks in real-world XSS worms, we conservatively examine file contents. We do not begin our URI detection process until the decoding process completes.

Once we have all the decoded code, we perform a similarity detection routine in search of a possible XSS worm payload. We use a string similarity detection algorithm based on trigrams [3] for its robustness in dealing with some JavaScript obfuscation techniques such as code shuffling and code nesting.

In our implementation, we use character-level trigrams, which are three character substrings of given strings, to detect the similarity between two strings. Note that a trigram is a special case of an n-gram where $n = 3$. We introduce formal definitions of the trigram algorithm as follows.

Definition 2.1 $\mathscr{T}(s)$ denotes the set of character-level trigrams of string s.

Definition 2.2 $\mathscr{S}(p,d)$ denotes the similarity between two strings p and d, where $p \in \mathscr{P}$, $d \in \mathscr{D}$, and both $\mathscr{P}$ and $\mathscr{D}$ are sets of strings.

We compute the similarity of p and d in the following way:

$$\mathscr{S}(p,d) = \frac{\left|\mathscr{T}(p) \cap \mathscr{T}(d)\right|}{\left|\mathscr{T}(p) \cup \mathscr{T}(d)\right|} \tag{2.1}$$

In our current settings, $\mathscr{P}$ is the set of parameter values and contents of retrieved external files, and $\mathscr{D}$ is the set of scripts extracted from the DOM tree and contents of cached external files. $\mathscr{S}(p,d)$ has a value between 0 and 1, inclusive.

To make our implementation scalable, we sort the elements in each set into ascending order before calculating their union or intersection. The average-case time complexity of the trigram algorithm is determined by the time complexity of the sorting algorithm, which is $\mathscr{O}(n \log n)$.

Definition 2.3 If $\exists\ p \in \mathscr{P}$, $d \in \mathscr{D}$ such that $\mathscr{S}(p,\ d)$ **exceeds** a customized threshold t, we say that an outgoing HTTP request may contain an XSS worm payload because it exhibits self-propagating behavior.

2.3 Implementation

We have developed a standard cross-platform Firefox 3.0 extension to detect XSS worms. Most Firefox extensions are written in JavaScript because the bindings between JavaScript and XPCOM are strong and well-defined. We wrote most components of our extension in JavaScript, except that we implemented the trigram algorithm using C++ for its better execution efficiency over JavaScript.

We show key user interfaces that we have used in our extension in Figure 2.2. This diagram is taken from Mozilla Cross-Reference to show the collaboration among the interfaces.

nsIObserverService. To monitor parameter values in each HTTP request, we first get a service from the XPCOM component `observer-service` through the interface `nsIObserverService`. We then

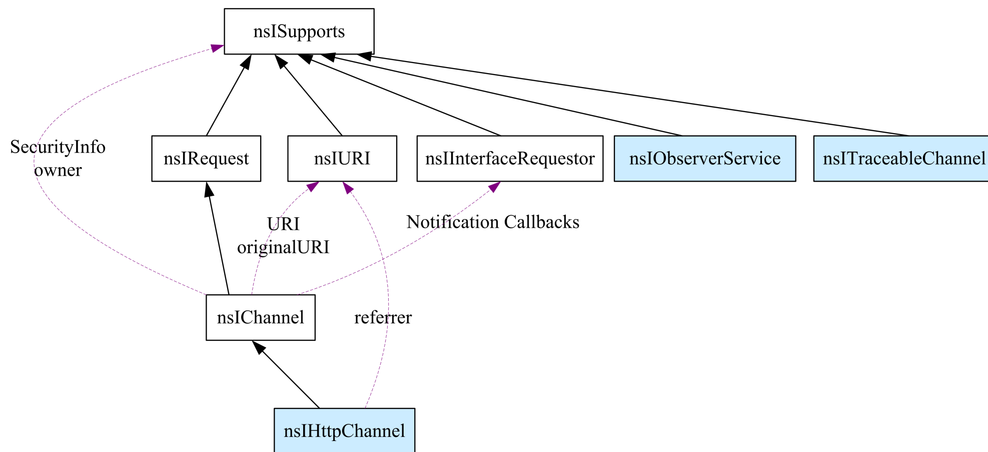

Figure 2.2: The interface collaboration diagram.

register an `http-on-modify-request` observer and an `http-on-examine-response` observer in our extension with the observer service we just obtained.

nsIHttpChannel. Through an `nsIHttpChannel` object, we can obtain an `nsIURI` object to read its `asciiSpec` property for an ASCII representation of the requested URI. To get the body of a POST request from an `nsIHttpChannel` object, we gain access to the post data by obtaining a pointer to the `nsIUploadChannel` interface, and then rewind the stream of the post data with a pointer to the `nsISeekableStream` interface. If an XSS worm is detected, we obtain a pointer to the `nsIRequest` interface, and then call the cancel method in that interface to cancel the malicious HTTP request.

nsITraceableChannel. The `nsITraceableChannel` interface enables us to directly retrieve external files from cached HTTP responses rather than sending asynchronous requests and waiting for their responses. This interface was recently introduced in Firefox 3.0.4, which was released in November 2008. Through this interface, we can replace a channel's original listener with a new one, and collect all the data we need by intercepting `OnDataAvailable` calls. We examine the Multipurpose Internet Mail Extensions (MIME) types of all HTTP responses to determine which responses might trigger script interpretation.

2.4 Empirical Evaluation

This section shows the effectiveness of our approach on real-world XSS worms, as well as obfuscated JavaScript code produced by some common JavaScript obfuscators. We then present the performance overhead results, reason about parameter settings, and discuss potential threats to the validity of our approach.

2.4.1 Real-World XSS Worms

The XSS worms examined in this section are all XSS worms released on popular real-world Web applications. All of these worms exploited persistent XSS vulnerabilities in different Web applications.

The Samy Worm

Based on the source code of the Samy worm [39], we wrote and tested an XSS worm on a small-scale Web application that we constructed. We named our Web application SamySpace. SamySpace mimics the necessary functionality of MySpace to allow the propagation of an XSS worm. We stored each user's profile in a backend `MySQL` database. We modified the original Samy worm code [39] as little as possible to make it work on SamySpace. As in the Samy worm, our worm sends five AJAX requests in total.

The original Samy worm is only slightly obfuscated using short variable names and few newline characters to fit itself into the limited space of the `interest` field in a user's profile. Our implementation is able to detect the XSS worm released on SamySpace by observing a high similarity of 91% between a parameter value sent in a request and a script extracted from the DOM tree.

The Orkut Worm

Different from the Samy worm, the Orkut worm is a heavily obfuscated XSS worm. During its outbreak, a user's account on Orkut was infected when the user simply read a `scrap` sent by the user's infected friend. The Orkut worm payload is contained in an external JavaScript file named `virus.js`, the URI

XSS worm	Propagation method	Triggering method	Payload location	Release date
Samy Worm	XHR	javascript: URIs	DOM	Oct. 2005
Xanga	XHR	javascript: URIs	DOM	Dec. 2005
Yamanner	XHR	`onload` event handler	server	Jun. 2006
SpaceFlash	XHR	javascript: URIs	URI link	Jul. 2006
MyYearBook	form submission	`innerHTML`	DOM	Jul. 2006
Gaia	XHR	`src` attribute	URI link	Jan. 2007
U-Dominion	XHR	`src` attribute	URI link	Jan. 2007
Orkut	XHR	`src` attribute	URI link	Dec. 2007
Hi5	form submission	`-moz-binding` / `expression`	URI link	Dec. 2007
Justin.tv	XHR	`src` attribute	URI link	Jun. 2008
Twitter	XHR	`src` attribute / `expression`	URI link	Apr. 2009

Table 2.1: Statistics of XSS worms in the wild.

of which is included in an `<embed>` element. The `<embed>` element is injected into the value of a parameter, which is used to store the message body of a scrap.

The Orkut worm works in three steps. To begin with, the worm payload embedded in `virus.js` reconstructs itself. It then propagates the worm payload to everyone present in the victim's friend list. Finally, it sends an asynchronous request to add the victim to a community, which tracks the total number of infected users, without the user's approval.

The parameter values sent by the Orkut worm include a malicious URI link. Since the URI link contained in the outgoing request is also embedded in the DOM, we are able to detect the propagation behavior of this XSS worm.

XSS Worms In the Wild

We carefully examined the available source code of XSS worms in the wild. Table 2.1 lists eleven of them. The first column of the table shows the names of XSS worms. We use the names of the infected websites to represent released worms when there is no ambiguity. The second column shows the propagation methods that were used. XMLHttpRequest (XHR) denotes asynchronous requests sent in the background, while form submission denotes HTTP requests which are sent when HTML forms are submitted or links are clicked. The third column of the table shows the triggering methods of

worm payloads. The `expression` function, which takes a piece of JavaScript code as its parameter, is supported in both Internet Explorer and Netscape; the `-moz-binding` attribute, which binds JavaScript code to a DOM element, is supported in both the Firefox and the Netscape browsers. The fourth column shows where worm payloads were extracted for payload reconstruction. Finally, the last column shows the release dates.

Yamanner was released on `Yahoo!Mail`; the `Xanga` worm was released on a blog; the `Gaia` worm and the `U-Dominion` worm were released on gaming websites; the `Justin.tv` worm was released on a video hosting Web site; and all of the six remaining worms were released on social networking websites. MySpace, a popular social networking Web site, is one of the favorite targets of XSS worms. Both the Samy worm and the SpaceFlash worm were released on it.

With regard to worm propagation methods, most attackers chose to use XHR for its advantage over form submission: asynchronous requests sent quietly in the background often go unnoticed. There are various kinds of XSS vulnerabilities, and so are script triggering methods. Most methods work on major Web browsers, except *-moz-binding* and *expression*, which are browser specific.

For XSS worms that propagate by reconstructing their payload from the loaded DOM tree, our approach is able to detect high similarity between parameter values of outgoing HTTP requests and DOM scripts; for XSS worms which propagate by sending links to external files, our approach is able to detect the existence of identical URI links that appear in both parameter values of outgoing requests and HTML attributes of the current DOM tree.

The Yamanner worm is a special case of XSS worms because its propagation process is actually performed on the server side rather than on the client side. Back in 2005 when the Yamanner worm was unleashed, the Yahoo!Mail system provided two ways to forward an email: either as inline text or as an attachment. For forwarded email with inline text, the message body of the original email was embedded in the message body of the forwarded email. For forwarded email with an attachment, only the message ID of the original email was embedded in the forwarded email; the message body of the original email was later retrieved when the attachment was opened or downloaded. The Yamanner worm used the attachment method to forward malicious emails; therefore, outgoing requests sent by Yamanner to forward emails only contained the message IDs of original emails. To obtain the message

body with the actual worm payload on the client side, we would need to write application-specific code to retrieve message bodies from the application server.

2.4.2 Effectiveness of Our Approach for Obfuscated JavaScript Code

When JavaScript code obfuscation was introduced a few years ago, it was mainly used to provide control over intellectual property theft. With the advent of XSS worms, we see the increasing abuse of legitimate JavaScript obfuscators by malware authors. We have seen an online XSS worm tutorial suggesting people obfuscate their code using a legitimate JavaScript packer [25]. We believe unsophisticated malware authors normally would not take the effort to write their own JavaScript obfuscators. When we first examined the obfuscated source code of the Orkut worm, we noticed that it used an unusual decoding function which has six parameters: *"(p, a, c, k, e, d)"*. After some research, it turned out that the obfuscated code was generated by a legitimate and publicly available JavaScript packer written by Dean Edwards [25]. The decoding function acts as a signature function of his JavaScript packer.

The purpose of using JavaScript obfuscators is to turn JavaScript source code into functionally equivalent JavaScript code that is more difficult to study, analyze and modify. Eventually any obfuscated JavaScript code must be read and correctly interpreted by a JavaScript engine. Common JavaScript obfuscation techniques include the following:

- Code shuffling and code nesting.
- String manipulations such as string reversion, split and concatenation.
- Character encoding.
- Insertions or deletions of arbitrary comments, spaces, tabs, and newline characters.
- Variable renaming and randomized function names.
- The use of encryption and decryption.

With the combination of our automatic decoder and trigram algorithm, our approach is robust in dealing with the first four obfuscation techniques. For example, the code shuffling technique has

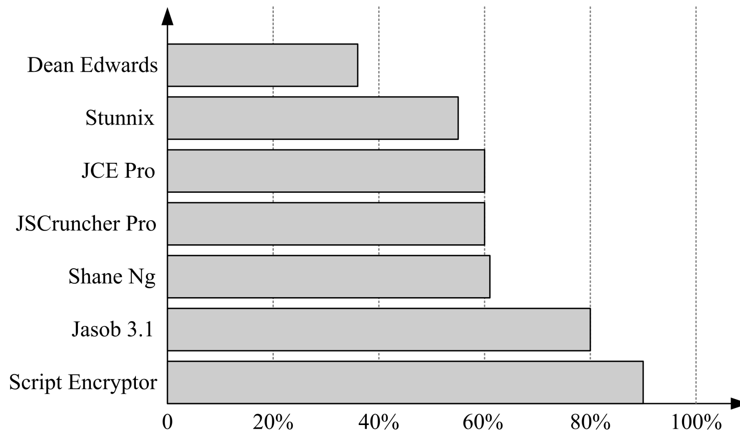

Figure 2.3: Similarity results for common JavaScript obfuscators.

no impact on our string-based similarity detection process. We tested our approach on widely used JavaScript obfuscators that we have collected. We used each obfuscator to obfuscate five real-world XSS worms, namely the Samy, Orkut, Yamanner, Hi5, and Justin.tv worms. We computed the similarity between original source code and obfuscated code, and calculated the average similarity for each JavaScript obfuscator. We show the results in Figure 2.3.

Based on the real-world XSS worms and obfuscated JavaScript code that we have collected, we set the string similarity threshold for the trigram algorithm to 20%. As can be seen in Figure 2.3, the similarity results for these obfuscators all exceed the conservative 20% threshold. Although the similarity result for Dean Edwards's JavaScript packer is relatively low, code obfuscated by this tool can be easily discerned by its signature function. An alternative is to simply replace the `eval` function with a `print` equivalent function for the complete exposure of the original worm payload. We observed that Stunnix, a commercial JavaScript obfuscator, applied multiple encoding routines using different character encoding schemes. Thanks to our automatic decoder, our extension was still able to detect a high similarity between the original and obfuscated code generated by Stunnix.

2.4.3 Overhead Measurements

To estimate the performance overhead imposed by our extension, we visited the top 100 most popular United States websites ranked by Alexa [4]. We measured the page load time for the top 100 websites

Web site	**w/o avg(s)**	**w/ avg(s)**	**overhead average**	**Web site**	**w/o avg(s)**	**w/ avg(s)**	**overhead average**
Google	0.240	0.243	1.17%	AOL	3.796	3.878	2.16%
Yahoo!	0.724	0.738	1.96%	Blogger	0.745	0.762	2.34%
Facebook	0.662	0.683	3.14%	Amazon	2.616	2.696	3.06%
YouTube	1.738	1.784	2.65%	Go	3.574	3.614	1.12%
MySpace	0.914	0.948	3.72%	CNN	6.450	6.746	4.59%
MSN	1.192	1.214	1.85%	Microsoft	1.756	1.782	1.48%
Windows Live	0.361	0.371	2.66%	Flickr	0.618	0.641	3.76%
Wikipedia	1.308	1.336	2.14%	ESPN	2.694	2.786	3.41%
Craigslist.org	0.283	0.285	0.78%	Photobucket	1.274	1.308	2.67%
eBay	0.712	0.722	1.41%	Twitter	1.098	1.118	1.82%

Table 2.2: Performance overhead for top 20 websites.

with Firebug [28], an open source Firefox Extension for Web development. To eliminate the impact of cache on measured time, we disabled browser cache by setting both `browser.cache.disk.enable` and `browser.cache.memory.enable` to `false` in our browser configuration. We first visited each Web page five times with our extension disabled, and then visited the same page another five times with our extension enabled. Due to space limitation, we show in Table 2.2 the performance overhead imposed by our extension for the top 20 websites. Column 1 and 5 show the names of the websites; column 2 and 6 show the average page load time in seconds without our extension; column 3 and 7 show the average page load time in seconds with our extension enabled; column 4 and 8 show the performance overhead for each Web site tested.

For all the 100 websites we visited, the number of requests a page load generates ranged from 3 to 390. All generated HTTP requests were monitored by our Firefox extension. The average number of requests a Web page generates on a page load event was 89; the average content length of corresponding HTTP responses was 666KB. Our extension increased page load time by 2.62% on average.

The primary overhead of running our extension is due to the latency introduced by the decoding and similarity detection processes. Websites that sent more outgoing HTTP requests with a large number of parameters incurred higher performance overhead than other websites. Among the listed top 20 websites, `CNN` sent out the largest number of requests, 249 requests in total, for a page load event; our extension incurred the highest overhead on `CNN`. Most websites ran smoothly with our extension

enabled. Of all the Web pages tested, `Craigslist.org` generated the fewest number of requests; we observed only minor performance overhead for `Craigslist.org`. Overall, the overhead incurred by our extension is reasonably low for everyday use.

2.4.4 Parameter Settings

Minimum URI link length

To avoid unnecessary computation and increase the efficiency of our algorithm, we conservatively set the minimum URI link length to 12. Protocol declarations, such as `http://` which has seven characters, are also counted in URI links. If the length of a parameter value is of length less than 12, we skip that parameter value all together without searching for URI links in it.

Minimum length of XSS worm payloads

We conservatively set the minimum length of an XSS worm payload p to 155 characters according to the result of an XSS worm replication contest [2]. This contest aimed to find the smallest amount of code necessary for XSS worm propagation. Having to deal with application-specific requirements, real-world XSS worms should have larger code base than the winners of this contest. In order for an XSS worm to propagate itself, it needs to at least obtain, reconstruct and embed its payload, and send an HTTP request. The smallest real-world XSS worm we have seen so far, the MyYearBook worm, is composed of 769 characters after a normalization process. If any element in set $\mathscr{P}$ or set $\mathscr{D}$ is of length less than p, we remove it from the set.

2.4.5 Discussion

Of all the websites that we have tested, most of them exhibit normal behavior. Only a few websites propagate advertisement URI links with long query strings similarly to the way that XSS worms propagate their payloads. However, files pointed to by these URI links are not dynamically loaded into the DOM tree and thus do not pose any threats. To limit the impact of these URI links on the similarity detection process, we used a URI query string filter to remove such query strings. We tested

our extension on the top 100 websites and enabled our extension in our everyday browsing for two weeks. With the current settings, we have not observed any false positives.

Some websites include identical style sheets or JavaScript library modules in multiple Web pages. However, such websites do not raise false alarms because these remote files are only loaded in the Web pages, but not propagated through outgoing requests.

As mentioned in Section 2.4.1, our approach does not deal with server-side self-replicating worms. This is because worms that propagate on the server side store their payloads directly on the Web application server rather than in parameter values or external files. In such cases, triggering the worm payload requires additional user actions rather than simply viewing Web pages. For this reason, server-side XSS worms are relatively rare in the wild. To detect such XSS worms, server-side coordination is necessary.

We have shown that our solution is effective with regard to popular JavaScript obfuscators. However, determined attackers might be willing to take the effort to create their own obfuscation techniques and write their own encryption and decryption routines to create highly obfuscated worms. For advanced obfuscation techniques, we expect behavior-based approaches to be more effective.

2.5 Related Work

We survey closely related work in this section.

2.5.1 Worm Detection

Researchers have proposed signature-based approaches to detect polymorphic worms. The key idea for signature generation is to find invariant substrings [46] or structural similarities in all variants of a payload. Such techniques can also be applied to highly obfuscated XSS worms.

Techniques for the detection of traditional Internet worms include content filtering [19], network packet analysis [21], honeypots, and worm propagation behavior analysis. Wang *et al.* [75, 76] proposed a similarity-based approach using an n-gram based detection algorithm.

Spectator [48] also detects JavaScript worms by identifying the propagation behavior of JavaScript

worms. In particular, it tags the traffic between browsers and web applications, and sets a threshold to detect worm-like long propagation chains. Although it is effective in detecting JavaScript worms, their approach can only detect JavaScript worms that have propagated far enough. Different from their approach, we can detect XSS worms in a timely manner.

2.5.2 Client-Side Protection

Several client-side approaches have been proposed to address XSS vulnerabilities. Most of them are not purely client-side solutions and require server-side cooperation.

Client-side policy enforcement mechanisms aim to enforce security policies provided by web application developers to make sure that browsers interpret web pages in expected ways. BEEP [37] provides two kinds of policies: a whitelist policy for trusted scripts and a blacklist policy for DOM sandboxing. Similarly, Noncespaces [32] use whitelist and ancestry-based sandbox policies along with the Instruction Set Randomization technique to constrain the capabilities of untrusted content.

To prevent injection attacks, several approaches [51, 55, 65] in the literature rely on the preservation of intended parsing behavior. BLUEPRINT [51] seeks to minimize the trust placed on browsers for interpreting untrusted content by enabling a web application to effectively take control of parsing decisions. DSI [55] ensures the structural integrity of HTML documents.

Noxes [42] is the first purely client-side solution to mitigate XSS attacks. It works as a personal web firewall that helps mitigate XSS attacks with both manually and automatically generated rules to protect against information leakage. However, as most of the above approaches, it aims to detect XSS attacks but may not work in the face of XSS worms, which exploit the trust between users and cause damage without accessing sensitive user information.

2.5.3 Server-Side Analysis

Previous server-side techniques mostly address web application vulnerabilities using information flow analysis. Static analysis [80, 83] aims to detect cross-site scripting vulnerabilities before the deployment of web applications, while dynamic analysis [12, 47, 56] aims to provide detailed information of

vulnerabilities and exploits at run time. Saner [8] uses both static analysis and dynamic analysis to analyze custom sanitization processes. Sekar [63] recently proposed a black-box taint-inference technique that works by observing inputs and outputs of a web application.

The challenge of applying server-side analysis lies in the difficulty of finding all XSS vulnerabilities in web applications. Our approach, in comparison, seeks to detect XSS worm payloads rather than to find all XSS vulnerabilities within a web application.

2.6 Conclusion

This chapter presents the first purely client-side solution to effectively detect XSS worms by observing the propagation of worm payloads. The main idea of our approach is to identify similar strings between the set of parameter values in outgoing HTTP requests and retrieved external files, and the set of DOM scripts and loaded external files. We implemented our approach as a cross-platform Mozilla Firefox extension. We evaluated its effectiveness on some real-world XSS worms and its resilience against some common JavaScript obfuscators. Finally we measured the performance overhead incurred by our Firefox extension on the top 100 U.S. websites. Our empirical results show that our extension is effective in detecting self-replicating XSS worms on the client side with reasonably low performance overhead. Because our approach is general and effective, it can be applied to other browsers besides Firefox to detect self-replicating XSS worms on the client side.

Chapter 3

Static Detection of Access Control Vulnerabilities in Web Applications

Access control vulnerabilities, which cause privilege escalations, are among the most dangerous vulnerabilities in web applications. Unfortunately, due to the difficulty in designing and implementing perfect access checks, web applications often fall victim to access control attacks. In contrast to traditional injection flaws, access control vulnerabilities are application-specific, rendering it challenging to obtain precise specifications for static and runtime enforcement. On one hand, writing specifications manually is tedious and time-consuming, which leads to non-existent, incomplete or erroneous specifications. On the other hand, automatic probabilistic-based specification inference is imprecise and computationally expensive in general.

This chapter describes the first static analysis that automatically detects access control vulnerabilities in web applications. The core of the analysis is a technique that statically infers and enforces *implicit access control assumptions*. Our insight is that source code implicitly documents intended accesses of each *role* and any successful *forced browsing* to a privileged page is likely a vulnerability. Based on this observation, our static analysis constructs sitemaps for different roles in a web application, compares per-role sitemaps to find privileged pages, and checks whether forced browsing is successful for each privileged page. We implemented our analysis and evaluated our tool on several real-world

web applications. The evaluation results show that our tool is scalable and detects both known and new access control vulnerabilities with few false positives.

3.1 Introduction

Web applications often restrict privileged accesses to authorized users. While bringing the convenience of accessing a large amount of information and operations from anywhere into people's daily lives, web applications have opened a new door for attacks and the number of web-based attacks is on the rise. A Symantec Internet security threat report published in April 2011 points out that the volume of web-based attacks in 2010 increased by 93% over the volume observed in 2009[1]. Researchers of web security have focused their attention on injection vulnerability, which is the most common vulnerability in web applications. Although not as prevalent as injection vulnerability, access control vulnerability poses a more serious threat because of exposed privileges, and has started attracting the attention of researchers [22]. Compared with those in traditional software, access checks in web applications are harder to get right because of the stateless nature of the HTTP protocol. In traditional software, once a user has passed an authentication check, the system remembers the identity of the user until she logs out or a timeout event happens. This is not the case for web applications, which must parse each new HTTP request to identify a previously logged-in user. A statistics report published in 2007 states that 14.15% of the surveyed web applications suffer from vulnerabilities of insufficient authorization[2].

Traditional injection vulnerabilities such as Cross-Site Scripting (XSS) and SQL injection are not application-specific and have a clear and general definition [65]: an injection vulnerability exists when an untrusted input flows into a sensitive sink without proper sanitization. To detect injection vulnerabilities, it is sufficient to analyze individual pages separately to examine where untrusted user inputs can flow. In contrast, access control vulnerabilities are application-specific, and it is necessary to examine connections between pages.

Web application developers frequently make implicit assumptions of allowed accesses and protect privileged pages by hiding links to these pages from unauthorized users. However, security by obscurity

[1] http://www.symantec.com/business/threatreport
[2] http://projects.webappsec.org/f/wasc_wass_2007.pdf

is insufficient to prevent a determined and skilled attacker from accessing these pages, viewing sensitive data or performing dangerous operations. As an example, Business Wire used a web server to store files of important trade information, which were supposed to be accessible to registered members only. Although the URLs to these files were hidden in the presentation layer from unauthorized users, the date-based URLs were highly predictable. By simply accessing these privileged files, an investment bank Lõhmus Haavel & Viisemann profited over eight million dollars based on the disclosed trade information[3]. Similarly, in November 2010, Blooming News obtained and published valuable financial earnings data of Disney and NetApp to its subscribers hours before official data releases by predicting resource locations inside secure corporate networks. As yet another example, accesses to the videos of USENIX conference presentations are restricted to USENIX members for a short period after a conference. However, the authors of this chapter were able to predict the author-name-based URLs of the videos and download a few videos as public users.

Researchers have proposed various static and dynamic analysis techniques [7, 22, 27, 35] to detect violations of application logic, including access control attacks. Unfortunately, these techniques have limited effectiveness on detecting access control vulnerabilities. Dynamic analyses have difficulty finding hidden pages and determining intended accesses for each role. Furthermore, sitemaps covered by dynamic executions tend to be shallow and incomplete as user inputs are usually limited. Despite that static analyses typically have better coverage, they often require good specifications in order to generate useful reports, whose false positives do not overwhelm users. In practice, deriving precise specifications is challenging, especially when diverse authentication and access control management schemes are in use. As manually writing specifications is time-consuming and probabilistic-based inference is error-prone, it is desirable to precisely infer implicit assumptions on intended accesses from the source code of applications.

In this chapter, we use *role* to represent a unique set of privileges that a group of users has. Most web applications have at least three types of roles: the role for administrators, the role for normal logged-in users and the role for public or anonymous users. Access control checks must be performed before granting access to any privileged resource to prevent privilege escalation attacks. When implicit

[3] http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf

assumptions are not matched by explicit access checks, unauthorized accesses are possible.

We propose the first role-based static analysis to detect access control vulnerabilities with automatic inference on implicit access control assumptions. Our key observations are that each role represents a unique set of privileges, and intended accesses for each role are reflected in explicit links shown in the presentation layer of an application. Guided by these observations, our analysis automatically derives specifications on privileged accesses by comparing explicit links presented to different roles. It then directly accesses privileged pages for unprivileged roles, and examines whether these accesses are allowed to detect vulnerable pages which have missing or insufficient access checks. Our main contributions are:

- A formal definition of access control vulnerabilities in web applications.
- The first role-based static analysis which automatically detects access control vulnerabilities in web applications with minimal manual efforts.
- An implementation of our analysis which constructs intended per-role sitemaps. Given role-based specifications, our prototype can systematically explore feasible execution paths based on the satisfiability of constraints.
- An evaluation of our tool on real-world web applications. Our tool works on unmodified code, and is able to detect both new and known vulnerabilities before the deployment of web applications. The evaluation results show that our approach is scalable and effective, with few false positives.

The rest of the chapter is organized as follows. We first use an example to illustrate the main steps of our approach (Section 3.2) and then present our formalization of access control vulnerability in web applications (Section 3.3). Section 3.4 describes our detailed algorithms. Section 3.5 presents the implementation details of our static analyzer, and Section 3.6 shows the effectiveness, coverage and performance of our analyzer on real-world web applications. Finally, we survey related work (Section 3.7) and conclude (Section 3.8).

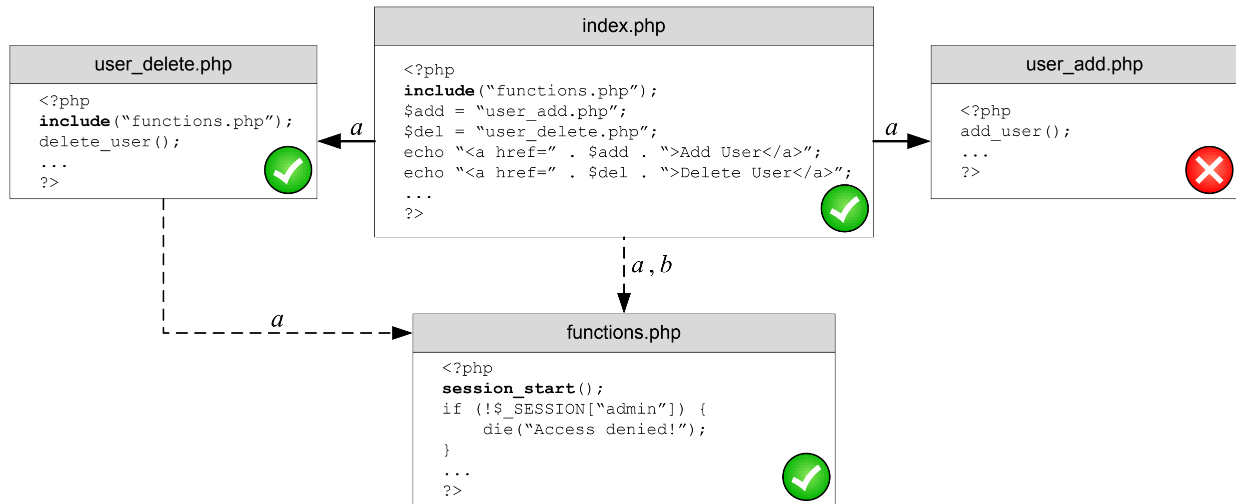

Figure 3.1: An Example of Access Control Vulnerability. Solid arrows represent explicit links, and dashed arrows represent inclusion relationship between pages. Arrows correspond to edges in sitemaps and are labeled with roles. The intended sitemap for privileged role a has four edges while the intended sitemap for role b has only one edge.

3.2 Illustrative Example

Figure 3.1 shows a simple web application based on one of the real-world web applications in our test suite. For illustration, suppose that the application has two roles: role a for administrators and role b for normal users. In our approach, we require developers to only specify application entry points and role-based application states, which serve as the basis for automatically inferring the set of privileged pages. Suppose that in the given specifications, the entry sets for both roles are identical and contain only "index.php", and the value of $\$_SESSION$["admin"] is specified as **true** for role a but **false** for role b. As we can see from the source code, only "functions.php" checks accesses. This file is included via PHP inclusion in both "index.php" and "user_delete.php", but not "user_add.php." Consequently, access checks are missing in "user_add.php" but present in the other three pages.

The first step of our analysis constructs per-role sitemaps with a worklist-based algorithm. Initially, worklists for both roles are ["index.php"]. While a worklist is not empty, our analysis pops a work node from the front of the worklist each time. Let us look at the sitemap construction for role a first. The first analyzed node is "index.php". From this node, users of role a can explicitly reach both "user_add.php"

and “user_delete.php” via anchor tags, and “functions.php” via a file inclusion. Thus, our analysis adds three new edges in the sitemap and appends the newly discovered nodes to the worklist, which is now [“user_add.php”, “user_delete.php”, “functions.php”]. The second analyzed node is “user_add.php”. This node can not reach any nodes, and thus our analysis pops “user_delete.php” and the worklist becomes [“functions.php”]. Role a can reach “functions.php” from “user_delete.php”, and thus our analysis adds a new edge in the sitemap. Because “functions.php” is already in the worklist, it is not appended to the current worklist. Finally, our analysis pops “functions.php”. This node can not reach any nodes and our analysis stops because the worklist is now empty. Now let us look at the sitemap construction for role b. The first popped node is still “index.php”. However, role b can only explicitly reach “functions.php” via a file inclusion from this node. The links to “user_delete.php” and “user_add.php” are hidden from users of role b in “index.php” via the access check in “functions.php”. Therefore, our analysis adds only one new edge and stops because the worklist is now empty. The edges of constructed per-role sitemaps are shown in Figure 3.1.

The second step of our analysis infers the set of privileged pages and attempts to access these pages directly to detect access control vulnerabilities. Comparing the sets of explicitly reachable nodes for role a and role b, our analysis infers that “user_add.php” and “user_delete.php” are privileged pages intended for users of role a only. Consequently, these two pages should have access checks to ward off users of role b. Unfortunately, only “user_delete.php” is safeguarded and “user_add.php” is left unprotected. Therefore, a direct access to “user_delete.php” fails, whereas a direct access to “user_add.php” succeeds, indicating that “user_delete.php” is guarded and “user_add.php” is vulnerable.

3.3 Approach Formulation

This section formulates our high-level approach. We define the notions of *role*, *explicit link*, *forced browsing*, *web application* and *access control vulnerability*, and present two assumptions we make with regard to roles and intended accesses.

Definition 3.1 (Role). A *role* $r \in R$ captures the set of allowed accesses for all users of role r where set R denotes roles that a web application has. Each role r represents a distinctive set of privileges.

Assumption 3.1: We assume that roles in R form a lattice $\langle R, \sqsubseteq \rangle$, where $\sqsubseteq$ denotes the ordering relationship between any two roles. Under this assumption, accessing a privileged resource as an unprivileged role is considered a privilege escalation attack. Roles at the same level of the lattice are not ordered by $\sqsubseteq$ as they may represent different sets of allowed accesses. The role for administrators is $\top$; the role for public users is $\bot$; and the role for normal logged-in users lies in the middle of the lattice.

Definition 3.2 (Explicit Link). In a web application, there exists an *explicit link* from page n_i to a different page n_j when it is possible to jump to n_j via an explicit URL in n_i, incurring no exceptions or errors. URLs might appear in file inclusions, header redirections, HTML tags for anchors, forms, meta refresh headers, frames, iframes, scripts, images or links.

Definition 3.3 (Forced Browsing). *Forced browsing* is the act of directly accessing privileged pages rather than following explicit links in a web application. Attackers often harness brute force techniques to access hidden pages with predictable locations. We consider forced browsing successful when HTML pages presented to two different roles are identical, and no redirections, exceptions or errors occur during the page rendering process.

Definition 3.4 (Web Application). Let *node* represent a web page. Suppose that a web application contains k nodes. Given a user role $r \in R$, we abstract the *web application* as $P_r = (S_r, Q_r, E_r, I_r, \Pi_r, N_r)$, where

- *Entry set* S_r contains the entry nodes to the web application. We include index pages in all directories in the entry set. Different roles may have different entry sets.
- *State set* $Q_r = \{q_i \mid 0 \leq i < k\}$ is a set of application states. For each node n_i, an application state q_i captures critical information at that node. It might include session values, cookie values, request parameter values, database records, variable values or function return values.
- *Explicit edge set* $E_r = \{\langle n_i, n_j \rangle \mid 0 \leq i, j < k\}$. An explicit edge from node n_i to n_j exists iff n_i in state q_i contains an explicit link to n_j.
- *Implicit edge set* $I_r = \{\langle n_i, n_j \rangle \mid 0 \leq i, j < k\}$. An implicit edge from node n_i to n_j exists iff forced

browsing enables one to jump to n_j from n_i in state q_i. Accesses via implicit edges are allowed but often unintended.

- *Navigation path set* $\Pi_r = \{(n_i)_{0 \le i < l} \mid 0 < l < k \wedge n_0 \in S_r \wedge \langle n_i, n_{i+1} \rangle \in (E_r \cup I_r)\}$. It consists of all possible navigation paths for role r, including explicit edges as well as implicit edges.
- *Explicitly reachable node set* N_r consists of nodes that are reachable from application entries in S_r via explicit edges in E_r. It can be easily computed with a graph reachability analysis.

Assumption 3.2: For each node in a web application, if multiple roles can reach this node on navigation paths composed of only explicit edges, we assume that the privilege level required to access this node is determined by the least privileged role.

Definition 3.5 (Access Control Vulnerability). Let $a, b \in R$ denote two roles that can be ordered in a web application where role b is less privileged than role a, *i.e.*, $b \sqsubset a$. An *access control vulnerability* exists at node n when:

$$n \in N_a \wedge n \notin N_b \wedge \exists\ \pi_b \in \Pi_b\ (n \in \pi_b)$$

In this definition, destination node n is a privileged node intended to be accessible to role a but not role b. We use $n \in \pi_b$ to denote that n is on navigation path π_b. This node is vulnerable to access control attacks when a user of role b is able to access n via an allowed, but probably unintended, navigation path π_b.

3.4 Analysis Algorithm

In this section, we introduce the three major algorithms of our approach. Section 3.4.1 describes how our analysis automatically infers specifications of implicit access control assumptions and detects access control vulnerabilities from a high-level view. Section 3.4.2 shows the algorithm that we use to build per-role sitemaps. Finally, we present the detailed link extraction algorithm in Section 3.4.3.

```
DetectVuls(Spec_a, Spec_b, reg)
 1  Vuls ← ∅
 2  nfa ← Reg2NFA(reg)
 3  dfa ← NFA2DFA(nfa)
 4  N_a ← BuildSitemap(Spec_a, dfa)
 5  N_b ← BuildSitemap(Spec_b, dfa)
 6  Privileged ← N_a \ N_b
 7  for each n in Privileged
 8  do ⟨cfg_a, R_a⟩ ← GetCFG(n, Spec_a)
 9     ⟨cfg_b, R_b⟩ ← GetCFG(n, Spec_b)
10     if SizeOf(cfg_a) = SizeOf(cfg_b) and R_a = R_b
11        then Vuls ← Vuls ∪ {n}
12  return Vuls
```

Figure 3.2: Algorithm for Vulnerability Detection.

3.4.1 Vulnerability Detection

Figure 3.2 presents the vulnerability detection algorithm which is the core of our approach. This algorithm infers privileged nodes from the source code of a web application and identifies nodes that are not properly protected.

Let $Spec_a$ and $Spec_b$ denote specifications for role a and role b respectively. Initially, the set of vulnerable nodes *Vuls* is empty. First, this algorithm parses the regular expression *reg*, which captures HTML tags where a link might appear, into a non-deterministic finite automaton (NFA). Then, the algorithm transforms the NFA into a deterministic finite automaton (DFA). Either NFA or DFA could be used for extracting links, and we chose DFA for its advantage on performance and the ease of FA state management.

Throughout this chapter, we assume role a is more privileged than role b. Following Definition 3.4, we use N_a and N_b to denote the sets of explicitly reachable nodes for roles a and b respectively. Function BuildSitemap, whose details are shown later in Section 3.4.2, computes these two sets. Relying on Assumption 3.2, the algorithm infers privileged nodes that are present in N_a but not in N_b (Line 6). For the example in Section 3.2, N_a ={“index.php”, “user_add.php”, “user_delete.php”, “functions.php”} and N_b ={“index.php”, “functions.php”}.

Access checks at privileged locations may be missing or insufficient. This algorithm analyzes each

privileged node n twice with function GETCFG, once for role a to create an oracle for the intended server response (Line 8), and once for role b to emulate forced browsing (Line 9). Given a role r and a privileged node n, GETCFG returns a context-free grammar (CFG) cfg_r and the set of page redirections R_r.[4] The obtained cfg_r is an approximation of the dynamic HTML output of node n. We observe that when an access check succeeds, users are often granted accesses to sensitive information or operations; otherwise, they are redirected to another page, or presented with error messages or login forms. In the latter case, CFG sizes of the two roles are different because of the different HTML outputs that are presented. Consequently, if the sizes of the two CFGs or the two redirection sets differ, node n is considered guarded; otherwise, n may be vulnerable (Line 11). For the privileged page "user_delete.php" shown in Figure 3.1, $\text{SIZEOF}(cfg_a) \neq \text{SIZEOF}(cfg_b)$ and $R_a = R_b = \emptyset$, indicating that the page is guarded; for the privileged page "user_add.php", $\text{SIZEOF}(cfg_a) = \text{SIZEOF}(cfg_b)$ and $R_a = R_b = \emptyset$, indicating that the page is vulnerable.

3.4.2 Building Sitemaps

Function BUILDSITEMAP shown in Figure 3.3 builds a per-role sitemap with specifications $Spec_r$ for role r and the DFA *dfa*. We use a worklist-based algorithm to traverse nodes in a web application in a breath-first manner. Initially, both the visited node set *Visited* and the edge set E_r are empty, and the worklist *WkLst* is initialized with the entry set S_r specified in $Spec_r$ (Line 3).

In each iteration of the loop, function GETWORKNODE pops a working node n_i from the front of list *WkLst* and retrieves its associated state q_i from $Spec_r$ (Line 5) to find outgoing edges of this working node. Next, this algorithm constructs a CFG that represents the possible HTML outputs of node n_i (Line 6). Besides cfg_i, function CONSTRUCTCFG also returns the page redirection set R_i and the file inclusion set F_i as links in these two sets also contribute to outgoing edges in a sitemap. Then, function EXTRACTLINKS extracts a set of matched links L_i that are present in cfg_i based on *dfa* (Line 7). The details of EXTRACTLINKS are presented later in Section 3.4.3. The set of reachable nodes N_j for n_i is the union of L_i, R_i and F_i (Line 8). We conservatively include F_i in this union because included files may present sensitive information or operations. The algorithm adds an outgoing edge $\langle n_i, n_j \rangle$

[4] Throughout this chapter, CFG stands for context-free grammar rather than control-flow graph.

```
BuildSitemap(Spec_r, dfa)
 1  E_r ← ∅
 2  Visited ← ∅
 3  WkLst ← GetEntries(Spec_r)
 4  while WkLst
 5  do ⟨n_i, q_i⟩ ← GetWorkNode(WkLst, Spec_r)
 6       ⟨cfg_i, R_i, F_i⟩ ← ConstructCFG(n_i, q_i)
 7       L_i ← ExtractLinks(cfg_i, dfa)
 8       N_j ← L_i ∪ R_i ∪ F_i
 9       for each n_j in N_j
10       do E_r ← E_r ∪ {⟨n_i, n_j⟩}
11       Visited ← Visited ∪ {n_i}
12       N ← Active(N_j) \ (Visited ∪ WkLst)
13       WkLst ← Append(WkLst, N)
14  return GetNodes(E_r)
```

Figure 3.3: Algorithm for Building Sitemaps.

to the explicit edge set E_r for each node $n_j \in N_j$ (Line 10) and then adds n_i to the visited node set (Line 11). To determine which nodes to analyze, we partition nodes into active nodes and inactive nodes, and only analyze active ones. Active nodes may have outgoing edges in a sitemap, whereas inactive nodes are dead ends. For example, a PDF file is considered an inactive node, while a PHP page is considered an active node. Finally, the algorithm adds the newly discovered active nodes to the worklist, excluding the ones that have been visited or are already in the worklist (Line 12, 13). The loop terminates when *WkLst* becomes empty, indicating that the construction of a per-role sitemap is complete. At this point, function BuildSitemap returns the set of explicitly reachable nodes N_r based on E_r (Line 14). When work node $n_i =$"index.php" shown in Figure 3.1 is analyzed for role a in a loop iteration, $L_i =${"user_delete.php", "user_add.php"}, $R_i = \emptyset$ and $F_i =${"functions.php"}. Therefore, three new outgoing edges from "index.php" are added to E_a. In contrast, when "index.php" is analyzed for role b, $L_i = R_i = \emptyset$ and $F_i =${"functions.php"}. In this case, only one new edge is added to E_b.

3.4.3 Link Extraction

We use C to denote a CFG, and F to denote an FA. In our setting, a CFG represents the dynamic HTML output of a node and an FA matches a single link-introducing HTML tag of various forms. Let $\mathscr{L}(C)$

be the set of words in the language for the CFG and $\mathcal{L}(F)$ be the set of words in the language for the FA. Suppose that function SUBSTR returns **true** only when w' is a substring of w. The output of EXTRACTLINKS on C and F is defined as follows:

$$\begin{aligned}\textsc{ExtractLinks}(C,F) = \{\ w' \ \mid\ & w \in \mathcal{L}(C) \wedge \\ & w' \in \mathcal{L}(F) \wedge \\ & \textsc{Substr}(w', w)\ \}\end{aligned}$$

We could use a straight-forward three-step approach to extract links. In the first step, we could use the standard CFG-reachability algorithm [52] to compute a CFG representing the intersection of the two languages for C and F', where F' matches HTML outputs that contain at least one link-introducing tag. The subtle difference between F' and F is that F' matches link-introducing tags as well as link-irrelevant HTML outputs, while F only matches link-introducing tags. In the second step, we could generate all possible HTML outputs of the CFG. In the third step, we could use an HTML parser to extract links from the generated HTML outputs. Nevertheless, this approach is not ideal for two reasons. The first is that the words of a CFG can be infinite and we can only generate a finite set of possible HTML outputs. The second is that the generated HTML outputs are likely being highly similar, and thus we may repetitively parse similar HTML outputs. For better performance, we designed a new algorithm that does not generate intermediate HTML outputs, but directly extracts links from the CFG.

In a CFG $\langle V, \Sigma, P, S_0 \rangle$, V is a finite set of variables (*i.e.* non-terminals); Σ is a finite set of terminals which is the alphabet of the language; $P = \{v \rightarrow rhs \mid v \in V \wedge rhs \in (V \cup \Sigma)^*\}$ is a finite set of grammar productions; and S_0 is the start variable. In an FA $\langle Q, \Sigma', q_0, \delta, Q_f \rangle$, Q is a finite, non-empty set of states; Σ' is the input alphabet; $q_0 \in Q$ is the start state; $\delta : Q \times \Sigma \rightarrow Q$ is the state-transition relation; and $Q_f \subseteq Q$ is the set of final states.

Figure 3.4 shows our link extraction algorithm where function EXTRACTLINKS is the entry point. We use set *VQW* to store $\langle v, q, w \rangle$ tuples where v represents a CFG variable, q is an FA state and w is a partially matched link string. Completely matched links are stored in set *Words*. To begin with, this algorithm walks the CFG with the start CFG symbol S_0, the start FA state q_0, and the empty string which

```
WalkTerminal(t, q, w)
 1  q′ ← δ(q, t)
 2  if q′ = q0
 3     then return ⟨q0, “” ⟩
 4  w′ ← Append(w, t)
 5  if q′ ∈ Qf
 6     then Words ← Words ∪ {w′}
 7            w′ = “”
 8  return ⟨q′, w′⟩

WalkVar(v, q, w)
 10  VQW ← VQW ∪ {⟨v, q, w⟩}
 11  RHS ← Productions(v, P)
 12  if IsSigma(RHS) or RHS = ∅
 13     then return {⟨q, w⟩}
 14  QW ← ∅
 15  for each rhs in RHS
 16  do if IsEpsilon(rhs)
 17         then QW ← QW ∪ {⟨q, w⟩}
 18         else QW ← QW ∪ WalkSymbols(rhs, q, w)
 19  return QW

WalkSymbol(s, QW)
 21  Result ← ∅
 22  for each ⟨q, w⟩ in QW
 23  do if IsTerminal(s)
 24         then QW′ ← {WalkTerminal(s, q, w)}
 25         else if ⟨s, q, w⟩ ∈ VQW
 26                 then QW′ ← {⟨q, w⟩}
 27                 else QW′ ← WalkVar(s, q, w)
 28      Result ← Result ∪ QW′
 29  return Result

WalkSymbols(rhs = [γ], q, w)
 31  QW ← {⟨q, w⟩}
 32  for each s_i in [γ]
 33  do QW ← WalkSymbol(s_i, QW)
 34  return QW

ExtractLinks(cfg = ⟨V, Σ, P, S0⟩, fa = ⟨Q, Σ′, q0, δ, Qf⟩)
 36  VQW ← ∅
 37  Words ← ∅
 38  WalkVar(S0, q0, “” )
 39  return Valid(Words)
```

Figure 3.4: Algorithm for Link Extraction.

represents the terminals that have been partially matched (Line 38).

Function WalkTerminal is the only function that advances an FA state q to a new state q' based on the FA transition function δ and an input character t (Line 1). If q' is the FA start state q_0, which indicates a mismatch, the algorithm clears the partially matched terminals and returns (Line 3); otherwise, it appends t to w (Line 4) and examines q' again (Line 5). If q' is a final FA state in Q_f, the algorithm adds the completely matched link to *Words* (Line 6) and resets w' to the empty string. In this way, we filter out noises that are irrelevant to links in the CFG and only keep track of link-introducing HTML outputs.

Recursive function WalkVar walks the grammar productions of variable v under an FA state q and a partially matched word w. Function Productions retrieves the set of productions which have v as the left-hand-side variable from the CFG production set P, and returns the set of right-hand sides *RHS* (Line 11). The different elements in *RHS* indicate how the dynamic HTML output might diverge for v. Function IsSigma checks whether a set is equivalent to the CFG alphabet Σ. A link of value Σ^* can point to any file in the application and therefore should be discarded. If *RHS* forms the alphabet or the empty set, the function returns the pair of unchanged q and w in a set (Line 13); otherwise, it walks the elements in set *RHS* one by one. In each loop iteration, if a right-hand side *rhs* has no symbols, the HTML output remains the same (Line 17); otherwise, the algorithm searches the set of new possible outcomes QW' with a call to function WalkSymbols (Line 18).

Recursive function WalkSymbols walks the symbols in list $[\gamma]$ in order. Consequently, links in the CFG are matched in the order of their appearances in a possible HTML output. Here $[\gamma] = (s_i)^* \wedge s_i \in (V \cup \Sigma)$, representing a sequence of right-hand-side symbols. For each symbol s_i in the list, the algorithm transitions the set of possible outcomes to a new set (Line 33).

Recursive function WalkSymbol walks a right-hand-side symbol s under each possible outcome $\langle q, w \rangle$. In each loop iteration, the algorithm first examines the symbol s (Line 23). If s is a terminal, the FA state is deterministically advanced via function WalkTerminal (Line 24). Otherwise, if the symbol is a variable, this algorithm recursively calls function WalkVar for s (Line 27) when v is associated with a new q or a new w. The use of set *VQW* ensures the termination of the algorithm. This algorithm stops when all reachable grammar productions have been explored at least once. A concrete example of how

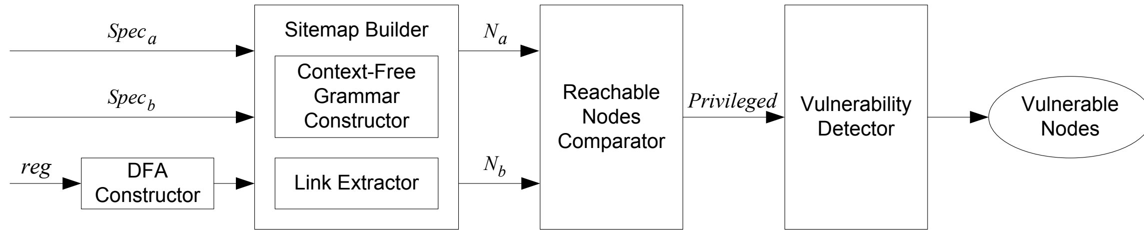

Figure 3.5: System Architecture.

this algorithm works is given in Section 3.5.2.

3.5 Implementation

As PHP is one of the most popular programming languages for web applications, we implemented our approach by extending Wassermann and Minamide's PHP string analyzer [53, 80], which is written in OCaml. The original PHP string analyzer was developed to detect injection vulnerabilities in web applications, and it analyzes individual pages in isolation and explores all execution paths. To detect access control vulnerabilities, we modified the string analyzer to build per-role sitemaps and examined connections between different pages. In particular, we introduced the concept of role into the static analyzer, added new specification rules for application states and entry sets, and strategically explored paths based on branch feasibilities. To explore only feasible execution paths, we keep track of both arithmetic constraints and string constraints. For arithmetic constraints, the analyzer consults a Satisfiability Modulo Theories (SMT) solver Z3 [23]; for string constraints, it consults a custom-built string constraint solver. Furthermore, we designed and implemented the algorithm shown in Figure 3.4 to efficiently extract explicit links from CFGs, added support for 176 built-in PHP functions, and modified both the specification lexer and parser to support specifications for the values of integers, floating-point numbers and strings.

Figure 3.5 shows our system architecture. A web application can have multiple roles, and our analysis compares a pair of ordered roles each time. Initially, the *DFA constructor* transforms the given regular expression reg into a DFA. The detection of access control vulnerabilities is carried out in two major steps. First, the *sitemap builder* explores the given web application based on parsed

specifications and the DFA. Second, the *reachable nodes comparator* infers what privileged nodes are, and the *vulnerability detector* performs forced browsing to detect nodes that are vulnerable to access control attacks.

3.5.1 Specification Rules

In our analysis, specifications are parsed with a lexer and a parser. For each role r, we only require developers to specify the entry set S_r and the set of critical application states Q_r. Multiple roles can share the same set of entry points. Either index pages or active pages with no incoming edges can be entry nodes. Index pages often have conventional names such as "index.php" and "index.html", and can be easily identified with a file scan; active pages with no incoming edges can be specified as entry nodes by developers. The types of application states that we support are listed in Definition 3.4. The state values that can be specified include abstract types and concrete values of built-in PHP types, and string values that can be represented by a regular expression. For function invocations, we allow developers to pinpoint an invocation by specifying the filename and line number where the invocation occurs. This is especially useful when function invocations return different values at different call sites.

Optionally, developers can explicitly specify a set of privileged nodes. In contrast to implicit navigation paths which involve forced browsing, explicit navigation paths are often tested more thoroughly. However, it is still possible that an allowed access to a sensitive node via an explicit navigation path of an unprivileged role is unauthorized, violating Assumption 3.2. In this case, when an unprivileged user can explicitly navigate to a privileged node, we would have false negatives. To solve this problem, we allow developers to explicitly specify privileged nodes. Such a node may be vulnerable to access control attacks even if it is explicitly accessible for both roles.

3.5.2 Sitemap Builder

The sitemap builder has two components: the *context-free grammar constructor* and the *link extractor*. With these two components, our analysis constructs a CFG for each explicitly reachable node, and extracts links embedded in the CFG to find outgoing edges of the node.

Context-Free Grammar Constructor

For each web page, our analyzer first parses the page into an Abstract Syntax Tree (AST), and then transforms the AST into an Intermediate Representation (IR), distinguishing every variable occurrence. Interested readers can refer to Wassermann's work [80] for more details.

To build a per-role CFG, our analyzer explores the IR only when necessary by predicting branch feasibilities with an inter-procedural path-sensitive analysis. It analyzes statements in the IR in a top-down manner, updating path conditions for both string constraints and arithmetic constraints. For arithmetic constraints, our analyzer resorts to the integrated Z3 to check the satisfiability of constraints; for string constraints, it feeds possible values of string variables and their aliases to our string constraint solver in exchange of answers. Our prototype string constraint solver supports string constraints which may contain multiple variables, regular expressions, equality and inequality operators, and checks on string lengths. We tried to solve string constraints with HAMPI [40], but it does not support multiple string variables yet. When constraints of a conditional is unsolvable, the analyzer explores both branches, updating path conditions for both the true branch and the false branch. For each function call, our analyzer first checks its calling context and then explores the function only when the context is new. Next, it propagates constraints on the arguments and related global variables of the function call. The IR exploration terminates when all possible branches have redirections or exits, indicating that none of the unexplored branches are feasible. In our implementation, we do not consider different contexts of page accesses and assume the parameters of HTTP requests to be Σ^* unless specified. In this way, we analyze each page only once, making our analyzer scalable at the expense of obtaining over-approximations of outgoing edges.

Finding the targets of PHP includes is a non-trivial task. It requires value resolution of possible string variables that are used for filename construction. Furthermore, it is necessary to find the directories that a PHP include file may reside in. When resolving PHP include paths, the following steps are performed in order:

- The `include_path` in the configuration of a PHP application is checked first;
- If no matching file is found under `include_path`, the directory of the calling script is checked;

```
function checkUser() {
  if (!isset($_SESSION["validUser"])
    || $_SESSION["validUser"] != true) {
    header("Location: login.php");
  }
}
checkUser();
sensitiveOperation();
```

Figure 3.6: An Example of Path Exploration.

- If no matching file is found in the directory of the calling script, the current working directory is checked;
- If no matching file is found in the current working directory, the inclusion finally fails.

We illustrate our basic exploration strategy with a simple example shown in Figure 3.6 based on one of the web applications that we have analyzed. Function `checkUser` checks whether an access should be allowed for a given user. Function `SensitiveOperation` will only be executed when the user has passed the access check. Suppose that $\$_SESSION$["validUser"] is a critical application state which determines the privileges of a role, and its value should be specified as **true** for role a and **false** for role b. Our analyzer explores the statements of the IR in order. Besides function definitions, the first statement it encounters is the function call `checkUser()`. Therefore, it retrieves the corresponding function body and continues from the first statement in the function. Because the first statement is an `if` statement, the analyzer attempts to solve the satisfiability of constraints to determine branch feasibilities. If the given role is b, only the true branch is feasible. As the true branch has a header redirection, the analyzer stops exploring the statements after this function call. Otherwise, when the role is a, only the false branch is feasible, and the analyzer continues exploring the statements after this function call, and eventually reaches function call `SensitiveOperation()`.

Path sensitivity prevents us from exploring infeasible paths. For example, suppose we have predicate $\$x > 1$ in the current path condition when the exploration reaches an `if` statement, the branch target of which depends on a conditional $\$x < 0$. To determine the feasibilities of the two possible branches, our analyzer sends two queries to Z3. The first query appends the new constraint to the existing path

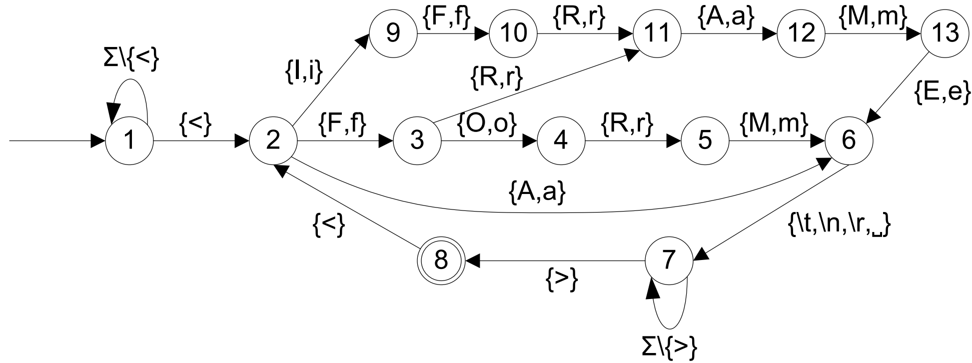

Figure 3.7: A Deterministic Finite Automaton Example.

condition, while the second query appends the negation of the new constraint to the existing path condition. Z3 will conclude that ($\$x > 1 \wedge \$x < 0$) is unsatisfiable, but ($\$x > 1 \wedge \neg(\$x < 0)$) is satisfiable. Thus, only the false branch is feasible and our analyzer will not explore the infeasible true branch of the `if` statement.

Link Extractor

Our link extractor extracts links to different web pages within a given web application. Since we are interested in constructing sitemaps, our link extractor filters links that point to pages outside of the application. We did not reuse the implementation from the previous work [80], which is based on the standard graph-reachability algorithm, but instead implemented the new link extraction algorithm shown in Figure 3.4 to eliminate the need of computing intermediate HTML outputs. As an example, Figure 3.7 shows an FA which matches anchor, form, frame and iframe tags in HTML outputs based on a simple regular expression:

```
/<([Aa]
     |[Ff][Oo][Rr][Mm]
     |[Ii]?[Ff][Rr][Aa][Mm][Ee]
  )\s[^>]*>/
```

We only show state-advancing edges in Figure 3.7 and omit state-resetting edges. In this FA, the start state $q_0 = 1$ and the final state set $Q_f = \{8\}$. For any FA state, a state-resetting edge directs

the current FA state back to the start FA state on input characters other than the ones shown on the state-advancing edges. We use the following simplified PHP code taken from one of our test subjects to show how our link extractor works.

```
echo "<div><a href="
    . $lang
    . ".php>Anchor</a></div>";
```

The above PHP code dynamically generates a link depending on the value of variable $\$lang$, which has three possible candidates: “english”, “spanish” and “french”. For this code, a CFG with five variables and seven grammar productions will be generated:

$S_0 \rightarrow S_1 S_2$

$S_1 \rightarrow$ “<div><a href=”

$S_2 \rightarrow S_3 S_4$

$S_3 \rightarrow$ “english” | “spanish” | “french”

$S_4 \rightarrow$ “.php>Anchor</div>”

In this CFG, $V = \{S_0, S_1, S_2, S_3, S_4\}$ and S_0 is the start symbol. Note that S_3 has three associated grammar productions separated by bars. For the algorithm in Figure 3.4, the link extraction starts with function call WALKVAR(S_0, 1, “”) (Line 38). Since S_0 maps to only one production, $RHS = \{[S_1 S_2]\}$ (Line 11) and our algorithm issues WALKSYMBOLS($[S_1 S_2]$, 1, “”) (Line 18). Then, it examines the symbols in list $[S_1 S_2]$ (Line 32) in order to derive the set of possible outcomes QW, the initial value of which is $\{\langle 1, “” \rangle\}$ (Line 31). Our algorithm sees that the first symbol S_1 is a variable and thus issues WALKVAR(S_1, 1, “”) (Line 27). For S_1, RHS ={“<div><a href=”} (Line 11), and the algorithm issues WALKSYMBOLS(“<div><a href=”,1,“”) (Line 18). Now our algorithm examines these terminals in order with function WALKTERMINAL. The first character is ‘<’, thus the algorithm transits the FA state from 1 to 2 along a state-advancing edge in Figure 3.7, and appends ‘<’ to w which is now “<”. The second character is ‘d’, thus the algorithm resets the FA state to the start state 1, and clears the matched

terminals in w. The third character is 'i', thus the algorithm stays at the FA start state 1, and w is still the empty string. Our algorithm continues like this and by the time it gets to variable S_3, the FA is in state 7 with w ="<a href=". For S_3, RHS ={"english", "spanish", "french"} (Line 11), and our algorithm walks these three elements one by one (Line 15). There are three possible outcomes, and thus the return value QW of WalkSymbols(S_3, 7, "< a href=") is {⟨7, "<a href=english"⟩, ⟨ 7, "<a href=spanish"⟩, ⟨7, "<a href=french"⟩} (Line 19). Our algorithm continues until all the seven grammar rules have been explored. Upon termination, it returns {"english.php", "spanish.php", "french.php"} (Line 39).

3.5.3 Vulnerability Detector

When the construction of per-role sitemaps is complete, our analyzer compares the two reachable node sets to infer privileged nodes. As HTML outputs presented to different roles are usually different, the set of privileged nodes is not empty in most cases. After obtaining the set of privileged nodes, our analyzer uses the same context-free grammar constructor again to approximate the outcomes of forced browsing. Finally, it compares derived redirection sets and the sizes of CFGs to determine whether forced browsing attemps are successful.

Even when forced browsing is successful, it is possible that the corresponding page does not contain any sensitive information or operations and is therefore considered safe. We observed that some pages used as file inclusions only contain function and class definitions. Such pages normally serve as inclusion files and are safe on their own. When the automatic vulnerability detection is over, we identify such safe pages with manual analysis, report them as false positives, and then mark the remaining pages as potentially vulnerable pages.

3.6 Empirical Evaluation

To evaluate the effectiveness and performance of our approach, we tested our tool on seven real-world PHP applications, two of which have patched versions. We picked these applications because they have reported vulnerabilities, which include injection vulnerabilities as well as access control vulnerabilities. The test subjects include both traditional web applications and Web 2.0 applications which use AJAX for

Subject	Files	LOC	
		PHP	HTML
SCARF	25	1,318	0
Events Lister	37	2,076	544
PHP Calendars	67	1,350	0
PHPoll	93	2,571	0
PHP iCalendar	183	8,276	0
AWCM	668	12,942	5,106
YaPiG	134	4,801	1,271

Table 3.1: Statistics on Evaluation Subjects.

client-server communications. The source code of all these PHP applications is publicly available. For each of the test subjects, we provide a specification file of at most ten lines. We ran all the tests on a PC with a quad-core CPU (2.40GHz) and 4 GB of RAM.

Our tool supports multiple roles and each role should have a set of distinctive application states. Typically, the administrator role has the most privileges; the normal user role has necessary privileges for common user operations; and the public user role has the least privileges. Although our tool can detect access control violations for any two roles, we chose to detect access control violations between administrators and normal users for two reasons. First, the operations and information that administrators can access are of greater importance than those that normal users can access. Second, it is often difficult for attackers to legally obtain administrator accounts, but easy to obtain normal user accounts.

Table 3.1 shows the total number of files as well as the lines of code for each web application. For the two web applications that have patched versions, we only list the statistics for the patched versions in the table. The lines of code in each application are counted for both PHP and HTML, excluding comments and empty lines. Our analysis translates HTML code into equivalent PHP `echo` statements.

3.6.1 Analysis Results

Table 3.2 shows the analysis results for the nine web applications. Note that we include two versions of SCARF and AWCM for vulnerability analysis. Columns "Vulnerable" and "FP" denote the numbers of

Project	Privileged	Vulnerable	FP	Guarded	Admin		Normal	
					Node	Edge	Node	Edge
SCARF	4	1	0	3	19	149	15	69
SCARF (patched)	4	0	0	4	19	149	15	69
Events Lister 2.03	9	2	2	5	23	113	14	26
PHP Calendars	3	1	0	2	19	35	19	30
PHPoll v0.97 beta	3	3	0	0	21	63	19	58
PHP iCalendar v1.1	1	0	0	1	51	292	50	292
AWCM v2.1	47	1	0	46	176	2,634	129	2,438
AWCM v2.2 final	47	0	0	47	180	2,851	133	2,612
YaPiG 0.95	11	0	0	11	54	260	44	154

Table 3.2: Vulnerability Analysis Results.

detected true vulnerabilities and manually confirmed false positives respectively. Column "Guarded" shows the number of privileged pages that are protected by access checks. The last four columns show numbers of explicitly reachable nodes and explicit edges in per-role sitemaps.

In summary, our tool found eight different access control vulnerabilities, four of which are previously unknown. It only has two false positives and correctly reports 119 guarded pages as not vulnerable. We manually confirmed all vulnerabilities and false positives on deployed web applications. In addition, the by-products of our analysis, the generated per-role sitemaps, provide high-level views of the test subjects and can be useful for understanding or modifying the structures of these web applications.

SCARF

SCARF is the Standford Conference And Research Forum. A critical access control checks whether the value of $\$_SESSION$["privilege"] equals "admin" in functions `is_admin` and `require_admin`.

Our tool detected a previously reported vulnerability (CVE-2006-5909). In this application, only users of role a are supposed to edit the configuration of the application in page "generaloptions.php". However, there is no access check for this edit privilege. Although the link is hidden from users of role b, they could still access and edit the configuration which affects the whole system. Our tool correctly reported the other three privileged pages "addsession.php", "editpaper.php" and "editsession.php" as guarded. Even if users of role b know the locations of these pages, forced browsing would fail because

of the presence of access checks in these pages. The latest version of SCARF fixed the vulnerability, and this is reflected in the vulnerability analysis result for SCARF (patched).

Events Lister

Events Lister is a PHP application that allows users to manage their events. Function `checkUser` implements an access control by checking whether $\$_SESSION$["validUser"] equals **true**.

Our tool found a new vulnerability in this application as well as a previously known one (CVE-2009-3168). We discovered that page "admin/setup.php" has no access checks and allows users of role b to repeatedly insert test events into the database of the application. It is even possible to create new tables in the database if none exists yet. The known vulnerability in page "admin/user_add.php" permits users of role b to add new users into the system. This privilege should only belong to users of role a. We consider the other two reports on privileged pages "admin/recover.php" and "admin/form.php" false positives. Page "admin/recover.php" allows users of role b to reset an administrator's password by sending a new password to the administrator's email address. Since only the administrator has access to her own email address, the password reset action does not pose any serious threats. Page "admin/form.php" contains an HTML form which is included in other container pages. On its own, this page does not expose any privileged operations or information, and is therefore considered safe. The notion of "safe" is sometimes a subjective matter. In a manual case study of another web application, we found that public users can view the list of all registered users with forced browsing. Such a list is also available for normal users and one can easily register for a normal user account. Consequently, it is unclear to us if the implicit access to the list of registered users is intended. As such, we would rather report such cases to developers for them to decide.

PHP Calendars

PHP Calendars is an online calendar management system. It protects privileged pages in the application by checking whether $\$_SESSION$["admin"] equals "yes" in page "admin/access.php".

Our tool detected a known vulnerability (CVE-2010-0380) in page "install.php" of this application.

The README file in this application warns administrators to delete this page after installation, but does not check if the file has indeed been deleted. If "install.php" exists in a deployed application, any users of role b could modify the configuration of the application by directly accessing this page. Because there is an explicit link to this page, we manually added this page to the privileged node set in the specification file. The other two privileged pages "admin/import.php" and "powerfeed.php" are not vulnerable. Note that N_a is not necessarily a superset of N_b. In this application, $|N_a| = |N_b|$, but $N_a \neq N_b$.

PHPoll

PHPoll is an online poll system where only users of role a can pass access checks by providing correct values of $\$_COOKIE[\$string_cook_login]$ and $\$_COOKIE[\$string_cook_password]$. Note that the cookie-based access controls are safe in this case because unauthorized users have no knowledge of valid cookie values.

Our tool detected three new access control vulnerabilities in this application and we manually confirmed them on a deployed application of PHPoll. All three pages have no access checks. The first page "modifica_configurazione.php" allows users of role b to modify login IDs and passwords, truncate the configuration table, and insert new entries into the configuration table of the application. The second page "modifica_votanti.php" lets users of role b delete votes or update polls stored in the MySQL database. The third page "modifica_band.php" does not prevent users of role b from reading, updating, or deleting poll results from the database with POST requests. These access control vulnerabilities pose serious threats to the security of the application, yet they have not been reported to the best of our knowledge.

PHP iCalendar

PHP iCalendar is another calendar application which displays calendar information to users. The only privileged page is "admin.php", and it is guarded by an access check which examines the value of $\$HTTP_SESSION_VARS$["phpical_loggedin"].

This application does not have any access control vulnerabilities. As Table 3.2 shows, users of role a can reach 51 pages which include "admin.php", while users of role b can only reach 50 pages which exclude "admin.php".

AWCM

AWCM (AR Web Content Manage system) differentiates role a from role b by determining whether $_*SESSION*["awcm_cp"] equals "yes" in a PHP include file "control/common.php".

Our tool detected a previously known vulnerability (CVE-2010-1066) in "control/db_backup.php" which dumps all the database information onto a web page. The cause of this access control vulnerability is that "control/db_backup.php" includes "common.php" instead of "control/common.php". Since access checks are only present in "control/common.php" but not "common.php", page "control/db_backup.php" is not guarded and can be accessed via forced browsing. Most pages in the "control" directory are intended for administrators only and our tool detected 47 privileged nodes in total. Our tool correctly recognized the access checks in the other 46 privileged pages and only reported "control/db_backup.php" to be vulnerable. The latest version of AWCM fixed the vulnerability, and this is reflected in the analysis result shown in Table 3.2. Although this application is AJAX-heavy, our tool covered nearly 80% of the active nodes, indicating that a majority of the links appear in PHP and HTML code which can be well handled with our tool.

YaPiG

YaPiG (Yet Another PHP Image Gallery) validates passwords and determines the privilege level of users with an access check in function `check_admin_login`.

An interesting thing about YaPiG is that all the five unreachable pages result from an uncovered execution path. In our implementation, we assume that an HTTP parameter \$*v* could have any values. Therefore, our tool infers that function call `isset($v)` returns **true** even if *v* is undefined. When a conditional depends on such a function call, the false branch is left unexplored. Our implementation does not yet support the specification of an optional value, which can either be defined or undefined.

Project	Nodes			Context-Free Grammar		Coverage	Time (s)
	Entry	Active	Orphan	Variables	Productions		
SCARF	1	19	0	158	719	100.00%	6.02
SCARF (patched)	1	19	0	159	719	100.00%	6.01
Events Lister v2.03	4	23	5	100	2,083	100.00%	3.84
PHP Calendars	3	15	0	48	255	80.00%	5.09
PHPoll v0.97 beta	5	21	6	115	224	100.00%	4.26
PHP iCalendar v1.1	2	52	2	811	4,774	90.38%	760.62
AWCM v2.1	17	208	22	410	422	79.33%	89.48
AWCM v2.2 final	16	209	14	451	484	79.90%	108.51
YaPiG 0.95	7	59	3	332	532	91.53%	208.38

Table 3.3: Coverage and Performance Results.

3.6.2 Performance Evaluation

In our evaluation, we collect links that point to files within an application, excluding those that point to CSS files which are of no interest to us. Currently, we treat PHP, HTML and XML files to be active nodes and analyze them to extract links. A page can contain links to both active nodes and inactive nodes. Although inactive nodes do not provide sensitive operations, they may contain sensitive information and therefore should also be checked.

Table 3.3 shows the coverage and performance of our tool. Column "Entry" shows the number of specified entry nodes for each application. Column "Active" lists the number of all active nodes. Column "Orphan" lists the number of specified *orphan nodes* which are non-entry active nodes with no incoming edges. Column "Coverage" lists the coverage of our tool on active nodes in an application, excluding orphan nodes. We list the average numbers of variables and grammar productions of all CFGs for each web application. Note that the numbers are counted on CFGs that have been simplified with grammar-reachability analysis. The last column shows the total analysis time spent for each application in terms of seconds.

Active nodes may have outgoing edges and may not have any incoming edges. An active node with no incoming edges can be optionally specified as either an entry node or an orphan node. When it is specified as an entry node, it is analyzed in the sitemap construction process to find outgoing edges; when it is specified as an orphan node, which indicates that this node should be outside any sitemaps,

Project	Time (s)		
	Admin Sitemap	Normal Sitemap	Forced Browsing
SCARF	3.15	1.70	1.15
Events Lister	2.29	1.00	0.53
PHP Calendars	1.81	1.67	1.61
PHPoll	2.39	1.54	0.33
PHP iCalendar	371.28	370.85	18.46
AWCM	55.36	49.11	3.85
YaPiG	85.59	44.91	77.86

Table 3.4: Analysis Time.

it is excluded from the coverage calculation; when it is unspecified, it may affect the coverage result. Let $Active$, $Orphan$ and $Reachable$ denote the sets of all active nodes, specified orphan nodes and explicitly reachable nodes respectively. We calculate the coverage as:

$$Coverage = \frac{|Reachable|}{|Active| - |Orphan|}$$

In our evaluation, we conservatively identify orphan nodes with a simple manual analysis and the obtained orphan sets may be incomplete, especially for large and complex applications. Therefore, the real coverages of our analysis might be better than the ones shown in the table because uncovered nodes might indeed be unreachable.

Our static analyzer achieved good coverage of active nodes: 100% for four applications, about 90% for two, and about 80% for the remaining three. The total analysis time listed in Table 3.3 demonstrates that our approach is scalable. For the smaller test applications SCARF, Events Lister, PHP Calendars and PHPoll, our tool finished within seven seconds; for the largest test application AWCM, our tool took less than two minutes to analyze the active nodes in the whole application. The analysis time for iCalendar is the longest because of the inlining of dynamic PHP files and the complexity of PHP code. As can be seen in Table 3.3, the number of grammar productions for PHP iCalendar is also the largest. We show the break down of analysis time in Table 3.4. Columns “Admin Sitemap” and “Normal Sitemap” list the time spent on constructing the sitemaps for roles a and b respectively. Column “Forced Browsing”

shows the time spent on detecting access control vulnerabilities via forced browsing. It is obvious from the data in the table that building sitemaps consumes the majority of the analysis time.

3.6.3 Discussions

As we mentioned earlier, our prototype did not find all kinds of links in web applications. The major reason is that our prototype did not identify all the links generated by JavaScript code or HTML templates, or those constructed with unresolvable string variables. Extracting links from JavaScript code is especially challenging because of the dynamic features of the JavaScript language. Our prototype works better on traditional web applications than AJAX-heavy ones. Incorporating JavaScript analysis could possibly improve the coverage. Furthermore, our test applications may not be representative of general web applications.

What a node represents determines the granularity of the analysis. Our prototype treats a web page as a node, but the general approach still applies when the granularity is refined to functionalities within a page. Performing the analysis at a refined granularity would be especially useful for complex web pages which contain multiple functionalities within a single page. The techniques proposed by Halfond *et al.* [33] could be used to identify important parameters in web applications to distinguish functionalities. Because a privilege is often granted with a set of atomic database operations, advancing the granularity to the level of database operations might be too fine-grained.

Our prototype does not handle all object-oriented features in PHP. This prevents us from parsing some PHP pages in large PHP applications. We leave it as future work to enhance our static analyzer for additional object-oriented features of the PHP language.

The current implementation of the string constraint solver is rudimentary. For either unsolvable constraints or non-determinism in a conditional, we conservatively explore both branches. This might lead to false negatives when infeasible paths for a less privileged role are explored. For access checks that involve non-determinism, such as password-based authentication and CSRF protection that uses random tokens, we rely on role-based specifications to determine which execution paths to explore. Non-determinism affects path explorations but not link extractions. Furthermore, when Assumption 3.2

does not hold, we would also have false negatives introduced by explicit accesses to privileged nodes.

Our tool generated false positives. Even when access checks are missing in hidden pages, these pages may not contain any sensitive information or operations and are therefore safe to access for any role in the application. We manually examined the analysis results and marked such safe pages as false positives.

3.7 Related Work

In this section, we discuss the most relevant work, including specification inference, workflow violation detection, privilege separation based on user roles, language-based approaches to secure web applications, and program analysis for web security.

The capability of automated tools in detecting vulnerabilities or bugs can only be as good as the specifications given to them. Since manually writing specifications is tedious, time-consuming and error-prone, a wide range of techniques have been proposed to automatically infer specifications from the source code of programs. For intrusion detection, Wagner and Dean [74] apply static analysis to derive a model of normal application behavior as an oracle. Based on the observation that bugs are deviant behavior [26], researchers have proposed probabilistic-based approaches [43, 69] to infer specifications from applications. However, without taking into account of roles in web applications, it is difficult to infer privileged pages which are only intended for a group of users.

Recently, workflow violations have attracted the interests of researchers. Nemesis [22] uses dynamic information flow tracking to detect authentication and access control vulnerabilities in web applications. It requires developers to specify access control lists for resources. Similarly, Hallé *et al.* [35] proposed a runtime enforcement mechanism to only allow navigations that conform to a state machine model specified by developers. Researchers have proposed various techniques to automatically infer correct workflows. Swaddler [20] first learns internal states of web applications, and then detects abnormal state violations at critical points. Targeting the detection of AJAX intrusion attacks, Guha *et al.* [31] leverage static analysis on client-side JavaScript code to infer expected server-side behavior. To detect multi-module vulnerabilities, MiMoSA [7] takes into account the interactions of different web pages.

However, it is not always easy to distinguish an intended path from an unintended one because of flexible navigation paths that web applications allow. Its follow-up work Waler [27] uses a combination of dynamic analysis and symbolic model checking to first infer invariants from dynamic program executions, and then report violations of the invariants as logic vulnerabilities. From a high-level view, the likely invariants that Waler generates with heuristics are subject to errors. Furthermore, the inferred invariants may not always hold due to the limited coverage of dynamic analysis. Access control vulnerabilities can be considered a special case of workflow vulnerabilities where cross-role workflow assumptions are violated. Cross-role comparisons allow us to precisely reason about privileged pages in most cases.

To reduce least-privilege incompatibilities, researchers distinguish different user roles and separate privileges based on different roles. Aiming at identifying dependencies on `admin` privileges in traditional software applications, Chen *et al.* [14] run applications without `admin` privileges and collect dynamic execution traces. We take a step further and use roles to represent sets of privileges in web applications. In our setting, roles form a lattice and its height is not limited. To reduce developer's burden on securing web applications, the CLAMP project [60] prevents leakage of sensitive information by restricting the flows of user data and isolating the authentication module of an application. While they also minimize developers' effort, they secure web applications by modifying application code at critical points. Web application vulnerability scanners can also automatically detect access control vulnerabilities. However, they often build shallow and incomplete sitemaps, missing deep and invisible pages that are only accessible when valid form data are submitted. This undermines the capabilities of web scanners in both discovering privileged nodes as well as successfully performing forced browsing with valid form data.

Previous work has proposed language-based approaches to secure web applications in a principled way. SIF [16] accepts specifications either as program annotations at compile time, or as user requirements at run time to guarantee confidentiality and integrity with information flow analysis. Recently, Krishnamurthy *et al.* [44] presented an object-capability language for fine-grained privilege separation for web applications. Unfortunately, theses language-based approaches do not apply to the large set of legacy code that is not written in the newly designed languages.

In the past few years, researchers have focused their attention on detecting injection vulnerabilities in web applications with both static analysis [49, 50, 65, 70, 79, 80, 81, 83] and dynamic analysis [8, 12, 57, 63]. Similar to our static analyzer, Pixy [38] is also a static analyzer built to analyze PHP applications. It takes advantage of taint analysis to detect injection vulnerabilities with specifications on taint sources and sinks. Its implementation hinders it from scaling to large applications as Pixy has no support for include resolution and object-oriented features.

3.8 Conclusion

Developers should enforce access controls throughout web applications for every privileged page. This chapter proposes a novel approach to detect access control vulnerabilities in web applications with minimal manual effort. Based on the observation that sitemaps presented to different roles are not identical, our analysis first automatically infers the set of privileged pages from the source code of a web application, and then detects access control vulnerabilities via forced browsing. We added support for role-based specification rules, and integrated constraint-solving capabilities with our static analyzer to systematically explore program paths. Our tool is able to achieve good coverage and scale to real-world applications. The evaluation results demonstrate that it is capable of detecting both unknown and known access control vulnerabilities in unmodified web applications with only a few lines of specifications. For future work, we plan to support additional language features of PHP, enhance the string constraint solver, and scale the analysis to larger web applications.

Chapter 4

Detecting Logic Vulnerabilities in E-commerce Applications

E-commerce has become a thriving business model. With easy access to various tools and third-party cashiers, it is straightforward to create and launch e-commerce web applications. However, it remains difficult to create *secure* ones. While third-party cashiers help bridge the gap of trustiness between merchants and customers, the involvement of cashiers as a new party complicates logic flows of checkout processes. Even a small loophole in a checkout process may lead to financial loss of merchants, thus logic vulnerabilities pose serious threats to the security of e-commerce applications. Performing manual code reviews is challenging because of the diversity of logic flows and the sophistication of checkout processes. Consequently, it is important to develop automated detection techniques.

This chapter proposes the first *static detection* of logic vulnerabilities in e-commerce web applications. The main difficulty of automated detection is the lack of a general and precise notion of *correct payment logic*. Our key insight is that secure checkout processes share a *common invariant*: A checkout process is secure when it guarantees the *integrity* and *authenticity* of critical payment status (order ID, order total, merchant ID and currency). Our approach combines symbolic execution and taint analysis to detect violations of the invariant by tracking tainted payment status and analyzing critical logic flows among merchants, cashiers and users. We have implemented a symbolic execution framework for PHP. In our

evaluation of 22 unique payment modules, our tool detected 12 logic vulnerabilities, 11 of which are new. We have also performed successful proof-of-concept experiments on live websites to confirm our findings.

4.1 Introduction

E-commerce web applications, a special type of web applications designed for online shopping, play an important role in the modern world. The U.S. Census Bureau of the Department of Commerce estimated that U.S. retail e-commerce sales for the second quarter of 2013 reached $64.8 billion, an 18.4% increase from the previous year [71]. The prevalence of Internet and the rise of smart mobile devices contribute to the rapid growth of e-commerce web applications. Unfortunately, the complexity of e-commerce applications and the diversity of third-party cashier APIs make it difficult to implement perfectly secure checkout processes. Since logic attacks are tied directly to financial loss and merchant embarrassment, the impact of logic vulnerabilities in e-commerce applications is often severe.

Business or application logic refers to *application-specific* functionality and behavior. Besides general functionality (such as user authentication), each application has its unique handling of user inputs, user actions and communications with third-party components. A report by WhiteHat Security lists seven examples of logic flaws [30]. In contrast to a bug which typically prevents an application from doing what is intended, a logic vulnerability typically exists when an attacker abuses legitimate application-specific functionality against developers' intentions via unexpected user inputs or actions [18]. Although logic vulnerability is not the most common type of web vulnerabilities, it often has serious impact and is easily exploitable.

Logic vulnerabilities in e-commerce applications, being a *subset* of general logic vulnerabilities, allow attackers to purchase products or services with incorrect or no payment at the expenses of merchants. Developers often make assumptions about what user inputs are and how users navigate web pages in checkout processes. However, when such assumptions do not hold and developers fail to implement proper security checks, attackers can exploit logic vulnerabilities in e-commerce applications for financial gains. CVE-2009-2039 [17] describes our motivating example where Luottokunta (version

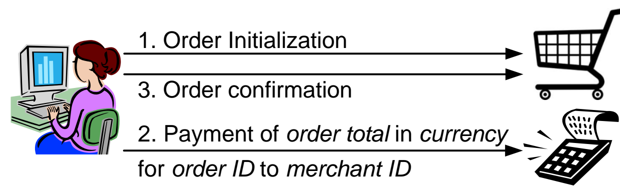

Figure 4.1: Logic Flows in E-Commerce Web Applications.

1.2), a payment module in the osCommerce software [1], has a logic vulnerability that allowed attackers to tamper with order ID, order total and merchant ID. The latest version of Luottokunta (version 1.3) was released to patch this vulnerability by adding logic checks on some components of payment status. However, upon close examination, we were surprised to discover that it is still vulnerable. The added check on order ID is insufficient, thus attackers can pay for one order and bypass payments for future orders. This is one of the new vulnerabilities that we detected.

The use of third-party cashiers in e-commerce applications introduces new security concerns even if the cashiers themselves are secure. For flexibility, a modern web application often presents several payment options during checkout by using one payment module for each third-party cashier. However, the integration of cashiers also increases the complexity of logic flows in checkout processes. Figure 4.1 illustrates three critical steps in a typical checkout process that involves a merchant, a cashier and a user: 1) order initiation on the merchant's server, 2) payment transaction on the cashier's server, and 3) order confirmation on the merchant's server. In the first step, the merchant initiates the basic payment information of an order. From then on, both the merchant and the cashier track the payment status of the order. Ideally, the merchant should either explicitly check *every* component of important payment status or directly communicate with the cashier. In practice, miscommunications between the merchant and the cashier may harm the integrity or authenticity of payment status. Insufficient or missing logic checks on payment status can allow an attacker to skip the second step or carry it out incorrectly. In a successful attack, the merchant is led to believe mistakenly that the order has been paid in full, while the cashier actually receives no payment or partial payment.

With the goal of confirming the real dangers that logic vulnerabilities in e-commerce applications pose, we designed responsible proof-of-concept experiments following the example set by

Figure 4.2: Received Products from Vulnerable Websites.

Wang *et al.* [77]. Each experiment was performed on a live website that used a vulnerable payment module. Specifically, we received three products (Figure 4.2) from three websites which integrate vulnerable payment modules. First, for payment module RBS WorldPay, we received a Ubuntu notebook from the Ubuntu online shop by Canonical Ltd. We paid less by changing the currency from British pounds to U.S. dollars. Second, for payment module Authorize.net Credit Card SIM, we received a diaper game package from a baby products online shop. We paid nothing by replaying tokens from a previous order. Third, for payment module PayPal Standard, we received three chocolate pieces from a California chocolate online shop. We paid nothing to the merchant by changing the merchant ID from the chocolate shop owner's ID to our ID. After having received the products, we immediately compensated the three merchants for the respective correct full amounts. These experiments clearly demonstrate that insecure uses of third-party cashiers, such as the heavily vetted cashier PayPal, may give merchants a false sense of protection.

The detection of logic vulnerabilities in e-commerce applications is challenging for both manual and automated analyses since any weak link in a checkout chain can result in a logic vulnerability. On one hand, manual code review is time-consuming and error-prone. Security analysts often spend much time understanding different logic flows in an e-commerce application before examining security checks of payment status. In contrast, most payment module developers are familiar with logic flows but not various attack vectors. In either case, a thorough manual code review of all possible logic flows in a checkout process is a nontrivial task. On the other hand, automatic code scanners cannot detect logic vulnerabilities without the knowledge of application-specific business context. E-commerce applications have various application-specific logic flows and each payment method has its unique APIs and security

checks. Consequently, it is challenging to create general rules to automate the detection process.

Researchers have proposed various techniques to detect different logic vulnerabilities, including abnormal logic behavior [27], multi-module vulnerabilities [7] and single sign-on vulnerabilities [78, 84]. Each technique targets a particular domain of logic vulnerabilities and checks web applications against specifications in the given domain. Wang *et al.* [77, 84] are the first to perform security analysis on Cashier-as-a-Service based e-commerce applications. They found several serious logic vulnerabilities in a few popular e-commerce applications via manual code reviews [77] and proposed a proxy-based approach to dynamically secure third-party web service integrations which include the integration of cashiers [84].

In this chapter, we propose the first *static detection* of logic vulnerabilities in e-commerce applications. Our key observation is that an invariant must be verified to secure a payment: A merchant M should accept an order O from a user if and only if the user has actually made a payment to the cashier in the correct amount and currency for that specific order O associated with merchant M. Based on this observation, we designed a symbolic execution framework that explores critical control flows exhaustively, tracking taint annotations for the critical components of payment status (order ID, order total, merchant ID and currency) and exposed signed tokens. Our main contributions are:

- We give an application-independent invariant for detecting logic vulnerabilities in e-commerce web applications. We found a new attack vector: Tampering with currency.

- We propose the first static analysis to detect logic vulnerabilities in e-commerce applications based on symbolic execution and taint tracking of payment status.

- We implemented a scalable symbolic execution framework for PHP web applications. Our analyzer systematically explores control flows to examine logic flows in checkout processes.

- We evaluated our tool on 22 unique real-world payment modules from various cashiers. We detected logic vulnerabilities in 12 out of 22 payment modules and performed responsible experiments on live websites. Of the 12 detected vulnerabilities, 11 vulnerabilities are new. The evaluation results show that our approach is effective and scalable.

The rest of the chapter is organized as follows. We first give an example to illustrate the main steps of our approach (Section 4.2). Section 4.3 describes our detailed algorithm and approach. Section 4.4 presents the implementation of the automated analyzer we developed, and Section 4.5 shows the vulnerability report, the details of our experiments on live websites and the performance of our tool on real-world e-commerce payment modules. Finally, we survey related work (Section 4.6) and conclude (Section 4.7).

4.2 Illustrative Example

This section uses payment module Luottokunta (version 1.3) to illustrate the major steps of our approach. This module, which patched the vulnerability described in CVE-2009-2039 [17], is still vulnerable because of an insufficient check on untrusted order ID. During checkout, a user sends out the following four critical HTTP requests, the last two of which are redirections from HTTP responses with a status code of 302:

```
R1. User > Merchant(checkout_confirmation.php)
R2. User > Cashier(https://dmp2.luottokunta.fi)
R3. User > Merchant(checkout_process.php), 302
R4. User > Merchant(checkout_success.php), 302
```

With this payment module, a merchant can integrate the service of third-party cashier Luottokunta. Of the four requests, the second one is sent to the cashier and the rest are sent to the merchant. The first request (R1) initializes the checkout process for an order when the user navigates to page `checkout_confirmation.php`. The second request (R2) lets the user pass on the order information generated by the merchant to the cashier. After the user has completed the payment transaction on the cashier's server, the cashier sends the user a response that redirects the user to page `checkout_process.php` (R3) on the merchant's server to process the order. If the order is accepted, the merchant redirects the user to page `checkout_success.php` (R4).

Our symbolic execution starts from the first merchant page `checkout_confirmation.php` in the checkout process. To model the first request (R1), it symbolically executes the intermediate representation (IR) of this page and simultaneously parses its symbolic HTML output in search of critical

HTML form elements. The analysis eventually finds an HTTP form that serves as a communication channel between the merchant and the cashier. Its elements record the order information and its action URL points to the cashier's URL (`https://dmp2.luottokunta.fi`). This form also contains a return URL (`checkout_process.php`), which will be used by the cashier to redirect the user back to the merchant's server once a payment transaction has been completed.

Since our analysis treats cashiers as black boxes that work correctly, we assume that the cashier would properly complete the payment transaction with the user (R2) and redirect the user back to the merchant (R3). To continue exploring logic flows, our analyzer symbolically executes page `checkout_process.php` which is a part of the return URL. A thorough examination requires the modeling of all possible responses from the cashier. Therefore, we use the symbolic *top* value ($\top$), *i.e.*, the most conservative value that denotes any possible value, for the request variables of R3. Our analyzer first propagates the end execution states from the previous page `checkout_confirmation.php` to the current page `checkout_process.php`, and then symbolically executes the IR of the current page. The execution eventually reaches function `before_process()` which has the following checks on payment status:

```
function before_process() {
  if (!isset($_GET['orderID'])) {
    tep_redirect(FILE_PAYMENT);
  } else {
    $orderID = $_GET['orderID'];
  }

  $price = $_SESSION['order']->info['total'];
  $tarkiste = SECRET_KEY
    . $price . $orderID . MERCHANT_ID;
  $mac = strtoupper(md5($tarkiste));

  if (($_POST['LKMAC'] != $mac)
      && ($_GET['LKMAC'] != $mac)) {
    tep_redirect(FILE_PAYMENT);
  }
}
```

Because request variable `$_GET['orderId']` has a symbolic top value, both branches of the first

`if` statement are feasible. For the `true` branch, the user is redirected to merchant page `FILE_PAYMENT`. This redirection forms a backward flow, which does not contribute to the detection of logic vulnerabilities. Therefore, this backward logic flow is automatically discarded. For the `false` branch, an `MD5` value is calculated and stored in variable `$mac`. Note that the value of `$orderID` used in the calculation comes from an untrusted request variable `$_GET['orderId']` which is under attackers' control.

Our taint analysis tracks the components of critical payment status across logic flows in the checkout process. Initially, order ID, order total, merchant ID, currency and secret are all tainted. *Secret* refers to an unpredictable value that only the merchant and the cashier know. Therefore, the cashier can use it to sign messages. For taint manipulation, we have a set of rules. One rule removes a taint annotation when a conditional check verifies an untrusted value against a trusted component. For the last conditional in function `before_process()`, we have the following symbolic constraints for the `false` branch:

```
[ or
  ($_POST['LKMAC'] = strtoupper(md5(SECRET_KEY
    . $_SESSION['order']->info['total']
    . $_GET['orderID'] . MERCHANT_ID)));
  ($_GET['LKMAC'] = strtoupper(md5(SECRET_KEY
    . $_SESSION['order']->info['total']
    . $_GET['orderID'] . MERCHANT_ID)));
]
```

Among the symbolic values in the above constraints, `$_SESSION['order']->info['total']` is a trusted session value, while `MERCHANT_ID` and `SECRET_KEY` are trusted constants defined in the merchant's database. This conditional check guarantees that the cashier has received a payment in full on behalf of the merchant. Therefore, our analyzer removes the taint annotations of order total, merchant ID and secret. In contrast, `$_GET['orderId']` is an untrusted request variable, and there is no check for currency.

After exploring `before_process()`, the symbolic execution eventually redirects the user to the final page `checkout_success.php` (R4). When the symbolic execution reaches this page, it means that the checkout process is complete and our analysis generates a final vulnerability report. In the report of this payment module, order ID and currency are still tainted, indicating that this module is vulnerable to two types of logic attacks. The first type of attacks allows an attacker to pay for one order

and avoid payments of future orders by replaying the value of `$_POST['LKMAC']` or `$_GET['LKMAC']` of the paid order. Note that the attacker can easily intercept the value of `$_POST['LKMAC']` or `$_GET['LKMAC']` of any paid order by changing the return URL to her own choice in R2. The second type of attacks allows an attacker to pay less by paying the cashier the correct amount indicated by `$_SESSION['order']->info['total']`, but in a different currency. For example, the merchant might list product prices in European euros, but an attacker can pay in U.S. dollars instead. [1]

4.3 Approach

This section presents our high-level approach. We first define logic vulnerability in e-commerce applications, lay out our assumption, and then describe the core algorithm of our approach.

4.3.1 Definitions

Definition 4.1 (Merchant). A merchant accepts an order when it has been properly paid through a cashier by a user. Merchant is the central role in e-commerce applications, responsible for initializing orders, tracking payment status, recording order details, finalizing orders, shipping products and providing services to users. During a checkout process, a merchant co-ordinates communications between a user and a cashier.

Definition 4.2 (Cashier). A third-party cashier accepts the payment of an order from a user on behalf of a merchant. Cashiers bridge the gap between merchants and users when they lack mutual trust. Users trust cashiers with their private information; merchants expect cashiers to correctly charge users.

Definition 4.3 (User). A user initiates a checkout process on a merchant's website, chooses a trusted cashier, makes a payment to the cashier and receives products or service from the merchant. User inputs and actions drive the logic flows of a checkout process. Some users are malicious, and it is the merchant's responsibility to check against untrusted user inputs and actions.

[1] This attack can be launched when the cashier accepts multiple currencies for payments.

Definition 4.4 (Logic Flow). Logic flows in e-commerce applications often involve three types of parties: merchant node, cashier node and user. Note that one merchant web page may be divided into multiple merchant nodes based on runtime values of a request. For instance, one page may perform an "insert", "update" or "delete" operation depending on the value of `$_GET['action']`. A logic flow serves as a communication channel between two parties, and any flow in a checkout process may influence payment status. Our analysis starts at the beginning merchant node n_0 of a checkout process and ends at the destination merchant node n_k where an order is accepted, tracking taint annotations of payment status and signed tokens across logic flows. Suppose for any valid node n_i in the checkout process, we start the analysis of n_i with execution state set Q_i. At the end of the analysis of n_i, we would have execution state set Q_j and a node to be visited next, namely n_j. Formally, logic flows in an e-commerce application can be represented as $\Pi = \{(n_i, Q_i) \rightarrow (n_j, Q_j) \mid 0 \leq i, j \leq k\}$.

Definition 4.5 (Logic State). A logic state, which is included in a symbolic execution state, consists of taint annotations and links to other valid nodes of a checkout process. The propagation of logic states reflects the changes of payment status. Specifically, for any order that a user places on a merchant's website with the integration of a third-party cashier, a logic state stores the taint annotations for the following payment status components and exposed signed tokens:

- ***Order ID***. The identifier of the current order that should be paid for.
- ***Order total***. The total amount that the cashier should receive from the user on behalf of the merchant.
- ***Merchant ID***. The identifier for the merchant who is selling products or service. The cashier will ultimately transfer the received money from the user to the merchant.
- ***Currency***. The currency (system of money) in which the payment for the order should be paid.
- ***Exposed signed token***. An encrypted value that is signed with a secret between the merchant and the cashier. It can act as a cashier's signature and is considered exposed when it is visible to users in the Document Object Model (DOM) tree of a merchant page.

Definition 4.6 (Logic Vulnerability in an E-commerce Application). Inspired by and developed from an extensive study of cashier documentation, open-source e-commerce applications and related work [77, 84], we consider a payment secure when the integrity and authenticity of payment status are ensured. Tampering with currency is a new attack vector we discovered in our study. A *logic vulnerability in an e-commerce application* exists when for any accepted order ID, the merchant cannot verify that the user has correctly paid the cashier the amount of order total in the expected currency to merchant ID.

Assumption 4.1: *Third-party cashiers are secure.* We treat third-party cashiers as black boxes and assume that they are perfectly secure. Most third-party cashiers' source code is unavailable, but many cashiers have been vetted heavily. The security of a third-party cashier does not correlate with the security of its integration in an e-commerce application. Developers of payment modules are often less security-conscious than those of cashiers, and payment modules are generally more prone to logic vulnerabilities.

Given a logic vulnerability in an e-commerce web application, it is easy to launch attacks on live websites. Simply using a browser extension, attackers can withhold an HTTP request, modify a request or forge a request from scratch. Moreover, attackers can exploit a signed token to pose as a cashier, reuse payment information from previous orders or intercept cashiers' responses by changing return URLs of HTTP forms.

In summary, a logic vulnerability in an e-commerce application is caused by the following five types of taint annotations:

- ***Tainted order ID***. To avoid payment for an order, attackers can replay the payment information of a previous order from the same merchant. As long as the order total and currency of the previous order match the ones of the current order, the check would pass because order ID is not verified.
- ***Tainted order total***. Attackers can pay an arbitrary amount for an order by tampering with the order total sent to a cashier if the order total is not verified. A payment to the cashier is still necessary for the order to be accepted.
- ***Tainted merchant ID***. When merchant ID is tainted, an attacker can set up her own merchant

account with the same cashier and send a payment to herself instead of the merchant for the order placed on the merchant's website. Note that a check on the secret between the merchant and the cashier can replace a check on merchant ID because the secret is a unique verifiable value set by the merchant.

- ***Tainted currency***. For cashiers that accept multiple currencies, it is possible to pay less for an order in a different currency without changing the order total.

- ***Exposed signed token***. An exposed signed token invalidates any security checks against trusted symbolic values since a signed request may not actually come from a trusted cashier. It may be used in a forged request by an attacker.

4.3.2 Automated Analysis

Section 4.3.2 presents our detection algorithm which explores critical logic flows in e-commerce applications among three parties (merchant, cashier and user). Section 4.3.2 describes taint manipulation rules which reflect changes of payment status.

Logic Vulnerability Detection Algorithm

Figure 4.3 presents our vulnerability detection algorithm which forms the core of our approach. It integrates symbolic execution of merchant nodes and taint analysis, and connects individual nodes to explore valid logic flows in e-commerce applications. We have four possible pairs of HTTP requests from the client side to the server side: (user, merchant), (user, cashier), (cashier, merchant) and (merchant, cashier). An attacker may skip user-to-cashier requests, but still send the same number of requests to the merchant to carry out all necessary steps of the checkout process. Consequently, each merchant node in the checkout process is analyzed in order.

Function DetectVuls is the main function of our analysis algorithm. Function AnalyzeNode analyzes each merchant node individually, while function GetNextNode connects nodes together for valid logic flows. The analysis begins from start node n_s with a start execution state q_s. An execution state q contains a logic state, memory maps for the values of global and local variables, alias information etc.

```
DetectVuls(Spec)
 1  n_s ← GetStartNode(Spec)
 2  q_s ← InitState(Spec)
 3  q_s ← AddComm(user, Merchant(n_s), q_s)
 4  Q_f ← AnalyzeNode(n_s, q_s, ∅, Spec)
 5  Vuls ← CheckLogicVuls(Q_f)
 6  return Vuls

AnalyzeNode(n, q, Q_f, Spec)
  1  n_f ← GetFinalNode(Spec)
  2  if n = n_f
  3     then Q_f ← Q_f ∪ {q_f}
  4           return Q_f
  5  q ← PropagateNodeState(n, q)
  6  Q ← SymbolicExecution(n, q, Spec)
  7  for each q_i in Q
  8  do ⟨n′, q_i⟩ ← GetNextNode(q_i, Spec)
  9      Q_f ← AnalyzeNode(n′, q_i, Q_f, Spec)
 10  return Q_f

GetNextNode(q, Spec)
  1  n ← ResetRedirection(q)
  2  if n = null
  3     then n ← ResetFormAction(q)
  4  if IsCashier(n, Spec)
  5     then q ← AddComm(user, Cashier(n), q)
  6           n ← ResetCallbackUrl(q)
  7           if n = null
  8              then n ← ResetReturnUrl(q)
  9  q ← AddComm(user, Merchant(n), q)
 10  return ⟨n, q⟩
```

Figure 4.3: Algorithm for Vulnerability Detection.

The first logic flow to be analyzed is $(user, \textsc{Merchant}(n_s))$. Our analysis then continues until all valid logic flows are explored. Finally, for each execution state q_f in the final execution state set Q_f, function CheckLogicVuls checks the logic state in q_f and reports any detected logic vulnerabilities.

Function AnalyzeNode recursively analyzes merchant nodes of valid logic flows until the final node n_f is reached. The final execution state set Q_f is only updated when a new final execution state q_f has a uniquely new logic state. The reason behind this update strategy is that other information in an execution state has no impact on the final vulnerability result. Function PropagateNodeState propagates an execution state q from one merchant node to another, performing a few operations on q. Specifically, this function updates runtime constants such as `$_SERVER['PHP_SELF']`, updates array `$_GET` for the query string of the new merchant node, updates array `$_POST` for the form elements of the previous merchant node, and resets the memory map of local variables. By default, request variables have the symbolic top value, which represents all possible values including `null`. A merchant node n is symbolically executed via function SymbolicExecution, and Q is the end execution state set for n. During symbolic execution, HTML form action URLs, form elements and parameters for `cURL` [2] merchant-to-cashier requests are monitored in search of links to other merchant nodes or cashier nodes.

Function GetNextNode examines four types of links to connect nodes for valid logic flows: redirection URL, form action URL, callback URL and return URL. A redirection URL or a form action URL can point to either a cashier node or a merchant node, while a callback URL or a return URL can only point to a merchant node. To navigate only along valid logic flows, we discard URLs that form backward or self-cycle logic flows, and URLs that are irrelevant to checkout. Each reset function within function GetNextNode stores in n the current value of a particular type of URL, and resets the URL to `null`. Header redirection URL (obtained via function ResetRedirection) and form action URL (obtained via function ResetFormAction) are checked first. When URL n points to a cashier node, a logic flow (user, $\textsc{Cashier}(n)$) is added. We then model the cashier's response by checking callback URL (obtained via function ResetCallbackUrl) and return URL (obtained via function ResetReturnUrl). Note that both callback URL and return URL can only be set after a cashier has been visited. Callback URL is optional and can be visited first by the cashier to notify the merchant that a payment transaction has been

[2] http://curl.haxx.se/

established. The cashier must send a request to the return URL to inform the merchant that a payment has been completed and the order should be accepted. The return value of function GetNextNode is a pair of merchant node n that should be visited next and the updated state q.

Taint Rules

To keep track of the integrity and authenticity of payment status, we designed a few taint manipulation rules. The integrity of payment status can thwart HTTP parameter tampering attacks, and the authenticity of payment status defends against forged payment status which is coined with predictable or exposed values of request variables. Untainted order ID, order total, merchant ID and currency ensure the integrity, while no exposure of signed tokens ensures the authenticity. The underlying assumptions of the taint rules are: 1) requests from users are untrusted; 2) unsigned cashier requests sent via insecure channels are untrusted; and 3) cashier responses that are relayed by users to merchants via HTTP redirection (status code 302) are also untrusted. Initially, order ID, order total, merchant ID and currency are all tainted.

When a merchant correctly verifies a payment status component, the taint annotation should be removed for the component. Our approach uses three taint removal rules for the following cases:

- ***Conditional checks***. When an (in)equality conditional check verifies an untrusted value against a trusted symbolic value of payment status component, remove taint from the payment status component.

- ***Writes to merchant database***. When a tainted value is written into the merchant's database with `INSERT` or `UPDATE` queries, conservatively remove taint from the component. Before a merchant employee ships a product or service for an order, she needs to review order details retrieved from database tables. She can easily spot a tampered payment status component in order details and thus reject the order.

- ***Secure communication channels***. For synchronous merchant-to-cashier `cURL` requests, remove taint for order total, merchant ID or currency when such a component is included in a URL

parameter; remove taint for order ID unconditionally. Synchronous requests offer a secure communication channel, and thus can guarantee the authenticity of payment status changes that pass through such a channel.

Our approach has one taint addition rule: When a conditional check for a cashier-to-merchant request relies on an exposed signed token, add taint to the exposed signed token. We keep track of all signed token values that are disclosed in DOM trees to users (typically in hidden HTTP form elements). Although hidden HTTP form elements are invisible in the presentation of an HTML page, attackers can obtain their values by simply viewing the source code of the page. Note that not all exposed signed tokens are tainted; the taint addition rule only applies when an exposed signed token is used as an unpredictable value in a conditional check for a cashier-to-merchant request. Once a signed token is exposed, it is no longer unpredictable and therefore should not be used in a conditional check. For example, suppose we have a signed token in a hidden HTML form with symbolic value `md5($secret.$orderId.$orderTotal)`. If we encounter an equality check `$_GET['hash'] == md5($secret.$_GET['oId'].$_GET['oTotal'])`, our approach adds taint to the exposed signed token. This is because although `$secret` is unpredictable, the three request variables are predictable. An attacker can easily pass the check by using the exposed signed token for `$_GET['hash']`, the current order ID for `$_GET['oId']` and the current order total for `$_GET['oTotal']`.

4.4 Implementation

We developed a symbolic execution framework that integrates taint analysis for PHP, one of the most prevalent languages for building web applications. We extended the PHP lexer and parser of a static string analyzer [53, 66, 79] written in OCaml. Our tool handles object-oriented features of PHP, including classes, objects and method calls. We wrote transfer functions for built-in PHP library functions, which include string functions, database functions, I/O functions etc. Our tool consults Satisfiability Modulo Theories (SMT) solver Z3 [23] for branch feasibility, supporting arithmetic constraints, simple string constraints and some other types of constraints. Our implementation contains a total of 25,113 lines of OCaml code. Although our implementation targets the PHP language, the

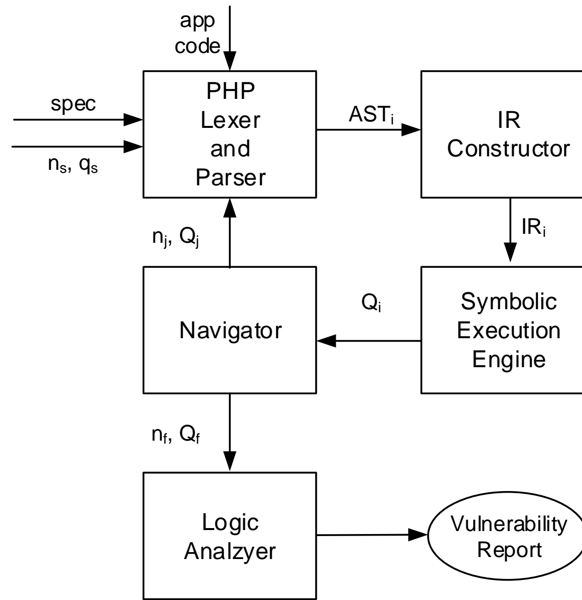

Figure 4.4: Symbolic Execution Framework.

high-level approach is general and applicable to e-commerce software written in other languages.

Figure 4.4 shows the architecture of our framework. Given the source code of an e-commerce application and a specification for it, our analysis starts with a single execution state q_s at merchant node n_s, the first node in the checkout process. For each merchant node n_i, our PHP lexer and parser transform the corresponding merchant page into an Abstract Syntax Tree AST_i, which is then transformed into an Internal Representation IR_i by our IR constructor. After the symbolic execution engine explores all possible control flow paths of IR_i, we have a set of end execution states Q_i. Next, the navigator searches for valid logic flows, and continues symbolic execution for new merchant nodes until the final merchant node n_f is reached. Finally, the logic analyzer checks all the unique logic states of final execution state set Q_f, and then reports any detected logic vulnerabilities.

To guide our automated analysis, we need developers to specify application-specific variable names of payment status components, critical merchant pages during checkout, cashier URLs, callback URL, return URL, values of configurable constants defined in the database and runtime values of a few variables that are used for the resolution of dynamic file inclusion and class construction. For instance,

a payment class can be dynamically constructed based on a user's choice of payment methods. If the user chooses PayPal Standard as the payment method, we can specify the value of runtime variable `$_SESSION['payment']` to be "paypal_standard" to precisely resolve the target of class `$payment`.

4.4.1 Symbolic Execution

For each web page during checkout, our PHP lexer and parser transform its source code into an IR. We followed the PHP language reference and carefully wrote parsing rules to resolve reduce/reduce conflicts, assigned operator precedence to resolve shift/reduce conflicts and used associativity to resolve other conflicts. We observed that a PHP page can either statically or dynamically include other pages via PHP include or `iframe`, and the pages that are included can in turn include other pages. To fully expand a PHP page, our analyzer infers static targets of the included pages when possible, and resorts to specification when targets can only be decided at run time. For example, static include `require(DIRS_CLASSES.'cart.php')` depends on the value of constant `DIRS_CLASSES`, while dynamic include `require($language.'.php')` depends on the runtime variable `$language`.

For heap modeling, our tool uses five variable maps: a variable-to-symbolic-value memory map, an instance-to-class-name map, an alias-to-variable map, an array-parent-to-array-elements map and an object-parent-to-object-properties map. First, the variable-to-symbolic-value map allows us to model a heap symbolically. A symbolic value is a recursive data structure composed of the following types: literal, basic symbolic value for a PHP variable, library function call, concatenation of two symbolic values, arithmetic expression, comparison expression and symbolic PHP resource value. For instance, symbolic value `md5("hello".$_GET['orderID'])` represents a call to library function `md5` with a symbolic argument of a concatenation of two symbolic values: a string literal "hello" and a basic symbolic value of type integer for `$orderID`. Second, given a class instance and a method name, the instance-to-class-name map enables us to quickly retrieve the corresponding class method definition. Third, the alias-to-variable map allows us to correctly update a symbolic heap. Aliases are created when: a method is called from within an object context (`$this` becomes available); a variable is assigned by reference; and a function/method has pass-by-reference arguments or returns a reference. Last, the two

maps for array and object variables enable us to track the children of arrays and objects respectively. Our tool uses one memory map for global variables and one memory map for local variables.

To model arrays and objects in PHP, we adopt the McCarthy rule for list manipulations [23]. Given an array a, an array element e and array index i, let $a[i]$ represent an array select and $a\{i \leftarrow e\}$ represent an array store with the element at index i set to e. By the McCarthy rule, we have the following:

$$(\forall\ \texttt{array}\ a)(\forall\ \texttt{element}\ e)(\forall\ \texttt{index}\ i, j)$$
$$i = j \longrightarrow a\{i \leftarrow e\}[j] = e$$
$$\wedge\ i \neq j \longrightarrow a\{i \leftarrow e\}[j] = a[j]$$

Our implementation precisely retrieves and updates array elements (or object properties) whenever possible. Otherwise, when an index of an array variable (or the field of an object property) is $\top$, all possible values of the array elements (or object properties) are merged. For example, suppose we have a simple array `$arr=array(1=>"x",2=>"y")`. If the value of array index `$i` is $\top$, the value of `$arr[$i]` is either "x" or "y". We also use the McCarthy rule to symbolically represent arrays and objects. As an example, the symbolic representation of `$arr` is:

$$array_update(array_update(array(), 1, \text{“}x\text{”}), 2, \text{“}y\text{”})$$

4.4.2 Path Exploration

Given a start execution state, our goal is to explore all possible intra-procedural and inter-procedural edges in a control-flow graph (CFG). We use a work-list-based algorithm and explore CFG edges with a depth-first strategy. On one hand, to explore all possible control flows within a function/method body, a work list stores execution states for feasible branches that have not been explored yet. Each execution state includes a program counter (consists of a basic block number and a statement number within the basic block), a logic state, path condition, memory maps of global and local variables, etc. We set a configurable quota for the maximum number of similar execution states in a work list to avoid

```
$error = false;
if ($_POST['x_response_code'] == '1') {
  if (tep_not_null(AUTHORIZENET_MD5_HASH) &&
    ($_POST['x_MD5_Hash'] != strtoupper(
      md5(AUTHORIZENET_MD5_HASH .
      AUTHORIZENET_LOGIN_ID .
      $_POST['x_trans_id'] .
      $this->format_raw($order->info['total'])
    )))) {
    $error = 'verification';
  } elseif ($_POST['x_amount'] !=
      $this->format_raw($order->info['total'])) {
    $error = 'verification';
  }
} elseif ($_POST['x_response_code'] == '2') {
  $error = 'declined';
} else {
  $error = 'general';
}

if ($error != false) {
  tep_redirect(tep_href_link(
    FILENAME_CHECKOUT_PAYMENT,
    'payment_error=' . $this->code .
    '&error=' . $error, 'SSL', true, false));
}
```

Figure 4.5: Example for Path Exploration.

state explosion. When the quota for a work list is exhausted, we only add an execution state to the work list if it has either a new program counter or a new logic state. On the other hand, to explore all possible inter-procedural edges, our approach adopts a global call stack which stores snapshots of previous function environment before function calls. A function environment snapshot includes a parameter-argument map for the inter-procedural function call to be explored, the work list and end execution states of the current function, etc.

Our tool consults the SMT solver Z3 for constraint solving. When a conditional is encountered during symbolic execution, our analyzer transforms the conditional into a formula of the `smtlib2` format, conjuncts the new formula with the current path condition, and feeds the merged path condition to Z3 to get an answer. When both branches are feasible, we select one branch to explore first, and add the other branch to the current work list. We support the following types in our constraints: boolean, integer, real, string, array, object, resource, `null` and $\top$. We try to infer the satisfiability of simple string constraints, which can contain literals, string variables, and operators such as $=, \neq, <, \leq, >$ and $\geq$. To symbolically represent PHP library function calls, we use `define-fun` in Z3 for function declarations.

Consider the example in Figure 4.5 for path exploration. Since the default values of request

variables are $\top$, all possible control-flow edges are explored. Only one exploration path in the example leads to a valid logic flow, while the other paths redirect users to the node for payment with an error message in `$error`. For the second `if` conditional on the valid path, there is a method call `$this->format_raw($order->info['total'])`. To follow this inter-procedural edge, our analyzer first looks up the class name of object `$this` and then the definition of method `format_raw` in the corresponding class. Next, the analyzer updates aliases for pass-by-reference parameters which include `$this`, initializes parameter values based on the arguments of the method call, passes on the memory map of global variables and pushes a snapshot of the current function environment into the global call stack. At the end of the symbolic execution for `format_raw`, we have a set of execution states Q. After the method call returns, our analyzer pops the function environment from the global call stack and maps Q to Q' to update method arguments that have pass-by-reference parameters. To continue path exploration after the call, Q' is added to the current work list. When all possible paths are explored, merchant ID and order total are untainted in the execution state that keeps the value of `$error` unchanged.

4.4.3 Logic Flows

The focus of our analysis is critical logic flows of a successful checkout process. We discard backward flows, error flows or aborted flows since they are irrelevant to our security analysis. First, a *backward flow* happens when an error has occurred in a merchant node n, and the user is redirected to a previous merchant node or the same merchant node n. Second, an *error flow* refers to a redirection to a special error page or a visited page with an error message in a request variable. In the first case, the special error page does not belong to the critical checkout process and the flow to this page is discarded. In the second case, flows to pages with a symbolic error message variable are backward flows, which are automatically discarded. Last, an *aborted flow* happens when a serious error occurs and the rendering of the merchant page is stopped with an `exit` statement.

In search of links to other nodes, our analyzer parses symbolic values of HTTP forms and `cURL` parameters. Since string literal is not the only type that a symbolic value can represent, we cannot simply

use regular expressions such as <form \s*action\s*=\s*[^>]*> to extract links. Consequently, our parser recursively examines each component of a symbolic value to correctly handle non-literals. In most cases, merchants embed URLs in HTTP requests to cashiers and our parser can find such URLs. However, a merchant may also store the configurations of callback URL and return URL on a cashier's server. For this case, we need to specify the pre-configured merchant URLs to continue exploring logic flows after a user-to-cashier request.

Requests from cashiers often store critical payment status in the parameters. Although the names of request parameters vary for different cashiers, it is not necessary to associate their names with payment status components unless their values are written to a database. On one hand, when an untrusted request parameter is compared against a trusted payment status component, our tool can infer which payment status component a request parameter is associated with, and apply taint rules for involved payment status components. For instance, for $_POST['x_amount'] ==$order->info['total'], our analyzer removes taint from order total because of the trusted payment status component $order->info['total'] rather than the untrusted $_POST['x_amount']. On the other hand, when untrusted request variables from cashiers are written to a database via INSERT or UPDATE queries, we need a specification of which payment status component a request parameter is associated with. For example, suppose a specification associates $_GET['v1'] with order ID and $_GET['v2'] with order total. If these two request parameters are written to a merchant's database, they will be read from the database and displayed clearly to a merchant employee. Since she needs to review order details before accepting an order, she may reject any order with abnormal payment status. Consequently, the taint annotations of order ID and order total should be removed based on the specification for $_GET['v1'] and $_GET['v2'].

4.5 Empirical Evaluation

To evaluate the effectiveness and performance of our approach, we performed experiments on osCommerce [1], one of the most popular open-source e-commerce applications. It has a long history of 13 years, powering more than 14,000 registered sites [1]. The latest stable release (version 2.3) of

Cashier	**Modules**	**Unique**	**Callback**
2Checkout	1	1	N
Authorize.net	2	2	N
ChronoPay	1	1	Y
inpay	1	1	Y
iPayment	3	1	Y
Luottokunta	2	2	N
Moneybookers	23	1	Y
NOCHEX	1	1	N
PayPal	5	5	Y
PayPoint.net	1	1	N
PSiGate	1	1	N
RBS WorldPay	1	1	Y
Sage Pay	3	3	Y
Sofortüberweisung	1	1	Y
Sum	46	22	8

Table 4.1: Payment Modules for Cashiers.

osCommerce contains 987 files with 38,991 lines of PHP code. It supports various third-party cashiers and multiple currencies with different payment modules, which are integrated in the main framework as add-ons. Each payment module provides a payment method that a user can choose during checkout.

In total, We evaluated 46 payment modules, 22 of which have distinct CFGs. There are 928 payment modules for osCommerce, and new payment modules have been actively added since 2003. In addition, payment modules evolve over time. For example, module Luottokunta (version 1.2) was reported to be vulnerable [17], and Luottokunta (version 1.3) was released to patch the reported vulnerability. 46 payment modules are included in osCommerce by default, and 44 of them are developed to integrate third-party cashiers. The two remaining payment modules are irrelevant to our security analysis: One allows merchants to accept cash on delivery, and the other enables merchants to accept mailed money orders. The 44 payment modules that accept online payment have 20 unique CFGs. Modules that differ slightly from one another in terms of variable names and cashier URLs may have identical CFGs. Therefore, we evaluated 20 default payment modules that have unique CFGs as well as the two Luottokunta payment modules. All the experiments are run on a desktop PC with a quad-core CPU (2.40 GHz) and 4GB of RAM.

Table 4.1 shows payment modules from 14 different cashiers. Column “Modules” shows the number

Module	Tainted / Exposed					Safe
	OrderId	OrderTotal	MerchantId	Currency	SignedToken	
2Checkout	✗	✗	✗	✗		✘
Authorize.net Credit Card AIM						✓
Authorize.net Credit Card SIM	✗			✗		✘
ChronoPay	✗	✗	✗	✗	✗	✘
inpay						✓
iPayment (Credit Card)	✗					✘
Luottokunta (v1.2)	✗	✗	✗	✗		✘
Luottokunta (v1.3)	✗			✗		✘
Moneybookers						✓
NOCHEX	✗	✗	✗	✗		✘
PayPal Express						✓
PayPal Pro - Direct Payments						✓
PayPal Pro (Payflow) - Direct Payments						✓
PayPal Pro (Payflow) - Express Checkout						✓
PayPal Standard			✗			✘
PayPoint.net SECPay	✗	✗		✗		✘
PSiGate	✗	✗	✗	✗		✘
RBS WorldPay Hosted				✗	✗	✘
Sage Pay Direct						✓
Sage Pay Form		✗		✗		✘
Sage Pay Server						✓
Sofortüberweisung Direkt				✗		✓*
Total	9	7	6	11	2	9 + 1*

Table 4.2: Logic Vulnerability Analysis Results.

of payment modules that a cashier has, and column "Unique" lists the number of payment modules that have unique CFGs. All the payment modules are in their latest versions except Luottokunta, for which we included two versions with different CFGs. Cashier Moneybookers provides 23 payment modules for various countries and currencies, but we observed that all of them share the same CFG. Therefore, it is sufficient to pick just one Moneybookers module for our security analysis. In contrast, PayPal has 5 payment modules and each of them is unique.

4.5.1 Analysis Results

Table 4.2 shows the analysis results for the 22 unique payment modules. Columns under "Tainted/-Exposed" show the existence of tainted components of payment status and exposed signed tokens for each module. For these columns, a table cell marked with "✗" means that a payment status component

is tainted or a signed token is exposed. The last column "Safe" summarizes the safety of a payment module. When a payment module verifies all the components of payment status and exposes no signed tokens, it is considered safe and marked with "✓"; otherwise, it is marked with "✘".

Table 4.2 shows that when a payment module is unsafe, it is often vulnerable to several types of logic attacks on different components of payment status. First, 9 modules fail to correctly verify order ID. This allows attackers to pay once for an order, and reuse the payment status values of the paid order to bypass payment for future orders. Second, 7 modules fail to verify order total, allowing attackers to pay an arbitrary amount. Third, 6 modules fail to verify merchant ID, allowing attackers to pay themselves instead. Note that the verification of secret can replace the verification of merchant ID. Fourth, 11 modules fail to verify currency, making it the most neglected component of payment status. When a cashier is configured to accept only one currency for a merchant, not verifying currency is safe and acceptable. However, we believe that the best practice is to always verify currency so that additional currencies can be easily added in the future. Last, 2 signed tokens are accidentally exposed in plain text, allowing attackers to pose as cashiers. We also tracked exposed secrets in our evaluation. When a secret is exposed, an attacker can arbitrarily forge values for order ID, order total, merchant ID and currency. Fortunately, none of the modules makes such a mistake.

In summary, as shown in the last column of Table 4.2, 9 out of 22 modules are safe; module Sofortüberweisung Direkt is safe when cashiers accept only one currency; the remaining 12 modules are vulnerable. We expected the patched version of Luottokunta (v1.3) to be safe at first but were surprised to see that it is still vulnerable. This shows the difficulty of writing a perfectly secure payment module. We manually confirmed the vulnerabilities on a local deployment of osCommerce, successfully performed responsible experiments on live web stores powered by osCommerce and communicated with the developers of osCommerce about the detected vulnerabilities. We classified the detected logic vulnerabilities into the following categories.

Untrusted Request Variables

Payment module developers sometimes make the mistake of checking payment status based on untrusted request variables. Verifying untrusted request variables guarantees neither the integrity nor the authenticity of payment status, but may give developers a false sense of security. Four modules, namely, Authorize.net Credit Card AIM, iPayment (Credit Card), Luottokunta (v1.3) and PayPoint.net SECPay fall into this category. The values of untrusted request variables that pass such insufficient checks may be inconsistent with actual payment status components. For example, module iPayment (Credit Card) performs a check on order ID based on untrusted request variable `$_GET['ret_booknr']` in the following code.

```
$_GET['ret_param_checksum'] !=
md5(MODULE_PAYMENT_IPAYMENT_CC_USER_ID
  . ($this->format_raw($order->info['total'])
  * 100) . $currency
  . $_GET['ret_authcode'] . $_GET['ret_booknr']
  . IPAYMENT_CC_SECRET_HASH_PASSWORD)
```

An attacker could pay once for an order and intercept the cashier-to-merchant request of the paid order by modifying the return URL of the preceding merchant-to-cashier request. For the above example, the attacker needs to record the values of `$_GET['ret_param_checksum']`, `$_GET['ret_authcode']` and `$_GET['ret_booknr']`. For future orders, the attacker can purchase different products and bypass payment as long as the order total and currency are the same as the paid order. Note that `$_GET['ret_param_checksum']` is supposed to be an unpredictable and unique value signed with secret `IPAYMENT_CC_SECRET_HASH_PASSWORD`. However, simply replaying the intercepted values of the three GET variables would allow the attacker to pass the above payment status check. The check in the example is insufficient because the value of order ID in the conditional comes from untrusted `$_GET['ret_booknr']`.

Exposed Signed Tokens

An exposed signed token nullifies the verification of payment status. Two modules ChronoPay and RBS WorldPay Hosted expose their signed tokens. Verification based on exposed signed tokens fails to ensure the authenticity of payment status. An attacker could record the values of signed tokens hidden in HTML forms and forge a request to fake a completed payment. The following exposed signed token from the form element `M_hash`, for example, nullifies the verification on order ID, order total and merchant ID (secret `RBSWORLDPAY_HOSTED_MD5_PASSWORD` can also uniquely identify a merchant).

```
tep_draw_hidden_field('M_hash',
  md5(tep_session_id() . $customer_id
    . $order_id . $language
    . number_format($order->info['total'], 2)
    . RBSWORLDPAY_HOSTED_MD5_PASSWORD));
```

Fundamentally, exposed signed tokens are caused by using the same secret for both merchant signature and cashier signature. We observed that a signed token is often exposed when a merchant wishes to use it to authenticate herself to a cashier. A signed token can work both as a merchant signature and a cashier signature for non-cURL HTTP requests. When a signed token is used for both purposes, it is considered exposed if attackers can intercept cashier-to-merchant requests. There are two methods to fix the problem. The first one is to use just one secret but two ways of calculation to make the signed tokens different. For example, by simply changing the orders of the components of payment status in a calculation, we can generate different signed tokens with the same secret. A better method is to use two secrets to avoid exposing important signed tokens. We can use one secret to authenticate a merchant and the other to authenticate a cashier using the same calculation, without worrying about the security of signed tokens.

Incomplete Payment Verification

Payment modules sometimes only partially verify the components of payment status. In other words, checks of some components of payment status are missing rather than insufficient. Three modules,

namely, Sage Pay Form, Sofortüberweisung Direkt and PayPal Standard belong to this category. Module Sage Pay Form writes partial payment status into the database, but misses checks on order total and currency. Module Sofortüberweisung Direkt does not verify currency and therefore is vulnerable to currency tampering attacks if cashiers are configured to support multiple currencies. Module PayPal Standard misses the check on merchant ID, allowing an attacker to pay herself instead.

Missing Payment Verification

Some payment modules are not designed with logic security in mind and have no security checks of payment status at all. They could easily become the playground for attackers. The following five payment modules unfortunately fall into this category: ChronoPay, Luottokunta (v1.2), NOCHEX, 2Checkout and PSiGate. Such payment modules should be patched as soon as possible.

4.5.2 Experiments on Live Websites

To show the feasibility and ease of attacks based on the detected logic vulnerabilities listed in Table 4.2, we conducted experiments on three live websites in a responsible manner. We consulted lawyers at our university and followed the example of Wang *et al.* in setting up attacker anonymity, purchasing a VISA gift card at a supermarket with cash, and registering accounts on third-party cashiers [77]. The Google Chrome browser with no extensions suffices as our attack tool. Although we initially paid nothing or less to the merchants for the three orders we placed, we paid in full amounts to the merchants after we received the products shown in Figure 4.2. We reported our findings to osCommerce developers. The details of the experiments are elaborated in the following.

The Ubuntu online shop by Canonical Ltd. (RBS WorldPay Hosted). RBS WorldPay is a cashier mainly used in the U.K. and supports multiple currencies. The Ubuntu online shop is a featured osCommerce shop, and it uses the vulnerable module RBS WorldPay Hosted. As Table 4.2 shows, this payment module is vulnerable to currency attacks. We placed an order in U.K. pounds but paid cashier WorldPay in U.S. dollars of the same amount. About one week later, we received a Ubuntu notebook (shown in Figure 4.2) even though we did not pay the full amount at first.

A baby products online shop (Authorize.net Credit Card SIM). Module Authorize.net Credit Card SIM is vulnerable to order ID attacks. In our experiments on the baby products online shop, we placed two orders of the same order total but only paid for the first order. We set up a simple web page on our server to record the values of HTTP request variables. For the first order, we changed the value of return URL from the merchant URL to that of our web page. This change lets cashier Authorize.net send the payment notification request to us instead of the merchant. We replayed the recorded values of the request variables from the first order for the cashier-to-merchant request of the second order. We paid nothing for the second order at first but received a dirty diaper game package shipped from California.

A chocolate online shop (PayPal Standard). Module PayPal Standard is vulnerable to merchant ID attacks. PayPal is one of the most popular cashiers in the U.S., yet it is not used securely in this payment module. In our experiment on the chocolate online shop, we simply changed the merchant ID from the chocolate merchant's PayPal account to our own PayPal account for the user-to-cashier payment request. In this way, we received three pieces of chocolate although the payment was not made to the chocolate merchant at first.

4.5.3 Performance Evaluation

Table 4.3 shows some data that we collected during symbolic execution to demonstrate the performance of our tool. For each IR, we report the number of parsed files (column "Files"), the number of nodes and node coverage (column "Nodes (%)"), the number of edges and edge coverage (column "Edges (%)") and the number of statements and statement coverage (column "Stmts (%)"). Additionally, column "States" shows the total number of end execution states; column "Flows" shows the total number of logic flows among user, cashier and merchant during checkout; and column "Time (s)" shows the total analysis time in seconds for each payment module.

Merchant nodes are non-trivial to analyze. The number of files that each merchant node includes ranges from 98 to 106, with an average of 102.73. An IR has 5,173 basic blocks (nodes), 6,162 control flow edges and 8,376 statements on average. To estimate the efforts of manual code review, we have

Payment Module	Files	Nodes (%)	Edges (%)	Stmts (%)	States	Flows	Time (s)
2Checkout	105	5,194 (19.09%)	6,176 (19.15%)	8,385 (25.01%)	40	4	16.04
Authorize.net Credit Card AIM	105	5,274 (19.95%)	6,284 (19.96%)	8,545 (25.97%)	43	4	17.65
Authorize.net Credit Card SIM	105	5,221 (19.66%)	6,221 (19.72%)	8,435 (25.52%)	46	4	16.89
ChronoPay	99	5,013 (15.67%)	5,969 (15.61%)	8,084 (20.75%)	69	5	31.51
inpay	100	5,118 (18.31%)	6,109 (18.42%)	8,408 (23.68%)	335	6	125.29
iPayment (Credit Card)	99	4,999 (16.09%)	5,932 (16.14%)	7,918 (21.62%)	38	5	21.86
Luottokunta (v1.2)	105	5,158 (18.94%)	6,127 (18.96%)	8,291 (24.72%)	34	4	15.33
Luottokunta (v1.3)	105	5,164 (18.99%)	6,135 (19.03%)	8,308 (24.80%)	35	4	15.33
Moneybookers	99	5,082 (15.90%)	6,059 (15.85%)	8,215 (21.08%)	66	4	80.85
NOCHEX	105	5,145 (18.90%)	6,111 (18.89%)	8,237 (24.67%)	33	4	15.03
PayPal Express	104	5,351 (12.63%)	6,379 (12.64%)	8,596 (17.95%)	62	11	42.15
PayPal Pro - Direct Payments	105	5,302 (19.85%)	6,339 (19.77%)	8,700 (25.61%)	65	4	20.76
PayPal Pro (Payflow) - Direct Payments	105	5,302 (19.92%)	6,339 (19.85%)	8,714 (25.71%)	63	4	20.85
PayPal Pro (Payflow) - Express Checkout	99	5,128 (14.41%)	6,107 (14.35%)	8,197 (20.08%)	31	10	31.95
PayPal Standard	99	5,040 (16.03%)	6,006 (16.01%)	8,170 (21.04%)	68	6	33.01
PayPoint.net SECPay	105	5,174 (19.09%)	6,152 (19.10%)	8,332 (24.97%)	40	4	15.80
PSiGate	106	5,231 (19.07%)	6,228 (19.04%)	8,436 (24.95%)	44	4	16.82
RBS WorldPay Hosted	99	5,019 (15.84%)	5,977 (15.92%)	8,121 (21.09%)	79	5	36.12
Sage Pay Direct	106	5,447 (20.71%)	6,515 (20.55%)	8,984 (25.97%)	95	4	26.20
Sage Pay Form	106	5,315 (19.52%)	6,329 (19.54%)	8,762 (24.55%)	55	4	19.96
Sage Pay Server	101	5,100 (14.72%)	6,067 (14.62%)	8,268 (19.78%)	42	6	28.26
Sofortüberweisung Direkt	98	5,038 (16.01%)	6,003 (15.96%)	8,160 (21.20%)	97	5	43.86
Average	102.73	5,173 (17.70%)	6,162 (17.69%)	8,376 (23.21%)	67.27	5.05	31.43

Table 4.3: Performance Results.

also counted the lines of code that are related to the checkout process for payment modules. In general, the number of lines of code is slightly higher than the number of statements listed in Table 4.3 for each payment module. For example, for module PayPal Express, there are 8,727 lines of code in total to review while its IR has 8,596 statements. In addition to code, manual reviewers need to examine database tables and cashiers' documentation.

The coverage of nodes, edges and statements is calculated for the main function, function bodies and method bodies. Some defined functions, defined class methods and even some branches of the main function may not be executed at all in the checkout process. On average, the symbolic execution of each merchant node has a CFG node coverage of 17.70%, an edge coverage of 17.69% and a statement coverage of 23.21%. Our tool explores paths in the CFGs based on branch feasibilities. Note that one merchant page is often divided into multiple merchant nodes based on different values in the query string of a request. Our exploration is based on merchant nodes, but the coverage is calculated using merchant pages. This explains why some modules have low coverage. The callback page of PayPal Express for example, has a `switch` statement based on the value of request variable

$_GET['osC_Action'] near the beginning of the page. It has different branches to handle "cancel", "callbackSet", "retrieve" and default actions. For a merchant node of this page, only one switch branch is taken.

On average, it takes 31.43 seconds to explore 67.72 execution states in 5.05 logic flows for each payment module. In simple cases, it takes only 4 logic flows to initiate the checkout process, make a payment on a cashier's server, notify the merchant of the payment and complete the order. Module PayPal Express has the most complex logic flows. It uses 11 logic flows to obtain a ppe_token for each payment transaction, start an express checkout with function setExpressCheckout, make a payment on a PayPal server, get payer details with function getExpressCheckoutDetails and complete the sale with function doExpressCheckoutPayment. Module inpay has the longest analysis time (125.29 seconds) and also the largest number of execution states (335 states). The performance results show that our automated detection is more efficient and comprehensive than manual analysis. When we manually confirmed the detected logic vulnerabilities, we need around 30 minutes for each payment module. We spent about 15 minutes to read control flows of merchant pages and cashier documentation, and another 15 minutes to find valid inputs that lead to logic attacks.

We have adopted a few optimizations to speed up our analysis and two of them significantly reduced the analysis time. The first optimization sets the maximum number of similar execution states in a work list to one. This means that whenever the analysis stores a new execution state in a work list, it first checks if there already exists an execution state with the same program counter and the same logic state. If yes, the new execution state is discarded. Since such two execution states often differ only slightly, discarding the second state has no impact on the vulnerability analysis result. The analysis time for each payment module is limited to 10 minutes. When we increased the length of a work list to two, timeout events occurred before the analyses were completed. The second optimization sets some symbolic session variables to be not null, just like what they should be in a normal checkout process. For example, $_SESSION['customer_id'] and $_SESSION['cartId'] are specified as not null. The first few basic blocks in the IR of a merchant node often check whether some session variables are null. The second optimization rules out irrelevant branches at an early state of a symbolic execution process. This accelerates our analysis considering that the number of states usually grows at an exponential rate.

4.5.4 Discussions

The implementation of our detection tool is neither sound nor complete. For all the logic vulnerabilities detected by our tool, we carefully examined and tested each one to confirm that they are true positives. There is no observed false positives to the best of our knowledge. We cannot guarantee the absence of logic vulnerabilities because of the difficulty of exploring all possible logic flows in large real-world e-commerce applications. We hope our tool can help developers write secure payment modules and raise their security awareness.

Our static analysis still faces nontrivial challenges which include dynamic features of PHP, constraint solving and regular expressions. Typically, static analyses are limited in handling dynamic language features (e.g. dynamic includes, dynamic class, array and object construction), and the dynamic features of PHP also most significantly influence the scalability and precision of our analysis. For a precise resolution of dynamic features, specifications are incorporated for some critical code.

Our current implementation does not support JavaScript analysis yet. It is possible that some links to merchant nodes or cashier nodes are generated by JavaScript code on the client side. We did not encounter any JavaScript links in our experiments but our test subject may not be representative of other e-commerce applications. Detecting links in JavaScript code is a difficult task because of the various dynamic features of the JavaScript language. For e-commerce applications that heavily use JavaScript, we may need to incorporate JavaScript analysis to detect critical URLs that are dynamically generated.

Automated analysis incurs significant engineering efforts and the amortized development cost can be kept low for e-commerce software with a large number of payment modules. Symbolic execution allows systematic exploration and is particularly useful to model HTTP requests/responses from cashiers and users as symbolic values can be used (rather than concrete values). In contrast, manual code review is error-prone, and it is difficult to cover all possible attack vectors and important control-flows (which may explain why many serious vulnerabilities still exist). The number of payment modules (928 for osCommerce) and the two vulnerable Luottokunta modules illustrate the difficulty of detecting missing/insufficient checks. However, for basic e-commerce software with only a few payment modules,

manual code review may be a viable alternative.

It is possible that there exist multiple execution states with unique logic states when we reach the final merchant node during checkout. There is no universal criterion as to which logic state should be picked over another for valid logic flows, and we leave the selection of logic states to developers who have the best judgment. Our current tool includes all taint operations and flows in logic states as a reference, and uses heuristics based on our observations to rank logic states. The logic state that should be picked is often the one that has the least number of taint annotations, excluding exposed signed tokens. The reason is that our symbolic execution may conservatively explore a branch that will not be taken in practice, and only the opposite branch contains checks on payment status.

4.6 Related Work

Logic vulnerabilities in e-commerce applications. The uniqueness of logic vulnerabilities, together with their great impact, has attracted the attention of researchers in recent years. Wang *et al.* [77] are the first to analyze logic vulnerabilities in Cashier-as-a-Service based web stores. Through manual security analysis, they found serious logic flaws that can lead to inconsistent payment status between a merchant's server and a cashier's server. Their follow-up work, InteGuard [84], offers dynamic protection of third-party web service integrations, including the integration of cashier service in merchants' websites. In contrast to their work, we seek to comprehensively examine various attack vectors on payment status and automatically detect logic vulnerabilities during checkout before the deployments of e-commerce applications. We discovered a new attack vector which allows an attacker to modify the currency of a payment to her advantage, and designed a symbolic execution framework to systematically explore critical logic flows during checkout.

Parameter pollution vulnerabilities in web applications. Another active line of research is HTTP Parameter Pollution (HPP) in web applications. It is a common attack vector for various vulnerabilities which include logic vulnerabilities. WAPTEC [10] takes a white-box approach that combines symbolic execution and dynamic analysis to detect parameter tampering vulnerabilities in PHP applications, while

NoTamper [9] and PAPAS [6] adopt black-box based approaches. NoTamper [9] detects insufficient server-side validations where a server fails to replicate the validations on the client side. PAPAS [6] aims at automated discovery of parameter pollution based on a black-box scanning technique for vulnerable parameters. Our approach also makes the assumption that user inputs are untrusted. However, in contrast to parameter pollution detection which examines parameters in isolation, our approach detects logic vulnerabilities in e-commerce applications by linking and analyzing the logic flows of a checkout process.

Other logic vulnerabilities in web applications. Besides attacks on e-commerce applications, logic vulnerabilities also open doors to other attacks which include access control attacks, single sign-on attacks and workflow violations in web applications. First, access control vulnerability exposes privileged functionality or resources to unauthenticated users. Nemesis [22] performs dynamic information flow tracking based on specified access control lists, while static approaches analyze source code to detect unprotected accesses [24, 64, 66]. Second, Wang *et al.* discovered new single sign-on attacks [78], and InteGuard moves a step forward [84] by using a proxy-based approach which checks a set of inferred invariants to let merchants safely integrate third-party web services. Third, to detect deviations of normal workflows, it is important to first establish a good guideline of correct workflows. Such a guideline can be specified by developers [35], inferred from client-side validations which should be replicated on the server side [9, 31], or obtained from dynamic analyses [7, 20, 27, 45]. An alternative way of thwarting logic attacks is secure-by-construction. Both Swift [15] and Ripley [72] aim to offload some computations to the client side while ensuring the consistency of logic states between servers and clients for modern web applications. Logic vulnerabilities in e-commerce applications are one important subtype of general logic vulnerabilities in web applications. Focusing on this particular domain, we are able to design an invariant of secure payments to detect logic vulnerabilities which are application-specific.

Symbolic execution and taint analysis. Symbolic execution and taint analysis are two widely used techniques in security research. Schwartz *et al.* [62] provide a high-level view of dynamic taint analysis

and forward symbolic execution. Symbolic execution is a powerful technique that can be adopted for a diverse set of languages and problem settings ever since the seminal work by King [41]. For traditional programs, KLEE [11] is capable of automatically generating tests that achieve high coverage on even complex programs. For server-side languages, Halfond *et al.* [34] apply symbolic execution to precisely identify interfaces in the Java Enterprise Edition (JEE) framework, while Rubyx [13] detects security vulnerabilities based on specifications by symbolically executing Ruby-on-Rails web applications. For JavaScript, a client-side language widely used in web applications, Saxena *et al.* [61] designed and implemented a symbolic execution framework which can handle string constraints. Pixy [38] is a static taint analyzer built for PHP applications, and it detects injection vulnerabilities with taint analysis based on specifications of taint sources and sinks. Our approach combines symbolic execution with taint analysis in a novel way to detect potential logic attacks on payment status.

4.7 Conclusion

Merchants should carefully verify each critical component of payment status to ensure the consistency of payment status on merchants' severs and cashiers' servers. This chapter proposes the first static approach to automatically detect logic vulnerabilities in e-commerce web applications. Our key observation is that secure checks on payment status must verify the integrity and authenticity of order ID, order total, merchant ID and currency. Our framework integrates symbolic execution with taint analysis to track critical logic states, which include payment status, across merchant nodes during checkout. Our tool explored important logic flows, scaled to 22 unique real-world payment modules and detected 11 unknown vulnerabilities along with one known vulnerability. For future work, we plan to support additional path exploration strategies for symbolic execution, add function summaries to improve performance and apply our analysis to a larger number of popular e-commerce applications.

Chapter 5

Conclusion

This dissertation has presented novel program analyses for web application security, with a focus on detecting application-specific vulnerabilities. It has revealed the fundamental self-replicating characteristic of XSS worms, and formally defined access control vulnerabilities in web applications and logic vulnerabilities in e-commerce applications. Aside from these conceptual contributions, this dissertation has presented practical, scalable algorithms and tools that can detect known and new vulnerabilities in real-world web applications. Its body of work has laid a solid foundation for future research in this important direction.

5.1 Summary

XSS worms are based on XSS vulnerabilities, which are the most common type of vulnerabilities in web applications. The silent infection process and rapid propagation of XSS worms in community-driven web applications can significantly increase the impact and spread of XSS attacks. Chapter 2 has presented the first purely client-side solution to detect XSS worms at an early stage of their propagation. This approach first analyzes retrieved external files, DOM scripts and loaded external files, and then uses them as references to detect self-replicating payloads in parameter values of outgoing HTTP requests. Consequently, worms can be stopped in the beginning of their propagation. The implemented cross-platform tool effectively detected real-world XSS worms and is resilient against common JavaScript

obfuscators. Moreover, the evaluation shows that the tool incurs reasonably low performance overhead.

Access control vulnerabilities expose sensitive information and functionality in web applications, yet they are easy to exploit when privileged URLs are predictable. Attackers can simply use forced browsing to directly access privileged URLs rather than follow explicit links in web applications. Chapter 3 has proposed a novel approach to detect access control vulnerabilities in web applications with minimal manual effort. Role is a central definition introduced in this approach and it captures a unique set of privileges that a user group has. A role can represent administrators, normal logged-in users or public users. The key observation is that sitemaps presented to different roles are not the same. Differences between role-based sitemaps form the core of the automated specification inference of privileged accesses. The analysis presented in Chapter 3 first automatically infers the set of privileged pages and then detects access control vulnerabilities via forced browsing. The evaluation results show that the implemented tool can detect both unknown and known access control vulnerabilities in unmodified web applications with minimal specifications.

Logic vulnerabilities in e-commerce applications are one of the most serious types of vulnerabilities in web applications because of their direct financial impact. A payment is secure only when the consistency of payment status between a merchant and its third-party cashier is guaranteed. It is essential for merchants to verify the integrity and authenticity of order ID, order total, merchant ID and currency for each order. Any missing or insufficient check on the components of payment status can lead to security breaches. Chapter 4 has presented the first static detection of logic vulnerabilities in e-commerce applications. This approach combines symbolic execution and taint analysis to track payment status across different logic flows of checkout processes. The evaluation demonstrates that the implemented tool is effective and scalable. It uncovered 11 unknown logic vulnerabilities in 22 unique real-world payment modules with no false positives.

5.2 Future Work

General logic vulnerabilities in web applications are widespread, diverse and application-specific, posing a substantial challenge for automated detection. They cover other types of vulnerabilities besides access

control vulnerabilities and logic vulnerabilities in e-commerce applications. Current research focuses on logic vulnerabilities caused by various user inputs, while the exploration of different navigation paths has remained largely unexplored. To utilize a sensitive functionality in a web application, benign users may share an identical navigation path. In contrast, to abuse a sensitive functionality with unexpected logic flows, attackers may visit nodes on a navigation path out of order, skip a node or repeatedly visit a node. If we use a state to capture critical application logic, general logic vulnerabilities can be modeled in terms of logic states. For a web application, given the start nodes of its critical navigation paths and start logic states for these start nodes, a logic vulnerability may exist when an end node can be reached with divergent logic states. This modeling can help automate the detection of general logic vulnerabilities.

For sophisticated Web 2.0 applications, coupling client-side code analysis with server-side code analysis can contribute to a better understanding of the overall functionality of an application. Although the majority of application source code is often server-side code, client-side code sometimes includes important information that should not be overlooked. JavaScript is a highly dynamic language frequently used for client-side code, featuring dynamic typing and runtime evaluation. Scripts written in JavaScript can access DOM trees, interact with users and exchange data asynchronously between web browsers and servers to avoid full page reloads. Due to its dynamic nature, JavaScript analysis is challenging. For example, its `eval` function, which executes statements provided as strings at run time, can be invoked in many different ways. Fortunately, analyzing benign scripts embedded in a web application is easier than analyzing malicious scripts. This makes automated analyses of benign JavaScript code possible. Researchers have proposed various JavaScript analyses in recent years [5, 29, 61]. For program analyses of web applications, the incorporation of JavaScript analysis can be a promising research direction.

The scalability and precision of the symbolic execution framework presented in Chapter 4 can be further improved. First, this symbolic execution can benefit from enhanced support of dynamic features of the PHP language. For *dynamic includes* in PHP, a web page sometimes dynamically includes tens of other web pages and it is nontrivial to manually specify the targets of all dynamic includes; for *dynamic object construction*, objects and their properties can be constructed on the fly based on runtime values; other common dynamic features include PHP variable variables, the `eval` function

and dynamic array manipulations, all of which depend on runtime values. Currently, the handling of dynamic language features in PHP influences the most the scalability and precision of this symbolic execution. Second, when coverage plays an important role, it is desirable to combine different path exploration strategies, such as breath-first, depth-first and randomized path explorations. The current path exploration relies on depth-first search, but a mixed path exploration strategy may yield better coverage. In addition, this symbolic execution framework can be enhanced by adding support for additional important library functions, such as database and I/O functions. Precise handling of database queries and I/O manipulations can improve the precision of this symbolic execution framework.

Bibliography

[1] osCommerce Online Merchant. http://www.oscommerce.com/.

[2] Diminutive XSS worm replication contest. http://sla.ckers.org/forum/read.php?2,18790,page=19, 2008.

[3] Tarek Ahmed. The trigram algorithm. http://search.cpan.org/dist/String-Trigram/Trigram.pm.

[4] Alexa. Top sites in United States. http://www.alexa.com/topsites.

[5] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of International Conference on Software Engineering*, 2011.

[6] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of Network and Distributed System Security*, 2011.

[7] Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of Computer and Communications Security*, 2007.

[8] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of Symposium on Security and Privacy*, 2008.

[9] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of Computer and Communications Security*, 2010.

[10] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of Computer and Communications Security*, 2011.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of Operating Systems Design and Implementation*, 2008.

[12] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of Computer and Communications Security*, 2008.

[13] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of Computer and Communications Security*, 2010.

[14] Shuo Chen, John Dunagan, Chad Verbowski, and Yi-Min Wang. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proceedings of Network and Distributed System Security*, 2005.

[15] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Symposium on Operating Systems Principles*, 2007.

[16] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the USENIX Security Symposium*, 2007.

[17] Common Vulnerabilities and Exposures. CVE-2009-2039. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2039`, 2009.

[18] Common Weakness Enumeration. CWE-840 business logic errors. `http://cwe.mitre.org/data/definitions/840.html`.

[19] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of Systems and Operating Systems Principles*, 2005.

[20] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of Recent Advances in Intrusion Detection*, 2007.

[21] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of Computer and Communications Security*, 2005.

[22] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication and access control vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium*, 2009.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[24] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of Computer and Communications Security*, 2011.

[25] Dean Edwards. Dean Edwards' JavaScript packer. `http://dean.edwards.name/packer/`.

[26] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of Symposium on Operating Systems Principles*, 2001.

[27] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium*, 2010.

[28] Firebug. http://getfirebug.com/.

[29] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Proceedings of Symposium on Principles of Programming Languages*, 2012.

[30] Jeremiah Grossman. Seven business logic flaws that put your website at risk. http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf, 2007.

[31] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for AJAX intrusion detection. In *Proceedings of World Wide Web*, 2009.

[32] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of Network and Distributed System Security*, 2009.

[33] William G. J. Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of Foundations of Software Engineering*, 2008.

[34] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of International Symposium on Software Testing and Analysis*, 2009.

[35] Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of Automated Software Engineering*, 2010.

[36] Robert Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html.

[37] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with Browser-Enforced Embedded Policies. In *Proceedings of World Wide Web*, 2007.

[38] Nenad Jovanovic, Christopher Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of Symposium on Security and Privacy*, 2006.

[39] Samy Kamkar. The Samy worm. `http://namb.la/popular/tech.html`, 2005.

[40] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of International Symposium on Software Testing and Analysis*, 2009.

[41] James C. King. Symbolic execution and program testing. In *Communications of ACM*, 1976.

[42] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Symposium on Applied Computing*, 2006.

[43] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of Operating Systems Design and Implementation*, 2006.

[44] Akshay Krishnamurthy, Adrian Mettler, and David Wagner. Fine-grained privilege separation for web applications. In *Proceedings of World Wide Web*, 2010.

[45] Xiaowei Li and Yuan Xue. BLOCK: A black-box approach for detection of state violation attacks towards web applications. In *Proceedings of Annual Computer Security Applications Conference*, 2011.

[46] Zhichun Li, Manan Sanghi, Yan Chen, Ming yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of Symposium on Security and Privacy*, 2006.

[47] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of Computer and Communications Security*, 2005.

[48] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of JavaScript worms. In *USENIX Annual Technical Conference*, 2008.

[49] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of Programming Language Design and Implementation*, 2009.

[50] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

[51] Mike Ter Louw and V.N. Venkatakrishnan. BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of Symposium on Security and Privacy*, 2009.

[52] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[53] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of World Wide Web*, 2005.

[54] Mozilla Corporation. Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[55] Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A robust basis for cross-site scripting defense. In *Proceedings of Network and Distributed System Security*, 2009.

[56] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed System Security*, 2005.

[57] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of International Information Security Conference*, 2005.

[58] Shira Ovide. Bloomberg news was the NetApp earnings leaker. `http://blogs.wsj.com/deals/2010/11/17/bloomberg-news-was-the-netapp-earnings-leaker/`, 2010.

[59] OWASP. `http://www.owasp.org`.

[60] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proceedings of Symposium on Security and Privacy*, 2009.

[61] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of Symposium on Security and Privacy*, 2010.

[62] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of Symposium on Security and Privacy*, 2010.

[63] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the Network and Distributed System Security*, 2009.

[64] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Proceedings of Network and Distributed System Security*, 2013.

[65] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of Principles of Programming Languages*, 2006.

[66] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium*, 2011.

[67] Symantec Corporation. Symantec global internet security threat report, volume 13, 2008.

[68] Symantec Corporation. Symantec global Internet security threat report, volume 18, 2013.

[69] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the USENIX Security Symposium*, 2008.

[70] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Proceedings of Programming Language Design and Implementation*, 2009.

[71] U.S. Census Bureau. Quarterly retail e-commerce sales. `http://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf`, 2013.

[72] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically securing Web 2.0 applications through replicated execution. In *Proceedings of Computer and Communications Security*, 2009.

[73] W3C. `http://www.w3.org/`.

[74] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of Symposium on Security and Privacy*, 2001.

[75] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of Recent Advances in Intrusion Detection*, 2005.

[76] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of Recent Advances in Intrusion Detection*, 2006.

[77] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online – security analysis of Cashier-as-a-Service based web stores. In *Proceedings of Symposium on Security and Privacy*, 2011.

[78] Rui Wang, Shuo Chen, and XiaoFeng 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 Symposium on Security and Privacy*, 2012.

[79] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of Programming Language Design and Implementation*, 2007.

[80] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of International Conference on Software Engineering*, 2008.

[81] Yao wen Huang, Fang Yu, Christian Hang, Chung hung Tsai, D. T. Lee, and Sy yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of World Wide Web*, 2004.

[82] WhiteHat Security. Website security statistics report, 2013.

[83] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, 2006.

[84] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. InteGuard: Toward automatic protection of third-party web service integrations. In *Proceedings of Network and Distributed System Security*, 2013.