UNIVERSITY OF CALIFORNIA
Santa Barbara

Advanced Automated Web Application Vulnerability Analysis

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

Doctor of Philosophy

in

Computer Science

by

Adam Loe Doupé

Committee in Charge:

Professor Giovanni Vigna, Chair

Professor Christopher Kruegel

Professor Ben Hardekopf

September 2014

The Dissertation of
Adam Loe Doupé is approved:

Professor Christopher Kruegel

Professor Ben Hardekopf

Professor Giovanni Vigna, Committee Chairperson

April 2014

Advanced Automated Web Application Vulnerability Analysis

Acknowledgements

I would like to thank the following people who, with their love and support, encouraged and motivated me to finish this dissertation.

Giovanni is the reason that I became addicted to computer security: From the moment that I took his undergrad security class I was hooked. I am forever indebted to him because he has constantly invested his time in me. First, by inviting me to join his hacking club. Then, he took a chance on mentoring a Master's student, and, upon graduation for my Master's degree, told me that I could "come back for the real thing." One year later I did, and I don't regret it for a second. I hope that I always have the enthusiasm and energy that Giovanni brings to research and life. He is truly a role model.

From Chris, I learned a great deal about what it means to be an academic and a scientist. He is able to focus so intensely on the details of the project while not loosing sight of the bigger picture—if I am able to do this half as well I will consider myself a success. He constantly inspires and encourages me to heights I never dreamed possible.

I would never have been able to finish this dissertation without the help and encouragement of all the past and current seclab members. I am forever grateful to the now departed members of the seclab who were here when I started my Masters. You took me under your wing, taught me, and created and infused within me the *seclab culture.*

I would like to thank: Wil, Vika, Sean, Nick, Brett, Lorenzo, Fede, Martin, Max, and especially Marco, for his wisdom and guidance.

The only way I was able to stay sane during my PhD was because of my fellow seclab members. We forged a bond during those 3am deadlines, iCTF all-nighters, walks to Freebirds!, costume parties, and soccer games—a bond which makes us brothers-in-arms. In no particular order, thanks to: Gianluca, Alexandros, Luca, Antonio, Gorkem, Ali, Patrick, Kevin, Jacopo, Dhilung, Kyle, Yan, Fish, Bob, Ludo, Manuel, Gregoire, and Bryce.

A very special thanks to my Mom, Dad, and brother for their love, support, and constant ribbing and teasing.

And finally, thank you Becca, for being my rock.

Curriculum Vitæ

Adam Loe Doupé

Education

2010 – 2014	PhD in Computer Science
	University of California, Santa Barbara
2008 – 2009	Master’s Degree in Computer Science
	University of California, Santa Barbara
2004 – 2008	Bachelor’s Degree in Computer Science with Honors
	University of California, Santa Barbara

Research Experience

2010 – 2014	Research Assistant, University of California, Santa Barbara
2013 Summer	Visiting PhD Student, Stanford University
	Advisor: John C. Mitchell
2012 Summer	Research Intern, Microsoft Research
	Advisor: Weidong Cui
2009	Research Assistant, University of California, Santa Barbara

Industry Experience

2009 – 2010	Software Developer Engineer, Microsoft
2008 Summer	Software Developer Engineer Intern, Microsoft
2007 – 2008	Software Developer, AT&T Government Solutions
2005 – 2010	Founder/Developer, WootWatchers.com
2004 – 2005	Mobile Product Manager, VCEL, Inc.

Teaching Experience

October 2013	Taught class lecture for undergraduate security class on web security, and created web application vulnerability homework by request of Richard Kemmerer
Fall 2013	Co-created and Co-taught "Recent Trends in Computing Research," a 2-unit seminar graduate class
November 2012	Created web application vulnerability homework and designed in-class lecture for undergraduate security class by request of Richard Kemmerer
April 2012	Created and ran a three hour hands-on workshop at UC Santa Barbara by request of the Web Standard Group entitled "Into the Mind of the Hacker"
October 2011	Taught class on crypto analysis for undergraduate security class by request of Richard Kemmerer

Fall 2010	Teaching Assistant for CS 279 (Advanced Topics in Computer Security), won Outstanding Teaching Assistant Award from the Computer Science Department
Fall 2008	Teaching Assistant for CS 177 (Introduction to Computer Security), won Outstanding Teaching Assistant Award from the Computer Science Department

Abstract

Advanced Automated Web Application Vulnerability Analysis

Adam Loe Doupé

Web applications are an integral part of our lives and culture. We use web applications to manage our bank accounts, interact with friends, and file our taxes. A single vulnerability in one of these web applications could allow a malicious hacker to steal your money, to impersonate you on Facebook, or to access sensitive information, such as tax returns. It is vital that we develop new approaches to discover and fix these vulnerabilities before the cybercriminals exploit them.

In this dissertation, I will present my research on securing the web against current threats and future threats. First, I will discuss my work on improving black-box vulnerability scanners, which are tools that can automatically discover vulnerabilities in web applications. Then, I will describe a new type of web application vulnerability: Execution After Redirect, or EAR, and an approach to automatically detect EARs in web applications. Finally, I will present deDacota, a first step in the direction of making web applications secure by construction.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Web applications are a fundamental part of our lives and culture. We use web applications in almost every facet of society: socializing, banking, health care, taxes, education, news, and entertainment, to name a few. These web applications are always available from anywhere with an Internet connection, and they enable us to communicate and collaborate at a speed that was unthinkable just a few decades ago.

As more and more of our lives and data move to web applications, hackers have shifted their focus to web applications. In 2011, hackers stole 1 million usernames and passwords from Sony [17]. In 2007, hackers stole 45 million customer credit cards from TJ Maxx [76]. In 2009, hackers stole 100 million customer credit cards from Heartland Payment Systems [1]. In 2012, hackers stole 24,000 Bitcoins[1] from BitFloor, a major Bitcoin exchange [84]. What all of these instances have in common is that hackers

[1]These Bitcoins are worth about $10 million at this time of writing.

exploited vulnerabilities in a web application to steal either usernames and passwords, credit cards, or Bitcoins.

Those same properties that make web applications so attractive to users also attract hackers. A web application never closes, so they are always available for hackers. Web applications also house a vast treasure-trove of data, which hackers can use for monetary gain. Finally, as we will explore in the next section, web applications are a complex hodgepodge of various technologies. This complexity, combined with the intense time-to-market pressure of companies and people that build web applications, is a breeding ground for bugs and vulnerabilities.

The situation is dire. We must focus on new ways to secure web applications from attack. We must develop new tools in order to find the vulnerabilities before a hacker does. We must, because web applications and the data they store are too important.

1.1 History of Web Applications

The World Wide Web was created by Sir. Tim Berners-Lee in 1989 as a means of sharing information for the CERN research organization. What began as a way to share simple hyper-linked textual documents over the nascent Internet quickly exploded in popularity over the proceeding years.

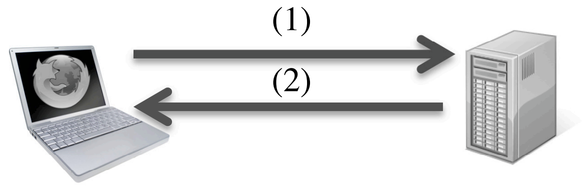

Figure 1.1: Example interaction between a web browser and a web server. In (1), the web browser makes an HTTP request to the webserver, and in (2) the web server sends the web browser an HTTP response containing the HTML of the web page.

The core of the web has remained relatively the same throughout the years: a web browser (operated by a user) connects to a web server using the Hypertext Transfer Protocol (HTTP) [49]. The web server then sends back a response, typically in the form of a HyperText Markup Language (HTML) page [16]. The web browser then parses the raw HTML page to create a graphical web page that is displayed to the user. The fundamental underlying principle, and the definition of HyperText, is that an HTML page contains links to *other* HTML pages.

Figure 1.1 shows a graphical representation of the interaction between the web browser and the web server. In (1), the web browser will make an HTTP request to the web server, to request a resource. Then, the web server will respond, as in (2), with an HTTP response which contains the HTML of the requested web page.

The beginning of the web was envisioned as a set of documents with links pointing to other documents[2]. In other words, the web was mostly a set of read-only documents (from the perspective of the user with the web browser). This is where the term *web*

[2]This is where the name *web* comes from, as each link forms a strand in the web.

site comes from: a web site is typically thought of as a collection of documents that exist under the same domain name.

As the web evolved, web sites started to shift from static, read-only documents. Developers realized that the HTML response returned to the client could be dynamic—that is, the content of the HTML response could vary programmatically. This shift in thinking caused web sites to transition to *web applications* which emulated features of traditional desktop applications. Web applications enabled scenarios that caused the web's popularity to increase: e-commerce, news sites, and web-based email clients. It is hard to overstate the impact that web applications had on the uptake of the web.

Now, with web applications, the architecture of the web changed. When the web browser makes an HTTP request to the server, instead of returning a static HTML response, the web server typically will invoke server-side code. This server-side code is responsible for returning a response, typically HTML, to the browser. The server-side code can use any number of inputs to determine the response, but typically the server-side code reads the parameters sent in the browser's HTTP request, consults an SQL database, and returns an HTML response.

Figure 1.2 shows an example web application with a back-end SQL database. Now, when the web browser sends an HTTP request to the web application, as in (1), the web application's server-side code will start to execute. Then, as (2) shows, the server-side code can make one or more request to the SQL database, when executes the queries

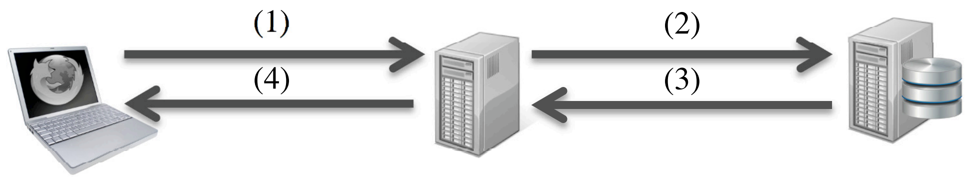

Figure 1.2: Sample web application with servers-side code and a back-end database. In (1), the web browser makes an HTTP request to the web application. Then the server-side code can issue one or more SQL queries to the back-end SQL database, shown as (2). The SQL server returns the data in (3), which the web application will use to generate an HTTP response with HTML, as in (4).

and returns the data to the server-side code in (3). Finally, the web application finishes processing the request and sends an HTTP response with an HTML web page to the web browser in (4).

The HTTP mechanism is, by design, stateless: Each HTTP request that the web server receives is independent of any other request. It is difficult to build an interactive application on top of a stateless protocol, thus a standard was developed to add state to the HTTP protocol [87]. This standard added the *cookie* mechanism to the HTTP layer. In this way, a web server can ask the web browser to set a cookie, then, in subsequent requests, the web browser will include the cookie. Therefore, a web server or web application can link the requests into a *session* based on the common cookie and thus develop state-aware web applications.

Even after the advent of web applications, the server-side code would return an HTML page that was statically rendered and displayed to the user. To change to content on the page or otherwise interact with the web application, the browser must perform another HTTP request and receive a response based on a link the user clicked or a form the user submitted. In 1997, Brendan Eich, a programmer at Netscape, created a client-side scripting language called JavaScript [46]. The user's web browser implemented an interpreter for this scripting language so that it could manipulate the web page. Now, with JavaScript, web developers could programmatically alter the content on the web page without making a request to the web server. The final linchpin which enabled web applications to truly rival traditional applications was the creation and standardization of the XMLHttpRequest JavaScript API [137]. This API allowed the client-side JavaScript code to make asynchronous requests to the web application and then update the content of the web page according to the response from the web application. Combined together, these web application development technologies came to be known as AJAX [56], which rivaled traditional desktop applications in functionality.

This architecture of a web application is what we will use throughout this chapter to discuss the security aspects of web applications. In this dissertation, other details and complexities of web applications will be explained in the chapter where it is needed.

1.2 Web Application Vulnerabilities

The security properties of a web application are similar to the security of any other software system: confidentially of the data, integrity of the data, and availability of the application. In this dissertation, we will focus on attacks that compromise the confidentially or integrity of the web application's data.

1.2.1 Injection Vulnerabilities

This class of vulnerabilities occur when an attacker is able to control or influence the value of parameters that are used as part of an outside[3] query, command, or language. If the attacker can manipulate and change the semantics of the query, command, or language, and this manipulation compromises the security of the application, then that is an injection vulnerability.

There are many types of injection vulnerabilities in web applications, and the types depend on the query, command, or language that is being injected. These include SQL queries, HTML responses, OS commands, email headers, HTTP headers, and many more. Next we will focus on two of the most serious and prevalent classes of injection vulnerabilities in web applications: SQL injection and Cross-Site Scripting (XSS).

SQL Injection

SQL injection vulnerabilities, while declining in the number reported compared to XSS vulnerabilities, are still numerous and are incredibly critical when they occur.

[3]Outside from the perspective of the web application's server-side language.

```
1 $name = $_GET['name'];
2 $q = "select * from users where name = '" . $name . "';";
3 $result = mysql_query($q);
```

Listing 1.1: Example of an SQL injection vulnerability in a PHP web application. The attacker-controlled `$name` parameter is used unsanitized in the SQL query created on Line 2 and issued on Line 3.

The root cause of SQL injection vulnerabilities is that the server-side code of the web application, to issue an SQL query to the SQL database, concatenates strings together. This format allows the queries to be parameterized, and therefore the server-side code can be more general.

The code in Listing 1.1 shows a sample PHP web application that contains an SQL injection vulnerability. In Line 1 of this sample, the variable `$name` is set based on the value of the query parameter called `name`. The `$name` variable is used in Line 2 to construct an SQL query to look up the given user by name in the SQL table `users`. The web application issues the query on Line 3.

The problem is that, according to the server-side language, the resulting query is simply a string, whereas when that string is passed to the SQL server, the SQL server parses the string into a SQL query. Therefore, what the server-side code treats as a simple string is a complex language with syntax and semantics.

In Listing 1.1, the vulnerability comes from the fact that the query parameter `name` comes from the user and therefore may be modified by an attacker. As seen in the example, the `$name` variable is used in the SQL query to select based on the SQL `name` column. In order to do this, the programmer constrains the value to be in between matching `'` which is SQL query syntax for delimiting data. Therefore, for the attacker to alter the semantics of the query, the attacker need only input something like the following: `'or 1=1; #`. This input would cause the SQL query that the web application issues to the database to be the following:

```
select * from users where name = ''or 1=1; #';
```

The `#` is an SQL comment which means that everything after that in the query is ignored. Now the attacker has been able to alter the semantics of the SQL query, in this case by adding another SQL clause (`or 1=1`) that was not in the original statement.

Thus, in order for an attacker to not alter the semantics of the SQL query, a web developer must be careful to properly *sanitize* all potentially attacker-controlled input. Here, sanitize means to transform the input from the user to a form that renders it neutral in the target language. In the case of SQL, this typically means converting any `'` (which are used by an attacker to escape out of an SQL query) to the inert `\'`.

With an SQL injection vulnerability, an attacker can violate both the confidentially and integrity of the application's data. An attacker can insert arbitrary data into the database, potentially adding a new admin user to the web application. Also, an attacker can exfiltrate any data that the database user can access (typically all data that the web application can access). Finally, the attacker can also delete all of the web application's data. All of these consequences are the result of a single SQL injection vulnerability, and that is why SQL injection vulnerabilities can critically compromise a web application's security.

To prevent SQL injections with sanitization, a developer must be extremely careful that no user-supplied data is used in an SQL statement, including any paths that the data could have taken through the web application. In practice, this is (understandably) difficult for developers to always accomplish. Therefore, developers should use *prepared statements,* which is a way to tell the database the structure of an SQL query *before* the data is given. In this way, the database already knows the structure of the SQL query, and therefore there is no way for an attacker to alter the structure and semantics of the query. Almost every server-side language or framework has support for prepared statements. Unfortunately, even with widespread support for prepared statements, SQL injections are still frequently found in web applications.

Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerabilities are similar in spirit to SQL injection vulnerabilities. Instead of an injection into a SQL query, XSS vulnerabilities are injections into the HTML output that the web application generates. XSS vulnerabilities are frequently in the top three of reported vulnerabilities in *all* software systems.

```php
$name = $_GET['name'];
echo "Hello <b>" . $name . "</b>";
```

Listing 1.2: Example of a XSS vulnerability in a PHP web application. The attacker-controlled `$name` parameter is used unsanitized in the HTML output on Line 2.

The root cause of XSS vulnerabilities is that the server-side code of a web application, in order to create the web application's HTML response, essentially concatenates strings together.

Listing 1.2 shows an example PHP web application that has an XSS vulnerability. In Line 1, the variable `$name` is retrieved from the query parameter `name`. Then, `$name` is used in Line 2 as an argument to PHP's `echo` function, which sends its string argument to the HTTP response. The goal of this code is to output the user's name in bold. This is accomplished in HTML by wrapping the user's name in a bold tag (`<b>`).

If an attacker is able to control the HTML output of the web application, as the `$name` parameter in Listing 1.2, then the attacker can trick the user's web browser into executing the attacker's JavaScript. This can be accomplished in a variety of ways, one example would be inputting the following for the `name` query parameter:

```
<script>alert('xss');</script>
```

Matching `<script>` HTML tags is the way for the web application to tell the user's browser to execute JavaScript.

The fundamental building block of JavaScript security in the web browser is the *Same Origin Policy*. In essence, this security policy means that only JavaScript that comes from the same origin[4] can interact. In practice, what this means is that JavaScript running on a web browser from `hacker.com` cannot interact with or affect JavaScript running on the same web browser from `example.com`.

The name *Cross-Site Scripting* is derived from the fact that XSS circumvents the browser's Same Origin Policy. By using an XSS vulnerability, an attacker is able to trick a user's browser to execute JavaScript code of their choosing in the web application's origin. This is because, from the browser's perspective, the JavaScript came from the web application, so the browser happily executes the attacker's JavaScript along with the web application's JavaScript.

With an XSS vulnerability, an attacker can compromise a web application significantly. A popular XSS exploitation technique is to steal the web application's cookies and send them to the attacker. Typically the web application's cookies are used to authenticate and keep state with the web application, which could allow the attacker to impersonate the user.

By executing JavaScript in the same origin as the web application, the attacker's JavaScript has total control over the graphical appearance of the web page. What this means is that the attacker can completely alter the look of the web page, and could, for

[4]Here, we omit the definition of the same origin. We will define it later in the dissertation when necessary.

instance, force the page to resemble the web application's login form. However, once the user puts their information into the form, the attacker's JavaScript could steal that information. In this way, the attacker is able to phish the user's credentials, except in this instance the user is on the proper domain name for the web application.

Another insidious thing that an attacker's JavaScript can do if it executes in the user's browser is interact with the web application on behalf of the user[5]. In practice, what this means is that the attacker's JavaScript can interact with the web application, and the web application has no way of knowing that the requests did not come from the user. Imagine an attacker's JavaScript sending emails on a user's behalf or initiating a bank transfer.

XSS vulnerabilities can be fixed by proper sanitizaiton at all program points in the web application that output HTML. This sanitization process typically will convert entities that are significant in parsing HTML to their display equivalent. For instance, the HTML < character is transformed to its HTML entity equivalent `&gt;`, which means to display a < character on the resulting web page, rather than starting an HTML tag.

There are a number of practical difficulties that make properly sanitizing output for XSS vulnerabilities particularly challenging (especially when compared to SQL injection vulnerabilities). One difficulty is that, as shown by Saxena, Molnar, and

[5]This defeats any CSRF protection that the web application has enabled, as the attacker's JavaScript can read the web application's CSRF tokens.

Livshits [125], there are numerous types of sanitization for XSS vulnerabilities, and which type of sanitization to use depends on *where* the output is used in the resulting HTML page. This means that the developer must reason not only about all program paths that a variable may take to get to a specific program point (to see if an attacker can influence its value), but also about all the different places in the HTML output where the variable is used. The complex nature of XSS vulnerabilities contribute to the reason that it is still the most frequent web application vulnerability.

Unfortunately XSS vulnerabilities have no easy, widely supported fix, as prepared statements are to SQL injection vulnerabilities. However, in Chapter 6 we will look at an approach to fundamentally solve a large class of XSS vulnerabilities.

1.2.2 Logic Flaws

Logic flaws are a class of vulnerabilities that occur when the implemented logic of the web application does not match with the developer's intended logic of the web application. One popular example would be, on an ecommerce application, if a user is able to submit a coupon multiple times, until the price of the item is zero. Another example might be a financial services web application which accidentally sends confidential financial reports to unauthorized users.

An injection vulnerability can affect any web application, and the fix of the vulnerability will be the same, regardless of the underlying web application. In contrast, logic flaws are specific and unique to the web application. Identical behavior that appears in two web applications may be a logic flaw in one but a security vulnerability in the other. Consider the behavior of an unauthenticated user altering the content of a web page. In most applications, this would represent a vulnerability, however it is the core mechanic and defining feature of a wiki, such as Wikipedia. The distinguishing feature of logic flaw vulnerabilities is that the web application code is functioning correctly—that is, an attacker is not able to alter how the code executes or execute code of her choosing, however the behavior that the code executes violates the developer's security model of the application. Therefore, these vulnerabilities are incredibly difficult to detect in an automated fashion, as the automated tool must reverse engineer the developer's intended security model.

In Chapter 5, we will describe a novel class of logic flaw vulnerabilities called Execution After Redirect.

1.3 Securing Web Applications

Given their rise in popularity, ensuring that web applications are secure is critical. Security flaws in a web application can allow an attacker unprecedented access to secret and sensitive data.

There are numerous approaches to secure web applications, depending on where the defense is put into place. One approach is to detect attacks as they happen and block the attack traffic. Another approach is to construct the web application in a way such that it is not vulnerable to entire classes of security vulnerabilities. Finally, and the approach taken in the majority of this dissertation, is automated tools to automatically find vulnerabilities in web applications.

1.3.1 Anomaly Detection

One way to secure web applications is to have tools and approaches that look for attacks against web applications in the inbound web traffic [118]. There are many approaches in this area, but most of them involve first creating a model of the normal behavior of the web application. Then, after this model is created, a monitoring/detection phase starts which analyzes inbound web application traffic looking for anomalous web requests which signify an attack. Depending on the anomaly detection system, the request can be blocked or prevented at that time.

Anomaly detection systems are good for preventing unknown exploits against the web application. However, the effectiveness of the anomaly detection depends on the creation of the web application model and the presence of extensive attack-free traffic. In practice, it is difficult to automatically create extensive attack-free traffic.

Modern web application can use anomaly detection systems in production environments as a defense-in-depth approach.

1.3.2 Vulnerability Analysis Tools

Vulnerability analysis is the art of finding vulnerabilities in software. The idea is to find vulnerabilities either before an application is deployed or before an attacker is able to find the vulnerability.

Manual vulnerability analysis is when a team of humans manually analyze an application for vulnerabilities. These manual vulnerability analyses, frequently called *pentesting,* employ a team of experts to find vulnerabilities in a software system. The downside is that an expert's time is costly, and therefore, due to the cost, a company will very infrequently do an external pentest of its web applications.

Vulnerability analysis tools are automated approaches to find vulnerabilities in software. The goal of this type of software is to find all possible vulnerabilities in an application. The core idea is to develop software that can encapsulate a human security expert's knowledge.

Because vulnerability analysis tools are automated, they can be used against a variety of applications. Furthermore, they are significantly less expensive than hiring a team of human experts, so they can be used much more frequently throughout the software development process.

Vulnerability analysis tools can be categorized based on what information of the web application they use. In the following sections we will describe the difference between white-box, black-box, and grey-box vulnerability analysis tools.

White-Box

A white-box vulnerability analysis tool looks at the source code of the web application to find vulnerabilities. By analyzing the source code of the web application, a white-box tool can see *all* potential program paths throughout the application. This enables a white-box tool to potentially find vulnerabilities along all program paths. Typically approaches leverage ideas and techniques from the program analysis and static analysis communities to find vulnerabilities.

The biggest strength of white-box tools is that they are able to see all possible program paths through the application. However, as precisely identifying all vulnerabilities in an application via static analysis is equivalent to the halting problem, trade-offs must be made in order to create useful tools. The trade-off that is often made in white-box tools is one of being sound rather than complete. What this means is that a white-box tool will report vulnerabilities that are not actual vulnerabilities. This is usually because the static analysis will over-approximate the program paths that the application can take. Thus, there will be vulnerabilities reported that cannot occur in practice.

The downside of white-box tools is that they are tied to the specific language or framework. A white-box vulnerability analysis tool written for PHP will not work for Ruby on Rails without significant engineering work. These tools are tightly coupled to not only language features, but also framework features.

Black-Box

In contrast to white-box tools, black-box vulnerability analysis tools assume no knowledge of the source-code of the web application. Instead of using the source code, black-box tools interact with the web application being tested just as a user with a web browser would. Specifically, this means that the black-box tools issue HTTP requests to the web application and receive HTTP responses containing HTML. These HTML pages tell the black-box tool how to generate new HTTP requests to the application.

Black-box tools first will crawl the web application looking for all possible *injection vectors* into the web application. An injection vector is any way that an attacker can feed input into the web application. In practice, web application injection vectors are: URL parameters, HTML form parameters, HTTP cookies, HTTP headers, URL path, and so on.

Once the black-box tool has enumerated all possible injection vectors in the application, the next step is to give the web application input which is intended to trigger or expose a vulnerability in the web application. This process is typically called *fuzzing*.

The specifics of choosing which injection vectors to fuzz and when are specific to each black-box tool.

Finally, the black-box tool will analyze the HTML and HTTP response to the fuzzing attempts in order to tell if the attempt was successful. If it was, the black-box tool will report it as a vulnerability.

There are two major benefits of black-box tools as opposed to white-box tools. The first is that black-box tools are general and can find vulnerabilities in *any* web application, regardless of what language the server-side code is written in. In this way, black-box tools emulate an external hacker who has no access to the source code of the application. Therefore, black-box tools are applicable to a much larger number of web applications.

The second major benefit is that black-box tools have significantly lower false positives[6] than white-box tools. This is because the fuzzing attempt actually tries to trigger the vulnerability, and, for most web vulnerabilities, a successful exploitation will be evident in the resulting HTML page. Ultimately, lower false positives causes the developers who run these tools against their own web applications to trust the output of a black-box tool over a white-box tool.

The drawback of a black-box tool is that it is not guaranteed to find all vulnerabilities in your web application. This limitation is because a black-box tool can only find

[6] A *false positive* is a vulnerability that the tool reports which is not actually a vulnerability.

vulnerabilities along program paths that it executes, whereas a white-box tool can see all program paths through an application.

Grey-Box

As the name suggests, grey-box tools are a combination of white-box techniques and black-box techniques. The main idea is to use white-box static analysis techniques to generate possible vulnerabilities. Then, there is a confirmation step where the tool will actually try to exploit the vulnerability. Only if this step is successful will the tool report the vulnerability.

Grey-box tools inherit the benefits of white-box tools: The ability to find vulnerabilities in all program paths along with the low false positive rate associated with black-box tools (as the vulnerabilities are verified by the black-box techniques). However, grey-box tools also inherit the drawbacks of white-box tools: Applicability to a single web application language or framework. Therefore, these types of tools are not as popular as white-box and black-box tools.

1.4 Securing the Web

Given the empowering nature of web applications, it is clear that securing web applications is important. Specifically, we must focus on the needs of the users: making sure that their data is safe, and that they are safe while browsing the web. To accomplish this, I believe that we must make the necessary strides to create automated tools that are able to automatically find security vulnerabilities. These tools can be used by developers with no security expertise, thus putting developers on a level playing field with the attackers.

In this dissertation, I make the following contributions to securing web applications from attack:

- I methodically analyze existing black-box web application vulnerability scanners. We develop a known-vulnerable web application, then evaluate several real-world black-box web application vulnerability scanners to identify their strengths and weaknesses.
- Then, using the previously developed work as a guide, I aim to solve the biggest problem restricting modern black-box web application vulnerability scanners: They do not understand that they are analyzing a web *application* with state. I develop an approach to automatically reverse-engineer the state machine of a web application solely through black-box interactions. Incorporating this knowledge into a black-box web application vulnerability scanner enables the scanner to test significantly more of the web application.

- I identify and study a novel class of web application vulnerabilities, called Execution After Redirect, or EARs. These logic flaw vulnerabilities can affect web applications written in a number of languages or frameworks. In addition to studying this class of vulnerabilities, we developed a white-box static analysis tool to automatically identify EARs in Ruby on Rails web applications. By applying this tool to a large corpus of real-world open-source web application, we found many previously unknown vulnerabilities.

- Finally, I propose a new approach to fundamentally solve Cross-Site Scripting vulnerabilities. By using the fundamental security principles of Code and Data separation, we can view XSS vulnerabilities as a problem of Code and Data separation. New applications can be designed with Code and Data separation in mind, however it is difficult to separate Code and Data manually. To prevent XSS vulnerabilities in existing web applications, I created a tool to automatically perform Code and Data separation for legacy web applications. After applying this tool, the web applications are fundamentally secure from server-side XSS vulnerabilities.

Chapter 2

Related Work

Automated web application vulnerability analysis tools are an area of research that has received considerable study. In this chapter, we will discuss works related to different areas of web application vulnerability scanners: how black-box web vulnerability scanners are evaluated, the history of black-box and white-box tools, and finally the various proposed defenses for Cross-Site Scripting vulnerabilities.

2.1 Evaluating Black-Box Web Vulnerability Scanners

Our work on evaluating black-box vulnerability scanners in Chapter 3 is related to two main areas of research: the design of web applications for assessing vulnerability analysis tools and the evaluation of web scanners.

Designing test web applications. Vulnerable test applications are required to assess web vulnerability scanners. Unfortunately, no standard test suite is currently available or accepted by the industry and research community. HacmeBank [53] and WebGoat [105] are two well-known, publicly-available, vulnerable web applications, but their design is focused more on teaching web application security rather than testing automated scanners.

SiteGenerator [104] is a tool to generate sites with certain characteristics (e.g., classes of vulnerabilities) according to its input configuration. While SiteGenerator is useful to automatically produce different vulnerable sites, we found it easier to manually introduce in WackoPicko the vulnerabilities with the characteristics that we wished to test.

Evaluating web vulnerability scanners. There exists a growing body of literature on the evaluation of web vulnerability scanners. For example, Suto compared three scanners against three different applications and used code coverage, among other metrics, as a measure of the effectiveness of each scanner [134]. In a follow-up study, Suto [135] assessed seven scanners and compared their detection capabilities and the time required to run them. Wiegenstein et al. ran five unnamed scanners against a custom benchmark [144]. Unfortunately, the authors do not discuss in detail the reasons for detections or spidering failures. In their survey of web security assessment tools, Curphey and Araujo reported that black-box scanners perform poorly [39]. Peine examined in depth the functionality and user interfaces of seven scanners (three commercial) that were run against WebGoat and one real-world application [111]. Kals et al. developed a new web vulnerability scanner and tested it on approximately 25,000 live web pages [82]. Because no ground truth is available for these sites, the authors did not discuss false negative rate or failures of their tool. AnantaSec released an evaluation of three scanners against 13 real-world applications, three web applications provided by the scanner vendors, and a series of JavaScript tests [5]. While this experiment assessed a large number of real-world applications, only a limited number of scanners are tested and no explanation is given for the results. In addition, Vieira et al. tested four web vulnerability scanners on 300 web services [138]. They also report high rates of false positives and false negatives.

2.2 Black-Box Vulnerability Scanners

Automatic or semi-automatic web application vulnerability scanning has been a hot topic in research for many years because of its relevance and its complexity. In Chapter 4 we will discuss the creation of a new black-box vulnerability scanner technique. Here, we review the relevant literature.

Huang et al. developed a tool (WAVES) for assessing web application security with which we share many points [71]. Similarly to our work, they have a scanner for finding the entry points in the web application by mimicking the behavior of a web browser. They employ a learning mechanism to sensibly fill web form fields and allow deep crawling of pages behind forms. Attempts to discover vulnerabilities are carried out by submitting the same form multiple times with valid, invalid, and faulty inputs, and comparing the result pages. Differently from WAVES, we are using the knowledge gathered by the understanding of the web application's state to help the fuzzer detect the effect of a given input. Moreover, black-box vulnerability scanner aims not only at finding relevant entry-points, but rather at building a complete state-aware navigational map of the web application.

A number of tools have been developed to try to automatically discover vulnerabilities in web applications, produced as academic prototypes [11,48,61,72,81,82,89], as open-source projects [26,33,117], or as commercial products [2,70,73,113].

Multiple projects [14, 135, 138], as well as Chapter 3 tackled the task of evaluating the effectiveness of popular black-box scanners (in some cases also called *point-and-shoot* scanners). The common theme in their results is a relevant discrepancy in vulnerabilities found across scanners, along with low accuracy. Authors of these evaluations acknowledge the difficulties and challenges of the task [59, 138]. In particular, we highlighted how more deep crawling and reverse engineering capabilities of web applications are needed in black-box scanners, and we also developed the *WackoPicko* web application which contains known vulnerabilities described in Chapter 3. Similarly, Bau et al. investigated the presence of room for research in this area, and found improvement is needed, in particular for detecting second-order XSS and SQL injection attacks [14].

Reverse engineering of web applications has not been widely explored in the literature, to our knowledge. Some approaches [42] perform static analysis on the code to create UML diagrams of the application.

Static analysis, in fact, is the technique mostly employed for automatic vulnerability detection, often combined with dynamic analysis.

Halfond et al. developed a traditional black-box vulnerability scanner, but improved its results by leveraging a static analysis technique to better identify input vectors [61].

Pixy [81] employed static analysis with taint propagation in order to detect SQL injection, XSS, and shell command injection, while *Saner* [11] used sound static analysis to detect failures in sanitization routines. Saner also takes advantage of a second phase of dynamic analysis to reduce false positives. Similarly, *WebSSARI* [72] also employed static analysis for detecting injection vulnerabilities, but, in addition, it proposed a technique for runtime instrumentation of the web application through the insertion of proper sanitization routines.

Felmetsger et al. investigated an approach for detecting logic flaw vulnerabilities by combining execution traces and symbolic model checking [48]. Similar approaches are also used for generic bug finding (in fact, vulnerabilities are considered to be a subset of the general bug category). Csallner et al. employ dynamic traces for bug finding and for dynamic verification of the alerts generated by the static analysis phase [37]. Artzi et al., on the other hand, use symbolic execution and model checking for finding general bugs in web applications [6].

On a completely separate track, efforts to improve web application security push developers toward writing secure code in the first place. Security experts are tying to achieve this goal by either educating the developers [129] or designing frameworks which either prohibit the use of bad programming practices or enforce some security constraints in the code. Robertson and Vigna developed a strongly-typed framework which statically enforces separation between structure and content of a web page, preventing XSS and SQL injection [119]. Also Chong et al. designed their language for developers to build web applications with strong confidentiality and integrity guarantees, by means of compile-time and run-time checks [34].

Alternatively, consequences of vulnerabilities in web applications can be mitigated by attempting to prevent the attacks before they reach potentially vulnerable code, such as, for example, in the already mentioned *WebSSARI* [72] work. A different approach for blocking attacks is followed by Scott and Sharp, who developed a language for specifying a security policy for the web application; a gateway will then enforce these policies [126].

Another interesting research track deals with the problem of how to explore web pages behind forms, also called the *hidden web* [115]. McAllister et al. monitor user interactions with a web application to collect sensible values for HTML form submission and generate test cases that can be replayed to increase code coverage [95]. Although not targeted to security goals, the work of Raghavan and Garcia-Molina is relevant for their contribution in classification of different types of dynamic content and for their novel approach for automatically filling forms by deducing the domain of form fields [115]. Raghavan and Garcia-Molina carried out further research in this direction, by reconstructing complex and hierarchical query interfaces exposed by web applications.

Moreover, Amalfitano et al. started tackling the problem of reverse engineering the state machine of client-side AJAX code, which will help in finding the web application server-side entry points and in better understating complex and hierarchical query interfaces [4].

Finally, there is the work by Berg et al. in reversing state machines into a *Symbolic Mealy Machine* (SMM) model [15]. Their approach for reversing machines cannot be directly applied to the case of web applications because of the infeasibility of fully exploring all pages for all the states, even for a small subset of the possible states. Nevertheless, the model they propose for a SMM is a good starting point to model the web application's state.

2.3 Automated Discovery of Logic Flaws

In Chapter 5, we discuss and analyze a novel class of web application vulnerabilities, called Execution After Redirect. In this section, we review the relevant literature applicable to Execution After Redirect vulnerabilities and, more generally, logic flaws.

Isolated instances of Execution After Redirect (EAR) vulnerabilities have been previously identified. Hofstetter wrote a blog post alerting people to not forget to exit after a redirect when using the PHP framework CakePHP [66]. This discussion resulted in a bug being filed with the CakePHP team [27]. This bug was resolved by updating the CakePHP documentation to indicate the redirect method did not end execution [28].

Felmetsger et al. presented a white-box static analysis tool for J2EE servlets to automatically detect logic flaws in web applications. The tool, Waler, found Execution After Redirect vulnerabilities in a web application called Global Internship Management System (GIMS) [48]. However, neither Felmetsger nor Hofstetter identified EARs as a systemic flaw among web applications.

Wang et al. manually discovered logic flaws in the interaction of Cashier-as-a-Service (CaaS) APIs and the web applications that use them [140]. This work is interesting because there is a three-way interaction between the users, the CaaS, and the web application. In Chapter 5, we consider one specific type of logic flaw across many applications.

Our white-box EAR detection tool uses the Ruby Intermediate Language (RIL) developed by Furr et al. [54]. RIL was used by An et al. to introduce static typing to Ruby on Rails [68]. They use the resulting system, DRails, on eleven Rails applications to statically discover type errors. DRails parses Rails applications by compiling them to equivalent Ruby code, making implicit Rails conventions explicit. This differs from our tool, which operates directly on the Rails application's control flow graph. Moreover, we are looking for a specific logic flaw, while DRails is looking for type errors.

Chaudhuri and Foster built a symbolic analysis engine on top of DRails, called Rubyx [31]. They are able to analyze the security properties of Rails applications using symbolic execution. Rubyx detected XSS, CSRF, session manipulation, and unauthorized access in the seven applications tested. Due to the symbolic execution and verifying of path conditions, false positives are reduced. However, Rubyx requires the developer to manually specify an analysis script that defines invariants on used objects, as well as the security requirements of the applications. Our tool, on the other hand, operates on raw, unmodified Rails applications, and does not require any developer input. This is due to the different focus; we are focusing on one specific type of flaw, while Rubyx is broader and can verify different types of security violations.

The static analysis EAR detection tool that we develop and describe in Chapter 5 is also related to numerous white-box tools that have previously been published. Huang et al. were one of the first to propose a static analysis tool for a server-side scripting language, specifically PHP. They implemented taint propagation to detect XSS, SQL injection, and general injection [72]. Livshits and Lam proposed a static analysis technique for Java web applications that used points-to analysis for improved precision [92]. Their tool detected 29 instances of SQL injection, XSS, HTTP response splitting, and command injection in nine open-source applications. Jovanovic et al. developed Pixy, an open-source static analysis tool to discover XSS attacks by performing flow-sensitive, inter-procedural, and context-sensitive data flow analysis on PHP web applications [80]. They later improved Pixy, adding precise alias analysis, to discover hundreds of XSS vulnerabilities in three PHP applications, half of which were false positives [79]. Balzarotti et al. used static and dynamic analysis to develop MiMoSa, a tool that performs inter-module data flow analysis to discover attacks that leverage several modules, such as stored XSS. They found 27 data flow violations in five PHP web applications [12].

All of these static analysis tools differ from our white-box tool because we are not looking for injection vulnerabilities, but rather for unexpected execution that a developer did not intend.

2.4 Cross-Site Scripting Defense

A broad variety of approaches have been proposed to address different types of XSS, though no standard taxonomy exists to classify these attacks and defenses. In general, XSS defenses employ schemes for input sanitization or restrictions on script generation and execution. Differences among various techniques involve client- or server-side implementation and static or dynamic operation. We group and review XSS defenses in this context.

2.4.1 Server-Side Methods

There has been much previous research in server-side XSS defenses [11, 18, 58, 60, 80, 93, 101, 112, 123, 125, 133]. Server-based techniques aim for dynamically generated pages free of XSS vulnerabilities. This may involve validation or injection of appropriate sanitizers for user input, analysis of scripts to find XSS vulnerabilities, or automatic generation of XSS-free scripts.

Server-side sanitizer defenses either check existing sanitization for correctness or generate input encodings automatically to match usage context. For example, Saner [11] uses static analysis to track unsafe inputs from entry to usage, followed by dynamic analysis to test input cases for proper sanitization along these paths. SCRIPTGARD [125] is a complementary approach that assumes a set of "correct" sanitizers and inserts them to match the browser's parsing context. BEK [67] focuses on creating sanitization functions that are automatically analyzable for preciseness and correctness. Sanitization remains the main industry-standard defense against XSS and related vulnerabilities.

A number of server-side defenses restrict scripts included in server-generated pages. For example, XSS-GUARD [18] determines valid scripts dynamically and disallows unexpected scripts. The authors report performance overheads of up to 46% because of the dynamic evaluation of HTML and JavaScript. Templating approaches [58,119,123] generate correct-by-construction scripts that incorporate correct sanitization based on context. In addition, schemes based on code isolation [3,8,91] mitigate XSS by limiting DOM access for particular scripts, depending on their context.

Certain XSS defenses [78,80,92,94,101,112,124,136,145] use data-flow analysis or taint tracking to identify unsanitized user input included in a generated web page. These approaches typically rely on sanitization, encoding, and other means of separating unsafe inputs from the script code. Some schemes prevent XSS bugs dynamically, while others focus on static detection and elimination.

Other approaches [60,93,100] combine server-side processing with various client-side components, such as confinement of untrusted inputs and markup randomization. Such schemes may parse documents on the server and prevent any modifications of the resulting parse trees on the client. In addition, randomization of XHTML tags can render foreign script code meaningless, defeating many XSS injection attacks.

2.4.2 Client-Side Methods

Client-side XSS defenses [77,83,97,131,139,142] mitigate XSS while receiving or rendering untrusted web content. Some of these schemes rely on browser modifications or plug-ins, often reducing their practical applicability. Others use custom JavaScript libraries or additional client-side monitoring software. CSP [131] is a browser-based approach that allows only developer specified JavaScript to execute, and its incorporation into WWW standards should facilitate wide acceptance and support by all popular browsers.

Some client-side XSS defenses focus on detecting and preventing leakage of sensitive data. For example, Noxes [83] operates as a personal-firewall browser plug-in that extracts all static links from incoming web pages, prompting the user about disclosure of information via dynamically generated links. Vogt et al. [139] also aim to address this problem, but use taint-tracking analysis within a browser to check for sensitive data released via XSS attacks.

Client-side HTML security policies mitigate XSS via content restrictions, such as disallowing unsafe features or executing only “known good” scripts. Using a browser’s HTML parser, BEEP [77] constructs whitelists of scripts, much like XSS-GUARD’s server-side approach [18]. BEEP assumes that the web application has no dynamic scripts whose hashes cannot be pre-computed, limiting its practicality with modern web applications; moreover, it has been shown that even whitelisted scripts may be vulnerable to attacks [8]. Another custom content security policy is BLUEPRINT's page descriptions, which are interpreted and rendered safely by a custom JavaScript library [93]. Script policies enforced at runtime [62, 97] are also useful for mitigating XSS exploits.

In general, standardized HTML security policies [131, 142] offer promise as a means of escaping the recent proliferation of complex, often ad hoc XSS defenses. CSP simplifies the problem by enforcing fairly strong restrictions, such as disabling `eval()` and other dangerous APIs, prohibiting inline JavaScript, and allowing only local script resources to be loaded. While new web applications can be designed with CSP in mind, legacy code may require significant rewriting.

Chapter 3

An Analysis of Black-Box Web Application Vulnerability Scanners

First, we will turn our attention to the problem of black-box web application vulnerabilities scanners—that is, automated tools that attempt to find security vulnerabilities in web applications. The goal of this chapter is study the current state of black-box web application vulnerability scanners.

Web application vulnerabilities, such as cross-site scripting and SQL injection, are one of the most pressing security problems on the Internet today. In fact, web application vulnerabilities are widespread, accounting for the majority of the vulnerabilities reported in the Common Vulnerabilities and Exposures database [40]; they are frequent targets of automated attacks [128]; and, if exploited successfully, they enable serious attacks, such as data breaches [103] and drive-by-download attacks [114]. In this scenario, security testing of web applications is clearly essential.

A common approach to the security testing of web applications consists of using *black-box web vulnerability scanners*. These are tools that crawl a web application to enumerate all the reachable pages and the associated input vectors (e.g., HTML form fields and HTTP GET parameters), generate specially-crafted input values that are submitted to the application, and observe the application's behavior (e.g., its HTTP responses) to determine if a vulnerability has been triggered.

Web application scanners have gained popularity, due to their independence from the specific web application's technology, ease of use, and high level of automation. (In fact, web application scanners are often marketed as "point-and-click" pentesting tools.) In the past few years, they have also become a requirement in several standards, most notably, in the Payment Card Industry Data Security Standard [110].

Nevertheless, web application scanners have limitations. Primarily, as most testing tools, they provide no guarantee of soundness. Indeed, in the last few years, several reports have shown that state-of-the-art web application scanners fail to detect a significant number of vulnerabilities in test applications [5,111,134,135,144]. These reports are valuable, as they warn against the naive use of web application scanners (and the false sense of security that derives from it), enable more informed buying decisions, and prompt to rethink security compliance standards.

However, knowing that web application scanners miss vulnerabilities (or that, conversely, they may raise false alerts) is only part of the question. Understanding *why* these tools have poor detection performance is critical to gain insights into how current tools work and to identify open problems that require further research. More concretely, we seek to determine the root causes of the errors that web application scanners make, by considering all the phases of their testing cycle, from crawling, to input selection, to response analysis. For example, some of the questions that we want to answer are: Do web application scanners correctly handle JavaScript code? Can they detect vulnerabilities that are "deep" in the application (e.g., that are reachable only after correctly submitting complex forms)? Can they precisely keep track of the state of the application?

To do this, we built a realistic web application, called WackoPicko, and used it to evaluate eleven web application scanners on their ability to crawl complex web applications and to identify the associated vulnerabilities. More precisely, the WackoPicko application uses features that are commonly found in modern web applications and that make their crawling difficult, such as complex HTML forms, extensive JavaScript and Flash code, and dynamically-created pages. Furthermore, we introduced in the application's source code a number of vulnerabilities that are representative of the bugs commonly found in real-world applications. The eleven web application scanners that we tested include both commercial and open-source tools. We evaluated each of them under three different configuration settings, corresponding to increasing levels of manual intervention. We then analyzed the results produced by the tools in order to understand how the tools work, how effective they are, and what makes them fail. The ultimate goal of this effort is to identify which tasks are the most challenging for black-box vulnerability scanners and may require novel approaches to be tackled successfully.

The main contributions of this chapter are the following:

- We performed the most extensive and thorough evaluation of black-box web application vulnerability scanners so far.
- We identify a number of challenges that scanners need to overcome to successfully test modern web applications both in terms of crawling and attack analysis capabilities.
- We describe the design of a testing web site for web application scanners that composes crawling challenges with vulnerability instances. This site has been made available to the public and can be used by other researchers in the field.
- We analyze in detail *why* the web application vulnerability scanners succeed or fail and we identify areas that need further research.

3.1 Background

Before discussing the design of our tests, it is useful to briefly discuss the vulnerabilities that web application scanners try to identify and to present an abstract model of a typical scanner.

3.1.1 Web Application Vulnerabilities

Web applications contain a mix of traditional flaws (e.g., ineffective authentication and authorization mechanisms) and web-specific vulnerabilities (e.g., using user-provided inputs in SQL queries without proper sanitization). Here, we will briefly describe some of the most common vulnerabilities in web applications (for further details, the interested reader can refer to the OWASP Top 10 List, which tracks the most critical vulnerabilities in web applications [107]):

- **Cross-Site Scripting (XSS):** XSS vulnerabilities allow an attacker to execute malicious JavaScript code as if the application sent that code to the user. This is the first most serious vulnerability of the OWASP Top 10 List, and WackoPicko includes five different XSS vulnerabilities, both reflected and stored.
- **SQL Injection:** SQL injection vulnerabilities allow one to manipulate, create or execute arbitrary SQL queries. This is the second most serious vulnerability on the OWASP Top 10 List, and the WackoPicko web application contains both a reflected and a stored SQL injection vulnerability.
- **Code Injection:** Code injection vulnerabilities allow an attacker to execute arbitrary commands or execute arbitrary code. This is the third most serious vulnerability on the OWASP Top 10 List, and WackoPicko includes both a command line injection and a file inclusion vulnerability (which might result in the execution of code).

- **Broken Access Controls:** A web application with broken access controls fails to properly define or enforce access to some of its resources. This is the tenth most serious vulnerability on the OWASP Top 10 List, and WackoPicko has an instance of this kind of vulnerability.

3.1.2 Web Application Scanners

In abstract, web application scanners can be seen as consisting of three main modules: a *crawler* module, an *attacker* module, and an *analysis* module. The crawling component is seeded with a set of URLs, retrieves the corresponding pages, and follows links and redirects to identify all the reachable pages in the application. In addition, the crawler identifies all the input points to the application, such as the parameters of GET requests, the input fields of HTML forms, and the controls that allow one to upload files.

The attacker module analyzes the URLs discovered by the crawler and the corresponding input points. Then, for each input and for each vulnerability type for which the web application vulnerability scanner tests, the attacker module generates values that are likely to trigger a vulnerability. For example, the attacker module would attempt to inject JavaScript code when testing for XSS vulnerabilities, or strings that have a special meaning in the SQL language, such as ticks and SQL operators, when testing for SQL injection vulnerabilities. Input values are usually generated using heuristics or using predefined values, such as those contained in one of the many available XSS and SQL injection cheat-sheets [121, 122].

The analysis module analyzes the pages returned by the web application in response to the attacks launched by the attacker module to detect possible vulnerabilities and to provide feedback to the other modules. For example, if the page returned in response to input testing for SQL injection contains a database error message, the analysis module may infer the existence of a SQL injection vulnerability.

3.2 The WackoPicko Web Site

A preliminary step for assessing web application scanners consists of choosing a web application to be tested. We have three requirements for such an application: it must have clearly defined vulnerabilities (to assess the scanner's detection performance), it must be easily customizable (to add crawling challenges and experiment with different types of vulnerabilities), and it must be representative of the web applications in use today (in terms of functionality and of technologies used).

We found that existing applications did not satisfy our requirements. Applications that deliberately contain vulnerabilities, such as HacmeBank [53] and WebGoat [105], are often designed to be educational tools rather than realistic testbeds for scanners. Others, such as SiteGenerator [104], are well-known, and certain scanners may be optimized to perform well on them. An alternative then is to use an older version of an open-source application that has known vulnerabilities. In this case, however, we would not be able to control and test the crawling capabilities of the scanners, and there would be no way to establish a false negative rate.

Therefore, we decided to create our own test application, called WackoPicko. It is important to note that WackoPicko is a realistic, fully functional web application. As opposed to a simple test application that contains just vulnerabilities, WackoPicko tests the scanners under realistic conditions. To test the scanners' support for client-side JavaScript code, we also used the open source Web Input Vector Extractor Teaser (WIVET). WIVET is a synthetic benchmark that measures how well a crawler is able to discover and follow links in a variety of formats, such as JavaScript, Flash, and form submissions.

3.2.1 Design

WackoPicko is a photo sharing and photo-purchasing site. A typical user of WackoPicko is able to upload photos, browse other user's photos, comment on photos, and purchase the rights to a high-quality version of a photo.

Authentication. WackoPicko provides personalized content to registered users. Despite recent efforts for a unified login across web sites [108], most web applications require a user to create an account in order to utilize the services offered. Thus, WackoPicko has a user registration system. Once a user has created an account, he/she can log in to access WackoPicko's restricted features.

Upload Pictures. When a photo is uploaded to WackoPicko by a registered user, other users can comment on it, as well as purchase the right to a high-quality version.

Comment On Pictures. Once a picture is uploaded into WackoPicko, all registered users can comment on the photo by filling out a form. Once created, the comment is displayed, along with the picture, with all the other comments associated with the picture.

Purchase Pictures. A registered user on WackoPicko can purchase the high-quality version of a picture. The purchase follows a multi-step process in which a shopping cart is filled with the items to be purchased, similar to the process used in e-commerce sites. After pictures are added to the cart, the total price of the cart is reviewed, discount coupons may be applied, and the order is placed. Once the pictures are purchased, the user is provided with links to the high-quality version of the pictures.

Search. To enable users to easily search for various pictures, WackoPicko provides a search toolbar at the top of every page. The search functionality utilizes the tag field that was filled out when the picture was uploaded. After a query is issued, the user is presented with a list of all the pictures that have tags that match the query.

Guestbook. A guestbook page provides a way to receive feedback from all visitors to the WackoPicko web site. The form used to submit feedback contains a "name" field and a "comment" field.

Admin Area. WackoPicko has a special area for administrators only, which has a different login mechanism than regular users. Administrators can perform special actions, such as deleting user accounts, or changing the tags of a picture.

3.2.2 Vulnerabilities

The WackoPicko web site contains sixteen vulnerabilities that are representative of vulnerabilities found in the wild, as reported by the OWASP Top 10 Project [107]. In the following we provide a brief description of each vulnerability.

Publicly Accessible Vulnerabilities

A number of vulnerabilities in WackoPicko can be exploited without first logging into the web site.

Reflected XSS: There is a XSS vulnerability on the search page, which is accessible without having to log into the application. In fact, the query parameter is not sanitized before being echoed to the user. The presence of the vulnerability can be tested by setting the query parameter to `<script>alert('xss')</script>`. When this string is reflected to the user, it will cause the browser to display an alert message. (Of course, an attacker would leverage the vulnerability to perform some malicious activity rather than alerting the victim.)

Stored XSS: There is a stored XSS vulnerability in the guestbook page. The `comment` field is not properly escaped, and therefore, an attacker can exploit this vulnerability by creating a comment containing JavaScript code. Whenever a user visits the guestbook page, the attack will be triggered and the (possibly malicious) JavaScript code executed.

Session ID: The session information associated with administrative accounts is handled differently than the information associated with the sessions of normal users. The functionality associated with normal users uses PHP's session handling capabilities, which is assumed to be free of any session-related vulnerabilities (e.g., session fixation, easily-guessable session IDs). However the admin section uses a custom session cookie to keep track of sessions. The value used in the cookie is a non-random value that is incremented when a new session is created. Therefore, an attacker can easily guess the session id and access the application with administrative rights.

Weak password: The administrative account page has an easily-guessable username and password combination: admin/admin.

Reflected SQL Injection: WackoPicko contains a reflected SQL injection vulnerability in the `username` field of the login form. By introducing a tick into the `username` field it is possible to perform arbitrary queries in the database and obtain, for example, the usernames and passwords of all the users in the system.

Command Line Injection: WackoPicko provides a simple service that checks to see if a user's password can be found in the dictionary. The `password` parameter of the form used to request the check is used without sanitization in the shell command: `grep ^<password>$ /etc/dictionaries-common/words`. This can be exploited by providing as the password value a dollar sign (to close grep's regular expression), followed by a semicolon (to terminate the grep command), followed by extra commands.

File Inclusion: The admin interface is accessed through a main page, called *index.php*. The index page acts as a portal; any value that is passed as its `page` parameter will be concatenated with the string ".php", and then the resulting PHP script will be run. For instance, the URL for the admin login page is `/admin/index.php?page=login`. On the server side, *index.php* will execute *login.php* which displays the form. This design is inherently flawed, because it introduces a file inclusion vulnerability. An attacker can exploit this vulnerability and execute remote PHP code by supplying, for example, `http://hacker/blah.php%00` as the `page` parameter to *index.php*. The `%00` at the end of the string causes PHP to ignore the ".php" that is appended to the page parameter. Thus *index.php* will download and execute the code at `http://hacker/blah.php`.

Unauthorized File Exposure: In addition to executing remote code, the file inclusion vulnerability can also be exploited to expose local files. Passing `/etc/passwd%00` as the "page" GET parameter to *index.php* of the admin section will cause the contents of the `/etc/passwd` file to be disclosed.

Reflected XSS Behind JavaScript: On WackoPicko's home page there is a form that checks if a file is in the proper format for WackoPicko to process. This form has two parameters, a file parameter and a name parameter. Upon a successful upload, the name is echoed back to the user unsanitized, and therefore, this represents a reflected vulnerability. However, the form is dynamically generated using JavaScript, and the target of the form is dynamically created by concatenating strings. This prevents a crawler from using simple pattern matching to discover the URL used by the form.

Parameter Manipulation: The WackoPicko home page provides a link to a sample profile page. The link uses the "userid" GET parameter to view the sample user (who has id of 1). An attacker can manipulate this variable to view any profile page without having a valid user account.

Vulnerabilities Requiring Authentication

A second class of vulnerabilities in WackoPicko can be exploited only after logging into the web site.

Stored SQL Injection: When users create an account, they are asked to supply their first name. This supplied value is then used unsanitized on a page that shows other users who have a similar first name. An attacker can exploit this vulnerability by creating a user with the name "' ; DROP users;#" then visiting the similar users page.

Directory Traversal: When uploading a picture, WackoPicko copies the file uploaded by the user to a subdirectory of the *upload* directory. The name of the subdirectory is the user-supplied tag of the uploaded picture. A malicious user can manipulate the tag parameter to perform a directory traversal attack. More precisely, by pre-pending "`../../`" to the tag parameter the attacker can reference files outside the upload directory and overwrite them.

Multi-Step Stored XSS: Similar to the stored XSS attack that exists on the guestbook, comments on pictures are susceptible to a stored XSS attack. However, this vulnerability is more difficult to exploit because the user must be logged in and must confirm the preview of the comment before the attack is actually triggered.

Forceful Browsing: One of the central ideas behind WackoPicko is the ability of users to purchase the rights to high-quality versions of pictures. However, the access to the links to the high-quality version of the picture is not checked, and an attacker who acquires the URL of a high-quality picture can access it without creating an account, thus bypassing the authentication logic.

Logic Flaw: The coupon system suffers from a logic flaw, as a coupon can be applied multiple times to the same order reducing the final price of an order to zero.

Reflected XSS Behind Flash: On the user's home page there is a Flash form that asks the user for his/her favorite color. The resulting page is vulnerable to a reflected XSS attack, where the "value" parameter is echoed back to the user without being sanitized.

3.2.3 Crawling Challenges

Crawling is arguably the most important part of a web application vulnerability scanner; if the scanner's attack engine is poor, it *might* miss a vulnerability, but if its crawling engine is poor and cannot reach the vulnerability, then it will *surely* miss the vulnerability. Because of the critical nature of crawling, we have included several types of crawling challenges in WackoPicko, some of which hide vulnerabilities.

HTML Parsing. Malformed HTML makes it difficult for web application scanners to crawl web sites. For instance, a crawler must be able to navigate HTML frames and be able to upload a file. Even though these tasks are straightforward for a human user with a regular browser, they represent a challenge for crawlers.

Multi-Step Process. Even though most web sites are built on top of the stateless HTTP protocol, a variety of techniques are utilized to introduce state into web applications. In order to properly analyze a web site, web application vulnerability scanners must be able to understand the state-based transactions that take place. In WackoPicko, there are several state-based interactions.

Infinite Web Site. It is often the case that some dynamically-generated content will create a very large (possibly infinite) crawling space. For example, WackoPicko has the ability to display a daily calendar. Each page of the calendar displays the agenda for a given day and links to the page for the following day. A crawler that naively followed the links in the WackoPicko's calendar would end up trying to visit an infinite sequence of pages, all generated dynamically by the same component.

Authentication. One feature that is common to most web sites is an authentication mechanism. Because this is so prevalent, scanners must properly handle authentication, possibly by creating accounts, logging in with valid credentials, and recognizing actions that log the crawler out. WackoPicko includes a registration and login system to test the scanner's crawlers ability to handle the authentication process correctly.

Client-side Code. Being able to parse and understand client-side technologies presents a major challenge for web application vulnerability scanners. WackoPicko includes vulnerabilities behind a JavaScript-created form, as well as behind a Flash application.

Link Extraction. We also tested the scanners on WIVET, an open-source benchmark for web link extractors [106]. WIVET contains 54 tests and assigns a final score to a crawler based on the percent of tests that it passes. The tests require scanners to analyze simple links, multi-page forms, links in comments and JavaScript actions on a variety

Name	Version Used	License	Type	Price
Acunetix	6.1 Build 20090318	Commercial	Standalone	$4,995-$6,350
AppScan	7.8.0.0 iFix001 Build: 570 Security Rules Version 647	Commercial	Standalone	$17,550-$32,500
Burp	1.2	Commercial	Proxy	£125 ($190.82)
Grendel-Scan	1.0	GPLv3	Standalone	N/A
Hailstorm	5.7 Build 3926	Commercial	Standalone	$10,000
Milescan	1.4	Commercial	Proxy	$495-$1,495
N-Stalker	2009 - Build 7.0.0.207	Commercial	Standalone	$899-$6,299
NTOSpider	3.2.067	Commercial	Standalone	$10,000
Paros	3.2.13	Clarified Artistic License	Proxy	N/A
w3af	1.0-rc2	GPLv2	Standalone	N/A
Webinspect	7.7.869.0	Commercial	Standalone	$6,000-$30,000

Table 3.1: Characteristics of the scanners evaluated.

of HTML elements. There are also AJAX-based tests as well as Flash-based tests. In our tests, we used WIVET version number 129.

3.3 Experimental Evaluation

We tested 11 web application scanners by running them on our WackoPicko web site. The tested scanners included 8 proprietary tools and 3 open source programs. Their cost ranges from free to tens of thousands of dollars. We used evaluation versions of each software, however they were fully functional. A summary of the characteristics of the scanners we evaluated is given in Table 3.1.

We ran the WackoPicko web application on a typical LAMP machine, which was running Apache 2.2.9, PHP 5.2.6, and MySQL 5.0.67. We enabled the `allow_url_fopen` and `allow_url_include` PHP options and disabled the `magic_quotes` option. We ran the scanners on a machine with a Pentium 4 3.6GHz CPU, 1024 MB of RAM, and Microsoft Windows XP, Service Pack 2.

3.3.1 Setup

The WackoPicko server used in testing the web vulnerability scanners was run in a virtual machine, so that before each test run the server could be put in an identical initial state. This state included ten regular users, nine pictures, and five administrator users.

Each scanner was run in three different configuration modes against WackoPicko, with each configuration requiring more setup on the part of the user. In all configuration styles, the default values for configuration parameters were used, and when choices were required, sensible values were chosen. In the INITIAL configuration mode, the scanner was directed to the initial page of WackoPicko and told to scan for all vulnerabilities. In the CONFIG setup, the scanner was given a valid username/password combination or login macro before scanning. MANUAL configuration required the most work on the part of the user; each scanner was put into a "proxy" mode and then the user browsed to each vulnerable page accessible without credentials; then, the user logged in and visited each vulnerability that required a login. Additionally a picture was uploaded, the rights to a high-quality version of a picture were purchased, and a coupon was applied to the order. The scanner was then asked to scan the WackoPicko web site.

Scanner	Reflected XSS	Stored XSS	Reflected SQL Injection	Command-line Injection
Acunetix	INITIAL	INITIAL	INITIAL	
AppScan	INITIAL	INITIAL	INITIAL	
Burp	INITIAL	MANUAL	INITIAL	INITIAL
Grendel-Scan	MANUAL		CONFIG	
Hailstorm	INITIAL	CONFIG	CONFIG	
Milescan	INITIAL	MANUAL	CONFIG	
N-Stalker	INITIAL	MANUAL	MANUAL	
NTOSpider	INITIAL	INITIAL	INITIAL	
Paros	INITIAL	INITIAL	CONFIG	
w3af	INITIAL	MANUAL	INITIAL	
Webinspect	INITIAL	INITIAL	INITIAL	

Scanner	File Inclusion	File Exposure	XSS via JavaScript	XSS via Flash
Acunetix	INITIAL	INITIAL	INITIAL	
AppScan	INITIAL	INITIAL		
Burp		INITIAL		MANUAL
Grendel-Scan				
Hailstorm				MANUAL
Milescan				
N-Stalker		INITIAL	INITIAL	MANUAL
NTOSpider				
Paros				MANUAL
w3af	INITIAL			MANUAL
Webinspect	INITIAL		INITIAL	MANUAL

Table 3.2: Detection results. For each scanner, the simplest configuration that detected a vulnerability is given. Empty cells indicate no detection in any mode.

3.3.2 Detection Results

The results of running the scanners against the WackoPicko site are shown in Table 3.2 and, graphically, in Figure 3.1. The values in the table correspond to the simplest configuration that discovered the vulnerability. An empty cell indicates that the given scanner did not discover the vulnerability in any mode. The table only reports the vulnerabilities that were detected by at least one scanner. Further analysis of why the scanners missed certain vulnerabilities is contained in Sections 3.3.3 and 3.3.4.

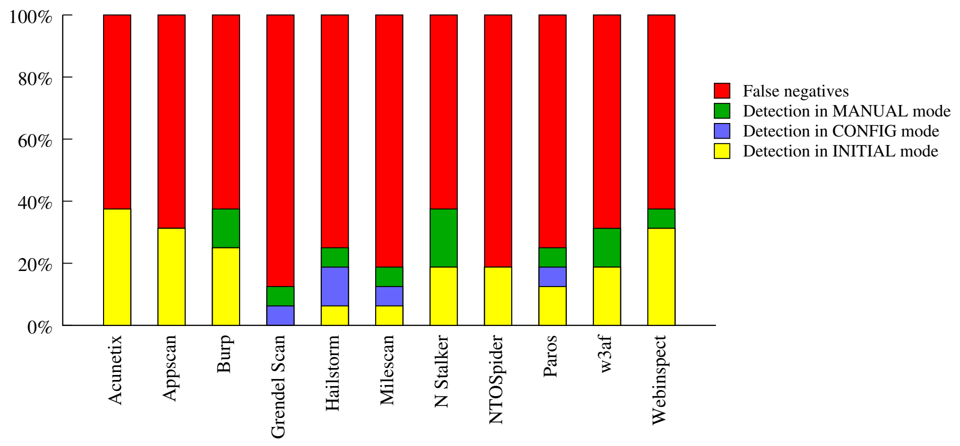

Figure 3.1: Detection performance (true positives and false negatives) of the evaluated scanners.

The running time of the scanners is shown in Figure 3.2. These times range from 74 seconds for the fastest tool (Burp) to 6 hours (N-Stalker). The majority of the scanners completed the scan within a half hour, which is acceptable for an automated tool.

False Negatives

One of the benefits of developing the WackoPicko web application to test the scanners is the ability for us to measure the false negatives of the scanners. An ideal scanner would be able to detect all vulnerabilities. In fact, we had a group composed of students with average security skills analyze WackoPicko. The students found all vulnerabilities except for the forceful browsing vulnerability. The automated scanners did not do as well; there were a number of vulnerabilities that were not detected by any scanner. These vulnerabilities are discussed hereinafter.

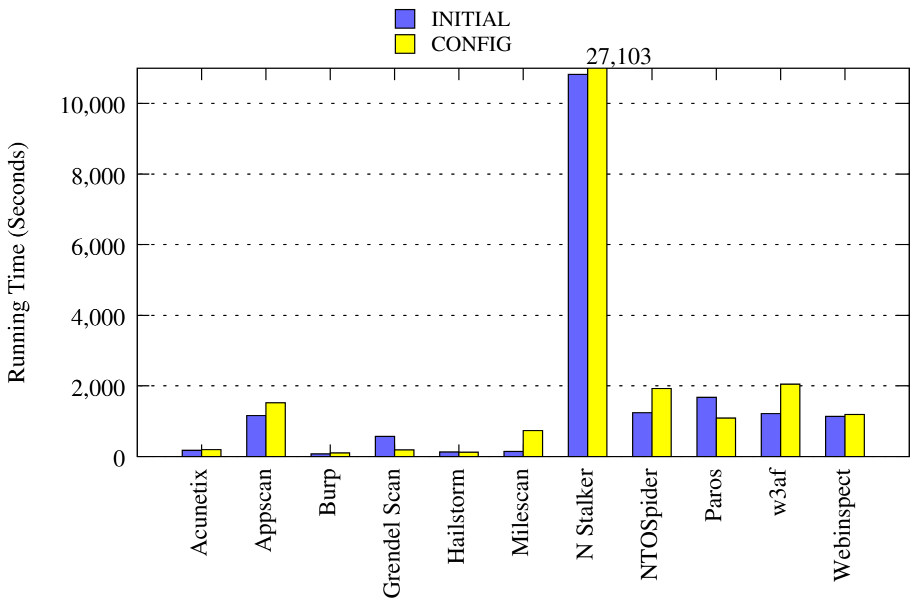

Figure 3.2: A graph of the time that it took each of the scanners to finish looking for vulnerabilities.

Session ID: No scanner was able to detect the session ID vulnerability on the admin login page. The vulnerability was not detected because the scanners were not given a valid username/password combination for the admin interface. This is consistent with what would happen when scanning a typical application, as the administration interface would include powerful functionality that the scanner should not invoke, like view, create, edit or delete sensitive user data. The session ID was only set on a successful login, which is why this vulnerability was not detected by any scanner.

Weak Password: Even though the scanners were not given a valid username/password combination for the administrator web site, an administrator account with the combination of admin/admin was present on the system. NTOSpider was the only scanner that successfully logged in with the admin/admin combination. However, it did not report it as an error, which suggests that it was unable to detect that the login was successful, even though the response that was returned for this request was different from every other login attempt.

Parameter Manipulation: The parameter manipulation vulnerability was not discovered by any scanner. There were two causes for this: first, only three of the scanners (AppScan, NTOSpider, and w3af) input a different number than the default value "1" to the `userid` parameter. Of the three, only NTOSpider used a value that successfully manipulated the `userid` parameter. The other reason was that in order to successfully detect a parameter manipulation vulnerability, the scanner needs to determine which pages require a valid username/password to access and which ones do not and it is clear that none of the scanners make this determination.

Stored SQL Injection: The stored SQL injection was also not discovered by any scanners, due to the fact that a scanner must create an account to discover the stored SQL injection. The reasons for this are discussed in more detail in Section 3.3.4.

Directory Traversal: The directory traversal vulnerability was also not discovered by any of the scanners. This failure is caused by the scanners being unable to upload a picture. We discuss this issue in Section 3.3.4, when we analyze how each of the scanners behaved when they had to upload a picture.

Name	INITIAL	CONFIG	MANUAL
Acunetix	1	7	4
AppScan	11	20	26
Burp	1	2	6
Grendel-Scan	15	16	16
Hailstorm	3	11	3
Milescan	0	0	0
N-Stalker	5	0	0
NTOSpider	3	1	3
Paros	1	1	1
w3af	1	1	9
Webinspect	215	317	297

Table 3.3: False positives.

Multi-Step Stored XSS: The stored XSS vulnerability that required a confirmation step was also missed by every scanner. In Section 3.3.4, we analyze how many of the scanners were able to successfully create a comment on a picture.

Forceful Browsing: No scanner found the forceful browsing vulnerability, which is not surprising since it is an application-specific vulnerability. These vulnerabilities are difficult to identify without access to the source code of the application [12].

Logic Flaw: Another vulnerability that none of the scanners uncovered was the logic flaw that existed in the coupon management functionality. Also in this case, some domain knowledge about the application is needed to find the vulnerability.

False Positives

The total number of false positives for each of the scanning configurations are show in Table 3.3. The number of false positives that each scanner generates is an important metric, because the greater the number of false positives, the less useful the tool is to the end user, who has to figure out which of the vulnerabilities reported are actual flaws and which are spurious results of the analysis.

The majority of the false positives across all scanners were due to a supposed "Server Path Disclosure." This is an information leakage vulnerability where the server leaks the paths of local files, which might give an attacker hints about the structure of the file system.

An analysis of the results identified two main reasons why these false positives were generated. The first is that while testing the application for file traversal or file injection vulnerabilities, some of the scanners passed parameters with values of file names, which, on some pages (e.g., the guestbook page), caused the file name to be included in that page's contents. When the scanner then tested the page for a Server Path Disclosure, it found the injected values in the page content, and generated a Server Path Disclosure vulnerability report. The other reason for the generation of false positives is that WackoPicko uses absolute paths in the `href` attribute of anchors (e.g., `/users/home.php`), which the scanner mistook for the disclosure of paths in the local system. Webinspect generated false positives because of both the above reasons, which explains the large amount of false positives produced by the tool.

Some scanners reported genuine false positives: Hailstorm reported a false XSS vulnerability and two false PHP code injection vulnerabilities, NTOSpider reported three false XSS vulnerabilities and w3af reported a false PHP `eval()` input injection vulnerability.

Measuring and Comparing Detection Capabilities

Comparing the scanners using a single benchmark like WackoPicko does not represent an exhaustive evaluation. However, we believe that the results provide insights about the current state of black-box web application vulnerability scanners.

One possible way of comparing the results of the scanners is arranging them in a lattice. This lattice is ordered on the basis of *strict dominance*. Scanner A *strictly dominates* Scanner B if and only if for every vulnerability discovered by Scanner B, Scanner A discovered that vulnerability with the same configuration level or simpler, and Scanner A either discovered a vulnerability that Scanner B did not discover or Scanner A discovered a vulnerability that Scanner B discovered, but with a simpler configuration. *Strictly dominates* has the property that any assignment of scores to vulnerabilities must preserve the *strictly dominates* relationship.

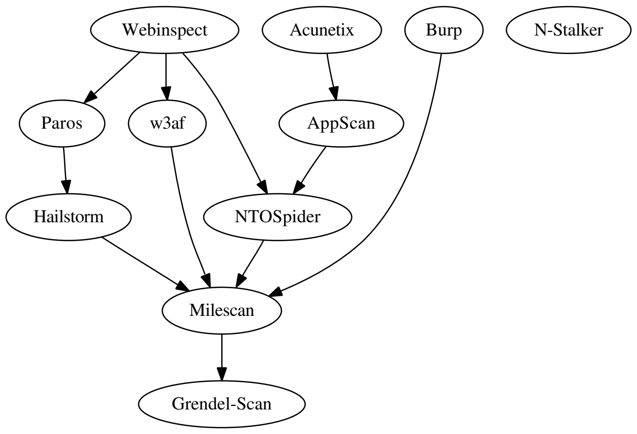

Figure 3.3: Dominates graph.

Figure 3.3 shows the *strictly dominates* graph for the scanners, where a directed edge from Scanner A to Scanner B means that Scanner A *strictly dominates* Scanner B. Because *strictly dominates* is transitive, if one scanner *strictly dominates* another it also *strictly dominates* all the scanners that the dominated scanner dominates, therefore, all redundant edges are not included. Figure 3.3 is organized so that the scanners in the top level are those that are not *strictly dominated* by any scanners. Those in the second level are strictly dominated by only one scanner and so on, until the last level, which contains those scanners that strictly dominate no other scanner.

Some interesting observations arise from Figure 3.3. N-Stalker does not strictly dominate any scanner and no scanner strictly dominates it. This is due to the unique combination of vulnerabilities that N-Stalker discovered and missed. Burp is also interesting due to the fact that it only dominates two scanners but no scanner dominates Burp because it was the only scanner to discover the command-line injection vulnerability.

Name	Detection	INITIAL Reachability	CONFIG Reachability	MANUAL Reachability
XSS Reflected	1	0	0	0
XSS Stored	2	0	0	0
SessionID	4	0	0	0
SQL Injection Reflected	1	0	0	0
Commandline Injection	4	0	0	0
File Inclusion	3	0	0	0
File Exposure	3	0	0	0
XSS Reflected behind JavaScript	1	3	3	0
Parameter Manipulation	8	0	0	0
Weak password	3	0	0	0
SQL Injection Stored Login	7	7	3	3
Directory Traversal Login	8	8	6	4
XSS Stored Login	2	8	7	6
Forceful Browsing Login	8	7	6	3
Logic Flaws - Coupon	9	9	8	6
XSS Reflected behind flash	1	9	7	1

Table 3.4: Vulnerability scores.

While Figure 3.3 is interesting, it does not give a way to compare two scanners where one does not strictly dominate the other. In order to compare the scanners, we assigned scores to each vulnerability present in WackoPicko. The scores are displayed in Table 3.4. The "Detection" score column in Table 3.4 is how many points a scanner is awarded based on how difficult it is for an automated tool to detect the existence of the vulnerability. In addition to the "Detection" score, each vulnerability is assigned a "Reachability" score, which indicates how difficult the vulnerability is to reach (i.e., it reflects the difficulty of crawling to the page that contains the vulnerability). There are three "Reachability" scores for each vulnerability, corresponding to how difficult it is

Name	Score
Acunetix	14
Webinspect	13
Burp	13
N-Stalker	13
AppScan	10
w3af	9
Paros	6
Hailstorm	6
NTOSpider	4
Milescan	4
Grendel-Scan	3

Table 3.5: Final ranking.

for a scanner to reach the vulnerability when run in INITIAL, CONFIG, or MANUAL mode. Of course, these vulnerability scores are subjective and depend on the specific characteristics of our WackoPicko application. However, their values try to estimate the crawling and detection difficulty of each vulnerability in this context.

The final score for each scanner is calculated by adding up the “Detection” score for each vulnerability the scanner detected and the “Reachability” score for the configuration (INITIAL, CONFIG and MANUAL) used when running the scanner. In the case of a tie, the scanners were ranked by how many vulnerabilities were discovered in INITIAL mode, which was enough to break all ties. Table 3.5 shows the final ranking of the scanners.

3.3.3 Attack and Analysis Capabilities

Analyzing how each scanner attempted to detect vulnerabilities gives us insight into how these programs work and illuminates areas for further research. First, the scanner would crawl the site looking for injection points, typically in the form of GET or POST parameters. Once the scanner identifies all the inputs on a page, it then attempts to inject values for each parameter and observes the response. When a page has more than one input, each parameter is injected in turn, and generally no two parameters are injected in the same request. However, scanners differ in what they supply as values of the non-injected parameters: some have a default value like `1234` or `Peter Wiener`, while others leave the fields blank. This has an impact on the results of the scanner, for example the WackoPicko guestbook requires that both the `name` and `comment` fields are present before making a comment, and thus the strategy employed by each scanner can affect the effectiveness of the vulnerability scanning process.

When detecting XSS attacks, most scanners employed similar techniques, some with a more sophisticated attempt to evade possible filters than others. One particularly effective strategy employed was to first input random data with various combinations of dangerous characters, such as `/ ,",',<, and >`, and then, if one of these combinations was found unchanged in the response, to attempt the injection of the full range of XSS attacks. This technique speeds up the analysis significantly, because the full XSS attack is not attempted against every input vector. Differently, some of the scanners took an exhaustive approach, attempting the full gamut of attacks on every combination of inputs.

When attempting a XSS attack, the thorough scanners would inject the typical `<script> alert('xss') </script>` as well as a whole range of XSS attack strings, such as JavaScript in a tag with the `onmouseover` attribute, in an `img`, `div` or `meta` tag, or `iframe`. Other scanners attempted to evade filters by using a different JavaScript function other than `alert`, or by using a different casing of `script`, such as `ScRiPt`.

Unlike with XSS, scanners could not perform an easy test to exclude a parameter from thorough testing for other Unsanitized Input vulnerabilities because the results of a successful exploit might not be readily evident in the response. This is true for the command-line injection on the WackoPicko site, because the output of the injectable command was not used in the response. Burp, the only scanner that was able to successfully detect the command line injection vulnerability, did so by injecting `` `ping -c 100 localhost` `` and noticing that the response time for the page was much slower than when nothing was injected.

This pattern of measuring the difference in response times was also seen in detecting SQL injections. In addition to injecting something with a SQL control character, such as tick or quote and seeing if an error is generated, the scanners also used a time-delay SQL injection, inputting `waitfor delay '0:0:20'` and seeing if the execution was delayed. This is a variation of the technique of using time-delay SQL injection to extract database information from a blind SQL vulnerability.

When testing for File Exposure, the scanners were typically the same; however one aspect caused them to miss the WackoPicko vulnerability. Each scanner that was looking for this vulnerability input the name of a file that they knew existed on the system, such as `/etc/passwd` on UNIX-like systems or `C:\boot.ini` for Windows. The scanners then looked for known strings in the response. The difficulty in exploiting the WackoPicko file exposure was including the null-terminating character (`%00`) at the end of the string, which caused PHP to ignore anything added by the application after the `/etc/passwd` part. The results show that only 4 scanners successfully discovered this vulnerability.

The remote code execution vulnerability in WackoPicko is similar to the file exposure vulnerability. However, instead of injecting known files, the scanners injected known web site addresses. This was typically from a domain the scanner's developers owned, and thus when successfully exploited, the injected page appeared instead of the regular page. The same difficulty in a successful exploitation existed in the File Exposure vulnerability, so a scanner had to add `%00` after the injected web site. Only 3 scanners were able to successfully identify this vulnerability.

3.3.4 Crawling Capabilities

The number of URLs requested and accessed varies considerably among scanners, depending on the capability and strategies implemented in the crawler and attack components. Table 3.6 shows the number of times each scanner made a `POST` or `GET` request to a vulnerable URL when the scanners were run in INITIAL, CONFIG, and MANUAL mode. For instance, from Table 3.6 we can see that Hailstorm was able to access many of the vulnerable pages that required a valid username/password when run in INITIAL mode. It can also be seen that N-Stalker takes a shotgun-like approach to scanning; it has over 1,000 accesses for each vulnerable URL, while in contrast Grendel-Scan never had over 50 accesses to a vulnerable URL.

In the following, we discuss the main challenges that the crawler components of the web application scanners under test faced.

HTML

The results for the stored XSS attack reveal some interesting characteristics of the analysis performed by the various scanners. For instance, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, and w3af were unable to discover the stored XSS vulnerability in INITIAL configuration mode. Burp and N-Stalker failed because of defective HTML parsing. Neither of the scanners correctly interpreted the `<textarea>` tag as an input to the HTML form. This was evident because both scanners only sent the `name` parameter when attempting to leave a comment on the guestbook. When run in MANUAL mode, however, the scanners discovered the vulnerability, because the user provided values for all these fields. Grendel-Scan and Milescan missed the stored XSS vulnerability for the same reason: they did not attempt a `POST` request unless the user used the proxy to make the request.

Hailstorm did not try to inject any values to the guestbook when in INITIAL mode, and, instead, used `testval` as the `name` parameter and `Default text` as the `comment` parameter. One explanation for this could be that Hailstorm was run in the default "turbo" mode, which Cenzic claims catches 95% of vulnerabilities, and chose not to fuzz the form to improve speed.

Finally, w3af missed the stored XSS vulnerability due to leaving one parameter blank while attempting to inject the other parameter. It was unable to create a guestbook entry, because both parameters are required.

Uploading a Picture

Being able to upload a picture is critical to discover the Directory Traversal vulnerability, as a properly crafted `tag` parameter can overwrite any file the web server can access. It was very difficult for the scanners to successfully upload a file: no scanner was able to upload a picture in INITIAL and CONFIG modes, and only AppScan and Webinspect were able to upload a picture after being showed how to do it in MANUAL configuration, with AppScan and Webinspect uploading 324 and 166 pictures respectively. Interestingly, Hailstorm, N-Stalker and NTOSpider never successfully uploaded a picture, even in MANUAL configuration. This surprising result is due to poor proxies or poor in-application browsers. For instance, Hailstorm includes an embedded Mozilla browser for the user to browse the site when they want to do so manually, and after repeated attempts the embedded browser was never able to upload a file. The other scanners that failed, N-Stalker and NTOSpider, had faulty HTTP proxies that did not know how to properly forward the file uploaded, thus the request never completed successfully.

Client-side Code

The results of the WIVET tests are shown in Figure 3.4. Analyzing the WIVET results gives a very good idea of the JavaScript capabilities of each scanner. Of all the 54 WIVET tests, 24 required actually executing or understand JavaScript code; that is, the test could not be passed simply by using a regular expression to extract the links on the page. Webinspect was the only scanner able to complete all of the dynamic JavaScript challenges. Of the rest of the scanners, Acunetix and NTOSpider only missed one of the dynamic JavaScript tests. Even though Hailstorm missed 12 of the dynamic JavaScript tests, we believe that this is because of a bug in the JavaScript analysis engine and not a general limitation of the tool. In fact, Hailstorm was able to correctly

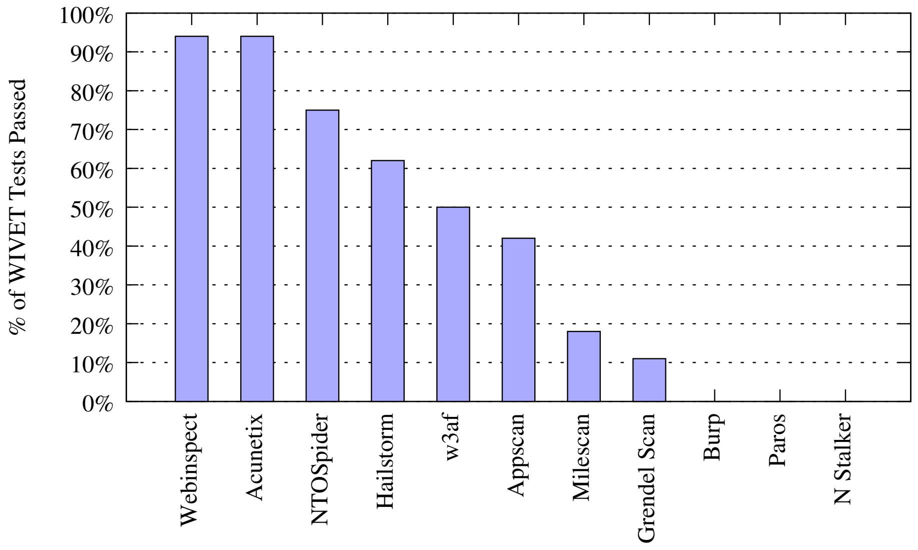

Figure 3.4: WIVET results.

handle JavaScript on the `onmouseup` and `onclick` parametrized functions. These tests were on parametrized `onmouseout` and `onmousedown` functions, but since Hailstorm was able to correctly handle the `onmouseup` and `onclick` parametrized functions, this can be considered a bug in Hailstorm's JavaScript parsing. From this, it can also be concluded that AppScan, Grendel-Scan, Milescan, and w3af perform no dynamic JavaScript parsing. Thus, Webinspect, Acunetix, NTOSpider, and Hailstorm can be claimed to have the best JavaScript parsing. The fact that N-Stalker found the reflected XSS vulnerability behind a JavaScript form in WackoPicko suggests that it can execute JavaScript, however it failed the WIVET benchmark so we cannot evaluate the extent of the parsing performed.

In looking at the WIVET results, there was one benchmark that no scanner was able to reach, which was behind a Flash application. The application had a link on a button's `onclick` event, however this link was dynamically created at run time. This failure shows that none of the current scanners processes Flash content with the same level of sophistication as JavaScript. This conclusion is supported by none of the scanners discovering the XSS vulnerability behind a Flash application in WackoPicko when in INITIAL or CONFIG mode.

Authentication

Table 3.7 shows the attempts that were made to create an account on the WackoPicko site. The Name column is the name of the scanner, "Successful" is the number of accounts successfully created, and "Error" is the number of account creation attempts that were unsuccessful. Note that Table 3.7 reports the results of the scanners when run in INITIAL mode only, because the results for the other configurations were almost identical.

Table 3.7 shows the capability of the scanners to handle user registration functionality. As can be seen from Table 3.7, only five of the scanners were able to successfully create an account. Of these, Hailstorm was the only one to leverage this ability to visit vulnerable URLs that required a login in its INITIAL run.

Creating an account is important in discovering the stored SQL injection that no scanner successfully detected. It is fairly telling that even though five scanners were able to create an account, none of them detected the vulnerability. It is entirely possible that none of the scanners actively searched for stored SQL injections, which is much harder to detect than stored XSS injections.

In addition to being critically important to the WackoPicko benchmark, being able to create an account is an important skill for a scanner to have when analyzing any web site, especially if that scanner wishes to be a point-and-click web application vulnerability scanner.

Multi-step Processes

In the WackoPicko web site there is a vulnerability that is triggered by going through a multi-step process. This vulnerability is the stored XSS on pictures, which requires an attacker to confirm a comment posting for the attack to be successful. Hailstorm and NTOSpider were the only scanners to successfully create a comment on the INITIAL run (creating 25 and 1 comment, respectively). This is important for two reasons: first, to be able to create a comment in the INITIAL run, the scanner had to create an account and log in with that account, which is consistent with Table 3.7. Also, all 25 of the comments successfully created by Hailstorm only contained the text `Default` `text`, which means that Hailstorm was not able to create a comment that exploited the vulnerability.

All scanners were able to create a comment when run in MANUAL configuration, since they were shown by the user how to carry out this task. However, only AppScan, Hailstorm, NTOSpider, and Webinspect (creating 6, 21, 7, and 2 comments respectively) were able to create a comment that was different than the one provided by the user. Of these scanners only Webinspect was able to create a comment that exploited the vulnerability, `<iFrAmE sRc=hTtP://xSrFtEsT .sPi/> </iFrAmE>`, however Webinspect failed to report this vulnerability. One plausible explanation for not detecting would be the scanners' XSS strategy discussed in Section 3.3.3. While testing the `text` parameter for a vulnerability, most of the scanners realized that it was properly escaped on the preview page, and thus stopped trying to inject XSS attacks. This would explain the directory traversal attack comment that AppScan successfully created and why Hailstorm did not attempt any injection. This is an example where the performance optimization of the vulnerability analysis can lead to false negatives.

Infinite Web Sites

One of the scanners attempted to visit all of the pages of the infinite calendar. When running Grendel-Scan, the calendar portion of WackoPicko had to be removed because the scanner ran out of memory attempting to access every page. Acunetix, Burp, N-

Stalker and w3af had the largest accesses (474, 691, 1780 and 3094 respectively), due to their attempts to exploit the calendar page. The other scanners used less accesses (between 27 and 243) because they were able to determine that no error was present.

3.4 Lessons Learned

We found that the crawling of modern web applications can be a serious challenge for today's web vulnerability scanners. A first class of problems we encountered consisted of implementation errors and the lack of support for commonly-used technologies. For example, handling of multimedia data (image uploads) exposed bugs in certain proxy-based scanners, which prevented the tools from delivering attacks to the application under test. Incomplete or incorrect HTML parsers caused scanners to ignore input vectors that would have exposed vulnerabilities. The lack of support for JavaScript (and Flash) prevented tools from reaching vulnerable pages altogether. *Support for well-known, pervasive technology should be improved.*

The second class of problems that hindered crawling is related to the design of modern web applications. In particular, applications with complex forms and aggressive checking of input values can effectively block a scanner, preventing it from crawling the pages "deep" in the web site structure. Handling this problem could be done, for example, by using heuristics to identify acceptable inputs or by reverse engineering the input filters. Furthermore, the behavior of an application can be wildly different depending on its internal "state," i.e., the values of internal variables that are not explicitly exposed to the scanner. The classic example of application state is whether the current user is logged in or not. A scanner that does not correctly model and track the state of an application (e.g., it does not realize that it has been automatically logged out) will fail to crawl all relevant parts of the application. *More sophisticated algorithms are needed to perform "deep" crawling and track the state of the application under test.*

Current scanners fail to detect (or even check for) application-specific (or "logic") vulnerabilities. Unfortunately, as applications become more complex, this type of vulnerabilities will also become more prevalent. *More research is warranted to automate the detection of application logic vulnerabilities.*

In conclusion, far from being point-and-click tools to be used by anybody, web application black-box security scanners require a sophisticated understanding of the application under test and of the limitations of the tool, in order to be effective.

3.5 Conclusions

This chapter presented the evaluation of eleven black-box web vulnerability scanners. The results of the evaluation clearly show that the ability to crawl a web application and reach “deep” into the application’s resources is as important as the ability to detect the vulnerabilities themselves.

It is also clear that although techniques to detect certain kinds of vulnerabilities are well-established and seem to work reliably, there are whole classes of vulnerabilities that are not well-understood and cannot be detected by the state-of-the-art scanners. We found that eight out of sixteen vulnerabilities were not detected by *any* of the scanners.

We have also found areas that require further research so that web application vulnerability scanners can improve their detection of vulnerabilities. Deep crawling is vital to discover all vulnerabilities in an application. Improved reverse engineering is necessary to keep track of the state of the application, which can enable automated detection of complex vulnerabilities.

Finally, we found that there is no strong correlation between cost of the scanner and functionality provided as some of the free or very cost-effective scanners performed as well as scanners that cost thousands of dollars.

Scanner	Reflected XSS			Stored XSS			Reflected SQL Injection		
	INITIAL	CONFIG	MANUAL						
Acunetix	496	638	498	613	779	724	544	709	546
AppScan	581	575	817	381	352	492	274	933	628
Burp	256	256	207	192	192	262	68	222	221
Grendel-Scan	0	0	44	1	1	3	14	34	44
Hailstorm	232	229	233	10	205	209	45	224	231
Milescan	104	0	208	50	0	170	75	272	1,237
N-Stalker	1,738	1,162	2,689	2,484	2,100	3,475	2,764	1,022	2,110
NTOSpider	856	679	692	252	370	370	184	5	5
Paros	68	68	58	126	126	110	151	299	97
w3af	157	157	563	259	257	464	1,377	1,411	2,634
Webinspect	108	108	105	631	631	630	297	403	346
Scanner	Command-line Injection			File Inclusion / File Exposure / Weak password			XSS Reflected - JavaScript		
	INITIAL	CONFIG	MANUAL						
Acunetix	495	637	497	198	244	200	670	860	671
AppScan	189	191	288	267	258	430	0	0	442
Burp	68	68	200	125	316	320	0	0	178
Grendel-Scan	1	1	3	2	2	5	0	0	2
Hailstorm	180	160	162	8	204	216	153	147	148
Milescan	0	0	131	80	0	246	0	0	163
N-Stalker	2,005	1,894	1,987	1,437	2,063	1,824	1,409	1,292	1,335
NTOSpider	105	9	9	243	614	614	11	13	13
Paros	28	28	72	146	146	185	0	0	56
w3af	140	142	253	263	262	470	0	0	34
Webinspect	164	164	164	239	237	234	909	909	0
Scanner	Parameter Manipulation			Directory Traversal			Logic Flaw		
	INITIAL	CONFIG	MANUAL						
Acunetix	2	0	2	35	1,149	37	0	0	5
AppScan	221	210	222	80	70	941	0	0	329
Burp	192	194	124	68	68	394	0	0	314
Grendel-Scan	3	3	6	1	1	3	0	0	6
Hailstorm	3	143	146	336	329	344	131	132	5
Milescan	105	0	103	8	0	163	0	0	1
N-Stalker	1,291	1,270	1,302	22	2,079	4,704	0	0	3
NTOSpider	107	115	115	11	572	572	0	11	11
Paros	72	72	72	14	14	0	0	0	114
w3af	128	128	124	31	30	783	0	0	235
Webinspect	102	102	102	29	29	690	0	8	3
Scanner	Forceful Browsing			XSS Reflected behind flash					
	INITIAL	CONFIG	MANUAL						
Acunetix	0	0	206	1	34	458			
AppScan	0	0	71	0	0	243			
Burp	0	0	151	0	0	125			
Grendel-Scan	0	0	1	0	0	3			
Hailstorm	102	102	105	0	0	143			
Milescan	0	0	60	0	0	68			
N-Stalker	0	0	2	0	0	1,315			
NTOSpider	0	0	0	0	11	11			
Paros	0	0	70	0	0	60			
w3af	0	0	270	0	0	119			
Webinspect	0	118	82	0	0	97			

Table 3.6: Number of accesses to vulnerable web pages in INITIAL, CONFIG, and MANUAL modes.

Name	Successful	Error
Acunetix	0	431
AppScan	1	297
Burp	0	0
Grendel-Scan	0	0
Hailstorm	107	276
Milescan	0	0
N-Stalker	74	1389
NTOSpider	74	330
Paros	0	176
w3af	0	538
Webinspect	127	267

Table 3.7: Account creation.

Chapter 4

A State-Aware Black-Box Web Vulnerability Scanner

We identified the biggest problem common to black-box web application vulnerability scanners in the previous chapter: The scanners treat the web application as if it was a web site, and ignore the fact that it is an *application* with state. In this chapter, we describe methods and approaches to automatically infer the state of a web application in a black-box manner, and we apply this to a black-box web application vulnerability scanner.

Web applications are the most popular way of delivering services via the Internet. A modern web application is composed of a back-end, server-side part (often written in Java or in interpreted languages such as PHP, Ruby, or Python) running on the provider's server, and a client part running in the user's web browser (implemented in

JavaScript and using HTML/CSS for presentation). The two parts often communicate via HTTP over the Internet using Asynchronous JavaScript and XML (AJAX) [56].

The complexity of modern web applications, along with the many different technologies used in various abstraction layers, are the root cause of vulnerabilities in web applications. In fact, the number of reported web application vulnerabilities is growing sharply [52, 132].

The occurrence of vulnerabilities could be reduced by better education of web developers, or by the use of security-aware web application development frameworks [34, 119], which enforce separation between structure and content of input and output data. In both cases, more effort and investment in training is required, and, therefore, cost and time-to-market constraints will keep pushing for the current fast-but-insecure development model.

A complementary approach for fighting security vulnerabilities is to discover and patch bugs before malicious attackers find and exploit them. One way is to use a white-box approach, employing static analysis of the source code [11, 48, 71, 81]. There are several drawbacks to a white-box approach. First, the potential applications that can be analyzed is reduced to only those applications that use the target programming language. In addition, there is the problem of substantial false positives. Finally, the source code of the application itself may be unavailable.

The other approach to discovering security vulnerabilities in web applications is by observing the application's output in response to a specific input. This method of analysis is called *black-box* testing, as the application is seen as a sealed machine with unobservable internals. Black-box approaches are able to perform large-scale analysis across a wide range of applications. While black-box approaches usually have fewer false positives than white-box approaches, black-box approaches suffer from a discoverability problem: They need to reach a page to find vulnerabilities on that page.

Classical black-box web vulnerability scanners crawl a web application to enumerate all reachable pages and then fuzz the input data (URL parameters, form values, cookies) to trigger vulnerabilities. However, this approach ignores a key aspect of modern web applications: Any request can change the state of the web application.

In the most general case, the state of the web application is any data (database, filesystem, time) that the web application uses to determine its output. Consider a forum that authenticates users, an e-commerce application where users add items to a cart, or a blog where visitors and administrators can leave comments. In all of these modern applications, the way a user interacts with the application determines the application's state.

Because a black-box web vulnerability scanner will never detect a vulnerability on a page that it does not see, scanners that ignore a web application's state will only explore and test a (likely small) fraction of the web application.

In this chapter, we propose to improve the effectiveness of black-box web vulnerability scanners by increasing their capability to understand the web application's internal state. Our tool constructs a partial model of the web application's state machine in a fully-automated fashion. It then uses this model to fuzz the application in a state-aware manner, traversing more of the web application and thus discovering more vulnerabilities.

The main contributions of this chapter are the following:

- A black-box technique to automatically learn a model of a web application's state.
- A novel vulnerability analysis technique that leverages the web application's state model to drive fuzzing.
- An evaluation of our technique, showing that both code coverage and effectiveness of vulnerability analysis are improved.

4.1 Motivation

Crawling modern web applications means dealing with the web application's changing state. Previous work in detecting workflow violations [12, 36, 48, 88] focused on navigation, where a malicious user can access a page that is intended only for admin-

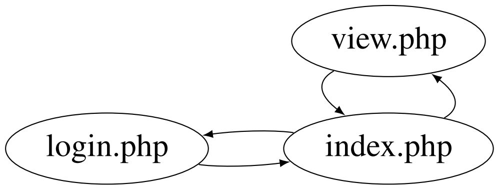

Figure 4.1: Navigation graph of a simple web application.

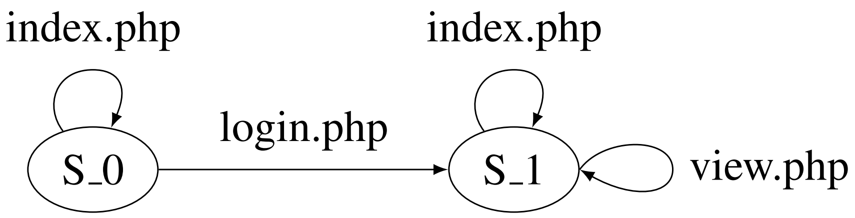

Figure 4.2: State machine of a simple web application.

istrators. This unauthorized access is a violation of the developer's intended work-flow of the application.

We wish to distinguish a navigation-based view of the web application, which is simply derived from crawling the web application, from the web application's internal state machine. To illustrate this important difference, we will use a small example.

Consider a simple web application with three pages, `index.php`, `login.php`, and `view.php`. The `view.php` page is only accessible after the `login.php` page is accessed. There is no logout functionality. A client accessing this web application might make a series of requests like the following:

⟨`index.php`, `login.php`, `index.php`, `view.php`,
 `index.php`, `view.php`⟩

Analyzing this series of requests from a navigation perspective creates a navigation graph, shown in Figure 4.1. This graph shows which page is accessible from every other page, based on the navigation trace. However, the navigation graph does not represent the information that `view.php` is only accessible after accessing `login.php`, or that `index.php` has changed after requesting `login.php` (it includes the link to `view.php`).

What we are interested in is not how to navigate the web application, but how the requests we make influence the web application's internal state machine. The simple web application described previously has the internal state machine shown in Figure 4.2. The web application starts with the internal state `S_0`. Arrows from a state show how a request affects the web application's internal state machine. In this example, in the initial state, `index.php` does not change the state of the application, however, `login.php` causes the state to transition from `S_0` to `S_1`. In the new state `S_1`, both `index.php` and `view.php` do not change the state of the web application.

The state machine in Figure 4.2 contains important information about the web application. First, it shows that `login.php` permanently changes the web application's state, and there is no way to recover from this change. Second, it shows that the `index.php` page is seen in two different states.

Now the question becomes: "How does knowledge of the web application's state machine (or lack thereof) affect a black-box web vulnerability scanner?" The scanner's goal is to find vulnerabilities in the application, and to do so it must fuzz as many execution paths of the server-side code as possible[1]. Consider the simple application described in Figure 4.2. In order to fuzz as many code paths as possible, a black-box web vulnerability scanner must fuzz the `index.php` page in both states `S_0` and `S_1`, since the code execution of `index.php` can follow different code paths depending on the current state (more precisely, in state `S_1`, `index.php` includes a link to `view.php`, which is not present in `S_0`).

A black-box web vulnerability scanner can also use the web application's state machine to handle requests that change state. For example, when fuzzing the `login.php` page of the sample application, a fuzzer will try to make several requests to the page, fuzzing different parameters. However, if the first request to `login.php` changes the state of the application, all further requests to `login.php` will no longer execute along the same code path as the first one. Thus, a scanner must have knowledge of the web application's state machine to test if the state was changed, and if it was, what requests to make to return the application to the previous state before continuing the fuzzing process.

We have shown how a web application's state machine can be leveraged to improve a black-box web vulnerability scanner. Our goal is to infer, in a black-box manner, as much of the web application's state machine as possible. Using only the sequence of

[1]Hereinafter, we assume that the scanner relies on fuzzer-based techniques. However, any other automated vulnerability analysis technique would benefit from our state-aware approach.

requests, along with the responses to those requests, we build a model of as much of the web application's state machine as possible. In the following section, we describe, at a high level, how we infer the web application's state machine. Then, in Section 4.3, we provide the details of our technique.

4.2 State-Aware Crawling

In this section, we describe our state-aware crawling approach. In Section 4.2.1, we describe web applications and define terms that we will use in the rest of the chapter. Then, in Section 4.2.2, we describe the various facets of the state-aware crawling algorithm at a high level.

4.2.1 Web Applications

Before we can describe our approach to inferring a web application's state, we must first define the elements that come into play in our web application model.

A web application consists of a server component, which accepts HTTP requests. This server component can be written in any language, and could use many different means of storage (database, filesystem, memcache). After processing a request, the server sends back a response. This response encapsulates some content, typically

HTML. The HTML content contains links and forms which describe how to make further requests.

Now that we have described a web application at a high level, we need to define specific terms related to web applications that we use in the rest of this chapter.

- Request—The HTTP request made to the web application. Includes anything (typically in the form of HTTP headers) that is sent by the user to the web application: the HTTP Method, URL, Parameters (`GET` and `POST`), Cookies, and User-Agent.
- Response—The response sent by the server to the user. Includes the HTTP Response Code and the content (typically HTML).
- Page—The HTML page that is contained in the response from a web application.
- Link—Element of an HTML page that tells the browser how to create a subsequent request. This can be either an anchor or a form. An anchor always generates a `GET` request, but a form can generate either a `POST` or `GET` request, depending on the parameters of the form.
- State—Anything that influences the web application's server-side code execution.

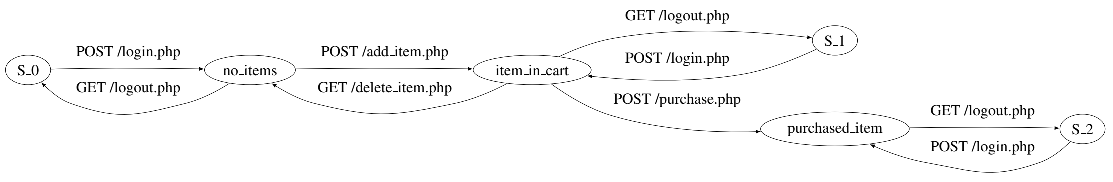

Figure 4.3: The state machine of a simple e-commerce application.

Web Application Model

We use a *symbolic Mealy machine* [15] to model the web application as a black-box. A Mealy machine is an automaton where the input to the automaton, along with the current state, determines the output (i.e., the page produced by the response) and the next state. A Mealy machine operates on a finite alphabet of input and output symbols, while a symbolic Mealy machine uses an infinite alphabet of input and output symbols.

This model of a web application works well because the input to a web application, along with the current state of the web application, determines the output and the next state. Consider a simple e-commerce web application with the state machine show in Figure 4.3. In this state graph, all requests except for the ones leaving a state bring the application back to the same state. Therefore, this state graph does not show all the request that can be made to the application, only the subset of requests that change the state.

For instance, in the initial state `S_0`, there is only one request that will change the state of the application, namely `POST /login.php`. This change logs the user into the web application. From the state `no_items`, there are two requests that can change the state, `GET /logout.php` which returns the user to the initial state `S_0` and `POST /add_item.php` to add an item to the user's shopping cart.

Note that the graph shown in Figure 4.3 is not a strongly connected graph—that is, every state cannot be reached by every other state. In this example, purchasing an item is a permanent action, it irrecoverably changes the state (there is no link from `purchased_item` to `item_in_cart`). Another interesting aspect is that one request, `GET /logout.php`, leads to three different states. This is because once the web application's state has changed, logging out, and then back in, does not change the state of the cart.

4.2.2 Inferring the State Machine

Inferring a web application's state machine requires the ability to detect when the state of the web application has changed. Therefore, we start with a description of the state-change detection algorithm, then explain the other components that are required to infer the state machine.

The key insight of our state-change algorithm is the following: We detect that the state of the web application has changed when we make an identical request and get a different response. This is the only externally visible effect of a state-change: Providing the same input causes a different output.

Using this insight, our state-change detection algorithm works, at a high level, as follows: (1) Crawl the web application sequentially, making requests based on a link in the previous response. (2) Assume that the state stays the same, because there is no evidence to the contrary. (3) If we make a request identical to a previous request and get a different response, then we assume that some request since the last identical request changed the state of the web application.

The intuition here is that a Mealy machine will, when given the same input in the same state, produce the same output. Therefore, if we send the same request and get a different output, the state must have changed. By detecting the web application's state changes only using inputs and outputs, we are agnostic with respect to both *what* constitutes the state information and *where* the state information is located. In this way, we are more generic than approaches that only consider the database to hold the state of the application, when in fact, the local file system or even memory could hold part of the web application's state.

The state-change detection algorithm allows us to infer when the web application's state has changed, yet four other techniques are necessary to infer a state machine: the clustering of similar pages, the identification of state-changing requests, the collapsing of similar states, and navigating.

Clustering similar pages. We want to group together pages that are similar, for two reasons: To handle infinite sections of web applications that are generated from the same code (e.g., the pages of a calendar) and to detect when a response has changed.

Before we can cluster pages, we model them using the links (anchors and forms) present on the page. The intuition here is that the links describe *how* the user can interact with the web application. Therefore, changes to what a user can do (new or missing links) indicate when the state of the web application has changed. Also, infinite sections of a web application will share the same link structure and will cluster together.

With our page model, we cluster pages together based on their link structure. Pages that are in different clusters are considered different. The details of this approach are described in Section 4.3.1.

Determining the state-changing request. The state-change detection algorithm only says that the state has changed, however we need to determine *which* request actually changed the state. When we detect a state change, we have a temporal list of requests with identical requests at the start and end. One of the requests in this list changed the state. We use a heuristic to determine which request changed the state. This heuristic favors newer requests over older requests, `POST` requests over `GET` requests, and requests that have previously changed the state over those that have never changed the state. The details are described in Section 4.3.2.

Collapsing similar states. The state-change detection algorithm detects only when the state has changed, however, we need to understand if we returned to a previous state. This is necessary because if we detect a state change, we want to know if this is a state we have previously seen or a brand new state. We reduce this problem to a graph coloring problem, where the nodes are the states and an edge between two nodes means that the states cannot be the same. We add edges to this graph by using the requests and responses, along with rules to determine when two states cannot be the same. After the graph is colored, states that are the same color are collapsed into the same state. Details of this state-merging technique are provided in Section 4.3.3.

Navigating. We have two strategies for crawling the web application.

First, we always try to pick a link in the last response. The rational behind choosing a link in the last response is that we emulate a user browsing the web application. In this way, we are able to handle multi-step processes, such as previewing a comment before it is committed.

Second, for each state, we make requests that are the least likely to change the state of the web application. The intuition here is that we want to first see as much of a state as possible, without accidentally changing the state, in case the state change is permanent. Full details of how we crawl the web application are provided in Section 4.3.4

4.3 Technical Details

Inferring a web application's state machine requires concretely defining aspects such as page clustering or navigation. However, we wish to stress that this is one implementation of the state machine inference algorithm and it may not be optimal.

4.3.1 Clustering Similar Pages

Our reason for grouping similar pages together is twofold: Prevent infinite scanning of the website by grouping the "infinite" areas together and detect when the state has changed by comparing page responses in an efficient manner.

Page Model

The output of a web application is usually an HTML document (it can actually be any arbitrary content, but we only consider HTML content and HTTP redirects). An HTML page is composed of navigational information (anchors and forms) and user-readable content. For our state-change detection algorithm, we are not interested in changes to the content, but rather to changes in the navigation structure. We focus on navigation changes because the links on a page define how a user can interact with the application, thus, when the links change, the web application's state has changed.

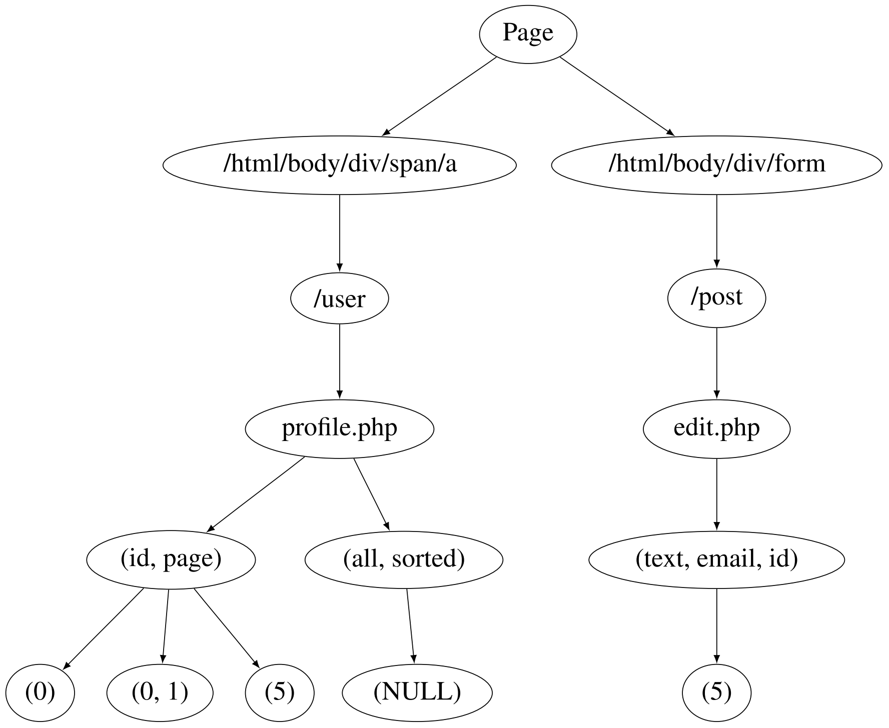

Figure 4.4: Representation of a page's link vectors stored in a prefix tree. There are five links present on this tree, as evidenced by the number of leaf nodes.

Therefore, we model a page by composing all the anchors and forms. First, every anchor and form is transformed into a vector constructed as follows:

$$\langle dompath,\ action,\ params,\ values \rangle$$

where:

- $dompath$ is the DOM (*Document Object Model*) path of the HTML link (anchor or form);
- $action$ is a list where each element is from the `href` (for anchors) or `action` (for forms) attribute split by '/';

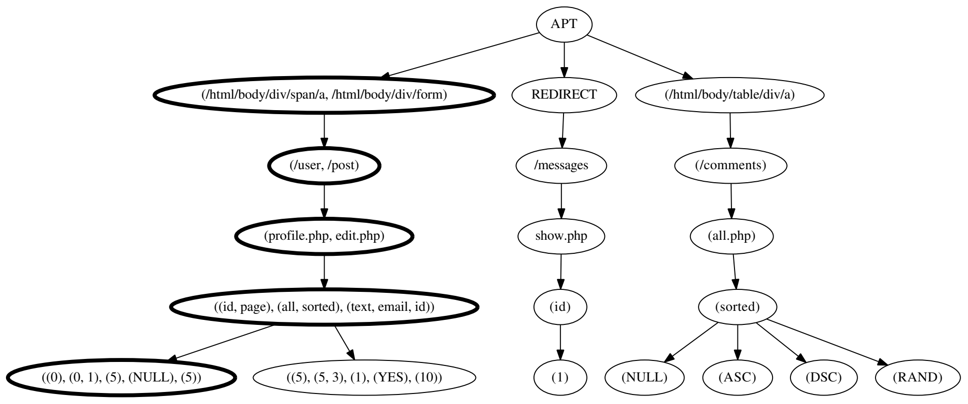
APT
(/html/body/div/span/a, /html/body/div/form)
REDIRECT
(/html/body/table/div/a)
(/user, /post)
/messages
(/comments)
(profile.php, edit.php)
show.php
(all.php)
((id, page), (all, sorted), (text, email, id))
(id)
(sorted)
((0), (0, 1), (5), (NULL), (5))
((5), (5, 3), (1), (YES), (10))
(1)
(NULL)
(ASC)
(DSC)
(RAND)

Figure 4.5: Abstract Page Tree. Every page's link vector is stored in this prefix tree. There are seven pages in this tree. The page link vector from Figure 4.4 is highlighted in bold.

- $params$ is the (potentially empty) set of parameter names of the form or anchor;
- $values$ is the set of values assigned to the parameters listed in $params$.

For instance, an anchor tag with the `href` attribute of `/user/profile.php?id=0&page` might have the following link vector:

$$\langle\text{/html/body/div/span/a},\ \text{/user},\ \text{profile.php},\ \text{(id, page)},\ \text{(0)}\rangle$$

All link vectors of a page are then stored in a prefix tree. This prefix tree is the model of the page. A prefix tree for a simple page with five links is shown in Figure 4.4. The link vector previously described is highlighted in bold in Figure 4.4.

HTTP redirects are handled as a special case, where the only element is a special *redirect* element having the target URL as the value of the *location* attribute.

Page Clustering

To cluster pages, we use a simple but efficient algorithm. As described in the previous section, the model of a page is a prefix tree representing all the links contained in the page.

These prefix trees are translated into vectors, where every element of this vector is the set of all nodes of a given level of the prefix tree, starting from the root. At this point, all pages are represented by a *page link vector.* For example, Figure 4.4 has the following page link vector:

$$\begin{aligned}\langle(&\text{/html/body/div/span/a},\ \text{/html/body/div/form}),\\ &(\text{/user},\ \text{/post}),\\ &(\text{profile.php},\ \text{edit.php}),\\ &((\text{id},\ \text{page}),\ (\text{all},\ \text{sorted}),\ (\text{text},\ \text{email},\ \text{id})),\\ &((0),\ (0,\ 1),\ (5),\ (NULL),\ (5))\rangle\end{aligned}$$

The page link vectors for all pages are then stored in another prefix tree, called the *Abstract Page Tree* (APT). In this way, pages are mapped to a leaf of the tree. Pages which are mapped to the same leaf have identical page link vectors and are considered to be the same page. Figure 4.5 shows an APT with seven pages. The page from Figure 4.4 is bold in Figure 4.5.

However, we want to cluster together pages whose page link vectors do not match exactly, but are similar (e.g., shopping cart pages with a different number of elements in the cart). A measure of the similarity between two pages is how many elements from the beginning of their link vectors are the same between the two pages. From the APT perspective, the higher the number of ancestors two pages (leaves) share, the closer they are.

Therefore, we create clusters of similar pages by selecting a node in the APT and merging into one cluster, called an *Abstract Page*, all the leaves in the corresponding subtree. The criteria for deciding whether to cluster a subtree of depth n from the root is the following:

- The number of leaves is greater than the median number of leaves of all its siblings (including itself); in this way, we cluster only subtrees which have a larger-than-usual number of leaves.

- There are at least $f(n)$ leaves in the subtree, where $f(n)$ is inversely related to n. The intuition is that the fewer ancestors a subtree has in common (the higher on the prefix tree it is), the more pages it must have to cluster them together. We have found that the function $f(n) = 8(1 + \frac{1}{n+1})$ works well by experimental analysis on a large corpus of web pages.

- The pages share the same $dompath$ and the first element of the $action$ list of the page link vector; in this way, all the pages that are clustered together share the same link structure with potentially different parameters and values.

4.3.2 Determine the State-Changing Request

When a state change is detected, we must determine which request actually changed the web application's state. Recall that we detect a state change when we make a request that is identical to a previous request, yet has different output. At this point, we have a list of all the requests made between the latest request R and the request R' closest in time to R such that R is identical to R'. We use a heuristic to determine which request in this list changed the web application's state, choosing the request i between R' and R which maximizes the function:

$$\text{score}(n_{i,transition}, n_{i,seen}, distance_i)$$

where:

- $n_{i,transition}$ is the number of times the request caused a state transition;
- $n_{i,seen}$ is the number of times the request has been made;
- $distance_i$ is how many requests have been made between request R and request i.

The function $score$ is defined as:

$$score(n_{i,transition}, n_{i,seen}, distance_i) =$$
$$1 - (1 - \tfrac{n_{i,transition}+1}{n_{i,seen}+1})^2 + \tfrac{\text{BOOST}_i}{distance_i+1}$$

BOOST_i is .2 for `POST` requests and .1 for `GET` requests.

We construct the $score$ function to capture two properties of web applications:

1. A POST request is more likely to change the state than a GET request. This is suggested by the HTTP specification, and $score$ captures this intuition with BOOST_i.

2. Resistant to errors. Because we cannot prove that the selected request changed the state, we need to be resistant to errors. That is why $score$ contains the ratio of $n_{i,transition}$ to $n_{i,seen}$. In this way, if we accidentally choose the wrong state-changing request once, but then, later, make that request many times without changing the state, we are less likely to choose it as a state-changing request.

4.3.3 Collapsing Similar States

Running the state detection algorithm on a series of requests and responses will tell us when the state has changed. At this point, we consider each state unique. This initial state assignment, though, is not optimal, because even if we encounter a state that we have seen in the past, we are marking it as new. For example, in the case of a sequence of login and logout actions, we are actually flipping between two states, instead of entering a new state at every login/logout. Therefore, we need to minimize the number of different states and collapse states that are actually the same.

The problem of state allocation can be seen as a graph-coloring problem on a non-planar graph [75]. Let each state be a node in the graph G. Let two nodes a and b be connected by an edge (meaning that the states cannot be the same) if either of the following conditions holds:

1. If a request R was made when the web application was in states a and b and results in pages in different clusters. The intuition is that two states cannot be the same if we make an identical request in each state yet receive a different response.

2. The two states a and b have no pages in common. The idea is to err on the conservative side, thus we require that two states share a page before collapsing the states into one.

After adding the edges to the graph by following the previous rules, G is colored. States assigned the same color are considered the same state.

To color the nodes of G, we employ a custom greedy algorithm. Every node has a unique identifier, which is the incremental number of the state as we see it in the request-response list. The nodes are ordered by identifier, and we assign the color to each node in a sequential way, using the highest color available (i.e., not used by its neighbors), or a new color if none is available.

This way of coloring the nodes works very well for state allocation because it takes into account the temporal locality of states: In particular, we attempt to assign the highest available color because it is more likely for a state to be the same as a recently seen state rather than one seen at the beginning of crawling.

There is one final rule that we need to add after the graph is colored. This rules captures an observation about transitioning between states: If a request, R, transitions the web application from state a_1 to state b, yet, later when the web application is in state a_2, R transitions the web application to state c, then a_1 and a_2 cannot be the same state. Therefore, we add an edge from a_1 to a_2 and redo the graph coloring.

We continue enforcing this rule until no additional edges are added. The algorithm is guaranteed to converge because only new edges are added at every step, and no edges are ever removed.

At the end of the iteration, the graph coloring output will determine the final state allocation—all nodes with the same color represent the same state (even if seen at different stages during the web application crawling process).

4.3.4 Navigating

Typical black-box web vulnerability scanners make concurrent HTTP requests to a web application to increase performance. However, as we have shown, an HTTP request can influence the web application's state, and, in this case, all other requests would occur in the new state. Also, some actions require a multi-step, sequential process, such as adding items to a shopping cart before purchasing them. Finally, a user of the web application does not browse a web application in this parallel fashion, thus, developers assume that the users will browse sequentially.

Our scanner navigates a web application by mimicking a user browsing the web application sequentially. Browsing sequentially not only allows us to follow the developer's intended path through the web application, but it enables us to detect *which* requests changed the web application's state.

Thus, a state-aware crawler must navigate the application sequentially. No concurrent requests are made, and only anchors and forms present in the last visited page are used to determine the next request. In the case of a page with no outgoing links we go back to the initial page.

Whenever the latest page does not contain unvisited links, the crawler will choose a path from the current page towards another page already seen that contains links that have not yet been visited. If there is no path from the current page to anywhere, we go back to the initial page. The criteria for choosing this path is based on the following intuitions:

- We want to explore as much of the current state as possible before changing the state, therefore we select links that are less likely to cause a state transition.

```
def fuzz_state_changing( fuzz_request ):
  make_request( fuzz_request )
  if state_has_changed():
    if state_is_reversible():
      make_requests_to_revert_state()
      if not back_in_previous_state():
        reset_and_put_in_previous_state()
    else:
      reset_and_put_in_previous_state()
```

Listing 4.1: Psuedocode for fuzzing state-changing request.

- When going from the current page to a page with an unvisited link, we will repeat requests. Therefore, we should choose a path that contains links that we have visited infrequently. This give us more information about the current state.

The exact algorithm we employ is Dijkstra Shortest Path Algorithm [43] with custom edge length. This edge length increases with the number of times we have previously visited that link. Finally, the edge length increases with how likely the link is to cause a state change.

4.4 State-Aware Fuzzing

After we crawl the web application, our system has inferred, as much as possible, the web application's state machine. We use the state machine information, along with the list of request–responses made by the crawler, to drive a state-aware fuzzing of the web application, looking for security vulnerabilities.

To fuzz the application in a state-aware manner, we need the ability to reset the web application to the initial state (the state when we started crawling). We do not use this ability when crawling, only when fuzzing. It is necessary to reset the application when we are fuzzing an irreversible state-changing request. Using the reset functionality, we are able to recover from these irreversible state changes.

Adding the ability to reset the web application does not break the black-box model of the web application. Resetting requires no knowledge of the web application, and can be easily performed by running the web application in a virtual machine.

Our state-aware fuzzing starts by resetting the web application to the initial state. Then we go through the requests that the crawler made, starting with the initial request. If the request does not change the state, then we fuzz the request as a typical black-box scanner. However, if the request is state-changing, we follow the algorithm shown in Listing 4.1. The algorithm is simple: We make the request, and if the state has changed, traverse the inferred state machine to find a series of requests to transition the web application to the previous state. If this does not exist, or does not work, then we reset the web application to the initial state, and make all the previous requests that the crawler made. This ensures that the web application is in the proper state before continuing to fuzz.

Our state-aware fuzzing approach can use *any* fuzzing component. In our implementation, we used the fuzzing plugins of an open-source scanner, w3af [117]. The

Application	Description	Version	Lines of Code
Gallery	Photo hosting.	3.0.2	26,622
PhpBB v2	Discussion forum.	2.0.4	16,034
PhpBB v3	Discussion forum.	3.0.10	110,186
SCARF	Stanford conference and research forum.	2007-02-27	798
Vanilla Forums	Discussion forum.	2.0.17.10	43,880
WackoPicko v2	Intentionally vulnerable web application.	2.0	900
WordPress v2	Blogging platform.	2.0	17,995
WordPress v3	Blogging platform.	3.2.1	71,698

Table 4.1: Applications that we ran the crawlers against to measure vulnerabilities discovered and code coverage.

fuzzing plugins take an HTTP request and generate variations on that request looking for different vulnerabilities. Our state-aware fuzzing makes those requests while checking that the state does not unintentionally change.

4.5 Evaluation

As shown in Chapter 3, fairly evaluating black-box web vulnerability scanners is difficult. The most important, at least to end users, metric for comparing black-box web vulnerability scanners is true vulnerabilities discovered. Comparing two scanners that discover different vulnerabilities is nearly impossible.

There are two other metrics that we use to evaluate black-box web vulnerability scanners:

- False Positives. The number of spurious vulnerabilities that a black-box web vulnerability scanner reports. This measures the accuracy of the scanner. False positives are a serious problem for the end user of the scanner—if the false positives are high, the user must manually inspect each vulnerability reported to determine the validity. This requires a security-conscious user to evaluate the reports. Moreover, false positives erode the user's trust in the tool and make the user less likely to use it in the future.

- Code Coverage. The percentage of the web application's code that the black-box web vulnerability scanner executes while it crawls and fuzzes the application. This measures how effective the scanner is in exercising the functionality of the web application. Moreover, code coverage is an excellent metric for another reason: A black-box web vulnerability scanner, by nature, cannot find a vulnerability along a code path that it does not execute. Therefore, greater code coverage means that a scanner has the potential to discover more vulnerabilities. Note that this is orthogonal to fuzzing capability: A fuzzer—no matter how effective—will never be able to discover a vulnerability on a code path that it does not execute.

We use both the metrics previously described in our evaluation. However, our main focus is on code coverage. This is because a scanner with greater code coverage will be able to discover more vulnerabilities in the web application.

However, code coverage is not a perfect metric. Evaluating raw code coverage percentage numbers can be misleading. Ten percent code coverage of an application could be horrible or excellent depending on how much functionality the application exposes. Some code may be intended only for installation, may be only for administrators, or is simply dead code and cannot be executed. Therefore, comparing code coverage normalized to a baseline is more informative, and we use this in our evaluation.

4.5.1 Experiments

We evaluated our approach by running our state-aware-scanner along with three other vulnerability scanners against eight web applications. These web applications range in size, complexity, and functionality. In the rest of this section, we describe the web applications, the black-box web vulnerability scanners, and the methodology we used to validate our approach.

Web Applications

Table 4.1 provides an overview of the web applications used with a short description, a version number, and lines of executable PHP code for each application. Because our approach assumes that the web application's state changes only via requests from the user, we made slight code modifications to three web applications to reduce the influence of external, non-user driven, forces, such as time.

This section describes the web applications along with the functionality against which we ran the black-box web vulnerability scanner.

Gallery is an open-source photo hosting application. The administrator can upload photos and organize them into albums. Guests can then view and comment on the uploaded photos. Gallery has AJAX functionality but gracefully degrades (is fully functional) without JavaScript. No modifications were made to the application.

PhpBB v2 is an open-source forum software. It allows registered users to perform many actions such as create new threads, comment on threads, and message other users. Version 2 is notorious for the amount of security vulnerabilities it contains [14], and we included it for this reason. We modified it to remove the "recently online" section on pages, because this section is based on time.

PhpBB v3 is the latest version of the popular open-source forum software. It is a complete rewrite from Version 2, but retains much of the same functionality. Similar to PhpBB v2, we removed the "recently online" section, because it is time-based.

SCARF, the Stanford Conference And Research Forum, is an open-source conference management system. The administrator can upload papers, and registered users can comment on the uploaded papers. We included this application because it was used by previous research [12, 36, 88, 89]. No modifications were made to this application.

Vanilla Forums is an open-source forum software similar in functionality to PhpBB. Registered users can create new threads, comment on threads, bookmark interesting threads, and send a message to another user. Vanilla Forums is unique in our test set in that it uses the path to pass parameters in a URL, whereas all other applications pass parameters using the query part of the URL. For instance, a specific user's profile is `GET /profile/scanner1`, while a discussion thread is located at `GET /discussion/1/how-to-scan`. Vanilla Forums also makes extensive use of AJAX, and it does not gracefully degrade. For instance, with JavaScript disabled, posting a comment returns a JSON object that contains the success or failure of the comment posting, instead of an HTML response. We modified Vanilla Forums by setting an XSRF token that it used to a constant value.

WackoPicko v2 is an open-source intentionally vulnerable web application which was originally created to evaluate many black-box web vulnerability scanners, and was described in Chapter 3. A registered user can upload pictures, comment on other user's pictures, and purchase another user's picture. Version 2 contains minor tweaks from the original paper, but no additional functionality.

WordPress v2 is an open-source blogging platform. An administrator can create blog posts, where guests can leave comments. No changes were made to this application.

WordPress v3 is an up-to-date version of the open-source blogging platform. Just like the previous version, administrators can create blog posts, while a guest can comment on blog posts. No changes were made to this application.

Scanner	Description	Language	Version
wget	GNU command-line website downloader.	C	1.12
w3af	Web Application Attack and Audit Framework.	Python	1.0-stable
skipfish	Open-source, high-performance vulnerability scanner.	C	2.03b
state-aware-scanner	Our state-aware vulnerability scanner.	Python	1.0

Table 4.2: Black-box web vulnerability scanners that we compared.

Black-Box Web Vulnerability Scanners

This section describes the black-box web vulnerability scanners that were compared against our approach, along with the configuration or settings that were used. Table 4.2 contains a short description of each scanner, the scanner's programming language, and the version number.

wget is a free and open-source application that is used to download files from a web application. While not a vulnerability scanner, wget is a crawler that will make all possible `GET` requests it can find. Thus, it provides an excellent baseline because vulnerability scanners make `POST` requests as well as `GET` requests and should discover more of the application than wget.

wget is launched with the following options: recursive, download everything, and ignore robots.txt.

w3af is an open-source black-box web vulnerability scanner which has many fuzzing modules. We enabled the blindSqli, eval, localFileInclude, osCommanding, remoteFileInclude, sqli, and xss fuzzing plugins.

skipfish is an open-source black-box web vulnerability scanner whose focus is on high speed and high performance. Skipfish epitomizes the "shotgun" approach, and boasts about making more than 2,000 requests per second to a web application on a LAN. Skipfish also attempts to guess, via a dictionary or brute-force, directory names. We disabled this behavior to be fair to the other scanners, because we do not want to test the ability to guess a hidden directory, but how a scanner crawls a web application.

state-aware-scanner is our state-aware black-box vulnerability scanner. We use HtmlUnit [55] to issue the HTTP requests and render the HTML responses. After crawling and building the state-graph, we utilize the fuzzing plugins from w3af to generate fuzzing requests. Thus, any improvement in code coverage of our crawler over w3af is due to our state-aware crawling, since the fuzzing components are identical.

Scanner Configuration

The following describes the exact settings that were used to run each of the evaluated scanners.

- wget is run in the following way:

  ```
  wget -rp -w 0 --waitretry=0 -nd
    --delete-after --execute robots=off
  ```

- w3af settings:

  ```
  misc-settings
  set maxThreads 0
  back
  plugins
  discovery webSpider
  audit blindSqli, eval,
    localFileInclude, osCommanding,
    remoteFileInclude, sqli, xss
  ```

- skipfish is run in the following way:

  ```
  skipfish -u -LV -W /dev/null -m 10
  ```

Methodology

We ran each black-box web vulnerability scanner against a distinct, yet identical, copy of each web application. We ran all tests on our local cloud [102].

Gallery, WordPress v2, and WordPress v3 do not require an account to interact with the website, thus each scanner is simply told to scan the test application.

For the remaining applications (PhpBB v2, PhpBB v3, SCARF, Vanilla Forums, and WackoPicko v2), it is difficult to fairly determine how much information to give the scanners. Our approach only requires a username/password for the application, and by its nature will discover the requests that log the user out, and recover from them. However, other scanners do not have this capability.

Thus, it is reasonable to test all scanners with the same level of information that we give our scanner. However, the other scanners lack the ability to provide a username and password. Therefore, we did the next best thing: For those applications that require a user account, we log into the application and save the cookie file. We then instruct the scanner to use this cookie file while scanning the web application.

While we could do more for the scanners, like preventing them from issuing the logout request for each application, we believe that our approach strikes a fair compromise and allows each scanner to decide how to crawl the site. Preventing the scanners from logging out of the application also limits the amount of the application they will see, as they will never see the web application from a guest's perspective.

4.5.2 Results

Table 4.3 shows the results of each of the black-box web vulnerability scanners against each web application. The column "% over Baseline" displays the percentage of code coverage improvement of the scanner against the wget baseline, while the column "Vulnerabilities" shows total number of reported vulnerabilities, true positives, unique true positives among the scanners, and false positives.

Scanner	Application	% over Baseline	Vulnerabilities			
			Reported	True	Unique	False
state-aware-scanner	Gallery	16.20%	0	0	0	0
w3af	Gallery	15.77%	3	0	0	3
skipfish	Gallery	10.96%	0	0	0	0
wget	Gallery	0%				
state-aware-scanner	PhpBB v2	38.34%	4	3	1	1
skipfish	PhpBB v2	5.10%	3	2	0	1
w3af	PhpBB v2	1.04%	5	1	0	4
wget	PhpBB v2	0%				
state-aware-scanner	PhpBB v3	115.45%	0	0	0	0
skipfish	PhpBB v3	60.21%	2	0	0	2
w3af	PhpBB v3	16.16%	0	0	0	0
wget	PhpBB v3	0%				
state-aware-scanner	SCARF	67.03%	1	1	1	0
skipfish	SCARF	55.66%	0	0	0	0
w3af	SCARF	21.55%	0	0	0	0
wget	SCARF	0%				
state-aware-scanner	Vanilla Forums	30.89%	0	0	0	0
w3af	Vanilla Forums	1.06%	0	0	0	0
wget	Vanilla Forums	0%				
skipfish	Vanilla Forums	-2.32%	17	15	2	2
state-aware-scanner	WackoPicko v2	241.86%	5	5	1	0
skipfish	WackoPicko v2	194.77%	4	3	1	1
w3af	WackoPicko v2	101.15%	5	5	1	0
wget	WackoPicko v2	0%				
state-aware-scanner	WordPress v2	14.49%	0	0	0	0
w3af	WordPress v2	12.49%	0	0	0	0
wget	WordPress v2	0%				
skipfish	WordPress v2	-18.34%	1	0	0	1
state-aware-scanner	WordPress v3	9.84%	0	0	0	0
w3af	WordPress v3	9.23%	3	0	0	3
skipfish	WordPress v3	3.89%	1	0	0	1
wget	WordPress v3	0%				

Table 4.3: Results of each of the black-box web vulnerability scanners against each application. The table is sorted by the percent increase in code coverage over the baseline scanner, wget.

The prototype implementation of our state-aware-scanner had the best code coverage for every application. This verifies the validity of our algorithm: Understanding state is necessary to better exercise a web application.

Figure 4.6 visually displays the code coverage percent improvement over wget. The most important thing to take from these results is the improvement state-aware-scanner has over w3af. Because we use the fuzzing component of w3af, the only difference

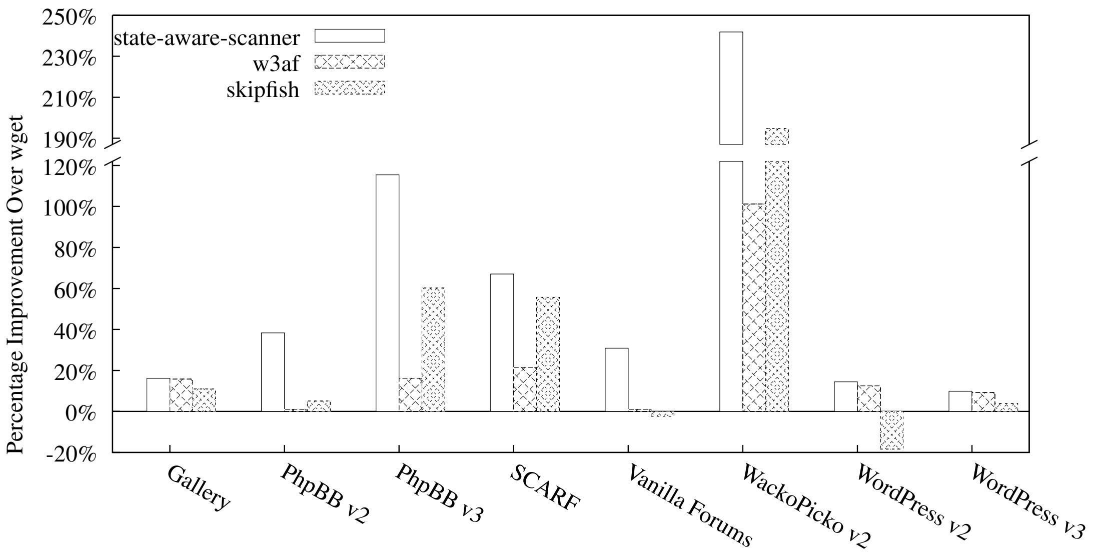

Figure 4.6: Visual representation of the percentage increase of code coverage over the baseline scanner, wget. Important to note is the gain our scanner, state-aware-scanner, has over w3af, because the only difference is our state-aware crawling. The y-axis range is broken to reduce the distortion of the WackoPicko v2 results.

is in our state-aware crawling. The results show that this gives state-aware-scanner an increase in code coverage from as little as half a percent to 140.71 percent.

Our crawler discovered three unique vulnerabilities (vulnerabilities that no other scanner found), one each in PhpBB v2, SCARF, and WackoPicko v2. The SCARF vulnerability is simply a XSS injection on the comment form. w3af logged itself out before fuzzing the comment page. skipfish filed the vulnerable page under "Response varies randomly, skipping checks." However, the content of this page does not vary randomly, it varies because skipfish is altering it. This *random* categorization also prevents skipfish from detecting the simple XSS vulnerability on WackoPicko v2's guestbook.

This result shows that a scanner needs to understand the web application's internal state to properly decide *why* a page's content is changing.

Skipfish was able to discover 15 vulnerabilities in Vanilla Forums. This is impressive, however, 14 stem from a XSS injection via the referer header on an error page. Thus, even though these 14 vulnerabilities are on different pages, it is the same root cause.

Surprisingly, our scanner produced less false positives than w3af. All of w3af's false positives were due to faulty timing detection of SQL injection and OS commanding. We believe that using HtmlUnit prevented our scanner from detecting these spurious vulnerabilities, even though we use the same fuzzing component as w3af.

Finally, our approach inferred the state machines of the evaluated applications. These state machines are very complex in the large applications. This complexity is because modern, large, application have many actions which modify the state. For instance, in WackoPicko v2, a user can log in, add items to their cart, comment on pictures, delete items from their cart, log out of the application, register as a new user, comment as this new user, upload a picture, and purchase items. All of these actions interact to form a complex state machine. The state machine our scanner inferred captures this complex series of state changes. The inferred WackoPicko v2 state machine is presented in Figure 4.7.

4.6 Limitations

Although dynamic page generation via JavaScript is supported by our crawler as allowed by the HtmlUnit framework [55], proper AJAX support is not implemented. This means that our prototype executes JavaScript when the page loads, but does not execute AJAX calls when clicking on links.

Nevertheless, our approach could be extended to handle AJAX requests. In fact, any interaction with the web application always contains a request and response, however the content of the response is no longer an HTML page. Thus, we could extend our notion of a "page" to typical response content of AJAX calls, such as JSON or XML. Another way to handle AJAX would be to follow a Crawljax [96] approach and covert the dynamic AJAX calls into static pages.

Another limitation of our approach is that our scanner cannot be used against a web application being accessed by other users (i.e., a public web application), because the other users may influence the state of the application (e.g., add a comment on a guestbook) and confuse our state change detection algorithm.

4.7 Conclusion

We have described a novel approach to inferring, as much as possible, a web application's internal state machine. We leveraged the state machine to drive the state-aware fuzzing of web applications. Using this approach, our crawler is able to crawl—and thus fuzz—more of the web application than a classical state-agnostic crawler. We believe our approach to detecting state change by differences in output for an identical response is valid and should be adopted by all black-box tools that wish to understand the web application's internal state machine.

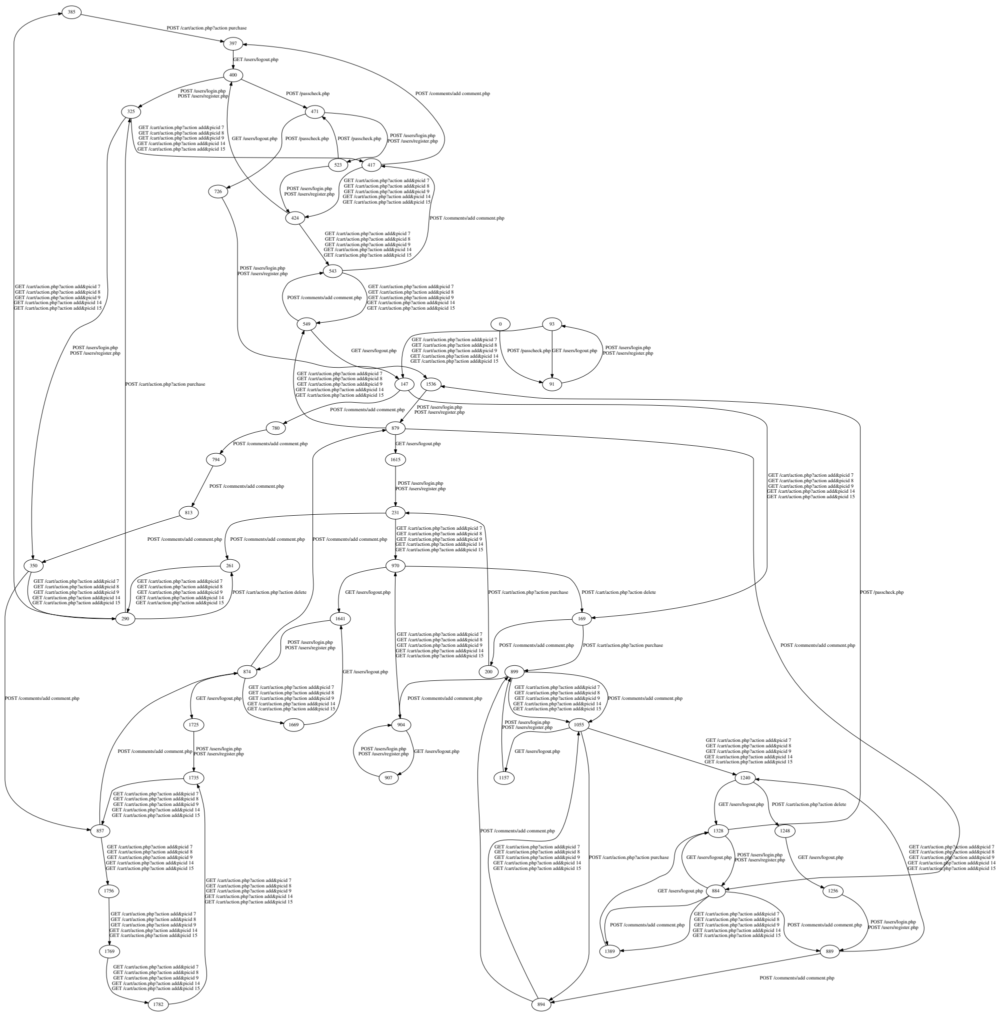
Figure 4.7: State machine that state-aware-scanner inferred for WackoPicko v2.

Chapter 5

Discovering and Mitigating Execution After Redirect Vulnerabilities

Now, we turn our attention to the study of a novel class of web application vulnerabilities called Execution After Redirect. In this chapter, we describe the vulnerability class and create a tool to statically find Execution After Redirect vulnerabilities in Ruby on Rails web applications.

An increasing number of services are being offered on-line. For example, banking, shopping, socializing, reading the news, and enjoying entertainment are all available on the web. The increasing amount of sensitive data stored by web applications has attracted the attention of cyber-criminals, who break into systems to steal valuable information such as passwords, credit card numbers, social security numbers, and bank account credentials.

Attackers use a variety of vulnerabilities to exploit web applications. In 2008, Albert Gonzalez was accused and later convicted of stealing 40 million credit and debit cards from major corporate retailers, by writing SQL injection attacks [74, 109]. Another common vulnerability, cross-site scripting (XSS), is the second highest-ranked entry on the OWASP top ten security risks for web applications, behind injection attacks like SQL injection [107]. Thus, SQL injection and XSS have received a large amount of attention by the security community. Other popular web application vulnerabilities include cross site request forgery (XSRF) [13], HTTP parameter pollution (HPP) [10, 30], HTTP response splitting [85], and clickjacking [9, 63].

In this chapter, we present an in-depth study of a little-known real-world web application logic flaw; one we are calling Execution After Redirect (EAR). An EAR occurs because of a developer's misunderstanding of how the web application framework operates. In the normal workflow of a web application, a user sends a request to the web application. The web application receives this request, performs some server-side processing, and returns an HTTP response. Part of the HTTP response can be a notification that the client (a web browser) should look elsewhere for the requested resource. In this case, the web application sets the HTTP response code to `301`, `302`, `303`, or `307`, and adds a `Location` header [51]. These response codes instruct the browser to look for the resource originally requested at a new URL specified by the web application in the HTTP `Location` header [50]. This process is known as redirection[1]; the web application redirects the user to another resource.

Intuitively, one assumes that a redirect should end execution of the server side code; the reason is that the browser immediately sends a request for the new location as soon as the redirection response is received, and it does not process the rest of the web application's output. Some web frameworks, however, do not halt execution on a redirect. This can lead to EAR vulnerabilities.

Specifically, an EAR can be introduced when a web application developer writes code that issues an HTTP redirect under the assumption that the redirect will automatically halt execution of the web application. Depending on the framework, execution can continue after the call to the redirect function, potentially violating the security properties of the web application.

We define *halt-on-redirect* as a web framework behavior where server-side code execution halts on a redirect, thus preventing EARs. Unfortunately, some languages make halt-on-redirect difficult to implement, for instance, by not supporting a `goto`-type statement. Therefore, web frameworks differ in supporting halt-on-redirect behavior. This difference in redirect method semantics can increase the developer's confusion when developing applications in different frameworks.

[1]In this chapter, we consider only HTTP server-side redirection. Other forms of redirection, executed on the client, exist such as JavaScript redirect or HTML meta refresh.

In this chapter, we present a comprehensive study of Execution After Redirect vulnerabilities: we provide an overview of EARs and classify EARs into different types. We also analyze nine web application frameworks' susceptibility to EARs, specifying their redirect semantics, as well as detailing what exactly makes them vulnerable to EARs. Moreover, we develop a novel static analysis algorithm to detect EARs, which we implemented in an open-source tool to analyze Ruby on Rails web applications. Finally, we discovered hundreds of vulnerabilities in open-source Ruby on Rails web applications, with a very low false positive rate.

In summary, this chapter provides the following contributions:

- We categorize EARs and provide an analysis of nine frameworks' susceptibility to various types of EARs.
- We discuss the results from the EAR challenge contained within our 2010 International Capture the Flag Competition.
- We present an algorithm to statically detect EARs in Ruby on Rails applications.
- We run our white-box tool on 18,127 open-source Ruby on Rails applications, which found 3,944 EARs.

5.1 Overview of EARs

An Execution After Redirect vulnerability is a logic flaw in web applications that results from a developer's misunderstanding of the semantics of redirection. Very often this misunderstanding is caused by the web framework used by the developer[2]. In particular, developers typically assume that the web application will halt after calling a function of the web framework that performs a redirect. Certain web frameworks, however, do not halt execution on a redirect, and instead, execute all the code that follows the redirect operation. The web browser perpetuates this misunderstanding, as it obediently performs the redirect, thus falsely indicating that the code is correct. As a result, when the developer tests the web application using the browser, the observed behavior seems in line with the intentions of the developer, and, consequently, the application is assumed to be correct.

Note that an EAR is not a code injection vulnerability; an attacker cannot execute arbitrary code, only code already present after the redirect. An EAR is also different from XSS and SQL injection vulnerabilities; it is not an input validation flaw, but rather a mismatch between the developer's intentions and the actual implementation.

As an example, consider the EAR vulnerability in the Ruby on Rails code shown in Listing 5.1. The code appears to redirect the current user to "/" if she is not an

[2] This misunderstanding was confirmed by a developer who responded to us when we notified him of an EAR in his code, who said, "I wasn't aware at all of this problem because I thought ruby on rails will always end any execution after a redirect." This example shows that developers do not always understand how their web framework handles redirects.

```
 1 class TopicsController < ApplicationController
 2  def update
 3   @topic = Topic.find(params[:id])
 4   if not current_user.is_admin?
 5     redirect_to("/")
 6   end
 7   @topic.update_attributes(params[:topic])
 8   flash[:notice] = "Topic updated!"
 9  end
10 end
```

Listing 5.1: Example of an Execution After Redirect vulnerability in Ruby on Rails.

administrator (Line 5), and, if she is an administrator, `@topic` will be updated with the parameters sent by the user in the `params` variable (Line 7). The code does not execute in this way, because Ruby on Rails does not support halt-on-redirect behavior. Thus, *any* user, not only the administrator, can update the topic, violating the intended authorization and compromising the security of the web application.

The simple way to fix Listing 5.1 is to add a `return` after the `redirect_to` call on Line 5. This will cause the `update` method to terminate after the redirect, thus, no additional code will be executed. Adding a return after all redirects is a good best practice, however, it is insufficient to prevent all EARs. Listing 5.2 depicts an example of an EAR that cannot be prevented by adding a return after a redirect. Here, the `redirect_to` on Line 4 is followed by a `return`, so there is no EAR in the `ensure_admin` method. However, `ensure_admin` is called by `delete` on Line 10, which calls `redirect_to` on Line 4. The `return` call on Line 5 will return the control flow back into the `delete` method, and execution will continue on

Line 11. Thus, the `@user` object will still be deleted on Line 12, regardless of whether the `current_user` is an administrator or not, introducing an EAR. Unfortunately in some frameworks, the developer cannot simply use `exit` instead of return to halt execution after a redirect because the web application is expected to handle multiple requests. Therefore, calling `exit` would kill the web application and prevent further requests.

5.1.1 EAR History

Execution After Redirect vulnerabilities are not a new occurrence; we found 17 Common Vulnerabilities and Exposures (CVE) EAR vulnerabilities dating back to 2007. These CVE entries were difficult to find because EARs do not have a separate vulnerability type; the EAR CVE vulnerabilities we found[3] were spread across different Common Weakness Enumeration Specification (CWE) types: "Input Validation," "Authentication Issues," "Design Error," "Credentials Management," "Code Injection," and "Permissions, Privileges, and Access Control." These vulnerabilities types vary greatly, and this indicates that EARs are not well understood by the security community.

[3]The interested reader is directed to the following EARs: CVE-2009-2168, CVE-2009-1936, CVE-2008-6966, CVE-2008-6965, CVE-2008-0350, CVE-2007-6652, CVE-2007-6550, CVE-2007-6414, CVE-2007-5578, CVE-2007-4932, CVE-2007-4240, CVE-2007-2988, CVE-2007-2776, CVE-2007-2775, CVE-2007-2713, CVE-2007-2372, and CVE-2007-2003.

```
class UsersController < ApplicationController
  def ensure_admin
    if not current_user.is_admin?
      redirect_to("/")
      return
    end
  end

  def delete
    ensure_admin()
    @user = User.find(params[:id])
    @user.delete()
    flash[:notice] = "User Deleted"
  end
end
```

Listing 5.2: Example of a complex Execution After Redirect vulnerability in Ruby on Rails.

5.1.2 EARs as Logic Flaws

While logic flaws are typically thought of as being unique to a specific web application, we believe EARs are logic flaws, even though they are systemic to many web applications. Because an EAR is the result of the developer's misunderstanding of the web application framework, there is an error in her logic. The intuition is that the redirect is an indication of the developer's intent for ending server-side processing. A redirect can be thought of as a `goto` - the developer, in essence, wishes to tell the user to look somewhere else. However, it does not act as a `goto`, because the server-side control flow of the application is not terminated, even though that is how it appears from the perspective of the client.

There are almost no valid reasons to have code executed after a redirect method. The few exceptions are: performing cleanup actions, such as closing open files, and starting long-running processes, such as encoding a video file. In the former case, the cleanup code can be executed before a redirect, and in the latter case, long-running processes can be started asynchronously, alleviating the need to have code executed after a redirect.

Because there is no reason to execute code after a redirect, we can infer that the presence of code executed after a redirect is a logic flaw.

5.1.3 Types of EARs

Execution After Redirect logic flaws can be of two types: benign or vulnerable. A benign EAR is one in which no security properties of the application are violated, even though additional, unintended, code is executed after a redirect. For example, the code executed after the redirect could set a local variable to a static string, and the local variable is not used or stored. Although no security properties are violated, a benign EAR may indicate that a developer misunderstood the redirect semantics of the web framework, posing the risk that code will, in the future, be added after the redirect, elevating the EAR from benign to vulnerable.

A vulnerable EAR occurs when the code executed after the redirect violates the security properties of the web application. More specifically, in a vulnerable EAR

```php
$current_user = get_current_user();
if (!$current_user->is_admin())
{
   header("Location: /");
}
echo "Sensitive Information";
```

Listing 5.3: Example of an information leakage Execution After Redirect vulnerability in PHP. If the `current_user` is not an administrator, the PHP `header` function will be called, redirecting the user to "/". However, the sensitive information will still be returned in the output, thus leaking information. The fix is to call the `exit` function after the `header` call.

the code executed after the redirect allows unauthorized modification to the state of the web application (typically the database), and/or causes leakage (reads and returns to the browser) of data to an unauthorized user. In the former case (e.g., see Listing 5.1), the integrity of the web application is compromised, while in the latter case, the confidentiality of the web application is violated (e.g., see Listing 5.3). Thus, every vulnerable EAR is an instance of broken/insufficient access controls, because the redirect call is an indication that the user who made the request is not allowed to access the requested resource.

EAR vulnerabilities can be silent. In a silent EAR, the execution of code does not produce any output. This lack of information makes silent EARs difficult to detect via a black-box approach, while information leakage EARs are easier to detect with black-box tools. Listings 5.1 and 5.2 are examples of silent EARs, and Listing 5.3 is an example of an information leakage EAR.

5.1.4 Framework Analysis

Web application frameworks vary on supporting halt-on-redirect behavior. Therefore, different frameworks provide protection against different kinds of EAR vulnerabilities. The differing semantics of redirects increases the confusion of developers. A developer we contacted said, "I didn't realize that [Ruby on Rails'] redirect_to was like PHP's header redirect and continued to run code." Thus, an understanding of the web framework's redirect semantics is essential to produce correct, EAR-free, code.

We analyzed nine of the most popular web frameworks to see how they differ with respect to their built-in redirect functions. The nine frameworks were chosen based on their StackOverflow activity, and include one framework for each of the Ruby, Groovy, and Python languages, three frameworks for the PHP language, one framework that can be applied to both C# and Visual Basic, and two frameworks for the Java language [22]. While the frameworks selected for analysis are not exhaustive, we believe they are diverse and popular enough to be representative of real-world usage.

To analyze the frameworks, we created nearly identical copies of a simple web service in each of the nine web frameworks. This web service provided access to four pages within the web application. The first was the root page, "/", which simply linked to the other three pages. The second was the redirect page, "/redirect", which was used to test proper redirect behavior. The third was the EAR page, "/ear", which called the framework's redirect function, appended a message to a log file regarding the request, and finally attempted to return a rendered response to the browser. The last page was the log page, "/log", which simply displayed the contents of the log file.

Using this design for the web application allowed us to check for integrity violations, represented by the appended log message, and confidentiality violations, represented by output sent after the HTTP redirect response when requesting the EAR page. We approached the implementation of this web application in each framework as many developers new to that framework would. That is, whenever possible, we followed the recommended tutorials and coding practices required to build a web application in the framework.

A brief background on the model-view-controller (MVC) software architecture is necessary to follow our analysis, as each framework analyzed fits the MVC pattern. The MVC architecture supports the separation of the persistent storage (model), the user interface (view), and the control flow (controller) [116]. More precisely, the models interact with the database, the views specify the output to return to the client, and the controllers are the glue that puts everything together. The controller must handle HTTP requests, fetch or update models, and finally return a view as an HTTP response. When following the MVC paradigm, a controller is responsible for issuing a redirect call.

The following sections describe our analysis of each framework's susceptibility to EAR vulnerabilities based on their redirect functions' use and documentation. We developed the test application in the latest stable version of each framework available at the time. The version numbers are listed adjacent to the framework name in the section headers.

Ruby on Rails 3.0.5

Ruby on Rails, commonly referred to as Rails, is a popular web application framework. Unfortunately, Rails is susceptible to EAR vulnerabilities. Rails provides the `redirect_to` function, which prepares the controller for sending the HTTP redirect. However, the redirect is not actually sent at this point, and code continues to execute following the call to `redirect_to`. In Rails, there is no mechanism to ensure that code halts following a redirect, thus if exit is called, a developer must return from the controller's entry function without executing additional code.

As previously mentioned in Section 5.1, the Ruby `exit` command cannot be used to halt the execution of a controller after a redirect. This is for two reasons: the first is that `redirect_to` does not immediately send output when it is called, thus if `exit` is called, the user will never see the redirect. The second reason is that Rails web applications are long-running processes that handle multiple incoming requests, unlike PHP, which typically spawns a new instance for each request. Therefore, calling `exit` to halt execution is not feasible, as it will terminate the Rails application, preventing it from handling further requests.

On a positive note, information leakage EARs are impossible in Rails web applications because a controller can either perform a redirect, or `render` a response (view) to the user. Any call to `render` after a redirect will result in Rails throwing a `DoubleRenderError`. This exception is thrown in all possible combinations: render after a redirect, render after a render, redirect after a render, and redirect after a redirect.

Grails 1.3.7

Grails is a framework written in Groovy, which was modeled after the Ruby on Rails framework. Thus, Grails behaves in a manner nearly identical to Rails with respect to redirects. Specifically, code will continue to execute following a call to the redirect function, and, therefore, the developer must take precautions to avoid creating an EAR vulnerability. Unfortunately, as of this writing, nowhere in the Grails documentation on redirects does it mention that code will continue to execute following a redirect [130].

Unlike Ruby on Rails, the behavior of Grails is somewhat less predictable when it comes to the order of view rendering and/or calls to redirect. To explain, we will say that to "render" means to output a view, and to "redirect" means to call the redirect function. As previously mentioned in Section 5.1.4, in Rails, only one render or one redirect may be called in a controller; a `DoubleRenderError` is thrown in the case of multiple calls. In Grails, however, the only redirect exception, `CannotRedirect-`

`Exception`, occurs when a redirect is called following another redirect. In cases where multiple calls to render are made, the final render is the only one that is sent to the browser. More importantly, in cases where both redirect and render are called, regardless of their order, the redirect is actually sent to the browser and the render call is simply ignored. Due to this behavior of Grails, it is not vulnerable to an information leakage EAR. However, like Rails, it is still vulnerable to silent EARs that violate the integrity of the application.

Django 1.2.5

Django is a Python web application framework that differs in its handling of redirects compared to the other frameworks (save for ASP.NET MVC). Rather than calling functions to render or perform the redirect, Django requires the developer to return an `HttpResponse` object from each controller. Django's documentation makes it clear that calling Django's `redirect` function merely returns a subclass of the `HttpResponse` object. Thus, there is no reason for the developer to expect the code to halt when calling `redirect`. The actual HTTP redirect is sent to the browser only if this object is also returned from the controller's entry point, thereby removing the possibility of further code execution [45]. Because the controller's entry point can only return a single `HttpResponse` object, the developer can rely completely on her browser for testing purposes. This behavior makes Django impervious to all EARs.

ASP.NET MVC 3.0

ASP.NET MVC is a web application framework developed by Microsoft that adds a Model-View-Controller paradigm on top of traditional ASP.NET, which includes the languages C# and Visual Basic [7]. ASP.NET MVC is similar to Django, in that all controllers must return an `ActionResult` object. In order to perform redirection, either a `RedirectResult` or `RedirectToRouteResult` object must be returned, which are both subclasses of `ActionResult`. Like Django, this behavior makes ASP.NET MVC impervious to all EARs.

Zend Framework 2.3

By default, the PHP based Zend Framework is not susceptible to EAR vulnerabilities because its redirect methods immediately result in the termination of server-side code. This default behavior is consistent in the two methods used to perform a redirect in the Zend Framework. The simplest method is by using the `_redirect` method of the controller, however, the recommended method is to use the `Redirector` helper object [147].

While the default behavior is not vulnerable to EARs, the Zend Framework supports disabling halt-on-redirect for both methods. The `_redirect` method will not halt when the keyword argument `exit=False` is provided as part of the call. Disabling halt-on-redirect when using the `Redirector` helper object requires calling the method `SetExit(False)` on the `Redirector` helper object prior to making the redirect call. The latter method is particularly interesting because any code executed during the request has the ability to modify the behavior of redirects called using the `Redirector` helper. Fortunately, even when using the `Redirector` helper, the developer has the option of using a set of functions suffixed with "AndExit" that always halt-on-redirect.

When halt-on-redirect is disabled in Zend, it becomes vulnerable to integrity violation EARs. However, the default view rendering behavior no longer occurs. Thus, even when modifying the default behavior, information leakage EARs will never occur in the Zend Framework.

CakePHP 1.3.7

Similar to the Zend Framework, the CakePHP framework is also not susceptible to EAR vulnerabilities out of the box. By default, CakePHP's single redirect method immediately results in the termination of the PHP script. In a manner similar to the Zend Framework, this default behavior can be modified by setting the third argument of `redirect` to `False`, which in turn also disables the default mechanism for view rendering [29]. Thus CakePHP is vulnerable to EARs in exactly the same way as the Zend Framework.

CodeIgniter 2.0.0

Unlike the Zend Framework and CakePHP, CodeIgniter is a very lightweight PHP framework, and thus, it does not offer much out of the box. Nevertheless, the framework still provides a `url` helper class that contains a redirect method [47]. CodeIgniter's redirect method always exits after setting the redirect header; a behavior that cannot be changed. Therefore CodeIgniter is impervious to EARs when developers use only the provided redirect function. Unfortunately, the url helper class must be included manually. As a result, there is the risk that developers will not use the provided redirect function and instead introduce EARs by neglecting to call `exit` following a call to `header("Location:<path>")`.

J2EE 1.4

Java 2 Platform, Enterprise Edition (J2EE) defines a servlet paradigm for the development of web applications and web application frameworks in Java. Thus, to perform a redirect in J2EE, or a J2EE-based framework, the developer calls `HttpServlet-Response.sendRedirect`. This redirect function will clear out everything previously in the output buffer, set the `Location` header to the redirect location, set the response code to 302, and finally flushes the output buffer to the browser. However, `sendRedirect` does not halt execution of the servlet. Thus, only silent EARs are present in J2EE web applications, or any framework that is based on J2EE servlets.

Struts 2.2.3

Apache Struts is an MVC framework that is built on top of the servlet model provided by J2EE. Thus, Struts inherits all the potential vulnerabilities of the J2EE framework, specifically that silent EARs are possible but information leakage EARs are not possible. This inheritance is possible because to perform a redirect, the `HttpServletResponse.sendRedirect` method of J2EE must be called.

5.1.5 EAR Security Challenge

Each year since 2003, we have organized and hosted a security competition called the International Capture the Flag (iCTF). The competition pits dozens of teams from various universities across the world against each other in a test of their security skills and knowledge. While each iCTF has a primary objective, the competitions typically involve secondary security challenges tangential to the primary objective [32].

For the 2010 edition of the iCTF, we constructed a security challenge to observe the familiarity of the teams to Execution After Redirect vulnerabilities. The challenge involved a vulnerable EAR that violated both the confidentiality and the integrity of the web application. The confidentiality was violated when the web application's administrator view was leaked to unauthorized users following a redirect; the unauthorized users were "correctly" redirected to an error page. The information contained in the leaked view provided enough information to allow for an integrity violation had the database not purposefully been in a read-only state. More importantly, the initial data leak provided the means to leak further information, thus allowing teams to successfully solve the challenge [21].

The crux of the EAR challenge relied on the automatic redirecting of web browsers and other web clients, such as `wget` and `curl`. To our surprise, many of the teams relied only on the output produced by their web browser, and, therefore, failed to notice the leaked information. It is important to note that the teams in this competition are primarily made up of graduate and undergraduate level students from various universities; many would not be considered security professionals. Nevertheless, we assumed that the meticulous eye of a novice-to-intermediate level hacker attempting to break into a web service would be more likely to detect information leakage when compared to a web developer testing their application for “correct” page flow.

Of the 72 teams in the competition, 69 contacted the web server at least once. 44 of these 69 teams advanced past the first step, which required them to submit a file as per the web application’s specifications. 34 of the 44 teams advanced past the second step, which required them to brute force a two-digit password. It was at this point that the EAR vulnerability was exposed to the teams, resulting in both a redirect to the unauthorized error page and the leakage of the administrator page as part of the HTTP redirect response. Of the 34 teams who made it this far, only 12 successfully discovered and exploited the vulnerability. The fact that only 12 out of 34 teams were successfully able to discover the information leaked to their browser in a hacking competition indicated that more research and exposure was necessary for EAR vulnerabilities.

5.2 EAR Detection

In this section, we discuss the design and implementation of our system to detect EAR vulnerabilities. This system uses static source code analysis to identify cases in which code might be executed after the call to a redirect function. We also introduce a heuristic to distinguish benign EARs from vulnerable EARs.

Our tool targets the Ruby language, specifically the Ruby on Rails web framework. We chose this framework for two reasons. First, Ruby on Rails is a very popular web framework, thus, there is a large number of open-source Ruby on Rails web applications available for inspection (e.g., on GitHub [57]). Second, due to the characteristics discussed in Section 5.1.4, all EARs present in Rails are silent. Thus, it is necessary to use a white-box tool to detect EARs in Ruby on Rails web applications. Again, it is important to note that redirects originate within the controllers[4], thus, our white-box tool operates specifically on controllers.

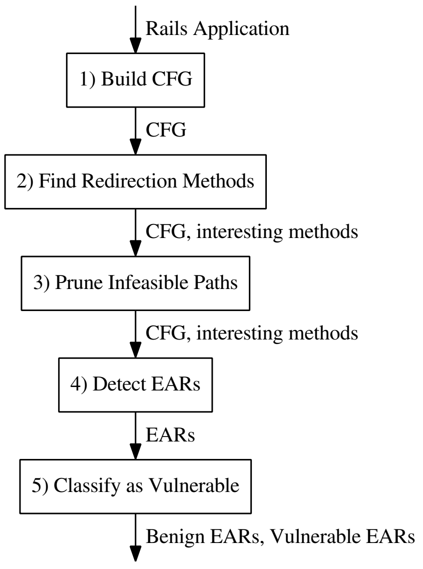
Rails Application
1) Build CFG
CFG
2) Find Redirection Methods
CFG, interesting methods
3) Prune Infeasible Paths
CFG, interesting methods
4) Detect EARs
EARs
5) Classify as Vulnerable
Benign EARs, Vulnerable EARs

Figure 5.1: The logical flow of the white-box tool.

5.2.1 Detection Algorithm

The goal of our EAR detector is to find a path in the controller's Control Flow Graph (CFG) that contains both a call to a redirect method and code following that redirect method. An overview of our algorithm is given in Figure 5.1. The algorithm operates in five steps: (i) generate the CFG of the controller; (ii) find redirection methods; (iii) prune infeasible paths in the CFG to reduce false positives; (iv) detect EARs by finding a path in the CFG where code is executed after a redirect method is called; (v) use a heuristic to differentiate between benign and vulnerable EARs.

[4]Redirects can also occur in Rails' routing, before the request gets to the controller. However, EARs cannot occur in this context, because control flow never reaches a controller. Thus, we are not concerned with these redirects.

Step 1: Building the Control Flow Graph

We built our system on top of the Ruby parser presented by Furr et al. [54]. This parser first compiles Ruby into a subset of the Ruby language called Ruby Intermediate Language, or RIL. The purpose of RIL is to simplify Ruby code into an easier-to-analyze format. The simplification is performed by removing ambiguities in expressions, reducing Ruby's four different branches to one canonical representation, making method calls explicit, and adding explicit returns. At the end of the transformation, every statement in RIL is either a statement with one side effect or a branch. The parser generates the CFG of RIL.

Due to Ruby's dynamic nature, this CFG might be incomplete. In particular, strings containing Ruby code can be evaluated at run-time using the `eval` function, object methods can be dynamically called at run-time using the `send` function, and methods can be added to objects at run-time. We do not address EAR vulnerabilities associated with these language features. However, we have found that these features are rarely used in practice (see Section 5.2.2).

Step 2: Finding Redirection

To detect EARs, we must first find all program paths (from any program entry to any program exit point) in the CFG that call the Ruby on Rails method `redirect_to`. The reason is that we need to check these paths for the presence of code execution between the redirect call and the program exit point. Note that intra-procedural analysis is not enough to find all EARs. Consider the code in Listing 5.2. Simply looking in `ensure_admin` for code execution *after* the call to `redirect_to` and *before* the end of this method is not sufficient. Thus, we need to perform inter-procedural analysis to find all possible ways in which code execution can continue after a `redirect_to` call until the end of the program.

Our inter-procedural analysis proceeds as follows: we start by finding all methods that directly call `redirect_to`. These methods are added to a set called *interesting methods*. Then, for each method in the *interesting methods* set, we add to this set all methods that call it. This process is iterated until a fixpoint is reached, and no new interesting methods are found.

At this point, every element (method) in *interesting methods* can eventually lead to a `redirect_to` call. Whenever a call to an interesting method returns, its execution will continue after the call site in the caller. Thus, all paths from invocations of `redirect_to` until the end of the program are captured by the paths from all invocations (call sites) of interesting methods to the end of the methods that contain these calls. Now, to detect an EAR, we can simply look for code that is executed on a path from the call site of an interesting method until the end of the method that contains this call.

```
 1 class UsersController < ApplicationController
 2   def ensure_logged_in
 3     if not current_user
 4       redirect_to("/") and return false
 5     end
 6     @logged_in_users += 1
 7     return true
 8   end
 9
10   def delete_all
11    if not ensure_logged_in()
12      return
13    User.delete(:all)
14   end
15 end
```

Listing 5.4: Example of a potential false positive.

Step 3: Prune Infeasible Paths

Looking for all paths from the `redirect_to` method to the program exit point might lead to false positives due to infeasible paths. Consider the example in Listing 5.4. There are no EARs in this code. The `redirect_to` on Line 4 will always return `true`, thus, `return false` (also on Line 4) will execute as well. Because of this, `ensure_logged_in` will always return `false` after performing a redirect. As a result, the call to `ensure_logged_in` on Line 11 will always return `false`, and the `return` on Line 12 will always occur.

The CFG for the code in Listing 5.4 is shown in Figure 5.2. With no additional processing, we would incorrectly report the path from `redirect_to` on Line 4 to the statement in Line 6. Moreover, we would also report an EAR because of the path

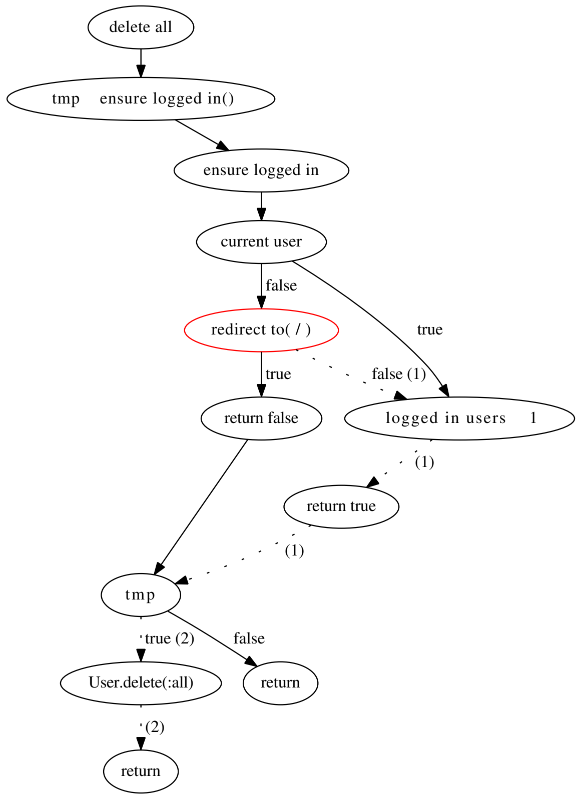

Figure 5.2: Control Flow Graph for the code shown in Listing 5.4. The dotted lines are paths removed from the CFG by Step 3 of the EAR detection algorithm.

from the redirect to the `User.delete` on Line 13. The first path is denoted as (1) in Figure 5.2, the second path as (2).

To prune infeasible paths in the CFG, we explore all paths that follow an interesting method. If *all* paths following an interesting method call return the same Boolean value, we propagate this Boolean constant to all the call sites of this method. Then, we recursively continue constant value propagation at all the call sites, pruning infeasible paths everywhere after the interesting method is called. We iteratively continue this process throughout the CFG; whenever we find a constant return value, we propagate this return value to all call sites.

Figure 5.2 shows the results of performing our pruning process on the CFG of Listing 5.4. Initially, all paths after the `redirect_to` in `ensure_logged_in` do not return the same Boolean, so we cannot conclude anything about the return value of `ensure_logged_in`. However, `redirect_to` always returns `true`. Therefore, we perform constant value propagation on the return value of `redirect_to`, which is used in a branch. As a consequence, we can prune all of the paths that result from the `false` branch. The edges of this path are labeled with (1) in Figure 5.2. Now, all paths from `redirect_to` return `false`, which means that `ensure_logged_in` will always return `false` after a redirect. We now perform constant value propagation at all the call sites of `ensure_logged_in`, removing all the paths labeled with (2). At this point, there is nothing more to be pruned, so we stop. It can be seen that there is no path from `redirect_to` to state-changing code (defined in the next step) along the solid lines.

Step 4: Detecting EARs

Once the CFG of the controller has been simplified and interesting method information has been extracted, we perform EAR detection. This is a fairly simple process; we traverse the CFG of every method to see if potentially problematic code can be executed after a call to an interesting method. We conservatively define such code as any statement that could possibly modify the program state, excluding statements that alter the control flow. This excludes `return` and branches, but includes assignment and method calls. As a special case, we also disregard all operations that set the `flash` or `session` array variable. These arrays are used in the former case to set a message to be displayed on the destination page, and in the latter case to store some information in the user's session. These calls are disregarded because they do no affect the state of the web application and are frequently called after redirection. We report as a potential EAR each method that executes potentially problematic code between the invocation of an interesting method and its return statements.

Step 5: Distinguishing Between Benign and Vulnerable EARs

We also introduce a heuristic to identify vulnerable EARs. This heuristic looks for paths from an interesting method to a function that modifies the database. If one is found, the EAR is marked as vulnerable. We used the Rails documentation to determine the 16 functions that modify the database. Of course, this list can be easily extended. This process is not sound, because we perform no type analysis, and look only at the method names being called. Moreover, we do not analyze the models, only looking for this specific list. Despite these limitations, our results (Section 5.3.1) show that this heuristic is still a good indicator of potentially vulnerable EARs that deserve the developer's attention.

5.2.2 Limitations

The white-box EAR detector is limited to analyzing Ruby on Rails applications, although the detection algorithm can be extended to any programming language and web framework. Detection is neither sound nor complete. False negatives can occur when a Rails application uses Ruby's dynamic features such as `eval` or `send` to execute a redirect. While such dynamic features are used extensively in the Ruby on Rails framework itself, they are rarely used by web applications written in Rails. Of the 3,457,512 method calls in controllers that we tested our tool on, there were 428 (0.012%) `eval` method calls and 2,426 (0.07%) `send` method calls, which shows how infrequently these are used in Rails web applications.

The white-box tool can report two types of false positives: false EARs, that is, the tool reports an EAR although no code can be executed after a redirect, or false vulnerable EARs, where the tool mistakes a benign EAR as vulnerable.

False EARs can occur for several reasons. One reason is that the path from the redirect function to the code execution that we found is infeasible. A typical example is when the redirect call and the code execution occur in opposite branches. The branch conditions for these are mutually exclusive, so there can never be a path from the redi-

Type of EAR reported	Number reported
Benign	3,089
Vulnerable	855
Total	3,944
Total Projects	18,127
Any EAR	1,173
Only Benign	830
At least one vulnerable EAR	343

Table 5.1: Results of running the white-box detector against Ruby on Rails applications, 6.5% of which contained an EAR flaw. 2.9% of the projects had an EAR classified as vulnerable.

rect call to the code execution. Examples of this type of false positive are discussed in Section 5.3.1, and these could be mitigated by introducing better path sensitivity.

False vulnerable EARs are a problem caused by the heuristic that we use. The biggest issue is that we simply look for method calls that have the same name as method calls that update/change the database. However, we do not perform any type analysis to determine the *object* that the method is called on. Thus, methods such as `delete` on a hash table will trigger a false vulnerable EAR, since `delete` is also a method of the database object. Improved heuristics could be developed, for instance, that include the type of the object the method is being invoked on.

Despite these limitations, our experiments demonstrate that the tool works very well in practice. In addition, Ruby on Rails controllers are typically very small, as most application logic is present in the models. Thus, our tool works very well on these types of controllers. We provide[5] our tool to the community at large, so that others may use it to detect EARs in their code.

5.3 Results

We used our EAR detection tool to find real-world EARs in open-source Ruby on Rails web applications. First, we downloaded 59,255 open-source projects from GitHub [57] that were designated as Ruby projects and that were not a fork of another project. We identified 18,127 of the downloaded Ruby projects that had an `app/controllers` folder, indicating a Ruby on Rails application.

Table 5.1 summarizes the results. In total, we found 3,944 EAR instances in 1,173 projects. 855 of these EARs, present in 343 projects, were classified as vulnerable by our system. This means that 6.5% of Rails applications we tested contained at least one EAR, and 29.3% of the applications containing EARs had an EAR classified as vulnerable.

Of the 1,173 projects that contained at least one EAR, we notified those project owners that had emails listed in their GitHub profile, for a total of 624. Of these project owners, 107 responded to our email. Half of the respondents, 49, confirmed the EARs we reported. 26 other respondents told us that the GitHub project was no longer being maintained or was a demo/toy. Three respondents pointed out false positives, which we

[5]`https://github.com/adamdoupe/find_ear_rails`

Classification after manual analysis	Number
True Vulnerable EARs	485
Benign EARs	325
No EARs (False Positives)	45

Table 5.2: Results of manually inspecting the 855 vulnerable EARs reported by our white-box tool. 40.1% were benign, and 5.3% were not EARs.

confirmed, while 6 of the project owners said that there were not going to fix the EAR because there was no security compromise. The rest of the responses thanked us for the report but did not offer a confirmation of the reported EAR.

5.3.1 Detection Effectiveness

To determine the effectiveness of our tool, we manually inspected all 855 vulnerable EARs. The results are shown in Table 5.2. We manually verified that 485, or 59.9%, were true positives. Many of these were caused by ad-hoc authorization checks, where the developer simply introduced a redirect when the check failed. Some examples of security violations were allowing non-administrators access to administrator functionality, allowing modifications to items not belonging to the current user, and being able to sign up for a conference even though it was full.

Listing 5.5 shows an interesting example adapted from a real EAR where the redirect is followed by `and return` (Line 3), however, due to Ruby's semantics, this code contains an EAR. In Ruby, a `return` with no arguments returns `false`[6], thus,

[6]Technically `nil`, but `nil` and `false` are equivalent for Boolean comparisons.

```
 1 class BanksController < ApplicationController
 2   def redirect_to_login
 3     redirect_to("/login") and return
 4   end
 5
 6   def create
 7     if not current_user.is_admin?
 8       redirect_to_login() and return
 9     end
10     @bank = Bank.create(params[:bank])
11   end
12 end
```

Listing 5.5: True positive Execution After Redirect vulnerability in Ruby on Rails.

`redirect_to_login` will always return `false` (because of the `return` call with no arguments on Line 3). The result is that the `return` on Line 8 will never be executed, because `redirect_to_login` will always return `false`, and the short-circuit logic of `and` will cause Line 10 to be executed. This example shows that our tool discovers non-obvious EARs.

For vulnerable EARs, we consider two different types of false positives: false *vulnerable* EARs, which are benign EARs mistakenly reported as vulnerable, and false EARs (false positives).

As shown in Table 5.2, the white-box tool generated 45 false EARs, for a false positive rate of 5.3%. These false positives came from two main categories. About half of the false positives were due to impossible paths from the redirect methods to some code. An example of this is when a redirect method was called at the end of a branch that checked that the request was an HTTP GET, while the code executed after a redirect was in a branch that checked that the request was an HTTP POST. These two conditions are mutually exclusive, thus, this path is impossible. The other half of false positives were due to local variables that had the same name as a redirect method. The parsing library, RIL, mistakenly identified the local variable access as a method call to a redirect method. We are currently looking into fixing this issue in RIL, which will almost halve our false positive rate.

While our false EAR rate was only 5.5%, our vulnerable EAR detection heuristic had a higher false detection rate of 40.1%. The biggest culprit for false vulnerable EARs (72.9% of the instances) was due to no feasible path from the redirect to the method that changed the state of the database. For instance, the redirect method occurred in a branch that was taken only when a certain object was `nil`[7]. Later, the database method was called on this object. Thus, when the redirect happens, the object will be `nil`. Because of the presence of an EAR flaw, execution will continue and reach the database access method. However, since the object is `nil`, the database will not be affected. Because our heuristics cannot detect the fact that, after the redirect, the database function will always be called with a `nil` object, we report a vulnerability. The other common false vulnerable EAR were instances where the redirect method was called before code was executed, however, it was clear that the developer was fully aware of the redirect semantics and intended for the code to be executed.

[7]`nil` is Ruby's `null`.

We also checked that the false EAR rate did not differ significantly among the benign EARs by manually inspecting 200 random EARs reported as benign. We saw 13 false EARs in the manual inspection, for a false positive rate of 6.5%. Thus, the total false positive rate among the instances we manually inspected is 5.5%. We also did not see any vulnerable EARs among the benign EARs, thus, we did not see any false negative vulnerable EARs in our experiments.

From our results, we can conclude that we detect EARs well. However, it is more difficult to distinguish between benign and vulnerable EARs. Classification could be improved by using a better heuristic to detect intended redirects. However, even though certain EARs might not be vulnerable at the moment, they are still programming errors that should be fixed. This is confirmed by the responses that we received from developers who were grateful for error reports even though they are not exploitable at the moment. Also, our tool reports one true vulnerability for every benign EAR mistakenly classified as vulnerable. This is well in line with the precision of previous static analysis tools [72, 79, 92].

5.3.2 Performance

To evaluate the performance of our tool, we measured the running time against the 18,127 Ruby on Rails applications. We ran our experiments on an Intel Core i7 with 12 gigabytes of RAM. Our algorithm scales linearly with the size of the CFG and is fast; no project took longer than 2.5 seconds even with the largest CFG size of 40,217 statements.

5.4 Prevention

The old adage "an ounce of prevention is worth a pound of cure" is true in software. Boehm showed that the later in an application's life-cycle bugs are caught, the more expensive they are to fix [23]. Thus, preventing certain types of bugs from even being introduced is attractive from both an economic standpoint, and a security perspective. Our recommendation to web frameworks, therefore, is to make Execution After Redirect vulnerabilities impossible to occur, by having every invocation of the redirect method halt execution, which we call halt-on-redirect behavior.

As we have shown in Section 5.1.4, some frameworks have already either adopted the approach of making EARs impossible, or their approach to generating HTTP responses makes EARs highly unlikely. For existing frameworks that wish to decrease the chance of EARs being introduced, such draconian measures may not be acceptable because they break backward-compatibility. Our suggestion in these cases is to make an application-wide setting to enable halt-on-redirect behavior, along with an argument to the redirect function to halt execution after the redirect. Of course, we suggest making halt-on-redirect the default behavior, however each framework will have to properly balance security and backward-compatibility.

To improve the security of Ruby on Rails, we are in discussions with the Rails development team about our proposed change. The difficulty with implementing halt-on-redirect behavior in Rails is that there are no `goto`s, and Rails applications run in a single-threaded context. This limits the two obvious forms of implementing halt-on-redirect: we cannot use a goto or language equivalent statement to jump from the end of the `redirect_to` method to the code after the controller is called. Moreover, we also cannot, at the end of the `redirect_to` method, send the HTTP response and cause the current thread to stop execution. PHP frameworks can use the `exit` function to implement halt-on-redirect behavior, because each request spawns a new PHP process.

Our proposed solution is to throw a new type of exception, `RedirectOccured-Exception`, at the end of the `redirect_to` body. In the Ruby on Rails framework core, where the controller is called, there is a catch block for this exception. While this will prevent almost all EARs, there is a possibility for code to be executed in an `ensure` block, Ruby's equivalent of a "finally" block. Code in this block will be executed regardless of a redirect. However, we believe this is semantically in line with the way the language should work: ensure blocks will always be executed, no matter what happens, and this is clear to the programmer via the language's semantics.

5.5 Conclusions

We have described a new type of vulnerability, Execution After Redirect, and developed a novel static analysis tool to effectively find EARs. We showed that EARs are difficult to differentiate between benign and vulnerable. This difficulty is due to vulnerable EARs violating the specific logic of the web application. Better understanding of the application's logic should help differentiate vulnerable and benign EARs and it will be the focus of future work.

Chapter 6

Toward Preventing Server-Side XSS via Automatic Code and Data Separation

Automatically finding vulnerabilities, as we have done in the previous chapters, is a great way to find vulnerabilities in web applications. However, there is always the risk that the automated tools do not find a vulnerability, and these tools give no guarantees that all vulnerabilities are found. Another avenue to secure web applications from attack is to write the application in such a way as to make vulnerabilities impossible. In this chapter, we examine Cross-Site Scripting vulnerabilities as having the root cause of Code and Data mixing. By properly applying the basic security principles of Code and Data separation we can automatically prevent a wide swath of Cross-Site Scripting vulnerabilities.

Web applications are prevalent and critical in today's computing world, making them a popular attack target. Looking at types of vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [41], web application flaws are by far the leading class.

Modern web applications have evolved into complex programs. These programs are no longer limited to server-side code that runs on the web server. Instead, web applications include a significant amount of JavaScript code that is sent to and executed on the client. Such client-side components not only provide a rich and fast user interface, they also contain parts of the application logic and typically communicate with the server-side component through asynchronous JavaScript calls. As a result, client-side scripts are an integral component of modern web applications, and they are routinely generated by server-side code.

There are two kinds of cross-site scripting (XSS) vulnerabilities: server-side and client-side. The latter is essentially caused by bugs in the client-side code, while the former is caused by bugs in the server-side code. In this chapter we focus on server-side XSS vulnerabilities (unless specified otherwise, we will use XSS to refer to server-side XSS). XSS vulnerabilities allow attackers to inject client-side scripting code (typically, JavaScript) into the output of web applications. The scripts are then executed by the browser as it renders the page, allowing malicious code to run in the context of the web application. Attackers can leverage XSS attacks to leak sensitive user information, impersonate the victim to perform unwanted actions in the context of the web application, or launch browser exploits.

There has been a significant amount of research effort on eliminating XSS vulnerabilities. The main line of research has focused on sanitizing untrusted input [11, 58, 67, 80, 90, 92, 101, 123, 125, 136, 141, 143, 145, 146]. Sanitization attempts to identify and "clean up" untrusted inputs that might contain JavaScript code. Performing correct sanitization is challenging, for a number of reasons. One reason is that it is difficult to guarantee coverage for all possible paths through the application [11, 143]. As part of this problem, it is necessary to find all program locations (sources) where untrusted input can enter the application, and then verify, along all program paths, the correctness of all sanitization functions that are used before the input is sent to the client (sinks). Furthermore, it is not always clear how to properly sanitize data, because a single input might appear in different contexts in the output of the application [125].

The root cause of XSS vulnerabilities is that *the current web application model violates the principle of code and data separation*. In the case of a web page, the data is the HTML content of the page and the code is the JavaScript code. Mixing JavaScript code and HTML data in the same channel (the HTTP response) makes it possible for an attacker to convince a user's browser to interpret maliciously crafted HTML data as JavaScript code. While sanitization tries to turn untrusted input, which could potentially contain code, into HTML data, we believe the fundamental solution to XSS is to separate the code and data in a web page—the way HTML and JavaScript should have been designed from the start. Once the code and data are separated, a web application can communicate this separation to the browser, and the browser can ensure no code is executed from the data channel. Such communication and enforcement is supported by the new W3C browser standard Content Security Policy (CSP) [131].

While new web applications can be designed with code and data separated from the start, it has been a daunting task to achieve code and data separation for legacy applications. The key challenge is to identify code or data in the output of a web application. Previous solutions have relied on either developers' manual annotations or dynamic analysis. For example, BEEP [77] requires developers to manually identify inline JavaScript code. BLUEPRINT [93] requires developers to manually identify the data by specifying which application statements could output untrusted input. XSS-GUARD dynamically identifies application-intended JavaScript code in a web page by comparing it with a shadow web page generated at run time [18]. The main problem preventing these solutions from being adopted is either the significant manual effort required from application developers or the significant runtime performance overhead. In fact, Weinberger et al. [142] showed how difficult it is to manually separate the code and data of a web application.

In this chapter, we present DEDACOTA, the first system that can automatically and statically rewrite an existing web application to separate code and data in its web pages.

Our novel idea is to use static analysis to determine all inline JavaScript code in the web pages of an application. Specifically, DEDACOTA performs static data-flow analysis of a given web application to approximate its HTML output. Then, it parses each page's HTML output to identify inline JavaScript code. Finally, it rewrites the web application to output the identified JavaScript code in a separate JavaScript file.

The problem of statically determining the set of (HTML) outputs of a web application is undecidable. However, as we observe in our evaluation, the problem is typically tractable for real-world web applications. These applications are written by benign developers and tend to have special properties that allow us to compute their outputs statically. For instance, the majority of the inline JavaScript code is static in the web applications we tested.

Dynamic inline JavaScript presents a second-order problem. Here, the JavaScript code itself (rather than the HTML page) is generated dynamically on the server and may depend on untrusted inputs. Again, the potential for XSS vulnerabilities exists. DEDACOTA provides a partial solution to this problem by producing alerts for all potentially dangerous instances of dynamic JavaScript generation in the application and by safely sanitizing a large subclass of these instances.

We implemented a prototype of DEDACOTA to analyze and rewrite ASP.NET [98] web applications. We applied DEDACOTA to six open-source, real-world ASP.NET applications. We verified that all known XSS vulnerabilities are eliminated. We then performed extensive testing to ensure that the rewritten binaries still function correctly. We also tested DEDACOTA's performance and found that the page loading times between the original and rewritten application are indistinguishable.

The main contributions of this chapter are the following:

- A novel approach for automatically separating the code and data of a web application using static analysis (Section 6.3).
- A prototype implementation of our approach, DEDACOTA, applied to ASP.NET applications (Section 6.4).
- An evaluation of DEDACOTA, showing that we are able to apply our analysis to six real-world, open-source, ASP.NET applications. We show that our implementation prevents the exploitation of know vulnerabilities and that the semantics of the application do not change (Section 6.5).

6.1 Background

In this section, we provide the background necessary for understanding the design of DEDACOTA.

6.1.1 Cross-Site Scripting

Modern web applications consist of both server-side and client-side code. Upon receiving an HTTP request, the server-side code, which is typically written in a server-side language, such as PHP or ASP.NET, dynamically generates a web page as a response, based on the user input in the request or data in a backend database. The client-side code, which is usually written in JavaScript and is executed by the browser, can be either inline in the web page or external as a standalone JavaScript file.

Cross-site scripting (XSS) vulnerabilities allow an attacker to inject malicious JavaScript into web pages to execute in the client-side browser, as if they were generated by the trusted web site. If the vulnerability allows the attacker to store malicious JavaScript on the server (e.g., using the contents of a message posted on a newsgroup), the vulnerability is traditionally referred to as “stored” or “persistent XSS.” When the malicious code is included in the request and involuntarily reflected to the user (copied into the response) by the server, the vulnerability is called “reflected XSS.” Finally, if the bug is in the client-side code, the XSS vulnerability is referred to as “DOM-based XSS” [86]. We call the first two types of vulnerabilities “server-side XSS vulnerabilities” and the latter “client-side XSS vulnerabilities.”

The root cause for server-side XSS is that the code (i.e., the client-side script) and the data (i.e., the HTML content) are mixed together in a web page. By crafting some malicious input that will be included into the returned web page by the server-side code, an attacker can trick the browser into confusing his data as JavaScript code.

6.1.2 Code and Data Separation

The separation of code and data can be traced back to the Harvard Architecture, which introduces separate storage and buses for code and data. Separating code and data is a basic security principle for avoiding code injection attacks [69]. Historically, whenever designs violate this principle, there exists a security hole. An example is the stack used in modern CPUs. The return addresses (code pointers) and function local variables (data) are co-located on the stack. Because the return addresses determine control transfers, they are essentially part of the code. Mixing them together with the data allows attackers to launch stack overflow attacks, where data written into a local variable spills into an adjacent return address. In the context of web applications, we face the same security challenge, this time caused by mixing code and data together in web pages. To fundamentally solve this problem, we must separate code and data in web pages created by web applications.

6.1.3 Content Security Policy

Content Security Policy (CSP) [131] is a mechanism for mitigating a broad class of content injection vulnerabilities in web applications. CSP is a declarative policy that allows a web application to inform the browser, via an HTTP header, about the sources from which the application expects to load resources such as JavaScript code. A web browser that implements support for CSP can enforce the security policy declared by the web application.

A newly developed web application can leverage CSP to avoid XSS by not using inline JavaScript and by specifying that only scripts from a set of trusted sites are allowed to execute on the client. Indeed, Google has required that all Chrome browser extensions implement CSP [3]. However, manually applying CSP to a legacy web application typically requires a non-trivial amount of work [142]. The reason is that the authors of the web application have to modify the server-side code to clearly identify which resources (e.g., which JavaScript programs) are used by a web page. Moreover, these scripts have to be separated from the web page.

CSP essentially provides a mechanism for web browsers to enforce the separation between code and data as specified by web applications. Our work solves the problem of automatically transforming legacy web applications so that the code and data in their web pages are separated. The transformed web applications can then directly leverage the browser's support for CSP to avoid a wide range of XSS vulnerabilities.

6.2 Threat Model

Before discussing the design of DEDACOTA, we need to state our assumptions about the code that we are analyzing and the vulnerabilities we are addressing.

Our approach involves rewriting a web application. This web application is written by a benign developer—that is, the developer has not intentionally obfuscated the code as a malicious developer might. This assumption also means that the JavaScript and HTML are benign and not intentionally taking advantage of browser parsing quirks (as described in BLUEPRINT [93]).

DEDACOTA will only prevent *server-side* XSS vulnerabilities. We define server-side XSS vulnerabilities as XSS vulnerabilities where the *root cause* of the vulnerability is in server-side code. Specifically, this means XSS vulnerabilities where unsanitized input is used in an HTML page. We explicitly do not protect against client-side XSS vulnerabilities, also called DOM-based XSS. Client-side XSS vulnerabilities occur when untrusted input is interpreted as JavaScript by the client-side JavaScript code using methods such as `eval`, `document.write`, or `innerHTML`. The root cause of these vulnerabilities is in the *JavaScript code*.

In this chapter, we focus solely on separating inline JavaScript code (that is, JavaScript in between `<script>` and `</script>`). While there are other vectors where JavaScript can be executed, such as JavaScript code in HTML attributes (event handlers such as `onclick`) and inline Cascading Style Sheet (CSS) styles [64], the techniques described here can be extended to approximate and rewrite the HTML attributes and inline CSS.

Unfortunately, code and data separation in an HTML page is not a panacea for XSS vulnerabilities. In modern web applications, the inline JavaScript code is sometimes dynamically generated by the server-side code. A common scenario is to use the dynamic JavaScript code to pass data from the server-side code to the client-side code. There may be XSS vulnerabilities, even if code and data are properly separated, if the data embedded in the JavaScript code is not properly sanitized. DEDACOTA provides a partial solution to the problem of dynamic JavaScript (see Section 6.3.5).

6.3 Design

Our goal is to statically transform a given web application so that the new version preserves the application semantics but outputs web pages where all the inline JavaScript code is moved to external JavaScript files. These external files will be the *only* JavaScript that the browser will execute, based on a Content Security Policy.

There are three high-level steps to our approach. For each web page in the web application: (1) we statically determine a conservative approximation of the page's HTML output, (2) we extract all inline JavaScript from the approximated HTML out-

```
1  <html>
2    <% Title = "Example";
3     Username = Request.Params["name"]; %>
4    <head><tile><%= Title %></title></head>
5    <body>
6      <script>
7        var username = "<%= Username %>";
8      </script>
9    </body>
10 </html>
```

Listing 6.1: Example of a simple ASP.NET Web Form page.

put, and (3) we rewrite the application so that all inline JavaScript is moved to external files.

Hereinafter, we define a running example that we use to describe how deDacota automatically transforms a web application, according to the three steps outlined previously.

6.3.1 Example

Listing 6.1 shows a simplified ASP.NET Web Form page. Note that everything not in between the `<%` and `%>` is output directly to the browser. Everything between matching `<%` and `%>` is C# code. A subtle but important point is that `<%=` is used to indicate that the C# code will output a string at that location in the HTML output.

In Listing 6.1, Line 2 sets the title of the page, and Line 3 sets the `Username` variable to the `name` parameter sent in the query string. The `Username` is output to the browser inside a JavaScript string on Line 7. This is an example of the C# server-

```
 1 void Render(TextWriter w) {
 2   w.Write("<html>\n  ");
 3   this.Title = "Example";
 4   this.Username = Request.Params["name"];
 5   w.Write("\n  <head><tile>");
 6   w.Write(this.Title);
 7   w.Write("</title></head>\n  <body>\n    <script>\n      var
        username = \"");
 8   w.Write(this.Username);
 9   w.Write("\";\n    </script>\n  </body>\n</html>");
10 }
```

Listing 6.2: The compiled C# output of Listing 6.1.

side code passing information to the JavaScript client-side code, as the intent here is for the JavaScript `username` variable to have the same value as the C# `Username` variable.

Internally, ASP.NET compiles the ASP.NET Web Form page to C#, either when the application is deployed, or on-demand, as the page is accessed. The relevant compiled C# output of Listing 6.1 is shown in Listing 6.2. Here, the ASP.NET Web Form page has been transformed into an equivalent C# program. The ASP.NET compiler creates a class (not shown) that represents the ASP.NET Web Form. A method of the class is given a `TextWriter` object as a parameter. Anything written to this object will be sent in the HTTP response. `TextWriter.Write` is a method call equivalent of writing to the console in a traditional command-line application.

From comparing Listing 6.1 to Listing 6.2, one can see that output not between `<%` and `%>` tags is written to the `TextWriter` object. The code between the `<%` and `%>` tags is inlined into the function (Lines 3 and 4), and the code that is between the

`<%=` and `%>` tags is written to the `TextWriter` object (Lines 6 and 8). We also note that `TextWriter.Write` is one of a set of methods used to write content to the HTTP response. However, for simplicity, in the remainder of this chapter, we will use `TextWriter.Write` to represent all possible ways of writing content to the HTTP response.

6.3.2 Approximating HTML Output

In the first phase of our approach, we approximate the HTML output of a web page. This approximation phase is a two-step process. First, we need to determine, at every `TextWriter.Write` location, what is being written. Second, we need to determine the order of the `TextWriter.Write` function invocations.

We use a different static analysis technique to answer each of the two questions. To determine what is being written at a `TextWriter.Write`, we use the points-to analysis algorithm presented in [38] modified to work on .NET byte-code, instead of C. This points-to analysis algorithm is inclusion-based, demand-driven, context-sensitive, field-sensitive, and partially flow-sensitive. The points-to analysis algorithm computes the set of strings that alias with the parameter of `TextWriter.Write`. If all strings in the alias set are constant strings, the output at the `TextWriter.Write` will be defined as the conjunction of all possible constant strings. Otherwise, we say the output is statically undecidable. To determine the ordering of all `TextWriter.Write` method

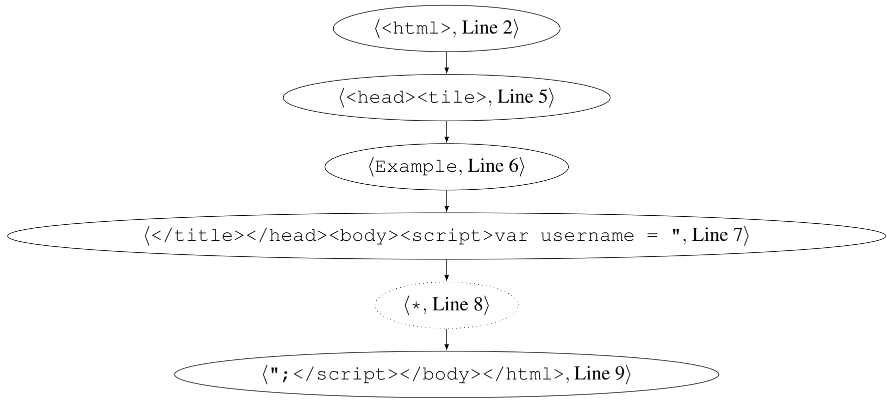
⟨<html>, Line 2⟩
⟨<head><tile>, Line 5⟩
⟨Example, Line 6⟩
⟨</title></head><body><script>var username = ", Line 7⟩
⟨⋆, Line 8⟩
⟨";</script></body></html>, Line 9⟩

Figure 6.1: Approximation graph for the code in Listing 6.1 and Listing 6.2. The dotted node's content is not statically determinable.

calls, we build a control-flow graph, using standard techniques, that only contains the `TextWriter.Write` method calls.

We encode the information produced by the two static analyses—the ordering of `TextWriter.Write` method calls and their possible output—into a graph that we call an *approximation graph.* Figure 6.1 shows the approximation graph for the code in Listing 6.1 and Listing 6.2. Each node in the graph contains the location of the `TextWriter.Write` that this node represents as well as the possible constant strings that could be output at this `TextWriter.Write` location. Content that cannot be determined statically is represented by a wild card $\star$ (the dotted node in Figure 6.1). The strings that may be output at the `TextWriter.Write` will be used to identify inline JavaScript, and the location of the `TextWriter.Write` will be used for rewriting the application.

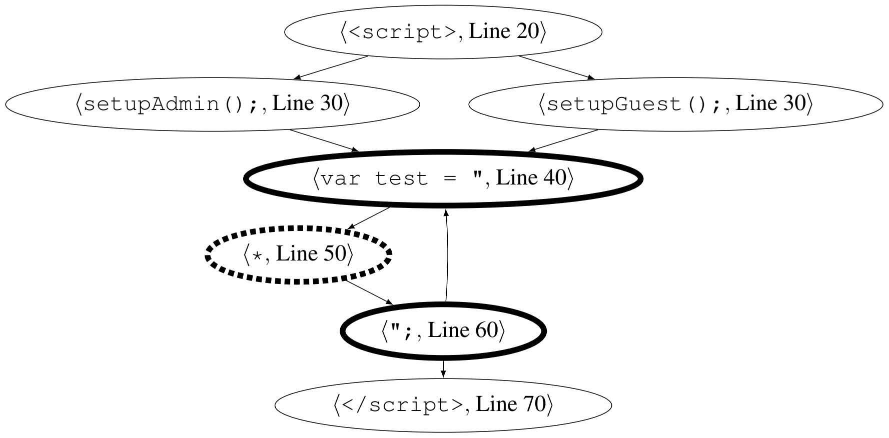

Figure 6.2: Approximation graph with branches and a loop. The loop will be collapsed into one node to create the final approximation graph.

In Figure 6.2 we show the approximation graph of a more complex page. The graph in Figure 6.2 contains a branch, where each node in the branch maps to the same `TextWriter.Write` method. This happens when the points-to analysis says that the `TextWriter.Write` method can output one of multiple strings. The other way there can be a branch in the approximation graph is when there is a branch in the control flow of the web application. The graph in Figure 6.2 also contains a loop that includes the nodes shown in bold. However, because we cannot statically determine the number of times a loop may execute, and we want our analysis to be conservative, we collapse all nodes of a loop (in the approximation graph) into a single node. This new node now has undecidable content (represented by a $\star$). The new node also keeps track of all the `TextWriter.Write` methods that were part of the original loop.

After collapsing all loops in the graph, we derive a conservative approximation of the HTML output of a web page. The approximation graph is a directed acyclic graph (DAG), and any path from the root node to a leaf node will represent one possible output of the web page.

6.3.3 Extracting Inline JavaScript

In the second phase, our approach uses the approximation graph described previously to extract all possible inline JavaScript. The output of this phase is a set containing all possible inline JavaScript that may appear in the web page.

In an approximation graph, each unique path from the root node to a leaf node represents a potential output of the page. A naïve algorithm would enumerate all paths and, thus, all outputs, and parse each output string to identify inline JavaScript. However, even without loops, the number of unique paths even in a simple web page may quickly explode and become unmanageable (this is the path-explosion problem faced in static analysis).

To reduce the impact of the path explosion problem, we extract the inline JavaScript directly from the approximation graph. We first search for the opening and closing tags of HTML elements in the graph. We ignore tags that appear in comments. Then, for each pair of JavaScript tags (i.e., `<script>` and `</script>`), we process all the unique paths between the opening and closing tags. For each path, we obtain an inline JavaScript that the program might output.

While our current prototype is relatively simplistic in parsing the starting and ending JavaScript files, it could be possible to use the parsing engine from a real browser. However, this is not as straight-forward as it seems, as our input is a graph of all potential HTML output, not a single document. We leave this approach to future work.

All identified inline JavaScript pieces are then passed to the last phase of our approach, which decides how to rewrite the application.

6.3.4 Application Rewriting

The goal of the third phase is to rewrite the application so that all identified inline JavaScript will be removed from the HTML content and saved in external JavaScript files. In the HTML code, an inline JavaScript is replaced with a reference to the external JavaScript file as follows:

```
<script src="External.js"></script>
```

It is not uncommon that multiple possible inline JavaScript snippets exist between an opening and closing JavaScript tag because there may be branches between the tags in the approximation graph. To know which exact inline JavaScript is created, we need to track the execution of the server-side code.

The inline JavaScript identified in the previous phase falls into two categories: static and dynamic (i.e., contains undecidable content). Because we cannot statically decide the content of a dynamic inline JavaScript, we must track the execution of the server-side code to create its external JavaScript file(s) at runtime. Therefore, we can avoid tracking the execution of the server-side code *only* for the case in which there is a *single, static* inline JavaScript code.

For a pair of opening and closing script tags that require tracking the execution of the server-side code, we rewrite the application in the following way. At the `TextWriter.Write` that may output the opening script tag, we first check if the output string contains the tag. We must perform this check because a `TextWriter.Write` site may be used to output either inline JavaScript code or other HTML. If we find the opening script tag in the output, we use a session flag to indicate that an inline JavaScript rewriting has started. We write out everything before the start of the opening script tag. We remove the opening script tag itself. The remaining content is stored into a session buffer. Note that both session flag and buffer are unique to each opening script tag. Then, for all subsequent `TextWriter.Write` method calls that are part of the inline JavaScript we are rewriting, except for the last (that writes the closing tag), we append their output to the session buffer if the session flag is on. For the last `TextWriter.Write` method call (i.e., the one that writes the closing script tag), any string content that occurs before the closing script tag is appended to the session

```
w.Write("</title></head>\n  <body>\n   ");

Session["7"] = "\n    var username = \"");
Session["7"] += this.Username;
Session["7"] += "\";\n    ";

var hashName = Hash(Session["7"]) + ".js";
WriteToFile(hashName, Session["7"]);

w.Write("<script src=\"" + hashName + "\"></script>");

w.Write("\n  </body>\n</html>");
```

Listing 6.3: The result of the rewriting algorithm applied to Listing 6.2. Specifically, here we show the transformation of Lines 7–9 in Listing 6.2.

buffer. Any content after the closing script tag is just written to the output. At this point, the session buffer contains the entire inline JavaScript code. We save this code to an external file and add a `TextWriter.Write` method call that outputs the reference to this JavaScript file.

To support JavaScript caching on the client side, the name of the JavaScript file is derived from its content, using a cryptographic hash of the JavaScript content. An unintended benefit of this approach is that inline JavaScript that is included on multiple pages will be cached by the browser, improving application performance by reducing the size of the page and saving server requests.

Listing 6.3 shows the result of applying this rewriting process to the inline JavaScript code in Listing 6.2. The changes shown are only those made to Lines 7–9 in Listing 6.2.

6.3.5 Dynamic Inline JavaScript

At this point in our analysis, we have successfully separated the JavaScript code from the HTML data in the web application. If the web application's JavaScript is static, and by static we mean statically decidable, then the application is now immune to XSS vulnerabilities. However, if the web application dynamically generates JavaScript with undecidable content, and that content is not properly sanitized inside the JavaScript code, an attacker can exploit this bug to inject a malicious script. The approach discussed so far does not mitigate this attack, because it simply moves the vulnerable JavaScript to an external file.

To understand how dynamic JavaScript can result in a vulnerability, consider our example application in Listing 6.2. There is an XSS vulnerability on Line 8 because the `Username` variable is derived from the `name` parameter and output directly to the user, without sanitization. An attacker could exploit this vulnerability by setting the `name` parameter to `";alert('xss')//`. This would cause the resulting inline JavaScript to be the following, thus executing the attacker's JavaScript code:

```
<script>
  var username = "";alert('xss')//";
</script>
```

Therefore, the code section of the application is dynamically generated with untrusted input and even with the code and data separated, there is still an XSS vulnerability.

We attempt to mitigate this problem, and therefore improve the security of the application, in two ways. First, we identify cases in which we can safely rewrite the application. Second, we notify the developer when we make an inline to external transformation that is potentially unsafe.

For the first case, when the undetermined output is produced in certain JavaScript contexts, we can include it in a safe fashion via sanitization. Specifically, during static analysis we pass the dynamic inline JavaScript to a JavaScript parser. Then, we query the parser to determine the contexts in which the undetermined output (i.e., the $\star$ parts) is used. Here, for context we are referring specifically to the HTML parsing contexts described by Samuel et al. [123]. Possible contexts are JavaScript string, JavaScript numeric, JavaScript regular expression, JavaScript variable, etc. If an undetermined output is in a string context, we sanitize them in a way similar to how BLUEPRINT [93] handles string literals in JavaScript.

Like BLUEPRINT, on the server side we encode the string value and store the encoded data in JavaScript by embedding a call to a decoding function. Then when the JavaScript is executed on the client side, the decoding function will decode the encoded data and return the string. Unlike BLUEPRINT, we do not require any developer annotations because our static analysis can automatically identify which JavaScript context an undetermined output is in.

6.3.6 Generality

While the description of our approach so far was specific to ASP.NET Web Forms, the high-level idea of automatically separating code and data in a legacy web application can be generalized to any other web application frameworks or templating languages. There are still challenges that remain to apply our approach to another language, or even another template in the same language. The two main steps of our approach that must be changed to accommodate a different language or templating language are: (1) understand how the output is created by the web application and (2) understand how to rewrite the web application. Only the first step affects the analysis capability (as the rewriting process is fairly straightforward).

To automatically separate the code and data of a different language or templating language, one must understand how the language or template generates its output. After that, one would need to implement a static analysis that can create an approximation graph. For instance, in the default Ruby on Rails template, ERB, variables are passed to the template either via a hash table or class instance variables [120]. Therefore, one could approximate the output of an ERB template by statically tracking the variables added to the hash table and class instance variables (using points-to analysis). Once an approximation graph is created, detecting inline JavaScript can be performed in the manner previously described.

The main factor to affect the success of applying our approach to another web application framework or templating language is the precision of the static analysis, or in other words, how precise and detailed the approximation graph would be. The more dynamicism in the language or framework, such as run-time code execution and dynamic method invocation, the more difficult the analysis will be. Simply, the more of the control-flow graph that we are able to determine statically, the better our analysis will be. As an example the default templating language in Django only allows a subset of computation: iterating over a collection instead of arbitrary loops [44]. This restriction could make the analysis easier and therefore the approximation graph more precise.

6.4 Implementation

We implemented the automated code and data separation approach described in Section 6.3 in a prototype called DEDACOTA. This prototype targets ASP.NET Web Forms applications. ASP.NET is a widely used technology; of the Quantcase top million websites on the Internet, 21.24% use ASP.NET [25].

DEDACOTA targets *binary* .NET applications. More precisely, it takes as input ASP.NET Web Forms binary web applications, performs the three steps of our approach, and outputs an ASP.NET binary that has all inline JavaScript code converted into external JavaScript files. We operate at the binary level because we must be able to analyze the ASP.NET system libraries, which are only available in binary form.

We leverage the open-source Common Compiler Infrastructure (CCI) [99] for reading and analyzing the .NET Common Language Runtime byte-code. CCI also has modules to extract basic blocks and to transform the code into single static assignment (SSA) form. We also use CCI to rewrite the .NET binaries.

For the static analysis engine, we leverage the points-to analysis engine of KOP (also known as MAS) [38]. KOP was originally written for the C programming language. Therefore, we wrote (using CCI) a frontend that processes .NET binaries and outputs the appropriate KOP points-to rules. Then, after parsing these rules, the static analysis engine can provide either alias analysis or points-to analysis. The KOP points-to analysis is demand-driven, context-sensitive, field-sensitive, and, because of the CCI single static assignment, partially flow-sensitive.

An important point, in terms of scalability, is the demand-driven ability of the static analysis engine. Specifically, we will only explore those parts of the program graph that are relevant to our analysis, in contrast to traditional data-flow techniques which track data dependencies across the entire program. The demand-driven nature of the static analysis engine offers another scalability improvement, which is parallelism. Each analysis query is independent and, therefore, can be run in parallel.

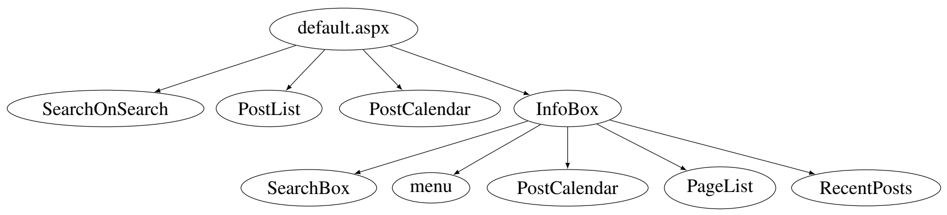

Figure 6.3: Control parent-child relationship between some of the controls in `default.aspx` from the application BlogEngine.NET. The siblings are ordered from left to right in first-added to last-added order.

We also extend the KOP points-to analysis system to model string concatenation. We do this by including special edges in the program graph that indicate that a variable is the result of the concatenation of two other variables. When computing the alias set of a variable, we first do so in the original way (ignoring any concatenation edges). Then, for each variable in the alias set that has concatenation edges, we compute the alias set for each of the two variables involved in the concatenation operation. We concatenate strings in the two alias sets and add them to the original alias set. The undecidable variables are tracked, so that their concatenated result contains a wildcard. This process is recursive, and handles arbitrary levels of concatenation.

ASP.NET uses the idea of reusable components, called `Controls`. The idea is that a developer can write a control once and then include it in other pages, and even other controls. This relationship of including one control inside another creates a parent-child relationship between the controls (the parent being the control that contains the child control).

In an ASP.NET Web Form, child controls are first added to the parent's `Child-Controls` collection, which is similar to an array. Then, during rendering, a parent renders its child controls either by iterating over the `ChildControls` or by referencing a child control based on its index in the `ChildControls`. Because the KOP points-to analysis does not model the array relation, we cannot precisely decide which child Control is being selected during rendering. To handle this problem, we need to track the parent-child relationships directly.

These parent-child relationships form a tree. Figure 6.3 shows the parent-child relationship of some of the user controls of `default.aspx` in the application BlogEngine.NET (one of the programs used in our evaluation). When building the control graph, we must statically recreate this tree.

To create this relationship statically, we take an approach similar to approximating the HTML output. The entry function for an ASP.NET page is `Framework-Initialize`, which is similar to the `main` function for a C program. Starting from this method, we create a control-flow graph of all calls to `AddParsedSubObject`, which is the function that adds a child control to a parent. This gives us the order of the `AddParsedSubObject` calls. At each of the calls, we use the points-to analysis to find which control is the parent and which is the child. This information, along with the order of the calls to `AddParsedSubObject`, allows us to recreate the parent-child control tree.

Application	Version	Known Vuln.	# Web Forms	# Controls	ASP LOC	C# LOC	Total LOC
BugTracker.NET	3.4.4	CVE-2010-3266	115	0	27,257	8,417	35,674
BlogEngine.NET	1.3	CVE-2008-6476	19	11	2,525	26,987	29,512
BlogSA.NET	1.0 Beta 3	CVE-2009-0814	29	26	2,632	4,362	6,994
ScrewTurn Wiki	2.0.29	CVE-2008-3483	30	4	2,951	9,204	12,155
WebGoat.NET	e9603b9d5f	2 Intentional	67	0	1,644	10,349	11,993
ChronoZoom	Beta 3	N/A	15	0	3,125	18,136	21,261

Table 6.1: ASP.NET Web Form applications that we ran DEDACOTA on to test its applicability to real-world web applications.

6.5 Evaluation

There are three properties that we must look at to evaluate the effectiveness of DEDACOTA. First, do we prevent XSS vulnerabilities in the data section of the application by applying code and data separation? Second, do we correctly separate the code and data of the application—that is, does the rewriting preserve the application's semantics? Third, what is the impact on the application's performance? To evaluate the security of our approach, we look at ASP.NET applications with known vulnerabilities. To evaluate the correctness of our rewriting procedure, we apply our approach to applications that have developer-created integration tests. Then, we carried out performance measurements to answer the third question. Finally, we discuss the relation between separating code and data in the output and sanitizing the input.

6.5.1 Applications

We wish to evaluate DEDACOTA on ASP.NET web applications that are real-world, are open-source, and contain known vulnerabilities. Real-world applications are important for showing that our approach works on real-world code, open-source is important for other researchers to replicate our results, and known-vulnerable is important because we aim to automatically prevent these known vulnerabilities.

Unfortunately, there is no standard (or semi-standard) ASP.NET web application benchmark that meets all three requirements. Furthermore, finding these application proved to be a challenge. Compared to other languages such as PHP, there are fewer open-source ASP.NET applications (as most ASP.NET applications tend to be proprietary). Therefore, here we present a benchmark of six real-world, open-source, ASP.NET applications, four of which are known-vulnerable, one of which is intentionally vulnerable for education, and one of which has a large developer-created test suite.

Table 6.1 contains, for each application, the version of the application used in our evaluation, the CVE number of the vulnerability reported for the application, the number of ASP.NET Web Form pages, and the number of developer-written ASP.NET `Controls`. To provide an idea of the size of the applications, we also show the number of lines of code (LOC) of the ASP.NET controls (Web Forms and Controls) and C# code.

The open-source web applications BugTracker.NET [24], BlogEngine.NET [19], BlogSA.NET [20], and ScrewTurn Wiki [127] all contain an XSS vulnerability as defined in the associated CVE.

WebGoat.NET [65] is an open-source ASP.NET application that is intentionally vulnerable. The purpose is to provide a safe platform for interested parties to learn about web security. Among the vulnerabilities present in the application are two XSS vulnerabilities.

ChronoZoom Beta 3 [35], is an open-source HTML5 “interactive timeline for all of history.” Parts are written in ASP.NET Web Forms, but the main application is a JavaScript-heavy HTML page. We use ChronoZoom because, unlike the other applications, it has an extensive test suite that exercises the JavaScript portion of the application. To evaluate the correctness of our rewriting, we converted the main HTML page of ChronoZoom, which contained inline JavaScript, into an ASP.NET Web Form page, along with nine other HTML pages that were used by the test suite.

These six real-world web applications encompass the spectrum of web application functionality that we expect to encounter. These applications constitute a total of 100,000 lines of code, written by different developers, each with a different coding style. Some had inline JavaScript in the ASP.NET page, some created inline JavaScript in C# directly, while others created inline JavaScript in C# using string concatenation. Furthermore, while analyzing each application we also analyzed the entire .NET framework (which includes ASP.NET); all 256 MB of binary code. As our analysis handles ASP.NET, we are confident that our approach can be applied to the majority of ASP.NET applications.

6.5.2 Security

We ran DEDACOTA on each of our test applications. Table 6.2 shows the total number of inline JS scripts per application and a breakdown of the number of static inline JS scripts, the number of safe dynamic inline JS scripts, and the number of unsafe dynamic inline JS scripts. There were four dynamic inline JS scripts created by the ASP.NET framework, and these are represented in Table 6.2 in parentheses. We chose to exclude these four from the total dynamic inline JS scripts because they are not under the developer's control, and, furthermore, they can and should be addressed by changes to the ASP.NET library. Furthermore, it is important to note that our tool found these dynamic inline JS scripts within the ASP.NET framework automatically.

From our results it is clear that modern web applications frequently use inline JS scripts. The applications used a range of five to 46 total inline JS scripts. Of these total inline JS scripts 22% to 100% of the inline JS scripts were static.

DEDACOTA was able to safely transform, using the technique outlined in Section 6.3.5, 50% to 70% of the dynamic inline JS scripts. This result means that our mitigation technique worked in the majority of the cases, with only zero to four actual unsafe dynamic inline JS scripts per application.

We looked for false negatives (inline JavaScript that we might have missed) in two ways. We manually browsed to every ASP.NET Web Form in the application and looked for inline JavaScript. We also searched for inline JavaScript in the original source code of the application to reveal possible scripts the previous browsing might have missed. We did not find any false negatives in the applications.

To evaluate the security improvements for those applications that had known vulnerabilities, we manually crafted inputs to exploit these know bugs. After verifying that the exploits worked on the original version of the application, we launched them against the rewritten versions (with the Content Security Policy header activated, and with a browser supporting CSP). As expected, the Content Security Policy in the browser, along with our rewritten applications, successfully blocked all exploits.

6.5.3 Functional Correctness

To evaluate the correctness of our approach, and to verify that we maintained the semantics of the original application, we used two approaches. First, we manually browsed web pages generated by each rewritten application and interacted with the web site similar to a normal user. During this process, we looked for JavaScript errors, unexpected behaviors, or CSP violations. We did not find any problems or deviations. Second, and more systematically, we leveraged the developer-written testing suite in ChronoZoom. Before we applied our rewriting, the original application passed 160 tests. After rewriting, all 160 tests executed without errors.

Application	Total JS	Static	Safe Dynamic	Unsafe Dynamic
BugTracker.NET	46	41	3	2 (4)
BlogEngine.NET	18	4	10	4 (4)
BlogSA.NET	12	10	1	1 (4)
ScrewTurn Wiki	35	27	4	4 (4)
WebGoat.NET	6	6	0	0 (4)
ChronoZoom	5	5	0	0 (4)

Table 6.2: Results of running DEDACOTA against the ASP.NET Web Form applications. Safe Dynamic is the number of dynamic inline JS scripts that we could safely transform, and Unsafe Dynamic is the number of dynamic inline JS scripts that we could not safely transform.

Application	Page Size	Loading Time
ChronoZoom (original)	50,827	0.65
ChronoZoom (transformed)	20,784	0.63
BlogEngine.NET (original)	18,518	0.15
BlogEngine.NET (transformed)	19,269	0.16

Table 6.3: Performance measurements for two of the tested applications, ChronoZoom. Page Size is the size (in bytes) of the main HTML page rendered by the browser, and Loading Time is the time (in seconds) that the browser took to load and display the page.

6.5.4 Performance

To assess the impact of DEDACOTA on application performance, we ran browser-based tests on original and transformed versions of two of the tested applications. Our performance metric was page-loading time in Internet Explorer 9.0, mainly to determine the impact of moving inline JavaScript into separate files. The web server was a 3 GB Hyper-V virtual machine running Microsoft IIS 7.0 under Windows Server 2008 R2, while the client was a similar VM running Windows 7. The physical server was an 8 GB, 3.16 GHz dual-core machine running Windows Server 2008 R2.

Table 6.3 shows test results for two web applications, summarizing performance data from page-loading tests on the client. The table columns list the average sizes of the main HTML pages retrieved by the browser by accessing the main application URLs, along with the average time used by the browser to retrieve and render the pages in their entirety. All the numbers were averaged over 20 requests.

As Table 6.3 indicates, DEDACOTA's transformations incurred no appreciable difference in page-loading times. Because the original ChronoZoom page contained a significant amount of script code, the transformed page is less than half of the original size. On the other hand, the BlogEngine.NET page is slightly larger because of its small amount of script code, which was replaced by longer links to script files. The page-loading times mirror the page sizes, also indicating that server-side processing incurred no discernible performance impact.

6.5.5 Discussion

The results of our rewriting shed light on the nature of inline JavaScript in web applications. Of the four applications that have dynamic JavaScript, 12.2% to 77.8% of the total inline JavaScript in the application is dynamic. This is important, because one of BEEP's XSS prevention policies is a whitelist containing the SHA1 hash of allowed JavaScript [77]. Unfortunately, in the modern web JavaScript is not static and frequently includes dynamic elements, necessitating new approaches that can handle dynamic JavaScript.

The other security policy presented in BEEP is DOM sandboxing. This approach requires the developer to manually annotate the sinks so that they can be neutralized. BLUEPRINT [93] works similarly, requiring the developer to annotate the outputs of untrusted data. Both approaches require the developer to manually annotate the sinks in the application in order to specify the trusted JavaScript. To understand the developer effort required to manually annotate the sinks in the application, we counted the sinks (i.e., `TextWriter.Write` call sites) inside the 29 Web Forms of BlogSA.NET and there were 407. In order to implement either BEEP or BLUEPRINT a developer must manually analyze all sinks in the application and annotate any that could create untrusted output.

Unlike BEEP and BLUEPRINT, DEDACOTA is completely automatic and does not require any developer annotations. DEDACOTA cannot prevent XSS vulnerabilities in dynamic inline JavaScript completely. If a developer wishes to prevent all XSS vulnerabilities after applying DEDACOTA, they would only need to examine the sinks that occur *within* the unsafe dynamic inline JavaScript. In BlogSA.NET, there are three sinks within the single unsafe dynamic JavaScript. One could further reduce the number of sinks by using taint analysis to check if untrusted input can reach a sink in the dynamic JavaScript.

6.6 Limitations

The goal of deDacota is to automatically separate the JavaScript code from the HTML data in the web pages of a web application using static analysis. We have shown that deDacota is effective with real-world web applications. In this section, we discuss its limitations in general.

The programming language of .NET has the following dynamic language features: dynamic assembly loading, dynamic compilation, dynamic run-time method calling (via reflection), and threading. The use of these features may compromise the soundness of any static analysis including ours in deDacota. However, these language features are rarely used in ASP.NET web applications in practice. For instance, those applications we tested did not use any of these features. Furthermore, deDacota is affected only if the use of these features determines the HTML output of an application.

On one hand, we handle loops conservatively by approximating that a loop can produce anything. On the other hand, we treat the output of a loop as a $\star$ in the approximation graph and assume it does not affect the structure of the approximation graph in a way that impacts our analysis. For instance, we assume the output of a loop does not contain the opening or closing script tag. Our analysis will be incorrect if this assumption is violated. While we found that this assumption holds for all the web applications we tested, it is possible that this assumption will not hold for other programs, thus requiring a different approach to handling loops.

We do not offer any formal proof of the correctness of DEDACOTA. While we believe that our approach is correct in absence of the dynamic language features, we leave a formal proof of this to future work.

DEDACOTA currently supports the analysis of string concatenations. The support for more complex string operations such as regular expressions is left for future work. A potential approach is to leverage an automata-based string analysis engine [146].

Our approach to sanitizing dynamic JavaScript code may not preserve an application's semantics when the dynamic content being sanitized as a string is meant to be used in multiple JavaScript contexts.

When deploying DEDACOTA in practice, we recommend two practices to mitigate its limitations. First, all tests for the original web application should be performed on the rewritten binary to detect any disruptions to the application's semantics. Second, CSP's "Report Only" mode should be used during the testing and initial deployment. Under this mode, the browser will report violations back to the web server when unspecified JavaScript code is loaded. This helps detect inline JavaScript code that is missed by DEDACOTA.

Finally, our prototype does not handle JavaScript code in HTML attributes. We do not believe that there is any fundamental limitation that makes discovering JavaScript attributes more difficult than inline JavaScript. The only minor difficulty here is in the rewriting. In order to separate a JavaScript attribute into an external JavaScript, one must be able to uniquely identify the DOM element that the JavaScript attribute affects. To do this, it would require generating a unique identifier for the HTML element associated with the JavaScript attribute.

6.7 Conclusion

Cross-site scripting vulnerabilities are pervasive in web applications. Malicious users frequently exploit these vulnerabilities to infect users with drive-by downloads or to steal personal information.

While there is currently no silver bullet to preventing every possible XSS attack vector, we believe that adhering to the fundamental security principle of code and data separation is a promising approach to combating XSS vulnerabilities. DEDACOTA is a novel approach that gets us closer to this goal, by using static analysis to automatically separate the code and data of a web application. While not a final solution, DEDACOTA and other tools that automate making web applications secure by construction are the next step in the fight against XSS and other kinds of vulnerabilities.

Chapter 7

Conclusions

Throughout this dissertation, we have discussed and analyzed the state of web security today. I have proposed new approaches that aim to find vulnerabilities before a malicious attacker has the chance. It is in this vein of preemptively finding vulnerabilities that I believe will have the greatest return-on-investment. By finding vulnerabilities early on in the development process, the vulnerabilities will be easier and cheaper to fix.

In this spirit, for moving forward I see the web security community moving to approaches that create web applications that are secure by construction. Therefore, vulnerabilities can be prevented, just by designing an application in a certain way, or perhaps by creating a new language or framework that is easy to statically analyze. As shown throughout this dissertation, web application vulnerabilities are incredibly prevalent, and show no signs of stopping. In order to counteract this trend, we require novel ideas: new ways of designing applications, new tools to automatically find security vulnerabilities, or new approaches to web applications. The web is too important to wait—we must take responsibly for securing this popular platform.

Bibliography

[1] B. Acohido. Hackers breach Heartland Payment credit card system. *USA TODAY*, Jan. 23, 2009.

[2] Acunetix. Acunetix Web Vulnerbility Scanner. `http://www.acunetix.com/`.

[3] D. Akhawe, P. Saxena, and D. Song. Privilege Separation in HTML5 Applications. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2012.

[4] D. Amalfitano, A. Fasolino, and P. Tramontana. Reverse Engineering Finite State Machines from Rich Internet Applications. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2008.

[5] AnantaSec. Web Vulnerability Scanners Evaluation. `http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html`, Jan. 2009.

[6] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, 2010.

[7] ASP.NET MVC. `http://www.asp.net/mvc`.

[8] E. Athanasopoulos, V. Pappas, and E. P. Markatos. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2009.

[9] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A Solution for the Automated Detection of Clickjacking Attacks. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2010.

[10] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2011.

[11] 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*, 2008.

[12] 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)*, 2007.

[13] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

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

[15] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines using Domains with Equality Tests. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2008.

[16] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. HTML5. `http://www.w3.org/TR/2014/CR-html5-20140204/`, Feb. 2014.

[17] N. Bilton and B. Stelter. Sony Says PlayStation Hacker Got Personal Data. *The New York Times*, Apr. 27, 2011.

[18] P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[19] blogengine.net - an innovative open source blogging platform. `http://www.dotnetblogengine.net`, 2013.

[20] BlogSA.NET. `http://www.blogsa.net/`, 2013.

[21] B. Boe. UCSB's International Capture The Flag Competition 2010 Challenge 6: Fear The EAR. `http://cs.ucsb.edu/~bboe/r/ictf10`, Dec. 2010.

[22] B. Boe. Using StackOverflow's API to Find the Top Web Frameworks. `http://cs.ucsb.edu/~bboe/r/top-web-frameworks`, Feb. 2011.

[23] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[24] BugTracker.NET - Free Bug Tracking. `http://ifdefined.com/bugtrackernet.html`, 2013.

[25] Top in Frameworks - Week beginning Jun 24th 2013. `http://trends.builtwith.com/framework`, 2013.

[26] D. Byrne. Grendel-Scan. `http://www.grendel-scan.com/`.

[27] Include exit with a redirect call. `http://replay.web.archive.org/20061011152124/https://trac.cakephp.org/ticket/1076`, Aug. 2006.

[28] docs should mention redirect does not "exit" a script. `http://replay.web.archive.org/20061011180440/https://trac.cakephp.org/ticket/1358`, Aug. 2006.

[29] Cake Software Foundation, Inc. The CakePHP 1.3 Book. `http://book.cakephp.org/view/982/redirect`, 2011.

[30] L. Carettoni and S. Di Paola. HTTP Parameter Pollution. OWASP AppSec Europe 2009, May 2009.

[31] A. Chaudhuri and J. Foster. Symbolic Security Analysis of Ruby-on-Rails Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[32] N. Childers, B. Boe, L. Cavallaro, L. Cavedon, M. Cova, M. Egele, and G. Vigna. Organizing large scale hacking competitions. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2010.

[33] Chinotec Technologies. Paros. `http://www.parosproxy.org/`.

[34] S. Chong, K. Vikram, and A. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2007.

[35] Chronozoom - A Brief History of the World. `http://chronozoom.cloudapp.net/firstgeneration.aspx`, 2013.

[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)*, 2007.

[37] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–37, 2008.

[38] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2012.

[39] M. Curphey and R. Araujo. Web Application Security Assessment Tools. *IEEE Security and Privacy*, 4(4):32–41, 2006.

[40] CVE. Common Vulnerabilities and Exposures. `http://www.cve.mitre.org`.

[41] CVE Details. Vulnerabilities by Type. `http://www.cvedetails.com/vulnerabilities-by-types.php`, 2013.

[42] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. WARE: a tool for the Reverse Engineering of Web applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2002.

[43] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[44] Django. `http://djangoproject.com`, 2013.

[45] Django Software Foundation. Django shortcut functions. `http://docs.djangoproject.com/en/dev/topics/http/shortcuts/#django.shortcuts.redirect`, 2011.

[46] Ecma International. ECMAScript: A general purpose, cross-platform programming language. `http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf`, June 1997.

[47] EllisLab, Inc. CodeIgniter User Guide Version 2.0.2. http://codeigniter.com/user_guide/helpers/url_helper.html, 2011.

[48] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2010.

[49] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, June 1999.

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 Header Field Definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.30, June 1999.

[51] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 Status Code Definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, June 1999.

[52] M. Fossi. Symantec Global Internet Security Threat Report. Technical report, Symantec, Apr. 2009. Volume XIV.

[53] Foundstone. Hacme Bank v2.0. http://www.foundstone.com/us/resources/proddesc/hacmebank.htm, May 2006.

[54] M. Furr, J. hoon (David) An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *Proceedings of the ACM SIGPLAN Dynamic Languages Symposium (DLS)*, 2009.

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

[56] J. J. Garrett. Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php, Feb. 2005.

[57] GitHub. http://github.com.

[58] Google. Google AutoEscape for CTemplate. http://code.google.com/p/ctemplate/.

[59] J. Grossman. Challenges of Automated Web Application Scanning, 2004.

[60] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2009.

[61] W. G. Halfond, S. R. Choudhary, and A. Orso. Penetration Testing with Improved Input Vector Identification. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009.

[62] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.

[63] R. Hansen. Clickjacking. `http://ha.ckers.org/blog/20080915/clickjacking/`, Sept. 2008.

[64] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[65] J. Hoff. WebGoat.NET. `https://github.com/jerryhoff/WebGoat.NET`, 2013.

[66] D. Hofstetter. Don't forget to exit after a redirect. `http://cakebaker.wordpress.com/2006/08/28/dont-forget-to-exit-after-a-redirect/`, Aug. 2006.

[67] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2011.

[68] J. hoon An, A. Chaudhuri, and J. Foster. Static Typing for Ruby on Rails. In *Proceedings of the IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2009.

[69] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, second edition, 2003.

[70] HP. WebInspect. `https://download.hpsmartupdate.com/webinspect/`.

[71] 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)*, 2003.

[72] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. 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)*, 2004.

[73] IBM. AppScan. `http://www-01.ibm.com/software/awdtools/appscan/`.

[74] Indictment in U.S. v. Albert Gonzalez. `http://www.justice.gov/usao/ma/news/IDTheft/Gonzalez,%20Albert%20-%20Indictment%20080508.pdf`, Aug. 2008.

[75] T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience Series on Discrete Mathematics and Optimization. Wiley, 1994.

[76] M. Jewell. Data Theft Believed to Be Biggest Hack. *The Washington Post*, Mar. 29, 2007.

[77] 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)*, 2007.

[78] M. Johns and C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2007.

[79] 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*, 2006.

[80] 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)*, 2006.

[81] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

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

[83] 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)*, 2006.

[84] J. Kirk. BitCoin exchange loses $250,0000 after unencrypted keys stolen. `http://www.computerworld.com/s/article/9230919/BitCoin_exchange_loses_250_0000_after_unencrypted_keys_stolen`, Sept. 5, 2012.

[85] A. Klein. "Divide and conquer": HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. `http://www.packetstormsecurity.org/papers/general/whitepaper/httpresponse.pdf`, 2004.

[86] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, 2005.

[87] D. Kristol and L. Montulli. RFC 2109: HTTP State Management Mechanism. `http://www.w3.org/Protocols/rfc2109/rfc2109`, Feb. 1997.

[88] X. Li and Y. Xue. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[89] X. Li, W. Yan, and Y. Xue. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[90] B. Livshits and S. Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2013.

[91] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

[92] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2005.

[93] M. T. Louw and V. Venkatakrishnan. BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[94] M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2008.

[95] S. McAllister, C. Kruegel, and E. Kirda. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[96] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by Inferring User Interface State Changes. In *Proceedings of the International Conference on Web Engineering (ICWE)*, 2008.

[97] L. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[98] Microsoft. ASP.NET. `http://www.asp.net/`.

[99] Microsoft Research. Common Compiler Infrastructure. `http://research.microsoft.com/en-us/projects/cci/`, 2013.

[100] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-Site Scripting Defense. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2008.

[101] A. Nguyen-tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[102] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2009.

[103] Open Security Foundation. OSF DataLossDB: Data Loss News, Statistics, and Research. `http://datalossdb.org/`.

[104] Open Web Application Security Project (OWASP). OWASP SiteGenerator. `http://www.owasp.org/index.php/OWASP_SiteGenerator`.

[105] Open Web Application Security Project (OWASP). OWASP WebGoat Project. http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[106] Open Web Application Security Project (OWASP). Web Input Vector Extractor Teaser. http://code.google.com/p/wivet/.

[107] Open Web Application Security Project (OWASP). OWASP Top Ten Project. http://www.owasp.org/index.php/Top_10, 2010.

[108] OpenID Foundation. OpenID. http://openid.net/.

[109] C. Ortiz. Outcome of sentencing in U.S. v. Albert Gonzalez. http://www.justice.gov/usao/ma/news/IDTheft/09-CR-10382/GONZALEZ%20website%20info%205-11-10.pdf, Mar. 2010.

[110] PCI Security Standards Council. PCI DDS Requirements and Security Assessment Procedures, v1.2, Oct. 2008.

[111] H. Peine. Security Test Tools for Web Applications. Technical Report 048.06, Fraunhofer IESE, Jan. 2006.

[112] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluations. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[113] PortSwigger. Burp Proxy. http://www.portswigger.net/burp/.

[114] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2008.

[115] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[116] T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

[117] A. Riancho. w3af – Web Application Attack and Audit Framework. http://w3af.sourceforge.net/.

[118] W. Robertson. *Detecting and Preventing Attacks Against Web Applications*. PhD thesis, University of California, Santa Barbara, June 2009.

[119] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2009.

[120] Ruby on Rails. http://rubyonrails.org/, 2013.

[121] RSnake. Sql injection cheat sheet. `http://ha.ckers.org/sqlinjection/`.

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

[123] M. Samuel, P. Saxena, and D. Song. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[124] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[125] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[126] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *Proceedings of the International World Wide Web Conference (WWW)*, 2002.

[127] ScrewTurn Wiki. `http://www.screwturn.eu/`, 2013.

[128] 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 (USENIX)*, 2008.

[129] SPI Dynamics. Complete Web Application Security: Phase 1 – Building Web Application Security into Your Development Process. SPI Dynamics Whitepaper, 2002.

[130] SpringSource. Contollers - Redirects. `http://www.grails.org/Controllers+-+Redirects`, 2010.

[131] 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)*, 2010.

[132] C. Steve and R. Martin. Vulnerability Type Distributions in CVE. *Mitre report, May*, 2007.

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

[134] L. Suto. Analyzing the Effectiveness and Coverage of Web Application Security Scanners. Case Study, Oct. 2007.

[135] L. Suto. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, Feb. 2010.

[136] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[137] A. van Kesteren and D. Jackson. The XMLHttpRequest Object. `http://www.w3.org/TR/2006/WD-XMLHttpRequest-20060405/`, Apr. 2006.

[138] M. Vieira, N. Antunes, and H. Madeira. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*, 2009.

[139] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, 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)*, 2007.

[140] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[141] 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)*, 2007.

[142] J. Weinberger, A. Barth, and D. Song. Towards Client-side HTML Security Policies. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[143] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2011.

[144] A. Wiegenstein, F. Weidemann, M. Schumacher, and S. Schinzel. Web Application Vulnerability Scanners—a Benchmark. Technical report, Virtual Forge GmbH, Oct. 2006.

[145] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2006.

[146] F. Yu, M. Alkhalaf, and T. Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[147] Zend Technologies Ltd. Zend Framework: Documentation: Action Helpers - Zend Framework Manual. `http://framework.zend.com/manual/en/zend.controller.actionhelpers.html#zend.controller.actionhelpers.redirector`, 2011.