TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

In-Depth Security Testing of Web Applications

carried out at the

Institute of Computer Aided Automation
Automation Systems Group
Vienna University of Technology

under the guidance of

Priv.Doz.Dipl.-Ing.Dr.techn. Christopher Kruegel

and

Priv.Doz.Dipl.-Ing.Dr.techn. Engin Kirda

by

Sean Mc Allister
Ennsgasse 13/6
Vienna 1020
Austria

May 8, 2008

Acknowledgements

To my family and friends who offered me continuous support and encouragement during my studies and the writing of these pages. Without them this would never have been completed.

I would like to thank my advisors, Engin Kirda and Christopher Kruegel, for their invaluable advice and support while working on this thesis and the preceding paper. This work has been supported by the Austrian Science Foundation (FWF).

Abstract

Over the last years, the complexity of web applications has grown significantly, challenging desktop programs in terms of functionality and design. Along with the rising popularity of web applications, the number of exploitable bugs has also increased. Web application flaws, such as cross-site scripting or SQL injection bugs, now account for more than two thirds of the reported security vulnerabilities.

Black-box testing techniques are a common approach to improve software quality and detect bugs before deployment. There exist a number of vulnerability scanners, or fuzzers, that expose web applications to a barrage of malformed inputs in the hope to identify input validation errors. Unfortunately, these scanners often fail to test a substantial fraction of a web application's logic, especially when this logic is invoked from pages that can only be reached after filling out complex forms that aggressively check the correctness of the provided values. Also, there are cases in which certain functionality (e.g., credit card payment) is enabled only after the user has executed a number of previous steps (e.g., add items to cart and checkout) in the correct order.

In this thesis I will introduce a number of techniques that make it possible to increase the overall coverage of these tools. One technique leverages information from existing use cases. This information enables the scanner to correctly fill out forms and exercise parts of the functionality that other tools cannot reach. The test generation process also abstracts from the available use cases, allowing the scanner to further expand the search, analyze more pages and, as a result, create more persistent database objects. The ability to create database objects is important to expose stored XSS vulnerabilities. This use-case-driven testing technique has been implemented and used to analyze a number of web applications.

Building on the guided crawling of applications the need arises to not only reach more depth within the test subject, but also to discover unknown functionality. Possible solutions to this problem are also presented and evaluated.

Zusammenfassung

Die Komplexität von webbasierten Applikationen hat in den vergangenen Jahren stetig zugenommen, oft ziehen sie in Bezug auf Funktionalität und Design mit herkömmlichen Desktopapplikationen gleich. Gemeinsam mit der steigenden Popularität wurden auch sicherheitsrelevante Fehler in diesen Applikationen alltäglich, so sehr, dass mittlerweile sogar mehr als zwei Drittel aller gemeldeten Sicherheitslücken diese Programmgattung betreffen.

Black-Box Tests sind ein gängiges Mittel um die Qualität von Software zu verbessern und Fehler darin aufzuspüren. Für webbasierte Applikationen existieren viele Programme, die das Entdecken von sicherheitsrelevanten Fehlern automatisieren und vereinfachen sollen. So genannte *Scanner* oder *Fuzzer* konfrontieren die Applikation mit fehlerhaften Eingabewerten und schließen aus den Antworten auf das Vorhandensein einer Sicherheitslücke. Diese Programme haben allerdings beträchtliche Nachteile beim Navigieren einer webbasierten Applikation, da sie oft an Formularen scheitern, die ein Vordringen in die Tiefen der Applikation verhindern. Ebenso stellt sie eine Abfolge an Unteraufgaben vor unlösbare Probleme, wenn die Reihenfolge dieser Schritte von Bedeutung ist, wie beispielsweise das Befüllen des virtuellen Einkaufwagens und das nachfolgende Bezahlen mit Kreditkarte in einem Webshop.

Diese Arbeit präsentiert, implementiert und evaluiert einige Methoden die es sich zum Ziel setzen eine höhere Testabdeckung innerhalb einer webbasierten Applikation zu erreichen. Zuerst werden aus bestehenden Anwendungsfällen der Applikation Informationen gewonnen die es sowohl erlauben einen Scanner mit sinnvollen Eingabedaten anzureichern als auch komplexe Arbeitsabläufe zu ermöglichen, die ein Vordringen in die Tiefen der Applikation erst erlauben. Weiters ist es möglich von diesen vorgegebenen Informationen zu abstrahieren und so automatisiert weitere Testfälle zu erzeugen und auszuführen.

Contents

List of Figures

List of Tables

1 Introduction

The first web applications were collections of static files, linked to each other by means of HTML references. Over time, dynamic features were added, and web applications started to accept user input, changing the presentation and content of the pages accordingly. This dynamic behavior was traditionally implemented by CGI scripts. Nowadays, more often then not, complete web sites are created dynamically. To this end, the site's content is stored in a database. Requests are processed by the web application to fetch the appropriate database entries and present them to the user. Along with the complexity of the web sites, the use cases have also become more involved. While in the beginning user interaction was typically limited to simple request-response pairs, web applications today often require a multitude of intermediate steps to achieve the desired results.

When developing software, an increase in complexity typically leads to a growing number of bugs. Of course, web applications are no exception. Moreover, web applications can be quickly deployed to be accessible to a large number of users on the Internet, and the available development frameworks make it easy to produce (partially correct) code that works only in most cases. As a result, web application vulnerabilities have sharply increased. For example, in the last two years, the three top positions in the annual Common Vulnerabilities and Exposures (CVE) list published by Mitre [37] were taken by web application vulnerabilities.

To identify and correct bugs and security vulnerabilities, developers have a variety of testing tools at their disposal. These programs can be broadly categorized as based on black-box approaches or white-box approaches. White-box testing tools, such as those presented in [14, 33, 51, 56], use static analysis to examine the source code of an application. They aim at detecting code fragments that are patterns of instances of known vulnerability classes. Since these systems do not execute the application, they achieve a large code coverage, and, in theory, can analyze all possible execution paths. A drawback of white-box testing tools is that each tool typically supports only very few (or a single) programming language. A second limitation is the often significant number of false positives. Since static code analysis faces undecidable problems, approximations are necessary. Especially for large software applications, these approximations can quickly lead to warnings about software bugs that do not exist.

Black-box testing tools [26] typically run the application and monitor its execution. By providing a variety of specially-crafted, malformed input values, the goal is to find cases in which the application misbehaves or crashes. A significant advantage of black-box testing is that there are no false positives. All problems that are reported are due to real bugs. Also, since the testing tool provides only input to the application, no knowledge about implementation-specific details (e.g., the used programming language) is required. This allows one to use the same tool for a large number of different applications. The drawback of black-box testing tools is their limited code coverage. The reason is that certain program paths are exercised only when specific input is provided.

Black-box testing is a popular choice when analyzing web applications for security errors. This is confirmed by the large number of open-source and commercial black-box tools that are available [12, 34, 39, 53]. These tools, also called web vulnerability scanners or fuzzers, typically check for the presence of well-known vulnerabilities, such as cross-site scripting (XSS) or SQL injection flaws. To check for security bugs, vulnerability scanners are equipped with a large database of test values that are crafted to trigger XSS or SQL injection bugs. These values are typically passed to an application by injecting them into the application's HTML form elements or into URL parameters.

Web vulnerability scanners, sharing the well-known limitation of black-box tools, can only test those parts of a web site (and its underlying web application) that they can reach. To explore the different parts of a web site, these scanners frequently rely on built-in web spiders (or crawlers) that follow links, starting from a few web pages that act as seeds. Unfortunately, given the increasing complexity of today's applications, this is often insufficient to reach "deeper" into the web site. Web applications often implement a complex workflow that requires a user to correctly fill out a series of forms. When the scanner cannot enter meaningful values into these forms, it will not reach certain parts of the site. Therefore, these parts are not tested, limiting the effectiveness of black-box testing for web applications.

In this thesis, I present techniques that improve the effectiveness of web vulnerability scanners. To this end, the improved scanner leverages input from real users as a starting point for its testing activity. More precisely, starting from recorded, actual user input, test cases are generated that can be replayed. By following a user's session, fuzzing at each step, the tool is able to increase the code coverage by exploring pages that are not reachable for other tools. Moreover, the techniques allow a scanner to interact with the web application in a more meaningful fashion. This often leads to test runs where the web application creates a large number of persistent objects (such as database entries). Creating objects is important to check for bugs that manifest when malicious input is stored in a database, such as in the case of stored cross-site scripting (XSS) vulnerabilities. Finally, when the vulnerability scanner can exercise some control over the program under test, it can extract important feedback from the application that helps in further improving the scanner's effectiveness.

In addition to leveraging existing user input to increase the coverage, the presented system also uses feedback that is obtained from the application under test while scanning for vulnerabilities. This feedback allows me to create a knowledge base that establishes a mapping between back-end server functions and the input data that these functions operate on. Using this knowledge base, the system can reuse input data that was collected for one point of the application to generate valid input data at another, seemingly unrelated, point. This allows the scanner to provide meaningful input to forms and thereby further increase the overall coverage.

I have implemented these techniques in a vulnerability scanner that can analyze applications that are based on the django web development framework [22]. The experimental results demonstrate that the presented tool achieves larger coverage and detects more vulnerabilities than existing open-source and commercial fuzzers.

Section 3 describes the current approaches to automated penetration testing of web applications and describes the weaknesses of current implementations and how I plan to improve coverage by providing real user input as a starting point for the fuzzing activities. In Section 4.2, this thesis provides a short introduction to a popular software design pattern that is often used to implement web applications. Furthermore, I will propose techniques to improve web application scanners based on attributes that can be exported from web applications built on top of these.

2 Security in Web Applications

There are multiple known attack vectors against web applications. Many of them target the server hosting the application while others target the users of a particular site. The most common vulnerabilities [37] that affect software products are found in the realm of web applications. This has numerous reasons, one being the widespread availability of these applications that makes them both interesting as well as susceptible to attackers [36]. Furthermore they are less involving than classical exploits such as stack and heap based overflows and do not require the same amount of technical sophistication from the attacker, making them easier to detect and subsequently easier to exploit. This thesis focuses on the detection of cross-site scripting vulnerabilities, according to the Mitre [37] vulnerability type distribution report [36] the most common type of vulnerability reported in todays applications.

2.1 HTTP attack vectors

A web application can be defined as a client-server architecture style application that communicates by the means of the HTTP protocol [30]. Commonly web applications are rendered inside a web browser and encapsulate their logic on the server side, with the users browser acting as a thin client to the functionality the server provides. The attack surface of a web application is rather limited and it is possible to define the entry points that are controllable by the user or by an attacker of an application. According to [49], the possible points of attack are:

- the URL, including any GET data
- the POST data contained in a request
- the cookies

Tainting of any one of these parameters can lead to a security breach within the attacked application and there are numerous known attacks that can affect both the server (i.e. SQL injections) or the other users of the site (i.e. cross-site scripting). Note that the kind of attacks under discussion try to compromise the application itself and not any layers in between, such as the network stack of the server or the firewall.

2.2 Cross-Site Scripting

Cross-site scripting, often referred to as XSS, is a vulnerability typically found in Web Applications. It is based on the principle that user input, as can be passed to the application using either one of the three methods described in 2.1, is echoed back to the user by the web application [19]. The attack usually has the intention of injecting javascript into the HTML source of a site that will be executed by the browser when the page is visited. The real danger of XSS results from the fact that the injected javascript has access to all properties of the page it is shown on. Note that this does not only mean that it can change the DOM (resulting in a defacement of the site) of the page but can also access cookies (which are often used to authorize users) and form values as well as form actions. This can potentially lead to a form submission being redirected to a page controlled by the attacker and while this is not much of a problem for the user of a search engine it could have devastating results on an online banking site, where the login form can be redirected. A simple example should explain the basics of this attack.

XSS Example A search engine that upon querying returns a result page that includes the original search term. The imaginary URL of the results page would now be

```
http://www.example.com/search.php?q=searchterm
```

Listing 1: URL with get data

and this leads to the following HTML snippet being generated by the server and displayed in the browser:

```
<p>
        Results for "searchterm".
</p>
```

Listing 2: HTML Snippet displayed on the search result page

The problem here could be that user supplied input (i.e. the search term) is embedded unfiltered into the results page returned by the server and thereby also embedded into HTML code. The browser has no way of differentiating between the static parts of the page (i.e. the HTML code the results page is made of) and the dynamically generated part (i.e. the reflected search term). An attacker could now use the query parameter to inject code into the page, making this an example of a reflected cross-site scripting attack (see 2.2.1 for details). By replacing the GET parameters in the URL with the following string,

```
q=%3Cscript%3Ealert(%27XSS%27);%3C/script%3
```

Listing 3: XSS attack string embedded into GET data

where the value for the q parameter is the URL escaped version of this piece of javascript code:

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

Listing 4: XSS payload

The results returned by the server now generates the following HTML snippet:

```
<p>
        Results for "<script>alert('XSS');</script>".
</p>
```

Listing 5: HTML snippet with embedded javascript

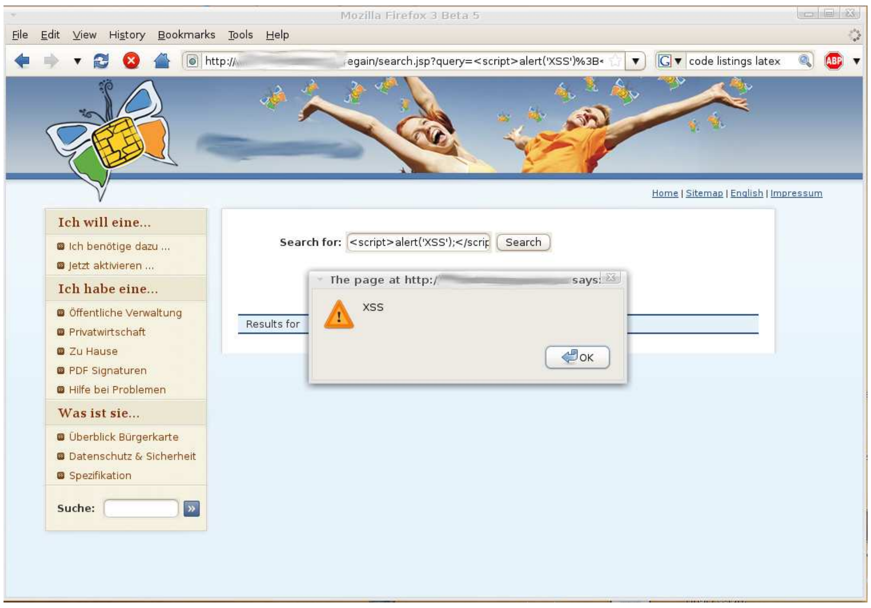
The script tags that were injected into the HTML source are valid and will be executed by a browser that has javascript enabled. This simple example results in an annoying but harmless alert box, as shown in 1. Using a similar technique it is also possible to construct a URL that contains the cookie value of the site and points to a server controlled by the attacker. The browser can not differentiate between legitimate and malicious requests, eventhough the same origin policy for javascript is in place [25]. Used this way the attacker has effectively taken over the victims session.
This was a very basic example and has the sole reason to demonstrate the basic concepts of XSS attacks. Further information, explanations and a number of sample attacks can be found in [20] and [50]. A XSS vulnerability known as the Samy Worm made the headlines in 2005 [10] and was targeted at the social networking website MySpace. With the use of a stored XSS vulnerability within the profile page Samy was able to alter the profiles of visiting users. Furthermore the worm copied it's own payload to the victims profile, thereby propagating to millions of users and finally leading to down time of the MySpace servers.

2.2.1 Types of XSS

There are three known types of XSS, each with their own characteristics.

Reflected XSS This is the simplest form of XSS and at the same time the easiest to detect. As described in 2.2 this vulnerability is most often encountered on dynamic pages that reflect (hence the name) user supplied input parameters. The attack payload is delivered to the victim with the help of a crafted URL, such as shown in the example in 2.2. The execution of the attack is then staged within a single request-response transaction to the server, which is why reflected XSS is sometimes referred to as first-order XSS.

Stored XSS This is the most dangerous form of XSS as it does not involve delivering the attack URL to the potential victim. The single unique feature of this attack is that the payload is stored on the server and displayed to users of the site. The storage usually takes place in a back-end database system that is queried by the web application in order to display it's contents. A


Mozilla Firefox 3 Beta 5

http://[redacted]egain/search.jsp?query=<script>alert('XSS')%3B<

Search for: <script>alert('XSS');</scrip

The page at http:/[redacted] says:

XSS

Figure 1: A reflected XSS vulnerability

typical example would be the comment section of a blog that allows the reader of the article to post his thoughts on the subject at hand. A vulnerability within this system has the potential to reach every reader of the article. Considering the fact that many sites require registration, and thereby a login form, in order to post contents on a site this is all the more interesting for attackers wishing to acquire login credentials or session cookies. There is no technical difference in the possible attack payloads as compared to reflected XSS attacks. The way in which the payload reaches the site on the other hand can vary, either by ways provided by the application itself, such as the aforementioned comment form or via off-site means such as sending a tainted email message to users of a webmail service. This attack usually involves a number of steps, namely those for the attacker delivering the malicious content to the site and those that the unaware user undertakes to retrieve (and eventually execute) the payload. This attack is sometimes referred to as second-order XSS.

DOM based XSS This kind of attack is relatively new and was first described in 2005 in [13]. It relies on the existence of exploitable javascript on a page and a crafted URL. The use case this attack exploits is found when javascript accesses the URL of the currently shown document, extracts values from it and injects them into the pages HTML source. The attacker can taint parts of the URL with executable javascript that is subsequently copied to the HTML and then executed. The difference in comparison with the reflected XSS attack is that the attack payload is not echoed by the server, but is both inserted and executed by the client (i.e. the browser).

2.2.2 Mitigating XSS attacks

The example shown in 2.2 is very simple and is based upon the assumption that no sanitization whatsoever is performed upon the user input. This is often not the case in real world situations, where web developers have become more aware to the problems raised by user interaction with a web application. Often filters are used to clean user input so that it can be safely used within a web application. A simple but effective way to sanitize input strings such as the example above is to convert all HTML special characters into their escaped form. Other approaches try to do selective filtering of HTML tags in order to allow custom HTML and/or CSS supplied by the user, to strip the input from any malicious code it might contain.

Using such an escaping filter for the XSS attack string from 4 yields the following string that can be safely embedded into HTML:

```
&lt;script&gt;alert('XSS');&lt;/script&gt;
```

Listing 6: escaped XSS payload

This string is not executable by a browser, but would be rendered in the graphic presentation, making an attack upon this particular form impossible. Unfortunately there are other ways to stage XSS attacks, depending on how the site under test composes the response. The escaping of HTML entities might not be sufficient depending on the location inside the HTML code where user supplied input is being echoed by the application.
If for example the user submitted string reappears inside of javascript code or in other locations where javascript is allowed outside of script tags, such as event handlers, the escaping of HTML entities is not sufficient.

3 Web Application Testing

The realm of black-box testing tools for websites is closely related to the scanning for vulnerabilities inside the same. I will first discuss general black-box testing tools that target the audience of web developers, then round up current solutions to vulnerability scanning of websites and their limitations.

3.1 Black-box testing tools for web applications

The selection of tools for black-box testing of web applications has steadily grown over the last years, for python alone there exist many solutions that differ in usage, focus and maturity [3]. The same is even more true for other popular programming languages like Java [2] and Ruby [7]. The aim of many of these tools is to integrate well with unit testing frameworks, which is the reason many of them are language dependant. Some frameworks, such as django [22] even supply their own black-box testing tools for web applications [6] and have the advantage that they offer functionality that would not be possible without framework integration.
More general testing tools such as Twill [54] or Selenium [4] offer a more generalized way of describing test cases, as both are scriptable without knowledge of a programming language, but also are programmable from within applications. For this Selenium offers a component called Remote Control (RC [5]) that is available for numerous programming languages (Python, Java, PHP,..). Twill on the other hand offers an API that is similar to real browsers, including back buttons and reload functions from within python.
These tools are generally classifiable into two types, namely protocol (or application) drivers (such as Twill) and browser drivers (such as Selenium). While protocol drivers can interact with web servers, they are often limited in the way they handle non-HTML content, hardly any of them can handle and execute javascript. Furthermore they need to deal with malformed HTML. Spiders that crawl and parse websites for information such as those used by search engines are examples of protocol drivers. Browser drivers on the other hand do not have to rely on their ability to handle malformed HTML, as they use and control a real browser through the use of plugins. Browser drivers are potentially more powerfull for the purpose of testing for certain vulnerabilities that affect the browser, as they can eliminate the occurrence of false positives and can potentially detect vulnerabilities within embedded content such as Flash.

While protocol drivers, due to the missing display functionality, offer a speed advantage the strength of browser drivers lies within the ability to detect certain non-functional characteristics, such as cross-browser compatibility.

3.2 Web Application Vulnerability Scanners

Penetration testers auditing a web site have numerous possibilities to achieve their goals, depending on the vulnerabilities they wish to find the selection of tools varies greatly. Vulnerability detection software for web applications mostly operates in the realm of black box testing. This is due to the client-server architecture of common web applications where the content generated on the server is sent to the clients web browser and displayed there using markup languages. With the advent of the so called Web 2.0 techniques this strict separation of the server generating markup and the client displaying the same has been changed to the role of the server delivering content that is then interpreted and displayed by the client. The markup of choice is still HTML, but the responsibility of generating markup can now be shifted towards the client.

One way to quickly and efficiently identify flaws in web applications is the use of vulnerability scanners. These scanners test the application by providing malformed inputs that are crafted so that they trigger certain classes of vulnerabilities. Typically, the scanners cover popular vulnerability classes such as cross-site scripting (XSS) or SQL injection bugs. These vulnerabilities are due to input validation errors. That is, the web application receives an input value that is used at a security-critical point in the program without (sufficient) prior validation. In case of an XSS vulnerability [24], malicious input can reach a point where it is sent back to the web client. At the client side, the malicious input is interpreted as JavaScript code that is executed in the context of the trusted web application. This allows an attacker to steal sensitive information such as cookies. In case of a SQL injection flaw, malicious input can reach a database query and modify the intended semantics of this query. This allows an attacker to obtain sensitive information from the database or to bypass authentication checks.

By providing malicious, or malformed, input to the web application under test, a vulnerability scanner can check for the presence of bugs. Typically, this is done by analyzing the response that the web application returns. For example, a scanner could send a string to the program that contains malicious JavaScript code. Then, it checks the output of the application

for the presence of this string. When the malicious JavaScript is present in the output, the scanner has found a case in which the application does not properly validate input before sending it back to clients. This is reported as an XSS vulnerability.
To send input to web applications, scanners only have a few possible injection points, as defined in 2.1. These points are often derived from form elements that are present on the web pages. That is, web vulnerability scanners analyze web pages to find injection points. Then, these injection points are fuzzed by sending a large number of requests that contain malformed inputs.
There are of course many other security related issues that can not be tested using scanners, these being mainly in the realm of of incomplete authorization or flaws in the business logic of an application [28], but many scanners also fail to identify the flaws they are supposed to detect, simply because they can not interpret the output of a web application on a semantic level. This brings us to the challenges faced by current vulnerability scanners.

3.3 Limitations

Automated scanners have a significant disadvantage compared to human testers in the way they can interact with the application. Typically, a user has certain goals in mind when interacting with a site. On an e-commerce site, for example, these goals could include buying an item or providing a rating for the most-recently-purchased goods. The goals, and the necessary operations to achieve these goals, are known to a human tester. Unfortunately, the scanner does not have any knowledge about use cases; all it can attempt to do is to collect information about the available injection points and attack them. More precisely, the typical workflow of a vulnerability scanners consists of the following steps:

- First, a web spider crawls the site to find valid injection, or entry, points. Commonly, these entry points are determined by collecting the links on a page, the action attributes of forms, and the source attributes of other tags. Advanced spiders can also parse JavaScript to search for URLs. Some even execute JavaScript to trigger requests to the server.
- The second phase is the audit phase. During this step, the scanner fuzzes the previously discovered entry points. It also analyzes the application's output to determine whether a vulnerability was triggered.

- Finally, many scanners will start another crawling step to find stored XSS vulnerabilities. In case of a stored XSS vulnerability, the malicious input is not immediately returned to the client but stored in the database and later included in another request. Therefore, it is not sufficient to only analyze the application's immediate response to a malformed input. Instead, the spider makes a second pass to check for pages that contain input injected during the second phase.

The common workflow outlined above and shown in figure 3.3 yields good results for simple sites that do not require a large amount of user interaction. Unfortunately, it often fails when confronted with more complex sites. The reason is that vulnerability scanners are equipped with simple rules to fill out forms. These rules, however, are not suited well to advance "deeper" into an application when the program enforces constraints on the input values that it expects. To illustrate the problem, I briefly discuss an example of how a fuzzer might fail on a simple use case.

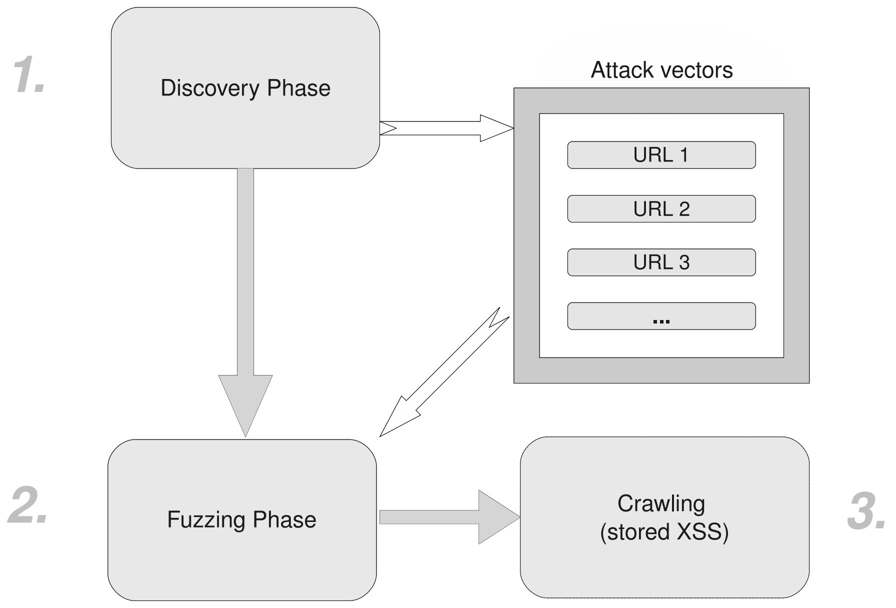


Figure 2: Workflow of common web application vulnerability scanners.

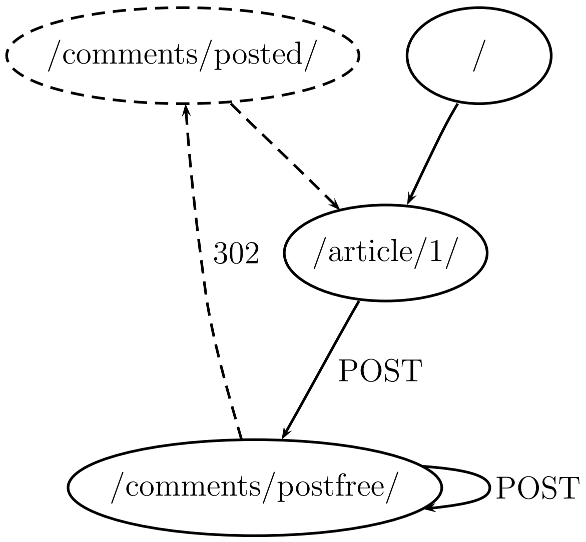


Figure 3: A user interaction example of a visitor posting a comment to the blog.

The example involves a blogging site that allows visitors to leave comments to each entry. To leave a comment, the user has to fill out a form that holds the content of the desired comment. Once this form is submitted, the web application responds with a page that shows a preview of the comment, allowing the user to make changes before submitting the posting. When the user decides to make changes and presses the corresponding button, the application returns to the form where the text can be edited. When the user is satisfied with her comment, she can post the text by selecting the appropriate button on the preview page. A graphical representation of the user interaction required to successfully post a comment to the is given in figure 3. The dotted lines show the parts of the workflow a spider was not able to follow.
The problem in this case is that the submit button (which actually posts the message to the blog) is activated on the preview page only when the web application recognizes the submitted data as a valid comment. This requires that both the name of the author and the text field of the comment are filled in. Furthermore, it is required that a number of hidden fields on the page remain unchanged. When the submit button is successfully pressed, a comment is created in the application's database, linked to the article, and subsequently shown in the comments section of the blog entry.

For a vulnerability scanner, posting a comment to a blog entry is an entry point that should be checked for the presence of vulnerabilities. Unfortunately, all of the tools evaluated in the experiments (details in Section 6.3) failed to post a comment. That is, even a relatively simple task, which requires a scanner to fill out two form elements on a page and to press two buttons in the correct order, proved to be too difficult for an automated scanner. Clearly, the situation becomes worse when facing more complex use cases.
During the evaluation of existing vulnerability scanners, I found that, commonly, the failure to detect a vulnerability is not due to the limited capabilities of the scanner to inject malformed input or to determine whether a response indicates a vulnerability, but rather due to the inability to generate enough valid requests to reach the vulnerable entry points. Of course, the exact reasons for failing to reach entry points varies, depending on the application that is being tested and the implementation of the scanner.
There are obviously attack vectors that rely on invalid input data, such as unescaped error messages when a form is filled in incorrectly, but for the sake of detecting the existence of stored XSS vulnerabilities the supplied datasets need to be at least as valid as the server side validation allows. It is therefore necessary to find a way that helps the fuzzer to apply it's attack strings to valid input data.

3.4 Current solutions

As shown in 3.3 and evaluated in 6 there is a need to teach web application scanners to crawl deep into existing applications and at the same time not destroy the state of their session. There are a number of current solutions that aim to solve this problems.

Proxying requests Most web application vulnerability scanners can function as a proxy for manual input through a web browser. For this the penetration testerś browser is configured to utilize a local proxy that is supplied by the scanner. The tester then navigates the web application while the proxy monitors the (previously undiscovered) endpoints and the values submitted with each request. This can for obvious reasons greatly improve the coverage a scanner can reach within a web application and at the same time supplies sane input data, that it can then use to audit forms. Sometimes this is also the only way to actually collect endpoints, as a site may link to important URLs only within Flash animations or Java applets, both formats that the spider components of the scanners are unlikely to support. From the evaluated auditing tools each supports this feature in one way or the other: While w3af [55] does so via the so called spiderMan plugin, the burp suite even has it's main focus on the proxy functionality. Acunetix offers a proxy and even a specialized browser that records login sequences. The downside this approach suffers from in most implementations is that the discovery and the auditing phases are decoupled. The proxy, that is part of the auditing phase, is able to collect endpoints, but the endpoints and the name-value pairs associated with it may no longer be valid in the auditing phase for various reasons, such as dynamic URLs or a complex workflow that requires certain actions to be completed in the correct order.

Blocking URLs Especially in applications that support some kind of session management it is essential to differ between registered and anonymous users. The way an application acts might greatly depend on this factor, or an application might even only be available to authenticated users. In the latter case it is definitely undesireable to follow the logout link, as a spider might be unable to restore it's session or the applications logic might prevent it from doing so. Similar rules must be put in place when testing the administrative interface of a site, where it is probably undesireable, even within a controlled testing environment, to delete all database entries. The current solutions involve a blacklist of URLs that the spider does not follow. Again each tool supports this feature, and Acunetix additionaly offers the ability to graphically select log out links, that it will then block during the auditing phase. Depending on the web application under test the effort to correctly configure the tool may vary greatly, and penetration tester might not even be aware of all links that would require blocking. On the other hand rigorous blocking of functionality may lead to bad results in terms of coverage.

Guessing form values It is essential for a scanner to submit forms when testing for XSS vulnerabilities. Furthermore they need to fill these forms with reasonable values in order to proceed and eventually find vulnerabilities. With no input data supplied to the scanners the fall back to using use a number of heuristics to find reasonable input values. Starting with the burp spider that fills out every textfield with the same user configurable string and leading to w3af that parses the input elements labels to find useable values based on these labels, the techniques are as diverse as they are unreliable for large forms with rigorous server side validation of the submitted values.

4 Increasing Test Coverage

To address the limitations of existing tools, I propose several techniques that allow a vulnerability scanner to detect more entry points. These entry points can then be tested, or fuzzed, using existing databases of malformed input values such as [9]. The first technique, described in Section 4.1, introduces a way to leverage inputs that are recorded by observing actual user interaction. This allows the scanner to follow an actual use case, achieving more depth when testing. The second technique, presented in Section 4.2, discusses a way to abstract from observed user inputs, leveraging the steps of the use case to achieve more breadth. The third technique, described in Section 4.3, makes the second technique more robust in cases where the broad exploration interferes with the correct replay of a use case.

4.1 Increasing Testing Depth

One way to improve the coverage, and thus, the effectiveness of scanners, is to leverage actual user input. In order to achieve this it is necessary to first collect a small set of inputs that were provided by users that interacted with the application. These interactions correspond to certain use cases, or workflows, in which a user carries out a sequence of steps to reach a particular goal. Depending on the application, this could be a scenario where the user purchases an item in an on-line store or a scenario where the user composes and sends an email using a web-based mail program. Based on the recorded test cases, the vulnerability scanner can replay the collected input values to successfully proceed a number of steps into the application logic. The reason is that the provided input has a higher probability to pass server-side validation routines. Of course, there is, by no means, a guarantee that recorded input satisfies the constrains imposed by an application at the time the values are replayed. While replaying a previously recorded use case, the scanner can fuzz the input values that are provided to the application.

Collecting input. There are different locations where client-supplied input data can be collected. One possibility is to deploy a proxy between a web client and the web server, logging the requests that are sent to the web application (see 3.4) for details on current implementations. Another way is to record the incoming requests at the server side, by means of web server log files or application level logging. For simplicity, the prese nted solution

records requests directly at the server, logging the names and values of all input parameters.
For the presented implementation I chose to save the requests on the server side, which, given the overall goal to improve the testing abilities for the developers of web applications, is often the same machine that runs the tests. Furthermore there are the big advantages of not having to set up any kind of proxy or browser plug-in and being able to create test cases by simply using the application, a task that most web developers will have to do in order to verify their code in the first place. This also gives the developer the ability to save test cases that can be reused at a later point or even packaged with the application as a regression test or as a starting point for testing the deployment of an application.
Taking the reverse approach it is now also possible to record the input that is produced during regular, functional testing of applications. Typically, developers need to create test cases that are intended to exercise the complete functionality of the application. When such test cases are available, they can be immediately leveraged by the vulnerability scanner. Another alternative is to deploy the collection component on a production server and let real-world users of the web application generate test cases. This obviously results in a performance penalty on the web server, but might be applicable in environments with small amounts of traffic.
In any case, the goal is to collect a number of inputs that are likely correct from the application's point of view, and thus, allow the scanner to reach additional parts of the application that might not be easily reachable by simply crawling the site and filling out forms with essentially random values.

Replaying input. Each use case consists of a number of steps that are carried out to reach a certain goal. For each step, requests were recorded and based upon these input values, the vulnerability scanner can replay a previously collected use case. To this end, the vulnerability scanner replays a recorded use case, one step at a time. After each step, a fuzzer component is invoked. This fuzzer uses the request issued in the previous step to test the application. More precisely, it uses a database of malformed values to replace the valid inputs within the request sent in the previous step. In other words, after sending a request as part of a replayed use case, the tool attempts to fuzz this request. Then, the previously recorded input values stored for the current step are used to advance to the next step. This process of fuzzing a

request and subsequently advancing one step along the use case is repeated until the test case is exhausted. Alternatively, the process stops when the fuzzer replays the recorded input to advance to the next page, but this page is different from the one expected. This situation can occur when a previously recorded input is no longer valid.
Using this technique the evaluation will demonstrate that it is possible are able to generate requests that will with some certainty pass server side validation and will subsequently lead to a tainted object being stored in the back-end storage.
When replaying input, the vulnerability scanner does not simply re-submit a previously recorded request. Instead, it scans the page for elements that require user input. Then, it uses the previously recorded request to provide input values for those elements only. This is important when an application uses cookies or hidden form fields that are associated with a particular session. Changing these values would cause the application to treat the request as invalid. Thus, for such elements, the scanner uses the current values instead of the "old" ones that were previously collected. The rules used to determine the values of each form field aim to mimic the actions of a benign user. That is, hidden fields are not changed, as well as read-only widgets (such as submit button values or disabled elements).
By not forcing the web application to deal with potentially broken data that it does not expect this technique can reach a level of robustness that results in a higher number of requests being accepted by the aplication as valid. This allows the scanner to create a high number of database objects which is an advantage when searching for stored XSS vulnerabilities.

Guided fuzzing. I call the process of using previously collected traces to step through an application *guided fuzzing*. Guided fuzzing improves the coverage of a vulnerability scanner because it allows the tool to reach entry points that were previously hidden behind forms that expect specific input values. That is, this method can increase the depth that a scanner can reach into the application.

4.2 Increasing Testing Breadth

With guided fuzzing, after each step that is replayed, the fuzzer only tests the single request that was sent for that step. That is, for each step, only a single entry point is analyzed. A straightforward extension to guided fuzzing

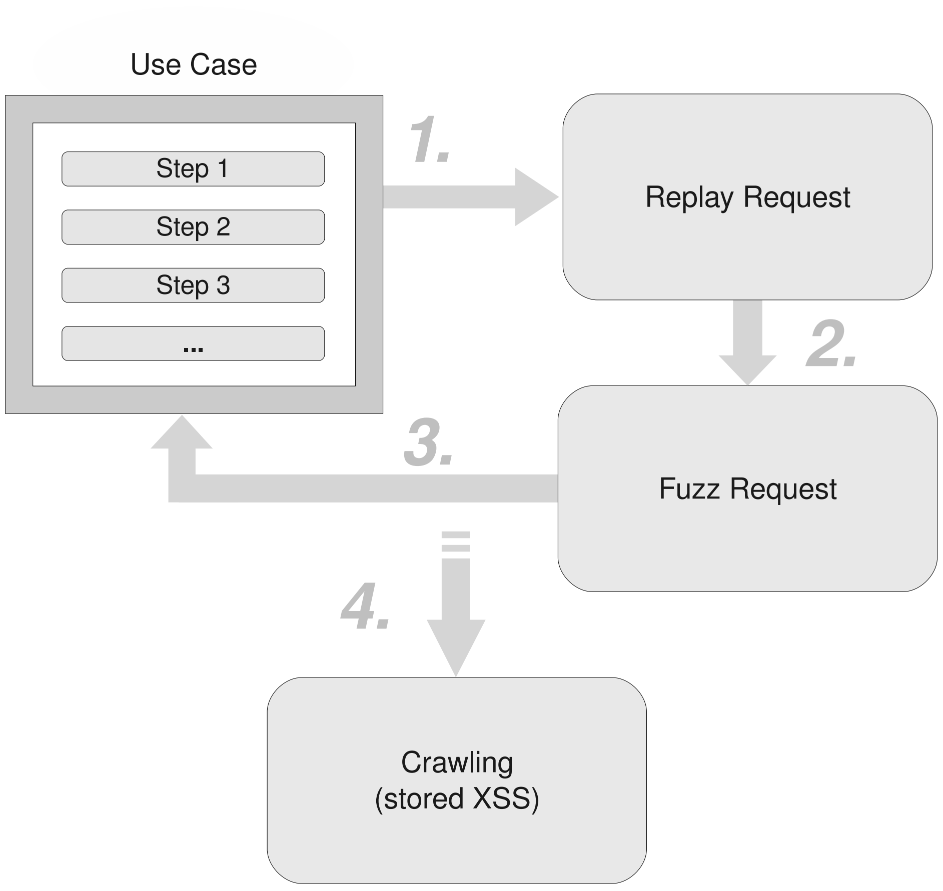


Figure 4: Workflow of the guided fuzzing approach.

is to not only test the single entry point, but to use the current step as a starting point for fuzzing the complete site that is reachable from this point. That is, the fuzzer can use the current page as its starting point, attempting to find additional entry points into the application. Each entry point that is found in this way is then tested by sending malformed input values. In this fashion, the workflow does not only increase the depth of the test cases, but also their breadth. For example, when a certain test case allows the scanner to bypass a form that performs aggressive input checking, it can then explore the complete application space that was previously hidden behind that form. In this thesis this approach is referred to as *extended, guided fuzzing*. The schematic representation of this approach can be seen in figure 4.2.

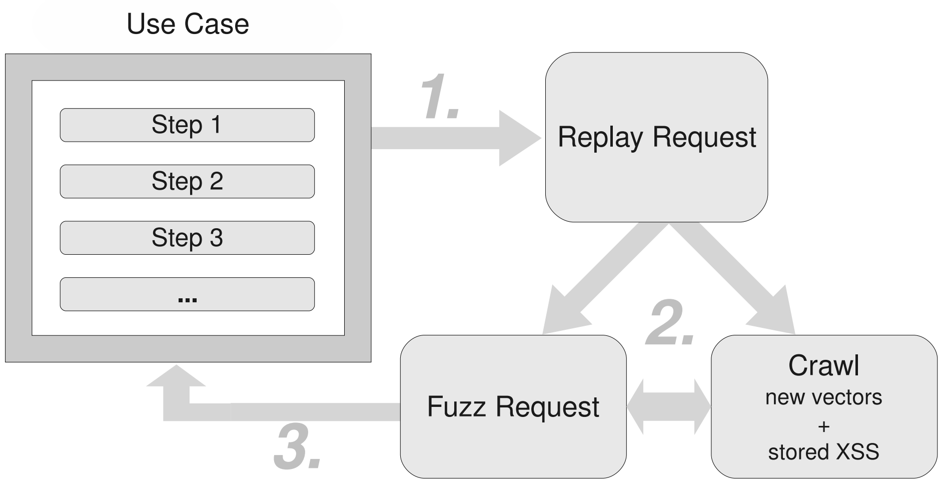


Figure 5: Workflow of the extended, guided fuzzing approach.

Extended, guided fuzzing has the potential to increase the number of entry points that a scanner can test.

4.2.1 Problems and pitfalls

However, alternating between a comprehensive fuzzing phase and advancing one step along a recorded use case can also lead to problems. To see this, consider the following example. Assume an e-commerce application that uses a shopping cart to hold the items that a customer intends to buy. The

vulnerability scanner has already executed a number of steps that added an item to the cart. At this point, the scanner encounters a page that shows the cart's inventory. This page contains several links; one link leads to the checkout view, the other one is used to delete items from the cart. Executing the fuzzer on this page can result in a situation where the shopping cart remains empty because all items are deleted. This could cause the following steps of the use case to fail, for example, because the application no longer provides access to the checkout page. A similar situation can arise when administrative pages are part of a use case. Here, running a fuzzer on a page that allows the administrator to delete all database entries could be very problematic. In general terms, the problem with extended, guided fuzzing is that the fuzzing activity could interfere in undesirable ways with the use case that is replayed. In particular, this occurs when the input sent by the fuzzer changes the state of the application such that the remaining steps of a use case can no longer be executed. This problem is difficult to address when assumed that the scanner has no knowledge and no control of the inner workings of the application under test. In the worst case the first fuzzing phase breaks the use case, in which case the results would be the same as without any modifications to the workflow of the fuzzer.
In the following Section 4.3, I consider the case in which the test system can interact more tightly with the analyzed program. In this case, it will be possible to prevent the undesirable side effects (or interference) from the fuzzing phases.

4.3 Stateful Fuzzing

The techniques presented in the previous sections work independently of the application under test. That is, the system builds black-box test cases based on previously recorded user input, and it uses these tests to check the application for vulnerabilities.
In this subsection, I consider the case where the scanner has considerable control over the application under test to adress the shortcomings described in 4.2.1
In order to understand how state information can be captured it is also essential to discuss the pattern that makes this approach possible.
One solution to the problem of undesirable side effects of the fuzzing step when replaying recorded use cases is to *take a snapshot* of the state of the application after each step that is replayed. Then, the fuzzer is allowed

to run. This might result in significant changes to the application's state. However, after each fuzzing step, the application is *restored* to the previously taken snapshot. At this point, the replay component will find the application in the expected state and can advance one step. After that, the process is repeated - that is, a snapshot is taken and the fuzzer is invoked. In this thesis this process will be called *stateful fuzzing*. The workflow of this approach can be seen in figure 4.3.
In principle, the concrete mechanisms to take a snapshot of an application's state depend on the implementation of this application. Unfortunately, this could be different for each web application. As a result, it would be necessary to customize the test system to each program, making it difficult to test a large number of different applications. Clearly, this is very undesirable. Fortunately, the situation is different for web applications. Over the last years, the model-view-controller (MVC) scheme has emerged as the most popular software design pattern for applications on the web.

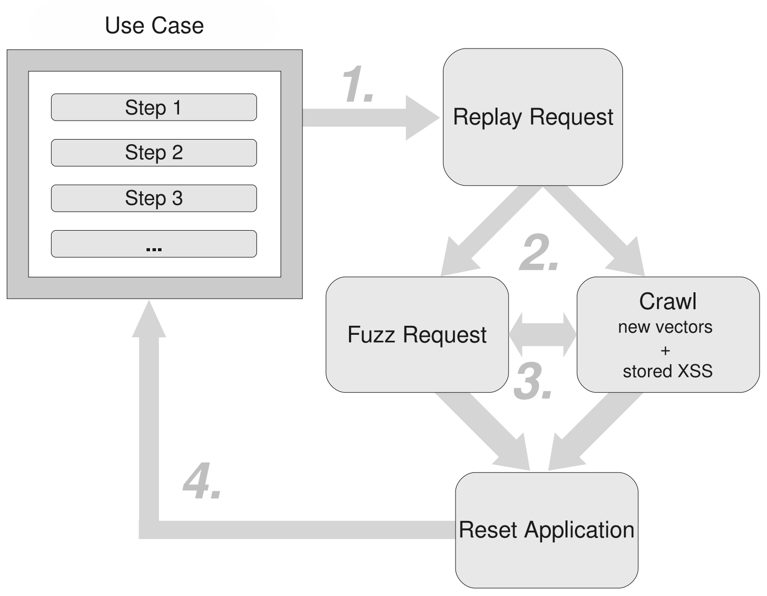


Figure 6: Workflow of the stateful fuzzing approach.

The MVC design pattern and web application development. Java programmers often refer to it as "Model 2" and not only since Ruby on Rails was released in 2004 many web development frameworks have made good use of the advantages offered by this paradigm. The goal of the MVC scheme is to separate three layers that are present in almost all web applications. These are the data layer, the presentational layer, and the application logic layer. The data layer represents the data storage that handles persistent objects. Typically, this layer is implemented by a backend database and an object (relational) manager. The application logic layer uses the objects provided by the data layer to implement the functionality of the application. It uses the presentational layer to format the results that are returned to clients. The presentation layer is frequently implemented by an HTML template engine. Moreover, as part of the application logic layer, there is a component that maps requests from clients to the corresponding functions or classes within the program. The terms used in the different frameworks built on this stack often vary and lead to some confusion amongst developers and users alike [1], but the basic principles remain the same.

Based on the commonalities between web applications that follow an MVC approach, it is possible (for most such applications) to identify general interfaces that can be instrumented to implement a snapshot mechanism. To be able to capture the state of the application and subsequently restore it, I am interested in the objects that are created, updated, or deleted by the object manager in response to requests. Whenever an object is modified or deleted, a copy of this object is serialized and saved. This way, it becomes possible, for example, to undelete an object that has been previously deleted, but that is required when a use case is replayed. In a similar fashion, it is also possible to undo updates to an object and delete objects that were created by the fuzzer.

The information about the modification of objects can be extracted at the interface between the application and the data layer (often, at the database level). At this level, a component is inserted that can serialize modified objects and later restore the snapshot of the application that was previously saved. Clearly, there are limitations to this technique. One problem is that the state of an application might not depend solely on the state of the persistent objects and its attributes. Nevertheless, this technique has the potential to increase the effectiveness of the scanner for a large set of programs that follow a MVC approach. This is also confirmed by the experimental results presented in Section 6.

4.4 Application feedback

Given that stateful fuzzing already requires the instrumentation of the program under test, it should be considered what additional information might be useful to further improve the vulnerability scanning process.
Another piece of feedback from the application that I consider useful is the *mapping of URLs to functions*. This mapping can be typically extracted by analyzing or instrumenting the controller component, which acts as a dispatcher from incoming requests to the appropriate handler functions. Using the mappings between URLs and the program functions, I can increase the effectiveness of the extended, guided fuzzing process. To this end, the scanner attempts to find a set of forms (or URLs) that all invoke the same function within the application. When the system has previously seen user input for one of these forms, it can reuse the same information on other forms as well (when no user input was recorded for these forms). The rationale is that information that was provided to a certain function through one particular form could also be valid when submitted as part of a related form. By reusing information for forms that the fuzzer encounters, it is possible to reach additional entry points.
When collecting user input (as discussed in Section 4.1), this system records all input values that a user provides on each page. More precisely, for each URL that is requested, the system store all the name-value pairs that a user submits along with the request. In case the scanner can obtain application feedback, it also stores the name of the program function that is invoked by the request. In other words, the system records a unique identifier of the function that the requested URL maps to. When the fuzzer later encounters an unknown action URL of a form (i.e., the URL where the form data is submitted to), the application is queried to determine which function this URL maps to. Then, the tool searches the collected information to see whether the same function was called previously by another URL. If this is the case, it examines the name-value pairs associated with this other URL. For each of those names, it attempts to find a form element on the current page that has a similar name. When a similar name is found, the corresponding, stored value is supplied. As mentioned previously, the assumption is that valid data that was passed to a program function through one form might also be valid when used for a different form, in another context. This can help in correctly filling out unknown forms, possibly leading to unexplored entry points and vulnerabilities.

As an example, consider a forum application where each discussion thread has a reply field at the bottom of the page. The action URLs that are used for submitting a reply could be different for each thread. However, the underlying function that is eventually called to save the reply and link it to the appropriate thread remains the same. Thus, when the tool encounters one case where a user submitted a reply, it would recognize other reply fields for different threads as being similar. The reason is that even though the action URLs associated with the reply forms are different, they all map to the same program function. Moreover, the name of the form fields are (very likely) the same. As a result, the fuzzer can reuse the input value(s) recorded in the first case on other pages.
It is of course questionable if it is at all desirable to fuzz the same back-end function over and over again and in reality this decision must be made by the penetration tester. At the very least this is also the way scanners work that have no knowledge about the applications structure and depending on the complexity of the application this might yield additional results, as attributes that are not foreseeable for the test case without using white box testing techniques can influence the servers response.
If such behavior is in deed not wanted a similar technique could be used to block scanners completely from fuzzing the same back-end callable over and over again, thereby reducing the time it takes for the tests to run.

4.5 Summary

In this section I have presented various techniques that can help to improve the test coverage of web application scanners. To summarize the approaches, here is a short rundown of the methods that were implemented and evaluated in the course of this thesis.

Guided Fuzzing. Current scanners do not have the necessary skills to fill out complex forms or follow workflows within web applications. By feeding them with real user input and guiding them through workflows that a ret ypical for the application under test it is possible to greatly enhance the effectiveness.

Extended, Guided Fuzzing. Using the guided fuzzing approach it is possible to crawl deep into web applications. Invoking a crawling and fuzzing stage after each step of the use case it is possible to increase coverage in terms of breadth of the application.

Stateful Fuzzing. As extended, guided fuzzing can have unwanted side effects, that will cause the use case to fail, there is the need to take snapshots of the applications state at the time when the fuzzer starts his work. As soon as the fuzzer returns the applications state is reset, so that the use case can proceed.

Mapping of callables This is a more general approach that can and will be applied to both the extended and guided fuzzing techniques. The knowledge about the functions that are invoked when a client requests specific URLs is important feedback that can help to improve the coverage of the vulnerability scanner. The reason is that, in some cases, it is possible to identify the types of inputs that a certain function expects. When the tool knows which URLs invoke this function, it can speculated that all those URLs use similar input parameters. If this speculation turns out to be correct, the scanner can correctly fill out HTML forms on pages for which no user input was previously recorded.

5 Implementation Details

A custom vulnerability scanner was developed that implements the techniques outlined above.

As discussed in the last section, some of the techniques require that a web application is instrumented **(i)** to capture and restore objects manipulated by the application, and **(ii)** to extract the mappings between URLs and functions.

The implementation of this observing component was done as a stand-alone Django [22] application that can be easily integrated into an existing project. One of the goals was to make the program as independent as possible and so there are no dependencies on other 3rd party Django applications.

The reason Django was chosen is that it is a web framework that implements the model-view-controller (MVC) scheme. Other candidates for the implementation were Ruby on Rails [21] and Java Servlets [52], both popular web development frameworks based upen the MVC pattern. This choice implies that the current implementation can only test web applications that are built using Django. Note, however, that the previously introduced concepts are general and can be ported to other development frameworks, but for obvious reasons require the core mechanisms, namely the capturing and mapping logics, to be implemented in their respective language and adapted to the needs of the framework.

The automated replaying and deep-crawling stage of the test procedure is then done by a component that utilizes a python web browser, namely twill [54]. Twill is a black box testing tool for web applications that enables the user to script site navigation, form-filling and -submission. It also supports a simple scripting language that can be used to create testcases without knowledge of python, but can also be embedded into python applications to enable programmatic browsing of web applications.

By themselves the aforementioned components can not attack web applications, but actually try to focus on the task of generating valid requests that will pass server side validation and enable a spider to reach new and unknown points within a web application. In order to audit the security of the application being tested a attack component was implemented that reuses the XSS plugin provided by the w3af [55] security auditing framework. This plugin provides a number of attack strings that can be injected into an application and the logic necessary to detect the existence of XSS vulnerabilites.

The full list of components that were combined to support the extensive testing procedures shown in this thesis are now:

- **django** [22] - a web development framework, the
- **twill** [54] - a testing tool for web applications
- **w3af** [55] - a security auditing framework for web applications

5.1 Extending the web framework

The main modules that monitor the applications state as well as all incoming requests are two middleware classes. Middleware in the django workflow has access to every incoming request. Any incoming request is first tunneled through the corresponding functions of all configured middleware classes before reaching the core application. Right after the view to be called has been determined and the arguments for the same prepared (this happens in the URL dispatcher, which matches the incoming request's path to a function via regular expressions), but before the actual view is invoked another function of the middleware is called, that gets the view function and it's applied positional and keyword arguments passed over. Then the view does it's work, gathers data from the database, renders the template and creates a HTTP-response object, which is returned. Before the response is returned to the client another function of the middleware is invoked that now also has access to the response object. A graphic representation of how the framework deals with requests and responses as well as errors can be seen in figure 7.
The information that is captured is stored in a relational database and since the implementation was made as a django application the tables used are represented as django models. The documentation for each model in use can be seen in Appendix A.

5.1.1 Capturing web requests.

The first task was to extend a Django application such that it can record the inputs that are sent when going through a use case. This makes it necessary to log all incoming requests together with the corresponding parameters. This was implemented in the form of a middleware class that intercepts every request made to the server, extracts information 5.1.1 from it and saves it to a database. At this point, the presented solution can log the complete request information. So far this implementation could have also been implemented

by parsing verbose server log files, but by extending this middleware it is also possible to capture additional information that is not available to the web server component.
Another middleware method of the same class captures a unique identifier of

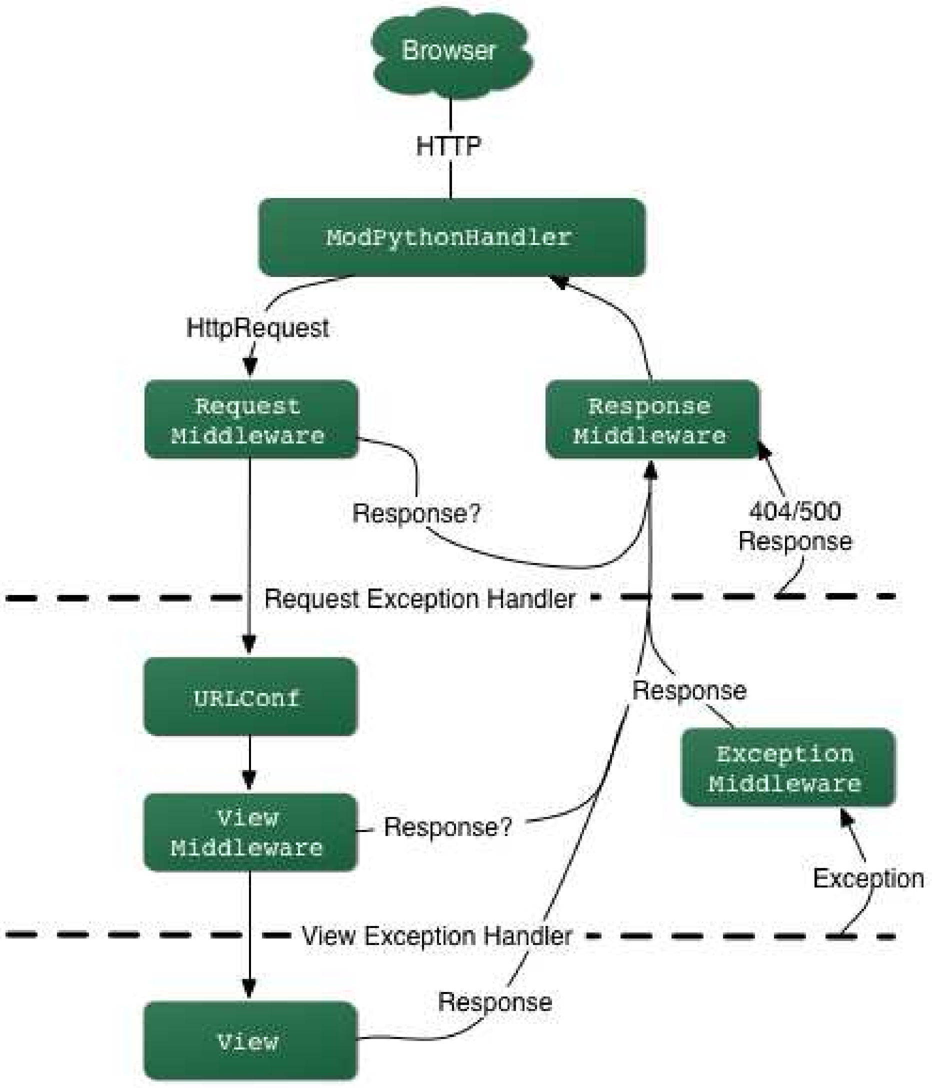


Figure 7: How the django framework processes a HTTP request. [29]

the view that is called to serve the request.
On returning the response the corresponding method of the middleware is invoked. Here all the information is commited to the database backend and (if necessary) a custom cookie is set that tracks the users interaction with the website throughout the session.
This is the complete list of properties that can be captured by the middleware:

- the resource being requested,
- HTTP request headers,
- the request method (POST/GET),
- any GET or POST data (if available),
- the response code that was returned for the request,
- the response headers,
- the view that was called to process the request, and
- a list of templates that were used for rendering the response.

For a detailed description of the middleware please refer to Appendix B.

5.1.2 Capturing object manipulations.

The implementation for this part is again a django middleware class. A custom eventlistener object is assigned to each request that subscribes to signals that are sent by the framework at various stages throughout the workflow. These signals, as described in [46], can be sent anywhere in the code and a dispatcher component routes these notifications to any methods or objects that subscribe to them, making this approach a convenient solution for decoupling tasks that need to be performed before or after an action has taken place, from the components where these actions are actually invoked.
The eventlistener component registers and waits for signals that are raised every time an (database) object is created, updated, or deleted. The signals are handled synchronously, meaning that the execution of the code that sent the signal is postponed until the signal handler has finished. I can exploit this fact to create copies of objects before they are saved to or deleted from the backend storage, allowing us to later restore any object to a previous

state. Note that these objects usually correspond to database entries, but this is not necessarily so. For example, objects could also be stored in memory, using a mechanism such as memcached [35]. It is however necessary that modifications of these objects omit a signal that can be caught by this implementation.
Each request to the application now triggers the creation of a new state and, depending on the actions performed within the application, any number of object states, which are serialized copies of the original object. Take a blogging application as an example where a user can create comments, the request that triggers the storage of the comment to the database would create a state, that when rolled back would now delete the object from the database. When rolling back to a state, all intermediate states that were created in between are reverted as well as all attached objectstates. I had to apply a small patch to the framework that adds a small number of signals to make the implementation aware that intermediate tables that are used for many to many relationships in the database design are being altered. Out of the box this functionality is not provided by django, but this was necessary to fully capture the state that an application is in, especially in cases where no objects are altered but only these intermediate tables are affected.
After a request has been processed, but before returning the response to the clients machine all state information is saved to a persistent storage location (in this case again a database).

5.2 Altering the workflow of web application scanners

With the help of the properties described in 5.1 it is now possible to replay real user interactions, that will be used as a basis for the advanced fuzzing techniques described in this thesis.

5.2.1 Replaying use cases

Once a use case, which consists of a series of requests, has been collected, it can be used for replaying. To this end, I have developed a test case replay component based on twill [54], a testing tool for web applications. This component analyzes a page and attempts to find the form elements that need to be filled out, based on the previously submitted request data.
This process is more robust than simply replaying the complete request that was recorded because it can leave values in hidden fields and read-only ele-

ments untouched, while replaying those values that were entered into editable form elements.
Using some of these properties it is possible to recreate a user session and replay it. In order to simply replay HTTP sessions it would be enough to capture the request/response pairs using some other mechanism that doesn't require server side support and this is also the approach that security testing tools use, but as I will show later some of the framework specific properties that were captured along with the usual data can be put to good use when testing a site.
For simple operations in the process of navigating the tested site, such as the clicking of links the test runner instructs the browser to find and subsequently follow the link described in the testcase. The process of form submission is naturally more involving and in order to reach a certain level of robustness within the testcases a special form filler takes care of these steps by examining the data recorded in the test case and the currently browsed page. It subsequently tries to find a suitable form within the page that matches the given user data. This is more than a simple replay of user session as the implementation and also the testing tool takes care of applying the correct values to forms that are being filled out and finally submitted. This means that depending on the type of HTML form widget encountered on the page the test browser decides whether to copy the supplied user values from the (known good) form submission data or to make it's own decision on what the value should be. This decision is based on a number of rules that are laid out to generate valid form submissions that the server is likely to accept.
From this knowledge it then creates a HTTP Request that can be submitted to the server. The test browser is instructed to place the request and handle the response. A optional callback function reports request and response objects to external handlers. This function is used later on in order to create fuzzable requests.

5.2.2 Fuzzer component

An important component of the vulnerability scanner is the fuzzer. The task of the fuzzer component is to expose each entry point that it finds to a set of malformed inputs that can expose XSS vulnerabilities. Typically, it also features a web spider that uses a certain page as a starting point to reach other parts of the application, checking each page that is encountered.
Because the focus of this work is not on the fuzzer component but on tech-

niques that can help to make this fuzzer more effective, I decided to use an existing web application testing tool. The choice was made for the "Web Application Attack and Audit Framework," or shorter, w3af [55], mainly because the framework itself is easy to extend and actively maintained.
W3af works in what the developers call phases [45]. The first phase is called discovery. This phase crawls an application to find entry points. To this end, the scanner follows links and attempts to fill out forms, so as to gather as much information as possible. As the experiments demonstrate (in Section 6.3), despite the presence of the simple web crawler, the effectiveness of the w3af scanner can be significantly improved when using the techniques proposed within this thesis.
After the discovery phase, the audit phase is launched. This phase searches for vulnerabilities in the endpoints discovered in the previous phase. For the audit phase, there are numerous plug-ins available that focus on different classes of vulnerabilities, ranging from cross-site request forgery detection to buffer overflows. As cross-site scripting the most common vulnerability found in web applications today [11], I chose to enable this plug-in for the experiments. Of course, more plug-ins could be easily added.
One problem I had with the implementation as it is, is that these phases are completely decoupled and it is not easy for a plug-in running in one part of the application to instruct another plug-in to do some work. A simple wrapper was created that allows the easy usage of the XSS plug-in from the scripts that control the guided fuzzing phases.

6 Evaluation

For the evaluation of the approaches, I installed three publicly available, real-world web applications based on Django (SVN Version 6668):

- The first application was a blogging application, called Django-basic-blog [23]. I did not install any user accounts, as there was no functionality on the site that required an authenticated session. Initially, the blog was filled with three articles. Comments were enabled for each article, and no other links were present on the page. That is, the comments were the only interactive component of the site.
- The second application was a forum software, called Django-Forum [44]. To provide all fuzzers with a chance to explore more of the application, every access was performed as coming from a privileged user account, which was ensured by a server side middleware component. Thus, each scanner was making requests as a user that could create new threads and post replies. Initially, a simple forum structure was created that consisted of three forums.
- The third application was a web shop, the Satchmo online shop 0.6 [47]. This site was much larger than the previous two applications, and, therefore, more challenging to test. The online shop was populated with the test data included in the package, and one user account was created.

I selected these three programs because they represent common archetypes of applications on the Internet. The test setup was run on Apache 2.2.4 (with pre-forked worker threads) and mod_python 3.3.1. Note that before a scanner was tested on a site, the application was restored to its initial state, so as to have level playgrounds for all applications.

6.1 Test Methodology

These three web applications were tested for XSS vulnerabilities by three freely available scanners as well as the tool presented in this thesis. The 3rd party scanners used were Burp Spider 1.21 [17], w3af spider [55], and Acunetix Web Vulnerability Scanner 5.1 (Free Edition) [12].

Each scanner is implemented as a web spider that can follow links on web pages. All scanners also have support for filling out forms and, with the exception of the burp spider, a fuzzer component to check for XSS vulnerabilities. For each page that is found to contain an XSS vulnerability, a warning is issued. In addition to the three vulnerability scanners and the presented tool, I also included the statistics of a very simple web spider into the tests. This web spider serves as the lower bound on the number of pages that should be found and analyzed by each vulnerability scanner.
When testing my tool, I first recorded a simple use case for each of the three applications. The use cases included posting a comment for the blog, creating a new thread and a post on the forum site, and purchasing an item in the online store. Then, I executed the system in one of three modes. First, guided fuzzing was used. In the second run, I used extended, guided fuzzing (together with application feedback). Finally, I scanned the program using stateful fuzzing.

Reference Spider This is a very simple self written webspider, that follows the first link found on a page and puts the location string onto a stack. It does so until it can't find any new locations to visit and then backtracks to the previous page until it has found all URLs. It does not fill out forms or parse anything but <a> tags with href attributes, so its presence in the tables is mainly for having a base line.

Burp Spider This is the spider component of the burpsuite, which has some basic form filling skills and also searches the source of the response for useable URLs (such as those embedded in javascript code). The default configuration was used for this spider, with the exceptions that the proxy option was deactivated and the maximum link depth was turned off as well as the option to search for session dependant pages. Furthermore the form filling option was activated and the maximum submissions allowed for each form was set to 10, the default behavior being to prompt the user for input. The version of the spider was 1.21 which comes with version 1.1 of the suite.

w3af The web spider discovery and XSS audit plug-in of the w3af suite. The spider has some simple functions for filling out forms and can therefore find more locations than the reference spider. The webSpider plug-in was used for finding injection points, which auto-enables another plug-in name allowedMethods. The number of checks was set to the maxim um of 10. For all tests the svn revision 800 was used.

Acunetix Web Vulnerability Scanner (Free Edition) The free edition of a commercial tool, that can check for various vulnerabilities in web applications. This edition is limited to auditing for XSS vulnerabilities and hence, served the purpose well. The default configuration was used besides enabling the Extensive Scan feature , optimizing for mod_python and checking for stored XSS. The version in use reported itself as 5.1 (Build 20080313).

6.2 Measuring the effectiveness of scanners

There are different ways to assess the effectiveness or coverage of a web vulnerability scanner. One metric is clearly the number of vulnerabilities that are reported. Unfortunately, this number could be misleading because a single program bug might manifest itself on many pages. For example, a scanner might find a bug in a form that is reused on several pages. In this case, there is only a single vulnerability, although the number of warnings could be significantly larger. Thus, the number of unique bugs, or vulnerable *injection points*, is more representative than the number of warnings.

Another way to assess coverage is to count the number of *locations* that a scanner visits. A location represents a unique, distinct page (or, more precisely, a distinct URL). Of course, visiting more locations potentially allows a scanner to test more of the application's functionality. Assume that, for a certain application, Scanner A is able to explore significantly more locations than Scanner B. However, because Scanner A misses one location with a vulnerability that Scanner B visits, it reports fewer vulnerable injection points. In this case, one might still conclude that Scanner A is better, because it achieves a larger coverage. Unfortunately, this number can also be misleading, because different locations could result from different URLs that represent the same, underlying page (e.g., the different pages on a forum, or different threads on a blog).

Finally, for the detection of vulnerabilities that require the scanner to store malicious input into the database (such as stored XSS vulnerabilities), it is more important to create many different database objects than to visit many locations. Thus, I also consider the number and diversity of different (database) objects that each scanner creates while testing an application.

The coverage a web application vulnerability scanner achieves is a useable metric to measure it's effectiveness. Clearly, this is true for discovering reflected XSS vulnerabilities, assuming that the detection algorithm can properly determine if the injected code is executable by the browser. For the detection of stored XSS, on the other hand, it is essential that the spider can actually generate datasets that are then saved in a backend storage by the web application (i.e., the injected script is retrieved at a later point in time). By monitoring the activities of the spiders on the server side, it is possible to create listings of generated objects for each test run. In this setup, these numbers directly correspond the number of database objects created by the actions of a spider. Hence, the more objects a spider is able to create, the higher its coverage.

6.3 Experimental Results

In this section, I present and discuss the results that the different scanners achieve when analyzing the three test applications. For each application, I list the number of locations that the scanner has visited, the number of reported vulnerabilities, the number of injection points (unique bugs) that these reports map to, and the number of relevant database objects that were created.

	Locations	POST/GET	XSS Warnings		Injection Points	
		Requests	Reflected	Stored	Reflected	Stored
Spider	4	4	-	-	-	-
Burp Spider	8	25	0	0	0	0
w3af	9	133	0	0	0	0
Acunetix	9	22	0	0	0	0
Use Case	4	4	-	-	-	-
Guided Fuzzing	4	64	0	1	0	1
Extended Fuzz.	6	189	0	1	0	1
Stateful Fuzz.	6	189	0	1	0	1

Table 1: Scanner effectiveness for blog application.

Blogging application. Table 1 shows the results for the simple blog application. Compared to the simple spider, one can see that all other tools have reached more locations. This is because all spiders requested the root of

	Comments created
Spider	-
Burp Spider	0
w3af	0
Acunetix	0
Use Case	1
Guided Fuzzing	12
Extended Fuzz.	12
Stateful Fuzz.	12

Table 2: Object creation statistics for the blogging application.

each identified directory. When available, these root directories can provide additional links to pages that might not be reachable from the initial page. As expected, it can be seen that extended, guided fuzzing reaches more locations than guided fuzzing alone, since it attempts to explore the application in breadth. Moreover, there is no difference between the results for the extended, guided fuzzing and the stateful fuzzing approach. The reason is that, for this application, invoking the fuzzer does not interfere with the correct replay of the use case.
None of the three existing scanners was able to create a valid comment on the blogging system. This was because the posting process is not straightforward: Once a comment is submitted, the blog displays a form with a preview button. This allows a user to either change the content of the comment or to post it. The problem is that the submit button (to actually post the message) is not part of the page until the server-side validation recognizes the submitted data as a valid comment. To this end, both comment fields (name and comment) need to be present. The commenting system is similar to the one described in 3.3. Here, the advantage of guided fuzzing is clear. Because the presented system relies on a number of previously recorded test cases, the fuzzer can correctly fill out the form and post a comment. This is beneficial, because it is possible to include malicious javascript into a comment and expose the stored XSS vulnerability that is missed by the other scanners.

Forum application. For the forum application, the scanners were able to generate some content, both in the form of new discussion threads and replies. Table 3 shows that while Burp Spider [17] and w3af [55] were able to create new discussion threads, only the Acunetix scanner managed to post

	Locations	POST/GET Requests	XSS Warnings		Inject. Points	
			Reflect	Stored	Reflect	Stored
Spider	8	8	-	-	-	-
Burp Spider	8	32	0	0	0	0
w3af	14	201	0	3	0	1
Acunetix	263	2,003	63	63	0	1
Use Case	6	7	-	-	-	-
Guided Fuzzing	16	48	0	1	0	1
Extended Fuzz.	85	555	0	3	0	1
Stateful Fuzz.	85	555	0	3	0	1

Table 3: Scanner effectiveness for the forum application.

	Threads created	Replies posted
Spider	-	-
Burp Spider	0	0
w3af	29	0
Acunetix	687	1,486
Use Case	1	2
Guided Fuzzing	12	22
Extended Fuzz.	36	184
Stateful Fuzz.	36	184

Table 4: Object creation statistics for the forum application.

replies as well. w3af correctly identified the form's action URL to post a reply, but failed to generate valid input data that would have resulted in the reply being stored in the database. However, since the vulnerability is caused by a bug in the routine that validates the thread title, posting replies is not necessary to identify the flaw in this program.
Both the number of executed requests and the number of reported vulnerabilities differ significantly between the vulnerability scanners tested. It can be seen that the Acunetix scanner has a large database of malformed inputs, which manifests both in the number of requests sent and the number of vulnerabilities reported. For each of the three forum threads, which contain a link to the unique, vulnerable entry point, Acunetix sent 21 fuzzed requests. Moreover, the Acunetix scanner reports each detected vulnerability twice. That is, each XSS vulnerability is reported once as reflected and once as stored XSS. As a result, the scanner generated 126 warnings for a single bug. w3af, in comparison, keeps an internal knowledge base of vulnerabilities

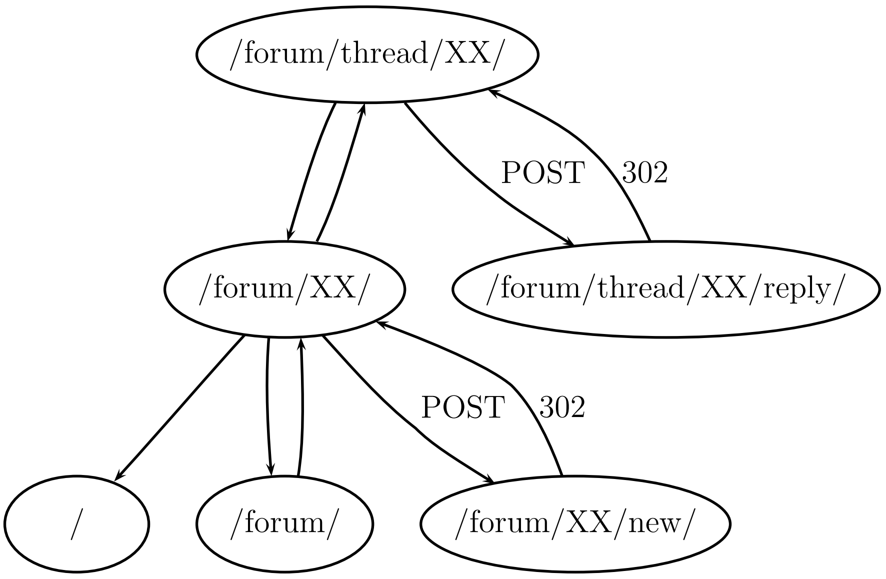


Figure 8: A user interaction example of the forum application.

that it discovers. Therefore, it reports each vulnerability only once (and the occurrence of a stored attack replaces a previously found, reflected vulnerability). However, it also reports three vulnerabilities, one for the occurrence of the vulnerable injection point in each forum.
The results show that all the applied techniques were able to find the vulnerability that is present in the forum application. Similar to the Acunetix scanner (but unlike w3af), they were able to create new threads and post replies. Again, the extended, guided fuzzing was able to visit more locations than the guided fuzzing alone (it can be seen that the extended fuzzing checked all three forum threads that were present initially, while the guided fuzzing only analyzed the single forum thread that was part of the recorded use case). Moreover, the fuzzing phase was not interfering with the replay of the use cases. Therefore, the stateful fuzzing approach did not yield any additional benefits.
The extended approach successfully submitted all forms found on the initial site, thereby improving the raw numbers of created objects and vulnerabilities and bringing them on par with the results of the commercial scanner.
Figure 8 shows the user interaction with the application as it was used for the use case that the results for the guided fuzzing approaches are based upon.

	Locations	POST/GET Requests	XSS Warnings		Injection Points	
			Reflected	Stored	Reflected	Stored
Spider	18	18	-	-	-	-
Burp Spider	22	52	0	0	0	0
w3af	21	829	1	0	1	0
Acunetix #1	22	1,405	16	0	1	0
Acunetix #2	25	2,564	8	0	1	0
Use Case	22	36	-	-	-	-
Guided Fuzzing	22	366	1	8	1	8
Extended Fuzz.	25	1,432	1	0	1	0
Stateful Fuzz.	32	2,078	1	8	1	8

Table 5: Scanner effectiveness for the online shopping application.

Online shopping application. The experimental results for the online shopping application are presented in Tables 5 and 6. Table 5 presents the scanner effectiveness based on the number of locations that are visited and the number of vulnerabilities that are detected, while Table 6 compares the number of database objects that were created by both the Acunetix scanner and the presented approaches. Note that the Acunetix scanner offers a feature that allows the tool to make use of login credentials and to block the logout links. For this experiment, two test were run with the Acunetix scanner: The first run (#1) as an anonymous user and the second test run (#2) by enabling this feature.

Both w3af and Acunetix identified a reflected XSS vulnerability in the login form. However, neither of the two scanners was able to reach deep into the application. As a result, both tools failed to reach and correctly fill out the form that allows to change the contact information of a user. This form contained eight stored XSS vulnerabilities, since none of the entered input was checked by the application for malicious values. However, the server checked the phone number and email address for their validity and would reject the complete form whenever one of the two values was incorrect. Other server side checks would also result in rejection of the form submission. A small user interaction for the account management functions of the e-commerce application can be seen in Figure 9. The dotted line is a redirect that is issued by the server after a successful update of the users profile. Only the approaches based on guided fuzzing were able to test this vector.

In contrast to the existing tools, guided fuzzing was able to analyze a large part of the application, including the login form and the user data form.

Object Class	Acunetix #1	Acunetix #2	Use Case	Guided Fuzzing	Extended Fuzzing	Stateful Fuzzing
OrderItem	-	-	1	1	-	2
AddressBook	-	-	2	2	-	7
PhoneNumber	-	-	1	3	-	5
Contact	1	-	1	1	1	2
CreditCardDetail	-	-	1	1	-	2
OrderStatus	-	-	1	1	-	1
OrderPayment	-	-	1	1	-	2
Order	-	-	1	1	-	2
Cart	2	1	1	1	3	3
CartItem	2	1	1	1	5	5
Comment	-	-	1	21	11	96
User	1	-	1	1	1	1

Table 6: Object creation statistics for the online shopping application.

Thus, this approach reported a total of nine vulnerable entry points. In this experiment, one can also observe the advantages of stateful fuzzing. With extended, guided fuzzing, the fuzzing step interferes with the proper replay of the use case (because the fuzzer logs itself out and deletes all items from the shopping cart).
The stateful fuzzer, on the other hand, allows the scanner to explore a broad range of entry points, and, using the snapshot mechanism, keeps the ability to replay the test case. The number of database objects created by the different approaches (as shown in Table 6) also confirms the ability of my techniques to create a large variety of different, valid objects, a result of analyzing large portions of the application Furthermore simple, but usually irreversible operations like confirming the order can be undone.

6.4 Discussion

All vulnerabilities that were found during the experiments were previously unknown, and were reported to the developers of the web applications. The results show that the novel fuzzing techniques consistently find more (or, at least, the same amount) of bugs than other open-source and commercial scanners. Moreover, it can be seen that the different approaches carry out meaningful interactions with the web applications, visiting many locations and creating a large variety of database objects. The different techniques

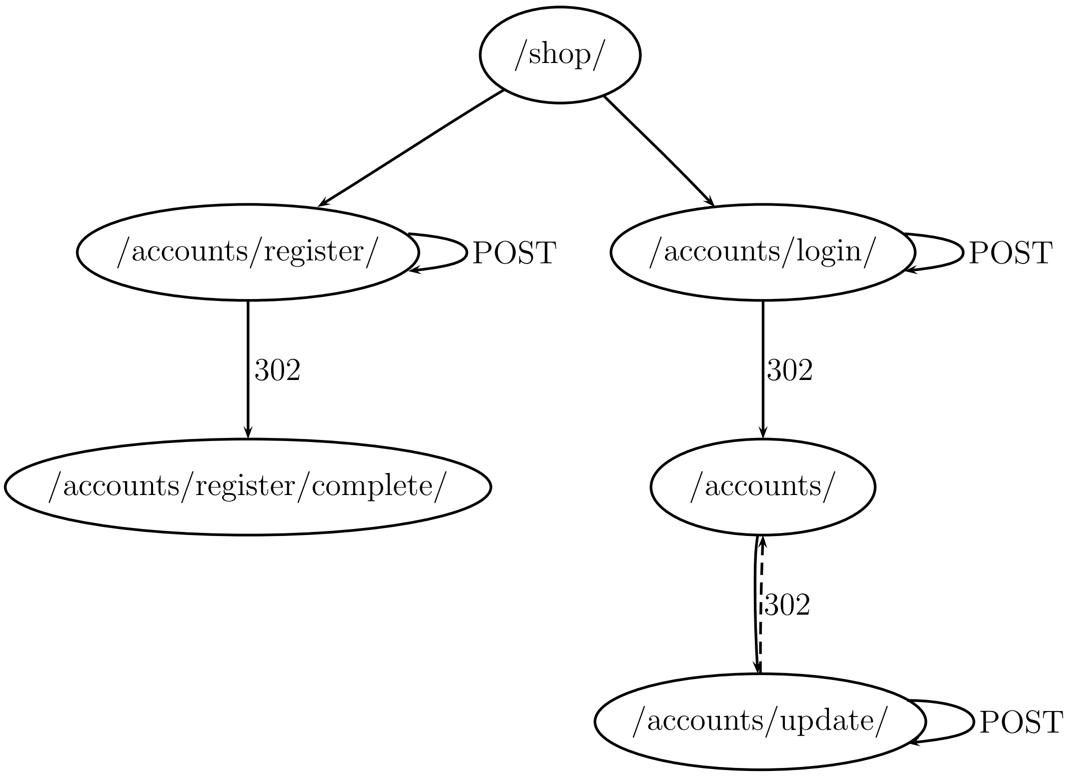


Figure 9: Account actions for the e-commerce application

exhibit different strengths.
The guided fuzzing approach reaches exactly the same coverage as the use cases, but can in some circumstances, where the application allows submission of the same form over and over, lead to a higher number of created database objects as the request pairs are tainted as they are sent to the server. This approach can be abstracted to a level where manual usage of the application can lead to automatic fuzzing being executed in the background, which is similar to the way the proxying functionality of web application vulnerability scanners work, with the advantage that this techniques generates reuseable black box testcases that can be used for deployment testing, an important aspect of web application testing.
Extending guided fuzzing has the potential to increase the breadth within the tested application and can be used to discover additional endpoints that are not covered by the test cases. This has obvious advantages and often increases the raw numbers of fuzzed endpoints, on the other hand the e-commerce application is a good example of where this approach has its limitations. When the state of an application matters this technique can lead to unwanted results and the penetration tester might consider switching to guided fuzzing, as the intermediate broad auditing stages hinder the use cases from completing. This could be avoided by providing the tool with configuration options that define which links to follow and which to ignore (such as log-out links, delete buttons etc.), but this partly diminishes the advantages gained by guided fuzzing. As can be seen in the forum application the knowledgebase adds value to the test results as all forms found on the site can be submitted, without the need to guess any form values, eventhough other applications fare well with guessing values for the simple forms found within this application.
The advantages of stateful fuzzing become especially apparent when the tested application is more complex and sensitive to the fuzzing steps, because the technique allows to undo unwanted operations that might have been executed by the fuzzing component and would render the testcase useless in the case of the extended guided fuzzing approach. So this clearly adds stability to the previous method at the expense of a larger overhead to the application itself.

7 Related Work

Concepts such as vulnerability testing, test case generation, and fuzzing are well-known concepts in software engineering and vulnerability analysis [15, 16, 26]. When analyzing web applications for vulnerabilities, black-box fuzzing tools [12, 17, 55] are most popular. However, as shown by the experiments, they suffer from the problem of test coverage. Especially for applications that require complex interactions or that expect specific input values to proceed, black-box tools often fail to fill out forms properly. As a result, they can scan only a small portion of the application. This is also true for SecuBat [34], a web vulnerability scanner that was previously developed at the Secure Systems Lab. SecuBat can detect reflected XSS and SQL injection vulnerabilities. However, it cannot fill out forms and, thus, was not included in the experiments.

In addition to web-specific scanners, there exist a large body of more general vulnerability detection and security assessment tools. Many of these tools (e.g., Nikto [39], Nessus [53]) rely on a repository of known vulnerabilities that are tested. The presented system, in contrast, aims to discover unknown vulnerabilities in the application under analysis. Besides application-level vulnerability scanners, there are also tools that work at the network level, e.g., nmap [32]. These tools can determine the availability of hosts and accessible services. However, they are not concerned with higher-level vulnerability analysis. Other well-known web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [48] and Huang et al.'s [31] vulnerability detection tool that automatically executes SQL injection attacks. Moreover, there are a large number of static source code analysis tools [33, 51, 56] that aim to identify vulnerabilities.

A field that is closely related to this work is automated test case generation. The methods used to generate test cases can be generally summarized as random, specification-based [40, 42], and model-based [41] approaches. Fuzzing falls into the category of random test case generation. By introducing use cases and guided fuzzing, it is possible to improve the effectiveness of random tests by providing some inputs that are likely valid and thus, allow the scanner to reach "deeper" into the application.

A well-known application testing tool, called WinRunner, allows a human tester to record user actions (e.g., input, mouse clicks, etc.) and then to replay these actions while testing. This could be seen similar to guided fuzzing, where inputs are recorded based on observing real user interaction.

However, the testing with WinRunner is not fully-automated. The developer needs to write scripts and create check points to compare the expected and actual outcomes from the test runs. By adding automated, random fuzzing to a guided execution approach, it is possible to combine the advantages provided by a tool such as WinRunner with black-box fuzzers. Moreover, the system provides techniques to generalize from a recorded use case.
Finally, a number of approaches [18, 27, 38] were presented in the past that aim to explore the alternative execution paths of an application to increase the analysis and test coverage of dynamic techniques. The work presented in this thesis is analogous in the sense that the techniques aim to identify more code to test. The difference is the way in which the different approaches are realized, as well as their corresponding properties. When exploring multiple execution paths, the system has to track constraints over inputs, which are solved at branching points to determine alternative paths. The presented system however leverages known, valid input to directly reach a large part of an application. Then, a black-box fuzzer is started to find vulnerabilities. This provides better scalability, allowing the tool to quickly examine large parts of the application and expose it to black-box tests.
To the best of my knowledge, to date, no blackbox approaches have been proposed in literature that are able to detect stored XSS vulnerabilities.

8 Conclusions

In this thesis, I presented a web application testing tool to detect reflected and stored cross-site scripting (XSS) vulnerabilities. The core of the system is a black-box vulnerability scanner. Unfortunately, black-box testing tools often fail to test a substantial fraction of a web application's logic, especially when this logic is invoked from pages that can only be reached after filling out complex forms that aggressively check the correctness of the provided values. To allow the presented scanner to reach "deeper" into the application, I introduced a number of techniques to create more comprehensive test cases. One technique, called guided fuzzing, leverages previously recorded user input to fill out forms with values that are likely valid. This technique can be further extended by using each step in the replay process as a starting point for the fuzzer to explore a program more comprehensively. When feedback from the application is available, the scanner can reuse the recorded user input for different forms during this process. Finally, I introduced stateful fuzzing as a way to mitigate potentially undesirable side-effects of the fuzzing step that could interfere with the replay of use cases during extended, guided fuzzing. I have implemented all these use-case-driven testing techniques and analyzed three real-world web applications. The experimental results demonstrate that these approaches are able to identify more bugs than several open-source and commercial web vulnerability scanners.

8.1 Future work

This leaves the question how to apply these approaches to existing vulnerability scanners. A sample implementation was given that makes use of an existing fuzzer and yields good results, but since w3af, due to its phases, is not designed to support the guided fuzzing approach the presented solution is limited to using the XSS plug-in. It would be desireable to integrate such functionality into an existing tool, but the way current vulnerability scanners work (strict separation of discovery and audit phase) makes it hard to do so without re-implementing the core functionality.

Improving existing solutions. In my opinion, backed by the results of the experiments, modern web applications can not be fully tested by enumerating their endpoints and subsequently fuzzing them but need to be treated and subsequently tested as the stateful applications that they are. Penetration testers know about the shortcomings of these tools and can leverage these by extensive test configuration or manual testing methods. Unaware users on the other hand are given a false sense of security when their scanner reports no vulnerabilities, because it has simply failed to supply sufficiently correct input to trigger the vulnerability or the spider component was unable to reach certain parts of the application that might be vulnerable. At the very least users of the scanners should be made aware of these limitations and the tools should issue warnings to reflect this. As a quick fix I would propose to implement some heuristics to determine the success of automated form filling. The user should be issued a warning when a submitted form reappears in the reply from the server, as this might indicate that the supplied input did not pass server side validation.

Another obvious place for improvement is the test runner itself. In its current state it is useable for a large number of web applications, but fails on javascript-heavy sites, because twill, the underlying tool of the browsing component, does not execute javascript.

Improving the test runner. In order to reliably detect cross-site scripting vulnerabilities on web pages that make use of javascript to generate content it is necessary to integrate a javascript engine that will execute the code in a similar fashion that a browser would. Perhaps the best way to combine functional black-box tests with the security auditing methods described in this thesis would be with the help of a in-browser tool such as Selenium [4]. This would obviously have a negative impact on the speed of the auditing process, but is necessary when confronted with web applications that make extensive use of javascript.

As the techniques that were presented in this thesis are strongly related to testing of software in general and black-box testing in particular it would be interesting to see implementations of security fuzzers that can be directly used inside native testing tools for web applications.

Integration with existing test suites. As described in [6], django comes with its own test client to test applications without the overhead of using the HTTP protocol, but by interfacing directly via wsgi [8, 43] with the application. Similar tools are available for other programming languages and frameworks, some of them even language independent [4, 5], so the the tools available to support black-box testing of web applications are in place. But there is a gap between functional testing tools and security auditing tools that is hard to bridge with the current solutions. This thesis presents a way to combine traditional testing techniques, namely use cases, with fuzzing and auditing strategies for penetration testing. Combining security and functional tests is in many cases a feasible way to fully test a web application and it would be desireable if this were respected in future tools.

Appendix

A Models

A.1 Functions

safedumps(*dic*, *types*=`_canpickle`)

A helper function for pickling unsafe dicts. It checks for simple python types inside the dicts values and removes key value pairs that can not be pickled correctly.

Parameters

`dic:`	a python dictionary to be pickled.
`types:`	an optional iterable of python types that will be used to check for compatibility with pickle.

Return Value

a pickled string of the possibly shortened dict.

revert(*self*)

Reverts all many to many relationships that may have existed between 2 models.

A.2 Class Session

django.db.models.Model
    └ **models.Session**

The Session model.
Handles session ids that are used to track a users movement on the site.

A.2.1 Methods

__unicode__(*self*)

A.3 Class SimpleStep

django.db.models.Model ┐
models.SimpleStep

The model class that represents a single request-response pair.
It's attributes represent the properties that are captured for each request response pair. Together with the Session model this enable the tracking of user interactions inside the application.

A.3.1 Methods

__init__(*self*, **args*, ***kwargs*)

__unicode__(*self*)

save(*self*)

Pickles it's attributes and saves the resulting object to the database backend.

A.4 Class State

django.db.models.Model ┐
models.State

The primary state model that all state related models link to.
A SimpleStep object links to exactly one State, which in turn consists of a number of ModelState and SimpleM2MState objects

A.4.1 Methods

revert(*self*)

The main revert() method of a state object.

I calls revert on all related ModelState and Many2ManyState objects. When calling this method, it is adviseable to do so within a single database transaction, as dependencies are only guaranteed to be resolved when the state has been completely reverted.

A.5 Class ModelState

django.db.models.Model

models.ModelState

This model keeps track of the state of models (database entries). It stores the action that was taken during a particular step and saves a serialized copy of the object. This also takes into account ForeignKeyFields, as this property is stored inside a field of the model.

A.5.1 Methods

__unicode__(*self*)

restore_model(*self*)

load_model(*self, data*)

Loads and returns a model, based on the serialized version of it.

revert(*self*)

Reverts models to their previous state.

Depending on the action that was taken by the application it restores the state of a models instance. If an instance was deleted, it will be recreated. If it was updated, the changes will be undone. If a new instance was created, it will be deleted.

A.6 Class SimpleM2MState

django.db.models.Model

models.SimpleM2MState

The model that saves many to many relationships that can be altered within a request response pair.
For Many2ManyFields we need a special model that stores all relations that are changed during a operation. Relations can be either added or deleted. A new SimpleM2MState model is created for each instance whose relations are changed.

B Tracker middleware

B.1 Functions

makeSessionId(*st*)

Creates a unique session id, from a string and the current timestamp.

B.2 Class SimpleClientTracker

object

middleware.simpletracker.SimpleClientTracker

The SimpleClientTracker middleware class applies to each request and monitors the user input (request) as well as the response. Usage of this middleware usually results in one database insert operation for each request, so if this is acceptable it can be used on production machines to gather use cases.
It attaches a SimpleTracker object to each request that is then populated in the appropriate methods of this middleware class.
This middleware monitors and save the following properties of a request-response pair:
Request:

- request path
- POST and GET data
- headers

View:

- name of the view called
- list of templates used to render the response (if any)

Response:

- response status code
- response headers

B.2.1 Methods

process_request(*self*, *request*)

Depending on the order of the middleware this method is called after the request has been received.

It first checks for a existing session cookie and creates one if needed. If template tracking is enabled it monkeypatches djangos templating system to record the templates in use by attaching a signal to the template loader that is fired every time the templating system renders a template.

For production machines turn off the template system patching.

process_view(*self*, *request*, *view_func*, *view_args*, *view_kwargs*)

This function is called just before the actual view is called. The data collected from this method can be used to map out the application in another way than simply collecting URLs.

process_response(*self*, *request*, *response*)

Before returning the generated response this method saves the SimpleTracker instance to the database.

C TestClient middleware

C.1 Class TestClientMiddleware

object ─┐

middleware.testclient.TestClientMiddleware

This middleware must be put after the tracker middleware, but should be placed before any middleware that changes objects within the database.
This middleware first checks if the requesting client is a testclient. If not, then the request is skipped and return immediately. Otherwise it creates a new state and attaches a StateMachine object to the request (request._SM). The StateMachine is then connected to the model signals to listen for changes to objects that are affected by the request. Furthermore it sets a custom header in the response that the client can use to reset the applications state.
The middleware connect to the following signals, defined in django.db.models.signals:

- pre_save, in the case of an update the object that emits this signals is serialized, before the changes are written to the database.
- post_save, with the help of this signal the implementation is able to determine what objects were created during a request.
- pre_delete, is needed to intercept delete operations, so that deleted objects can be later restored.

The reason that both save signals need to be listened for by the middleware is that it is in some cases impossible to determine if an object is created or updated when listening for the pre_save signals, whereas the post_save signal has an argument created that indicates if the objects was created in the database.
The StateMachine that is attached by this middleware uses in an internal cache so that it doesn't collect multiple versions of the same object during one state, but only the first alteration of it, which is important in the case of a operation running multiple times on the same object.

C.1.1 Methods

process_request(*self*, *request*)

This function is called after the request has been received (but before any view is called). Depending on the order of the middleware no interaction with the database should have occured at this point, with the exception that other middlewares might have changed objects. Therefore it is important that this middleware appears early within the middleware settings in the global settings file.

The request object stays the same during the whole workflow of django so this is the best place to attach the StateMachine.

process_response(*self*, *request*, *response*)

This method is run after the application has finished it's processing.

The StateMachine is instructed to dump it's collected data to a persistent storage. Also the custom header containing the state-id is applied to the response.

References

[1] Django project homepage - frequently asked questions.

[2] Open Source Web Testing Tools in Java. `http://java-source.net/open-source/web-testing-tools`.

[3] PythonTestingToolsTaxonomy. `http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy#WebTestingTools`.

[4] Selenium Homepage. `http://selenium.openqa.org/`.

[5] Selenium-RC Homepage. `http://selenium-rc.openqa.org/`.

[6] Testing Django Applications. `http://www.djangoproject.com/documentation/testing/`.

[7] Watir Homepage. `http://wtr.rubyforge.org/`.

[8] WSGI Homepage. `http://www.wsgi.org/wsgi`.

[9] XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[10] Cross-Site Scripting Worm Floods MySpace . `http://it.slashdot.org/it/05/10/14/126233.shtml?tid=172&tid=95&tid=220`, 2005.

[11] A. van der Stock. OWASP Top 10. `http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf`, 2007.

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

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

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

[15] B. Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, 1984.

[16] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

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

[18] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *ACM Conference on Computer and Communication Security*, 2006.

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

[20] S. Cook. A Web developer's guide to cross-site scripting . `http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.pdf`, 2003.

[21] David Hannson. Ruby on Rails. `http://www.rubyonrails.org/`, 2008.

[22] Django. The Web Framework for Professionals with Deadlines. `http://www.djangoproject.com/`, 2008.

[23] Basic Django Blog Application. `http://code.google.com/p/django-basic-blog/`.

[24] D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.

[25] The Mozilla Foundation. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

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

[27] P. Godefroid, N. Klarlund, and K. Sen. DART. In *Programming Language Design and Implementation (PLDI)*, 2005.

[28] Jeremiah Grossman. Technology alone cannot defeat web application attacks: Understanding technical vs. logical vulnerabilities. `http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1189767,00.html`, 2006.

[29] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Apress, 2007.

[30] RFC: Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`.

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

[32] Insecure.org. NMap Network Scanner. `http://www.insecure.org/nmap/`, 2008.

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

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

[35] memcached. memcached: A Distributed Memory Object Caching System. `http://twill.idyll.org/`, 2008.

[36] Mitre. Vulnerability Type Distributions in CVE. `http://cwe.mitre.org/documents/vuln-trends/index.html`.

[37] Mitre. Common Vulnerabilities and Exposures. `http://cve.mitre.org/`.

[38] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[39] Nikto. Web Server Scanner. `http://www.cirt.net/code/nikto.shtml`, 2008.

[40] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. *Second International Conference on the Unified Modeling Language*, 1999.

[41] J. Offutt and A. Abdurazik. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Third International Conference on UML*, 2000.

[42] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State-based Specifications. *Journal of Software Testing, Verification and Reliability*, 2003.

[43] Phillip J. Eby. PEP 333 - Python Web Server Gateway Interface. `http://www.python.org/dev/peps/pep-0333/`.

[44] R. Poulton. Django Forum Component. `http://code.google.com/p/django-forum/`.

[45] A. Riancho. w3af User Guide. `http://w3af.sourceforge.net/documentation/user/w3afUsersGuide.pdf`, 2007.

[46] Matt Riggott. Django signals . `http://www.mercurytide.co.uk/whitepapers/django-signals/`, 2006.

[47] Satchmo. `http://www.satchmoproject.com/`.

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

[49] WhiteHat Security. Web Application Security 101 . `http://www.whitehatsec.com/articles/webappsec101.pdf`, 2005.

[50] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook*. Wiley, 2007.

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

[52] Sun. Java Servlets. `http://java.sun.com/products/servlet/`, 2008.

[53] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2008.

[54] Twill. Twill: A Simple Scripting Language for Web Browsing. `http://twill.idyll.org/`, 2008.

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

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