Expanding Human Interactions for In-Depth Testing of Web Applications

Sean McAllister[1], Engin Kirda[2], and Christopher Kruegel[3]

[1] Secure Systems Lab, Technical University Vienna, Austria
sean@seclab.tuwien.ac.at
[2] Institute Eurecom, France
kirda@eurecom.fr
[3] University of California, Santa Barbara
chris@cs.ucsb.edu

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 significantly. 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.
In this paper, we present an automated testing tool that can find reflected and stored cross-site scripting (XSS) vulnerabilities in web applications. The core of our system is a black-box vulnerability scanner. This scanner is enhanced by techniques that allow one to generate more comprehensive test cases and explore a larger fraction of the application. Our experiments demonstrate that our approach is able to test more thoroughly these programs and identify more bugs than a number of open-source and commercial web vulnerability scanners.

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 [17] 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 [2, 15, 27, 32], 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 [11] 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 [1, 16, 19, 29]. 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 paper, we present techniques that improve the effectiveness of web vulnerability scanners. To this end, our scanner leverages input from real users as a starting point for its testing activity. More precisely, starting from recorded, actual user input, we generate test cases that can be replayed. By following a user's session, fuzzing at each step, we are able to increase the code coverage by exploring pages that are not reachable for other tools. Moreover, our 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.

We have implemented our techniques in a vulnerability scanner that can analyze applications that are based on the Django web development framework [8]. Our experimental results demonstrate that our tool achieves larger coverage and detects more vulnerabilities than existing open-source and commercial fuzzers.

2 Web Application Testing and Limitations

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 [10], 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. According to [26], the possible points of attack are the URL, the cookie, and the POST data contained in a request. 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.

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 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 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, we briefly discuss an example of how a fuzzer might fail on a simple use case.

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.

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 our experiments (details in Section 5.2) 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 our evaluation of existing vulnerability scanners, we 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 vary, depending on the application that is being tested and the implementation of the scanner.

3 Increasing Test Coverage

To address the limitations of existing tools, we 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. The first technique, described in Section 3.1, 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 3.2, 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 3.3, makes the second technique more robust in cases where the broad exploration interferes with the correct replay of a use case.

3.1 Increasing Testing Depth

One way to improve the coverage, and thus, the effectiveness of scanners, is to leverage actual user input. That is, we 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. 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, we record requests directly at the server, logging the names and values of all input parameters.

It is 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. 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. This approach raises some concerns towards the nature of the captured data. The penetration tester must be aware of the fact that user input is being captured and stored in cleartext form. This might be acceptable for some sites but not for others, because the unencrypted storage of sensitive information such as passwords and credit card numbers might at the very least be a security problem or even illegal. In these cases it is advisable to perform all input capturing and tests in a controlled testbed, which is in many cases the only sensible place to perform penetration tests.

Replaying input. Each use case consists of a number of steps that are carried out to reach a certain goal. For each step, we have recorded the requests (i.e., the input values) that were submitted. Based on 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, we attempt 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.

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). Of course security vulnerabilities can also be triggered by malicious input data within these hidden fields, but this is of no concern at this stage because the idea is to generate benign and valid input and then apply the attack logic to these values. Later on, during the attacking stage, the fuzzer will take care that all parameters will be tested.

Guided fuzzing. We 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, we can increase the depth that a scanner can reach into the application.

3.2 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 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, we do 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. We call this approach *extended, guided fuzzing*.

Extended, guided fuzzing has the potential to increase the number of entry points that a scanner can test. 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 we assume that the scanner has no knowledge and control of the inner workings of the application under test. In the following Section 3.3, we consider the case in which our test system can interact more tightly with the analyzed program. In this case, we are able to prevent the undesirable side effects (or interference) from the fuzzing phases.

3.3 Stateful Fuzzing

The techniques presented in the previous sections work independently of the application under test. That is, our 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, we consider the case where the scanner has some control over the application under test.

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. We call this process *stateful fuzzing*.

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, we would have to customize our 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. 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 presentation 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 presentation 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.

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, we are 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, we can, for example, 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, we insert a component 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 our experimental results presented in Section 5.

Application feedback. Given that stateful fuzzing already requires the instrumentation of the program under test, we should consider what additional information might be useful to further improve the vulnerability scanning process.

One piece of feedback from the application that we 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, we can increase the effectiveness of the extended, guided fuzzing process. To this end, we attempt to find a set of forms (or URLs) that all invoke the same function within the application. When we have previously seen user input for one of these forms, we 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 3.1), we record all input values that a user provides on each page. More precisely, for each URL that is requested, we store all the name-value pairs that a user submits with this request. In case the scanner can obtain application feedback, we also store the name of the program function that is invoked by the request. In other words, we record the name 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), we query the application to determine which function this URL maps to. Then, we search our collected information to see whether the same function was called previously by another URL. If this is the case, we examine the name-value pairs associated with this other URL. For each of those names, we attempt 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 we have encountered one case where a user submitted a reply, we 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.

4 Implementation Details

We developed a vulnerability scanner 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. Therefore, we were looking for a web development framework that supports the model-view-controller (MVC) scheme. Among the candidates were most popular web development frameworks, such as Ruby on Rails [7], Java Servlets [28], or Django [8], which is based upon Python. Since we are familiar with Python, we selected the Django framework. That is, we extended the Django framework such that it provides the necessary functionality for the vulnerability scanner. Our choice implies that we can currently 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 (i.e., with some additional engineering effort, we could use our techniques to test applications based upon other frameworks).

Capturing web requests. The first task was to extend Django 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. In Django, all incoming requests pass through two middleware classes before reaching the actual application code. One of these classes is a URL dispatcher class that determines the function that should be invoked. At this point, we can log the complete request information. Also, the URL dispatcher class provides easy access to the mapping between URLs and the functions that are invoked.

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, we have developed a small test case replay component based on twill [30], 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.

Capturing object manipulations. Our implementation uses the Django middleware classes to attach event listeners to incoming requests. These event listeners wait for signals that are raised every time an 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. We exploit this fact to create copies of objects before they are saved to the backend storage, allowing us to later restore any object to a previous state.

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 techniques that can help to make this fuzzer more effective, we decided to use an existing web application testing tool. The choice was made for the "Web Application Attack and Audit Framework," or shorter, w3af [31], mainly because the framework itself is easy to extend and actively maintained.

5 Evaluation

For our experiments, we 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 [9]. We did not install any user accounts. 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 [23]. To provide all fuzzers with a chance to explore more of the application, every access was performed as coming from a privileged user account. 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 [24]. This site was 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.

We selected these three programs because they represent common archetypes of applications on the Internet. For our experiments, we used Apache 2.2.4 (with preforked worker threads) and mod_python 3.3.1. Note that before a new scanner was tested on a site, the application was restored to its initial state.

5.1 Test Methodology

We tested each of the three aforementioned web applications with three existing web vulnerability scanners, as well as with our own tool. The scanners that we used were Burp Spider 1.21 [5], w3af spider [31], and Acunetix Web Vulnerability Scanner 5.1 (Free Edition) [1]. 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 Suite 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 our tool, we also included a very simple web spider into the tests. This self-written spider follows all links on a page. It repeats this process recursively for all pages that are found, until all available URLs are exhausted. This web spider serves as the lower bound on the number of pages that should be found and analyzed by each vulnerability scanner.

We used the default configuration for all tools. One exception was that we enabled the form filling option for the Burp Spider. Moreover, for the Acunetix scanner, we activated the "extensive scan feature," which optimizes the scan for mod_python applications and checks for stored XSS.

When testing our own tool, we 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, we executed our system in one of three modes. First, guided fuzzing was used. In the second run, we used extended, guided fuzzing (together with application feedback). Finally, we scanned the program using stateful fuzzing.

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, we 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, we also consider the number and diversity of different (database) objects that each scanner creates while testing an application.

Eventhough we only tested for XSS vulnerabilities within web applications many other attacks can be performed against these pages. XSS is a very common and well understood vulnerability and therefore we chose this attack for all testing tools, but the approaches described in this paper also apply to injection attacks in general as the automatic testing routines can benefit from greater coverage of the application.

5.2 Experimental Results

In this section, we present and discuss the results that the different scanners achieve when analyzing the three test applications. For each application, we present 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 Requests	Comments	XSS Warnings		Injection Points	
				Reflected	Stored	Reflected	Stored
Spider	4	4	-	-	-	-	-
Burp Spider	8	25	0	-	-	-	-
w3af	9	133	0	0	0	0	0
Acunetix	9	22	0	0	0	0	0
Use Case	4	4	1	-	-	-	-
Guided Fuzzing	4	64	12	0	1	0	1
Extended Fuzz.	6	189	12	0	1	0	1
Stateful Fuzz.	6	189	12	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 (except the simple one) requested the root of 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. Here, the advantage of guided fuzzing is clear. Because our system relies on a previously recorded test case, 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. Concerning the number of injection points that are higher for some tested scanners it has to be noted that this is caused by the way some scanners try to find new attack points. When finding a new URL they also make requests for all subdirectories of this injection point, which, depending on the application, will lead to new pages being found, a redirect or simply a 404 error page. As our approach tries to focus on use cases this logic was not implemented in our test runner.

	Locations	POST/GET Requests	Threads Created	Replies Created	XSS Warnings		Inject. Points	
					Reflect	Stored	Reflect	Stored
Spider	8	8	-	-	-	-	-	-
Burp Spider	8	32	0	0	-	-	-	-
w3af	14	201	29	0	0	3	0	1
Acunetix	263	2,003	687	1,486	63	63	0	1
Use Case	6	7	1	2	-	-	-	-
Guided Fuzzing	16	48	12	22	0	1	0	1
Extended Fuzz.	85	555	36	184	0	3	0	1
Stateful Fuzz.	85	555	36	184	0	3	0	1

Table 2. Scanner effectiveness for the forum application.

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 2 shows that while Burp Spider [5] and w3af [31] were able to create new discussion threads, only the Acunetix scanner managed to post 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 that it discovers. Therefore, it reports each vulnerability only once (and the occurrence of a stored attack replaces a previously found, reflected vulnerability).

The results show that all our 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.

	Locations	POST/GET Requests	XSS Warnings		Injection Points	
			Reflected	Stored	Reflected	Stored
Spider	18	18	-	-	-	-
Burp Spider	22	52	-	-	-	-
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 3. Scanner effectiveness for the online shopping application.

Online shopping application. The experimental results for the online shopping application are presented in Tables 3 and 4. Table 3 presents the scanner effectiveness based on the number of locations that are visited and the number of vulnerabilities that are detected, while Table 4 compares the number of database objects that were created by both the Acunetix scanner and our 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, we made two test runs with the Acunetix scanner: The first run (#1) as 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.

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 4. Object creation statistics for the online shopping application.

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. Thus, this approach reported a total of nine vulnerable entry points. In this experiment, we 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 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 4) also confirms the ability of our techniques to create a large variety of different, valid objects, a result of analyzing large portions of the application.

Discussion. All vulnerabilities that we found in our experiments were previously unknown, and we reported them to the developers of the web applications. Our results show that our 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. Finally, the different techniques exhibit different strengths. For example, stateful fuzzing becomes useful especially when the tested application is more complex and sensitive to the fuzzing steps.

6 Related Work

Concepts such as vulnerability testing, test case generation, and fuzzing are well-known concepts in software engineering and vulnerability analysis [3, 4, 11]. When

analyzing web applications for vulnerabilities, black-box fuzzing tools [1, 5, 31] are most popular. However, as shown by our experiments, they suffer from the problem of test coverage. Especially for applications that require complex interactions or 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 [16], a web vulnerability scanner that we developed previously. SecuBat can detect reflected XSS and SQL injection vulnerabilities. However, it cannot fill out forms and, thus, was not included in our experiments.

In addition to web-specific scanners, there exist a large body of more general vulnerability detection and security assessment tools. Most of these tools (e.g., Nikto [19], Nessus [29]) rely on a repository of known vulnerabilities that are tested. Our tool, 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 [14]. 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 [25] and Huang et al.'s [13] vulnerability detection tool that automatically executes SQL injection attacks. Moreover, there are a large number of static source code analysis tools [15, 27, 32] that aim to identify vulnerabilities.

A field that is closely related to our work is automated test case generation. The methods used to generate test cases can be generally summarized as random, specification-based [20, 22], and model-based [21] approaches. Fuzzing falls into the category of random test case generation. By introducing use cases and guided fuzzing, we 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, we combine the advantages provided by a tool such as WinRunner with black-box fuzzers. Moreover, we provide techniques to generalize from a recorded use case.

Finally, a number of approaches [6, 12, 18] 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 we present in this paper 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. Our system, instead, 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 us to quickly examine large parts of the application and expose it to black-box tests.

7 Conclusions

In this paper, we presented a web application testing tool to detect reflected and stored cross-site scripting (XSS) vulnerabilities in web applications. The core of our 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 our scanner to reach "deeper" into the application, we introduce 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, we can reuse the recorded user input for different forms during this process. Finally, we introduce 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. We have implemented our use-case-driven testing techniques and analyzed three real-world web applications. Our experimental results demonstrate that our approach is able to identify more bugs than several open-source and commercial web vulnerability scanners.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grants P-18764, P-18157, and P-18368 and the Secure Business Austria Competence Center.

References

[1] Acunetix. Acunetix Web Vulnerability Scanner. `http://www.acunetix.com/`, 2008.
[2] 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.
[3] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.
[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
[5] Burp Spider. Web Application Security. `http://portswigger.net/spider/`, 2008.
[6] 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.
[7] David Hannson. Ruby on Rails. `http://www.rubyonrails.org/`, 2008.
[8] Django. The Web Framework for Professionals with Deadlines. `http://www.djangoproject.com/`, 2008.

[9] Basic Django Blog Application. http://code.google.com/p/django-basic-blog/.
[10] D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.
[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.
[12] P. Godefroid, N. Klarlund, and K. Sen. DART. In *Programming Language Design and Implementation (PLDI)*, 2005.
[13] Y. Huang, S. Huang, and T. Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *12th World Wide Web Conference*, 2003.
[14] Insecure.org. NMap Network Scanner. http://www.insecure.org/nmap/, 2008.
[15] 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.
[16] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.
[17] Mitre. Common Vulnerabilities and Exposures. http://cve.mitre.org/.
[18] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.
[19] Nikto. Web Server Scanner. http://www.cirt.net/code/nikto.shtml, 2008.
[20] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. *Second International Conference on the Unified Modeling Language*, 1999.
[21] J. Offutt and A. Abdurazik. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Third International Conference on UML*, 2000.
[22] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State-based Specifications. *Journal of Software Testing, Verification and Reliability*, 2003.
[23] R. Poulton. Django Forum Component. http://code.google.com/p/django-forum/.
[24] Satchmo. http://www.satchmoproject.com/.
[25] D. Scott and R. Sharp. Abstracting Application-level Web Security. *11th World Wide Web Conference*, 2002.
[26] WhiteHat Security. Web Application Security 101 . http://www.whitehatsec.com/articles/webappsec101.pdf, 2005.
[27] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.
[28] Sun. Java Servlets. http://java.sun.com/products/servlet/, 2008.
[29] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. http://www.nessus.org/, 2008.
[30] Twill. Twill: A Simple Scripting Language for Web Browsing. http://twill.idyll.org/, 2008.
[31] Web Application Attack and Audit Framework. http://w3af.sourceforge.net/.
[32] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.