Penetration Testing with Improved Input Vector Identification

William G.J. Halfond, Shauvik Roy Choudhary, and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond, shauvik, orso}@cc.gatech.edu

Abstract

Penetration testing is widely used to help ensure the security of web applications. It discovers vulnerabilities by simulating attacks from malicious users on a target application. Identifying the input vectors of a web application and checking the results of an attack are important parts of penetration testing, as they indicate where an attack could be introduced and whether an attempted attack was successful. Current techniques for identifying input vectors and checking attack results are typically ad-hoc and incomplete, which can cause parts of an application to be untested and leave vulnerabilities undiscovered. In this paper, we propose a new approach to penetration testing that addresses these limitations by leveraging two recently-developed analysis techniques. The first is used to identify a web application's possible input vectors, and the second is used to automatically check whether an attack resulted in an injection. To empirically evaluate our approach, we compare it against a state-of-the-art, alternative technique. Our results show that our approach performs a more thorough penetration testing and leads to the discovery of more vulnerabilities.

1. Introduction

Many companies use web applications to maintain and build relationships with their customers. These applications often store sensitive and valuable information, such as customer details and payment information, which has made them the target of attacks by malicious users. The cost of these attacks has dramatically increased the importance of techniques for improving the security of web applications. One such technique, penetration testing, evaluates the security of a system by simulating attacks by malicious users and assessing whether the attacks are successful. Penetration testing can provide developers with a list of vulnerabilities and security issues in the tested web applications, which can be used to improve the security of the applications.

Penetration testing has become a widely used and integral part of quality assurance techniques for web applications. In fact, many government agencies and trade groups, such as the Communications and Electronic Security Group in the U.K., OWASP,[1] and OSSTMM,[2] accredit penetration testers and sanction penetration testing "best practices." Penetration testing is a useful technique for several reasons: 1) it generally produces a low rate of false positives since it discovers vulnerabilities by exploiting them and thus producing counter examples; 2) it tests applications in context, which allows for the discovery of vulnerabilities that arise from specific configuration and environment issues; and 3) it provides a set of concrete inputs that exploit the discovered vulnerabilities and that can be used to guide developers in correcting the code.

The process of penetration testing can be broadly divided into three phases: information gathering, attack generation, and response analysis. Figure 1 shows a high-level overview of a generic penetration testing process. In the *information gathering* phase, testers use a wide variety of techniques, such as automated scanning, web crawlers, and social engineering, to gain information about the target application. This information is used to drive the *attack generation* phase, in which testers use the identified information, together with domain knowledge about possible vulnerabilities, to generate attacks. Penetration testers typically use a range of commercial and open-source tools to automate the generation of attacks. Finally, the *response analysis* phase checks whether an attack has succeeded and, if so, logs information about the attack. The final result of the penetration testing process is a report that details the discovered vulnerabilities and corresponding attacks. Developers can use this information to eliminate the vulnerabilities and improve the security of their software.

The collection of accurate high-quality information during the information gathering phase is vital for the success of penetration testing. Better information about an application generally leads to more effective testing and a higher confidence in the thoroughness of the results. Of particu-

[1]`http://www.owasp.org/`
[2]`http://www.osstmm.org/`

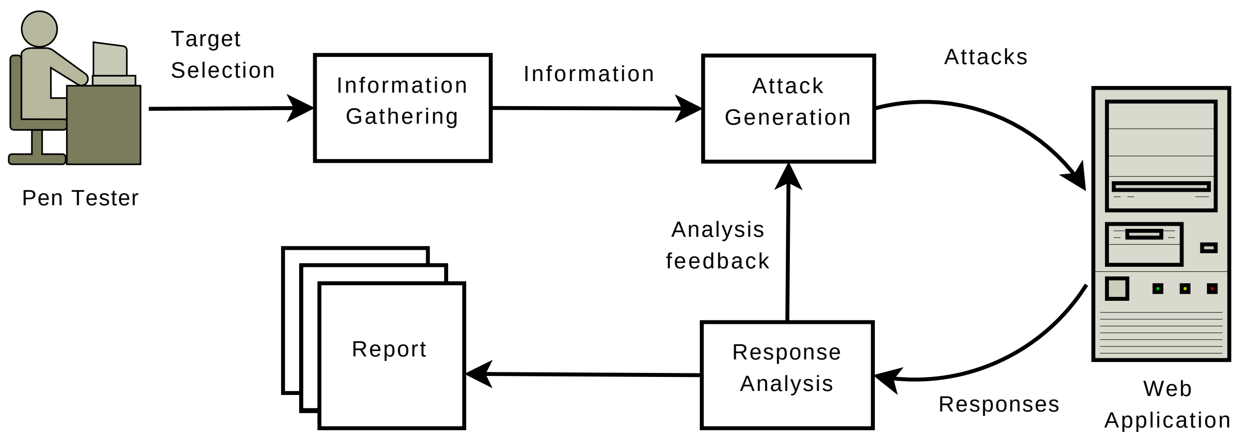

Figure 1. The penetration testing process.

lar importance to penetration testers is the identification of an application's *input vectors* (IVs)—points in an application where an attack may be introduced, such as user-input fields and cookie fields. Most penetration testers rely on automated scanners, such as web crawlers, to identify IVs in the target web application. Web crawlers traverse the HTML content generated by a web application and analyze each page to identify information related to IVs. Although web crawlers are widely used, the information they gather is generally incomplete. This incompleteness is due to the fact that web crawlers are typically unable to visit every page in the web application or need to interact with the web application in some specific way in order for certain elements to be displayed. Performing attack generation using an incomplete set of IVs can limit the effectiveness of penetration testing dramatically.

In addition to accurately identifying IVs, another important aspect of penetration testing is response analysis to determine whether an attempted attack was successful. In most cases, this assessment is performed manually by the tester, which can be time consuming and error prone. Some techniques try to automate the check by using various heuristics. In our experience, such heuristics can work well in simple cases, but are fairly ineffective when used for realistic applications.

In this paper, we propose a penetration testing approach and tool that addresses the aforementioned shortcomings of existing approaches. To *improve the information gathering phase*, our approach leverages a static analysis technique for discovering IVs that we previously developed [12]. Specifically, our approach combines a conservative static analysis that identifies all possible IVs with a dynamic penetration testing technique that effectively generates attacks for such IVs. Although it is common to assume that penetration testing is a black-box approach, current best practices (*e.g.,* OWASP[1] and OSSTMM[2]) recommend that penetration testers assume that attackers have access to one or more versions of the source code of the application. By building on this and leveraging static analysis of the source code, our approach can outperform the typical black-box-only approaches to penetration testing. To *improve the response analysis phase*, our technique incorporates and adapts a dynamic analysis technique that we also developed in previous work [11]. The technique performs automated identification of injection attacks by leveraging dynamic tainting and allows our approach to perform fully-automated detection of successful attacks in most cases.

We implemented our approach in a prototype tool, SDAPT, and used the tool to perform an extensive empirical evaluation of our approach. In the evaluation, we used SDAPT to perform penetration testing on nine web applications. Our empirical results show that our approach was able to (1) exercise the subject applications more thoroughly and (2) discover a considerably higher number of vulnerabilities than a traditional penetration testing approach.

The contributions of this paper are:

- An approach for penetration testing based on improved input vector identification and automated response analysis.
- An implementation of the approach in a prototype tool.
- Four empirical studies that assess the practicality, thoroughness, and effectiveness of the approach on nine web applications.

2. Motivating Example

In this section we introduce a small motivating example to illustrate the drawbacks of traditional penetration testing. We also use this example in Section 3 to illustrate our approach.

Figure 2 shows an excerpt from a Java-based web application. This example is implemented as a *servlet*, which

```
 1. public void service(HttpServletRequest req) {
 2.  String action = req.getParameter("userAction");
 3.  if (action.equals("createLogin")) {
 4.    String password = req.getParameter("password");
 5.    String loginName = req.getParameter("login");
 6.    if (isAlphaNumeric(password)) {
 7.       Connection conn =
               new Connection("mysql://localDB");
 8.       Statement stmt = conn.getStatement();
 9.       stmt.execute("insert into UserTable "
                        + "(login, password) values ("
                        + loginName + ", "
                        + password + ")");
10.       displayAddressForm();
11.    } else {
12.       displayErrorPage("Bad password.");
13.    }
14.  } else if (action.equals("provideAddress")) {
15.     String loginName = req.getParameter("login");
16.     String address = req.getParameter("address");
17.     Connection conn =
             new Connection("mysql://localDB");
18.     Statement stmt = conn.getStatement();
19.     stmt.execute("update UserTable set"
                        + " address ='"+address+"'"
                        + " where loginName = "
                        + loginName);
20.  } else{
21.     displayCreateLoginForm();
22.  }
23. }
```

Figure 2. Example servlet.

is the basic implementation unit in the Java Enterprise Edition (JEE) framework for developing web applications. This servlet allows a user to register a new login and password by filling out a series of web forms. When a user first accesses this servlet, execution begins in function `service`, which is the standard entry point for all servlets. At line 2, the servlet accesses an input parameter named "userAction." In general, parameters are passed from an end-user to a servlet as name-value pairs. To access these parameters, a servlet calls a *parameter function*, passes to it the name of the desired parameter, and receives its corresponding value. At lines 3 and 14, the servlet checks the values of parameter "userAction." The first invocation of the servlet is performed with an empty set of parameters, which means that the execution continues at line 21. Function `displayCreateLoginForm` generates a web form that is sent to the user browser and allows the user to enter their desired login name and password. This function also sets a hidden input field called "userAction" to the value "createLogin."

When the user enters the data in this web form and submits it, the browser bundles the three input fields (login, password, and the hidden field) as name-value pairs and sends them to the servlet. On this second execution of the servlet, the condition at line 3 is true, and the servlet retrieves the password and login (lines 4–5). If the password is not alphanumeric, an error message is returned to the end user (line 12). Otherwise, the servlet generates a query to the database that creates the user account (lines 7–9). The servlet then calls function `displayAddressForm`, which generates another web form that allows the user to enter their address (line 10). This function also sets two hidden input fields in the form: "userAction" is set to "provideAddress," and "login" is set to the user-chosen login. When the user submits this form, the condition at line 14 is true, so the servlet retrieves the address and login fields (lines 15–16) and updates the entry in the database with the supplied information (lines 17–19). At this point, the registration is done.

This example servlet contains several vulnerabilities to SQL injection attacks (SQLIA). An application can be vulnerable to SQLIAs when it directly uses user input in a database query. An attacker can take advantage of this situation and insert database commands that will be directly executed by the database. For our example servlet, if an attacker enters the string "`name, secret); drop table UserTable --` " as their chosen login, the following SQL query would be executed on the database at line 9: "`insert into UserTable (login, password) values (name, secret); drop table UserTable -- )`". Besides creating an account with login "name" and password "secret," this query would execute a drop command that would delete all of the user information in the user table. (Note that "`--`" is the SQL comment operator, so the extra parenthesis after the query would be ignored.) In general, the queries at lines 9 and 19 are vulnerable to a wide range of SQLIAs [13].

When a web crawler targets this servlet, it would be able to discover the web form generated by `displayCreateLoginForm` because this is the default page created by the servlet. From this page, the web crawler would then be able to identify that the names of three IVs for this application are "userAction," "login," and "password." At this point, most web crawlers would generate values for the identified IVs in an attempt to access subsequent pages that may contain additional information. However, unless the crawler could correctly guess that "password" must be alphanumeric, the servlet would not reveal any additional useful information, and the penetration tester would not discover the vulnerabilities at lines 9 and 19.

3. Approach and Implementation

The goal of our approach is to improve penetration testing of web applications. To do this, our approach focuses on two areas where current techniques are limited: identifying the IVs of a web application and detecting the outcome of an attempted attack. We developed a new approach to penetration testing that leverages two recently-developed analysis techniques. The first is a static analysis technique for identifying potential IVs, and the second is a dynamic analysis to automate response analysis. In the information gathering phase, our approach leverages the static analysis technique to analyze the code of the application and identify IVs, how they are grouped (*i.e.,* which sets of IVs are accessed together by a servlet), and their domain information (*i.e.,* IVs' relevant values and type constraints). In the attack generation phase, our approach targets the identified IVs and uses the domain and grouping information to generate realistic values for the penetration test cases. Finally, in the response analysis phase, our approach uses the dynamic analysis technique to assess in an automated way whether an attack was successful.

In the rest of this section, we explain the details of our approach to penetration testing. We divide the discussion of or approach based on the three phases of penetration testing and explain how our approach performs each one. Where applicable, we illustrate the advantages of our approach using the example from Section 2. We also present, in Section 3.4, the details of our tool implementation.

3.1. Information Gathering

During the information gathering phase, testers analyze the target application to identify information that may be useful to generate attacks. In particular, testers are interested in gathering information about the application's IVs—their names, groupings, and domain information. Web crawlers are currently one of the most popular and widely-used techniques for discovering this type of information. However, as we discussed in Section 2, web crawlers are limited by their purely black-box nature and generally discover incomplete information. Conversely, our static analysis technique is complete since it performs a conservative analysis to identify IVs. Although the conservative nature of the analysis may lead to the identification of spurious IVs, this does not affect the effectiveness of penetration testing and, in the worst case, can only result in the generation of additional test inputs.

The static analysis performed by our approach is based on a technique, WAM, that was developed by two of the authors [12]. We provide a high-level overview of this analysis and explain how we leveraged it for the purpose of information gathering during penetration testing. The WAM technique analyzes the code of a web application in two phases and produces a listing of all of the IVs in the application together with their groupings and domain information. In the first phase, WAM computes domain information for IVs in the application by identifying chains of definitions and uses (*DU chains*) that start with the return value of a parameter function. For each DU chain, WAM checks for typecasts, value equality comparisons, and specific API calls that allow for inferring information about the domain of the IVs accessed via that parameter function. (Note that, at this time, WAM does not account for more complex domain constraints, such as regular expressions or relational operators.) To illustrate with an example, consider the IV accessed at line 2 of the example servlet in Figure 2, whose value is assigned to variable `action`. Variable `action` is used at lines 3 and 14. From these uses, WAM can infer that "createLogin" and "provideAddress" are relevant values for the accessed IV. Similarly, for the IV accessed at line 4, WAM can infer from the use at line 6 that the domain of the IV should be alphanumeric. In its second phase, the WAM analysis groups IVs accessed along the same path of execution and identifies the names of the individual IVs. To avoid an expensive per-path analysis, WAM uses an iterative data-flow algorithm to compute the groupings. To identify the names of IVs, WAM tries to resolve the value of the parameter passed to the parameter functions. In our example, this is always a string constant, but the analysis can handle more complicated cases where this is a string expression involving values defined in different methods. In the example servlet the second phase of WAM would identify the following three groupings of IVs: {"userAction"}, {"userAction," "login," "password"} and {"userAction," "login," "address"}.

Our approach leverages the WAM analysis to produce IV information that is then provided as input to the attack generation phase. To better tailor the analysis to penetration testing, our approach extends WAM with heuristics for identifying error checking patterns in the web application code, such as checks for empty strings or null values. Our approach eliminates values associated with such checks from the domain information because we found that they typically lead to test inputs that cause a properly handled error and are unlikely to facilitate successful attacks.

3.2. Attack Generation

During the attack generation phase, testers use the information gathered in the previous phase to create attacks on the target application. To do this, a tester typically targets each identified IV using a set of attack heuristics, while supplying realistic and "harmless" input values for the other IVs that must be part of a complete request. The identification of suitable realistic input values for the IVs not involved in an attack is a crucial part of this process. Traditionally, testers would determine such values by interacting with the developers, using values supplied as defaults in the web pages examined during the previous phase, or generating random strings. Although practical, these approaches may not provide realistic values that will enable a vulnerability to be exposed, as we illustrated in Section 2.

Our approach addresses this problem by using the domain and grouping information identified by the WAM analysis to provide relevant values for all IVs that are not being injected with potential attacks. Our approach does not create new attack heuristics; it provides a way to generate more realistic and relevant values for the penetration test cases.

To illustrate with an example, we use one of the IV groupings identified by the information gathering phase for the example servlet in Figure 2: {“login,” “password,” “userAction”}. During attack generation, the testers would target each of these IVs with possible injections based on some attack heuristics. When the first IV, “login,” is targeted, both our approach and traditional approaches would generate an attack string and use it as the value for “login.” The difference between our approach and other approaches is how the values for the other IVs are determined. Our approach leverages the domain information discovered by WAM, which would result in using an alphanumeric value for “password” and setting “userAction” first to “createLogin” and then to “provideAddress.” The use of this domain information allows the penetration test cases to pass the checks at lines 3 and 6, and thus successfully exploit the vulnerability at line 9. In contrast, approaches that do not have this domain information would have to either involve the developer, which would affect the practicality of the approach, or use random values, which would be unlikely to satisfy the domain constraints on the IVs.

3.3. Response Analysis

In the response analysis phase, testers analyze each web page that the target application returns after an attempted attack. The purpose of the analysis is to determine if the attack succeeded and extract any additional information that was revealed in the response. Because manual checking of web pages is extremely time consuming and error-prone, testers typically use automatable heuristic-based tools to check whether an attack was successful. For example, some tools search the text in the resulting web page for exceptions thrown by the database. Other tools compare the content generated by a servlet in response to an attack and a legal access and look for significant changes that would indicate that the attack was successful. Unfortunately, the success of these approaches is highly application specific, and it is difficult to identify automated heuristics that are broadly applicable. In fact, our experience shows that attempts to do so can be highly ineffective. (See Section 4.4.)

In our approach, we improve the accuracy of response analysis by incorporating an automated injection detector based on a technique, WASP, that was developed by two of the authors in previous work [10, 11]. The WASP technique uses positive tainting to track all of the trusted strings in an application that may be used to build a database command. At runtime, WASP uses syntax-aware evaluation to ensure that only trusted strings are used to form the parts of a database command that correspond to SQL keywords and operators. If a database command violates this policy, it is prevented from executing on the database. To use WASP in the context of penetration testing, we extended it so that it adds a special HTTP header to the application’s response when it detects an attack. The header informs the response analysis that a performed attack was successful. The response analysis can thus correlate this information with the information provided by the attack generator to identify and report each vulnerable IV and the attack that was able to reveal the vulnerability.

To illustrate the response analysis with an example, consider the SQLIA presented in Section 2 that targets line 9 of the servlet in Figure 2. Before the servlet executes, WASP performs positive tainting and marks all of the trusted strings in the servlet. In the example, the trusted strings are all of the hard-coded strings used to build database queries at lines 9 and 19. (The other hard-coded strings are also marked as trusted, but are not used to build database queries, so we do not discuss them further.) At runtime, WASP tracks the trust markings on the strings. When the servlet attempts to execute a database query, WASP checks the string that contains the query to be executed. In this check WASP parses the string using the database’s parser and verifies that every substring that represents a keyword or operator was generated using a trusted string. Referring back to the example SQLIA, this check would reveal that the keyword “DROP” was generated using a string that was not trusted. This causes WASP to block the attack and return the special HTTP header that flags a detected attack.

3.4. Implementation

We implemented our approach in a prototype tool called SDAPT (Static and Dynamic Analysis based Penetration Testing). SDAPT is implemented in Java, works on Java-based web applications, and performs penetration testing for discovering SQLIA vulnerabilities. The high-level architecture of SDAPT is shown in Figure 3. SDAPT inputs the code of a web application (*i.e.,* a set of servlets in bytecode format) and produces a report with a list of the successful SQLIAs and the corresponding vulnerable IVs. We chose SQL injection as our attack type because there are a large number of web applications that contain SQLIA vulnerabilities.

The *information gathering* module analyzes the servlets’ code and outputs information about the IVs of each servlet. For this module, we used the adapted implementation of the WAM analysis described in Section 3.1.

The *attack generation* module consists of several submodules. The *controller* inputs the IV-related information and passes the IV groups, one at a time, to the *IV selector*. The *IV selector*, in turn, iterates over each of the IVs in a group and, for each selected IV, passes it to the *attack heuristics* module, which generates possible attack strings for the IV. The *injection engine* generates penetration test cases by combining these attack strings for the selected IV and legitimate values for the remaining IVs in the current IV group. To generate legitimate values, the engine lever-

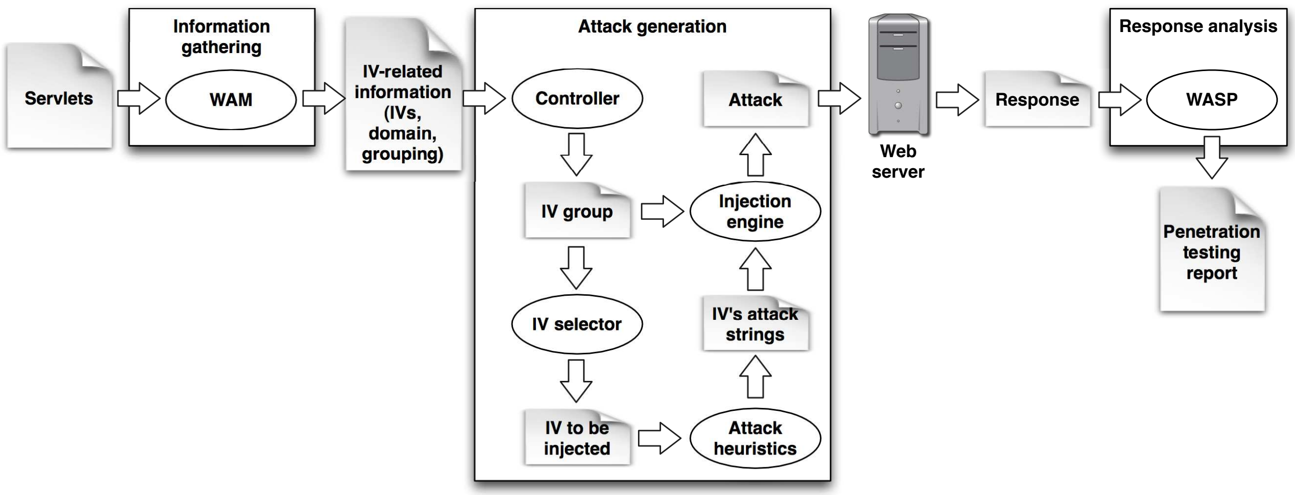

Figure 3. High-level architecture of the SDAPT tool.

ages the IVs' domain information. The generated attacks are then sent to the target web application. In our implementation, the *controller* was built from scratch, whereas to implement the *IV selector*, *attack heuristics*, and *injection engine* modules, we modified parts of the code base of SQLMAP.[3] We chose to use SQLMAP as the basis for our implementation for several reasons. First, SQLMAP is a widely used, popular, and actively maintained penetration testing tool for discovering SQLIA vulnerabilities. Second, the architecture of SQLMAP is highly modular, which made it easier to integrate it in our tool. Finally, SQLMAP contains heuristics for performing many different types of SQLIAs and can interact with a wide range of applications that communicate using different HTTP based mechanisms.

The *response analysis* module receives the HTML responses generated by the target web application and analyzes them to determine whether the attack was successful. It then associates this information with the IV under test. After all of the responses have been analyzed, the output of this module is a report that lists all of the vulnerable IVs with the test inputs that were able to reveal the vulnerability. The implementation of this module is based on an extended version of WASP (see Section 3.3), which was developed by two of the authors in previous work [10, 11].

4. Empirical Evaluation

The goal of our empirical evaluation is to assess the usefulness of our penetration testing approach, implemented in the SDAPT tool, when compared to a traditional penetration testing tool. To do this, we measure SDAPT's *practicality* in terms of the time and resources needed to perform the information gathering and attack generation phases, *thoroughness* in terms of the number of IVs and components tested, and *effectiveness* in terms of the number of vulnerabilities discovered. Our evaluation addressed the following research questions:

RQ1: Is SDAPT practical in terms of its time and resource requirements?

RQ2: Does SDAPT result in more thorough testing of a web application than a traditional approach?

RQ3: Is SDAPT's response analysis more accurate than a heuristic-based approach?

RQ4: Does SDAPT's information gathering lead to the discovery of more vulnerabilities than a traditional approach?

As an instance of a traditional approach, we implemented an improved version of SQLMAP,[3] the penetration testing tool we discussed in Section 3.4. The improved tool, SQLMAP++, extends SQLMAP in two ways. First, we integrated a web crawler into SQLMAP to perform information gathering. Web crawling is one of the most widely-used techniques for gathering information about a web application and is thus a good representative of current approaches. We used a web crawler based on the OWASP WebScarab[1] project and modified it so that it collects IVs and any default values for these IVs in the web pages it visits. (The default values are used as possible values for the IVs during attack generation.) Second, we integrated our improved response analysis (see Sections 3.3 and 3.4) into SQLMAP.

Note that, in the implementation of SDAPT and SQLMAP++, we maximized the amount of code reused wherever possible. In particular, the two tools use the same attack heuristics from the original SQLMAP tool. Also SQLMAP++ and SDAPT use the same implementation of the response analysis.

[3]http://sqlmap.sourceforge.net/

Subject	*LOC*	*Classes*	*Servlets*
Bookstore	19,402	28	27
Checkers	5,415	59	32
Classifieds	10,702	18	18
Daffodil	18,706	119	70
Empl. Dir.	5,529	11	9
Events	7,164	13	12
Filelister	8,671	41	10
Office Talk	4,670	63	39
Portal	16,089	28	27

Table 1. Subject web applications.

	Analysis time (s)		Number of test cases	
Subject	SQLMAP++	SDAPT	SQLMAP++	SDAPT
Bookstore	40	2,322	802	14,711
Checkers	5	146	5	492
Classifieds	79	1,797	544	8,557
Daffodil	13	1,271	442	20,698
Empl. Dir.	15	449	223	3,237
Events	11	853	106	3,746
Filelister	6	862	45	4,465
Office Talk	5	477	18	208
Portal	45	726	393	9,266

Table 2. Practicality results.

4.1. Experiment Subjects

In our evaluation, we used the nine Java-based web applications listed in Table 1. Five of these applications (Bookstore, Classifieds, Empl. Dir., Events, and Portal) are commercial open-source products available from GotoCode (`http://www.gotocode.com/`). Two of the subjects, Checkers and Officetalk, are student-developed projects that have been used in previous studies [9, 10]. Filelister and Daffodil are open source projects available from SourceForge (`http://sourceforge.net/`).

The GotoCode and student-developed subjects contain a wide range of security vulnerabilities. The remaining two applications contain specific and known vulnerabilities that have been reported in the Open Source Vulnerability Database (`http://osvdb.org/`). Table 1 lists, for each subject application, its number of lines of code (*LOC*), classes (*Classes*), and classes that are servlets (*Servlets*)

4.2. RQ1: Practicality

To evaluate the practicality of our approach, we compared the analysis time of SDAPT and SQLMAP++, and the number of test cases they generated during penetration testing. For the analysis time of SQLMAP++, we measured the time needed to crawl and analyze an application's web pages. For SDAPT, we measured the time to statically analyze each application. The results of this study are shown in Table 2. The table lists, for each subject and each of the two tools, the *analysis time* and the *number of test cases* generated.

The results in the table show that both analysis time and number of test cases are higher for SDAPT than for SQLMAP++. The analysis time of SQLMAP++ ranges from five to 79 seconds, with an average of about 24 seconds. The analysis time of SDAPT ranges from two to almost 39 minutes, with an average of about 16 minutes. Despite being higher than SQLMAP++'s analysis time, SDAPT's analysis time is still clearly practical. Moreover, this time cost is typically incurred only once per application because the IV information is computed once and then simply reused during attack generation.

In terms of number of test cases generated, SDAPT consistently generated at least an order of magnitude more test cases than SQLMAP++. This result is somehow expected, given SDAPT's more complete identification of IV-related information; richer IV information is likely to result in more test cases being generated. Although a higher number of test cases results in more testing time, the maximum testing time for the subject considered was below ten hours, which would not prevent the test cases from being run overnight. Moreover, as our results for RQ2 and RQ4 show, the additional test cases always result in a more thorough penetration testing and in the discovery of more vulnerabilities.

4.3. RQ2: Thoroughness

To evaluate the thoroughness of our approach, we measured the number of IVs and components tested by SDAPT and SQLMAP++. In general, a higher number of tested IVs and components indicates that more points in the application are being exercised to assess whether they contain vulnerabilities. To determine the number of IVs tested, we analyzed the attacks generated by the two tools and counted the number of unique IV names targeted for each servlet of each subject application. Similarly, to determine the number of components tested, we counted the number of unique components targeted in each application. Table 3 shows the results of this analysis. For each subject and each tool, the table lists the number of unique IVs (*Number of IVs*) and the number of unique components (*Number of Comp.*) exercised during the penetration testing.

As the results in the table show, SDAPT resulted in a consistently higher number of tested IVs and components than SQLMAP++. On average, SDAPT tested 111 IVs and 20 components per application, compared to the 56 IVs and eight components per application tested by SQLMAP++.

To better understand the reason for SDAPT's performance, we manually inspected the code of several servlets in the subject applications. We found that SQLMAP++

Subject	Number of IVs		Number of Comp.	
	SQLMAP++	SDAPT	SQLMAP++	SDAPT
Bookstore	104	189	15	27
Checkers	5	69	2	20
Classifieds	61	118	10	18
Daffodil	107	165	7	39
Empl. Dir.	36	66	6	9
Events	44	79	8	12
Filelister	12	46	1	9
Office Talk	16	58	5	20
Portal	123	211	20	27
Average	**56**	**111**	**8**	**20**

Table 3. Evaluation of thoroughness.

Subject	Number of Vulnerabilities		
	SQLMAP++$_{NORA}$	SQLMAP++	SDAPT
Bookstore	0	7	11
Checkers	0	0	2
Classifieds	0	4	14
Daffodil	0	6	11
Empl. Dir.	0	1	11
Events	0	4	11
Filelister	0	1	1
Office Talk	0	2	12
Portal	0	11	17
Total	**0**	**36**	**90**

Table 4. Evaluation of effectiveness.

tested less components mainly because many of the web pages in the web applications are not linked to each other. Therefore, the crawler was not able to reach all of the pages in an application, and the attack generation based on the information collected by the crawler never targeted the unreachable pages. These unreachable pages also partly explain why SDAPT was able to test a higher number of IVs. However, this was not the only reason for this difference, as our inspection also revealed two other reasons. The first reason is that several components require the crawler to provide specific IV values in order to display subsequent web forms (as in our example in Figure 2). Because the crawler was not able to guess these values, it could not reach these subsequent web forms and missed their IV information. The second reason is that several components have IVs that do not have a corresponding web form—they were "hidden" IVs. These IVs may have been developer errors or IVs intended only for use by other components without going through a web form.

Overall, the higher number of tested IVs and components provides evidence that our penetration testing approach can result in a more thorough testing of a web application.

4.4. RQ3: Response Analysis Effectiveness

We evaluated the effectiveness of our technique for response analysis independently from the effects of the improved information gathering approach. To do this, we also implemented a version of SQLMAP++ that did not include our response analysis technique and used the standard heuristic-based response analysis provided by SQLMAP. We call this version SQLMAP++$_{NORA}$. We then measured the number of vulnerabilities discovered by SQLMAP++ and SQLMAP++$_{NORA}$ when run on all subject applications. The results of this study are shown in Table 4 under the columns titled SQLMAP++$_{NORA}$ and SQLMAP++.

As the results in the table show, SQLMAP++$_{NORA}$ was unable to recognize any vulnerabilities in the subject applications. In contrast, using the improved response analysis, SQLMAP++ discovered a total of 36 vulnerabilities. This result indicates that, although SQLMAP may be able to generate inputs that cause SQLIAs, its response analysis is totally ineffective. Our manual inspection of the generated attacks and results revealed that, in most cases, an attempted attack did not have any observable effect on the HTML response from the attacked servlet. In the few cases where there was a change in the HTML response, the change was subtle enough that the heuristic-based analysis was not able to determine whether it was a normal variation in output or the result of a successful attack. For example, one case involved a page that listed results extracted from a table. A successful attack caused the page to list a specific set of results, but nothing about these results clearly indicated the effect of an attack (*i.e.,* the same results might have been generated by a successful query). Therefore, a heuristic that simply checks differences between HTML responses would be unable to determine if the variation in the listed results was due to an attack. Manual inspection of the results might have been able to recognize the attack, but manually checking each attempted attack is, in general, impractical and error prone.

Overall, these results motivate the need for improved response analysis and indicate that our proposed technique for response analysis is effective and is an important part of our approach for penetration testing.

4.5. RQ4: Information Gathering Effectiveness

To evaluate the effectiveness of our technique for information gathering, we measured the number of vulnerabilities discovered by SQLMAP++ and SDAPT. As with RQ3, we ran both tools against each of the subject applications. Table 4 shows the results of this study. For each application, we list the number of vulnerable IVs discovered by SQLMAP++ and SDAPT.

The results in the table show that SDAPT was able to discover considerably more vulnerabilities than SQLMAP++. SDAPT discovered a total of 90 vulnerable IVs as compared to 36 found by SQLMAP++. Of particular interest are the results for the applications with known vulnerabilities. For Filelister, both tools were able to discover the single known vulnerable IV. For Daffodil, there were two known vulnerable IVs. SQLMAP++ discovered an additional 4, and SDAPT discovered an additional 9.

In addition to discovering more vulnerabilities, our approach also had a very low false positive rate. We manually inspected each reported vulnerability in order to determine if it was a real vulnerability or a false positive. We found that SQLMAP++ reported three false positives and SDAPT reported two false positives. These were not included in the vulnerability totals in Table 4. For both approaches, the false positives were caused by limitations in the implementation of WASP, and could be eliminated with further engineering.

Overall, our results show that, at least for the subjects considered, our approach can outperform more traditional penetration testing techniques and that our information gathering technique plays an important role in the effectiveness of our approach.

4.6. Threats to Validity

In this section, we outline the possible threats to validity of our empirical evaluation and explain how we addressed each threat.

Construct Validity: Construct validity is straightforward in our empirical evaluation, as we use typical metrics for evaluating the thoroughness, effectiveness, and practicality of our approach. For thoroughness, the number of IVs and components tested is a common measure for both penetration and regular testing. For effectiveness of penetration testing, the number of vulnerabilities is by far the most commonly accepted metric. Similarly, for practicality, analysis time and used resources are generally accepted metrics.

Internal Validity: For internal validity, we must ensure that variances in the dependent variable (measure of thoroughness, effectiveness, and practicality) can be attributed to variances in the independent variable (the information gathering and response analysis). To ensure this, we used the same attack heuristics in both tools. The tools differed only in the information gathering, use of the gathered information, and response analysis. We also deployed the subject applications with the same configuration when testing them with the two approaches.

External Validity: The primary concern is whether the results could generalize to more web applications and automated penetration testing tools. Our set of subject applications consisted of nine subjects that came from several different sources: commercial open-source, auto-generated code, and student-developed projects. More subjects would obviously enhance the validity of the studies, but we feel that the range in size, source, and type of application provides us with a reasonably representative set of subjects. SQLMAP is a widely used penetration testing tool that has an architecture and approach similar to many other penetration testing tools. In particular, it uses widely-recognized attack heuristics for discovering vulnerabilities to SQLIAs in web applications.

5. Related Work

A technique by Miller, Fredricksen, and So [17], called *fuzzing*, was an early influential work that led to the development of many subsequent penetration testing techniques. In their work, Miller and colleagues submitted byte streams of random data to common UNIX utilities to assess whether they could crash them. This technique was later adopted and expanded by many testers to discover bugs and security vulnerabilities [20]. Although the concepts and principles behind penetration testing have been known for quite some time, it was not until recently that penetration testing began to receive significant attention [21]. Geer and Harthorne provided an early definition of the goals and techniques of penetration testers [8]. Subsequent work has motivated the need for penetration testing and proposed ways to incorporate the technique into software engineering processes [2, 3].

In the area of information gathering techniques to support penetration testing, there has been very limited research work. Most of the work in the area has been commercially oriented and focused on improving web crawling techniques, such as OWASP's WebScarab web crawler,[1] or on developing new vulnerability scanners, such as Nessus[4] and Nikto.[5] Notable research contributions in this area include the development of an advanced web crawler by Huang and colleagues [14], and a technique by Elbaum and colleagues [5] that interacts with a web application at runtime to identify IVs and possible domain information. Since these techniques are both dynamic techniques based on web crawling, they cannot provide guarantees of completeness and have limitations similar to those of other web crawling techniques. However, as compared to our static approach, these types of technique would be advantageous in situations where the source code is unavailable or cannot be statically analyzed due to resource constraints.

Our information gathering technique more broadly relates to techniques that address the problem of interface identification in web applications. In this area, there has been a fair amount of work. Early techniques relied on developer-provided specifications [1, 15, 18], which does

[4]`http://www.nessus.org/`
[5]`http://www.cirt.net/nikto2`

not fit well into the usage scenario of penetration testing, where oftentimes the vulnerabilities are found in IVs that are unknown or untested by the developers. Several techniques [6, 7, 16, 19] use session data and user logs to identify relevant information about the monitored web applications. These techniques have limitations similar to those associated with web crawlers, as they can only analyze parts of a web application that have been visited.

The technique most closely related to our information gathering technique is the one proposed by Deng, Frankl, and Wang [4]. Their technique scans the code of an application and identifies the names of all IVs. However, unlike our technique, it does not group them based on paths of execution or determine possible relevant values for each IV, which reduces the effectiveness of the technique for penetration testing.

6. Conclusion

In this paper we proposed an approach and tool for penetration testing that addresses two of the shortcomings of existing penetration testing techniques. First, our approach identifies the input vectors of a web application in a conservative way and improves on traditional, purely black-box approaches. Second, our approach uses an automated technique to assess whether an attempted attack was successful, which results in the identification of a higher number of attacks and, ultimately, in the discovery of a higher number of vulnerabilities. We created a prototype tool, SDAPT, that implements our penetration testing approach. In our empirical evaluation, we compared SDAPT against a state-of-the-art penetration testing tool in terms of practicality, thoroughness, and effectiveness in testing nine web applications. The results of our evaluation show that SDAPT can perform a more thorough testing and discover more vulnerabilities than a traditional tool. Therefore, the results provide evidence that our penetration testing approach is, at least for the applications considered, practical and effective.

Acknowledgements

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

References

[1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing Web Applications by Modeling with FSMs. In *Software Systems and Modeling*, pages 326–345, July 2005.

[2] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84 – 87, 2005.

[3] M. Bishop. About Penetration Testing. *IEEE Security & Privacy*, 5(6):84–87, 2007.

[4] Y. Deng, P. Frankl, and J. Wang. Testing Web Database Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[5] S. Elbaum, K.-R. Chilakamarri, M. F. II, and G. Rothermel. Web Application Characterization Through Directed Requests. In *International Workshop on Dynamic Analysis*, May 2006.

[6] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. In *International Conference on Software Engineering*, November 2003.

[7] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging User-Session Data to Support Web Application Testing. *IEEE Transactions On Software Engineering*, 31(3):187–202, March 2005.

[8] D. Geer and J. Harthorne. Penetration testing: a duet. In *Proceedings of the 18th Annual Computer Security Applications Conference.*, December 2002.

[9] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th International Conference on Software Engineering*, May 2004.

[10] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 2006.

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

[12] W. G. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the ESEC/SIGSOFT Symposium on the Foundations of Software Engineering*, Sep. 2007.

[13] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proc. of the Intl. Symp. on Secure Software Engineering*, Mar. 2006.

[14] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the 12th International World Wide Web Conference* , May 2003.

[15] X. Jia and H. Liu. Rigorous and Automatic Testing of Web Applications. In *6th IASTED International Conference on Software Engineering and Applications*, November 2002.

[16] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[17] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[18] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *International Conference on Software Engineering*, May 2001.

[19] J. Sant, A. Souter, and L. Greenwald. An Exploration of Statistical Models for Automated Test Case Generation. In *Proceedings of the International Workshop on Dynamic Analysis*, May 2005.

[20] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[21] H. H. Thompson. Application penetration testing. *IEEE Security & Privacy*, 3(1):66 – 69, 2005.