Network and Web Security

Tutorial 5: SQL Injection Vulnerabilities*

February 15, 2021

This tutorial continues where Tutorial 4 left off: we'll continue using the copy of the *Damn Vulnerable Web Application (DVWA)* on the `dvwa` VM to investigate *SQL injection*, one of the most common server-side web attacks.

1 SQL injection vulnerabilities in DVWA

Modern web applications typically offer a view of some data stored on a database server such as MySQL or PostgreSQL. This involves reading input from a user of the web application, including it as part of a query to the database server to retrieve relevant information from it, then showing a digested form of the retrieved data to the user.

If the web application is poorly-written, a malicious user may be able to provide some specially-crafted input that subverts the meaning of the query that the web developer intended to execute on the database server; this is often possible when the developer builds the query dynamically by concatenating strings, mixing SQL with user input, without checking whether the user input could change the meaning of the surrounding SQL after all of the strings have been concatenated. This type of attack is known as *SQL injection*.

Depending on the intended purpose of the original query and how its results are used in the web application, a SQL injection attack could cause the web application to disclose more data than it intended to, overwrite data, or destroy data entirely.

Part of DVWA is specifically designed to be vulnerable to SQL injection: let's try performing some attacks against it.

1. In `kali-vm`, open Chrome or Firefox and visit `http://<dvwa ip>/dvwa/`.
2. Log in to DVWA with the user name `admin` and the password `password`.
3. From the left-hand menu, select **SQL Injection**.
4. This part of DVWA allows you to query a table of users by user ID (an integer), displaying the forename and surname of the user (if any) with the matching ID. Perform a SQL injection against this part of DVWA, causing it to reveal the password hashes of all five users. (The part of the web application that you need to exploit is in the box with the solid black border; the PHP source code for this part of the page is shown when you click the **View Source** button.)

You aren't given the name of the table containing the user information or the name of the column in that table containing the password hash. To discover them, you have three options:

Guess them. It's good practice to name SQL tables and columns sensibly according to the kind of data they contain. Most web developers follow this practice, which also makes it easier for attackers to blindly guess them.

Get the web application to disclose them to you. If the web application is badly-written, a failed SQL injection may trigger an exception in the web application code that reveals a database server error message. These

*Thanks to Chris Novakovic `c.novakovic@imperial.ac.uk` for preparing this material.

should not be displayed publicly, because they usually contain sensitive information, such as the query that the web application attempted to execute — and potentially the user name and even the password used to connect to the database server, although this is rare these days.

Get the database server to disclose them to you. Many modern database servers expose metadata about the databases stored on the database server *as a database itself*. This metadata database contains a wealth of information about the structure of the server's "real" databases and tables. Information exfiltrated from this database is often very useful for mounting subsequent SQL injection attacks.

Move through the security levels as you find SQL injection attacks that succeed. This category is complete when you successfully perform a SQL injection against the code for the *high* security level.

If you're stuck, here are some helpful resources:

- `dvwa` uses MySQL as its backend database server. The *MySQL Functions* chapter of the *PHP Language Reference* (`https://secure.php.net/manual/en/ref.mysql.php`) provides information about PHP's most commonly-used API for MySQL. (A similar chapter exists for PostgreSQL: `https://secure.php.net/manual/en/ref.pgsql.php`.)
- The *OWASP Testing for SQL Injection guide* presents a detailed and methodical process for detecting and exploiting SQL injection vulnerabilities in web applications: `https://www.owasp.org/index.php/Testing_for_SQL_Injection`.
- The *OWASP SQL Injection Prevention Cheat Sheet* contains a comprehensive list of ways that web developers can ensure their code is not vulnerable to SQL injection: `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`.

2 Blind SQL injection vulnerabilities in DVWA

A variant of the standard SQL injection attack is the *blind SQL injection* attack, in which the attacker exploits the presence of a *side-channel* that causes the database server to indirectly leak information about the data it contains via the web application, even if that information is never explicitly shown to the attacker.

The most common side-channel used in conjunction with a blind SQL injection is the amount of time a query takes to execute. If an attacker can submit a query to the database server that takes a noticeably different amount of time to execute depending on some condition set by the attacker, the attacker effectively learns the truth value of the condition by observing how long it takes for the database server to return the result of the query (this is analagous to the concept of a *timing attack* in cryptography). This type of blind SQL injection attack normally involves the following elements:

A synchronous web application. Web applications typically request user input, use a processed form of that input as part of an SQL query, send the query to a database server, wait for the database server to produce a results table, process the results table, and send the result of the processing back to the user. Unless implemented carefully, this pipeline is vulnerable to timing attacks: a slow response from the database server results in an equally slow response to the overall request, which may reveal information about the intermediate processing steps to the user.

A database server that supports a conditional construct. Most do: MySQL has an `IF` expression; PostgreSQL has a `CASE` expression.

An SQL function that takes a long time to return. This could be something as complex as a hashing operation repeated tens of thousands of times, or something as simple as causing the database server to pause for a while (MySQL has the `SLEEP()` function; PostgreSQL has the similar `pg_sleep()` function).

A true/false question to be asked. The information the attacker wants to learn is encoded as a question with two possible outcomes (e.g. "is the first character of Chris's password `a`?").

Repetition. The answer to a single true/false question is rarely interesting; the answer to a series of true/false questions (e.g. "is the first character of Chris's password `a`?"; "is the first character of Chris's password `b`?"; "is the first character of Chris's password `c`?"; ...) often is.

An attacker can combine each of these elements to extract one bit of interesting information from the database server per query, even if the web application in front of the database server doesn't appear to respond any differently to each query.

Let's try some blind SQL injection attacks against DVWA.

1. From the left-hand menu in DVWA, select **SQL Injection (Blind)**.
2. Similarly to section 1, this part of DVWA allows you to query a table of users by user ID, but this time only the presence or absence of a user with the given ID is revealed. Perform a SQL injection against this part of DVWA, causing it to reveal the password hash of Gordon Brown's user account (user ID 2). (The part of the web application that you need to exploit is in the box with the solid black border; the PHP source code for this part of the page is shown when you click the **View Source** button.)

Verify that you recovered Gordon Brown's password hash correctly by cracking the password hash (hint: use John the Ripper, introduced in Tutorial 2, or save time by checking whether someone else has cracked it already!), logging out of DVWA and attempting to log back in with the plaintext password. If you succeed, increase the security level; higher security levels may attempt to introduce noise into the side-channels you're relying on, so you might have to change your tactics. When you find a blind SQL injection attack that succeeds against the code for the *high* security level, congratulations: you've completed this tutorial.

1. Which lines of the DVWA source code in each category are vulnerable to SQL injection attacks?
2. Which lines of the DVWA source code in each category attempt to fix these vulnerabilities? Why are they inadequate?
3. How could these vulnerabilities be fixed properly? Compare your answer with the source code for the *impossible* security level (which is invulnerable to SQL injection) in each category.

3 Automating a blind SQL injection attack against DVWA (Optional)

Because a blind SQL injection attack can only provide an answer to a single binary question, attempting to extract any meaningful amount of information from a database server via blind SQL injection is a drawn-out affair. Here are some ways we could decrease the amount of time it takes to compromise a user's password hash:

- Rather than guessing each character manually, we could write a program to automate a brute-force attack against each character in the hash, moving on to the next character after detecting a difference in the web application's response.
- We could make an algorithmic improvement to our attack: if we know — or assume — that the characters in the password hash belong to a fixed range (in this case, `[0-9a-f]`), we can perform a binary search on the range of characters to reduce the number of true/false questions that need to be asked in order to reveal the value of a single character (e.g. "does the first character of Gordon's password have a value less than `8`?"; "does the first character of Gordon's password have a value less than `8` and greater than `4`?"; ...).

1. Write a program that automatically performs blind SQL injections to recover the password hash for a given DVWA user ID (it needn't be particularly efficient). Make it measure the number of HTTP requests it makes to DVWA, and the amount of time it takes to complete the attack.
2. Implement the algorithmic improvement above (or a more efficient one, if you can think of one), and measure the difference in the number of HTTP requests sent to DVWA and the amount of time taken to complete the attack compared to the program in step 1.