Cyber Advent Challenge - 23/12

Santa's been inundated with Facebook messages containing Christmas wishlists, so Elf Jr. has taken an online course in developing a North Pole-exclusive social network, LapLANd! Unfortunately, he had to cut a few corners on security to complete the site in time for Christmas and now there are rumours spreading through the workshop about Santa! Can you gain access to LapLANd and find out the truth once and for all?

NB: You may attempt to bruteforce the login page using a tool such as THC-Hydra for this challenge, your mileage may vary!

SQL Injection (SQLi)

SQL databases are a type of relational database widely utilised in web applications. There are several implementations of SQL servers available from vendors such as Microsoft and Oracle, as well as open-source products such as MariaDB.

Within an SQL database, there are tables containing rows and columns. Each row is an entry in the table, and the data in each column can represent a wide range of things! For example, we could have a table called 'Customers' which contains information about people who shop at our online store. We could have columns such as "CustomerName" to store their name, "City" to store the city they reside in, "DOB" to store their date of birth and so on. Each row would represent a unique customer, and the data stored in each column on that row would be associated with that customer (entry).

SQL (***S**tructured **Q**uery **L**anguage)* queries are fairly straightforward and offer an English-like syntax. For example, you could query a table called 'Customers' and retrieve the customer names and the cities they reside in using the following query:

```
SELECT CustomerName, City FROM Customers;
```

Wildcard operators you may be familiar with, such as ***** also exist within SQL, for example, we could modify our above query to retrieve all details associated with each entry in the Customers table.

```
SELECT * FROM Customers;
```

So, from a security perspective, how can we take over a SQL database? There's quite a motivation to retrieve valuable information from a corporate database after all.

It's quite unlikely that we'd be exploiting a vulnerability in the SQL server itself, but instead we can always rely on poor implementation by lazy or inexperienced developers! The Cyber Advent

challenge today is based on a very popular Udemy course, but a few corners were cut to get the web application released in time for Christmas!

It's relatively simple to connect a web application to a SQL server, and it's also particularly easy to implement this in a very insecure manner. In this example, let's look at a SQL statement implemented in the PHP programming language:

```
$email = $_POST['log_email'];

$check_database_query = mysqli_query($con, "SELECT * FROM users WHERE email='$email' AND password='$password'");
```

Don't be put off if this doesn't make a lot of sense right now! What's important is understanding the process that's occurring here. The first line is specific to PHP, and is assigning the input from a form (user-defined input) into a variable, $email. The second line then queries the database to SELECT information from the SQL database, identified by the email AND password that the user supplied (this is a basic authentication system).

The fundamental issue with this implementation is that the user's input is not being sanitised in any way, we are blindly trusting our users, which is a terrible idea at the best of times. See how the $email variable is surrounded by single-quotation marks? As an attacker, we can take advantage of this! We can cleverly craft our 'email' input to instead break out of these quotation marks and allow us to execute arbitrary SQL queries of our choosing.

For example, if there was no input sanitation on the email field, we could simply put a ' before a SQL statement of our choosing. We could potentially exploit boolean-logic to bypass this authentication system by crafting a SQL query that always returns as True. Confused? That's to be expected. Let's take a deeper thought.

We know that 1 is always equal to 1, there's no possible scenario where this would not be true. We can use this fact to bypass any conditions an existing SQL statement is looking for. Let's take a section the query from above:

```
SELECT * FROM users WHERE email='$email'
```

The query will only return entries if the email column matches the email input provided by the user. OR we could use our quotation mark in combination with the knowledge of 1=1 to create a completely different SQL statement:

```
SELECT * FROM users WHERE email='$email'OR WHERE 1=1--'
```

Now it doesn't matter what we enter as our email, because this statement will check if the email entered matches OR if 1 is equal to 1, which we know is always true. Therefore, our new statement will just pull all the records from the users table, because 1 will always equal 1.
If you're eagle-eyed, you'd have noticed the additional dashes -- appended to that statement. This is sometimes necessary. The double dash comments out the rest of the existing SQL query that exists after our injected query, this double dash ensures our query is a valid SQL query and there aren't any mismatched quotation marks (remember we inserted an additional one before).

In practice, you could undertake this by simply entering your email address as:

```
'OR WHERE 1=1--
```

In the e-mail field of the website you're attacking.

This is a very basic example of SQL injection and thankfully most developers know to sanitise their inputs, at least to an extent. You can learn more about SQL injection in various rooms on TryHackMe - https://tryhackme.com/room/sqli, https://tryhackme.com/room/uopeasy, https://tryhackme.com/room/jurassicpark, https://tryhackme.com/room/ccpentesting.

It can be rather time-consuming to get the perfect SQLi for any given scenario. Thankfully, automated tools exist! The two most popular being SQLmap and the ever-useful Burp Suite. It's important to note that automated SQLi tools are rather easy to detect from a SysAdmin perspective, think of them as analogous to a bull in a china shop! SQLi tools typically attempt lots of techniques in a very short period of time and do not look remotely similar to normal user behaviour. It's still very valuable to learn the fundamentals of SQLi by hand.

SQLMap is a command-line tool that is pre-installed on most pentesting distributions. It offers several options depending on the kind of web application being attacked, but also usefully offers a wizard to guide newcomers through the process. SQLMap is rather intelligent as it suggests parameters that may be worth investigating further, as well as eliminating attacks that may not be relevant (for example, MSSQL specific injections when a MySQL DBMS has been detected). In some scenarios, we don't get visual feedback from a SQL command when attempting a SQLi, this is known as a blind injection. Blind injections are much more difficult, as you won't know you've got it right (or are getting close) until everything clicks! SQLMap can perform blind timing-based attacks which exploit the execution time of queries. By timing the response times on a variety of queries, SQLMap can enumerate data from a database, albeit at a significantly slower rate than just being able to dump all the information at once. SQLMap also automates a lot of the more laborious tasks, such as determining UNION select queries.

There are various levels of "risk" and "depth" that can be configured for SQLMap, should no possible SQLis be detected at lower levels. Sometimes the tool may find a sub-optimal SQLi (for example, a very slow timing attack) for scenarios where faster SQLis exist. This may occur in

this Cyber Advent challenge if you decide to use the tool (try flushing the session, forcing a dbms and increasing the risk and/or level).

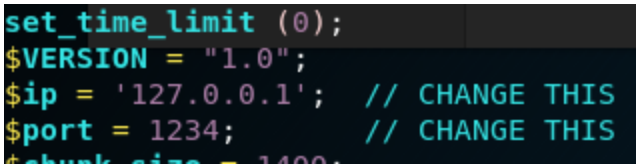
If you're unsure as to which tables may exist in one particular database, you may be able to just dump the whole contents of the database using the *--dump* option, but it may be much more sensible to target important tables of interest (dumping a whole database with a time-based blind attack may take a long time!). A better technique may involve enumerating the layout of the database (tables), enumerating the column names of a table of interest and from there, extracting information. For example, if you're looking for a particular user's email and password and you know their first name, it would make sense to find the table containing this information (users) and extracting only the useful columns you'd want (first_name, email, password). Table names might not be exactly what you expect, hence why it's generally a good idea to enumerate the layout first, so you don't waste time.

Uploading web shells

Beyond SQL injections, it's even more important to sanitise any files you allow users to upload to a web application! Improperly configured file uploads could leave a server wide-open to an attacker to upload various nefarious scripts, in particular web shells.

A reverse web-shell is a script that is executed on a target machine which sends out a connection to an attacker-controlled machine. The attacker-controlled machine is then able to execute commands on the target machines, which gives the opportunity for data exfiltration and privilege escalation. There are several web shells included in Kali Linux (*/usr/share/webshells/*) for a variety of web server languages. These scripts are designed to be easily modified for use.

For example, let's take a look at the PHP reverse shell included in Kali Linux:



It's highlighted we need to change the IP and port for our own usage. This IP and port should be the IP of the attacker machine (i.e. your Kali VM) and the port should be an available port to be used for the reverse-shell connection. 1234 is typically okay to use, 4444 is also another popular option.

Once this file is uploaded to a target machine, it will need to be executed. In the case of a PHP reverse shell, this is usually triggered by navigating to the location of the uploaded PHP script. It is normal (and good news!) for the web page to infinitely load when trying to navigate to the reverse shell script, this usually means the script is successfully running.

So how do we actually control the target machine? We need to have a listener set up for when the reverse shell is run and the connection request is made. The easiest way to do this is to use netcat, which is installed on most pen-testing distributions by default. For example:

```
nc -nvlp 4444
```

Would set up a listener on port 4444. If the PHP reverse shell is configured to operate on port 4444, your listener would receive the connection request when the script is run on the target machine. If all goes as planned, you will have a command terminal prompt on your screen!

The reverse shell is typically fairly basic and the connection is reliant on the web server running the PHP file being run (so try to avoid accidentally closing your tab!). It's usually a good idea to try and establish persistence and/or upgrade the shell you have to something a bit nicer). If you lose your reverse shell connection, try and reload the script you uploaded (and hope it hasn't been deleted by an angry SOC analyst!).

Be wary, developers actively try and prevent this from happening (for good reason) so may blacklist files from being uploaded based on filetype/file extension! You may need to think of alternative formats that bypass blacklists, but are also still interpreted by the web server for execution (**hint hint**). If you are attempting the challenge and your reverse shell seems to not work (the web server just prints the contents of the uploaded file), this means the web server is not interpreting the file as PHP! Try a different file format, there's quite a few