BTN 710 Deliverable 2 - Group 1

Neil An - Team Leader

Tyerelle Toppin - Web Master

Alan Sciulli - Report Lead

Andrew Donets - Video Presentation Lead

Table of Contents

Introduction

This paper explores the vulnerabilities that are planned to be implemented in our website. Exploits are outlined in detail, and explanations of accompanying attack processes are provided. The website has a host of vulnerabilities, but this paper focuses on three main exploits: SQL injection, poor password hashing, and cross-site scripting. After the vulnerabilities are described, the system and hardware are outlined, and examples of attacks are provided. The research aims to find solutions for all three of the vulnerabilities outlined.

Section 1: Vulnerabilities

The three vulnerabilities we are going to explore in this report are SQL injection, incorrectly and poorly salted/hashed passwords, and cross-site scripting (XSS).

SQL Injection

The first vulnerability, SQL injection, is one of the most famous vulnerabilities of all time. It's a technique that sends a malicious SQL string to a web application that uses SQL to perform unauthorised database queries. In the worst case scenario, we would have full access to the database's contents, we would be able to modify the entirety of the database, and we would even be able to completely delete the database. The best way to prevent SQL injections is to validate user input, use parameterized queries and prepared statements, and sanitising the user input at every stage.

A website is vulnerable to this attack because a lot of websites have a login system. In particular, our website will have a login system that will be susceptible to SQL injection attacks. SQL injection attacks occur through faults in the login system. For example, instead of entering a username in the username field on the website, someone can instead execute a SQL statement that would retrieve all users in the database. For example, the SQL statement:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

This statement would return all users from the Users table since 1=1 is always true.

Poorly Hashed & Salted Passwords

The second vulnerability is hashing passwords using a weak hashing algorithm, and improperly securing salts. When we store passwords, it's incredibly crucial that we do not store those passwords as plain-text, we want to obscure the real password as much as we can, and we can do that by hashing and salting a password. By hashing a password, we put the password into a hashing algorithm (like SHA), and the result is a string of random characters and numbers. Crucial to these algorithms is that the same input will always result in the same output, and that we cannot perform the process we used to generate the hash in reverse. There are, however, weak hashing algorithms that were designed to be fast first, making them very bad for password hashing, such as MD5, SHA-1 and RIPEMD. If we hash

our passwords using these outdated (or broken) algorithms, it can take mere seconds, minutes, or hours to bruteforce a hashed password that uses one of these algorithms. If we salt those passwords before putting them through a stronger hashing algorithm, our passwords will be a lot harder to crack. Securing our salt however is also critical, if an attacker obtains the salt we used to salt our passwords, then we must consider that our passwords are effectively reverted back to their unsalted state.

Again, a website with a login system would be vulnerable to password cracking attempts. Also, if an attacker was able to get into our database, then they would be able to access our stored passwords. If those passwords are stored in plain text, or hashed with the aforementioned poor hashing algorithms, then everyone's account would be compromised.

Cross-Site Scripting

Cross-Site Scripting (XSS) is an exploit where the attacker compromises the user's interactions with the web application. The attacker requests the user to carry out actions that they have permission to do, and accesses the user's data. This allows the attacker to have full control over the web application's functionality and data. XSS attacks can be prevented by using combinations of different methods such as filtering users' input on what it's expected to be or validate it, encode data output from the user to server, use correct response headers and implement Content Security Policy.

A website can be vulnerable to cross-site scripting attacks in multiple ways. One possible way is through injection, where the attacker will inject some malicious script into a user input field, such as a username field. Another way is through storing the malicious code inside the database from user input. The final way is by storing the malicious script inside the website itself, where the script will execute whenever a user loads the site.

Section 2: System Setup

Hardware

The hardware we are going to be using to demonstrate our attacks is on a virtual hardware situated on our stations that was setted up through the software called Workstation 5 Pro.

Operating System (OS)

The Operating System (OS) we are going to be using to demonstrate our attacks on is Kali Linux.

Protocols/Services/Application

The Application we are going to be using to present our attacks is called BeEf (The Browser Exploitation Framework).

Descriptions And Diagrams Of Network

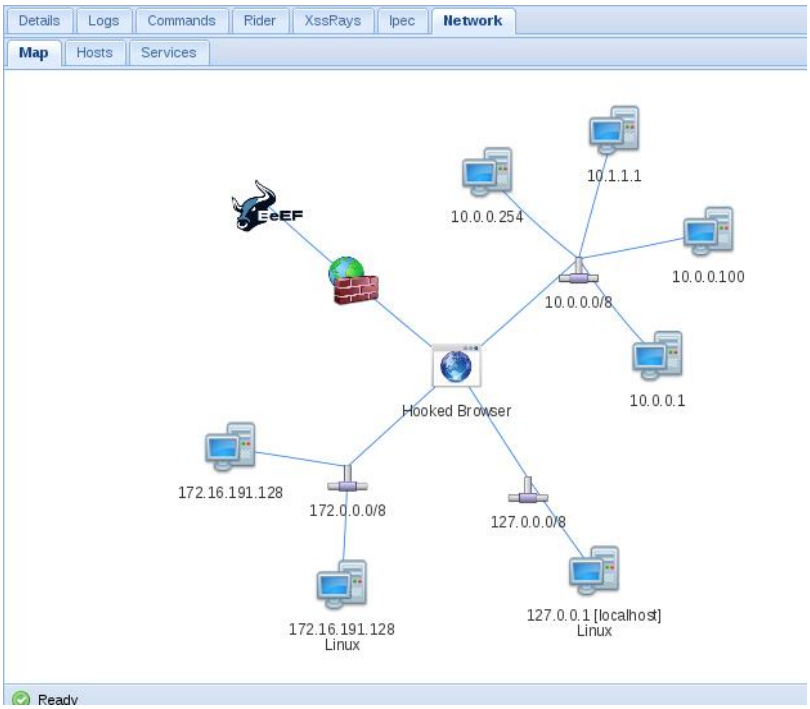

Fig 2.1 (BeEF hooked Network)

Protocol/Service Description

BeEF is a penetration testing testing tool that is used and mainly focuses on the web browser. With BeEf we can access the security posture of a targeted environment using client side attacks, which exceeds network perimeters and client systems, and examines any possible exploitable open doors known as the web browser. Once a web browser is hooked, we will use BeEf to carry out commanded attack modules against a user from within the browser.

Section 3: Exploits

SQL Injection

SQL injection works by allowing an attacker to retrieve/modify data from a database through an application such as a website. In the case of a website, SQL code is injected through a form or field such as the page where user's login. Websites are vulnerable to these attacks, because when they send a login request to the backend, they can instead send SQL code and interact with the database in the backend. An example of an attack is found in the diagram below:

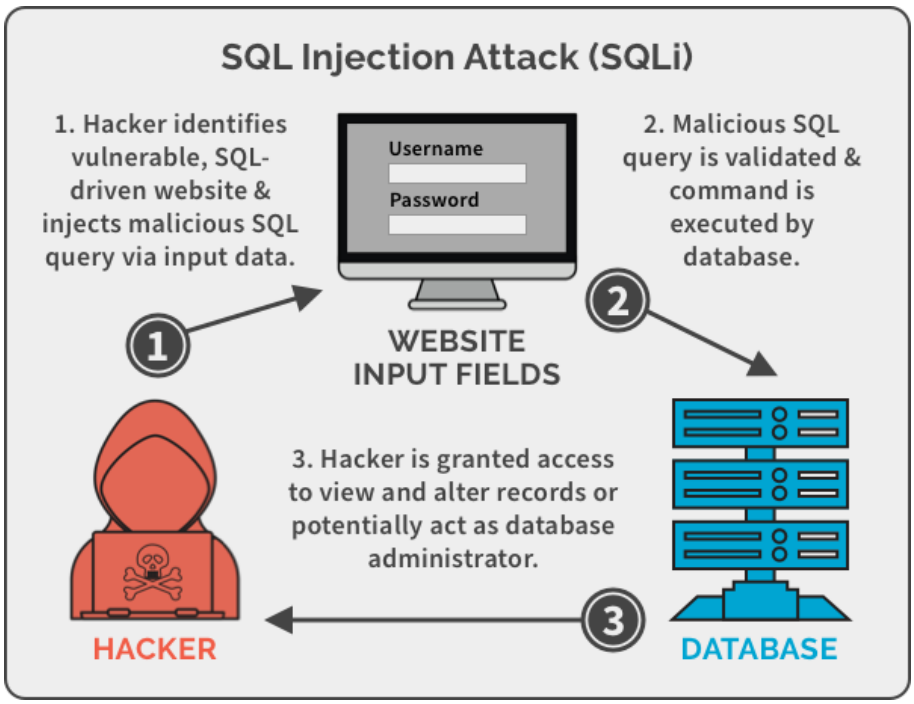

Fig 3.1 (SQLi)

Anyone can attack using SQL injection, as long as there is some form or field on the website that is connected to the database in the backend. No special tools or programs are needed for the exploit to work.

To execute the exploit, the attacker must find a vulnerable website. There are many web vulnerability scanners available online that can test to see if a website is vulnerable to attacks. An attacker can also test if any errors or anomalies occur by entering a single quote (') into the entry point of the website. If the website is found to be vulnerable, then the attacker can start injecting SQL into the website. The diagram below shows an example of how SQL code can grab user data.

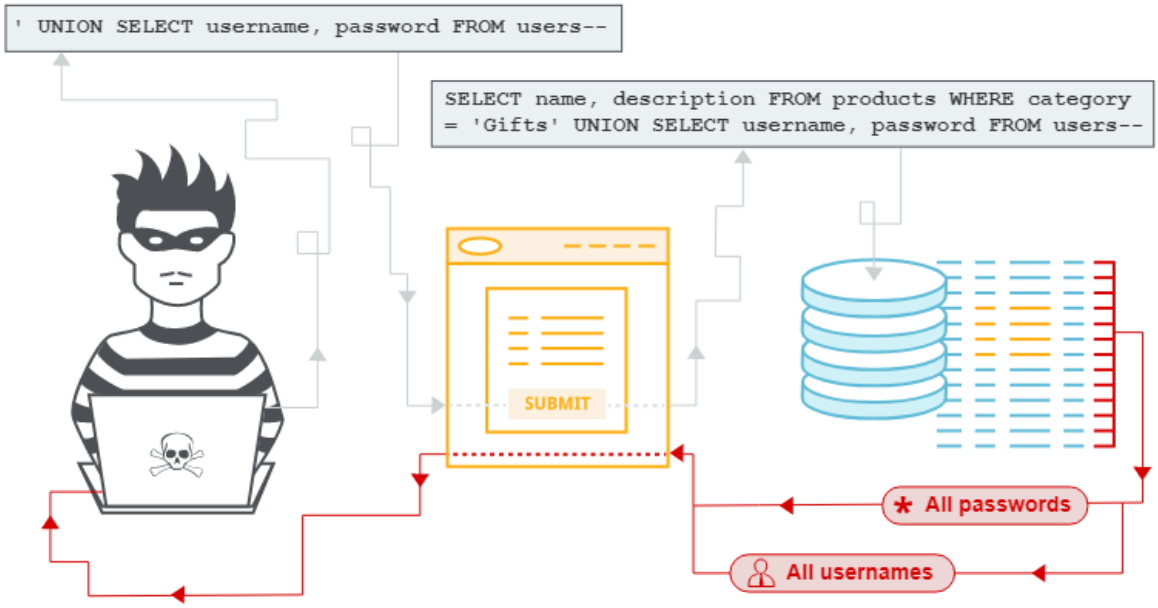

Fig 3.2 (SQL injection examples)

There are multiple ways an attacker can use SQL code to get the information they want. They can use the UNION keyword to grab data from another table like in the figure above. The attacks can also be more direct. For example, an attacker can enter:

```
" or ""="
```

into the username and password field and the server code will automatically create and execute the following SQL statement:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The database would return all rows from the Users table.

There are no signatures or traces of the attack left with SQL injection. This is because the attacks occur through the website and are executed in the backend where no logs for queries can be found.

Cross-Site-Scripting (XSS)

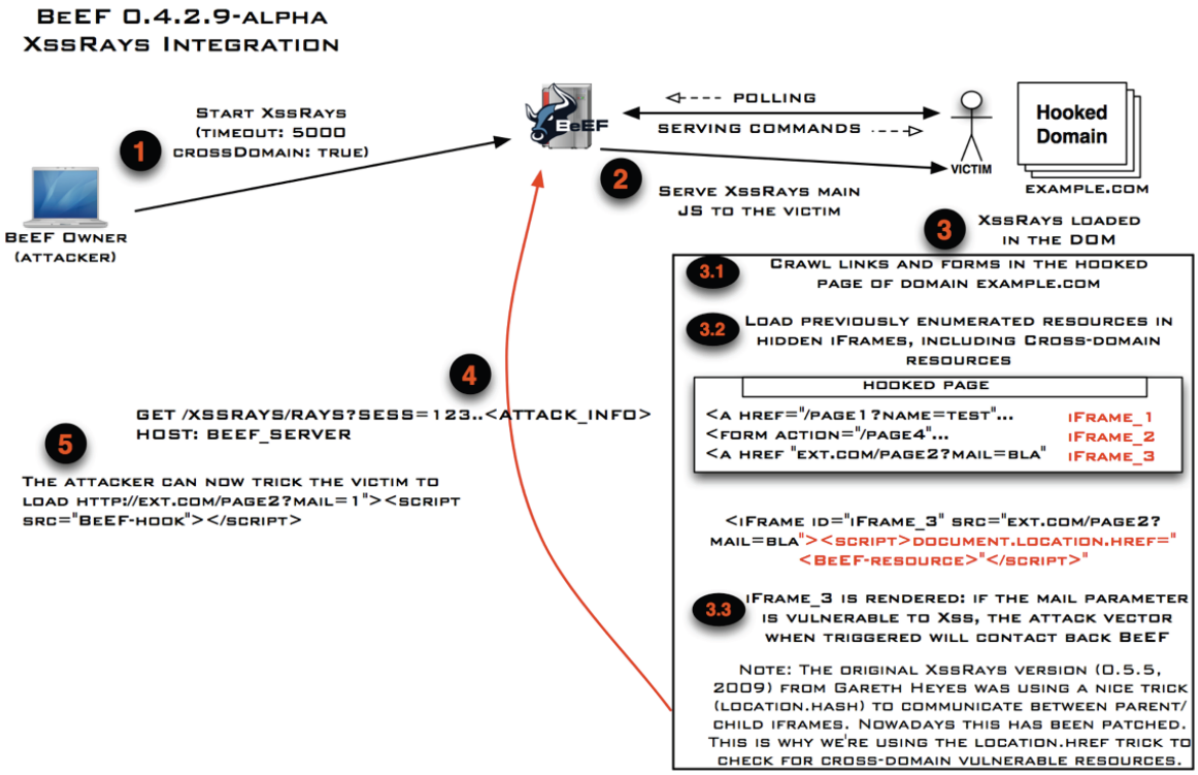

Fig 3.3 (BeEf XSS Rays Integration)

First we will have the user get hooked by clicking on a link embedded in the website. After the user is hooked we will then proceed to inject the XSS Rays into the browser to scan for XSS vulnerability. Once a vulnerability is found, it can be found in the XssRays -> Logs.

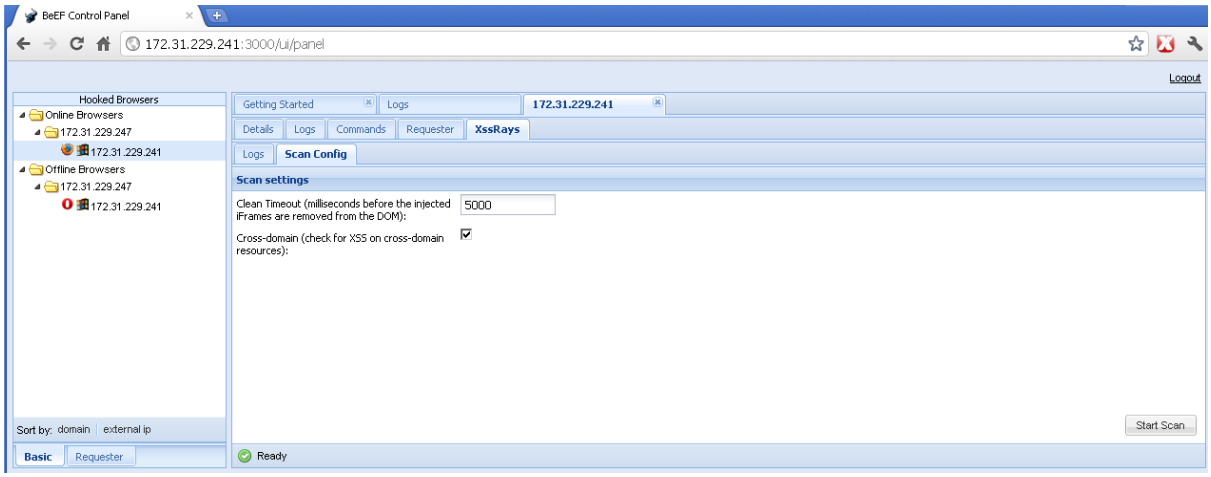

Fig 3.4 (Scan config)

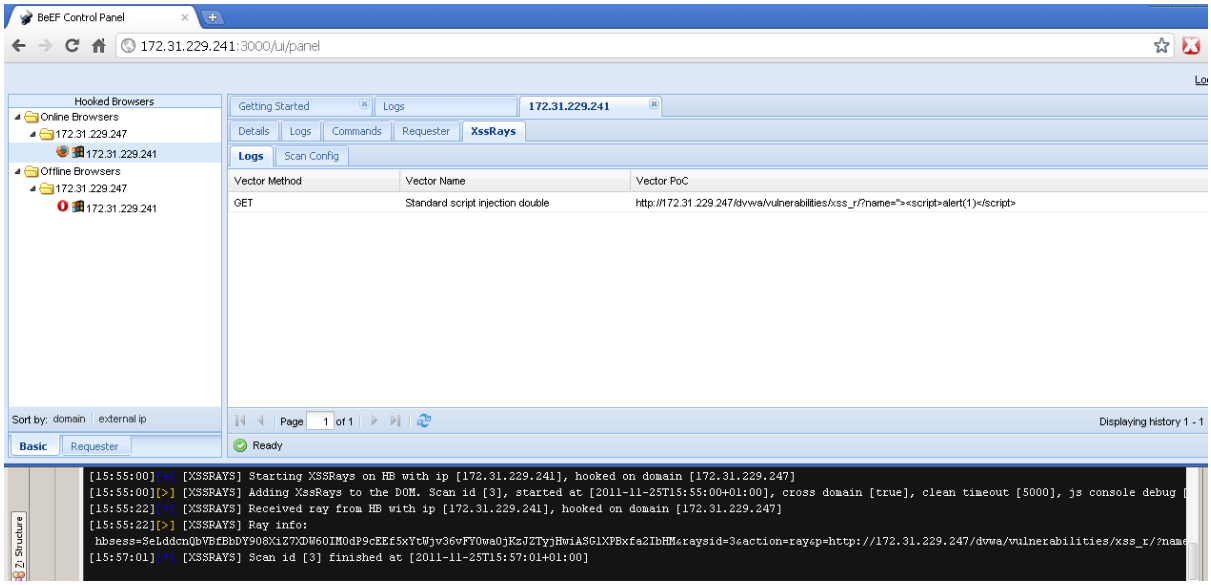

Fig3.5 (XSS Rays Logs)

With direct access to the user's web application, we can test the XSS Rays finding through the use of PoC that is provided in the Logs tab. If direct access isn't available at the time of our XSS attack, a command module can be utilised, that will trigger the user to open a link that is pointing towards a vulnerability resource. After utilising this tool, our attack surface has been expanded, in which we will have both the user's online and offline hooks, and the application hooked with the XSS found by XSS Rays.

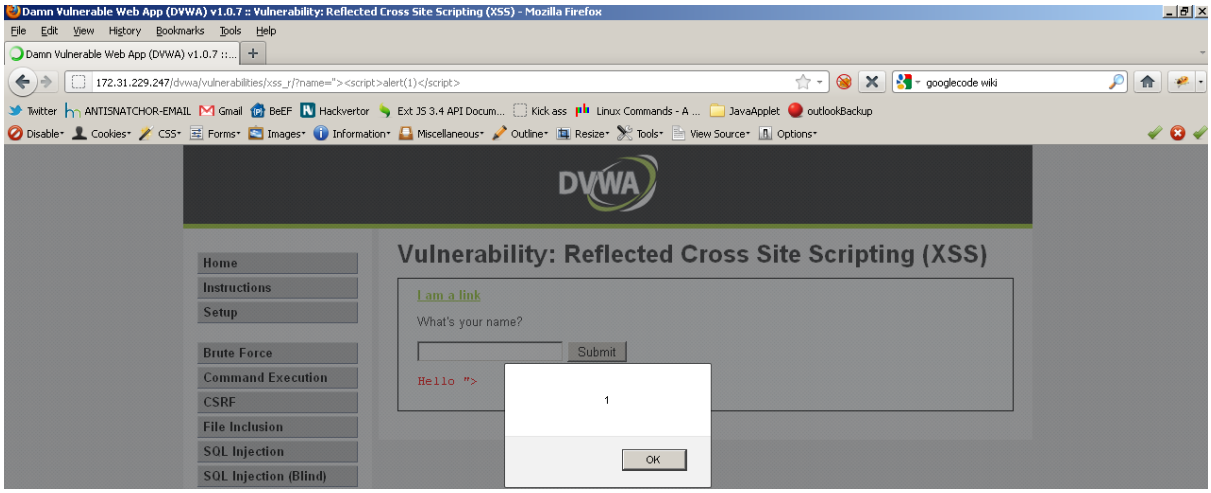

Fig 3.6 (XSS Attack)

Poor Password Hashing Algorithms

This exploit involves an attacker bypassing the system's password protection protocols. The system hashes user passwords with the MD5 hash encryption algorithm, which leaves the passwords in a vulnerable state.

How the Exploit Works

MD5 is a hash algorithm that is primarily used for data authentication. It encrypts data by running it through a mathematical algorithm and converting it from plain text into a string of 32 characters (Avast, 2022). The resulting hashes are most commonly used to verify that the proper data is received without that data being tampered with. Although ill-advised, it is possible to use MD5 for the purpose of data encryption. A problem results from exclusively using MD5 to encrypt data, as the algorithm was designed to be fast. It is vulnerable to not only rainbow table attacks, but even today's home computers can brute-force or billions of MD5 hashes per second.

When a piece of data is inputted into the MD5 hashing algorithm, the output will be the same every time. This is crucial to how any hashing algorithm works, however this leaves us vulnerable to rainbow table and word-list attacks. For example, attempting to encrypt the word 'frog' will always result in the following hash: 938c2cc0dcc05f2b68c4287040cfcf71 (Avast, 2022). Therefore, attackers can easily crack the encrypted data by following a series of steps, outlined below (screenshots taken from personal machines):

1. Obtain the hash of the password to be cracked. One example of a method an attacker could use to accomplish this would be to access the database itself, where the hashed passwords are stored.
2. Determine what algorithm was used for the hash. The attacker analyzes the properties of the hash string, looking for key features to determine which algorithm was used (for example, as explained above, an MD5 algorithm would contain 32 characters, but there could be other indicators in a system).
3. Insert the hash into a tool to decrypt it. An example of a tool the attacker could use for this step would be 'Hashcat'. The following steps outline how an attacker can decrypt a hash using Hashcat and a word dictionary.
    a. Open a command line window on a machine with Hashcat installed.
    b. Retrieve the hash. For example, 938c2cc0dcc05f2b68c4287040cfcf71.
    c. Navigate to a directory with a word dictionary, which will be used for the decryption. A word dictionary is a file with many words and phrases in it.
    d. Run the following command: "hashcat -a 0 -m 0 938c2cc0dcc05f2b68c4287040cfcf71 dictionary.txt" ('-a' = attack mode, in this case '0' is 'straight'; '-m' = mode, in this case '0' is 'MD5'; 'dictionary.txt' = the word dictionary. See example below)

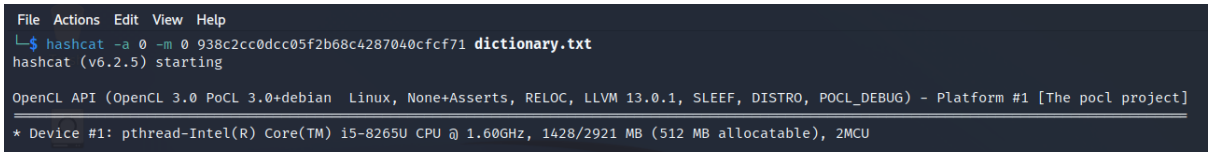

Fig. 3.7 (Hashcat command)

e. Results will show the decrypted data if the dictionary found a match. For example:

```
938c2cc0dcc05f2b68c4287040cfcf71:frog

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 938c2cc0dcc05f2b68c4287040cfcf71
Time.Started.....: Sat Oct 15 21:19:03 2022 (1 sec)
Time.Estimated...: Sat Oct 15 21:19:04 2022 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (dictionary.txt)
```

Fig. 3.8 (Results of Hashcat Decryption Command)

It should be noted that even if a given MD5 hash isn't in a dictionary list or word-list, we can still attempt to brute-force it out-right. The following demonstrates how trivial it is to brute-force an 8 character hashed password that contains upper and lower case characters, and numbers. Below is our password, "Btn710a1" hashed using MD5:

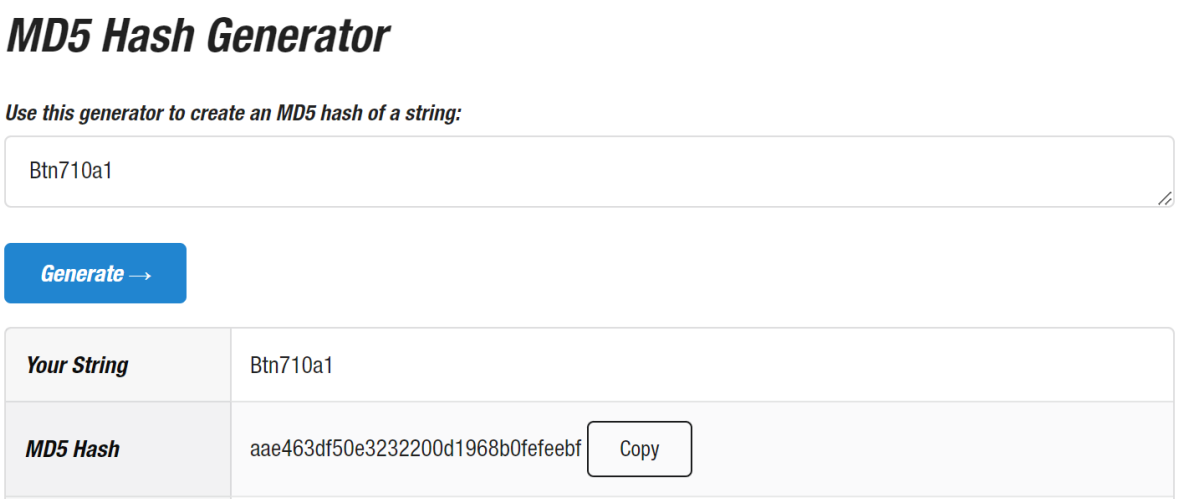

Fig. 3.9 (MD5 Hash result of "Btn710a1")

Now we'll once again use Hashcat to crack our hash "aae463df50e3232200d1968b0fefeebf", but this time we will use the brute-force attack option. Entering the command "hashcat -O -m 0 -a 3 a1.txt", we can run our brute-forcer. Note we change our attack flag from 0 to 3 to indicate the switch to brute-forcing, -O is added for performance purposes only.

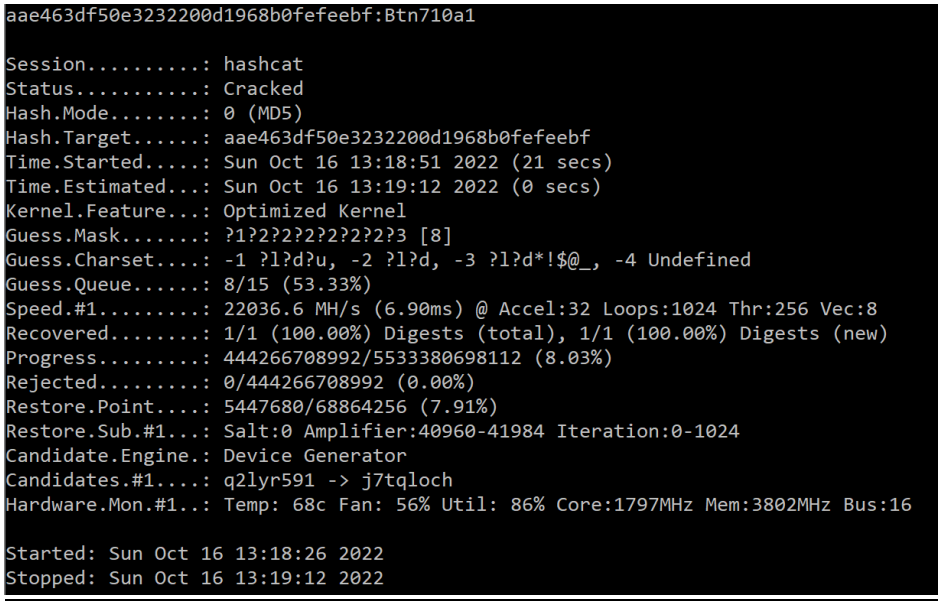

Fig. 3.10 (Successful brute-force of MD5 hash "aae463df50e3232200d1968b0fefeebf")

As we can see in figure 3.10, our 8 character password was brute-forced by Hashcat in only 21 seconds. To emphasise just how fast MD5 hashes can be brute-forced, we can see that hashcat ran through over 444 Billion hashes in those 21 seconds.

Signature of the Attack

In this exploit, virtually no signature is left by the attack. The reason for this is because this is an 'offline' attack (Avast, 2022). This attack is done independent of the application – it involves the attacker gleaning data, then putting that data to use somehow. The only traces that are left are network traffic created by the attacker logging in to a user's account, and that assumes the attacker logs in to the account in the first place.

Section 4: Security Policy and Controls

How To Protect Against SQL Injection

How can the user running vulnerable software protect themselves?

It's difficult for the end-user to know if a system is vulnerable to SQL Injection, especially if they are not tech savvy. Users in this case should follow general principles of safe data sharing online, as can be applied to any other website or program. Users should avoid entering and sharing their personal information on websites they may not trust, or use often. Users should create randomly generated passwords for each website, so that a data breach in one website will not compromise the unique passwords used on other websites. Throwaway and secondary email addresses can also be considered.

How can the vendor fix the vulnerability?

There is no method of fully patching SQL injection, however we can adhere to the following practices that can help prevent the easiest methods of attack.

The greatest method of defence against SQL Injection is to use parameterized statements in place of string concatenation. Parameterized (also known as prepared) statements should generally always be used when user input can never be trusted in a query we're running, especially if we're using WHERE, INSERT or UPDATE statements.

```
String ourQuery = "SELECT * FROM ourtable WHERE something = '"+ input ="'";
Statement runStatement = connection.createStatement();
ResultSet ourResults = runStatement.executeQuery(ourQuery);
```

Fig. 4.1 (Our unsafe SQL code that concatenates the user input string into the statement)

Figure 4.1 shows our vulnerable SQL code. Because of string concatenation, it becomes clear how a malicious input now becomes part of our query. It's relatively simple to fix this code however, as shown in Figure 4.2.

```
PreparedStatement runStatement = connection.prepareStatement("SELECT * FROM ourtable WHERE something = ?");
runStatement.setString(1, input);
ResultSet ourResults = runStatement.executeQuery();
```

Fig. 4.2 (Our Prepared Statement)

Figure 4.2 shows our now (relatively) safe SQL code using prepared statements, though while this is a good demonstration, it's still not considered perfect for all implementations, such as table or column names, but that is considered already very poor design, and should not be used in the first place.

Prepared statements aren't the only mitigations against SQL injection, it's generally recommended to perform penetration testing wherever you can on your website or implementation. The rule of the lowest level of privilege should be applied wherever user input cannot be trusted (which should be never), application modules that might rely on user input should prevent returning or writing data outside of the scope of what the user is realistically allowed to do.

How To Protect Against A Poor Hashing Algorithm

How can the user running vulnerable software protect themselves?

This exploit is largely an 'offline' exploit, which means that the attacker(s) do not interact directly with the website or application, instead with already obtained hashes. This means that there is no real way for the user to protect themselves, as the user is not running vulnerable software in the first place. A user can't know if a vendor or website is storing their passwords and other data using a strong, or weak hashing algorithm, however preventative measures can be taken by the user.

As previously discussed, rainbow-table and dictionary attacks are typically composed of common, and weak passwords, such as *password, awesome1, trees,* etc. A user should create a simple to remember password, but one that can't immediately be found in a list, a book, or thought up by someone, combining multiple words is even better. The use of lowercase, uppercase and numbers is strong against human guesses, but not against computers, but adding in special characters makes it even more time consuming for computers. In the previous MD5 hash brute-force example, hashcat was only brute-forcing using upper, lower and number characters, we can also add special characters into the brute-forcing, but this now creates billions of new combinations we must brute-force.

The ultimate deterrent a user can perform against brute-forcing however is creating a unique long password using the four character sets mentioned for every website. When we brute-forced our 8 character MD5 password, while it took Hashcat 21 seconds, we got lucky as we only went through the first 8% of all possibilities before finding the match, though the theoretical max would be 4 minutes 22 seconds. A single character increase however exponentially increases the amount of time it takes to brute-force all possibilities.

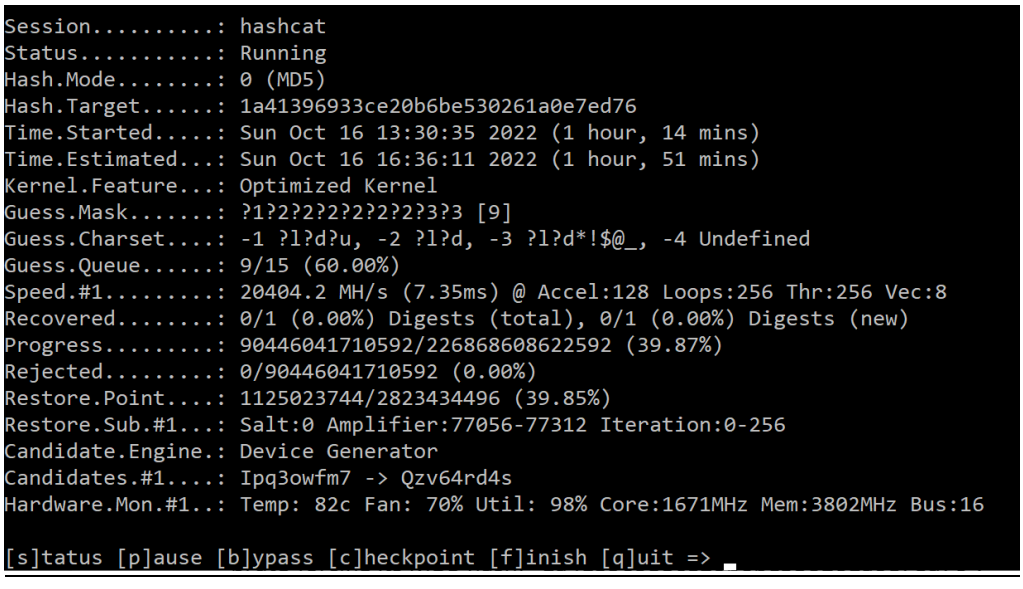

Fig. 4.3 (1.25 hours passed trying to brute-force MD5 hash "1a41396933ce20b6be530261a0e7ed76" with an estimated 2 hours remaining)

For example, in Fig. 4.3, we created the 9 character long password "9CharLong", and we've begun brute-forcing its MD5 hash, "1a41396933ce20b6be530261a0e7ed76". At the

time of the screenshot, the hash was still not yet cracked, even after over 1 hour and 14 minutes of brute-forcing. This password uses the same 3 character charset as our 8 character password, the only difference being a single character. 3.5 hours is still a short amount of time for a dedicated cracker, however if we simply add more characters, estimated run-times can reach months, years, or decades. If we add special characters, this adds even more combinations, and therefore time for a brute-forcer. While this is no excuse for a piece of website using MD5 password hashing, a user can realistically be safe if they generate a unique password for every website containing a minimum of 12 characters that uses uppercase, lowercase, numbers, and special characters.

How can the vendor fix the vulnerability?

The vulnerabilities of weak hashing algorithms like SHA-1 and MD5 cannot be patched; these algorithms are fundamentally designed to be fast, and in turn they are not safe for password hashing. Realistically, a vendor should never implement this at any stage for password hashing. If it's already implemented, the vendor should immediately switch to a more secure password hashing algorithm such as B-CRYPT for new password creation. This however, complicates things as we can't just convert our already generated MD5 hashes to B-CRYPT hashes. A vendor could implement a system where users who have generated MD5 hashed passwords must reset their password on their next login, then hashing that new password with B-CRYPT, deleting the old password, however we can implement salting to protect not just our vulnerable MD5 hashes, but to even further secure our newly generated B-CRYPT hashes.

Salting takes a value generated by us, and crucially only known by us, and combines it with what we hash. For example if our salt is "mySaltThatIsUsually32CharsOrMore", we add it to "usersPass", and then hash it, the resulting hash practically impossible to be brute-forced, or lookup by any list. It is critical that our salt be safely stored and used separately, if an attacker knows our salt, then the original hash is effectively exposed.

How To Protect Against Cross-Site Scripting

How can the user running vulnerable software protect themselves?

Cross-Site Scripting is an attack that primarily focuses on the host and vendor of a given website for the most part. The goal of attackers using cross-site scripting is typically to access either the user data and/or the site's data. Users should exercise safe online practices as discussed in the SQL Injection protection section as there is not much a website user can do against this exploit.

How can the vendor fix the vulnerability?

Cross-Site scripting is difficult to fully protect against due to all the possible entry points of attack depending on a website's implementation. The following isn't an exhaustive list, especially as the countless number of web frameworks have their own large lists of XSS mitigation policies, however this will outline the most effective, and generally recommended guidelines to protect against the most common forms of XSS.

Much like SQL Injection, we cannot trust any user input. A vendor's implementation should filter all received user input, and compare whether the given input is expected or valid. This can be difficult to fully implement in especially larger applications, as it can be easy to miss a certain edge case. That's why thorough input validation through unit testing is widely recommended.

A vendor's implementation might be expected to output user-controlled data in its HTTP responses, but to prevent it as being interpreted as active content, that data should be encoded, either through Javascript, HTML, URL encoding dependent on the implementation.

HTTP responses are also susceptible to XSS, specifically in headers that shouldn't contain any HTML, Javascript or CSS. We can help prevent this by implementing `Content-Type and X-Content-Type-Options` in those headers to ensure the browser is interpreting everything correctly.

One of the best lines of defence against many XSS attacks is to set up a Content Security Policy for a given implementation. This can be created by configuring our implemented web server to return the `Content-Security-Policy` header, or placing it into a `<meta>` tag. The policy can be set up so that it dictates whether all content should only come from a vendor's web server, trusted domains, image, audio and video sources, whether it must use TLS, and many more options. A vendor should thoroughly research what content policies it should implement that best fit their implementation and provide the most security to their service.

Conclusion

In conclusion, all three vulnerabilities can be difficult to completely fix, as there is no way to guarantee that the data is protected. However, there are strategies that both users and system administrators can take in order to reduce the likelihood of successful attacks.

If form fields and other input methods are not secured properly, SQL injection can penetrate the application or website and deal damage to the systems in place. No signatures are left with SQL injection, as the queries are all executed in the backend, and that makes it especially dangerous. It emphasises the need for users to create unique passwords for every account with sensitive data, so that other accounts are not compromised in the event of an SQL injection breach.

Proper steps must be taken in order to protect the passwords of users. Basic encryption is not enough to keep the data safe, as there are many decryption tools available that can crack simple algorithms. If using a vulnerable algorithm such as MD5, one must first take actions such as salting the data so that the algorithm encryption is not so easily breached.

Bibliography

Avast. (2022, October 6). *What is The MD5 Hashing Algorithm and How Does it Work?* Retrieved October 13, 2022 from https://www.avast.com/c-md5-hashing-algorithm

Asamborski (2017, April 7). *News*. Network Security News. (n.d.). Retrieved October 16, 2022, from https://asamborski.github.io/cs558_s17_blog/

Beefproject. (n.d.). *Home · beefproject/Beef wiki*. GitHub. Retrieved October 5, 2022, from https://github.com/beefproject/beef/wiki/

Content security policy (CSP) - http: MDN. HTTP | MDN. (n.d.). Retrieved October 16, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP#:~:text=Content%20Security%20Policy%20(CSP)%20is,site%20defacement%2C%20to%20malware%20distribution.

Oza, S. (2020, February 4). *SQL injection attacks - web-based app security, part 4*. Spanning. Retrieved October 15, 2022, from https://spanning.com/blog/sql-injection-attacks-web-based-application-security-part-4/

Pauli, D. (2016, December 23). *Security! experts! slam! yahoo! management! for! using! old! crypto!* The Register® - Biting the hand that feeds IT. Retrieved October 16, 2022, from https://www.theregister.com/2016/12/15/yahoos_password_hash/

Penetration testing and ethical hacking linux distribution. Kali Linux. (2022, October 4). Retrieved October 5, 2022, from https://www.kali.org/

Protecting against SQL Injection. Hacksplaining. (n.d.). Retrieved October 7, 2022, from https://www.hacksplaining.com/prevention/sql-injection

simonwhitaker. (n.d.). *Passwords: Of MD5 and mistresses*. Gist. Retrieved October 15, 2022, from https://gist.github.com/simonwhitaker/4079504

The Browser Exploitation Framework Project. BeEF. (n.d.). Retrieved October 5, 2022, from https://beefproject.com/

What is cross-site scripting (XSS) and how to prevent it?: Web security academy. What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy. (n.d.). Retrieved October 16, 2022, from https://portswigger.net/web-security/cross-site-scripting

What is SQL injection? tutorial & examples: Web security academy. What is SQL Injection? Tutorial & Examples | Web Security Academy. (n.d.). Retrieved October 15, 2022, from https://portswigger.net/web-security/sql-injection