Introduction

The topic of this essay is SQL injection attacks. More specifically we will be discussing what role they play in 2018, in light of the plethora of technological innovations we have witnessed so far in the 21st century. Before we do that however, it would seem fitting to proceed with the following structure; First, we will have a brief look at the history of SQL injection attacks. Subsequently an explanation will be given in regard to how some of the most popular attacks work. Following this, we will attempt to show how the aforementioned attacks can be prevented. Finally, we move on to the core question of this essay: What role does SQL injection attacks play today? It is crucial that this question comes last, as the essay builds up in a way that makes the reader understand why the question is relevant. The last section will contain my thoughts and reflections on the question.

Before all of this, a definition of SQL injection attacks should be given. Microsoft defines SQL injection attacks, or SQLi for short, in this manner: 'SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution.' (Microsoft, Microsoft Docs, 2012). Essentially, this means that the philosophy of these attacks revolves around the abuse of data input from the client side of the system. This is not to confused with cross-site-scripting, which is an attack that stems from the opposite side of the system – the server side.

History of SQL injection attacks

As we know by now, an SQLi attack is where hackers input commands in queries that are supposed to receive strings. This enables the attacker to extract information from databases that is supposed to be private or confidential. This vulnerability was, to the best of my knowledge, first discussed in December 1998 in an article published on the hackers magazine Phrack (Foristal, 1998). Naturally, the problem was around before that, and someone even claims to have discovered SQLi before that, in 1995 (Plato, 2018).

At that time, most webpages were not using full Microsoft SQL databases, so the issue did not gain any substantial attention. Ironically, this vulnerability would soon become one of the most common to be abused by hackers. The Open Web Application Security Project, OWASP for short, is a worldwide non-profit organization with the purpose of mitigating software vulnerabilities. Today, SQLi consistently tops the charts on OWASP's 'Top 10 Application Security Risks' (OWASP, OWASP, 2017).

SQLi is indeed an old vulnerability, but that does not make it any less prevalent. It is far from a 'problem of the past', and the news headlines that consistently show up in relation to SQLi is a testament to that. Chances are, if you are an adult and American, you have probably heard of the Equifax data breach. This data breach, which took place last year, compromised the personal data of more than 145 million Americans. This included social security numbers, usernames, passwords, and credit card information. This was achieved through a vulnerability named CVE-2017-5638, affecting the Struts framework, which is used in a wide range of web applications. NIST (National Institute of Standards and Technology) has published details regarding this particular vulnerability. According to them, CVE-2017-5638 '…has incorrect exception handling and error-message generation during file-upload attempts, which allows remote attackers to execute arbitrary commands via a crafted Content-Type, Content-Disposition, or Content-Length HTTP header, as exploited in the wild in March 2017 …' (NIST, 2017). In other words, a remote code execution enabled this attack to take place. The interesting notion about the attack, is that Equifax was warned about it six months in advance, without taking precautions (Franceschi-Bichhierai, 2017). Fifteen years later, and SQLi is not taken as seriously as it should be. Hence becoming the topic of this essay.

The common types of SQL injection attacks

Unsanitized inputs

One of the most common methods of SQLi is where the attacker sends user input that does not get thoroughly sanitized for special characters that normally would be escaped. Unsanitized input could also mean an attack in which the input is not validated for what is expected from the relevant query. An example of this attack is shown in an episode of The Modern Rogue, a channel that showcases different types of hacks and vulnerabilities. In this particular episode, they showcase a prime example of bad input sanitization. The episode hosts visit a webpage (demo.testfire.net) with the purpose of testing different SQLi attacks. They enter a simple login form, which is expected to receive a string of username and password. The trick here is to add a tick (‘) and two dashes followed by a space(--) after the supposed username in the username field. What this does, is turn the following code and checks into a comment. This means that the password check that is supposed to follow, never happens. In other words, if you enter a generic username, lets say “Bob” and do the procedure mentioned above, it will look like this: Username: "Bob'-- ". Anything you enter in the password field is irrelevant, as it is now treated as a comment and not as code.

This is just a simple exploit where you bypass login authentication. The full potential of this attack is far more sinister. In fact, with this type of exploit, it is possible to run commands that return private information; usernames, passwords, addresses, credit card information etcetera. This is what we hear about in the news (e.g. above: Equifax data breach). The attack is so potent, that it can give a total compromise of the victim’s servers. The solution to it is actually very simple, but that will be discussed later in this essay. Ironically, despite the simple solution(s) to this exploit, it still happens frequently even in 2018, fifteen years after its discovery.

Error-based SQLi attacks

Error based SQLi attacks are attacks that rely on error messages given by the web application. Usually, in order for these attacks to work, the desired database should be connected to the web application. This is the first thing to confirm when doing an error-based attack. Let us assume a webpage with the following URL structure: '**www.uio.no/studier.php?id=3**'. One way of determining if the database is connected to the webpage, is to add a tick after the id number, which gives us something like this: "**www.uio.no/studier.php?id=3'**". What this tick does, is disturb the syntax of the query. If the webpage is connected to the database, an error message will be displayed. Now that we have verified that this page is connected to the database, we can start planning the attack.

There are different ways to structure and plan the attack, but they usually do all of the following in no particular order: Verifying the database version, extracting the table names, extracting the column names, and finally accessing the data in these columns. The nature of this type of attack, is that the information you need will be displayed in the error codes returned by the web application. Say you want to access the aforementioned URL, but instead of id=3, you want to browser the page with id=integer value of MySQL version. This page does not exist of course, but the error message you will receive, will give you what you need. Assuming all of this is done correctly, you should receive an error message telling you that a page where the id is 'version of MySQL' does not exist. Following this structure, you can infer all of the information you want to extract.

Inferential SQLi attacks

So far we have mentioned unsantized input exploitation, and error-based exploitation, which are attacks where the attacker uses the same connection to attack and gather information. These attacks are referred to as in-band SQLi, because you use the same connection to attack and extract. Inferential SQLi attacks are on the other side of the spectrum. In these types of attacks, no information is retrieved through the webpage application. The attacker can also not

see the application's response to his/her attack, hence the method being dubbed 'blind SQL injection'. As a result, inferential attacks consume more time, and are not as common as in-band SQLi.

The way these attacks work, is as follows. The attacker sends out multiple instructions to the web application and observes how the application reacts to it. There are generally two common ways to do this. OWASP has defined these methods in their webpage (OWASP, 2013). According to them, the two most used methods are content-based and time-based. Content-based inferential injection attacks occur when the perpetrator compares the content of the web application in relation to his/her injected queries. If the content looks different depending on the query's boolean value, then that means it is susceptible to the attack. Let us say the attacker injects a query that always returns TRUE, and then does the same with a query that always returns FALSE. If there is a difference in the content based on the query's boolean value, that means it can be attacked through a content-based attack.

Time-based attacks follow the same general concept of deriving the result from the web applications behaviour in relation to the sent queries. The difference here is that it is not based on the content of the application, but rather the time it takes for the application to respond to the sent query. The difference in response time is what determines whether the returned result of the query is TRUE or FALSE. Generally speaking, a FALSE value is returned almost immediately, and a TRUE value is returned if it takes any longer. Once this pattern has been established, the attacker only has to enumerate each letter of the targeted database. This can be done with simple if statements (if the first letter of the database's name is "A", wait 10 seconds etc.). Again this is a time consuming attack, but there are tools out there that help with this enumeration process.

Defending against SQLi

SQL injections have been around for over 20 years, but despite this they still dominate as the most widespread cyberattack. You would not be wrong to think that after all these years, we

would have learned our lesson and made corrections – and we have. The problem is that these corrections are neglected due to lack of information or plain laziness. In this section of the essay we will investigate the preventive and corrective actions to take in regards of SQLi.

There is no silver bullet here. There is no one single solution that will mitigate the problem of SQLi. There are however procedures to follow that will ensure a decreased risk of injection attacks. Let us have look.

Prepared statements

The first thing you should do as a database designer, is to avoid dynamic SQL. What does this mean? It means preventing user input to be directly put into SQL statements, which is how these attacks work in the first place. Instead, you should use prepared statements with parameterized queries. The basic idea of prepared statements with parameterized queries, is that you first write all of your SQL code and then pass in the parameters afterwards. Let us assume you want to insert some values into a table named Grades. Normally, it would look something like this: 'INSERT INTO Grades VALUES (userinput, userinput, userinput)'. With prepared statements, you have placeholders instead of user input. These placeholders look different based on your database environment, but in PHP the symbol for placeholders is '?'. The result of a parameterized query would then look like this: 'INSERT INTO Grades VALUES (?,?,?)'.

This separates the user input (data) from the SQL statement (code), and makes it harder for attackers to exploit. Not only that, but it is more intuitive and easier to understand because the user input is separated from the SQL statements. Another benefit of this method is that it generally improves performance, by reducing network traffic to the database. Due to it using templates for the queries, the preparation (parsing, compiling, optimization) is done only once, meaning that the application does not have to repeat the preparation for each time the user inputs data. There are however instances where prepared statements harm performance. In such cases, it is recommended to either strongly validate all data, or escaping all user input using an escaping routine specific to your database (OWASP, 2018).

Stored procedures

Stored procedures work similarly to prepared statements, with one exception. They are similar in the way that they both revolve around using placeholders for the user input, but the difference is that the stored procedures are stored and defined in the database itself, contrary to prepared statements. This makes it slightly more susceptible to injections, but when implemented correctly should work just as well as prepared statements. Both of them provide an extra layer of protection between the user interface and the database. It is a matter of choosing what works best for you in terms of cost, effort, and performance.

There is one thing in particular to consider when it comes to choosing stored procedures as a primary defence for a database. When using stored procedures, you are essentially taking advantage of ownership chaining to return data to the user. For stored procedures to work, you need to grant the EXECUTE role to the procedures. This is how they are set up. A stored procedure can call other stored procedures, or access multiple tables. If all objects in the chain of execution have the same owner, then SQL only checks the EXECUTE permission for the caller, not the caller's permission on other objects (Microsoft, 2017). This means that only stored procedures should be granted the EXECUTE permission, and all underlying tables should deny all permissions and remove existing ones.

The problem with this design, is that the EXECUTE role is not available by default. In some instances, web applications had to run under the db_owner role for stored procedures to work. Naturally this caused a problem. If the application was to be breached, the attacker would have db_owner rights (all permissions and full access), meaning that the consequences of a breach would be critical. However, if you follow the guidelines given by Microsoft, this should not be a problem (Microsoft, 2017).

Escaping input

If for whatever reason you are not able to either use prepared statements or stored procedures, the next best alternative you have is escaping user input. As shown earlier, most injection attacks revolve around inputting strings that disturb the intended intention of a query. Anything from ticks to dashes, and other special characters are used for this purpose. In other words, escaping input is a way of sanitizing once unsanitized input. The way this works, is that there are functions in various database environments that are intended to check for these irregularities, and remove them. This is as good of a preventive method as the other ones mentioned, because not all SQL injections revolve around the abuse of special character symbols.

Principle of least privilege

No matter which approach you take to secure your database from attacks, the principle of least privilege is a philosophy you should follow to further increase your protection against SQL injections. Essentially, the philosophy of this principle, is that every subject only has the roles that enables it to do its intended function – no more, no less. When looking at this from the user end of the spectrum, the term ‘least user access’ (LUA) becomes relevant. Instead of looking at what roles a user should not have, you start from zero roles and add roles until you have a functioning model. This is considered a secondary layer of protection, unlike the primary defence alternative mentioned above.

Data encryption

Again, this is also a secondary layer of protection, but should be used regardless of your approach. This one is quite simply to incorporate in your defence strategy. Let us assume gains access to your database. Leaving private information unencrypted is quite careless, but let us assume you did not encrypt your data. This gives the attacker a free way to your information, making his job easier. If you would have encrypted your data, it would not guarantee its protection, but would make it harder for the hacker to access it. Encryption is

encouraged, because sometimes the hacker is not able to decrypt the data. But even if he manages to decrypt the data, it buys time for you to take preventive actions (identify the culprits, change passwords, freeze accounts etc.).

What role does SQLi attacks play in 2018?

As we have seen, SQLi attacks are possible to prevent if you follow a good coding discipline. But that is also its downfall. The common stereotype that companies do not acknowledge a security problem until it becomes a problem is sadly true, based on the news articles that follow after a major attack. Take for instance the Equifax data breach, or the Ticketmaster data breach (Priday, 2018). Most of these companies are warned beforehand, but do not do anything until it is too late. You might ask yourselves why.

There are many things to consider when taking preventive actions against discovered vulnerabilities. The one that is usually considered first is the cost vs consequence relation. Is the fix going to be worth the damage it mitigates? Most of the time the answer is yes. Another thing to take into consideration is the fact that the problem is due to bad code practice. A complex and big database system might have been built with bad input sanitization and little to none regards to injection attacks. This arises the question of effort. Lazy CEOs will sometimes try to stall and avoid confrontation with the security consultants.

Sometimes the developers responsible for fixing these issues are also pressured into finishing new projects before taking preventive action against old ones. This happens because the manager prioritizes the short-term profit of publishing new project vs the long-term profit of securing old projects. In the end, management is often the cause of the attacks we hear about in the news, not the developers. Even so, both developers and managers have a long way to go before we start seeing less of these attacks happening.

The bottom line in my opinion is to take the problem seriously. Whether you are just now starting to build your database, or if you already have it built – commit to good code practice

and use your online sources to build a secure database environment. Not only for your own sake, but for your users as well. CEOs and other employees with administrative backgrounds and weak IT knowledge should take introductory courses to grasp the importance of these attacks. It is 2018 after all, society is quite digitalized already, but it will become even more so in the future. There is no excuse for people in these positions to not know about the implications of data breaches.

References

Andrews, T. M. (2018). *The Washington Post*. Retrieved from https://www.washingtonpost.com/news/arts-and-entertainment/wp/2018/06/05/ticketfly-has-been-hacked-heres-what-you-need-to-know/?noredirect=on&utm_term=.bfc633074038

Foristal, J. (1998, December 25). *Phrack*. Retrieved from Phrack Magazine: http://phrack.org/issues/54/8.html#article

Franceschi-Bichhierai, L. (2017, October). *Vice*. Retrieved from Equifax Was Warned - Motherboard: https://motherboard.vice.com/en_us/article/ne3bv7/equifax-breach-social-security-numbers-researcher-warning

Microsoft. (2012, April 10). *Microsoft Docs*. Retrieved from SQL Injection | Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)

Microsoft. (2017). Retrieved from https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/managing-permissions-with-stored-procedures-in-sql-server

NIST. (2017). *NVD*. Retrieved from NVD - CVE-2017-5638: https://nvd.nist.gov/vuln/detail/CVE-2017-5638

OWASP. (2013). Retrieved from Blind SQL Injection: https://www.owasp.org/index.php/Blind_SQL_Injection

OWASP. (2017). *OWASP*. Retrieved from OWASP: https://www.owasp.org/index.php/Top_10-2017_Top_10

OWASP. (2018). Retrieved from https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Plato, A. (2018, 11 04). *LinkedIn*. Retrieved from LinkedIn: https://www.linkedin.com/in/andrewplato

Priday, R. (2018). *Wired*. Retrieved from https://www.wired.co.uk/article/ticketmaster-data-breach-monzo-inbenta