**Slide 1**

Hello, my name is Marvin Newlin and today I am going to be talking to you about SQL Injection Attacks. As a disclaimer, “The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Air Force, the Department of Defense, or the U.S. Government.”

**Slide 2**

In overview, today I will be showing an illustration of a simple SQL injection attack and discussing who is vulnerable to SQL Injection, How SQL Injection works, and how to prevent SQL Injection attacks from happening to you.

**Slide 3**

Imagine that we have a web application with that takes in a username and password and then passes that into a back-end database.

**Slide 4**

In most cases with a normal friendly user, they will enter their username and password as intended and go on about their day.

**Slide 5 (REDO)**

But, if we throw our grey hacker hats on for a minute we can imagine a scenario in which we can attack this application. Suppose that our database contains a table called “authtable” and the SQL query that our application provides takes in the user’s input without checking it at all.

Suppose the hacker enters “x; drop authtable --”. I’ll explain the semantics of an SQL query in just a bit but here is how this specific one works.

The semicolon after the X splits this one “query” into two separate queries and the double dashes cause anything after it to be read as a comment, effectively disabling anything after them.

The application will send this string straight into the database and will completely delete the “authtable” table from the database, and now we’ve been hacked. This is obviously a big problem and something that should never happen.

This is in a nutshell, how SQL Injection attacks occur.

**Slide 6 (REDO)**

So who is vulnerable to SQL Injection?? Basically, any application that sends user input into a database can be vulnerable to SQL injection, particularly any application that is built in-house.

SQL Injection Attacks can be incredibly damaging and are listed in the top 10 vulnerability list put out by the Open Web Application Security Project.

The damaging thing about SQL injection vulnerabilities is that they generally go unnoticed until they are exploited by a hacker. SQL Injection is concerning because a SQL injection attack can lead to compromised or stolen data and potentially private data, so companies have to be on the lookout for this vulnerability. In addition to having data stolen, companies can be held legally liable for stolen private data if they do not properly secure their information. This makes protecting against SQL Injection Attacks imperative.

**Slide 7 (REDO)**

SQL stands for Structured Query Language and is a language developed for interacting with databases. The main element of a database is a table. Every table has a table name, each table is composed of columns called fields, and rows called records. A record is a collection of fields, and a table is a list of records.

The basic makeup of an SQL query is three key words, SELECT, FROM, and WHERE. In our example table, to extract a specific record we would say, SELECT fieldName1 FROM tablename WHERE fieldname1 = ‘$VALUE’;. The WHERE key word specifies a Boolean condition that must be met to return a record or field from the SQL query. Often times in a SQL Injection Attack, the malicious query passed in will contain a tautology, a condition that is always true, in the WHERE field in order to automatically return the contents of the table.

SQL Queries end with a semi-colon and the $ sign indicates a variable value inserted by the programming language creating the SQL query. The process of building an SQL query inside of an application that allows for variable inputs is what gets exploited by an injection attack

**Slide 8 (REDO)**

SQL Injection at its root, works when user input is sent to the database without filtering or checking it first. The actual injection happens when the malicious user enters data that is itself an SQL query into a field that isn’t expecting it. In the code, this input is joined to an SQL query by concatenation and the unfiltered query is sent into the DB. This causes havoc in the DB and can cause the application to crash or worse allow the attacker to steal data from the application.

The example code on the right shows an example of an SQL query that is vulnerable to an injection attack. Here we see the creation of the SQL query string variable, SQLString. It selects the ccnum field from the cust table where the id field is equal to some passed in value indicated by the concatenation with the ID variable. This is the bad concatenation of user input that allows for SQL injection attacks to occur.

That SQLString variable is then used to create an SQL Command and then the execution of that command is supposed to return the string that represents a credit card number. However, if a SQL Injection Attack has occurred and the SQL String is invalidated in such a way as illustrated in the earlier example, then the execution command will instead successfully execute the SQL Injection Attack and the attacker will now have access to the table that contains all the credit card numbers!

**Slide 9 (REDO)**

So, how to we keep ourselves safe from an SQL Injection Attack?

First, always always always check user input. Never send user input directly to the database without checking the contents first! Validation is a key aspect in preventing SQL attacks.

Second, in code, always utilize prepared statements rather than joining user input directly to the query.

Prepared Statements allow the developer to completely structure the entire SQL query beforehand and then plug in any user input with the ‘@’ tag. Prepared Statements coupled with input validation can go a very long way in preventing SQL Injection Attacks.

The example code on the right shows how to properly prepare an SQL Prepared Statement that takes in a user provided parameter called ‘ID’. First, the user input is validated by ensuring that it is a valid ID number only containing numbers and is then plugged into the Prepared Statement using the ‘@ID’ tag.

The SQL query is now not vulnerable to an SQL Injection Attack in two ways. First, the validation of input prevents characters that shouldn’t be in the query from being contained in it. Second, adding as a parameter allows the programming language to wrap the validated input with all of the other SQL requirements like a semicolon to end the statement rather than the programmer doing it manually.

With these safeguards, any attempt to inject SQL commands into the query will be caught and an exception will be thrown, preventing the injection attack from executing.

**Slide 10 (REDO)**

So, to wrap it up I have demonstrated a simple SQL Injection attack, I have talked about who is vulnerable to an SQL injection attack and the ramifications it can have, discussed how SQL injection Attacks work, and how to protect against SQL injection Attacks. Thank you for your time

**Slide 11**

These are the references I made throughout the presentation. Thank you again!

End