SQL Exploitation Notes

Finally, we get to talk about security. Just like with many other topics, we leave it to our colleagues in the other departments to worry about the science and the algorithms, and we look at we can do with the tools that they provide us. In this case, we are going to look at what the *hackers* can do with the tools and come up with ways to defeat their efforts.

Let's get started by going to the first slide. It's very easy to find accounts in the news how about systems being hacked and data being revealed. Hackers getting in and finding out someone's password or other personal information is all too common. And, as a database person, I find this very annoying because most of the time the hacker's success is due to the laziness of the database developer. It literally takes a few seconds to the write the code to hash a password. Sure, with enough horsepower pretty much any password hashing scheme can be hacked, but that's not usually what's going on here. If you read the accounts, you'll see that in most cases the passwords were stored in clear text.

We're going to see, very shortly, that there are also some very simple ways that a hacker can you use our SQL against us to either gain entry into the system or find out valuable information. And much like the effort required to hash the password stored in the database, you'll see that it is relatively easy for us to thwart their attempts. We do not have to live in a world where hackers can break into our database systems and cause major havoc or have everyone panic. It's just too easy to prevent.

So, let's go to the next slide and get started. For the sake of discussion, we're going to break SQL exploitation into two categories, or vectors as the security folks like to call them. The first one is it malformed queries. The essence of a malformed query is an attempt to cause the system blow up. If the hacker can cause our query to have a syntax error or some other problem, the database engine will reject it with an error message that can contain some very useful information for the hacker. This is important, and a good test question, **a malformed query is one that will cause the system to throw an error and reveal information.**

The other type of exploitation is referred to as an SQL injection. In this technique, the hacker tries to modify the meaning of the SQL statement. They do not want it to cause an error, they want it to run. They're just going to try to get it to return some additional information beyond what was intended. So, the key here is that **SQL injection is used to reveal additional data beyond that which was intended**.

Okay, let's go to the next slide and dive a little deeper into malformed inquiries. Sometimes we can cause malformed queries ourselves by simply having poor syntax in our SQL query! You may wonder how that could possibly happen when, being a good developer, you test the query before you put it into your code. Well, you're correct that you probably would do that for a well-defined query. But sometimes we use a technique called dynamic SQL where we build the query on-the-fly. Perhaps we were waiting for input from the user to determine exactly how we would construct the query. Once we get the input, we do some simple string concatenation.

But what if we messed up in our string building and while concatenating a couple of pieces, we lost a space with the result being some weird combination of keywords? That would generate an error when the engine parsed it.

That's is, what about the hacker? Perhaps we were asking the user to enter something that we are going to filter on, that is, we're going to put it into our WHERE clause. If the hacker enters some funky characters or some odd expression, and we run the query with that value, it could cause things to blow up. So how do we guard against this? Let's go to the next slide.

There are two approaches we can take: a proactive approach, and a reactive approach. In the proactive approach, we take a look at the SQL statement before we try to run it. We might limit the type of statements that can be run against the database. For example, we might only allow a SELECT statement so that no damage could be done to the database. Of course, that's not always practical nor is it sufficient. Even with a SELECT statement, the hacker can still find out information we don't want him to.

Another proactive method is sanitization. This means we're going to look at the SQL statement and be sure that there are no funky characters in there and that it is syntactically correct. We are effectively parsing the SQL statement just as the database engine would do. The difference is that instead of blowing up the way the database engine well and throwing error, we are going to trap the problem and just reject the query. There are actually some libraries that can help with this. That's good because it is a difficult task.

Now while these proactive methods are good, they do not guarantee that they'll catch everything. As a result, we will always employ reactive methods. Let me restate that for emphasis, **regardless of whether we use any proactive methods we will always use reactive methods.** And what is the reactive method we're going to use? We are going to catch our exceptions! It is really that simple. As long as I run all my SQL queries in a try-catch block, I do not have to worry about hackers using malformed queries. If they are successful in their attempt get the SQL to fail and the database engine throws an error, that's okay, I will catch it and they will see nothing as a result.

This goes to my comment earlier about developers being lazy. All we have to do is catch our exceptions, a standard best practice for coding, and we will totally shut down this vector for hackers.

Let's go to the next slide that discusses SQL injection. Remember that unlike malformed queries, the hacker wants the SQL injection effort to run. They just want to change the meaning of the statement. One of the techniques a hacker will use is a UNION. You may or may not remember what a UNION does with the SQL statement. We use a UNION in SQL when we want to run two or more queries and combine the results into a single resultset. So, if I know that the result of a query is going to yield a result set with, let's say, three string datatype columns, then I can UNION that with any other query against any other table as long as I am extracting three string datatype columns. As an example, the original query may be trying to select the name

and address of clients, but a hacker could UNION that with a query that selects user ID and password from the Users table.

How would the hacker know about a user Id field and a password field in the Users table? Unlike the movies, hackers don't just type a few key phrases in order to gain entry into a system. It's a process that takes time and patience. They try something that gives them a little bit of info. Then, they use that info to try something else that gives them a little bit more, and on and on. So, perhaps they first tried some malformed queries and got back information about your table structures. Don't believe me? Take a close look at the error message the next time you have bad syntax in one of your queries!

But even with that, you're probably sitting there thinking, "Okay, but how does the hacker get to the actual SQL statement? That is in my code to which is on the server." Well, what is the entry point into your application? It's the data entry form up in the presentation layer. And, once again, it's those evil text boxes that are the culprit.

When I ask the user for some data, how do I control what they actually type in the text box? If they have a good sense of the query I'm about to run, then they can actually enter SQL code in the text box have that get inserted into my query state. That's why it's called SQL injection, they're injecting their own SQL code into my SQL statement.

Let's look at the next slide for a humorous example of this. This XKCD comic shows what happens in this fictitious situation where the user types in some SQL code. First, in black, I've listed it out the actual SQL statement that the developer wants to run. But now consider the value that is entered for the variable, stuName. You see that in green in the second line. When we do the substitution, we get to the third line.

Notice a couple of things here. First, we have created two separate SQL statements the first one does a SELECT and ends with the ";" The second statement does a DROP TABLES and also ends with a ":" Finally, the "--" denotes that the rest of the text is a comment. Look at this carefully and understand what the hacker has done. He (or she) anticipated (or knew from previous poking) the syntax of the SQL statement and cleverly constructed a string that would change the intent of the query without causing an error. The use of a semi-colon to permit multiple statements, and the use of the comment symbol ("--") are key factors in this approach.

Let's go to the next slide for yet another example. We start with a classic query for authenticating a user: "SELECT username, password, name FROM users WHERE userName= some value from user and password equals some value from user". In the next part of the slide, we see the values that are entered. For userName they entered any value, but for password they entered a phrase that contained Boolean logic. At the bottom of the slide, you can see the result of the substitution. Look at the WHERE clause and think back to the rules of Boolean logic. We have a compound statement with an AND and an OR. Recall the rules of Boolean logic, first we evaluate the phrases, then we process the ANDs, and then we process the ORs. Evaluating the phrases, we end up with:

false AND false OR true

Next we process the AND and reduce the overall expression to

false OR true

This, of course, evaluates to True. As a result, the query runs and returns the username, password, and name for every user in the table!

All of this is made possible because the developer is using dynamic SQL. That is, he's building the SQL statements on-the-fly using input from user. Go to next slide and you will see the simple way to prevent this: don't use dynamic SQL, use prepared statements. That's it!

Once again, we have a relatively straightforward and simple process of using prepared statements to completely thwart the hacker from using this vector. Remember our discussions of prepared statements. We said that one of the important characteristics of a prepared statement is that the database engine will already have a query plan figured out for the specified tables and fields. It is not possible to change the meaning of the SQL statement because the query plan has already been set.

Let's go to the next slide to wrap up SQL exploitation. Data cleansing and exception handling will effectively guard against malformed queries. Data cleansing is a nice idea and is often used in large systems, but the required thoroughness, or the required money for a good library, often makes it not practical. As a result, many systems only employ exception handling as a guard against malformed queries and that is perfectly fine.

Prepared statements and their cousins, stored procedures, are effective guards against SQL injection. In both cases, it is not possible to change the meaning of the statement (or procedure) once the query plan has been created.

You may have gathered from my description of developers as being lazy if they don't implement these guards, that I take this very seriously. Keep that in mind. I do not expect to see any queries in your project that are not prepared statements or that are not executed inside a try-catch block. There may be exceptions here and there, but they should be rare.

Unfortunately, SQL exploitation is not the only way a hacker can't get into our system. Go to the next slide for some other vectors. The examples listed here are mostly things that are beyond our control. The DBMS is an application itself and it will have bugs just like any other application. Hackers will look for those bugs and try to exploit them. Depending on the vendor, the DBMS may also be vulnerable 0-day exploits or bugs introduced with maintenance releases. Fortunately for us developers, these are the things that the DBA normally worries about. But, if you're in a small shop, you may be both the database developer and the DBA.

One item here that is under our control is the language workarounds. Be very careful about using deprecated features or old, out of date versions. There could be security holes there that a hacker would take advantage of. If you're going to be a developer in language X, make it your business to learn everything you possibly can about that language. Don't just learn the cool features the language offers, keep up on news releases and reports of vulnerabilities and enhancements and be sure you understand their ramifications. That's what being a professional is all about—thoroughness of understanding.

And finally, the next slide wraps things up. Security is not something you can check off and say, "I'm done." It's an ongoing game that we play with the hackers. They poke and prod and find holes in our approaches. They take advantage of them and then we fix our code. Always try to write code ask defensively as possible. In your code, never ever trust input from the user. When you're writing your exceptions, only pass back minimal information. Make all your variables and methods private unless there's a reason to make them public. These are just a few things that can help you write your code defensively and help protect your application from hackers.

That's it. Pretty short and pretty simple; please be sure you utilize these protection methods.