My name is Warren Moynihan and I am a member of the Security Systems Ethical Hacking team. In this video I will be talking about one of the more common weaknesses found in insecure code: Injection.

This type of vulnerability is currently ranked #1 on the OWASP Top 10 chart which means that it is responsible for a large portion of public disclosures and security breaches.

So what is an injection vulnerability? Well, there are actually several types. Some of the most common types include SQL injection, code injection and LDAP injection. With the different types of injection, the attacker will construct his attack in a different way.

For example if an attacker wants to perform SQL inject they could write a SQL query to replace or concatenate an existing query used by the application, using characters like ‘ or -- to bypass the existing query’s logic.

For an OS commanding injection they may include a shell command within their injection using a ; or | character to include their commands. Each attack could be tailored to the attacker’s goal, the target server’s infrastructure, and which inputs can bypass the application’s existing logic.

All injection attacks involve allowing untrusted or manipulated requests, commands, or queries to be executed by a web application.
Injection vulnerabilities also present some of the most significant risks when effectively exploited. Some of these risks include:
* Data loss or corruption
* Data theft
* Unauthorized access
* Denial of service
* Complete host system takeover

The consequences of any of these risks can seriously impact your ability to conduct business, your customers, and your organization’s reputation.
So how can you protect yourself from these types of vulnerabilities? Many strategies can be used.
You could use a vetted library or framework that does not allow this weakness to occur, or provides constructs that make it easier to avoid.
* Use a parameterized API
* Ensure you’re running the application with minimum privileges
* Escape all special characters used by any interpreter
* And also input validation or sanitization

Assume all input is malicious. Use an "accept known good" input validation strategy: a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on blacklisting malicious or malformed inputs. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

I will be demonstrating some basic SQL injection exploits. Open a browser and navigate to demo.testfire.net. This is a sample banking website. In order to find a SQL Injection vulnerability, attackers may inject a variety of characters or different inputs into the web application. Characters that have a special meaning in a SQl query. A great example of this is single quotes.

Here I’m going enter a simple username. It has to be a valid username. And I’m going to put in a single quote in the password field. Now we’ll attempt to log in. You can see that although I wasn’t able to log in successfully, the target application threw a very detailed error message and showing a variety of information about the SQL query it tried to run in the backend. Using this information I can choose to construct a more elaborate attack that will be more useful.

Here you can see it’s passing a username and a password with an AND operator. Let’s see if I can do a different login operator and try to bypass the logic of the application.

So we’ll start the injection off in much the same way with a single quote, add an OR operator in this case and let’s add a condition that’s going to be “true”. 1 equals 1, and let’s comment out the SQL line, to make a more well-formed statement.

Let's inject that into the password field. You can see now we've successfully logged into the application and quickly bypassed the authentication part of the application using a SQL injection. As an added bonus, you can see that I'm actually logged in as an administrator.

Now let's try a more elaborate example. Let's log in as J. Smith. The password is 'demo1234'. And let's try a common exploit to glean additional information from the database that we should not have access to.

Let's go to Recent Transactions. And like before, let's put in a single quote into the 'After' field. Again, you can see the request wasn't successful, but I can see that a SQL query was in fact run, and I can see some basic information about that query.

Analyzing these results, I can construct a more elaborate injection. In this case, I'm using a 'union' statement which is going to try to extract the username and password from the database. I'll place that in the same place as the last injection, and there you have it. I've used SQL injection to gain the usernames and passwords of everybody in this web application.

AppScan's test policy includes a wide variety of injection types and variants, including those we mentioned earlier. You can use the search dialogue in the test policy window to search for specific tests or variants you may want to test, and then check them to enable or disable them as desired.

Here's how to configure AppScan to automatically detect a SQL injection vulnerability. Open up AppScan, click Configuration and enter our Starting URL: demo.testfire.net. It's important to always record the login and enable in-session detection when you're running scans with AppScan.

Log in with jsmith, demo1234. Record the login, and always enable in-session detection. You can see that the green check mark indicates it was able to successfully validate a login.

Let's go to Test Policy. In the interest of saving time, we're going to use an isolated test policy. Most of the injection vulnerabilities including SQL injection are included in the Default application test policy. Let's go ahead and turn everything off for a moment here. In the type defined field here I'll enter Injection. Okay, and as you can see there is a number of injection vulnerabilities, many of which we discusses earlier available in AppScan.

In this example, we're focusing on the SQL Injection vulnerability, so we'll enable that one test, or a series of tests I should say.
And at this point we could run a full scan. Again in the interest of saving time, let's record a manual explore of just the content we're interested in. So we found a vulnerability on the login page and under Recent Transactions, let's just go in here and edit transactions.

AppScan's analyzing the explored data, and we'll select Test Only. We can see AppScan has already identified some injection vulnerabilities, three of them on the Transaction page, in the 'before' and 'after' field.

Oh, there's the one on the login page, in both the username and password fields.

The scan is still running, so maybe we'll find other vulnerabilities.

And there you have it; we found a couple of SQL injection vulnerabilities, both in the application manually and also demonstrated how AppScan can help you identify them.

This concludes our coverage of the 1st OWASP Top 10 category. I hope you found this information useful. I wish you best of luck in writing and maintaining secure software!