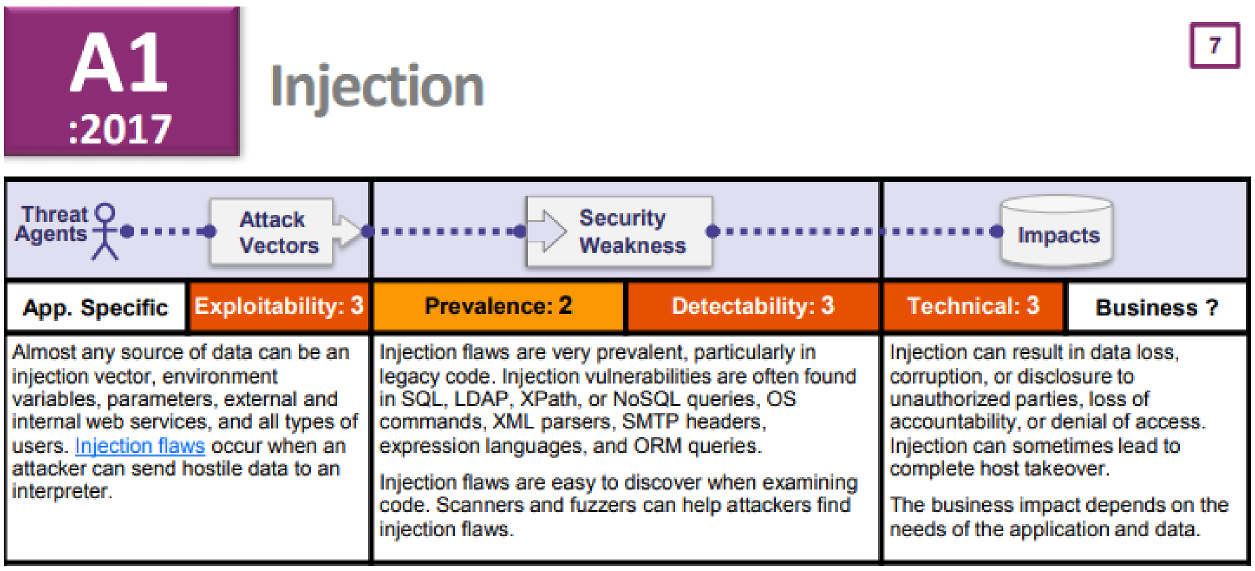
Querying Your Database While Avoiding SQL Injections

In this lesson, we'll look at the dangers of an SQL injection attack and how they can be avoided.

We'll cover the following

Introduction

Right off the bat, you're probably thinking, "*I've heard about injections,*" and that's probably because they were the #1 vulnerability in the "2017 OWASP Top 10: The Ten Most Critical Web Application Security Risks".



A1
:2017
Injection
7

Threat Agents	Attack Vectors	Security Weakness		Impacts	
App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Overview of injection vulnerabilities as presented in the 2017 OWASP Top 10

But guess what, injections made the #1 spot in the 2010 and 2013 version of the list. There's a strong chance you might be familiar with any type of injection risk. The only thing you need to remember when fighting an injection attack is to never trust the client. If you receive data from a client, make sure it's validated, filtered and innocuous, then pass it to your database.

An example

A typical example of an injection vulnerability is the following SQL query:

```
SELECT * FROM db.users WHERE name = "$name"
```

Suppose `$name` comes from an external input, like the URL `https://example.com/users/search?name=LeBron`. An attacker can then craft a specific value for the variable that will significantly alter the SQL query being executed. For example, the URL `https://example.com/users/search?name=anyone%22%3B%20TRUNCATE%20TABLE%20users%3B%20--` would result in this query being executed.

```
SELECT * FROM db.users WHERE name = "anyone"; TRUNCATE TABLE users; --"
```

This query would return the correct search result, but also destroy the users' table with catastrophic consequences.

Most frameworks and libraries provide you with the tools needed to sanitize data before feeding it to a database. The simplest solution is to use prepared statements, a mechanism offered by most databases that prevents SQL injections altogether.

> ### ℹ Prepared statements: Behind the scenes
>
> Wondering how prepared statements work? They're very straightforward, but often misunderstood. The typical API of a prepared statement looks like this:
>
> ```
> query = `SELECT * FROM users WHERE id = ?`
> db.execute(query, id)
> ```
>
> As you can see, the base query is separated from the external variables that need to be embedded in the query. What most database drivers will eventually do is send the query to the database so that it can prepare an execution plan for the query itself. That execution plan can also be reused for the same query using different parameters, so prepared statements have performance benefits as well. Separately, the driver will also send the parameters to be used in the query.
>
> At that point the database will sanitize the parameters and execute the query with the

> At that point the database will sanitize them, and execute the query with the sanitized parameters.
>
> There are two key takeaways in this process:
>
> - The query and parameters are never joined before being sent to the database, as it’s the database itself that performs this operation
> - You delegate sanitization to a built-in database mechanism, and that is likely to be more effective than any sanitization mechanism we could have come up with by ourselves

---

In the next lesson, we’ll look at a few dependencies with known vulnerabilities.