Chapter 15
Introduction to Injection Attacks

Revision 3.1, December 2025.

Objectives

- Understand the basic principle behind injection attacks.

15.1 Background and Motivation

There are several types of injection attacks, depending on what part of a system is being attacked. However, they all follow a common pattern. We start with some component of the program that accepts commands in text form. This component might be a command shell, the SQL interpreter, XML parser, or even the interpreters for a language such as Python or JavaScript. Injection attacks are possible whenever four criteria are satisfied:

1. The program is using some form of command interpreter.
2. The commands being sent to the interpreter are constructed by the program while it is executing. In other words, the commands are not constant strings.
3. At least part of the data being used to construct the strings comes from user input. In other words, at least part of the input comes from the attack surface.
4. The program does not correctly prevent the user input from changing the way that the programmer intended the command to be interpreted.

Items 1 through 3 above may be critical to how you construct your program. For example, you may have a database of valid users and passwords, so need to use SQL queries to check that database (item 1). The SQL query will need to be different for each user who tries to log in (item 2). And the user name and password will likely come from what is typed into a web form and sent to the server (item 3).

Flaws in your programs can enable successful injection attacks. Such flaws are based on user input confusing your program into allowing commands to execute that you did not intend (item 4). Typically, such confusions come from improper escaping of metacharacters (punctuation) or improper quoting. As a result, we end up with text that was intended to be string data that becomes part of the command itself, resulting in a very different command that does something different than intended.

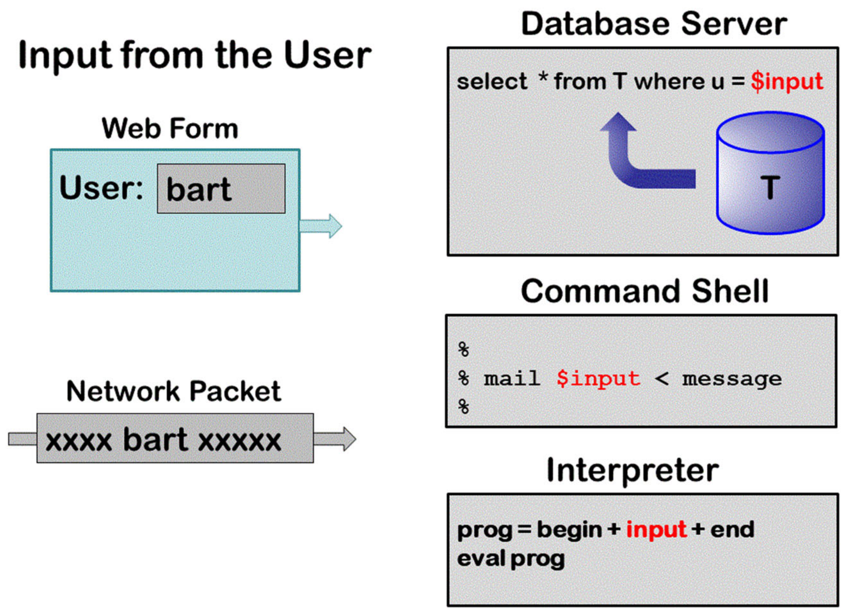

Figure 1: Different User Input Sources and Three Types of Injections

In the picture below, we show some simple examples of how user input might be used in a few different scenarios. As we mentioned above, the places where the user input can reach the program are collectively called the attack surface. We call the places in the program where this user input can affect the program's behavior (shown as **input** in picture) the *impact surface*.

Note that in each case the input from the user is being used to construct a command string to a different type of command interpreter.

15.2 A Human-Language Example

We will start with an informative but somewhat silly example based on the famous and eminently enjoyable Mad Libs[1] children's word game. The idea of the game is that you start with a list of categories for words. The children first select a word to fit each category. The words are then used to fill in the blanks in a story. Since you cannot see the story before you start choosing the words, the result is usually absurd and quite funny

So, we will start with the list of words on the lefthand side of Figure 2. You can make your own selection to play along. Make sure not to peek at the story on the right while you are selecting your words.

[1] https://madlibs.com/

Ask for a list of words:	Then use them in story:
❶ A vehicle: ❷ An outdoor location: ❸ A food: ❹ Another food: ❺ A sport: ❻ A relaxing activity:	It was a lovely day for a picnic, so we packed the __❶__ and headed to the __❷__. The basket was loaded full of delicious __❸__ and __❹__. We spread out our blanket and first decided to play __❺__ and then __❻__ for a while.

Figure 2: A Game with Word Categories to Choose and Accompanying Story

Now, we will pick our choice of words on the left in Figure 3, and see how they are filled into story on the right. As you can see, it ends up with quite a silly story.

❶ A vehicle: **chariot** ❷ An outdoor location: **rooftop** ❸ A food: **scramble eggs** ❹ Another food: **pickles** ❺ A sport: **javelin throwing** ❻ A relaxing activity: **stand on our heads**	It was a lovely day for a picnic, so we packed the **chariot** and headed to the **rooftop**. The basket was loaded full of delicious **scramble eggs** and **pickles**. We spread out our blanket and first decided to play **javelin throwing** and then **stand on our heads** for a while.

Figure 3: Word Game with Word Choices Made

But things can take a darker turn. Now, for item ❻ in Figure 4, after the selected word, we added a period and then a whole additional sentence. The result of reading this sentence is that the reader of story instructs the other children to go out and rob a bank. Because they are all obedient children, they promptly do this robbery, steering them into a life of crime. This is all tongue-in-cheek but is representative of a computer that follows our literal instructions.

❶ A vehicle: **chariot** ❷ An outdoor location: **rooftop** ❸ A food: **scramble eggs** ❹ Another food: **pickles** ❺ A sport: **javelin throwing** ❻ A relaxing activity: **relax. Hey kids, now go to the bank and rob it, while we stay here**	It was a lovely day for a picnic, so we packed the **chariot** and headed to the **rooftop**. The basket was loaded full of delicious **scramble eggs** and **pickles**. We spread out our blanket and first decided to play **javelin throwing** and then **relax. Hey kids, now go to the bank and rob it, while we stay here** for a while.

Figure 4: Word Game with a Malicious Word Choice Added

So, what went wrong here? The fundamental problem is that the creator of the game made some implicit assumptions about the inputs to the story, that is how the players would provide words. They trusted the players to somehow be reasonable in their word choices. And, just as important, they did not provide any checks to see if the players were being reasonable in their choices.

As a result, in the place where we were expecting a single word, we ended up with a whole additional sentence. That extra sentence was possible because the malicious player inserted a punctuation mark after the first word, allowing the following sentence. And there was no checking or enforcement of the rules, so the punctuation mark was allowed, along with a second sentence. Third, the listeners of the story followed the instructions literally, to their own harm.

15.3 Injection Attacks in Software

We can now take these ideas and see how they apply to an attack on software. We are interested in programs that constructs command strings that will then be used as input to a library or program that then interprets (in other words, executes) the command.

An injection attack can occur when user input is used to construct a command string, which is then executed. In other words, if there is a path from the attack surface to the impact points at which a command is executed, and if the input is not sufficiently checked or sanitized, then there is risk of an injection.

When we look at Figure 1, we see two examples of points on the attack surface, the contents of a web form and contents of a network packet arriving at a server. We also see three examples of impact points, where the user data is used to construct a command to a SQL interpreter, command shell, and language interpreter. We can also have injection impact points at XML interpreters, web browsers (cross site scripting attacks in HTML, described in Chapter 21), and for file access (directory traversal attacks, described in Chapter 12).

A common way that a malicious user can fool a command interpreter into doing the wrong thing is by including metadata, that is punctuation, such as semicolons or quotes, in their input. In the example with our kids' word game, we added a period in the input to end the first sentence, then started a second, malicious sentence. In real injection attacks, we find exactly the same situation.

You can avoid injection attacks by preventing the user from including metadata in their input or somehow neutralizing it by sanitizing the input.

You will see examples of how to prevent or neutralize metadata when you view the modules on specific types of injection attacks. Though, in general, rejecting dangerous input is better than trying to sanitize it.

Typically, the programming flaws that enable injection attacks involve incomplete or incorrect understanding of the rules for command syntax. The most common injection mistake is not detecting all the appropriate metadata characters in the user input. String delimiters (various types of quotation), command terminators (such as period, semicolon, and new line), and escape characters (often then backslash) are the ones that you find most often. However, as we will see in the following chapters, there can be unexpected cases that can vary from programming language to language.

It is **essential** that we understand the causes of injection attacks and how to prevent them. The OWASP Top 25 list of most common security coding flaws (CWEs) in the real world shows that injection attacks account for six of the top flaws.

15.4 Summary

Injection attacks are always a concern when commands are dynamically constructed from user input. Metacharacters such as quotes and escape characters can be both the means of attack and potentially also the mitigation, but writing *ad hoc* code to validate inputs or transforming potentially dangerous string inputs into safe ones is both a lot of work and also difficult to get exactly right.

15.5 Exercises

1. Write your own Mad Libs type of story and show how, by filling in more than a single word or sentence in a given blank, you can completely change the meaning of the story.