Introduction to Software Security

Chapter 3.8: Introduction to Injection Attacks

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 2.1, March 2024.

1 Objectives

- Understand the basic principle behind injection attacks.

2 General Injection Attacks

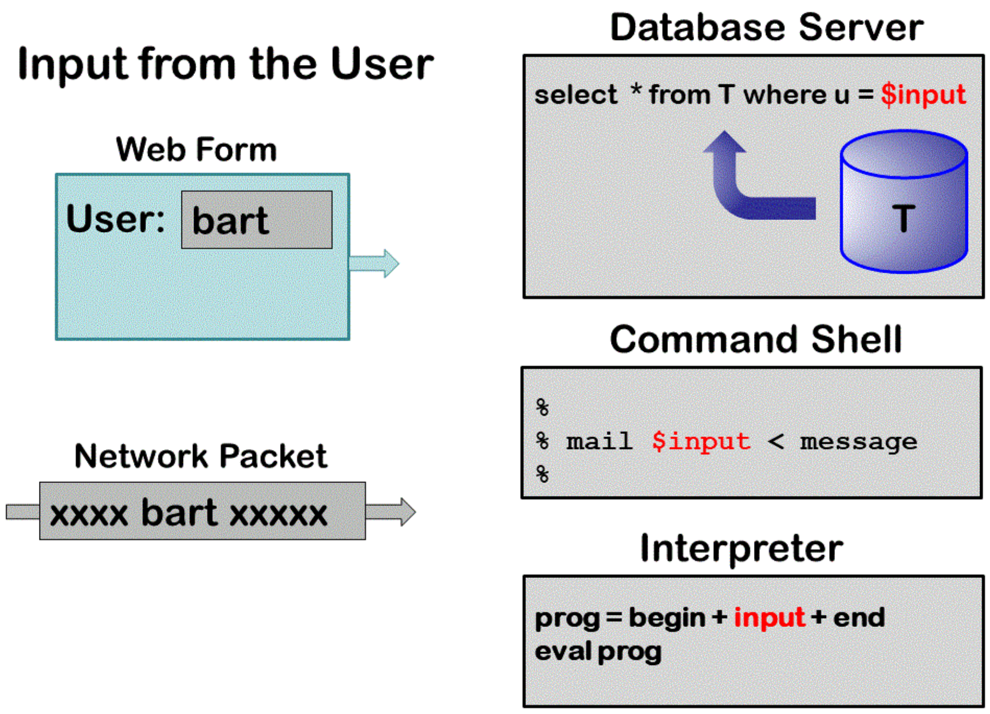
There are several types of injection attacks, depending on what part of a system is being attacked. However, they all follow a common pattern. We start with some component of the program that accepts commands in text form. This component might be a command shell, the SQL interpreter, XML parser, or even 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.

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.

3 A Human-Language Example

Seeing an actual injection makes this much clearer so to begin, consider this somewhat contrived English language example. Suppose that Aaron has agreed to convey a message from Monica to Barbara — and Aaron is rather simple minded so he just uses a basic template as follows:

```
Hello, Barbara: Monica asked me to tell you WHATEVER.
```

Monica gets to provide whatever content she likes to substitute for the term `WHATEVER` for Aaron to faithfully speak as he is told. That is, Monica gets to put words in Aaron's mouth. To dramatize the effect we assume that Aaron is honor bound to follow along.

First, as an example of intended usage, Monica might tell Aaron to convey a message such as:

```
that she will meet you at your house tomorrow noon
```

So Aaron finds Barbara and says,

```
    Hello, Barbara: Monica asked me to tell you
that she will meet you at your house tomorrow noon.
```

Now we consider what Monica might say if she were mischievous, for example:

```
that the moon is made of green cheese
```

While Aaron might feel silly conveying the message, it's clear that Monica is just being silly.

```
Hello, Barbara: Monica asked me to tell you
that the moon is made of green cheese.
```

However, a malicious Monica might do something unexpected that abuses the intended form where Aaron is clearly conveying her words. Consider if Monica said the following:

```
something that I forgot. But while I'm here, please take my wallet from my
back pocket; remove all the money, and give it to Monica right away. I owe
her the money and you will be doing me a great favor.
```

This time the effect is completely different. Note how there is no message conveyed from Monica to Barbara anymore. More importantly, Monica has literally put words into Aaron's mouth and the surrounding context couches the sentences as Aaron's own words.

```
Hello, Barbara: Monica asked me to tell you
something that I forgot. But while I'm here, please take my wallet from my
back pocket; remove all the money, and give it to Monica right away. I owe
her the money and you will be doing me a great favor.
```

While people are not so stupid as to blindly fall for this, that is exactly how software works.

In just this way attackers exploit injection bugs. Instead of providing some harmless data as intended in the context of a command, sloppy coding can allow attackers to corrupt the command itself. In doing so, the attackers get to issue commands of their choosing with the full authority of the mediating software — as if the software had commanded itself.

Whenever any kind of command mixes with user supplied data this very same trick becomes possible, so you need to be alert and take precautions lest "words be put in your mouth".

4 Syntax and encoding

When you are constructing statements or commands that may be subject to injection, it is essential to carefully handle inputs that are influenced from the attack surface. You want to ensure that all possible input values will result in valid constructions that do what they are intended to do. Following chapters provide specific examples but the common principles will generally apply.

The constructed statements or commands typically have specific rules about quoting of strings and other metacharacters such as statement separators, command-argument prefixes (e.g., `-option` in Linux or `/option` in Windows), and line continuation; these elements must be carefully handled. Be sure to enclose strings in quotes where needed, and to escape quotes that might appear inside the quotes. Escaping uses a prefix character, often backslash (e.g., tab is `\t`) to name a special character. The escape character also requires special handling, so to actually use a backslash, you need to escape it as `\\`.

In addition to syntactic concerns, it is important to handle multibyte character encodings properly. Most multibyte encodings (e.g., UTF-8) are sequences of bytes where the first byte determines if the character has one or two or more bytes. One possible danger with such encodings occurs when the input byte sequence ends with a byte that requires more following bytes to represent a character, and those following bytes are omitted. If this happens and you catenate byte sequence with other strings then the next byte may get consumed as the second byte of a multibyte character and effectively become lost, breaking out of quotes for example.

5 Types of Injection Attacks

In software, whenever a string is synthesized from data such as user input, and that string is later parsed and interpreted as a command, this risk of unexpected effects exists. Examples include:

- SQL statements
- Command line statements, such as in a command shell
- Code injection in interpreted languages like JavaScript, Python, Perl or Ruby
- XML injections when processing stored to network-transmitted XML data
- Cross-site scripting in HTML

Typically the programming flaws that enable injection attacks involve botching the handling of user data in the formation of commands when untrusted data is involved. In practice this means accepting data from an attack surface where attackers can enter whatever strings they like via a web page form or an URL parameter, and from that constructing a SQL query that the more privileged server code then executes against its database.

The most common injection mistake is naively believing that putting quote characters before and after an arbitrary string necessarily forms a single quoted string. Alternatively, unexpected metacharacters may allow the user supplied data to break out of its intended use and become part of a command. Even when the programmer anticipates the injection and checks the user string for tricky things like embedded quotes, unless the code designed to neutralize attacks anticipates all possible malicious forms it may allow some obscure attacks to still achieve the injection.

6 Summary

Injection attacks are always a concern when commands are dynamically synthesized using untrusted inputs. Metacharacters such as quotes and escape characters are typically both the means of attack and potentially also the mitigation, but writing *ad hoc* code to validate inputs or morph potentially dangerous string inputs into safe ones is both a lot of work and also risky to get exactly right.

SQL injection is the classic most common case, but the same kind of flaw can arise with shell commands, XML, or constructed customized scripts. Your next step is to study each of these types of injection attacks in more detail. You can continue your exploration of this topic in each of these four chapters:

3.8.1: SQL Injection Attacks

3.8.2: Command Injections

Exercises

1. The game MadLibs is a childhood favorite, where a given story is provided with blank spaces to be filled in. The players choose their words before they actually see the story, so there is usually a pretty fun mismatch between the intent of the story and the one that includes the players' words. Write your own MadLibs type of story and show how by filling in more than a single word in a given blank, you can completely change the meaning of the story.