Introduction to Software Security

Chapter 3.8.2: Command Injections

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 1.7, March 2019.

Objectives

- Briefly review the general problem of injections.
- Understand how command injections work in some examples.
- Learn how to mitigate the risk of command injections.

Examples from Perl, Python and Ruby.

Command Injection Attacks

Injection attacks are a general vulnerability pattern where a string constructed by the attacker, usually via user input, is inadvertently interpreted not as an arbitrary string but in such a way as to influence the behavior of the program. SQL injection is the most well-known example of this class of attack, but in this chapter we explore shell command injection vulnerabilities.

These vulnerabilities generally result from sloppy handling of arbitrary data coming from the attack surface used in combination with predefined shell commands intended to do some safe operation. Usually these vulnerabilities arise from improper handling of metacharacters such as punctuation, quote marks, or escape characters in the input that confuses the parsing of the resultant command.

In the case of command injection, the vulnerability is due to clumsily constructing a shell command that incorporates runtime data that the attacker can influence (most commonly input from an untrusted source). This class of injection attacks that is the subject of this chapter is also called “shell injection”. The essence of these attacks is that malicious input from an attack surface is used to coerce a constructed command to perform unanticipated harmful actions when executed by a shell the program invokes. The difference in naming is just between the changed meaning of the command, or the changed effect of the shell process executing that same command.

A very simple example illustrates the potential pitfalls best: suppose our program simply wants to take user input such as “Hello” (to be specific, meaning that the input is the five characters inside the quotes, not including the quotes themselves), and execute the shell command shown below which will print that message (without the quotes) to the console.

```
echo “Hello”
```

To see the vulnerability we must look closely at how that command was formed by catenating the untrusted five letter string injected between quotes. The following table shows several possible input values and the resulting command that would be formed.

Untrusted input string value	**Resultant command to be executed**
`simple`	`echo “simple”`
`invalid\`	`echo “invalid\”`
`!”; rm -rf /*; echo “!`	`echo “!”; rm -rf /*; echo “!”`

The first simple example shown above is the expected case and it works fine. The second one produces an invalid command because the backslash quotes the trailing quote such that the quoted string is unterminated: this is a bug, and the resulting malformed command is rejected without doing any harm. The final example is an actual exploit since it produces three commands (since semicolon is a command separator and each resulting command is well formed), the second of which removes all files on the system (if executed with the necessary privileges).

When an untrusted source supplies values that are potentially controlled by an attacker, always assume the string may be maliciously crafted. Inputs that contain special characters that may affect how the shell command is parsed must be validated, with troublesome characters quoted, escaped, or otherwise avoided.

There are a few strategies, the simplest is to choose a safe subset of characters that cover all reasonable uses of the input and you know will not influence command parsing, and reject strings that contain any other characters being sure not to execute the resulting command. Alternatively, scan and remove any offending characters and proceed using only the safe characters that remain.

The downside of the latter approach of removing problematic characters and proceeding is that the resulting command may not do what the user intended, since they did not expect input characters to be filtered out. For example, forming an email sending command you might worry about hyphens that could be interpreted as (Linux) command arguments and drop these; however, hyphens are valid domain name characters so the email intended for `admin@ex-ample.com` would go to `admin@example.com` instead.

In order to handle command metacharacters then instead of rejecting them, you need to accommodate them safely. Typically this means escaping or quoting the special characters so they will be properly handled in the final command. This sounds simple but it is critical that you handle each and every one of these just right as any mistakes are an opportunity for an attack. Doing this accurately for all inputs requires a very detailed understanding of shell command syntax and then writing program logic that protects against all the potential unintended consequences.

Note that these vulnerabilities are not attacks on the shell, but rather are attacks that create malicious command lines that are executed by the shell.

Signs of shell command injection vulnerability to watch out for include:

- Use of (Linux) `popen`, or `system`.
- Programmatic execution of a shell such as `sh`, or `tcsh`, or `bash`.
- Argument injections (use of `exec`), allowing arguments to begin with "-" can be dangerous.

Example: A Server Sending Email

Let's consider an in-depth example to more fully understand the threat of command injection attacks and how to correctly defend against it. This example illustrates that secure implementation must not only anticipate some potential attacks, but that it is necessary to defend against all of them.

For example, consider a website that allows customers to order something and will use email as a communication link to provide updates during delivery of goods or services. The specifics are unimportant for our purposes, but commonly servers will send email to users to let them know about events such as: Your package arrived; Your flight is canceled; You are over the credit limit.

The email address comes from input that is provided by the user via a dialog such as shown here. In this case, the user provides their email address (filling in the top blank line) and optionally checks the box and adds a customized message text (the lower blank line).

Let us notify you on delivery: _____

☐ Add a message: _____

The naive programmer would expect an email address such as `me@example.com` to be provided, and a common (and risky) way for a program to send email is to generate a command-line such as:

```
/bin/mailx -s "Your package" me@example.com
```

While this works fine for normal and valid email address inputs, consider what happens with malicious input:

```
you@bad.com ; evil -cmd
```

Now the resulting command that gets executed is actually two commands: a mailx command followed by a second generic evil command of the attacker's choosing.

```
/bin/mailx -s "Your package" you@bad.com ; evil -cmd
```

Same Example: A More Arcane Attack

Suppose we anticipated and sanitized some input characters, such as double quotes (“), semicolon (;), ampersand (&), and vertical bar (|), from the email address input field in the example described just above and closed the door on that attack surface. That’s progress, but based on a real vulnerability the authors discovered in an security audit of production code for a client, this server was not yet secure.

Taking the attacker’s perspective and accepting that the first input field is now fully protected, consider how else a command injection attack might be done. By checking the message box we could try specifying a text message, but that just comprises the email text that will be sent to ourselves as a message … could that possibly be used to inject a command?

Linux commands have a long history extending back to the original Unix commands (over forty years) and inevitably over time commands accrue many obscure arguments and extensions over time. Sure enough, the mailx command offers a feature called “tilde extensions” that must be enabled with a command line argument; when enabled, message body text lines beginning with a tilde (the ~ character) are parsed as special extensions and can be used to do all kinds of things. One tilde example (~!) prefixes an arbitrary shell command to be executed with the output inserted into the email body text, and that’s exactly the opening needed for a command injection attack.

Let us notify you on delivery: **`you@bad.com -~`**

☑ Add a message: **`Goodbye, filesystem:`**
`~! rm -rf /`

Now the resulting command that gets executed is the following:

```
/bin/mailx -s "Your package" you@bad.com -~
```

With the given message body containing the tilde escape prefixed command **`rm -rf /`** that gets executed and destroys the system by removing every file the process has privileges to do so.

Command Injection Mitigations

Building shell command strings using data that can be provided or influenced from an attack surface is best avoided in the first place since it is very difficult to do securely. Instead of invoking a shell with an ad hoc command, use a standard library that provides the desired functionality. In the case of sending email, examples of appropriate services for some common languages include:

- Java: There are many choices, such as the standard JavaMail API.
- Python: Also many choices, such as the standard email package.
- Perl: Also choices, such as the popular MIME::Lite or Email::Stuffer packages.

There are also language-independent solutions for this email problem; you can use a generic Web-based service to send the notifications, such as mailgun, MailChimp, Drip, and SendGrid.

If you are going to build command lines and invoke them in a shell, despite the risks of abuse, here are some of the issues you will need to defend against. Note that there are a number of different shells, each with their own particular syntax and quirks.

- Check user input for metacharacters such as ";" (statement separator), "\" (escape prefix), and quotes.
- Neutralize metacharacters (using escape or quotes) that can't be eliminated or rejected.
- Ensure that inputs are valid character encodings, specifically with multibyte characters.

As an additional layer of security, you can ensure that the shell executes at the least privilege necessary to perform its intended function. On Linux, first use `fork` to create a new process, then drop any excess privileges. Dropping privileges can include closing unneeded files and changing the user or group context to minimize access (using `setuid` and `setgid`). Then, use `exec` to run the shell. If a command injection attack should somehow succeed, then the scope of its damage can be limited in this way.

Perl Command Injections

Even if you do not intend to execute a command line in a shell it can happen as a side effect of using what looks like an innocuous library. For example, consider the Perl open function:

```
open(F, $filename)
```

Depending on the syntax of the string `$filename` this can: open files in various modes, start programs, run commands, or duplicate file descriptors. For example, if `$filename` is `"rm -rf /|"`, it will execute the rm command in a shell and delete the file system root (if privileges allow).

Here are some examples in Perl that are vulnerable to shell interpretation:

```
open(C, "$cmd|")        open(C, "-|", $cmd)
open(C, "|$cmd")        open(C, "|-", $cmd)
`$cmd`                  qx/$cmd/
system($cmd)
```

The string **`$cmd`** is parsed by the shell, so is subject to injection if user input is used to form this string.

Perl does have alternatives that are safer from shell interpretation:

```
open(C, "-|", @argList)
open(C, "|-", @argList)
system(@argList)
```

In these array argument forms, the program name and each argument are in a different location of the array `@argList`. By providing separate arguments as individual strings, attackers cannot use metacharacters to manipulate the interpretation of the command.

The following is an example of the dangerous form of Perl command shell invocation:

```
open(CMD, "|/bin/mail -s $sub $to");
```

For example, if `$to` is "`badguy@evil.com; rm -rf /`" then the file system root will be deleted.

Here is the safe and simple way to do the same thing:

```
open(cmd, "|-", "/bin/mail", "-s", $sub, $to);
```

Even if `$to` is "`badguy@evil.com; rm -rf /`" that entire string will be treated as the destination address for the email (which is invalid and simply prevents sending anything); the "rm -rf" is just part of the invalid email command and is not executed in a shell.

Other language Command Injections

In Ruby, functions prone to injection attacks include:

```
Kernel.system(command)
Kernel.exec(command)
`command`                    (backtick operator)
%x[command]
```

In Python, functions prone to injection attacks include:

```
os.system(command)      # execute a command in a subshell
os.popen(command)       # open a pipe to/from a command
```

Summary

Command injections provide an opening to attack when a program constructs shell commands at runtime from strings coming from or influenced by the attack surface. Shell commands are handy to do many things, but because they are powerful this constitutes a great opportunity to attackers.

Command injections can arise in almost any language, sometimes even unexpectedly via special syntax or hidden feature of a library call. Whenever possible, avoid trying to build command line strings directly; instead, use standard libraries to perform the functionality. If you must construct commands, familiarize yourself with the details of command line syntax, and be wary of inputs from a potential attack surface: carefully check input validity and handle quote and escape characters properly.

Exercises

1. Write a simple example program that constructs and executes a command line from user input strings. Study the shell command syntax and see how many different ways (with escape, quote, statement separators, multi-byte tricks) you can subvert the intended function. Hint: instead of harmful commands attackers would use, do something safe like `echo Gotcha`
2. Modify the program from Exercise 1 to defend against all attacks. Try the same attacks and confirm they are now harmless. Can you find any new attacks that still work? Try trading your code with a colleague and attacking (without causing harm) each other's defenses.

3. In a production system, when an invalid input potentially causing a command injection is detected, what are the pros and cons of the possible responses — reject with an error message, silently filter out offending parts of the input?
4. (Advanced) Read the command line specification for a popular shell and critique the design of command syntax for how easy or hard they make it to safely construct command strings. How could the command syntax be improved to be make it easier to avoid command injection attacks?