Chapter 17
Command Injection Attacks

Revision 3.2, December 2025.

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.

17.1 Background

Underlying a command injection attack is some component of your system that is going to use a command shell – like tcsh, bash, or PowerShell – to execute commands that come in the form of a character string. Executing a shell command from a program is a common feature found in most programming languages and operating systems. (We provide examples of such interfaces in Section 17.4.)

Your system is going to construct commands to cause the shell to execute some program. If input from the user is used to construct these commands, then there is a path from the attack surface to the command. In this case, the user's input helps to determine the meaning of the command. If you are not a careful programmer, and the attacker is clever about how they create their input, they can change the meaning of the command that you generate in ways that can create security problems.

In general, the cause of a command injection attack, like other types of injections, is allowing the user to include metacharacters such as semicolons, quotes, or escape characters in their input and not rejecting the input or neutralizing dangerous characters.

Note that a command injection attack is not an attack on the shell itself, but rather an attack that creates malicious command lines that are executed by the shell.

17.2 Example: A Server Sending Email

Now we will look at an example of a command injection attack and show how to defend against such attacks. This example comes from another real system for which we did a security evaluation.

For our example, consider a website that uses email to communicate with a user when something important happens. For example, when their package is delivered or their flight canceled or when they go over their credit limit.

The email address that is used may come from input that is provided by the user via a dialog such as shown in Figure 1. In this case, the user provides their email address filling in the blue box shown.

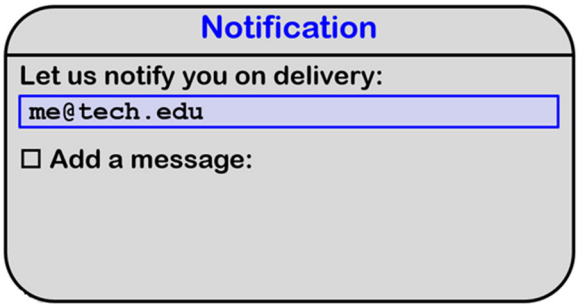

Figure 1: Web Interface to Enter a Delivery Notification Request

A common, easy (and risky) way for the web server to send an email is to generate a command-line such as:

```
/bin/mailx -s "Your package" me@tech.edu
```

This command line executes `mailx`[1], the POSIX standard command line email program. `mailx` is a descendent of the command line mail programs used on the earliest of Unix system from the 1970s. The `-s` option specifies the subject line for the message and the rest of the parameters (only `me@tech.edu` in this case) are addresses of the email recipients. The web server runs the above command by passing the command string to a function that executes a shell command.

The website expected a simple email address to be entered into the web form, such as `me@tech.edu`. However, a malicious user might try to craft an input for the email address field of the web form that would cause an unexpected and undesirable result. We see an example of such an input in Figure 2.

In this case, the malicious user entered:

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

Right away, you can see a classic sign of an injection attack: there is a command line separator, the semicolon, present in the input. The result of this input would be the command line:

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

[1] `https://man7.org/linux/man-pages/man1/mailx.1p.html`

Figure 2: Web Interface for Delivery Notification with Malicious Input

There was a path from the attack surface to the construction of the command. So, in addition to sending an email message, the attacker can add any command of their choice, creating a remote execute attack. In this example, we show this arbitrary command as `evil-cmd`.

17.3 Same Example: A More Arcane Attack

Suppose the programmer of the web server understood the threat of command injection attacks and checked carefully for the dangerous class of inputs in the email field. Such dangerous inputs might contain characters such as double quotes (`"`), single quotes (`'`), semicolons (`;`), ampersand (`&`), and pipes (`|`). Checking for these characters and rejecting inputs that contain them would prevent the attack that we showed in Figure 2. In our evaluation of the system that contained this vulnerability, the programmers fixed this weakness in the code in exactly this way.

However, in our security team meeting, we asked the question "Is there any remaining attack surface for this email notification feature?" The answer was clearly "yes", as the web form shown in Figure 1 and Figure 2 has another field. By checking the "Add a message:" box, an additional field appears for a message to accompanying the delivery notification, as shown in Figure 3.

Figure 3: Web Interface for Delivery Notification with Additional User Message

Linux commands have a long history extending back to the original Unix commands (over forty years) and some now-obscure features to these programs still remain. Sure enough, the `mailx` program has some of these features. One of these features is called the "tilde extensions" that allow a variety of special kind of lines to appear in the email message. For example, a line that starts `~t` specifies additional "To" recipients for the message; `~c` specifies additional "Cc:" recipients, `~b` specifies additional "Bcc:" recipients, and `~s` specifies the subject line for the message. So, you might see lines in an email message that look like:

```
~s Your package was delivered
~c dad@email.com
~b you@bad.com
```

These are all reasonable and clear features. However, there is another feature that seems more unexpected. If you specify `~!`, the remainder of that line contains a command to be executed by the shell. It simply executes the command, printing the output to the terminal, without affecting the contents of the email message. A line that uses this feature might look like:

```
~! ls -lt
```

It is reasonable to ask, from a modern perspective, why did they need to be able to execute a shell command that did not affect the message in the middle of typing the message? Why not just type the message in another window or suspend the program? The answer dates back to the 1970s when the first version of the command line mailer program appeared. In those days, there were no windows. Users logged on to a system from what is now known as a "dumb terminal", with 24 lines of 80 characters each. Effectively, every user was limited to a single window. And this was also before the control-Z feature that allows you to suspend a program from the shell to do something else. So, this feature allowed the mail user to execute a command (such as checking the contents of a file) while in the middle of typing an email message.

We now have the means to execute a command from the contents of the email message. However, there is one more hurdle that we have to overcome. To use the `~!` feature, you must run the `mailx` program with the `-~` command line option. So, putting it all together, a malicious user might fill out the form as shown in Figure 4. Note the extra command line option in the email address field and the command injection in the message contents.

The shell command that gets executed is:

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

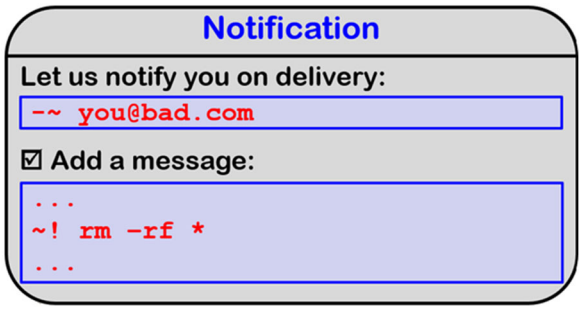

Figure 4: Web Interface for Delivery Notification with Malicious Message Contents

And the message body would use the tilde feature, which executes the command "`rm -rf /`".

Preventing this second attack is not difficult. Simply reject any line in the message field that starts with the tilde character. You can also check to prevent the `-~` option in the email field.

17.4 Language and Operating System Interfaces to Execute Shell Commands

There are a large variety of functions or methods that will take a string input and execute it with a command shell. The functions or methods can be called within a programming language or directly as a system call to the operating system. Here, we present a few of these. For each programming language and operating system that you use, you will need to be familiar with the functions or methods that are available.

17.4.1 Operating System Calls

There are variety of ways to execute a shell command using an operating system interface. The most basic way is to simply use the create-process feature to start a new program, specifying a shell as the program to run. On Unix, you would use one of the `exec`[2] system calls to start the new program running, presumably after using `fork` or `clone` to create a new process. On Windows using C#, you can use the `Start` method in the `Process` class to create the new process.

There are also direct calls to run a shell command. On Unix, you simply use the `system` library function. The string parameter to `system` is executed as a command to a shell. On Windows using C#, you would use the `PowerShell` class and run the `Invoke` method to execute a shell command.

[2] `https://man7.org/linux/man-pages/man3/exec.3.html`

17.4.2 Perl Command Injections

Sometimes the designers of programming languages are too clever. 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 all the files on the system (if privileges allow).

Essentially, code that processes the file name in Perl is a mini programming language. If the pipe ("`|`") character appears at the beginning or end of a file name, instead of doing a normal file open, it invokes the `popen` system library function (on Unix). `popen` starts a new process that is running the shell and creates a pipe (message channel) to that process. The string in the $filename parameter (minus the pipe character) is passed to the shell as command to be run. Any reads or writes (depending on where the pipe character is placed) on the file either sends (writes) input to the running command or receives (reads) output from command. The key issue is that something as innocent looking as an `open` call can actually start a shell.

Figure 5 shows the many ways that you can run a command in the shell from a Perl program. An interesting question is: why are there so many distinct ways to do this? There probably is not a good answer to that question. It might simply be due to a lack of discipline by the language designers.

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

Figure 5: The Many Ways to Execute a Shell Command in Perl

The string `$cmd` is parsed by the shell, so is subject to an injection if there is a path from the user input to the place where this string is constructed. If that input contains metacharacters like separators, quotes, or escape characters, the attacker might be able to control what is executed.

Perl does have alternatives that are safer from command injection:

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

These versions separate a shell command into an array of its component tokens, `@arglist`, so the command is never parsed. The first element of the array is the command (the file that contains the program to run). The remaining elements form the rest of the command line, i.e., the options and parameters to the command. Since the line is not parsed, metacharacters contained within the various tokens cannot affect parsing.

The following is an example of the dangerous form of invoking the shell from Perl:

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

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

Here is a safer 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 recipient address for the email. Since no valid email address could ever have spaces, a semicolon, and a slash, this send operation will fail.

The bottom line is that there are simply too many ways to invoke a shell in a Perl program. So, these programs require extra carefully scrutiny.

17.4.3 Other language Command Injections

Python has many fewer ways than Perl to execute a shell command and none of them built into the language. In Python, you use the `os` class, which provides access to the underlying operating system calls. In this case, Python provides access to the `system` and `popen` system calls:

```
os.system(command)      # execute a command in a shell
os.popen(command)       # start a shell and open a pipe
                        # to/from the command
```

Ruby provides a class called `Kernel` to provide the same kind of access to system calls. In addition, the language provides two features that will execute a command in a shell. The backtick symbol (`) syntax is also used in Perl and original comes from the C-shell (csh) written by Bill Joy in the late 1970s. The `%x[]` notation is an invention unique to Python, though why they needed another way to execute a shell command is open to question.

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

17.5 Command Injection Mitigations

Building shell command strings using data from the attack surface is best avoided since it is difficult to do securely. Instead of invoking a shell from within a program to accomplish a certain task, it is better to use a standard library, modules, or package to do the same thing.

For example, in the case of sending email as we described in Section 17.2, instead of starting a shell to run the mailx command, you might use one of these standard libraries, modules, packages:

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

By calling these packages directly, you avoid executing a command in a shell thereby preventing a command injection.

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`, Intuit `MailChimp`, `drip`, or Twilio `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 many different shells, each with its own particular syntax and quirks.

- Check user input for metacharacters such as statement separators, quotes, and escape characters.
- Alternatively, use an allow list to determine which user input is valid
- Reject input that appears unsafe or invalid. Do not try to fix it.

17.6 Summary

Command injections provide an opening to attack when a program constructs shell commands at runtime from strings coming from the attack surface. Shell commands are handy to do many things, but because they are so powerful this provides 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.

17.7 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. In Figure *5*, we listed a variety of methods by which a command could be run in a Perl program.
    a. Find the appropriate documentation and describe how each method works. Your explanation should be detailed enough so that you could write a program using the method.
    b. Write a simple program that demonstrates the use of each method. Make sure to execute shell commands that have no potentially dangerous effects.
3. Both Perl and Python support the backtick operation, where you can type `command` to execute a command in a shell. However, when you use the backtick operator, it does more than just execute the command in the string enclosed between the backticks. Explain this difference between using the backtick operators and simpler mechanisms that simply execute the shell command.
4. Consider a language other than the ones discussed in this chapter (Perl, Python, and Ruby), perhaps one that you regularly use for program development. For that language:
    a. Identify the features in that language that will execute a command in a shell.
    b. Identify what libraries, packages, or modules are available to that language for creating and sending email.
5. 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 or try to correct the offending parts of the input?
6. (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 make it easier to avoid command injection attacks?