Chapter 18
Code Injection Attacks

Revision 3.1, December 2025.

Objectives

- Describe how the general problem of injections applies to code injections (also known as “language injection”, “dynamic evaluation”, or just “eval” attacks).
- Understand what code injections are and see some examples of how they work.
- Learn how to mitigate code injection vulnerabilities.

18.1 Background and Overview

Most interpreted languages allow the program to create strings that contain code written in that language and then dynamically execute that code. Consider the Python code in Figure 1. The `eval` built-in function is typical of many programming languages; it takes a string and executes the code contained in that string. So when this example code is executed, it will print “`Hi mom.`” While this feature is extremely powerful, it must be used with great care. In general, avoiding using dynamic code execution is a good idea.

Code injections occur when user input is used in the construction of code that is dynamically executed with a built-in function such as `eval`. In other words, if there is a path from the attack surface to the dynamic execution of code, we have a code injection.

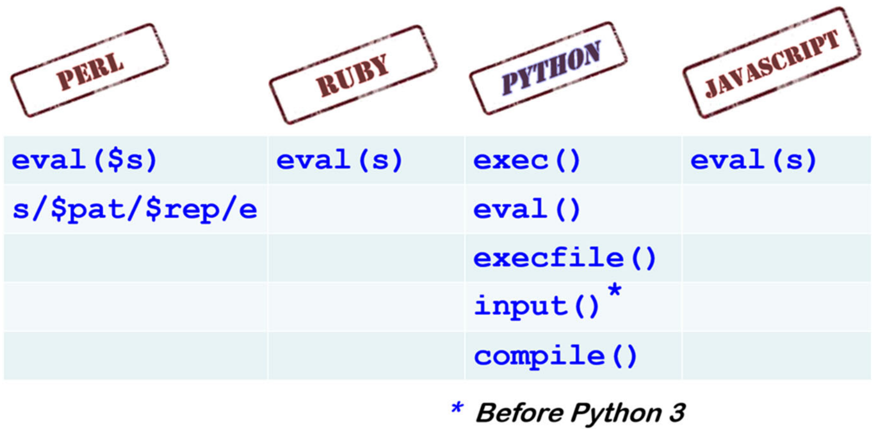
Since a code injection might allow an attacker to create and execute arbitrary program code, these attacks can be quite dangerous.

```
cmd = "print(\"Hi mom.\")"
eval(cmd)
```

Figure 1: Python Code to Demonstrate the Use of `eval`.

18.2 Language Mechanism to Dynamically Execute Code

We will start by describing various mechanisms by which interpreted languages can execute code dynamically. Each use of one of these mechanisms needs to be evaluated for the potential of a code injection attack. Figure 2 summarizes these features in four of the languages we will discuss.



PERL	RUBY	PYTHON	JAVASCRIPT
`eval($s)`	`eval(s)`	`exec()`	`eval(s)`
`s/$pat/$rep/e`		`eval()`	
		`execfile()`	
		`input()`*	
		`compile()`	

* *Before Python 3*

Figure 2: Summary of Dynamic Code Execution Features of Four Languages

18.2.1 Perl Language Injections

Mechanisms in Perl that can dynamically execute code are:

```
eval($s)
s/$pat/$replace/e
```

The first mechanism is the most obvious and typical, the `eval` built-in function. It takes a string as a parameter and then executes the code contained in that string.

The second form is subtle and strange. It looks like a regular expression substitution where a string is scanned for the pattern contain `$pat` and each place where that pattern occurs, it is replaced by the string in `$replace`. However, when the special “`e`” modifier is added at the end, it behaves quite differently and unexpectedly. Instead of using $replace as the value for replacement, this value is executed as Perl code and then the replacement string value is the *output of executing that code*.

And it gets even stranger. You can have more than one “`e`” at the end, so if the modifier at the end was “`ee`”, it would execute the code in $replace and then take the output of that code and execute it as code. The final value of the replacement string would be the output of this second round. You can have as many “e” modifiers at the end as you want, creating no end of confusion.

Another way to think about the “e” modifier is that `s/$pat/$replace/e` is equivalent to:

```
$temp = eval($replace)
s/$pat/$temp/
```

This is a good example of why it is important for programmers and analysts to be fluent with all the details of the languages that they use. And, when you combine a feature such as this one with the plethora of shell execution command mechanisms, you can see why Perl is often considered a risky language to use from a security perspective.

18.2.2 Ruby Language Injections

Ruby only has the straightforward form of `eval`, which works as much the same as it does in Perl. Naturally, the string value is interpreted as a Ruby statement and executed.

```
eval(string)
```

From a security point of view, the design of Ruby helps secure coding by making it clear when a code injection attack might be possible by having only one easily identifiable mechanism for dynamic code execution.

18.2.3 JavaScript Language Injections

Like Ruby, JavaScript has only the eval built-in function for dynamic code execution. Its syntax is similar to Perl and Ruby. The string argument can be a JavaScript expression, a statement, or a sequence of statements.

```
eval(string)
```

18.2.4 Python Ruby Language Injections

Python is more complex than Ruby and JavaScript, having several built-in functions that are associated with dynamic code execution:

```
exec(string)
eval(string)
execfile(string)
input(string)     # only risky before Python 3
compile(string)
```

The `exec` function in Python operates similarly to the Perl and JavaScript `eval` functions in that the string can contain arbitrary Python code. The `eval` function in Python can only take an expression to be evaluated. However, since that expression can contain function calls, it can dynamic execute a pretty wide variety of code.

The `execfile` built-in function is a combination of reading string input from a file and then executing the program contained in that string input. The `input` function is also a combination that reads one line from standard input and evaluates that string as an expression. However, this behavior was changed in Python 3, where the input function no longer evaluates the input string.

18.3 A Simple Code Injection Example

We will start our discussions of attacks with the simple example shown in Figure 3. This is Python code that implements a basic calculator that computes the result of the input provided by the user. While simple, this represents a common reason that programmers use dynamic code execution.

```
01 comp = input('\nYour computation? => ')
02 if not comp:
03   print ("No input")
04 else:
05   print ("Result = ", eval(comp))
```

Figure 3: Code that is Vulnerable to Code Injection

The `input` function on line 1 reads a line from standard input. The `eval` function on line 5 performs the evaluation of the expression provided. Given that `eval` evaluates the input as a Python expression, it can also calculate values if you prefer. For example, if the input was `30*12+5`, then it computes the value and outputs "`Result = 365`".

However, Python expressions can include some more dangerous things such as function calls. Consider this input to the program in Figure 3:

```
__import__('os').system('rm -rf /')
```

The `__import__` function dynamically imports the module named by the string provided, so evaluating this expression will call the standard `os.system` function that invokes a shell to execute the given command (`rm -rf /`).

```
comp = input('\nYour computation? => ')
if not comp:
    print ("No input")
else:
    if validate(comp):
       print ("Result =", eval(comp))
    else:
       print ("Error")
```

Figure 4: Code from Figure 3 with the Vulnerability Fixed

To mitigate this code injection vulnerability, it is necessary to validate the input supplied by the user before calling `eval`. In Figure 4, we added a call to `validate`, which should check that only characters belonging to an allow list are used. In this case, the allow list might contain only valid numeric characters plus arithmetic operators such as + or ×. Or it could use a block list to check to see if the input includes forbidden characters. Crafting either of these solutions is definitely tricky and requires a lot of attention. In the end, it would be best to avoid the use of `eval` altogether.

18.4 A More Arcane Code Injection Example in Perl

We will now look at a more complicated example of a code injection vulnerability. This is another that we found during one of our code evaluation exercises.

The scenario for this example started when a large multi-organization team was building a complex distributed system in Python. As you would expect from any sophisticated system, it included a logging facility to record significant security, performance and status events. Of course, as was the rest of the system, it was written in Python.

Sometime later, a new group joined this software team to contribute a module with new functionality. For some reason, this team implemented the new module in Perl instead of Python. The new team wanted to add logging to their module but had a problem: the logging module was written in Python and their code was written in Perl.

In this situation they had two obvious choices:

1. Write a compatible version of the logging module in Perl that interoperates with the existing Python version.
2. Rewrite the new system module in Python so it can use the existing logging module.

Both of these choices were unappealing as they would involve writing a lot of new code. So, this team selected choice 3, "none of the above".

Their solution was to record the log records locally in a file and then, sometime later, send the log records to the logging module. Their Perl code started up the Python interpreter, and then generated Python code on-the-fly to do the logging and sent that code to the Python interpreter to be executed.

```
@data = ReadLogFile('logfile');
open(PH,"|/usr/bin/python");
print PH "import LogIt\n";
foreach (@data) {
  print PH "LogIt.Name(‘$_')\n";
}
```

Figure 5: Perl Code to Start the Python Interpreter and Send it Logging Calls

As you might guess, this design resulted in a serious code injection vulnerability, and one that was not easy to notice given the complexity of their design. Figure 5 shows how they coded this in Perl. Since this code is quite trickly, we will go through it line by line.

Line 1 calls the function `ReadLogFile`, which that reads a file of log records containing names into an array named `@data`.

Line 2 used the pipe character feature of the Perl `open` function (as we discussed in Chapter 17) to start the Python interpreter. The handle for the open file, `PH`, was actually a pipe (message channel) to the Python interpreted that is used to send input the interpreter. Any writes to open file `PH` resulted in an input line being sent to Python.

Line 3 contains a print to PH, which causes the first input to be sent to Python. In this case, it was telling Python to load the logging module, named `LogIt`.

Line 4 loops through the `@data` array and for each name in the array and line 5 uses a `print` statement to send the line to the Python interpreter to create a log record.

To see how this code injection works, consider two input lines, first a normal line and then a malicious line that contains an attack.

```
name = John Smith
name = ');import os;os.system("evilprog");#
```

For the first normal input line that contains a name (John Smith), the Perl code generates the expected Python statement and sends it to the Python interpreter to be executed:

```
LogIt.Name('John Smith')
```

The Python interpreter parses this line and invokes the `Name` function in `LogIt`, as planned.

However, the second line is attended to be malicious so now we will look at how this string is process. When the `print` statement on line 5 executes, it will send the following line to the Python interpreter for execution:

```
LogIt.Name('');import os;os.system("evilprog");#')
```

This line consists of three statements, shown below on separate lines:

```
LogIt.Name('');
import os;
os.system("evilprog");
#')
```

The first line invokes `LogIt.Name` with an empty string parameter. The second statement imports the standard `os` module and the third statement invokes `os.system("evilprog")` that causes a shell to execute some malicious activity[1]. The last statement is just a comment that was added so the remaining characters do not cause a syntax error that would potentially prevent the attack from occurring or raise an error that might make discovery of this mischief more likely.

18.5 Mitigating this Example in Perl

The fix to this code, shown in Figure 6, was designed to prevent any of the metacharacters from affecting the way that the code executes by escaping them. The checking and escaping is done by calling function `QuotePyString` on line 5 before sending the value to the Python interpreter on line 6. The code for `QuotePyString` is shown in Figure 7.

```
01 @data = ReadLogFile('logfile');
02 open(PH,"|/usr/bin/python");
03 print PH "import LogIt\n";
04 foreach (@data) {
05   my $val = QuotePyString($_);
06   print PH "LogIt.Name('$val')\n"
07 }
```

Figure 6: A Fix to the Dangerous Perl Code in Figure 5

```
sub QuotePyString {
  my $s = $_[0];       # Copy input string parameter to $s
  $s =~ s/\\/\\\\/g;   # backslash \ becomes \\
  $s =~ s/'/\\'/g;     # ' becomes \'
  $s =~ s/"/\\"/g;     # " becomes \"
  $s =~ s/\n/\\n/g;    # \n becomes \\n
  return $s;
}
```

Figure 7: The Checking and Escaping Code Used in Figure 6

For the first normal input line (John Smith), the output does not change. However, thanks to the new escaping code, here is the result for the malicious attack string:

```
LogIt.Name('\');import os;os.system(\”evilprog\”);#')
```

Since all the single and double quotes of the input have been escaped, this is a single valid (but bizarre) string that the logging component can harmlessly log. This approach to fixing the vulnerability was chosen because it was the

[1] Note that this example has both a code injection and a command injection.

least amount of work to get a solution. However, because we are trying to fix the input, it is a fragile solution and not the best choice.

18.6 Defenses Against Code Injections

As we have discussed, if there is a path from the user input to the construction of the code string, you have to defend yourself against the user manipulating the code to accomplish a malicious goal. The defenses follow the same principles as we have previously discussed for SQL and command injection attacks.

Your best defense is to simply not construct code strings and dynamically execute them. Avoiding this scenario removes the whole problem.

If you decide that you need to construct code strings based on user input, you must ensure that the input has no metadata characters (and reject input that does have them). If you take this approach, you must thoroughly validate the input, detecting metacharacters that might introduce unintended escaping, quotation, or statement separators to alter the intended semantics of the result.

You might also try to neutralize the metacharacters by quoting or escaping them, though this is always a much riskier approach and should not be used unless there is a compelling reason not to reject questionable input.

Summary

- Code injections arise when a program constructs program code at runtime and there is a path from the attack surface (typically user input) to the string to be executed as code.
- To mitigate code injections, it is best to avoid constructing code at runtime for execution.
- If you must execute generated code then carefully validate the input for signs of injection. Avoid trying to sanitize the input as this is always a riskier approach.

18.7 Exercises

1. Write a simple example program that constructs and executes statements based on user input strings. See how many different ways you can trick this program into executing arbitrary code specified by various tricky inputs. Note: never actually do malicious attacks but instead, as a proof of principle, do something obvious and harmless.
2. The code in Figure 4 includes a call to a `validate` function that attempts to check to forbid anything except valid arithmetic

expressions. Write the code for this function. Try writing it using both allow list and block list approaches.

3. Choose a language with one of the languages listed in this chapter, or choose another interpreted language if you prefer that is capable of dynamically executing string values as code. How might you check a large body of source code to find possible code injection vulnerabilities?
4. Pick an open source project written in an interpreted language:
    a. See if it includes any of the danger signs indicating potential code injection vulnerability.
    b. Investigate if these are actual vulnerabilities or not (either by source code inspection or experimenting with running code). Of course, as always, set up a private instance and never attempt anything malicious.
    c. (Advanced) If you think that the code is vulnerable, try to create a proof-of-concept that demonstrates how the code might be attacked. If you are successful, responsibly let the project owners know about the vulnerability and propose necessary mitigation.