Introduction to Software Security

# Chapter 3.8.3: Code Injections

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

*DRAFT — Revision 2.0, January 2022.*

## Objectives

- Briefly review the general problem of injections, and apply 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 injections vulnerabilities.

Examples are presented from Python, Perl, JavaScript and Ruby.

## Code Injection Attacks

Code Injection is a specific type of injection attack where an executable program statement is constructed involving user input at an attack surface that becomes vulnerable when it can be manipulated in an unanticipated way to invoke functionality that can be used to cause harm. See Chapter 3.8 for details on the general form of these attacks.

As with other kinds of injection attacks, the key insight to understanding the source of vulnerability involves how metacharacters can potentially be used by a clever attacker resulting in a generated statement that does things never intended by the programmer. Defense and mitigation is best done by avoiding ad hoc string construction of programmatic statements in the first place, or if you must do so thoroughly validating input and carefully handling metacharacters that might introduce unintended escaping, quotation, or statement separators to alter the intended semantics of the result.

Code injection is a risk with languages that execute or interpret script because of the ease of running a string as an executable statement or statements at runtime (commonly called "eval"). For example, popular languages that have this ability include: JavaScript, Perl, Python, and Ruby.

Code injection is potentially the most dangerous form of these attacks since it literally provides an opportunity for the attacker to have their own arbitrary code injected and subsequently executed. Nonetheless, building statements at runtime is a powerful and tantalizing technique that is hard for programmers to resist. There are situations where this technique is a reasonable choice, but as with any powerful tool it's important that you recognize the danger and fully understand the risk to make a wise decision about it and when necessary know precisely how to protect against creating a bad vulnerability.

## Perl Danger Signs

Functions prone to code injection attacks include:

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

The first form is a straightforward "eval" function where the string `$s` is executed as a Perl statement. If the attacker can influence the value of the string `$s` that opens this up to attack.

The second form looks like a regular expression substitution, but the special "ee" modifier at the end causes the expression `eval($replace)` to be executed and the result is used as the substitution string. That is, the s/// statement above is equivalent to this:

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

While the first form is more obvious, the second is easily overlooked as an opportunity for injection. This is a good example of why it is important to be fluent with all the ins and outs of languages and libraries to understand exactly how they work to code securely.

## Ruby Danger Signs

Ruby only has the straightforward form of "eval" which works much as described already — 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 crystal clear when a code injection attack might be possible by having one specific syntax.

## JavaScript Danger Signs

JavaScript's "eval" syntax is similar to Ruby. The string argument can be a JavaScript expression, a statement, or a sequence of statements.

```
eval(string)
```

## Python Danger Signs

Python has a number of functions that are prone to injection attacks:

```
exec(string)     # dynamic execution of Python code
eval(string)     # returns the value of an expression or code object
execfile(string) # reads & executes Python script from a file
input(string)    # equiv to eval(raw_input()) before Python 3000 only
compile(string)  # compile the source string into an executable object
```

The first two forms interpret the string argument of the function as a statement (`exec`) for execution, or an expression (`eval`) returning the resultant value. Either of these is a potential code injection attack if the attacker can influence the string argument value.

The next two forms are shortcuts for reading input from a file (`execfile`) and executing the text as statements, or reading one line from standard input (`input`) and evaluating that as an expression and returning the result. Note that Python 3000 broke compatibility and changed the semantics of `input` and `raw_input`.

*Note: These danger sign sections are intended to illustrate major sources of code injection attacks in some common languages, but are not meant as a rigorous enumeration to all possibilities.*

## Simple Code Injection Example in Python 2

Consider this simple Python example code that implements a basic calculator to see how code injections works in practice. The calculator computes the result of the input provided by the user.

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

The (Python 3) `input` function reads a line from standard input. The `eval` function 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 is `30 * 12 + 5` then it computes the value and outputs: `Result = 365`.

However, Python expressions can do a lot of things, for example, consider this:

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

The `__import__` function dynamically imports the module named by the string provided, so this invokes the standard `os.system` function that invokes a shell to execute the given command (`rm -rf /`) which removes the filesystem root if the process has sufficient privileges.

## Mitigating this Simple Python Example

To mitigate the code injection vulnerability of the simple code below, it is necessary to validate the input supplied by the user before calling `eval`.

```
comp = input('\nYour computation? => ')
if not comp:
```

```
  print ("No input")
else:
  if validate(comp):
    print ("Result =", eval(comp))
  else:
    print ("Error")
```

In the above `validate(comp)` we should either check that only characters belonging to a whitelist are used, or check to see if the input includes the escape character (\) before any quotes so the whole sequence of characters provided by the attacker is interpreted as a single string. In this case, a white list might contain only valid numeric characters plus arithmetic operators such as + or ×.

## A More Arcane Example in Perl

Large companies over the years often have a mix of systems written at various times in different languages but still need everything to interoperate. Here is the scenario for this more advanced example: It started a few years ago when a large team built a complex distributed system in Python. This system included a logging module, written in Python of course, for reporting interesting events. Now another team is building a new component of this distributed system, working in Perl. Everything is almost complete, but now the new component needs to use the logging facility.

In this situation there are two obvious choices:

1. Write a compatible version of the logging module in Perl that interoperates seamlessly.
2. Rewrite the new system component in Python (from Perl) just so it can use the logging module.
3. Or, surprise: none of the above!

Faced with unappealing choices 1 and 2, somebody suggested a tricky third option: have the Perl code dynamically create Python code that gets executed to call the logging module. By the way, this example is based on a real vulnerability the authors discovered reviewing real code.

Programmers love to be clever and lazy, and this approach achieves both at once. What could go wrong? Here is what the quick and dirty solution to the problem looks like written in Perl:

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

To see how this bridges the gap from Perl to Python, let's walk through the code above.

The first line calls the function ReadLogFile that reads a file of lines and filters the names, resulting in the array data containing an array of names. This provides the data to be logged, returned in the array data. To see how this code injection works, consider two input lines: first a normal line, then a malicious line that constitutes the attack.

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

The second line opens a file handle to a pipe that provides input to the Python interpreter. Strings written to this file (PH) will be parsed and executed as Python source code.

The third line writes the first line of Python: an import statement naming the existing logging component already described (named LogIt).

The fourth line iterates through the array data already read from the logging input file shown above. Each line is expected to be a key/value pair with the key name. The fifth line is where the rubber meets the road: the value $_ is sent as a string to the logging component – LogIt.Name('$_') – to be precise here, the Python statement is formed by inserting the value $_ between the two single quotes of the statement prototype as shown.

For the first normal input line (John Smith) we get the expected Python statement:

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

The Python interpreter parses this line and invokes the LogIt component Name function as planned.

However, the second line is malicious so let's look at the example string that this code provides to the Python interpreter for execution:

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

According to Python language syntax, this is several statements, shown below on separate lines:

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

The first Python statement invokes LogIt.Name with an empty string parameter (perhaps this allows the attack to avoid being noticed since nothing is logged at all). The second statement imports the standard os module and the third statement invokes os.system("evilprog") that does some unspecified malicious activities. 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.

## Mitigating this Example in Perl

Fortunately, a clever developer on the team pointed out the problem and implemented a quick fix.

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

The crucial change added in the middle of the loop just before printing the Python statement: now the value string is processed by a new Perl function `QuotePyString($_)` that applies Python syntax quoting to the input. Here is the Perl code for this new code:

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

For the first normal input line (John Smith) we still get the expected Python statement:

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

Thanks to the new Python quoting code, here is the result for the malicious attack string:

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

Now 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.

## 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’s best to avoid constructing code at runtime for execution.
- If you must execute generated code then carefully sanitize the input (though this is a riskier approach).

## 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 "proof of principle" do something obvious but harmless.
2. Modify the program from Exercise 1 to be safe for all possible inputs you can think of. Try the same attacks and that they are now foiled. Can you find any new attacks that still work? Give your code to a colleague and challenge them to attack it (without causing harm).
3. Choose a language with "danger signs" listed in this chapter, or choose another interpreted language if you prefer that is capable of parsing and 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 principle" sample that demonstrates how the code might be attacked — if successful, responsibly let the project owners know and propose necessary mitigation.