Server-Side Template Injection (SSTI)

The "Mail Merge" for Websites

Template engines are powerful server-side tools that combine static template files with dynamic data to generate final HTML pages. Think of them as sophisticated mail merge systems for web applications.

01

Template

```
<h2>Hello, {{ username }}!</h2>
```

02

Data

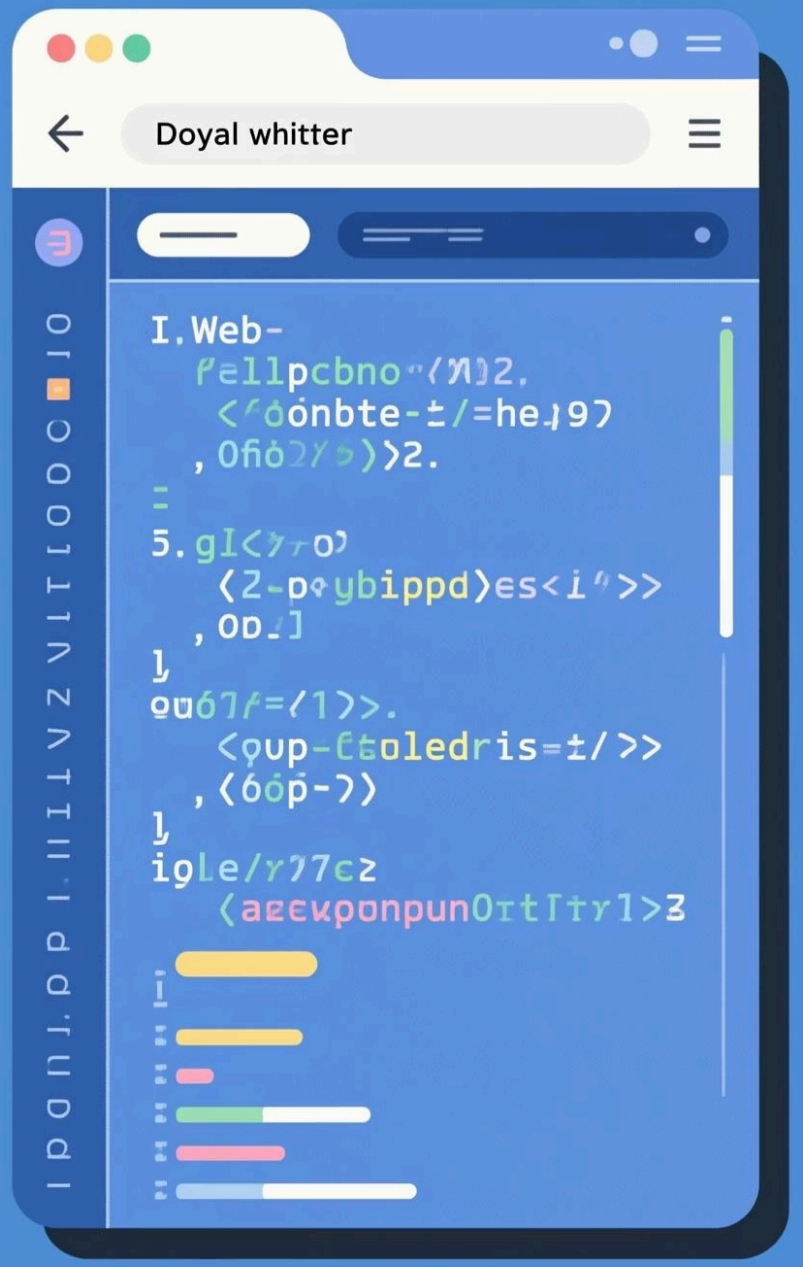
```
{"username": "Alice"}
```

03

Final HTML

```
<h2>Hello, Alice!</h2>
```

Common engines: Jinja2 (Python), Twig (PHP), FreeMarker (Java), ERB (Ruby), Handlebars (JavaScript)



What is Server-Side Template Injection?

When Data Becomes Code

SSTI occurs when an attacker injects malicious code directly into a template. The server mistakenly interprets the attacker's input as part of the template logic and executes it.

The critical flaw: The application inserts user-supplied, unsanitized data directly into the template without proper validation or escaping.

Instead of treating input as harmless data, the template engine processes it as executable code—opening the door to severe exploitation.

SSTI vs. XSS: A Critical Distinction

Understanding the difference between these vulnerabilities is crucial for assessing risk and prioritizing security efforts.

XSS (Client-Side)

- Code executes in the victim's browser
- Can steal cookies and session tokens
- Enables site defacement
- Limited to client-side impact

SSTI (Server-Side)

- Code executes on the web server
- Often leads to Remote Code Execution (RCE)
- Enables full server takeover
- Can compromise entire infrastructure

Key takeaway: SSTI is significantly more dangerous than XSS because it provides direct access to server resources and can lead to complete system compromise.

Probing for the Vulnerability

Identifying SSTI involves inserting template syntax into common input fields like search bars, forms, or URL parameters to observe how the server processes the input.

The "Polyglot" Test

Security researchers use a payload that tests multiple template engines simultaneously:

```
{{ 7 * 7 }}
```

Jinja2, Twig

```
A${7*7}
```

Mako, Velocity

```
<%= 7*7 %>
```

ERB, ASP

```
@{7*7}
```

Razor

Reading the Server's Response

Interpreting how the server handles injected payloads reveals whether a vulnerability exists and helps identify the template engine being used.

Test Scenario

A URL `.../?name=Chris` displays "Hello, Chris"

1

Test Payload

```
.../?name={{7*7}}
```

2

Vulnerable Response

Output: Hello, 49

The server executed the expression—SSTI confirmed!

3

XSS-like Response

Output: Hello, {{7*7}}

Reflected in HTML but not executed server-side

4

Safe Response

Output: Hello, {{7*7}}

Rendered as literal text—properly escaped and secure

From Injection to Remote Code Execution

More Than Just Math

Once an attacker confirms SSTI, calculating `7*7` is just the beginning. The real goal is achieving **Remote Code Execution (RCE)**—the ability to run arbitrary commands on the server.

The Attack Method

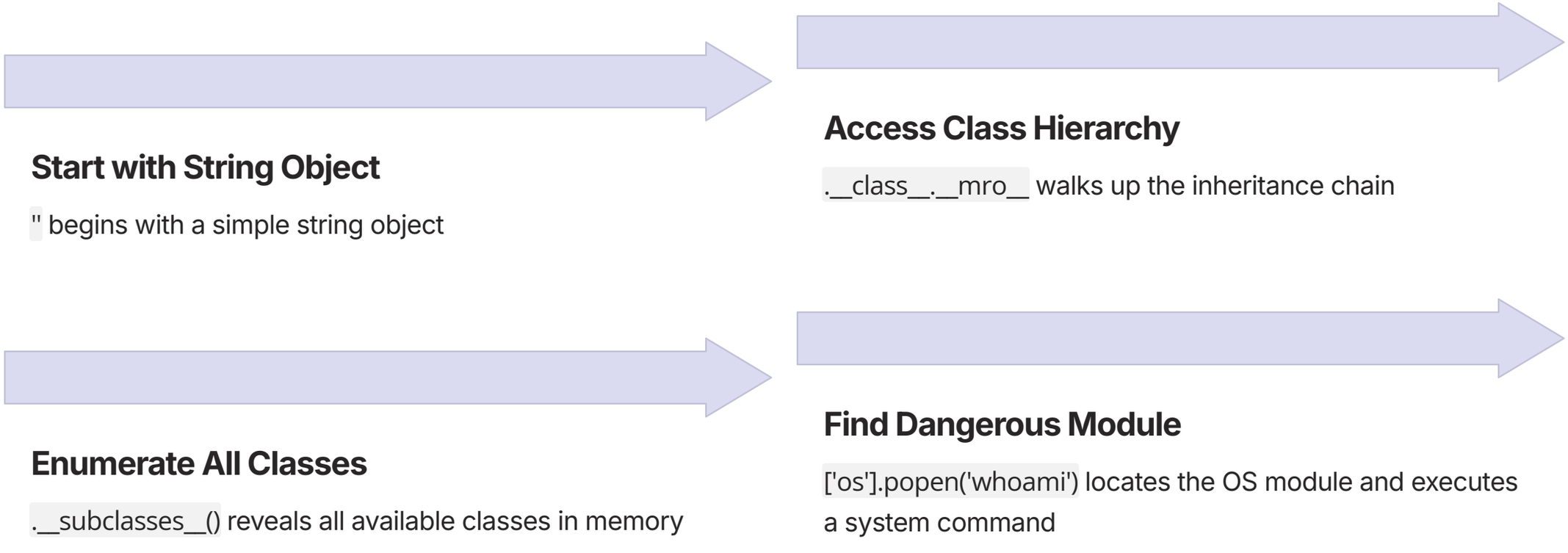
Attackers "escape" the template's sandbox by exploiting the programming language's object model. They navigate through object hierarchies to access dangerous built-in libraries like Python's `os` module or Java's `Runtime` class.

From there, they can execute system commands, read sensitive files, modify databases, or establish persistent backdoors.

"Walking the Object Tree"

Here's a conceptual example of how attackers chain together object references to achieve code execution in Python/Jinja2:

```
{{ ''.__class__.__mro__[1].__subclasses__()[...].__init__.__globals__['os'].popen('whoami').read() }}
```



Result: The page would render the output of the `whoami` command, such as "www-data" or "root"—proving complete server compromise.

Where is SSTI Found?

SSTI vulnerabilities commonly appear in applications that allow users to customize content, templates, or personalized messages.

Wikis & CMS Platforms

Content management systems that allow template editing or custom page layouts are frequent targets for SSTI attacks.

Email Marketing Tools

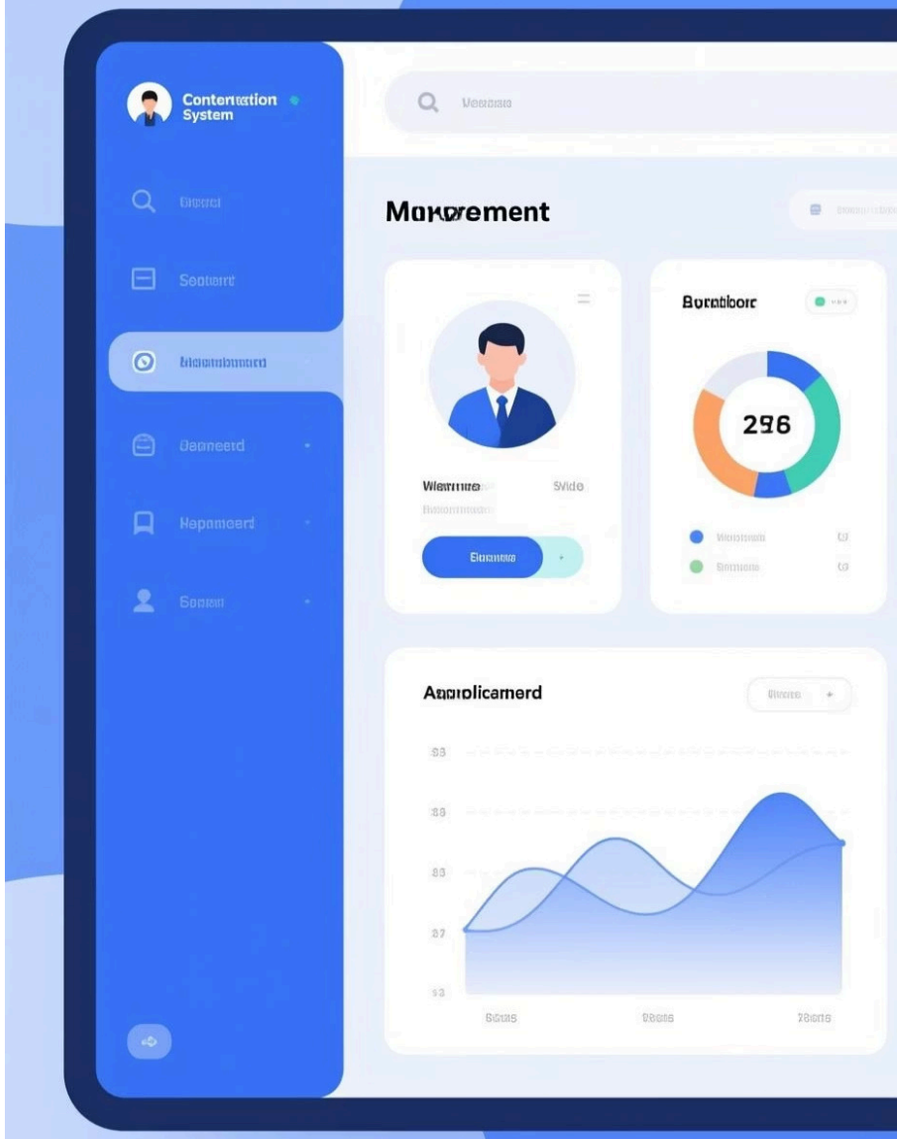
Applications enabling users to create personalized email templates using dynamic placeholders are vulnerable if input isn't sanitized.

Custom Profile Features

Web apps with customizable user profiles, signatures, or bio sections that render server-side are potential attack vectors.

Report Generators

Systems that generate dynamic reports, invoices, or documents by combining user input with templates can be exploited.



Prevention and Mitigation 🛡️

Securing your applications against SSTI requires a defense-in-depth approach combining input validation, architectural choices, and security-focused development practices.

1

Sanitize All User Input

Never pass raw user-supplied data directly to a template engine's rendering function. Always validate, escape, and treat input as untrusted data.

2

Use Sandboxing

If users must edit templates, implement a sandboxed environment that explicitly restricts access to dangerous functions and system objects.

3

Choose Secure Defaults

Modern frameworks like Django and Flask are secure by default. Vulnerabilities typically arise from manual template handling or bypassing built-in protections.

4

Separate Logic from Data

Maintain clear separation between template logic and user data. Store templates separately from user input and never allow users to modify core template files.

The Golden Rule: Always treat user input as *data*, never as executable *code*. This principle is fundamental to preventing SSTI and many other injection vulnerabilities.