**Server-side template injection**

Server-side template injection is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server. As the name suggests, server-side template injection payloads are delivered and evaluated server-side, potentially making them much more dangerous than a typical client-side template injection.

**What is the impact of server-side template injection?**

Server-side template injection vulnerabilities can expose websites to a variety of attacks depending on the template engine in question and how exactly the application uses it. In certain rare circumstances, these vulnerabilities pose no real security risk. However, most of the time, the impact of server-side template injection can be catastrophic.

At the severe end of the scale, an attacker can potentially achieve remote code execution, taking full control of the back-end server and using it to perform other attacks on internal infrastructure.

Even in cases where full remote code execution is not possible, an attacker can often still use server-side template injection as the basis for numerous other attacks, potentially gaining read access to sensitive data and arbitrary files on the server.

**How do server-side template injection vulnerabilities arise?**

Server-side template injection vulnerabilities arise when user input is concatenated into templates rather than being passed in as data.

Static templates that simply provide placeholders into which dynamic content is rendered are generally not vulnerable to server-side template injection. The classic example is an email that greets each user by their name, such as the following extract from a Twig template:

$output = $twig->render("Dear {first\_name},", array("first\_name" => $user.first\_name) );

This is not vulnerable to server-side template injection because the user's first name is merely passed into the template as data.

However, as templates are simply strings, web developers sometimes directly concatenate user input into templates prior to rendering. Let's take a similar example to the one above, but this time, users are able to customize parts of the email before it is sent. For example, they might be able to choose the name that is used:

$output = $twig->render("Dear " . $\_GET['name']);

In this example, instead of a static value being passed into the template, part of the template itself is being dynamically generated using the GET parameter name. As template syntax is evaluated server-side, this potentially allows an attacker to place a server-side template injection payload inside the name parameter as follows:

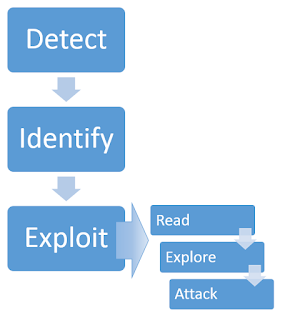
http://vulnerable-website.com/?name={{bad-stuff-here}}

Vulnerabilities like this are sometimes caused by accident due to poor template design by people unfamiliar with the security implications. Like in the example above, you may see different components, some of which contain user input, concatenated and embedded into a template. In some ways, this is similar to SQL injection vulnerabilities occurring in poorly written prepared statements.

However, sometimes this behavior is actually implemented intentionally. For example, some websites deliberately allow certain privileged users, such as content editors, to edit or submit custom templates by design. This clearly poses a huge security risk if an attacker is able to compromise an account with such privileges.

**Constructing a server-side template injection attack**

Identifying server-side template injection vulnerabilities and crafting a successful attack typically involves the following high-level process.



**Detect**

Server-side template injection vulnerabilities often go unnoticed not because they are complex but because they are only really apparent to auditors who are explicitly looking for them. If you are able to detect that a vulnerability is present, it can be surprisingly easy to exploit it. This is especially true in unsandboxed environments.

As with any vulnerability, the first step towards exploitation is being able to find it. Perhaps the simplest initial approach is to try fuzzing the template by injecting a sequence of special characters commonly used in template expressions, such as ${{<%[%'"}}%\. If an exception is raised, this indicates that the injected template syntax is potentially being interpreted by the server in some way. This is one sign that a vulnerability to server-side template injection may exist.

Server-side template injection vulnerabilities occur in two distinct contexts, each of which requires its own detection method. Regardless of the results of your fuzzing attempts, it is important to also try the following context-specific approaches. If fuzzing was inconclusive, a vulnerability may still reveal itself using one of these approaches. Even if fuzzing did suggest a template injection vulnerability, you still need to identify its context in order to exploit it.

**Plaintext context**

Most template languages allow you to freely input content either by using HTML tags directly or by using the template's native syntax, which will be rendered to HTML on the back-end before the HTTP response is sent. For example, in Freemarker, the line render('Hello ' + username) would render to something like Hello mrwebsecure.

This can sometimes be exploited for XSS and is in fact often mistaken for a simple XSS vulnerability. However, by setting mathematical operations as the value of the parameter, we can test whether this is also a potential entry point for a server-side template injection attack.

For example, consider a template that contains the following vulnerable code:

render('Hello ' + username)

During auditing, we might test for server-side template injection by requesting a URL such as:

http://vulnerable-website.com/?username=${7\*7}

If the resulting output contains Hello 49, this shows that the mathematical operation is being evaluated server-side. This is a good proof of concept for a server-side template injection vulnerability.

Note that the specific syntax required to successfully evaluate the mathematical operation will vary depending on which template engine is being used. We'll discuss this in more detail in the Identify step.

**Code context**

In other cases, the vulnerability is exposed by user input being placed within a template expression, as we saw earlier with our email example. This may take the form of a user-controllable variable name being placed inside a parameter, such as:

greeting = getQueryParameter('greeting')

engine.render("Hello {{"+greeting+"}}", data)

On the website, the resulting URL would be something like:

http://vulnerable-website.com/?greeting=data.username

This would be rendered in the output to Hello mrwebsecure, for example.

This context is easily missed during assessment because it doesn't result in obvious XSS and is almost indistinguishable from a simple hashmap lookup. One method of testing for server-side template injection in this context is to first establish that the parameter doesn't contain a direct XSS vulnerability by injecting arbitrary HTML into the value:

http://vulnerable-website.com/?greeting=data.username<tag>

In the absence of XSS, this will usually either result in a blank entry in the output (just Hello with no username), encoded tags, or an error message. The next step is to try and break out of the statement using common templating syntax and attempt to inject arbitrary HTML after it:

http://vulnerable-website.com/?greeting=data.username}}<tag>

If this again results in an error or blank output, you have either used syntax from the wrong templating language or, if no template-style syntax appears to be valid, server-side template injection is not possible. Alternatively, if the output is rendered correctly, along with the arbitrary HTML, this is a key indication that a server-side template injection vulnerability is present:

Hello mrwebsecre<tag>

**Identify**

Once you have detected the template injection potential, the next step is to identify the template engine.

Although there are a huge number of templating languages, many of them use very similar syntax that is specifically chosen not to clash with HTML characters. As a result, it can be relatively simple to create probing payloads to test which template engine is being used.

Simply submitting invalid syntax is often enough because the resulting error message will tell you exactly what the template engine is, and sometimes even which version. For example, the invalid expression <%=foobar%> triggers the following response from the Ruby-based ERB engine:

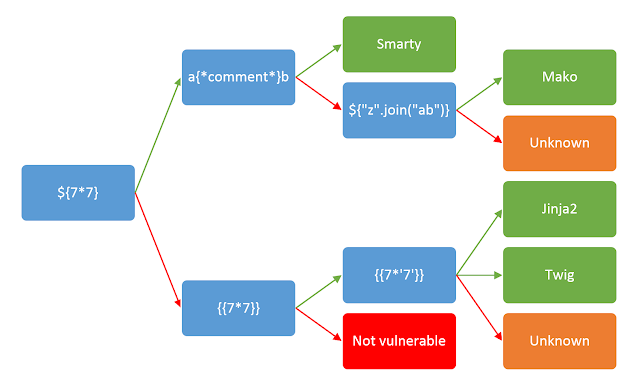
(erb):1:in `<main>': undefined local variable or method `foobar' for main:Object (NameError)

from /usr/lib/ruby/2.5.0/erb.rb:876:in `eval'

from /usr/lib/ruby/2.5.0/erb.rb:876:in `result'

from -e:4:in `<main>'

Otherwise, you'll need to manually test different language-specific payloads and study how they are interpreted by the template engine. Using a process of elimination based on which syntax appears to be valid or invalid, you can narrow down the options quicker than you might think. A common way of doing this is to inject arbitrary mathematical operations using syntax from different template engines. You can then observe whether they are successfully evaluated. To help with this process, you can use a decision tree similar to the following:



You should be aware that the same payload can sometimes return a successful response in more than one template language. For example, the payload {{7\*'7'}} returns 49 in Twig and 7777777 in Jinja2. Therefore, it is important not to jump to conclusions based on a single successful response.

**Exploit**

After detecting that a potential vulnerability exists and successfully identifying the template engine, you can begin trying to find ways of exploiting it.

**Exploiting server-side template injection vulnerabilities**

In this section, we'll look more closely at some typical server-side template injection vulnerabilities and demonstrate how they can be exploited using our high-level methodology. By putting this process into practice, you can potentially discover and exploit a variety of different server-side template injection vulnerabilities.

Once you discover a server-side template injection vulnerability, and identify the template engine being used, successful exploitation typically involves the following process.

* Read
  + Template syntax
  + Security documentation
  + Documented exploits
* Explore the environment
* Create a custom attack

**Read**

Unless you already know the template engine inside out, reading its documentation is usually the first place to start. While this may not be the most exciting way to spend your time, it is important not to underestimate what a useful source of information the documentation can be.

**Learn the basic template syntax**

Learning the basic syntax is obviously important, along with key functions and handling of variables. Even something as simple as learning how to embed native code blocks in the template can sometimes quickly lead to an exploit. For example, once you know that the Python-based Mako template engine is being used, achieving remote code execution could be as simple as:

<%

import os

x=os.popen('id').read()

%>

${x}

In an unsandboxed environment, achieving remote code execution and using it to read, edit, or delete arbitrary files is similarly as simple in many common template engines.

**Read about the security implications**

In addition to providing the fundamentals of how to create and use templates, the documentation may also provide some sort of "Security" section. The name of this section will vary, but it will usually outline all the potentially dangerous things that people should avoid doing with the template. This can be an invaluable resource, even acting as a kind of cheat sheet for which behaviors you should look for during auditing, as well as how to exploit them.

Even if there is no dedicated "Security" section, if a particular built-in object or function can pose a security risk, there is almost always a warning of some kind in the documentation. The warning may not provide much detail, but at the very least it should flag this particular built-in as something to investigate.

For example, in ERB, the documentation reveals that you can list all directories and then read arbitrary files as follows:

<%= Dir.entries('/') %>

<%= File.open('/example/arbitrary-file').read %>

**Look for known exploits**

Another key aspect of exploiting server-side template injection vulnerabilities is being good at finding additional resources online. Once you are able to identify the template engine being used, you should browse the web for any vulnerabilities that others may have already discovered. Due to the widespread use of some of the major template engines, it is sometimes possible to find well-documented exploits that you might be able to tweak to exploit your own target website.

**Explore**

At this point, you might have already stumbled across a workable exploit using the documentation. If not, the next step is to explore the environment and try to discover all the objects to which you have access.

Many template engines expose a "self" or "environment" object of some kind, which acts like a namespace containing all objects, methods, and attributes that are supported by the template engine. If such an object exists, you can potentially use it to generate a list of objects that are in scope. For example, in Java-based templating languages, you can sometimes list all variables in the environment using the following injection:

${T(java.lang.System).getenv()}

This can form the basis for creating a shortlist of potentially interesting objects and methods to investigate further. Additionally, for Burp Suite Professional users, the Intruder provides a built-in wordlist for brute-forcing variable names.

**Developer-supplied objects**

It is important to note that websites will contain both built-in objects provided by the template and custom, site-specific objects that have been supplied by the web developer. You should pay particular attention to these non-standard objects because they are especially likely to contain sensitive information or exploitable methods. As these objects can vary between different templates within the same website, be aware that you might need to study an object's behavior in the context of each distinct template before you find a way to exploit it.

While server-side template injection can potentially lead to remote code execution and full takeover of the server, in practice this is not always possible to achieve. However, just because you have ruled out remote code execution, that doesn't necessarily mean there is no potential for a different kind of exploit. You can still leverage server-side template injection vulnerabilities for other high-severity exploits, such as file path traversal, to gain access to sensitive data.

**Create a custom attack**

So far, we've looked primarily at constructing an attack either by reusing a documented exploit or by using well-known vulnerabilities in a template engine. However, sometimes you will need to construct a custom exploit. For example, you might find that the template engine executes templates inside a sandbox, which can make exploitation difficult, or even impossible.

After identifying the attack surface, if there is no obvious way to exploit the vulnerability, you should proceed with traditional auditing techniques by reviewing each function for exploitable behavior. By working methodically through this process, you may sometimes be able to construct a complex attack that is even able to exploit more secure targets.

**Constructing a custom exploit using an object chain**

As described above, the first step is to identify objects and methods to which you have access. Some of the objects may immediately jump out as interesting. By combining your own knowledge and the information provided in the documentation, you should be able to put together a shortlist of objects that you want to investigate more thoroughly.

When studying the documentation for objects, pay particular attention to which methods these objects grant access to, as well as which objects they return. By drilling down into the documentation, you can discover combinations of objects and methods that you can chain together. Chaining together the right objects and methods sometimes allows you to gain access to dangerous functionality and sensitive data that initially appears out of reach.

For example, in the Java-based template engine Velocity, you have access to a ClassTool object called $class. Studying the documentation reveals that you can chain the $class.inspect() method and $class.type property to obtain references to arbitrary objects. In the past, this has been exploited to execute shell commands on the target system as follows:

$class.inspect("java.lang.Runtime").type.getRuntime().exec("bad-stuff-here")

**Constructing a custom exploit using developer-supplied objects**

Some template engines run in a secure, locked-down environment by default in order to mitigate the associated risks as much as possible. Although this makes it difficult to exploit such templates for remote code execution, developer-created objects that are exposed to the template can offer a further, less battle-hardened attack surface.

However, while substantial documentation is usually provided for template built-ins, site-specific objects are almost certainly not documented at all. Therefore, working out how to exploit them will require you to investigate the website's behavior manually to identify the attack surface and construct your own custom exploit accordingly.

**How to prevent server-side template injection vulnerabilities**

The best way to prevent server-side template injection is to not allow any users to modify or submit new templates. However, this is sometimes unavoidable due to business requirements.

One of the simplest ways to avoid introducing server-side template injection vulnerabilities is to always use a "logic-less" template engine, such as Mustache, unless absolutely necessary. Separating the logic from presentation as much as possible can greatly reduce your exposure to the most dangerous template-based attacks.

Another measure is to only execute users' code in a sandboxed environment where potentially dangerous modules and functions have been removed altogether. Unfortunately, sandboxing untrusted code is inherently difficult and prone to bypasses.

Finally, another complementary approach is to accept that arbitrary code execution is all but inevitable and apply your own sandboxing by deploying your template environment in a locked-down Docker container, for example.