WHITE BOX PENTESTING *AND* EXPLOIT DEVELOPMENT

Finding vulnerabilities from source code review and creating scripts /o/

Twitter: https://twitter.com/trouble1_raunak
Github: https://github.com/TROUBLE-1

WHOAMI?

- Ty-bcom student
- Doing AWAE (OSWE)

Twitter: https://twitter.com/trouble1_raunak
Github: https://github.com/TROUBLE-1

WHAT WE ARE GOING TO LEARN?

Doing a source code review to exploit vulnerabilities like:

- Type juggling
- Advance 2nd order SQL Injection
- Pass-the-hash
- {{SSTI}} (Server Side Template Injection)

At the end creating a python script to automate the attack

TYPE JUGGLING

TYPE JUGGLING

It's about comparison

- String to String
- Integer to Integer
- Array to Array
- String to Integer
- String to float
- Integer to Array
- Integer to float

TYPE JUGGLING

LOOSE COMPARISON

==

STRICT COMPARISON

===

TYPE JUGGLING

```
4 == 4   // True
3 == "3"   // True
77.5 == "77.5000"   // True
"I am a string but " == 0   // True
"7 is not my roll no" == 7   // True
```

LOOSE COMPARISON

==

TYPE JUGGLING

```
3 === "3"    // False
77.5 === "77.5000"   // False
"I am a string but " === 0   // False
"7 is my roll no" === 7    // False
```

STRICT COMPARISON

===

TYPE JUGGLING

Few important comparison

```
"03456346" == 03456346 // False
"0e3456346" == 0 // True
"0a235235" == 0 // True
0e3425235 == 0 // True
"0e2367556" == "0" // True
{} == 0 // True
[] == 0 // False
```

TYPE JUGGLING

Any Question relating this ?

TYPE JUGGLING

Time for source code review
/o/

2ND ORDER SQL INJECTION

2ND ORDER SQL INJECTION

MYTH:

Escaping single quotes in a string based user input used for database transactions will prevent SQL injections

2ND ORDER SQL INJECTION

User inputs are sanitized and saved into database like adding backward slash \', \#, … etc and saved into database.

Then some another function call that user input which is saved into databses without sanitizing to create a *SQL QUERY*

2ND ORDER SQL INJECTION

For example,

user_input is: a') or 1=1 #

Sql Query: insert into student(username) values('a\'\) or 1=1\#')

Sql Query for search: search * from student where username like ('%a') or 1=1 #%')

This is how 2nd order sql injection works

2ND ORDER SQL INJECTION

Now you would really like to dig in more for sql injection's during pentesting

PASS-THE-HASH

PASS-THE-HASH

Doing sql injection and getting password hashes is cool but what if you are not able to decode it :(

You can't be able to login into the admin's account

Which makes you to think like:

Wish I could be able to login using hash 🤔

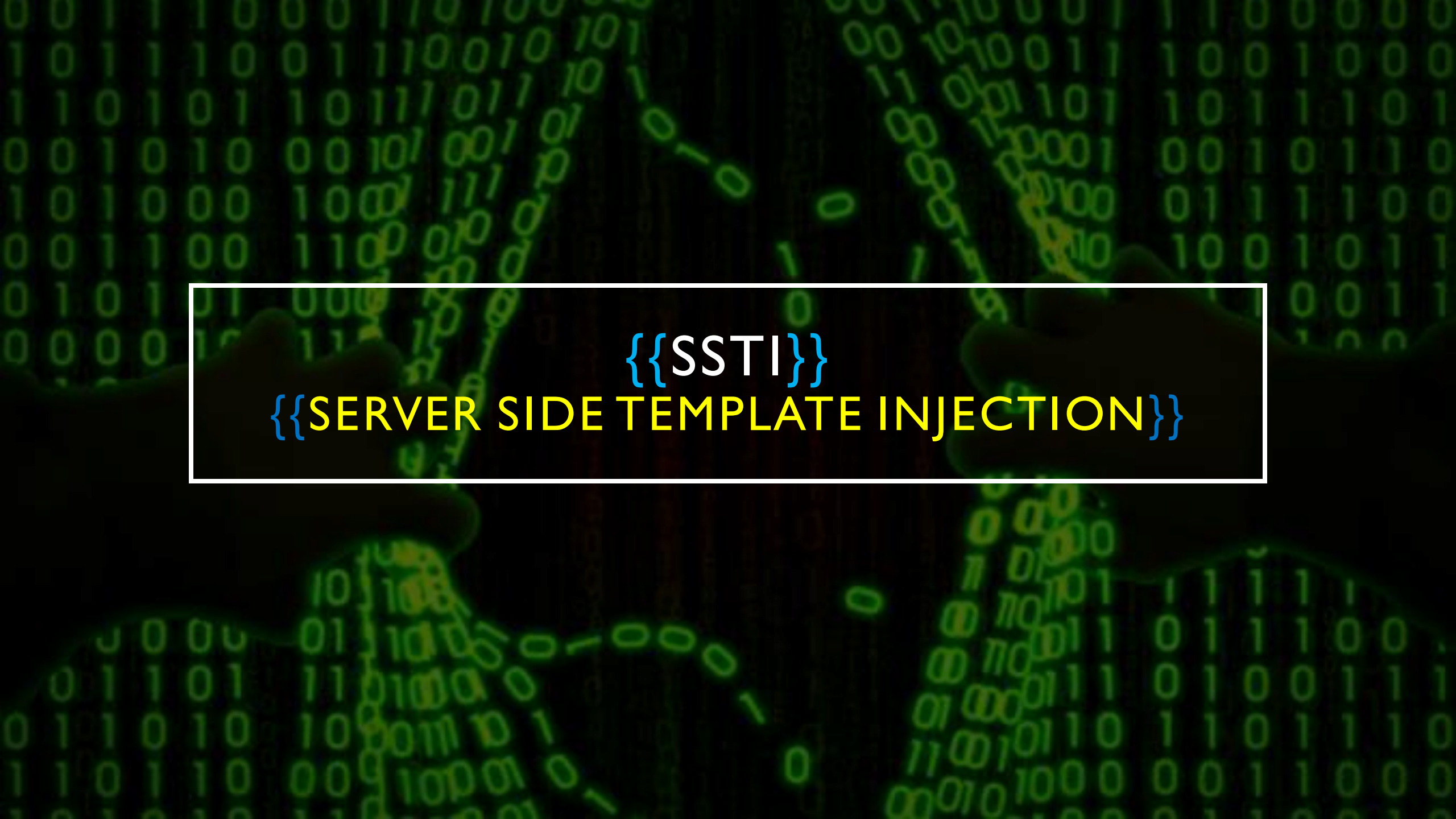
PASS-THE-HASH

Pass-the-hash is a vulnerability where an attacker is able to login using a hash that is without decrypting it.

Username: admin
Password: 69DFCD36E2787D59967299C981C3FCFBE3AA3A96

PASS-THE-HASH

Will see this vulnerability in dept later on during live demo :)


{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

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

But a malicious user input can lead to *server side template injection*

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

Template Injection can be used to directly attack web servers internals and often obtain Remote code execution {{RCE}}

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

Template engines likes *Twig* and *FreeMaker* to embed dynamic content.

Lets see the coding stuff

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

Consider a marketing application that sends bulk emails, and uses a Twig template to greet recipients by name.

```
$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) );
```

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

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

This takes user input from a get method which means users can execute template injection

hints for identifying server-side vulnerability:

custom_email={{7*7}}

Hello 49

custom_email=username}}<tag>

Hello username}}

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

The first step after finding template injection and identifying the template engine is to read the documentation

{{SSTI}}
{{SERVER SIDE TEMPLATE INJECTION}}

IN TWIG FRAMEWORK

{{_self.env.registerUndefinedFilterCallback("exec")}}

{{_self.env.getFilter("id")}}

Hello uid=33(www-data) gid=33(www-data) groups=33(www-data)

Payload for reverse shell in twig framework /o/

Reading documentation helps to get the malicious function.

TIME FOR SOURCE CODE REVIEW

Let's dig into the code and find some vulnerabilities

And chaining all the *vulnerabilites* into a one *python script* to *automate the exploit* :))

REFERENCES LINKS

- https://www.owasp.org/images/6/6b/PHPMagicTricks-TypeJuggling.pdf
- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Type%20Juggling
- https://www.esecforte.com/second-order-sql-injection/
- https://portswigger.net/research/server-side-template-injection

MY LABS

- https://github.com/TROUBLE-1/Type-juggling

THANK YOU

Twitter: https://twitter.com/trouble1_raunak
Github: https://github.com/TROUBLE-1