INJECTING SECURITY INTO WEB APPS AT RUNTIME

AJIN ABRAHAM
SECURITY ENGINEER
IMMUNIO
#WHOAMI

- Security Engineering @ Immunio
- Research on Runtime Application Self Defence
- Authored MobSF, Xenotix and NodeJSScan
- Teach Security: opsecx.com
- Runs: opensecurity.in
AGENDA: WHAT THE TALK IS ABOUT?

RASP

WHAT THE TALK IS NOT ABOUT?

WAF
APPSEC CHALLENGES

- Writing Secure Code is not Easy
- Most follows agile development strategies
- Frequent releases and builds
- Any release can introduce or reintroduce vulnerabilities
- Problems by design.
  Ex: Session Hijacking, Credential Stuffing
STATE OF WEB FRAMEWORK SECURITY

- Automatic CSRF Token - Anti CSRF
- Templates escapes User Input - No XSS
- Uses ORM - No SQLi

You need to use secure APIs or write Code to enable some of these

Security Bugs happens when people write bad code.
STATE OF WEB FRAMEWORK SECURITY

- Anti CSRF - Can easily be turned off/miss configurations
- Templates escapes User Input - Just HTML Escape -> XSS
  - https://jsfiddle.net/1c4f271c/
- Uses ORM - SQLi is still possible
  - http://rails-sqli.org/
STATE OF WEB FRAMEWORK SECURITY

- Remote OS Command Execution - No
- Remote Code Injection - No
- Server Side Template Injection RCE - No
- Session Hijacking - No
- Verb Tampering - No
- File Upload Restriction - No

The list goes on.....
WE NEED TO PREVENT EXPLOITATION

LET’S USE WAF
CAN A WAF SOLVE THIS?

- First WAF AppShield in 1999, almost 18 years of existence
- Quick question: How many of you run a WAF in defence/protection mode?
- Most organisations use them, but in monitor mode due to high rate false positives.
- Most WAFs use BLACKLISTS

[Pie chart showing False Negatives (20%), False Positives (10%), and Detection (70%)]

False Negatives   False Positive Detection
APPLICATION SECURITY RULE OF THUMB

HOW I FEEL...

WHEN THE BLACKLIST IS ON

Gets bypassed, today or tomorrow
WHAT WAF SEES?

ATTACK != VULNERABILITY
HOW WAF WORKS

- The strength of WAF is the **blacklist**
- They detect **Attacks** not **Vulnerability**
- WAF has no application context
- Doesn’t know if a vulnerability got exploited inside the app server or not.
**WAF PROBLEMS**

- How long they keep on building the black lists?

- WAFs used to downgrade your security.
  - No Perfect Forward Secrecy
  - Can’t Support elliptic curves like ECDHE
  - Some started to support with a Reverse Proxy

- Organisations are moving to PFS (Heartbleed bug)

- SSL Decryption and Re-encryption Overhead
TLS 1.3 COMING SOON . . . .

- Cipher Suites (19 suites)
  - Cipher Suite: Unknown (0xdada)
  - Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
  - Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
  - Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)

- Extension: signature_algorithms (len=20)
  - Type: signature_algorithms (13)
  - Length: 20
  - Signature Hash Algorithms Length: 18
  - Signature Hash Algorithms (9 algorithms)

- Extension: elliptic_curves (len=10)
  - Type: elliptic_curves (10)
  - Length: 10
  - Elliptic Curves Length: 8
  - Elliptic curves (4 curves)
    - Elliptic curve: Unknown (0x9a9a)
    - Elliptic curve: ecdh_x25519 (0x001d)
    - Elliptic curve: secp256r1 (0x0017)
    - Elliptic curve: secp384r1 (0x0018)
SO WHAT’S THE IDEAL PLACE FOR SECURITY?
We can do much better.

It’s time to evolve

WAF -> SAST -> DAST -> IAST -> RASP

Attack Detection & Prevention
Vulnerability Detection
Precise Vulnerability Detection
Attack Detection & Prevention/Neutralization + Precise Vulnerability Detection + Extras
RUNTIME APPLICATION SELF DEFENCE

- Detect both Attacks and Vulnerability
- Zero Code Modification and Easy Integration
- No Hardware Requirements
- Apply defence inside the application
- Have Code Level insights
- Fewer False positives
- Inject Security at Runtime
- No use of Blacklists
TYPES OF RASP

- Pattern Matching with Blacklist - Old wine in new bottle (Fancy WAF)
- Dynamic Tainting - Good but Performance over head
- Virtualisation and Compartmentalisation - Good, but Less Precise, Container oriented and not application oriented, Platform Specific (JVM)
- Code Instrumentation and Dynamic Whitelist - Good, but specific to Frameworks, Developer deployed
FOCUS OF RESEARCH

▸ RASP by API Instrumentation and Dynamic Whitelist
▸ Securing a vulnerable Python Tornado app with Zero Code change.
▸ Code Injection Vulnerabilities
  ▸ Preventing SQLi
  ▸ Preventing RCE
  ▸ Preventing Stored & Reflected XSS
  ▸ Preventing DOM XSS
▸ Other AppSec Challenges
  ▸ Preventing Header Injection
  ▸ File Upload Protection
▸ Ongoing Research
  ▸ Preventing Session Hijacking
  ▸ Preventing Layer 7 DDoS
  ▸ Credential Stuffing
RASP BY API INSTRUMENTATION AND DYNAMIC WHITELIST

- MONKEY PATCHING
- LEXICAL ANALYSIS
- CONTEXT DETERMINATION
MONKEY PATCHING

- Also known as Runtime Hooking and Patching of functions/methods.

- [https://jsfiddle.net/h1gves49/2/](https://jsfiddle.net/h1gves49/2/)
A lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Lexical analyzer generates error if it sees an invalid token.
Lexical Analysis and Token Generation

**Input:** `int value = 100; // value is 100`

**Normal Lexer**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>KEYWORD</code></td>
</tr>
<tr>
<td><code>value</code></td>
<td><code>IDENTIFIER</code></td>
</tr>
<tr>
<td><code>=</code></td>
<td><code>OPERATOR</code></td>
</tr>
<tr>
<td><code>100</code></td>
<td><code>CONSTANT</code></td>
</tr>
<tr>
<td><code>;</code></td>
<td><code>SYMBOL</code></td>
</tr>
</tbody>
</table>

**Custom Lexer**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>KEYWORD</code></td>
</tr>
<tr>
<td><code>value</code></td>
<td><code>IDENTIFIER</code></td>
</tr>
<tr>
<td><code>=</code></td>
<td><code>OPERATOR</code></td>
</tr>
<tr>
<td><code>100</code></td>
<td><code>CONSTANT</code></td>
</tr>
<tr>
<td><code>;</code></td>
<td><code>SYMBOL</code></td>
</tr>
<tr>
<td><code>//value is 100</code></td>
<td><code>COMMENT</code></td>
</tr>
</tbody>
</table>
CONTEXT DETERMINATION

HTML CODE

```html
<html>
  <head>
    Heading
  </head>
  <body id="iden">
    <h1>Hello</h1>
    <iframe src="http://google.com"></iframe>
  </body>
</html>
```

DOM TREE

```
<html>
  <head>
    Heading
  </head>
  <body>
    <h1>Hello</h1>
    <iframe @src="http://google.com"></iframe>
  </body>
</html>
```
PREVENTING CODE INJECTION VULNERABILITIES

Interpreter cannot distinguish between Code and Data

Solve that and you solve the code injection problems
PREVENTING CODE INJECTION VULNERABILITIES

▶ Preventing SQL Injection
▶ Preventing Remote OS Command Execution
▶ Preventing Stored & Reflected Cross Site Scripting
▶ Preventing DOM XSS
SQL INJECTION

SELECT * FROM <user_input>
SQL INJECTION : HOOK

SQL Execution API

cursor.execute('SELECT * FROM logs')
### SQL INJECTION : LEARN

**SELECT**  *  **FROM**  **logs**

<table>
<thead>
<tr>
<th>SYNTAX</th>
<th>TOKEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>KEYWORD</td>
</tr>
<tr>
<td></td>
<td>WHITESPACE</td>
</tr>
<tr>
<td>*</td>
<td>OPERATOR</td>
</tr>
<tr>
<td></td>
<td>WHITESPACE</td>
</tr>
<tr>
<td>FROM</td>
<td>KEYWORD</td>
</tr>
<tr>
<td></td>
<td>WHITESPACE</td>
</tr>
<tr>
<td>logs</td>
<td>STRING</td>
</tr>
</tbody>
</table>
**SQL INJECTION : PROTECT**

```
SELECT * FROM logs AND DROP TABLE admin
```
SQL INJECTION : PROTECT

Rule for Context: SELECT * FROM <user_input>

KEYWORD WHITESPACE OPERATOR WHITESPACE KEYWORD WHITESPACE STRING

SELECT * FROM logs

SELECT * FROM history

SELECT * FROM logs AND DROP TABLE admin

KEYWORD WHITESPACE OPERATOR WHITESPACE KEYWORD WHITESPACE STRING
WHITESPACE KEYWORD WHITESPACE KEYWORD WHITESPACE STRING
DEMO
REMOTE OS COMMAND INJECTION

`ping -c 3 <user input>`
REMOTE OS COMMAND INJECTION : HOOK

Command Execution API

`os.system('ping -c 3 127.0.0.1')`
**REMOTE OS COMMAND INJECTION : LEARN**

```
ping -c 3 127.0.0.1
```

<table>
<thead>
<tr>
<th>SYNTAX</th>
<th>TOKEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ping</td>
<td>EXECUTABLE</td>
</tr>
<tr>
<td>-c</td>
<td>ARGUMENT_DASH</td>
</tr>
<tr>
<td>3</td>
<td>NUMBER</td>
</tr>
<tr>
<td>127.0.0.1</td>
<td>IP_OR_DOMAIN</td>
</tr>
</tbody>
</table>
# Remote OS Command Injection: Protect

The command `ping -c 3 127.0.0.1 & cat /etc/passwd` can be parsed into tokens as follows:

<table>
<thead>
<tr>
<th>SYNTAX</th>
<th>TOKEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ping</td>
<td>EXECUTABLE</td>
</tr>
<tr>
<td>-c</td>
<td>ARGUMENT_DASH</td>
</tr>
<tr>
<td>3</td>
<td>NUMBER</td>
</tr>
<tr>
<td>127.0.0.1</td>
<td>IP_OR_DOMAIN</td>
</tr>
<tr>
<td>&amp;</td>
<td>SPLITTER</td>
</tr>
<tr>
<td>cat</td>
<td>EXECUTABLE</td>
</tr>
<tr>
<td>/etc/passwd</td>
<td>UNIX_PATH</td>
</tr>
</tbody>
</table>
REMOTE OS COMMAND INJECTION : PROTECT

Rule for Context: ping -c 3 <user_input>

EXECUTABLE WHITESPACE ARGUMENT_DASH WHITESPACE NUMBER WHITESPACE IP_ORDOMAIN

ping -c 3 127.0.0.1
ping -c 3 google.com

ping -c 3 127.0.0.1 & cat /etc/passwd

EXECUTABLE WHITESPACE ARGUMENT_DASH WHITESPACE NUMBER WHITESPACE IP_ORDOMAIN
WHITESPACE SPLITTER WHITESPACE EXECUTABLE WHITESPACE UNIX_PATH
DEMO
CROSS SITE SCRIPTING

<body><h1>hello {{user_input1}} </h1></body>
<script> var x='{{user_input2}}';</script>
CROSS SITE SCRIPTING : HOOK

Template Rendering API

template.render("<body><h1>hello {{user_input1}}</h1></body><script> var x='{{user_input2}}';</script>", user_input1, user_input2)
CROSS SITE SCRIPTING : CONTEXT DETERMINATION

```html
<body>
<h1>hello {{user_input1}}</h1>
</body>
<script>
var x='{{user_input2}}';
</script>
```

**Parsing the DOM Tree**

```
<table>
<thead>
<tr>
<th>HTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
</tr>
<tr>
<td>BODY</td>
</tr>
<tr>
<td>#text:</td>
</tr>
<tr>
<td>H1</td>
</tr>
<tr>
<td>#text: hello {{user_input1}}</td>
</tr>
<tr>
<td>#text:</td>
</tr>
<tr>
<td>SCRIPT</td>
</tr>
<tr>
<td>#text: var x='{{user_input2}}';</td>
</tr>
</tbody>
</table>
```

**HTML_CONTEXT**

**JAVASCRIPT_VALUE_CONTEXT**
CROSS SITE SCRIPTING : PROTECT

```html
<body><h1>hello {{user_input1}} </h1></body>
<script> var x='{{user_input2}}';</script>
<body><h1>hello World</h1></body>
<script> var x='Hello World';</script>
```

user_input1 = “World”
user_input2 = “Hello World”
**CROSS SITE SCRIPTING : PROTECT**

```javascript
user_input1 = "\"<script>alert(0)</script>\"
user_input2 = "\'`;alert(0);//</script>""
```

```html
<body><h1>hello &lt;script&gt;alert(0)&lt;/script&gt; \</h1></body>
<script> var x='\'`;alert(0);//\x3C/script\x3E';</script>
```
PREVENTING DOM XSS

- Inject Security into JavaScript Frameworks
- Common JavaScript Frameworks - jQuery, AngularJS, MustacheJS etc...
- DOMPurify - https://github.com/cure53/DOMPurify
- jPurify - https://github.com/cure53/jPurify

```javascript
jQuery.fn.unsafeHtml = jQuery.fn.html;
jQuery.fn.html = function() {
    var args = Array.prototype.slice.call(arguments);
    if (args && args[0]) {
        args = sanitize(args, 0);
    }
    return jQuery.fn.unsafeHtml.apply(this, args);
};
```

https://jsfiddle.net/vno23woL/3/
OTHER APPSEC CHALLENGES

▸ Preventing Header Injection
▸ File Upload Protection
▸ Preventing Path Traversal
PREVENTING HEADER INJECTION

- Unlike WAF we don’t have to keep a blacklist of every possible encoded combination of "\%0a" and "\%0d"

- **Hook** HTTP Request API

- **Look** for "\%0a, \%0d" in HTTP Request Headers

- **Block** if Present
FILE UPLOAD PROTECTION

- Classic File Upload Bypass
  `image.jpg.php, image.php3` etc.

- **Hook** File/IO API:
  ```
  io.open("/tmp/nice.jpg", 'wb')
  ```

- **Learn** file extensions to create a whitelist.

- **Block** any unknown file extensions
  ```
  io.open("/tmp/nice.py", 'wb')
  ```

DEMO
PREVENTING PATH TRAVERSAL

- WAF Looks for

```plaintext
/../{FILE}
/.../{FILE}
/..%2f..%2f{FILE}
/%2e%2e/%2e%2e/{FILE}
/..%252f..%252f..%252f{FILE}
/%252e%252e/%252e%252e/{FILE}
/%252e%252e%252e%252e%252e%252f{FILE}
/.../.../.../.../.../{FILE}
/%255c...%255c...%255c...%255c{FILE}
/%2e%2e%2e%2e%2e%2e%2e%2e%2e%2e%2e%2e%2e%2e/{FILE}
/%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c{FILE}
```
PREVENTING PATH TRAVERSAL

- **Hook** File/IO API:
  
  ```python
  io.open("/read_dir/index.txt", 'rb')
  ```

- **Learn** directories and file extensions

- **Block** any unknown directories and file extensions
  
  ```python
  io.open("/read_dir/../../etc/passwd", 'rb')
  ```

DEMO
ON GOING RESEARCH

▸ Preventing Session Hijacking
▸ Preventing Layer 7 DDoS
▸ Credential Stuffing
THE RASP ADVANTAGES

- Accurate and Precise in Vulnerability Detection & Prevention
- Code Level Insight (Line no, Stack trace)
- Not based on Heuristics - Zero/Negligible False Positives
- No SSL Decryption and Re-encryption overhead
- Doesn’t Downgrade your Security
- Preemptive security - Zero Day protection
- Zero Code Change and easy integration

```python
pip install rasp_module
import rasp_module
```
BIGGEST ADVANTAGE

Now you can deploy it on protection mode
CHARACTERISTICS OF AN IDEAL RASP

- Ideal RASP should have minimal Performance impact
- Should not introduce vulnerabilities
- Must not consume PII of users
- Should not learn the bad stuff
- Should be a “real RASP” not a fancy WAF with Blacklist.
- Minimal Configuration and Easy deployment
THAT’S ALL FOLKS!

- Thanks to
  - Zaid Al Hamami, Mike Milner, Steve Williams, Oliver Lavery (Team IMMUNIO inc).
  - Kamaiah, Francis, Bharadwaj, Surendar, Sinu, Vivek (Team Yodlee Security Office - YSO)

- Due Credits
  - Graphics/Image Owners

@ajinabraham
ajin25@gmail.com