INJECTING SECURITY INTO WEB APPS WITH RUNTIME PATCHING AND CONTEXT LEARNING

AJIN ABRAHAM

SECURITY ENGINEER

IMMUNIO

#WHOAMI

- Security Engineering @ IMMUNIO
- Research on Runtime Application Self Defence
- Authored MobSF, Xenotix and NodeJSScan
- Teach Security: https://opsecx.com
- Blog: http://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

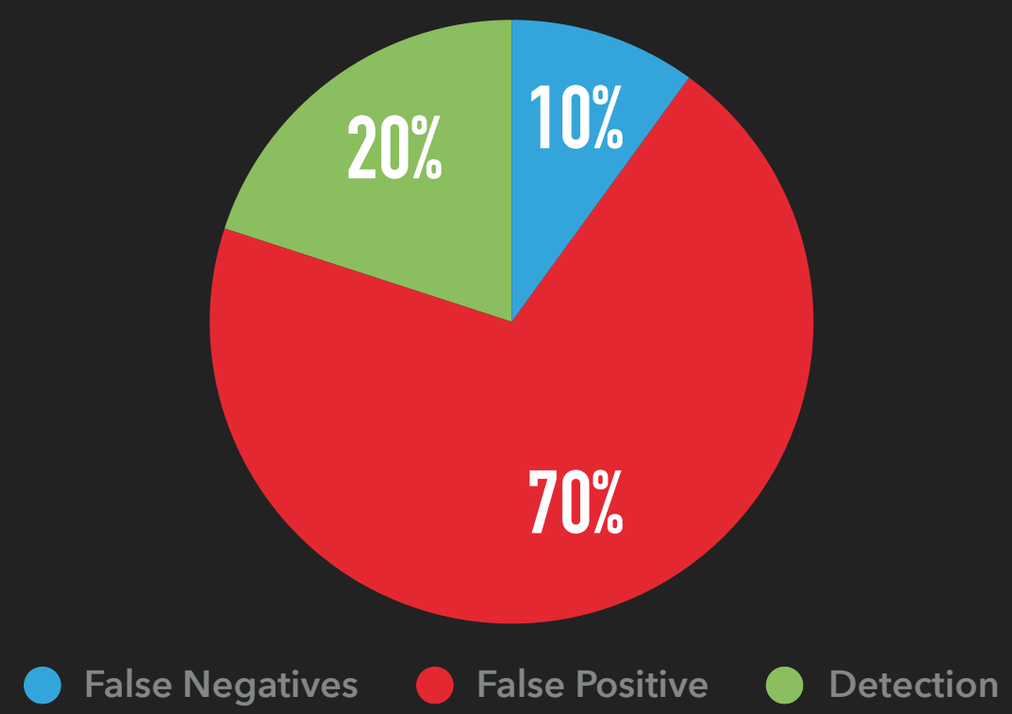
The list goes on.....

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 high rate false positives.
- Most WAFs use **BLACKLISTS**



APPLICATION SECURITY RULE OF THUMB

Gets bypassed, today or tomorrow

WHAT WAF SEES?

Request

Raw | Params | Headers | Hex

```
GET / HTTP/1.1
Host: opensecurity.in
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10_12_3) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.87 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,imag
e/webp,*/*;q=0.8
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6
Cookie:
__cfduid=d8a9396a6e5615f89a1777aacb30636a51467679500;
PHPSESSID=o92v6idjvrougfqll0dqiev663;
Connection: close
```

Response

Raw | Headers | Hex | HTML | Render

```
HTTP/1.1 200 OK
Date: Sun, 26 Feb 2017 09:29:32 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.14
Vary: Accept-Encoding,Cookie
Cache-Control: max-age=3, must-revalidate
WP-Super-Cache: Served supercache file from PHP
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Length: 55173

<!DOCTYPE html>
<html class="no-js" lang="en-US">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.
  <link rel="profile" href="http://gmpg.org/xfn/11">
  <link rel="pingback" href="http://opensecurity.in/xmlrpc.php">

  <title>OpenSecurity &#8211; In God we trust, rest we test.</title>
```

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.

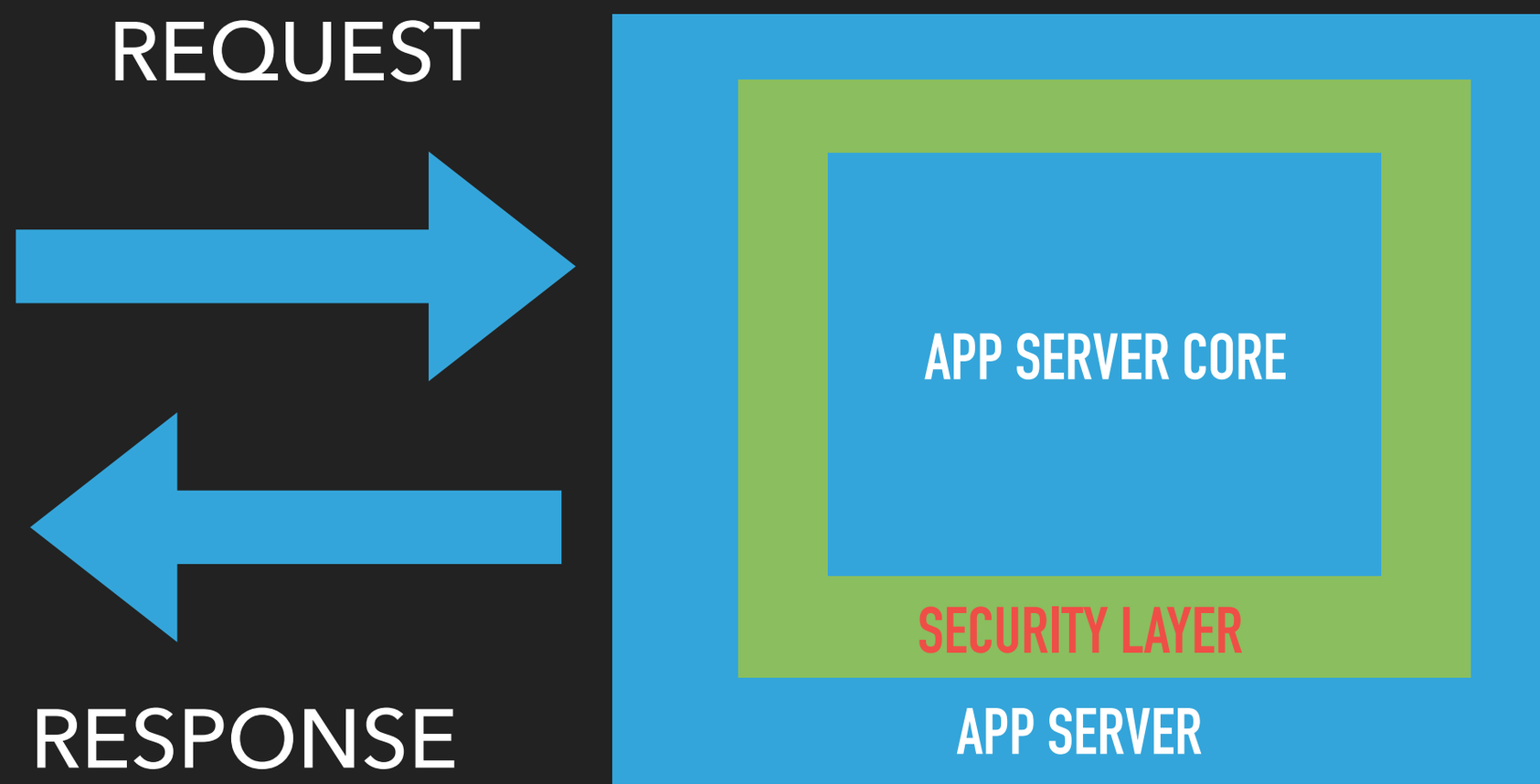
WAF PROBLEMS

- 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 - >

Attack Detection
&
Prevention

SAST -> DAST ->

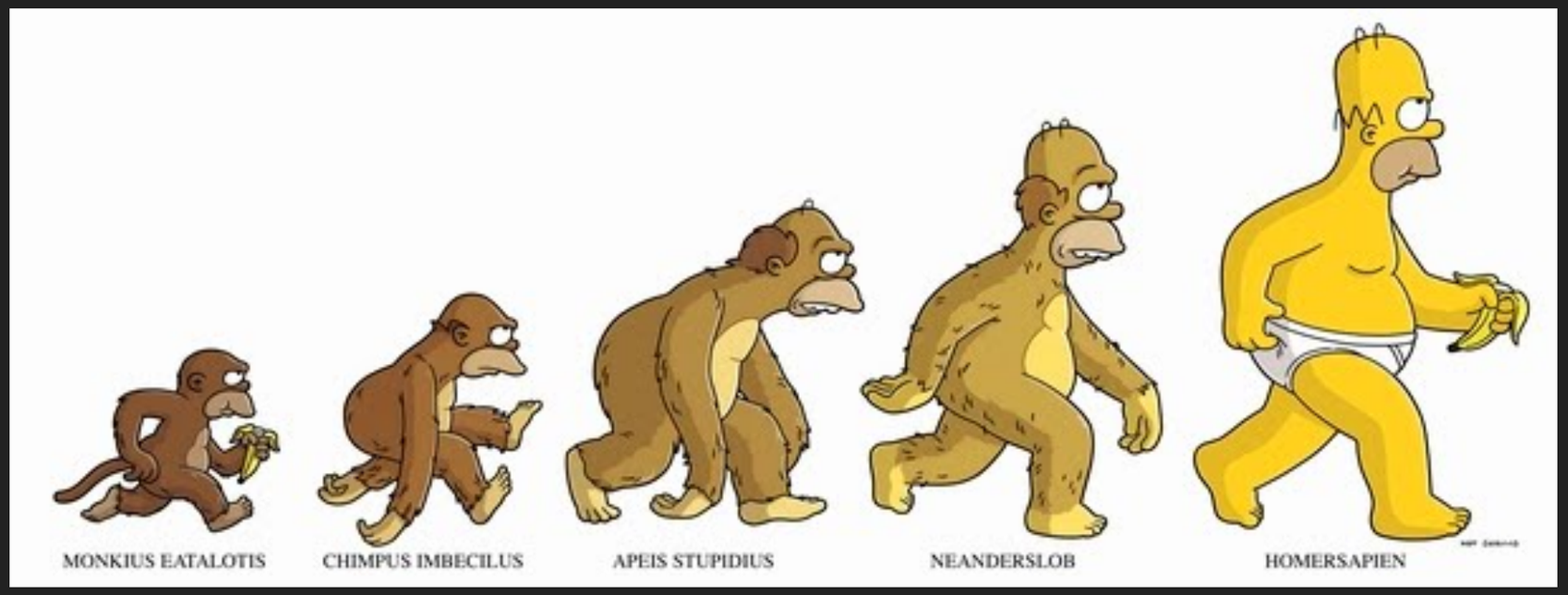
Vulnerability Detection

IAST ->

Precise
Vulnerability Detection

RASP

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 Verb Tampering
  - Preventing Header Injection
  - File Upload Protection
- Ongoing Research
  - Preventing Session Hijacking
  - Preventing Layer 7 DDoS
  - Credential Stuffing

RASP BY API INSTRUMENTATION AND DYNAMIC WHITELIST

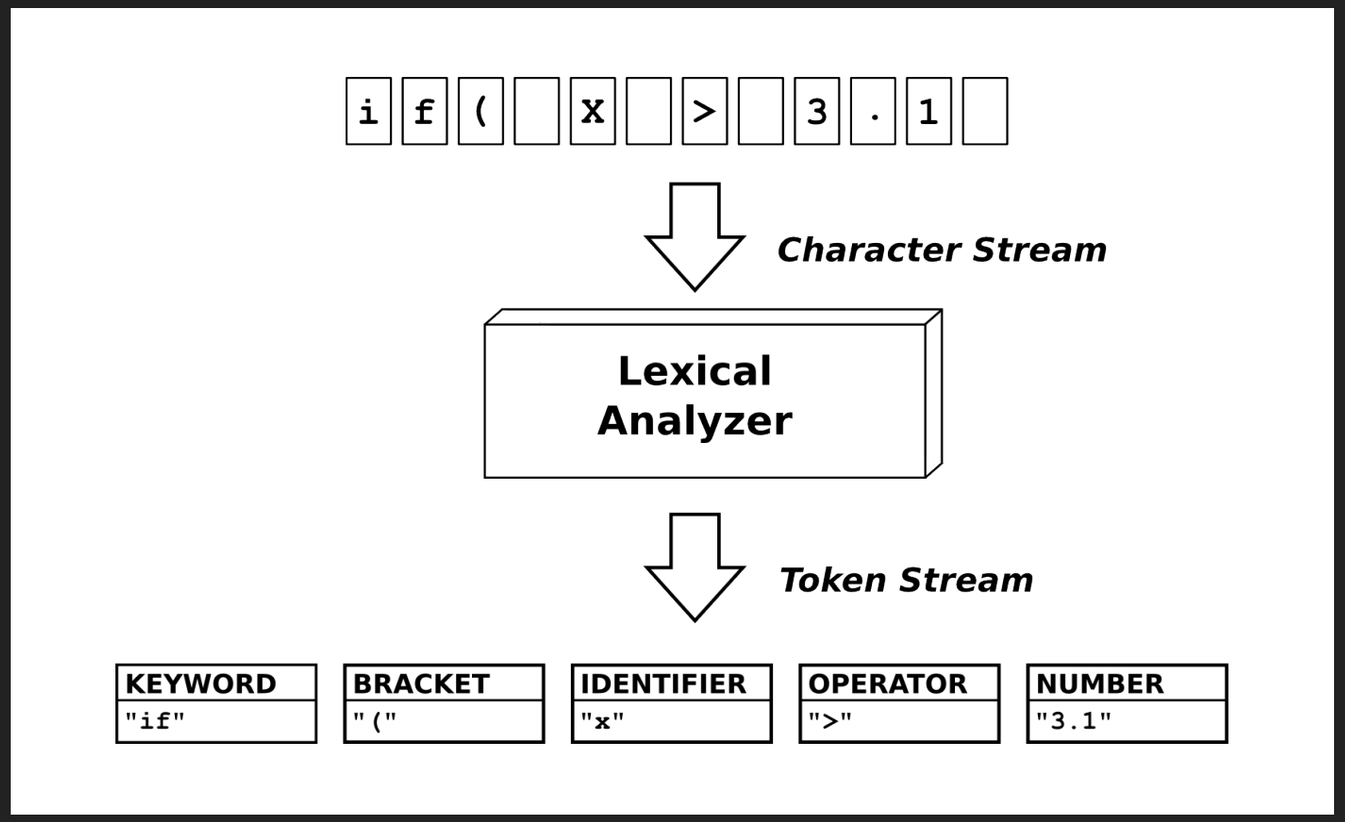
- MONKEY PATCHING
- LEXICAL ANALYSIS
- CONTEXT DETERMINATION

MONKEY PATCHING

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

- https://jsfiddle.net/h1gves49/2/

LEXICAL ANALYSIS AND TOKEN GENERATION



- 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

SYNTAX	TOKEN
int	KEYWORD
value	IDENTIFIER
=	OPERATOR
100	CONSTANT
;	SYMBOL

Custom Lexer

SYNTAX	TOKEN
int	KEYWORD
	WHITESPACE
value	IDENTIFIER
	WHITESPACE
=	OPERATOR
	WHITESPACE
100	CONSTANT
;	SYMBOL
//value is 100	COMMENT

CONTEXT DETERMINATION

HTML CODE

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

HTML PARSER

DOM TREE

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
```

SYNTAX	TOKEN
SELECT	KEYWORD
	WHITESPACE
*	OPERATOR
	WHITESPACE
FROM	KEYWORD
	WHITESPACE
logs	STRING

SQL INJECTION : PROTECT

```
SELECT * FROM logs AND DROP TABLE admin
```

SYNTAX	TOKEN
SELECT	KEYWORD
	WHITESPACE
*	OPERATOR
	WHITESPACE
FROM	KEYWORD
	WHITESPACE
logs	STRING
	WHITESPACE
AND	KEYWORD
	WHITESPACE
DROP	KEYWORD
	WHITESPACE
TABLE	KEYWORD
	WHITESPACE
admin	STRING

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 **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
```

SYNTAX	TOKEN
ping	EXECUTABLE
	WHITESPACE
-c	ARGUMENT_DASH
	WHITESPACE
3	NUMBER
	WHITESPACE
127.0.0.1	IP_OR_DOMAIN

REMOTE OS COMMAND INJECTION : PROTECT

```
ping -c 3 127.0.0.1 & cat /etc/passwd
```

SYNTAX	TOKEN
ping	EXECUTABLE
	WHITESPACE
-c	ARGUMENT_DASH
	WHITESPACE
3	NUMBER
	WHITESPACE
127.0.0.1	IP_OR_DOMAIN
	WHITESPACE
&	SPLITTER
	WHITESPACE
cat	EXECUTABLE
	WHITESPACE
/etc/passwd	UNIX_PATH

REMOTE OS COMMAND INJECTION : PROTECT

Rule for Context: `ping -c 3 <user_input>`

EXECUTABLE WHITESPACE **ARGUMENT_DASH** WHITESPACE **NUMBER** WHITESPACE **IP_OR_DOMAIN**

```
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_OR_DOMAIN**
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

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

Parsing the DOM Tree

```
└HTML
  ├HEAD
  └BODY
    ├#text:
    ├H1
    │ └#text:  hello {{user_input1}}
    ├#text:
    └SCRIPT
      └#text:  var x='{{user_input2}}';
```

HTML_CONTEXT

JAVASCRIPT_VALUE_CONTEXT

CROSS SITE SCRIPTING : PROTECT

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

user_input1 = "World"

user_input2 = "Hello World"

```
<body><h1>hello World </h1></body>
<script> var x='Hello World';</script>
```

CROSS SITE SCRIPTING : PROTECT

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

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

DEMO

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

```
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 Verb Tampering
- Preventing Header Injection
- File Upload Protection
- Preventing Path Traversal

PREVENTING VERB TAMPERING

- WAFs blindly blocks **TRACE** and **OPTION** Request
- **Hook** HTTP Request API
- **Learn** the HTTP Verbs and Generate Whitelist
- **Block** if not in the list

ROUTE	ALLOWED VERBS
/api/add_user	GET, POST
/api/auth	POST
/api/edit_user	POST, DELETE
/api/logout	GET

DEMO

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

DEMO

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

```
/../{FILE}
/../../{FILE}
/..%2f..%2f{FILE}
/%2e%2e/%2e%2e/{FILE}
/..%252f..%252f..%252f{FILE}
/%252e%252e/%252e%252e/{FILE}
/%252e%252e%252f%252e%252e%252f%252e%252e%252f{FILE}
/..\..\..\..\..\{FILE}
/..%255c..%255c..%255c..%255c{FILE}
/%2e%2e\%2e%2e\%2e%2e\%2e%2e\{FILE}
/%2e%2e%5c%2e%2e%5c%2e%2e%5c{FILE
```

PREVENTING PATH TRAVERSAL

- **Hook** File/IO API:
  `io.open("/read_dir/index.txt", 'rb')`
- **Learn** directories and file extensions
- **Block** any unknown directories and file extensions
  `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

```
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