Browser Exploits? Grab 'em by the Collar!

Presented By: Debasish Mandal (@debasishm89)

About Me

- Security researcher, currently working in McAfee IPS Vulnerability Research Team.
- Working in information security industry for past six years.
- At first was mostly focused on penetration testing of web applications and networks.
- Last three years at McAfee, primary focus has shifted to vulnerability research, reverse engineering, exploits, and exploitation techniques.
- In spare time, do security bug hunting, blogging.
- http://www.debasish.in/
- https://securingtomorrow.mcafee.com/author/debasish-mandal/

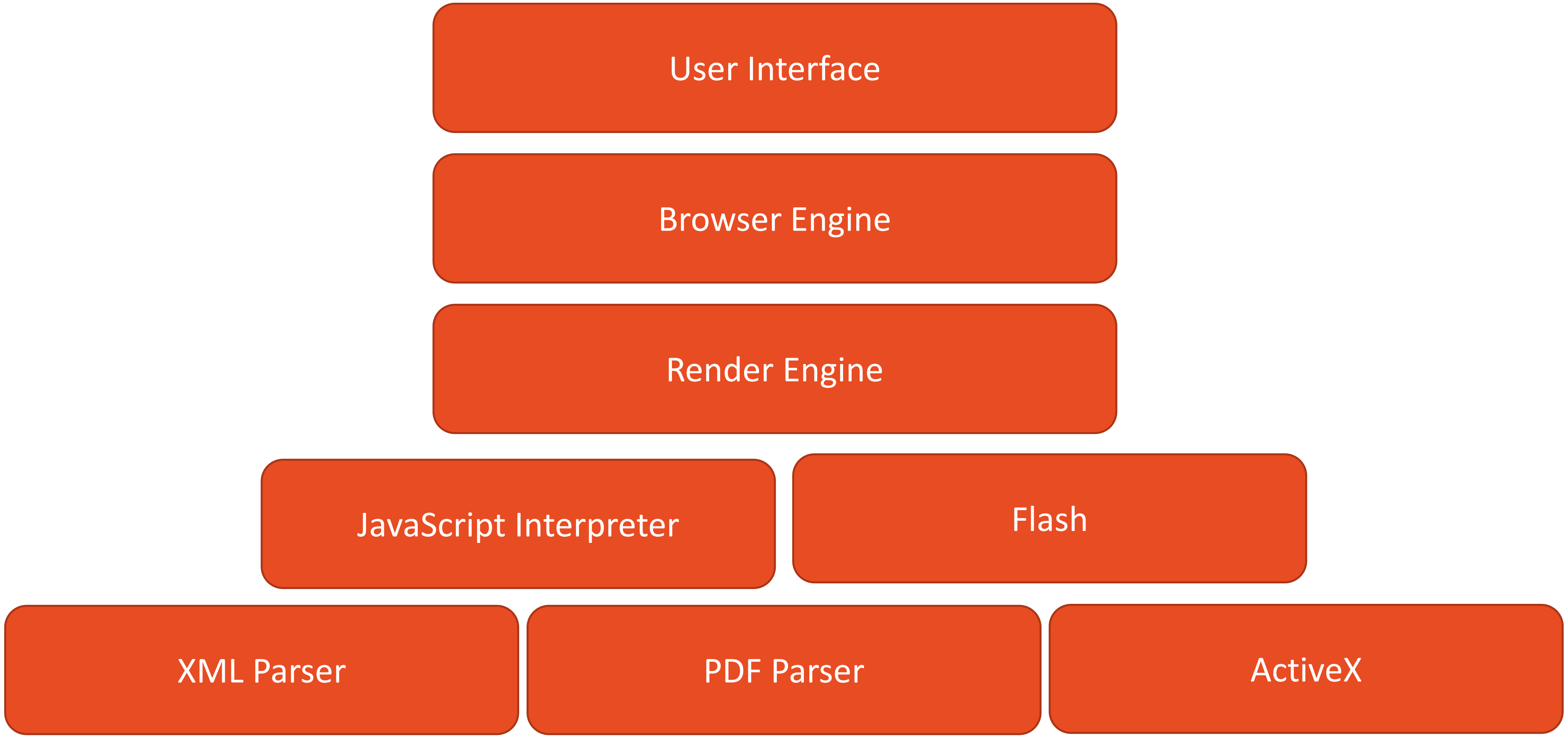
Agenda

- Brief overview of browser exploits and exploitation techniques.
- Current solutions to detect and catch browser exploits.
- Motivation behind this research.
- TCP live stream injection to catch browser exploits.
- Advantages over current exploit detection systems.
- Demo
- Closing remarks
- Q&A

Browser Exploits

- A form of malicious code that takes advantage of a flaw or vulnerability in a browser and compromises user security
- Targets different components of browser or operating system
- Exploit codes usually written in JavaScript/HTML
- Usually delivered in form of legitimate web page

Browser Architecture and Attack Surfaces



Browser Attack Surfaces

- HTML rendering engine
- JavaScript interpreter
- Third-party libraries
- XML parser
- ActiveX components
- Flash
- PDF parser
- Any component that deals with untrusted data

Types of Browser Exploits

Exploits abusing memory corruption

- Attacker tricks browser to unintentionally modify memory location, violating memory safety.
- Attacker does memory spraying to prepare a predictable memory layout.
- At final stage, shellcode is used to compromise system security.

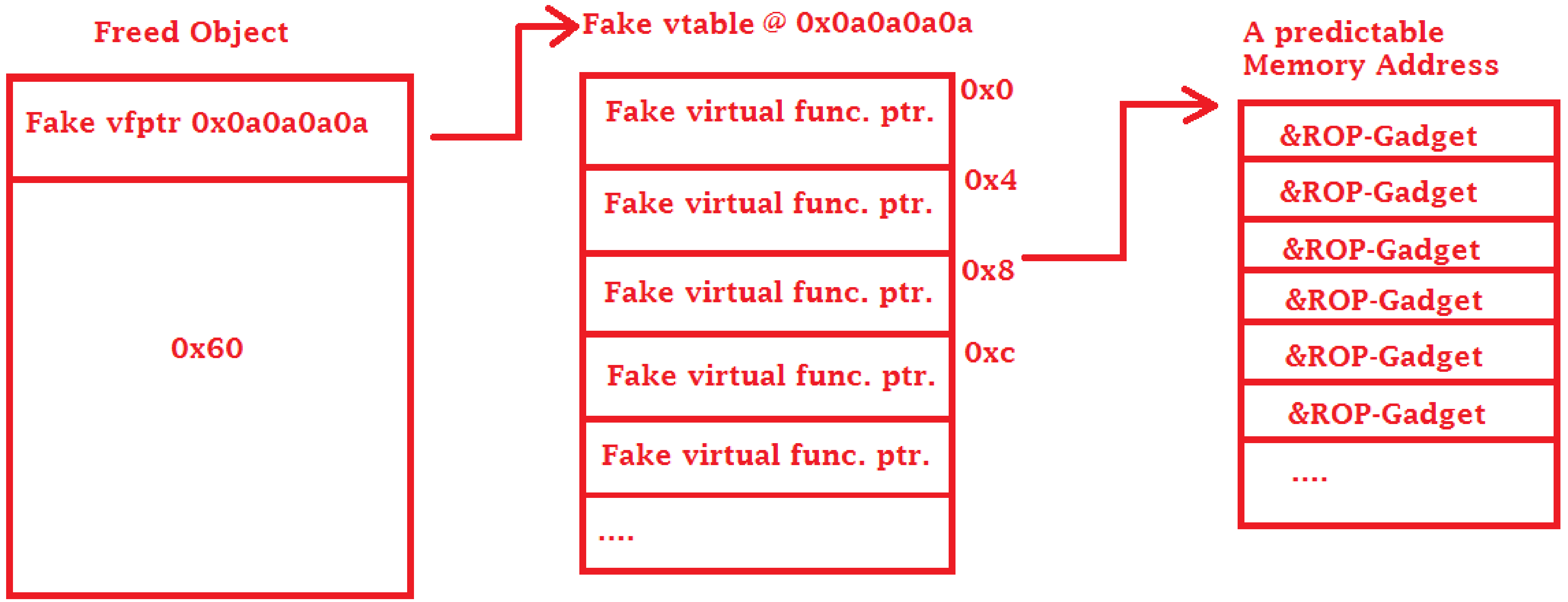
Exploits abusing logical/design flaws

- There is no generic technique to exploit logical design flaws in browsers.
- Exploitation depends completely on the anatomy of a vulnerability.

Exploiting Memory Corruption in Browser

1. Exploit delivery
2. The exploit deobfuscates itself (in case it is an obfuscated exploit)
3. Prepare desired memory layout for exploitation (by spraying heap with malicious code such as shellcode, ROP chain, etc.)
4. Trigger required browser vulnerability(s)
5. Bypass DEP/ASLR/other mitigation techniques (if there are any)
6. Transfer program control flow to the malicious code, which is placed in the desired memory location in Step 2.

Example: Exploiting Use-After-Free in Browsers



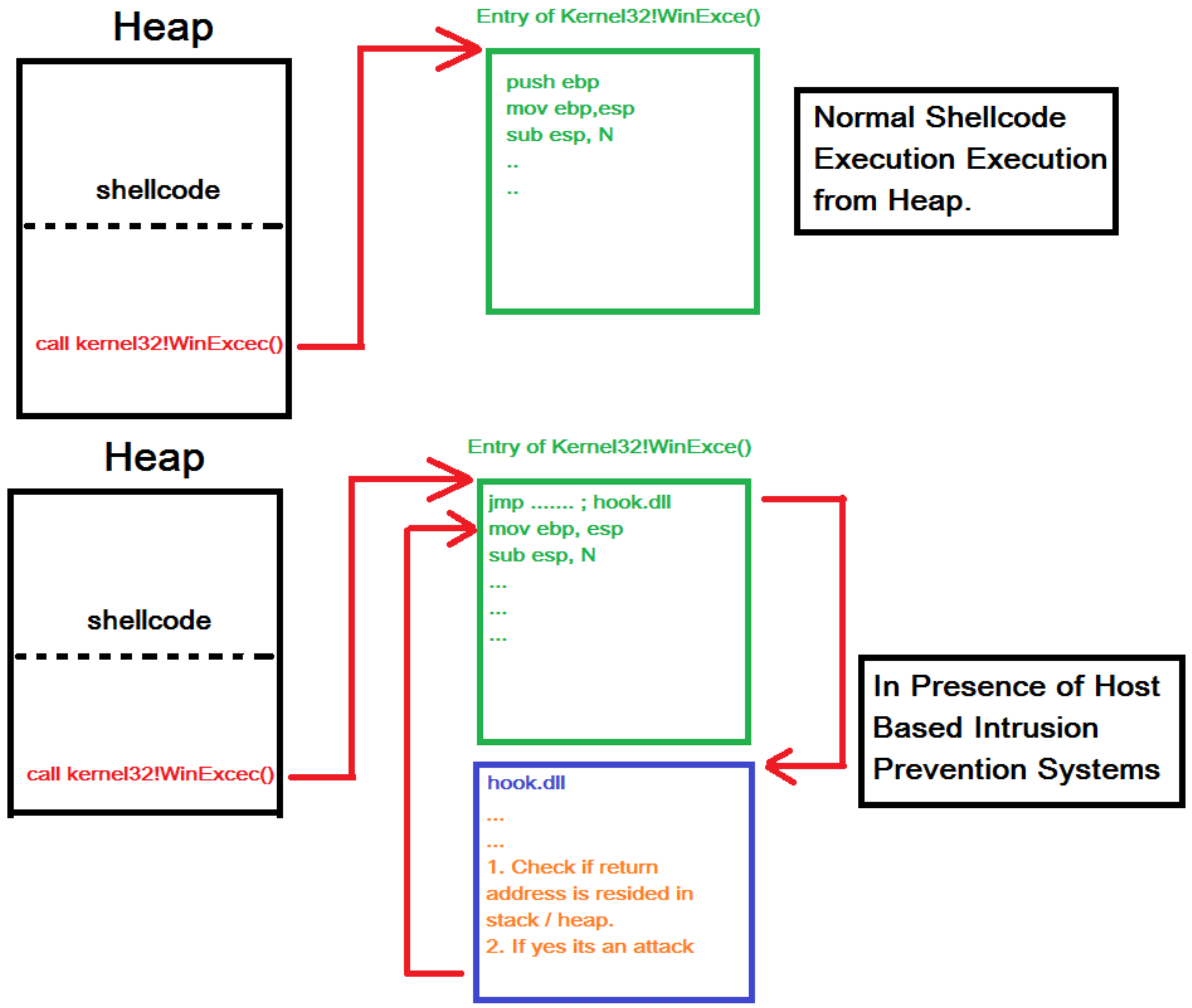
Existing Solutions to Detect and Prevent Browser Exploits

- Host-based intrusion prevention system
- Network-based intrusion prevention system
- Sandbox-based network intrusion prevention system

Existing Solutions to Detect and Prevent Browser Exploits: Host-Based Detection

- Installed on endpoints; monitors system for suspicious activity by analyzing events occurring within that host.
  - Hooks different OS APIs and monitors them
  - Inspects API arguments at runtime
  - Follows different heuristics to detect anomaly

Example of Host-Based Browser Exploit Detection



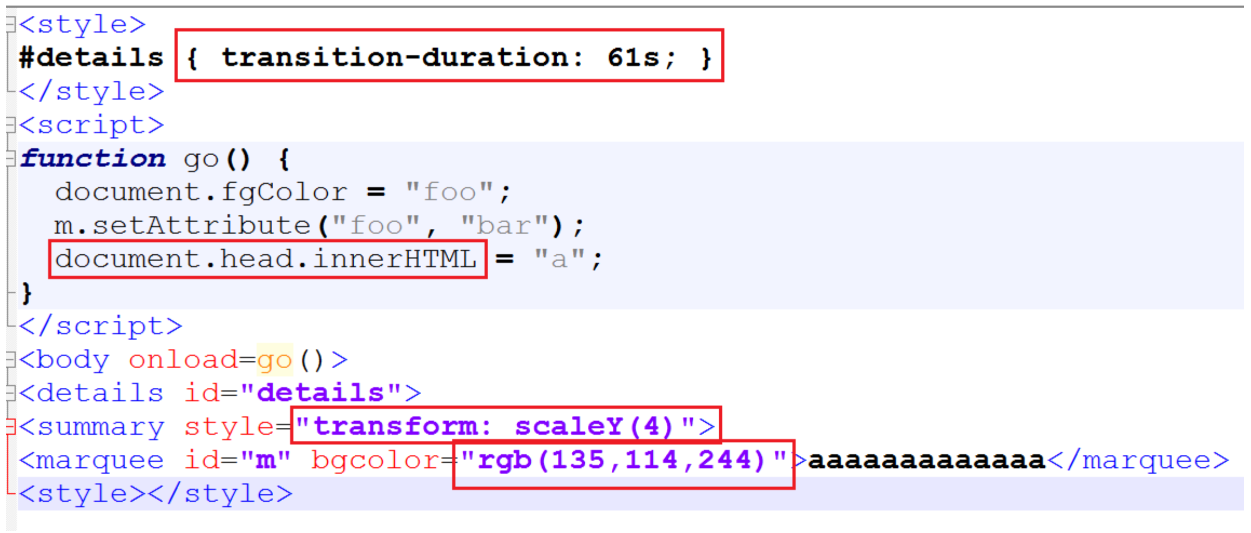
Heap
shellcode
call kernel32!WinExcec()
Entry of Kernel32!WinExce()
push ebp
mov ebp,esp
sub esp, N
..
..
Normal Shellcode Execution Execution from Heap.
Heap
shellcode
call kernel32!WinExcec()
Entry of Kernel32!WinExce()
jmp .... ; hook.dll
mov ebp, esp
sub esp, N
...
...
...
hook.dll
...
...
1. Check if return address is resided in stack / heap.
2. If yes its an attack
In Presence of Host Based Intrusion Prevention Systems

Limitations of Host-Based IPS

- Agent based
- Hook hopping is an old-school technique to bypass host-based IPS.
- Although capable of catching obfuscated exploits, host-based IPS has a significant impact on OS performance.
- Pushing new updates/signatures to every endpoint can be painful for any large organization.

Existing Solutions to Detect and Prevent Browser Exploits: Network-Based Detection

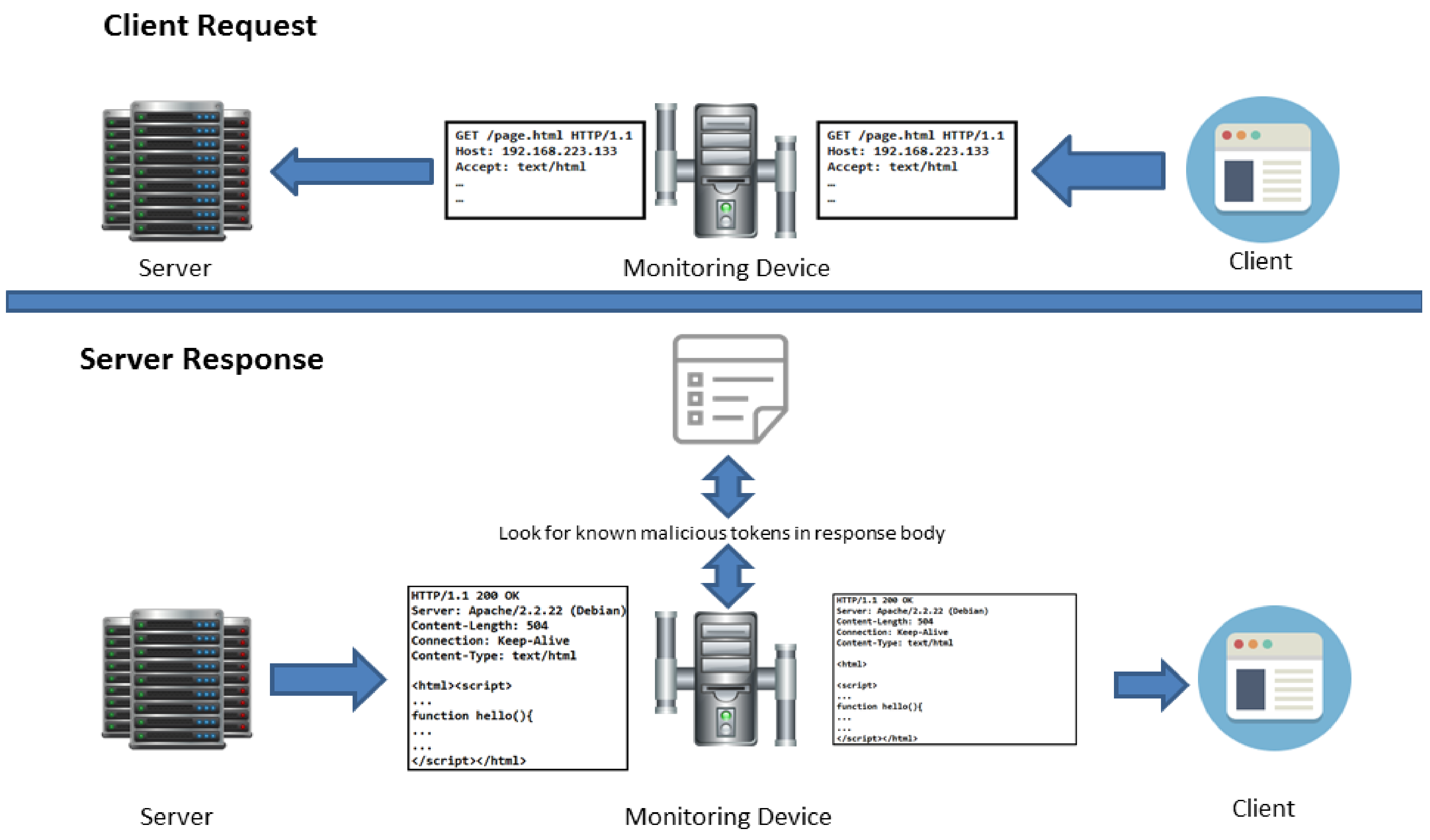
- Sits in corporate gateway
- Intercepts HTTP response
- Looks for malicious tokens in HTTP response
- Sandbox-based network IPS solutions execute/open files in a sandbox and look
  for suspicious behavior.



```
<style>
#details { transition-duration: 61s; }
</style>
<script>
function go() {
  document.fgColor = "foo";
  m.setAttribute("foo", "bar");
  document.head.innerHTML = "a";
}
</script>
<body onload=go()>
<details id="details">
<summary style="transform: scaleY(4)">
<marquee id="m" bgcolor="rgb(135,114,244)">aaaaaaaaaaaaa</marquee>
<style></style>
```

Example of Network-Based Browser Exploit Detection

Existing Signature Based Browser Exploit Detection System



```
GET /page.html HTTP/1.1
Host: 192.168.223.133
Accept: text/html
…
…
```

```
GET /page.html HTTP/1.1
Host: 192.168.223.133
Accept: text/html
…
…
```

```
HTTP/1.1 200 OK
Server: Apache/2.2.22 (Debian)
Content-Length: 504
Connection: Keep-Alive
Content-Type: text/html

<html><script>
...
function hello(){
...
...
</script></html>
```

```
HTTP/1.1 200 OK
Server: Apache/2.2.22 (Debian)
Content-Length: 504
Connection: Keep-Alive
Content-Type: text/html

<html>

<script>
...
function hello(){
...
...
</script></html>
```

Limitations and Drawbacks of Existing Exploit Detection System: Network Based

- Mainly malicious token-based detection mechanisms and browser exploits are very dynamic in nature. Very unreliable.
- Has significant impact on network monitoring device performance. Increases latency.
- Successful execution of browser exploits in a network sandbox is very difficult because they are highly dependent on the environment.

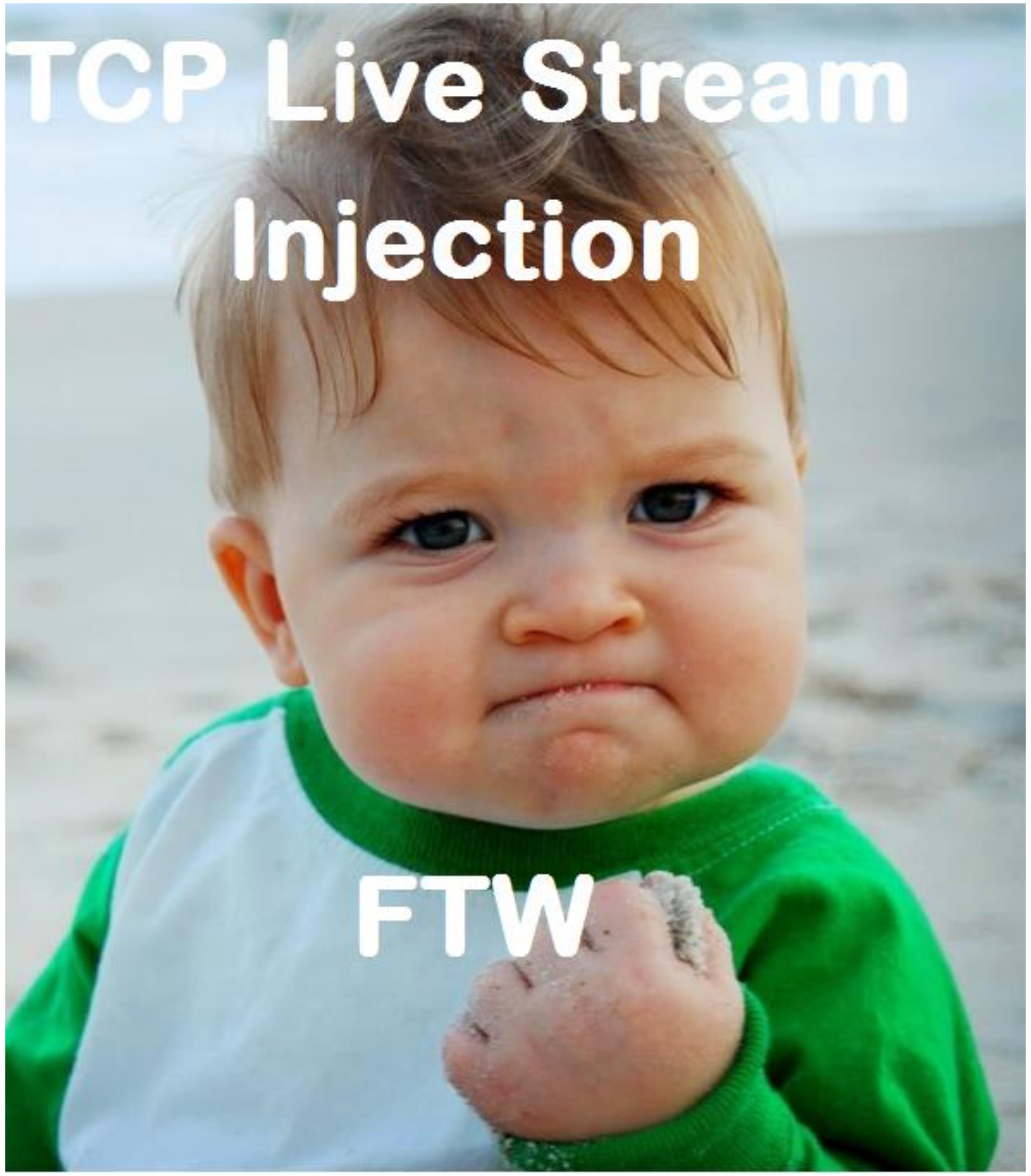
Motivation Behind This Research

- One browser-based exploit can be written/obfuscated in thousands of ways. Hence signature-/token-based network detection system fails drastically.
- Host-based IPS has its own limitations: impact on OS performance, effort required to push updates to each endpoint.
- For any host IPS, pushing new updates to each endpoint can be painful.

The Idea

- As with host-based IPS, if we can somehow place our application behavior-monitoring code (e.g., hooking code) into the user’s system, we can make the system generic and solve the problem of obfuscation up to a certain level.
- Can we do this without installing an agent on the endpoint?

We Can Do It!


TCP Live Stream
Injection
FTW

Basics of Network Packet/TCP Live Stream Injection Techniques

- Packet injection is a process of interfering with an established network connection, by constructing packets to appear as if they are part of the normal communication stream.

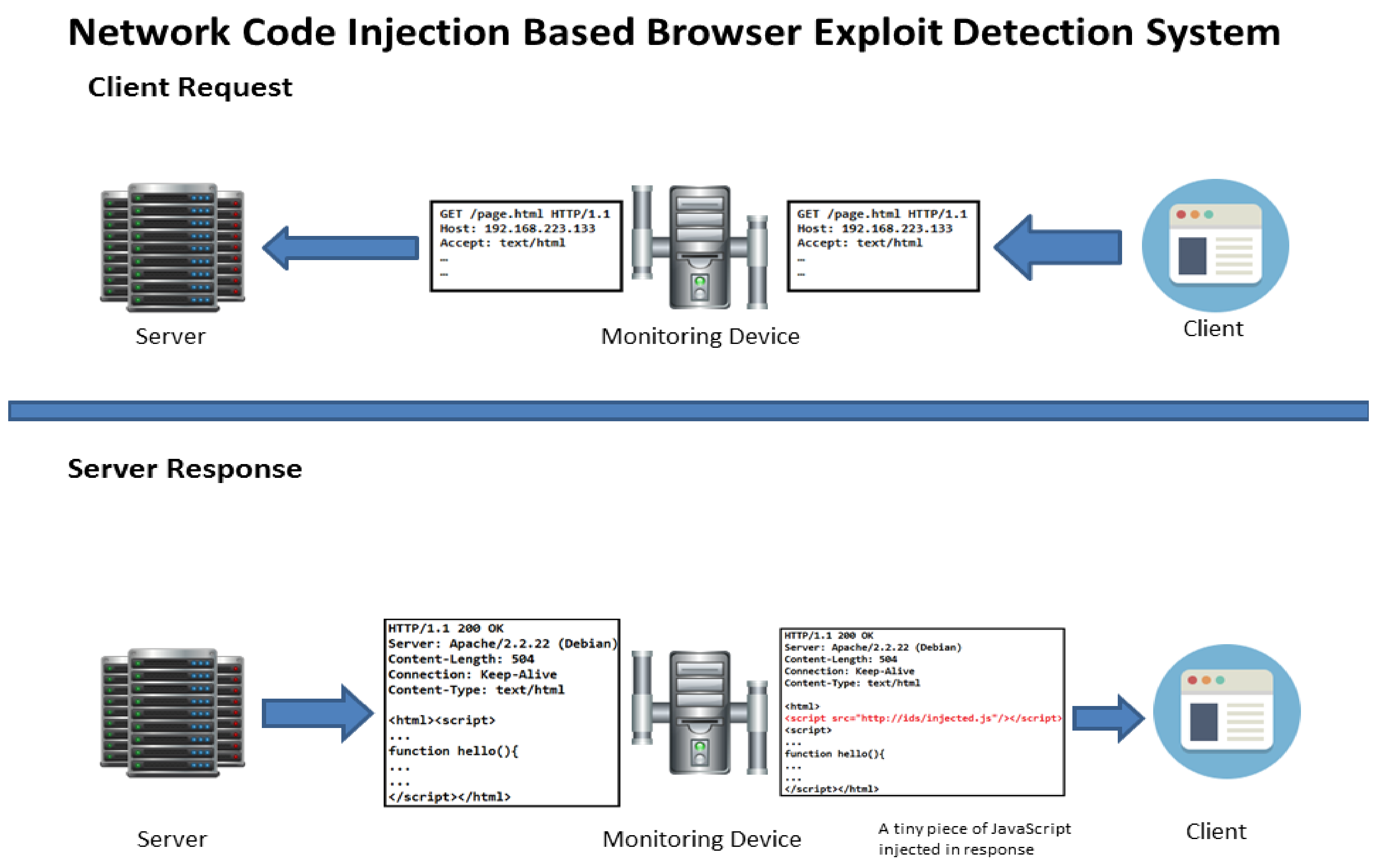
General Use of TCP Live Stream Injection Techniques

- Internet Service Providers, router vendors inject arbitrary advertisements into live web pages.
- Disrupting certain services
- MITM attacks

TCP Live Stream Code Injection for Browser Exploit Detection

- The detection system injects a tiny piece of JavaScript code into the (HTTP response body) page.
- Injected in such a manner that when the JavaScript is delivered to user’s browser, the injected JS code is executed first.
- Works by injecting our script at the top of the page.
- https://www.infoworld.com/article/2925839/net-neutrality/code-injection-new-low-isps.html
- https://arstechnica.com/tech-policy/2013/04/how-a-banner-ad-for-hs-ok/

Working Principle


Network Code Injection Based Browser Exploit Detection System
Client Request
Server
Monitoring Device
Client
Server Response
Server
Monitoring Device
A tiny piece of JavaScript injected in response
Client

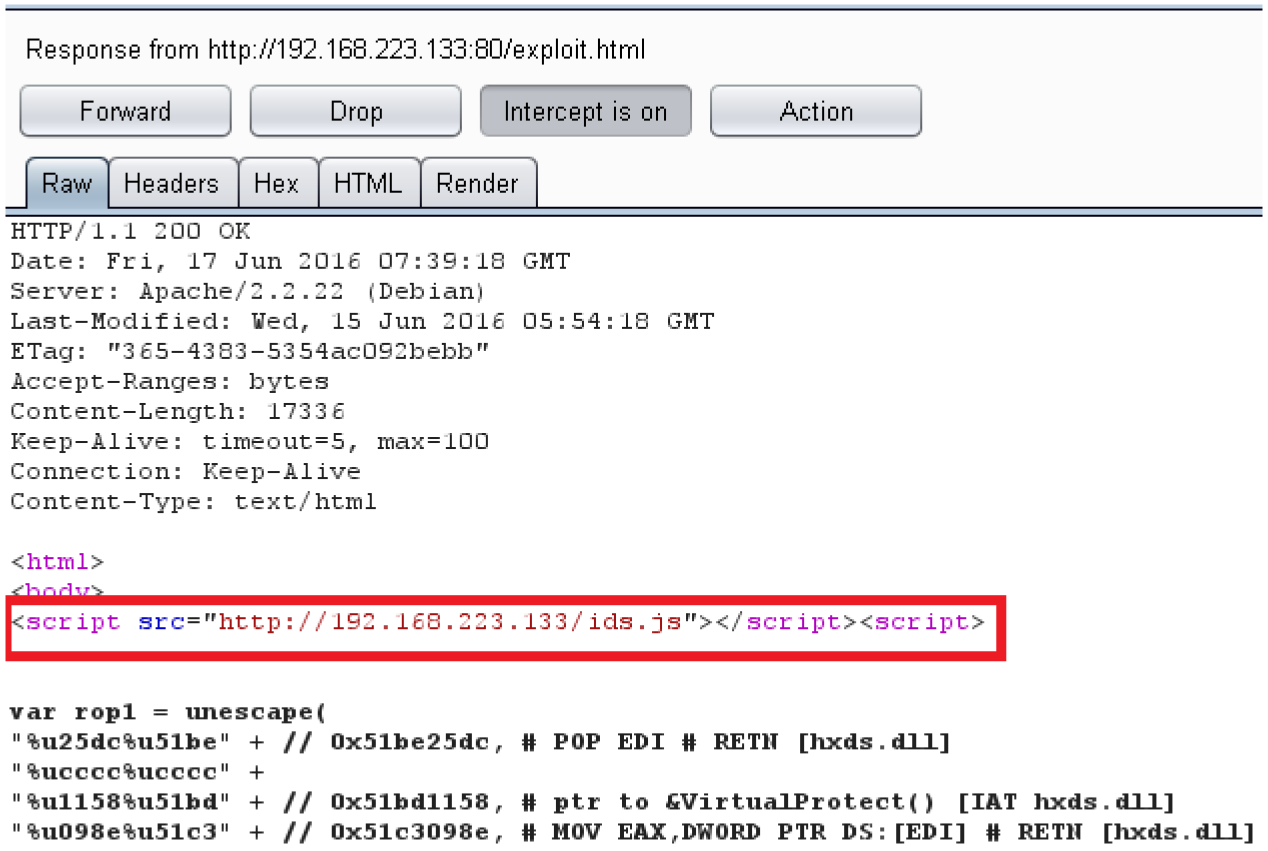
```
GET /page.html HTTP/1.1
Host: 192.168.223.133
Accept: text/html
…
…
```

```
GET /page.html HTTP/1.1
Host: 192.168.223.133
Accept: text/html
…
…
```

```
HTTP/1.1 200 OK
Server: Apache/2.2.22 (Debian)
Content-Length: 504
Connection: Keep-Alive
Content-Type: text/html

<html><script>
...
function hello(){
...
...
</script></html>
```

```
HTTP/1.1 200 OK
Server: Apache/2.2.22 (Debian)
Content-Length: 504
Connection: Keep-Alive
Content-Type: text/html

<html>
<script src="http://ids/injected.js"/></script>
<script>
...
function hello(){
...
...
</script></html>
```

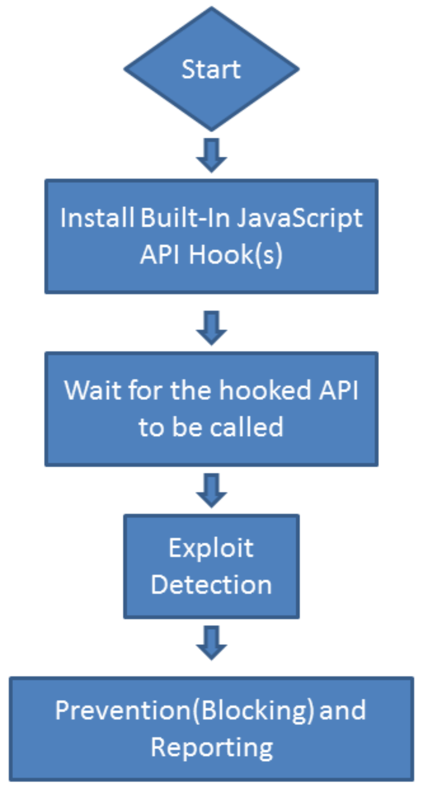
Working Principle, Continued



```
HTTP/1.1 200 OK
Date: Fri, 17 Jun 2016 07:39:18 GMT
Server: Apache/2.2.22 (Debian)
Last-Modified: Wed, 15 Jun 2016 05:54:18 GMT
ETag: "365-4383-5354ac092bebb"
Accept-Ranges: bytes
Content-Length: 17336
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<html>
<body>
<script src="http://192.168.223.133/ids.js"></script><script>

var rop1 = unescape(
"%u25dc%u51be" + // 0x51be25dc, # POP EDI # RETN [hxds.dll]
"%ucccc%ucccc" +
"%u1158%u51bd" + // 0x51bd1158, # ptr to &VirtualProtect() [IAT hxds.dll]
"%u098e%u51c3" + // 0x51c3098e, # MOV EAX,DWORD PTR DS:[EDI] # RETN [hxds.dll]
```

The injected code looks like this when we intercept it.

Working Principle, Continued

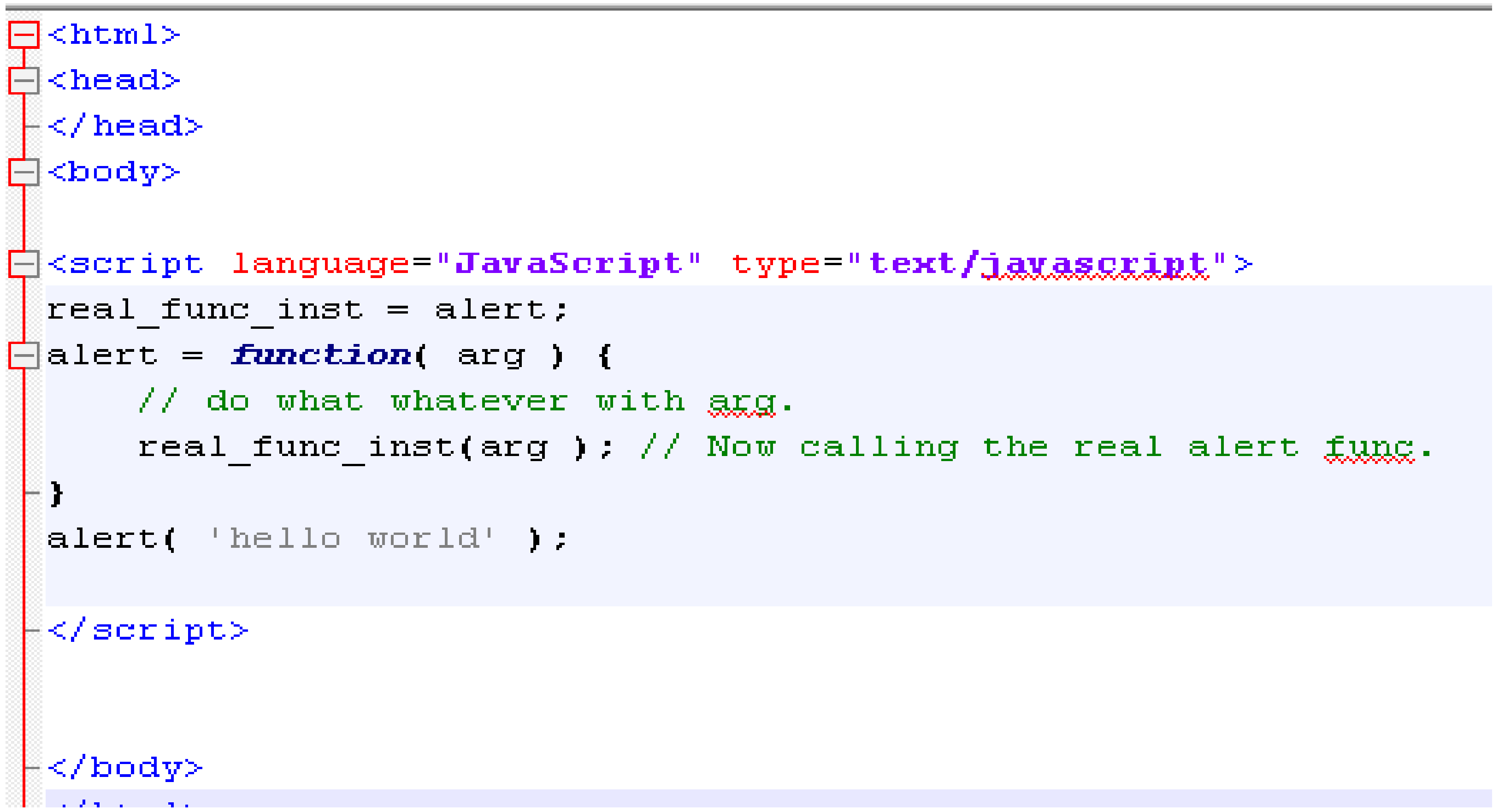

Start
Install Built-In JavaScript API Hook(s)
Wait for the hooked API to be called
Exploit Detection
Prevention(Blocking) and Reporting

Note : We are talking about hooking JavaScript API(s), not OS APIs.

Installing a Built-In JavaScript API Hook

- Injected JavaScript code is executed first.
- The injected code installs JavaScript API hooks.
- Mainly it hooks JavaScript APIs that are commonly used by malicious developers for exploitation, obfuscation, preparation of memory layout, etc.
- Once our JavaScript hooks are installed in the client’s browser, whenever those APIs are called from the page, we can intercept its arguments.

Installing a Built-In JavaScript API Hook

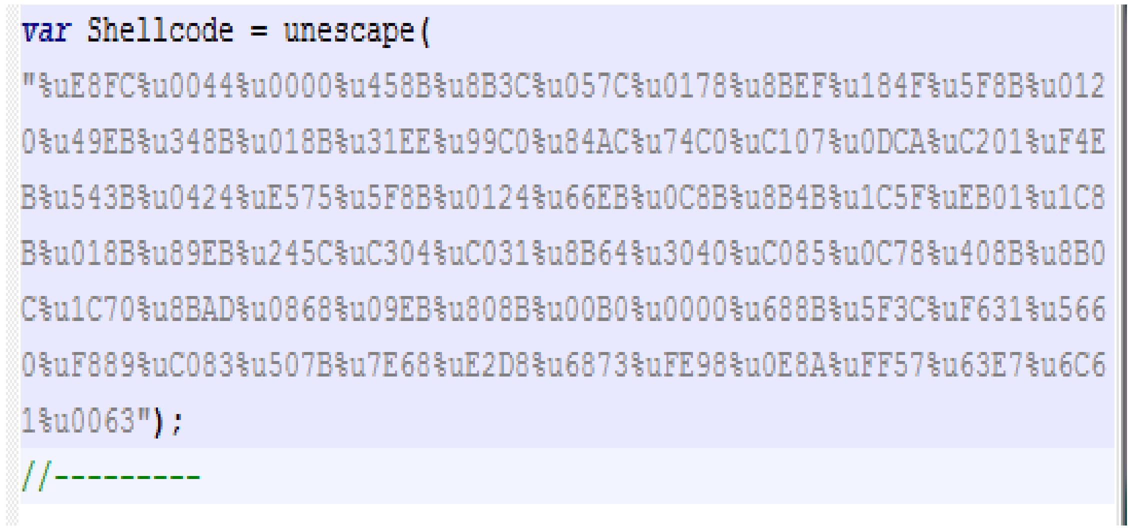


```
<html>
<head>
</head>
<body>

<script language="JavaScript" type="text/javascript">
real_func_inst = alert;
alert = function( arg ) {
    // do what whatever with arg.
    real_func_inst(arg ); // Now calling the real alert func.
}
alert( 'hello world' );

</script>


</body>
```

Commonly Used JavaScript APIs in Browser Exploitations

- escape()
- unescape()
- String operations related API such as substring(x,x,)
- Functions involved in Array() operations
- Functions involved in string operations.
- document.write(), document.createlement(), etc.
- Functions involved in ActiveXObject
- And hundreds more…

Built-In JavaScript API Hook Example



```
var Shellcode = unescape(
"%uE8FC%u0044%u0000%u458B%u8B3C%u057C%u0178%u8BEF%u184F%u5F8B%u012
0%u49EB%u348B%u018B%u31EE%u99C0%u84AC%u74C0%uC107%u0DCA%uC201%uF4E
B%u543B%u0424%uE575%u5F8B%u0124%u66EB%u0C8B%u8B4B%u1C5F%uEB01%u1C8
B%u018B%u89EB%u245C%uC304%uC031%u8B64%u3040%uC085%u0C78%u408B%u8B0
C%u1C70%u8BAD%u0868%u09EB%u808B%u00B0%u0000%u688B%u5F3C%uF631%u566
0%uF889%uC083%u507B%u7E68%uE2D8%u6873%uFE98%u0E8A%uFF57%u63E7%u6C6
1%u0063");
//---------
```

- If we have the JavaScript function unescape() hooked.
- We can easily intercept the argument passed to it…
- And perform various checks to determine if the parameter is malicious in nature.

Previous work on JavaScript Hooking

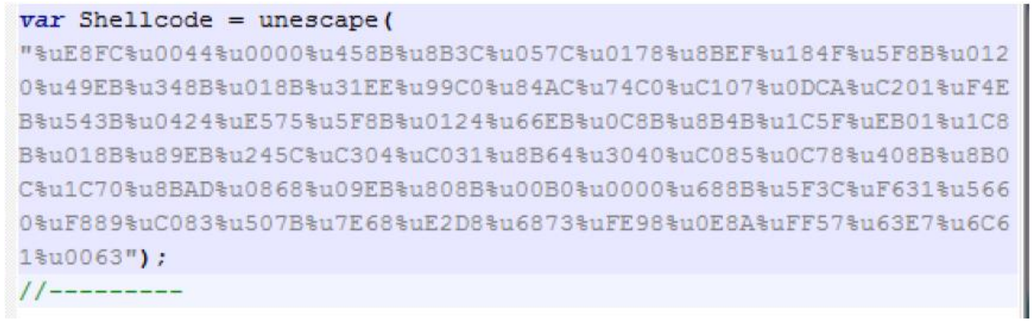
- BeEF: The Browser Exploitation Framework – hook.js
- Javascript Hooking for Malicious Website Research by Liran Englender and Kris Kaspersky

Demo 1: Built-In JavaScript API Hook

- The demo quickly shows how JavaScript API hooking works.

Example of Exploit Detection: Shellcode

- Often browser exploits handle shellcode within JavaScript code:

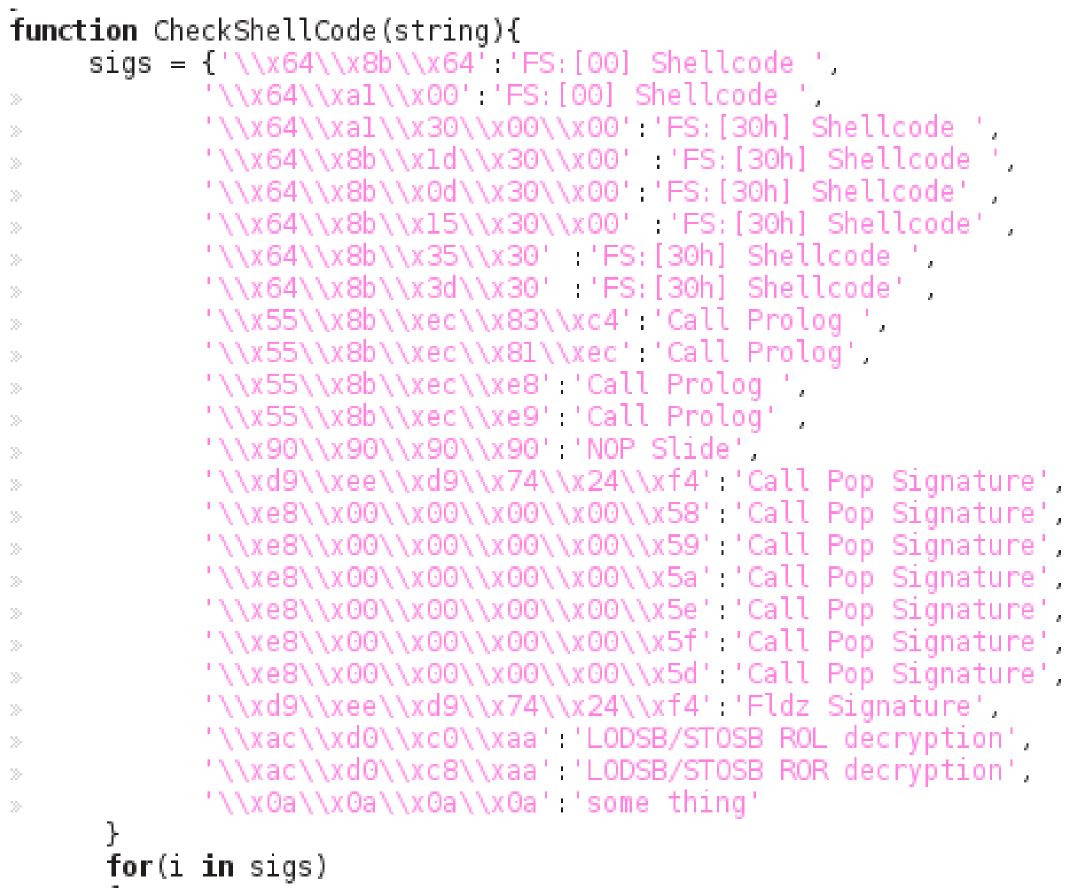


```
var Shellcode = unescape(
"%uE8FC%u0044%u0000%u458B%u8B3C%u057C%u0178%u8BEF%u184F%u5F8B%u012
0%u49EB%u348B%u018B%u31EE%u99C0%u84AC%u74C0%uC107%u0DCA%uC201%uF4E
B%u543B%u0424%uE575%u5F8B%u0124%u66EB%u0C8B%u8B4B%u1C5F%uEB01%u1C8
B%u018B%u89EB%u245C%uC304%uC031%u8B64%u3040%uC085%u0C78%u408B%u8B0
C%u1C70%u8BAD%u0868%u09EB%u808B%u00B0%u0000%u688B%u5F3C%uF631%u566
0%uF889%uC083%u507B%u7E68%uE2D8%u6873%uFE98%u0E8A%uFF57%u63E7%u6C6
1%u0063");
//---------
```

- When the system hooks the unescape() function and intercepts argument passed to it.
- And looks for malicious opcodes/patterns often found in shellcode.

Example of Exploit Detection: Shellcode

- Malicious opcodes such as
  - Call pop
  - FS:[00]
  - FS:[30h]
  - Call xxxx
  - nop sleds
  - Etc.



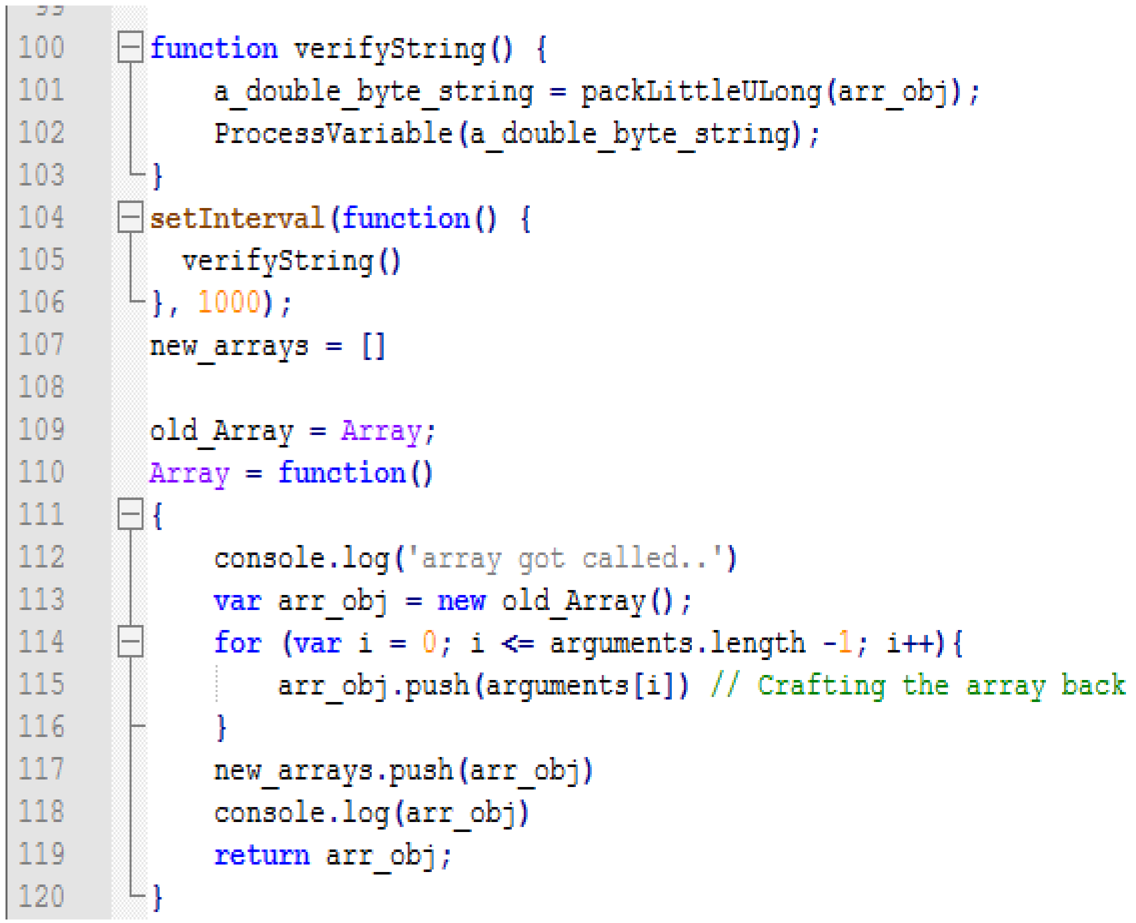
```
function CheckShellCode(string){
     sigs = {'\\x64\\x8b\\x64':'FS:[00] Shellcode ',
             '\\x64\\xa1\\x00':'FS:[00] Shellcode ',
             '\\x64\\xa1\\x30\\x00\\x00':'FS:[30h] Shellcode ',
             '\\x64\\x8b\\x1d\\x30\\x00' :'FS:[30h] Shellcode ',
             '\\x64\\x8b\\x0d\\x30\\x00':'FS:[30h] Shellcode' ,
             '\\x64\\x8b\\x15\\x30\\x00' :'FS:[30h] Shellcode' ,
             '\\x64\\x8b\\x35\\x30' :'FS:[30h] Shellcode ',
             '\\x64\\x8b\\x3d\\x30' :'FS:[30h] Shellcode' ,
             '\\x55\\x8b\\xec\\x83\\xc4':'Call Prolog ',
             '\\x55\\x8b\\xec\\x81\\xec':'Call Prolog',
             '\\x55\\x8b\\xec\\xe8':'Call Prolog ',
             '\\x55\\x8b\\xec\\xe9':'Call Prolog' ,
             '\\x90\\x90\\x90\\x90':'NOP Slide',
             '\\xd9\\xee\\xd9\\x74\\x24\\xf4':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x58':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x59':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x5a':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x5e':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x5f':'Call Pop Signature',
             '\\xe8\\x00\\x00\\x00\\x00\\x5d':'Call Pop Signature',
             '\\xd9\\xee\\xd9\\x74\\x24\\xf4':'Fldz Signature',
             '\\xac\\xd0\\xc0\\xaa':'LODSB/STOSB ROL decryption',
             '\\xac\\xd0\\xc8\\xaa':'LODSB/STOSB ROR decryption',
             '\\x0a\\x0a\\x0a\\x0a':'some thing'
      }
      for(i in sigs)
```

Example of Exploit Detection: Spray

- We reviewed several exploits and classified JavaScript APIs used for preparing memory layout and spraying.
- One very popular API for memory spraying is **Array()**. Real-world exploits frequently use APIs like Array(), Uint32Array(), (push, pop).
- During memory spraying these functions are very aggressively called by the exploit code.

Example of Exploit Detection: Spray

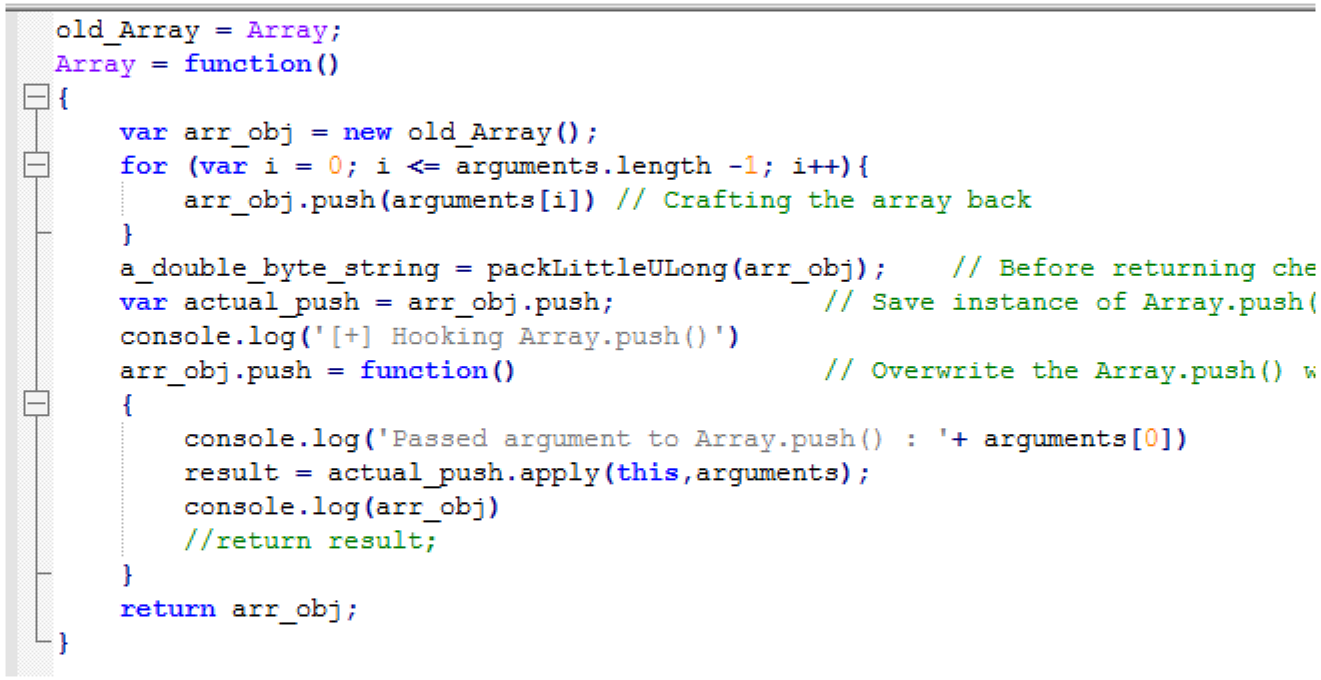
- Keep track of Array() created dynamically
- Keep checking when some operations are done on them and looks for suspicious tokens.



```
function verifyString() {
    a_double_byte_string = packLittleULong(arr_obj);
    ProcessVariable(a_double_byte_string);
}
setInterval(function() {
  verifyString()
}, 1000);
new_arrays = []

old_Array = Array;
Array = function()
{
    console.log('array got called..')
    var arr_obj = new old_Array();
    for (var i = 0; i <= arguments.length -1; i++){
        arr_obj.push(arguments[i]) // Crafting the array back
    }
    new_arrays.push(arr_obj)
    console.log(arr_obj)
    return arr_obj;
}
```

Example of Exploit Detection: Spray

- Intercepting Array.push() routine.

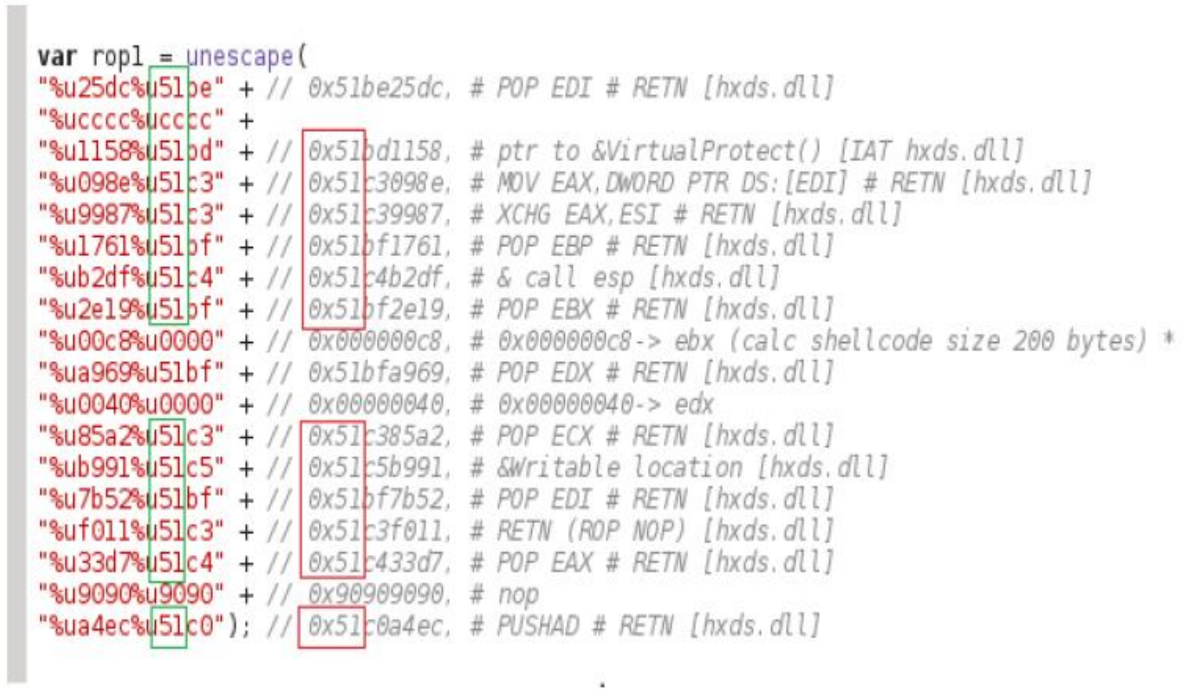


```
old_Array = Array;
Array = function()
{
    var arr_obj = new old_Array();
    for (var i = 0; i <= arguments.length -1; i++){
        arr_obj.push(arguments[i]) // Crafting the array back
    }
    a_double_byte_string = packLittleULong(arr_obj);    // Before returning che
    var actual_push = arr_obj.push;                    // Save instance of Array.push(
    console.log('[+] Hooking Array.push()')
    arr_obj.push = function()                          // Overwrite the Array.push() w
    {
        console.log('Passed argument to Array.push() : '+ arguments[0])
        result = actual_push.apply(this,arguments);
        console.log(arr_obj)
        //return result;
    }
    return arr_obj;
}
```

Demo 2: Heap Spray Detection

Example of Exploit Detection: ROP Chain

- The ROP chain will have a certain pattern, which can be used to detect whether any JavaScript string has an ROP chain.
- ROP gadgets are chosen from a single module; the most significant byte of addresses pointing to an ROP gadget will always remain same.



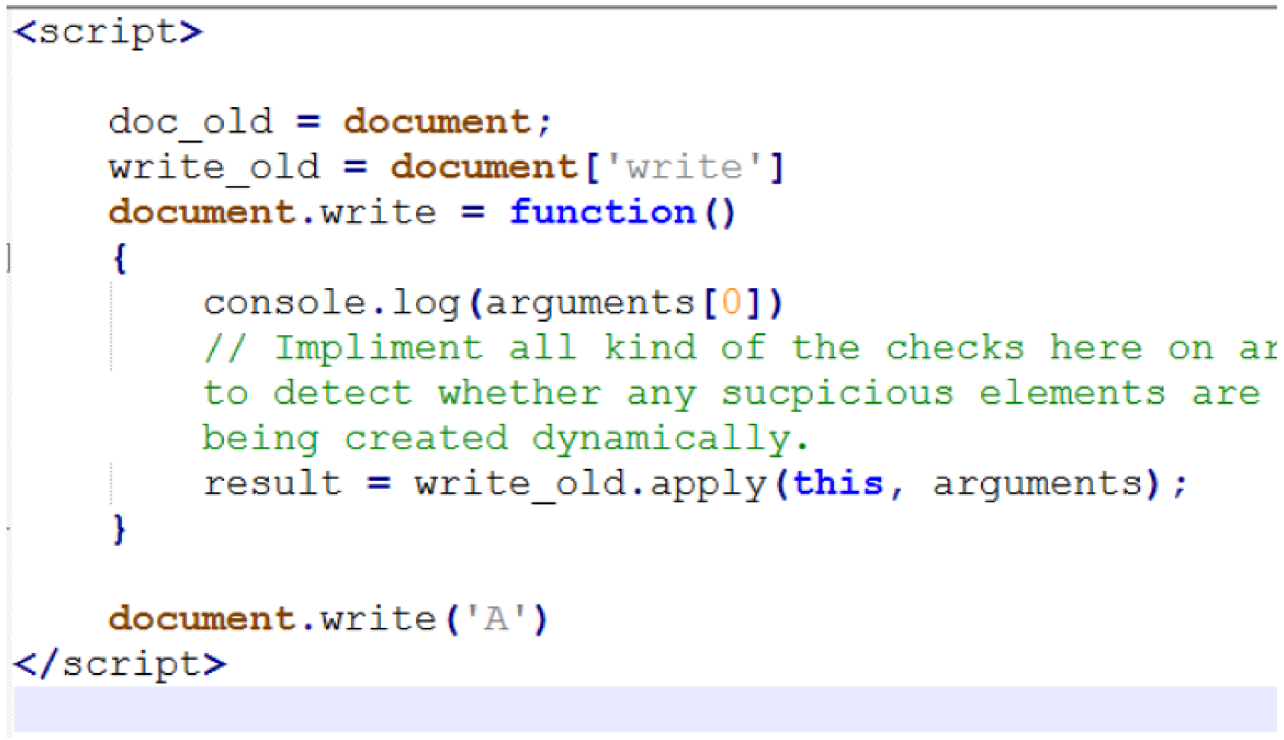
```
var rop1 = unescape(
"%u25dc%u51be" + // 0x51be25dc, # POP EDI # RETN [hxds.dll]
"%ucccc%ucccc" +
"%u1158%u51bd" + // 0x51bd1158, # ptr to &VirtualProtect() [IAT hxds.dll]
"%u098e%u51c3" + // 0x51c3098e, # MOV EAX,DWORD PTR DS:[EDI] # RETN [hxds.dll]
"%u9987%u51c3" + // 0x51c39987, # XCHG EAX,ESI # RETN [hxds.dll]
"%u1761%u51bf" + // 0x51bf1761, # POP EBP # RETN [hxds.dll]
"%ub2df%u51c4" + // 0x51c4b2df, # & call esp [hxds.dll]
"%u2e19%u51bf" + // 0x51bf2e19, # POP EBX # RETN [hxds.dll]
"%u00c8%u0000" + // 0x000000c8, # 0x000000c8-> ebx (calc shellcode size 200 bytes) *
"%ua969%u51bf" + // 0x51bfa969, # POP EDX # RETN [hxds.dll]
"%u0040%u0000" + // 0x00000040, # 0x00000040-> edx
"%u85a2%u51c3" + // 0x51c385a2, # POP ECX # RETN [hxds.dll]
"%ub991%u51c5" + // 0x51c5b991, # &Writable location [hxds.dll]
"%u7b52%u51bf" + // 0x51bf7b52, # POP EDI # RETN [hxds.dll]
"%uf011%u51c3" + // 0x51c3f011, # RETN (ROP NOP) [hxds.dll]
"%u33d7%u51c4" + // 0x51c433d7, # POP EAX # RETN [hxds.dll]
"%u9090%u9090" + // 0x90909090, # nop
"%ua4ec%u51c0"); // 0x51c0a4ec, # PUSHAD # RETN [hxds.dll]
```

Example of Exploit Detection: Dynamic Element Creation

- Once the exploit deobfuscated itself, it may try to dynamically create several elements to load the exploit.
- We have seen one very common technique used by exploit developers: to load the exploit through 0 x 0 or an invisible iframe.
- These iframes can be created dynamically in many ways. One is document.createElement("iframe");
- document.write(……)

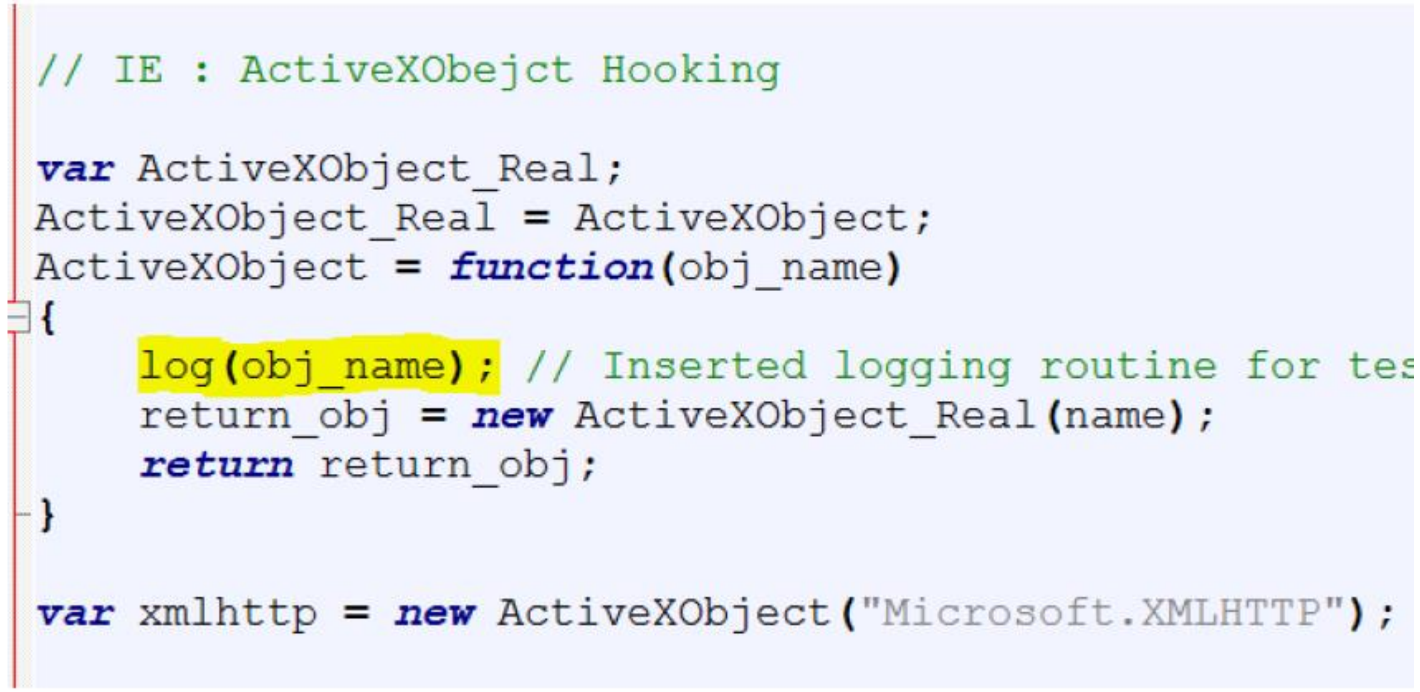
Example of Exploit Detection: Dynamic Element Creation

- This shows how our exploit detection system hooks into the document.write() function and intercepts arguments. Once the arguments are intercepted, we can perform various checks to decide if the write() call is suspicious.



```
<script>

    doc_old = document;
    write_old = document['write']
    document.write = function()
    {
        console.log(arguments[0])
        // Impliment all kind of the checks here on ar
        to detect whether any sucpicious elements are
        being created dynamically.
        result = write_old.apply(this, arguments);
    }

    document.write('A')
</script>
```

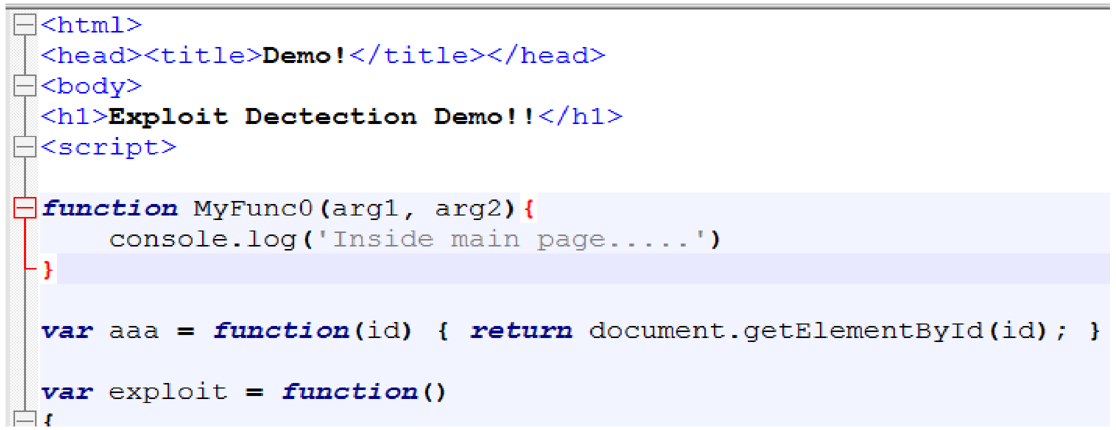
Exploit Detection: Use of ActiveXObject()



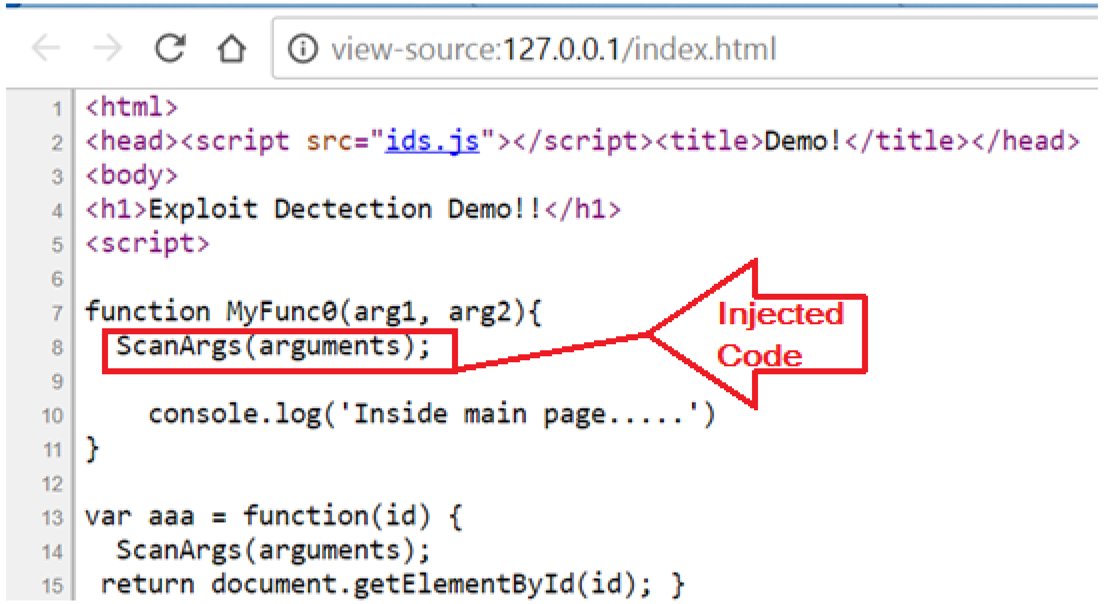
```
// IE : ActiveXObejct Hooking

var ActiveXObject_Real;
ActiveXObject_Real = ActiveXObject;
ActiveXObject = function(obj_name)
{
    log(obj_name); // Inserted logging routine for tes
    return_obj = new ActiveXObject_Real(name);
    return return_obj;
}

var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
```

Hooking User-Defined JavaScript Functions

- To hook a custom user-defined JavaScript function, the detection device should be able to identify functions in an HTTP response and inject a line of code into it.



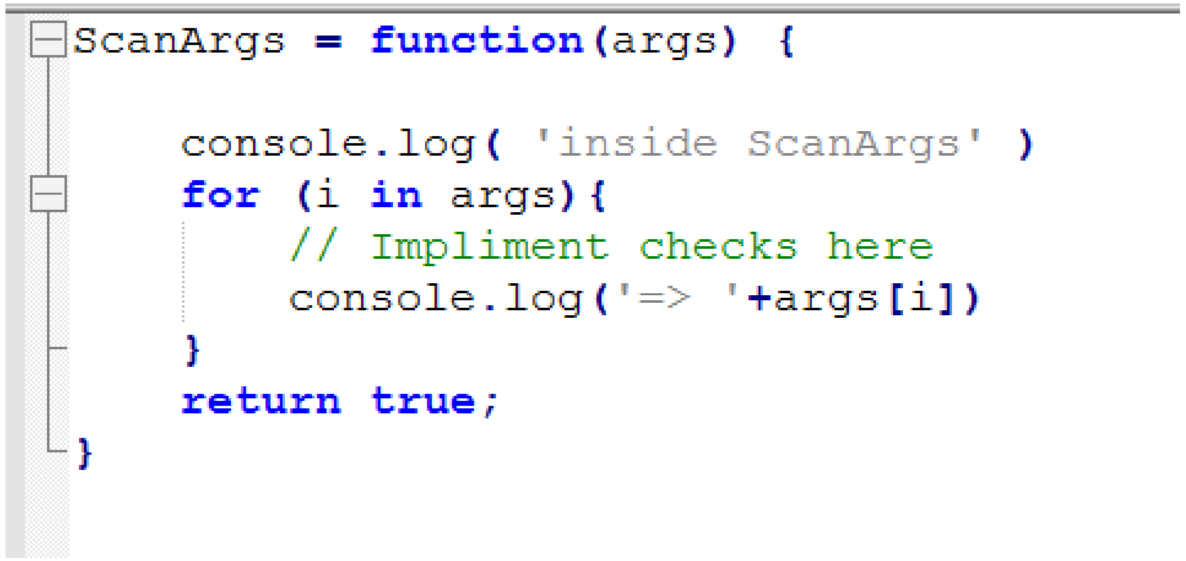
```
<html>
<head><title>Demo!</title></head>
<body>
<h1>Exploit Dectection Demo!!</h1>
<script>

function MyFunc0(arg1, arg2){
    console.log('Inside main page.....')
}

var aaa = function(id) { return document.getElementById(id); }

var exploit = function()
{
```



```
<html>
<head><script src="ids.js"></script><title>Demo!</title></head>
<body>
<h1>Exploit Dectection Demo!!</h1>
<script>

function MyFunc0(arg1, arg2){
  ScanArgs(arguments);

    console.log('Inside main page.....')
}

var aaa = function(id) {
  ScanArgs(arguments);
 return document.getElementById(id); }
```

Hooking Custom Functions (Inspecting Passed Arguments)

- The injected line of code passes **arguments** to the function ScanArgs().
- **“Arguments”** is a list that holds the arguments passed to any function.
- The example ScanArg() function performs several integrity checks to determine if an attack is in place.

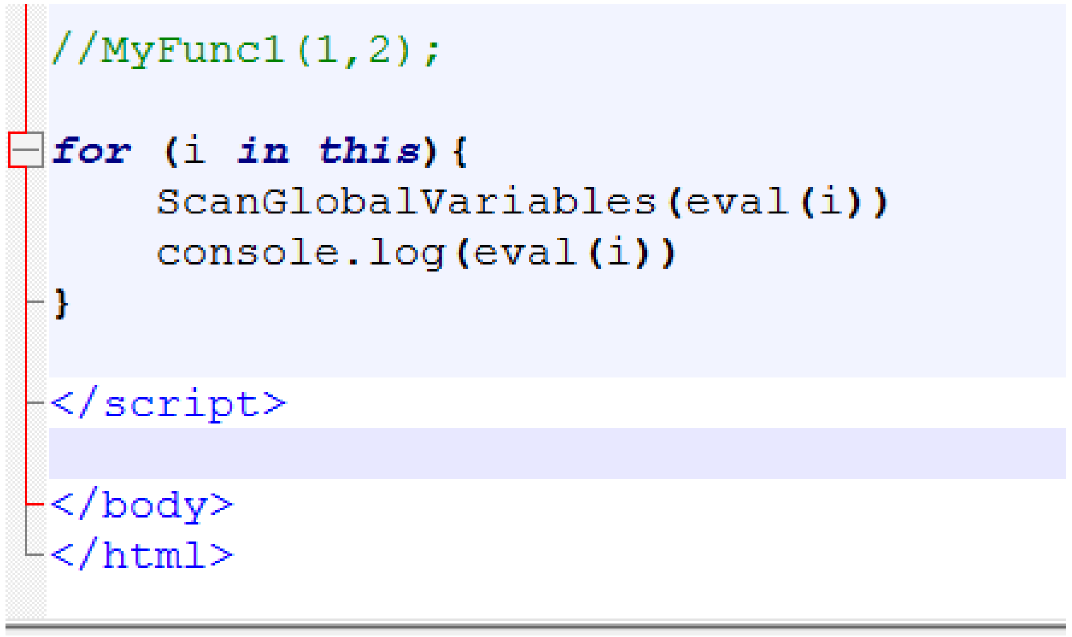


```
ScanArgs = function(args) {

    console.log( 'inside ScanArgs' )
    for (i in args){
        // Impliment checks here
        console.log('=> '+args[i])
    }
    return true;
}
```

Demo 3: Hooking Custom JavaScript Functions

Inspecting Strings Declared as Global Variables

- The detection system can injects a tiny piece of code in <script> blocks.
- In the global execution context (outside of any function) this refers to the global object.
- The code simply iterates "**this**" in any instance on a page and performs a few integrity checks.
- Some filtering is required because iterating **"this"** gives us a lot of unnecessary variables.



```
//MyFunc1(1,2);

for (i in this){
    ScanGlobalVariables(eval(i))
    console.log(eval(i))
}

</script>

</body>
</html>
```

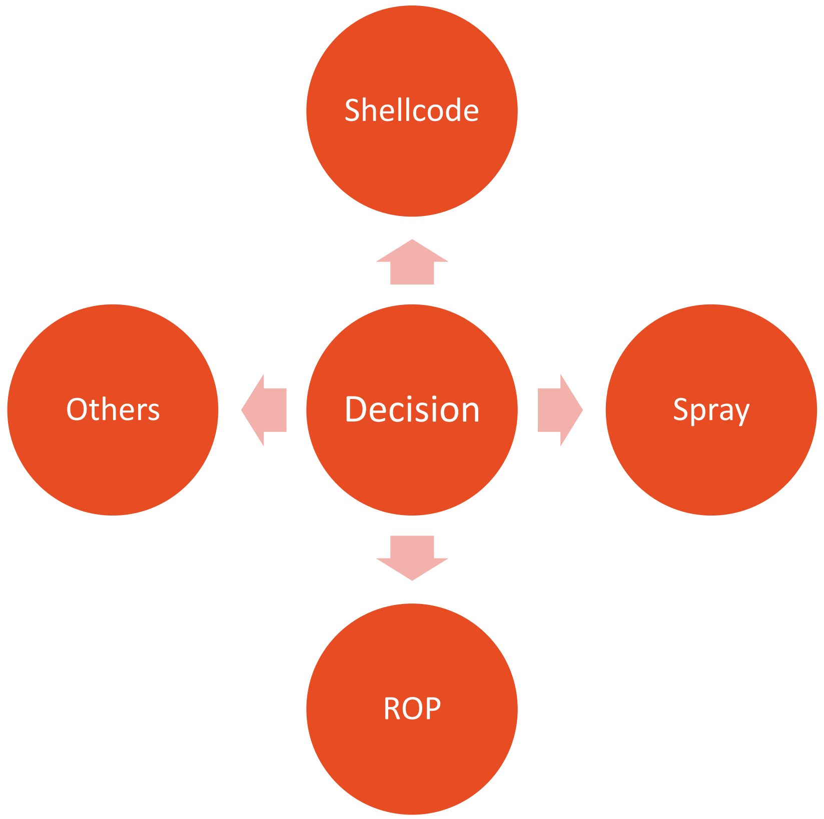
Demo 4: Catching Variables Declared as Global Variables

Anomaly Detection

- Suspicious API call sequence.
- Unusual API call count.
- Etc.

Combining Everything

- Once the subject web page goes through several stages of our hooking routine, the detection system has to decide whether the page is malicious in nature.
- The detection system can make that decision based on positive or negative results of several checks discussed earlier.



Prevention and Reporting

- If the injected code finds anything malicious, it tries to grab all the page content (both HTML and JavaScript).
- Sends results to separate logging servers, where they can be verified as false positive or real exploit.
- Once content is logged, the code immediately stops the page and prevents it from further loading.
- There are several ways a page can be stopped. The system uses window.location = "about:blank" to flush the page content.
- Or a simple HTTP redirect will do the job.

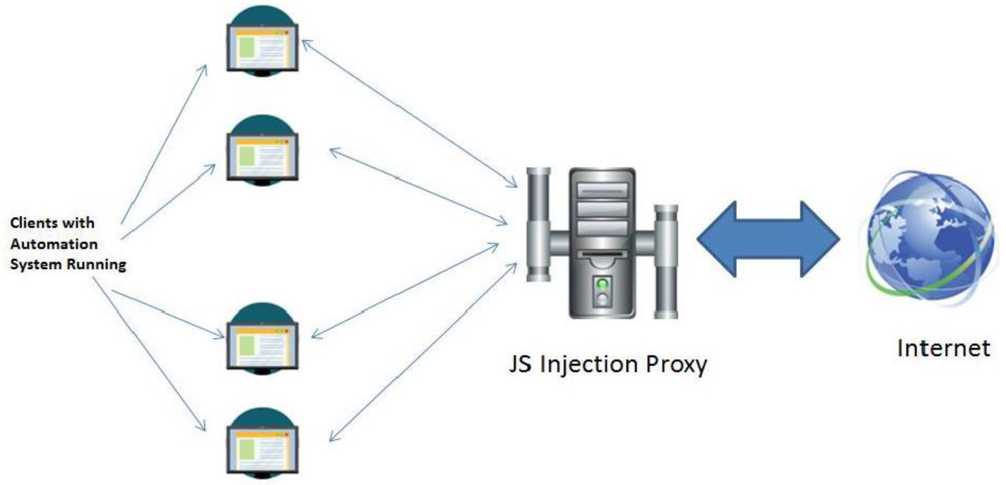
Making the System Smarter

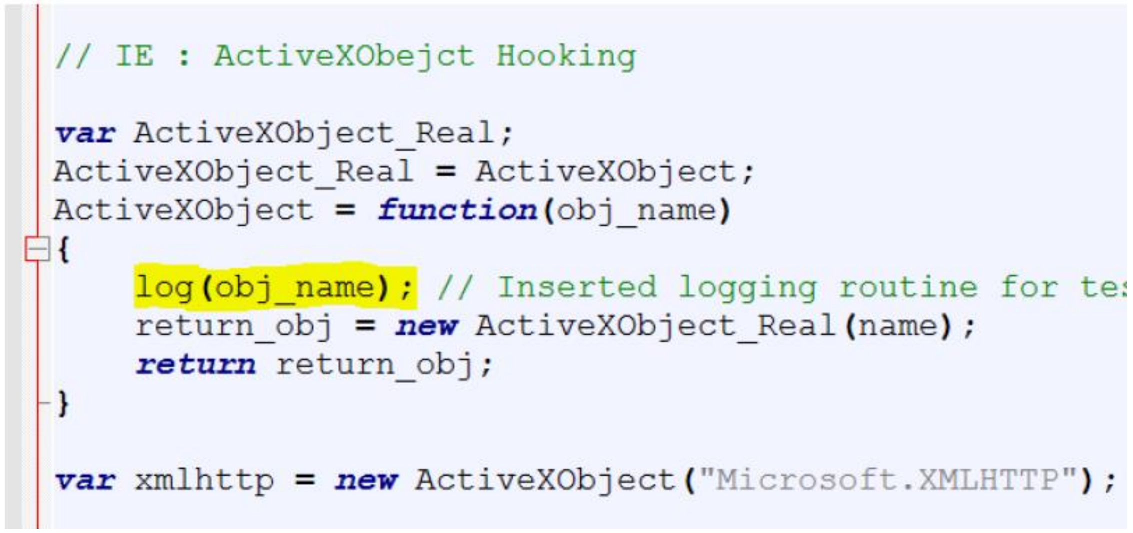
Making the System Smarter: Adding More Intelligence, Getting Rid of False Positives

- Because we are hooking JavaScript APIs—which are also used by legitimate web apps—the chances of false positives are very high.
- When the prototype was tested with real-world web traffic, the results were full of false positives and broke many web applications.
- Debugging such false-positive errors in a large-scale deployment is pretty difficult.

Making the System Smarter, Continued

- An automation system was developed.
- Once a list of websites was fed into the automation, the system recursively crawls the sites using Internet Explorer.
- To see how the injected code is reacting to real-world web apps, we modified our hooking routines by injecting logging routines into almost every step.

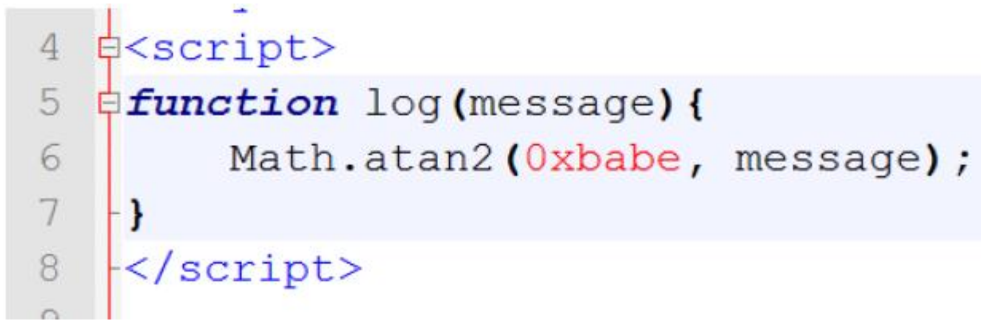




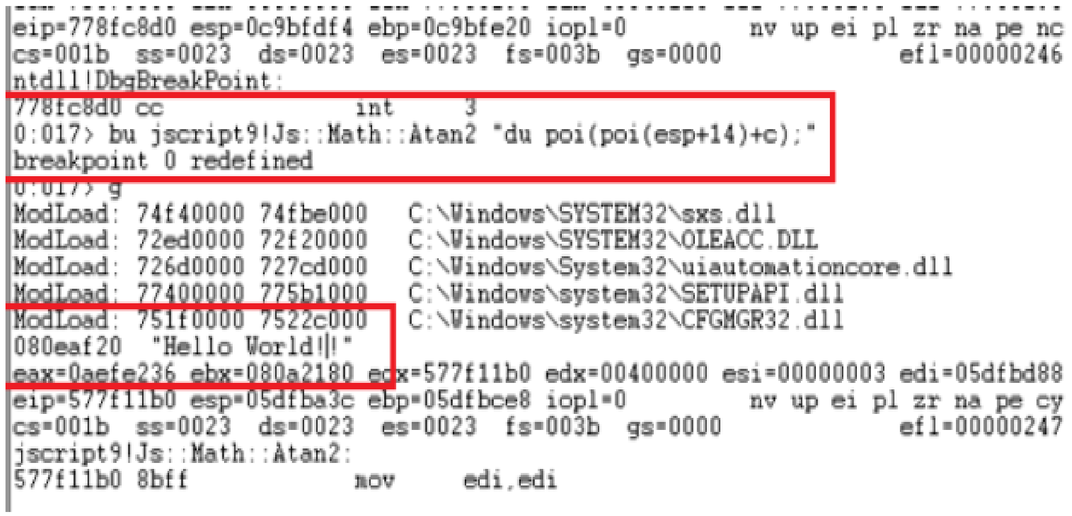
```
// IE : ActiveXObejct Hooking

var ActiveXObject_Real;
ActiveXObject_Real = ActiveXObject;
ActiveXObject = function(obj_name)
{
    log(obj_name); // Inserted logging routine for te
    return_obj = new ActiveXObject_Real(name);
    return return_obj;
}

var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
```

Making the System Smarter, Continued

- To catch and save the logging messages passed by the injected JavaScript hooking routine, we made some changes in Internet Explorer.
- The **log()** function uses the **Math.atan2** function.
- A custom DLL in every instance of Internet Explorer hooks (by offset) into **jscript9!Js::Math::Atan2** and intercepts arguments passed to this function.



```
<script>
function log(message){
        Math.atan2(0xbabe, message);
}
</script>
```



```
eip=778fc8d0 esp=0c9bfdf4 ebp=0c9bfe20 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
778fc8d0 cc              int     3
0:017> bu jscript9!Js::Math::Atan2 "du poi(poi(esp+14)+c);"
breakpoint 0 redefined
0:017> g
ModLoad: 74f40000 74fbe000   C:\Windows\SYSTEM32\sxs.dll
ModLoad: 72ed0000 72f20000   C:\Windows\SYSTEM32\OLEACC.DLL
ModLoad: 726d0000 727cd000   C:\Windows\System32\uiautomationcore.dll
ModLoad: 77400000 775b1000   C:\Windows\system32\SETUPAPI.dll
ModLoad: 751f0000 7522c000   C:\Windows\system32\CFGMGR32.dll
080eaf20  "Hello World!|!"
eax=0aefe236 ebx=080a2180 ecx=577f11b0 edx=00400000 esi=00000003 edi=05dfbd88
eip=577f11b0 esp=05dfba3c ebp=05dfbce8 iopl=0         nv up ei pl zr na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000247
jscript9!Js::Math::Atan2:
577f11b0 8bff            mov     edi,edi
```

Some Key Findings

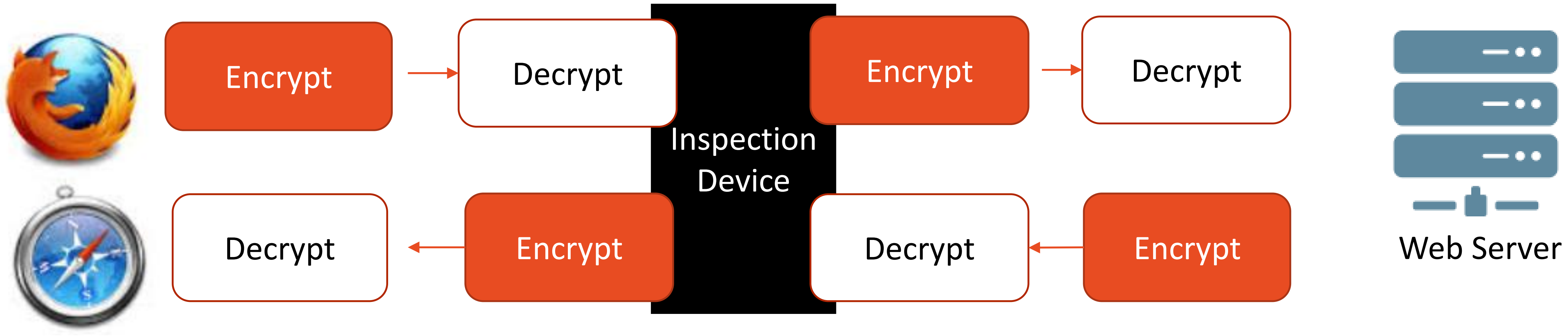
- To test false positives, we used legitimate websites. To test false negatives real browser exploits were used.
- The Array() hooking routines proven to be the most powerful.
- When tested against the Metasploit framework, we found unescape() and escape() hooking routines catch most Metasploit exploits because they are widely used.
- Few other heuristics-based routines (such as API call count, suspicious call sequence etc.) catch some exploits found in wild.

Using the System as a Browser Plug-In

- The JavaScript injection into the subject web page is the backbone of the exploit detection system.
- JavaScript can be injected in many ways.
- Web browser plug-ins can also be used to inject intrusion detection system JavaScript code into a web page.
- However using it as plug-ins reopen the problem of installing agents on the endpoint.

Dealing with HTTPS

- SSL inspection at the corporate gateway is an old-school technique.
- This detection system can be integrated with any network inspection device capable of decrypting HTTPS.



Advantages Over Current Signature-Based Detection Systems

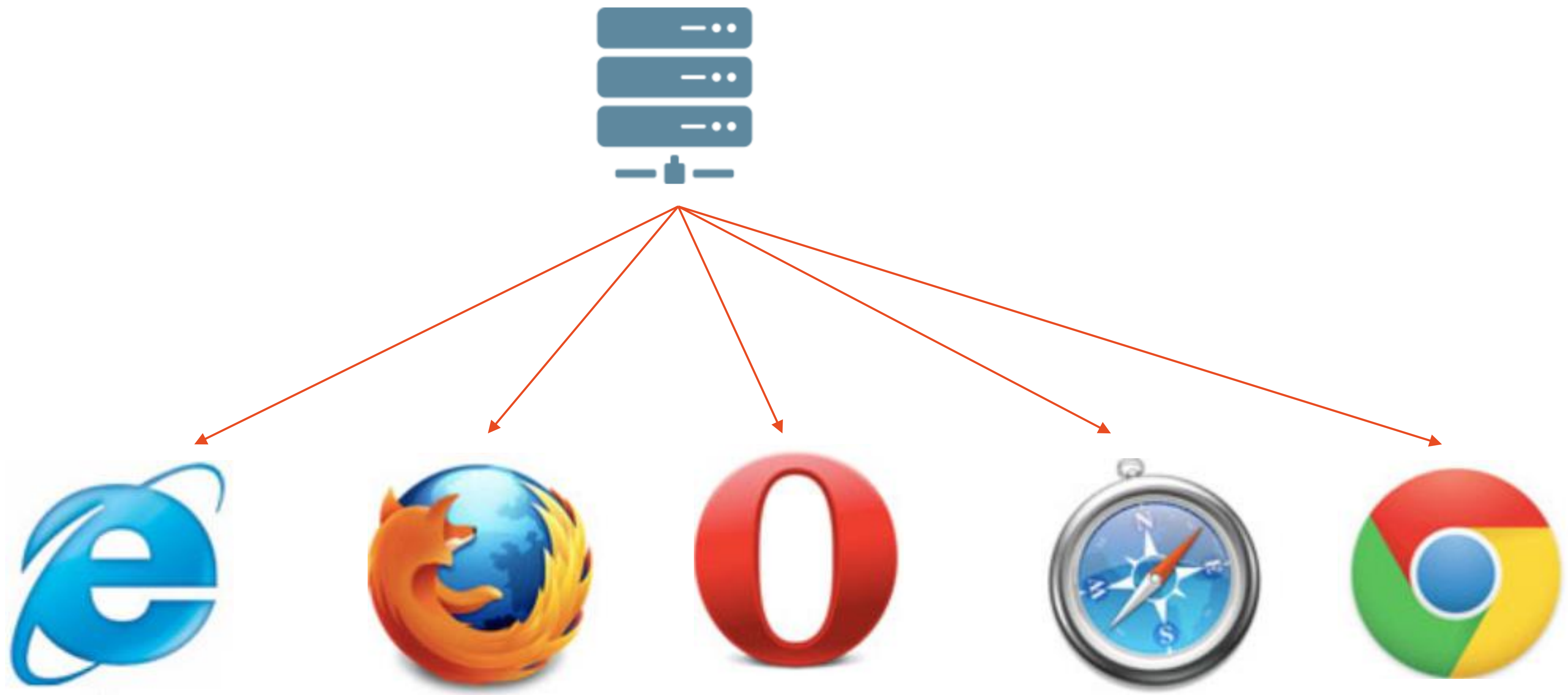
Advantages: Generic in Nature

- Browser exploits are very dynamic in nature.
- One exploit can be written (or obfuscated) in many ways.
- Unlike other signature-based exploit detection systems, this system does not catch exploits based on already known tokens.

Advantages: Agentless

- In any corporate environment, deploying a system behavior monitoring agent is quite challenging.
- However, with this solution we can closely monitor browser behavior with no agent required on endpoints.

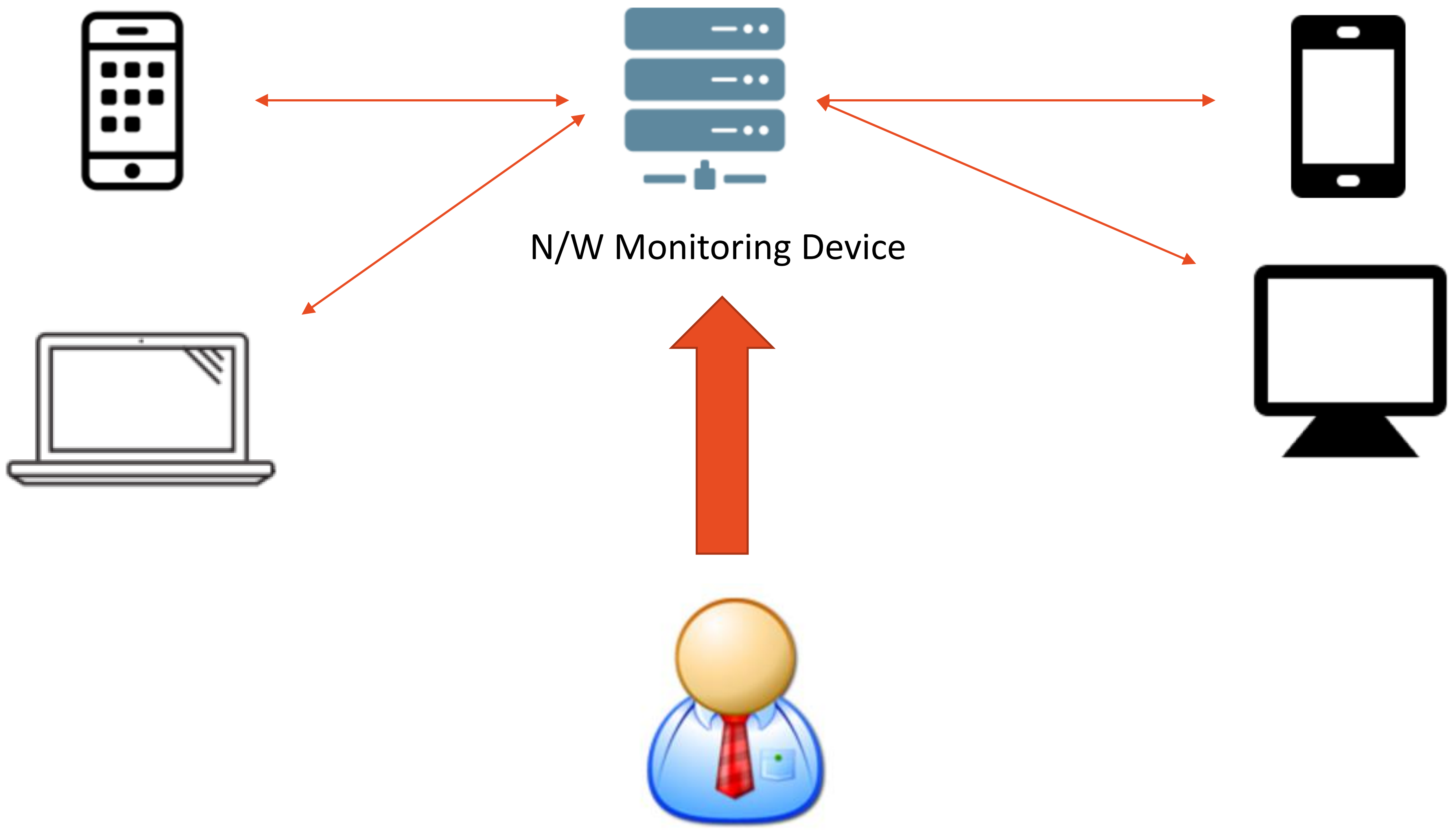
Advantages: Improves Network Monitoring Device Performance



Advantages: Platform Independent

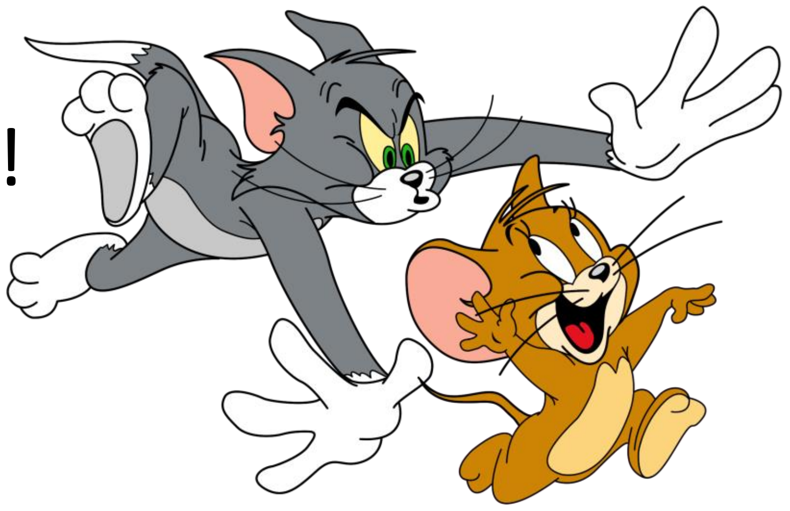
- JavaScript is <3.
- The core exploit detection logic is written and based on JavaScript. The solution can be considered and completely platform independent.

Advantages: Easy Update Shipments



Limitations and Bypasses

- Security is a cat-and-mouse game.
- Anti-JavaScript API hooking
- Anti-anti-JavaScript API hooking
- ..
- ..
- And it goes on...Cheers to JavaScript Ninjas!



Closing Remarks

- Endpoints are becoming more powerful everyday.
- JavaScript is a beautiful, powerful, and flexible language.
- JavaScript is the backbone of our exploit detection, which makes the system very powerful.
- On the other hand, this method gives attackers a lot of power to overcome security measures.

Demo 5

- Demonstration of a real browser exploit detection found in the wild.

Major References

- https://www.oomphinc.com/notes/2009/03/javascript-events-runtime/
- JavaScript Hooking as a Malicious Website Research - Liran Englender and Kris Kaspersky

Thank you ☺

- Special thanks to Bing Sun and Krish Patil for their valuable suggestions.
- Thanks to Dan Sommer for his help with the slides.
- https://twitter.com/debasishm89
- https://github.com/debasishm89