UPS: Observations on CVE-2015-3113, Prior Zero-Days and the Pirpi Payload

By Robert Falcone, Richard Wartell

Published: 2015-07-27 · Archived: 2026-04-02 12:20:58 UTC

A June 23 FireEye blog post titled "Operation Clandestine Wolf" discussed a cyber espionage group, known as APT3, that had been exploiting a zero-day vulnerability in Adobe Flash. Unit 42 also tracks the APT3 group using the name UPS, which is an intrusion set with Chinese origins that is known for having early access to zero-day vulnerabilities and delivering a backdoor called Pirpi.

The UPS group has exploited several zero-day vulnerabilities, most recently using the zero-days released in the Hacking Team breach that we discussed in our July 10 blog post, "APT Group UPS Targets US Government with Hacking Team Flash Exploit". However, the most recent original zero-day released by this group is tracked by CVE-2015-3113, which has similarities to the once zero-day vulnerabilities CVE-2014-1776 and CVE-2014-6332 exploited by UPS in May and November 2014, respectively. We'll discuss here the similarities observed between the various components used to exploit these two vulnerabilities, specifically focusing on the malicious Flash files and the payloads delivered.

Malicious Flash Files

Recent zero-day vulnerabilities exploited by UPS exploit or leverage Adobe Flash to exploit other applications on the system. Unit 42 recently analyzed malicious Flash files that exploited CVE-2015-3113, which was a zero-day vulnerability in Adobe Flash that was patched on June 23, 2015. During the analysis, we noticed similarities between this malicious Flash file, those that UPS used to exploit CVE-2014-1776, and the proof-of-concept code for CVE-2014-6332, albeit these two Flash files were used to exploit zero-day vulnerabilities in Internet Explorer.

Overlaps within ActionScript

Unit 42 analyzed the ActionScript within malicious Flash files created by UPS that exploited CVE-2014-1776 and CVE-2015-3113 and discovered shared code between the two. First, both ActionScripts contain a function named "hexToIntArray", which Figure 1 displays side-by-side for comparison. Not only do these files contain the same function name, but they also share the same exact operation codes (opcodes) to carry out its functionality. The existence of the hexToIntArray function in the CVE-2015-3113 sample is rather interesting, as it is never called or used within the ActionScript. We believe that the threat actor used the CVE-2014-1776 ActionScript as the basis for the CVE-2015-3113 file and forgot to remove the unused hexToIntArray function.

CVE-2014-1776	CVE-2015-3113
function private::hexToIntArray(String):__AS3__.vec::Vector.	function private::hexToIntArray(String):__AS3__.vec::Vector.

```
<int>
{
0    getlocal0
1    pushscope
2    pushnull
3    coerce_a
4    setlocal2
5    getlocal1
6    getproperty    length
8    coerce_a
9    setlocal3
10   pushbyte       0
12   coerce_a
13   setlocal       4
15   getlex         Vector
17   getlex         int
19   applytype      (1)
21   getlocal3
22   pushbyte       2
24   divide
25   construct      (1)
27   coerce         __AS3__.vec::Vector.<int>
29   setlocal       5
31   pushbyte       0
33   coerce_a
34   setlocal       6
36   jump           L1L2:
40   label
41   getlocal1
42   getlocal       4
44   callproperty
http://adobe.com/AS3/2006/builtin::charAt (1)
47   getlocal1
48   getlocal       4
50   pushbyte       1
52   add
53   callproperty
http://adobe.com/AS3/2006/builtin::charAt (1)
56   add
57   coerce_a
58   setlocal2
59   getlocal       5
```

```
<int>
{
0    getlocal0
1    pushscope
2    pushnull
3    coerce_a
4    setlocal2
5    getlocal1
6    getproperty    length
8    coerce_a
9    setlocal3
10   pushbyte       0
12   coerce_a
13   setlocal       4
15   getlex         Vector
17   getlex         int
19   applytype      (1)
21   getlocal3
22   pushbyte       2
24   divide
25   construct      (1)
27   coerce         __AS3__.vec::Vector.<int>
29   setlocal       5
31   pushbyte       0
33   coerce_a
34   setlocal       6
36   jump           L1L2:
40   label
41   getlocal1
42   getlocal       4
44   callproperty
http://adobe.com/AS3/2006/builtin::charAt (1)
47   getlocal1
48   getlocal       4
50   pushbyte       1
52   add
53   callproperty
http://adobe.com/AS3/2006/builtin::charAt (1)
56   add
57   coerce_a
58   setlocal2
59   getlocal       5
```

61 getlocal 6	61 getlocal 6
63 findpropstrict parseInt	63 findpropstrict parseInt
65 getlocal2	65 getlocal2
66 pushbyte 16	66 pushbyte 16
68 callproperty parseInt (2)	68 callproperty parseInt (2)
71 setproperty null	71 setproperty null
73 getlocal 4	73 getlocal 4
75 pushbyte 2	75 pushbyte 2
77 add	77 add
78 coerce_a	78 coerce_a
79 setlocal 4	79 setlocal 4
81 getlocal 6	81 getlocal 6
83 pushbyte 1	83 pushbyte 1
85 add	85 add
86 coerce_a	86 coerce_a
87 setlocal 6L1:	87 setlocal 6L1:
89 getlocal 4	89 getlocal 4
91 getlocal3	91 getlocal3
92 iflt L2	92 iflt L2
96 getlocal 5	96 getlocal 5
98 returnvalue	98 returnvalue
}	}

Figure 1. Side-by-side comparison of opcodes in hexToIntArray functions

Also, the Flash file exploiting CVE-2015-3113 had a main class named "flappyMan". This class name was also used in the Flash file that Unit 42 analyzed and discussed in its November 26, 2014 blog titled "Addressing CVE-2014-6332 SWF Exploit", as well as the proof-of-concept (PoC) for CVE-2014-6332 that is now publicly available in exploit-related forums. According to FireEye's "Operation Double Tap", UPS exploited CVE-2014-6332 in its November 2014 attacks; however, UPS used a VBScript to exploit the vulnerability instead of a Flash file. While purely speculation, this overlap in class names between the CVE-2014-6332 PoC and the Flash file exploiting CVE-2015-3113 may suggest that UPS also used Flash files to exploit CVE-2014-6332.

Shellcode Similarities

As with most remote code execution vulnerabilities, UPS' malicious documents execute shellcode in the event of successful exploitation of either CVE-2014-1776 or CVE-2015-3113. The shellcode found in the UPS delivery documents exploiting both of these vulnerabilities are not the same, but have similarities worth noting.

First, the delivery documents share the same technique of locating API functions, which involves using the rotate right (ror 7 to be specific) instruction on the function name in kernel32.dll and checking it with a specific value. The use of the same rotate right algorithm results in several common constants, such as 0xC917432 that both

shellcodes use to locate LoadLibraryA. Second, both shellcodes use a similar method of creating the Unicode string “kernel32.dll”, seen in Figure 2. The shellcodes use the Unicode string and the same method to find the base address of the loaded kernel32.dll module from the LDR structures obtained from the process environment block (PEB). Third, both shellcodes have similar single byte XOR algorithms used to decrypt and later execute the functional payload.

```
mov     [ebp+var_30], 6Bh ; 'k'
mov     [ebp+var_2F], 0
mov     [ebp+var_2E], 65h ; 'e'
mov     [ebp+var_2D], 0
mov     [ebp+var_2C], 72h ; 'r'
mov     [ebp+var_2B], 0
mov     [ebp+var_2A], 6Eh ; 'n'
mov     [ebp+var_29], 0
mov     [ebp+var_28], 65h ; 'e'
mov     [ebp+var_27], 0
mov     [ebp+var_26], 6Ch ; 'l'
mov     [ebp+var_25], 0
mov     [ebp+var_24], 33h ; '3'
mov     [ebp+var_23], 0
mov     [ebp+var_22], 32h ; '2'
mov     [ebp+var_21], 0
mov     [ebp+var_20], 2Eh ; '.'
mov     [ebp+var_1F], 0
mov     [ebp+var_1E], 64h ; 'd'
mov     [ebp+var_1D], 0
mov     [ebp+var_1C], 6Ch ; 'l'
mov     [ebp+var_1B], 0
mov     [ebp+var_1A], 6Ch ; 'l'
mov     [ebp+var_19], 0
mov     [ebp+var_18], 0
mov     [ebp+var_17], 0
```

```
mov     [ebp+var_28], 6Bh ; 'k'
mov     [ebp+var_27], bl
mov     [ebp+var_26], 65h ; 'e'
mov     [ebp+var_25], bl
mov     [ebp+var_24], 72h ; 'r'
mov     [ebp+var_23], bl
mov     [ebp+var_22], 6Eh ; 'n'
mov     [ebp+var_21], bl
mov     [ebp+var_20], 65h ; 'e'
mov     [ebp+var_1F], bl
mov     [ebp+var_1E], 6Ch ; 'l'
mov     [ebp+var_1D], bl
mov     [ebp+var_1C], 33h ; '3'
mov     [ebp+var_1B], bl
mov     [ebp+var_1A], 32h ; '2'
mov     [ebp+var_19], bl
mov     [ebp+var_18], 2Eh ; '.'
mov     [ebp+var_17], bl
mov     [ebp+var_16], 64h ; 'd'
mov     [ebp+var_15], bl
mov     [ebp+var_14], 6Ch ; 'l'
mov     [ebp+var_13], bl
mov     [ebp+var_12], 6Ch ; 'l'
mov     [ebp+var_11], bl
mov     [ebp+var_10], bl
mov     [ebp+var_F], bl
```

Figure 2. Comparison of Instructions in UPS Shellcodes that Builds Kernel32.dll Unicode String

Steganography to Conceal Payloads

While analyzing the malicious Flash file exploiting CVE-2015-3113, Unit 42 discovered that the ActionScript loaded an animated GIF image. The malware author used steganography to embed an encrypted payload within this animated GIF image. The payload in the CVE-2014-1776 was also embedded within an animated GIF. Ultimately, the shellcode executed in the event of successful exploitation of either of these vulnerabilities decrypt and execute the embedded payload, as mentioned in the previous section. While the animated GIFs themselves are vastly different, as seen in Figure 3 and 4 (payloads removed), the use of steganography and animated images as the carrier of the payload is common between the two campaigns.

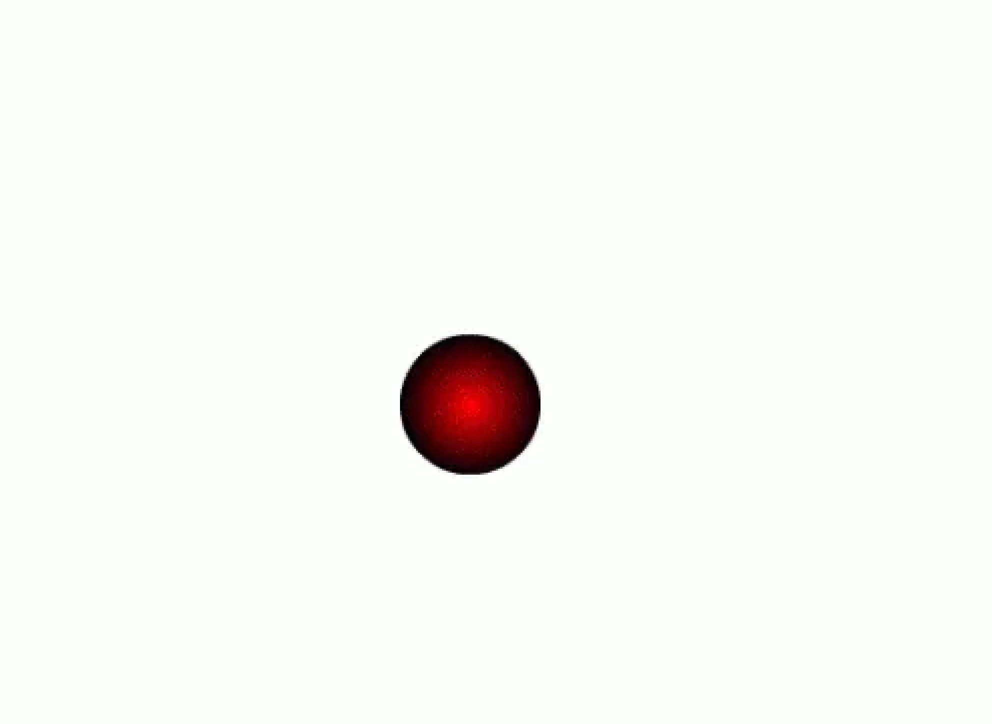

Figure 3. Animated GIF “v.gif” from UPS Campaign Exploiting CVE-2015-3113 (click to see .gif)

Figure 4. Animated GIF “anyway.gif” from UPS Campaign Exploiting CVE-2014-1776

Payload Comparison

With the amount of overlap between the other components in these separate campaigns, we decided to compare the Pirpi payloads delivered by the UPS group using CVE-2014-1776 and CVE-2015-3113. From here on, we will refer to these two payloads as Pirpi.2014 (CVE-2014-1776) and Pirpi.2015 (CVE-2015-3113), whose details are listed in Table 1. Unit 42 discovered several similarities between the two Pirpi variants, as well as a few equally important differences, both of which are worth discussing. We also compared the Pirpi.2014 and Pirpi.2015 payloads to other known Pirpi samples in an attempt to determine which variant they most closely resemble.

File Name	**File Type**	**Architecture**	**Size**
MD5			**Compile Time**
SHA256			
IePorxyv.dll (Pirpi.2014)	PE.DLL	X86	86016

B48E578F030A7B5BB93A3E9D6D1E2A83			04:29:14 00:44:04
81BD203EF3924BF497E8824ED5F224561487258FF3D8EE55F1E0907155FD5333			
{CVE-2015-3113 payload} (Pirpi.2015)	PE.DLL	X86	150528
1B0E6BA299A522A3B3B02015A3536F6F			06:07:15 01:51:27
0649A3DD632CDE57BC2E97B814BE81A7F45454FED2A73800DE476AA75CDBE8CD			

Table 1. File Details of Pirpi.2014 and Pirpi.2015 Samples

Similarities in C2 Communications

Both Pirpi variants perform an initial check to see if a configuration file exists at %APPDATA%\vcl.tmp or %TEMP%\vcl.tmp depending on the operating system. If it finds one, it decodes it and uses the configuration data it finds inside for C2 communication, otherwise it uses hardcoded C2 domains encoded inside the binary. The malware then creates threads to begin C2 communication.

The Pirpi.2014 and Pirpi.2015 payloads communicate with their C2 by issuing HTTP GET requests to the C2 domain hardcoded inside the payload or within its "vcl.tmp" configuration file. While the structure of the C2 URL differs between the two variants, both use the HTTP Cookie field to transmit data in encrypted form to the C2 domain. Figure 6 shows examples of C2 communications from Pirpi.2014 and Figure 7 shows communication with the C2 of Pirpi.2015 malware variants, both containing data within the Cookie field.

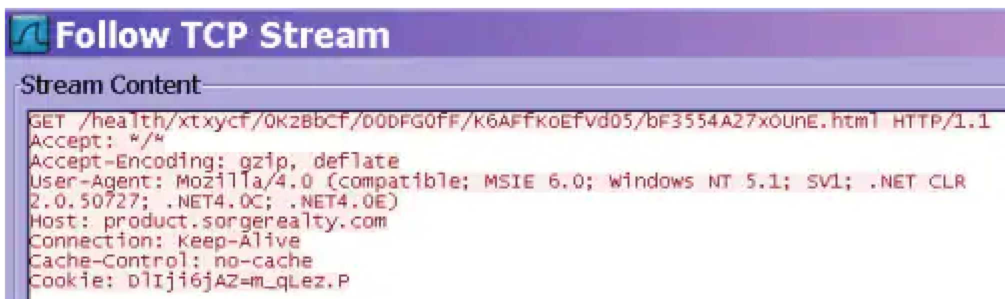

Figure 5. Pirpi.2014 C2 Communication using Cookie Field for Exfiltration

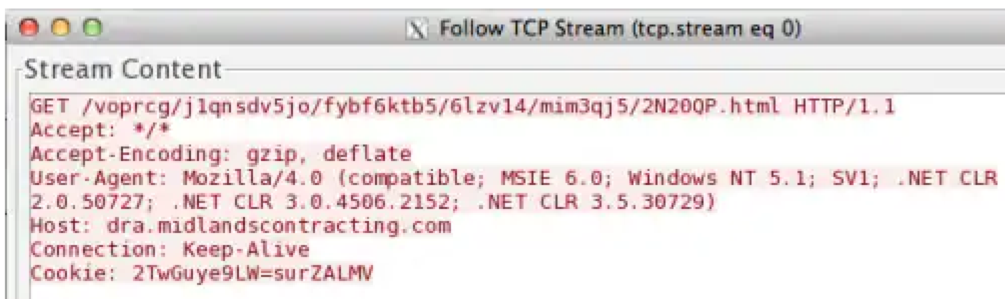

Figure 6. Pirpi.2015 C2 Communication using Cookie Field for Exfiltration

The GET request will return a web page that the malware will parse, specifically looking for encoded commands within two of the HTML tags.

Command Loop Overlap

Once the two Pirpi variants successfully communicate with their C2 server and parse the returned HTML for commands, Pirpi enters a command loop that processes the commands and carries out the respective activities. The command loop for the backdoor remains largely unchanged between Pirpi.2014 and Pirpi.2015 with only two of the commands differing between the two. Table 2 shows the commands that each malware can accept with only the 35 and 36 commands differing between the two Pirpi variants.

Value	Pirpi.2014 Command	Pirpi.2015 Command
1	Launch Process	
2	Process Listing	
3	Terminate Process	
4	Download a file from the C2, launch it, and then delete it	
5	Exit the malware	
6	Sleep	
7	Update C2 configuration and save it to %APPDATA%\vcl.tmp	
8	Download a file, load it into memory, then delete the file	
9	Load a DLL from %APPDATA% and execute one of its exported functions	
10	Do nothing	
11	Do nothing	
12	List all servers in the domain	
13	Get network adaptor information	
14	List TCP connection status (netstat)	
15	Retrieve information about connected users	
16	List servers in the primary domain	
17	Locates DCs on a domain	
32	Directory listing	
33	Upload a file to the C2	
34	Delete file	
35	Copy file and delete original	Copy file

36	Download and save file	Do Nothing
37	Echo	
38	Execute Process	
49	Get location of configuration file and set as current working directory	

Table 2. Commands Available within Pirpi.2014 and Pirpi.2015

Anti-Disassembly

The UPS threat group is a fan of one anti-disassembly trick that can be seen in both Pirpi.2014 and Pirpi.2015. It plays upon the order IDA Pro disassembles instructions. As you can see in the code sample in Figure 6 from Pirpi.2014 there is a "jump above" instruction, followed by a "jump below or equal" instruction which just falls through to the next instruction. This fall-through code path will never get executed since the jump occurs if 0x58693C96 > 0x0D7F31B4.

```
.text:10009127                 stc
.text:10009128                 mov     edi, 58693C96h
.text:1000912D                 cmp     edi, 0D7F31B4h
.text:10009133                 ja      short near ptr loc_1000913A+4
.text:10009135                 jbe     short $+2
.text:10009137
.text:10009137 loc_10009137:                           ; CODE XREF: .text:10009135↑j
.text:10009137                 mov     al, 0B8h
.text:10009139                 lodsd
.text:1000913A
.text:1000913A loc_1000913A:                           ; CODE XREF: .text:10009133↑j
.text:1000913A                 mov     ebx, 0C12F11D8h
.text:1000913F                 loope   loc_10009150
```

Figure 7. Code Showing Anti-Disassembly Technique used in Pirpi Tool

IDA Pro's disassembly sequence follows the fall-through branch of conditional jumps first, and thus in the previous instruction sequence, IDA keeps disassembling one instruction after another. When IDA goes back to disassemble the jump target for 0x10009133, it finds it pointing to the middle of an instruction. This stops IDA from being able to draw function borders, view a function in graph mode, or decompile with Hex-Rays. To solve this, undefine all of the code that will not be executed, and define code starting from the target of the conditional branch (in this case 0x1000913E), as seen in Figure 7.

```
.text:10009128                 mov     edi, 58693C96h
.text:1000912D                 cmp     edi, 0D7F31B4h
.text:10009133                 ja      short loc_1000913E
.text:10009133 ; ---------------------------------------------------------------
.text:10009135                 dd 0B8B00076h
.text:10009139                 dd 11D8BBADh
.text:1000913D                 db  2Fh ; /
.text:1000913E ; ---------------------------------------------------------------
.text:1000913E
.text:1000913E loc_1000913E:                           ; CODE XREF: .text:10009133↑j
.text:1000913E                 shl     ecx, 0Fh
.text:10009141                 not     bh
.text:10009143                 shr     ch, 13h
```

Figure 8. Fixing Anti-Disassembly Trick used by Pirpi Tool by Undefining Errant Instructions

You will now be able to create a function to improve your ability to do analysis. To make this easier, use an [IDA Pro script]() to fix these anti-disassembly tricks. Please note that this script specifically targets the anti-disassembly

used in Pirpi and other UPS samples. It may cause issues with malware that uses other anti-disassembly tricks. Use with caution.

Notable Differences

The first major difference between the Pirpi.2014 and Pirpi.2015 variants is in the way the command loop is executed in each backdoor. In Pirpi.2014, the malware uses a simple state machine that executes code blocks that correspond to a state value, which the malware updates at the end of each code block. Many of these code blocks include sleep functions, however, if the state value is set to the correct value, the malware executes a code block that contains the command loop. The purpose of this state machine is to intentionally delay the malware's execution of the command loop.

In Pirpi.2015, the malware implements a second state machine that executes the Pirpi.2014 state machine as one of its code blocks. The second state machine introduces a large number of randomized sleep functions, causing the malware to take much longer to execute its command loop. The majority of code blocks in the second state machine either sleep, or create threads and wait for them to finish. The malware author likely implemented these state machines as an anti-debugging technique and to defeat most modern sandbox solutions.

The second difference between the two Pirpi variants involves the encoding algorithm, which has improved greatly in the past year. Contained in the binary is an invertible math function for encoding and decoding of data. In Pirpi.2014 this function is rather simple, involving a few mathematical operations. However, in Pirpi.2015, the algorithm when decompiled is more than 300 source code lines of mathematical operations.

Other Pirpi Samples

FireEye released two reports in 2014 about APT3 phishing campaigns, Operation Doubletap and [Operation Clandestine Fox](). Each report containins md5s of other Pirpi samples that were available on VirusTotal. In addition, simple VirusTotal searches resulted in a few more Pirpi samples that came from the same code base. Table 3 contains the file information for each of these Pirpi samples.

<table>
<tr><th>File Name</th><th>File Type</th><th>Architecture</th><th>Size</th></tr>
<tr><th colspan="3">MD5</th><th rowspan="2">Compile Time</th></tr>
<tr><th colspan="3">SHA256</th></tr>
<tr><td>{FireEye Report Sample}</td><td>PE.EXE</td><td>X86</td><td>102400</td></tr>
<tr><td colspan="3">8849538EF1C3471640230605C2623C67</td><td rowspan="2">09:25:14
09:09:59</td></tr>
<tr><td colspan="3">854C6BA97B4BD01246AC6EF9258135D2337E6938676421131B6793ABF339FA94</td></tr>
<tr><td>msupd.dll</td><td>PE.DLL</td><td>X86</td><td>81920</td></tr>
<tr><td colspan="3">FA3578C2ABE3F37DDDA76EE40C5A1608</td><td rowspan="2">09:10:14
04:54:09</td></tr>
<tr><td colspan="3">CE7ACAE4CDB53C2FB526624855FC8E008608343B177DF348657295578312EB49</td></tr>
</table>

ieupd.dll	PE.DLL	X86	86016
1A4B710621EF2E69B1F7790AE9B7A288			05:27:14 08:48:13
12AE4A7072C95EAE0E433570B1D563C3D39FE3239816C04426C8E64A49BBE7D7			
IePorxyv.dll	PE.DLL	X86	86016
F4884C0458176AAC848A911683D3DEF5			04:29:14 00:45:45
8C64D673CB84F76124FDBDC76941396647FF03725BDDD1D59D0CD32D8EBAD81F			
IePorxyv.dll	PE.DLL	X86	81920
4CA97FF9D72B422589266AA7B532D6E6			04:29:14 00:32:43
4F677060D25A5E448BE986759FED5A325CD83F64D9FEF13FB51B18D1D0EB0F52			

Table 3. Details of Pirpi Samples from FireEye Reports and Samples that Share the Same Code Base

The sample listed as “{FireEye Report Sample}” in Table 3 is simply a dropper and loader for msupd.dll sample. Unit 42 compared all of the DLL samples listed in the table above and found that they are most closely related to Pirpi.2014. Table 4 below shows the statistics from Zynamics BinDiff from comparing each of the DLLs with Pirpi.2014 and Pirpi.2015.

Sample MD5	**Pirpi.2014 Bindiff**		**Pirpi.2015 Bindiff**	
	Similarity	**Confidence**	**Similarity**	**Confidence**
FA3578C2ABE3F37DDDA76EE40C5A1608	89.5%	98.6%	29.8%	69.5%
1A4B710621EF2E69B1F7790AE9B7A288	92.7%	98.8%	29.4%	69.5%
F4884C0458176AAC848A911683D3DEF5	91.4%	98.7%	29.6%	71.6%
4CA97FF9D72B422589266AA7B532D6E6	93.7%	98.7%	30.7%	71.6%
B48E578F030A7B5BB93A3E9D6D1E2A83	100%	100%	34.3%	73.0%
1B0E6BA299A522A3B3B02015A3536F6F	34.3%	73.0%	100%	100%

Table 4. Resulting Similarity and Confidence Rates of Pirpi Samples

Conclusion

The UPS threat group continues to exploit zero-day vulnerabilities in their campaigns, which shows that this group is quite sophisticated and has access to significant resources. Within their attack campaigns involving zero-days, UPS has consistently reused delivery techniques and code within various components of the attack. UPS has relied on steganography to conceal the payloads delivered after exploitation of zero-days by embedding payloads,

specifically the Pirpi backdoor within animated GIFs. This group also reuses portions of their ActionScript within their malicious Flash files used to exploit vulnerabilities, as well as sharing portions of shellcode that executes after exploitation.

In regards to similarities amongst payloads, UPS delivers variants of the Pirpi backdoor that are typically very similar to each other. The Pirpi backdoors we analyzed use the same configuration file, a common C2 communications channel and a similar command handler. Also, the author of Pirpi includes several notable fingerprints within the code, specifically using a unique state machine and anti-disassembly techniques. Organizations can use all of these overlaps and similarities to track and hopefully protect themselves from this advanced adversary. AutoFocus users can identify Pirpi payloads with the Pirpi tag (Figure 9). WildFire automatically classifies Pirpi samples as malicious and we have released IPS signature 14643 to detect Pirpi C2 communications.

Figure 9. Pirpi tag

Source: http://researchcenter.paloaltonetworks.com/2015/07/ups-observations-on-cve-2015-3113-prior-zero-days-and-the-pirpi-payload/