FLARE ON CHALLENGE SOLUTIONS: PART 2 OF 2

November 21, 2014 | by Richard Wartell, Mike Sikorski | Technology

The FireEye Labs Advanced Reverse Engineering (FLARE) team created and released the first FLARE On Challenge to the community in July. Last week we released Part 1 of the step-by-step solution series and showed how to solve Challenges 1-5.

Here in Part 2, we focus only on Challenges 6 and 7. They were more difficult so we thought they warranted their own blog post. Two members of the FLARE team took very different approaches to solving the Challenge 6, so we believe it would be helpful to the community to post both solutions. Before we dive into the solutions, we want to explain how we generated Challenge 6.

As a reminder, the goal of each challenge is to find a key in the form of an email address that allows you unlock the next challenge. The archive of challenges have been posted to the challenge website.

Stay tuned in early 2015 for details of the next FLARE On Challenge!

CHALLENGE 6 CREATION

For Challenge 6, known as “linhax,” a number of people expressed interest in how it was constructed. Based on the code in the binary it looks like we wrote a ridiculous amount of junk C++ code. We wrote a C++ obfuscation engine in Python that would create annoying, red herring C++ code that did nothing, over and over again, and randomly intersperse actual code throughout it. Malware authors sometimes use similar techniques during their build process to either hinder analysis by adding more code that needs to be reverse engineered, or trying to hinder signature and family classification rules. Figure 1 shows an example of the kind of C++ code our Python code would produce.

```
int int_4 = 8836178;
int int_5 = 9906622;
int int_6 = 9125925;
int int_7 = 5381201;
int int_8 = 5009290;
int int_9 = 5292660;
char char_0 = 124;
char char_1 = 99;
char char_2 = 53;
char char_3 = 124;
char char_4 = 87;
char char_5 = 41;
char char_6 = 124;
char char_7 = 73;
char char_8 = 60;
char* loc_string_0 = "mknod";
char* loc_string_1 = "0";
char* loc_string_2 = "SMTP";
char* loc_string_3 = "Heartbleed";
char char_temp = 0;
for(int i_388769 = 0; i_388769 < 4; i_388769++) {

}
char* temp_12790951 = string_A; b64shell[270] = temp_12790951[0]; // **EVIL**
return 14073; }
```

Figure 1: C code generated by Python

Depending on the number of lines of actually important code the binary needed to execute, the code would scale linearly. For this challenge, the generated .cpp file was 1.35MB in size.

CHALLENGE 6 SOLUTION BY WILLI BALLENTHIN

Here, I'd like to share how I solved Challenge 6. My technique was slightly different than most others since I recovered the key using only static analysis techniques.

Background

linhax is a statically linked 64-bit ELF executable file for GNU/Linux. IDA Pro identifies 2,373 functions, none of which the disassembler renames by default, and 1,450 strings. Many of the strings look interesting, such as external program names, URI fragments and vulnerability terms. After even a brief exploration, the astute reverse engineer will stumble across the function at virtual address 0x452079. Figure 2 shows an overview of the basic block graph for this function. Reversing, or rather, avoiding to reverse, this function is a major component of solving Challenge 6.

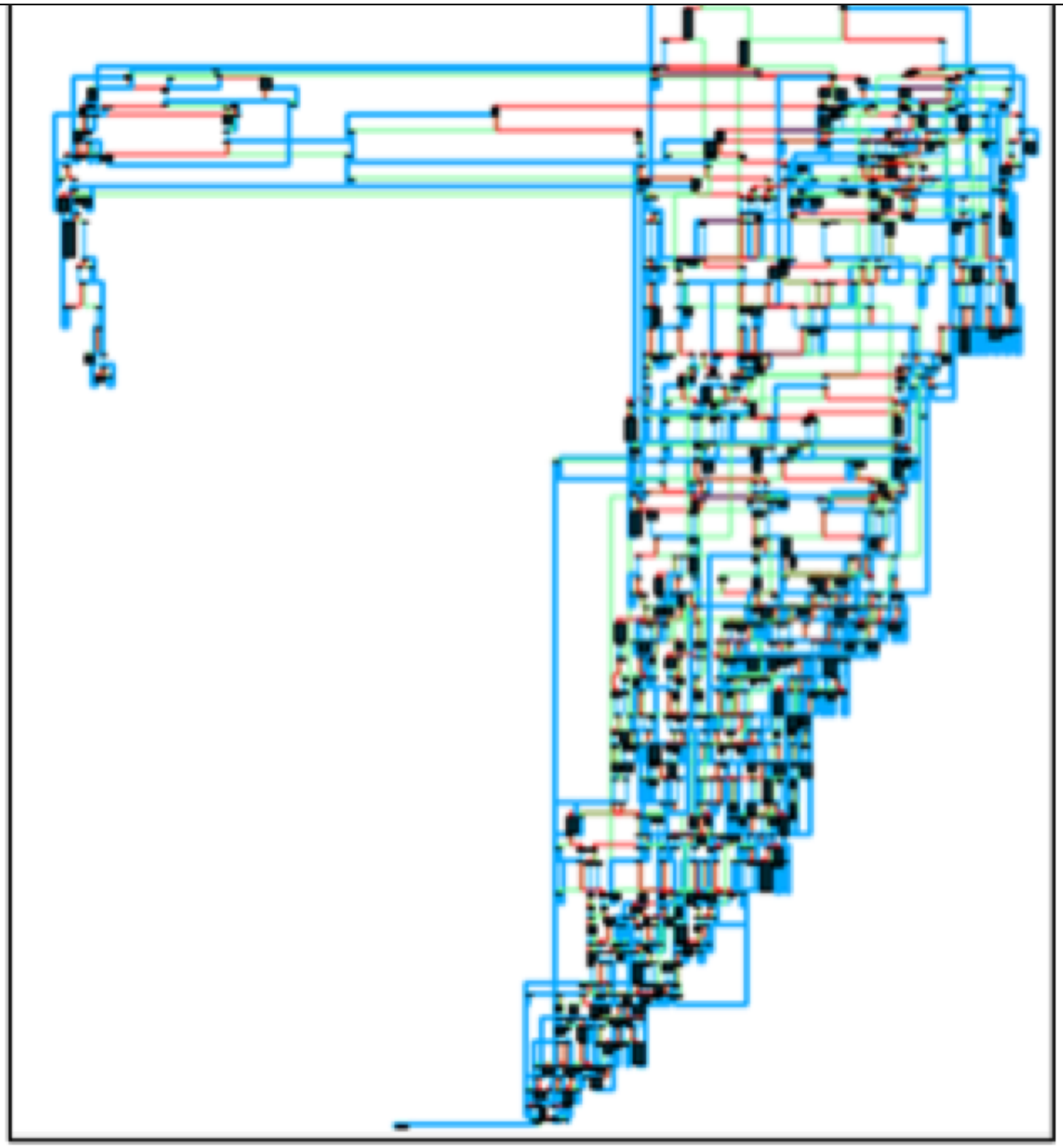

Figure 2: Graph overview of function at 0x45d19d

Part 1: Peeling back the first layer

To avoid directly analyzing this function I renamed it to “(not)_fun” and studied how the program uses its human-readable ASCII strings. Initially many string fragments looked relevant, but I quickly realized that the program does not display or manipulate most of the strings. The program typically fetches a single byte from a string or its address and stores it in a global byte array; however, reviewing the cross-references of the mov targets confirmed that the elements in the global array are often overwritten. Many elements are written to from one or two sources, as shown in Figure 3 and Figure 4. It seemed like these bytes are only written but never read.

Figure 3: Global byte array element with one cross reference

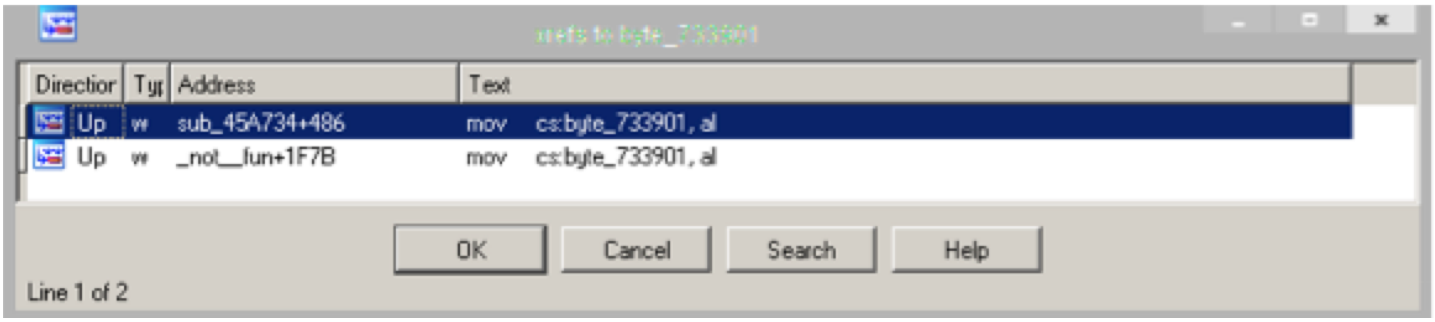

Figure 4: Global byte array element with two cross references

Upon closer inspection, I noticed that the first element of the global byte array has an additional cross reference. As show in Figure 5, the instruction at virtual address 0x44bb09 fetches and stores the address of the global byte array in the register EDI. This register is immediately used by the function at virtual address 0x401164, which becomes a prime candidate for further analysis.

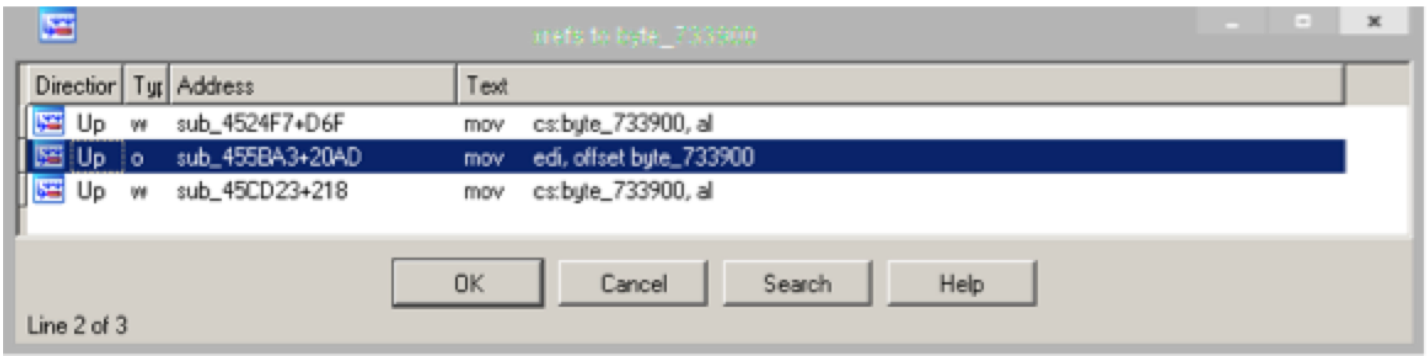

Figure 5: Global byte array element with three cross references

A short review of the function at virtual address 0x401164 should yield a suspicion that the function is a data encoding routine. Its four parameters are accessed throughout a for loop, possibly as an input and output buffer. Additionally I noticed the instruction “shl edx, 6” which I've most commonly seen in Base64-like encoding routines. Could this function post-process the final contents of that global byte array?

To determine the contents of the global byte array, I studied the two manners in which the program stored data into each element. In the first case, as show in Figure 6, the program fetches the value of the first character in a string, and stores it into the array element. Every element in the global byte array is set one at a time using this technique.

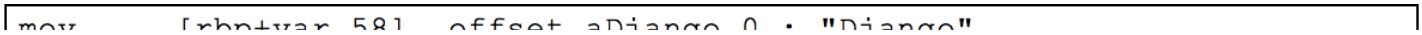

Figure 6: Storing a string's first character into an array element

The second case is more interesting. As shown in Figure 7, the program stores the first byte of the address of the string into the array element. This doesn't make much sense, since a programmer using their high-level language would probably not have control over the .text section layout, and therefore be unable to predict this byte value. Pair this with the fact that only some of the elements of the global byte array were set using this technique, and I hypothesized that this was a junk code component that I could ignore.

Figure 7: Storing a string's address into an array element

To test this theory I wrote an IDAPython script to compute the contents of the global byte array. You can review the entire script here. In essence, the script enumerated each cross reference to each element in the array looking for a sequence of instruction used in the "first case". Specifically, it tested for a global variable moved into a local variable, dereferenced, and then the first byte moved into the array element.

As I developed this script I found that there were no write-conflicts to any array elements, and ultimately, that the final contents appeared to be a Base64 string! Figure 8 shows the output of the script as it completes execution and dumps the contents of the global byte array.

```
('A', '0x41')
('=', '0x3d')
('0x733d0c', '0x405437L', 'not correct op type (1)')
SIn46AAAAABIixwkSIPDCusKSDHSSDHAsDwPBcAAWIAAnoAwzYAoOY
AA+8AIkoA4R3QC/+NIg8ABgCiZgACjgCgAgAAzgDiedAL/40iDwAHA
AJ+AKJaAON90Av/jSIPAAcAIccAIy4A4J3QC/+NIg8ABgACZgDgNdA
rACKmAMDzAAKOAMEuAOPt0Av/jSIPAAcAAcIAAjcAI0IA47nQC/+NI
gDiXdAL/40iDwAGAKHLAAD/ACHmAOL50Av/jSIPAAYAAFMAA34AoG4
jXgDhcdAL/40iDwAHAAGyAKJyAOEp0Av/jSIPAAYAogsAIuIAomMAI
SDH/SDH2SDHSTTHAagJfagFeagZaailYDwVJicBIMfZNMdJBUsYEJA
9qO1gPBZDr/pA=
```

Figure 8: Script output showing global byte array contents

Trying to decode the string as a Base64-encoded string does not generate any warnings, and results in mostly random-looking bytes. The illustration shows a hex dump of these results. Note, however, the ASCII string /bin/sh toward the end of the display. This string is fairly unlikely to appear in random or incorrectly decoded data, so I concluded that I'd unpeeled the first layer of Challenge 6.

```
0000280: 8030 fd80 382a 7402 ffe3 4883 c001 4831  .0..8*t...H...H1
0000290: c048 31ff 4831 f648 31d2 4d31 c06a 025f  .H1.H1.H1.M1.j._
00002a0: 6a01 5e6a 065a 6a29 580f 0549 89c0 4831  j.^j.Zj)X..I..H1
00002b0: f64d 31d2 4152 c604 2402 66c7 4424 029a  .M1.AR..$.f.D$..
00002c0: 02c7 4424 0409 3507 5648 89e6 6a10 5a41  ..D$..5.VH..j.ZA
00002d0: 505f 6a2a 580f 0548 31f6 6a03 5e48 ffce  P_j*X..H1.j.^H..
00002e0: 6a21 580f 0575 f648 31ff 5757 5e5a 48bf  j!X..u.H1.WW^ZH.
00002f0: 2f2f 6269 6e2f 7368 48c1 ef08 5754 5f6a  //bin/shH...WT_j
0000300: 3b58 0f05 90eb fe90                      ;X......
```

Figure 9: Hex dump of Base64 decode results

Part 2: Getting to the Center

By following the results of the Base64 decode function at virtual address 0x401164, I confirmed the appropriate interpretation of the decoded buffer as shellcode. The instruction at virtual address 0x44bb2b is an indirect call into the start of the decoded buffer, so I knew to begin disassembly at its first byte.

The shellcode consists of a sequence of byte-wise operations and comparisons against constants. The source appeared to be a user-supplied buffer from the parent program, and each byte in this buffer must satisfy a unique constraint. Figure 10 lists example checks against two of the bytes in the buffer. Trivially reversible operations composed these constraints, though I began to fear that I'd make a few hundred (error-prone) operations to hand-compute the solution bytes.

```
                add     byte ptr [rax], 28h ; '+'
                cmp     byte ptr [rax], 0B9h ; '¦' ; b9 - 2b + 39 ^ cd - 9e ror 58
                jz      short loc_32
                jmp     rbx
; ---------------------------------------------------------------------------

loc_32:                                 ; CODE XREF: seg000:000000000000002E↑j
                add     rax, 1
                xor     byte ptr [rax], 15h
                add     byte ptr [rax], 11h
                sub     byte ptr [rax], 99h ; 'Ö'
                cmp     byte ptr [rax], 9Ch ; '£'
                jz      short loc_46
                jmp     rbx
; ---------------------------------------------------------------------------
```

Figure 10: Two constraint checks

Instead, I developed a second IDAPython script to "reverse-emulate" the mathematic operations and solve for the expected bytes. You can review the entire script here. In summary, the script defines a reverse operation for each byte manipulation, dispatches to these routines based on the instruction mnemonics. I was surprised how few lines of Python code emulated the shellcode and solved for the expected bytes. The script runs in a fraction of a second, and generates the correct solution, as shown in Figure 11.

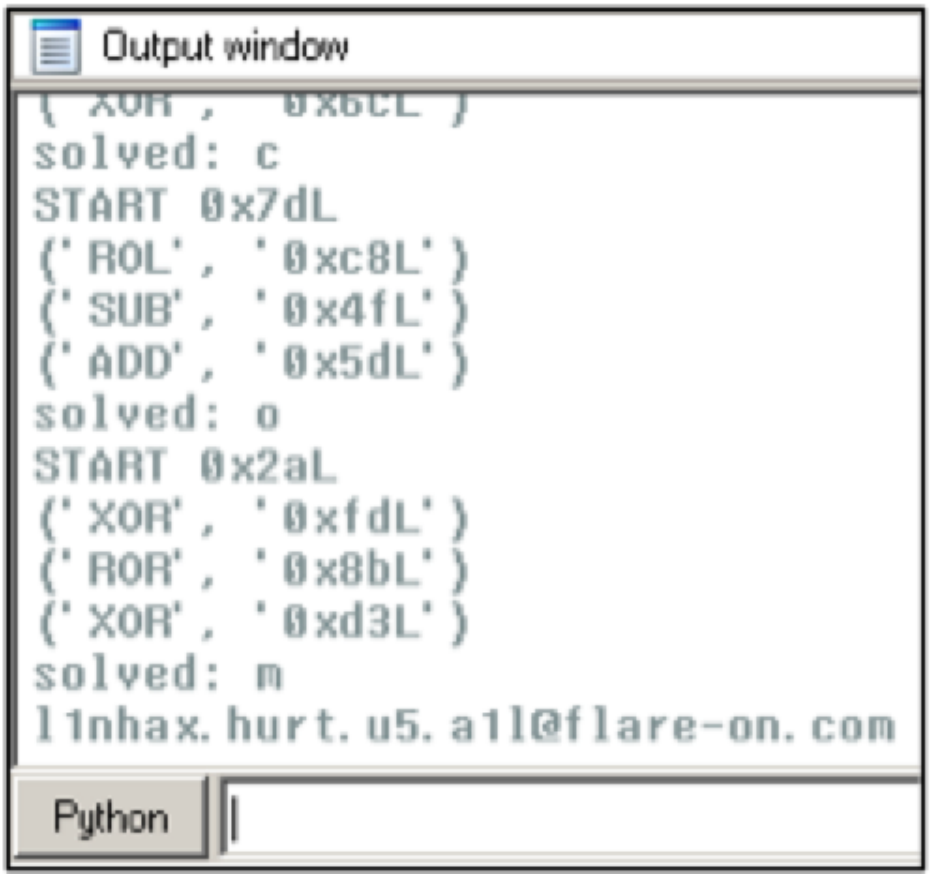

Figure 11: Solved output

CHALLENGE 6 SOLUTION BY BOB JUNG

Part 1: Figuring out the argument "situation"

After spending some time looking at the binary statically it was pretty obvious that I was getting nowhere quickly. The binary was heavily obfuscated and statically compiled, so I switched to debugging to see what I could get.

The first interesting part of the binary is that giving it any arguments causes the program to hang. Running it under strace solved that mystery pretty quickly as shown in Figure 12.

```
$ strace ./linhax derp
execve("./linhax", ["./linhax", "derp"], [/* 63 vars */]) = 0
uname({sys="Linux", node="ubuntu", ...}) = 0
brk(0)                                  = 0x19b0000
brk(0x19b11c0)                          = 0x19b11c0
arch_prctl(ARCH_SET_FS, 0x19b0880)      = 0
brk(0x19d21c0)                          = 0x19d21c0
brk(0x19d3000)                          = 0x19d3000
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL, [], 0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({3600, 0},
```

Figure 12: Running under strace

The sleep calls were easy enough to find in IDA and patch with a hex editor, which led to the next discovery that we see some interesting responses to different numbers of arguments as shown in Figure 13.

```
$ ./linhax_patched.nosleep
no
$ ./linhax_patched.nosleep derp
na
$ ./linhax_patched.nosleep derp derp
bad
$ ./linhax_patched.nosleep derp derp derp
stahp
$ ./linhax_patched.nosleep derp derp derp derp
stahp
```

Figure 13: Running with different arguments

By the looks of things, it seems like the program seems to care about the first two arguments. At this point I spent some time debugging in GDB and I found macros and watchpoints especially useful. Since the arguments to the program were obviously of interest, I was able to set watchpoints on both of them.

Breaking near the start of the program gives us access to the ARGV array which seems to be stored in $RSI. So for


```
(gdb) set $arg_addr = *(long*)($rsi+16)
(gdb) rwatch *$arg_addr
```

Figure 14: gdb commands

The watchpoint on the first argument made it easy to find the spot in the binary where the string was XORed with 0x56, compared with "bngcg`debd", and the program would terminate if they were different. This means that the first argument needs to be "bngcg`debd" ^ 0x56, which is “4815162342.”

The second argument is a completely different story.

Part 2: Second argument discovery

After some trial and error (and a lot of hitting “c” to continue), the second watchpoint eventually pays off and gets us to a place where we find the program executing on the stack. This is the shellcode discussed in the previous solution where each character in the second argument seems to be randomly shifted, added, subtracted, and XORed with seemingly random values before being compared to yet another value and then terminating if the values don’t match.

Figure 15 shows an example of the first two sets of instructions working on the first two characters.

```
0x7fffffffcf6c:    rolb   $0x58,(%rax)
0x7fffffffcf6f:    addb   $0x9e,(%rax)
0x7fffffffcf72:    xorb   $0xcd,(%rax)
0x7fffffffcf75:    subb   $0x39,(%rax)
0x7fffffffcf78:    addb   $0x2b,(%rax)
0x7fffffffcf7b:    cmpb   $0xb9,(%rax)
0x7fffffffcf7e:    je     0x7fffffffcf82
0x7fffffffcf80:    jmpq   *%rbx
0x7fffffffcf82:    add    $0x1,%rax
0x7fffffffcf86:    xorb   $0x15,(%rax)
0x7fffffffcf89:    addb   $0x11,(%rax)
0x7fffffffcf8c:    subb   $0x99,(%rax)
0x7fffffffcf8f:    cmpb   $0x9c,(%rax)
0x7fffffffcf92:    je     0x7fffffffcf96
0x7fffffffcf94:    jmpq   *%rbx
0x7fffffffcf96:    add    $0x1,%rax
```

Figure 15: Second argument access

After spending some time solving the first two on pen and paper, I realized that if I were to manually do this for all 30 characters it was going to take a long time. So as is often the case when analyzing programs it was time to try to do something clever and to write some code that would only have to execute correctly one time.

After spending some time thinking about how to solve this with code I decided an approach would be to start with the

each transformation, i.e., XOR, SUB, ROL, ROR, etc., the script stores a string of Python code that will cause the opposite transformation on the byte. Then, when we reach the end of each block, as indicated by the cmpb instruction, we print out all these instructions in reverse order.

```
#!/usr/bin/python
f = open("dis.txt")
print "from bitstring import BitStream"
insns = []
for line in f.readlines():
    if line.find('add    $0x1,%rax') != -1 and len(insns) > 0:
        #OK, hit the end of this block
        #print everything in reverse
        for insn in reversed(insns):
            print insn
        print "print \"%s (%s)\" % (str(chr(b.uint)), b.hex)"
        insns = []
    elif line.find("cmpb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b = BitStream(\'0x%02x\')" % b)
    elif line.find("xorb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b.uint ^= %s" % hex(b))
    elif line.find("addb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b.uint = (b.uint - %s) %% 256" % hex(b))
    elif line.find("subb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b.uint = (b.uint + %s) %% 256" % hex(b))
    elif line.find("rolb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b.ror(%s)" % hex(b))
    elif line.find("rorb") != -1:
        b = int(line.split('$')[-1].replace(',(%rax)', ''), 16)
        insns.append("b.rol(%s)" % hex(b))
```

Figure 16: Python code that writes code

So running the above script on the assembly results created the following script. Note that again for the sake of brevity I'm just showing the first two blocks that deal with the first two characters. I decided to use the BitStream class to represent the bytes since it gracefully lets you deal with rotate right/left, xor/or, add/subtract, etc. on a single byte. This Python module is pretty useful for doing weird calculations on bitstrings of arbitrary length.

```
b.uint ^= 0xcd
b.uint = (b.uint - 0x9e) % 256
b.ror(0x58)
print "%s (%s)" % (str(chr(b.uint)), b.hex)
b = BitStream('0x9c')
b.uint = (b.uint + 0x99) % 256
b.uint = (b.uint - 0x11) % 256
b.uint ^= 0x15
print "%s (%s)" % (str(chr(b.uint)), b.hex)
```

Figure 17: Example code output

After a couple tries it worked as seen in Figure 18! My software crime of writing code that writes yet more code seemed to do the trick, getting us the second argument and the email alias for this challenge!

```
a (61)
x (78)
. (2e)
h (68)
u (75)
r (72)
t (74)
. (2e)
u (75)
5 (35)
. (2e)
a (61)
1 (31)
l (6c)
@ (40)
f (66)
l (6c)
a (61)
r (72)
e (65)
- (2d)
o (6f)
n (6e)
. (2e)
c (63)
o (6f)
m (6d)
```

Figure 18: The key output

CHALLENGE 7

For Challenge 7, we wanted to test everyone's ability to deal with annoying anti-debugging, anti-VM and anti-disassembly checks. We see this all the time in malware that the FLARE team analyzes, so we threw a few of them into the binary before having it decode and launch another binary.

CHALLENGE 7 SOLUTION BY MATT GRAEBER

Challenge 7 contains 14 individual checks ranging from checking for the presence of a debugger, execution in a VM, execution at a certain date/time and the presence or absence of Internet connectivity. Each individual check decodes an embedded PE using a different multi-byte XOR key depending upon the execution environment.

Specifically, the executable checks the following conditions, and these are labeled in Figure 19:

2. Anti-debug – Check _PEB.BeingDebugged (equivalent to calling IsDebuggerPresent)

3. Anti-VM – Check if running in VMware specifically by using the SIDT instruction

4. Anti-VM – Check if running in VMware specifically by using the I/O port ‘VMXh’ trick

5. Anti-debug – OutputDebugString anti-debugger trick

6. Anti-debug – Check the count of (0xCC – int3) bytes in between two functions

7. Anti-debug – Check _PEB.NtGlobalFlag for the following flags associated with starting a process in a debugger: FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS

8. Temporal – Check if it’s Friday

9. Filename – Check if the filename is ‘backdoge.exe’

10. Internet connectivity – Check if www.dogecoin.com resolves to 127.0.0.1 or something else

11. Temporal – Check if the time is 1700 local

12. Filename – Use argv[0] as a 12 byte XOR key

13. Internet connectivity – Check if e.root-servers.net resolves. Since these IP addresses should not change, the expected resolved IP address should be 192.203.230.10. This IP treated as a multi-byte XOR key

14. Internet connectivity – Check for a magic string present in the @FireEye twitter account and use it as an XOR key

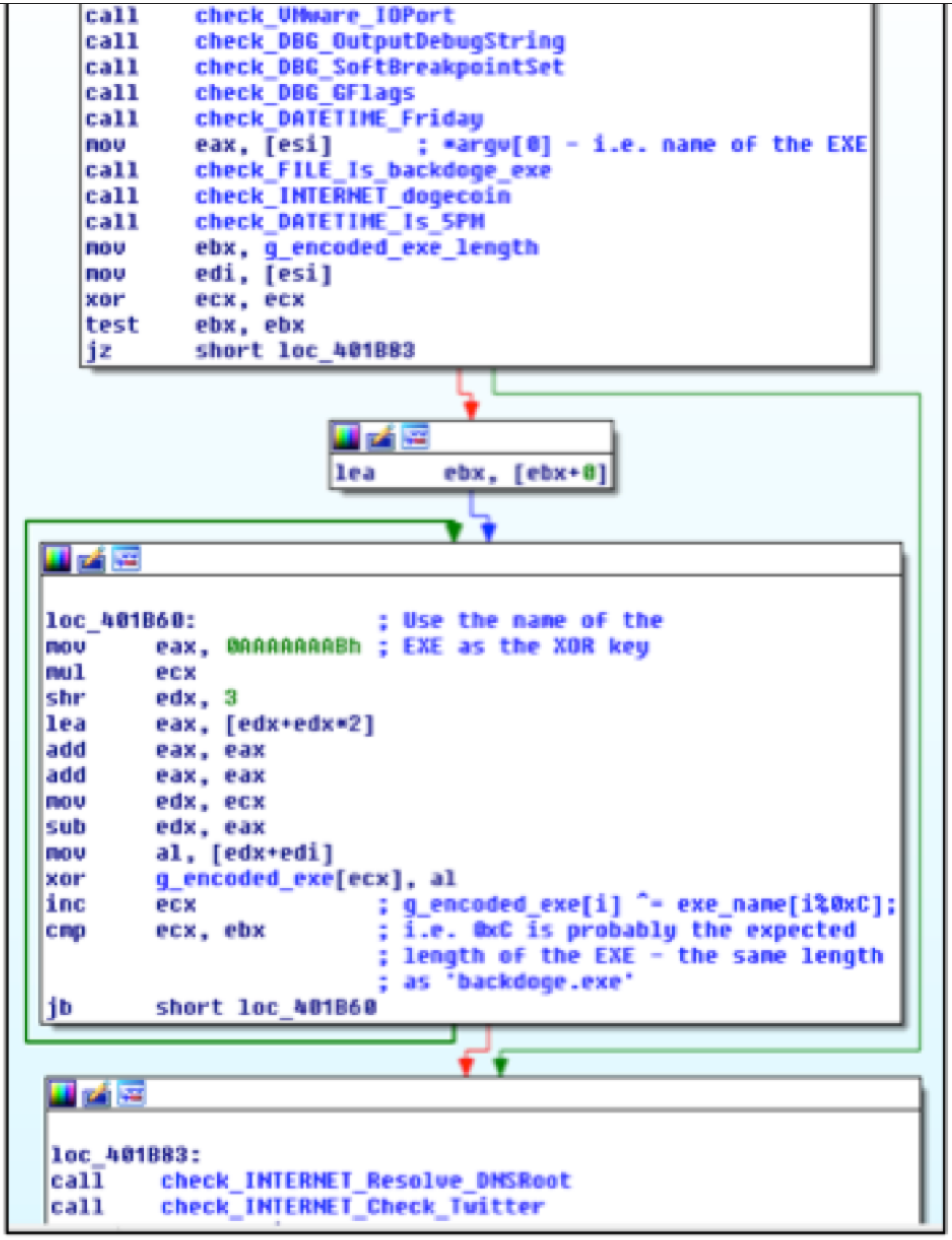
call check_VMware_IOPort
call check_DBG_OutputDebugString
call check_DBG_SoftBreakpointSet
call check_DBG_GFlags
call check_DATETIME_Friday
mov eax, [esi] ; *argv[0] - i.e. name of the EXE
call check_FILE_Is_backdoge_exe
call check_INTERNET_dogecoin
call check_DATETIME_Is_5PM
mov ebx, g_encoded_exe_length
mov edi, [esi]
xor ecx, ecx
test ebx, ebx
jz short loc_401B83
lea ebx, [ebx+0]
loc_401B60: ; Use the name of the
mov eax, 0AAAAAAABh ; EXE as the XOR key
mul ecx
shr edx, 3
lea eax, [edx+edx*2]
add eax, eax
add eax, eax
mov edx, ecx
sub edx, eax
mov al, [edx+edi]
xor g_encoded_exe[ecx], al
inc ecx ; g_encoded_exe[i] ^= exe_name[i%0xC];
cmp ecx, ebx ; i.e. 0xC is probably the expected
; length of the EXE - the same length
; as 'backdoge.exe'
jb short loc_401B60
loc_401B83:
call check_INTERNET_Resolve_DNSRoot
call check_INTERNET_Check_Twitter

Figure 19: Challenge 7 checks labeled in IDA Pro

execution environment. If there are 14 checks and two XOR keys for each check, that means there are 2^14 (16384) possible environment configurations.

Here was my strategy to tackle the challenge:

1) Extract the embedded, encoded PE.

2) Read the first 0x30 bytes of the embedded PE. This will speed things up when decoding each permutation. Since the binary is greater than a megabyte, decoding the full binary every time becomes computationally exhausting.

3) Generate each permutation by counting from 0-16383, convert the number to binary, and use each bit as the XOR key to use for that particular round of decoding.

4) After each round of decoding, check the four bytes after the MZ signature for 0x90,0x00,0x03,0x00 (common for most PEs). I don't check for an MZ because the first two bytes of the decoded PE are overwritten by the first argument provided at the command line (as seen in Figure 20). I assumed that the first bytes wouldn't decode to MZ.

5) If a match is found, read in the entire encoded PE and decode it with the correct XOR key sequence.

```
loc_401883:
call    check_INTERNET_Resolve_DNSRoot
call    check_INTERNET_Check_Twitter
mov     ecx, [esi+4]
movzx   edx, byte ptr [ecx] ; Replace the first two bytes
                            ; of the decoded EXE with the
                            ; first argument provided at
                            ; the command line
mov     g_encoded_exe_base, dl
mov     eax, [esi+4]        ; argv[1]
mov     cl, [eax+1]
mov     g_MZ_signature_byte2, cl ; argv[1][0]
mov     edx, [esi+8]
mov     al, [edx]
mov     byte_413278, al ; argv[1][1]
mov     ecx, [esi+8]
movzx   edx, byte ptr [ecx+1]
lea     eax, [ebp+var_10]
push    offset aWb          ; "wb"
```

Figure 20: Argv being written into extracted file

The following is a PowerShell script I wrote to perform the brute-forcing:

```
$HostExePath = Resolve-Path 'd69650fa6d4825ec2ddeecdc6a92228d'
$FileOffset = 0x113F8
$Length = 0x106240
$HostExeBytes = [IO.File]::ReadAllBytes($HostExePath)
[IO.File]::WriteAllBytes("$PWD\encoded.bin", $HostExeBytes[$FileOffset..($FileOffset+$Length-1)])

# Only read in the first 0x30 bytes. This will save time when decoding.
$FilePath = Resolve-Path 'encoded.bin' # Resolve full path of 'encoded.bin'
[Byte[]] $OriginalPEBytes = Get-Content $FilePath -TotalCount 0x30 -Encoding Byte
$PEBytes = $OriginalPEBytes

# Simple multi-byte xor function
function Xor([String] $Key, [Byte[]] $EncodedBytes) {
    $KeyBytes = [Text.Encoding]::ASCII.GetBytes($Key)

    foreach ($i in 0..($EncodedBytes.Length - 1)) {
        $EncodedBytes[$i] = $EncodedBytes[$i] -bxor $KeyBytes[$i % $KeyBytes.Length]
    }
}

# Used to convert non-printable characters to a string
$Enc = [Text.Encoding]::ASCII

# XOR keys extracted from IDA
$Keys = @(
    @('the final countdown', 'oh happy dayz'),
    @('UNACCEPTABLE!', 'omglob'),
    @("you're so bad", "you're so good"),
```

```
    @(([Char] 1), 'f'),
```



```
    @('Feel the sting of the Monarch!', ($Enc.GetString(@(9,0,0,1)))),
    @('! 50 1337', '1337'),
    @('MATH IS HARD', 'LETS GO SHOPPING'),
    @('SHOPPING IS HARD', 'LETS GO MATH'),
    @($Enc.GetString(@(7,0x77)), ($Enc.GetString(@(1,2,3,5,0,0x78,0x30,0x38,0x0D)))),
    @('backdoge.exe', 'd69650fa6d4825ec2ddeecdc6a92228d'),
    @('192.203.230.10', '127.0.0.1'),
    @('jackRAT', ([Char] 0))
)

# 0..16383 = the amount of possible encoder purmutations - 2^14

# Counting from 0-16383 and converting the current number to a
# binary string (e.g. 01100010101011) represents each permutation.

0..16383 | % {
    $PermuteString = [Convert]::ToString($_, 2).PadLeft(14, '0')
    $PermuteInstance = [Int[]]$PermuteString.ToCharArray()

    for ($i = 0; $i -lt $Keys.Length; $i++) {
        Xor ($Keys[$i][$PermuteInstance[$i] - 48]) $PEBytes
    }

    if ([Bitconverter]::ToInt32($PEBytes, 2) -eq 0x00030090) {
        Write-Host "Correct decode permutation found! $PermuteString"

        # Read in the entire file now
        [Byte[]] $DecodedPE = [IO.File]::ReadAllBytes($FilePath)
```

```
    for ($i = 0; $i -lt $Keys.Length; $i++) {
        Xor ($Keys[$i][$PermuteInstance[$i] - 48]) $DecodedPE
    }

    [IO.File]::WriteAllBytes("$PWD\gratz.exe_", $DecodedPE)

    break
  }

  # Restore the encoded byte array to its original form.
  $PEBytes = $OriginalPEBytes
}
```

After about 30 seconds of execution, the script successfully recovered the correct sequence of XOR keys. As it turns out, the malware expected the following environment to decode properly:

1. The EXE is not running in a debugger
2. The EXE is not running in a VM
3. The EXE is named backdoge.exe
4. The EXE executes on a Friday at 5PM
5. The EXE can connect to the Internet

In other words, the malware decodes if you allow an unknown, potentially malicious executable to run on your host OS.

The decoded EXE - gratz.exe is a .NET executable. I loaded the EXE into my go-to .NET disassembler/decompiler – ILSpy. I'm using the latest version in Figure 21. One of the features I like about the latest version is that it displays the metadata tokens of each class, method, etc. This helps out a lot when working with heavily obfuscated malware that uses unprintable Unicode characters.

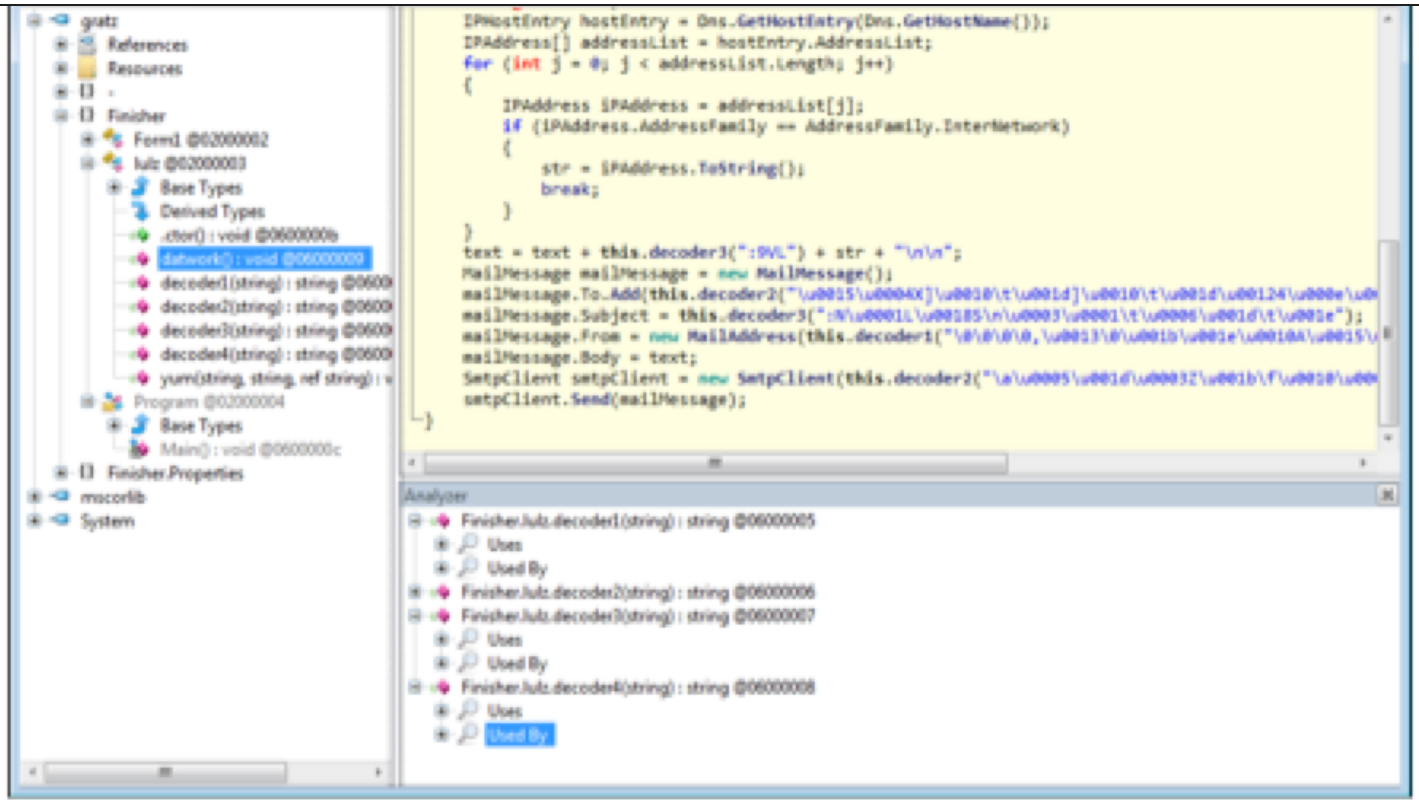

Figure 21: ILSpy dump of gratz.exe

In Figure 21, there are numerous encoded strings present. When working with .NET malware, decoding/decrypting strings and embedded byte arrays is typically my first course of action. I start by running a sample through de4dot, an incredibly powerful automated .NET deobfuscator. Unfortunately, in this case, it didn't work because it doesn't work well when it encounters instance methods as is the case in this example – i.e., the 'lulz' class must first be instantiated before the decoder methods can be called. Not a problem though. PowerShell can help us out with the task of decoding these strings.

When it comes to .NET executables, this is where I'm allowed to say that PowerShell is ideally suited for analysis since, after all, PowerShell is basically a glorified .NET interpreter. Using the .NET reflection API, we can call target methods in malware dynamically. The following PowerShell script performs the following actions:

1. Load gratz.exe. After it's loaded, we can begin to interact with its methods without executing any other malicious logic.
2. Instantiate the 'Finisher.lulz' class. Doing this allows us to call the four decoder methods.
3. Unescape the embedded Unicode strings.

4. Call the decoder method on each string. In other words, rather than implementing the decoding methods


```
$PEBytes = [IO.File]::ReadAllBytes("$PWD\gratz.exe_")
# Load the .NET executable
# Yes, .NET has an in-memory loader for .NET executables :D
$EvilAssembly = [Reflection.Assembly]::Load($PEBytes)

# All the decoder functions are instance methods of the 'lulz'
# class. Instantiate the 'lulz' class.
$Lulz = New-Object Finisher.lulz

function Get-DecodedStrings {
    # Encoded strings manually pulled out of gratz.exe_ in ILSpy
    $Decoder1Strings = @(
        '(\u0014\u0018Z.\u0010\r\u0019\u0003\u001bVpAXAWAXAWAXAWAXAWAXAWAXAWAXAWAXAp',
        '9\u0006\t\bVU',
        '\u001b\u0014\0\u0016\t\u0001B\u001e\r\u0001',
        '\0\0\0\0,\u0013\0\u001b\u001e\u0010A\u0015\u0002[\u000f\u0015\u0001'
    )
    $Decoder2Strings = @(
        '9\t\n\u001b\u001d\u0006\fIT',
        ';;I%\u0011\u001a\u001a\u001a\u001b\u0006SS',
'\u0015\u0004X]\u0010\t\u001d]\u0010\t\u001d\u00124\u000e\u0005\u0012\u0006\rD\u001c\u001aF\n\u001c\u0019',
        '\a\u0005\u001d\u0003Z\u001b\f\u0010\u0001\u001a\f\0\u0011\u001a\u001f\u0016\u0006F\a\u0016\0',
        '\u001b\u0005\u000eS\u001d\u001bI\a\u001c\u0001\u001aS\0\0\fS\u0006\r\b\u001fT\a\a\u0016K'
    )
    $Decoder3Strings = @(
        '&\u001a\t\u001e=\u001c\u0004\r\u0005\u0017II',
        '7\u001b\u0005\u001a\u001cII',
```

```
    ':9VL'.
```



```
    )

    $Decoder4Strings = @(

'\v\fP\u000e\u000fBA\u0006\rG\u0015I\u001a\u0001\u0016H\\\t\b\u0002\u0013/\b\t^\u001d\bJO\a]C\u001b\u0005'

    )


    # Call the decoder method on each encoded string.

    # No need to manually implement the logic of each decoder. :D

    $Decoder1Strings | % { $Lulz.decoder1([Regex]::Unescape($_)) }

    $Decoder2Strings | % { $Lulz.decoder2([Regex]::Unescape($_)) }

    $Decoder3Strings | % { $Lulz.decoder3([Regex]::Unescape($_)) }

    $Decoder4Strings | % { $Lulz.decoder4([Regex]::Unescape($_)) }

}


# Profit

Get-DecodedStrings
```

The output of this script is shown in Figure 22.

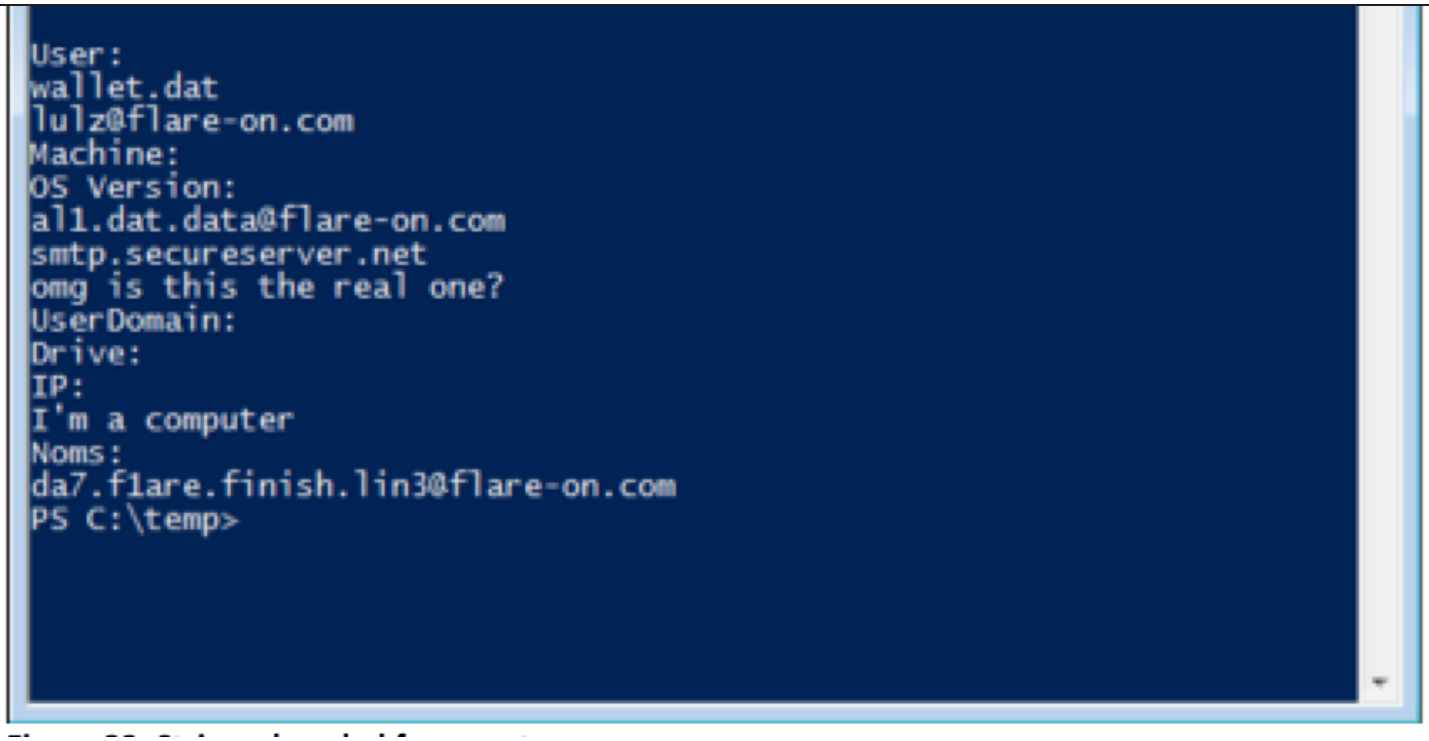

Figure 22: Strings decoded from gratz.exe

At this point, I don't care about what the malware actually does since I have three potential flag email addresses. I emailed all three and got a response from da7.f1are.finish.lin3@flare-on.com.

That was a lot of work. Isn't there an easier way? In fact, there is. I had already developed a PowerShell (non-public at the moment) tool that uses the de4dot library to automatically pull out encoded/encrypted strings, finds candidate decoder/decryption methods and tries to invoke those methods on the extracted strings. The Challenge 7 flag can then be pulled out in a one-liner as shown in Figure 23.

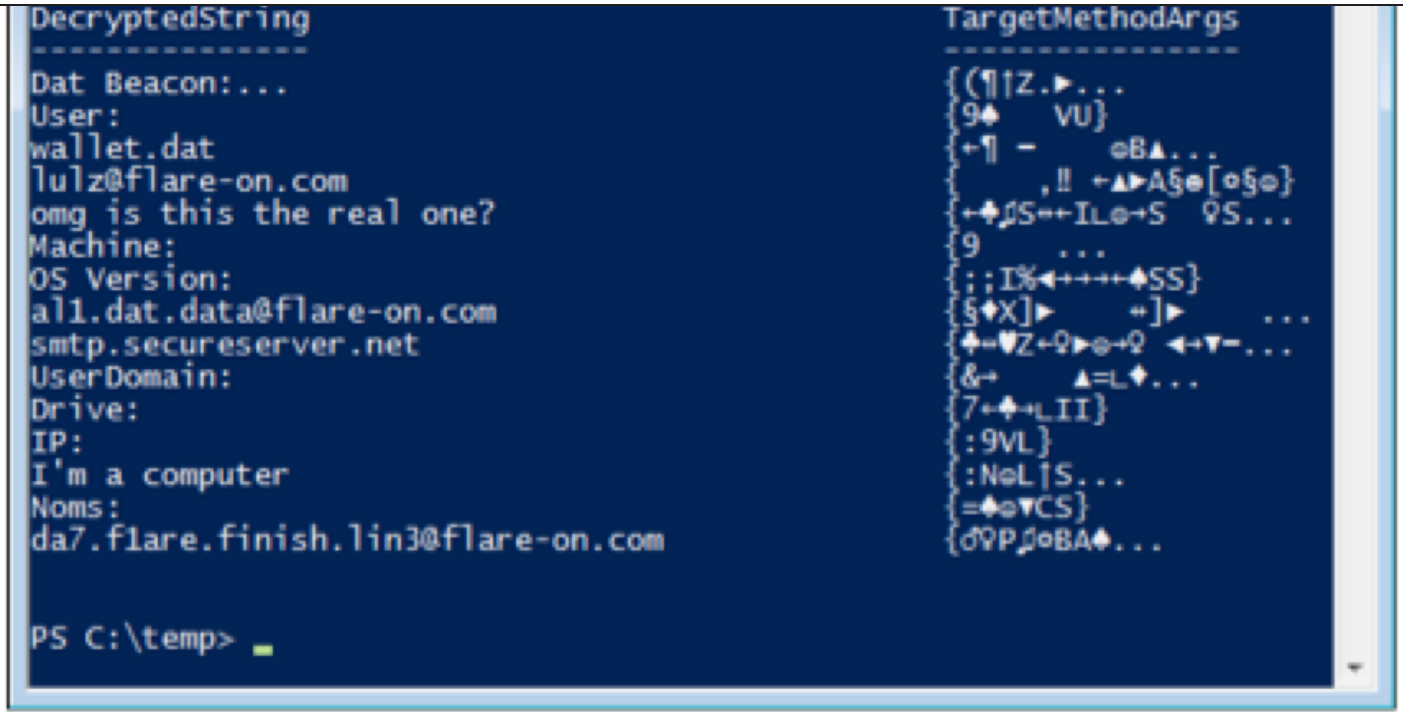

Figure 23: Automatic extraction of encoded strings

WRAP-UP

We hope you enjoyed this challenge and learned some new aspects of malware reverse engineering. We had fun developing the challenges and sharing them with the community. Stay tuned for the next FLARE On Challenge in early 2015.

This entry was posted on Fri Nov 21 10:00:00 EST 2014 and filed under Bad Encryption, Blog, Cybersecurity, Mike Sikorski, Richard Wartell and Technology.