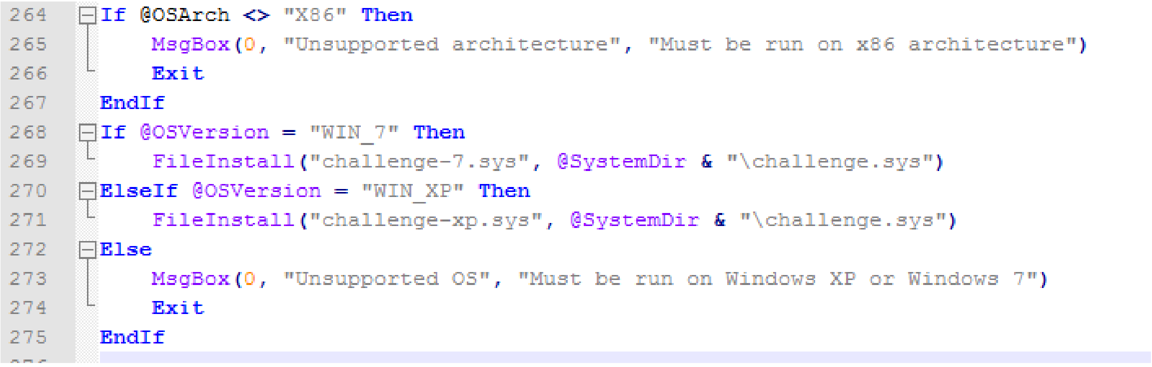
Challenge #10 Solution

by Tom Bennett

This challenge starts with a single executable file, `loader.exe`. Dropping it into PEview (or equivalent tool) tells us it is a 32-bit PE file. We start by taking a look at the strings to see if we can easily discover anything about the executable before digging deeper. As shown in Figure 1, a quick look through the strings tells us we are likely dealing with a compiled AutoIt script.

Figure 1 AutoIt strings

From AutoIt's website: "AutoIt v3 is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting." Now we want to search for any decompiler tools for compiled AutoIt scripts. A Web search shows us several options, in this case we will go with the Exe2Aut tool. As the disclaimer for the tool recommends, we drag and drop `loader.exe` onto Exe2Aut in a safe virtual machine environment since there is a chance it can be executed and perform malicious behaviors. The tool works, and we find ourselves with another executable `ioctl.exe`, two drivers `challenge-xp.sys` and `challenge-7.sys`, and the decompiled AutoIt script. The driver names suggest that they are each for a specific OS (Windows XP or Windows 7), so we take a closer a look at the script to see how these files are used and how they relate to each other. Scrolling down past all the service related functions, we find some evidence to support our deduction about the drivers as shown in Figure 2.



```
If @OSArch <> "X86" Then
    MsgBox(0, "Unsupported architecture", "Must be run on x86 architecture")
    Exit
EndIf
If @OSVersion = "WIN_7" Then
    FileInstall("challenge-7.sys", @SystemDir & "\challenge.sys")
ElseIf @OSVersion = "WIN_XP" Then
    FileInstall("challenge-xp.sys", @SystemDir & "\challenge.sys")
Else
    MsgBox(0, "Unsupported OS", "Must be run on Windows XP or Windows 7")
    Exit
EndIf
```

Figure 2 OS version check

Immediately following the OS checks, we find some obfuscated code that we can assume will do something with the files that it drops onto the system. Before getting messy with deobfuscating this code, we take a look at the executable and OS-specific driver of our choice. Thankfully, the executable only has one function and no obfuscation or anti-analysis tricks of any kind. It simply opens a handle to a device named `challenge` which is likely our driver, and sends an I/O request packet (IRP) with an I/O Control (IOCTL) code supplied as ASCII encoded hex via the command line. It waits for a response from the driver, but disregards it and exits. In summary, this binary is a simple tool to send a one way IRP to our challenge driver. This leads us to believe there is some IOCTL code that we need to discover to help us along with this challenge. With no other clues in this file, we move onto the driver. The first thing we notice when opening the driver in IDA Pro is how long it takes to perform its initial analysis. Once done, we poke around the functions and quickly discover a big mess as shown in Figure 3.

Figure 3 Big function

It turns out there are several functions in this driver that look much like this one. To make matters worse, the IRP handler function has cases for around 400 IOCTL codes! It is probably not a good idea for us to continue digging in the driver at this point, we need to find that IOCTL code. Perhaps one of those obfuscated AutoIt script lines will make use of `ioctl.exe` and give us the right code.

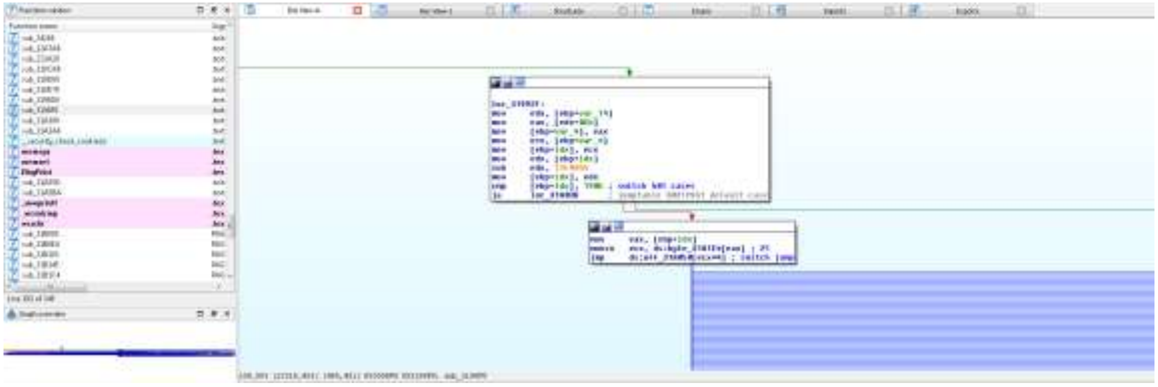
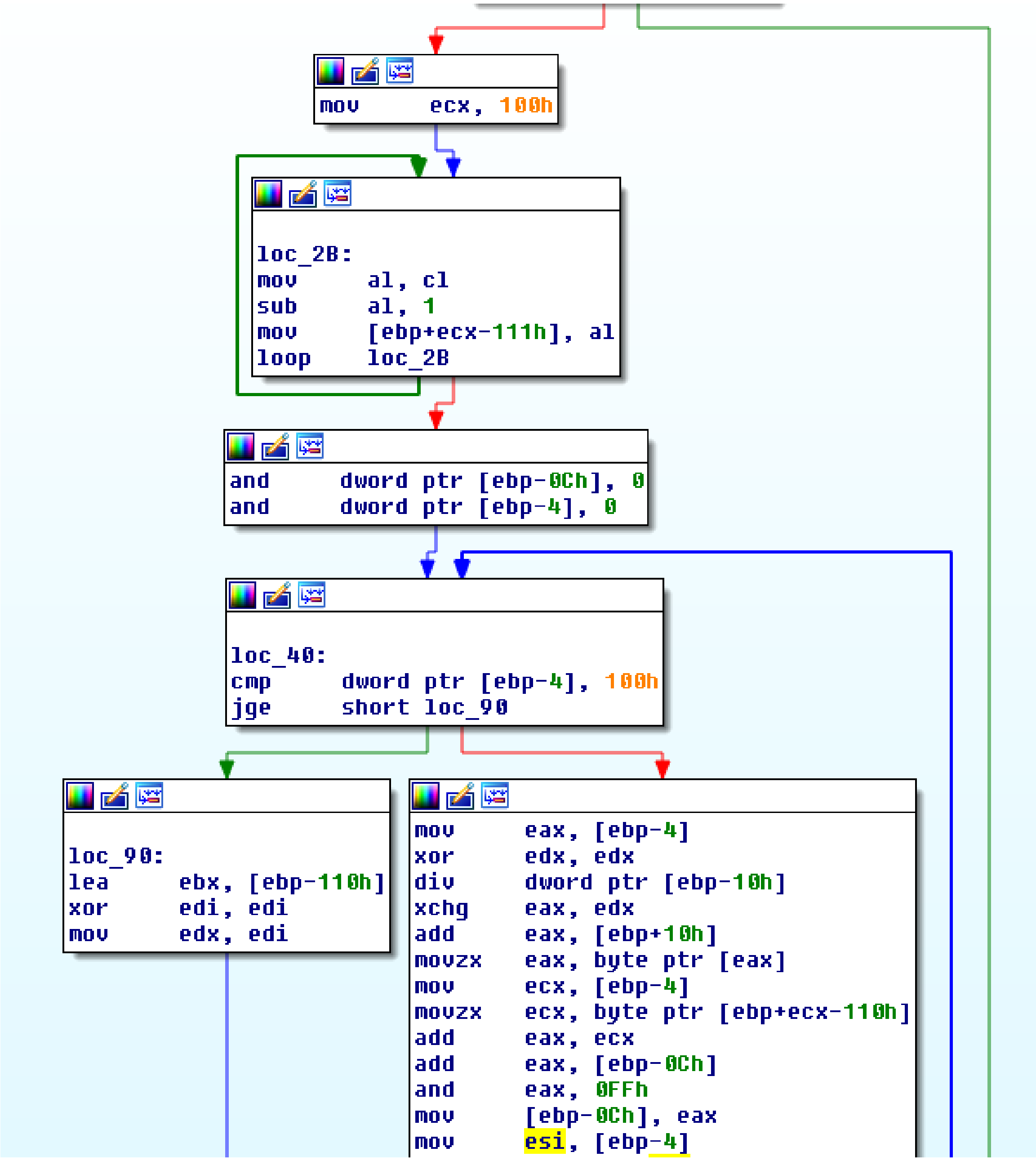


Figure 4 IRP handler function

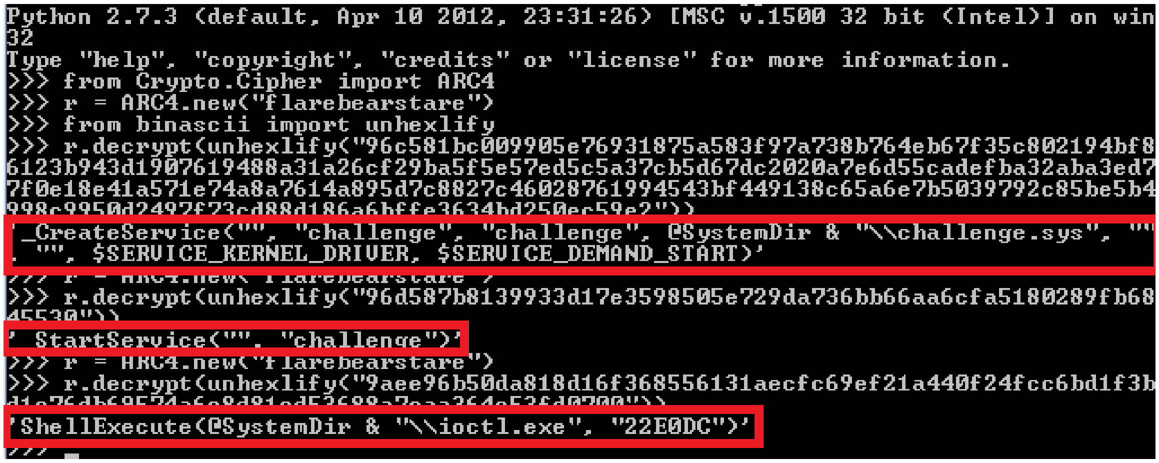
The obfuscation in the AutoIt script involves decrypting each line of code and executing it. It uses the `CallWindowProc` API to achieve arbitrary execution, in this case executing shellcode it places in memory using AutoIt's `DllStructCreate` function. This shellcode contains some kind of decryption routine used to decrypt the script lines with the key `flarebearstare`. To analyze the decryption routine, we copy the hex value out of the AutoIt script into a small Python script that uses the `unhexlify` function from the `binascii` module to convert it into binary and write it to file. Once we open this file in IDA Pro to view the disassembly, we can see there are two successive loops both with 256 iterations as displayed in Figure 5.



```
mov     ecx, 100h

loc_2B:
mov     al, cl
sub     al, 1
mov     [ebp+ecx-111h], al
loop    loc_2B

and     dword ptr [ebp-0Ch], 0
and     dword ptr [ebp-4], 0

loc_40:
cmp     dword ptr [ebp-4], 100h
jge     short loc_90

loc_90:
lea     ebx, [ebp-110h]
xor     edi, edi
mov     edx, edi

mov     eax, [ebp-4]
xor     edx, edx
div     dword ptr [ebp-10h]
xchg    eax, edx
add     eax, [ebp+10h]
movzx   eax, byte ptr [eax]
mov     ecx, [ebp-4]
movzx   ecx, byte ptr [ebp+ecx-110h]
add     eax, ecx
add     eax, [ebp-0Ch]
and     eax, 0FFh
mov     [ebp-0Ch], eax
mov     esi, [ebp-4]
```

Figure 5 Decryption function

Those with cryptography experience, or a fair amount of malware analysis experience, may recognize this as possibly being the key scheduling algorithm (KSA) for an implementation of the RC4 stream cipher. Further analysis confirms this to be the case, leaving us with the task of decrypting the three lines of AutoIt code. Once decrypted, we can see that the script installs and starts the `challenge` service, then executes `ioctl.exe` with the argument `22E0DC`. There is our IOCTL code!



```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> from Crypto.Cipher import ARC4
>>> r = ARC4.new("flarebearstare")
>>> from binascii import unhexlify
>>> r.decrypt(unhexlify("96c581bc009905e76931875a583f97a738b764eb67f35c802194bf8
6123b943d1907619488a31a26cf29ba5f5e57ed5c5a37cb5d67dc2020a7e6d55cadefba32aba3ed7
7f0e18e41a571e74a8a7614a895d7c8827c46028761994543bf449138c65a6e7b5039792c85be5b4
998c9950d2497f73cd88d186a6bffe3634bd250ec59e2"))
'_CreateService("", "challenge", "challenge", @SystemDir & "\\challenge.sys", ""
, "", $SERVICE_KERNEL_DRIVER, $SERVICE_DEMAND_START)'
>>> r = ARC4.new("flarebearstare")
>>> r.decrypt(unhexlify("96d587b8139933d17e3598505e729da736bb66aa6cfa5180289fb68
45530"))
' StartService("", "challenge")'
>>> r = ARC4.new("flarebearstare")
>>> r.decrypt(unhexlify("9aee96b50da818d16f368556131aecfc69ef21a440f24fcc6bd1f3b
d1e76db69574a6e8d81ed53688a7eaa364e53fd0700"))
'ShellExecute(@SystemDir & "\\ioctl.exe", "22E0DC")'
>>>
```

Figure 6 Decrypted AutoIt script lines

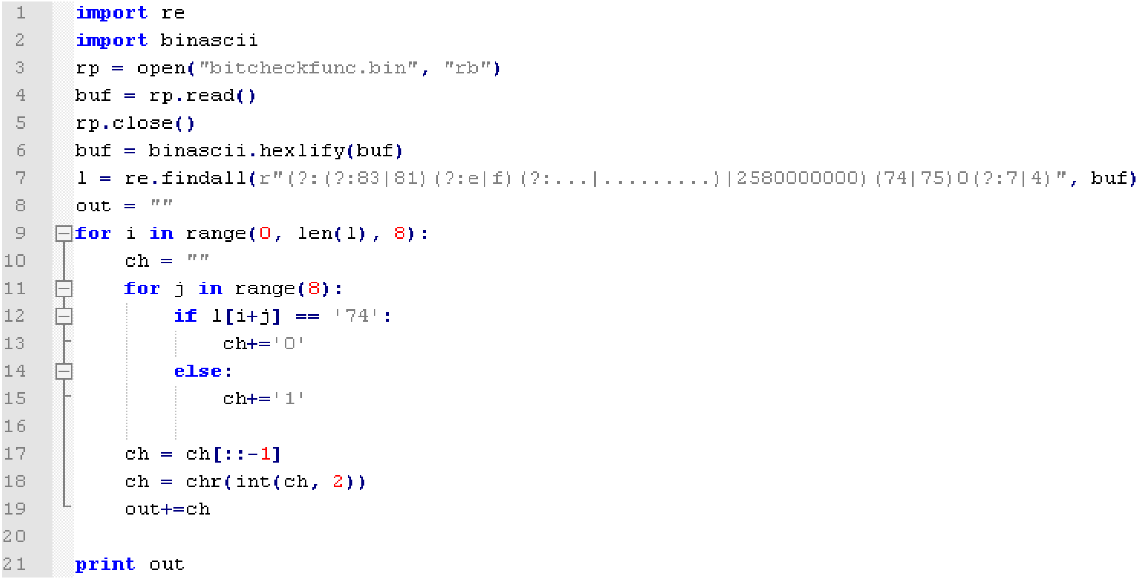
After calculating the proper jump table destination in the IRP handler function, we identify the function we need to look at next, which is partially illustrated in Figure 7. We see that this function is performing a bit test on each bit of the first byte of `var_1C`, which was initialized to zero. It then does the same thing for the next byte, and the byte after that, up to 22 bytes.

```
.text:00318D82                  xor     eax, eax
.text:00318D84                  mov     [ebp+var_1C], eax
.text:00318D87                  mov     [ebp+var_18], eax
.text:00318D8A                  mov     [ebp+var_14], eax
.text:00318D8D                  mov     [ebp+var_10], eax
.text:00318D90                  mov     [ebp+var_C], eax
.text:00318D93                  mov     [ebp+var_8], ax
.text:00318D97                  movzx   ecx, byte ptr [ebp+var_1C]
.text:00318D9B                  and     ecx, 1
```

```
.text:00318D9E                 jz      short loc_318DA7
.text:00318DA0                 xor     al, al
.text:00318DA2                 jmp     loc_3198B6
.text:00318DA7 ; ---------------------------------------------------
.text:00318DA7
.text:00318DA7 loc_318DA7:
.text:00318DA7                 movzx   edx, byte ptr [ebp+var_1C]
.text:00318DAB                 and     edx, 2
.text:00318DAE                 jz      short loc_318DB7
.text:00318DB0                 xor     al, al
.text:00318DB2                 jmp     loc_3198B6
.text:00318DB7 ; ---------------------------------------------------
.text:00318DB7
.text:00318DB7 loc_318DB7:
.text:00318DB7                 movzx   eax, byte ptr [ebp+var_1C]
.text:00318DBB                 and     eax, 4
.text:00318DBE                 jnz     short loc_318DC7
.text:00318DC0                 xor     al, al
.text:00318DC2                 jmp     loc_3198B6
.text:00318DC7 ; ---------------------------------------------------
.text:00318DC7
.text:00318DC7 loc_318DC7:
.text:00318DC7                 movzx   ecx, byte ptr [ebp+var_1C]
.text:00318DCB                 and     ecx, 8
.text:00318DCE                 jz      short loc_318DD7
.text:00318DD0                 xor     al, al
.text:00318DD2                 jmp     loc_3198B6
```

Figure 7 Bit test function snippet

We can assume that these bit tests are a clue to us of the bits that "should" be there, so we just need a way to translate these bit tests to actual bits. Considering that the determination of whether a bit should be "on" or "off" comes down to whether a `jz` or `jnz` instruction is used for a branch, we can write a small script to parse the code and do this for us. The resulting buffer turns out to be the string `try this ioctl: 22E068`.



```python
import re
import binascii
rp = open("bitcheckfunc.bin", "rb")
buf = rp.read()
rp.close()
buf = binascii.hexlify(buf)
l = re.findall(r"(?:(?:83|81)(?:e|f)(?:...|.........)|2580000000)(74|75)0(?:7|4)", buf)
out = ""
for i in range(0, len(l), 8):
    ch = ""
    for j in range(8):
        if l[i+j] == '74':
            ch+='0'
        else:
            ch+='1'

    ch = ch[::-1]
    ch = chr(int(ch, 2))
    out+=ch

print out
```

Figure 8 Bit test function deobfuscation script

This IOCTL code leads us to one of those gigantic, messy functions. However, there must be something special about this one. Browsing through the code, we see a lot of useless math being done on variables that are just thrown away. Ultimately, the only thing being done in this function that really matters is moving byte values into a global character array. The problem is that there are many branches in this function moving different values into different positions in the array. Which branches are the correct ones to take? How do we influence that? If we look at the very end of this large function, we can see that a pointer to some element of that global character array is pushed onto the stack as an argument for a function being called. A cursory look at this function reveals cryptography that operates on the buffer from this point in the array and that the second argument specifies the length of the buffer. Also, looking at the cross references to characters in the global array shows us that only from this point forward is every byte referenced. Before this point, there are many elements in the array that are not directly referenced.

```
.data:0031AF7D                      db     0
.data:0031AF7E byte_31AF7E          db 0                           ; DATA XREF: sub_1B440+37B↑w
.data:0031AF7F byte_31AF7F          db 0                           ; DATA XREF: sub_11ED0+33C↑w
.data:0031AF7F                                                     ; sub_1BB10+361↑w
.data:0031AF80 byte_31AF80          db 0                           ; DATA XREF: sub_12BF0+350↑w
.data:0031AF80                                                     ; sub_15FE0+2B7↑w
.data:0031AF81                      db     0
.data:0031AF82                      db     0
.data:0031AF83 byte_31AF83          db 0                           ; DATA XREF: sub_20740+376↑w
.data:0031AF84 byte_31AF84          db 0                           ; DATA XREF: sub_138B0+28F↑w
.data:0031AF85 byte_31AF85          db 0                           ; DATA XREF: sub_1B150+2D4↑w
.data:0031AF86 byte_31AF86          db 0                           ; DATA XREF: sub_241F0+2F9↑w
.data:0031AF87                      db     0
.data:0031AF88                      db     0
.data:0031AF89 byte_31AF89          db 0                           ; DATA XREF: sub_1BE90+2BB↑w
.data:0031AF89                                                     ; sub_1E480+370↑w
.data:0031AF8A byte_31AF8A          db 0                           ; DATA XREF: sub_21E80+31F↑w
.data:0031AF8B                      db     0
.data:0031AF8C                      db     0
.data:0031AF8D                      db     0
.data:0031AF8E                      db     0
.data:0031AF8F byte_31AF8F          db 0                           ; DATA XREF: sub_12220+373↑w
.data:0031AF90 byte_31AF90          db 0                           ; DATA XREF: sub_247D0+2DC↑w
.data:0031AF90                                                     ; sub_34260+F652D↑o
.data:0031AF91 byte_31AF91          db 0                           ; DATA XREF: sub_24AC0+2C7↑w
.data:0031AF92 byte_31AF92          db 0                           ; DATA XREF: sub_24DA0+30F↑w
.data:0031AF93 byte_31AF93          db 0                           ; DATA XREF: sub_250D0+2CC↑w
.data:0031AF94 byte_31AF94          db 0                           ; DATA XREF: sub_253B0+2BC↑w
.data:0031AF95 byte_31AF95          db 0                           ; DATA XREF: sub_25680+353↑w
.data:0031AF96 byte_31AF96          db 0                           ; DATA XREF: sub_259F0+345↑w
.data:0031AF97 byte_31AF97          db 0                           ; DATA XREF: sub_25D50+2C3↑w
.data:0031AF98 byte_31AF98          db 0                           ; DATA XREF: sub_26030+368↑w
.data:0031AF99 byte_31AF99          db 0                           ; DATA XREF: sub_263B0+332↑w
.data:0031AF9A byte_31AF9A          db 0                           ; DATA XREF: sub_26700+2E2↑w
.data:0031AF9B byte_31AF9B          db 0                           ; DATA XREF: sub_26A00+2A6↑w
.data:0031AF9C byte_31AF9C          db 0                           ; DATA XREF: sub_26CC0+2AA↑w
.data:0031AF9D byte_31AF9D          db 0                           ; DATA XREF: sub_26F80+2E2↑w
.data:0031AF9E byte_31AF9E          db 0                           ; DATA XREF: sub_27280+375↑w
```

Figure 9 Global array cross references

It seems then, that there must be a path through this function that will fill this array with the correct characters that will decrypt to something meaningful (hopefully the key). Looking at each conditional expression, an interesting pattern becomes clear: these conditionals are not really conditional at all! Shortly before each test operation, the variable being tested is set to zero. After checking a few branches, it becomes apparent that the branches filling the array that we care about are never taken with the code in its current state. There are several ways we could go about retrieving the buffer we are looking for, we will take the dynamic approach and apply patches to fix the branches for us. Since Windows performs an integrity check on a driver file before loading it, we will patch in memory to avoid having to deal with another obstacle. Using windbg, this can be accomplished by dumping the function's memory using the `.writemem` command, patching the function on disk, then reading it back to memory in the same place with the `.readmem` command. Since there are many places in the code that need to be patched and the patch is always the same, it is easier to do a simple find and replace operation. This can be done with the following Python code snippet.

```
string.replace(buf, "\xc6\x45\x9e\x00", "\xc6\x45\x9e\x01")
```

Figure 10 Patching code snippet

With the patched function in memory, we set a breakpoint on the call to the crypto function and use `ioctl.exe` to execute it. Stepping over the function and checking the buffer reveals the key `unconditional_conditions@flare-on.com`.