FireEye

FLARE

SECURITY REIMAGINED

Challenge #4 Solution

by Josh Homan

When executing `youPecks.exe` without any arguments you get the output “`2+2=4`”. Inspecting the binary closer, you’ll notice this challenge is packed with UPX. Section names starting with UPX followed by a number (`UPX0` and `UPX1`) and the string “`UPX!`” at the end of the PE header are both good indications the binary is packaged with UPX. For the next step you’ll want to unpack the challenge so it can be easily loaded into static analysis tools.

A common method of unpacking UPX packed binaries is the command line option `-d`. This option extracts the original packed binary. Figure 1 contains an example command line that creates an unpacked binary `youPecks.upx.exe` from the original `youPecks.exe`.

```
c:\>upx -d -o youPecks.upx.exe youPecks.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91w       Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     25088 <-     12800   51.02%    win32/pe     youPecks.upx.exe
```

Figure 1: UPX –d example

After executing `youPecks.upx.exe` you’ll notice the output is now “`2+2=5`”.

To understand what is happening, we first have to cover how “`upx -d`” operates. When unpacking a compressed file using the command line option, UPX references a structure that is located at the end of the PE header. This header includes checksums, packed and unpacked sizes, and compression details. Figure 2 shows the UPX structure from youPecks.exe (packhead.cpp).

```
000003D0  00 00 00 00 00 00 00 00 00 00 00 33 2E 39 31 00  ...........3.91.
000003E0  55 50 58 21 0D 09 02 08 D1 1E A6 44 50 6A BD 1A  UPX!......¦DPj..
000003F0  81 9A 00 00 3A 24 00 00 00 62 00 00 26 01 00 A5  ....:$...b..&…
```

Figure 2: UPX header for youPecks.exe

This process is done entirely within the UPX utility and no code is executed from the packed sample. In contrast, when a UPX packed program is executed a small unpacking stub embedded within the program is run first. This stub unpacks the executable to memory, loads the required imports and executes the program. This means the stub can be can patch or produce an entirely different executable than one created by “`upx -d`”.

Based on the different output from youPecks.exe (“`2+2=4`”) and youPecks.upx.exe (“`2+2=5`”) it’s likely code is being modified within the unpacking stub. Opening `youPecks.exe` in IDA Pro, notice an instruction at offset `0x0040B60B` is setting a memory address to ‘`4`’ (Figure 3). There is also a loop at offset `0x0040B601` that is XORing memory with `0x20`.

```
UPX1:0040B5F8  push    33h
UPX1:0040B5FA  pop     ecx
UPX1:0040B5FB  add     edi, 51B8h
UPX1:0040B601 loc_40B601:
UPX1:0040B601  xor     byte ptr [ecx+edi], 20h
UPX1:0040B605  loop    loc_40B601
UPX1:0040B607  xor     byte ptr [ecx+edi], 20h
UPX1:0040B60B  mov     byte ptr [esi+424Ch], '4'
UPX1:0040B612  pop     ecx
UPX1:0040B613  popa
UPX1:0040B614 lea     eax, [esp-80h]
UPX1:0040B618  loc_40B618:
UPX1:0040B618  push    0
UPX1:0040B61A  cmp     esp, eax
UPX1:0040B61C  jnz     short loc_40B618
UPX1:0040B61E  sub     esp, 0FFFFFF80h
UPX1:0040B621  jmp     near ptr word_403A8A
```

Figure 3: UPX Stub

By setting a breakpoint at `0x0040B601`, we can see the XOR loop is modifying the string “`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`”. The XOR loop swaps the case for the first `52` characters producing the string

"`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/`".

Inspecting the code at address `0x0040B60B`, we can see the value ‘`5`’ being overwritten with ‘`4`’.

At this point we know what is being modified after the file is unpacked and have a couple options. One option is to create an entirely new PE using your favorite process dumper before executing the original entry point (OEP). The jump to OEP is located at address `0x0040B601`. Another option, since these are fairly trivial changes, is to patch the UPX extracted binary (`youPecks.upx.exe`) using a hex editor. The modified string is at offset `0x3BB8` and the modified ‘`5`’ is at offset `0x3C4C`.

Now that we have a correctly unpacked executable, we can take a closer look to see what the file is doing. One of the first items to notice is the check to determine if the file was unpacked correctly. The code at address `0x00401442` converts the string at address `0x0040524C` from ASCII to integer and verifies it isn’t `5` (Figure 4). Since we have a correctly unpacked binary, this is no longer an issue.

```
.text:00401442  push    offset Str      ; "4"
```

```
.text:00401447  call    ds:atoi
.text:0040144D  add     esp, 4
.text:00401450  mov     esi, eax
...
.text:0040147C  cmp     esi, 5
.text:0040147F  jnz     short loc_4
```

Figure 4: Unpack check

Looking further we can see the challenge verifies that a single argument is provided from the command line (Figure 5). The challenge then converts the first argument to an integer at address `0x00401564` and the result is passed as a parameter to a function call at address `0x00401584`.

```
.text:004014EE  cmp     [ebp+argc], 2
.text:004014F2  jz      short loc_401560
...
.text:00401560  mov     eax, [edi+4]
.text:00401563  push    eax                 ; Str
.text:00401564  call    ds:atoi
.text:0040156A  add     esp, 4
.text:0040156D  lea     ecx, [esp+0F4h+var_34]
.text:00401574  push    ecx                 ; BYTE *
.text:00401575  lea     edx, [esp+0F8h+pbData]
.text:0040157C  push    edx                 ; pbData
.text:0040157D  mov     [esp+0FCh+pbData], al
.text:00401584  call    sub_4012E0
```

Figure 5: Argument check

Looking closer at `sub_4012E0`, we can see the function is calculating the MD5 sum of the single byte buffer (Figure 6).

```
...
.text:0040130F  push    ecx                 ; phHash
.text:00401310  push    0                   ; dwFlags
.text:00401312  push    0                   ; hKey
.text:00401314  push    CALG_MD5            ; Algid
.text:00401319  push    edx                 ; hProv
.text:0040131A  call    ds:CryptCreateHash
.text:00401320  mov     eax, [ebp+pbData]
.text:00401323  mov     ecx, [ebp+phHash]
.text:00401326  push    0                   ; dwFlags
.text:00401328  push    1                   ; dwDataLen
.text:0040132A  push    eax                 ; pbData
.text:0040132B  push    ecx                 ; hHash
.text:0040132C  call    ds:CryptHashData
```

Figure 6: MD5 Function

The next important observation is the challenge obtains the current time at address 0x0040193 and stores the current hour at address 0x004015A0 (Figure 7).

```
.text:00401589   lea      eax, [esp+0FCh+Time]
.text:0040158D   push     eax                ; Time
.text:0040158E   lea      ecx, [esp+100h+Tm]
.text:00401592   push     ecx                ; Tm
.text:00401593   call     ds:_localtime64_s
.text:00401599   add      esp, 10h
.text:0040159C   cmp      eax, ebx
.text:0040159E   jnz      short loc_4015A6
.text:004015A0   mov      edi, [esp+0F4h+Tm.tm_hour]
.text:004015A4   jmp      short loc_4015A9
```

Figure 7: Get current hour

The challenge then creates a vector containing 24 strings. The current hour is then used as an index into this vector and the result is passed to a function at address 0x00401000 (Figure 8).

```
.text:00401B22   lea      eax, ds:0[edi*8]
.text:00401B29   sub      eax, edi
.text:00401B2B   add      eax, eax
.text:00401B2D   add      eax, eax
.text:00401B2F   mov      [esp+0F4h+var_9C], eax
.text:00401B33   add      eax, [esp+0F4h+var_E4]
.text:00401B37   sub      esp, 1Ch
.text:00401B3A   mov      ecx, esp ; int
.text:00401B3C   mov      [esp+110h+var_A0], esp
.text:00401B40   push     eax ; int
.text:00401B41   mov      dword ptr [ecx+14h], 0Fh
.text:00401B48   mov      dword ptr [ecx+10h], 0
.text:00401B4F   or       eax, 0FFFFFFFFh
.text:00401B52   xor      ebx, ebx
.text:00401B54   mov      byte ptr [ecx], 0
.text:00401B57   call     sub_4032D0
.text:00401B5C   lea      eax, [esp+80h]
.text:00401B63   push     eax
.text:00401B64   call     sub_401000
```

Figure 8: Vector access using current hour

Looking closer at sub_401000, we can see the function is responsible for Base64 decoding (The function contains an error string "Non-Valid Character in Base 64!" and references the constant '='). Note the UPX stub modifies the character set for this function before executing the unpacked program. If the UPX unpacked binary `youPecks.upx.exe` is executed, the Base64 decoding would produce different results due to a different character set being used.

The challenge then compares Base64 decoded string against the MD5 hash of the first command line argument at address `0x00401BB0`. If the values do not match, the program exits. Now we know the challenge expects the first command line argument to be the current hour of the system.

The challenge then creates another vector containing 24 strings at address `0x00401BC7`. The current hour is used as an index into the new vector and the result is Base64 decoded with the customer character set (address `0x004023D3`). The challenge uses the MD5 hash of the current hour as an XOR key to decode the previously Base64 decoded value. Finally the challenge then prints the result `Uhr1thm3tic@flare-on.com`.

Figure 9 shows sample Python code that decodes the result for hour zero.

```
import base64

hour = "K7IfRF4nOiNn9Jsqt9wFCq==".swapcase()
crypted = "XTd3NiPLZBQ5N1FqkBN+a/Av6SpqBS/K".swapcase()

hour = base64.b64decode(hour)
crypted = base64.b64decode(crypted)

result = ""
for x in range(len(crypted)):
    result += chr(ord(hour[x % len(hour)]) ^ ord(crypted[x]))

print(result)
```

Figure 9: Sample Python code