



FLARE-ON CHALLENGE 9 SOLUTION
BY MICHAEL BAILEY (@MYKILL)

Challenge 4: darn_mice

Challenge Prompt

"If it crashes its user error."

-Flare Team

7-zip password: flare

Solution

The file `darn_mice.exe` is a console executable that terminates silently unless one command-line argument is provided. If an argument is provided, the program terminates abnormally, as shown in [Figure 1](#).

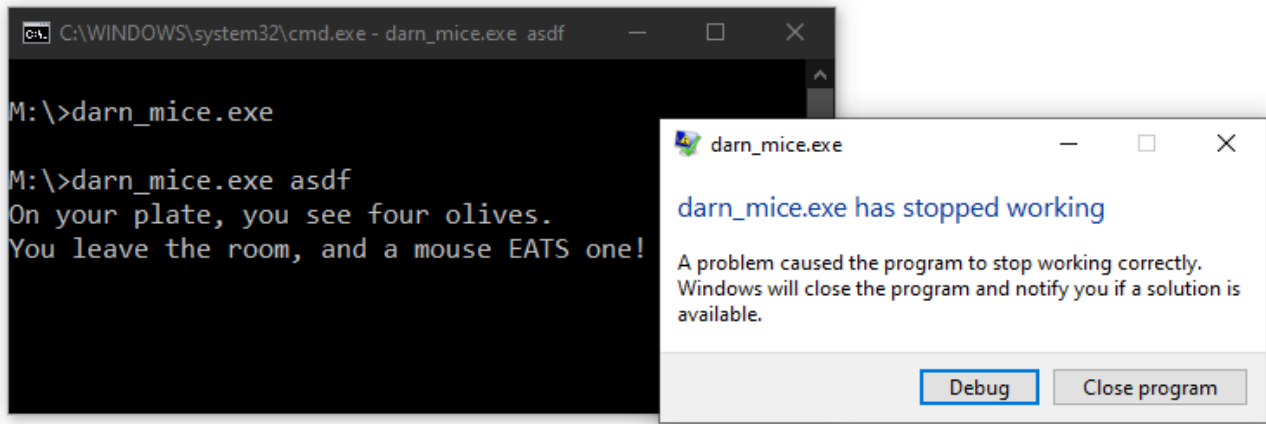


Figure 1: `darn_mice.exe` command-line behavior

Basic Static Analysis

There are several interesting strings, as shown in [Listing 1](#).

```
On your plate, you see four olives.  
No, nevermind.  
You leave the room, and a mouse EATS one!  
Nibble...  
When you return, you only: %s  
salty  
SHA512  
BCryptOpenAlgorithmProvider failed, %08x  
BCryptDeriveKeyPBKDF2 failed, %08x
```

Listing 1: Interesting strings

The imported functions are mostly typical of C runtime code, but three functions stand out:

DLL	Function
bcrypt.dll	BCryptOpenAlgorithmProvider
bcrypt.dll	BCryptDeriveKeyPBKDF2
KERNEL32.dll	VirtualAlloc

Table 1: Imports of interest

Advanced Static Analysis

Function main checks if the argument count in argc is 2, and if so, it passes the lone command-line argument to sub_401000. At the beginning of sub_401000, the program moves 35 values into a byte array starting at var_28, followed by a null. The beginning of this sequence is shown in [Figure 2](#).

```

.text:00401000 var_7= byte ptr -7
.text:00401000 var_6= byte ptr -6
.text:00401000 var_5= byte ptr -5
.text:00401000 var_4= dword ptr -4
.text:00401000 Str= dword ptr 8
.text:00401000
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 sub     esp, 34h
.text:00401006 mov     eax, ___security_cookie
.text:0040100B xor     eax, ebp
.text:0040100D mov     [ebp+var_4], eax
.text:00401010 mov     [ebp+var_2C], 0
.text:00401017 mov     [ebp+var_30], 0
.text:0040101E mov     [ebp+var_34], 0
.text:00401025 mov     [ebp+var_28], 50h ; 'P'
.text:00401029 mov     [ebp+var_27], 5Eh ; '^'
.text:0040102D mov     [ebp+var_26], 5Eh ; '^'
.text:00401031 mov     [ebp+var_25], 0A3h
.text:00401035 mov     [ebp+var_24], 4Fh ; 'O'
.text:00401039 mov     [ebp+var_23], 5Bh ; '['
.text:0040103D mov     [ebp+var_22], 51h ; 'Q'
.text:00401041 mov     [ebp+var_21], 5Eh ; '^'
.text:00401045 mov     [ebp+var_20], 5Eh ; '^'
.text:00401049 mov     [ebp+var_1F], 97h
.text:0040104D mov     [ebp+var_1E], 0A3h
.text:00401051 mov     [ebp+var_1D], 80h
.text:00401055 mov     [ebp+var_1C], 90h
.text:00401059 mov     [ebp+var_1B], 0A3h
.text:0040105D mov     [ebp+var_1A], 80h
.text:00401061 mov     [ebp+var_19], 90h
.text:00401065 mov     [ebp+var_18], 0A3h
.text:00401069 mov     [ebp+var_17], 80h
.text:0040106D mov     [ebp+var_16], 90h
.text:00401071 mov     [ebp+var_15], 0A3h
.text:00401075 mov     [ebp+var_14], 80h
    
```

Figure 2: Array of byte values in sub_401000

The push instruction at 0x4010B5 loads the string "On your plate, you see four olives.\n", which was seen in the output in [Figure 1](#). Based on this, sub_401240 can be renamed as printf, and ignored.

If the string length of the command-line argument is zero or is greater than 35, then the program prints:

```
No, nevermind.
```

Otherwise, the program prints:

```
You leave the room, and a mouse EATS one!
```

The program then enters a loop, iterating up to 35 times, iterating through the bytes of the command-line argument and the byte array in tandem. In the loop body, the program takes 3 steps:

1. Allocates a read/write/execute buffer (0x40113b);
2. Writes a single byte to the first element of the buffer, computed from the sum of the current byte from the command-line argument and the corresponding element in the byte array (0x40115a); and
3. Calls the buffer (0x401164).

This is shown in [Figure 3](#).

```

.text:0040112D push    PAGE_EXECUTE_READWRITE ; flProtect
.text:0040112F push    3000h                    ; flAllocationType
.text:00401134 push    1000h                    ; dwSize
.text:00401139 push    0                        ; lpAddress
.text:0040113B call    ds:VirtualAlloc
.text:00401141 mov     [ebp+rw_buf], eax
.text:00401144 mov     eax, [ebp+loop_iterator]
.text:00401147 movzx  ecx, [ebp+eax+byte_array] ; ecx = byte_array[i]
.text:0040114C mov     edx, [ebp+cmdline_arg]
.text:0040114F add     edx, [ebp+loop_iterator] ; edx = &cmdline_arg[i]
.text:00401152 movsx  eax, byte ptr [edx] ; eax = cmdline_arg[i]
.text:00401155 add     ecx, eax ; ecx = byte_array[i] + cmdline_arg[i]
.text:00401157 mov     edx, [ebp+rw_buf]
.text:0040115A mov     [edx], cl ; rw_buf[0] = byte_array[i] + cmdline_arg[i]
.text:0040115C call   [ebp+rw_buf]
.text:0040115F push   offset aNibble ; "Nibble...\n"
.text:00401164 call   printf
.text:00401169 add     esp, 4
.text:0040116C jmp     short increment i

```

Figure 3: Loop body

This accounts for the crash that occurs when the program is executed with arbitrary command-line arguments (for details, see

Appendix A: Crash Analysis).

If control flow does resume after the call into the buffer, then the program prints a message to the console:

Nibble...

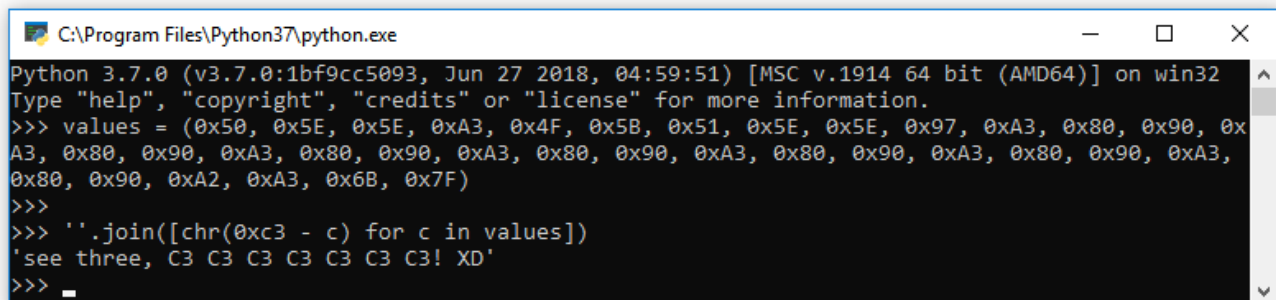
After the loop exits, the malware uses the bcrypt library to perform PBKDF2 key derivation and uses RC4 to then decrypt unidentified bytes that are subsequently printed to the console. Presumably, this is the flag.

The problem appears to be a matter of delivering control flow back to the end of the loop body to continue iterating through the loop.

Because the command-line argument only affords control of the first byte of each buffer, it is easiest to consider common single-byte instructions that would return control to the location after the call. The instruction that fits best is the single-byte near return (RETN) instruction with opcode 0xC3, since its purpose is to pop the saved return address off the stack and return control flow to the instruction after the call.

To test whether the RETN instruction can be used across the board requires computing all the corresponding character values that would sum with each byte array value to equal 0xC3; if these candidate input values are all printable character values, then they can be provided on the command line and might constitute a password that can be used to access the flag.

To collect the byte values in the sequence starting at 0x401025 in IDA Pro, it is reasonably convenient to display opcode bytes (Options → General... , Number of opcode bytes). The disassembly lines can be selected and filtered through awk (code: {print \$5}) or otherwise parsed. [Figure 4](#) shows an example of how to compute and print the argument characters that satisfy the criterion of producing RETN instructions for each iteration of the loop body.



```

C:\Program Files\Python37\python.exe
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> values = (0x50, 0x5E, 0x5E, 0xA3, 0x4F, 0x5B, 0x51, 0x5E, 0x5E, 0x97, 0xA3, 0x80, 0x90, 0x
A3, 0x80, 0x90, 0xA3, 0x80, 0x90, 0xA3, 0x80, 0x90, 0xA3, 0x80, 0x90, 0xA3, 0x80, 0x90, 0xA3,
0x80, 0x90, 0xA2, 0xA3, 0x6B, 0x7F)
>>>
>>> ''.join([chr(0xc3 - c) for c in values])
'see three, C3 C3 C3 C3 C3 C3 C3! XD'
>>> _

```

Figure 4: Computing character values from the difference of 0xC3 and each byte array value

This produces the result:

see three, C3 C3 C3 C3 C3 C3 C3! XD

To test this on the command line with darn_mice.exe, it is necessary to surround the argument in quotes to preserve the text as a single argument (due to the space characters). If this is provided as the sole argument to the program, the "Nibble..." messages are printed on the console and the "riddle" is solved (when you return, you only C3). [Figure 5](#) shows that the flag is:

i_w0uld_l1k3_to_RETurn_this_joke@flare-on.com

Appendix A: Crash Analysis

in the example from [Figure 1](#), where the argument "asdf" was provided, the sum of 0x50 (from the byte array) and 0x61 (the byte value of the ASCII letter 'a' from the command-line argument). This results in the buffer containing an invalid instruction sequence that precipitates an access violation, as shown in [Listing 2](#).

```
0:000> u 0x9b0000
009b0000 b100      mov     cl,0
009b0002 0000      add     byte ptr [eax],al
009b0004 0000      add     byte ptr [eax],al
009b0006 0000      add     byte ptr [eax],al
```

Listing 2: Disassembly of example buffer with argument "asdf"

In this example, the first instruction is valid and violates no rules, but the second and following instructions dereference the `eax` register, which contains a small value within the printable ASCII range due to its being populated from the command-line argument. The ensuing access to the null page causes an access violation exception.

