Flare-On 5: Challenge 7 Solution – WorldOfWarcraft.exe

Challenge Author: Ryan Warns

Summary

This challenge implements a 32-bit Windows binary meant to run in a Windows on Windows (WOW) environment.

Analysis

I often start my analysis of samples by quickly skimming the output of static analysis tools and looking through IDA. Performing basic static analysis on the binary we see that WorldOfWarcraft.exe is a simple 32-bit DLL. Running strings.exe on this binary shows us several strings that look like they might be related to the key.

```
USER32
WS2_32
%s@FLARE-On.com
Cannot read payload!
n1ght_4lve$_R_c00L.bin
A_l1ttl3_P1C_0f_h3aV3n
RSDS
R:\objchk_win7_x86\i386\WorldOfWarcraft.pdb
```

Figure 1 - strings in WorldOfWarcraft.exe

Opening the binary in IDA we can see that the binary doesn't appear to implement much in the way of functionality, with the main function only calling 3 subroutines. The subroutine at address `0x1001A60` contains references to our strings of interest.

```
strcpy(v4, "A_l1ttl3_P1C_0f_h3aV3n");
v5[0] = 21;
v5[1] = 55;
v5[2] = 93;
v5[3] = 66;
v5[4] = 43;
v5[5] = 69;
v5[6] = 31;
v5[7] = 108;
v5[8] = 43;
v5[9] = 56;
v5[10] = 2;
v5[11] = 28;
v5[12] = 40;
v5[13] = 66;
v5[14] = 86;
v5[15] = 49;
v5[16] = 15;
v5[17] = 108;
v5[18] = 10;
v5[19] = 101;
v5[20] = 74;
v5[21] = 49;
v5[22] = 0;
ms_exc.registration.TryLevel = 0;
if ( sub_1001910("n1ght_4lve$_R_c00L.bin", (int)&v3, (int)&v6) )
{
  for ( i = 0; i < &v4[strlen(v4) + 1] - &v4[1]; ++i )
    *(_BYTE *)(i + v3) ^= v4[i];
  for ( j = 0; j < &v4[strlen(v4) + 1] - &v4[1]; ++j )
  {
```

Figure 2 - Decompilation of function using interesting strings

I've cleaned up the decompilation in the screenshot above to be slightly more accurate. Quickly skimming `sub_1001910` reveals that this function grabs the contents of a file, so it looks like `sub_0x1001A60` will read the file `n1ght_4lve$_R_c00L.bin` and XOR the contents against the string `A_l1ttl3_P1C_0f_h3aV3n`. The result of this operation is compared to a static array on the stack, and if the two match the sample will print our key.

Thinking perhaps you might have gotten lucky and the maintainers of the Flare-On competition uploaded a buggy sample, you quickly curtail your analysis and calculate the data that the malware is looking for:

```
Th1s_1s_th3_wr0ng_k3y_
```

Figure 3 - Result of manually decoding the XOR loop

Additionally, running the sample reveals that not only do we not see the error message in the above decompilation, but a print message not shown in the IDA. Note that these screenshots are taken from my FLARE VM. Despite being a 32-bit binary, the sample appears to do nothing if run from a 32-bit operating system.

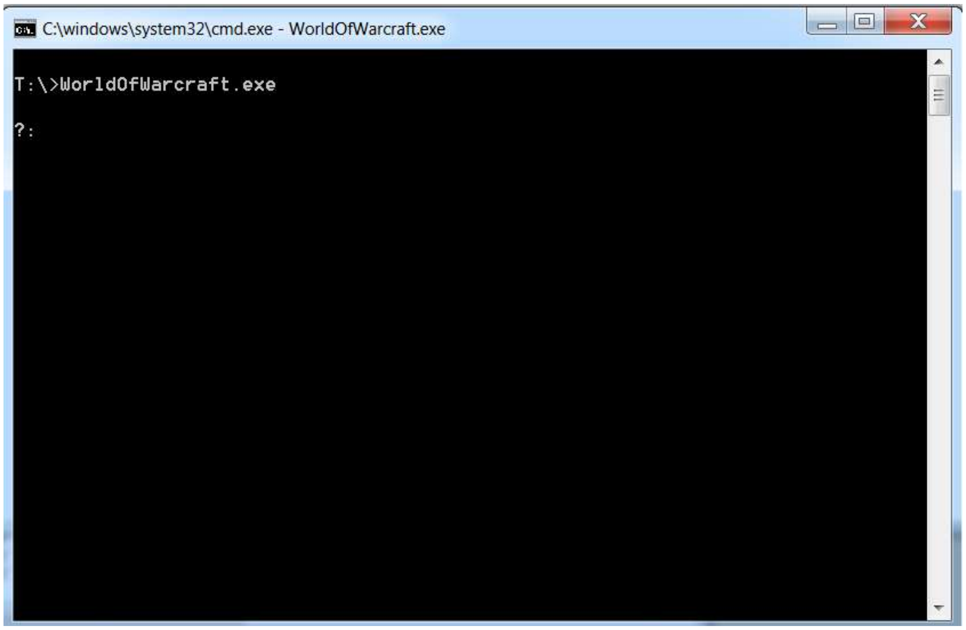

Figure 4 - Result of running WorldOfWarcraft.exe from the command line on a 64-bit system

Knowing that the initial strings are red herrings, we can start by looking at our main function to discover where the program diverges. The subroutine at address `0x1017D0` loads `WS2_32.dll`. The subroutines at `0x1001800` and

`0x1001600` validate that the sample is running in its intended environment. The function at address `0x1001800` calls `GetVersionExA` to determine the operating system it is executing on. The sample expects the major version to equal `6` and the minor version to equal `1`. Per MSDN these values correspond to Windows 7 or Windows Server 2008.

The function at address `0x1001600` is more complicated. The subroutines at address `0x1001410` and `0x10012A0` perform an export lookup using a simple ROR-based hash algorithm. These subroutines ultimately return the function `NtQueryInformationProcess` from NTDLL. This function is called with 26 as the `ProcessInformationClass`, which according to MSDN is `ProcessWow64Information`. This information class returns whether the target process (the current process in this case) is running in a WOW64 environment: if ProcessInformation is populated by the function, the process is a WOW64 process.

This information means that this binary expects to be running on a 64-bit Windows 7 operating system, running in a WOW environment.

Windows on Windows

Microsoft has consistently prioritized backwards compatibility throughout the development of the Windows operating system. One aspect of this is the Windows on Windows (WOW, or WOW64) subsystem, which provides a small compatibility layer so that 32-bit executables can run natively on a 64-bit system.

```
0:004> lm
start    end        module name
00c60000 00d20000   calc       (deferred)
72bd0000 72c0c000   oleacc     (deferred)
72c10000 72dae000   COMCTL32   (deferred)
72ef0000 72f03000   dwmapi     (deferred)
72f10000 7300b000   WindowsCodecs   (deferred)
73010000 73019000   VERSION    (deferred)
73030000 73062000   WINMM      (deferred)
73520000 735a0000   UxTheme    (deferred)
735a0000 73730000   gdiplus    (deferred)
74a30000 74a3c000   CRYPTBASE   (deferred)
74a40000 74aa0000   SspiCli    (deferred)
74b00000 74b0a000   LPK        (deferred)
74b10000 74bb0000   ADVAPI32   (deferred)
74bb0000 74bf6000   KERNELBASE   (deferred)
74dc0000 74e17000   SHLWAPI    (deferred)
74e50000 74fac000   ole32      (deferred)
750f0000 7519c000   msvcrt     (deferred)
75210000 75310000   USER32     (deferred)
75310000 75329000   sechost    (deferred)
753d0000 7601a000   SHELL32    (deferred)
761a0000 76200000   IMM32      (deferred)
76200000 76283000   CLBCatQ    (deferred)
762a0000 7632f000   OLEAUT32   (deferred)
764c0000 76550000   GDI32      (deferred)
76550000 7661c000   MSCTF      (deferred)
76820000 76930000   kernel32   (pdb symbols)
76930000 76a20000   RPCRT4     (deferred)
76a50000 76aed000   USP10      (deferred)
76ef0000 77070000   ntdll      (pdb symbols)
```

```
kd> lmuD
start             end                 module name
00000000`00c60000 00000000`00d20000   calc       (deferred)
00000000`73730000 00000000`73738000   wow64cpu   (deferred)
00000000`73740000 00000000`7379c000   wow64win   (deferred)
00000000`737a0000 00000000`737df000   wow64      (deferred)
00000000`76d10000 00000000`76eb9000   ntdll      (pdb symbols)
```

Figure 5 - Difference in loaded modules from 32-bit view (top) vs 64-bit (bottom)

At a high level, when a WOW process starts the OS creates a normal 64-bit "shell" process with a fake 32-bit environment, including 32-bit versions of the normal system DLLs (kernel32, ntdll, etc) and important process structures (PEB, TEB, etc). These structures allow the 32-bit binary to function as it if was running natively, despite being inside a 64-bit process. We can see how this is constructed by attaching 32-bit and 64-bit variants of windbg to a WOW process and dumping the loaded DLLs.

Under normal system operation, most Windows APIs end up calling an exported function inside of ntdll. These functions' names start with Zw or Nt and their only job is to issue a request to the kernel, usually via a sysenter or syscall instruction.

```
kd> u ntdll!ZwCreateFile
ntdll!ZwCreateFile:
00000000`76d61860 4c8bd1          mov     r10,rcx
00000000`76d61863 b852000000      mov     eax,52h
00000000`76d61868 0f05            syscall
00000000`76d6186a c3              ret
```

Figure 6 - Disassembly of normal system calls in NTDLL

In a WOW64 environment there are only four 64-bit DLLs loaded: `wow64cpu.dll`, `wow64win.dll`, `wow64.dll`, `ntdll.dll`. The emulated 32-bit environment works the same way described above, but instead of making system calls the 32-bit NTDLL uses a far jump to pass execution to `wow64cpu!CpupReturnFromSimulatedCode`. `wow64cpu` tracks the transitions between 32-bit and 64-bit code and maintains the 64-bit stack for threads executing in WOW mode.

```
0:004> u ntdll!NtCreateFile L6
ntdll!NtCreateFile:
76f100a4 b852000000      mov     eax,52h
76f100a9 33c9            xor     ecx,ecx
76f100ab 8d542404        lea     edx,[esp+4]
76f100af 64ff15c0000000  call    dword ptr fs:[0C0h]
76f100b6 83c404          add     esp,4
76f100b9 c22c00          ret     2Ch
0:004> dd fs:0c0 l1
0053:000000c0  73732320
0:004> u 73732320 L1
73732320 ea1e2773733300  jmp     0033:7373271E
```

```
kd> u 0x33:7373271e
wow64cpu!CpupReturnFromSimulatedCode:
0033:00000000`7373271e 67448b0424       mov     r8d,dword ptr [esp]
0033:00000000`73732723 458985bc000000   mov     dword ptr [r13+0BCh],r8d
0033:00000000`7373272a 4189a5c8000000   mov     dword ptr [r13+0C8h],esp
0033:00000000`73732731 498ba42480140000 mov      rsp,qword ptr [r12+1480h
0033:00000000`73732739 4983a4248014000000 and    qword ptr [r12+1480h],0
0033:00000000`73732742 448bda           mov     r11d,edx
wow64cpu!TurboDispatchJumpAddressStart:
0033:00000000`73732745 41ff24cf         jmp     qword ptr [r15+rcx*8]
```

Figure 7 - Transition from 32-bit (left) to 64-bit code (right)

There are several instructions, including far jumps, which use a segment selector. Other than the FS and GS segment registers, segmentation is a processor feature generally ignored by Windows which was originally used to provide a way to segregate and organize memory access on 16-bit systems. To support WOW Windows has set the appropriate Global Descriptor Table (GDT) entry for the `0x33` segment selector selector shown above. The far jump shown in Figure 7 is the mechanism used to jump to 64-bit code. Executing a similar instruction with a segment selector of `0x23` will jump back to 32-bit.

64 Bit Analysis

Now that we understand the execution environment for this sample we can continue our analysis. Despite being an executable, there is an export named `X64Call`. This function breaks IDA's stack analysis and uses the `retf` instruction.

This function implements a variant of what was first publicly described as Heaven's Gate[1]. Because segmentation is a feature of the processor, once we know the correct segment selectors to use Heaven's Gate is a technique to manually jump between 32-bit and 64-bit code without going through the normal WOW DLLs.

```
15A0 ?X64Call@@YG_KPAX@Z proc far            ; CODE XREF: sub_
15A0                                         ; DATA XREF: .tex
15A0
15A0 var_20          = dword ptr -20h
15A0 var_18          = qword ptr -18h
15A0 var_C           = dword ptr -0Ch
15A0 var_8           = dword ptr -8
15A0 var_4           = dword ptr -4
15A0
15A0                 mov     edi, edi
15A2                 push    ebp
15A3                 mov     ebp, esp
15A5                 sub     esp, 18h
15A8                 mov     eax, [ebp+8]
15AB                 cdq
15AC                 mov     dword ptr [ebp+var_18], eax
15AF                 mov     dword ptr [ebp+var_18+4], edx
15B2                 mov     [ebp+var_8], 0
15B9                 mov     [ebp+var_4], 0
15C0                 mov     [ebp+var_C], esp
15C3                 and     sp, 0FFF8h
15C7                 push    33h ; '3'
15C9                 call    $+5
15CE                 add     [esp+20h+var_20], 5
15D2                 retf
15D2 ?X64Call@@YG_KPAX@Z endp ; sp-analysis failed
15D2
15D3 ; ---------------------------------------------------------
15D3                 sub     esp, 20h
15D6                 call    dword ptr [ebp-18h]
15D9                 add     esp, 20h
15DC                 call    $+5
15E1                 mov     dword ptr [esp+4], 23h ; '#'
15E9                 add     dword ptr [esp], 0Dh
15ED                 retf
15EE ; ---------------------------------------------------------
15EE                 mov     esp, [ebp-0Ch]
15F1                 mov     esp, ebp
15F3                 pop     ebp
15F4                 retn    4
```

Figure 8 - Disassembly of X64Call

[1] http://rce.co/knockin-on-heavens-gate-dynamic-processor-mode-switching/

X64Call is used in subroutine at address 0x1001740. Analyzing this function, we can see that sub_1001740 accesses a global data buffer and XORs it with 0xDEEDEEB; the result of this decoding is a 64-bit DLL. X64Call is used to jump to 64-bit code and execute offset 0x580 in this decoded DLL.

Offset 0x580 corresponds to the single exported function inside the DLL. This function implements a variant of in-memory DLL loading, but instead of loading a target DLL the sample is loading itself into memory. The function begins by resolving several imports using the same hash algorithm detailed earlier in this report. Note that because this DLL is running in the 64-bit "heaven" area of the process all imports must come from NTDLL. After finding its PE header in memory, the sample fully loads itself. Note that the subroutine at 0x18000360A is used to jump to the fully loaded copy of the DLL; the debugger will not regain execution after stepping over this function.

```
sub_18000360A    proc near              ;
                 pop     rax
                 sub     rax, rcx
                 add     rdx, rax
                 add     rdx, r8
                 jmp     rdx
sub_18000360A    endp ; sp-analysis failed
```

Figure 9 - Function that calculates and jumps to the loaded 64-bit DLL

Once fully loaded, the new DLL begins modifying the 32-bit program state. The function at address 0x180001BA0 once again calls NtQueryInformationProcess with ProcessWow64Information but uses the return value like a pointer. In a WOW64 process (on this OS) NtQueryInformationProcess returns the 32-bit PEB for this ProcessInformationClass.

The subroutine at 0x1800023B0 uses this PEB and the same hashing function from previous lookups to resolve the base address of and functions exported by the 32-bit NTDLL. The subroutine at address 0x180001950 is called with the address of the 32-bit ZwDeviceIoControlFile. ZwDeviceIoControlFile is the NTDLL API called by DeviceIoControl, used to send IOCTLs to drivers. This subroutine searches this API's function body for the byte 0xB8, which corresponds to mov eax, <immediate value>. When using the syscall instruction, EAX contains the syscall number; this subroutine is meant to dynamically determine the system call number for ZwDeviceIoControlFile. This technique of dynamically walking code is used in more sophisticated malware samples for OS compatibility reason.

The subroutine at address 0x180003100 looks similar to the loader functionality from 0x1800032F0, but this function is used to load another 32-bit binary embedded in the .data section of this DLL.

```
.data:0000000180005120 dword_180005120 dd 1A00h
.data:0000000180005124                 align 10h
.data:0000000180005130 unk_180005130   db  4Dh ; M
.data:0000000180005131                 db  5Ah ; Z
.data:0000000180005132                 db  90h
```

Figure 10 - 32-bit DLL embedded in the 64-bit DLL's .data section

The syscall number resolved earlier is used in the function at address `0x1800013E0`. This function uses the syscall number resolved earlier to dynamically build a buffer of executable code. The sample then replaces the pointer at `fs:0C0` with this executable code buffer. Per earlier in this report, this offset in the PEB is used when transitioning between 32-bit and 64-bit code. The dynamic code buffer redirects syscalls from `ZwDeviceIoControlFile` to the subroutine at address `0x180001660`.

```
32.kd:x86> u 0x001e0000
00000000`001e0000 3d04000000      cmp     eax,4
00000000`001e0005 7406            je      001e000d
00000000`001e0007 6820237d73      push    offset wow64cpu!X86SwitchTo64BitMode
00000000`001e000c c3              ret
00000000`001e000d 8bcc            mov     ecx,esp
00000000`001e000f 6681e4f8ff      and     sp,0FFF8h
00000000`001e0014 6683ec20        sub     sp,20h
00000000`001e0018 ea601609003300  jmp     0033:00091660
```

Figure 11 - Dynamically constructed hook code inserted into the PEB

Finally, the function at address `0x1800035A2` searches the stack for a marker. By adding the values `0xCCB9984` and `0x1234567` the sample attempts to obfuscate that it is searching the stack for the value `0xDEEDEEB`, the same XOR key used earlier to decode this DLL. The sample overwrites 4 bytes before this location, which contains the original return address from `X64Call`, with address `0x10001220` from the newly loaded 32-bit DLL.

Final Analysis

This second 32-bit binary (helpfully labeled as `crackme` by the pdb string) is very small; only two functions are called from the subroutine we've returned to. After resolving several function pointers, the subroutine at address `0x10001390` implements the prompt we saw earlier.

This function runs a loop 29 times, where for each iteration the input from the prompt is taken, passed to `htons` and used in the `connect` API (to connect to localhost). After concluding the sample calls `recv` and prints the result to the screen in the format we want for our key.

The code does not do anything with the result of the connect call, so it may appear we have no indication of where to go next. However, a breakpoint on the `ZwDeviceIoControlHook` we saw earlier will be hit as part of the `connect` API call.

```
32.kd:x86> k
 # ChildEBP          RetAddr
00 002af550 736c3ad6 ntdll_76ef0000!ZwDeviceIoControlFile
01 002af604 736c5f26 mswsock!WSPBind+0x1fc
02 002af690 736c5da4 mswsock!SockDoConnect+0x25b
03 002af6b4 751d6c2f mswsock!WSPConnect+0x1f
04 002af700 000a14f6 WS2_32!connect+0x52
WARNING: Frame IP not in any known module. Following frames may be wrong.
05 002af9b0 000a123e 0xa14f6
06 002af9e0 00a81da9 0xa123e
07 002afa24 768333ca WorldOfWarcraft!X64Call+0x809
08 002afa30 76f29ed2 kernel32!BaseThreadInitThunk+0xe
09 002afa70 76f29ea5 ntdll_76ef0000!__RtlUserThreadStart+0x70
0a 002afa88 00000000 ntdll_76ef0000!_RtlUserThreadStart+0x1b
```

Figure 12 - Stack trace from connect to ZwDeviceIoControlFile

Socket functionality on Windows is implemented using I/O requests to `afd.sys`, the Ancillary Function Driver for Winsock. This driver implements management functionality for the Windows networking stack. Back in our hook function, the second parameter to this function is a pointer to the parameter stack from the 32-bit ntdll!ZwDeviceIoControlFile. According to MSDN[2] the sixth parameter to this function is the ioControlCode (IOCTL), designating what functionality the driver should run. We can see that the hook function switches on and implements several control codes but the only two we care about for this analysis are `0x12007` and `0x12017` which correspond to connect and recv, respectively. Note that there are other IOCTLs that end up in the hook code, but they stem from the `socket` api.

[2] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-ntdeviceiocontrolfile

```
case 0x12007:
  v8 = *(unsigned int *)(v6 + 32);
  qmemcpy(&v3, (const void *)(v8 + 14), 2ui64);
  if ( (unsigned int)dword_180006BE8 <= 0x74ui64 )
  {
    if ( v4 == dword_180006B40[dword_180006BE8] )
    {
      for ( i = dword_180006BE8 + 1; i < 29; ++i )
        dword_180006B40[i] ^= v4;
    }
    byte_180006BB8[dword_180006BE8] ^= LOBYTE(dword_180006B40[dword_180006BE8]);
    ++dword_180006BE8;
    *(_QWORD *)v10 = 0xC0000005i64;
    v9 = 0xC0000005i64;
  }
  else
  {
    *(_QWORD *)v10 = 5i64;
    v9 = 0xC0000005i64;
  }
  break;
case 0x12017:
  *(_QWORD *)&v7 = *(unsigned int *)(v6 + 32);
  v5 = *(unsigned int *)v7;
  qmemcpy((void *)*(unsigned int *)(v5 + 4), byte_180006BB8, *(unsigned int *)v5);
  *(_DWORD *)v10 = 0;
  *(_DWORD *)(v10 + 4) = *(_DWORD *)v5;
  break;
```

Figure 13 - Relevant decompilation of the ZwDeviceIoControlFile hook

The IOCTL handler for the `connect` API interacts with two global buffers. The seventh parameter to `ZwDeviceIoControlFile` is the input buffer for this API, which for the connect API call contains the `sockaddr_in` structure at offset 0xC. The hook function reads the port from this structure. One of the global buffers, which is also referenced in the `recv` IOCTL, is always XORed with the port number. For the other buffer, the hook function maintains a global counter of the number of times it has been called and if the port matches the current position in the buffer, the rest of the buffer is XORed with the port.

Now we have a good understanding of all the moving pieces of this binary:

1. The sample runs as a `WOW64` process with red herring code
2. The sample uses the function `X64Call` to jump to the 64-bit part of the process
3. A 64-bit DLL is self-loaded
4. The 64 bit DLL hooks the pointer in the PEB normally used to redirect system calls through `wow64cpu!CpupReturnFromSimulatedCode`

5. The 64 bit DLL then loads and redirects execution to another 32 bit DLL which contains the challenge prompt

6. The 32 bit DLL goes through a loop calling the `connect` API to "connect" to whatever port the user enters

7. The `connect` API is routed through the hooked pointer in the PEB and the 64-bit DLL performs an XOR operation on one or more lists depending on the port entered

Using the port check as a hint, entering the matching port number in the second list for each 29 iterations will give us the key: `P0rt_Kn0ck1ng_0n_he4v3ns_d00r@flare-on.com`

Appendix 1: Full list of correct port numbers

```
15
88
54
22
124
241
79
100
80
24
53
11
55
200
66
72
94
172
168
37
192
56
184
138
7
18
78
85
27
```