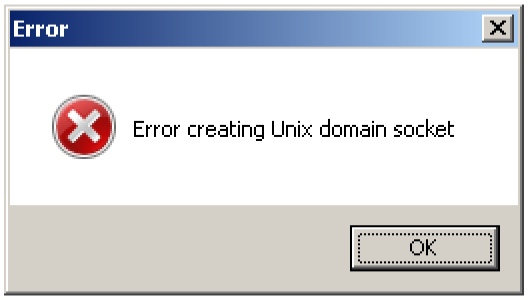
Flare-On 7: Challenge 8 – Aardvark

Challenge Author: Jacob Thompson

The name Aardvark is in reference to the AARD code[1], a section of code in early versions of Windows that verified they were running on Microsoft's version of DOS, as opposed to another version. This challenge binary verifies that it is running on Microsoft's Windows Subsystem for Linux, as opposed to another version of Linux.

Running ttt2.exe on Windows 7 produces an error dialog (Figure 1):


Error

Error creating Unix domain socket

Figure 1. The binary fails on Windows 7 due to an error creating a Unix domain socket.

A Unix domain socket is a special type of socket on POSIX operating systems designed for communication between processes on the same machine, with greater efficiency than using IP over the loopback 127.0.0.1 interface.

Reviewing the socket() call in IDA (Figure 2), we see that the code is indeed calling `socket(1, 1, 0);` which would have been compiled from `socket(AF_UNIX, SOCK_STREAM, 0)`;. In addition, we see indications that the name of the Unix domain socket is `496b9b4b.ed5`.

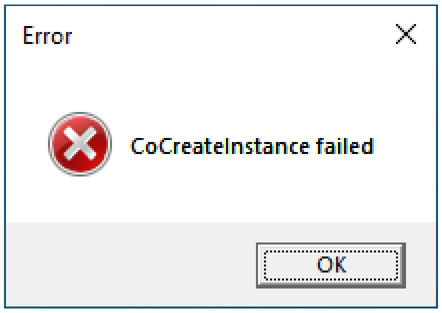
[1] https://en.wikipedia.org/wiki/AARD_code

```
loc_140001649:
; __unwind { // __GSHandlerCheck
mov     [rsp+3B8h+arg_0], rbx
xor     eax, eax
lea     r8, FileName    ; "496b9b4b.ed5"
mov     ebx, 1
 ...
xor     r8d, r8d        ; protocol
mov     edx, ebx        ; type
mov     ecx, ebx        ; af
call    cs:socket
```

Figure 2. The binary attempts to create a Unix domain socket on startup.

It makes sense that the socket call for AF_UNIX fails since this is Windows 7, not Unix. Why would a Windows EXE file attempt to create such a socket? It turns out that Windows began supporting Unix domain sockets with Windows 10 build 17063 (https://devblogs.microsoft.com/commandline/af_unix-comes-to-windows/), so we'll have to do the rest of the analysis on a sufficiently recent Windows 10.

When we re-run ttt2.exe on a fresh install of Windows 10 2004, we get a new error – `CoCreateInstance()` failed, as shown in Figure 3.


Error

CoCreateInstance failed

Figure 3. The CoCreateInstance function fails when running on a fresh install of Windows 10 2004.

`CoCreateInstance()` is the Windows COM function that allows a program to specify a COM Class ID and Interface ID, and obtain a pointer table allowing the program to call functions on that interface. If we use IDA's Imports tab and cross-reference feature to find the call to `CoCreateInstance()`, we find there are many calls. Using a debugger, we can set a breakpoint on the `CoCreateInstance()` function and find that the first call occurs at 0x14000223c (running on Windows 10 2004). Then we can go to that address in IDA, as shown in Figure 4, and see the parameters to the `CoCreateInstance()` function.

```
loc_140002204:
; __unwind { // __GSHandlerCheck
mov     [r11-18h], rsi
lea     rax, [rbp+250h+var_2C8]
mov     [r11-20h], r14
lea     r9, riid        ; riid
xor     edi, edi
mov     [r11-28h], r15
mov     r15, r8
mov     [rbp+250h+var_2C8], rdi
mov     r14d, edx
mov     [rsp+380h+ppv], rax ; ppv
mov     rsi, rcx
xor     edx, edx        ; pUnkOuter
lea     r8d, [rdi+4]    ; dwClsContext
lea     rcx, rclsid     ; rclsid
call    cs:CoCreateInstance
```

Figure 4. When running on Windows 10 2004, the first CoCreateInstance call occurs at 0x14000223c.

We can use IDA to determine what the rclsid and riid values are, as shown in Figure 5. That will help us determine what this binary is trying to do with COM.

```
.data:000000014001E008 ; IID rclsid
.data:000000014001E008 rclsid    dd 4F476546h           ; Data1
.data:000000014001E008                                  ; DATA XREF:
sub_140001B10+55↑o
.data:000000014001E008                                  ; sub_140001D60+54↑o ...
.data:000000014001E008           dw 0B412h              ; Data2
.data:000000014001E008           dw 4579h               ; Data3
.data:000000014001E008           db 0B6h, 4Ch, 12h, 3Dh, 0F3h, 31h, 0E3h, 0D6h; Data4
.data:000000014001E018 ; IID riid
.data:000000014001E018 riid      dd 536A6BCFh           ; Data1
.data:000000014001E018                                  ; DATA XREF:
sub_140001B10+4E↑o
.data:000000014001E018                                  ; sub_140001D60+30↑o ...
.data:000000014001E018           dw 0FE04h              ; Data2
.data:000000014001E018           dw 41D9h               ; Data3
.data:000000014001E018           db 0B9h, 78h, 0DCh, 0ACh, 2 dup(0A9h), 0B5h, 0B9h;
Data4
```

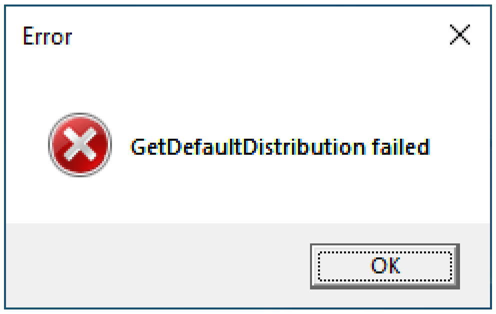
Figure 5. IDA cross-references allow us to determine the COM Class ID and Interface ID.

This tells us that the binary calls `CoCreateInstance()` with a CLSID of {4f476546-b412-4579-b64c-123df331e3d6} and an IID of {536a6bcf-fe04-41d9-b978-dcaca9a9b5b9}. Performing a Google search for the CLSID, as of this writing there are only nine results, and for the IID, only one result. As can be confirmed on a Windows 10 machine with Windows Subsystem for Linux installed, this IID and CLSID correspond to the ILxssUserSession interface of the LxssUserSession class. It turns out that this is an undocumented COM interface that an application can use to interact with WSL without going through the wsl.exe command line utility.

Despite being undocumented, several analysts have publicly published information that they have reverse-engineered about this interface, including GitHub user Biswa96 (https://github.com/Biswa96/wslbridge2/blob/master/src/GetVmId.cpp, https://github.com/Biswa96/WslReverse/blob/master/common/LxssUserSession.h) and Alex Ionescu (https://github.com/ionescu007/lxss/blob/master/inc/lxssmanager.h).

The reason there are multiple calls to `CoCreateInstance()` is that this interface is undocumented, and Microsoft therefore has changed it between different builds of Windows 10. The interface definition changed in Windows 10 1803, 1809, and 1903, and has remained compatible since 1903. This (along with testing for WSL 1 vs. WSL 2) is why the binary has multiple calls to `CoCreateInstance()` and tests the Windows build number to determine which to call.

The use of the ILxssUserSession interface, and the fact that `CoCreateInstance()` fails, suggests that WSL must be installed. After using the “Turn Windows features on or off” control panel option to enable WSL and rebooting, ttt2.exe can be re-launched. This time, the binary complains that the `GetDefaultDistribution()` call failed (Figure 6).



GetDefaultDistribution failed

Figure 6. The binary complains that GetDefaultDistribution failed.

WSL does not include any Linux software itself; it manages multiple distributions. It is possible to download and install Ubuntu 20.04 without going through the Windows store by downloading the install package from https://docs.microsoft.com/en-us/windows/wsl/install-manual and launching it.

After downloading, installing, and launching Ubuntu 20.04 for the first time, we can try to open ttt2.exe again. This time, we are greeted with a Tic-Tac-Toe game (Figure 7). As shown in Figure 8, however, the game is rigged as the computer player always goes first as the “X” player, and has just enough intelligence to not allow the “O” player to ever win.

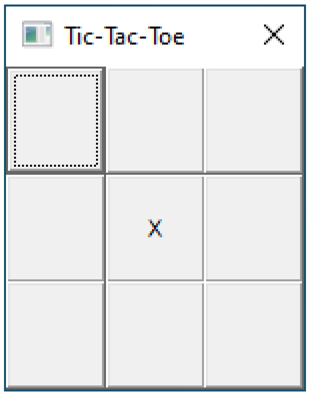


Figure 7. Once WSL and a Linux distribution are installed, ttt2.exe opens a Tic-Tac-Toe game.

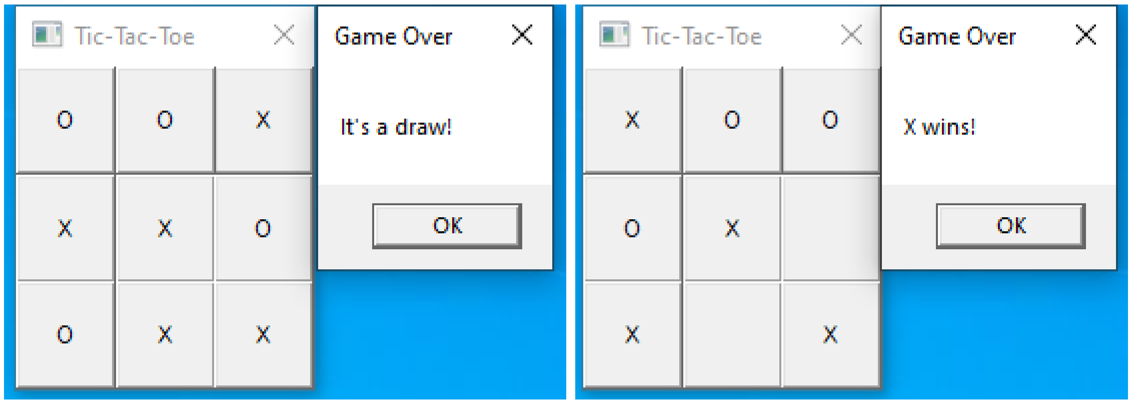
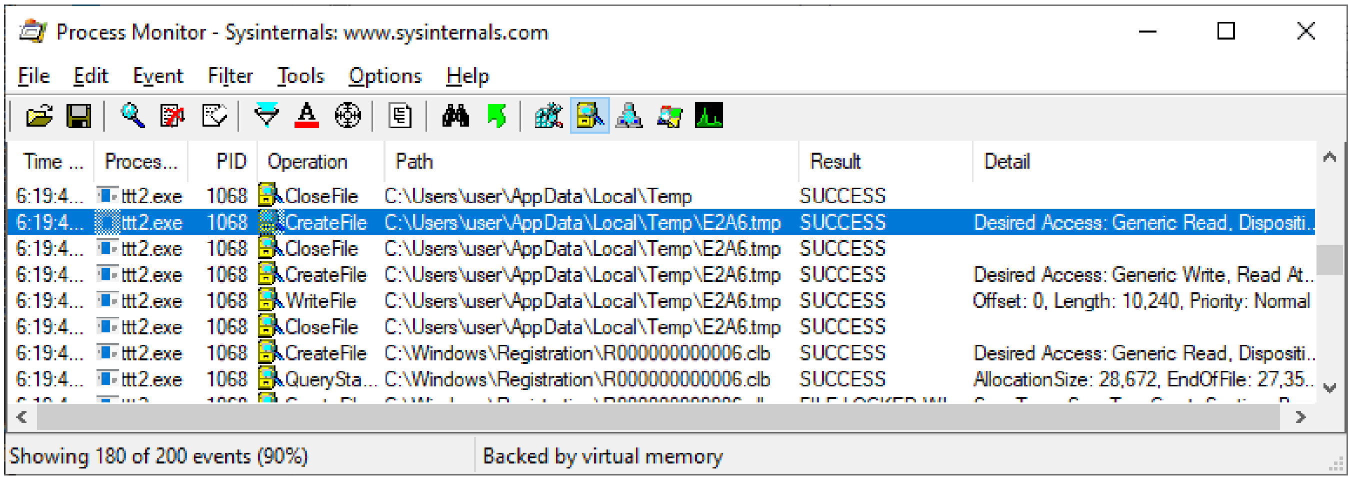


Figure 8. The Tic-Tac-Toe game has a computer player that will not allow the human player to win.

The remainder of the challenge therefore is how to win the game!

Using Procmon, we can find that ttt2.exe always deletes and re-creates a zero-byte file named %TEMP%\496b9b4b.ed5. This is a placeholder file for the program's Unix domain socket.

Another 10KB file is created under %TEMP%, with a random filename, as shown in Figure 9.

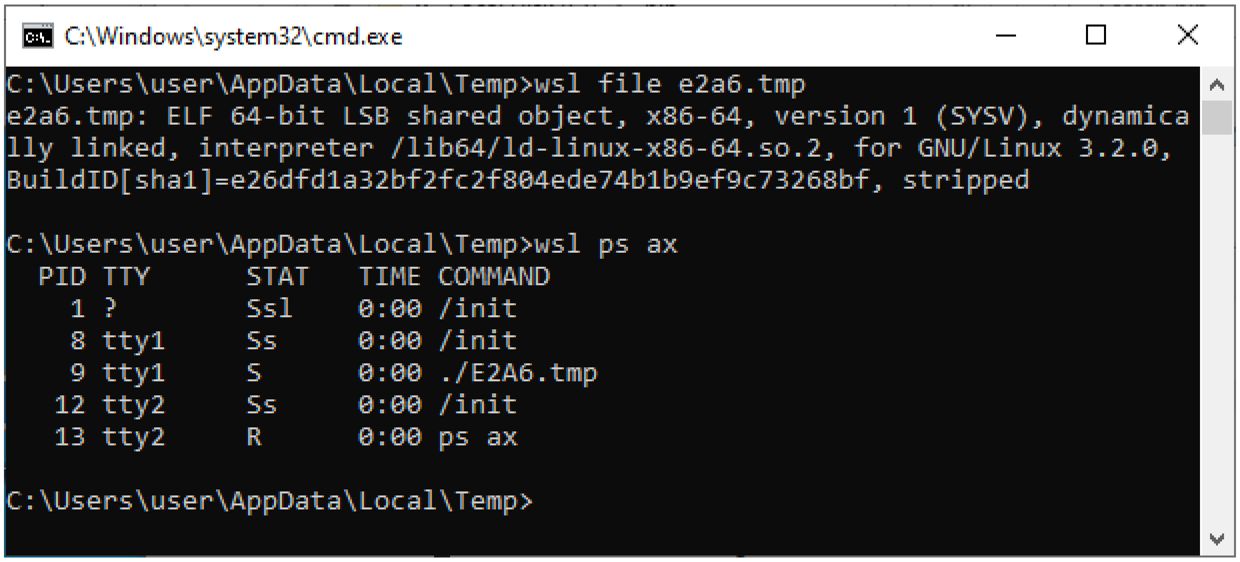

Process Monitor - Sysinternals: www.sysinternals.com

```
Time ...  Proces...  PID   Operation    Path                                          Result    Detail
6:19:4... ttt2.exe   1068  CloseFile    C:\Users\user\AppData\Local\Temp              SUCCESS
6:19:4... ttt2.exe   1068  CreateFile   C:\Users\user\AppData\Local\Temp\E2A6.tmp     SUCCESS   Desired Access: Generic Read, Dispositi...
6:19:4... ttt2.exe   1068  CloseFile    C:\Users\user\AppData\Local\Temp\E2A6.tmp     SUCCESS
6:19:4... ttt2.exe   1068  CreateFile   C:\Users\user\AppData\Local\Temp\E2A6.tmp     SUCCESS   Desired Access: Generic Write, Read At...
6:19:4... ttt2.exe   1068  WriteFile    C:\Users\user\AppData\Local\Temp\E2A6.tmp     SUCCESS   Offset: 0, Length: 10,240, Priority: Normal
6:19:4... ttt2.exe   1068  CloseFile    C:\Users\user\AppData\Local\Temp\E2A6.tmp     SUCCESS
6:19:4... ttt2.exe   1068  CreateFile   C:\Windows\Registration\R000000000006.clb     SUCCESS   Desired Access: Generic Read, Dispositi...
6:19:4... ttt2.exe   1068  QuerySta...  C:\Windows\Registration\R000000000006.clb     SUCCESS   AllocationSize: 28,672, EndOfFile: 27,35...

Showing 180 of 200 events (90%)        Backed by virtual memory
```

Figure 9. ttt2.exe always creates and writes 10KB to a temp file with a random name.

If we run the *file* utility, we find that e2a6.tmp is a 64-bit Linux ELF executable, and if we run the *ps* utility, we find a copy is already running (Figure 10):


C:\Windows\system32\cmd.exe

```
C:\Users\user\AppData\Local\Temp>wsl file e2a6.tmp
e2a6.tmp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamica
lly linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=e26dfd1a32bf2fc2f804ede74b1b9ef9c73268bf, stripped

C:\Users\user\AppData\Local\Temp>wsl ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ssl    0:00 /init
    8 tty1     Ss     0:00 /init
    9 tty1     S      0:00 ./E2A6.tmp
   12 tty2     Ss     0:00 /init
   13 tty2     R      0:00 ps ax

C:\Users\user\AppData\Local\Temp>
```

Figure 10. The temp file written by ttt2.exe is an ELF executable that executes inside WSL.

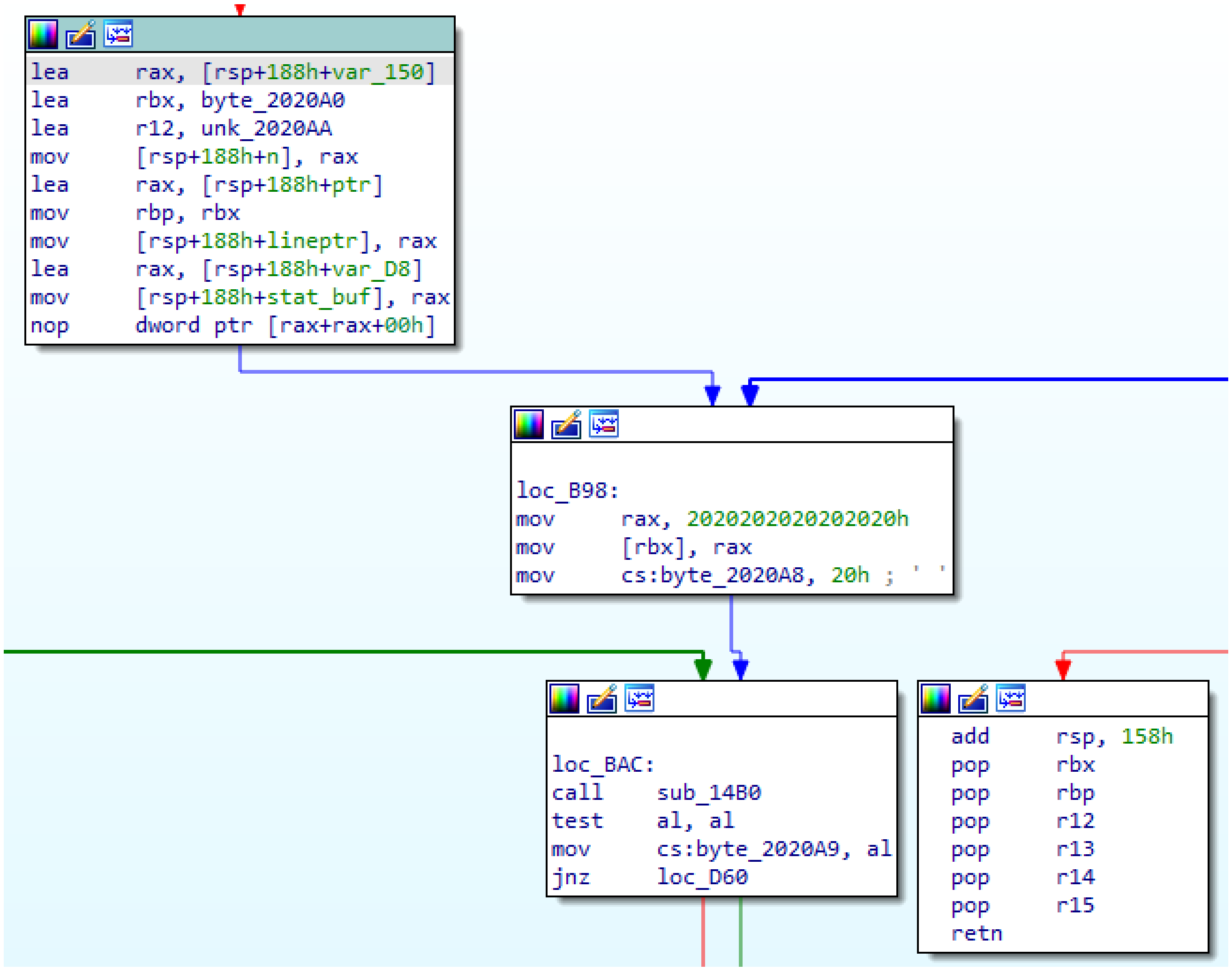
Next, we can take a look at an excerpt from the ELF executable's main() function in IDA in Figure 11. If we look at the arguments to the socket() call, we can see that, after accounting for byte order, the program opens the same 496b9b4b.ed5 Unix domain socket that ttt2.exe creates.

```
MOV     ECX, 1
MOV     RAX, 'B4B9B694'
MOV     EDI, 1             ; DOMAIN
MOV     [RSP+188H+ADDR.SA_FAMILY], CX
MOV     QWORD PTR [RSP+188H+ADDR.SA_DATA], RAX
MOV      DWORD PTR [RSP+188H+ADDR.SA_DATA+8], '5DE.'
MOV      [RSP+188H+ADDR.SA_DATA+0CH], 0
```

Figure 11. The ELF binary opens the same Unix domain socket that ttt2.exe created.

This means that ttt2.exe and the ELF binary perform interprocess communication despite one being a Windows binary and one being for Linux, running inside WSL. Such is the capability of WSL.

Early in the ELF binary's main() function, the program has a loop where a variable at 0x2020a0 is initializes with 9 spaces—suspiciously like a [3][3] array. We correctly surmise that the variable at 0x2020a0 holds the current board state.



```
lea     rax, [rsp+188h+var_150]
lea     rbx, byte_2020A0
lea     r12, unk_2020AA
mov     [rsp+188h+n], rax
lea     rax, [rsp+188h+ptr]
mov     rbp, rbx
mov     [rsp+188h+lineptr], rax
lea     rax, [rsp+188h+var_D8]
mov     [rsp+188h+stat_buf], rax
nop     dword ptr [rax+rax+00h]
```

```
loc_B98:
mov     rax, 2020202020202020h
mov     [rbx], rax
mov     cs:byte_2020A8, 20h ; ' '
```

```
loc_BAC:
call    sub_14B0
test    al, al
mov     cs:byte_2020A9, al
jnz     loc_D60
```

```
add     rsp, 158h
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
retn
```

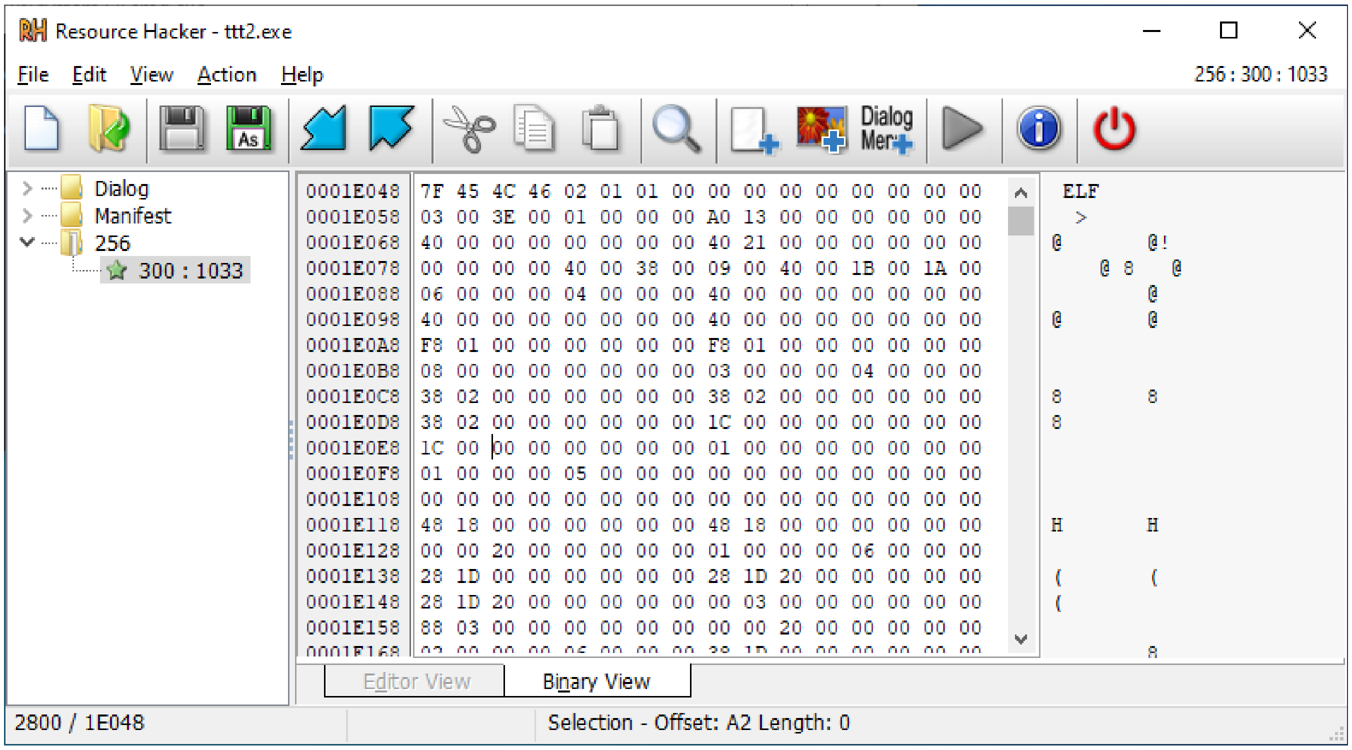
Figure 12. Early in the main function, the ELF binary initializes the board.

The simplest way to cheat and win the tic-tac-toe game is to "preload" the initial board state with some "O" marks. Because the computer player still goes first, if the supposedly-empty initial board is preloaded with two-ways for the human player to win, the computer player cannot block both opportunities on the “first” move.

To create the following “empty” board, we need to set the rax contents at loc_b98 to 202020204f204f4f.

O	O	
O		

Using Resource Hacker, we find the ELF file is stored in ttt2.exe as a resource, as shown in Figure 13. We know that the 2020202020202020 initial board load was at 0xb98 from Figure 12, so combining this with the file offset from Resource Hacker, we know the instruction must be at 0x1ebe0 within the ttt2.exe file.

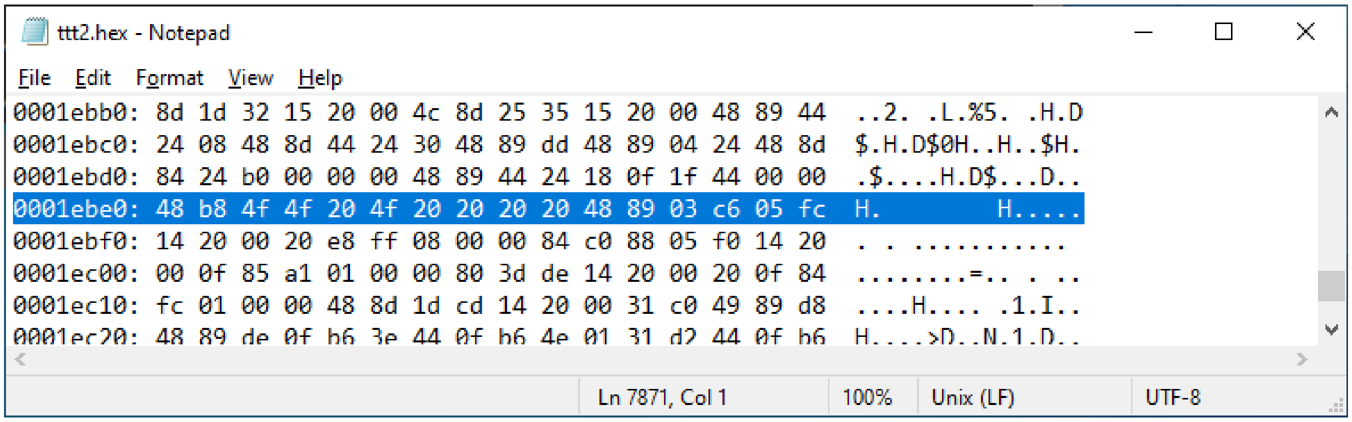


```
0001E048  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00   ELF
0001E058  03 00 3E 00 01 00 00 00 A0 13 00 00 00 00 00 00     >
0001E068  40 00 00 00 00 00 00 00 40 21 00 00 00 00 00 00  @       @!
0001E078  00 00 00 00 40 00 38 00 09 00 40 00 1B 00 1A 00      @ 8   @
0001E088  06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00          @
0001E098  40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  @       @
0001E0A8  F8 01 00 00 00 00 00 00 F8 01 00 00 00 00 00 00
0001E0B8  08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00
0001E0C8  38 02 00 00 00 00 00 00 38 02 00 00 00 00 00 00  8       8
0001E0D8  38 02 00 00 00 00 00 00 1C 00 00 00 00 00 00 00  8
0001E0E8  1C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
0001E0F8  01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00
0001E108  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001E118  48 18 00 00 00 00 00 00 48 18 00 00 00 00 00 00  H       H
0001E128  00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00
0001E138  28 1D 00 00 00 00 00 00 28 1D 20 00 00 00 00 00  (       (
0001E148  28 1D 20 00 00 00 00 00 00 03 00 00 00 00 00 00  (
0001E158  88 03 00 00 00 00 00 00 00 00 20 00 00 00 00 00
```

```
2800 / 1E048          Selection - Offset: A2 Length: 0
```

Figure 13. The dropped ELF executable is stored in ttt2.exe as a resource.

Armed with that information, we can use xxd and xxd -r as a hex editor to patch in the new initial board state (Figure 14):

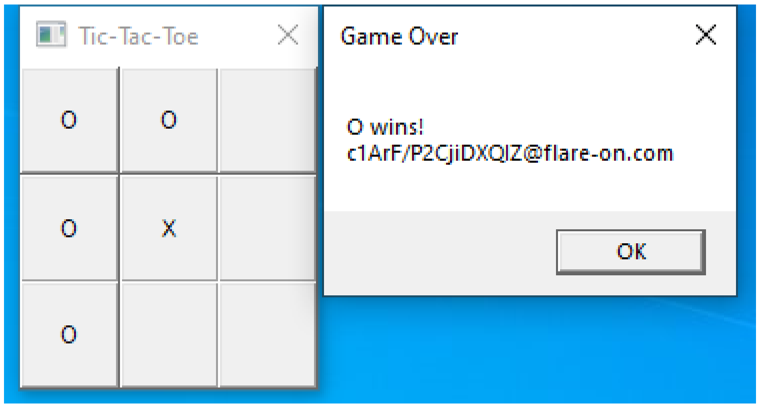


```
0001ebb0: 8d 1d 32 15 20 00 4c 8d 25 35 15 20 00 48 89 44  ..2. .L.%5. .H.D
0001ebc0: 24 08 48 8d 44 24 30 48 89 dd 48 89 04 24 48 8d  $.H.D$0H..H..$H.
0001ebd0: 84 24 b0 00 00 00 48 89 44 24 18 0f 1f 44 00 00  .$....H.D$...D..
0001ebe0: 48 b8 4f 4f 20 4f 20 20 20 20 48 89 03 c6 05 fc  H.          H.....
0001ebf0: 14 20 00 20 e8 ff 08 00 00 84 c0 88 05 f0 14 20  . . ...........
0001ec00: 00 0f 85 a1 01 00 00 80 3d de 14 20 00 20 0f 84  ........=.. . ..
0001ec10: fc 01 00 00 48 8d 1d cd 14 20 00 31 c0 49 89 d8  ....H.... .1.I..
0001ec20: 48 89 de 0f b6 3e 44 0f b6 4e 01 31 d2 44 0f b6  H....>D..N.1.D..
```

Ln 7871, Col 1 | 100% | Unix (LF) | UTF-8

Figure 14. The initial board state can be changed so as to give the human player the opportunity to win.

Playing the hacked tic-tac-toe game, we can now win and get the flag:



O wins!
c1ArF/P2CjiDXQIZ@flare-on.com

You will find that the flag does not appear in ttt2.exe anywhere, and also, we did not attempt to copy the ELF binary to a native Linux virtual machine and analyze it there. It turns out that the ELF binary is designed to XOR-decode the flag, not using a static key, but using information obtained from its environment. This makes static analysis to determine the flag frustrating.

By tracing through the ELF code in IDA as shown in Figure 14, we find that the main function takes a code path to 0xec2 if the "O" (the human player) wins. At 0xf09 we see a pointer to 0x202010 loaded into rdi; Figure 15 shows that this looks much like an encrypted key. Indeed, further reversing of sub_14b0 would find that it determines the winner and places the winning marker into al; the loop at 0xf20-0x0xf31 XORs each byte in the array at 0x202010 with the ASCII code for the winning marker. This is only the first level of encryption.

```
.TEXT:0000000000000EC2 LOC_EC2:                           ; CODE
XREF: MAIN+2B8↑J
.TEXT:0000000000000EC2                XOR     ECX, ECX    ; FLAGS
.TEXT:0000000000000EC4                MOV     EDX, 0AH    ; N
.TEXT:0000000000000EC9                MOV     RSI, RBP    ; BUF
```

```
.TEXT:0000000000000ECC                CALL    _SEND
.TEXT:0000000000000ED1                PXOR    XMM0, XMM0
.TEXT:0000000000000ED5                MOV     RAX, 0A21736E6977204FH
.TEXT:0000000000000EDF                XOR     EDX, EDX
.TEXT:0000000000000EE1                MOV     QWORD PTR
CS:XMMWORD_202060, RAX
.TEXT:0000000000000EE8                MOV     QWORD PTR
CS:XMMWORD_202060+8, RDX
.TEXT:0000000000000EEF                MOVAPS  CS:XMMWORD_202070,
XMM0
.TEXT:0000000000000EF6                MOVAPS  CS:XMMWORD_202080,
XMM0
.TEXT:0000000000000EFD                MOVAPS  CS:XMMWORD_202090,
XMM0
.TEXT:0000000000000F04                CALL    SUB_14B0
.TEXT:0000000000000F09                LEA     RDI, UNK_202010
.TEXT:0000000000000F10                MOV     [RSP+188H+VAR_160],
RDI
.TEXT:0000000000000F15                MOV     R13, RDI
.TEXT:0000000000000F18                MOV     R15, RDI
.TEXT:0000000000000F1B                NOP     DWORD PTR
[RAX+RAX+00H]
.TEXT:0000000000000F20 LOC_F20:                                 ; CODE
XREF: MAIN+471↓J
.TEXT:0000000000000F20                LEA     RDI, UNK_202020
.TEXT:0000000000000F27                XOR     [R15], AL
.TEXT:0000000000000F2A                ADD     R15, 1
.TEXT:0000000000000F2E                CMP     RDI, R15
.TEXT:0000000000000F31                JNZ     SHORT LOC_F20
```

```
.DATA:0000000000202010 UNK_202010     DB  4AH ; J           ; DATA
XREF: MAIN+449↑O
.DATA:0000000000202010                                      ;
MAIN+7EA↑O
.DATA:0000000000202011                DB  82H
.DATA:0000000000202012                DB  43H ; C
.DATA:0000000000202013                DB 0ABH
.DATA:0000000000202014                DB  95H
.DATA:0000000000202015                DB 0EDH
.DATA:0000000000202016                DB  8FH
.DATA:0000000000202017                DB  7EH ; ~
.DATA:0000000000202018                DB  9CH
.DATA:0000000000202019                DB 0BCH
.DATA:000000000020201A                DB 0ADH
.DATA:000000000020201B                DB  84H
.DATA:000000000020201C                DB  17H
.DATA:000000000020201D                DB  91H
.DATA:000000000020201E                DB    6
.DATA:000000000020201F                DB  15H
```

Figure 15. At 0x202010 is a byte array that looks like it could be an encrypted string.

Further analysis of the decryption code reveals what ties the binary to WSL. These are some interesting differences between WSL 1 and native Linux that allow the code to detect on which version it is running.

- At 0xf33, the code opens the /proc/modules file and looks for a module name starting with cpufreq_, and XORs the flag with the full name of that module. As /proc/modules is not present on WSL 1, this only serves to corrupt the flag when run on native Linux.
- At 0xfa3, the code opens the /proc/mounts file and looks for the type of the root file system. It looks for the 'f' character in the file system type, and XORs characters from the file system type against the flag. For WSL, the root file system type is lxfs or wslfs, so this is always the characters 'f' 's'. On native Linux, the file system type is likely ext4 which contains no 'f' 's' suffix, so the flag would not be decrypted properly.
- At 0x1010, the code opens /proc/version_signature and XORs against the first 9 characters in the file; these are expected to be "Microsoft".
- At 0x1073, the code parses the auxiliary vector (somewhat like a PEB) to find the ELF header for the VDSO shared library. This is a shared library injected into processes by the Linux kernel (not on disk). The code iterates over each program header in the VDSO, XORing the flag with the upper 48 bits of each program header's virtual offset. On WSL 1 these bits are 0xffffffffff70 while on native Linux they are 0.
- At 0x10c3, the code iterates over each non-directory entry in the /proc file system, XOR-ing the flag with the upper 16 bits of each file's i-number. On WSL 1 these i-numbers are low sequential numbers (1, 2, 3, …) with all-zero upper 16 bits, while on native Linux, some of those upper 16 bits are set, corrupting the flag.

As a result of these checks, the embedded ELF binary will fail to decrypt the flag on native Linux (or WSL 2, for that matter), while succeeding on WSL 1. It is a Linux program, but only for Microsoft's Linux, and thus the Aardvark name, harkening back to Microsoft's AARD code that allowed their programs to detect when they were running on cloned DOS versions.