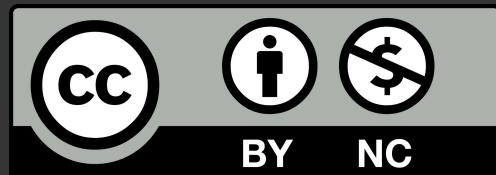

ETHL - Ethical Hacking Lab

0x07 - Binary Exploitation p1

Davide Guerri - davide[.]guerri AT uniroma1[.]it
Ethical Hacking Lab
Sapienza University of Rome - Department of Computer Science



This slide deck is released under Creative Commons
Attribution-NonCommercial (CC BY-NC)



ToC

1

Useful concepts

3

Shared libraries -
GOT and PLT

5

Shellcodes

2

Security measures

4

Calling
conventions



Useful Concepts - CPU Registers



Useful Concepts - CPU Registers

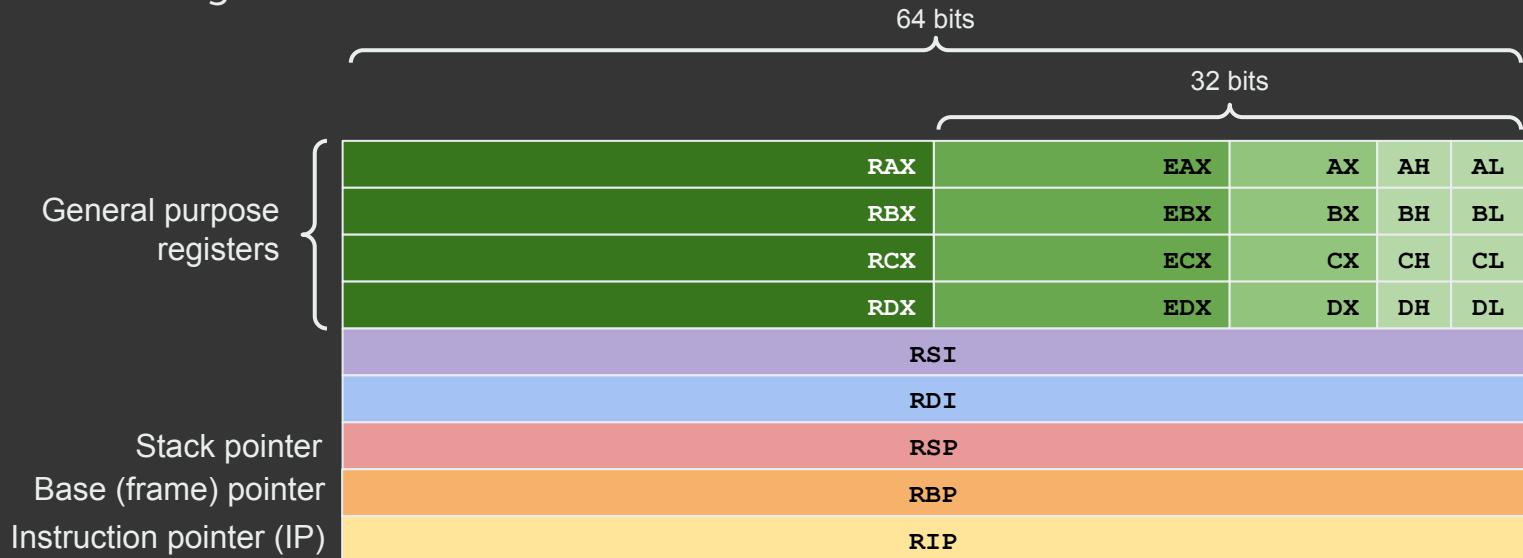
Quickly accessible memory locations, available to a computer's processor

- Usually consist of a small amount of fast storage, although some registers have specific hardware functions, and may be read-only or write-only



Useful Concepts - CPU Registers

Example: x86_64 registers



Useful Concepts - CPU Registers

Special (non-general purpose) registers for x86-64

- rbp - base or frame pointer: start of the function frame (*)
- rsp - stack pointer: current location in stack, growing downwards
- rip - instruction pointer: points to the next instruction the CPU will execute

Some other registers and their conventional use

- rsi - register source index (source for data copies)
- rdi - register destination index (destination for data copies)
- rcx - typically the index in loops

(*) We shall see later what a frame is in this context

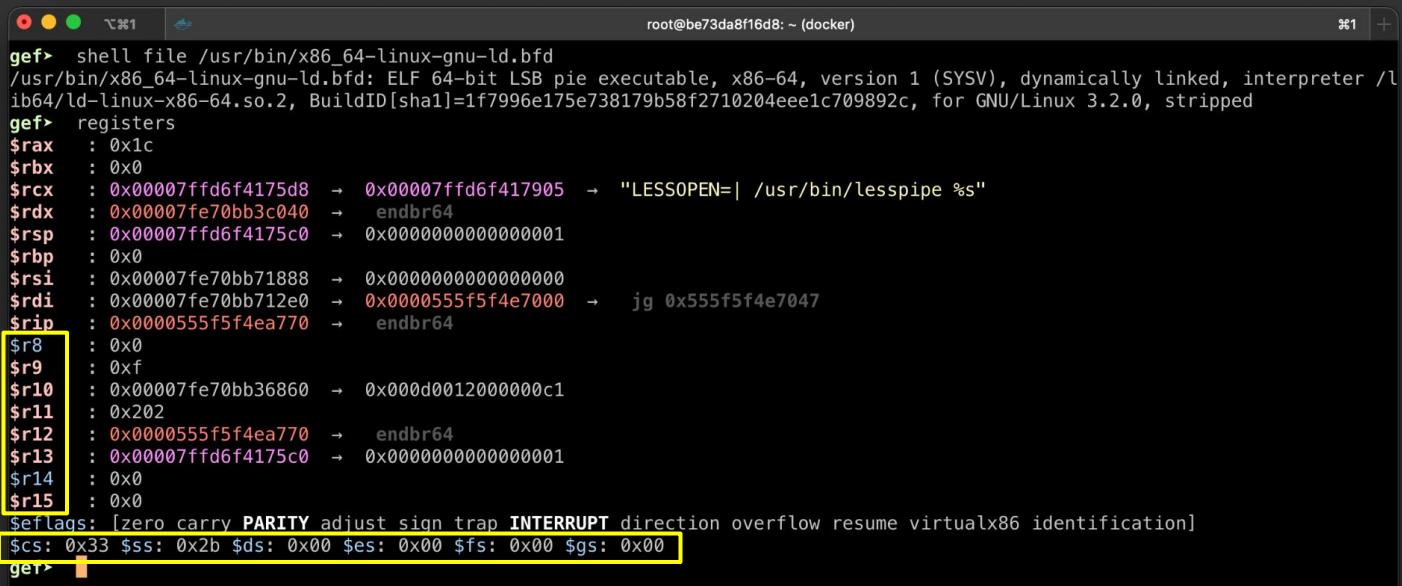


Useful Concepts - CPU

In this picture: Intel(R) Celeron(R) CPU J1900

x86_64 - 64-bit version of the x86 instruction set

Segment
Registers
E.g., **fs** is used
for canary



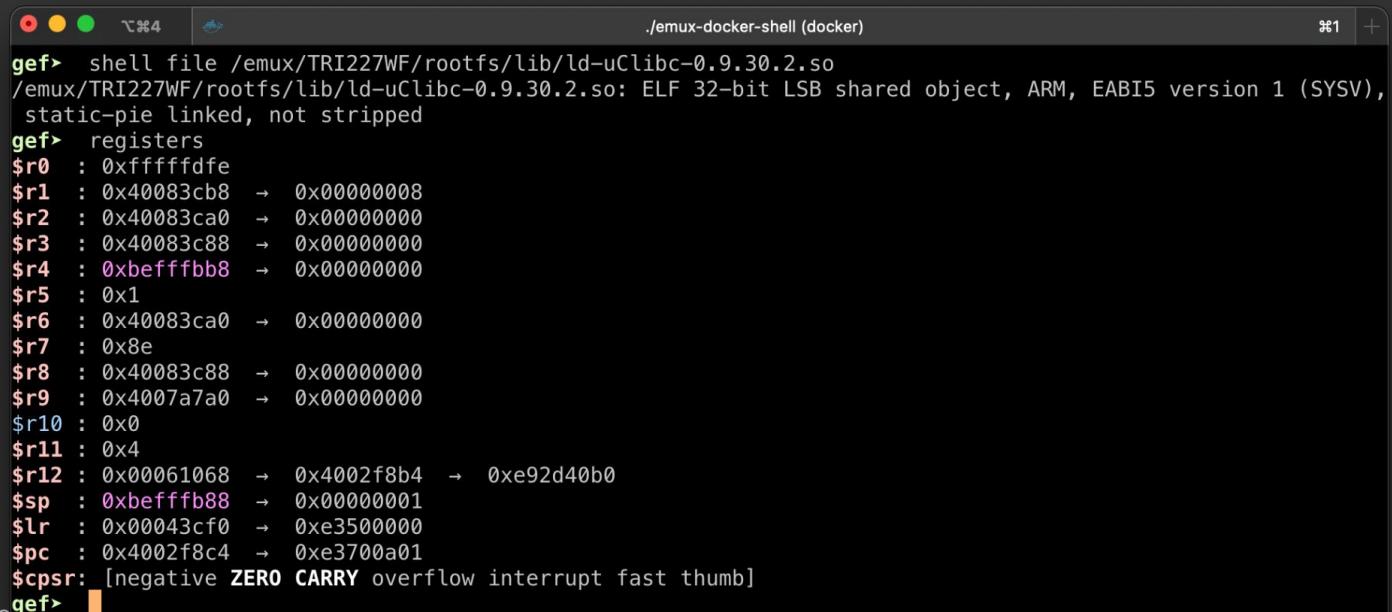
```
gef> shell file /usr/bin/x86_64-linux-gnu-ld.bfd
/usr/bin/x86_64-linux-gnu-ld.bfd: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=1f7996e175e738179b58f2710204eee1c709892c, for GNU/Linux 3.2.0, stripped
gef> registers
$rax : 0x1c
$rbx : 0x0
$rcx : 0x00007ffd6f4175d8 → 0x00007ffd6f417905 → "LESSOPEN=| /usr/bin/lesspipe %s"
$rdx : 0x00007fe70bb3c040 → endbr64
$rsp : 0x00007ffd6f4175c0 → 0x0000000000000001
$rbp : 0x0
$rsi : 0x00007fe70bb71888 → 0x0000000000000000
$rdi : 0x00007fe70bb712e0 → 0x0000555f5f4e7000 → jg 0x555f5f4e7047
$rip : 0x0000555f5f4ea770 → endbr64
$r8 : 0x0
$r9 : 0xf
$r10 : 0x00007fe70bb36860 → 0x000d0012000000c1
$r11 : 0x202
$r12 : 0x0000555f5f4ea770 → endbr64
$r13 : 0x00007ffd6f4175c0 → 0x0000000000000001
$r14 : 0x0
$r15 : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
gef>
```



Useful Concepts - CPU

In this picture: Trivision NC-227-WF IP Camera remote gdb debugging

ARM 32 bits - Typically on embedded / IoT Devices



The screenshot shows the gef debugger interface running on a Docker container. The title bar indicates the session is for the file `/emux/TRI227WF/rootfs/lib/ld-uClibc-0.9.30.2.so`. The command `registers` has been entered, displaying the current values of all ARM registers:

```
gef> registers
$r0 : 0xfffffdfe
$r1 : 0x40083cb8 → 0x00000008
$r2 : 0x40083ca0 → 0x00000000
$r3 : 0x40083c88 → 0x00000000
$r4 : 0xbefffb88 → 0x00000000
$r5 : 0x1
$r6 : 0x40083ca0 → 0x00000000
$r7 : 0x8e
$r8 : 0x40083c88 → 0x00000000
$r9 : 0x4007a7a0 → 0x00000000
$r10 : 0x0
$r11 : 0x4
$r12 : 0x00061068 → 0x4002f8b4 → 0xe92d40b0
$sp : 0xbefffb88 → 0x00000001
$lr : 0x00043cf0 → 0xe3500000
$pc : 0x4002f8c4 → 0xe3700a01
$cpsr: [negative ZERO CARRY overflow interrupt fast thumb]
```

The bottom status bar shows the gef logo.



Useful Concepts

ARM 64 bits

Modern ARM machines

In this picture:
Linux ARM VM on Mac M1

```
gef> shell file /usr/bin/aarch64-linux-gnu-ld.bfd
/usr/bin/aarch64-linux-gnu-ld.bfd: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-aarch64.so.1, BuildID[sha1]=a18e0b307d283d11814d78b3bfe81f4b42736276, for GNU/Linux 3.7.0
, stripped
gef> registers
$x0 : 0x0aaaaaaaaaa1bc0 → stp x29, x30, [sp, #-384]!
$x1 : 0x1
$x2 : 0x00ffffffffffff140 → 0x00ffffffffffff46c → "/usr/bin/cat"
$x3 : 0x0
$x4 : 0x0
$x5 : 0x00fffffc24e0 → <_dl_fini+0> stp x29, x30, [sp, #-80]!
$x6 : 0x0000000000000001
$x7 : 0x4554415649
$x8 : 0xd7
$x9 : 0x20
$x10 : 0x0
$x11 : 0x0
$x12 : 0x2e
$x13 : 0x3d8f538
$x14 : 0x00fffffc24e0 → 0x0000000000003ecc0
$x15 : 0x3
$x16 : 0x00fffffc24e0 → <_libc_start_main+0> stp x29, x30, [sp, #-96]!
$x17 : 0x0aaaaaaaaaa1bc0 → 0x00fffffc24e0 → <_libc_start_main+0> stp x29, x30, [sp, #-96]!
$x18 : 0x19d000
$x19 : 0x0
$x20 : 0x0
$x21 : 0x0aaaaaaaaaa1bc0 → nop
$x22 : 0x0
$x23 : 0x0
$x24 : 0x0
$x25 : 0x0
$x26 : 0x0
$x27 : 0x0
$x28 : 0x0
$x29 : 0x0
$x30 : 0x0aaaaaaaaaa1bc0 → bl 0xaaaaaaaaaa1a40 <abort@plt>
$sp : 0x0000000000000001
$pc : 0x00fffffc24e0 → <_libc_start_main+0> stp x29, x30, [sp, #-96]!
$cpsr: [NEGATIVE zero carry overflow interrupt fast]
$fpsr: 0x0
$fpcr: 0x0
gef>
```



Useful Concepts - Call Stack



Useful Concepts - Call Stack

What is a Call Stack?

A stack-like data structure

- Stores information about the **active subroutines** of a computer program



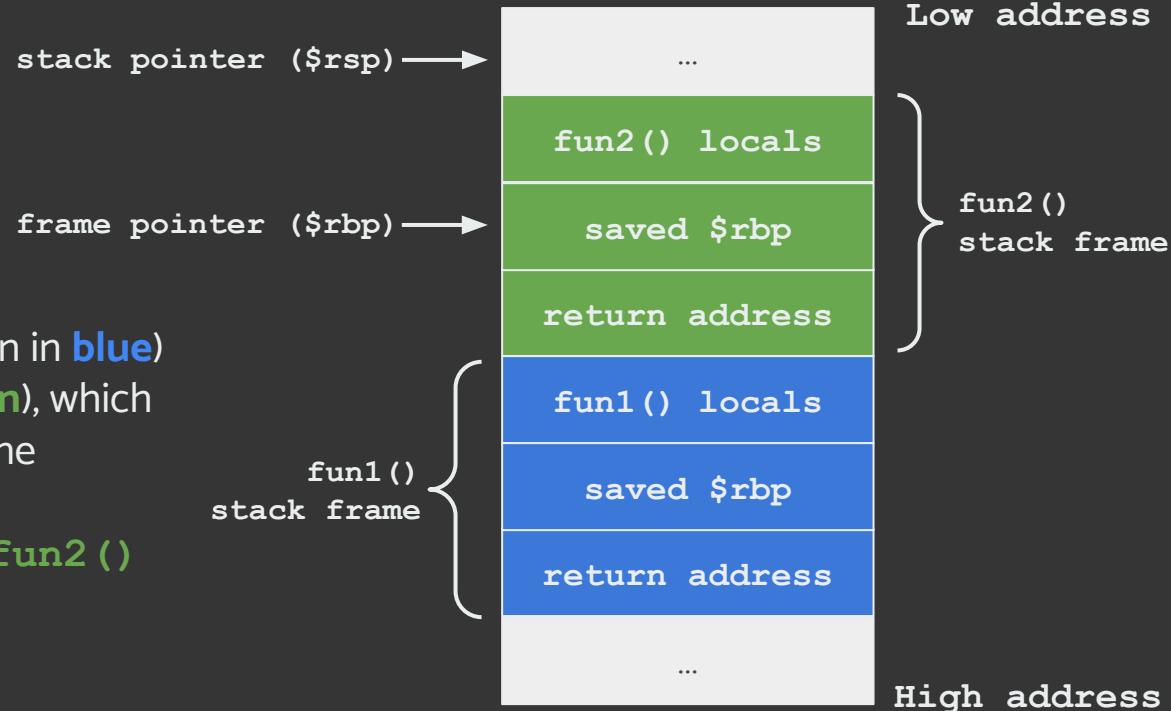
Useful Concepts

- Call Stack

In the picture:

Call Stack after **fun1()** (shown in **blue**) called **fun2()** (shown in **green**), which is the currently executing routine

In the picture, **fun1()** called **fun2()**



Useful Concepts - Function prologue and epilogue



Useful Concepts - Function prologue and epilogue

Function prologue - a few lines of code at the beginning of a function

- Prepares the stack and registers for use within the function

Function epilogue - appears at the end of the function

- Restores the stack and registers to the state they were in before the function was called



Useful Concepts - Function prologue and epilogue

```
1 #include <stdio.h>
2
3 int fun1(int p1) {
4     const int p2 = 2;
5     return p1 + p2;
6 }
7
8 int main() {
9     int res = fun1(4);
10    printf("Hello World, %d\n", res);
11    return 0;
12 }
```

int fun1(int) disassembly

prologue {

<+0>:	endbr64
<+4>:	push rbp
<+5>:	mov rbp, rsp
<+8>:	mov DWORD PTR [rbp-0x14], edi
<+11>:	mov DWORD PTR [rbp-0x4], 0x2
<+18>:	mov edx, DWORD PTR [rbp-0x14]
<+21>:	mov eax, DWORD PTR [rbp-0x4]
<+24>:	add eax, edx
<+26>:	pop rbp
<+27>:	ret

epilogue {



Security Measures - Stack Canary



Security Measures - Stack Canary

A stack canary is a **random value**

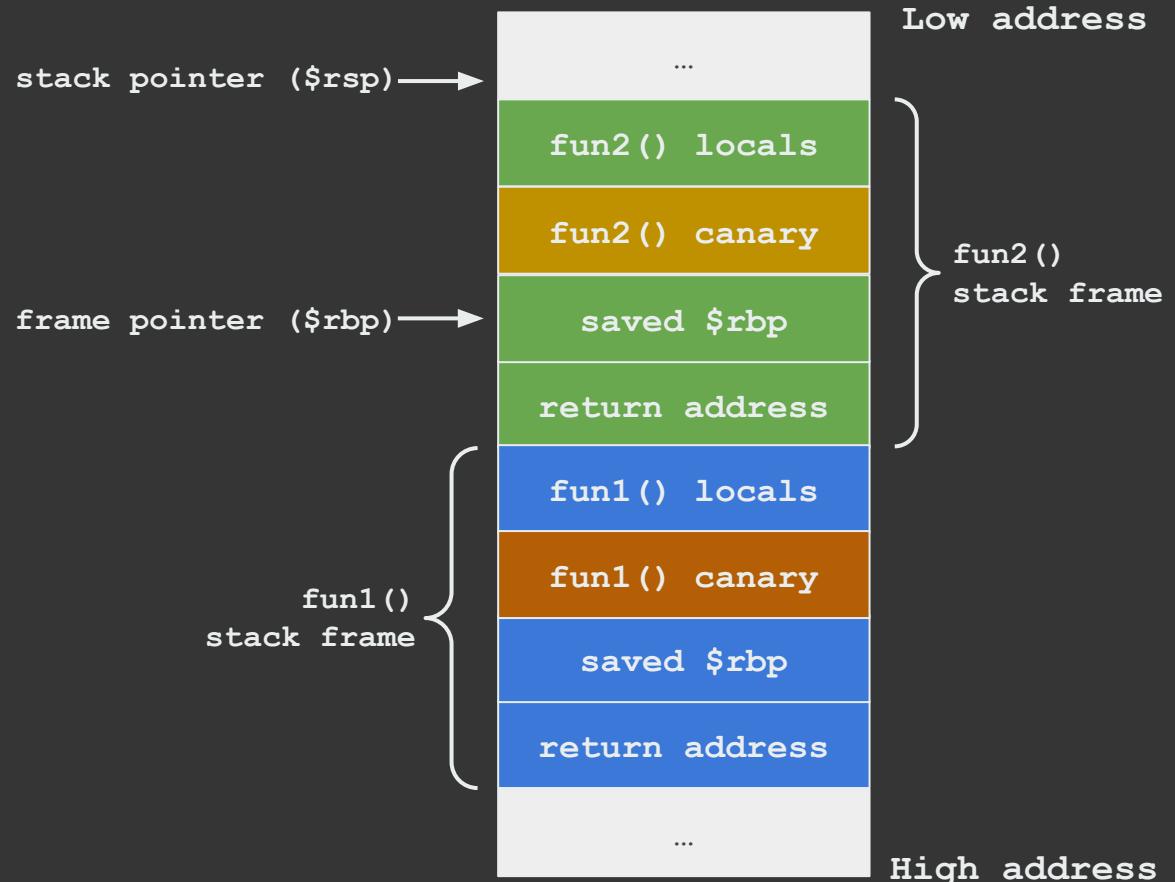
- **Put on the stack** between the local variables of a function and the saved `$rbp` register (i.e., the frame pointer) + the function return address
- The value is **checked when the function returns**
- If different from the original, the **program is terminated**

Canary logic is produced by the **compiler**



Security Measures - Stack Canary

How does the call stack
look like w/ canary?



Useful Concepts - Function prologue and epilogue

```
1 #include <stdio.h>
2
3 int fun1(int p1) {
4     char buff[16];
5     const int p2 = 2;
6     return p1 + p1;
7 }
8
9 int main() {
10    int res = fun1(4);
11    printf("Hello World, %d\n", res);
12    return 0;
13 }
14 }
```

int fun1(int) disassembly

prologue {

<+0>:	endbr64
<+4>:	push rbp
<+5>:	mov rbp, rsp
<+8>:	sub rsp, 0x40
<+12>:	mov DWORD PTR [rbp-0x34], edi
<+15>:	mov rax, QWORD PTR fs:0x28
<+24>:	mov QWORD PTR [rbp-0x8], rax
<+28>:	xor eax, eax
<+30>:	mov DWORD PTR [rbp-0x24], 0x2
<+37>:	mov eax, DWORD PTR [rbp-0x34]
<+40>:	add eax, eax
<+42>:	mov rdx, QWORD PTR [rbp-0x8]
<+46>:	sub rdx, QWORD PTR fs:0x28
<+55>:	je 0x11a7 <fun1+62>
<+57>:	call 0x1060 <__stack_chk_fail@plt>
<+62>:	leave
<+63>:	ret

epilogue {



Security Measures - Stack Canary

In the picture:

Linux, gcc Stack Canary

Note: on Linux, canary is always NULL terminated,
why?

The screenshot shows a debugger interface with assembly code and memory dump sections. Annotations highlight specific memory locations:

- A yellow arrow points from the text "local variable" to the memory location `0x000005605e26009f`, which is the value of register `$rip`.
- A yellow arrow points from the text "canary" to the memory location `0x000005605e26009b`, which is the value of register `$rbp`. This value is also annotated with "`<main+33> mov eax, 0x0`".
- A yellow arrow points from the text "return address" to the memory location `0x000005605e260089f`, which is the value of register `$rip` and annotated with "`<main+47> jg 0x5605e2600047`".

```
[ Legend: Modified register | Code | Heap | Stack | String ]  
registers —  
$rax : 0x9e616f98ad572400  
$rbx : 0x0  
$rcx : 0x000005605e2600a90 → <__libc_csu_init+0> push r15  
$rdx : 0x00007fff28729d58 → 0x00007fff2872be22 → "HOME=/root"  
$rsp : 0x00007fff28729bd0 → 0x0000000000000001  
$rbp : 0x00007fff28729be0 → 0x00007fff28729c30 → 0x0000000000000001  
$rsi : 0x00007fff28729d48 → 0x00007fff2872bdfe → "/root/binaries/pwn101/pwn107.pwn107"  
$rdi : 0x1  
$rip : 0x000005605e260089f → <setup+21> xor eax, eax  
$r8 : 0x00007f2888321f10 → 0x0000000000000004  
$r9 : 0x00007f288833b040 → endbr64  
$r10 : 0x00007f2888335908 → 0x000000120000000e  
$r11 : 0x00007f2888350660 → <_dl_audit_preinit+0> endbr64  
$r12 : 0x00007fff28729d48 → 0x00007fff2872bdfe → "/root/binaries/pwn101/pwn107.pwn107"  
$r13 : 0x000005605e2600992 → <main+0> push rbp  
$r14 : 0x0  
$r15 : 0x00007f288836f040 → 0x00007f28883702e0 → 0x000005605e2600000 → jg 0x5605e2600047  
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]  
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00  
stack —  
0x00007fff28729bd0 +0x0000: 0x0000000000000001 ← $rsp  
0x00007fff28729bd8 +0x0008: 0x9e616f98ad572400 ← $rbx ← canary  
0x00007fff28729be0 +0x0010: 0x00007fff28729c30 → 0x0000000000000001 ← $rbp ← canary  
0x00007fff28729be8 +0x0018: 0x000005605e26009b3 → <main+33> mov eax, 0x0  
0x00007fff28729bf0 +0x0020: 0x0000000000000000  
0x00007fff28729bf8 +0x0028: 0x0000000000000000  
0x00007fff28729c00 +0x0030: 0x0000000000000000  
0x00007fff28729c08 +0x0038: 0x0000000000000000  
code:x86:64 —  
0x5605e260088e <setup+4> sub    rsp, 0x10  
0x5605e2600892 <setup+8>  mov    rax, QWORD PTR fs:0x28  
0x5605e260089b <setup+17> mov    QWORD PTR [rbp-0x8], rax  
→ 0x5605e260089f <setup+21> xor    eax, eax  
0x5605e26008a1 <setup+23> mov    rax, QWORD PTR [rip+0x201778]      # 0x5605e2802020 <stdout@@GLIBC_2.2.5>  
0x5605e26008a8 <setup+30>  mov    ecx, 0x0  
0x5605e26008ad <setup+35>  mov    edx, 0x2  
0x5605e26008b2 <setup+40>  mov    esi, 0x0  
0x5605e26008b7 <setup+45>  mov    rdi, rax  
threads —  
[#0] Id 1, Name: "pwn107.pwn107", stopped 0x5605e260089f in setup (), reason: SINGLE STEP  
trace —  
[#0] 0x5605e260089f → setup()  
[#1] 0x5605e26009b3 → main()
```



Security measures - ASLR



Security measures - ASLR

Address Space Layout Randomization (ASLR)

Developed to **prevent exploitation of memory corruption vulnerabilities**

- Example: Prevent an attacker from reliably jumping to a particular function in memory

ASLR randomly arranges the address space positions of key data areas of a process, including

- the base of the executable
- the positions of the stack
- heap and libraries



Security measures - ASLR

It is implemented by the **operating system**

- can be expensive!
- could be turned off on some device

Codes meaning

- No randomisation
- Shared libraries, stack, `mmap()`, VDSO and heap are randomised
- Full randomisation: 1. + memory managed through `brk()` (e.g., program heap)

```
# cat /proc/cpuinfo
Processor      : ARM926EJ-S rev 5 (v5l)
BogoMIPS       : 339.96
Features       : swp half fastmult vfp edsp java
CPU implementer: 0x41
CPU architecture: 5TEJ
CPU variant    : 0x0
CPU part       : 0x926
CPU revision   : 5

Hardware       : ARM-Versatile PB
Revision       : 0000
Serial         : 0000000000000000
# cat /proc/sys/kernel/randomize_va_space
0
```

```
root@4b1c189a7265: ~ % uname -m
x86_64
root@4b1c189a7265: ~ % cat /proc/sys/kernel/randomize_va_space
2

root@4b1c189a7265: ~ %
```



Security measures - PIC/PIE



Security measures - PIC/PIE

PIC (Position Independent Code) / PIE (PI Executable)

Code that does not depend on being loaded in a particular memory address

PIC is used for **shared libraries**

- Shared code that can be “loaded” at any location within the linking program’s virtual address space

PIC is also used for **executable binaries** (PIE)

- Implemented for hardening purposes
- Default on modern Linux distros

```
d11ac41e9108 — davide@lechuck
> checksec minimal
[*] '/root/sources/basics/minimal'
Arch:      amd64-64-little
RELRO:    Full RELRO
Stack:    No canary found
NX:       NX enabled
PIE:     PIE enabled
> checksec minimal-lazy-nopie
[*] '/root/sources/basics/minimal-lazy-nopie'
Arch:      amd64-64-little
RELRO:    No RELRO
Stack:    No canary found
NX:       NX enabled
PIE:     No PIE (0x400000)

```



Shared libraries - GOT and PLT



Shared libraries - GOT and PLT

Shared libraries - Body of PIC, shared by multiple binaries (Libc is an example)

Executables don't know the location of functions they need in shared libs

The dynamic linker (that is `ld.so`, for Linux) is responsible for resolving those addresses:

- On each function call - aka **lazy binding**
- On program load
- On program load w/ RELRO (RELocation Read-Only) - **default for modern distros**



Shared libraries - GOT and PLT

Dynamic linking is implemented through

- ***Procedure Linkage Table (PLT)***
- ***Global Offset Table (GOT)***

These tables work together



Shared libraries - GOT and PLT

PLT

- Contains a set of stubs or trampolines
 - Responsible for redirecting control flow to the dynamic linker during the *first invocation* of a function in the shared library (assuming lazy binding)
 - Subsequent calls will go to the real function in the shared library
- Each entry in the PLT corresponds to a function in a shared library used by the executable (determined at compile time)



Shared libraries - GOT and PLT

GOT (Global Offset Table)

The GOT is a table that contains addresses of global data and functions

- Initially, the entries in the GOT point to the corresponding PLT stubs



Shared libraries - GOT and PLT

GOT (Global Offset Table)

The ***first time*** the dynamic linker resolves the addresses of library functions, it updates the GOT entries with the resolved addresses (assuming lazy binding)

From that moment, subsequent call to the same library functions go to the library



Super-minimal program to show dynamic linking, GOT and PLT

We force lazy binding for a simple binary (otherwise dynamic linking will happen at load time)

In this example I used GDB with GEF (Gnu DeBugger - GDB Enhanced Features)

```
ॐ 4d1fa104c464 — davide@lechuck
> ccat minimal.c
#include <stdio.h>

void main() {
    puts("Hello world!\n");
}

> gcc minimal.c -o minimal -z lazy -z norelro
> gdb minimal
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef` to start, `gef config` to configure
88 commands loaded and 5 functions added for GDB 12.1 in 0.01ms using Python engine 3.10
Reading symbols from minimal...
(No debugging symbols found in minimal)
gef> start
```



Before calling puts () the first time

GOT contains the
address of a stub
function of the dynamic
linker

```
gef> disassemble main
Dump of assembler code for function main:
 0x0000561c2703a149 <+0>:    endbr64
 0x0000561c2703a14d <+4>:    push   rbp
 0x0000561c2703a14e <+5>:    mov    rbp, rsp
=> 0x0000561c2703a151 <+8>:    lea    rax,[rip+0xeac]      # 0x561c2703b004
 0x0000561c2703a158 <+15>:   mov    rax,[rax]
 0x0000561c2703a15b <+18>:   call   0x561c2703a050 <puts@plt>
 0x0000561c2703a160 <+23>:   nop
 0x0000561c2703a161 <+24>:   pop    rbp
 0x0000561c2703a162 <+25>:   ret
End of assembler dump.
gef> b *0x0000561c2703a160
Breakpoint 1 at 0x561c2703a160
gef> disassemble 0x561c2703a050
Dump of assembler code for function puts@plt:
 0x0000561c2703a050 <+0>:    endbr64
 0x0000561c2703a054 <+4>:    bnd   jmp  QWORD PTR [rip+0x22cd]
 0x0000561c2703a05b <+11>:   nop    DWORD PTR [rax+rax*1+0x0]      # 0x561c2703c328 <puts@got.plt>
End of assembler dump.
gef> x/xg 0x561c2703c328
0x561c2703c328 <puts@got.plt>: 0x0000561c2703a030
gef> disassemble 0x0000561c2703a030, 0x0000561c2703a040
Dump of assembler code from 0x561c2703a030 to 0x561c2703a040:
 0x0000561c2703a030:    endbr64
 0x0000561c2703a034:    push   0x0
 0x0000561c2703a039:    bnd   jmp  0x561c2703a020
 0x0000561c2703a03f:    nop
End of assembler dump.
gef> continue
```

dynamic linker's code



After calling puts()

The GOT contains the address of the real puts() in Libc

```
gef> disassemble main
Dump of assembler code for function main:
0x0000561c2703a149 <+0>:    endbr64
0x0000561c2703a14d <+4>:    push   rbp
0x0000561c2703a14e <+5>:    mov    rbp,rs
0x0000561c2703a151 <+8>:    lea    rax,[rip+0xeac]      # 0x561c2703b004
0x0000561c2703a158 <+15>:   mov    rdi,rdx
0x0000561c2703a15b <+18>:   call   0x561c2703a050 <puts@plt>
=> 0x0000561c2703a160 <+23>:   nop
0x0000561c2703a161 <+24>:   pop    rbp
0x0000561c2703a162 <+25>:   ret
End of assembler dump.
gef> disassemble 0x561c2703a050
Dump of assembler code for function puts@plt:
0x0000561c2703a050 <+0>:    endbr64
0x0000561c2703a054 <+4>:    bnd   jmp QWORD PTR [rip+0x22cd]
0x0000561c2703a05b <+11>:   nop    DWORD PTR [rax+rax*1+0x0]      # 0x561c2703c328 <puts@got.plt>
End of assembler dump.
gef> x/xg 0x561c2703c328
0x561c2703c328 <puts@got=plt>: 0x00007efd5e2e7e50
gef> disassemble 0x00007efd5e2e7e50-0x00007efd5e2e7e60:
Dump of assembler code from 0x00007efd5e2e7e50 to 0x00007efd5e2e7e60:
0x00007efd5e2e7e50 <puts+0>: endbr64
0x00007efd5e2e7e54 <puts+4>: push   r14
0x00007efd5e2e7e56 <puts+6>: push   r13
0x00007efd5e2e7e58 <puts+8>: push   r12           glibc's code
0x00007efd5e2e7e5a <puts+10>:  mov    r12,rdi
0x00007efd5e2e7e5d <puts+13>:  push   rbp
0x00007efd5e2e7e5e <puts+14>:  push   rbx
0x00007efd5e2e7e5f <puts+15>:  sub    rsp,0x10
End of assembler dump.
```



Another way to see it

Dump the GOT (requires GEF)

```
gef> disassemble main
Dump of assembler code for function main:
0x000055ed3068d149 <+0>:    endbr64
0x000055ed3068d14d <+4>:    push   rbp
0x000055ed3068d14e <+5>:    mov    rbp,rs
=> 0x000055ed3068d151 <+8>:    lea    rax,[rip+0xeac]      # 0x55ed3068e004
0x000055ed3068d158 <+15>:   mov    rdi,rax
0x000055ed3068d15b <+18>:   call   0x55ed3068d050 <puts@plt>
0x000055ed3068d160 <+23>:   nop
0x000055ed3068d161 <+24>:   pop    rbp
0x000055ed3068d162 <+25>:   ret
End of assembler dump.
gef> b *0x000055ed3068d160
Breakpoint 1 at 0x55ed3068d160
gef> got
GOT protection: No RelRO | GOT functions: 1
[0x55ed3068f328] puts@GLIBC_2.2.5 → 0x55ed3068d030
gef> continue
Continuing.
Hello world!

Breakpoint 1, 0x000055ed3068d160 in main ()
gef> got
GOT protection: No RelRO | GOT functions: 1
[0x55ed3068f328] puts@GLIBC_2.2.5 → 0x7f7db0e38e50
gef>
```

[0]

"4d1fa104c464" 10:38 13-Jan-24



Calling conventions



Calling conventions

Describe the interface of called code (e.g., functions)

It's part of the ABI (Application Binary Interface)

- The **order** in which atomic (scalar) parameters, or individual parts of a complex parameter, are allocated
- **How parameters are passed** - pushed on the stack, placed in registers, or a mix of both
- Which registers the called function **must preserve** for the caller
- How the task of preparing the stack for, and restoring after, a function call is divided between the caller and the callee



Calling conventions

Order of parameters pushed onto the stack - RTL = Right To Left

Depend on the **hardware architecture, operating system and compiler**

Arch	Name	Operating system, compiler	Parameters	Stack order
			Registers	
x86-64	Microsoft x64 calling convention	Windows (Microsoft Visual C++, GCC, Intel C++ Compiler, Delphi), UEFI	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3	RTL (C)
	System V AMD64 ABI	Solaris, Linux, BSD, macOS, OpenVMS(GCC, Intel C++ Compiler, Clang, Delphi)	RDI, RSI, RDX, RCX, R8, R9, [XYZ]MM0–7	RTL (C)

Calling conventions

x86-64 Linux gcc

rdi, rsi, rdx, rcx, ...

In this example:

call to libc function puts (), putting
the pointer to a string in \$rdi.

The screenshot shows the GEF debugger interface with the following details:

- Registers:** Shows CPU registers (\$rip, \$r8, \$r9, \$r10, \$r11, \$r12, \$r13, \$r14, \$r15, \$cs, \$ss, \$ds, \$es, \$fs, \$gs) and their values.
- Stack:** Shows the stack memory starting at address 0x00007ffed5836060, containing various memory addresses.
- Code:** Shows the assembly code for the current function, labeled "code:x86_64". A specific instruction, `call 0x558b50185050 <puts@plt>`, is highlighted with a yellow box and a yellow arrow pointing to the corresponding C source code line.
- Source:** Shows the C source code for "minimal.c" with the line `puts("Hello world!\n");` highlighted.
- Threads:** Shows the current thread information.
- Trace:** Shows the trace of the current instruction.
- gef>:** The GEF command prompt.



Calling conventions - couple of useful notes

Calling convention for syscalls follows different rules

- Syscalls are wrapped in libc functions, these do follows SysV ABI (as we saw)
- We shall syscalls in a bit

Something useful to know for binary exploitation on amd64 SysV ABI (Linux):

- **Upon function call, the stack pointer must be aligned to 16 bytes**
- Some instructions (like MOVAPS) will cause SIGSEGV if the stack is not aligned



Shellcodes



Shellcodes

Small piece of code used as the payload in the exploitation of a software vulnerability

Written in machine code for the target system (hardware and OS)

It typically spawns a shell - but, being arbitrary code, it can for instance:

- Create a bind shell with TCP
- Create a reverse shell via TCP or UDP
- Open the Windows Calculator app



Shellcodes

To understand shellcodes, we need to understand syscalls

- Syscalls are the Kernel APIs - programmatic way for a program to request a service from the kernel, which controls the core functionalities of the system
- Bridge between a program and the operating system kernel
- Each syscall has a number



Shellcodes

Calling conventions for syscalls are different from “normal” functions

Some of them:

- The kernel interface uses `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9` - **syscall number** in `rax`
- Invoked via the `syscall` instruction
- Returning from the syscall, register `rax` contains the result of the system-call
 - A value in the range between -4095 and -1 indicates an error, it is `-errno`.
- Only values of class INTEGER or class MEMORY are passed to the kernel



Shellcodes

See:

- [Linux System Call Table for x86_64](#)
- [Searchable Linux Syscall Table](#)

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
57	sys_fork						
58	sys_vfork						
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			
60	sys_exit	int error_code					



Shellcodes

A minimal shellcode would just call `execve()` syscall to execute `/bin/sh`
E.g., used for Privileges Escalation

Disassembly

```
0x0000000000000000: 48 C7 C0 3B 00 00 00 00      mov    rax, 0x3b
0x0000000000000007: 48 BB 2F 62 69 6E 2F 73 68 00  movabs rbx, 0x68732f6e69622f
0x0000000000000011: 53                           push   rbx
0x0000000000000012: 54                           push   rsp
0x0000000000000013: 5F                           pop    rdi
0x0000000000000014: 48 31 F6                   xor    rsi, rsi
0x0000000000000017: 48 31 D2                   xor    rdx, rdx
0x000000000000001a: 0F 05                     syscall
```

/bin/sh



Shellcodes

```
mov rax, 0x3b  
movabs rbx, 0x68732f6e69622f  
push rbx ←  
push rsp  
pop rdi  
xor rsi, rsi ←  
xor rdx, rdx  
syscall
```

We need a **pointer** to the NULL terminated string “/bin/sh”

This is put in `rdi` by using the stack:

- Push the value itself (`rbx` content, `0x68732f6e69622f`)
- Push the address of the value (`rsp` points to `0x68732f6e69622f`)

No `argv` ⇒ `rsi` = 0 (NULL)

No `env` ⇒ `rdx` = 0 (NULL)

Escaped Hex Minimal Shellcode

```
\x48\xc7\xc0\x3b\x00\x00\x00\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\x54\x5f\x48\x31\xf6\x48\x31\xd2\x0f\x05
```



Links

- [Searchable Syscalls](#) (x86-64)
- [Decompiler explorer](#)
- [Online Assembler and Disassembler](#)
- GDB - [GEEF](#)
- GDB - [GEP](#) (GDB Enhanced Prompt)
- [Online GDB](#)
- [What does the endbr64 instruction actually do?](#)

