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

Created: 2024-03-05
Last modified: 2025-05-15

ToC

Useful Concepts - CPU Registers

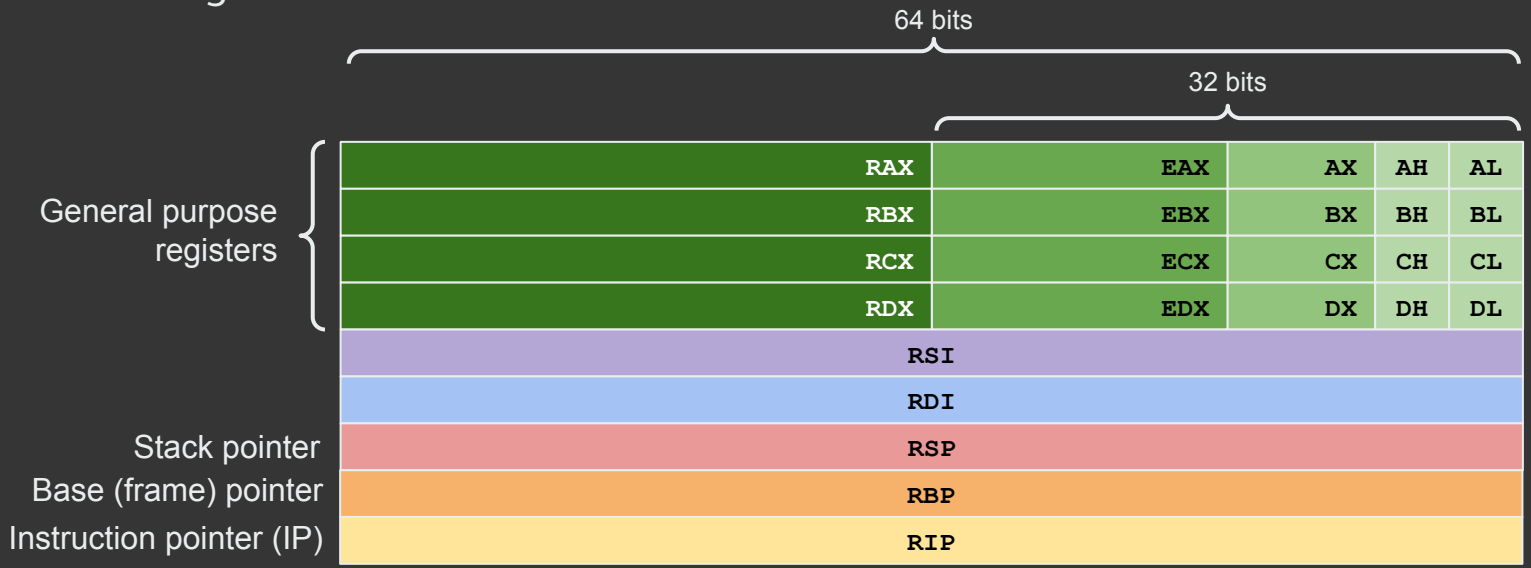
Useful Concepts - CPU Registers

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

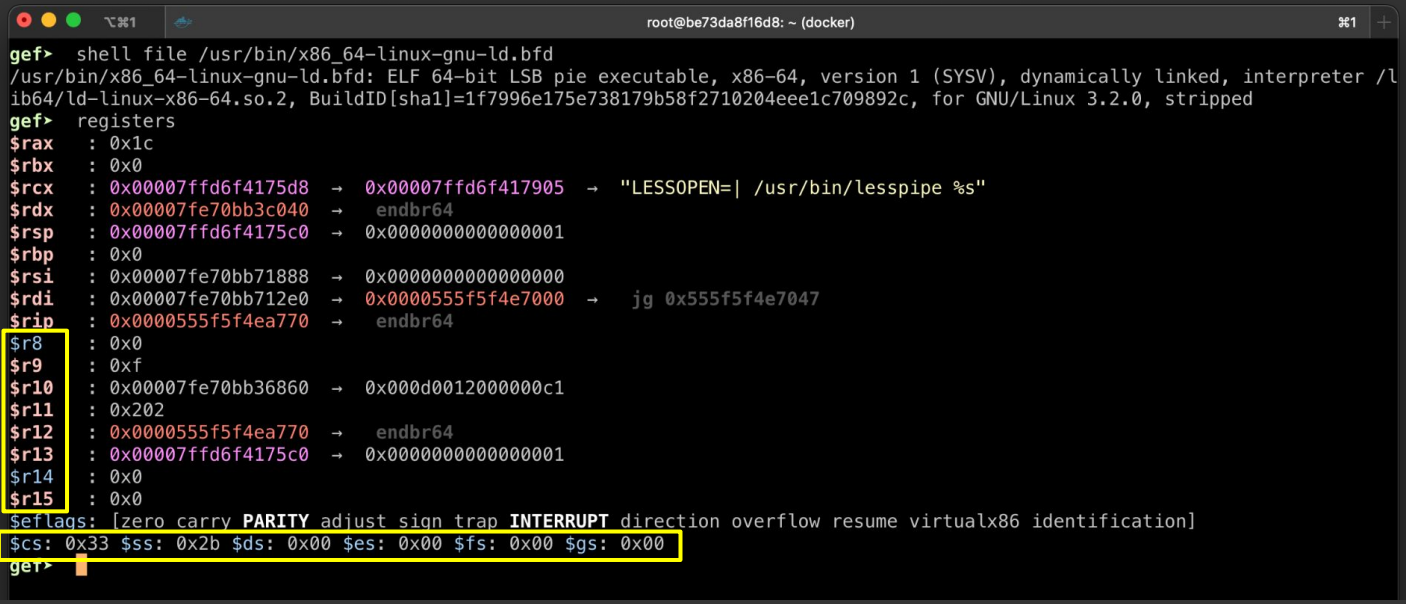
(*) We shall see later what a frame is in this context

Useful Concepts - CPU

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

x86_64 - 64-bit version of the x86 instruction set

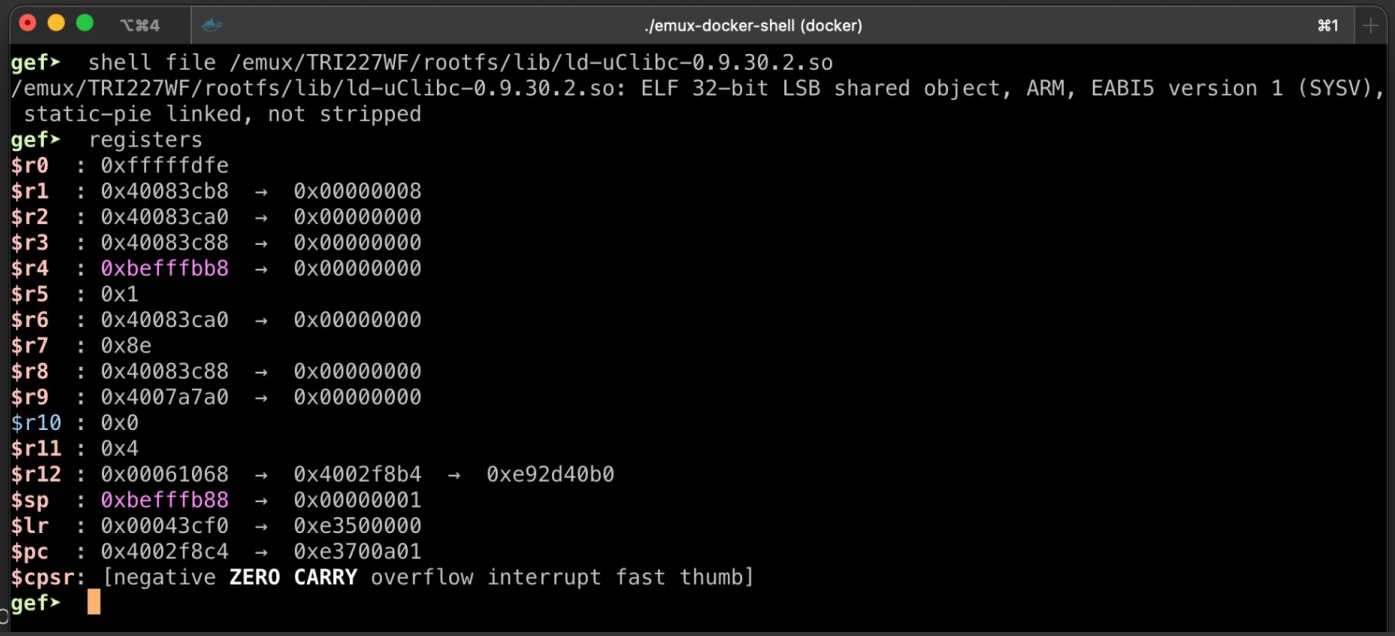
Segment Registers
E.g., `fs` is used for canary



Useful Concepts - CPU

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

ARM 32 bits - Typically on embedded / IoT Devices



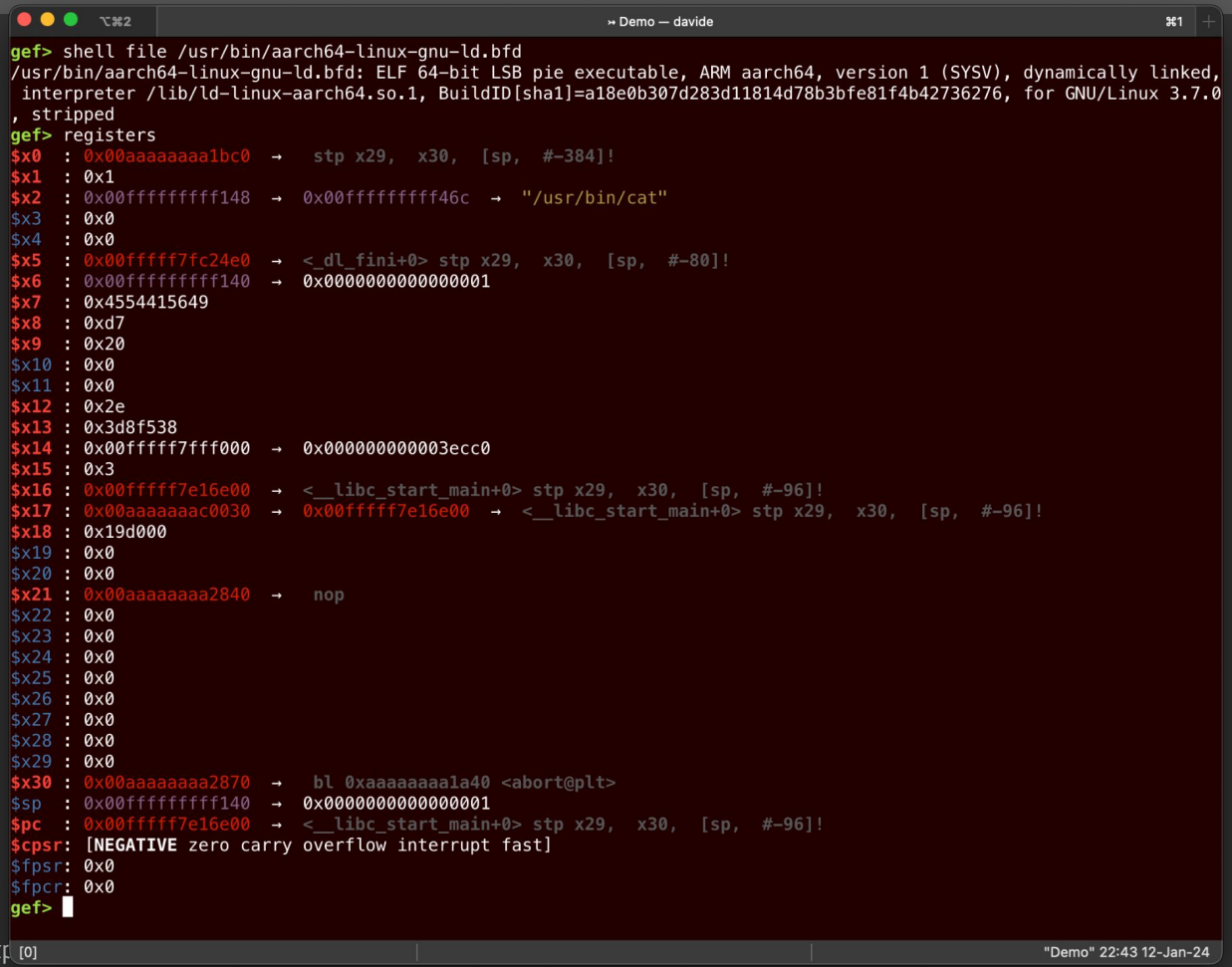
Useful Concepts

ARM 64 bits

Modern ARM machines

In this picture:

Linux ARM VM on Mac M1



Useful Concepts - Call Stack

Useful Concepts - Call Stack

What is a Call Stack?

A stack-like data structure

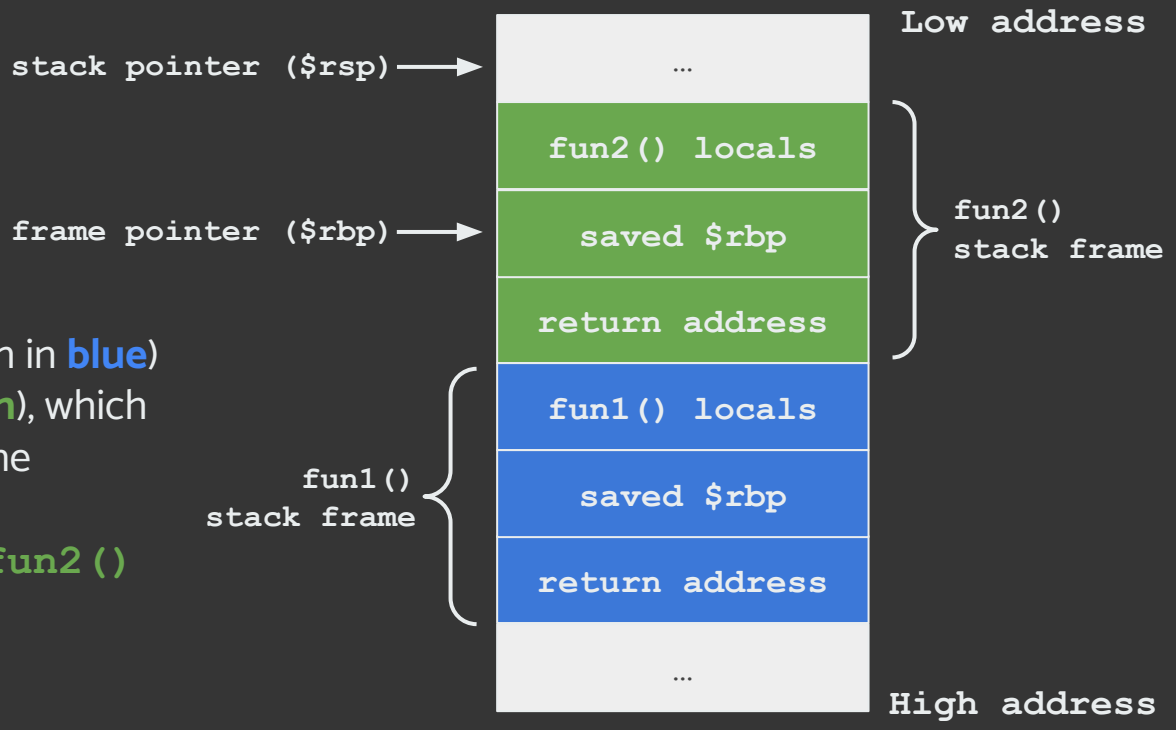
- Stores information about the **active subroutines** of a computer program

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

```c
#include <stdio.h>

int fun1(int p1) {
    const int p2 = 2;
    return p1 + p2;
}

int main() {
    int res = fun1(4);
    printf("Hello World, %d\n", res);
    return 0;
}
```

`int fun1(int)` disassembly

prologue

epilogue

```
<+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
```

Security Measures - Stack Canary

Security Measures - Stack Canary

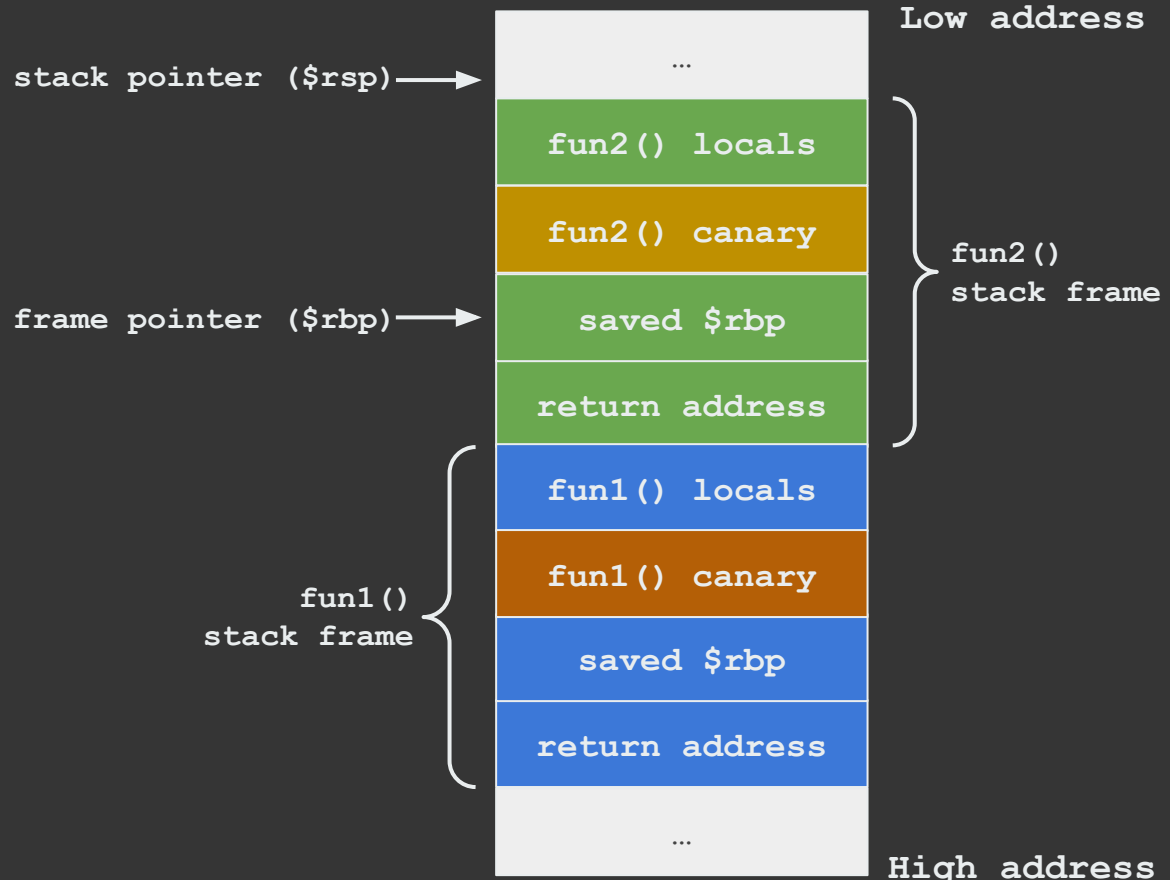
A stack canary is a **random value**

- **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

```c
#include <stdio.h>

int fun1(int p1) {
  char buff[16];
  const int p2 = 2;
  return p1 + p1;
}

int main() {
  int res = fun1(4);
  printf("Hello World, %d\n", res);
  return 0;
}
```

`int fun1(int)` disassembly

prologue

epilogue

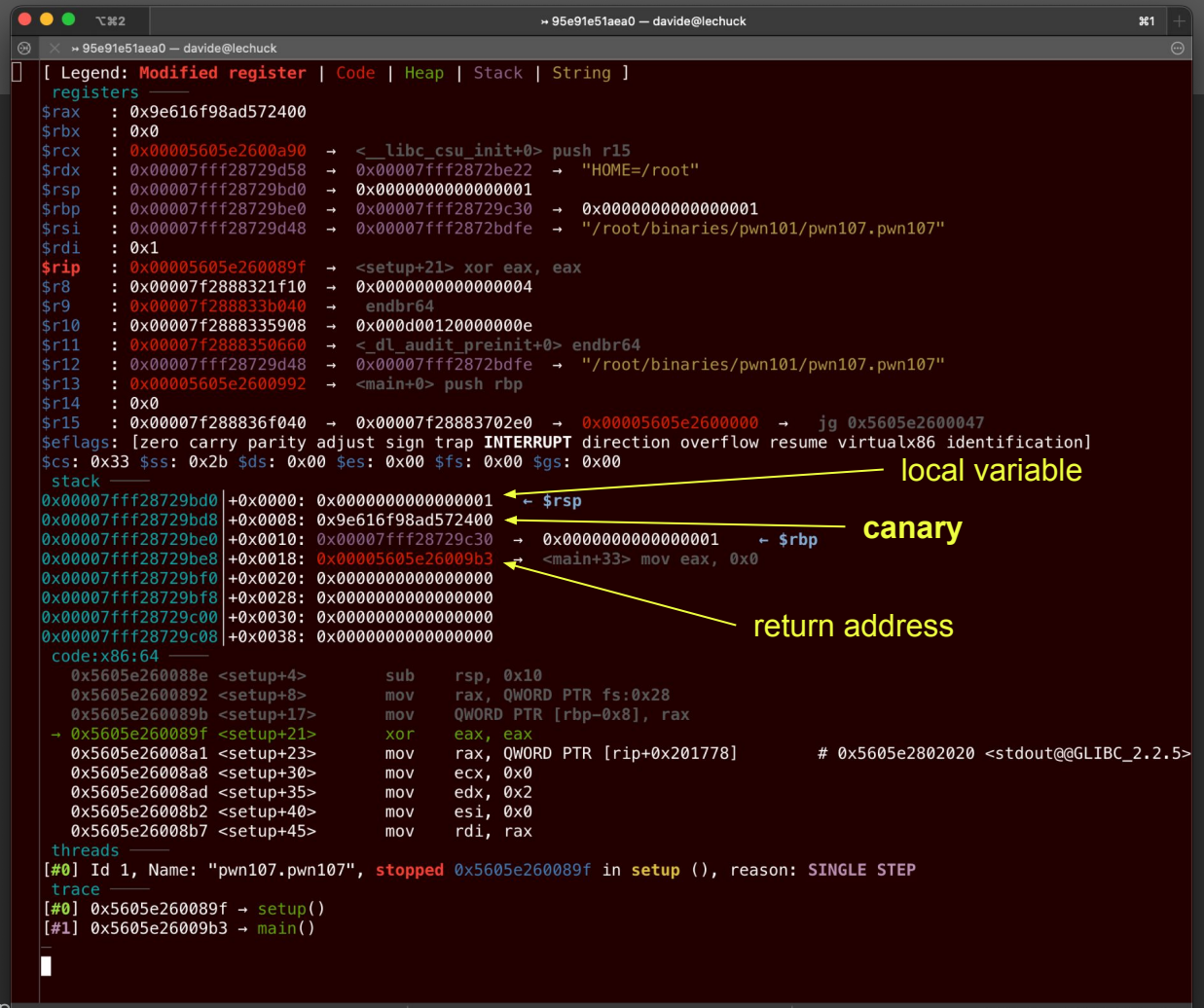
```
<+0>:     endbr64
<+4>:     push   rbp
<+5>:     mov    rbp,rsp
<+8>:     sub    rsp,0x40
<+12>:    mov    DWORD PTR [rbp-0x34],edi
<+15>:    mov    rax,QWORD PTR fs:0x28
<+24>:    mov    QWORD PTR [rbp-0x8],rax
<+28>:    xor    eax,eax
<+30>:    mov    DWORD PTR [rbp-0x24],0x2
<+37>:    mov    eax,DWORD 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
```

Security Measures - Stack Canary

In the picture:

Linux, gcc Stack Canary

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



Security measures - ASLR

Security measures - ASLR

Address Space Layout Randomization (ASLR)

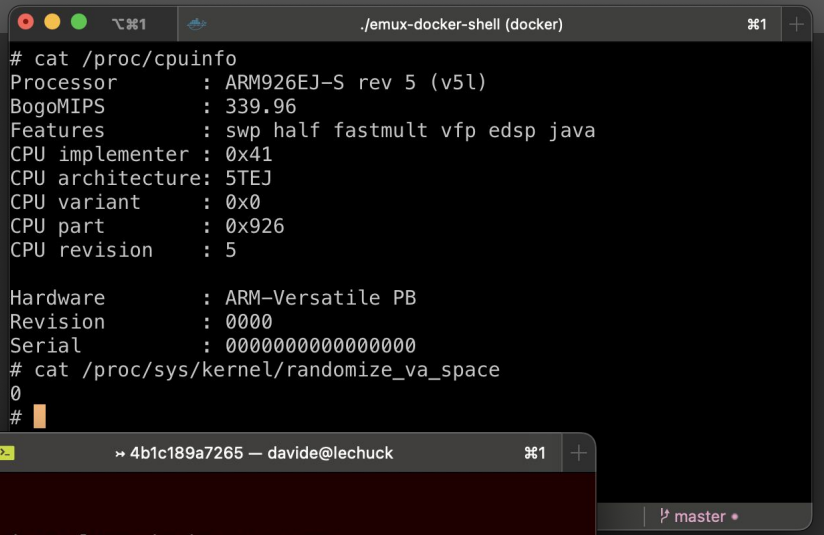
Developed to **prevent exploitation of memory corruption vulnerabilities**

- 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

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

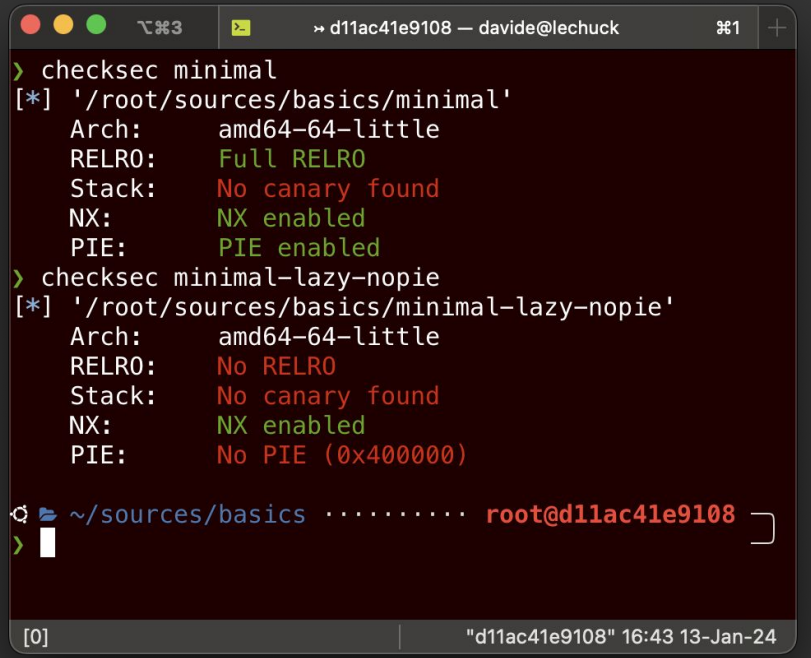
Security measures - PIC/PIE

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



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 (**G**nu **D**e**B**ugger - **G**DB **E**nhanced **F**eatures)

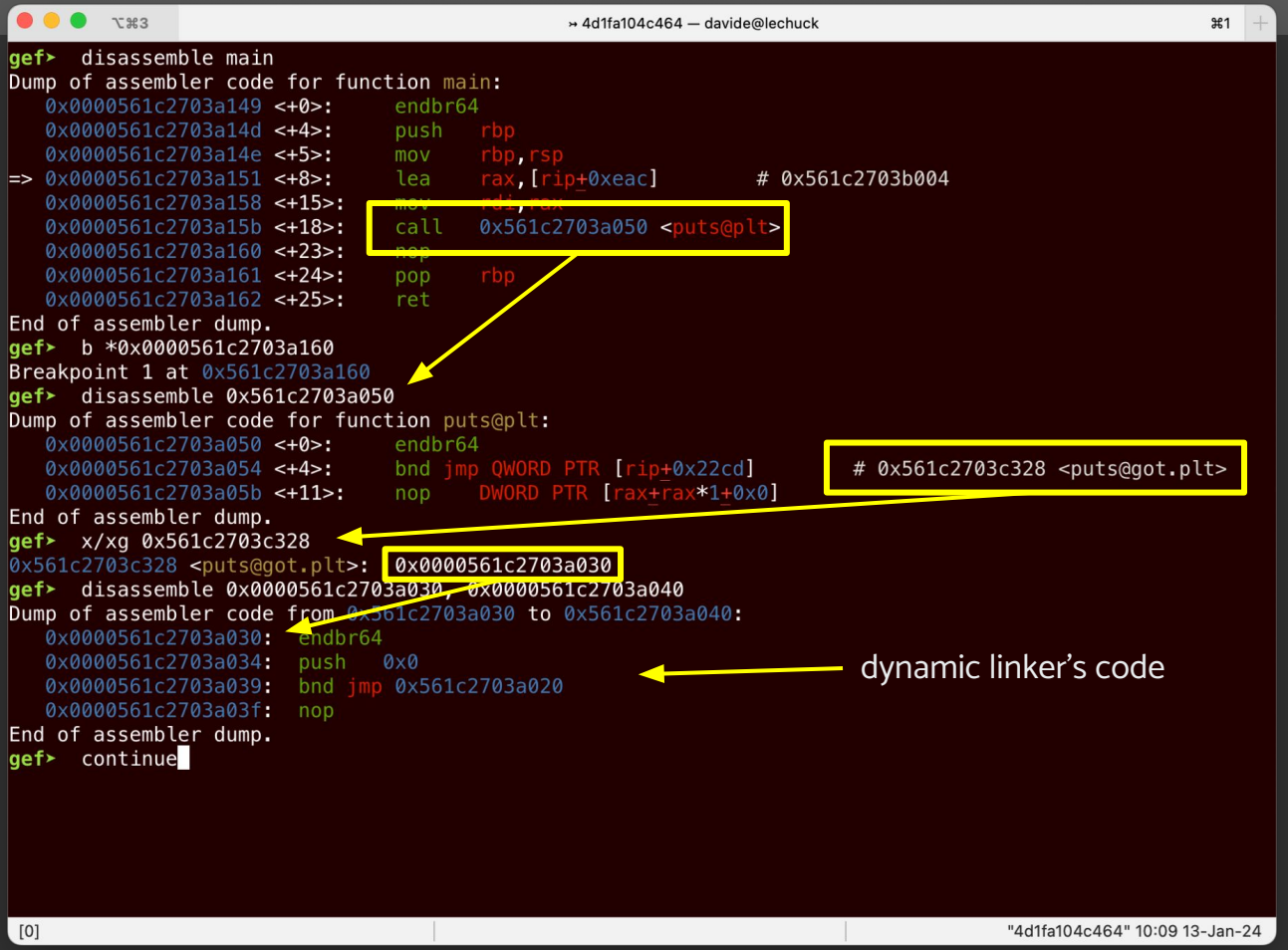
```
⌥⌘3                    ↣ 4d1fa104c464 — davide@lechuck
❯ ccat minimal.c
#include <stdio.h>

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
[0]                                                    "4d1fa104c
```

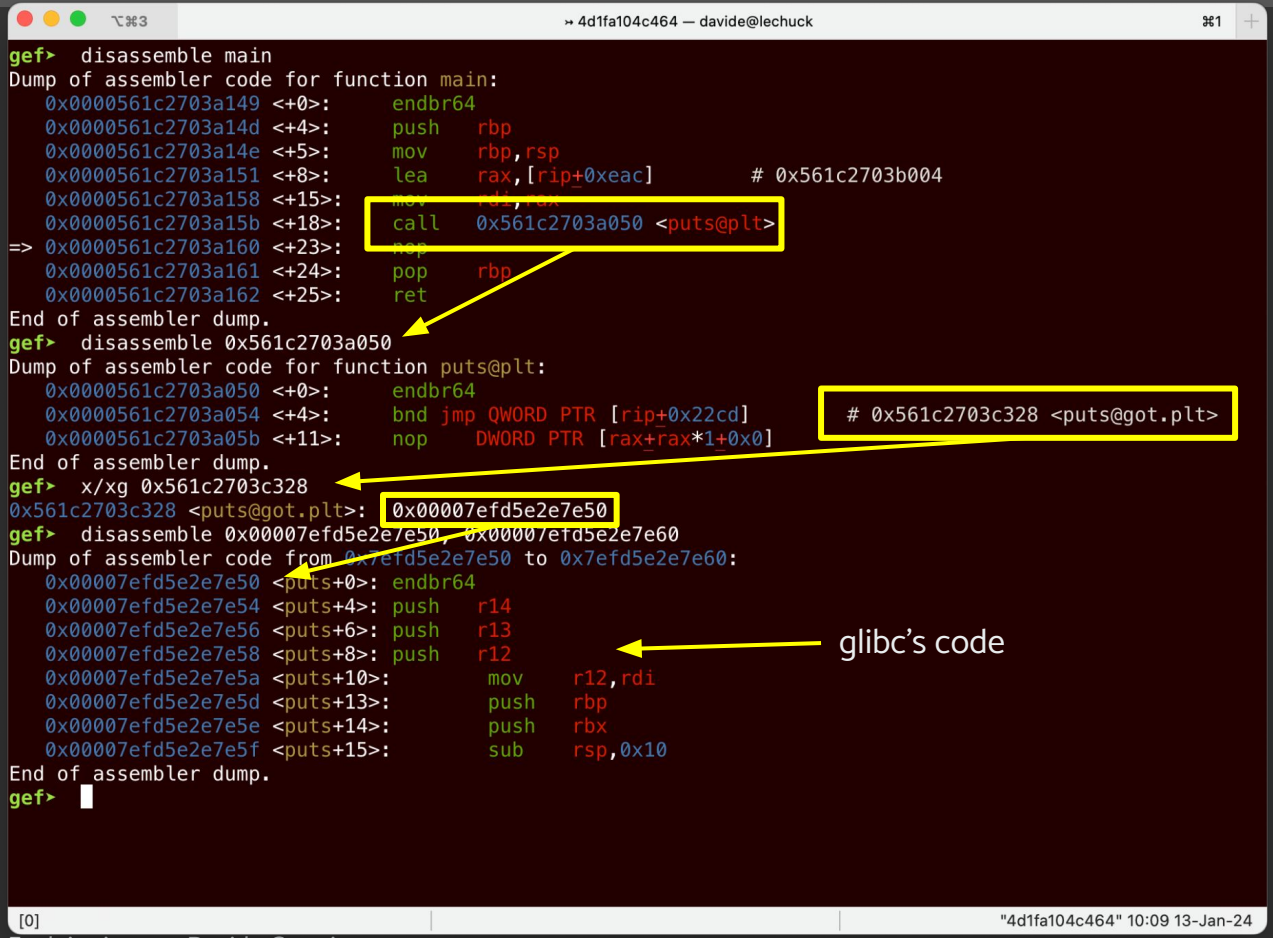
Before calling `puts()` the first time

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



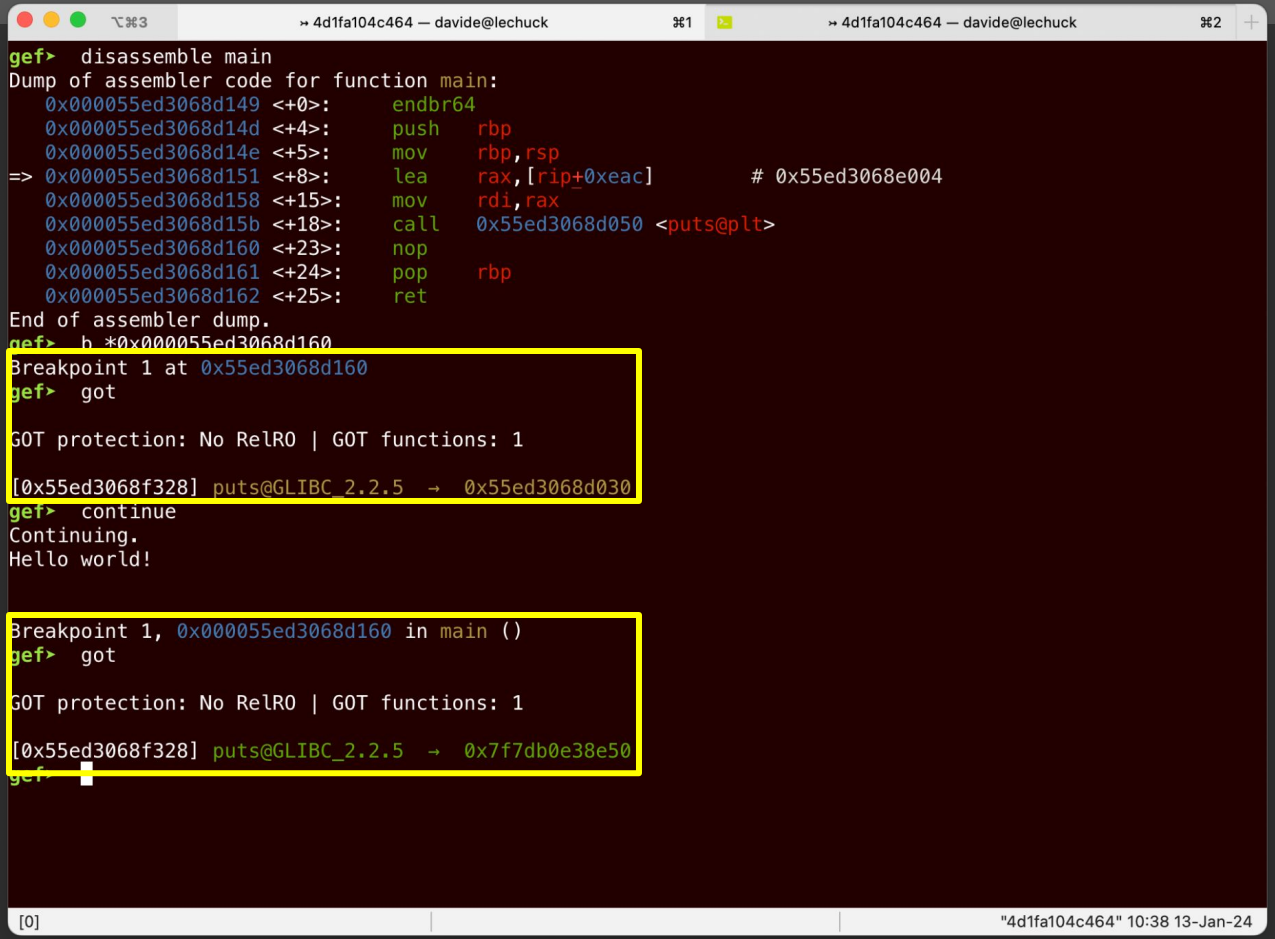
After calling `puts()`

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



Another way to see it

Dump the GOT (requires GEF)



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	
			Registers	Stack order
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)

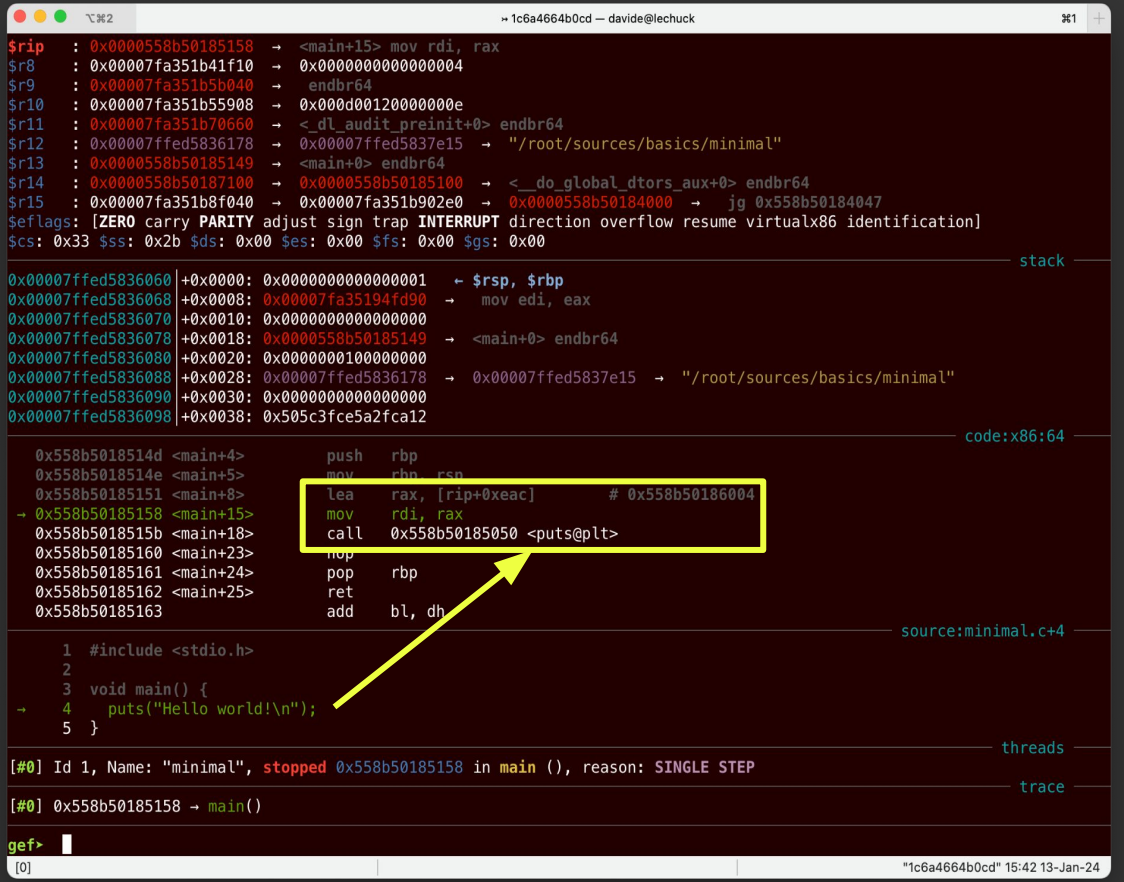
Calling conventions

x86-64 Linux gcc

`rdi`, `rsi`, `rdx`, `rcx`, ...

In this example:

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



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					

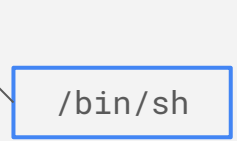
Shellcodes

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          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
```



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 - GEF
- GDB - GEP (GDB Enhanced Prompt)
- Online GDB
- What does the endbr64 instruction actually do?