Reverse engineering and binary analysis

Davide Berardi (D)

March 16, 2021

License

Agenda

- How programs are compiled
- ELF Structure
- Deassembly
- Decompilation
- Debug
- Anti debug
- Assembly
- Kernel space vs user space
- Systemcall vs library call
- Dynamic library vs static library
- Dynamic tracing

The real problem of C...

With Undefined Behavior, Anything is Possible

Aug 17, 2018

How programs are compiled

Suppose to have a C code like this:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    char who[] = world;
    print("Hello␣%s!\n", who);
    return 0;
}
```

Which are the passes to compile it?

Precompilation

The compiler is istructed to include code or translate code from other sources, therefore the content of stdio.h (which, in ubuntu is placed at /usr/include/stdio.h) is placed before the main. You can inspect the results using gcc -E or with cpp.

```
$ gcc -E /tmp/test.c | grep 'int printf'
extern int printf (const char *__restrict __format, ...);
```

Another example can be:

```
$ cat test.c
#define CIAO 5
int main(int argc, char **argv) {
    printf("%d\n", CIAO);
}
$ gcc -E test.c | grep printf
 printf("%d\n", 5);
```

Compilation

The compiler will translate (and optimize) the C code into Assembly code.

```
$ gcc -S test.c -o - 2>/dev/null | grep main: -A 15
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
```

Assembly

The optimized assembly code will therefore be translated in opcodes, the binary language understandable by the machine, using an assembler (as).

```
$ gcc -c -o test.o test.c
$ file test.o
test.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Linking

If we look inside the code we have just placeholder and not the code of the functions not declared in our C code. This code will be injected into the program by the linker (ld).

```
$ gcc -o test test.c
$ ./test
Hello world!
```

ELF Structure

The constructed binary is serialized in a structured file which is formatted in the Executable and Linkable Format. This format declares different areas called segments:

```
$ readelf -S $(which /bin/ls)
  ...
  [13] .text            PROGBITS          0000000000004040  00004040
       0000000000012db2 0000000000000000  AX       0     0     16
  [15] .rodata          PROGBITS          0000000000017000  00017000
       0000000000005309 0000000000000000   A       0     0     32
  [23] .data            PROGBITS          0000000000022000  00021000
       0000000000000268 0000000000000000  WA       0     0     32
  [24] .bss             NOBITS            0000000000022280  00021268
       00000000000012d8 0000000000000000  WA       0     0     32
  ...
```

ELF Structure

Some common segments are:

- **.interp** Contains the path of the interpreter which will be used tu run the program.
- **.text** Contains the code of the executable i.e. `return 3 + 4;`.
- **.rodata** Contains read-only data. i.e. the string ciao in `printf("ciao");`
- **.data** Contains read and writable data. i.e. static int x = 0x41;
- **.bss** Contains uninitialized global variables. i.e. static int x;

We will introduce some of the other segments, useful for security or exploit in a next lecture.

Information leak!

We can also read the plain-text readable portions of code!

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char *password = "super-secret-password";
    if (argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    if (!strcmp(password, argv[1]))
        printf("Access granted!\n");

    return 0;
}
```

```
$ gcc -o test test.c
$ strings test | grep pass
super-secret-password
```

Static analysis

If we invert the compilation process we can retrieve the assembly code of the program.
While the assembly is nearly 1:1 with the opcodes, the C program is not nearly 1:1 with the assembly code.
We can invert the assembly and the compilation phase with some techniques.

Going back: disassembly

The assembly phase is easily revertable, you can map the opcodes to their meaning in the x86_64 assembly. Beware that x86_64 does not use a different alphabet for data and code, therefore you can disassemble garbage (for example disassembling .rodata or .data).

```
$ objdump -D ./test | grep '^[0-9]\+ <main>' -A 10
0000000000001149 <main>:
    1149:     55                        push   %rbp
    114a:     48 89 e5                  mov    %rsp,%rbp
    114d:     48 83 ec 20               sub    $0x20,%rsp
    1151:     89 7d ec                  mov    %edi,-0x14(%rbp)
    1154:     48 89 75 e0               mov    %rsi,-0x20(%rbp)
    1158:     64 48 8b 04 25 28 00      mov    %fs:0x28,%rax
    115f:     00 00
    1161:     48 89 45 f8               mov    %rax,-0x8(%rbp)
    1165:     31 c0                     xor    %eax,%eax
    1167:     c7 45 f2 77 6f 72 6c      movl   $0x6c726f77,-0xe(%rbp)
```

Going back: disassembling

Some tools that can disassemble the code

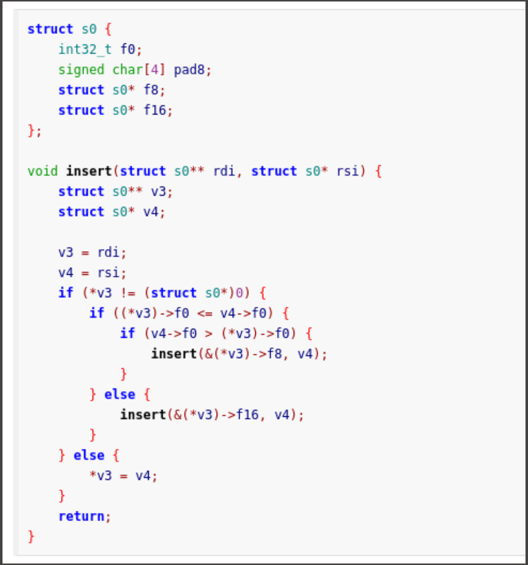
- **objdump** Open source, it is a basic disassembler, it can read most of the architecture but sometimes it can be disabled by simple tricks.
- **gdb** Open source, GNU debugger. It can disassemble most architectures.
- **radare2** Open source, it can disassemble most architectures but it is really bleeding edge and have a complex command set.
- **IDA** Closed source, available for linux, windows, and mac. A free version is available but it disassemble only x86_64 64-bit code.
- **Binary ninja** Closed source, available for linux, windows, and mac. A free version is available but it disassemble only x86_64 32-bit code.
- **Ghidra** Open source, developed by nsa

Going back: decompilation

From the assembly code we can try to reconstruct pseudo-C code. There are some decompilers which work using different techniques like:

- **Ghidra** Open source reverse engineering tool developed by NSA.
- **HexRay** Closed source decompiler, it can be attached to IDA.
- **Snowman** Open source decompiler.

Going back: decompilation, an example



```c
struct s0 {
    int32_t f0;
    signed char[4] pad8;
    struct s0* f8;
    struct s0* f16;
};

void insert(struct s0** rdi, struct s0* rsi) {
    struct s0** v3;
    struct s0* v4;

    v3 = rdi;
    v4 = rsi;
    if (*v3 != (struct s0*)0) {
        if ((*v3)->f0 <= v4->f0) {
            if (v4->f0 > (*v3)->f0) {
                insert(&(*v3)->f8, v4);
            }
        } else {
            insert(&(*v3)->f16, v4);
        }
    } else {
        *v3 = v4;
    }
    return;
}
```

Anti static analysis technique

Beware! The static analyzers are complex and most of the times try to analyze the code using heuristics and complex approaches. Expecially for open source code, never rely on a single tool.
An example of an anti debug technique can start by changing the headers searched by objdump, in this case this tool will not work.
There are two main approaches that a disassembler can use

- **Linear sweep**: scan the code and analyze it translating byte by byte.
- **Recursive descent**: during a Linear sweep, when a branch is encountered, a new linear sweep is executed on this new branch.

Anti static analysis technique: obfuscation

The code can also be obfuscated, by introducing strange optimizations or unuseful instructions to break the disassemblers.
N.B. x86 Intel syntax!!!

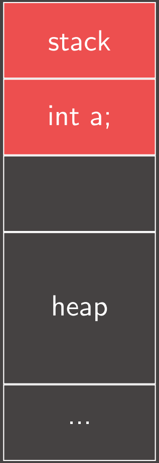
```
0000: B8 00 03 C1 BB   mov eax, 0xBBC10300
0005: B9 00 00 00 05   mov ecx, 0x05000000
000A: 03 C1            add eax, ecx
000C: EB F4            jmp $-10
```

If we read the code starting from 0x02 we get a totally different interpretation.

```
0002: 03 C1            add eax, ecx
0004: BB B9 00 00 00   mov ebx, 0xB9
0009: 05 03 C1 EB F4   add eax, 0xF4EBC103
000E: 03 C3            add eax, ebx
0010: C3               ret
```

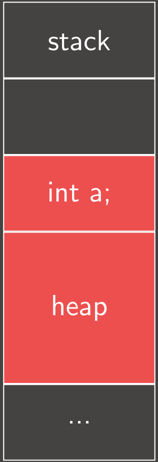
What is the stack?

The stack is the place where automatic memory of a program is placed, when you call a function or you declare an automatic variable it will be placed into this memory.

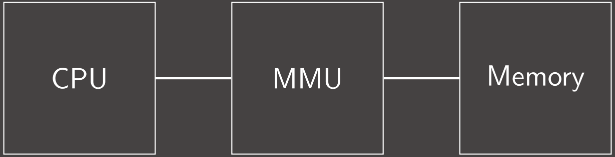


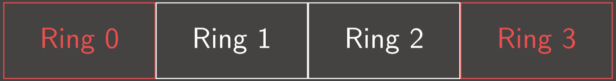
What is the heap?

The heap is the place where you can allocate memory (e.g. using malloc)



x86_64 Assembly





We will focus on x86_64 assembly because it is the most common in CTF nowadays.

GP Registers

A x86_64 cpu have 16 GP registers

- ***ax, *bx, *cx, *dx, *si, *di**
- ***sp, *bp**
- **r8-r15**

These can be accessed in 4 ways:

- full register (64 bit): **rax**
- half register (lowest 32 bit): **eax**
- 1/4 register (lowest 16 bit): **ax**
- 1/8 register (lowest 8 bit): **al**

x86_64 special registers

There are some registers which are automatically used by the CPU, these are:

- **rip** Points to the instruction that will be executed at the next step.
- **rsp** used automatically to keep the pointer to the stack frame with push and pop;
- **rflags** The flags set that describes the status of the current run (e.g. zero to indicate that the previous instruction returned 0).
- Also, **rbp** is not used by the CPU automatically but it is normally used to calculate offset from memory locations.

x86_64 SIMD registers

xmm0-15 registers are 16 registers which enables the use of SIMD instructions (Single Instruction Multiple Data). Usually there are 128-bit or 256-bit and can be used to accelerate cryptographical operations or vector graphics.

- AESENC xmm1,xmm2/m128 —Perform One Round of an AES Encryption Flow round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.
- AESENCLAST xmm1, xmm2/m128 —Perform Last Round of an AES Encryption Flow a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.
- AESKEYGENASSIST xmm1, xmm2/m128, imm8 Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

Example of x86_64 assembly code

```
xor %rax, %rax;
mov $$0xFF978CD091969DD1, %rbx ;
neg %rbx;
push %rbx;
push %rsp;
pop %rdi;
cdq;
push %rdx;
push %rdi;
push %rsp;
pop %rsi;
mov $$0x3b, %al;
syscall
```

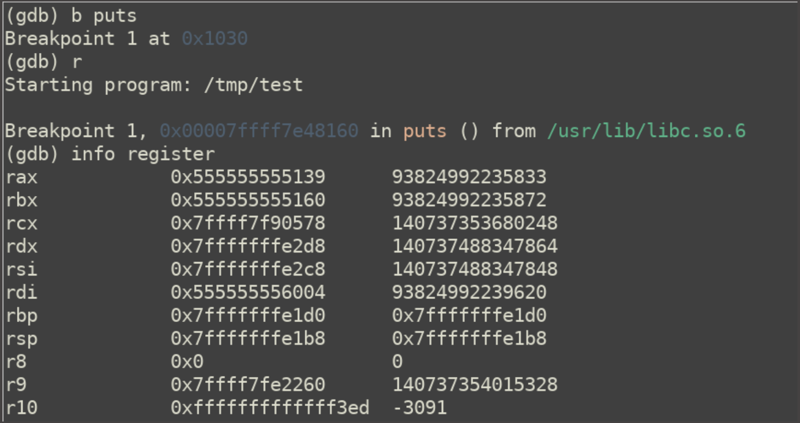
Example of x86_64 assembly code

Parameters for functions get passed in RDI, RSI, RDX, RCX, R8, R9, XMM0–7

```
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
mov     %edi,-0x4(%rbp)
mov     %rsi,-0x10(%rbp)
lea     0xeb5(%rip),%rdi
callq   1030 <system@plt>
mov     $0x0,%eax
leaveq
retq
nopl    0x0(%rax,%rax,1)
```

Dynamic analysis: debugging

We can use a debugger to see the code running and to analyze the instruction that are executed.



```
(gdb) b puts
Breakpoint 1 at 0x1030
(gdb) r
Starting program: /tmp/test

Breakpoint 1, 0x00007ffff7e48160 in puts () from /usr/lib/libc.so.6
(gdb) info register
rax            0x555555555139      93824992235833
rbx            0x555555555160      93824992235872
rcx            0x7ffff7f90578      140737353680248
rdx            0x7fffffffe2d8      140737488347864
rsi            0x7fffffffe2c8      140737488347848
rdi            0x555555556004      93824992239620
rbp            0x7fffffffe1d0      0x7fffffffe1d0
rsp            0x7fffffffe1b8      0x7fffffffe1b8
r8             0x0                 0
r9             0x7ffff7fe2260      140737354015328
r10            0xfffffffffffff3ed  -3091
```

Dynamic analysis: debugging

We will focus on gdb, some example commands of gdb are:

- **r** $<$ $<$**(shell command)** run the program with input from a shell script (like a pipe).
- **ni** next instruction without following calls.
- **si** step in following jumps.
- **info register** print the status of the
- **b printf** Break on printf invokation.
- **b *0x123456** Break on address 0x123456.
- **d3** Delete breakpoint 3.

GDB (and generally debuggers under linux) use a systemcall called **ptrace**, which can trace a process and retrieve the current status of the registers.

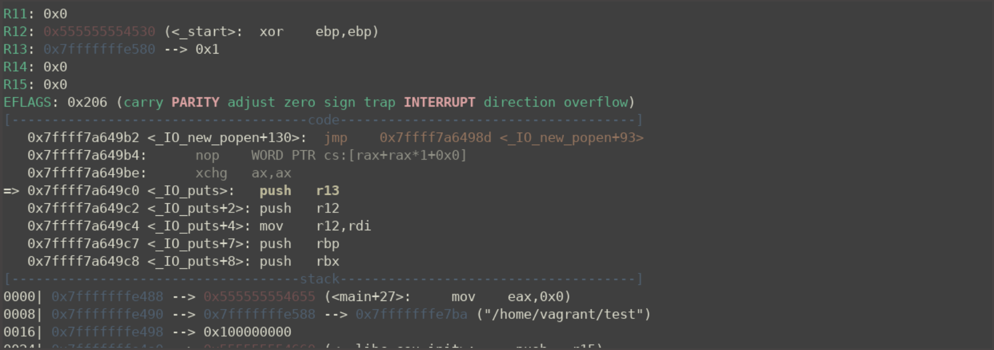
A glimpse of future!

GDB therefore can change the registers and modify the code of the process.
What happens if you attach a privileged process? and a privileged executable? (e.g. `setuid`)

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

- **peda**, Python Exploit Development Assistance. A python init script for gdb to debug the program in a more user-friendly way.
- `https://github.com/longld/peda`

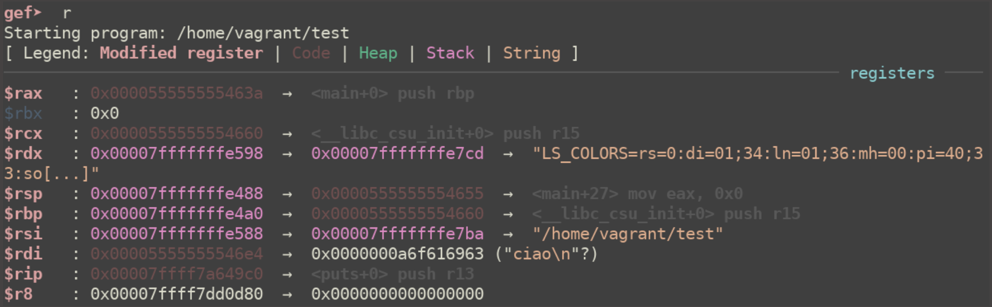


```
R11: 0x0
R12: 0x555555554530 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe580 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff7a649b2 <_IO_new_popen+130>:  jmp    0x7ffff7a6498d <_IO_new_popen+93>
   0x7ffff7a649b4:      nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7ffff7a649be:      xchg   ax,ax
=> 0x7ffff7a649c0 <_IO_puts>:   push   r13
   0x7ffff7a649c2 <_IO_puts+2>: push   r12
   0x7ffff7a649c4 <_IO_puts+4>: mov    r12,rdi
   0x7ffff7a649c7 <_IO_puts+7>: push   rbp
   0x7ffff7a649c8 <_IO_puts+8>: push   rbx
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe488 --> 0x555555554655 (<main+27>:     mov    eax,0x0)
0008| 0x7fffffffe490 --> 0x7fffffffe588 --> 0x7fffffffe7ba ("/home/vagrant/test")
0016| 0x7fffffffe498 --> 0x100000000
```

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

- **gef**, GDB Enhanced Features. Like peda, this init script makes the use of gdb more user-friendly. Super useful for heap analysis!
- `https://github.com/hugsy/gef`

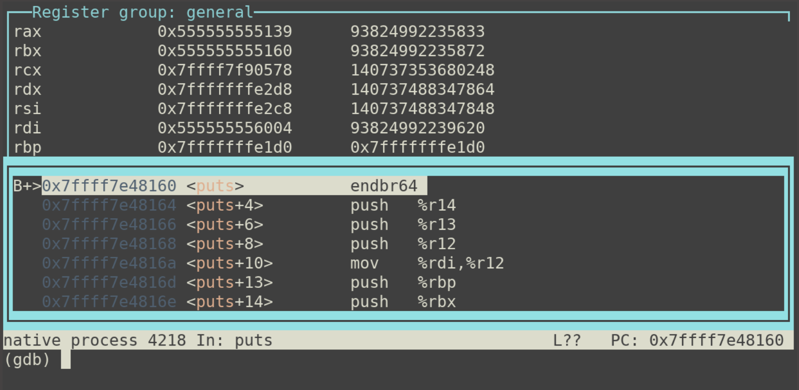


```
gef➤  r
Starting program: /home/vagrant/test
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x000055555555463a  →  <main+0> push rbp
$rbx   : 0x0
$rcx   : 0x0000555555554660  →  <__libc_csu_init+0> push r15
$rdx   : 0x00007fffffffe598  →  0x00007fffffffe7cd  →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;3
3:so[...]"
$rsp   : 0x00007fffffffe488  →  0x0000555555554655  →  <main+27> mov eax, 0x0
$rbp   : 0x00007fffffffe4a0  →  0x0000555555554660  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007fffffffe588  →  0x00007fffffffe7ba  →  "/home/vagrant/test"
$rdi   : 0x00005555555546e4  →  0x0000000a6f616963 ("ciao\n"?)
$rip   : 0x00007ffff7a649c0  →  <puts+0> push r13
$r8    : 0x00007ffff7dd0d80  →  0x0000000000000000
```

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

- **layout next**, while not being an extension, it can help the debug process and the concurrent visualization of the assembly and the status of the registries.
- `https://www.youtube.com/watch?v=PorfLSr3DDI`



```
┌─Register group: general─────────────────────────────────────────────┐
│rax            0x555555555139      93824992235833
│rbx            0x555555555160      93824992235872
│rcx            0x7ffff7f90578      140737353680248
│rdx            0x7fffffffe2d8      140737488347864
│rsi            0x7fffffffe2c8      140737488347848
│rdi            0x555555556004      93824992239620
│rbp            0x7fffffffe1d0      0x7fffffffe1d0

B+>0x7ffff7e48160 <puts>           endbr64
   0x7ffff7e48164 <puts+4>         push   %r14
   0x7ffff7e48166 <puts+6>         push   %r13
   0x7ffff7e48168 <puts+8>         push   %r12
   0x7ffff7e4816a <puts+10>        mov    %rdi,%r12
   0x7ffff7e4816d <puts+13>        push   %rbp
   0x7ffff7e4816e <puts+14>        push   %rbx

native process 4218 In: puts                              L??   PC: 0x7ffff7e48160
(gdb)
```

Library

A library is a collection of functions that exports symbols available to be linked to the other applications.

```
$ nm /lib64/libasan.so
0000000000116710 T __asan_address_is_poisoned
0000000000035b50 T __asan_addr_is_in_fake_stack
0000000000038710 T __asan_after_dynamic_init
0000000000035be0 T __asan_alloca_poison
...
0000000000188ea0 d _ZZN6__asan22ErrorAllocTypeMismatch5PrintEvE13dealloc_name
000000000019d758 b _ZZN6__asan26InitializeAsanInterceptorsEvE15was_called_onc
00000000001a2c74 b _ZZN6__asanL15AsanCheckFailedEPKciS1_yyE9num_calls
00000000001a2c78 b _ZZN6__asanL7AsanDieEvE9num_calls
```

Dynamic linking

The executable can also be linked to external library to save space and simplify the updates of the system.
This is achieved by the linker and is the standard behaviour of linux c compilers.

```
$ cat - <<_END_ >test.c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("hello world!");
    return 0;
}
$ gcc --static -o test test.c
$ du -h test
764K    test
```

```
$ cat - <<_END_ >test.c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("hello world!");
    return 0;
}
$ gcc -o test test.c
$ du -h test
20K     test
```

Dynamic linking analysis

When the program is executed the loader (**.interp** section) will load the libraries and update the reference in the code we will see the procedure in details in **Software security 2**.

```
$ ldd $(which ls)
linux-vdso.so.1 (0x00007ffe55373000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fd754796000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd7543a5000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fd754133000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fd753f2f000)
/lib64/ld-linux-x86-64.so.2 (0x00007fd754be0000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fd753d10000)
```

What is a kernel

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system.[1]

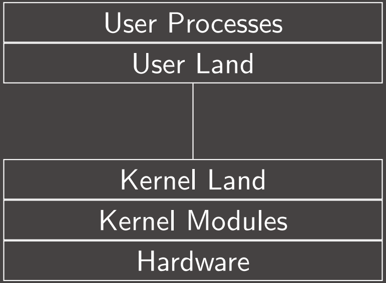
- We will focus on monolitic kernels (expecially linux and BSD).
- The kernel is responsible for:
  - Manage the lifecycle of the userland (processes);
  - Manage resources;
  - Interacting with hardware;
  - **Security of the system**.

[1] `http://www.linfo.org/kernel.html`

What is the userland

> *The term userland (or user space) refers to all code that runs outside the operating system's kernel.*[2]

Probably most of the code you wrote runs in user space. If you need to write a device driver for Linux it will be in kernel space.



[2] `https://en.wikipedia.org/wiki/User_space`

Process

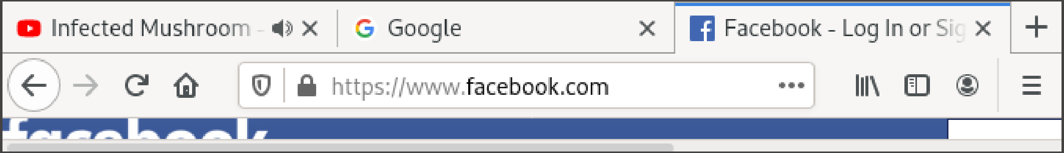
A process is an instance of a program. Beware that in linux the terms process, task, and thread are sometimes misleading!
In general term

- Task - the task of a process, what you want to achieve and how.
- Thread - an instance of a program, share the memory with other related threads.
- Process - a container of threads which share the same memory.

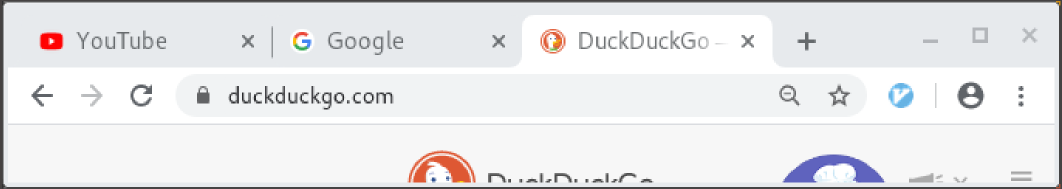
- In Linux a Thread and a process are the same thing! A process is a thread with separated memory from the other processes.

Examples of threads and process

In firefox every tab was a thread



In chrome every tab was a process



Process loading

When a program is launched, the interpreter (**loader**) is executed and the content of its **ELF** is loaded in memory by the **MMU** (the **.text**, **.rodata**, ...). The **.bss** section is therefore initialized to zero (mapped to an empty page).
The dynamic libraries are therefore loaded in memory and shared between common process (This could be a comment to docker if you want :)).

Segmented memory

The memory in a program is segmented similarly to the ELF. In this case the system will load different areas, comprending area allocated for the heap and the stack (allocated by the **MMU**).

```
$ cat /proc/self/maps
55ddcdbfe000-55ddcdc02000 r-xp 00002000 08:01 2232386 /usr/bin/cat
...
55ddced1e000-55ddced3f000 rw-p 00000000 00:00 0       [heap]
7f2fb5442000-7f2fb5467000 r--p 00000000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5467000-7f2fb55b3000 r-xp 00025000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5601000-7f2fb5604000 rw-p 001be000 08:01 2231702 /usr/lib/libc-2.31.so
...
7f2fb5604000-7f2fb560a000 rw-p 00000000 00:00 0
7f2fb563e000-7f2fb5640000 r--p 00000000 08:01 2231656 /usr/lib/ld-2.31.so
7f2fb5640000-7f2fb5660000 r-xp 00002000 08:01 2231656 /usr/lib/ld-2.31.so
...
7f2fb566a000-7f2fb566b000 rw-p 0002b000 08:01 2231656 /usr/lib/ld-2.31.so
7f2fb566b000-7f2fb566c000 rw-p 00000000 00:00 0
7ffc1ac7b000-7ffc1ac9c000 rw-p 00000000 00:00 0       [stack]
...
```

Library hijacking

When you declare a library call:

```
puts("ciao\n");
```

You get a call into the library

```
1154:      e8 d7 fe ff ff                callq  1030 <puts@plt>
```

So...Can we hijack the call changing the library?

Library hijacking

Yes you can! At loading time Without relinking the application :D

```
$ ./test
ciao
$ LD_LIBRARY_PRELOAD=./fakelib.so ./test
hacked
```

Using this environment variable you can hijack the library call hooking and changing the behaviour of the library functions.

Library hijacking: tracing

Using this trick you can trace the execution of a program without debugging it with conventional method.

```
$ cat test.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    if (argc < 2)
        return 1;
    if (!strcmp("super-secure-password", argv[1]))
        printf("Access granted!\n");
    return 0;
}
$ ltrace ./test ciao
strcmp("super-secure-password", "ciao")                  = 16
+++ exited (status 0) +++
```

Another question!

What happens if you use **LD_LIBRARY_PRELOAD** over a privileged (**setuid** or with ep capabilities) executable?

Systemcall

A systemcall is a procedure to communicate with the kernel. Issuing the systemcall the system will process the request and operate accordingly.
Some examples of systemcalls are:

- open
- read
- write
- socket

Systemcall: assembly

The operating system must agree on the procedure used by the userland program to retrieve the correct parameters from the userland.
An x86 32bit example:

```
int 0x80;
```

The trap way was too slow! It was microprogrammed to a dedicated opcode in x86_64

```
syscall
```

The systemcall number is loaded in rax (eax on 32 bit);
Parameters get passed in eax, ebx, ecx, edx, esi, edi, ebp.
and the return code for the systemcall is returned to the user throug rax (eax) register.
For 64 bit systems the parameters are passed in rdi, rsi, rdx, r10, r8, r9.

Library vs systemcall

A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char c;
  int counter = 0;
  FILE *f = fopen("/dev/urandom",
                  "r");

  while (counter < 100 * 1000) {
    fread(&c, 1, 1, f);
    if (c == '\xff')
      counter++;
  }
  printf("%d\n", counter);
  fclose(f);
  return 0;
}
```

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char **argv) {
  char c;
  int counter = 0;
  int f = open("/dev/urandom",
               O_RDONLY);
  while (counter < 100 * 1000) {
    read(f, &c, 1);
    if (c == '\xff')
      counter++;
  }
  printf("%d\n", counter);
  close(f);
  return 0;
}
```

Library vs systemcall

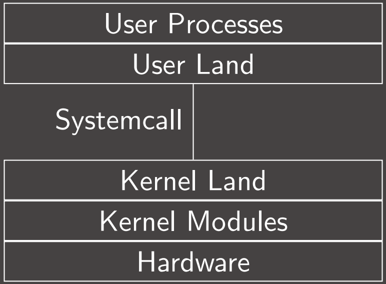
A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?

```
$ time ./test-fread
100000
./test-fread  0.46s user 0.16s system 93% cpu 0.667 total

$ time ./test-read
100000
./test-read  5.82s user 15.15s system 99% cpu 21.056 total
```

Library vs systemcall

A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?



Every read you are making a context switch, fread will read a block of data and return to you byte by byte without the switch.

Dynamic tracing of systemcall: ptrace

Ptrace is a systemcall which can control the behaviour of a traced program. As stated before is the systemcall that gdb uses to debug programs. It can be instructed to retrieve memory, registries and systemcall invoked by the traced process.

```
long ptrace(enum __ptrace_request request, pid_t pid,
                    void *addr, void *data);
```

Dynamic tracing of systemcall: strace

Strace is a tool based on ptrace that can dynamically analyze a program to print out all the systemcall that gets issued.

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
 int main(int argc, char **argv)
{
    char buffer[4096];
    int f;
    if (argc < 2)
        return 1;
    f = open("./password", O_RDONLY);
    read(f, buffer, sizeof(buffer));
    if (!strcmp(argv[1], buffer))
        printf("Access␣granted\n");
    close(f);
    return 0;
}
```

Dynamic tracing of systemcall: strace

Strace is a tool based on ptrace that can dynamically analyze a program to print out all the systemcall that gets issued.

```
$ gcc --static -o test-read2 test-read2.c
$ ltrace ./test-read2 ciao
Couldn't find .dynsym or .dynstr in "/proc/19647/exe"
$ strace ./test-read2 ciao 2>&1 | grep read
execve("./test-read2", ["./test-read2", "ciao"], 0x7ffda35b8618 /* 45 vars */
readlink("/proc/self/exe", "/tmp/test-read2", 4096) = 15
read(3, "super-secure-password", 4096) = 21
```

Anti dynamic analysis technique

Ptrace can be simply eluded by tracing itself and disabling ptrace mechanism, according to manpages:

```
EPERM The specified process cannot be traced.  This could be because the
tracer has insufficient privileges  (the required capability is
CAP_SYS_TRACE); unprivileged processes cannot trace processes that tehy
cannot send signals to or those running set-user-ID/set-group-ID
programs, for obvious reasons.  Alternatively, the process may already
be being traced, or (on kernels before 2.6.26) be init(1) (PID 1).
```

Also, sometimes language virtual machines are employied to obfuscate the code.

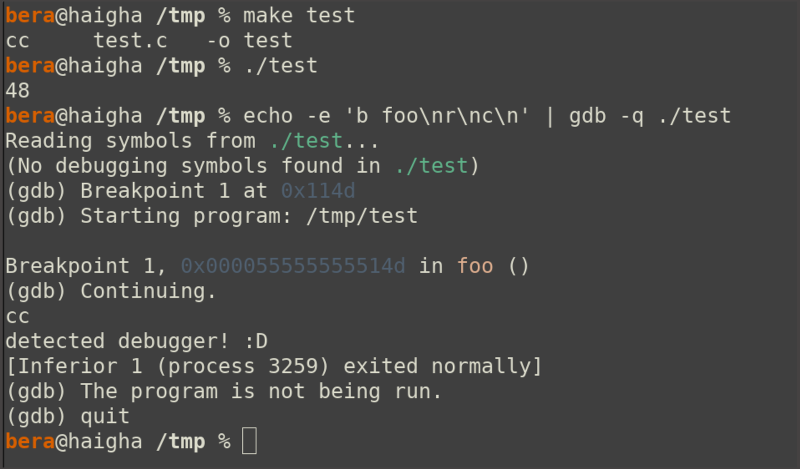
Anti dynamic analysis technique: anti breakpoint

Debuggers insert the opcode 0xcc (int 3) in the program to trace breakpoints. With this technique a program can check if it is being debugged checking for breakpoints.

```
#include <stdio.h>
#define PRINT_SIZE 16
int foo()
{
    unsigned char x = *(((unsigned char *)foo) + 4);
    printf("%02x\n", x);
    if (x == 0xcc)
        printf("detected debugger! :D\n");
    return 0;
}
int main(int argc, char **argv)
{
    foo();
    return 0;
}
```

Anti dynamic analysis technique: anti breakpoint

Debuggers insert the opcode 0xcc (int 3) in the program to trace breakpoints. With this technique a program can check if it is being debugged checking for breakpoints.



```
bera@haigha /tmp % make test
cc     test.c   -o test
bera@haigha /tmp % ./test
48
bera@haigha /tmp % echo -e 'b foo\nr\nc\n' | gdb -q ./test
Reading symbols from ./test...
(No debugging symbols found in ./test)
(gdb) Breakpoint 1 at 0x114d
(gdb) Starting program: /tmp/test

Breakpoint 1, 0x000055555555514d in foo ()
(gdb) Continuing.
cc
detected debugger! :D
[Inferior 1 (process 3259) exited normally]
(gdb) The program is not being run.
(gdb) quit
bera@haigha /tmp % 
```

Systemcall firewalls: seccomp

Systemcalls can be firewalled in linux using BGP. This can be loaded in the following way:
BPF (Berkeley packet filter, a concept ported from BSD systems) is a language virtual machine embedded in the linux kernel to accelerate the filtering of firewalls (like iptables). That language specifies rules to accept or drop a packet. In this case the systemcall parameters are mapped as packet field. In this way the systemcall can be filtered by the invoking program, limiting the set of usable systemcall.

Systemcall firewalls: seccomp

Other kernel have different implementation of this concept like **pledge(2)** in OpenBSD.
This will be clear in subsequent **Software security** lessons, for the moment imagine a firewall that can block the **ptrace** systemcall.
In this case the use of gdb will be blocked by the firewall itself.

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char buf[8];
    FILE *f;
    if (argc < 2)
        return 1;
    f = fopen("/dev/urandom", "r");
    fread(buf, 1, sizeof(buf), f);
    fclose(f);
    if (!memcmp(buf, argv[1], sizeof(buf)))
        printf("Wow!\n");
    return 1;
}
```

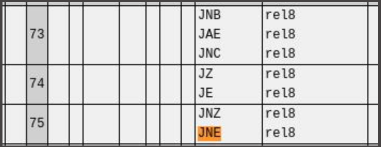
Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

```
11fc:       48 8b 08                mov    (%rax),%rcx
11ff:       48 8d 45 f0             lea    -0x10(%rbp),%rax
1203:       ba 08 00 00 00          mov    $0x8,%edx
1208:       48 89 ce                mov    %rcx,%rsi
120b:       48 89 c7                mov    %rax,%rdi
120e:       e8 5d fe ff ff          callq  1070 <memcmp@plt>
1213:       85 c0                   test   %eax,%eax
1215:       75 11                   jne    1228 <main+0x9f>
1217:       48 8d 3d f5 0d 00 00    lea    0xdf5(%rip),%rdi
121e:       b8 00 00 00 00          mov    $0x0,%eax
1223:       e8 38 fe ff ff          callq  1060 <printf@plt>
```

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.
We can see that the **jne** code is 0x75.



73	JNB	rel8
	JAE	rel8
	JNC	rel8
74	JZ	rel8
	JE	rel8
75	JNZ	rel8
	JNE	rel8

What happens if we change it to **je** (0x74)?

```
$ xxd -ps test > test.hex
$ sed -i 's/ffff85c07511/ffff85c07411/' test.hex
$ xxd -ps -r test.hex > test
```

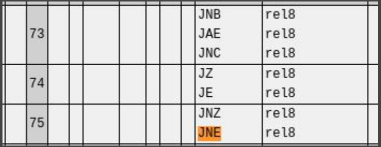
Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

```
11fc:       48 8b 08                  mov    (%rax),%rcx
11ff:       48 8d 45 f0               lea    -0x10(%rbp),%rax
1203:       ba 08 00 00 00            mov    $0x8,%edx
1208:       48 89 ce                  mov    %rcx,%rsi
120b:       48 89 c7                  mov    %rax,%rdi
120e:       e8 5d fe ff ff            callq  1070 <memcmp@plt>
1213:       85 c0                     test   %eax,%eax
1215:    74 11                      je     1228 <main+0x9f>
1217:       48 8d 3d f5 0d 00 00      lea    0xdf5(%rip),%rdi
121e:       b8 00 00 00 00            mov    $0x0,%eax
1223:       e8 38 fe ff ff            callq  1060 <printf@plt>
```

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.
We can see that the **jne** code is 0x75.



73	JNB	rel8
	JAE	rel8
	JNC	rel8
74	JZ	rel8
	JE	rel8
75	JNZ	rel8
	JNE	rel8

What happens if we change it to **je** (0x74)?

```
$ ./test ciao
Wow!
```

Symbolic analysis: angr

Another way to analyze binaries is the symbolic analysis. You encode your binary in a solver and delimit some constraints over your solution.

```
import angr

project = angr.Project("angr-doc/examples/defcamp_r100/r100", auto_load_libs=

@project.hook(0x400844)
def print_flag(state):
    print("FLAG_SHOULD_BE:", state.posix.dumps(0))
    project.terminate_execution()

project.execute()
```

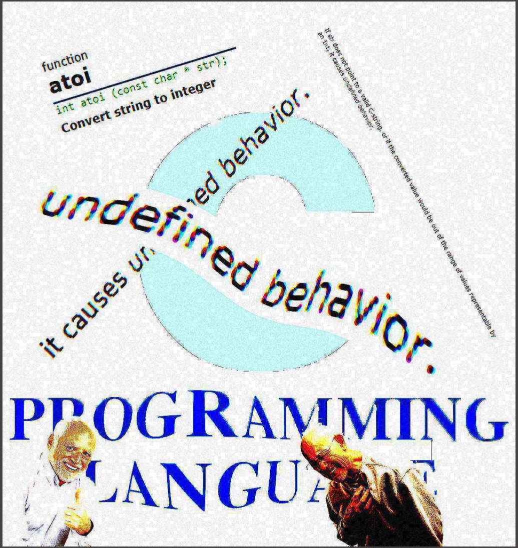
Buffer overflow

We will introduce pwn and buffer overflows in a following lecture. A buffer overflow is an attack that use a misconfigured buffer length:

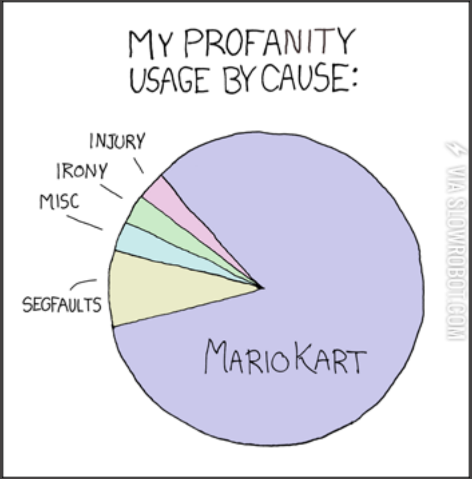
```c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char buf[8];
    strcpy(buf, argv[1]);
    return 0;
}
```

What happens if we copy more than 8 characters in the buf?

Nasal demons again!


function
atoi
int atoi (const char * str);
Convert string to integer
If str does not point to a valid C-string, or if the converted value would be out of the range of values representable by an int, it causes undefined behavior.
it causes undefined behavior.
undefined behavior.
PROGRAMMING
LANGUAGE

We meet again...my old nemesis!



If we are lucky, in CTFs sometimes we struggle to get a segmentation fault. :)
Sometimes...well...the program randomly segfaults.

Secure coding

In this case we can act defensively adding a check on the buffer and placing a terminating zero at the end of the string.

```
#include <stdio.h>
#include <string.h>
#define BUFLEN 8
int main(int argc, char **argv) {
    char buf[BUFLEN + 1];
    if (argc < 2)
        return 1;

    strncpy(buf, argv[1], BUFLEN);
    buf[BUFLEN] = '\0';

    return 0;
}
```

Format string leak

Even misplaced **printf** can be problematic for the security of a program, providing a leak. We will exploit this behaviour in a different lecture, but what happens if we write something like this?

```c
#include <stdio.h>
int main(int argc, char **argv) {
    char *password = "super-secret-password";
    if (argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    printf("Hello ");
    printf(argv[1]);
    printf("\n");
    /* ... */
    return 0;
}
```

Format string leak

[language=C] Even misplaced **printf** can be problematic for the security of a program, providing a leak. We will exploit this behaviour in a different lecture, but what happens if we write something like this?

```
$ ./test "%x%x%x%x%x%x"
Hello 2000c76cf2a0679457d18
```

Race condition: a real world example

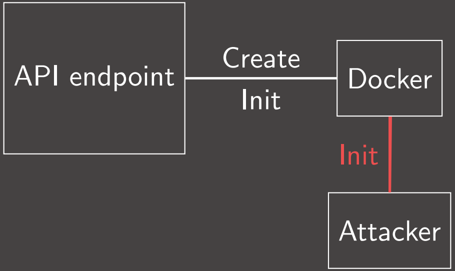
Suppose to have an application that have three methods in its **API**.

- **create**, the first phase is the creation of an environment where the code get executed. This is a privileged action.
- **init**, the init phase will inject the code to run in the run phase. In our case this was not a privileged action. If an application is already initialized, the init will fail.
- **run**, in the run phase the code will get executed and takes parameters from the user.

Can you spot the error here?

Race condition: a real world example

If we keep flooding the system with **init** we can reach the API endpoint before the authorized system, enabling our malicious program to take control over the infrastructure and tampering the system.



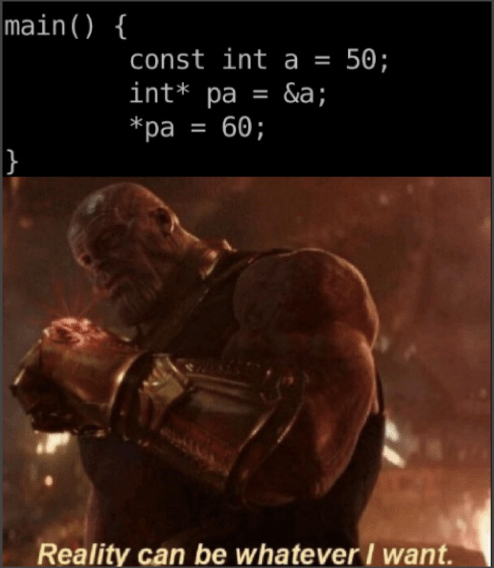
FYI the problem was even worst that this one ;)

Where nasal demons come from?

```
int x = 10;
int y = 10;
y=y*(2*(++x) - 2*(1-x--));

GCC: 400
Clang: 420
```

Well...again



```
main() {
        const int a = 50;
        int* pa = &a;
        *pa = 60;
}
```