Software Security 1

Binary Patching
Dynamic Analysis
Anti-(Reversing|Debugging) Techniques
SMT solvers & Symbolic Execution

Carlo Ramponi <carlo.ramponi@unitn.it>

Binary Patching

Carlo Ramponi

Binary Patching

Binary Patching consists of **making changes to a program**, by **directly modifying the binary**, when the source code is not available or the re-compilation is not possible.

Binary Patching can affect every part of the binary file:

- **Data Section**: tampering with the data saved in the binary file (e.g. a string)
- **Code Section**: overwriting binary instructions to change the behavior of the program.
  - Can't add or remove instructions
  - Can't replace instructions with longer or shorter instructions
- **ELF headers**: modifying the program's metadata

Binary Patching

There are many ways to patch a binary:

- Directly writing binary data in a specific location of the file
  - Hex editor (e.g. **`bless`**, **`hexedit`**)
  - Programmatically (e.g. pure Python)
- More complex editors that also include an assembler / disassembler
  - Ghidra
  - Python (with libraries such as **`python-capstone`**)

Binary Patching - Data section


```
carlo@carlo-pc ~/Desktop/EH24/Code ./es
Hello World!
Tough luck, the number is not 42
```

Let’s see if we can make the program print something else instead of **“Hello World!”**

1. Find where the string is stored in the binary, we can use **bgrep** (binary **grep**)

Binary Patching - Data section


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./es
Hello World!
Tough luck, the number is not 42
 carlo@carlo-pc  ~/Desktop/EH24/Code  bgrep '"Hello World"' es
0002008: 4865 6c6c 6f20 576f 726c 64              Hello World
```

Let’s see if we can make the program print something else instead of **“Hello World!”**

2. Write the new string at **0x2008**, we can use **dd**

Binary Patching - Data section


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./es
Hello World!
Tough luck, the number is not 42
 carlo@carlo-pc  ~/Desktop/EH24/Code  bgrep '"Hello World"' es
0002008: 4865 6c6c 6f20 576f 726c 64              Hello World
 carlo@carlo-pc  ~/Desktop/EH24/Code  echo 'Hacked dude!' | dd of=es conv=notrunc oseek=$((0x0002008)) bs=1
13+0 records in
13+0 records out
13 bytes copied, 0.00044937 s, 28.9 kB/s
```

Let’s see if we can make the program print something else instead of **“Hello World!”**

3. We are done, run the patched program!

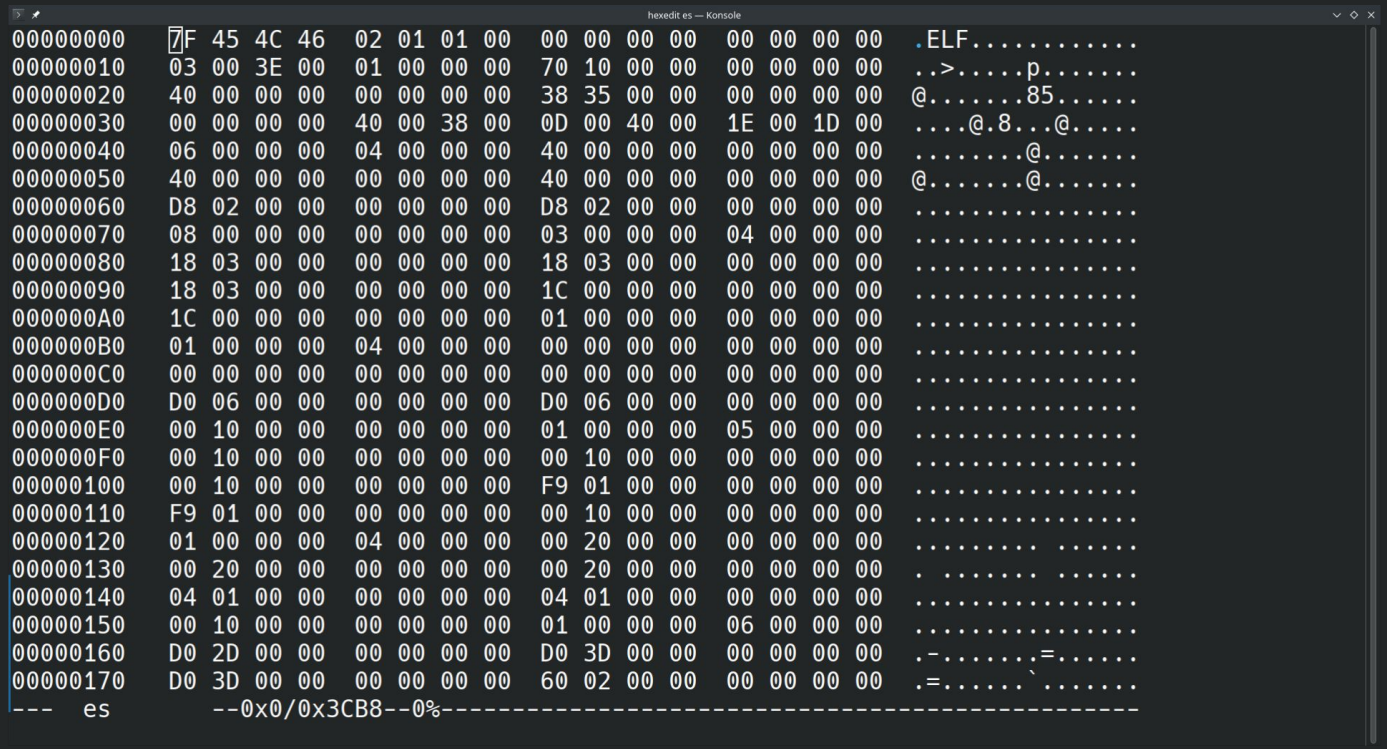
Binary Patching - Data section


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./es
Hello World!
Tough luck, the number is not 42
 carlo@carlo-pc  ~/Desktop/EH24/Code  bgrep '"Hello World"' es
0002008: 4865 6c6c 6f20 576f 726c 64             Hello World
 carlo@carlo-pc  ~/Desktop/EH24/Code  echo 'Hacked dude!' | dd of=es conv=notrunc oseek=$((0x0002008)) bs=1
13+0 records in
13+0 records out
13 bytes copied, 0.00044937 s, 28.9 kB/s
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./es
Hacked dude!

Tough luck, the number is not 42
```

Let’s see if we can make the program print something else instead of **“Hello World!”**

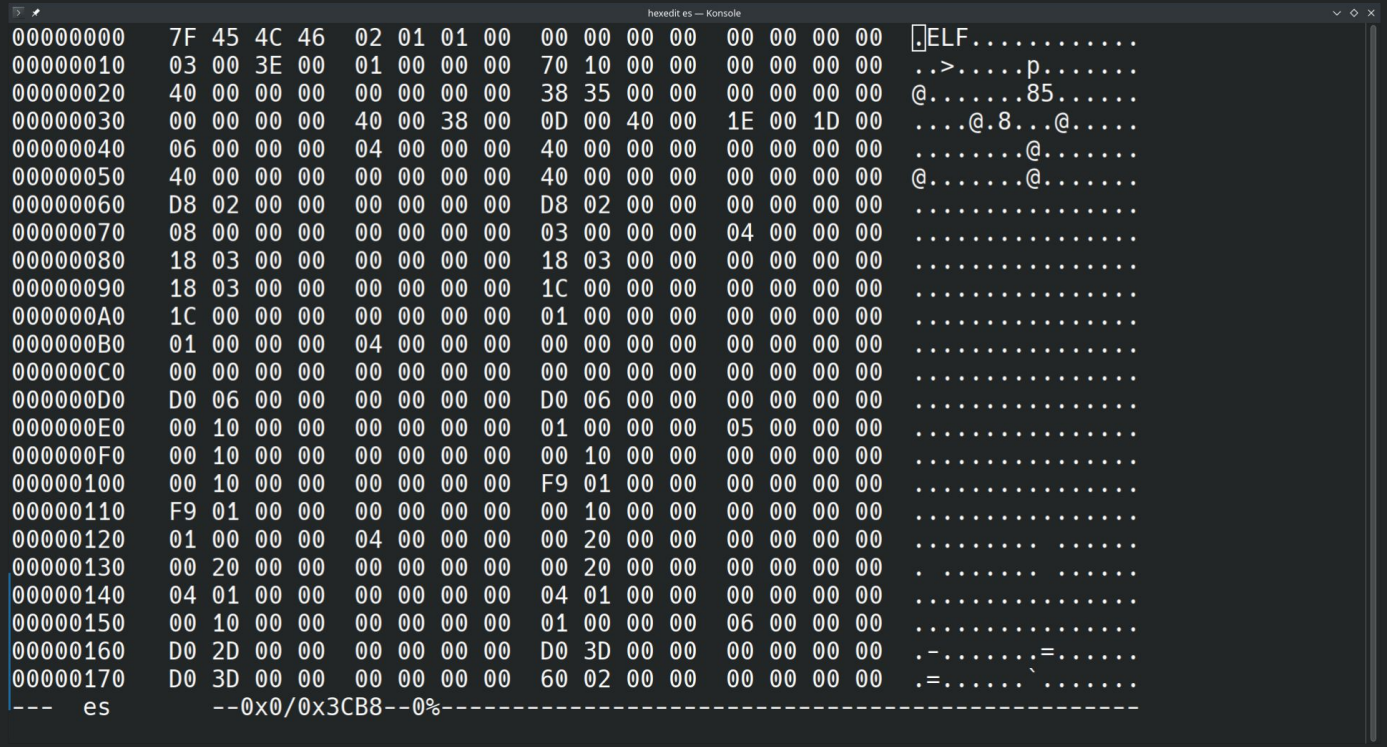
We can, let’s try doing it using different tools!

Binary Patching - Data section - `hexedit`



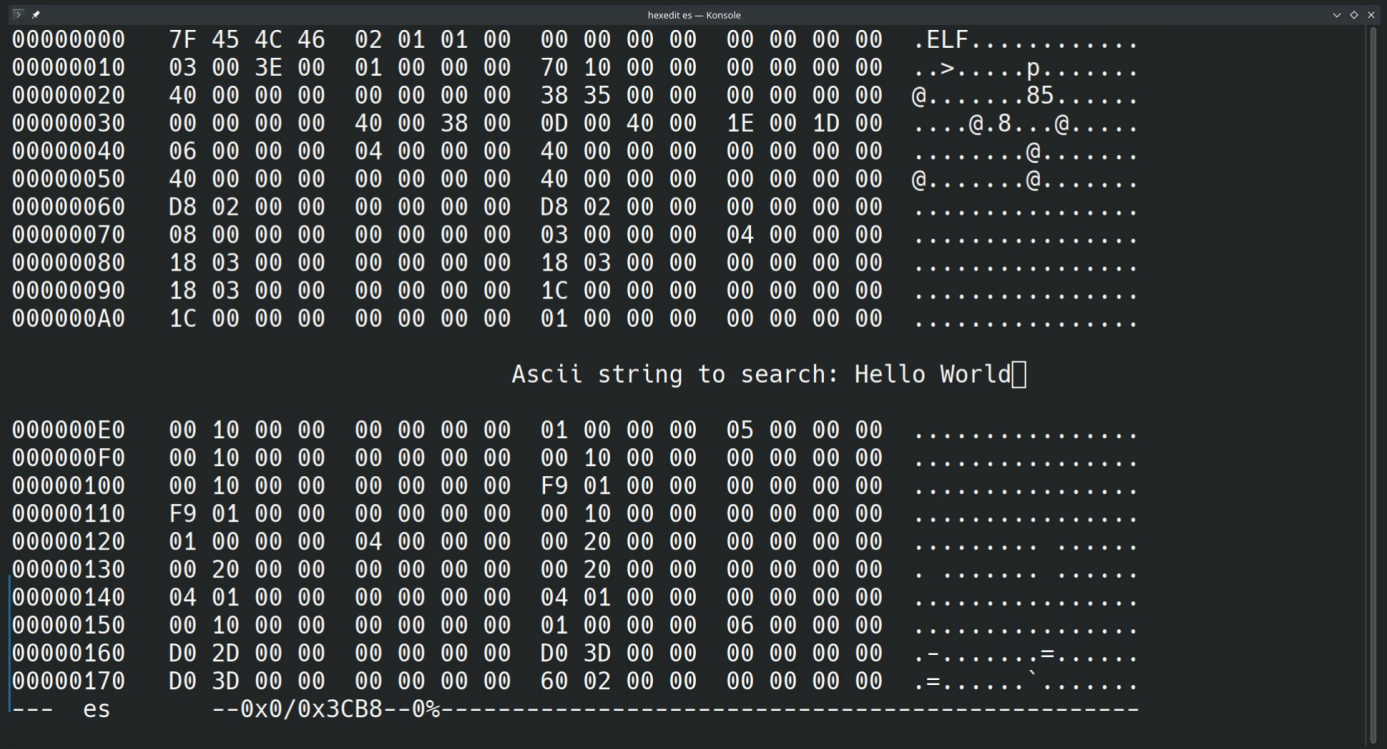
Press **TAB** to switch from **HEX** to **ASCII** editor

Binary Patching - Data section - `hexedit`

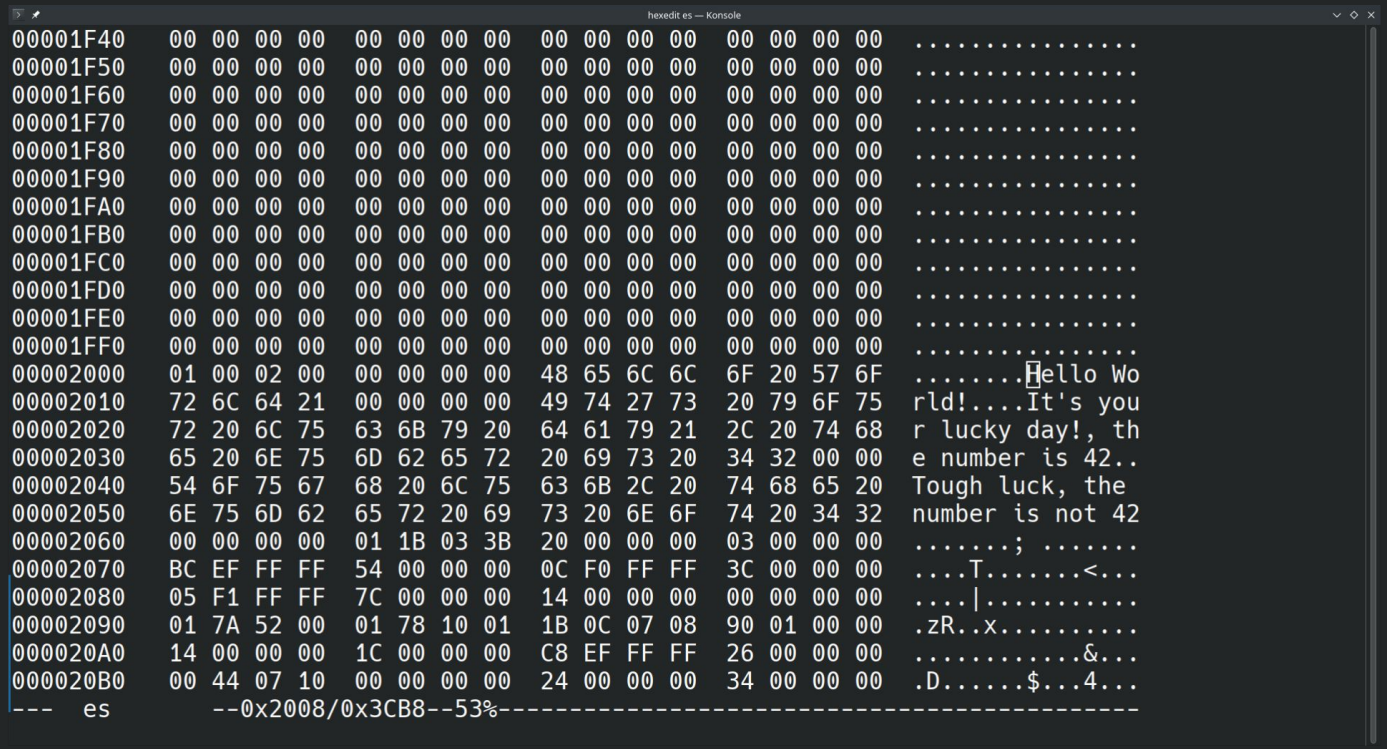


Press / to search for the string **`Hello World`**

Binary Patching - Data section - `hexedit`

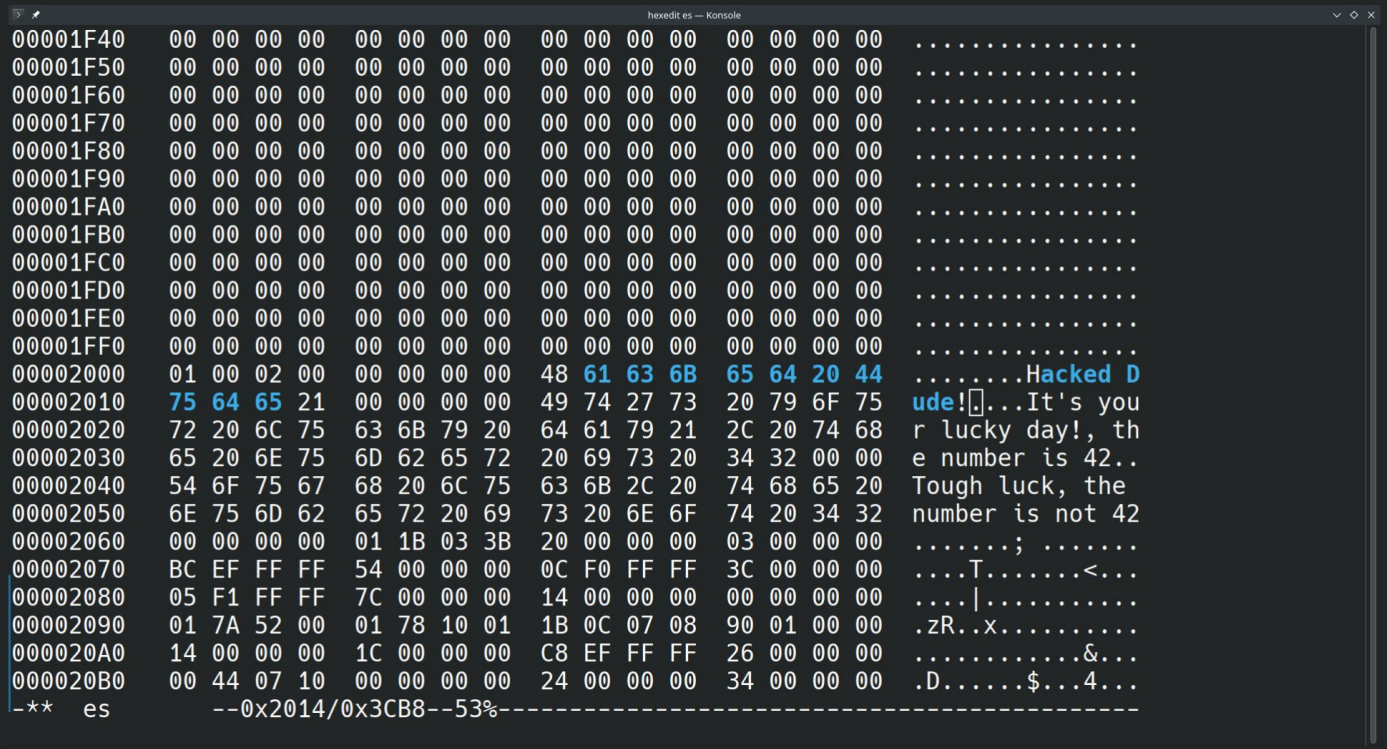


Binary Patching - Data section - `hexedit`



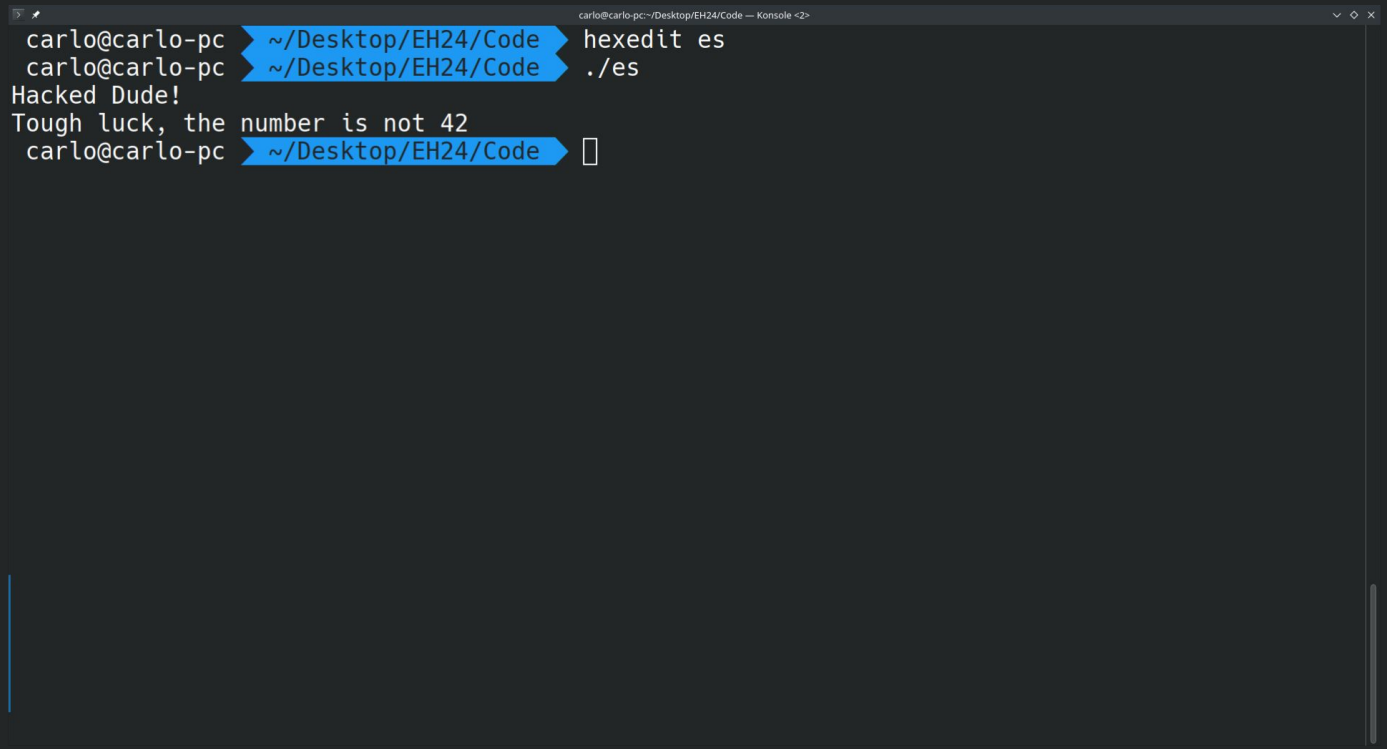
Start overwriting the string with something else

Binary Patching - Data section - `hexedit`



Press **Ctrl-X** to save and exit

Binary Patching - Data section - `hexedit`

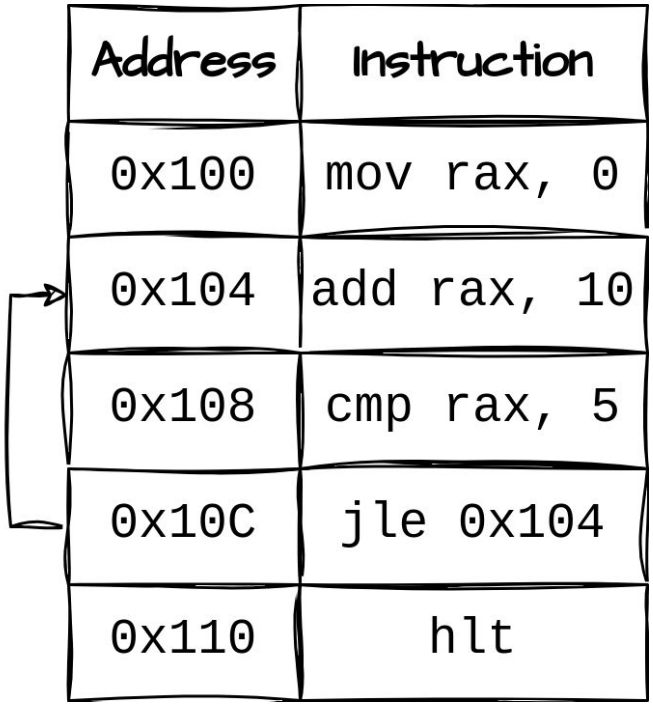


Done!

Binary Patching - Code section

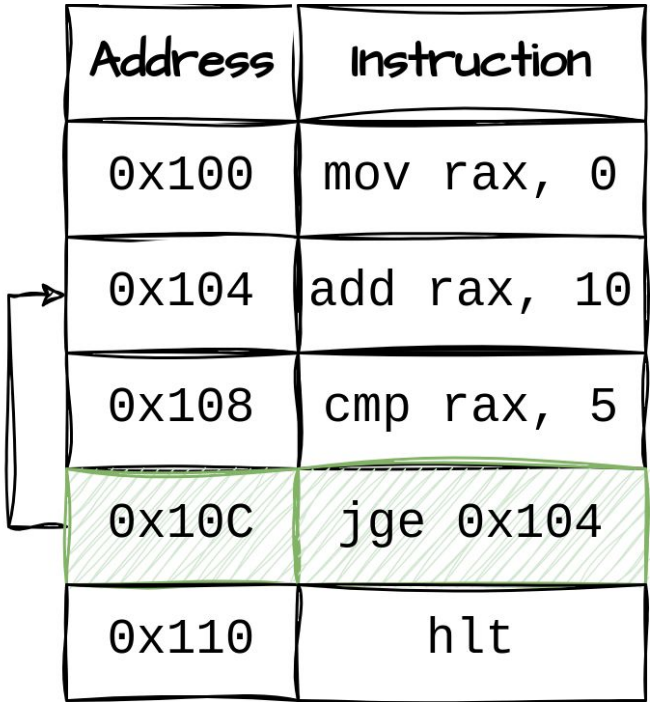
When we want to **change the code** of a binary file, we **can't add or remove instructions**, or more generally, we **can't modify the structure** of the code, but why?

Let's take this simple program as an example



Binary Patching - Code section

We can **REPLACE** instructions

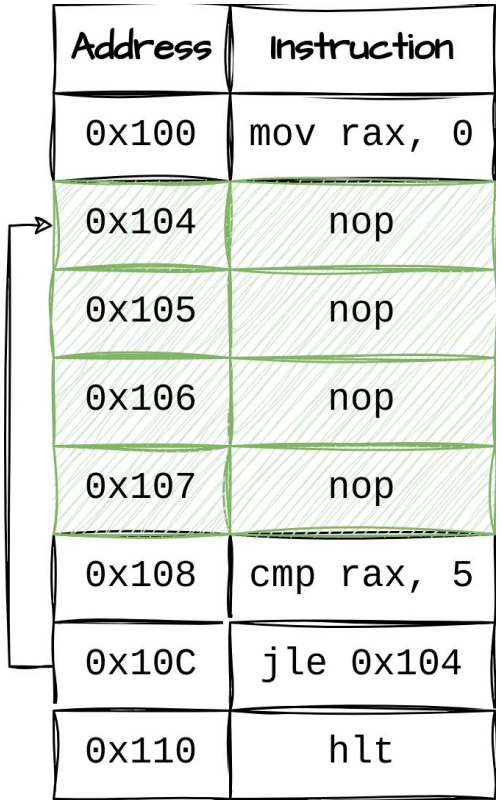


Binary Patching - Code section

We **CAN'T REMOVE** instructions,
but we can still replace then with the
NOP instruction

The **NOP** (**No OPeration**) instruction
simply jumps to the next one

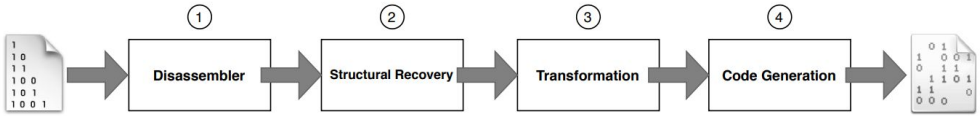
Note: the NOP instruction is 1-byte long,
while the ADD instruction is 4-bytes long,
hence we have to replace it with 4 NOP
instructions



Binary Patching - Code section

We **CAN'T ADD** instructions,
because that way we are modifying the structure of the program, breaking any reference that the program contains (e.g. jumps, calls, symbols, …)

Modifying the structure of a binary program is possible, but it's no easy task [1]



Address	Instruction
0x100	mov rax, 0
0x104	mov rbx, 0
0x108	mov rcx, 0
0x10C	add rax, 10
0x110	cmp rax, 5
0x114	jle 0x104
0x118	hlt

[1]: https://arxiv.org/pdf/2011.14067.pdf

Binary Patching - Code section - Ghidra

```
int main(void)

{
  int random_number;
  time_t tVar1;

  puts("Hello World!");
  tVar1 = time((time_t *)0x0);
  srand((uint)tVar1);
  random_number = rand();
  if (random_number % 1000 == 42) {
    puts("It\'s your lucky day!, the number is 42");
  }
  else {
    puts("Tough luck, the number is not 42");
  }
  return 0;
}
```

The decompiled version of the program we patched before looks like this.

We want to make the program print the string that would otherwise be printed once every thousand times.

Binary Patching - Code section - Ghidra

```
int main(void)

{
  int random_number;
  time_t tVar1;

  puts("Hello World!");
  tVar1 = time((time_t *)0x0);
  srand((uint)tVar1);
  random_number = rand(); = 42
  if (random_number % 1000 == 42) {
    puts("It\'s your lucky day!, the number is 42");
  }
  else {
    puts("Tough luck, the number is not 42");
  }
  return 0;
}
```

We could, for example, **skip the call** to the **rand** function and **replace it** with an **immediate assignment**.

Or, we could play with the **if** statement, modifying the **comparison or the conditional jumps**.

Let's look at the disassembly!

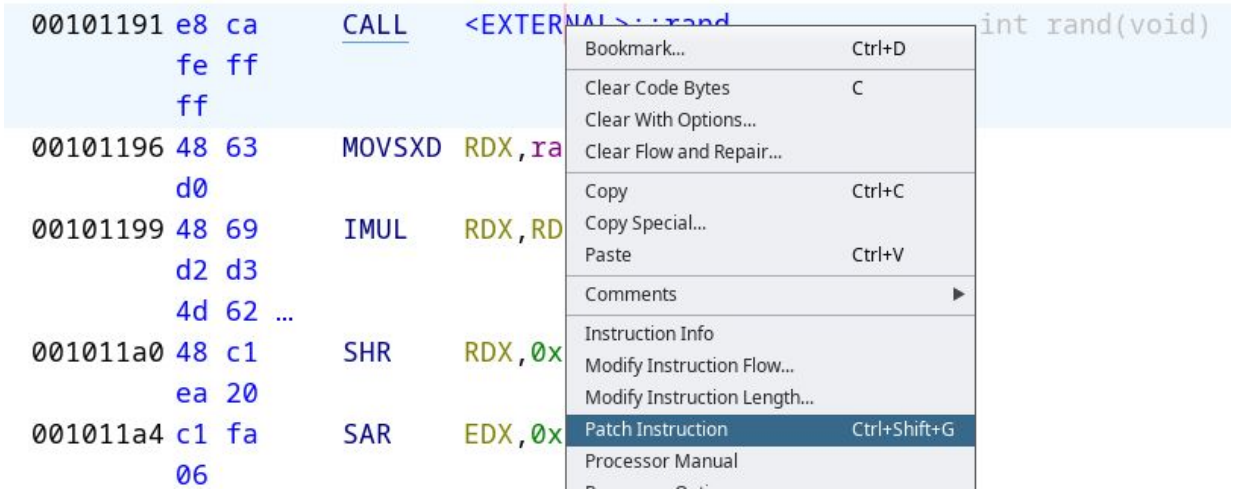
Binary Patching - Code section - Ghidra

```
00101191 e8 ca fe ff ff     CALL    <EXTERNAL>::rand            int rand(void)
00101196 48 63 d0           MOVSXD  RDX,random_number
00101199 48 69 d2 d3 4d 62 … IMUL   RDX,RDX,0x10624dd3
001011a0 48 c1 ea 20        SHR     RDX,0x20
001011a4 c1 fa 06           SAR     EDX,0x6
```

The instruction **`CALL <EXTERNAL>::rand`** calls the library function which places the result in the **`rax`** register [1], the other instructions implement the modulo operation, we don't care about them.

[1]: https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions#CDECL

Binary Patching - Code section - Ghidra



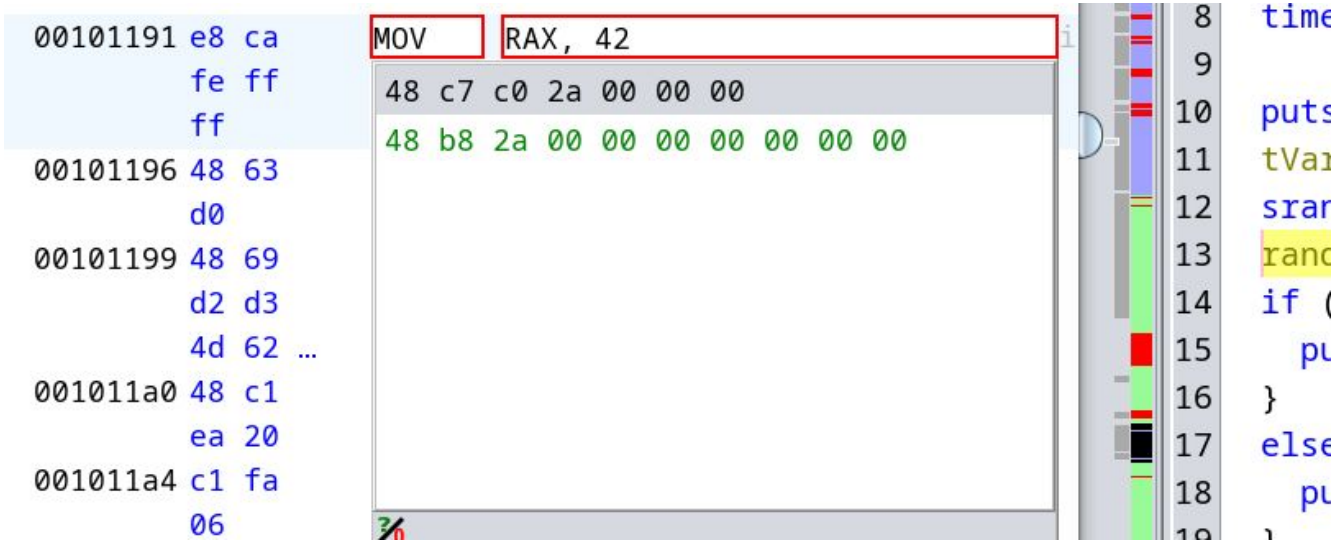
To patch the instruction, right click on it and select Patch Instruction

Binary Patching - Code section - Ghidra

```
00101191 e8 ca fe ff ff    CALL    0x00101060                    int rand(void)
00101196 48 63 d0          MOVSXD  RDX,random_number
00101199 48 69 d2 d3 4d 62 …  IMUL  RDX,RDX,0x10624dd3
001011a0 48 c1 ea 20       SHR     RDX,0x20
001011a4 c1 fa 06          SAR     EDX,0x6
```

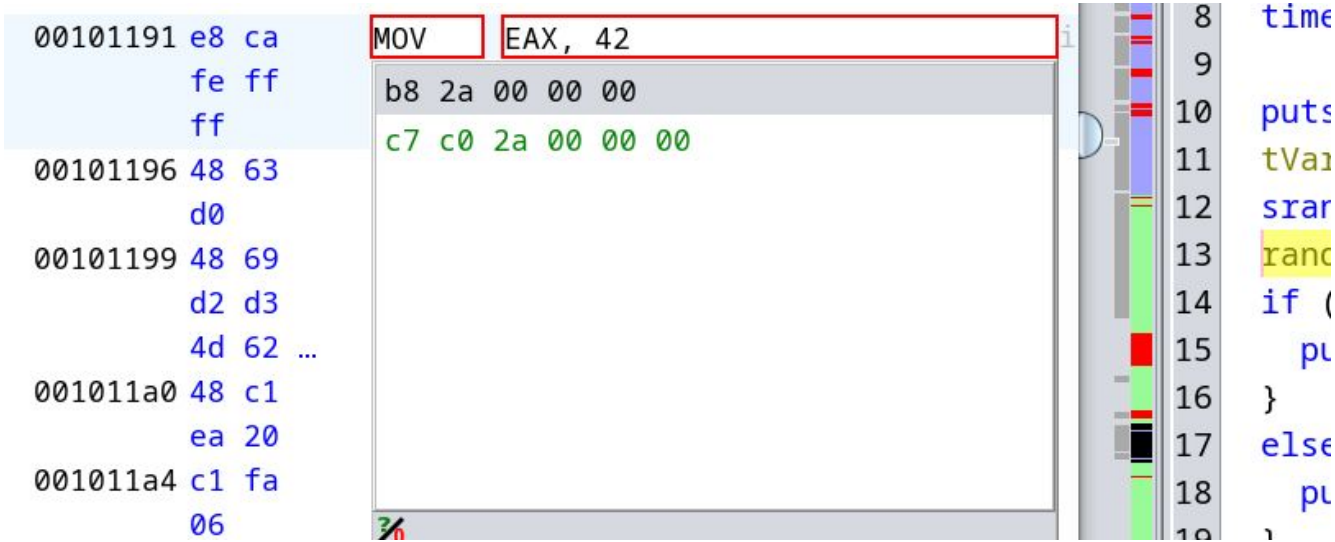
In this view, you can start typing for the new instruction

Binary Patching - Code section - Ghidra



Here you see a list of opcodes you can choose from, **be aware of the size**!
If you choose an instruction which is **bigger** than the one you are overwriting, you will also overwrite (part of) the next instruction(s)

Binary Patching - Code section - Ghidra



The 32 bit version matches the size of the instruction we are replacing (5 bytes), let's use this one instead.

Binary Patching - Code section - Ghidra

```
int main(void)

{
  time_t tVar1;

  puts("Hello World!");
  tVar1 = time((time_t *)0x0);
  srand((uint)tVar1);
  puts("It\'s your lucky day!, the number is 42");
  return 0;
}
```

This is how the decompiled code gets updated.

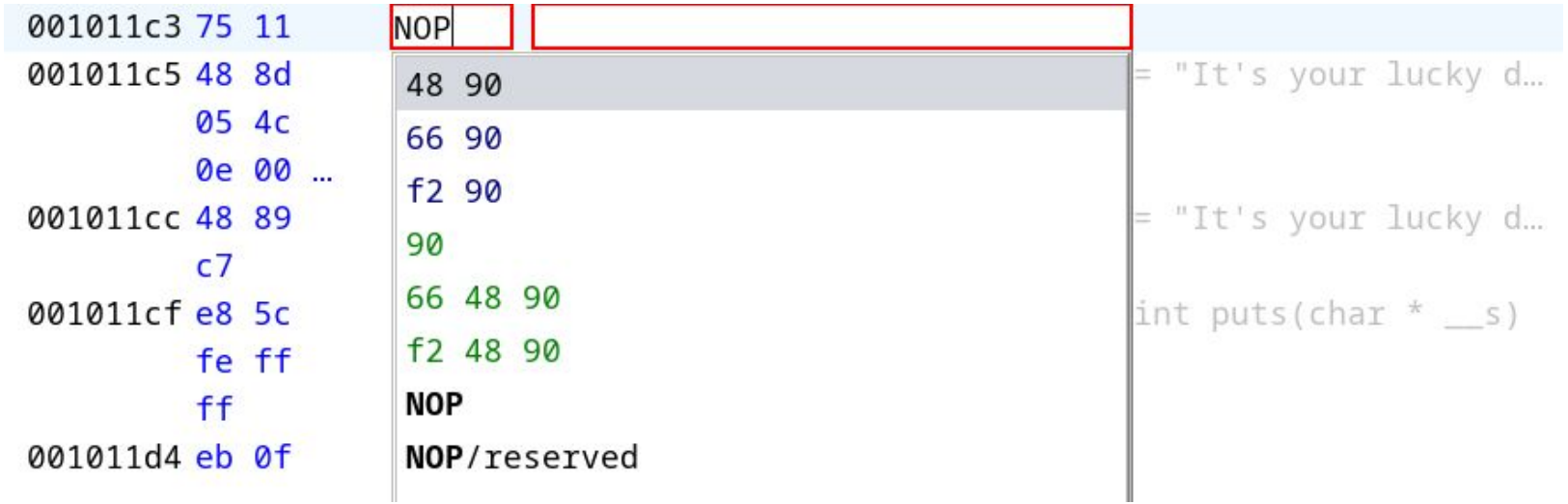
*NOTE: we **haven't removed the missing code** here, but Ghidra understands that it **has become unreachable**, and thus it removes it from the decompiled code.*

Binary Patching - Code section - Ghidra


```
001011c3 75 11      JNZ     LAB_001011d6
001011c5 48 8d      LEA     random_number,[s_It's_your_lu… = "It's your lucky d…
         05 4c
         0e 00 …
001011cc 48 89      MOV     RDI=>s_It's_your_lucky_day!,_… = "It's your lucky d…
         c7
001011cf e8 5c      CALL    <EXTERNAL>::puts                int puts(char * __s)
         fe ff
         ff
001011d4 eb 0f      JMP     LAB_001011e5
```

Another way to tackle this is to mess with the conditional jumps, we could for example **remove** the **`JNZ`** instruction.

Binary Patching - Code section - Ghidra



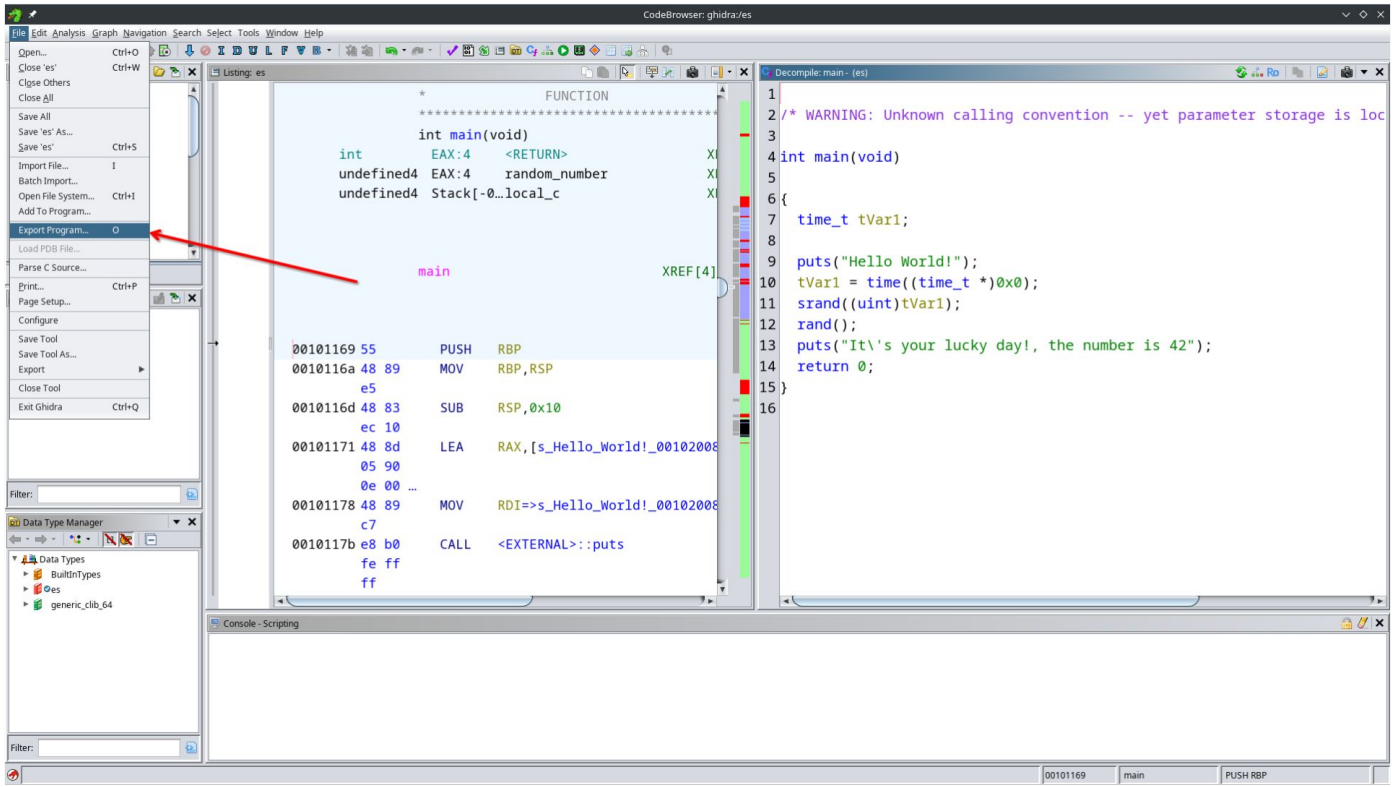
There are **different opcodes** that correspond to a **NOP** instruction, you should choose one that **matches the size** of the replaced instruction (in this case **2 bytes**, you can also replace it with **two 1-byte long NOP** instructions)

Binary Patching - Code section - Ghidra

```
int main(void)

{
  time_t tVar1;

  puts("Hello World!");
  tVar1 = time((time_t *)0x0);
  srand((uint)tVar1);
  rand();
  puts("It\'s your lucky day!, the number is 42");
  return 0;
}
```

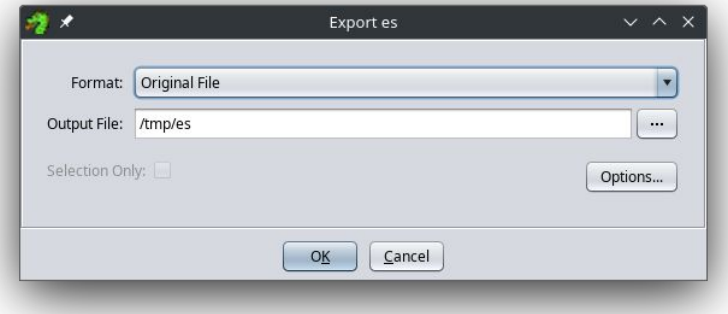
The updated decompiled code is pretty similar to the previous one, except for the **rand** call.

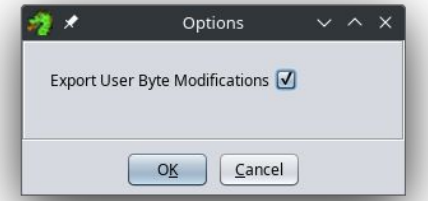
Binary Patching - Code section - Ghidra



You can export the patched binary by selecting:
`File → Export program…`

Binary Patching - Code section - Ghidra





Select **Original File** as format and check the **Export User Byte Modifications** box.

Dynamic Analysis

Carlo Ramponi

Dynamic Analysis

What we've done so far is called **Static Analysis**, i.e. a method that consists of **examining the source** code/compiled code **without having to execute the program**

Dynamic Analysis, instead, is the process of testing and evaluating a program, **while software is running**.

The two techniques can be combined to gain the best knowledge out the program under analysis.

ltrace

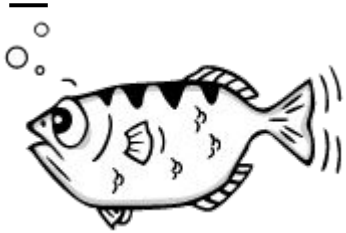
Display the calls a userspace application makes to shared libraries

```
> ltrace ./baby
puts("Insert key: ")                                = 13
fgets("test\n", 20, 0x7fce3f83d8c0)                 = 0x7ffeff909020
strcmp("test\n", "abcde122313\n")                   = 19
puts("Try again later."Try again later.")           = 17
+++ exited (status 0) +++
```

strace

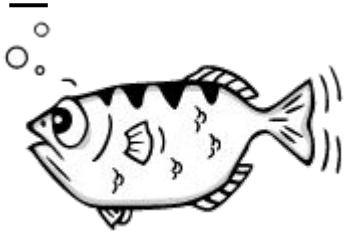
Monitor the interactions between processes and the Linux kernel

```
> strace ./baby
[ a lot of things related to the process' spawn ]
write(1, "Insert key: \n", 13)                          = 13
read(0, "test\n", 1024)                                 = 5
write(1, "Try again later.\n", 17)                      = 17
exit_group(0)                                           = ?
+++ exited with 0 +++
```



GDB: the GNU debugger

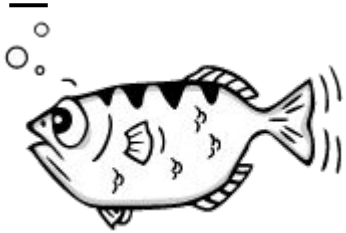
- First release: 1986
- But still a valid product
- Supports almost all architectures (for sure all the ones we will see)
- Advanced features offered by extensions (peda, gef,pwndbg…)



GDB: the GNU debugger

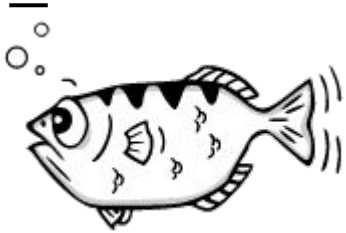
- GDB is preinstalled on the majority of unix-like OSs
- But if it is not on yours, let's install it!

```
# Is it installed?
> gdb --version

# Ubuntu (debian-based)
> sudo apt install gdb

# Arch-based
> sudo pacman -S gdb

# Others?
> Google it!
```



GEF - GDB Extended Features

- An extension for GDB that will make our life easier!
- Key features:
  - Enhance the display of gdb: colorize and display disassembly codes, registers, memory information during debugging.
  - Add commands to support debugging and exploit development

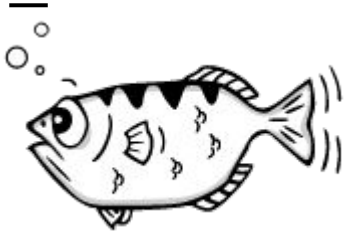


GEF - GDB Extended Features

Installation:

```
# We trust them not to backdoor our system
bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

As easy as that!

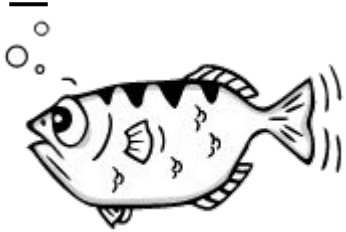


GEF - GDB Extended Features

Uninstall:

- edit ~/.gdbinit

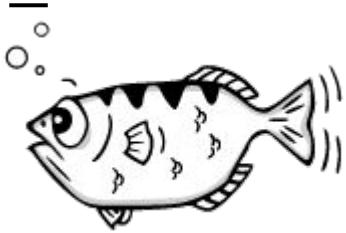
As easy as that!



GEF - GDB Extended Features

Without GEF:

```
(gdb) b main
Breakpoint 1 at 0x40113a
(gdb) r
Starting program: /tmp/bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, 0x000000000040113a in main ()
(gdb)
```



GEF - GDB Extended Features

With GEF:

The exact same situation doesn’t fit here, so see next slide lol

```
(gdb) b main
Breakpoint 1 at 0x40113a
(gdb) r
Starting program: /tmp/bin
[ Legend: Modified register | Code | Heap | Stack | String ]
```

registers

```
$rax   : 0x0000000000401136  →  <main+0> push rbp
$rbx   : 0x00007fffffffd558  →  0x00007fffffffda1a  →  "/tmp/bin"
$rcx   : 0x0000000000403df0  →  0x0000000000401100  →   endbr64
$rdx   : 0x00007fffffffd568  →  0x00007fffffffda23  →  "CASROOT=/usr"
$rsp   : 0x00007fffffffd440  →  0x0000000000000001
$rbp   : 0x00007fffffffd440  →  0x0000000000000001
$rsi   : 0x00007fffffffd558  →  0x00007fffffffda1a  →  "/tmp/bin"
$rdi   : 0x1
$rip   : 0x000000000040113a  →  <main+4> sub rsp, 0x40
$r8    : 0x0
$r9    : 0x00007ffff7fcce20  →   endbr64
$r10   : 0x00007fffffffd170  →  0x0000000000800000
$r11   : 0x206
$r12   : 0x0
$r13   : 0x00007fffffffd568  →  0x00007fffffffda23  →  "CASROOT=/usr"
$r14   : 0x00007ffff7ffd000  →  0x00007ffff7ffe2e0  →  0x0000000000000000
$r15   : 0x0000000000403df0  →  0x0000000000401100  →   endbr64
$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

```
0x00007fffffffd440|+0x0000: 0x0000000000000001   ← $rsp, $rbp
0x00007fffffffd448|+0x0008: 0x00007ffff7dc6cd0  →   mov edi, eax
0x00007fffffffd450|+0x0010: 0x00007fffffffd540  →  0x00007fffffffd548  →  0x0000000000000038 ("8"?)
0x00007fffffffd458|+0x0018: 0x0000000000401136  →  <main+0> push rbp
0x00007fffffffd460|+0x0020: 0x0000000100400040 ("@"?)
0x00007fffffffd468|+0x0028: 0x00007fffffffd558  →  0x00007fffffffda1a  →  "/tmp/bin"
0x00007fffffffd470|+0x0030: 0x00007fffffffd558  →  0x00007fffffffda1a  →  "/tmp/bin"
0x00007fffffffd478|+0x0038: 0xbb06b5368b9c2b52
```

code:x86:64

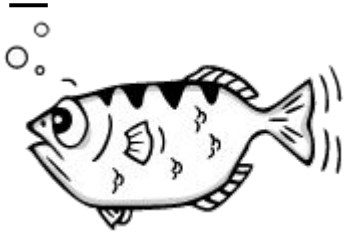
```
    0x401134                  jmp    0x4010c0
    0x401136 <main+0>         push   rbp
    0x401137 <main+1>         mov    rbp, rsp
 →  0x40113a <main+4>         sub    rsp, 0x40
    0x40113e <main+8>         mov    rax, QWORD PTR [rip+0x2edb]        # 0x404020 <stdout@GLIBC_2.2.5>
    0x401145 <main+15>        mov    esi, 0x0
    0x40114a <main+20>        mov    rdi, rax
    0x40114d <main+23>        call   0x401030 <setbuf@plt>
    0x401152 <main+28>        mov    rax, QWORD PTR [rip+0x2ed7]        # 0x404030 <stdin@GLIBC_2.2.5>
```

threads

```
[#0] Id 1, Name: "bin", stopped 0x40113a in main (), reason: BREAKPOINT
```

trace

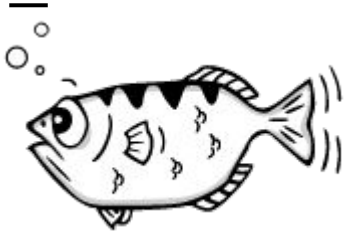
```
[#0] 0x40113a → main()
```



GEF - GDB Extended Features

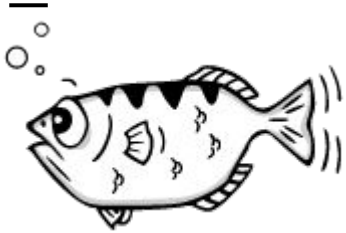
Not only that, GEF:

- Shows parameters a function is being called with
- Shows the canary (with `canary`),
- Pattern creation: `pattern create <n>`, `pattern search $rbp`
- Uses Intel syntax by default 🎉
- Much more



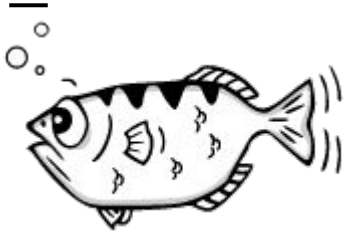
First steps with GDB - control flow

```
> gdb ./hello # debug a program called “hello”
gdb> break main # (b) breakpoint at start of function main
gdb> break *addr # (b) breakpoint at address “addr” notice the “*”
gdb> info break # (i) show information on breakpoints set
gdb> break *main+54 # break at the offset 54 of main
gdb> del breakpoint 1 # (d) delete breakpoint 1
gdb> run # (r) start the program
```



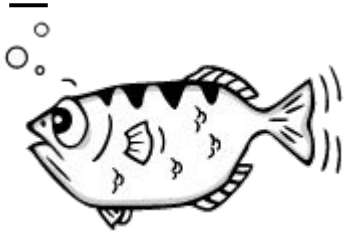
First steps with GDB - control flow

```
gdb> continue # (c) continue execution
gdb> ni # (si) next instruction
gdb> finish # execute until the end of the current function
gdb> call (return type) main(arg1, …) # call function main
gdb> jump *addr # (j) jump to address “addr”
```



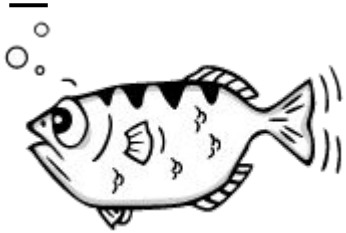
First steps with GDB - registers

```
gdb> info registers # display the value of all registers
gdb> set $rax=0xdeadbeef # edit a register value
gdb> jump *($rip+0x10) # jump to address pointed by rip + 0x10
```



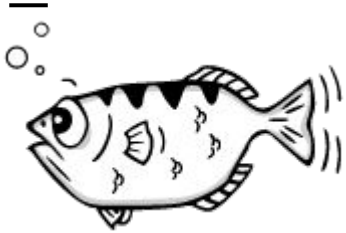
First steps with GDB - disassembling

```
gdb> disassemble main # (d) disassemble the main function
gdb> disassemble $rip # (d) disassemble the function where $rip is in
# if gdb doesn't find a function to disassemble you can:
gdb> x/10i $rip # read memory at $rip and print 10 instructions
```



First steps with GDB - memory

```
gdb> x addr # display the value of memory at address “addr”
gdb> x *addr # memory at address pointed by “addr”
gdb> x $rsp # memory at address pointed by register rsp
gdb> x $rsp - 10 # memory at address pointed by register rsp - 10
gdb> x main # memory at address pointed by symbol “main”
```



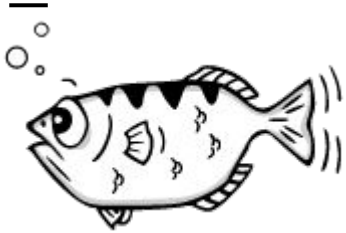
First steps with GDB - memory (1)

Every “x” from the previous slide can be replaced with: `x/nfu`

Where:

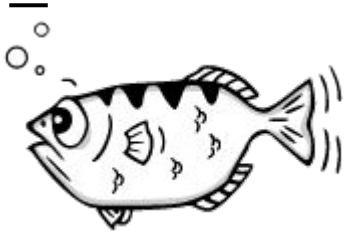
- n is the “the repeat count” (how many values to display)
- f is the format, one of [x, d, u, o, t, a, c, f, s]
  - ‘i’ (for machine instructions) and ‘m’ (for displaying memory tags)
- u is the unit size
  - b (bytes), h (halfwords), w (words), g (giant words)

Default is `x/1xw` (one hexadecimal word)



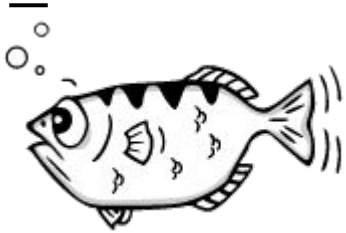
First steps with GDB - memory (2)

```
gdb> find “flag” # search for the string flag in memory
gdb> watch *addr # break when memory cell “addr” is written
gdb> rwatch *addr # break when memory cell “addr” is read
gdb> awatch *addr # [...] memory cell “addr” is written or read
# dump memory from “start” to “end” to binary file “filename”:
gdb> dump binary memory filename start end
gdb> set { unsigned int } addr = 12 # *addr = (unsigned int)12
gdb> set {char [12]} adr = "Hello there" # addr = “Hello there”
```



First steps with GDB - MISC

```
gdb> r <<< $(python3 -c ‘print(“A”*1024)’) # Run the program by
                    passing the output of the command to stdin
gdb> r < payload # Pass the content of a file to stdin
gdb> info functions # Show functions
gdb> info frame # Get info on the current stack frame
gdb> p main # Get address of the main function
```



First steps with GDB - all the rest

For all the other amazing functions of GDB (a lot) head to the GDB documentation:

https://www.sourceware.org/gdb/documentation/

Anti-Reversing & Anti-Debugging Techniques

Carlo Ramponi

Anti reversing

Why? You want to protect your program from being reverse engineered

- There's no way to completely hide what a program does
  - The processor must receive the instructions to be execute after all
- You can still make the life of the reverse engineer harder, though

Anti reversing - stripping

The easiest and fastest way to **harden the reverse engineering process** on your software is to **strip** all symbols such as function names, global variable names, …

Still, **shared object symbols** have to be **included** (the dynamic linker needs to know you want to call a shared function, e.g. **`printf`**)

→ You can statically link the shared libraries to also strip those symbols.

With **`gcc`**:

```
$ gcc -s -l:library_file ...
```

Anti reversing - anti disassembly

There aren't many disassemblers out there!
→ Defeating them with specific techniques is feasible

In particular, you can **exploit bugs or limitations** in the most used disassemblers (Ghidra, IDA, …) by **inserting bytes** which will **not interfere with the execution** but will **trick the disassemblers**.

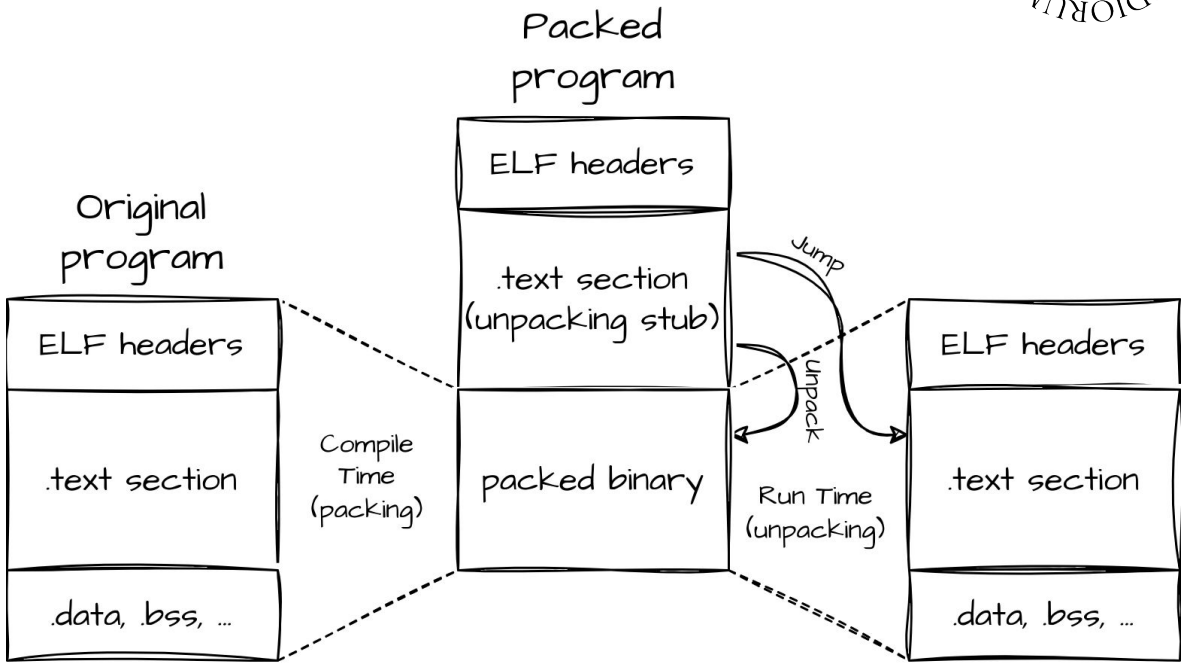
More at http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/anti-disas.pdf

Anti reversing - packing

Binary packing is the technique of **compressing executable files** to **obscure their content**, making it harder for security applications to detect or analyze them

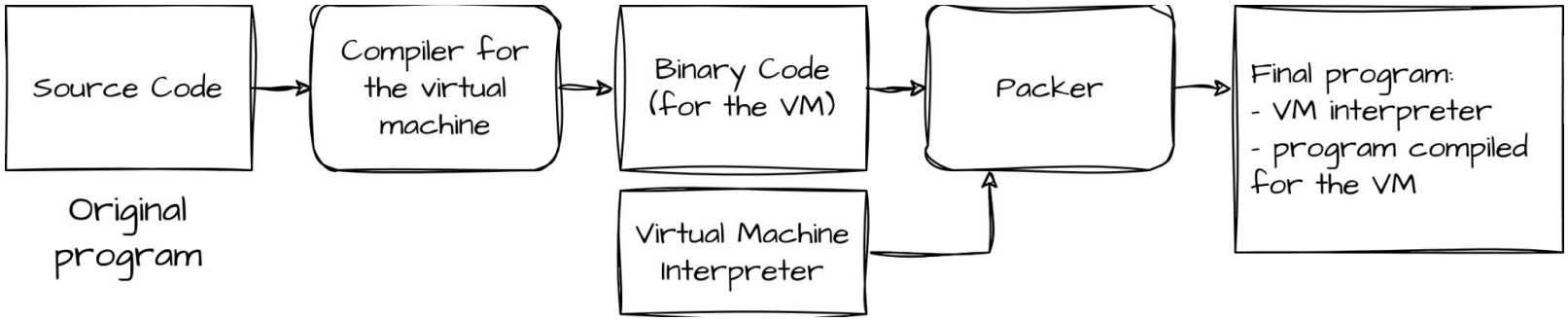
Can include **encryption**

Easily **defeated** by manually unpacking or by **debugging**



Anti reversing - Virtual Machine

A **Virtual Machine-based obfuscator** is a special kind of **binary packing**, where the stub doesn't only unpack and execute the original program, but instead it is an **interpreter** for a **newly-invented ISA** in which the **original program is compiled**



To **reverse engineer** such a program you'll have to:

1. **Reverse engineer the VM-interpreter** to understand how the VM works
2. **Implement a disassembler** for the VM language
3. **Reverse engineer the original program** (in the VM language)

Anti debugging

Why? Some anti-reversing techniques can be bypassed by dynamically analyzing the program, thus you want to protect your program from being debugged.

e.g. If the program is a Malware, it wants to know whether is being analyzed and change its behavior according to the environment

Anti debugging - `ptrace`


```
ptrace(2)                        System Calls Manual                        ptrace(2)

NAME
       ptrace - process trace

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/ptrace.h>

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

DESCRIPTION
       The  ptrace()  system call provides a means by which one process (the "tracer") may observe and con-
       trol the execution of another process (the "tracee"), and examine and change the tracee's memory and
       registers.  It is primarily used to implement breakpoint debugging and system call tracing.
```

- allows **one** process to control another one
- any process can be traced by **only one process** at a time
- returns an error if tracing request fails

Anti debugging - `ptrace`


```
ptrace(2)                        System Calls Manual                       ptrace(2)

NAME
       ptrace - process trace

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/ptrace.h>

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

DESCRIPTION
       The  ptrace()  system call provides a means by which one process (the "tracer") may observe and con-
       trol the execution of another process (the "tracee"), and examine and change the tracee's memory and
       registers.  It is primarily used to implement breakpoint debugging and system call tracing.
```

QUESTION: Given this syscall, is there a way to use it in order to check if the current process is being debugged?

Anti debugging - self tracing

A process can also trace itself, calling **`ptrace`** with the **`PTRACE_TRACEME`** macro,

- if the call **succeeds** → no one is tracing us (**remember**: only one tracer at a time)
- if the call **fails** (returns **`-1`**) → some other process is tracing us (a debugger, perhaps)

```c
#include <stdio.h>
#include <sys/ptrace.h>

int main() {
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1) {
        printf("You are being traced!\n");
        return 1;
    }

    printf("Hello World!\n");
    return 0;
}
```

Anti debugging - self tracing

```
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./ptrace
Hello World!
```

Anti debugging - self tracing


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  ./ptrace
Hello World!
 carlo@carlo-pc  ~/Desktop/EH24/Code  ltrace ./ptrace 2> /dev/null
You are being traced!
```

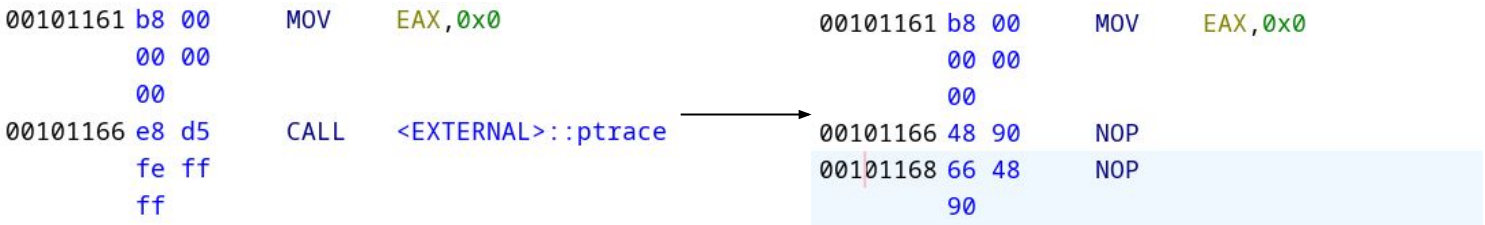
Anti debugging - self tracing


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  gdb ./ptrace
GNU gdb (GDB) 14.2
Copyright (C) 2023 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.
gef➤  r
Starting program: /home/carlo/Desktop/EH24/Code/ptrace
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
You are being traced!
[Inferior 1 (process 86561) exited with code 01]
```

Anti Anti debugging

How to defeat self tracing?

- by **patching** the binary (we know how to do that now!)
- **overwrite** the **`ptrace`** function by **preloading** a custom **shared library**



Shared library preloading

The **dynamic loader** looks for shared libraries in the **default system directories**, but you can also **specify where to look firstly**, that way you can **inject a shared library** before the default ones will be loaded

If a shared library you inject, for example, **defines a libc's function**, your implementation will be chosen instead of the one defined by libc.

This can be achieved by setting the environment variable **`LD_PRELOAD`**

Shared library preloading


```
carlo@carlo-pc ~/Desktop/EH24/Code bat my_puts.c
```

File: my_puts.c

```c
#include <stdio.h>
int puts(const char *str) {
    printf("I'm not printing your garbage, lol!");
    return 0;
}
```

```
carlo@carlo-pc ~/Desktop/EH24/Code gcc -c -fPIC -o my_puts.o my_puts.c
carlo@carlo-pc ~/Desktop/EH24/Code gcc -shared -fPIC -Wl,-soname,my_puts.so -o my_puts.so my_puts.o -lc
carlo@carlo-pc ~/Desktop/EH24/Code bat target.c
```

File: target.c

```c
#include <stdio.h>
int main() {
    puts("Hello World!");
    return 0;
}
```

```
carlo@carlo-pc ~/Desktop/EH24/Code LD_PRELOAD=./my_puts.so ./target
I'm not printing your garbage, lol!%
```

Shared library preloading

```
carlo@carlo-pc ~/Desktop/EH24/Code bat my_puts.c
```

File: **my_puts.c** ← The fake implementation of puts, it needs to match the signature of the original one

```c
#include <stdio.h>
int puts(const char *str) {
    printf("I'm not printing your garbage, lol!");
    return 0;
}
```

Compile the fake function

Generate the shared library file

```
carlo@carlo-pc ~/Desktop/EH24/Code gcc -c -fPIC -o my_puts.o my_puts.c
carlo@carlo-pc ~/Desktop/EH24/Code gcc -shared -fPIC -Wl,-soname,my_puts.so -o my_puts.so my_puts.o -lc
carlo@carlo-pc ~/Desktop/EH24/Code bat target.c
```

File: **target.c** ← The target program, which calls the puts function

```c
#include <stdio.h>
int main() {
    puts("Hello World!");
    return 0;
}
```

Set the environment variable

Run the target program

```
carlo@carlo-pc ~/Desktop/EH24/Code LD_PRELOAD=./my_puts.so ./target
I'm not printing your garbage, lol!%
```

← The fake implementation has been executed instead of the libc one

Shared library preloading

To **defeat** the **self tracing** technique we just need to provide the program with an implementation of **`ptrace`** that **never fails** (i.e. never returns **`-1`**)

```c
long ptrace(int request, int pid, int addr, int data) {
    return 0;
}
```

SMT solvers & Symbolic Execution

NOTE: this is an advanced topic, we won't go into details

Carlo Ramponi

Boolean Satisfiability Problem (SAT)

Given a **boolean formula**, does exist an interpretation that **satisfies** it?

$$p \lor \neg q \Rightarrow \text{SAT}$$
$$p \land \neg p \Rightarrow \text{UNSAT}$$

SAT is **NP-complete**!

Satisfiability Modulo Theory (SMT)

SMT is a **generalization** of a Boolean SAT instance in which various sets of variables are replaced by predicates from a **variety of underlying theories**.

E.g.:

- Linear real arithmetic constraints
- Arrays
- Uninterpreted functions
- Bit vectors

$$3x + 2y - z \geq 4$$

$$read(a, y - 2) = f\ (y - x + 1)$$

Z3 Theorem Prover

Z3 is a theorem prover from Microsoft Research that can **efficiently solve SMT problems**

Name	z3
Released	February, 2007
Size	300K+ loc (core)
License	MIT
Bindings	python, c/c++, . . .

Z3 - Python bindings

Z3 can be used as a standalone executable (using **SMTLIB** syntax) or as a library.
Both the executable and the python bindings can be installed using pip.

```
> pip3 install z3-solver
> from z3 import *
```

Z3 - Example

Is $(p \vee \neg q) \wedge (\neg p \vee q \vee r) \wedge \neg p$ satisfiable?

```python
from z3 import *

p = Bool('p') # Declaring variables
q = Bool('q')
r = Bool('r')
s = Solver() # Instantiating Z3 Solver
# Adding formula constraints to solver
s.add(
    Or(p, Not(q)),
    Or(Not(p), q, r),
    Not(p)
)

print(s.check()) # YES, it is SAT
print(s.model()) # [p = False, q = False, r = False]
```

Satisfiability | Validity

- A **formula is satisfiable** if it has an **interpretation** that makes it **logically true**
- A **formula is valid** if it is **logically true in any interpretation**

→ Proving that a **formula is a tautology** (i.e. valid) is equivalent to proving that its negation is not satisfiable!

Theories

INTEGER NUMBERS

```
x = Int('x')
y = Int('y')
s = Solver()
s.add( 2*x + y == 12 )
s.add( x == 14 )
s.check()
print(s.model())
```

```
[x = 14, y = -16]
```

BIT VECTORS

```
x = BitVec('x', 32)
y = BitVec('y', 32)
s = Solver()
s.add( Extract(31,24, x) == 10 )
s.add( x + y == 23 )
s.check()
print(s.model())
```

```
[y = 4127195159, x = 167772160]
```

Symbolic execution

- **Generalizes testing** by using **unknown symbolic variables** in evaluation
- If an **execution path** depends on a **symbolic value** the evaluator is **forked**
- Used to **check feasibility** of program paths
  - Determined by computing the path condition on the path and checking satisfiability
  - Enabled by advances in SMT solvers

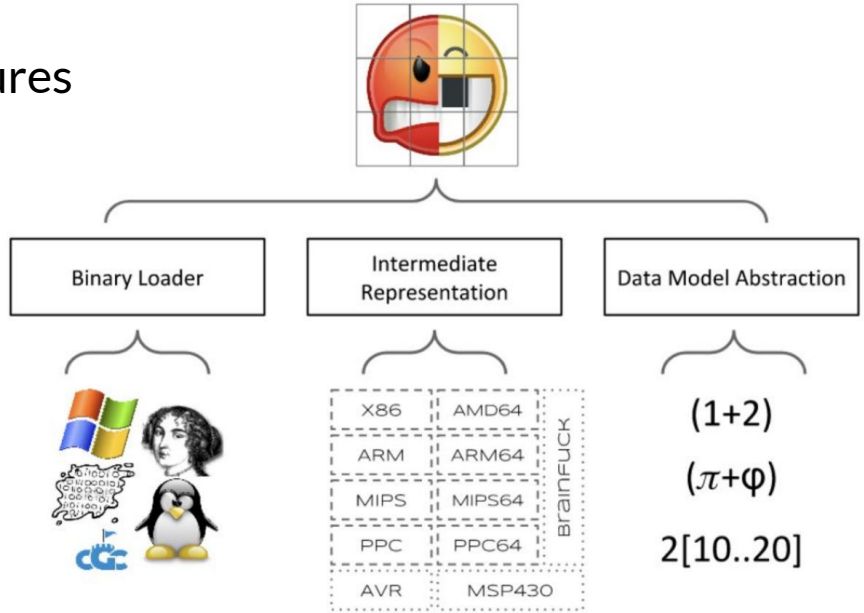
Angr

It's a **framework** which includes:

- An emulator
- A symbolic execution engine
- A collection of static analysis procedures

Integrates with:

- **CLE** (ELF, PE, . . .)
- **PyVEX** (valgrind VEX IR)
- **claripy** (Z3)



Angr - installation

Installation:

```
pip3 install angr --user
```

Usage:

```
import angr
import claripy
```

Angr - SimState

`angr.SimState`: Represents the **state of a program**, including its memory, registers, and so forth.

- `regs`: A convenient view of the state's registers
- `mem`: A convenient view of the state's memory
- `solver` / `se`: The solver engine for this state
- `step()`: Perform a step of symbolic execution using this state

Examples of SimState:

- **`p.factory.blank_state`**: a mostly-uninitialized state
- **`p.factory.entry_state`**: the entry point of the program
- **`p.factory.call_state`**: the start of a given function

Angr - SimulationManager

`angr.SimulationManager`: Simulation managers allow you to wrangle multiple states in a slick way.

`step()`: **Step a stash of states forward** and categorize the successors appropriately

`explore()`: **Tick the current stash forward** looking for condition `find` and avoiding `avoid`

Angr - Hooks

Angr allows to define **hooks** to:

- **Skip** and summarize **code** that causes problems
- Create **simulated procedures** that trade completeness for efficiency to replace binary (or library) defined functions
- **Instrument execution** or modify control flow

`angr.Project.hook`: Hook a section of code with a custom function.

```
p.hook(ADDR, hook=HOOKFN)        @p.hook(ADDR)
```

`angr.Project.hook_symbol`: Looks up the address of a given symbol and hooks that address

Angr - SimProcedures

`angr.SimProcedure`: A SimProcedure is a wonderful object which describes a procedure to run on a state.

- `run()`: Provides the procedure implementation
- `state`: SimState object that the procedure can mutate

Angr - Example

Fairlight - SecurityFest 2016

A simple reverse challenge that takes a **key** as a **command line argument** and checks it against **14 checks**

Reversing all those checks is tedious and cumbersome

Goal: **solve it using angr** without reversing any of the checks

Angr - Example

Fairlight - SecurityFest 2016

- The main idea is that we **want to reach a point of execution** where we have passed all the checks
  - This point can be the first instruction executed after the last check has been correctly passed
- We want to **avoid failing at each check** which can be seen as not failing only the last check
  - The **`to_avoid`** address is the first instruction which gets executed if we've failed the check
- The symbolic variable is the key, passed as a CLI argument
- Let **`angr`** do the rest

Angr - Performances and common problems

- Path explosion
- Loops
- Completeness and coverage
- Limits of SMT Solving

Angr - Example

```
1   import angr
2   import claripy
3
4   proj = angr.Project('./fairlight',
5                       load_options={"auto_load_libs": False})
6
7   # 0xE is the length of the key, each byte is 8 bits
8   argv1 = claripy.BVS("argv1", 0xE * 8)
9
10  initial_state = proj.factory.entry_state(
11      args=["./fairlight", argv1])
12
13  sm = proj.factory.simulation_manager(initial_state)
14  sm.explore(find=0x4018f7, avoid=0x4018f9)
15  found = sm.found[0]
16  print(found.solver.eval(argv1, cast_to=bytes))
```


That's All
Folks