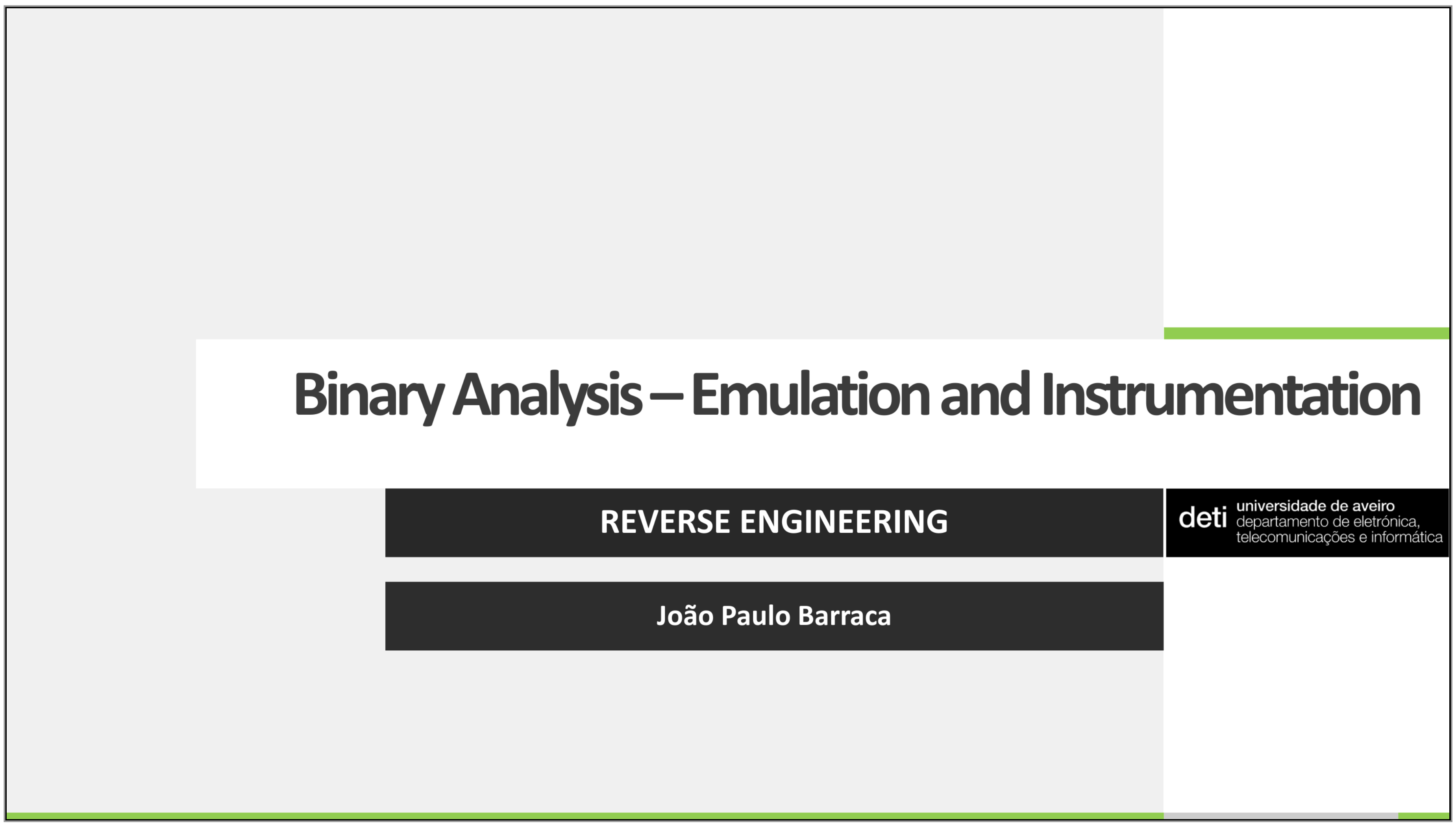
Binary Emulation and Instrumentation

Lecture Notes

Analyzing Binary Executable with focus on dynamic analysis mechanisms such as tracing, debugging, emulation and instrumentation.



Download here

Practical Tasks

Exercise 1

Tracers are tools allowing the analysis of the dynamic behavior of an application, in relation to the actions it takes with an operating system or devices. It uses hooks to hardware, or the operating system, and can register the interactions. It follows a black box approach, in the sense that there is no visibility over the code itself, only the interactions.

Simple tracing application that can be used are strace and ltrace. strace will trace over system calls, while ltrace will trace over library calls (and system calls with some additional switches). As all interactions an application has go through the kernel, and most applications do not have full implementation of all functions, but rely on external libraries, these tools provide a simple, yet effective, method to understand the interactions with the outside (files access, communication with external systems, etc…).

Take in consideration the following snippet. Compile it with `gcc -o ex1 ex1.c`.

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char** argv){
    printf("Hello ");
    fflush(stdout);
    sleep(1);
    printf("World\n");
    return 0;
}
```

Use these tools to analyze its behavior. Correlate what you obtain from the tracers, with the functions called in the program. For ltrace, the `-CifrS` options may be interesting.

From the documentation:

- `-C`: Decode (demangle) low-level symbol names into user-level names. Besides removing any initial underscore prefix used by the system, this makes C++ function names readable.
- `-i`: Print the instruction pointer at the time of the library call.
- `-f`: Trace child processes as they are created by currently traced processes as a result of the fork(2) or clone(2) system calls. The new process is attached immediately.
- `-r`: Print a relative timestamp with each line of the trace. This records the time difference between the beginning of successive lines.
- `-S`: Display system calls as well as library calls

Exercise 2

Most reversing operations will use existing debuggers, which offer a rich set of functionality and related tools. However, a basic debugger is not so complex to be implemented, and can provide insight regarding how applications can be analyzed. Custom debuggers may also be required for applications employing anti-debugging techniques.

The following code sample implements a simple `PTRACE` based debugger for tracing purposes. The code source originates from Eli Bendersky excellent debugging guide and was slightly adapted to our case.

In order to debug processes, this program will `fork` a copy of itself, which will run the debugged application as a child process. The parent will use `PTRACE` to effectively trace over all instructions.

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <signal.h>
#include <syscall.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/reg.h>
#include <sys/user.h>
#include <unistd.h>
#include <errno.h>

void procmsg(const char* format, ...) {
    va_list ap;
    fprintf(stdout, "[%d] ", getpid());
    va_start(ap, format);
    vfprintf(stdout, format, ap);
    va_end(ap);
}


void run_target(const char* programname) {
    procmsg("target started. will run '%s'\n", programname);

    /* Allow tracing of this process */
    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
        perror("ptrace");
        return;
    }

    /* Replace this process's image with the given program */
    execl(programname, programname, 0);
}


void run_debugger(pid_t child_pid) {
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");
    struct user_regs_struct regs;

    /* Wait for child to stop on its first instruction */
    wait(&wait_status);

    while (WIFSTOPPED(wait_status)) {
        icounter++;
        ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
        unsigned instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.rip, 0);

        procmsg("icounter = %u.  RIP = 0x%0lx.  instr = 0x%08x\n",icounter, regs.rip, instr);

        /* Make the child execute another instruction */
        if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
            perror("ptrace");
            return;
        }
        /* Wait for child to stop on its next instruction */
        wait(&wait_status);
    }

    procmsg("the child executed %u instructions\n", icounter);
}

int main(int argc, char** argv) {
    pid_t child_pid;

    if (argc < 2) {
        fprintf(stderr, "Expected a program name as argument\n");
        return -1;
    }

    child_pid = fork();
    if (child_pid == 0)
        run_target(argv[1]);
    else if (child_pid > 0)
        run_debugger(child_pid);
    else {
        perror("error with fork");
        return -2;
    }

    return 0;
}
```

Analyze the code and compile it. Then use the resulting application to debug a sample program of your choosing.

Breakpoints are very important, as they allow the debugger to let the application run, and then stop on a given event. A frequent case is breaking when the instruction pointer is at a given address. Using breakpoints involves replacing an instruction at a given position with `int 0x03` (inserting the `0xCC` byte). If we wish the application to resume execution, creating a breakpoint involves saving the content at the breakpoint address, replacing it with

`0xCC` and let the program execute. When the breakpoint is reached, the debugger will be notified, and memory needs to be patched again with the original byte.

```
unsigned addr = 0x80000; // Address to use for breakpoint
unsigned data = ptrace(PTRACE_PEEKTEXT, pid, addr, 0); // Save the byte from PID at ADDR
ptrace(PTRACE_POKETEXT, pid, addr, (data & 0xFFFFFF00) | 0xCC); // Patch it
ptrace(PTRACE_CONT, child_pid, 0, 0); // Let it run
wait(0); // Wait for BP

struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, 0, &regs);  // Get registers
regs.eip -= 1;                          // Decrease Instruction Pointer
ptrace(PTRACE_SETREGS, pid, 0, &regs);  // Set new Instruction Pointer
ptrace(PTRACE_POKETEXT, pid, (void*)addr, (void*)data); // Patch RAM with old data

ptrace(PTRACE_CONT, pid, 0, 0);         // Execute again. Original instruction will be executed
```

If the breakpoint is to stay active. In the last line, `PTRACE_CONT` should be replaced by `PTRACE_SINGLESTEP` and the memory needs to be patched again.

Build a simple use case to set breakpoints in your application. Consider both the case of a one shot breakpoint and a recurring breakpoint.

Exercise 3

Some applications are compiled to architectures that differ from host architecture. This is common for binaries from embedded devices, such as networking and IoT devices. Emulators, such as qemu or unicorn are required as they allow the execution of binaries from a multitude of architectures. They can also run full systems, which is interesting for firmware images.

For this exercise, consider the crackme-dyn-arm, which is compiled for `ARM` and has a hidden flag. Because of how the flag is hidden, dynamic analysis will be much better than a static approach.

We will use qemu-arm with gdb to recover the secret. The application will run in the emulated environment, while gdb will have access to it through a remote interface.

The first step is to load the binary:

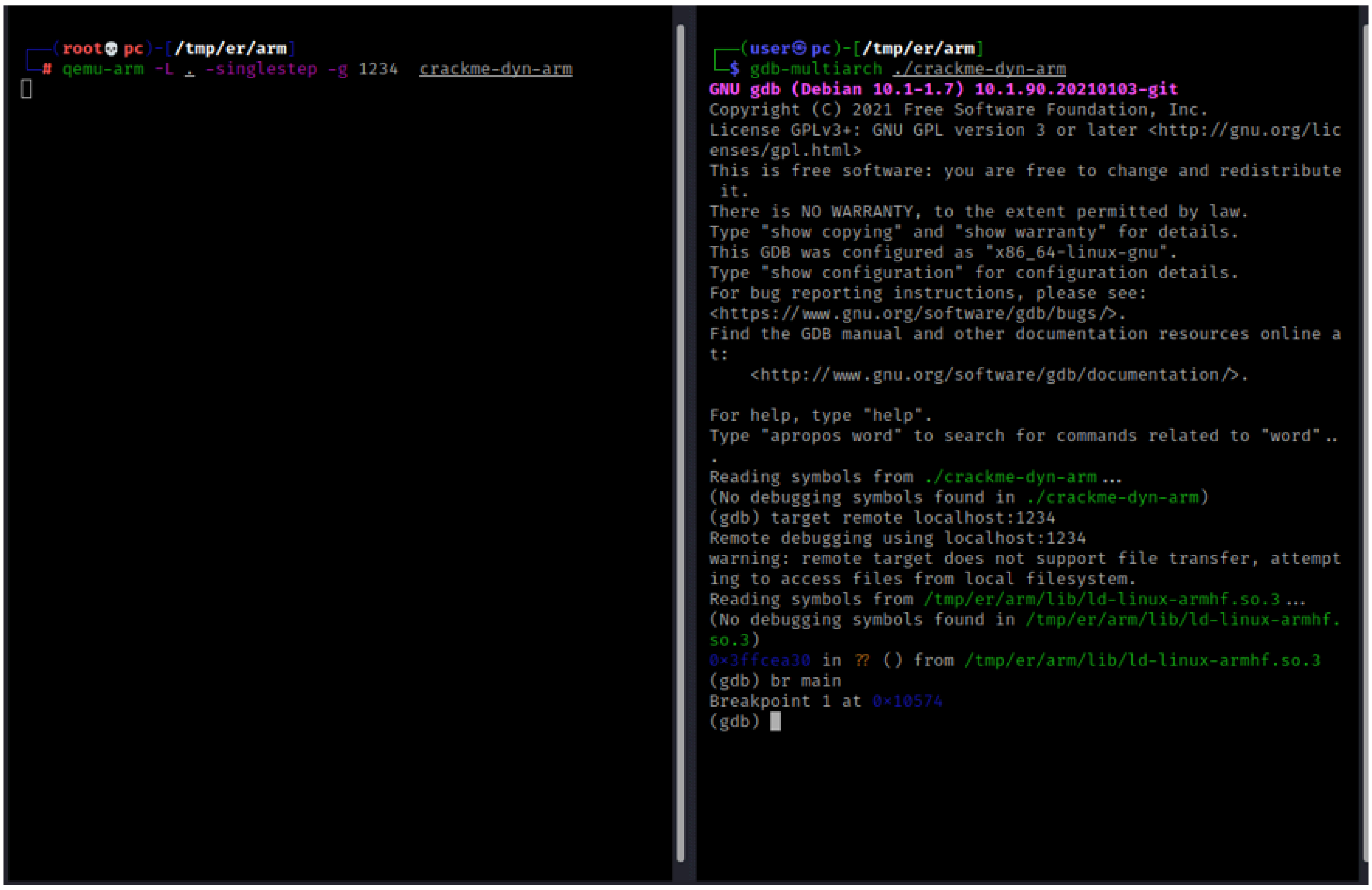
```
qemu-arm -L . -singlestep -g 1234 crackme-dyn-arm
```

qemu will provide a remote debugging interface, to which gdb can connect for remote debugging. It is important to notice, that gdb must have support for the architecture of the remote binary. Therefore, you will need to use `gdb-arm-none-eabi` or `gdb-multiarch`. These are commonly available on most Linux distributions.

After the binary is loaded, start gdb and connect to qemu.

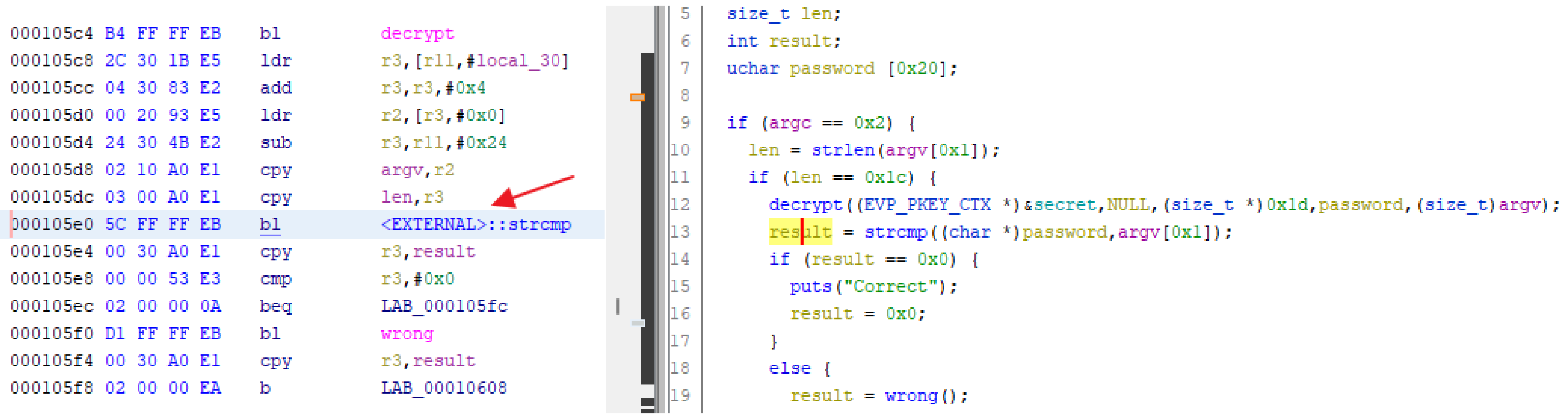
```
gdb ./crackme-dyn-arm

(gdb) target remote localhost:1234
(gdb) br main
```

The output should be the following.

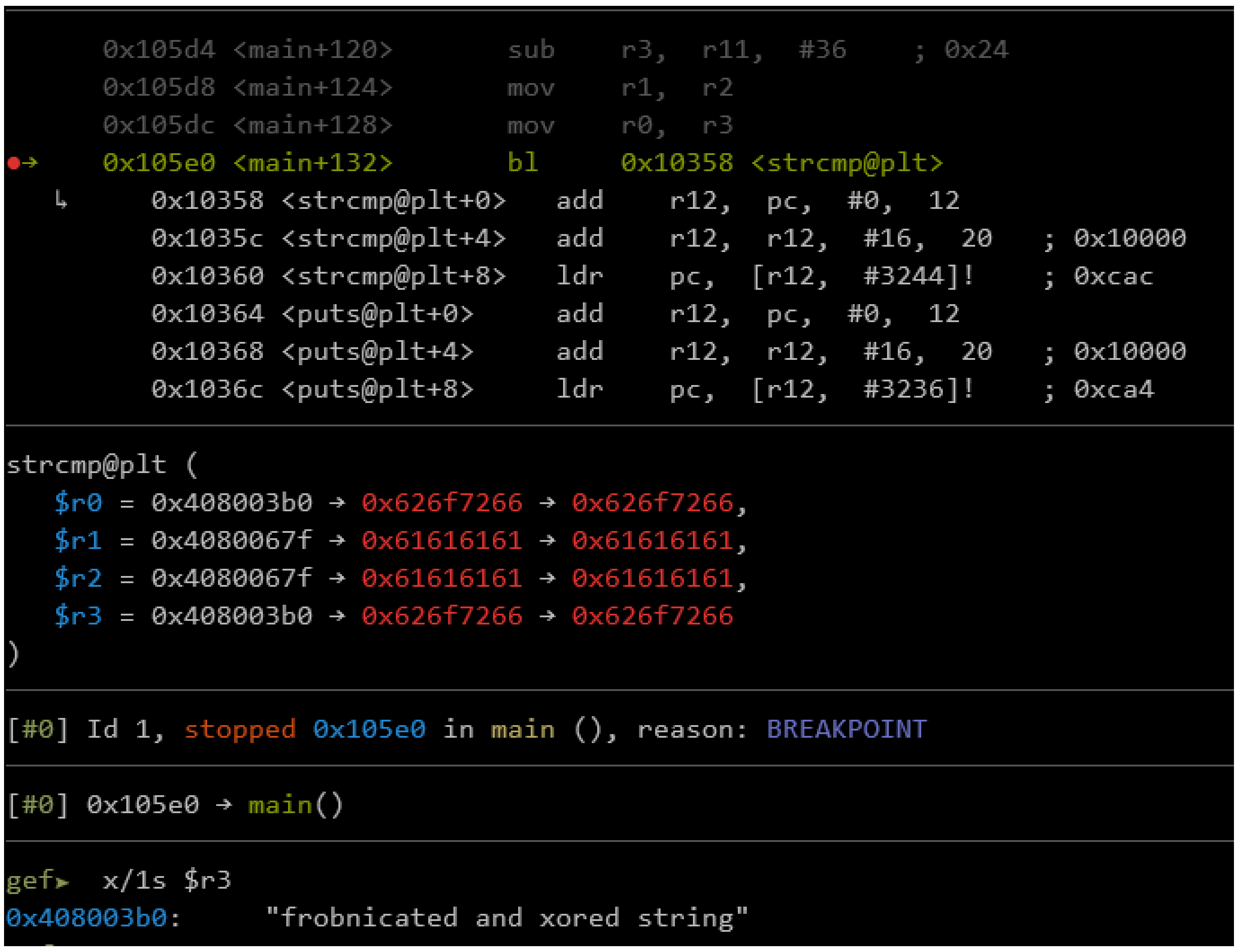


If you analyse this binary, you will notice that the password is present in the memory, and then compared with the output. Finding the correct password is just a matter of setting a breakpoint (`br *0x105e0`) to the correct location and then checking the memory content (`x/1s $r3`).

In ghidra you will be able to identify where the check is.



While in gdb you can set a breakpoint and dump the memory.



Exercise 4

Dumping a memory area at a known location when the instruction pointer is at a given location is also a trivial task for Qiling. In a debugger this would be called a `Watch` over a memory area (or a variable).

To summarize, we know that if the instruction pointer is at `0x105e0`, the register `$r3` will have the address of the correct password. A strategy for using Qiling would be to set a hook to `0x105e0`, and then get and print the memory to the terminal.

The following snippet will do this. It will call the binary with an argument of `0x1c` letters, and and set a hook. When the instruction pointer is at `0x105e0` the `dump` function is called. This function will simply read the memory at `r3`, decode it from bytes to string, clean it up and print the result.

```
from qiling import *


def dump(ql):
    buf = ql.mem.read(ql.arch.regs.r3, 30).decode().strip()

    print("Password: ", buf)
    ql.stop()

def sandbox(path, rootfs):
    global md

    ql = Qiling(path, rootfs, verbose=0)
    ql.hook_address(dump, 0x105e0)

    ql.run()

if __name__ == '__main__':
    sandbox(['./crackme-dyn-arm', 'a'*0x1c], 'rootfs')
```

A major difference between the last approach (with gdb), is that in this case we will executing the binary in a sandbox that automatically supports a wide range of architectures (ARM included). The libraries are still required, and we assume that they will be place in folder named `rootfs`.

Exercise 5

Lets apply the same concept to another file, from another architecture and platform: USB.zip. Find what file it is, and extract the flag.

As hints, take in consideration the following:

With Qiling you when you stop at some point, you can change the instruction pointer (`EIP`) and go to another address. The basic structure is:

Define the sandbox and a hook at an address:

```
def our_sandbox(path, rootfs):
    ...
    ql.hook_address(patch, 0x1000165c)
```

The hook will just change a register:

```
def patch(ql):
    ql.arch.regs.eip = 0x10001000 # Target address
```

The flag is available at `EBP - 0x2C` when the `EIP` is at the start of the `_MessageBoxThread@4`. In my case, the message with the flag is called with one argument (`PUSH EAX`), and the argument will have the value at `EBP + -0x2c`.

```
10001165 8d 45 d4        LEA        EAX=>str,[EBP + -0x2c]
10001168 50              PUSH       EAX
10001169 6a 00           PUSH       0x0
1000116b ff 15 34        CALL       dword ptr [->USER32.DLL::MessageBoxA]            = 0000282e
```

Following the previous example, you can set a hook to an address in this function and dump the memory with:

```
def pdb_break(ql):
    print(ql.mem.read(ql.arch.regs.ebp-0x2c, 38))
    ql.stop()
```

Also, your `rootfs` must have the following structure:

```
rootfs/Windows/System32/ntdll.dll
rootfs/Windows/System32/kernel32.dll
rootfs/Windows/System32/user32.dll
rootfs/Windows/System32/vcruntime140.dll
rootfs/Windows/System32/api-ms-win-crt-runtime-l1-1-0.dll
```

You can get this files from the internet (x32 bits versions!). The Registry hive files can be obtained from here: https://github.com/vivesg/RegistryToolbox

So, what is the flag?

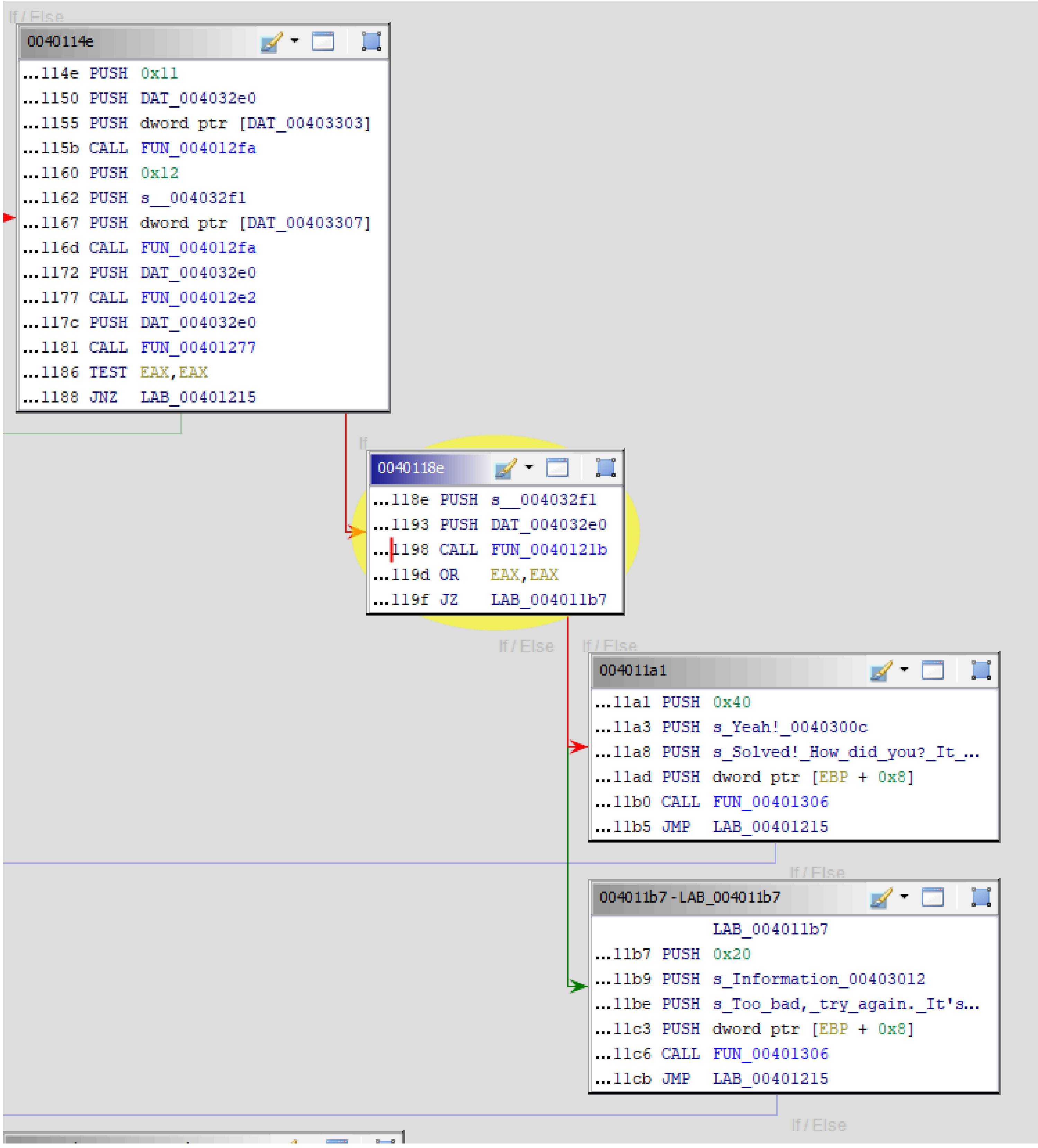
This file originated from the Nahamcon2022 CTF. We thank the authors for the availability of the challenge.

Exercise 6

In the previous exercise it was shown how a Windows binary can be instrumented, even if a Windows system is not available. Qiling doesn't not support all the required interfaces, but it allows us to manipulate and test specific parts of programs with ease.

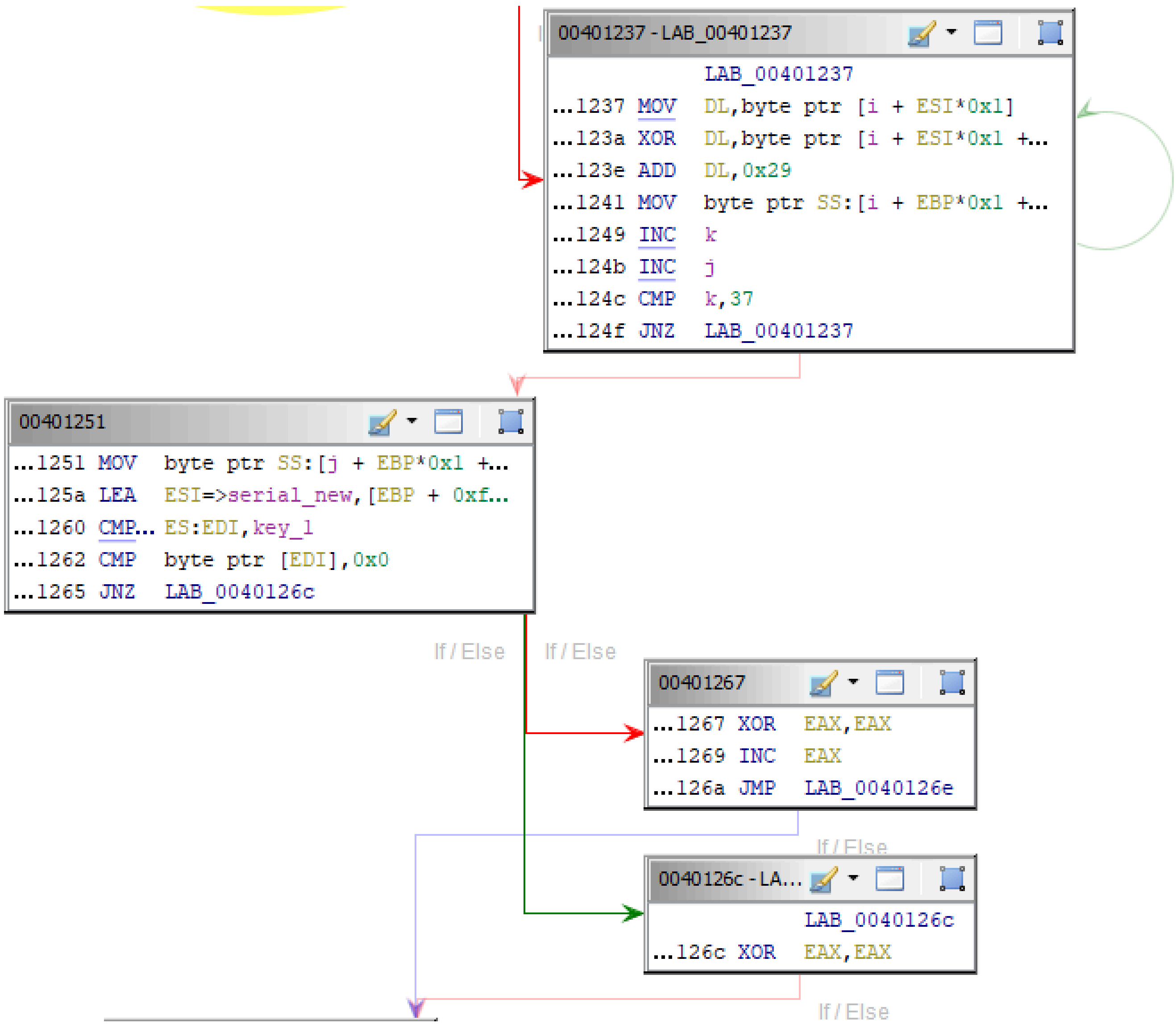
Let's apply this to reversing a keygen. Or at least to validate the reversing of a keygen, as reversing will be easy. In the process you will learn how to manipulate the process memory as you need. In this case, the objective will be to provide a username and check what serial key is produced by the keygen.

The keygen was is part of a series of crackmes by `The Dutch Cracker`, and can be found here. If you analyse the Function Graph of the program, you will notice that `FUN_0040121b` is called with two arguments (the two `PUSH` opcodes before it), and then the result (`EAX`) is evaluated. Two possible paths are present. One will print a success message, and the other will print a failure message.



If you analyse the function itself, you will notice that it will take the user name provided, calculate a serial for that user name, and then compare the result against the serial provided by the user. If the strings match, the serial is valid.

To start your reversing, reconstruct the algorithm used for creating the serial. You can find it at `0x0401237`, in the form of a loop.



After this loop, there is a string comparison. If you are using ghidra you will notice that the decompilation is a little different from the Assembly listing. Specially because the comparison is not using a loop, but a dedicated string comparison opcode called `CMPSB.REPE`.

```
00401251 36 C6 84 29 00 FF FF FF 00    MOV        byte ptr SS:[j + EBP*0x1 + 0xffffff00],0x0
0040125a 8D B5 00 FF FF FF             LEA        ESI=>serial_new,[EBP + 0xffffff00]
00401260 F3 A6                         CMPSB.REPE ES:EDI,key_1
00401262 80 3F 00                      CMP        byte ptr [EDI],0x0
00401265 75 05                         JNZ        LAB_0040126c
```

The interesting part is that when at `0x401251`, the new serial to be used for the validation is at `EBP-0x100`.

This exercise will consist in providing a user name (write to memory), call the function directly, and check the result in the middle of the function.

As a strategy do the following:

- Write whatever string (the user name) to memory. The program expects the value at `0x4032e0`.
- Set a hook to stop the program after the serial is calculated.
- Create the hook to print the memory at `EBP-0x100`.
- Start the program at a specific place, just before the function is called. You can supply the `begin=` argument to `ql.run` ((`ql.run(begin=ADDRESS)`)

Exercise 7

The unknown.zip file contains a nice example of unstructured binary developed for slightly different architecture. Using `file` we notice that it is recognized as a Master Boot Record (MBR). This is correct, and we learn that:

- MBR is loaded to address `0x7C00`
- MBR code runs in Intel x86 Real Mode (16bits)
- There are quite a few limitations and assumptions: IBM DOS 2.00 Master Boot Record (http://pcministry.com)
- There is no OS running. Input/Output must use BIOS Interrupts

As a bonus, take notice that the binary is encrypted. The first parts of the code will decrypt the following parts, until the main code is decrypted and then executed. Static analysis is almost useless due to this encryption. Actually, the main code uses data elements that are further encrypted, and are decrypted only when required.

This is a nice target for a full system emulator such as qemu as it allows us to dump the binary after it is decrypted, and then load the decrypted memory into a tool such as ghidra.

Before we start, consider this file gdb init file and place it in the same place where you will run gdb.

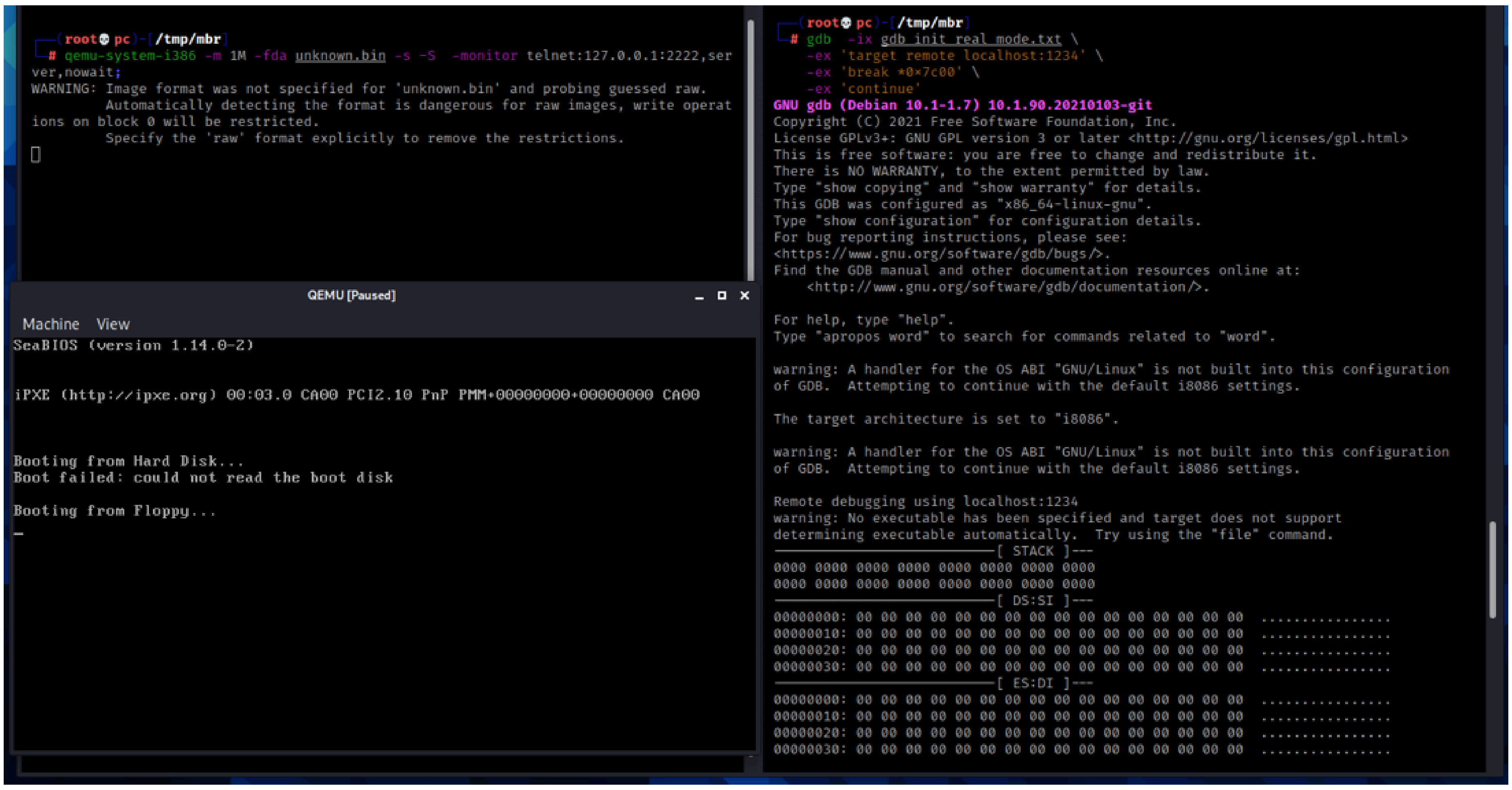
First step is to load the binary in qemu, this time using the full system mode.

```
qemu-system-i386 -m 1M -fda unknown.bin -s -S  -monitor telnet:127.0.0.1:2222,server,nowait
```

Then start gdb with the init file, connecting to the remote server (qemu). We also insert a breakpoint at `0x7c00` and let it run. This address is the entry point of the MBR, so the emulator will stop when it starts executing the code. The binary will be loaded at this location.

```
gdb -ix gdb_init_real_mode.txt -ex 'target remote localhost:1234' -ex 'break *0x7c00' -ex 'continue'
```

The result should be the following:



If you wish you can let it run and enjoy the ASCII art :D.

To analyze the loading process, issue a break at `0x7c85`. When the code jumps to this position, the second stage has been decrypted and is ready for analysis.

Then you can dump the memory and load it for analysis in ghidra. To dump the memory when stopped at `0x7c85` use `pmemsave 0 1048576 mem-at-7c85`. You can freely set breakpoints and dump the memory for analysis as required. Then load the binary in ghidra. Do not forget to select the correct architecture (i386, 16bits, real mode), and memory structure. When loading check the options and set a RAM block with base address of `0000:7c00`, file offset `0x00`, and length `0x3002`. This will place the binary in the correct location.

After you get to the main code, analyze the CFGs, rename variables, retype variables and functions and find the flags. There are 8 flags available in the binary. Some are available by inspecting the memory, some are encrypted, some require some konami kung fu.

Tools and links

- bvi: http://bvi.sourceforge.net/
- FileInsight: https://github.com/nmantani/FileInsight-plugins
- ghex: https://wiki.gnome.org/Apps/Ghex
- ghidra: https://ghidra-sre.org/
- HexEdit: https://hexed.it/
- HexWorkshop: http://www.hexworkshop.com/
- HxD: https://mh-nexus.de/en/hxd/
- ImHex: https://github.com/WerWolv/ImHex
- ltrace: https://man7.org/linux/man-pages/man1/ltrace.1.html
- objdump: https://man7.org/linux/man-pages/man1/objdump.1.html
- qemu: https://www.qemu.org/
- qiling: https://github.com/qilingframework/qiling
- readelf: https://man7.org/linux/man-pages/man1/readelf.1.html
- strace: https://man7.org/linux/man-pages/man1/strace.1.html
- unicorn: https://www.unicorn-engine.org/docs/beyond_qemu.html

PREVIOUS
Static Analysis of Applications

NEXT
Project 1 - Android Reversing

Last updated on 2 Apr 2024