Binary Analysis – Emulation and Instrumentation

REVERSE ENGINEERING

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Binary Analysis Process (cont.)

- Up to now we know how ELF files are structured, but the question remains: how do we analyse ELF files?
  - Or any other binary executable

- A possible flow can be:
  - File analysis (file, nm, ldd, content visualization, foremost, binwalk)
  - Static Analysis (disassemblers and decompilers
  - Behavioral Analysis (strace, LD_PRELOAD)
  - **Dynamic Analysis (debuggers and emulators)**

Dynamic Binary Analysis

- Allows capturing the dynamic behavior of some code
  - Behavior that depends on external input
  - Data structures and even code revealed during execution time

- Allows runtime validation/evaluation of binary code
  - A program, a firmware, part of a program, a sequence of instructions
  - Under a controlled context
  - On a different (more flexible, or controllable, or safe) environment

Dynamic Binary Analysis

How

- Load the binary and ***execute*** instructions of the target binary
    - The meaning of “execute” is broader than it may look

- Allow some interaction with the binary while it is running
    - Break the execution at some point
    - Inspect memory and process its content
    - Change memory, either variables or code
    - Execute code in a controlled manner: step by step, in chunks, until a given point

Dynamic Binary Analysis

Approaches

- Analysis of an execution flow can either be **passive** or **active**.
    - Choosing either one or the other has consequences on the soundness, the coverage, etc. of the results

- **Passive analysis**: observation
    - **register values**: return value of functions (`rax`), program counter (`pc`), stack frame (`rbp, rsp`), etc.
    - **stack inspection**: local variables, input parameters (according to some calling conventions), return address, etc.
    - **heap inspection**: the number of allocated blocks, their content, etc

- **Active analysis**: modification
    - Easily explore paths without finding inputs that actually activate them

Dynamic Binary Analysis

Caveats

- Binary applications **are more powerful and complex**
  - May be written in multiple languages, and have code that runs in a VM
  - May consider code that changes the host system, or is modified in runtime

- Binary analysis of complex applications requires a different toolset
  - The principles will be the same, but the tools will allow fine grained control and isolation
  - Side effects and execution impact may be subtle (remember Meltdown and Spectre)
  - Host systems may be more complex

Considerations

(Need for) Stability

- Reversing is significantly more difficult if execution is unstable.
  - Observations are affected by "random" factors, such as multithreaded execution, hardware behavior, user interactions with graphical interface and so on.
  - Applications being reversed should be isolated from external effects are much as possible.
- Determinism in a design results from stable execution of a program run
  - Thus it facilitates debugging and reversing.
  - State may also be deterministically altered for the entire program or for a specific function (fuzzing)
- Logs can be obtained from executions using monitor applications

Considerations

(Need for) Save and Replaying

- Reversing may need **tracing** from the current state to the code where a change was produced.
  - It implies moving "back in time".
  - To restore past program state, one must **re-run it** and try to find failure source.
  - This operation may be performed multiple times, **moving backward step-by-step, and then forward**.

- Deterministic replay reconstructs program execution using previously recorded input data.
  - The first program run is used to record these inputs into the log.
  - Then all following runs will reconstruct the same behavior, because the program uses only recorded inputs.
  - Should included all inputs (disk, network)

Considerations

(Need for) Safety

- Target binary **may be malicious (… it is always malicious until proven safe)**

- An important aspect of Reversing binaries is malware analysis
  - Malware is way to complex to be analyzed statically
  - But executing the malware may be dangerous
    - **Most important: dangerous in ways unknown to the reverse engineer**

- Solutions must create the adequate isolation boundaries between environments
  - If stability is required, no interactions with the software under analysis
  - Sometimes, isolation must be broken to trigger specific behavior
    - Network connection allowing contact with a C&C address or to download some payload
    - Disk or file presence
    - Whenever possible, such resource should be virtualized

Considerations

(Need for) Support of Heterogeneous Architectures

- Dynamic analysis requires the execution of the program under analysis.

- An analyst will mostly run on an Intel x86 64bits computer (a COTS laptop/server)
  - Most embedded devices are ARM, which has several variants
  - Microcontrollers frequently use 8085, AVR or PIC architectures (MIPS)
  - Several specialty SOCs use custom architectures (the list is large…)
  - Several binary formats are popular: ELF, PE, DWARF and then many others from IoT

- Frameworks must be extensible in order to support a wide range of architectures
  - And the related interfaces and customizations
  - While minimizing the need for new tools

Considerations

(Need for) Support of Peripherals and external entities

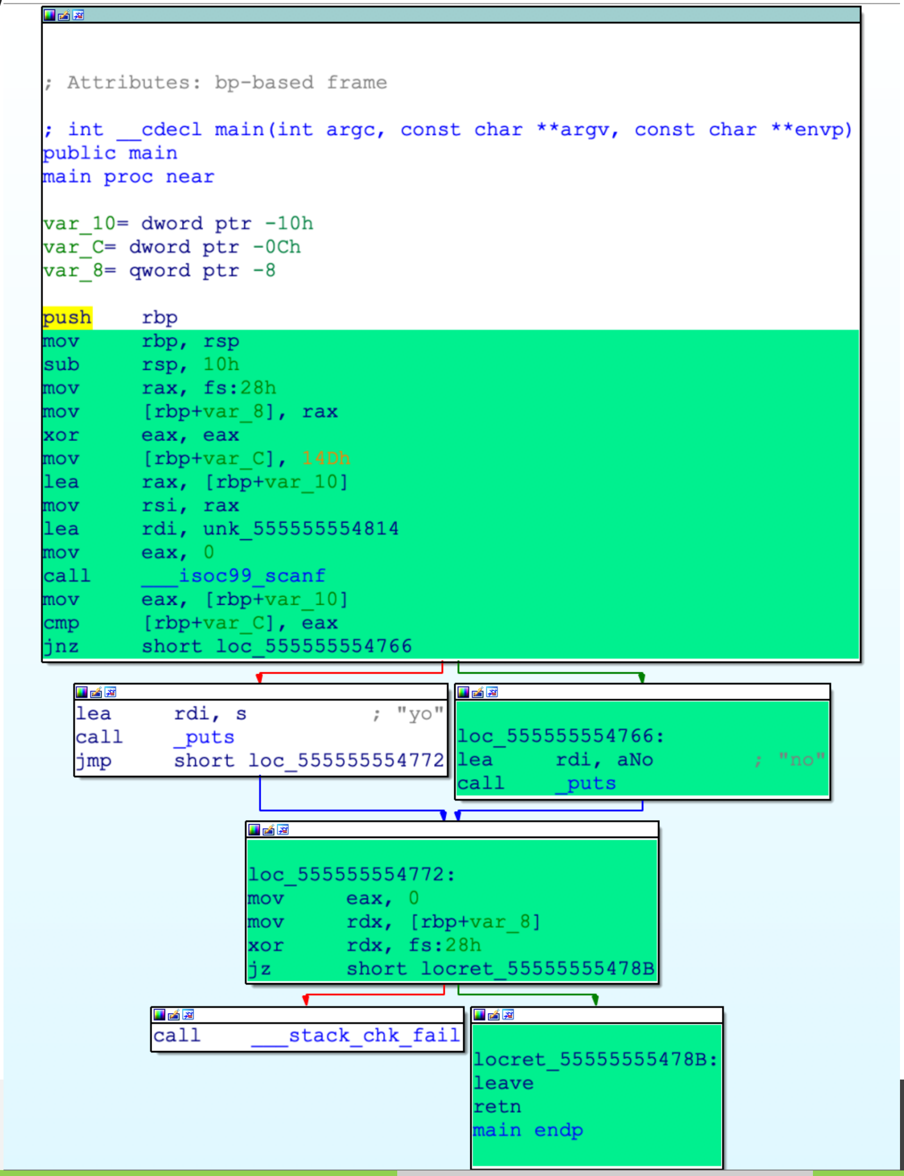
- Reversing an application with external interactions may require the existence of the related entities
  - Web sites, servers in fixed/dynamic IP addresses
  - Common physical devices for user input, storage, …
  - Exotic external devices communicating through known or unknown buses
  - Hardware Dongles

- Need to recreate the set of devices/entities required to trigger a specific path
  - Frequently resorts to device emulation with mock software constructs

Considerations

(Need for) Context manipulation (instrumentation)

- The main limitation of a dynamic approach is **coverage**.
  - Every path that is not covered by the instrumented executions cannot be analyzed.
  - This limitation can be slightly reduced by performing active instrumentation, and in particular by **forcing conditional branching**



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     [rbp+var_C], 14Dh
lea     rax, [rbp+var_10]
mov     rsi, rax
lea     rdi, unk_555555554814
mov     eax, 0
call    ___isoc99_scanf
mov     eax, [rbp+var_10]
cmp     [rbp+var_C], eax
jnz     short loc_555555554766
```

```
lea     rdi, s          ; "yo"
call    _puts
jmp     short loc_555555554772
```

```
loc_555555554766:
lea     rdi, aNo        ; "no"
call    _puts
```

```
loc_555555554772:
mov     eax, 0
mov     rdx, [rbp+var_8]
xor     rdx, fs:28h
jz      short locret_55555555478B
```

```
call    ___stack_chk_fail
```

```
locret_55555555478B:
leave
retn
main endp
```

Example of Intel PIN coverage output provided to IDA

https://hex-rays.com/products/ida/support/tutorials/pin/pin_tutorial.html

Considerations

(Need for) Context manipulation (instrumentation)

- A reversing task will need to observe **structure** and **behavior**
  - The analysis should have **enough coverage to recover the adequate level of detail**
  - But while static analysis aims for wide coverage, dynamic analysis aims for focus
  - What if a specific course of execution is not triggered?
  - **Results of dynamic analysis are dependent on the context of the execution**

- Context manipulation allows setting the adequate state to trigger a specific flow of execution, **increasing the reversing coverage**
  - Achieved by careful manipulation of execution state, registers and memory content
  - Problems:
    - May lead to the recovery of an incorrect design as the found flow may be a decoy!
    - May lead to the recovery of artificial vulnerabilities, that do not really exist

Considerations

Context manipulation (instrumentation)

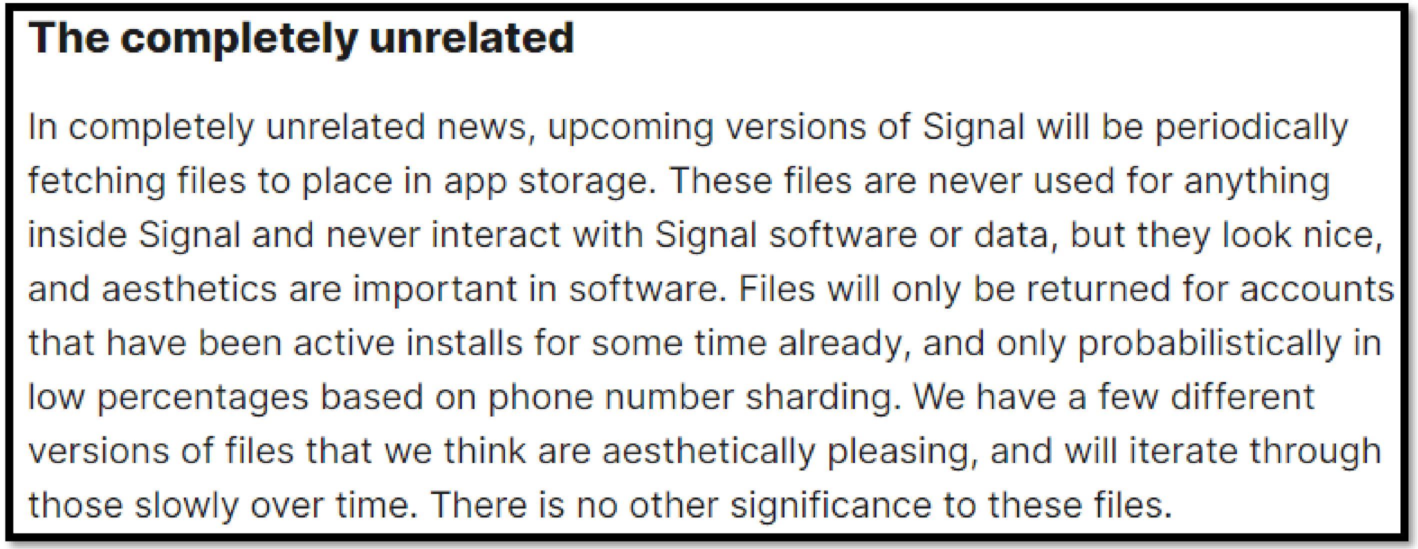
- **Live patching**: modifying RAM in a debugger/controlled environment
- **File Patching**: alter binaries files to replace their content
- **Binary Instrumentation**: Real time, automated modification

Considerations

Design Fidelity

- Program under analysis may **detect it and try to defend actively against analysis**.
  - For instance, it can hide a part of its behavior if it detects that it is being analyzed.
  - This anti-debugging and anti-instrumentation techniques are used by many malwares.

- So, when we achieve a hypothesis of a design, how correct it is?



The completely unrelated

In completely unrelated news, upcoming versions of Signal will be periodically fetching files to place in app storage. These files are never used for anything inside Signal and never interact with Signal software or data, but they look nice, and aesthetics are important in software. Files will only be returned for accounts that have been active installs for some time already, and only probabilistically in low percentages based on phone number sharding. We have a few different versions of files that we think are aesthetically pleasing, and will iterate through those slowly over time. There is no other significance to these files.

https://signal.org/blog/cellebrite-vulnerabilities/

Considerations

Design Fidelity: example of gdb+br detection

```
gef➤  disassemble evil
xDump of assembler code for function evil:
   0x0000000008001163 <+0>:     endbr64
   0x0000000008001167 <+4>:     push   rbp
   0x0000000008001168 <+5>:     mov    rbp,rsp
   0x000000000800116b <+8>:     lea    rax,[rip+0xe9c]        # 0x800200e
   0x0000000008001172 <+15>:    mov    rdi,rax
   0x0000000008001175 <+18>:    call   0x8001030 <puts@plt>
   0x000000000800117a <+23>:    nop
   0x000000000800117b <+24>:    pop    rbp
   0x000000000800117c <+25>:    ret
End of assembler dump.
```

endbr is `0xfa1e0ff3`

```
gef➤  br *0x0000000008001163
Breakpoint 1 at 0x8001163
```

br will modify address to trigger `int3`
opcode for `int3` is `0xcc`

```
gef➤  r
Starting program: main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
evil at: 8001163 val: fa1e0fcc
Good code
[Inferior 1 (process 2173) exited normally]
```

```
antibr  batcat main.c
       File: main.c
       #include<stdlib.h>
       #include<stdio.h>
       #include<stdint.h>

       void good(){
           printf("Good code\n");
       }

       void evil() {
           printf("Evil code\n");
       }
       int main(int argc, char** argv) {
           uint32_t* ptr = (void*) evil;

           printf("evil at: %x val: %x\n", ptr, *ptr);
           if(*ptr == 0xfa1e0ff3) {
               evil();
           }else {
               good();
           }
       }
antibr  gcc -fcf-protection -o main main.c
antibr  ./main
evil at: 778fc163 val: fa1e0ff3
Evil code
```

execution differs

Dynamic Binary Analysis of Binaries

Processes

- Tracing
- Debugging
- Sandboxing
- Emulation
- Instrumentation

Tracers

… Already briefly discussed in previous lectures

- **Tracers <u>execute a binary</u>, logging information about function and system calls**

- Binary is executed in the analyst's system
  - That is: In a VM!

- Tracer adds hooks to application or kernel to gain information about execution
  - Access to files, packets sent, registry access

- No confinement or security measures in place
  - Actually, there may be no interaction between the tracer and the application
    - Tracer monitors system through kernel debug interfaces

Tracers

… Already briefly discussed in previous lectures

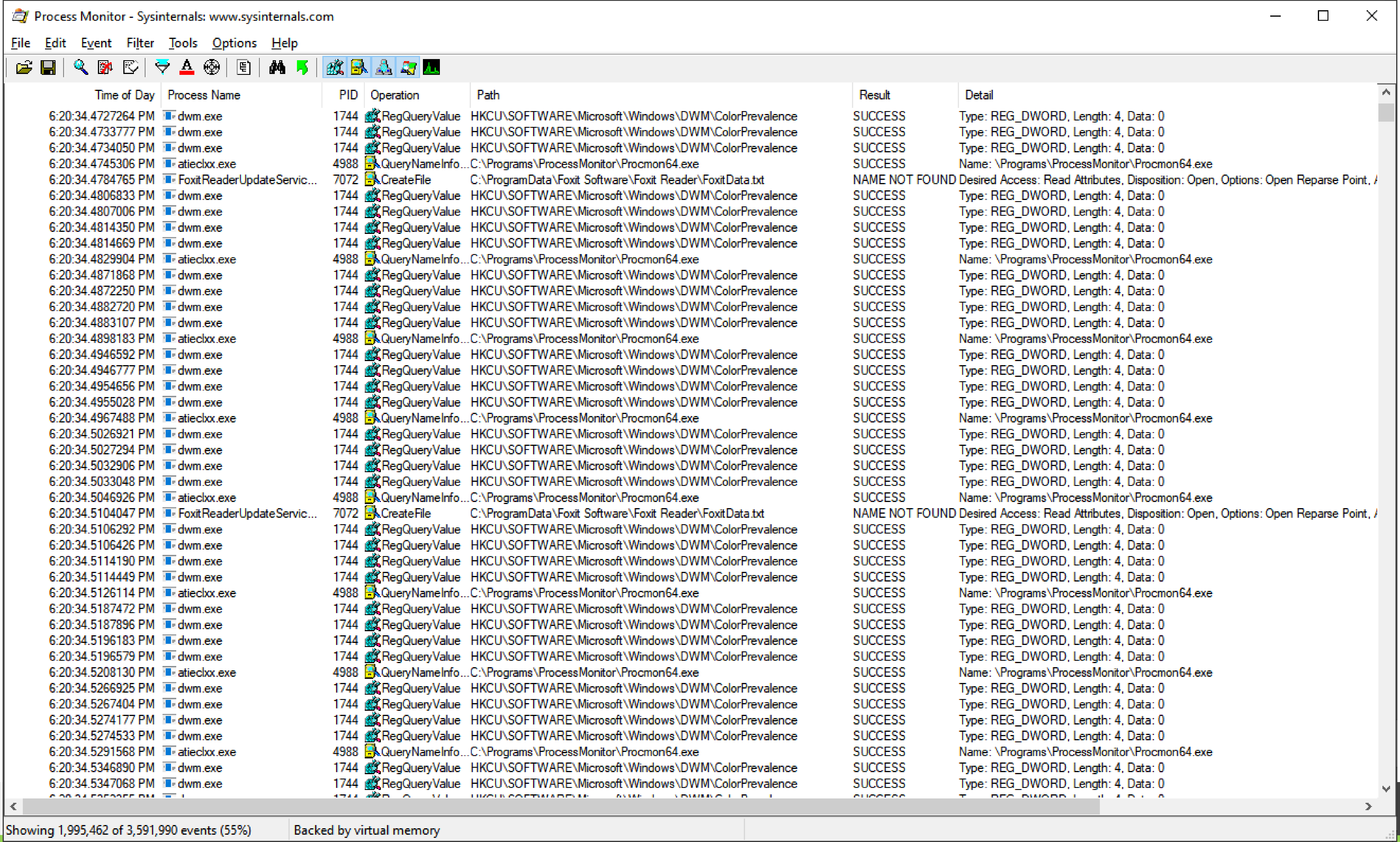
- **Limitations:**
  - No isolation, no capability to analyze malicious or harmful code
  - Can only inspect interactions between the application and the external environment
  - Host environment must be compatible with the target binary
    - No possibility of analyzing windows binaries on linux, vice-versa, embedded systems on windows, etc…

- **Linux**: `ltrace, strace (ptrace), bpftrace, wireshark, valgrind, cachegrind, callgrind, helgrind`

- **Windows**: `process monitor, wireshark`

```
$ ltrace -CfirS ./hello
[pid 5287]   0.000000 [0x7f7e47875307] SYS_brk(0)                                       = 0x55582397c000
[pid 5287]   0.000447 [0x7f7e47876363] SYS_mmap(0, 8192, 3, 34)                         = 0x7f7e47854000
[pid 5287]   0.000166 [0x7f7e478760a7] SYS_access("/etc/ld.so.preload", 04)            = -2
[pid 5287]   0.000192 [0x7f7e478761dd] SYS_openat(0xffffff9c, 0x7f7e4787e103, 0x80000, 0) = 3
[pid 5287]   0.000169 [0x7f7e47875fea] SYS_newfstatat(3, 0x7f7e4787ec84, 0x7ffd04c65030, 4096) = 0
[pid 5287]   0.000072 [0x7f7e47876363] SYS_mmap(0, 0x15267, 1, 2)                       = 0x7f7e4783e000
[pid 5287]   0.000113 [0x7f7e478760c7] SYS_close(3)                                     = 0
[pid 5287]   0.000110 [0x7f7e478761dd] SYS_openat(0xffffff9c, 0x7f7e47854140, 0x80000, 0) = 3
[pid 5287]   0.000077 [0x7f7e47876234] SYS_read(3, "\177ELF\002\001\001\003", 832)      = 832
[pid 5287]   0.000146 [0x7f7e4787625a] SYS_pread(3, 0x7ffd04c64db0, 784, 64)            = 784
[pid 5287]   0.000078 [0x7f7e47875fea] SYS_newfstatat(3, 0x7f7e4787ec84, 0x7ffd04c65030, 4096) = 0
[pid 5287]   0.000102 [0x7f7e4787625a] SYS_pread(3, 0x7ffd04c64c80, 784, 64)            = 784
[pid 5287]   0.000082 [0x7f7e47876363] SYS_mmap(0, 0x1e1f50, 1, 2050)                   = 0x7f7e4765c000
[pid 5287]   0.000286 [0x7f7e47876363] SYS_mmap(0x7f7e47682000, 0x155000, 5, 2066)      = 0x7f7e47682000
[pid 5287]   0.000094 [0x7f7e47876363] SYS_mmap(0x7f7e477d7000, 0x54000, 1, 2066)       = 0x7f7e477d7000
[pid 5287]   0.000123 [0x7f7e47876363] SYS_mmap(0x7f7e4782b000, 0x6000, 3, 2066)        = 0x7f7e4782b000
[pid 5287]   0.000109 [0x7f7e47876363] SYS_mmap(0x7f7e47831000, 0xcf50, 3, 50)          = 0x7f7e47831000
[pid 5287]   0.000113 [0x7f7e478760c7] SYS_close(3)                                     = 0
[pid 5287]   0.000071 [0x7f7e47876363] SYS_mmap(0, 0x3000, 3, 34)                       = 0x7f7e47659000
[pid 5287]   0.000121 [0x7f7e47870eb5] SYS_arch_prctl(4098, 0x7f7e47659740, 0xffff8081b89a5f30, 34) = 0
[pid 5287]   0.000071 [0x7f7e4786800a] SYS_set_tid_address(0x7f7e47659a10, 0x7f7e47659740, 0x7f7e478890b0, 34) = 5287
[pid 5287]   0.000088 [0x7f7e47868066] SYS_set_robust_list(0x7f7e47659a20, 24, 0x7f7e478890b0, 34) = 0
[pid 5287]   0.000067 [0x7f7e4786809d] SYS_334(0x7f7e4765a060, 32, 0, 0x53053053)       = 0
[pid 5287]   0.000176 [0x7f7e478763c7] SYS_mprotect(0x7f7e4782b000, 16384, 1)           = 0
[pid 5287]   0.000069 [0x7f7e478763c7] SYS_mprotect(0x555822a8c000, 4096, 1)            = 0
[pid 5287]   0.000096 [0x7f7e478763c7] SYS_mprotect(0x7f7e47886000, 8192, 1)            = 0
[pid 5287]   0.000097 [0x7f7e47758fa0] SYS_prlimit64(0, 3, 0, 0x7ffd04c65b70)           = 0
[pid 5287]   0.000121 [0x7f7e478763a7] SYS_munmap(0x7f7e4783e000, 86631)                = 0
[pid 5287]   0.003672 [0x555822a8a14c] puts("Hello Word" <unfinished ...>
[pid 5287]   0.000826 [0x7f7e4775301a] SYS_newfstatat(1, 0x7f7e477f1df3, 0x7ffd04c65cc0, 4096) = 0
[pid 5287]   0.000609 [0x7f7e476f0535] SYS_318(0x7f7e47836498, 8, 1, 4096)              = 8
[pid 5287]   0.000107 [0x7f7e477593f7] SYS_brk(0)                                       = 0x55582397c000
[pid 5287]   0.000070 [0x7f7e477593f7] SYS_brk(0x55582399d000)                          = 0x55582399d000
[pid 5287]   0.000081 [0x7f7e47753b00] SYS_write(1, "Hello Word\n", 11Hello Word
)                 = 11
[pid 5287]   0.000172 [0x555822a8a14c] <... puts resumed> )                             = 11
[pid 5287]   0.000084 [0x7f7e4772f995] SYS_exit_group(11 <no return ...>
[pid 5287]   0.000443 [0xffffffffffffffff] +++ exited (status 11) +++
```

Function calls
System calls



Time of Day	Process Name	PID	Operation	Path	Result	Detail
6:20:34.4727264 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4733777 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4734050 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4745306 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4784765 PM	FoxitReaderUpdateServic...	7072	CreateFile	C:\ProgramData\Foxit Software\Foxit Reader\FoxitData.txt	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point,
6:20:34.4806833 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4807006 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4814350 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4814669 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4829904 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4871868 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4872250 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4882720 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4883107 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4898183 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.4946592 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4946777 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4954656 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4955028 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.4967488 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5026921 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5027294 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5032906 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5033048 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5046926 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5104047 PM	FoxitReaderUpdateServic...	7072	CreateFile	C:\ProgramData\Foxit Software\Foxit Reader\FoxitData.txt	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point,
6:20:34.5106292 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5106426 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5114190 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5114449 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5126114 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5187472 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5187896 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5196183 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5196579 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5208130 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5266925 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5267404 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5274177 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5274533 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5291568 PM	atieclxx.exe	4988	QueryNameInfo...	C:\Programs\ProcessMonitor\Procmon64.exe	SUCCESS	Name: \Programs\ProcessMonitor\Procmon64.exe
6:20:34.5346890 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:20:34.5347068 PM	dwm.exe	1744	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\DWM\ColorPrevalence	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0

Showing 1,995,462 of 3,591,990 events (55%)

Backed by virtual memory

Debugging

- Applications that can control (trace) a target executing binary
  - Debuggers can **create a process** and analyze it or **attach to a running process**
    - Process usually executes in the host system
  - This is the “typical”, low tech way of dynamically analyzing a program
    - Reuses concepts/tools from the engineering process, applied to reverse engineering

- **Provide:** extensive, interactive control over a process execution flow
  - Frequently at the level of opcodes and assembly
  - Can be integrated with static analysis tools
    - Combining execution information with decompiled code, CFGs, disassembly

Debugging

Limitations

- Debugging **can be detected** and subverted by the target application
  - Especially popular in malware and DRM systems

- Target application must be executed in a full hosted environment
  - Without isolation measures, this provides a serious security risk
  - Remote debugging may be used to circumvent this limitation

- Host system architecture must match the target binary architecture
  - Binary is loaded to the host system as a standard process
  - No debugging of windows in Linux, ARM or MIPS in x86
  - No direct way of debugging shellcode or a binary blob (e.g firmware).

Debugging

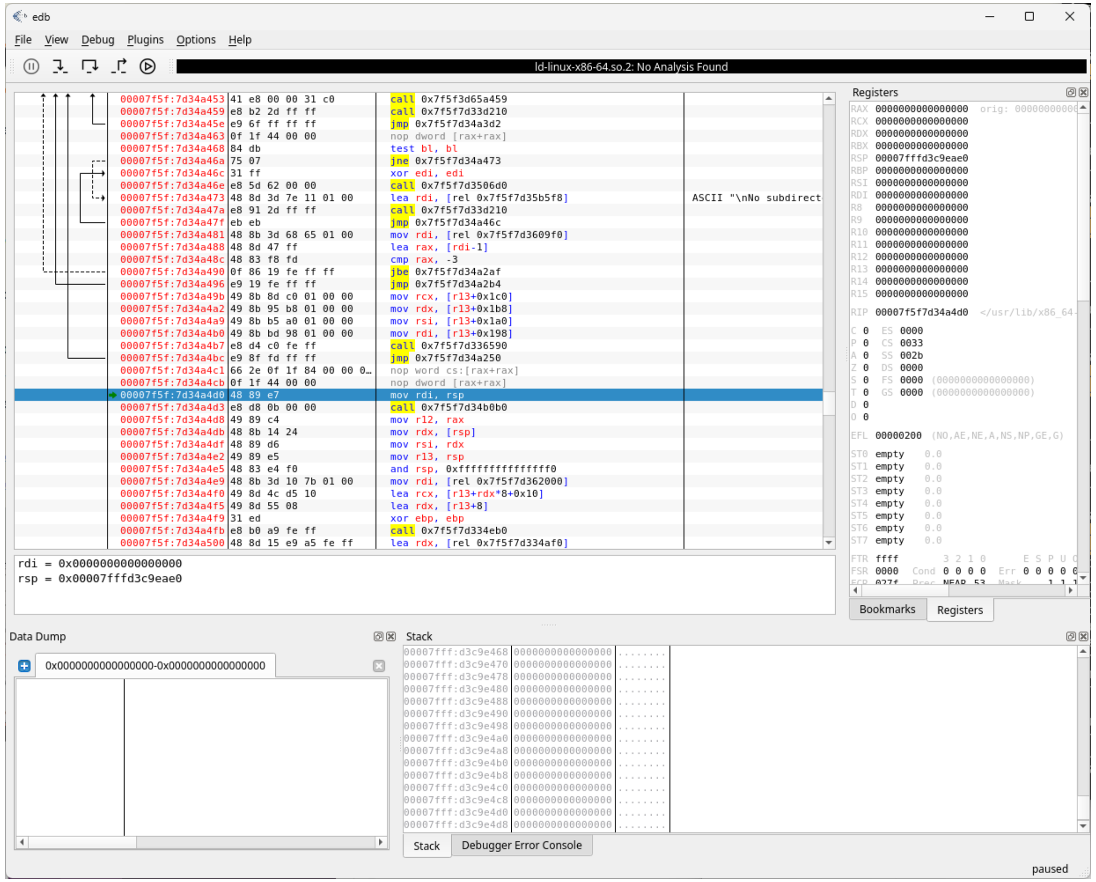
How debuggers work?

- Debuggers explore system calls provided by the operating system
  - Debuggers either:
    - create a child process, sharing the same address space
    - attach to an existing process given that the user has the correct permissions (e.g. root)
  - Linux: **`ptrace`**
  - Windows: provides API for process control
    - **`CreateProcess`** with specific **`dwCreationFlags`** (`DEBUG_PROCESS`)
    - **`OpenProcess`** with **`dwDesiredAccess`** `(PROCESS_VM_READ, PROCESS_VM_WRITE, PROCESS_VM_OPERATION)`

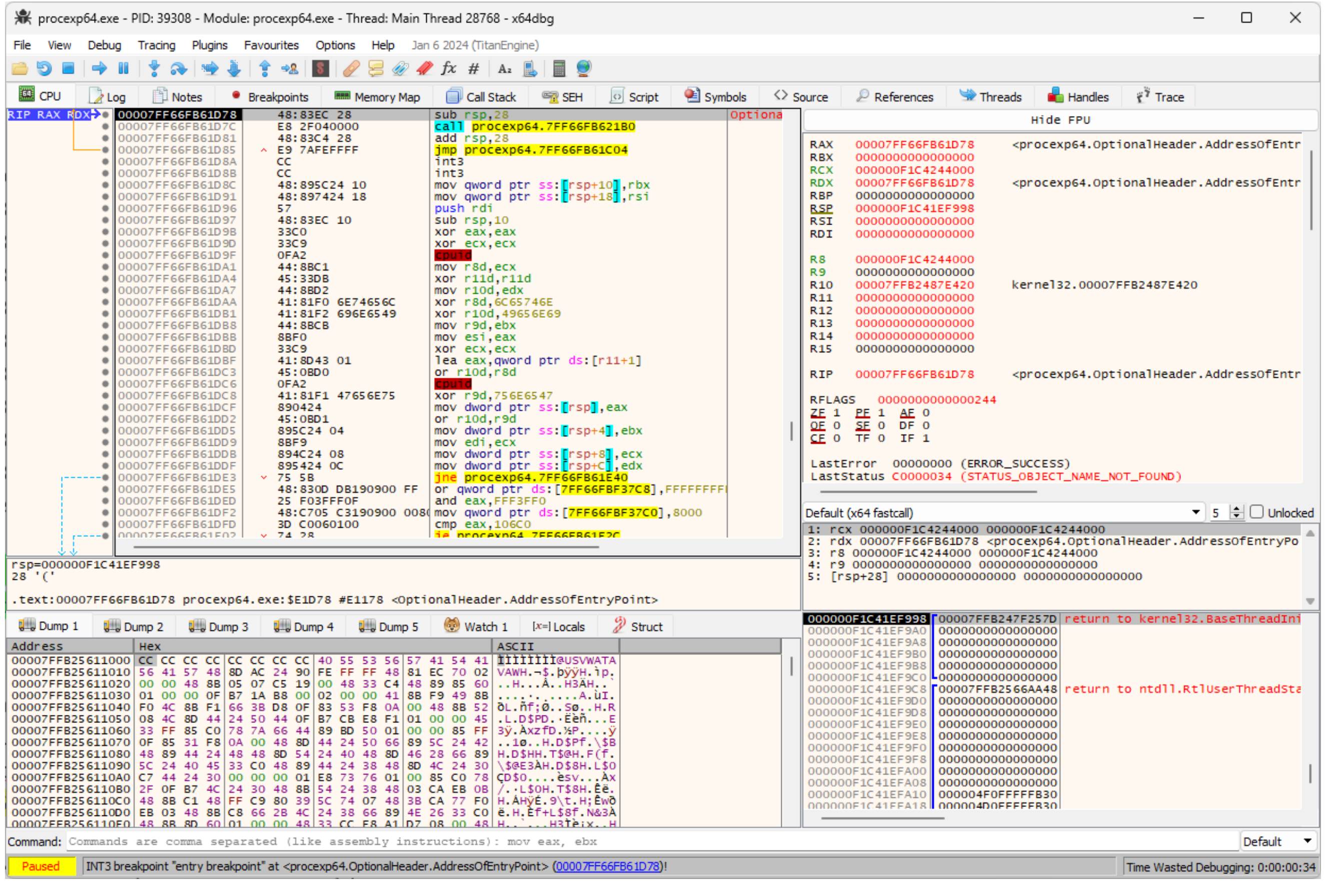
- Debuggers may attach to hardware devices providing external debugging
  - Used in embedded devices

Debugging

edb and x86dbg



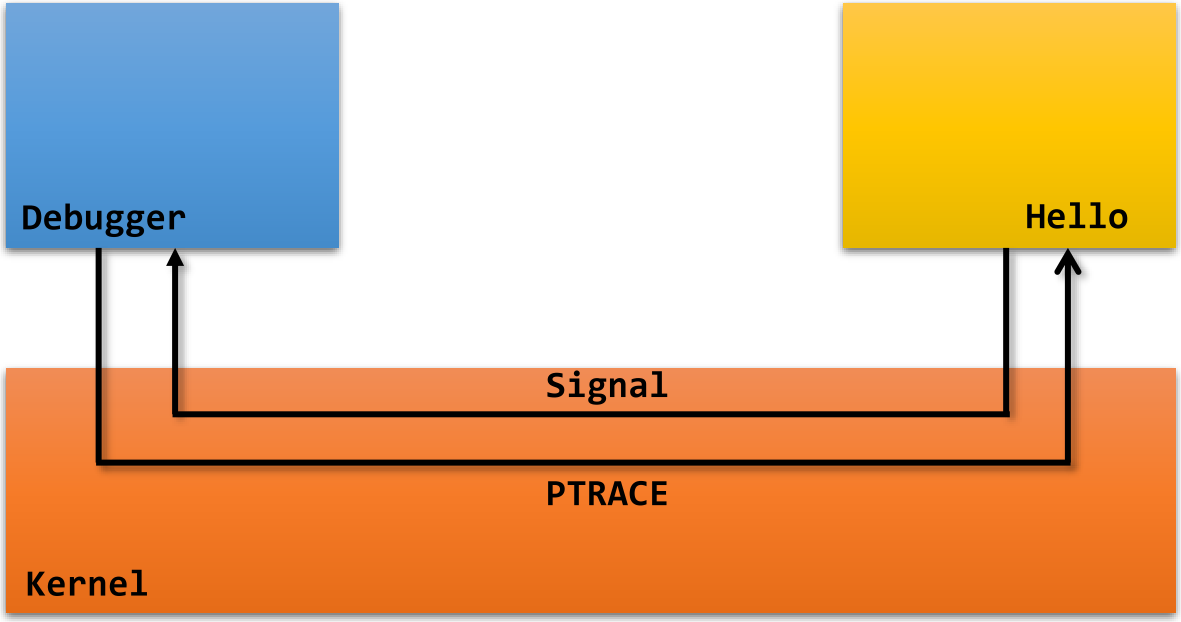
```
rdi = 0x0000000000000000
rsp = 0x00007fffd3c9eae0
```



```
rsp=000000F1C41EF998
28 '('

.text:00007FF66FB61D78 procexp64.exe:$E1D78 #E1178 <OptionalHeader.AddressOfEntryPoint>
```

INT3 breakpoint "entry breakpoint" at <procexp64.OptionalHeader.AddressOfEntryPoint> (00007FF66FB61D78)!

Debugging

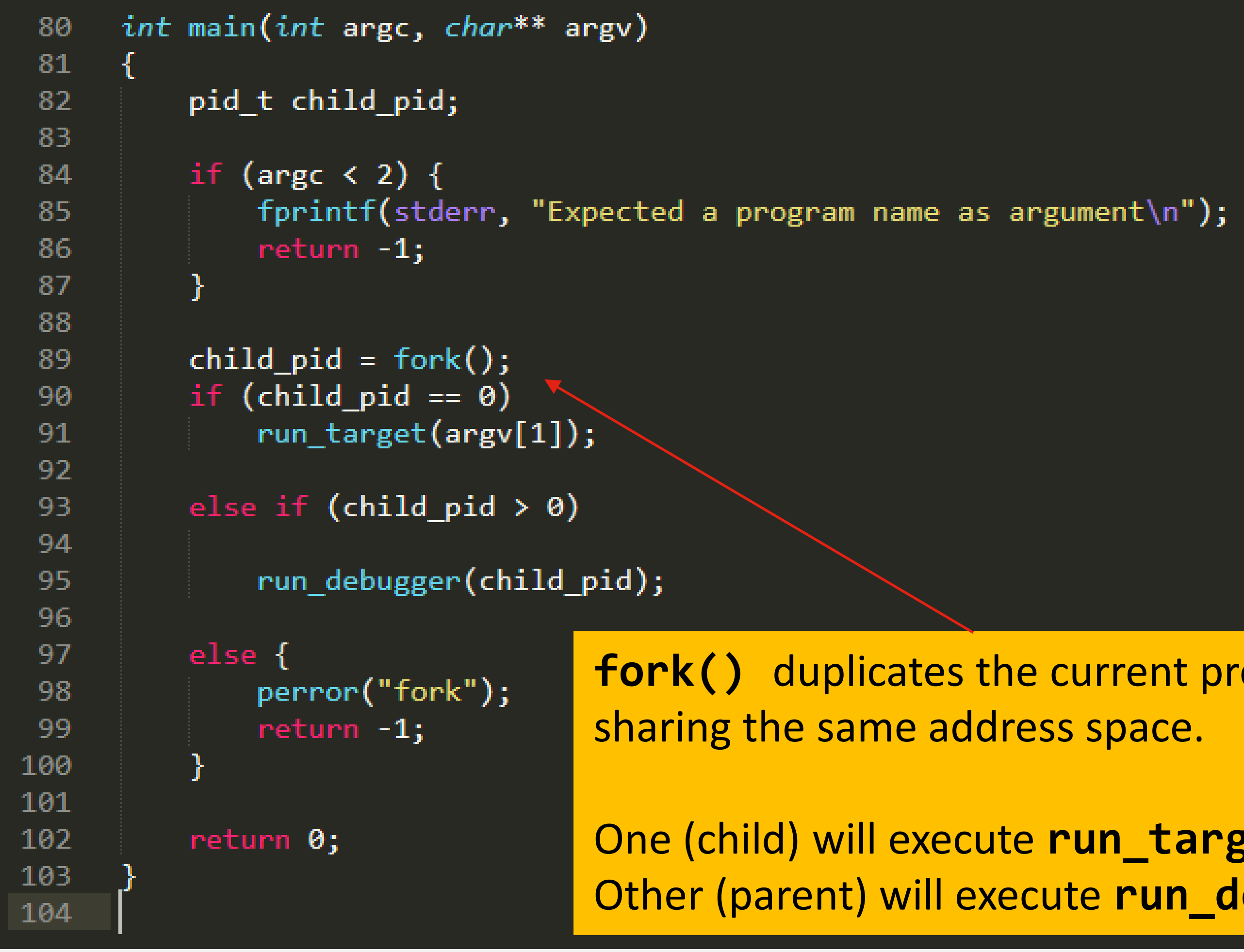


Debugger set breakpoints which Trigger SIGTRAP, returning control to the debugger.

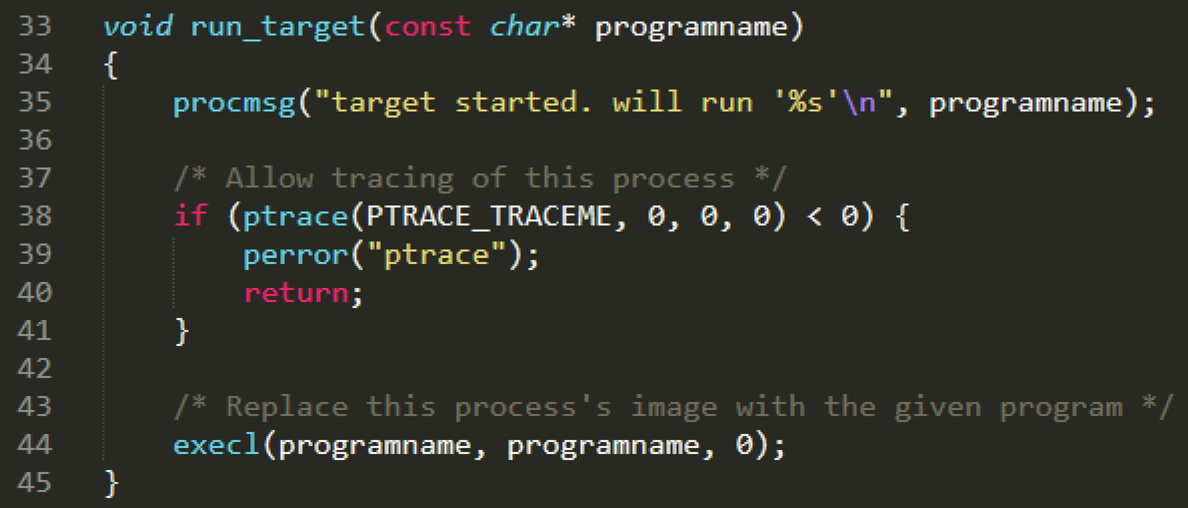
Patching the code with **`0xCC`** or using Hardware breakpoints (through **PTRACE**)

Debugging

debugger.c



```c
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("fork");
        return -1;
    }

    return 0;
}
```



```c
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);
}
```

`fork()` duplicates the current process. While sharing the same address space.

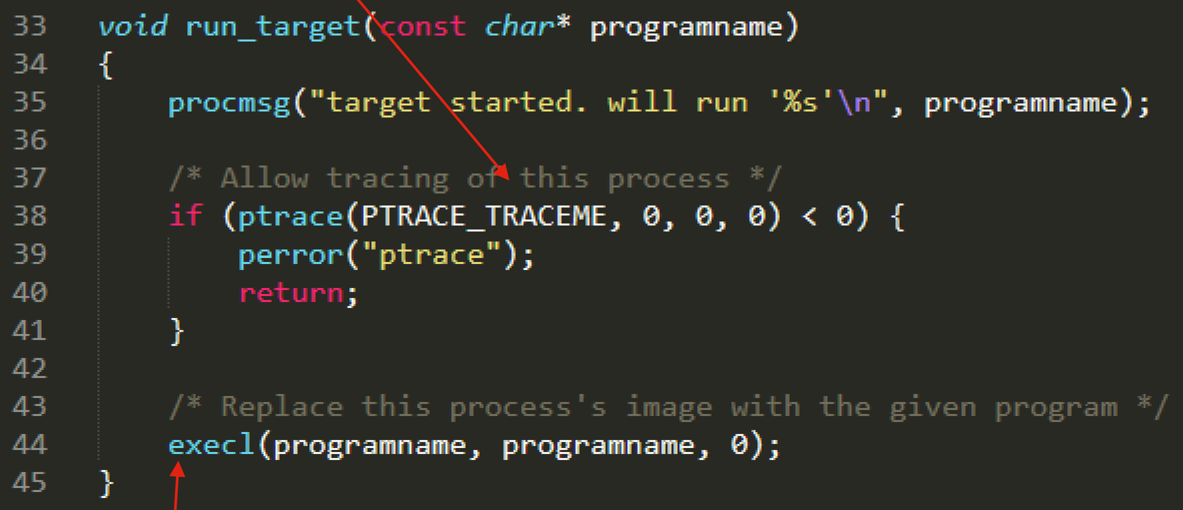
One (child) will execute **`run_target()`**
Other (parent) will execute **`run_debugger()`**

Debugging

debugger.c

Child process allows tracing

```c
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("fork");
        return -1;
    }

    return 0;
}

```



```c
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);
}
```

`execl` will replace the current process image with the binary loaded from the storage.

In this moment, the processes become different.

Debugging

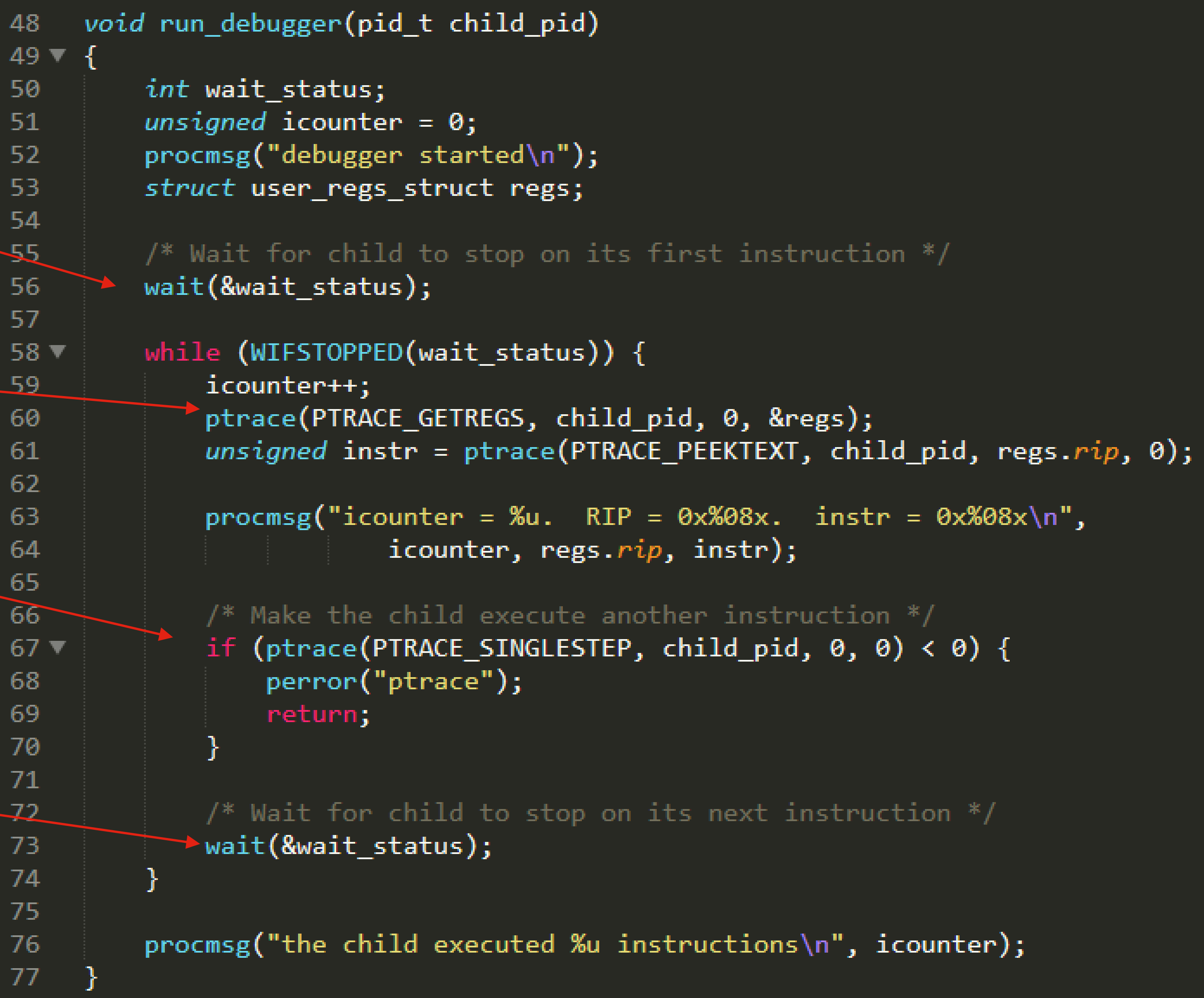
debugger.c

Wait for process to start

Get CPU registers

Single Step through one instruction (ASM)

Wait for instruction to finish



```c
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%08x.  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);
}
```

Sandboxing

- Sandboxing improves the control that debuggers provide
  - Creation of a distinct execution environment
    - Different libraries? Restricted view of the filesystem (minimal access to files)
  - Isolate some actions, providing some safety to analyze malicious applications

- Implementation: lightweight virtual machines or namespaces/containers
  - Supported my mechanisms of the Operating System or additional tools
  - Tools: `sandboxie, pyrebox, panda`

- An agent monitors interactions of the application inside the environment and may allow instrumentation
  - File access, network communication
  - Remote debugging

Emulators

- Emulators are common backends for secur sandboxes
  - May provide much better isolation as the guest and host environments are distinct
    - Kernel is not shared, hardware is emulated
  - Tools: **QEMU**, Virtualbox, Vmware

- Emulation types
  - Full system emulation
  - User mode emulation

Emulators

User Mode Emulation

- Launches **a processes** directly, but on a restricted environment
  - Process may be compiled for one CPU and executed on another CPU
  - Address space is restricted, such as filesystem and libraries available
  - Interaction with Host OS is mediated by the emulator

- Emulator process native CPU instructions (emulation/translation) and:
  - Provide means **to translate syscalls** from guest to host OS
  - Understand intrinsic characteristics such as clone
    - Clone is used to spawn new processes and will require the creation of a new emulation environment
  - Handle signals between analyzed binary and the host system

- May provide integration with debugging tools

Emulators

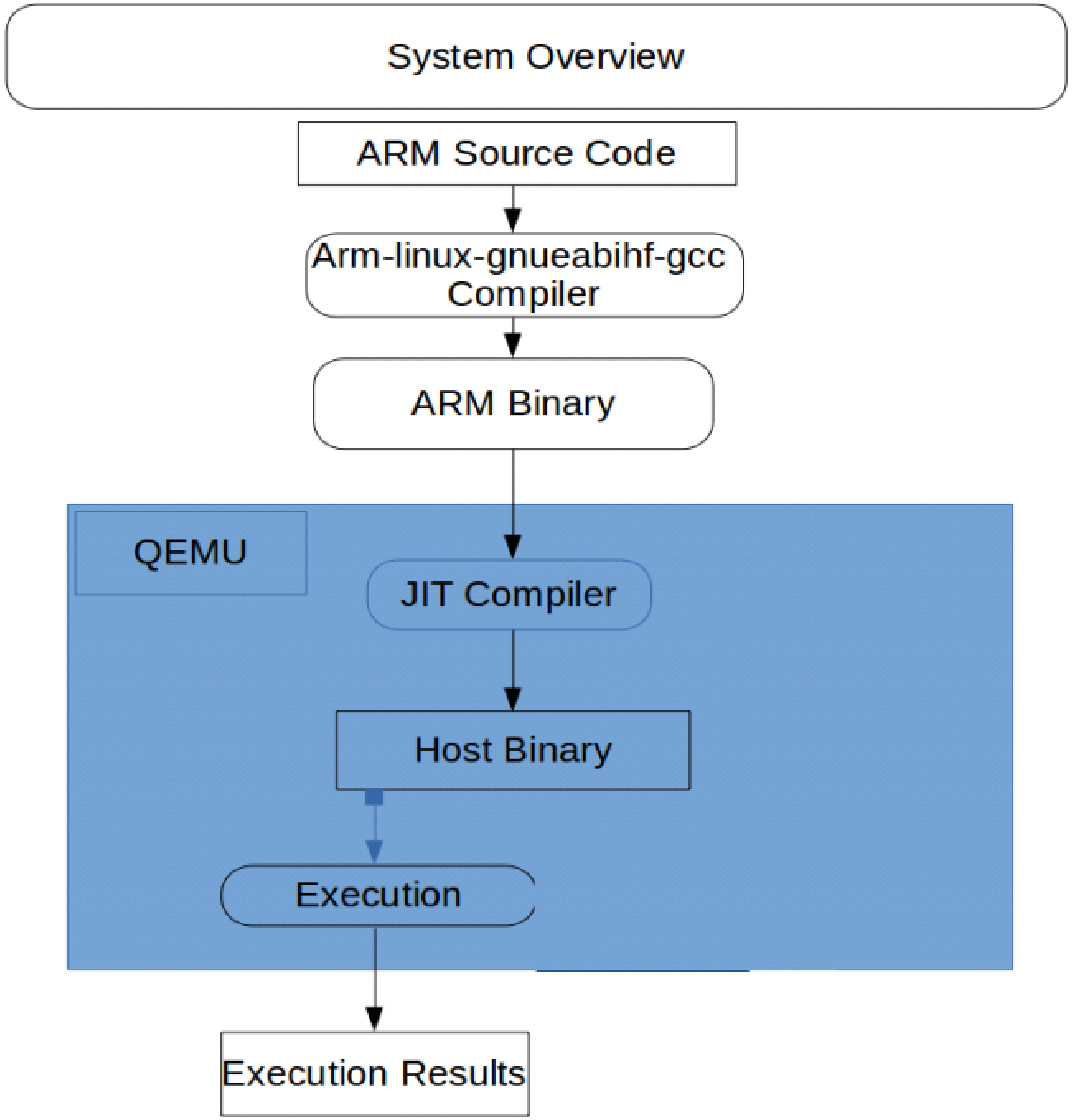
User Mode Emulation with QEMU

- QEMU allows user mode emulation **as long as the OS is kept the same**

- What it does:
  - Machine code translation from any CPU to any CPU
  - Syscall mapping
  - Data structure conversion (Bit-order and Bit-width conversions)
  - Extensive tracing capability to the level of Micro Ops

- Provides a **`gdbserver`** interface for interaction with GDB

- **Usefulness**: reverse engineering applications compiled to other architectures



System Overview

ARM Source Code

Arm-linux-gnueabihf-gcc Compiler

ARM Binary

QEMU

JIT Compiler

Host Binary

Execution

Execution Results

Emulators

Full System Emulation

- Basically: a full-blown virtual machine
  - Emulates a highly configurable set of hardware, including embedded devices
  - Maps interactions to Host resources (screen, disk, network)
  - RE aware software tools expose debugging interfaces (usually to **gdb**)

- Provides the best level of isolation
  - All accesses are mediated by the emulator, reducing the attack surface to emulator components
  - Allows analyzing other binaries besides standard executable files
    - Firmware, MBR, UEFI

- Malware frequently try to detect Virtual Machines, emulators and debuggers…
  - With variable sophistication

Remote debugging with emulators

gdb and gdbserver

- **gdb** can debug remote applications
  - It can even debug remote kernels and firmware
  - Why? Consider embedded devices, software inside an emulator

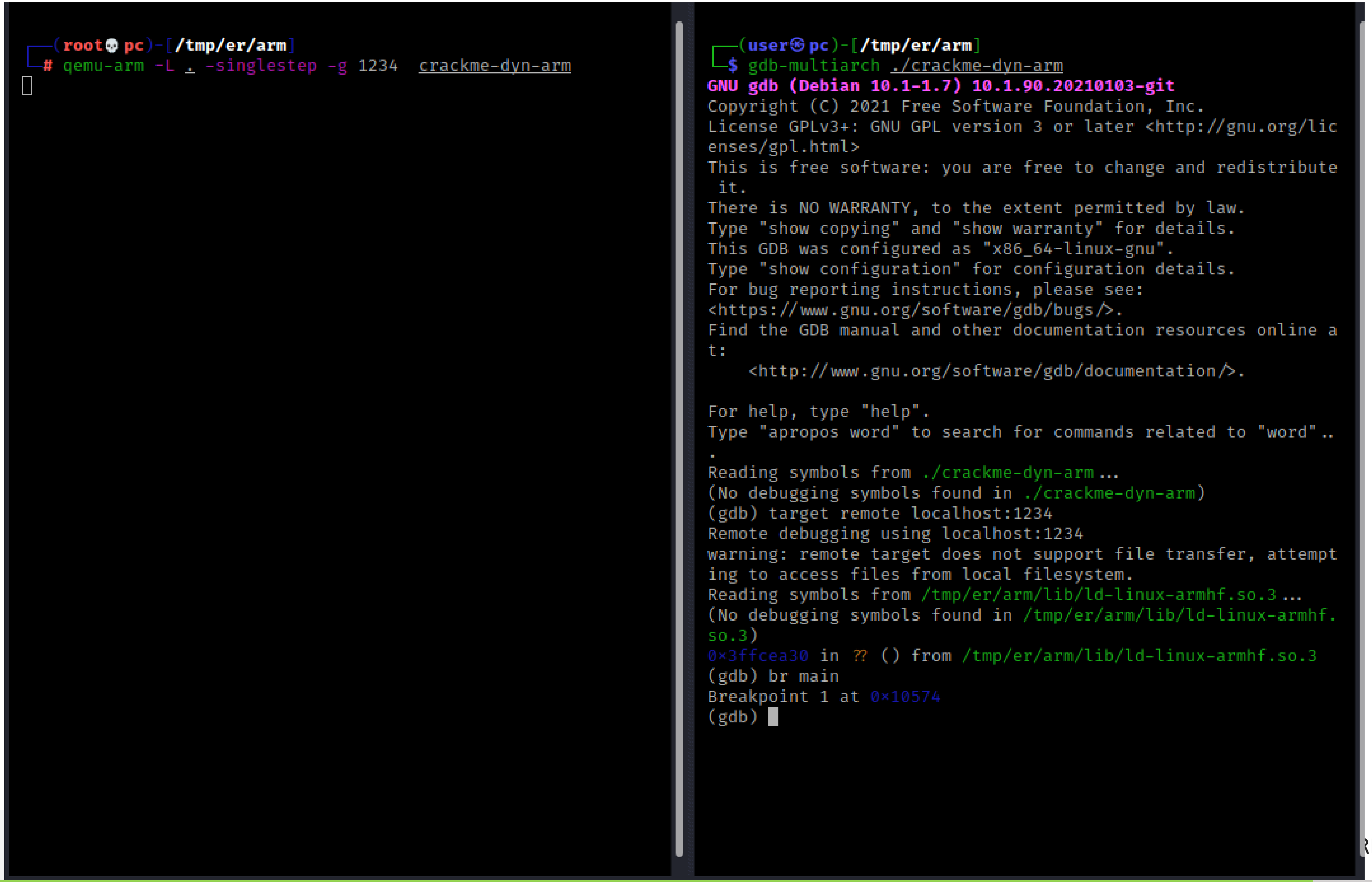
- **gdbserver** is launched on the target system, with the arguments:
  - Either a device name (to use a serial line) or a TCP hostname and portnumber, and the path and filename of the executable to be debugged
  - It then waits passively for the host **gdb** to communicate with it.

- **gdb** is run on the host, with the arguments:
  - The path and filename of the executable (and any sources) on the host, and
  - A device name (for a serial line) or the IP address and port number needed for connection to the target system.

- **Alternative**: the remote application is compiled with a stub that provides a **gdbserver** interface when the application is launched

Example

Reversing an ARM binary



```
┌──(root💀pc)-[/tmp/er/arm]
└─# qemu-arm -L . -singlestep -g 1234  crackme-dyn-arm
▯
```

```
┌──(user㉿pc)-[/tmp/er/arm]
└─$ gdb-multiarch ./crackme-dyn-arm
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/lic
enses/gpl.html>
This is free software: you are free to change and redistribute
 it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online a
t:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"..
.
Reading symbols from ./crackme-dyn-arm...
(No debugging symbols found in ./crackme-dyn-arm)
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: remote target does not support file transfer, attempt
ing to access files from local filesystem.
Reading symbols from /tmp/er/arm/lib/ld-linux-armhf.so.3...
(No debugging symbols found in /tmp/er/arm/lib/ld-linux-armhf.
so.3)
0x3ffcea30 in ?? () from /tmp/er/arm/lib/ld-linux-armhf.so.3
(gdb) br main
Breakpoint 1 at 0x10574
(gdb) █
```

Example

unknown.bin

- Remember the **`unknown.bin`** file?
  - Well… looks like a PDF (is a PDF)
  - but **`$ file unknown.bin`** returns "**`unknown.bin: DOS/MBR boot sector`**"

- What we may extrapolate from that:
  - Seems to be a DOS/Master Boot Record ([Master boot record – Wikipedia]())
  - DOS was only released for i386 (16bits and 32bits)
  - `qemu-system-i386` may boot it if used as a hard disk or floppy disk

Example

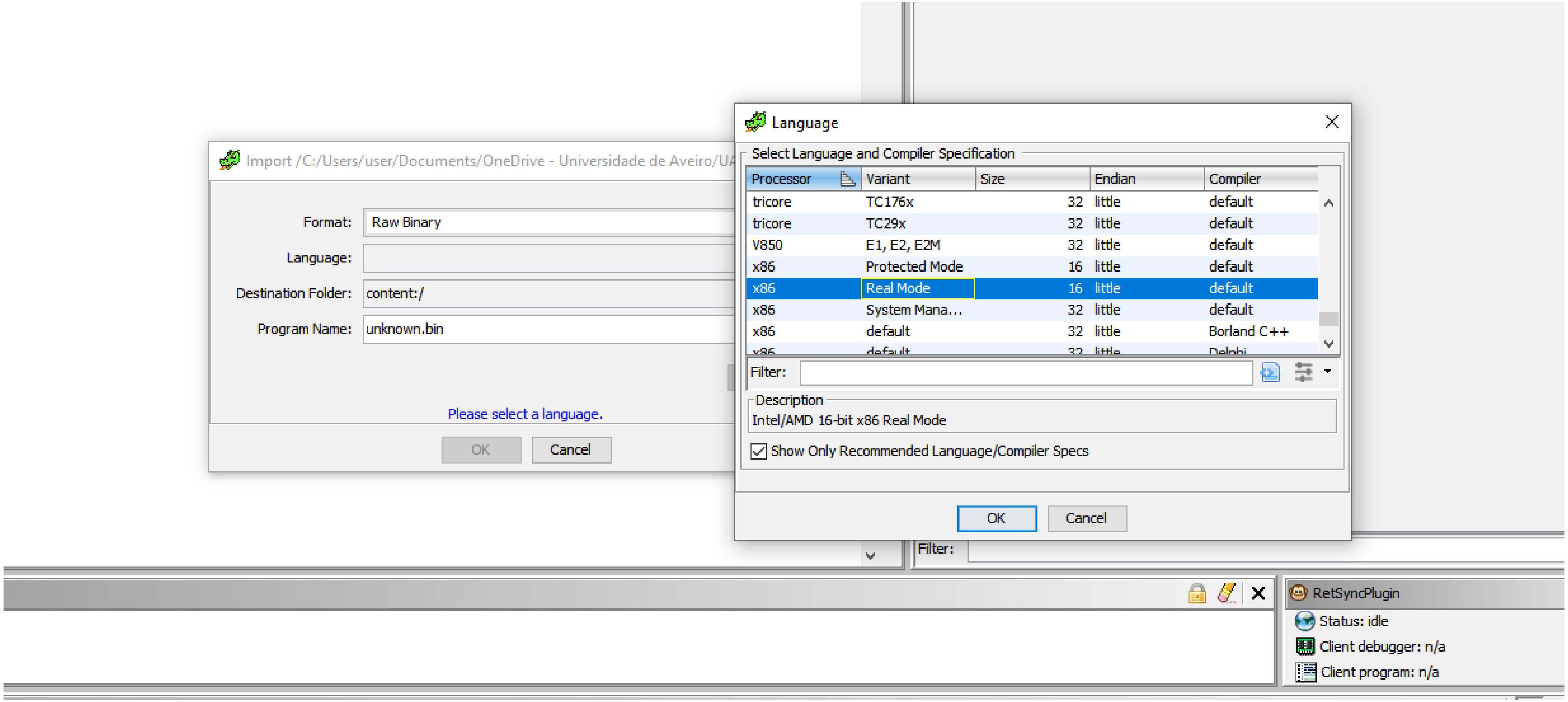
unknown.bin

- How to address such files?
  - Binary files other than ELFs (or PE or other similar) obey to a fixed set of rules
  - It is required to check the datasheets and gather information required to load the file.
  - Important:
    - CPU used, CPU mode, relevant or required peripherals: to know how to decode the binary instructions
    - Program Entry Point: to know where the program starts, and where disassembly should start

- From a Master Boot Record we may know:
  - 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 (pcministry.com)
  - There is no OS running. Input/Output must use BIOS Interrupts

Example

Loading the unknown.bin in ghidra



Import /C:/Users/user/Documents/OneDrive - Universidade de Aveiro/UA

Format: Raw Binary

Language:

Destination Folder: content:/

Program Name: unknown.bin

Please select a language.

Language

Select Language and Compiler Specification

Processor	Variant	Size	Endian	Compiler
tricore	TC176x	32	little	default
tricore	TC29x	32	little	default
V850	E1, E2, E2M	32	little	default
x86	Protected Mode	16	little	default
x86	Real Mode	16	little	default
x86	System Mana…	32	little	default
x86	default	32	little	Borland C++

Filter:

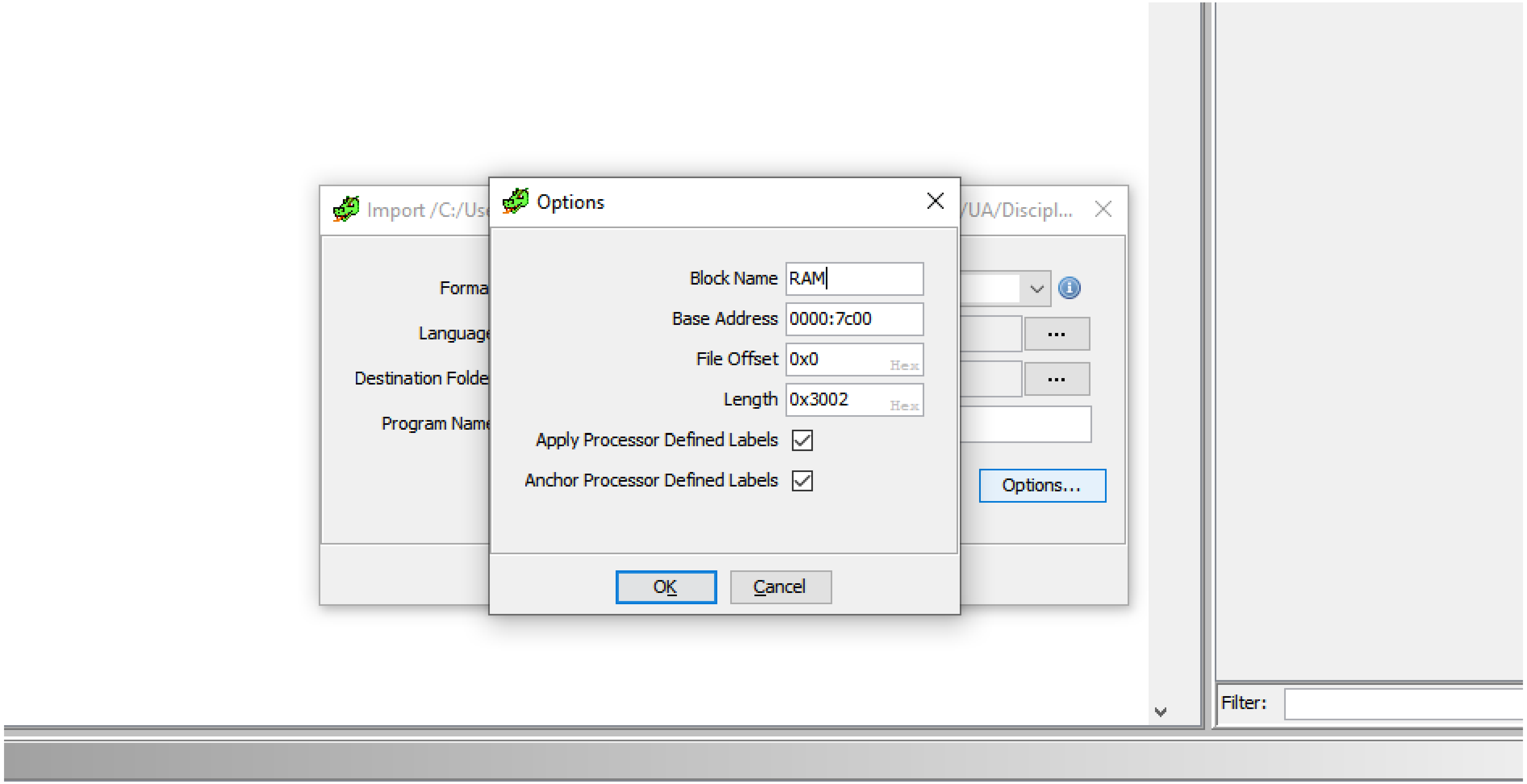
Description
Intel/AMD 16-bit x86 Real Mode

☑ Show Only Recommended Language/Compiler Specs

Status: idle
Client debugger: n/a
Client program: n/a

Example

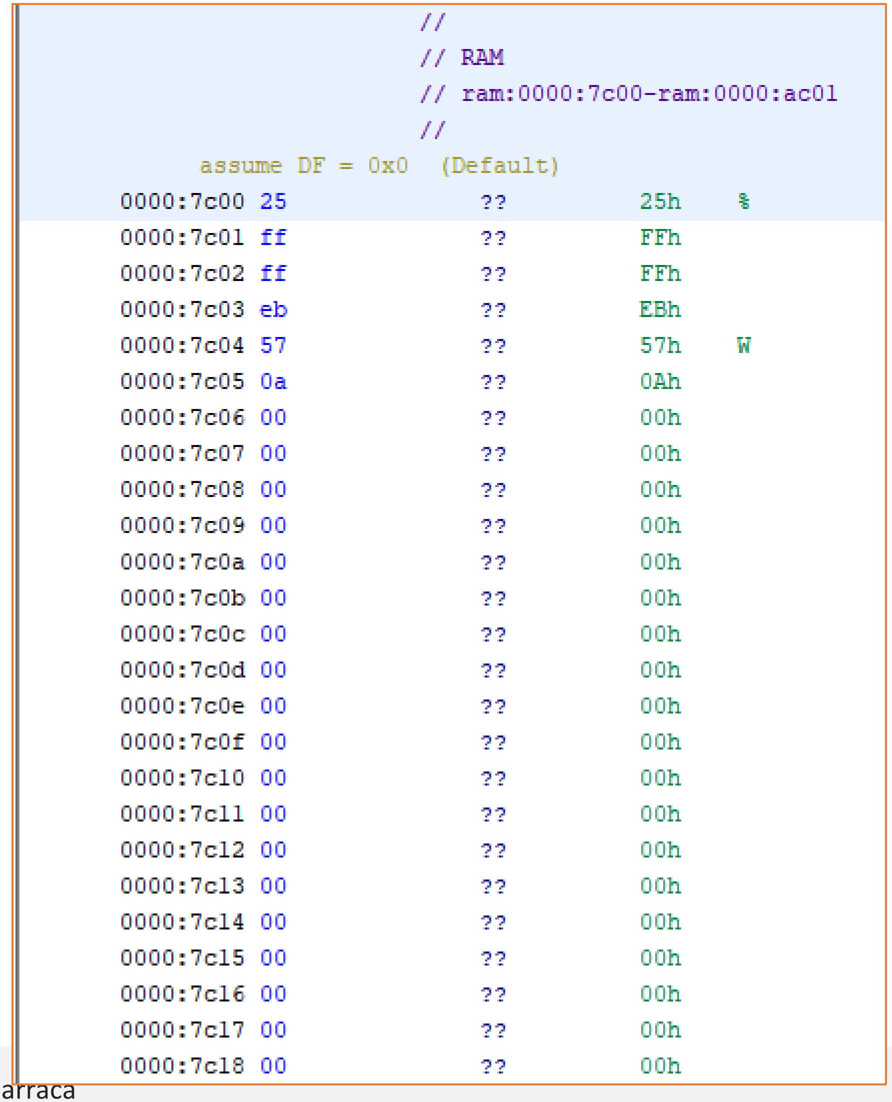
Loading the unknown.bin in ghidra



Block Name: RAM
Base Address: 0000:7c00
File Offset: 0x0
Length: 0x3002
Apply Processor Defined Labels ☑
Anchor Processor Defined Labels ☑

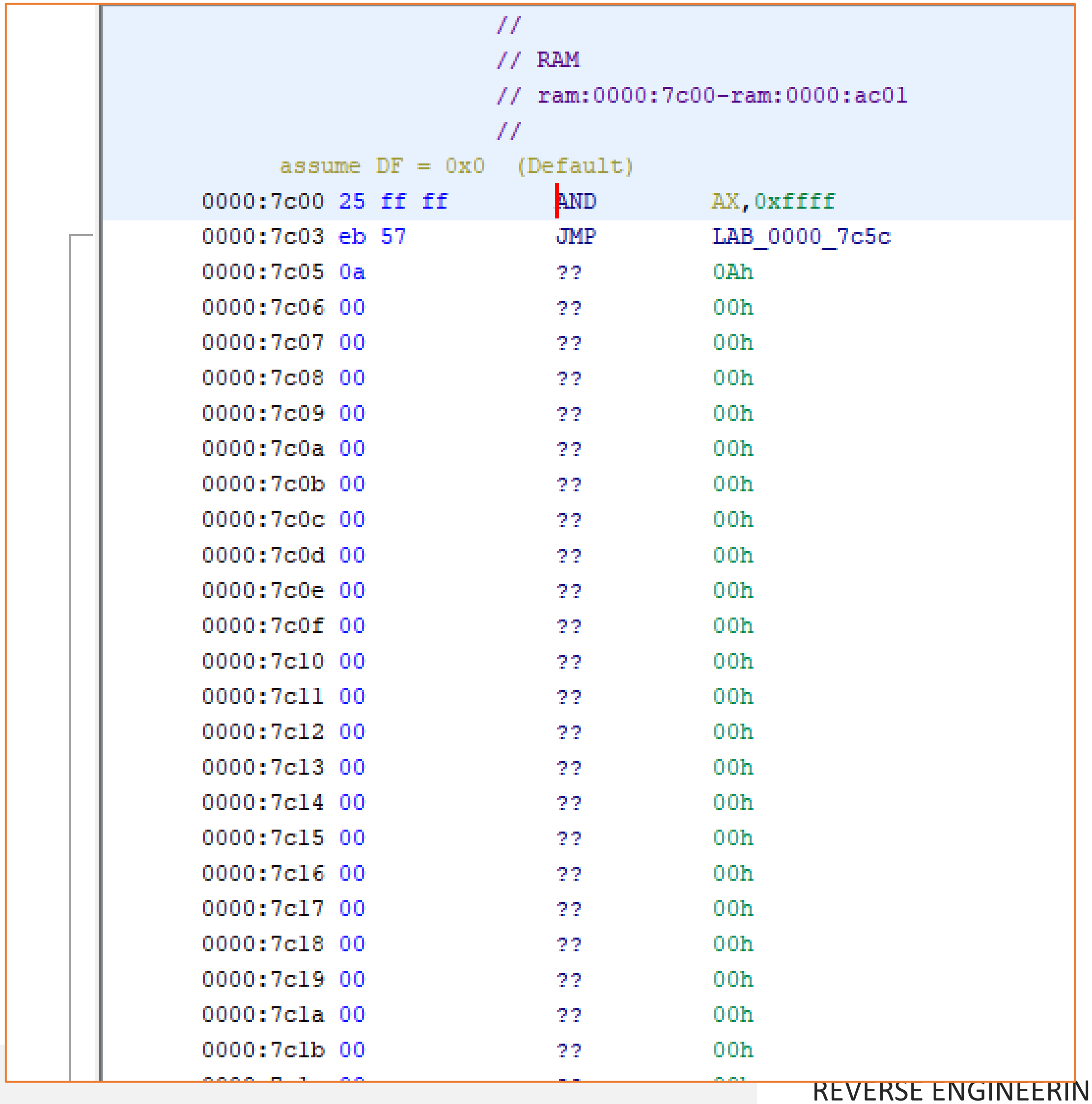
Example

Loading the unknown.bin in ghidra



```
                     //
                     // RAM
                     // ram:0000:7c00-ram:0000:ac01
                     //
        assume DF = 0x0  (Default)
     0000:7c00 25          ??        25h    %
     0000:7c01 ff          ??        FFh
     0000:7c02 ff          ??        FFh
     0000:7c03 eb          ??        EBh
     0000:7c04 57          ??        57h    W
     0000:7c05 0a          ??        0Ah
     0000:7c06 00          ??        00h
     0000:7c07 00          ??        00h
     0000:7c08 00          ??        00h
     0000:7c09 00          ??        00h
     0000:7c0a 00          ??        00h
     0000:7c0b 00          ??        00h
     0000:7c0c 00          ??        00h
     0000:7c0d 00          ??        00h
     0000:7c0e 00          ??        00h
     0000:7c0f 00          ??        00h
     0000:7c10 00          ??        00h
     0000:7c11 00          ??        00h
     0000:7c12 00          ??        00h
     0000:7c13 00          ??        00h
     0000:7c14 00          ??        00h
     0000:7c15 00          ??        00h
     0000:7c16 00          ??        00h
     0000:7c17 00          ??        00h
     0000:7c18 00          ??        00h
```

If we state that 0x7C00 has code, looks like we have something

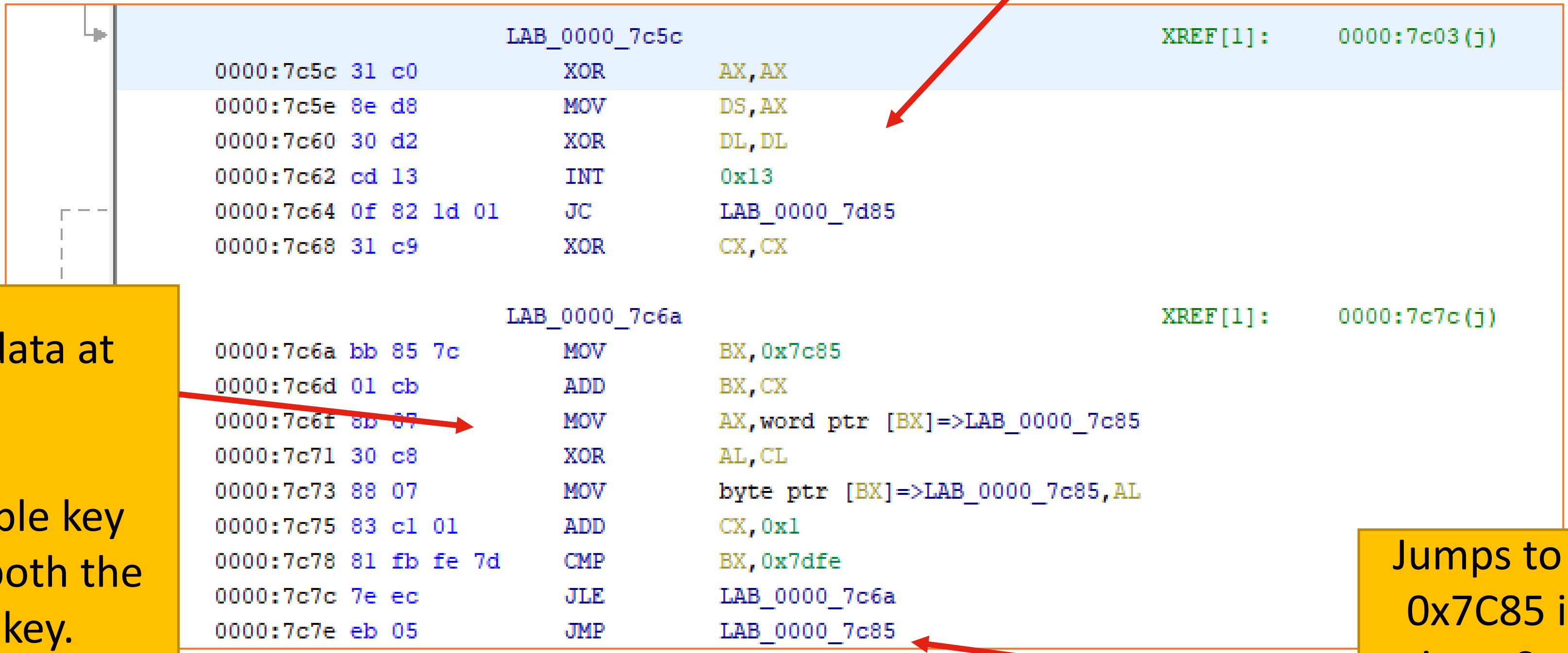


```
                     //
                     // RAM
                     // ram:0000:7c00-ram:0000:ac01
                     //
        assume DF = 0x0  (Default)
     0000:7c00 25 ff ff     AND       AX,0xffff
     0000:7c03 eb 57        JMP       LAB_0000_7c5c
     0000:7c05 0a           ??        0Ah
     0000:7c06 00           ??        00h
     0000:7c07 00           ??        00h
     0000:7c08 00           ??        00h
     0000:7c09 00           ??        00h
     0000:7c0a 00           ??        00h
     0000:7c0b 00           ??        00h
     0000:7c0c 00           ??        00h
     0000:7c0d 00           ??        00h
     0000:7c0e 00           ??        00h
     0000:7c0f 00           ??        00h
     0000:7c10 00           ??        00h
     0000:7c11 00           ??        00h
     0000:7c12 00           ??        00h
     0000:7c13 00           ??        00h
     0000:7c14 00           ??        00h
     0000:7c15 00           ??        00h
     0000:7c16 00           ??        00h
     0000:7c17 00           ??        00h
     0000:7c18 00           ??        00h
     0000:7c19 00           ??        00h
     0000:7c1a 00           ??        00h
     0000:7c1b 00           ??        00h
```

Example

Loading the unknown.bin in ghidra

Some check to int 13H
(HDD or Floppy)



```
                          LAB_0000_7c5c                                   XREF[1]:     0000:7c03(j)
     0000:7c5c 31 c0           XOR        AX,AX
     0000:7c5e 8e d8           MOV        DS,AX
     0000:7c60 30 d2           XOR        DL,DL
     0000:7c62 cd 13           INT        0x13
     0000:7c64 0f 82 1d 01     JC         LAB_0000_7d85
     0000:7c68 31 c9           XOR        CX,CX

                          LAB_0000_7c6a                                   XREF[1]:     0000:7c7c(j)
     0000:7c6a bb 85 7c        MOV        BX,0x7c85
     0000:7c6d 01 cb           ADD        BX,CX
     0000:7c6f 8b 07           MOV        AX,word ptr [BX]=>LAB_0000_7c85
     0000:7c71 30 c8           XOR        AL,CL
     0000:7c73 88 07           MOV        byte ptr [BX]=>LAB_0000_7c85,AL
     0000:7c75 83 c1 01        ADD        CX,0x1
     0000:7c78 81 fb fe 7d     CMP        BX,0x7dfe
     0000:7c7c 7e ec           JLE        LAB_0000_7c6a
     0000:7c7e eb 05           JMP        LAB_0000_7c85
```

A loop XORing data at 0x7C85.

XOR uses a variable key (register CL). It's both the index and the key.

Jumps to 0x7C85 but data at 0x7C85 is decrypted in real time. Static analysis cannot see it… ☹

Must use dynamic analysis ☺

```
for i in range(0x7dfe – 0x7c85):
    ram[0x7c85 + i] ^= i
```

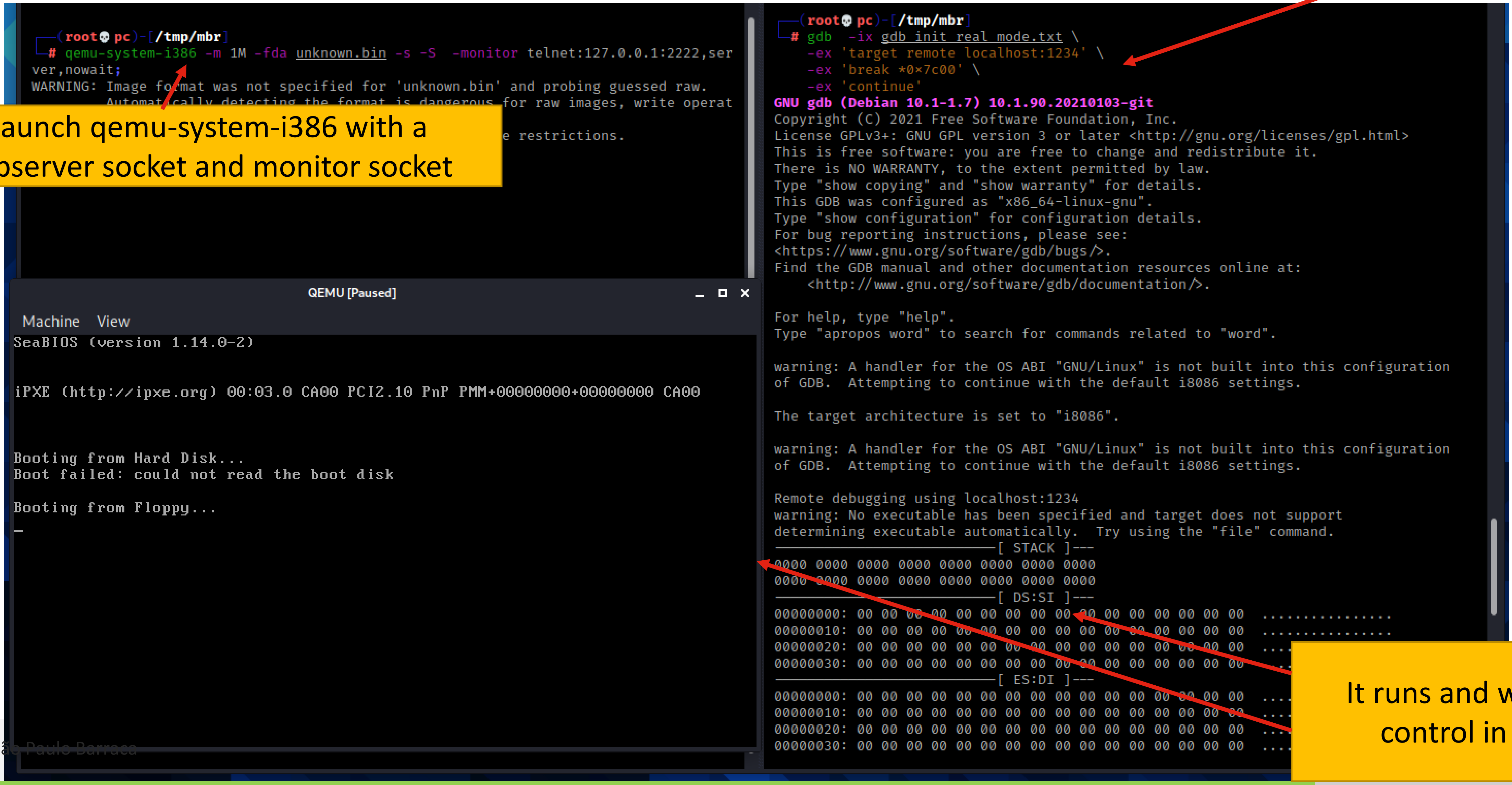
Example

Loading the unknown.bin in qemu with gdb

Execute GDB
Connect to the gdbserver
Do some initialization to set the CPU and display layout

Launch qemu-system-i386 with a gdbserver socket and monitor socket

It runs and we have control in GDB



```
┌──(root💀 pc)-[/tmp/mbr]
└─# qemu-system-i386 -m 1M -fda unknown.bin -s -S  -monitor telnet:127.0.0.1:2222,ser
ver,nowait;
WARNING: Image format was not specified for 'unknown.bin' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operat
                                                              e restrictions.
```

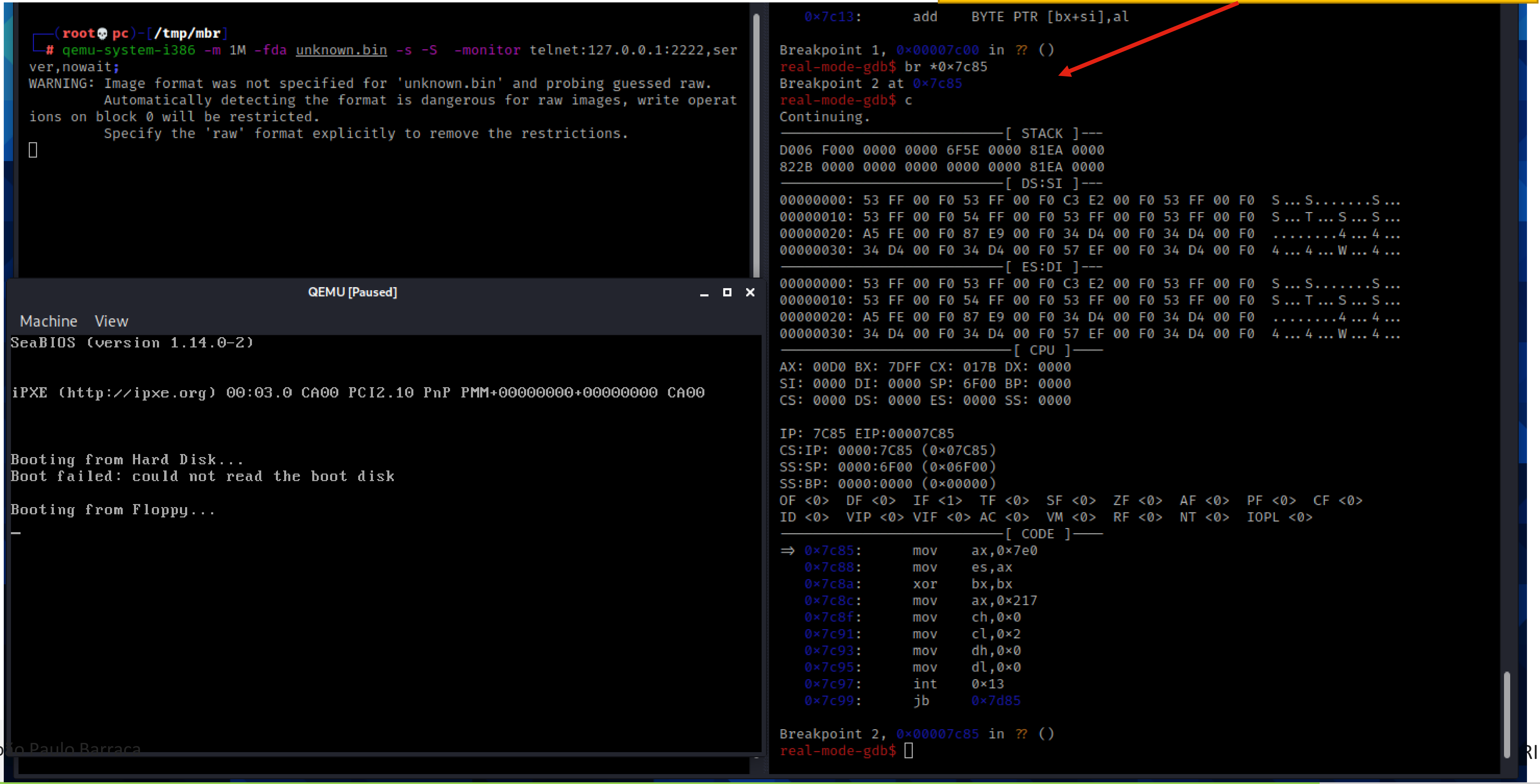
```
SeaBIOS (version 1.14.0-2)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+00000000+00000000 CA00



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
_
```

```
┌──(root💀 pc)-[/tmp/mbr]
└─# gdb  -ix gdb_init_real_mode.txt \
    -ex 'target remote localhost:1234' \
    -ex 'break *0x7c00' \
    -ex 'continue'
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".

warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is set to "i8086".

warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
───────────────────────────[ STACK ]---
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
───────────────────────────[ DS:SI ]---
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
───────────────────────────[ ES:DI ]---
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...
```

Example

Loading the unknown.bin in qemu with gdb

Approach:
- Set a breakpoint to 0x7c85
- Continue (let it decrypt)



```
┌──(root㉿pc)-[/tmp/mbr]
└─# qemu-system-i386 -m 1M -fda unknown.bin -s -S  -monitor telnet:127.0.0.1:2222,ser
ver,nowait;
WARNING: Image format was not specified for 'unknown.bin' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operat
ions on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
▯
```

QEMU [Paused]

Machine   View

```
SeaBIOS (version 1.14.0-2)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+00000000+00000000 CA00



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
_
```

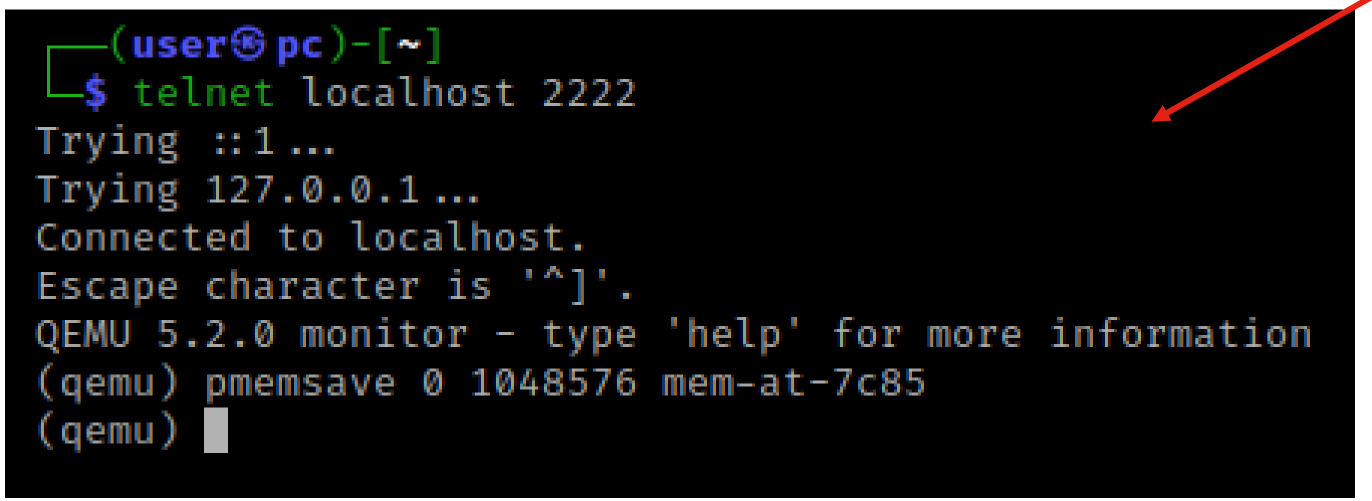
```
   0x7c13:      add    BYTE PTR [bx+si],al

Breakpoint 1, 0x00007c00 in ?? ()
real-mode-gdb$ br *0x7c85
Breakpoint 2 at 0x7c85
real-mode-gdb$ c
Continuing.
───────────────────────────────────────────[ STACK ]───
D006 F000 0000 0000 6F5E 0000 81EA 0000
822B 0000 0000 0000 0000 0000 81EA 0000
───────────────────────────────────────────[ DS:SI ]───
00000000: 53 FF 00 F0 53 FF 00 F0 C3 E2 00 F0 53 FF 00 F0  S...S.......S...
00000010: 53 FF 00 F0 54 FF 00 F0 53 FF 00 F0 53 FF 00 F0  S...T...S...S...
00000020: A5 FE 00 F0 87 E9 00 F0 34 D4 00 F0 34 D4 00 F0  ........4...4...
00000030: 34 D4 00 F0 34 D4 00 F0 57 EF 00 F0 34 D4 00 F0  4...4...W...4...
───────────────────────────────────────────[ ES:DI ]───
00000000: 53 FF 00 F0 53 FF 00 F0 C3 E2 00 F0 53 FF 00 F0  S...S.......S...
00000010: 53 FF 00 F0 54 FF 00 F0 53 FF 00 F0 53 FF 00 F0  S...T...S...S...
00000020: A5 FE 00 F0 87 E9 00 F0 34 D4 00 F0 34 D4 00 F0  ........4...4...
00000030: 34 D4 00 F0 34 D4 00 F0 57 EF 00 F0 34 D4 00 F0  4...4...W...4...
────────────────────────────────────────────[ CPU ]────
AX: 00D0 BX: 7DFF CX: 017B DX: 0000
SI: 0000 DI: 0000 SP: 6F00 BP: 0000
CS: 0000 DS: 0000 ES: 0000 SS: 0000

IP: 7C85 EIP:00007C85
CS:IP: 0000:7C85 (0x07C85)
SS:SP: 0000:6F00 (0x06F00)
SS:BP: 0000:0000 (0x00000)
OF <0>  DF <0>  IF <1>  TF <0>  SF <0>  ZF <0>  AF <0>  PF <0>  CF <0>
ID <0>  VIP <0> VIF <0> AC <0>  VM <0>  RF <0>  NT <0>  IOPL <0>
───────────────────────────────────────────[ CODE ]────
=> 0x7c85:      mov    ax,0x7e0
   0x7c88:      mov    es,ax
   0x7c8a:      xor    bx,bx
   0x7c8c:      mov    ax,0x217
   0x7c8f:      mov    ch,0x0
   0x7c91:      mov    cl,0x2
   0x7c93:      mov    dh,0x0
   0x7c95:      mov    dl,0x0
   0x7c97:      int    0x13
   0x7c99:      jb     0x7d85

Breakpoint 2, 0x00007c85 in ?? ()
real-mode-gdb$ ▯
```

Example

Loading the unknown.bin in qemu with gdb

Connect to the QEMU Control socket
Dump physical RAM (1MB)
This file can be loaded in ghidra and should contain the decrypted code! ☺

Can you recover the flags only with RE? (*)



```
┌──(user㉿pc)-[~]
└─$ telnet localhost 2222
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
QEMU 5.2.0 monitor - type 'help' for more information
(qemu) pmemsave 0 1048576 mem-at-7c85
(qemu) 
```

(*) there may be some additional steps involved. ☺
Analyze CFGs, rename, retype and combine with dynamic analysis whenever relevant
Enjoy the ASCII art and praise @zezadas for the great work with this binary.

Dynamic Binary Instrumentation (DBI)

What are they

- DBI system as an <u>application virtual machine</u> that interprets the ISA of a specific platform
  - usually (but not always) coinciding with the one where the system runs
  - offering instrumentation capabilities to support monitoring and altering instructions and data from an analysis tool component
  - Up to the level of a single instruction

- DBI systems expand standard Dynamic Binary Analysis tasks by
  - Fine grained monitoring capabilities
  - Full control over data and instructions, potentially increasing Reverse Engineering Scope

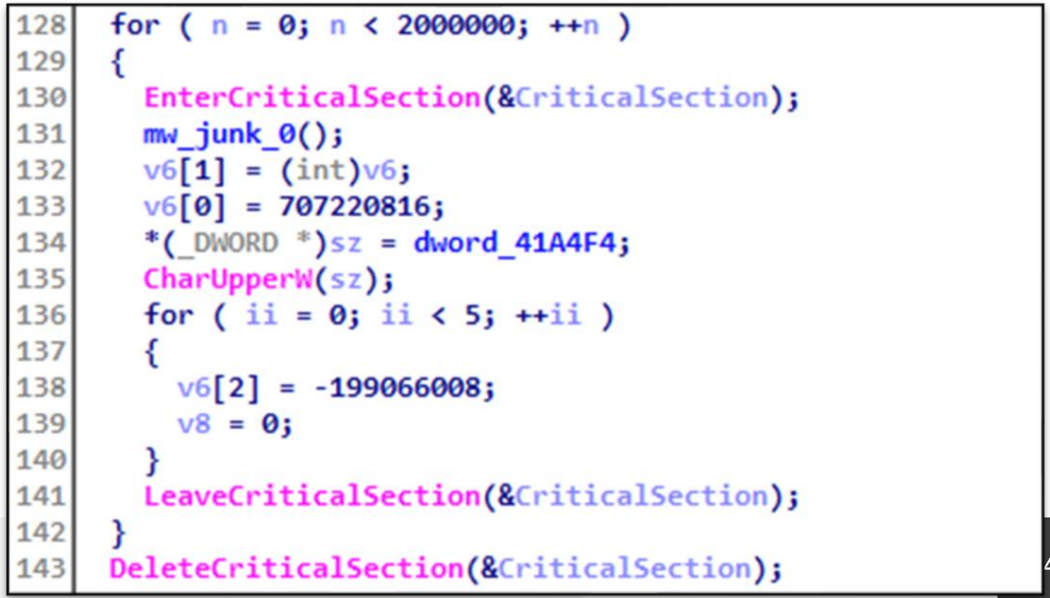
- Uses
  - Measure performance, Detect vulnerabilities, Force code execution, Fuzz binary programs at the scale of a group of instructions

Dynamic Binary Instrumentation (DBI)

caveats

- DBI is vulnerable to specific attacks targeting the emulator
  - Purpose: avoid the use of emulators or induce incorrect results
  - Exploit the fact that DBI tools are slow
  - Exploit the fact that the system is emulated and differs from a real system

- Some approaches
  - Extensive loops Timing measurements
  - Timing measurements
  - Testing for system specific behavior



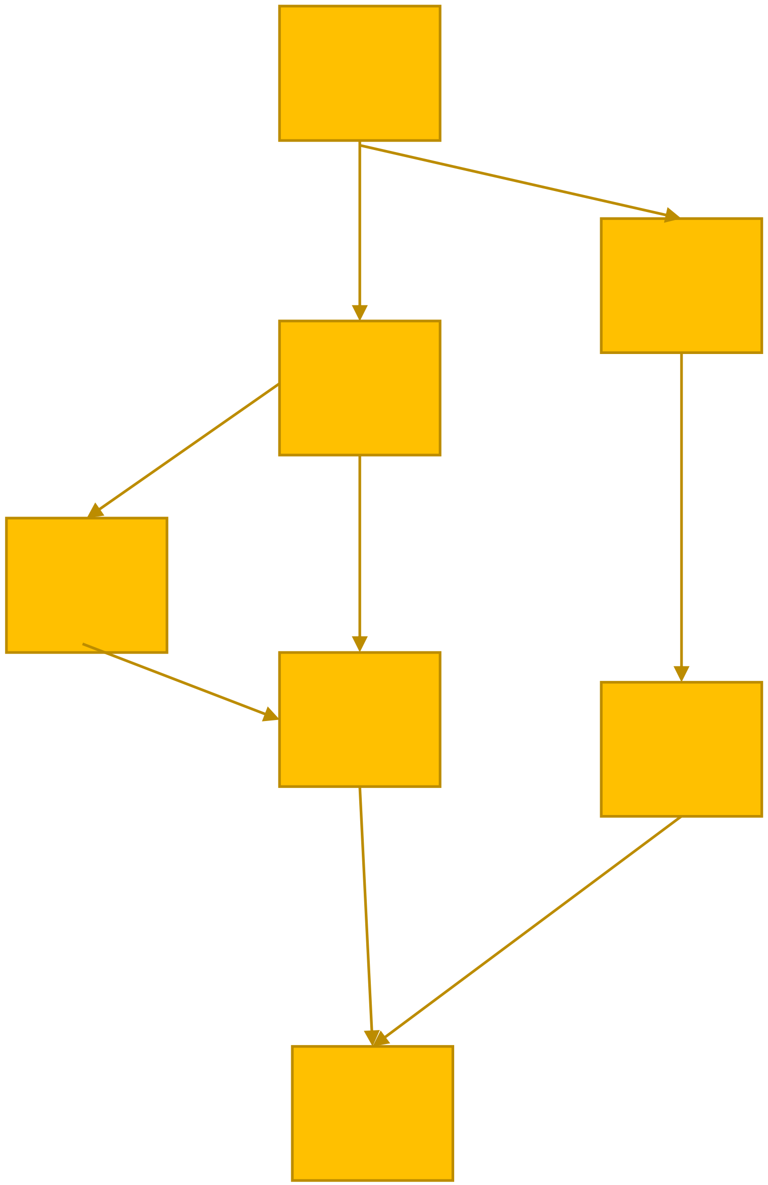
```
for ( n = 0; n < 2000000; ++n )
{
  EnterCriticalSection(&CriticalSection);
  mw_junk_0();
  v6[1] = (int)v6;
  v6[0] = 707220816;
  *(_DWORD *)sz = dword_41A4F4;
  CharUpperW(sz);
  for ( ii = 0; ii < 5; ++ii )
  {
    v6[2] = -199066008;
    v8 = 0;
  }
  LeaveCriticalSection(&CriticalSection);
}
DeleteCriticalSection(&CriticalSection);
```

Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code

- Dynamic Binary Instrumentation
  - “Running” Code

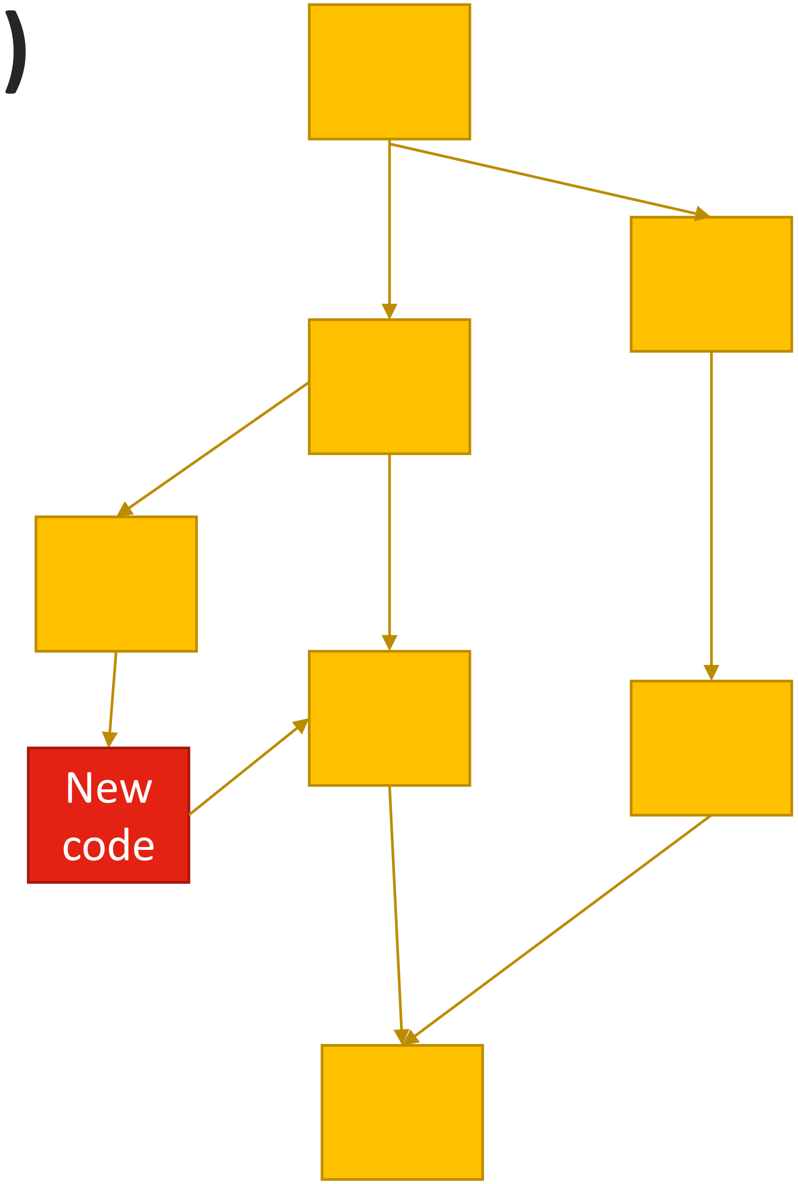


Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code

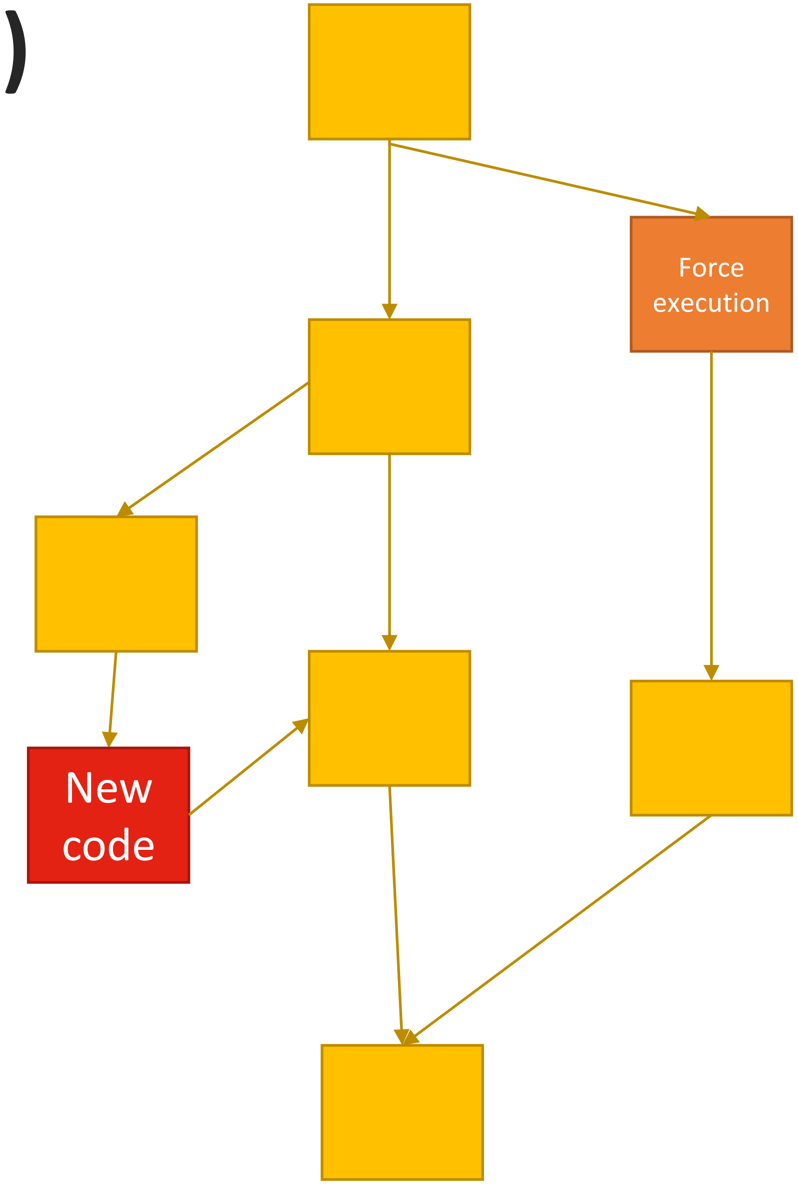
- Dynamic Binary Instrumentation
  - "Running" Code



Dynamic Binary Instrumentation (DBI)

What are they

- Instrumentation
  - Insert Code
- Dynamic Binary Instrumentation
  - “Running” Code



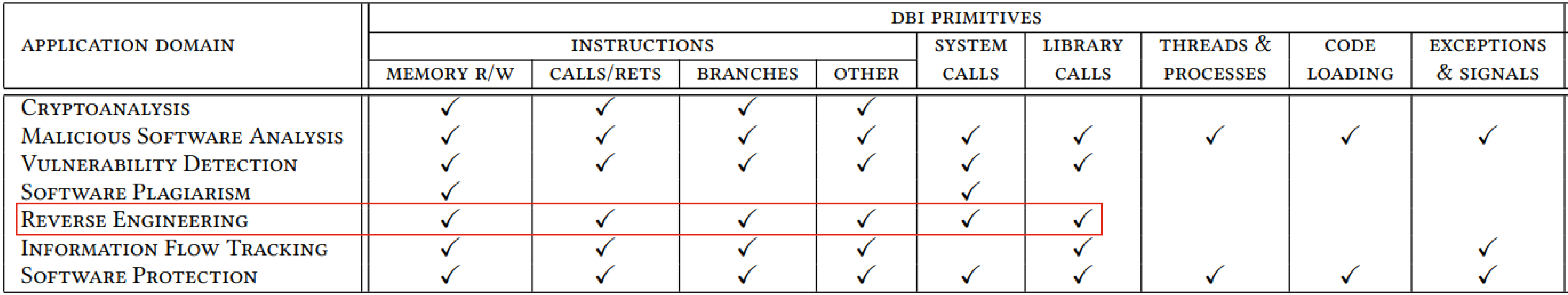
Dynamic Binary Instrumentation (DBI)

How they work?

- Rebuild a program binary code using some JIT technique
  - Insert trace points and hooks for inspection
  - Divert execution to **additional user specified functions**
  - Monitor access to memory regions
    - Potentially triggering callbacks on access
  - May reimplement access to IOs or even **`syscalls`** and interrupts
  - May create a fully Emulated Execution Environment
    - Can be combined with an Emulation platform such as QEMU or Unicorn (a fork from QEMU)

- Popular tools: valgrind, DynamoRIO, Intel PIN, DynInst, Qiling, Frida

Dynamic Binary Instrumentation (DBI)



APPLICATION DOMAIN	DBI PRIMITIVES								
	INSTRUCTIONS				SYSTEM CALLS	LIBRARY CALLS	THREADS & PROCESSES	CODE LOADING	EXCEPTIONS & SIGNALS
	MEMORY R/W	CALLS/RETS	BRANCHES	OTHER					
Cryptoanalysis	✓	✓	✓	✓					
Malicious Software Analysis	✓	✓	✓	✓	✓	✓	✓	✓	✓
Vulnerability Detection	✓	✓	✓	✓	✓	✓			
Software Plagiarism	✓				✓				
Reverse Engineering	✓	✓	✓	✓	✓	✓			
Information Flow Tracking	✓	✓	✓	✓		✓			✓
Software Protection	✓	✓	✓	✓	✓	✓	✓	✓	✓

Daniele D’Elia et al, SoK: Using Dynamic Binary Instrumentation for Security, AsiaCCS, 2019

DBI with Qiling

DBI tool that can perform:

- **Emulation**: Executes binary code step by step, replacing instructions
- **Binary instrumentation**: allows injection of user specified code
- **Cross-platform and cross-architectural analysis**: analyze one architecture or OS on another
- **Sandboxing**: I/O is redirected to fake devices (files, sockets)
- **On raw binaries**: used to analyze blobs from binary devices or shellcode

DBI with Qiling

Emulation

- Syscalls and interrupt are implemented in python
  - Program calls syscall/interrupt
  - Qiling invokes handler in python, which mimics a standard system
  - Implementation can be overridden by the user

- Host OS is never called, and result is provided by Qiling
  - Advantages:
    - Great control over the execution
    - Great isolation
  - Disadvantages:
    - Not all calls are implemented
    - Behavior mimics an ideal system and may deviate from reality

DBI with Qiling

Instrumentation

- User can define hooks to triggering callbacks on an event
  - Because an emulator is translating code in real time, instruction level hooks are possible

- Example
  - Code execution reaches a specific address
  - An address is written or read
  - A function is called, or is leaving
  - An instruction is executed

DBI with Qiling

Cross Platform and Cross Architecture

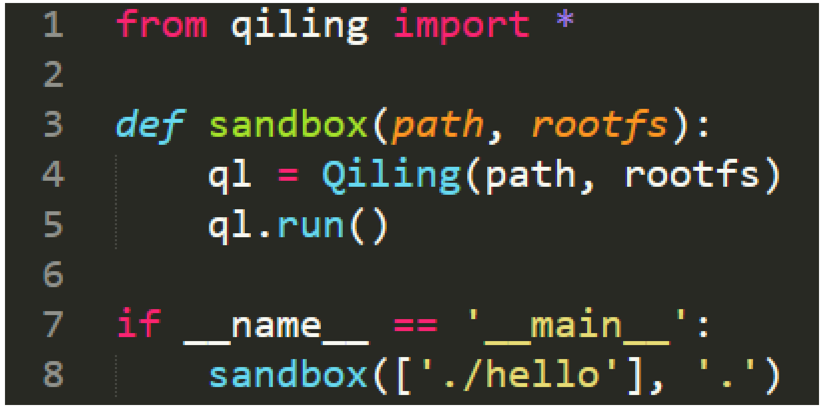
- Binary code is emulated, allowing cross architecture execution
  - Target architecture instructions are compiled to native instructions

- Because all syscalls and interrupts are emulated, host platform can differ from target platform
  - As Qiling is based on Unicorn (Qemu), a wide range of possibilities is available

DBI with Qiling

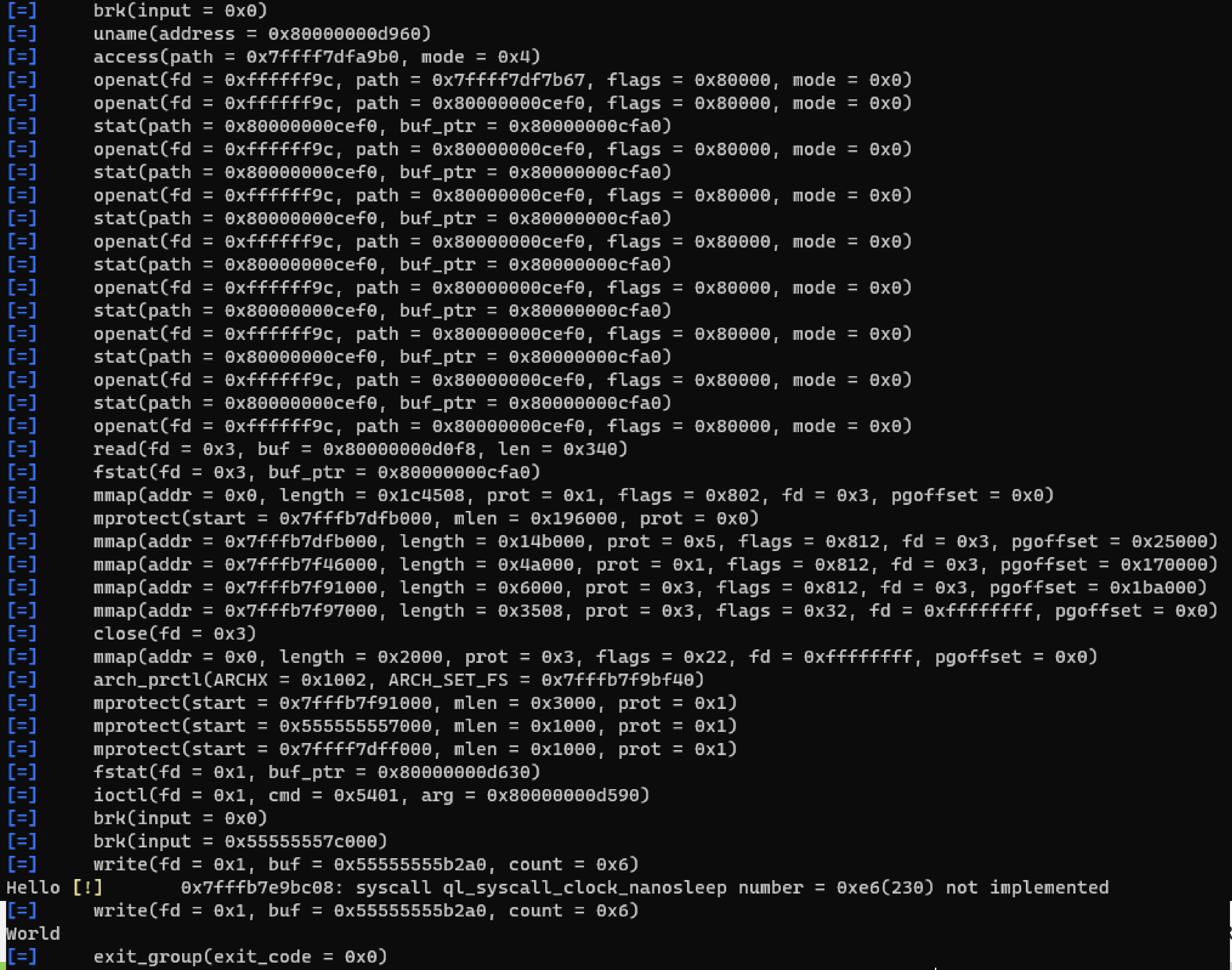
Loading an Elf

- Qiling has several loaders
  - MBR
  - PE, ELF, MachO
  - Unstructured binary (shellcode)



```python
from qiling import *

def sandbox(path, rootfs):
    ql = Qiling(path, rootfs)
    ql.run()

if __name__ == '__main__':
    sandbox(['./hello'], '.')
```

- Loader will make code available to be **emulated** on a secure **rootfs**
  - Calls to interrupts and syscalls are implemented in python

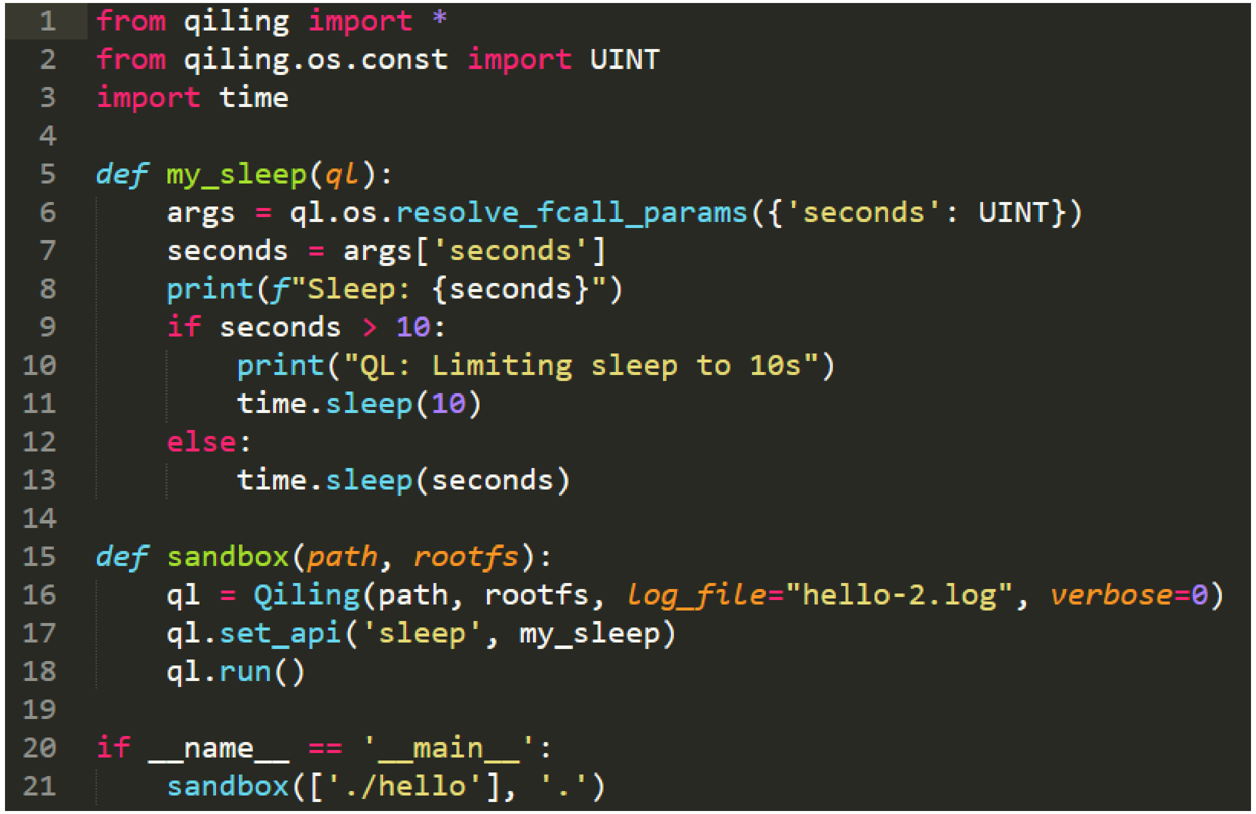


```
[=]     brk(input = 0x0)
[=]     uname(address = 0x80000000d960)
[=]     access(path = 0x7ffff7dfa9b0, mode = 0x4)
[=]     openat(fd = 0xffffff9c, path = 0x7ffff7df7b67, flags = 0x80000, mode = 0x0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     stat(path = 0x80000000cef0, buf_ptr = 0x80000000cfa0)
[=]     openat(fd = 0xffffff9c, path = 0x80000000cef0, flags = 0x80000, mode = 0x0)
[=]     read(fd = 0x3, buf = 0x80000000d0f8, len = 0x340)
[=]     fstat(fd = 0x3, buf_ptr = 0x80000000cfa0)
[=]     mmap(addr = 0x0, length = 0x1c4508, prot = 0x1, flags = 0x802, fd = 0x3, pgoffset = 0x0)
[=]     mprotect(start = 0x7fffb7dfb000, mlen = 0x196000, prot = 0x0)
[=]     mmap(addr = 0x7fffb7dfb000, length = 0x14b000, prot = 0x5, flags = 0x812, fd = 0x3, pgoffset = 0x25000)
[=]     mmap(addr = 0x7fffb7f46000, length = 0x4a000, prot = 0x1, flags = 0x812, fd = 0x3, pgoffset = 0x170000)
[=]     mmap(addr = 0x7fffb7f91000, length = 0x6000, prot = 0x3, flags = 0x812, fd = 0x3, pgoffset = 0x1ba000)
[=]     mmap(addr = 0x7fffb7f97000, length = 0x3508, prot = 0x3, flags = 0x32, fd = 0xffffffff, pgoffset = 0x0)
[=]     close(fd = 0x3)
[=]     mmap(addr = 0x0, length = 0x2000, prot = 0x3, flags = 0x22, fd = 0xffffffff, pgoffset = 0x0)
[=]     arch_prctl(ARCHX = 0x1002, ARCH_SET_FS = 0x7fffb7f9bf40)
[=]     mprotect(start = 0x7fffb7f91000, mlen = 0x3000, prot = 0x1)
[=]     mprotect(start = 0x555555557000, mlen = 0x1000, prot = 0x1)
[=]     mprotect(start = 0x7ffff7dff000, mlen = 0x1000, prot = 0x1)
[=]     fstat(fd = 0x1, buf_ptr = 0x80000000d630)
[=]     ioctl(fd = 0x1, cmd = 0x5401, arg = 0x80000000d590)
[=]     brk(input = 0x0)
[=]     brk(input = 0x55555557c000)
[=]     write(fd = 0x1, buf = 0x5555555b2a0, count = 0x6)
Hello [!]       0x7fffb7e9bc08: syscall ql_syscall_clock_nanosleep number = 0xe6(230) not implemented
[=]     write(fd = 0x1, buf = 0x5555555b2a0, count = 0x6)
World
[=]     exit_group(exit_code = 0x0)
```

DBI with Qiling

Overriding a library function

- Functions can be overridden with custom implementations
  - Code can access arguments of basic types (Strings, Ints, Floats)
  - Inside function, other external functions can be called
  - Entire set of registries and memory can be manipulated
  - Return is provided to the calling function to be **emulated** on a secure **rootfs**
  - Calls to interrupts and syscalls are implemented in python



```python
from qiling import *
from qiling.os.const import UINT
import time

def my_sleep(ql):
    args = ql.os.resolve_fcall_params({'seconds': UINT})
    seconds = args['seconds']
    print(f"Sleep: {seconds}")
    if seconds > 10:
        print("QL: Limiting sleep to 10s")
        time.sleep(10)
    else:
        time.sleep(seconds)

def sandbox(path, rootfs):
    ql = Qiling(path, rootfs, log_file="hello-2.log", verbose=0)
    ql.set_api('sleep', my_sleep)
    ql.run()

if __name__ == '__main__':
    sandbox(['./hello'], '.')
```