Concepts and Tooling for Reverse Engineering

Florian Magin <fmagin@ernw.de>

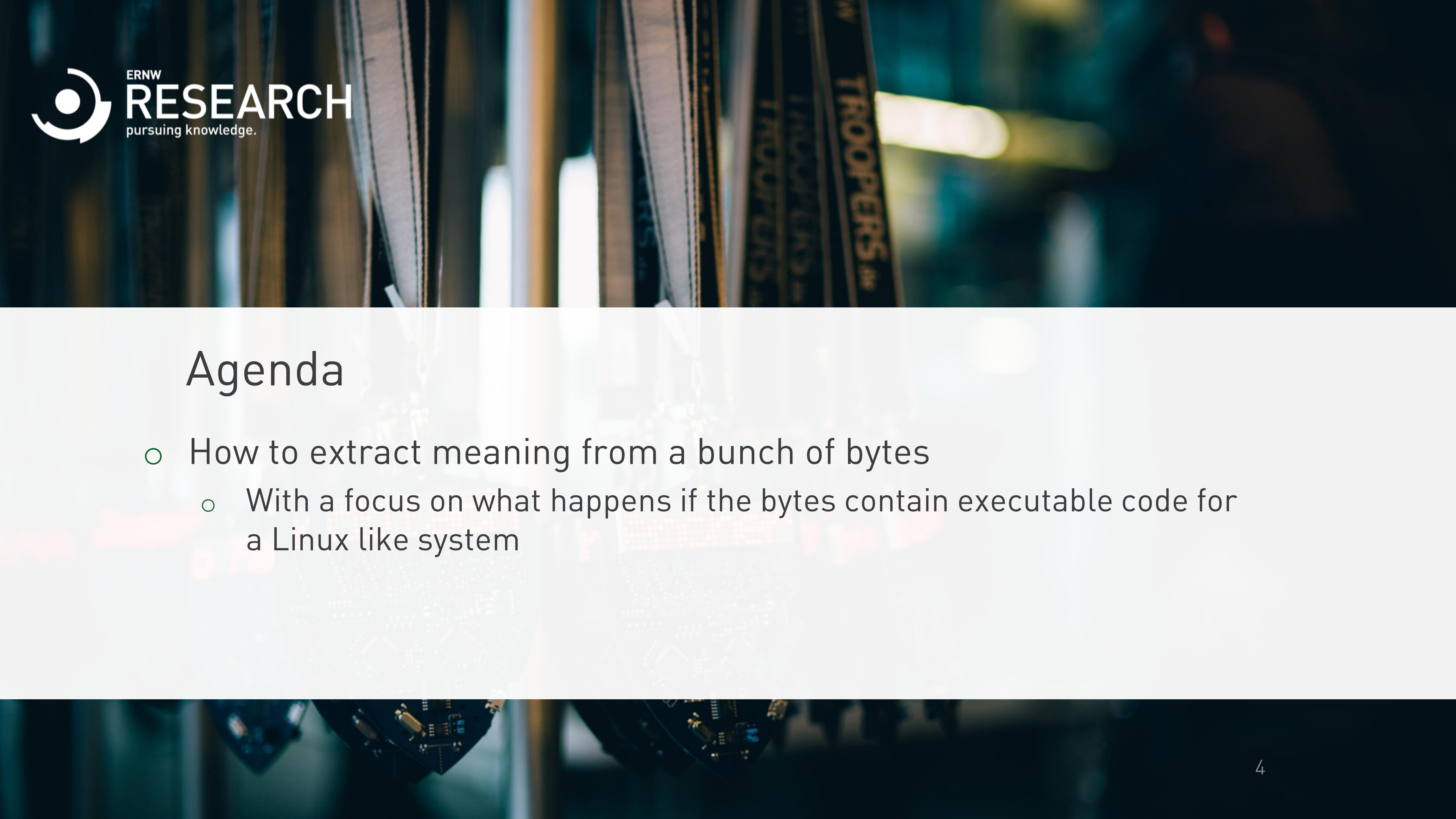
whoami

- Security Researcher at ERNW Research GmbH from Heidelberg, Germany
- Organizer of the Wizards of Dos CTF team from Darmstadt, Germany
- Reach me via:
  - Twitter: @0x464D
  - Email: fmagin@ernw.de

TROOPERS
MAKE THE WORLD A SAFER PLACE

Who we are

- Germany-based ERNW GmbH
- Blog: *www.insinuator.net*
- Conference: *www.troopers.de*



Agenda

- How to extract meaning from a bunch of bytes
  - With a focus on what happens if the bytes contain executable code for a Linux like system

Bunch of Bytes

- Find Patterns
  - The human brain is really good at that
- Throw some byte sequences into a search machine
- Contextualize
  - In most cases you know the rough context
- Just call ‘file’ or ‘binwalk’ on it
- Find a good enough parser
- If there is none, generate your own
  - More on that later

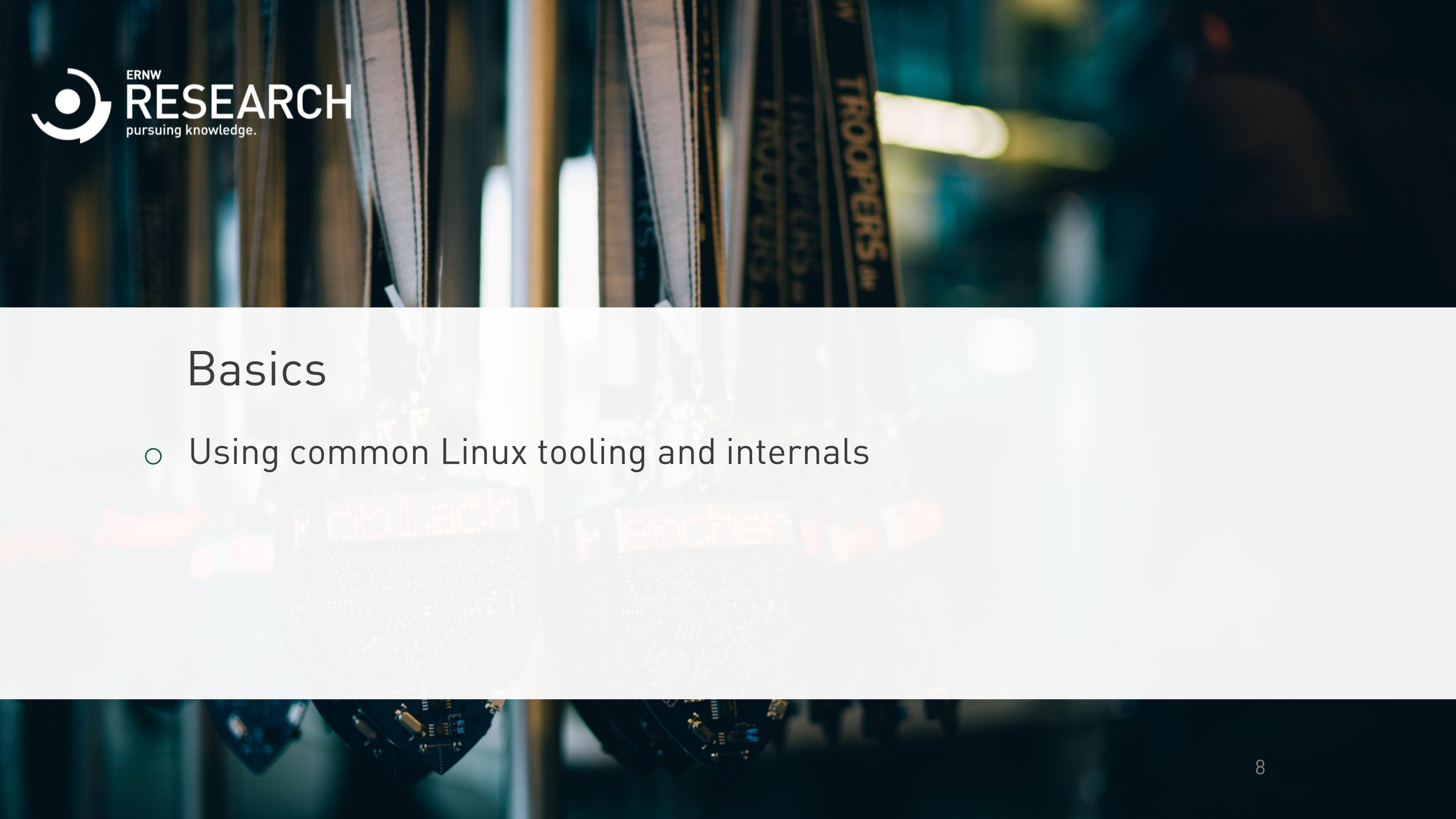
https://github.com/ReFirmLabs/binwalk

First Steps

- If it’s in a typical executable format
  - You are lucky
  - Plenty of parsers and support
  - Most information already available
- If it’s firmware
  - Manual work required

Firmware Information

- Determine the Architecture
  - Datasheet
  - Heuristics like grepping for function prologues/epilogues for various CPUs/CCs
- Determine Memory Layout
  - Datasheet
  - Memory Dump



Basics

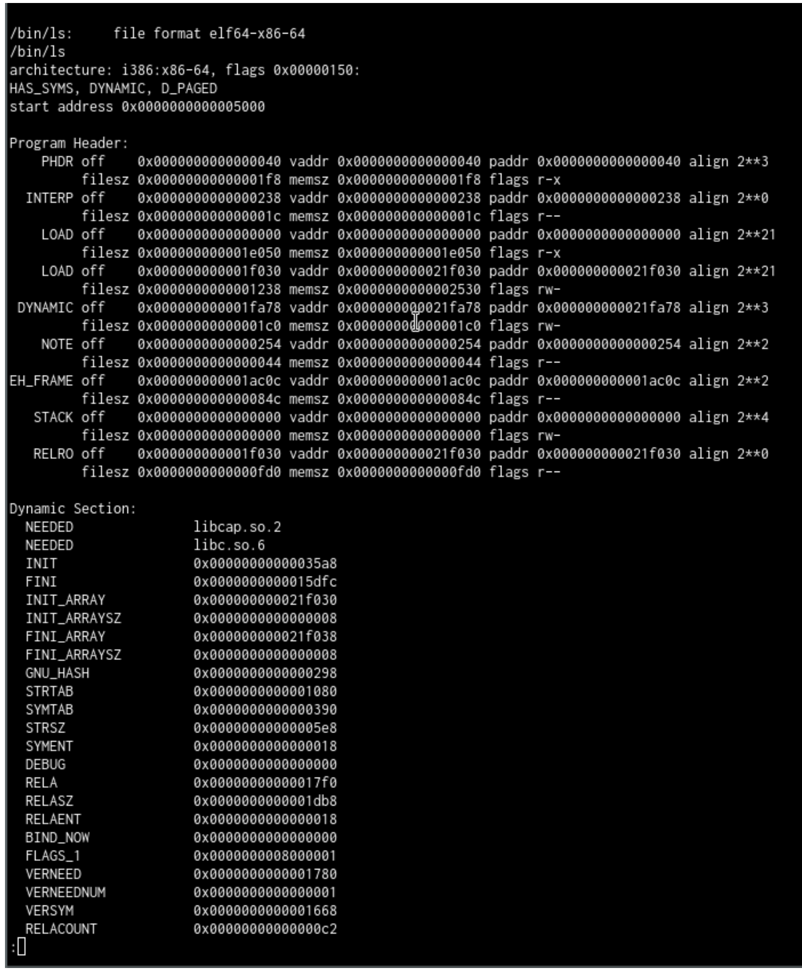
- Using common Linux tooling and internals

Executable Parsing

- ELF: Executable and Linking Format
- PE/MZ
- Mach-O
- Header contains all the information for the loader to setup the program
  - Memory layout
  - Entry point
  - Dependencies
  - etc
- Relevant:
  - External Library and Function names
  - Symbol Table if available

Executable Parsing: Tools

- Many overlapping tools
- ‘objdump’ if you want to get a first look
- Other tools will take care of this for you



```
/bin/ls:     file format elf64-x86-64
/bin/ls
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000005000

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000000238 paddr 0x0000000000000238 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**21
         filesz 0x000000000001e050 memsz 0x000000000001e050 flags r-x
    LOAD off    0x000000000001f030 vaddr 0x000000000021f030 paddr 0x000000000021f030 align 2**21
         filesz 0x0000000000001238 memsz 0x0000000000002530 flags rw-
 DYNAMIC off    0x000000000001fa78 vaddr 0x000000000021fa78 paddr 0x000000000021fa78 align 2**3
         filesz 0x00000000000001c0 memsz 0x00000000000001c0 flags rw-
    NOTE off    0x0000000000000254 vaddr 0x0000000000000254 paddr 0x0000000000000254 align 2**2
         filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
EH_FRAME off    0x000000000001ac0c vaddr 0x000000000001ac0c paddr 0x000000000001ac0c align 2**2
         filesz 0x000000000000084c memsz 0x000000000000084c flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
   RELRO off    0x000000000001f030 vaddr 0x000000000021f030 paddr 0x000000000021f030 align 2**0
         filesz 0x0000000000000fd0 memsz 0x0000000000000fd0 flags r--

Dynamic Section:
  NEEDED               libcap.so.2
  NEEDED               libc.so.6
  INIT                 0x00000000000035a8
  FINI                 0x0000000000015dfc
  INIT_ARRAY           0x000000000021f030
  INIT_ARRAYSZ         0x0000000000000008
  FINI_ARRAY           0x000000000021f038
  FINI_ARRAYSZ         0x0000000000000008
  GNU_HASH             0x0000000000000298
  STRTAB               0x0000000000001080
  SYMTAB               0x0000000000000390
  STRSZ                0x00000000000005e8
  SYMENT               0x0000000000000018
  DEBUG                0x0000000000000000
  RELA                 0x00000000000017f0
  RELASZ               0x0000000000001db8
  RELAENT              0x0000000000000018
  BIND_NOW             0x0000000000000000
  FLAGS_1              0x0000000008000001
  VERNEED              0x0000000000001780
  VERNEEDNUM           0x0000000000000001
  VERSYM               0x0000000000001668
  RELACOUNT            0x00000000000000c2
:
```

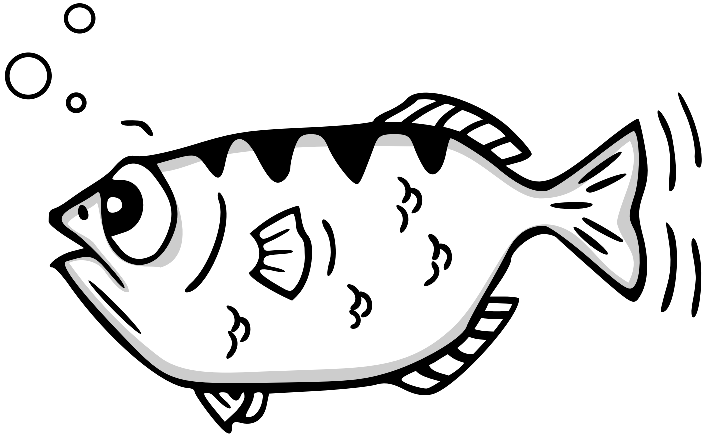
https://sourceware.org/binutils/

Tracing

- Running the program and collecting information
  - Called Library Functions (with Arguments!) with ‘ltrace’
  - Systemcalls (Files opened) with ‘strace’
- Examples:
  - Binary deobfuscates some hostname and connects to it, so check for the ‘connect’ systemcalls
  - If some application just hangs the last syscall or library call might give you a hint

Basic Runtime Influence: GDB

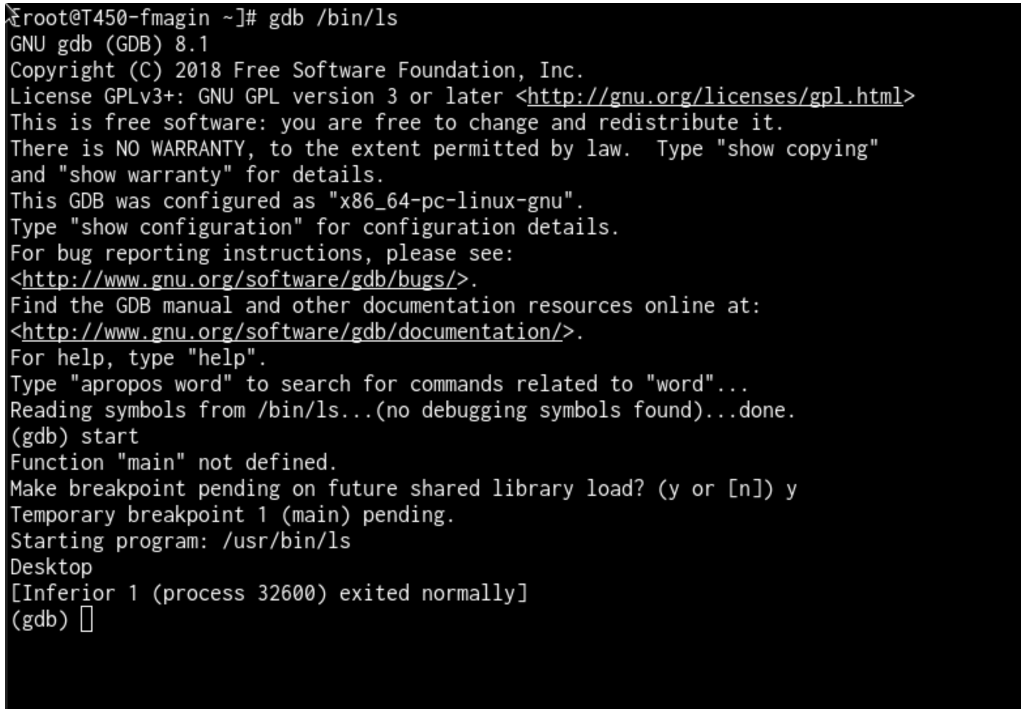
- GDB: GNU Debugger
  - Great for working with debugging symbols
  - Painful without them
- Can be enough for basic tasks on its own
  - Stop execution at certain addresses
  - Inspect registers and memory
- Plugins that help with analysis
  - https://github.com/longld/peda
  - https://github.com/pwndbg/pwndbg
  - https://github.com/hugsy/gef



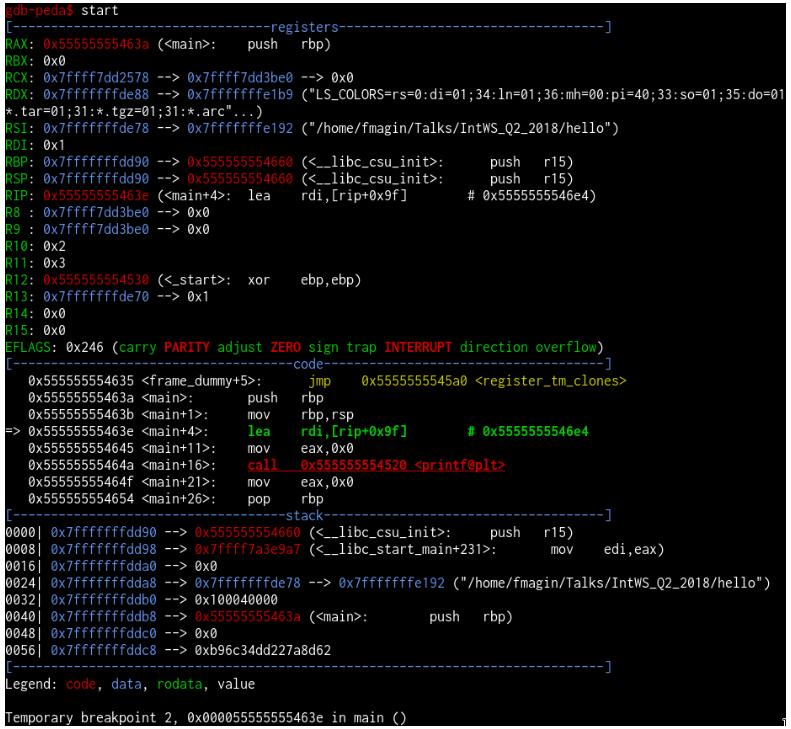
Source: https://www.gnu.org/software/gdb/mascot/

https://www.gnu.org/software/gdb/

Vanilla GDB vs Plugins



```
[root@T450-fmagin ~]# gdb /bin/ls
GNU gdb (GDB) 8.1
Copyright (C) 2018 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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"...
Reading symbols from /bin/ls...(no debugging symbols found)...done.
(gdb) start
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Temporary breakpoint 1 (main) pending.
Starting program: /usr/bin/ls
Desktop
[Inferior 1 (process 32600) exited normally]
(gdb) 
```

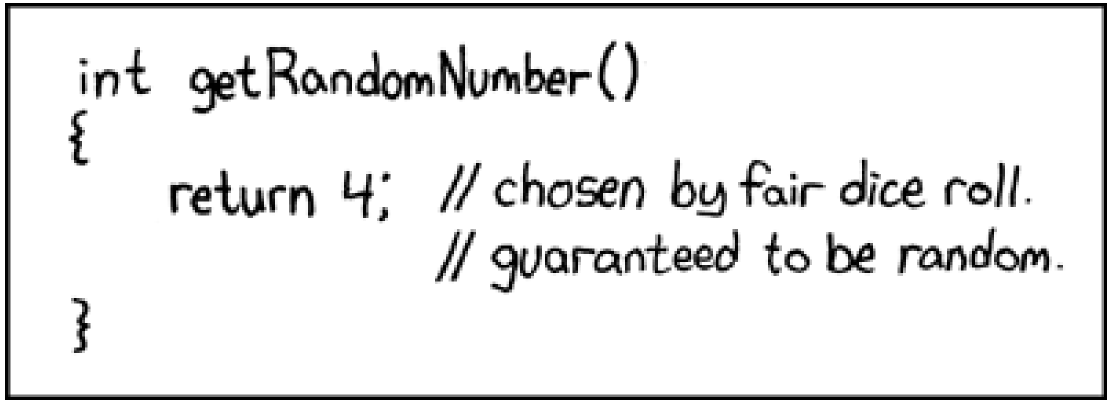


```
gdb-peda$ start
[----------------------------------registers-----------------------------------]
RAX: 0x55555555463a (<main>:	push   rbp)
RBX: 0x0
RCX: 0x7ffff7dd2578 --> 0x7ffff7dd3be0 --> 0x0
RDX: 0x7fffffffde88 --> 0x7fffffffe1b9 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01
*.tar=01;31:*.tgz=01;31:*.arc"...)
RSI: 0x7fffffffde78 --> 0x7fffffffe192 ("/home/fmagin/Talks/IntWS_Q2_2018/hello")
RDI: 0x1
RBP: 0x7fffffffdd90 --> 0x555555554660 (<__libc_csu_init>:	push   r15)
RSP: 0x7fffffffdd90 --> 0x555555554660 (<__libc_csu_init>:	push   r15)
RIP: 0x55555555463e (<main+4>:	lea    rdi,[rip+0x9f]        # 0x5555555546e4)
R8 : 0x7ffff7dd3be0 --> 0x0
R9 : 0x7ffff7dd3be0 --> 0x0
R10: 0x2
R11: 0x3
R12: 0x555555554530 (<_start>:	xor    ebp,ebp)
R13: 0x7fffffffde70 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x555555554635 <frame_dummy+5>:	jmp    0x5555555545a0 <register_tm_clones>
   0x55555555463a <main>:	push   rbp
   0x55555555463b <main+1>:	mov    rbp,rsp
=> 0x55555555463e <main+4>:	lea    rdi,[rip+0x9f]        # 0x5555555546e4
   0x555555554645 <main+11>:	mov    eax,0x0
   0x55555555464a <main+16>:	call   0x555555554520 <printf@plt>
   0x55555555464f <main+21>:	mov    eax,0x0
   0x555555554654 <main+26>:	pop    rbp
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd90 --> 0x555555554660 (<__libc_csu_init>:	push   r15)
0008| 0x7fffffffdd98 --> 0x7ffff7a3e9a7 (<__libc_start_main+231>:	mov    edi,eax)
0016| 0x7fffffffdda0 --> 0x0
0024| 0x7fffffffdda8 --> 0x7fffffffde78 --> 0x7fffffffe192 ("/home/fmagin/Talks/IntWS_Q2_2018/hello")
0032| 0x7fffffffddb0 --> 0x100040000
0040| 0x7fffffffddb8 --> 0x55555555463a (<main>:	push   rbp)
0048| 0x7fffffffddc0 --> 0x0
0056| 0x7fffffffddc8 --> 0xb96c34dd227a8d62
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 2, 0x000055555555463e in main ()
```

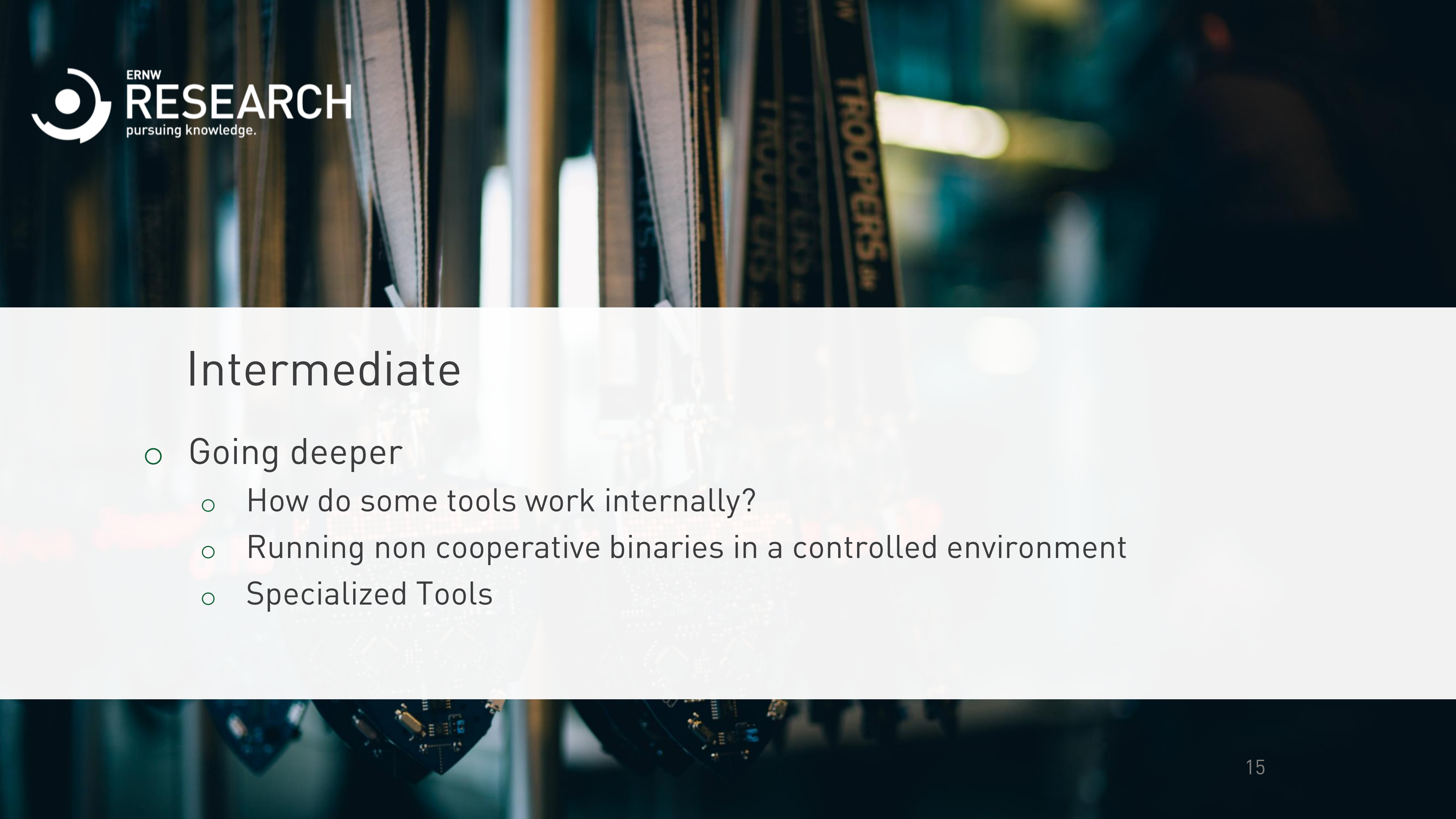
Basic Runtime Influence

- LD_PRELOAD Functionality
  - Load your libraries before the specified ones
- Those Functions get called instead of the intended ones
  - Replace “getRandomNumber” with “rand”
  - gcc -shared -fPIC unrandom.c -o unrandom.so
  - LD_PRELOAD=$PWD/unrandom.so ./binary
- No more randomness!



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Source: https://xkcd.com/221/



Intermediate

- Going deeper
  - How do some tools work internally?
  - Running non cooperative binaries in a controlled environment
  - Specialized Tools

Concept: Disassembly and Lifting

- Map from bytes to an instruction

[0x83, 0xc0,0x01] -> “add eax,1”

Concept: Disassembly and Lifting

- Map from bytes to an instruction
- Or lift to some other language that makes the semantics explicit

[0x83, 0xc0,0x01] -> “add eax,1”

add dstreg, immediate

dstreg += immediate

Tool: Capstone

- Disassembly Framework
- Python (and other bindings)
- Many (FOSS) tools use it in the background somewhere

- Same project provides Keystone for assembly

Source: http://www.capstone-engine.org/logo/

https://www.capstone-engine.org/

ERNW RESEARCH pursuing knowledge.

Disassembly Algorithms

- How do we or the tools know what part of the binary is code?
- ELF Information
  - Entrypoint
  - Possibly symbols

```
00000000: eb01 b848   ...H
00000004: c7c0 3905   ..9.
00000008: 0000 48c7   ..H.
0000000c: c37f 1d00   ....
00000010: 00ff d0     ...
```

Disassembly Algorithms

- Linear Sweep
  - Easy to implement
- Might yield confusing results
  - On architectures like x86 with variable instruction lengths and no forced alignment

```
eb01:         jmp 3
b848c7c039:   mov eax, 0x39c0c748
05000048c7:   add eax, 0xc7480000
c3:           ret
7f1d:         jg 0x1f
0000:         add byte [rax], al
ffd0:         call rax
```

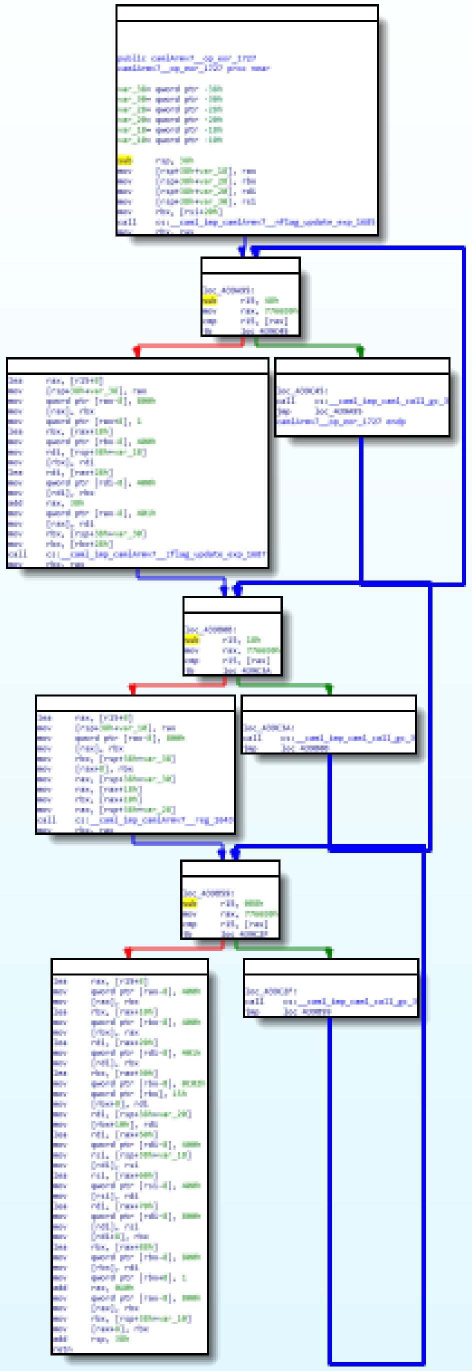
Disassembly Algorithms

- Recursive Descent Disassembly
  - Requires some semantic understanding
  - More accurate

```
eb01:           jmp 3
b8:             db 0xb8
48c7c039050000: mov rax, 1337
48c7c37f100000: mov rbx, 4223
ffd0:           call rax
```

Control Flow Graph Generation

- Graph of the possible control flows through the program
- Tradeoffs between accuracy and tractability
- Highly useful, it’s easy to get lost in disassembly
- Every good graphical disassembler should have this somewhere



Executable Parsing: Continued

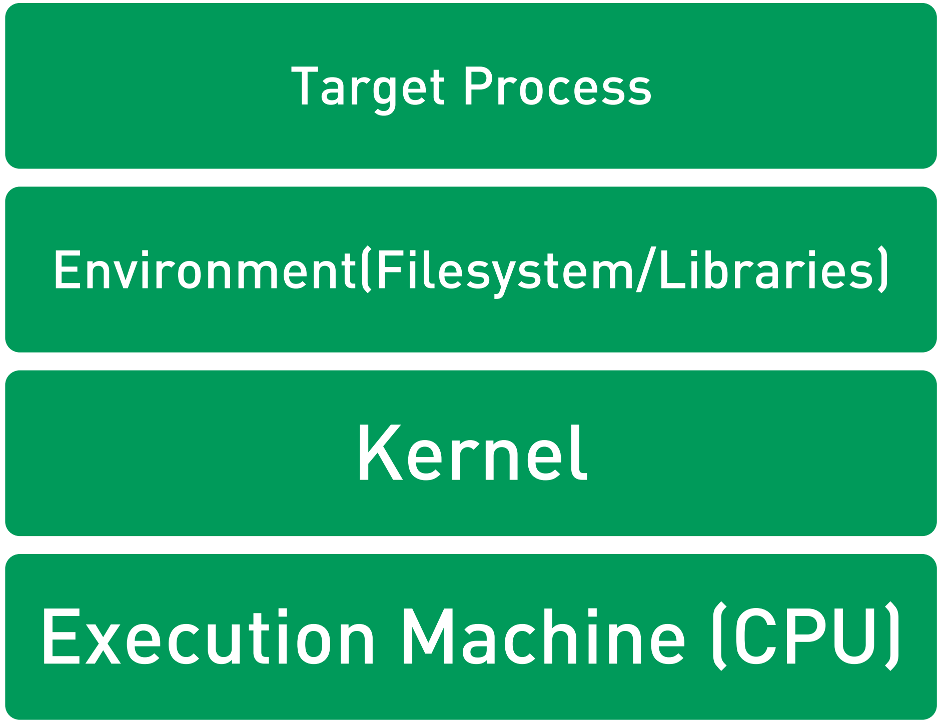
- You might want to build something that needs this
  - Sure, you could just use objdump and grep

- Small pure python library for ELF parsing: ‘pyelftools’

- If you want something more fancy: ‘lief'

Dynamic Analysis

- Observing and manipulating at runtime



Target Process

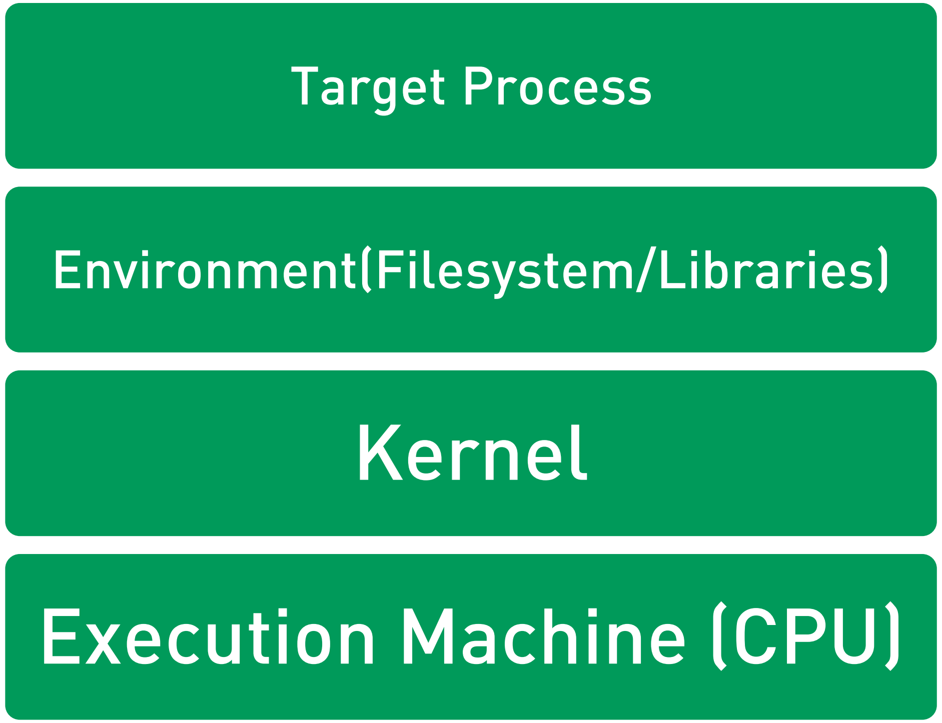
Environment(Filesystem/Libraries)

Kernel

Execution Machine (CPU)

Emulation

- Everything below some level of abstraction is emulated
- Level of Abstraction => Kind of emulation



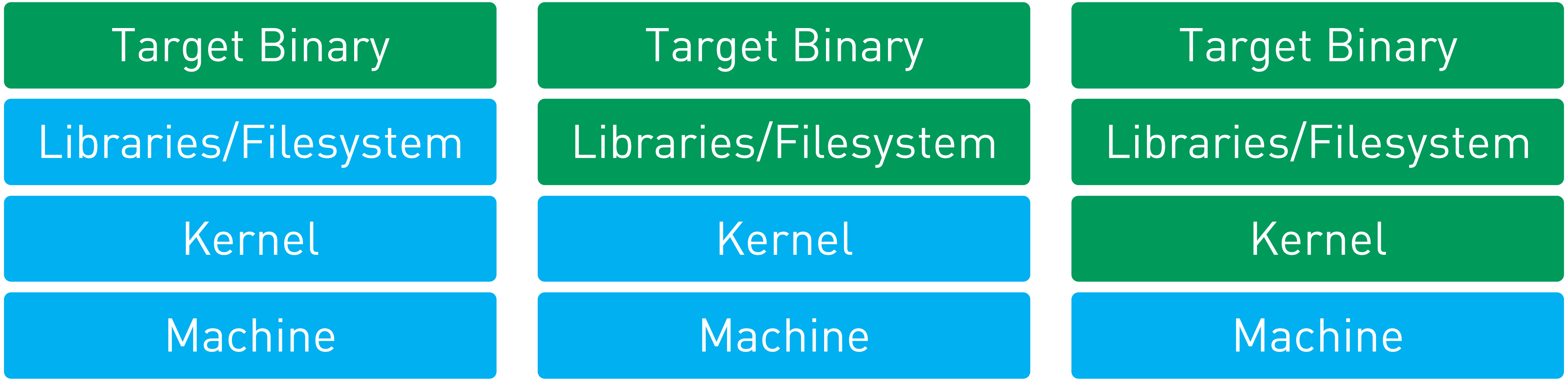
Target Process

Environment(Filesystem/Libraries)

Kernel

Execution Machine (CPU)

Emulation



Target Binary	Target Binary	Target Binary
Libraries/Filesystem	Libraries/Filesystem	Libraries/Filesystem
Kernel	Kernel	Kernel
Machine	Machine	Machine

Plain QEMU User mode	Chrooting into System Image with QEMU as Interpreter	QEMU System Mode

Emulation

- We are always at least in control of the execution machine
  - But we are slower than the real one
- Redefine instructions
  - Up is down, down is up, “inc reg” now decrements the register
- Add custom code to the emulation logic
  - Callback on every {jump, call, syscall} for analysis

- Fully emulating the environment might hard
  - Example: Windows API

Tool: QEMU

- Supports a lot of architectures
- Used for device emulation in KVM/Xen
- Decently fast
  - JITs and caches basic blocks

Source: https://wiki.qemu.org/Logo

https://www.qemu.org/

Tool: Unicorn

- Lightweight emulator
  - Just the CPU emulation core of QEMU
  - No device emulation
  - No syscalls
- Library
  - Use as the backend in some other tool
  - Emulate small code snippets

Source: https://www.unicorn-engine.org/images/unicorn.png

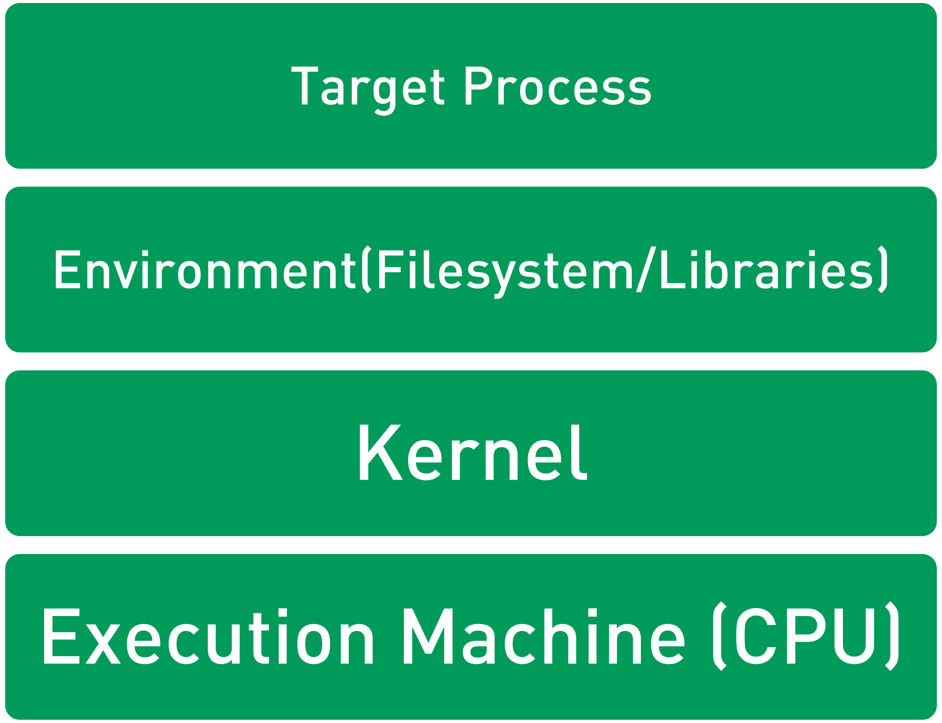
https://www.unicorn-engine.org/

Tools: Misc

- pyrebox
  - IPython shell for introspection and instrumentation of (mainly Windows) guests
  - Main Focus: Malware Analysis
  - https://github.com/Cisco-Talos/pyrebox
- panda2
  - Full system tracing and analysis based on QEMU
  - https://github.com/panda-re/panda

Dynamic Binary Instrumentation

- Rewrite the target code at runtime
  - Remove code
  - Add analysis code
  - Hook functions



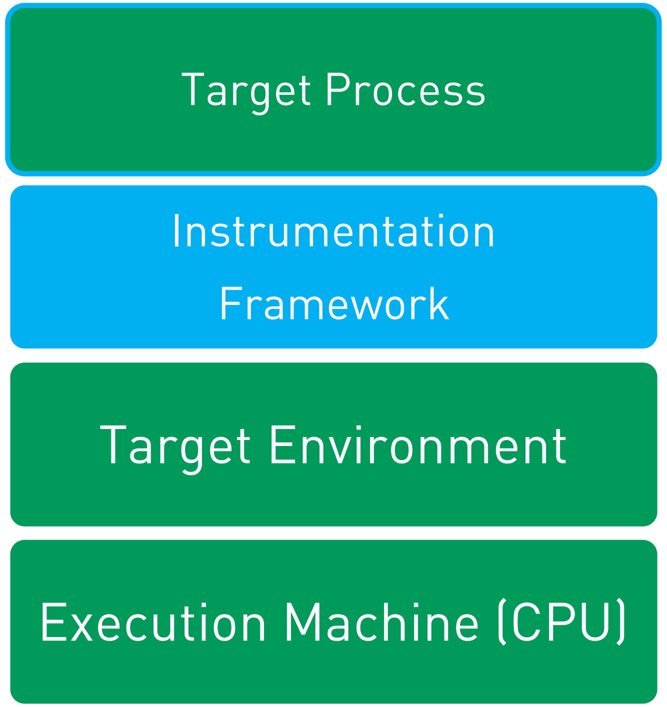
Target Process

Environment(Filesystem/Libraries)

Kernel

Execution Machine (CPU)

Some Dynamic Binary Instrumentation Approaches



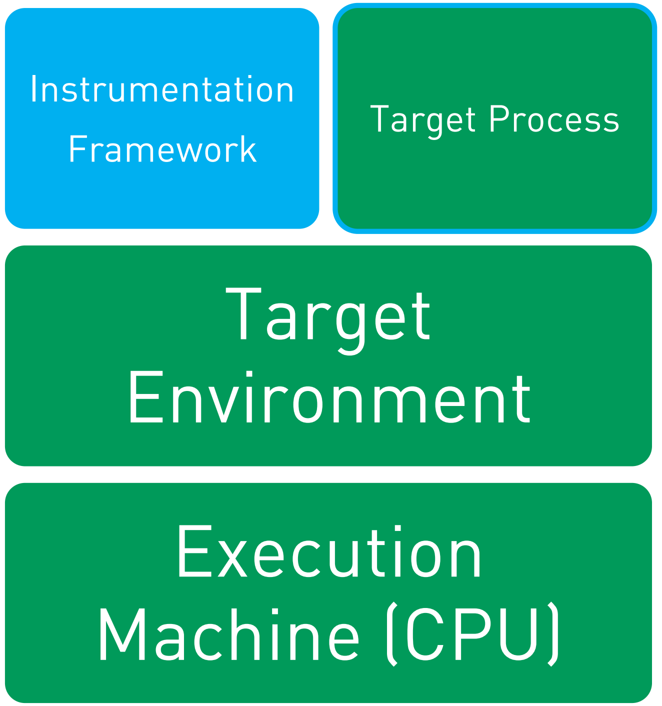
Target Process

Instrumentation Framework

Target Environment

Execution Machine (CPU)

Framework runs a provided Binary.
Example: DynamoRIO



Instrumentation Framework

Target Process

Target Environment

Execution Machine (CPU)

Framework is loaded
into an existing process.
Example: Frida

Dynamic Binary Instrumentation: Basic Idea

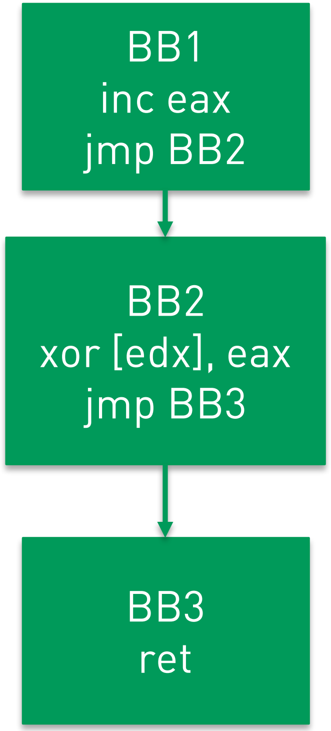


Fig 1: Typical Execution Flow

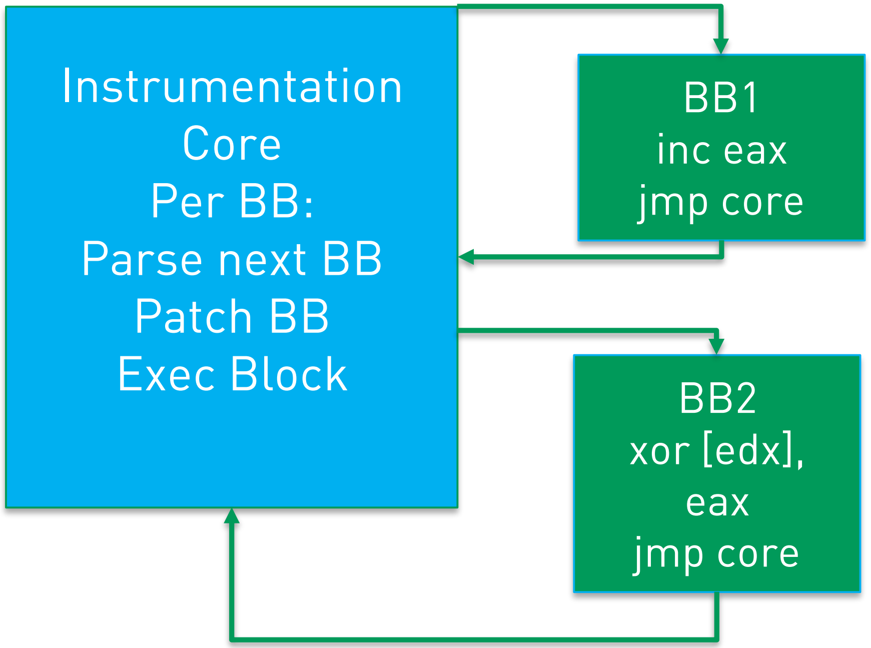


Fig 2: Execution Flow under Dynamic Instrumentation

Dynamic Binary Instrumentation: Use Cases

- Hook functions
  - Library call and System Calling Tracing
  - Tracing of any function call
  - Basic Block Tracing for Coverage (Fuzzing)
  - Change return values
- Example
  - Static Crypto Key Generation is obfuscated?
  - Just hook the call where it is used

Tool: Frida

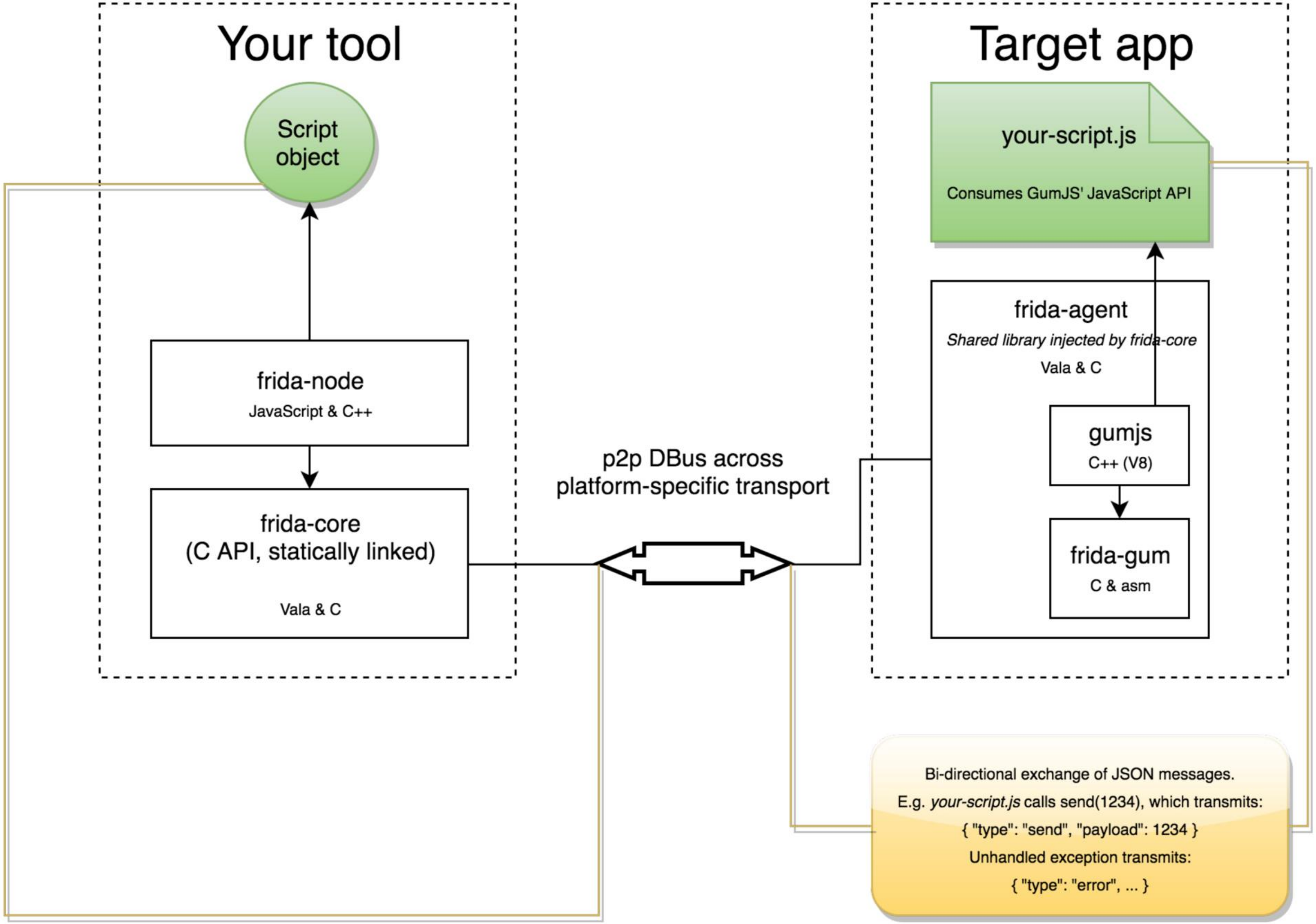
FЯIDA

Source: https://www.frida.re/img/logotype.svg

- Dynamic instrumentation toolkit
- Scriptable
- Multi-platform and multi-arch
  - Windows/Mac/Linux/Android/iOS/QNX – i386/AMD64/ARM/ARM64
- Bindings for Python, .NET, C and Node.js
  - But the actual scripts have to be written in Javascript...
- Very easy to Hook functions

https://www.frida.re/

Frida Architecture



Your tool

Script object

frida-node
JavaScript & C++

frida-core
(C API, statically linked)
Vala & C

p2p DBus across platform-specific transport

Target app

your-script.js
Consumes GumJS' JavaScript API

frida-agent
Shared library injected by frida-core
Vala & C

gumjs
C++ (V8)

frida-gum
C & asm

Bi-directional exchange of JSON messages.
E.g. *your-script.js* calls send(1234), which transmits:
{ "type": "send", "payload": 1234 }
Unhandled exception transmits:
{ "type": "error", ... }

https://www.coresecurity.com/system/files/publications/2016/10/Getting%20fun%20with%20Frida-Ekoparty-21-10-2016.pdf

DBI: Misc

- DynamoRIO
  - More mature
  - FOSS (BSD)

- Intel PIN
  - More mature
  - Proprietary, but free as in beer

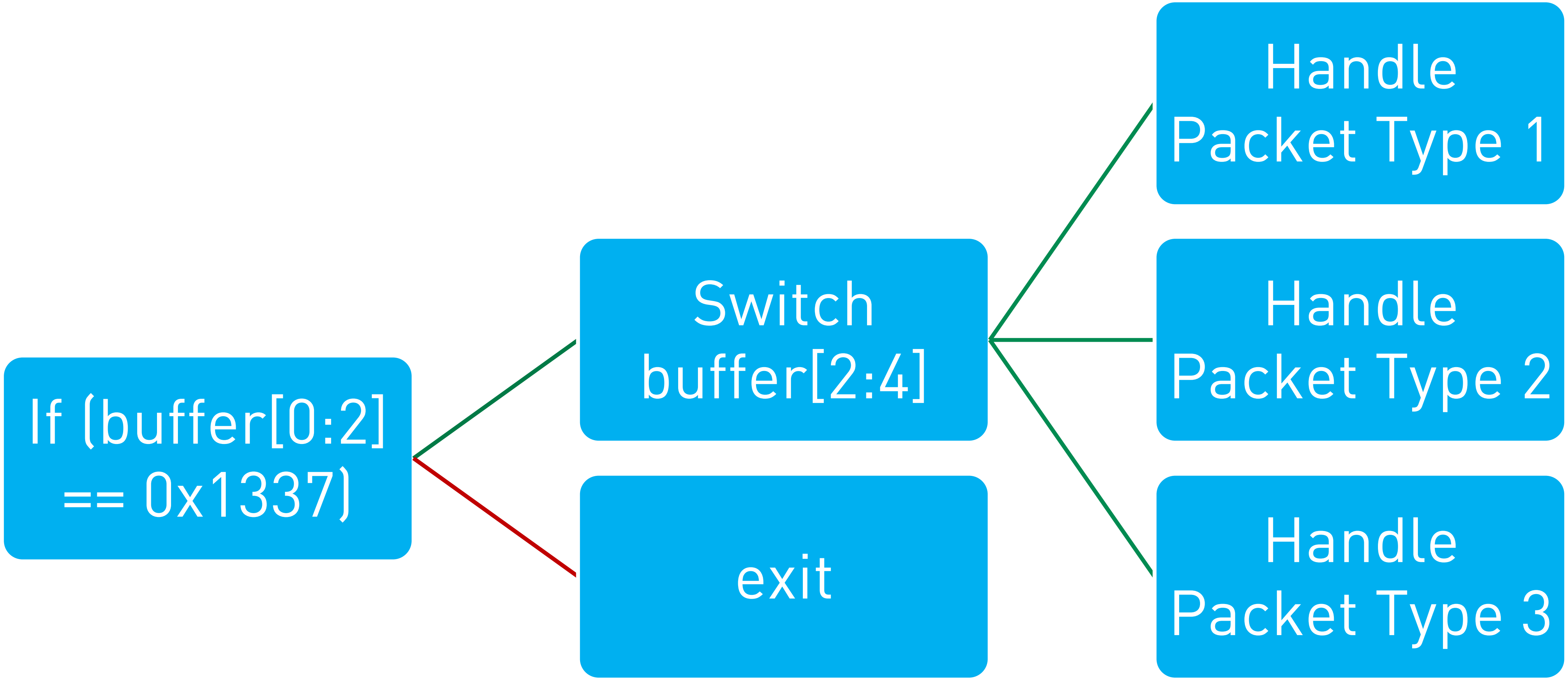
Advanced

- Formalizing
- Automation

Binary or Program Analysis

- The subfield of computer science dealing with automated analysis
- Massive improvements over the last years
  - Mainly due to the DARPA CGC

Symbolic Execution



Tool: angr

- Binary Analysis Framework
  - Lifting to VEX IR
  - Emulation
  - Symbolic Execution
  - CFG Generation
- Used in the DARPA CGC by Shellphish , won 3rd place
- Best used from an interactive IPython Shell
- Build tools upon or integrate into others

Source: http://angr.io/img/angry_face.png

https://github.com/angr/angr

Conclusion

- The right mature tooling makes your life a lot easier
- Initial learning overhead tends to be worth it

- Combine tooling to solve new problems

- Integrate new tooling into your existing tooling

Probably out of time?

Option 1: Q&A

Option 2: Misc

✉ fmagin@ernw.de

@0x464D

www.ernw.de

www.insinuator.net

Misc

CPU Features

- CPUs sometimes provide advanced features

- Hardware Watchpoints
  - Hard to detect
  - Break on data read/write and not just code

- Intel PT
  - Trace execution with your CPU

Tool: rr

- Extension to GDB that allows recording a trace and debugging it
  - Run or step the program in reverse

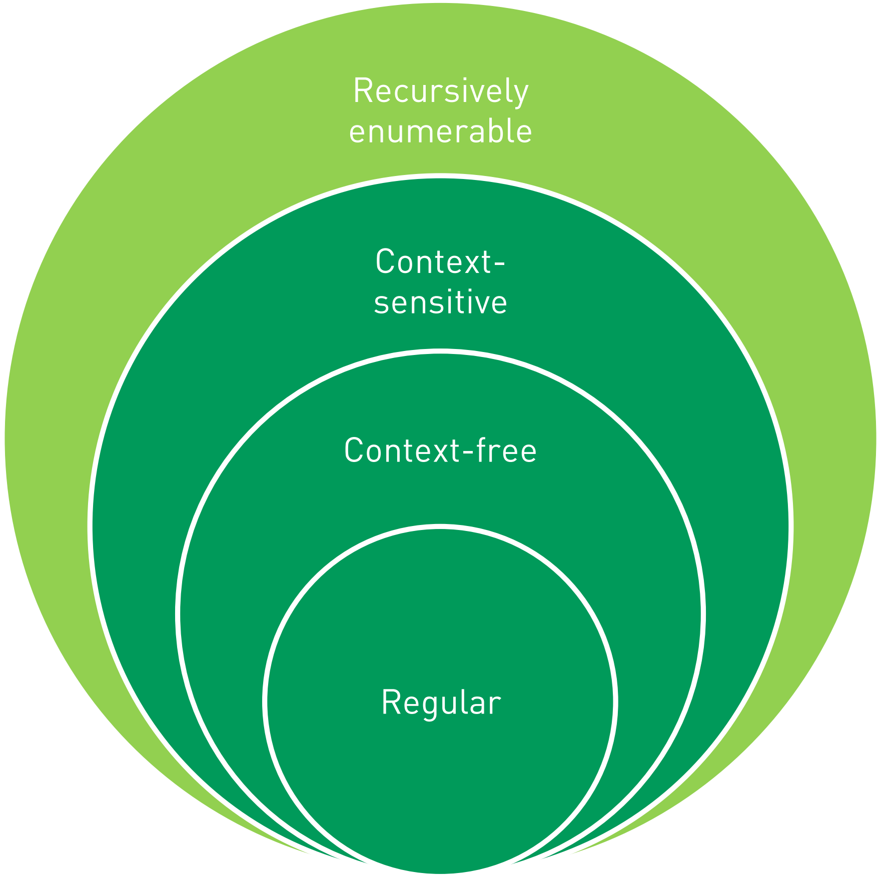
https://rr-project.org/

Concept: Parser Generation

- Unknown File Format
  - $proprietary protocol or file format
  - Some patterns might be obvious
  - Others can be derived from looking at an existing Parser
- Problem: Support for custom tools
  - Parser for Visualization
  - Parser/Serializer for Custom Client
  - Language Aware Fuzzer

Theory: Formal Languages

- Every protocol or file format is basically a formal language
- Every formal language is induced by a grammar
  - More than one even
- You can generate a parser for the language from the grammar
  - Theoretically
- But for every sane protocol this should work

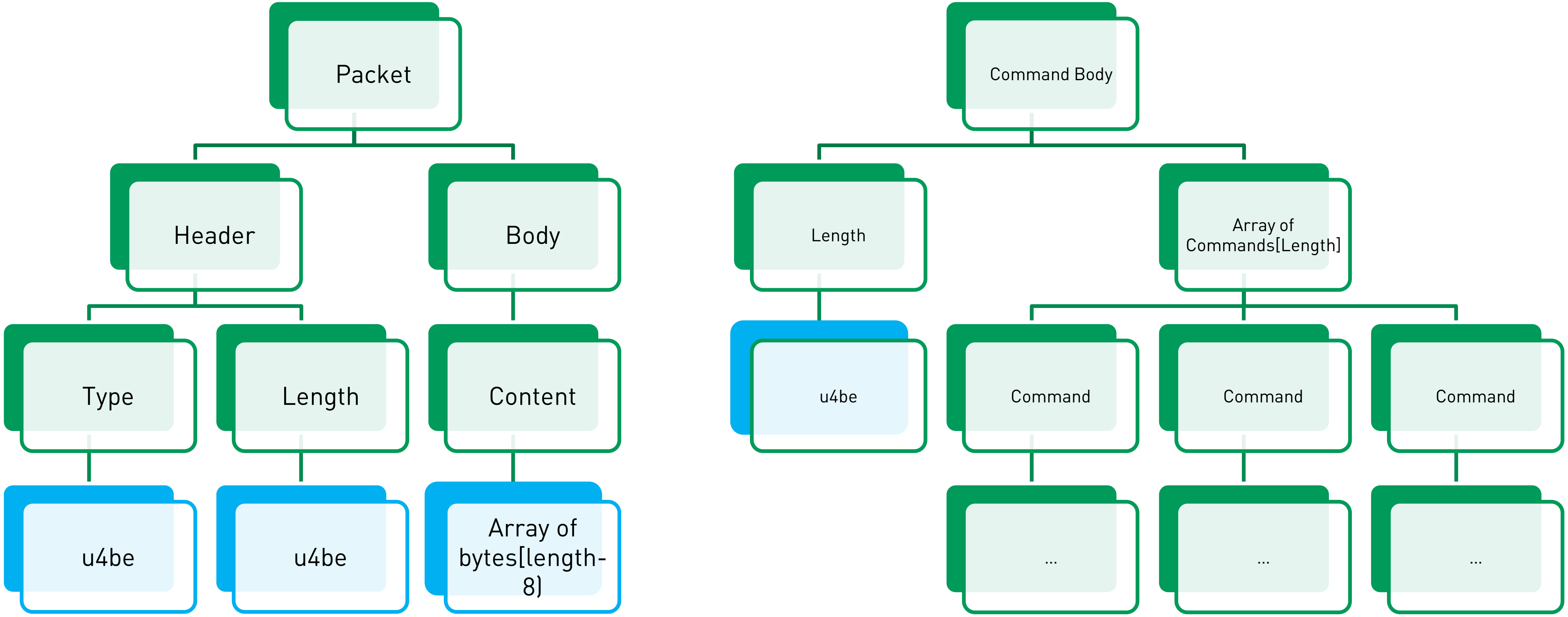


Kaitai Struct

- Generates parser in many languages from a spec
- Compiles to
  - C++/STL
  - Python, Ruby, Perl
  - Javascript
  - C#, Java
  - Lua
  - Others

Source: http://kaitai.io/img/kaitai_16x_dark.png

Parse Trees


Packet
Header
Body
Type
Length
Content
u4be
u4be
Array of bytes[length-8]
Command Body
Length
Array of Commands[Length]
u4be
Command
Command
Command
…
…
…

Kaitai Struct Use Cases

- Binary protocols over HTTP that you are intercepting with ‘mitmproxy’

- Wireshark Dissectors

- Burp Plugins