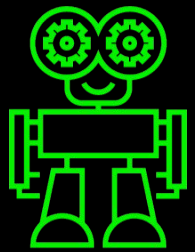
Codename: flip.re

Hypervisor-level debugging and beyond



Lars Haukli
@zutle
lars@flip.re

The malware landscape

350 000 new malware threats per day *

The challenge: Understanding the threats that matter to you

- Discover the malware threats and actors that can hurt you
- Handle the malware threats that end up causing security incidents

* source: AV-TEST

Malware analysis challenges

Even analyzing 1‰ (permyriad) of 350 000 would mean 35 per day!

A single threat can easily take 2 weeks to analyze

Sophisticated malware actively evade analysis

Effective malware analysis

“For me, reverse engineering efficiently has always been about combining dynamic and static techniques”

Dynamic analysis is a lot faster, and almost always useful for deeper analysis

- Get an idea of what the code is doing
- Locate the code you want to analyze further

Evasive Malware (anti-analysis techniques)

Designed to frustrate reverse engineers by actively making analysis challenging

- Code obfuscation / encryption
- VM/Sandbox detection
- Self-debugging
- Multi-stage code injection

Current malware sandboxes for analysis use

Often used as an initial (dynamic) analysis

Reports can be useful, but is typically superficial

All fully automated analysis solutions will be vulnerable to evasive malware. Always. It is not possible to fix.

Anti-evasion problem = Halting problem

The flip.re project

An interactive malware analysis system, combining dynamic and static analysis to

- Analyze advanced threats faster
- Triage threats faster

You need to understand how the malware works
before it’s too late.

And you need a human in the analysis loop.

The plugin

IO plugin for r2

Turns r2 into a hypervisor-level debugger

- Debug code running inside a Windows guest from the outside
- Enables debugging code in a very sneaky and stealthy way

Conceptually similar to the zdbg plugin (r2con 2017),
but written from scratch in Rust

Rust

Pros:

- Amazing language with an excellent modern toolchain
- Great replacement for C as a safer and more reliable systems programming language

Cons:

- Compiler can feel overly strict
- Unsafe Rust has a very similar feeling to C
  (type casting, memory management)

build.rs

Using bindgen, you can generate Rust FFI bindings to use the radare2 C plugin API

```
let bindings = bindgen::Builder::default()
    .rustfmt_bindings(true)
    .derive_default(true)
    .header("r2_wrapper.h")
    .clang_arg(format!("-I{}", r2_include_dir.trim()))
    .clang_arg(format!("-I{}/sdb", r2_include_dir.trim()))
    .whitelist_var("R_LIB_TYPE_.*")
    .whitelist_var("R_IO_SEEK_SET|R_IO_SEEK_CUR|R_IO_SEEK_END")
    .whitelist_type("RLibStruct|RIOPlugin|RDebugPlugin|RIODesc")
    .whitelist_type("RDebugPid|RDebugMap|RListIter")
    .no_copy("r_io_desc_t|r_debug_pid_t")
    .whitelist_function("r_list_append|r_list_sort")
    .generate()
    .unwrap_or_else(|()| {
        println!("Whoops!");
        process::exit(1);
    });
```

```
#include "r_lib.h"
#include "r_io.h"
#include "r_debug.h"
```

Plugin

```
static PLUGIN_DATA: RIOPlugin = RIOPlugin {
    name: "flip\0".as_ptr() as *mut c_char,
    desc: "flip.re plugin\0".as_ptr() as *mut c_char,
    version: "0.1.0\0".as_ptr() as *mut c_char,
    author: "zutle\0".as_ptr() as *mut c_char,
    license: "LGPL3\0".as_ptr() as *mut c_char,
    widget: std::ptr::null_mut(),
    uris: "flip://\0".as_ptr() as *mut c_char,
    listener: None,
    accept: None,
    check: Some(flux::check),
    close: Some(flux::close),
    create: None,
    extend: None,
    getbase: None,
    getpid: None,
    gettid: None,
    init: None,
    is_blockdevice: None,
    is_chardevice: None,
    isdbg: true,
    lseek: Some(flux::lseek),
    open: Some(open),
    open_many: None,
    read: Some(flux::read),
    resize: None,
    system: Some(flux::system),
    undo: RIOUndo {
```

build.rs

You can also compile C code together with your Rust code (use the cc crate)

```
let cfg = cc::Build::new()
    .include(r2_include_dir.trim())
    .include(format!("{}/sdb", r2_include_dir.trim()))
    .include("gdb/include")
    .file("gdb/src/libgdbr.c")
    .file("gdb/src/core.c")
    .file("gdb/src/messages.c")
    .file("gdb/src/packet.c")
    .file("gdb/src/utils.c")
    .compile("gdb");
```

Developing radare2 plugins in Rust

We are releasing an example Rust plugin for r2con 2020 for anyone interested in learning more about developing radare2 plugins in Rust

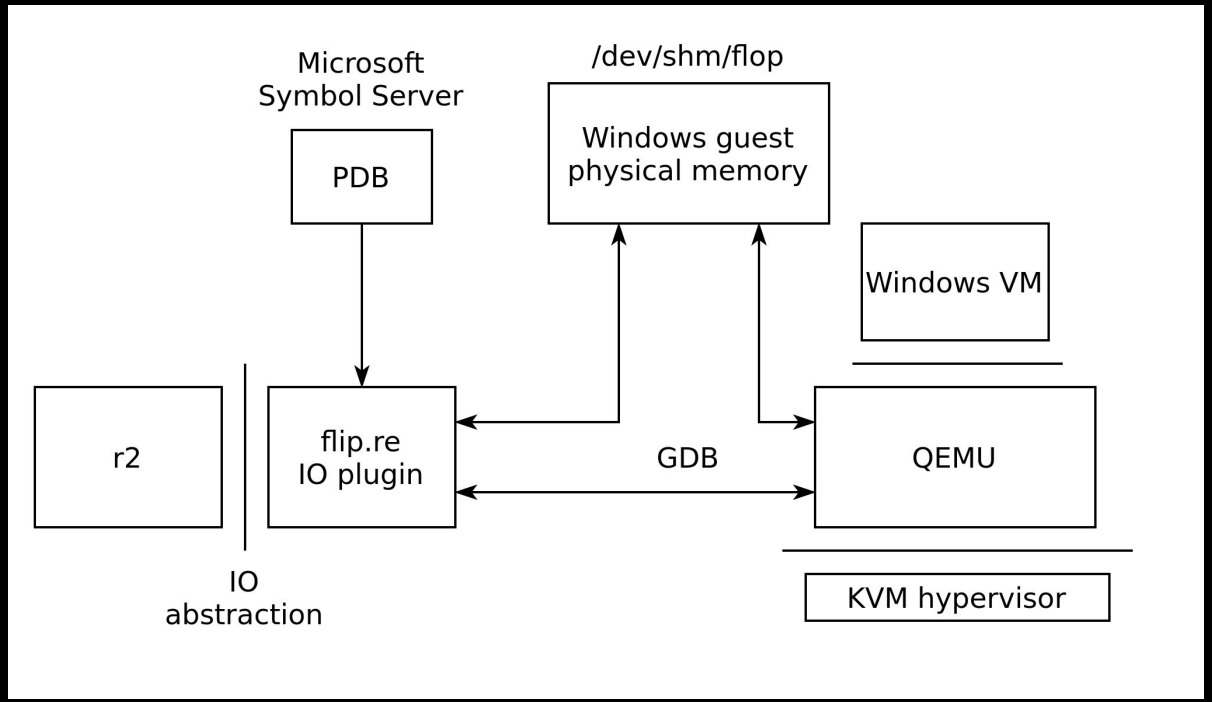
Available soon at:

https://github.com/flipreversing/r2con2020

Follow us on Twitter for updates:

@flipreversing

Architecture



GDB provides registers and debug control

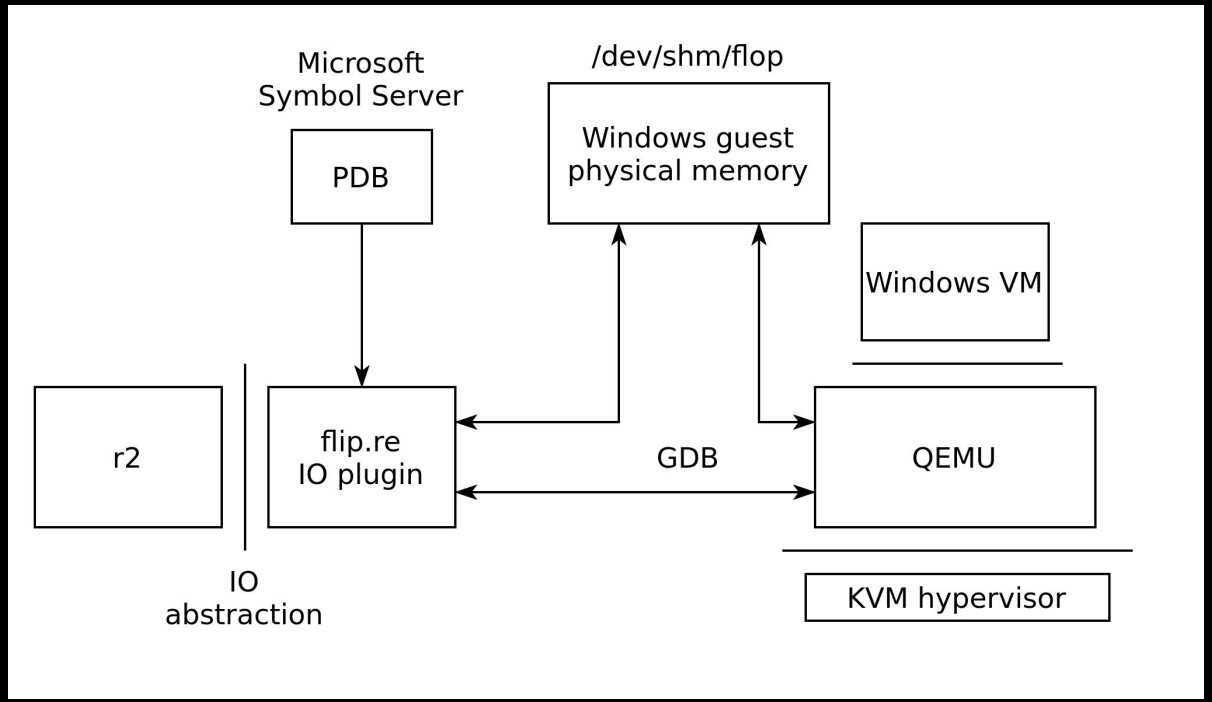
GDB

‘-s’ param in QEMU enables gdbstub (GDB server) on localhost:1234

Effectively enables us to debug the virtual CPU

- Query CPU registers
- Single Step
- Trap on Breakpoints

Architecture



Shared memory provides access to physical memory

Physical memory access

Using the memory-backend-file QEMU param

-object memory-backend-file,[...],mem-path=/dev/shm/flop,share=on

Essentially gives us access to guest physical RAM

Virtual memory access

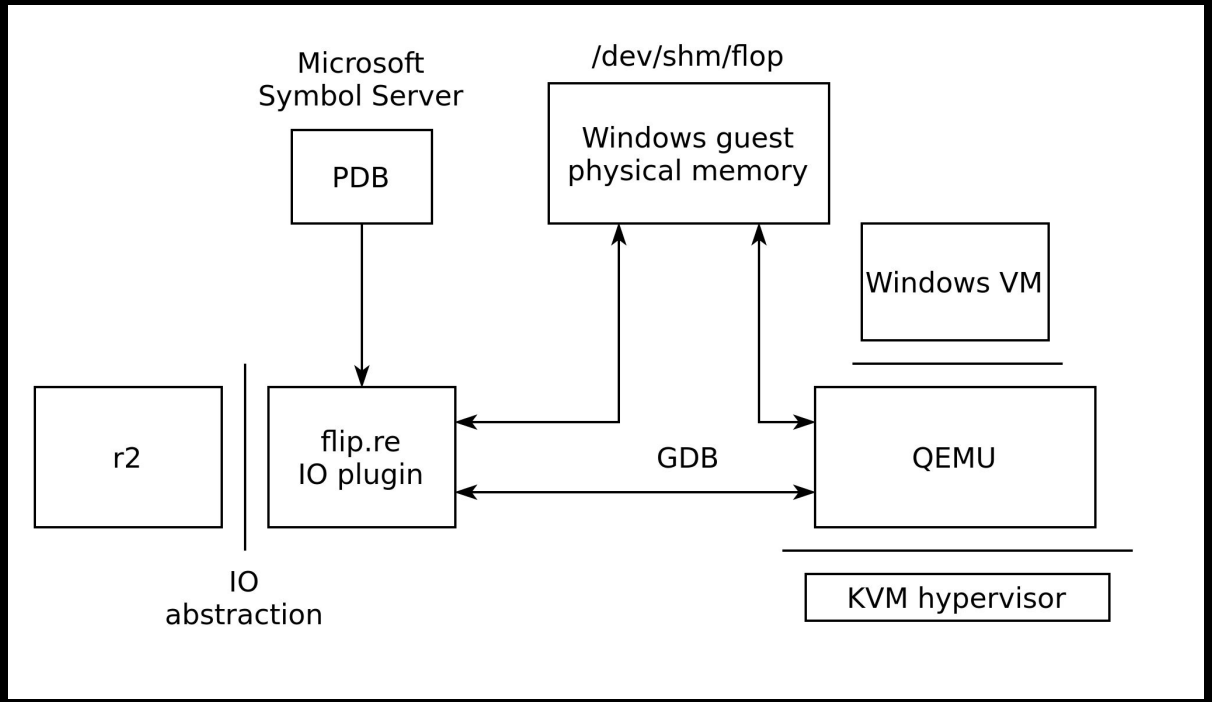
Each process has its own CR3 values (well actually two after Spectre/Meltdown)
- contains directory base table (first lookup table)

x86 architecture defines address translation logic
- Windows adds some features on top (r2con 2017)

Our plugin enables us to:

- Select individual processes to inspect
- Effectively work with virtual addresses

Architecture



PDB is our key to navigate the Windows kernel

PDB symbols

Our API to the Windows kernel

- Details on types and structures
  - Undocumented structures change over time
- Location in memory of important structures
  - Enables us to find things like the process list

PDB symbols

Available on the Microsoft Symbol server

We require the GUID: r2 -c “i~guid[1]” -q ntoskrnl.exe
and the pdb name: r2 -c “i~dbg_file[1]” -q ntoskrnl.exe

With this info, the plugin can download the correct PDB from Microsoft

Finally we parse the PDB file using the pdb crate (Rust)

- https://crates.io/crates/pdb

Live demo time!

- Attaching to a Windows VM
- Load kernel symbols as flags in r2
- Load a program inside the Windows VM
  - With some help from Frida ;)

Case Study: Anti-Anti analysis

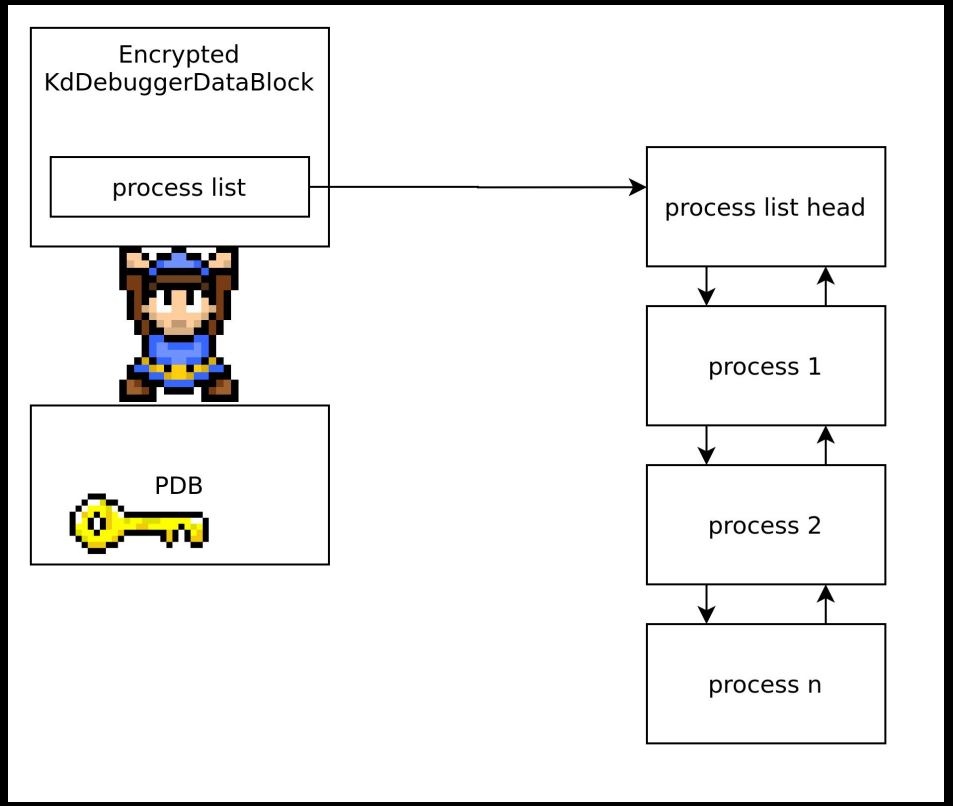
Tracing can be an effective technique to locate interesting code

- Frida is a great tool for this!

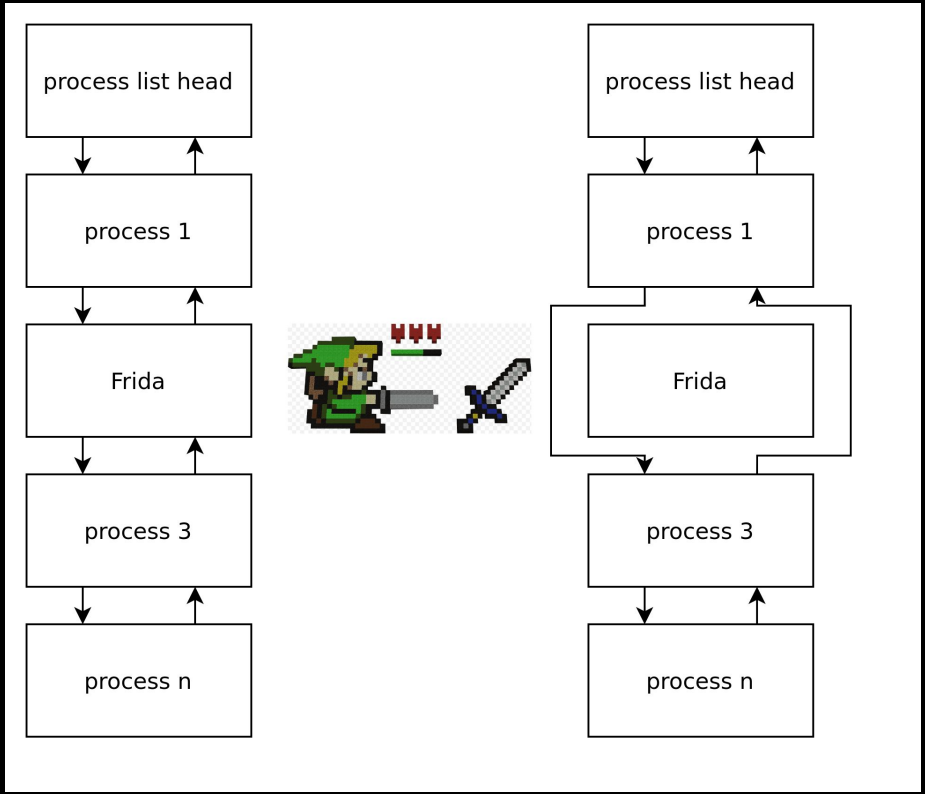
Malware can check for Frida’s processes on the system, and evade our analysis

- Our system can hide analysis components
- In this way we can fool the malware into running

Retrieving the process list



Hiding the Frida server process



Live demo time!

- Using rootkit techniques to hide the Frida process
- Fooling grandma into revealing her true self
- Twice. And then a bonus trick

Disclaimer:
This is an early alpha version of the plugin,
and some features were coded during the con :D

Participate!

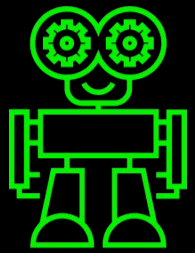
We will soon be looking for alpha testers and early adopters!

Follow us on Twitter: @flipreversing

Join our Telegram: https://t.me/flipreversing

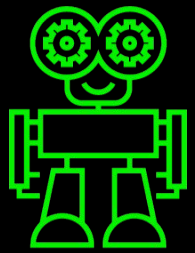
Send us an email: hello@flip.re

Question time!



Lars Haukli
@zutle
lars@flip.re

Thank you, and enjoy chiptunes!



Lars Haukli
@zutle
lars@flip.re