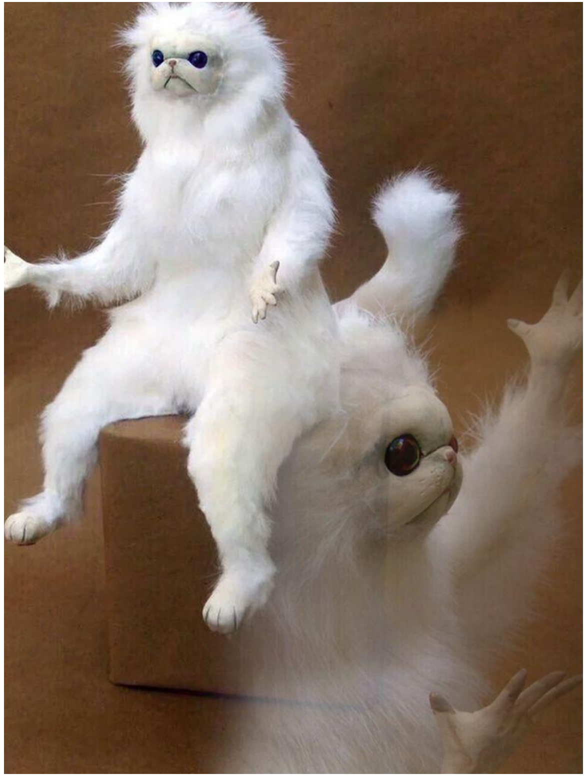
Tartiflette:
Snapshot fuzzing with KVM and libAFL

Tanguy Dubroca, César Belley

Help me



Fuzzing ?

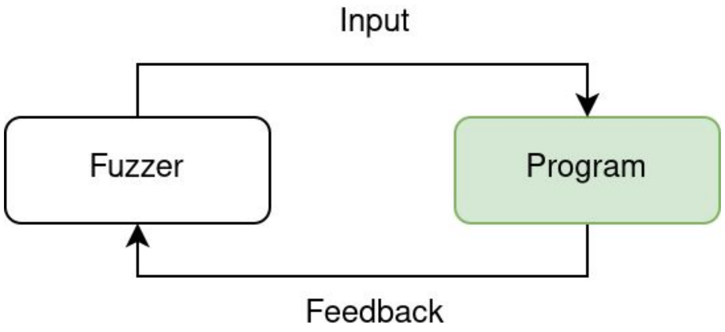
libAFL ??

KVM ???

Tartiflette ????

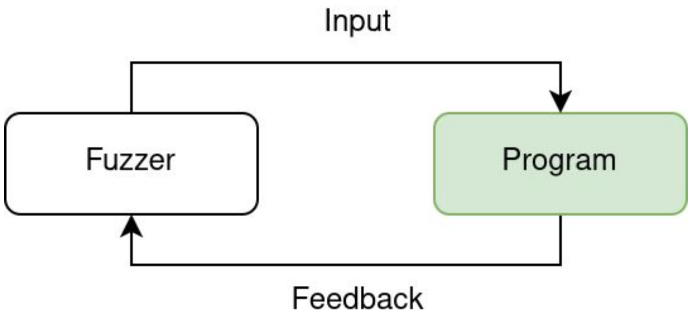
Fuzzing 101

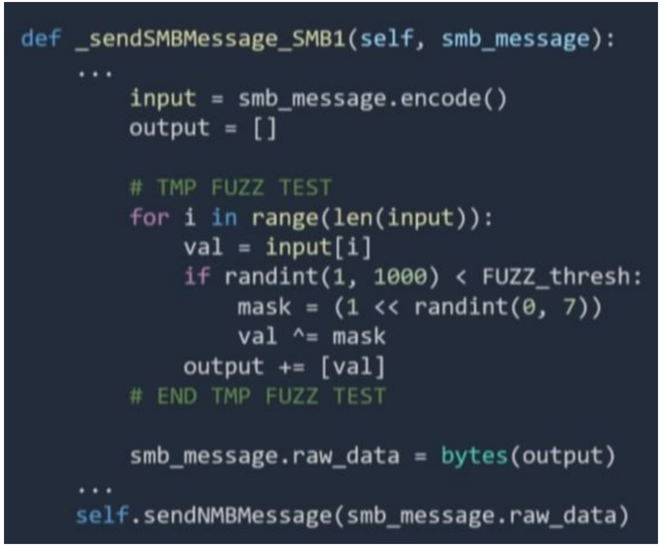
- Automatically tests robustness of programs
- We provide an input to the program:
  - Random
  - Corrupted (mutated valid input)
  - Generated (ex: from a grammar)
- We observe the effects on the execution:
  - Nothing (99% of the time)
  - Crash (very interesting)
- Not limited to memory corruption:
  - Differential fuzzing
  - Web fuzzing (xss)
  - Only requirement -> some form of feedback



Blackbox fuzzing

- The simplest type of fuzzing...
  - Very low feedback (crash oracle)
  - No need for source code !
  - Fast to build
  - ./my_prog < /dev/urandom
- .. but it works !
  - 3DS smb bug -> RCE [1]

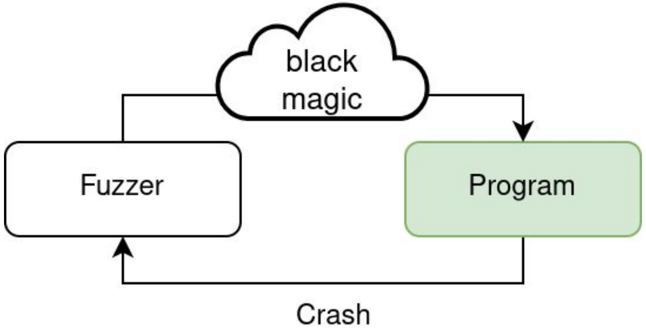




```
def _sendSMBMessage_SMB1(self, smb_message):
  ...
    input = smb_message.encode()
    output = []

    # TMP FUZZ TEST
    for i in range(len(input)):
      val = input[i]
      if randint(1, 1000) < FUZZ_thresh:
        mask = (1 << randint(0, 7))
        val ^= mask
      output += [val]
    # END TMP FUZZ TEST

    smb_message.raw_data = bytes(output)
  ...
  self.sendNMBMessage(smb_message.raw_data)
```

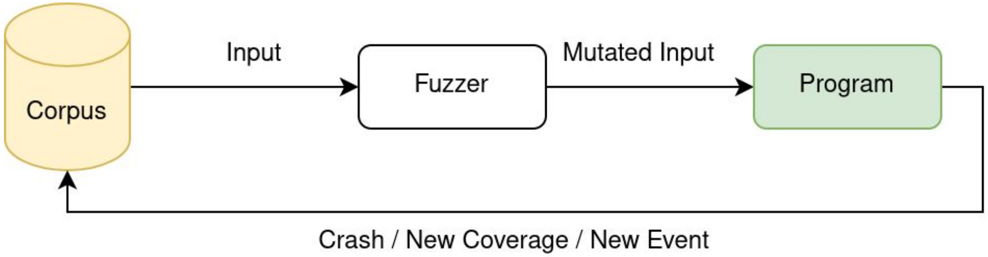
Whitebox fuzzing

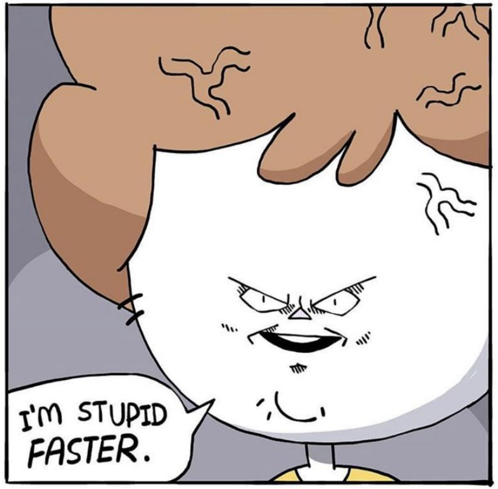
- Most complicated type of fuzzing:
  - Makes use of symbolic execution
  - Can solve complex constraints
  - With or without source code
  - Slow !
- But:
  - Can be combined with other fuzzing techniques

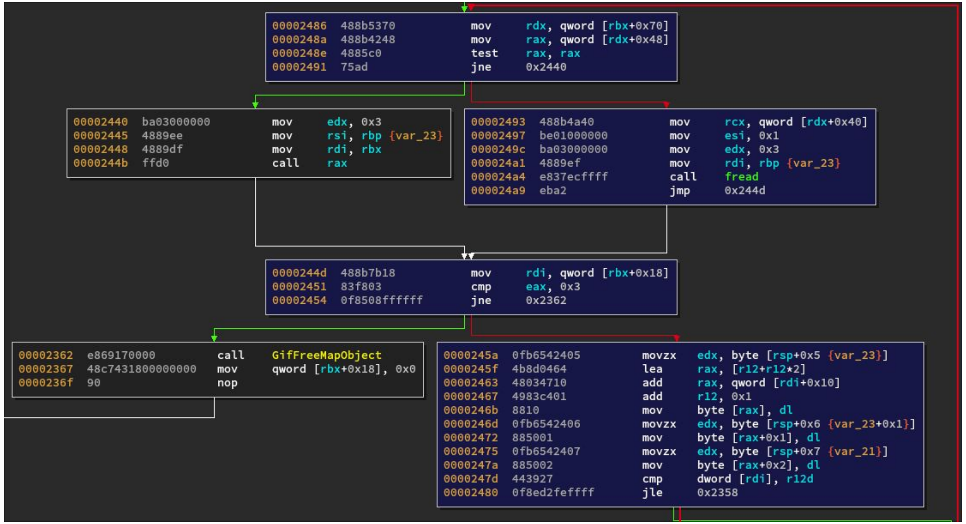


Greybox fuzzing

- The most common technique:
  - **AFL**, **libfuzzer**, **Honggfuzz**, etc…
  - Genetic algorithm using coverage*
  - With or without source code
  - Fast and efficient !
  - Good track record


Corpus
Input
Fuzzer
Mutated Input
Program
Crash / New Coverage / New Event





```
00002486  488b5370          mov     rdx, qword [rbx+0x70]
0000248a  488b4248          mov     rax, qword [rdx+0x48]
0000248e  4885c0            test    rax, rax
00002491  75ad              jne     0x2440
```

```
00002440  ba03000000        mov     edx, 0x3
00002445  4889ee            mov     rsi, rbp {var_23}
00002448  4889df            mov     rdi, rbx
0000244b  ffd0              call    rax
```

```
00002493  488b4a40          mov     rcx, qword [rdx+0x40]
00002497  be01000000        mov     esi, 0x1
0000249c  ba03000000        mov     edx, 0x3
000024a1  4889ef            mov     rdi, rbp {var_23}
000024a4  e837ecffff        call    fread
000024a9  eba2              jmp     0x244d
```

```
0000244d  488b7b18          mov     rdi, qword [rbx+0x18]
00002451  83f803            cmp     eax, 0x3
00002454  0f8508ffffff      jne     0x2362
```

```
00002362  e869170000        call    GifFreeMapObject
00002367  48c7431800000000  mov     qword [rbx+0x18], 0x0
0000236f  90                nop
```

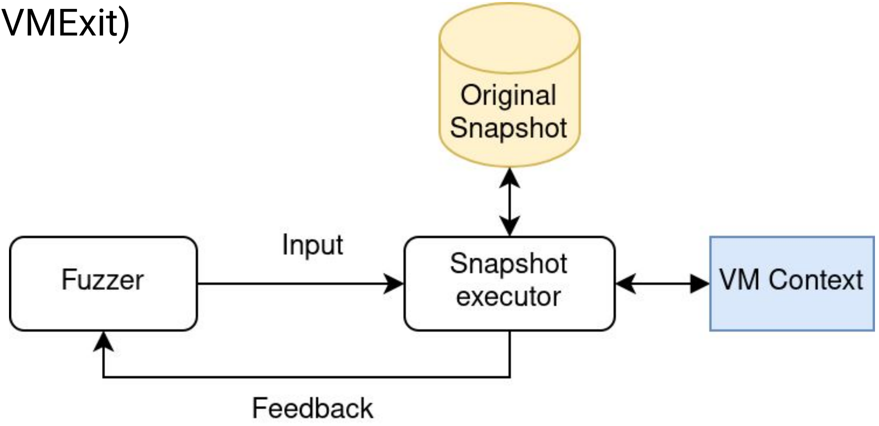
```
0000245a  0fb6542405        movzx   edx, byte [rsp+0x5 {var_23}]
0000245f  4b8d0464          lea     rax, [r12+r12*2]
00002463  48034710          add     rax, qword [rdi+0x10]
00002467  4983c401          add     r12, 0x1
0000246b  8810              mov     byte [rax], dl
0000246d  0fb6542406        movzx   edx, byte [rsp+0x6 {var_23+0x1}]
00002472  885001            mov     byte [rax+0x1], dl
00002475  0fb6542407        movzx   edx, byte [rsp+0x7 {var_21}]
0000247a  885002            mov     byte [rax+0x2], dl
0000247d  443927            cmp     dword [rdi], r12d
00002480  0f8ed2feffff      jle     0x2358
```

Snapshot fuzzing

- Running ~~programs~~ snapshots:
  - Process/VM memory
  - Process/VM CPU context
  - Metadata (coverage, symbols, opened files, etc…)
- Flexibility:
  - Fuzzing from any point of execution of a program
  - Userland, Kernel, Hypervisor
  - Windows kernel fuzzing on Linux
  - Fuzz another architecture is possible (emulators)
- But:
  - Requires a lot of boilerplate (syscall emulation layer)
  - Need for low level system knowledge
  - Hard to debug

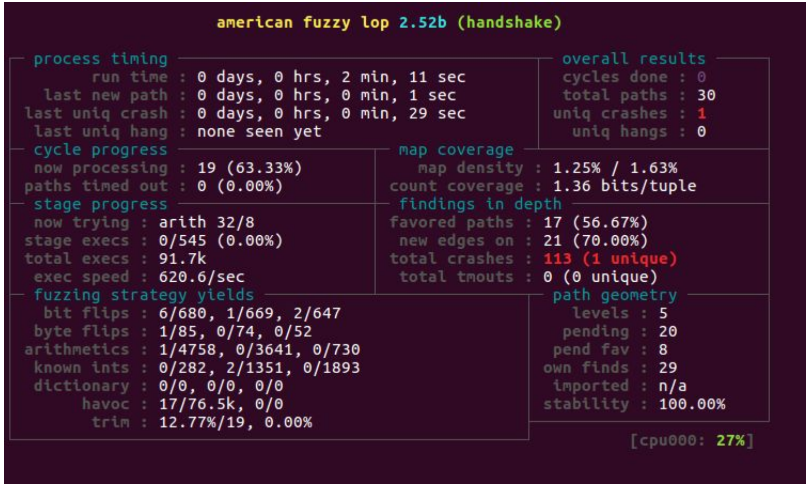
Snapshot fuzzing

- Processus:
  - Place the input data in memory
  - Run the snapshot until a chose exit point
  - Handle the coming events during runtime (VMExit)
    - Errors
    - Syscalls emulations
    - Instrumentation Hooks
  - Restore full VM state after execution
    - Gives deterministic execution
  - Give feedback to fuzzer



AFL

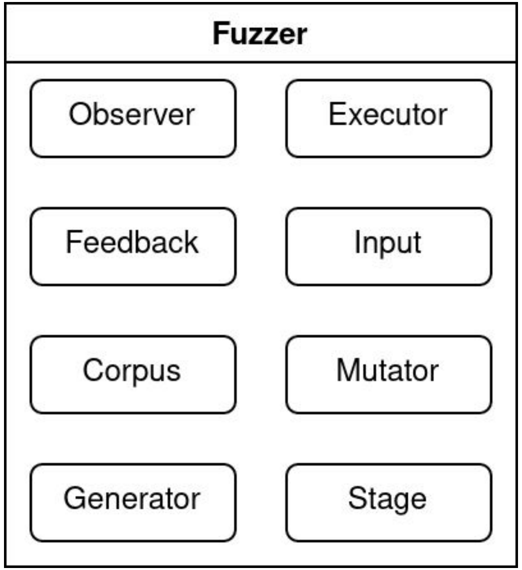
- First version released in 2014
- Impressive track record:
  - **Firefox**, **OpenSSL**, **sqlite**, **VLC**, etc…
- Easy to use
- Uses coverage for feedback:
  - With source: compile-time instrumentation
  - Without source: instrumentation with **QEMU**
- But:
  - Monolithic (new feature == fork)
    - **WinAFL**, **kAFL**, **TriforceAFL**, **afl-pt**, ...
  - Not optimized (too many syscalls)
  - Parallel fuzzing is a hack
  - Not actively maintained
    - **AFL++** is a great alternative



```
                 american fuzzy lop 2.52b (handshake)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 2 min, 11 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec        │  total paths : 30     │
│ last uniq crash : 0 days, 0 hrs, 0 min, 29 sec       │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 19 (63.33%)       │    map density : 1.25% / 1.63%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.36 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 32/8            │ favored paths : 17 (56.67%)            │
│ stage execs : 0/545 (0.00%)         │  new edges on : 21 (70.00%)            │
│ total execs : 91.7k                 │ total crashes : 113 (1 unique)         │
│  exec speed : 620.6/sec             │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 6/680, 1/669, 2/647                   │    levels : 5          │
│  byte flips : 1/85, 0/74, 0/52                      │   pending : 20         │
│ arithmetics : 1/4758, 0/3641, 0/730                 │  pend fav : 8          │
│  known ints : 0/282, 2/1351, 0/1893                 │ own finds : 29         │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 17/76.5k, 0/0                         │ stability : 100.00%    │
│        trim : 12.77%/19, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 27%]
```

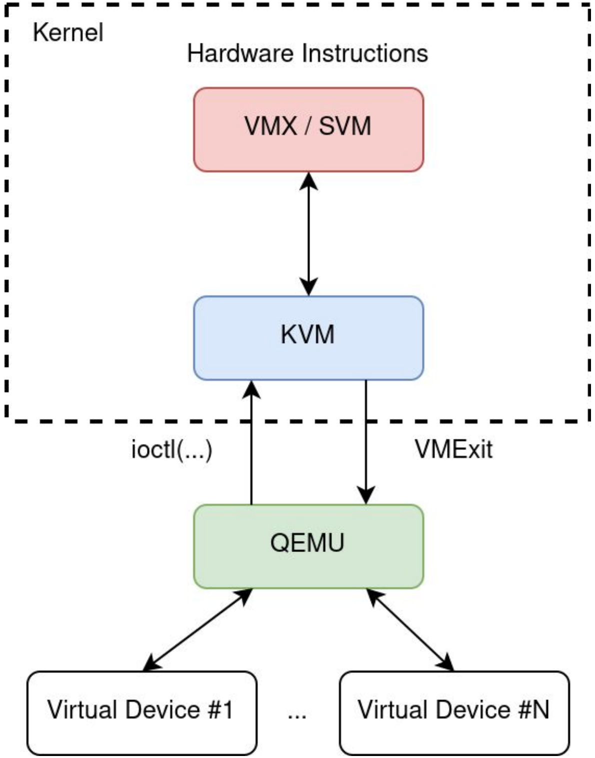
libAFL

- First version release in 2021
- By the **AFL++** authors
- Framework/library for fuzzing:
  - Extremely composable and flexible architecture
  - Designed for performance
  - Built for parallel and scalable fuzzing
- But:
  - Unstable API
  - Lack of documentation
  - Requires a lot of boilerplate code



KVM

- Kernel Virtual Machine
- Linux kernel hypervisor
- Exposes a userland interface:
  - VCPU manipulation
  - Memory manipulation
  - Interruption handling
  - Low level API
- Used as backend by other hypervisors:
  - QEMU/KVM
  - VirtualBox backend KVM



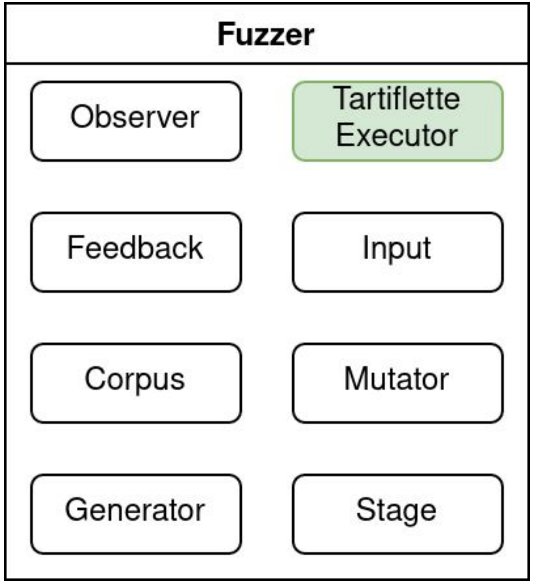
Tartiflette

libAFL + KVM

=

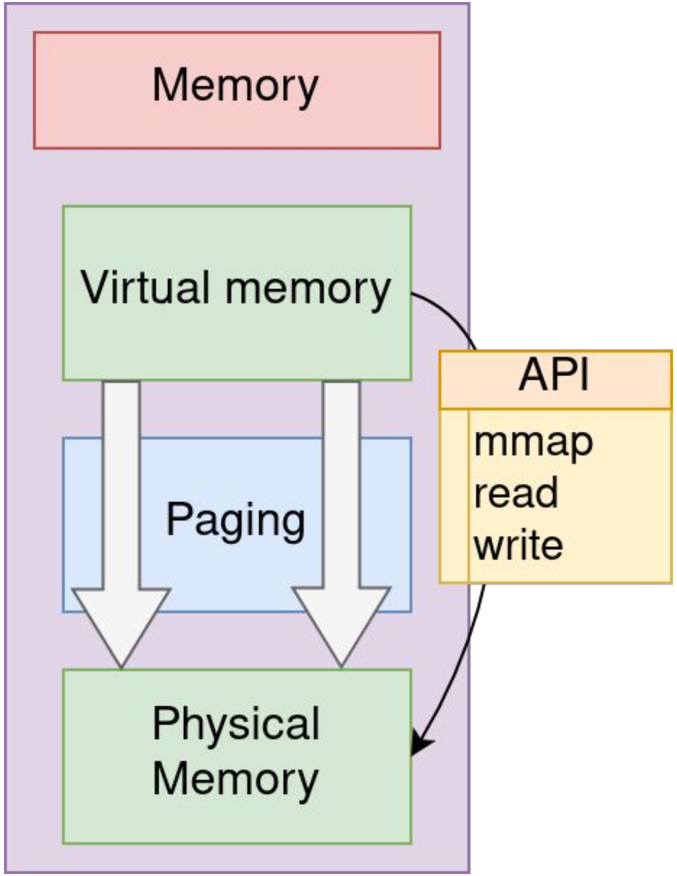
Tartiflette: Overview

- **Tartiflette-VM**
    - Abstraction layer over the **KVM** api
    - Provides virtual memory handling, cpu access, exception forwarding and state reset
- **Tartiflette-executor**
    - **libAFL** executor component using **Tartiflette-VM**
    - Provides coverage collection, code hooking, syscall hooking, timeout handling



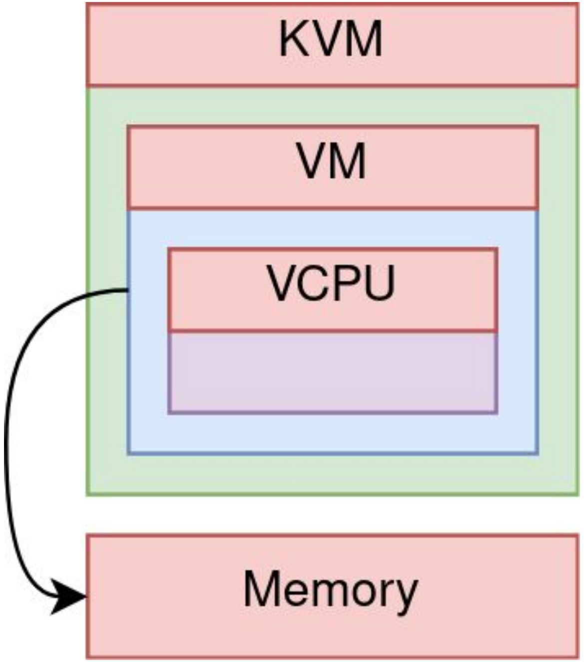
Tartiflette-vm: Memory handling

- **Tartiflette-vm** exposes virtual memory handling apis (mmap, read, write):
  - Much like Unicorn
- Paging is implemented by the host
- Advantages:
  - Lower memory consumption
  - More flexible api (mmap)
- Disadvantages:
  - Does not expect guest code to meddle with physical memory
    - Tricky to fuzz kernel code
  - Cannot handle context switches (userland/kernel)
    - CPU context in ring0
    - All memory is mapped in ring0



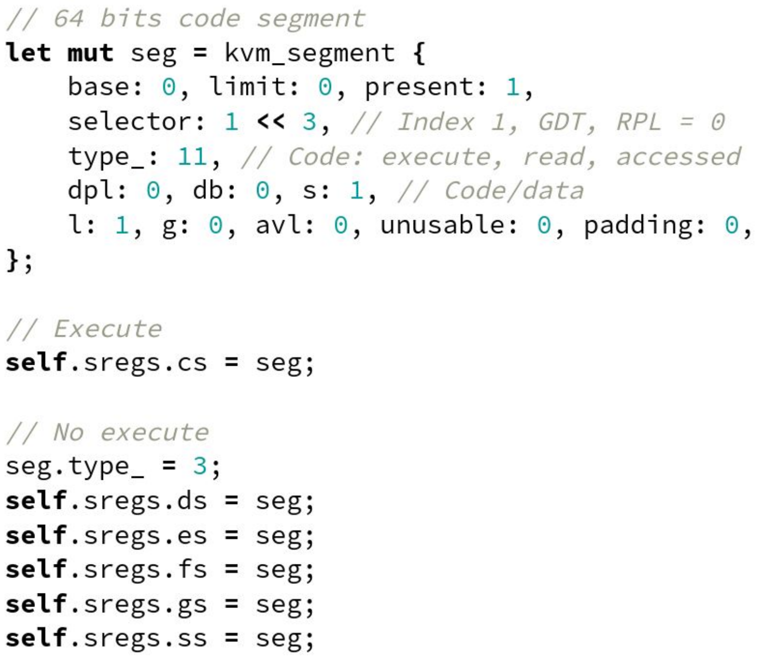
Tartiflette-vm: KVM setup

```
// 1 - Allocate the memory
let vm_memory = VirtualMemory::new(memory_size)?;

// 2 - Open the Kvm handles
let kvm_fd = Kvm::new()?;
let vm_fd = kvm_fd.create_vm()?;
let vcpu_fd = vm_fd.create_vcpu(0)?;

// 3 - Setup guest memory
let region = kvm_userspace_memory_region {
    slot: 0,
    guest_phys_addr: 0,
    memory_size: vm_memory.host_memory_size() as u64,
    userspace_addr: vm_memory.host_address(),
    flags: KVM_MEM_LOG_DIRTY_PAGES,
};
vm_fd.set_user_memory_region(region)?;
```

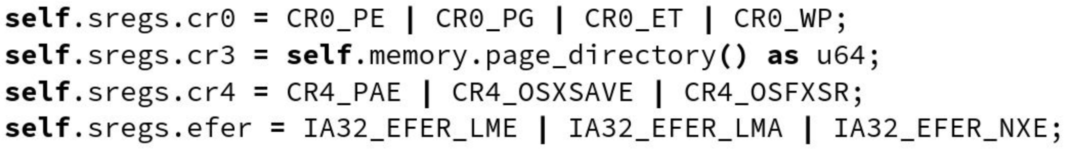


Tartiflette-vm: CPU setup

- Segments selectors:
  - 64 bits segments
- Controls registers
  - Paging
- Model specific registers:
  - 64 bits modes



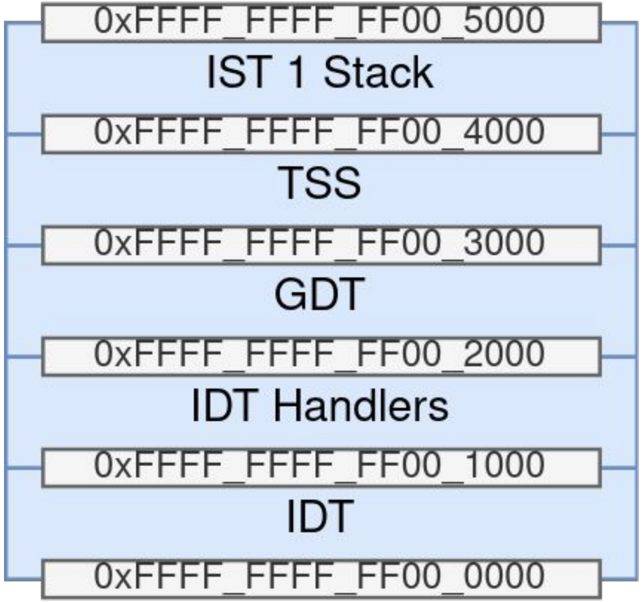
```
// 64 bits code segment
let mut seg = kvm_segment {
    base: 0, limit: 0, present: 1,
    selector: 1 << 3, // Index 1, GDT, RPL = 0
    type_: 11, // Code: execute, read, accessed
    dpl: 0, db: 0, s: 1, // Code/data
    l: 1, g: 0, avl: 0, unusable: 0, padding: 0,
};

// Execute
self.sregs.cs = seg;

// No execute
seg.type_ = 3;
self.sregs.ds = seg;
self.sregs.es = seg;
self.sregs.fs = seg;
self.sregs.gs = seg;
self.sregs.ss = seg;
```



```
self.sregs.cr0 = CR0_PE | CR0_PG | CR0_ET | CR0_WP;
self.sregs.cr3 = self.memory.page_directory() as u64;
self.sregs.cr4 = CR4_PAE | CR4_OSXSAVE | CR4_OSFXSR;
self.sregs.efer = IA32_EFER_LME | IA32_EFER_LMA | IA32_EFER_NXE;
```

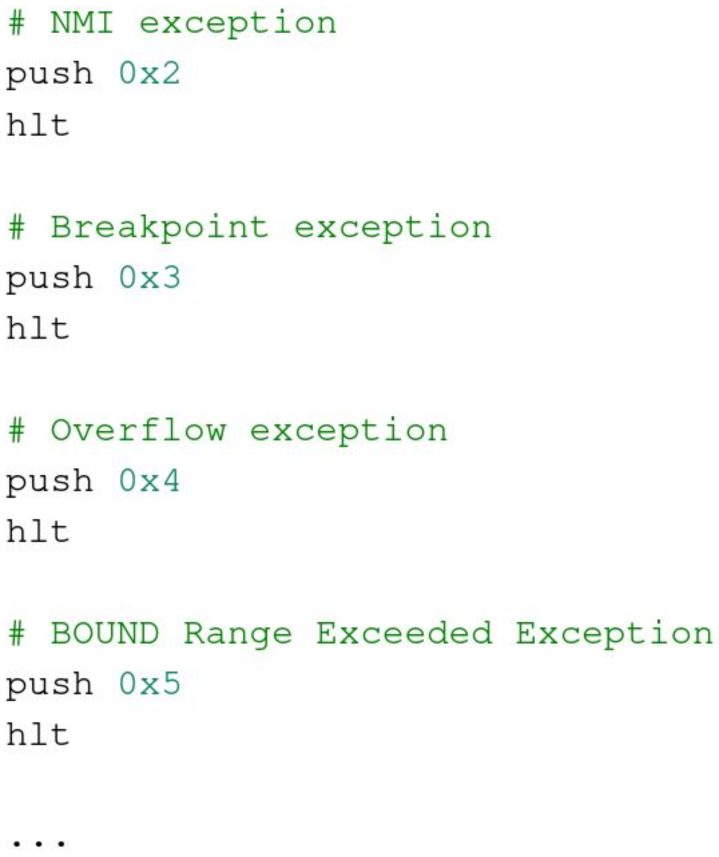
Tartiflette-vm: Exception handling setup

- GDT
  - NULL entry (as per the spec)
  - 64 bit dumb segment descriptor
  - TSS entry
  - Sets up **GDTR**
- IDT Handlers
  - Forwards guest exceptions to the host
- IDT
  - **IDT** entries pointing to corresponding handlers
  - Sets the **IST #1** as the interrupt stack to use
  - Sets up **IDTR**
- TSS
  - Sets up the first **IST** entry to point to the interrupt stack
- Interrupt stack
  - Separate stack to use during CPU exceptions



Tartiflette-vm: Exception forwarding

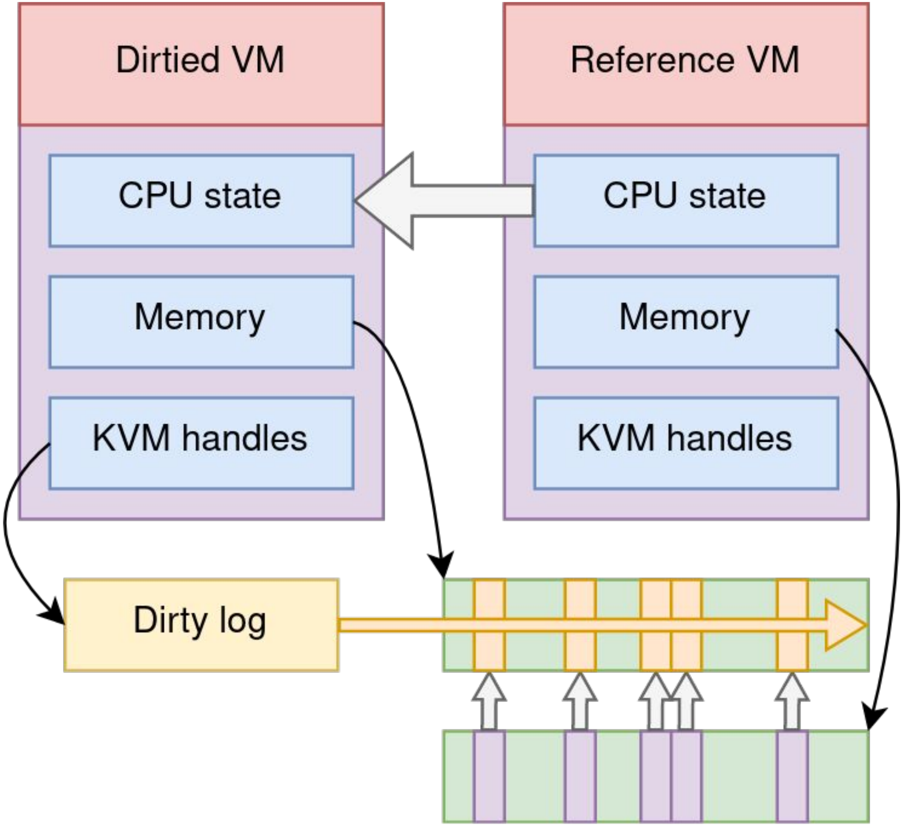
- Guest exceptions do not generate VMExits
    - Save for breakpoints with the **VM_GUESTDBG_ENABLE** and **KVM_GUESTDBG_USE_SW_BP** KVM
- Solution
    - Writes the trampoline entries to the **IDT Handlers** page, forwarding exceptions through a **HLT VMExit**
    - Point the **IDT** entries to the relevant trampolines



```
# NMI exception
push 0x2
hlt

# Breakpoint exception
push 0x3
hlt

# Overflow exception
push 0x4
hlt

# BOUND Range Exceeded Exception
push 0x5
hlt

...
```

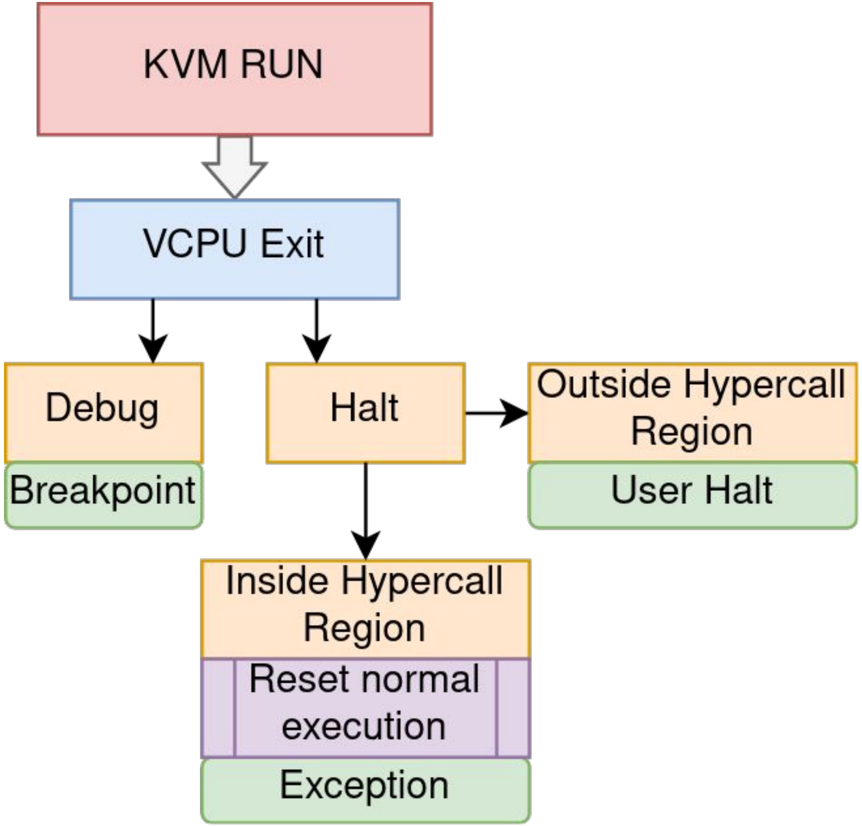
Tartiflette-vm: State reset

- Used to reset a VM after a fuzz case for another one
- Full CPU reset:
  - General purpose registers
  - Special registers (segments, crX)
  - Model specific registers (fs/gs)
- Differential memory reset:
  - Uses **KVM_LOG_DIRTY** bits bitmap to only reset dirty physical guest pages



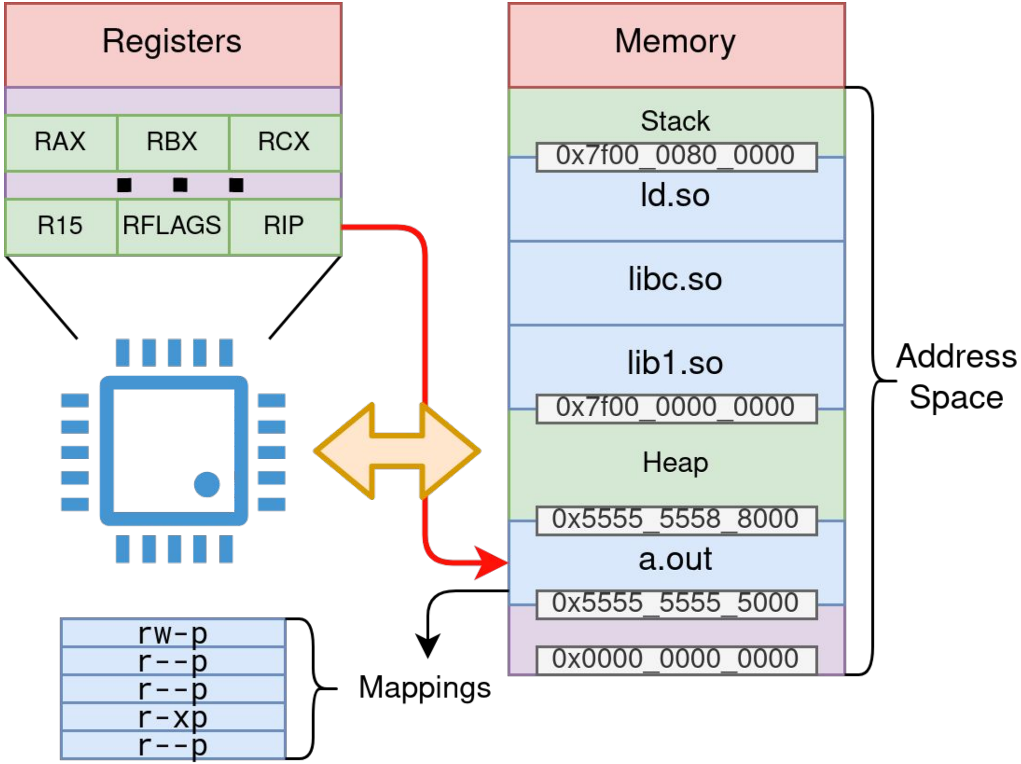
Tartiflette-vm: Run loop

- Let KVM run the VM’s VCPU
  - Until the first VCPU exit
- Get CPU full state
  - General purpose registers
  - Special registers (segments, crX)
- Handle the VCPU exit:
  - Debug: Forward breakpoint
  - Halt outside hypercall region: Forward breakpoint Halt
  - Halt inside hypercall region:
    - Get the exception frame
    - Reset execution before interruption
    - Forward interruption:
      - PageFault
      - Syscall: InvalidOpcode
      - ...



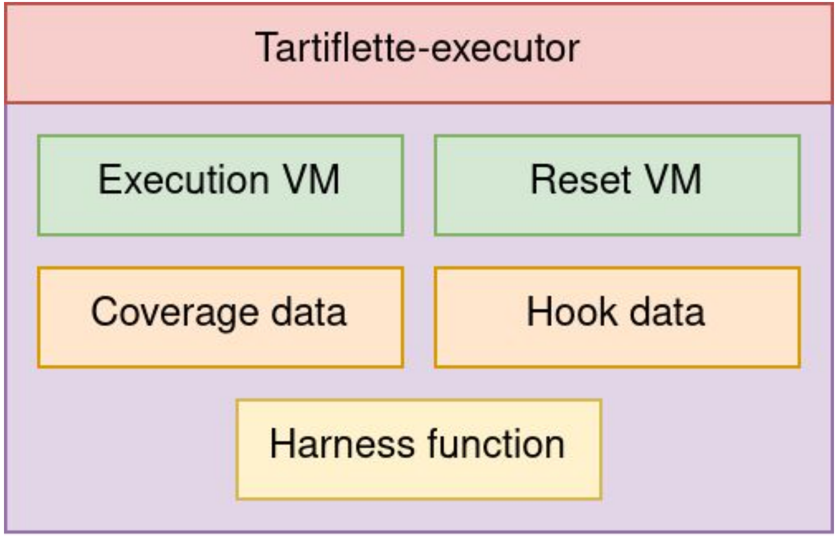
Tartiflette-vm: Snapshot format

- Binary file:
  - All dumped memory mappings
- JSON file:
  - CPU context
  - Mappings
    - Virtual start and end addresses
    - Page permissions
    - Physical offset into the binary file
    - Owning file (if any)
  - Symbols (if any)



Tartiflette-executor: Overview

- Implements the fuzzing runtime:
  - Harness:
    - Writes the input and sets up the Vm state
  - Coverage handling
    - Uses breakpoints on basic blocks
  - Timeout handling
  - Hook handling
  - State reset
- Exposes instrumentation apis:
  - Code hooks
  - Syscall hooks
  - Coverage hooks



Tartiflette-executor

Execution VM

Reset VM

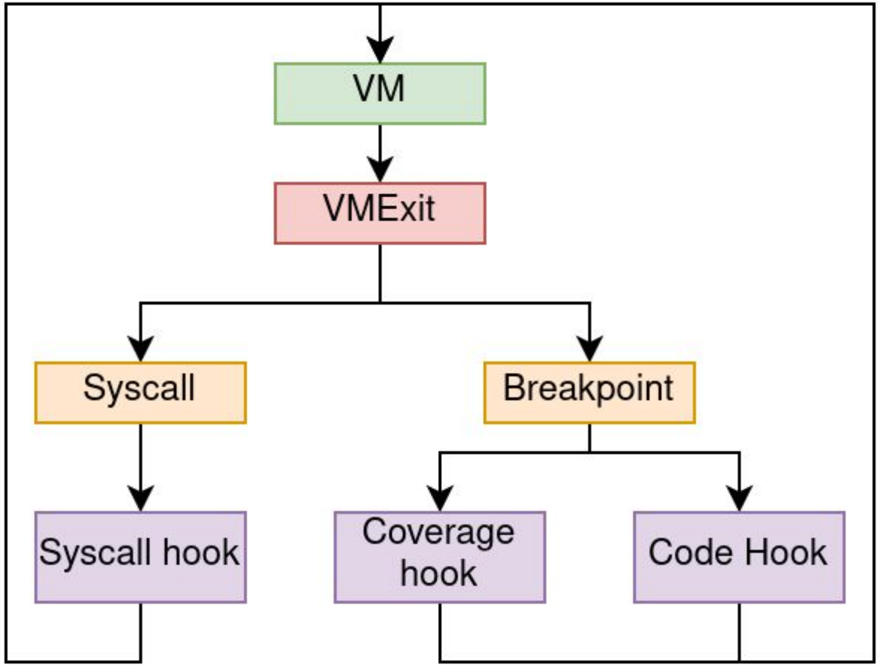
Coverage data

Hook data

Harness function

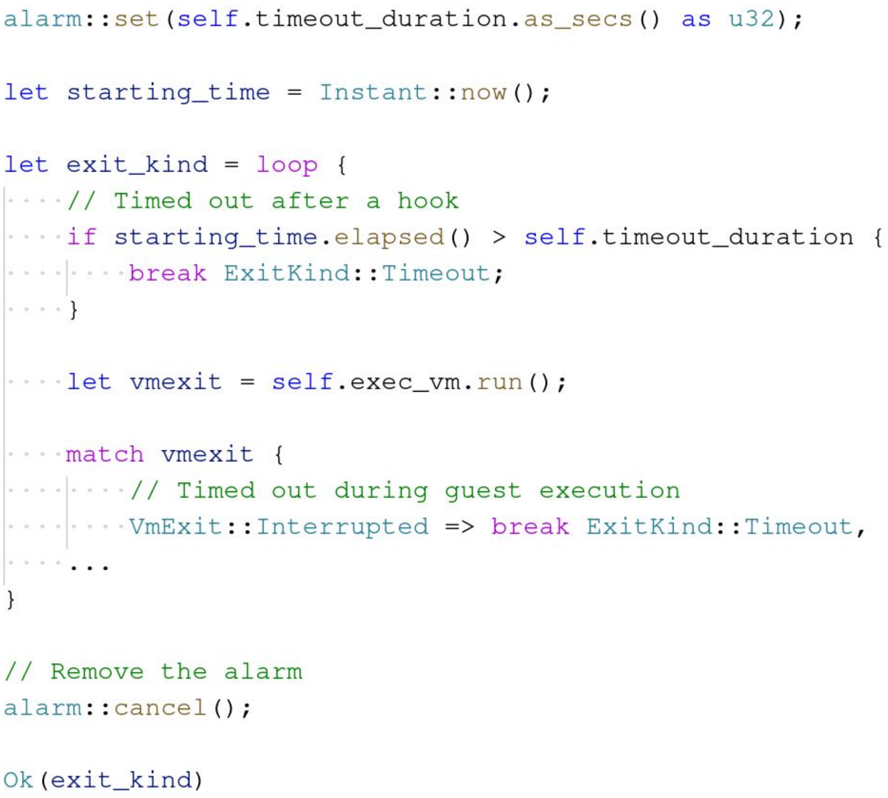
Tartiflette-executor: Hooks

- Code hooks:
  - For instrumentation
  - Returns whether the guest should continue, stop or crash
- Coverage hook:
  - For coverage logging (ex: lighthouse mod offset trace)
  - Called when a basic block is first encountered
- Syscall hooks:
  - For syscall emulation
  - Returns whether the execution should stop



Tartiflette-executor: Timeout

- **KVM** has no built-in timer to kick the VM out of virtualization
- We use alarm:
  - Kicks the kvm thread to handler
  - ioctl(KVM_RUN, …) returns EINTR -> we exit with timeout
  - If timeout occurred in vmexit handling, we check the starting time
- Alarm only offers a granularity in seconds :(



```
alarm::set(self.timeout_duration.as_secs() as u32);

let starting_time = Instant::now();

let exit_kind = loop {
    // Timed out after a hook
    if starting_time.elapsed() > self.timeout_duration {
        break ExitKind::Timeout;
    }

    let vmexit = self.exec_vm.run();

    match vmexit {
        // Timed out during guest execution
        VmExit::Interrupted => break ExitKind::Timeout,
    ...
}

// Remove the alarm
alarm::cancel();

Ok(exit_kind)
```

POC

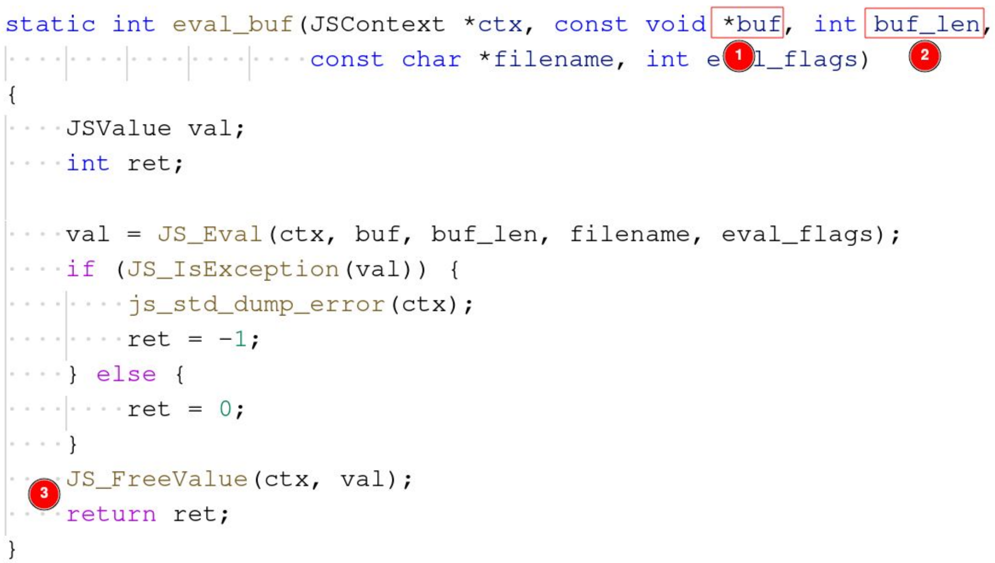
QuickJS

QuickJS: Overview

- Released in 2020
- Javascript Interpreter
- Project both small and complex:
  - Perfect for testing a fuzzer !
- Code not battle tested:
  - Bugs and exploits were found for older versions
    - http://rce.party/cracksbykim-quickJS.nfo
  - Potential to find bugs !

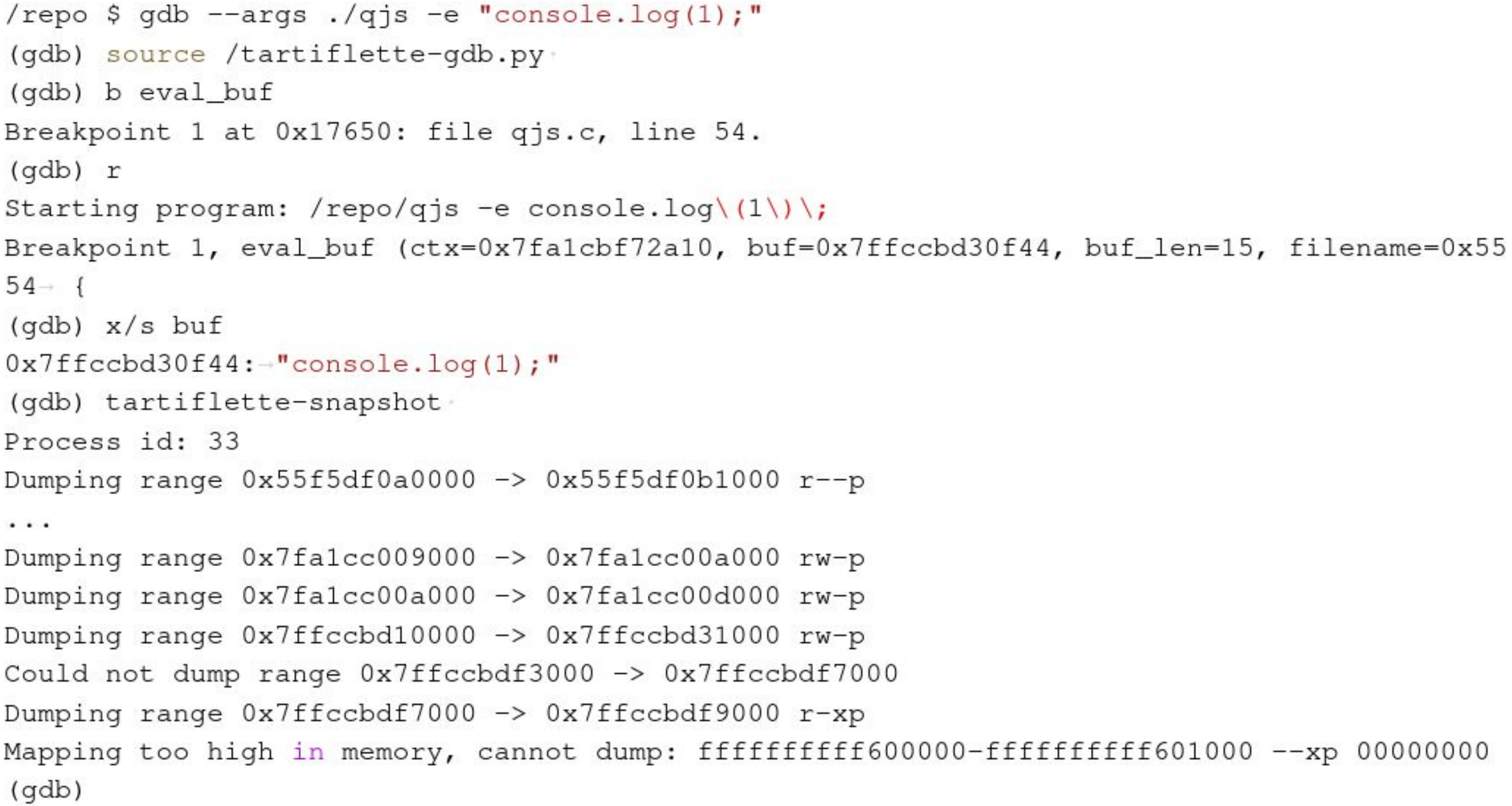
QuickJS: Capturing a snapshot

- We want to capture a snapshot before parsing and exec
- **qjs -e <expr>**
  - Executes the given expression and exits
  - Internally calls the **eval_buf** function
- With gdb:
  - Breakpoint at **eval_buf** and dump the program state



```
static int eval_buf(JSContext *ctx, const void *buf, int buf_len,
                    const char *filename, int eval_flags)
{
    JSValue val;
    int ret;

    val = JS_Eval(ctx, buf, buf_len, filename, eval_flags);
    if (JS_IsException(val)) {
        js_std_dump_error(ctx);
        ret = -1;
    } else {
        ret = 0;
    }
    JS_FreeValue(ctx, val);
    return ret;
}
```

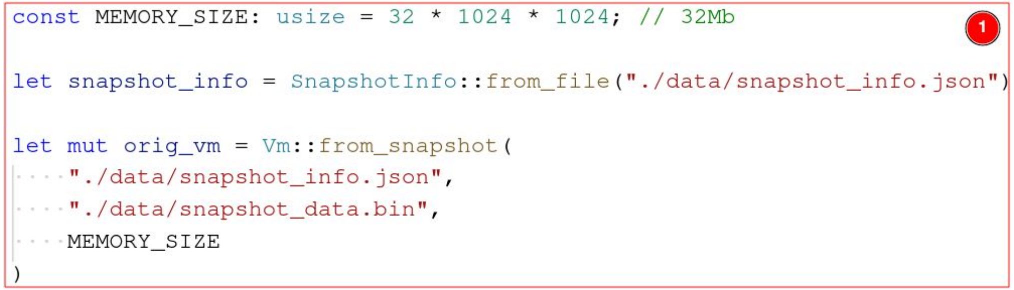
QuickJS: Capturing a snapshot



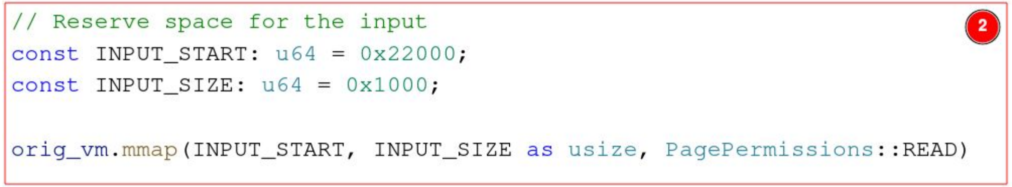
```
/repo $ gdb --args ./qjs -e "console.log(1);"
(gdb) source /tartiflette-gdb.py
(gdb) b eval_buf
Breakpoint 1 at 0x17650: file qjs.c, line 54.
(gdb) r
Starting program: /repo/qjs -e console.log\(1\)\;
Breakpoint 1, eval_buf (ctx=0x7fa1cbf72a10, buf=0x7ffccbd30f44, buf_len=15, filename=0x55
54	{
(gdb) x/s buf
0x7ffccbd30f44:	"console.log(1);"
(gdb) tartiflette-snapshot
Process id: 33
Dumping range 0x55f5df0a0000 -> 0x55f5df0b1000 r--p
...
Dumping range 0x7fa1cc009000 -> 0x7fa1cc00a000 rw-p
Dumping range 0x7fa1cc00a000 -> 0x7fa1cc00d000 rw-p
Dumping range 0x7ffccbd10000 -> 0x7ffccbd31000 rw-p
Could not dump range 0x7ffccbdf3000 -> 0x7ffccbdf7000
Dumping range 0x7ffccbdf7000 -> 0x7ffccbdf9000 r-xp
Mapping too high in memory, cannot dump: ffffffffff600000-ffffffffff601000 --xp 00000000
(gdb)
```

QuickJS Fuzzer: VM Setup

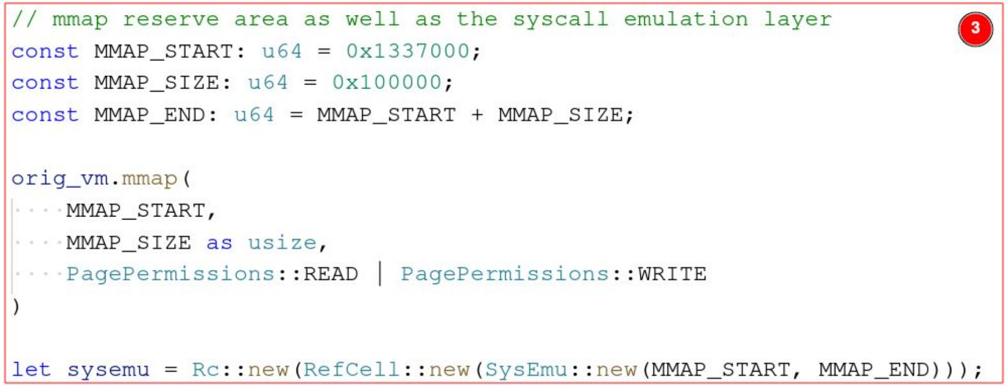
1. Load Vm from snapshot:
    a. Parse Snapshot information
    b. Load snapshot into a fresh VM
2. Allocate space for the Input
3. Setup the syscall emulation layer:
    a. To emulate syscalls triggered in code running in the vm
    b. Allocate space for possible returned resources accessible in the vm



```
const MEMORY_SIZE: usize = 32 * 1024 * 1024; // 32Mb

let snapshot_info = SnapshotInfo::from_file("./data/snapshot_info.json")

let mut orig_vm = Vm::from_snapshot(
    "./data/snapshot_info.json",
    "./data/snapshot_data.bin",
    MEMORY_SIZE
)
```



```
// Reserve space for the input
const INPUT_START: u64 = 0x22000;
const INPUT_SIZE: u64 = 0x1000;

orig_vm.mmap(INPUT_START, INPUT_SIZE as usize, PagePermissions::READ)
```



```
// mmap reserve area as well as the syscall emulation layer
const MMAP_START: u64 = 0x1337000;
const MMAP_SIZE: u64 = 0x100000;
const MMAP_END: u64 = MMAP_START + MMAP_SIZE;

orig_vm.mmap(
    MMAP_START,
    MMAP_SIZE as usize,
    PagePermissions::READ | PagePermissions::WRITE
)

let sysemu = Rc::new(RefCell::new(SysEmu::new(MMAP_START, MMAP_END)));
```

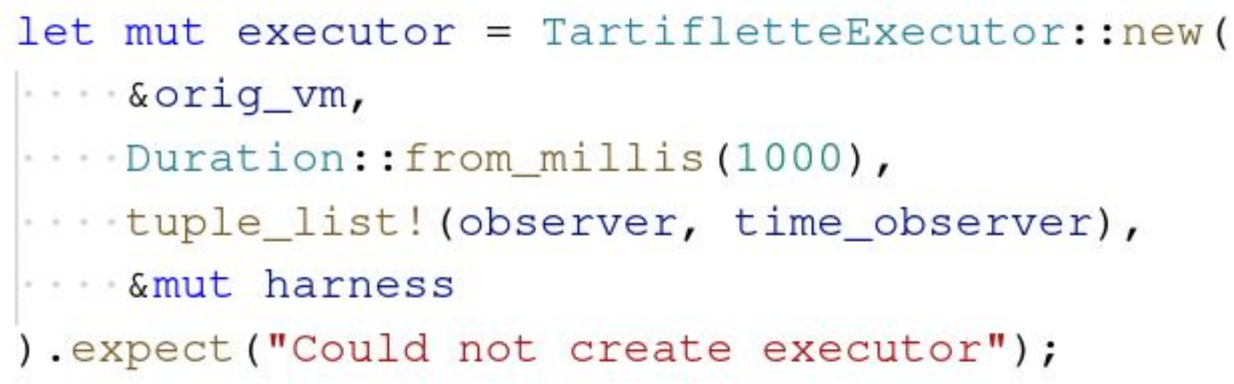
QuickJS Fuzzer: libAFL Setup

- Create the standard components:
    - Observer:
        - Channel through which collected events can be queried
    - Feedback:
        - Determine whether input is interesting, using observers data.
    - State
    - Fuzzer

```
static mut COVERAGE: [u8; 1 << 15] = [0u8; 1 << 15];

let observer = StdMapObserver::new("coverage", unsafe { &mut COVERAGE });
let time_observer = TimeObserver::new("time");

let feedback_state = MapFeedbackState::with_observer(&observer);
let feedback = feedback_or!(
    MaxMapFeedback::new(&feedback_state, &observer),
    TimeFeedback::new_with_observer(&time_observer)
);
let objective = CrashFeedback::new();

let mut state = StdState::new(
    // Randomness sources
    StdRand::with_seed(current_nanos()),
    // Input corpus
    InMemoryCorpus::new(),
    // Solution corpus
    InMemoryCorpus::new(),
    // Feedback states
    tuple_list!(feedback_state),
    )
});

let corpus_scheduler = QueueCorpusScheduler::new();
let mut fuzzer = StdFuzzer::new(corpus_scheduler, feedback, objective);
```

QuickJS Fuzzer: Tartiflette-Executor

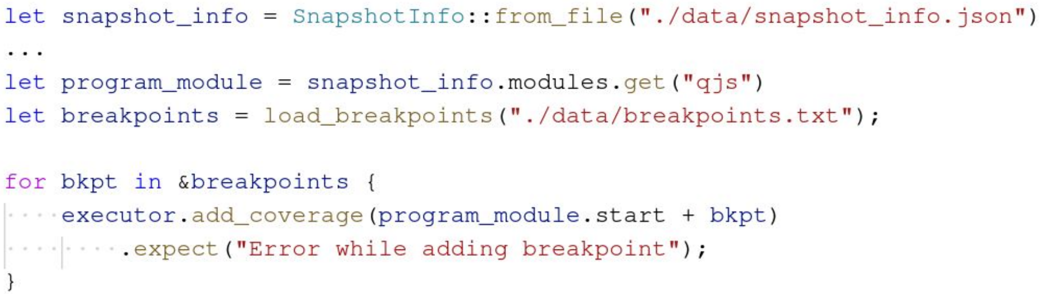
- Take an input
- Run the target with it
- Inside a Tartiflette-VM!



```
let mut executor = TartifletteExecutor::new(
    &orig_vm,
    Duration::from_millis(1000),
    tuple_list!(observer, time_observer),
    &mut harness
).expect("Could not create executor");
```

QuickJS Fuzzer: Coverage points

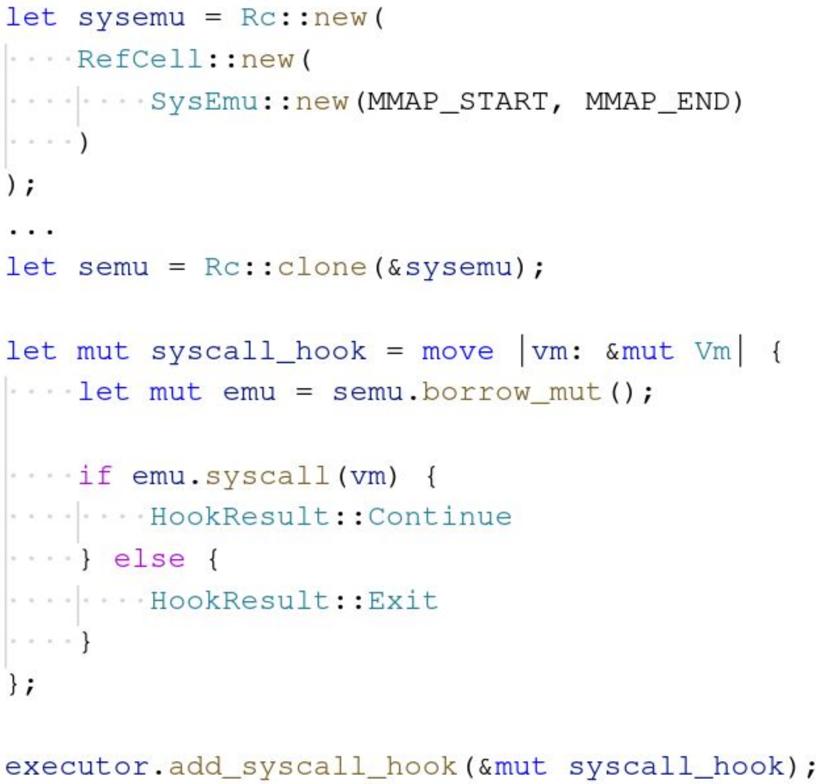
- A disassembler can be used to dump basic block addresses for coverage
- The snapshot info json file can give the base address of a module
- This gives flexibility without having to hardcode addresses in the fuzzer



```
let snapshot_info = SnapshotInfo::from_file("./data/snapshot_info.json")
...
let program_module = snapshot_info.modules.get("qjs")
let breakpoints = load_breakpoints("./data/breakpoints.txt");

for bkpt in &breakpoints {
    executor.add_coverage(program_module.start + bkpt)
        .expect("Error while adding breakpoint");
}
```

QuickJS Fuzzer: Syscall hook

- Called on each syscall
- **src/sysemu.rs** implements some useful standard syscalls:
    - **mmap**
    - **munmap**
    - **exit_group**
- Enough to make calls to malloc work and to stop the program cleanly on errors



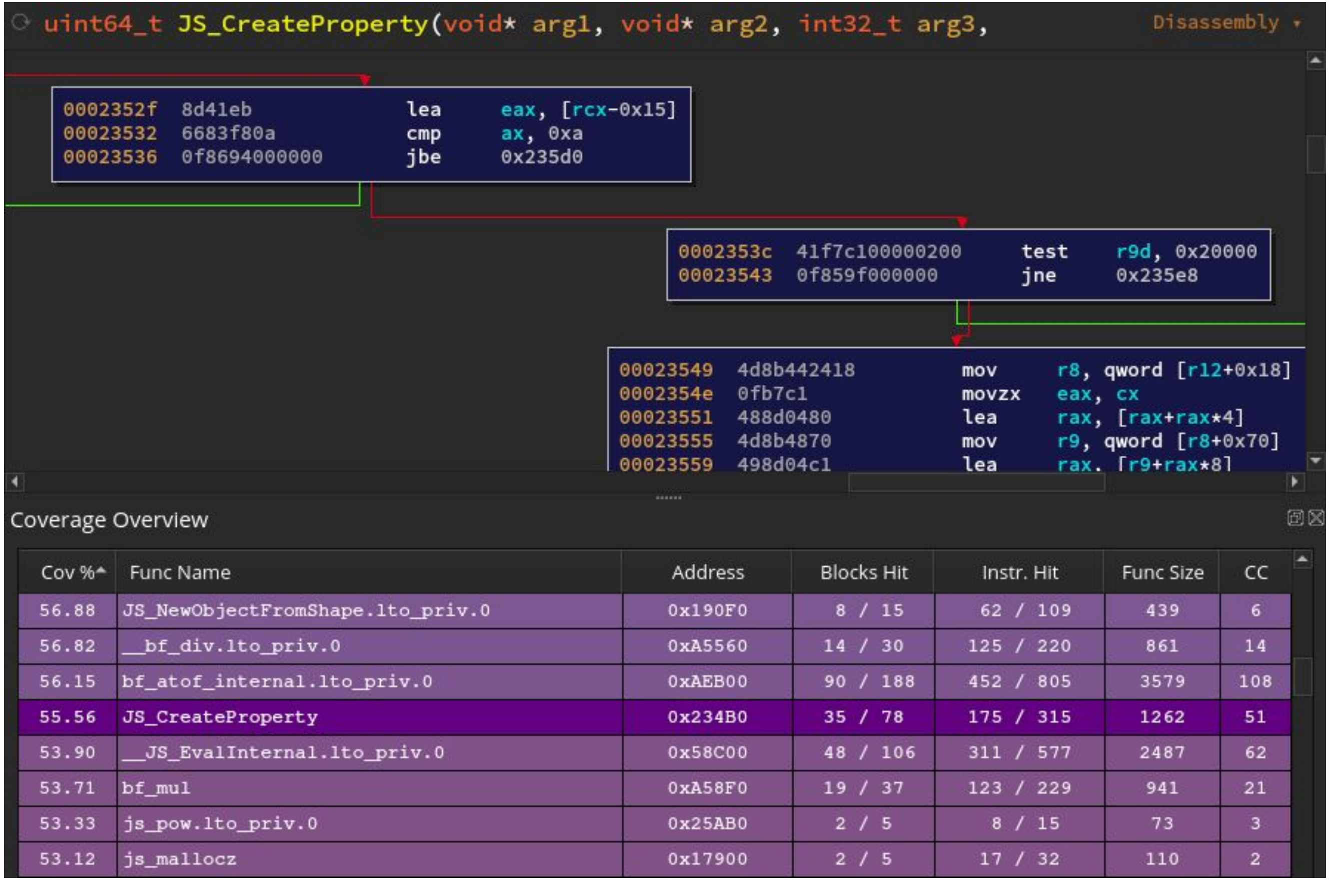
```
let sysemu = Rc::new(
    RefCell::new(
        SysEmu::new(MMAP_START, MMAP_END)
    )
);
...
let semu = Rc::clone(&sysemu);

let mut syscall_hook = move |vm: &mut Vm| {
    let mut emu = semu.borrow_mut();

    if emu.syscall(vm) {
        HookResult::Continue
    } else {
        HookResult::Exit
    }
};

executor.add_syscall_hook(&mut syscall_hook);
```

QuickJS Fuzzer: Coverage hook

- Called on each new basic block discovered
- Useful for generating traces (for tools such as lighthouse)

```
let cov_file = File::create("cov.txt");
let mut cov_file = LineWriter::new(cov_file);
let mod_base = program_module.start;

let mut coverage_hook = move |addr| {
    let offset = addr - mod_base;
    write!(cov_file, "qjs+0x{:x}\n", offset)
};

executor.add_coverage_hook(&mut coverage_hook);
```

QuickJS Fuzzer: Coverage hook



```
uint64_t JS_CreateProperty(void* arg1, void* arg2, int32_t arg3,

0002352f  8d41eb          lea    eax, [rcx-0x15]
00023532  6683f80a        cmp    ax, 0xa
00023536  0f8694000000    jbe    0x235d0

0002353c  41f7c100000200  test   r9d, 0x20000
00023543  0f859f000000    jne    0x235e8

00023549  4d8b442418      mov    r8, qword [r12+0x18]
0002354e  0fb7c1          movzx  eax, cx
00023551  488d0480        lea    rax, [rax+rax*4]
00023555  4d8b4870        mov    r9, qword [r8+0x70]
00023559  498d04c1        lea    rax, [r9+rax*8]
```

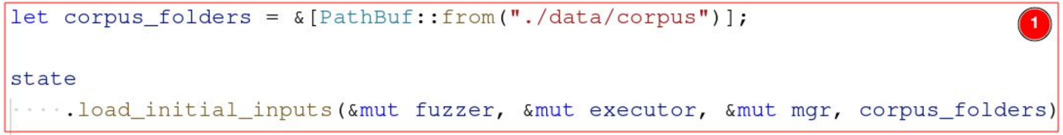
Coverage Overview

Cov %	Func Name	Address	Blocks Hit	Instr. Hit	Func Size	CC
56.88	JS_NewObjectFromShape.lto_priv.0	0x190F0	8 / 15	62 / 109	439	6
56.82	__bf_div.lto_priv.0	0xA5560	14 / 30	125 / 220	861	14
56.15	bf_atof_internal.lto_priv.0	0xAEB00	90 / 188	452 / 805	3579	108
55.56	JS_CreateProperty	0x234B0	35 / 78	175 / 315	1262	51
53.90	__JS_EvalInternal.lto_priv.0	0x58C00	48 / 106	311 / 577	2487	62
53.71	bf_mul	0xA58F0	19 / 37	123 / 229	941	21
53.33	js_pow.lto_priv.0	0x25AB0	2 / 5	8 / 15	73	3
53.12	js_mallocz	0x17900	2 / 5	17 / 32	110	2

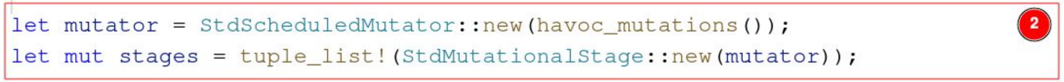
Coverage trace loaded in lighthouse

QuickJS Fuzzer: Launching the fuzzer

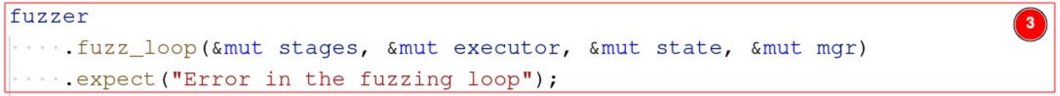
1. Load corpus from directory
2. Prepares the input transformation stages:
   a. We only use a standard byte mutator
   b. No pre/post preprocessing stages
3. Runs the fuzzer !



```
let corpus_folders = &[PathBuf::from("./data/corpus")];

state
    .load_initial_inputs(&mut fuzzer, &mut executor, &mut mgr, corpus_folders)
```



```
let mutator = StdScheduledMutator::new(havoc_mutations());
let mut stages = tuple_list!(StdMutationalStage::new(mutator));
```



```
fuzzer
    .fuzz_loop(&mut stages, &mut executor, &mut state, &mut mgr)
    .expect("Error in the fuzzing loop");
```

QuickJS Fuzzer: Javascript fuzzing

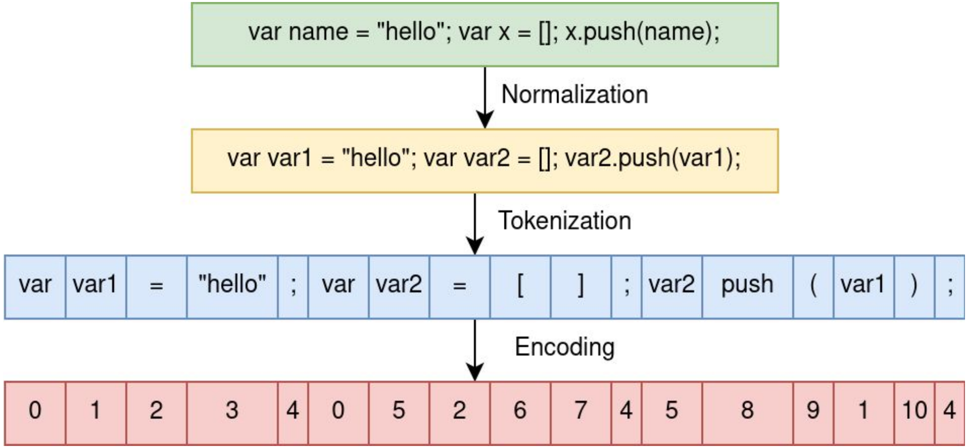
- Usual javascript fuzzing techniques:
  - Dictionary (**AFL**, **Jackalope**)
  - Grammar fuzzing (**domato**, **dharma**)
  - Black Magic (**Fuzzilli**)
- The standard libAFL mutator operates on bytes.
  - Bad for text based inputs
  - Bad for highly structured inputs
  - Bad for javascript fuzzing
- How to generate javascript code from bytes ?

QuickJS Fuzzer: Token-Level Fuzzing

- Taken from the USENIX ‘21 - Token-Level Fuzzing presentation
  - Augmented dictionary fuzzing
- Encodes javascript tokens as byte arrays:
  - Encoded input can be mutated by a classic byte mutator
  - Input is decoded back to javascript before being sent to the program
- Advantages:
  - Quick to build
  - Low cost of maintenance
- Disadvantages:
  - May not go as deep as grammar based fuzzers
  - High chance of syntactically invalid inputs

QuickJS Fuzzer: Token-Level Fuzzing (encoding)

- Code normalization:
  - Done using a javascript obfuscator and custom variable name dictionary
- Tokenization:
  - Done using the esprima js parser
- Output:
  - Encoded javascript code (per input)
  - Mapping of javascript tokens to their indices (shared across all inputs)



var name = "hello"; var x = []; x.push(name);

Normalization

var var1 = "hello"; var var2 = []; var2.push(var1);

Tokenization

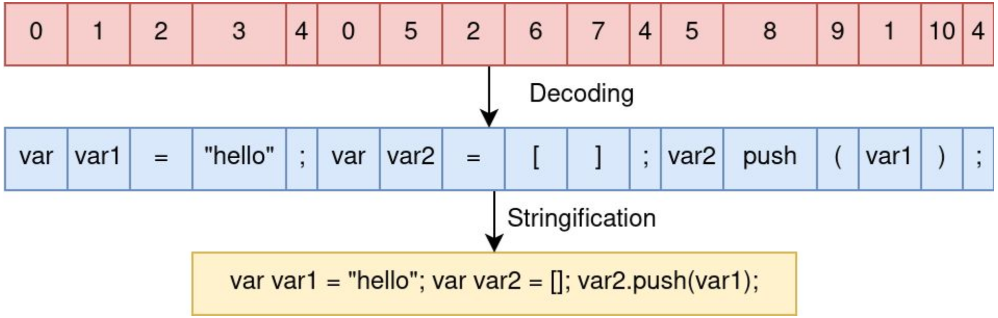
var	var1	=	"hello"	;	var	var2	=	[	]	;	var2	push	(	var1	)	;

Encoding

0	1	2	3	4	0	5	2	6	7	4	5	8	9	1	10	4

QuickJS Fuzzer: Token-Level Fuzzing (decoding)

- Inputs:
  - Encoded javascript code
  - Token mapping
- Decoding:
  - Map token indices to their string representation
- Stringification:
  - Concat everything
- Output:
  - Javascript code



0	1	2	3	4	0	5	2	6	7	4	5	8	9	1	10	4

Decoding

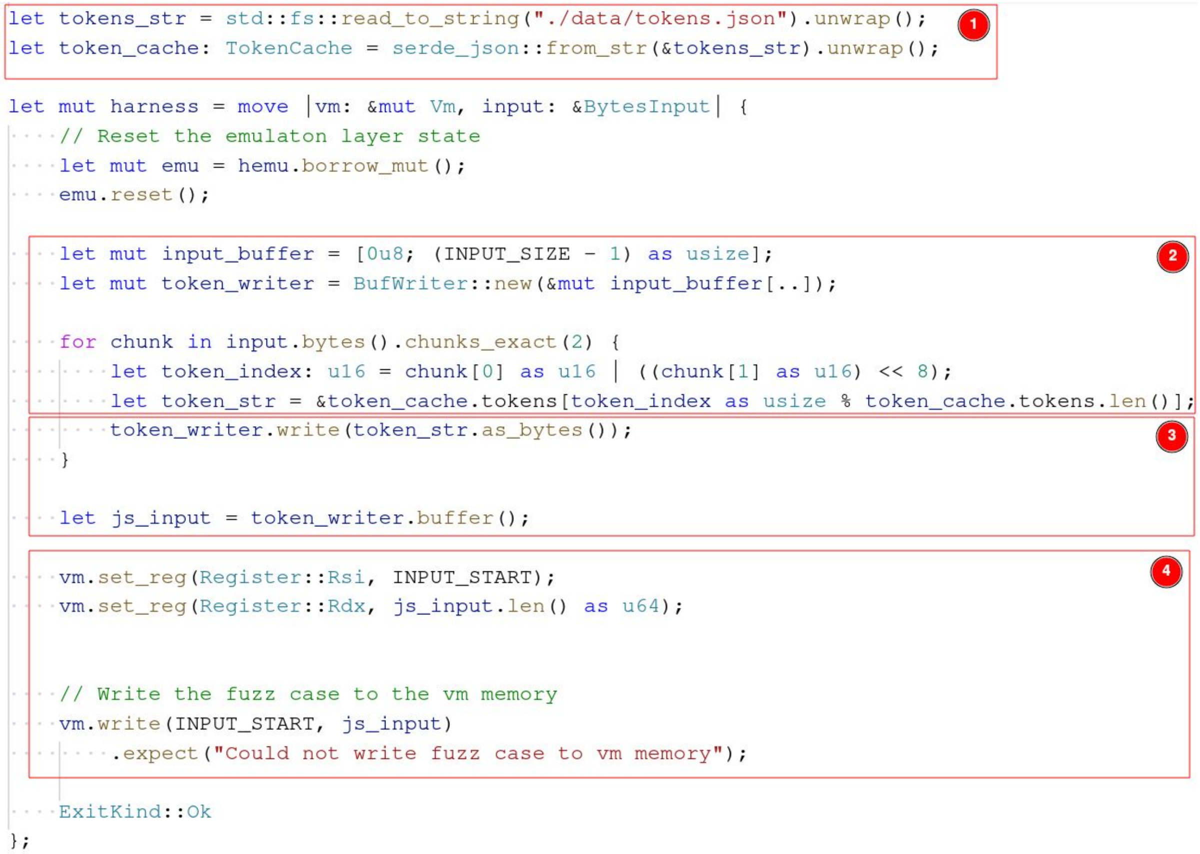
var	var1	=	"hello"	;	var	var2	=	[	]	;	var2	push	(	var1	)	;

Stringification

var var1 = "hello"; var var2 = []; var2.push(var1);

QuickJS Fuzzer: Token-Level Fuzzing (harness)

1. Load token map
2. Decode token to strings
3. Stringify
4. Write the input to the VM memory



```
let tokens_str = std::fs::read_to_string("./data/tokens.json").unwrap();
let token_cache: TokenCache = serde_json::from_str(&tokens_str).unwrap();

let mut harness = move |vm: &mut Vm, input: &BytesInput| {
    // Reset the emulaton layer state
    let mut emu = hemu.borrow_mut();
    emu.reset();

    let mut input_buffer = [0u8; (INPUT_SIZE - 1) as usize];
    let mut token_writer = BufWriter::new(&mut input_buffer[..]);

    for chunk in input.bytes().chunks_exact(2) {
        let token_index: u16 = chunk[0] as u16 | ((chunk[1] as u16) << 8);
        let token_str = &token_cache.tokens[token_index as usize % token_cache.tokens.len()];
        token_writer.write(token_str.as_bytes());
    }

    let js_input = token_writer.buffer();

    vm.set_reg(Register::Rsi, INPUT_START);
    vm.set_reg(Register::Rdx, js_input.len() as u64);


    // Write the fuzz case to the vm memory
    vm.write(INPUT_START, js_input)
        .expect("Could not write fuzz case to vm memory");

    ExitKind::Ok
};
```

Tartiflette

DEMO

Questions ?

Links

- Repository:
    - https://github.com/MattGorko/Tartiflette