Unicorefuzz:

On the Viability of Emulation for Kernelspace Fuzzing

WOOT '19

Dominik Maier
Benedikt Radtke
Bastian Harren

2019-08-12

Security in Telecommunications
Technische Universität Berlin

Unicorefuzz

- Simplifies emulation-based fuzzing of kernel parsers
- Coverage guided, blackbox
- Based on
  - AFL-Unicorn
  - avatar²
- Finds bugs
- On any GDB target
- Open sourced

AFL has been around for quite some time

We still find buffer overflows like it’s 1996*

*Aleph One. **Smashing the Stack for Fun and Profit**. Phrack 7, 1996

Fuzzing is Hard

- Legacy code is not written to be tested
- Depending on globals, proper initializations, state, …
- How do we get input to the right place?

⇒ Setting up a fuzz harness is challenging

Fuzzing Kernels is Hard

- Hardware interactions
- Restarting Kernels for each test case needs more effort
- “Did it just crash?”
- How do we get input in that thing?

⇒ Setting up a kernel fuzz harness is even worse

People Are Fuzzing Kernels

- Trinity
- DIFUZE
- TriforceAFL
- Syzkaller
- kAFL
- ...

Example: Triforce AFL

- AFL's QEMU Mode
- Ported for Kernel Emulation
- Runs until special hypercall
- Starts Forkserver at that point

But:

- QEMU forks before the forkserver starts may be "strange"
- VM is heavy, has interrupts, non-deterministic
- Has to be a VM...

People are not Fuzzing Our Way

- Trinity -> No Coverage
- DIFUZE -> No Coverage
- TriforceAFL -> Shaky with forks in QEMU, etc.
- Syzkaller -> No* coverage for blackbox OS fuzzing
- kAFL -> Awesome but x86(_64) only

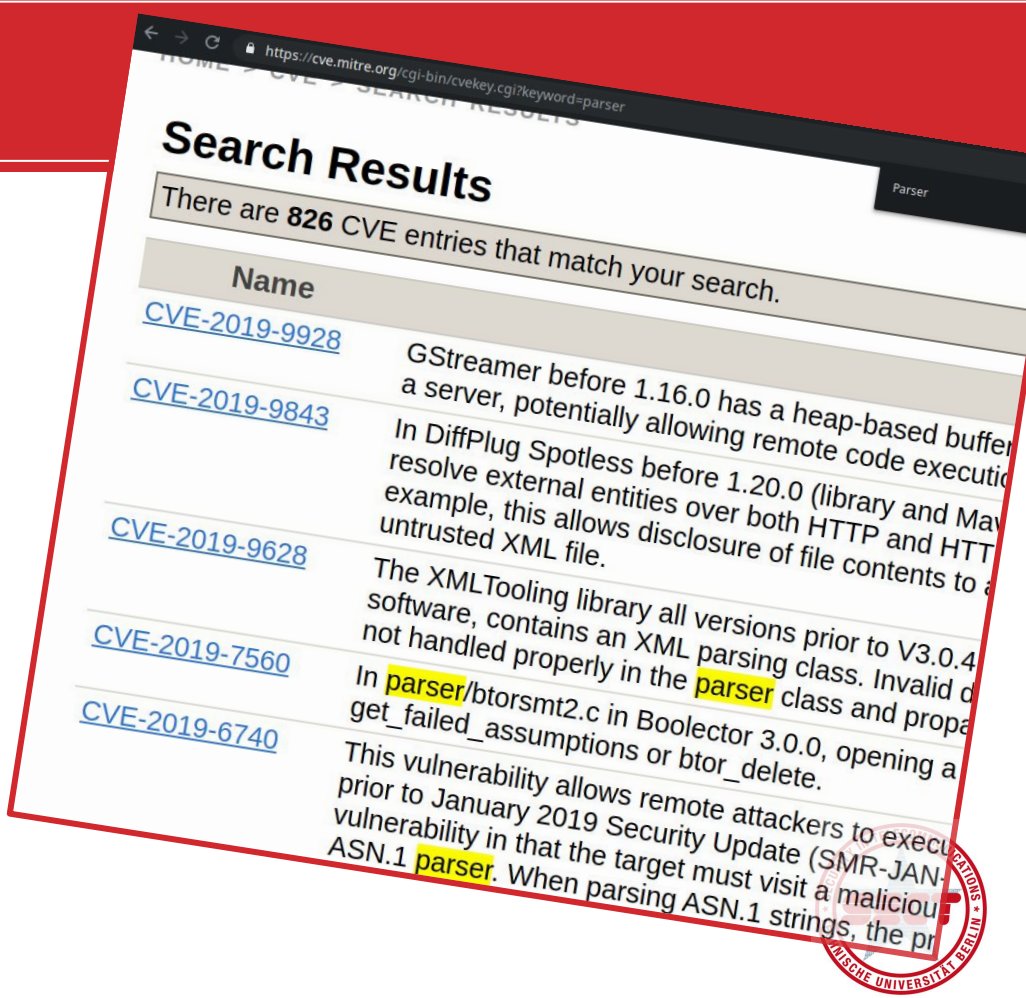
- Whatever Brandon Falk is doing -> Crazy ;)

* Apart from Windows fuzzing with kAFL's instrumentation

Idea: Rip out Parsers
Fuzz them somewhere else

Why Parsers

- They tend to break
- Often read from well-defined buffers
- little to no additional hardware interaction
- Have you seen bug-free ASN1 parsers?
- They tend to break



Copy&Paste Parsers to Userland?

Ideal solution!

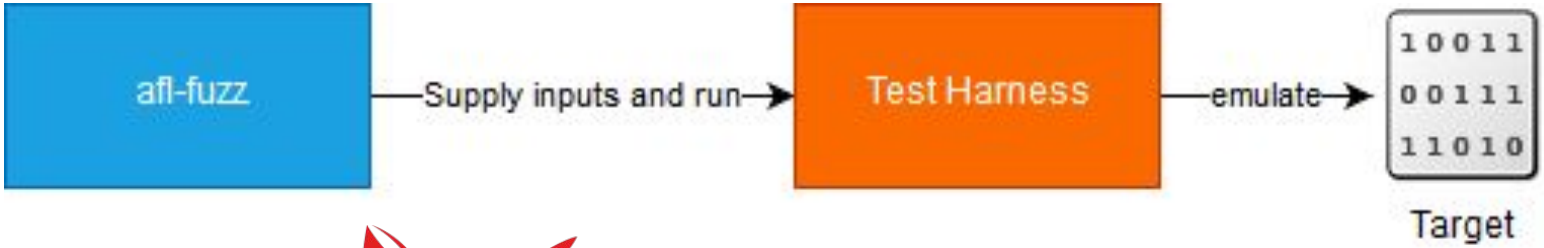
Problem: Code depends heavily on

- State and proper initializations
- All those kernel functions
- Source Code availability

⇒ Lots of work even with source

Idea: Rip out Parsers
Fuzz them in an Emulator

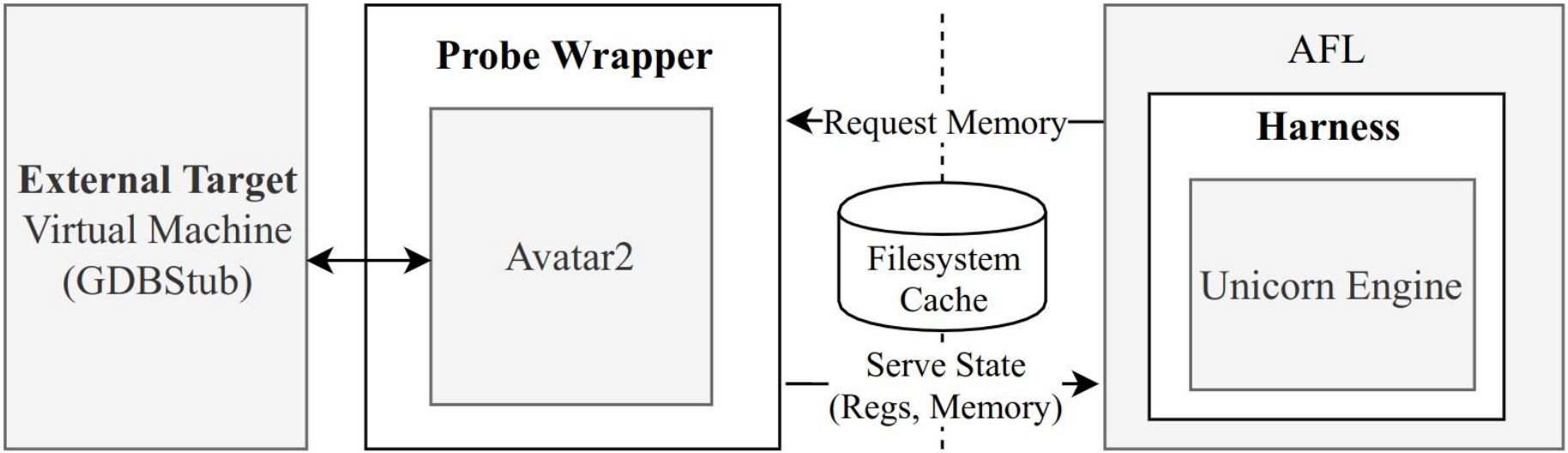
AFL-Unicorn



Unicorn:
CPU Emulator, Fork of QEMU
Multi-architectures: Arm, Arm64 (Armv8), M68K, Mips, Sparc, & X86 (include X86_64).

AFL-Unicorn:
Adds Instrumentation to Basic Blocks
Much like AFL QEMU

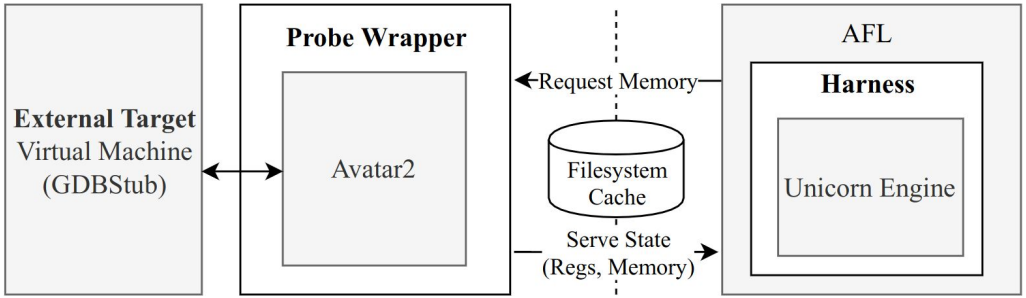
Unicorefuzz Architecture



Probe Wrapper

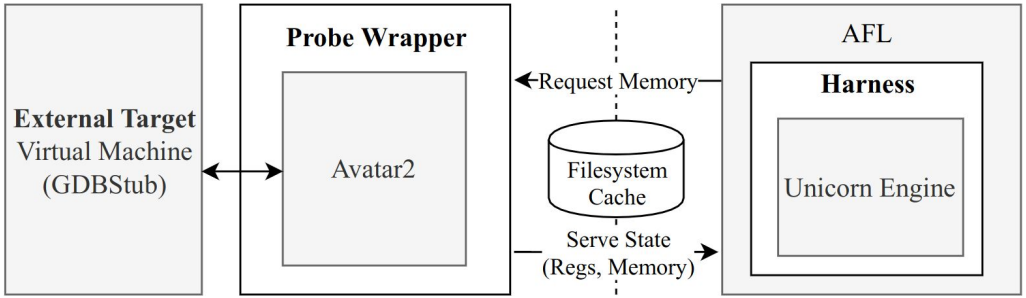
- Sets breakpoint on target
- Dumps all registers once bp triggers
- Waits for memory requests from harness
- Fetches memory via Avatar2/GDB on demand
- Memory exchange via file system

-> Can eventually be turned off

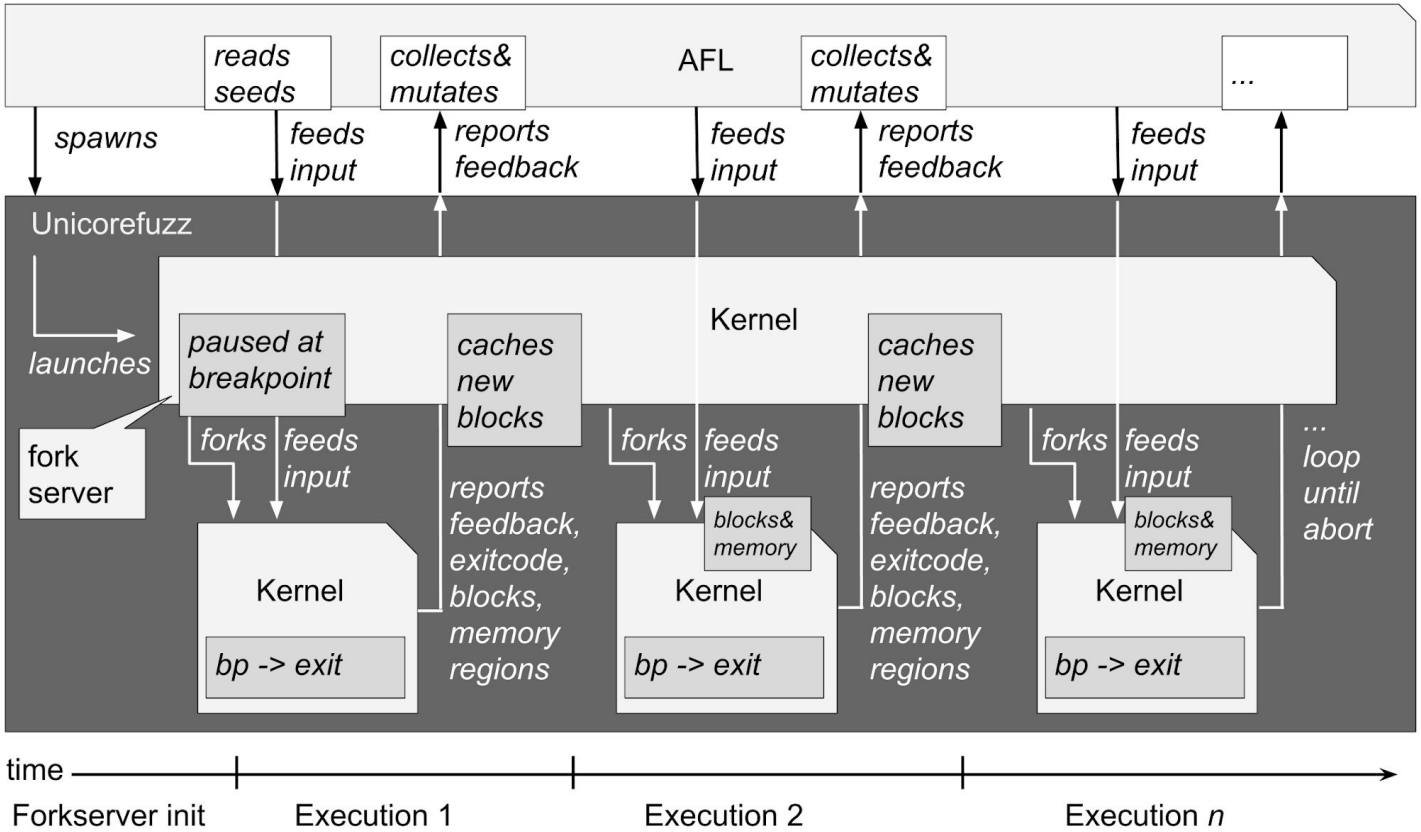


Unicorefuzz Harness

- Fork on first insn
- Child: between each code block
  - Request memory from Probe Wrapper if not mapped
  - Set bit for hash(from->to) in shared map
  - Cache the translated block in parent
  - Execute the translated block
- Fork again, with hot caches



AFL/Unicorefuzz Interactions



Unicorefuzz: Workflow

Workflow

Step 0: Download & make

Step 1: Find a parser (ghidra/ida/r2/...)

Step 2: Edit config.py

Step 3: Trigger parser

Step 4: Fuzz

Step 5: Triage

+Profit?

Step 0: Download & Install

```
git clone https://github.com/fgsect/unicorefuzz.git
cd unicorefuzz
./setupaflpp.sh
./setupdebug.sh #optional if you want to use uDdbg
./setaflops.sh # optional

[Get some target. To fuzz a QEMU VM, have a look at./startvm.sh]
```

Step 1: Find a Parser

Analyze the target

.../ghidra/ida/r2/...

Find a function that:

- takes input
- returns something
- actually gets called

+ find calling convention

Step 2: Edit config.py

For each target, the config.py needs to be altered.

Settings include:

- MODULE + BREAKOFFSET -> if fuzzing Linux .ko object
- BREAKADDR -> Breakpoint for anything else
- LENGTH & EXITS -> Where to exit
- implement init_func(uc, rip) -> if you need uc hooks
- Implement `place_input(uc, input)`

  ⇒ Function that drops AFL input at memory location

place_input() for Open vSwitch

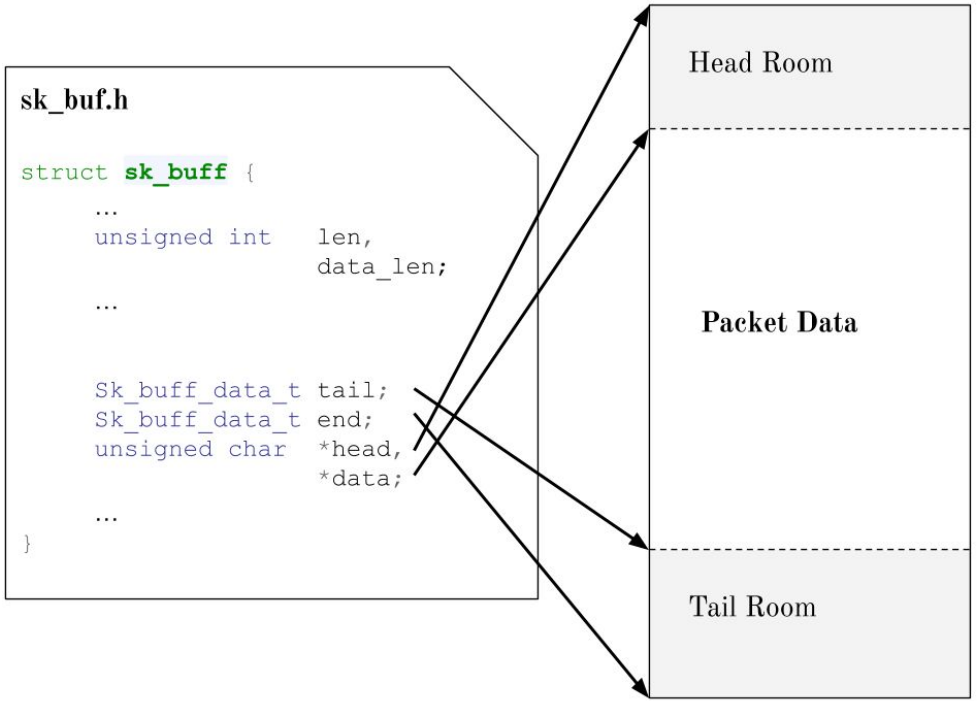
```
def place_input(uc, input):
    """
    Places the input in memory and alters the input.
    Example for sk_buff in openvswitch
    """
    ...
    if len(input) > 1500:
        import os
        os._exit(0) # too big!
    # read input to the correct position at param rdx here:
    rdx = uc.reg_read(UC_X86_REG_RDX)
    util.map_page_blocking(uc, rdx) # ensure sk_buf is mapped
    bufferPtr = struct.unpack("<Q",uc.mem_read(rdx + 0xd8, 8))[0]
    util.map_page_blocking(uc, bufferPtr) # ensure the buffer is mapped
    uc.mem_write(bufferPtr, input) # insert afl input
    uc.mem_write(rdx + 0xc4, b"\xdc\x05") # fix tail
```

Fuzzing OpenVSwitch

```
int ovs_flow_key_extract(struct sk_buff *skb, struct sw_flow_key *key)
```

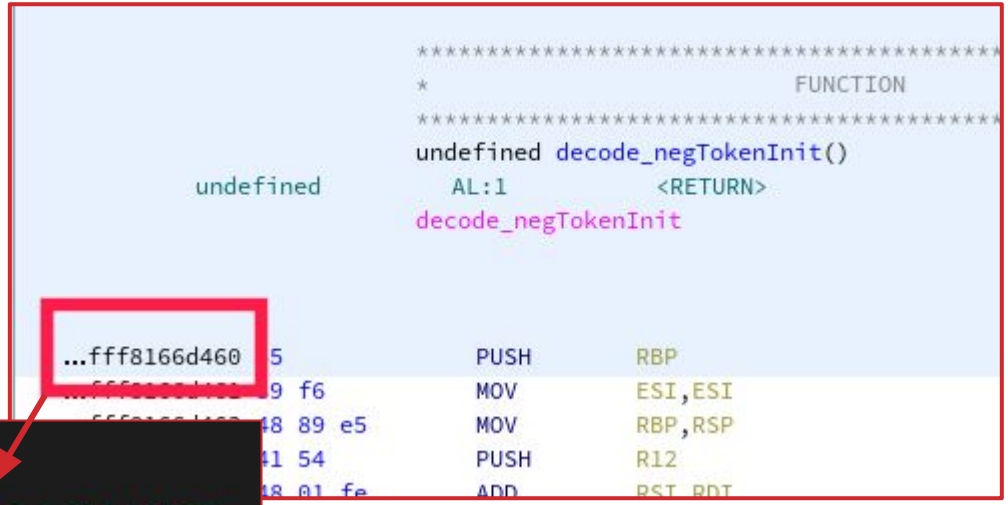
Drop input at
Packet Data

Fuzzing the whole skb would break all pointers
→ false positives...



Step 3: Trigger Parser

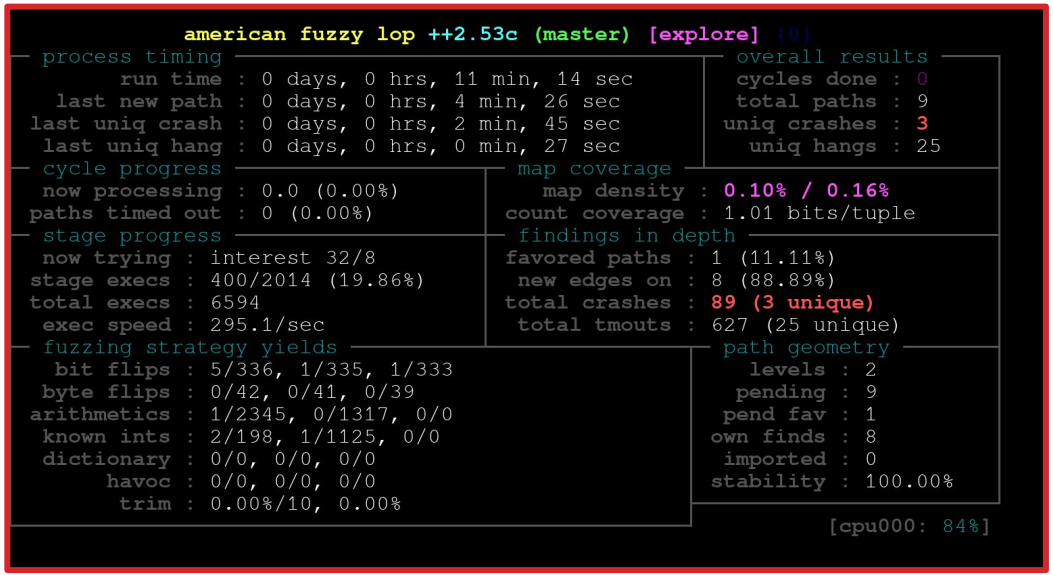
- ./probe_wrapper.py
- Make target exec to break point
- All right, let's fuzz!




```
❯ ./probe_wrapper.py
Breakpoint set at 0xffffffff8166d460
waiting for bp hit...
hit! dumping registers and memory
```

Step 4: Fuzz

- Add seeds to ./afl_inputs
- Run ./startafl.sh [workerid]
- Enjoy AFL



Step 5: Triage

Got a bug? Nice. Rerun the bug:

- On the target (hopefully)
- Using ./harness.py -t <input> for tracing
- Using ./harness.py -d <input> for some uDdb debugging

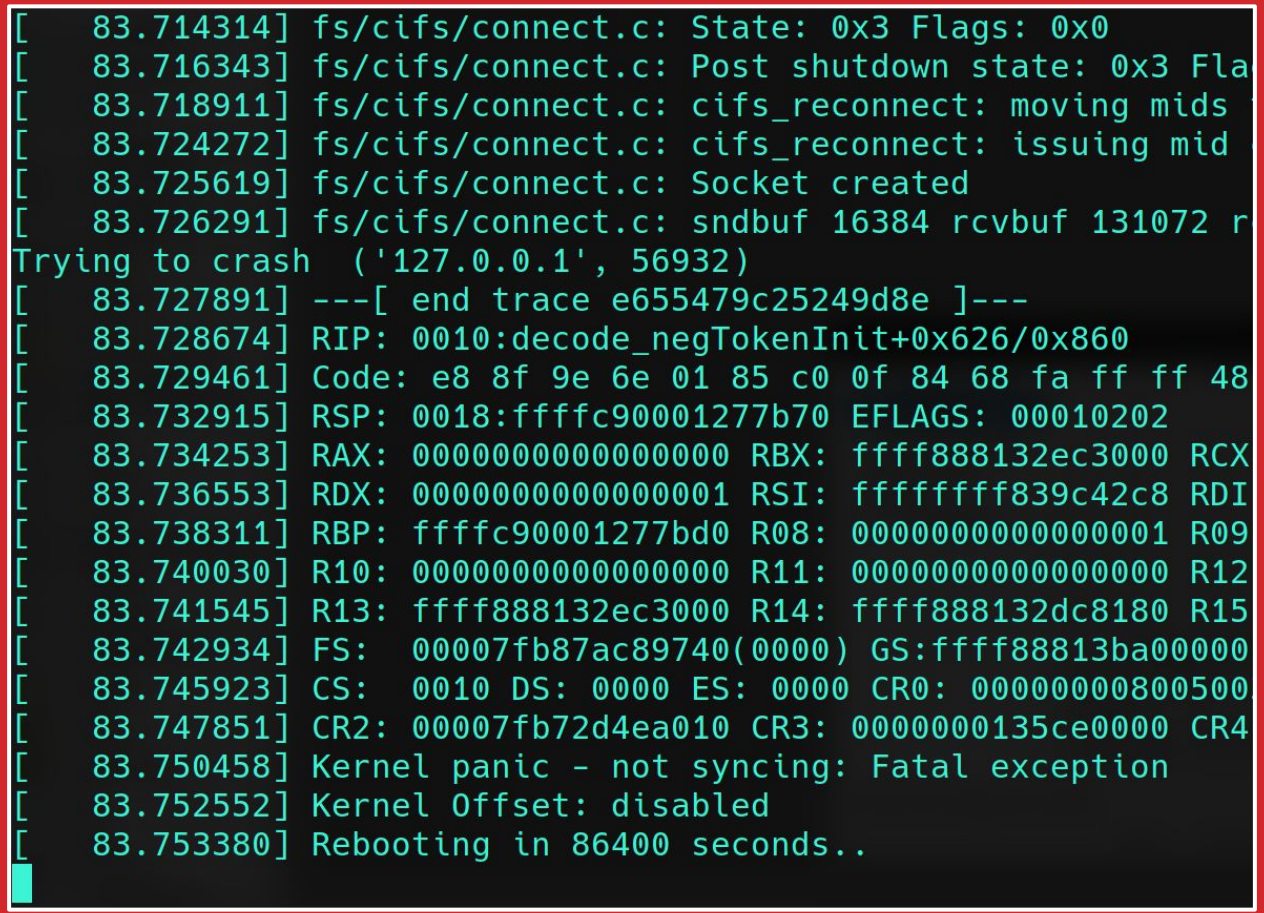
Trace

```
r: 0xffffffff8166da61:  test    eax, eax
r: 0xffffffff8166da63:  je      0xffffffffffffa74
Basic Block: addr=0xffffffff8166da69, size=0x000000000000002e
    Instr: 0xffffffff8166da69:  mov     rax, qword ptr [rbp - 0x50]
        >>> Read: addr=0xffffc90001457b80 size=8
    Instr: 0xffffffff8166da6d:  mov     rdi, -0x7bc511a8
    Instr: 0xffffffff8166da74:  mov     rsi, -0x7c63bd38
    Instr: 0xffffffff8166da7b:  mov     r8d, dword ptr [rbp - 0x5c]
        >>> Read: addr=0xffffc90001457b74 size=4
    Instr: 0xffffffff8166da7f:  mov     r9d, dword ptr [rbp - 0x58]
        >>> Read: addr=0xffffc90001457b78 size=4
    Instr: 0xffffffff8166da83:  mov     ecx, dword ptr [rbp - 0x60]
        >>> Read: addr=0xffffc90001457b70 size=4
    Instr: 0xffffffff8166da86:  movzx   edx, byte ptr [rax]
unicorn_debug_mem_invalid_access(uc=<unicorn.unicorn.Uc object at 0x7f
00000000000, size=1, value=0, ud=None)
        >>> INVALID Read: addr=0x0000000000000000 size=1
CAN I HAZ EXPLOIT?
[1]    24599 segmentation fault  ./harness.py -t ../unicorefuzz_cifs/c
```

Debug

```
                kffff0a                          16776970
                k3a                                 58
r13             0xffff888136bfe800  18446612687283546112
r14             0xffff8881358b83c0  18446612687263335360
r15             0xffff888136bfe800  18446612687283546112
eflags          0x202                              514
-----------------------------------------[ disasm ]------
0xffffffff8166d44b      ADD     BYTE PTR [RAx], AL
0xffffffff8166d44d      ADD     BYTE PTR [RAx], AL
0xffffffff8166d44f      XOR     EAx, EAx
0xffffffff8166d451      JMP     0xFFFFFFFF8166D2F1
0xffffffff8166d456      NOP     WORD PTR CS:[RAx + RAx]
0xffffffff8166d460      PUSH    RBP
0xffffffff8166d461      MOV     ESI, ESI
0xffffffff8166d463      MOV     RBP, RSP
0xffffffff8166d466      PUSH    R12
0xffffffff8166d468      ADD     RSI, RDI
0xffffffff8166d46b      PUSH    RBx
0xffffffff8166d46c      LEA     R8, [RBP - 0x58]
0xffffffff8166d470      MOV     RBx, RDx
0xffffffff8166d473      LEA     RCx, [RBP - 0x5C]
0xffffffff8166d477      SUB     RSP, 0x50
-----------------------------------------[ memory access ]----
WRITE   0x-36fffeba83a0 > 0xffffc90001457bd0
$>
```

DEMO


[83.714314] fs/cifs/connect.c: State: 0x3 Flags: 0x0
[83.716343] fs/cifs/connect.c: Post shutdown state: 0x3 Fla
[83.718911] fs/cifs/connect.c: cifs_reconnect: moving mids
[83.724272] fs/cifs/connect.c: cifs_reconnect: issuing mid
[83.725619] fs/cifs/connect.c: Socket created
[83.726291] fs/cifs/connect.c: sndbuf 16384 rcvbuf 131072 r
Trying to crash ('127.0.0.1', 56932)
[83.727891] ---[end trace e655479c25249d8e]---
[83.728674] RIP: 0010:decode_negTokenInit+0x626/0x860
[83.729461] Code: e8 8f 9e 6e 01 85 c0 0f 84 68 fa ff ff 48
[83.732915] RSP: 0018:ffffc90001277b70 EFLAGS: 00010202
[83.734253] RAX: 0000000000000000 RBX: ffff888132ec3000 RCX
[83.736553] RDX: 0000000000000001 RSI: ffffffff839c42c8 RDI
[83.738311] RBP: ffffc90001277bd0 R08: 0000000000000001 R09
[83.740030] R10: 0000000000000000 R11: 0000000000000000 R12
[83.741545] R13: ffff888132ec3000 R14: ffff888132dc8180 R15
[83.742934] FS: 00007fb87ac89740(0000) GS:ffff88813ba00000
[83.745923] CS: 0010 DS: 0000 ES: 0000 CR0: 00000000800500
[83.747851] CR2: 00007fb72d4ea010 CR3: 0000000135ce0000 CR4
[83.750458] Kernel panic - not syncing: Fatal exception
[83.752552] Kernel Offset: disabled
[83.753380] Rebooting in 86400 seconds..

Speed

- There is a ASN1 parser in the CIFS Filesystem driver
- So we start fuzzing at entrypoint
- ASN1 parser broken, as is tradition
- Input from remote, but needs local interaction
- In CIFS debug mode only (needs root to enable)

⇒ Not severe, but proves viability of Unicorefuzz

Speed

Single Core Speed Comparison for example.ko on a Laptop:

Framework	Execs/Sec
Unicorefuzz	459
AFL QEMU Mode	939
AFL	4860

TL;DR Not that great (yet).

But... simply throw more hardware at the problem.

Comparison Chart

Features	Trinity	TriforceAFL	In-Kernel AFL	syzkaller	kAFL	Userland Port	Unicorefuzz
Fuzz Anywhere	-	-	-	-	-	+	+
Peripherals	++	+	-	+	+	-	-
Binary-Only	-	+	-	-	+	-	+
Multi Arch	+	+	?	+	-	+	+
Speed	++	-	-	+	+	++	-
Instrumentation	-	AFL-QEMU	KCOV	KCOV	Intel PT	Any	AFL-Unicorn

Caveats

- State-dependent bugs won’t be found
- Code paths need to be triggered, first
- No interrupts/timers, no race conditions, …
- Speed could be better
- Unicorn…
- *Lots* of manual analysis

Unicorn…

POP QUIZ: Where is gs_base stored in memory?

Unicorn…

On x86_64 gs_base an actual register. Same for fs_base.

- Unicorn cannot write base registers (gs, fs) directly!

Workaround: map scratch address, emulate wrmsr

- cmpxchg16b instruction on Unicorn somewhat broken
- Probably more.

ARM insns have issues, too…

Hence no Unicorefuzz for ARM yet. :(

Nice things

- Fuzz allthethings
  → All GDB/Avatar2 targets should™ work
- Support for loadable kernel objects
- Debugger for test cases
- No ugly interrupts - Unicorn doesn’t have any
- Hooks can be set if fuzzer gets stuck
- Can fuzz deeply hidden functions

Future Work

- Embedded fuzzing
  - Fix ARM target
  - Add MIPS target
- Emulation performance: block chaining(?)
- Automate seed collection on BP hits
- Automate Triaging
- Unified (Proper) Evaluation Criteria for Kernel Fuzzers

Conclusion

- Coverage guided fuzzing is all the rage
- Fuzz anything you can attach GDB at
- No bug in Open vSwitch (yet)
- DoS in CIFS ASN1 parser
- Speed could be better or worse
- Finds bugs
- Open-sourced Unicorefuzz
  https://github.com/fgsect/unicorefuzz

Coverage guided fuzzing finds bugs early

Let’s find some kernel space overflows

```
while (questions());

char buf[16];
strncpy(buf, ""
    "Thank you for your attention."
    "\n", sizeof(buf));
printf("%s", buf);
```