A Not-So Fuzzy Art

"Better to have fuzzed and lost than to have never fuzzed at all"

Goal

- Make the *fuzzy* idea of fuzzing more concrete
- Teach you about some fuzzing tools / techniques
- Reassure you that fuzzing is a useful art

Where do we start?

```
$ while :; do
timeout 5 ./program < /dev/urandom;
done
```

Where do we start?

```
$ afl-fuzz -i inputs  \
           -o outputs \
           -- ./program
```

Where do we end?

- A hypervisor with COW memory to save and restore the entire OS state (and device drivers) to roll back time for each test case
- Per-block VM-exits as coverage indicators

Not so black-and-white

- Ok well, let's take a step back
- Do you have source?
  - No? Ok not the end of the world, we do this everyday
  - Yes? Lovely, that'll make this easier (hopefully)

Binary-only Fuzzing

"Live or die by your coverage" or

"Can I even run this?"

Binary-only Fuzzing

- Relevant especially to CTF
- AFL havoc mode or radamsa just won’t cut it
  - Don’t have benefit of afl-gcc
  - At the *very* least, need a strong input corpus

Binary-only Fuzzing

- You need some kind of instrumentation!
  - Block/branch/path coverage metrics
  - memcmp(), strcmp()
  - Symbolic execution
- Emulation or static binary rewriting

Binary-only Fuzzing

- Emulation and dynamic binary instrumentation are tractable but slow
- Static binary instrumentation is difficult but fast
  - Most solutions require private pdb's (Vulcan, Syzygy)

Binary-only Fuzzing

- Emulation and dynamic binary instrumentation are tractable but slow
- Static binary instrumentation is difficult but fast
  - Most solutions require private pdb's (Vulcan, Syzygy)
- Hardware acceleration? - Intel PT

Fuzzing “BC”

- Emphasis on black box fuzzing
- Extremely smart mutation engines, so we never “waste” a mutation

Age of Enlightenment

- History of modern fuzzing begins with AFL
- Key concepts:
    - Compiler instrumentation-guided
    - Coverage informs input generation

Case Study: Radamsa

piazzt@DESKTOP-IERJHAA: ~/radamsa

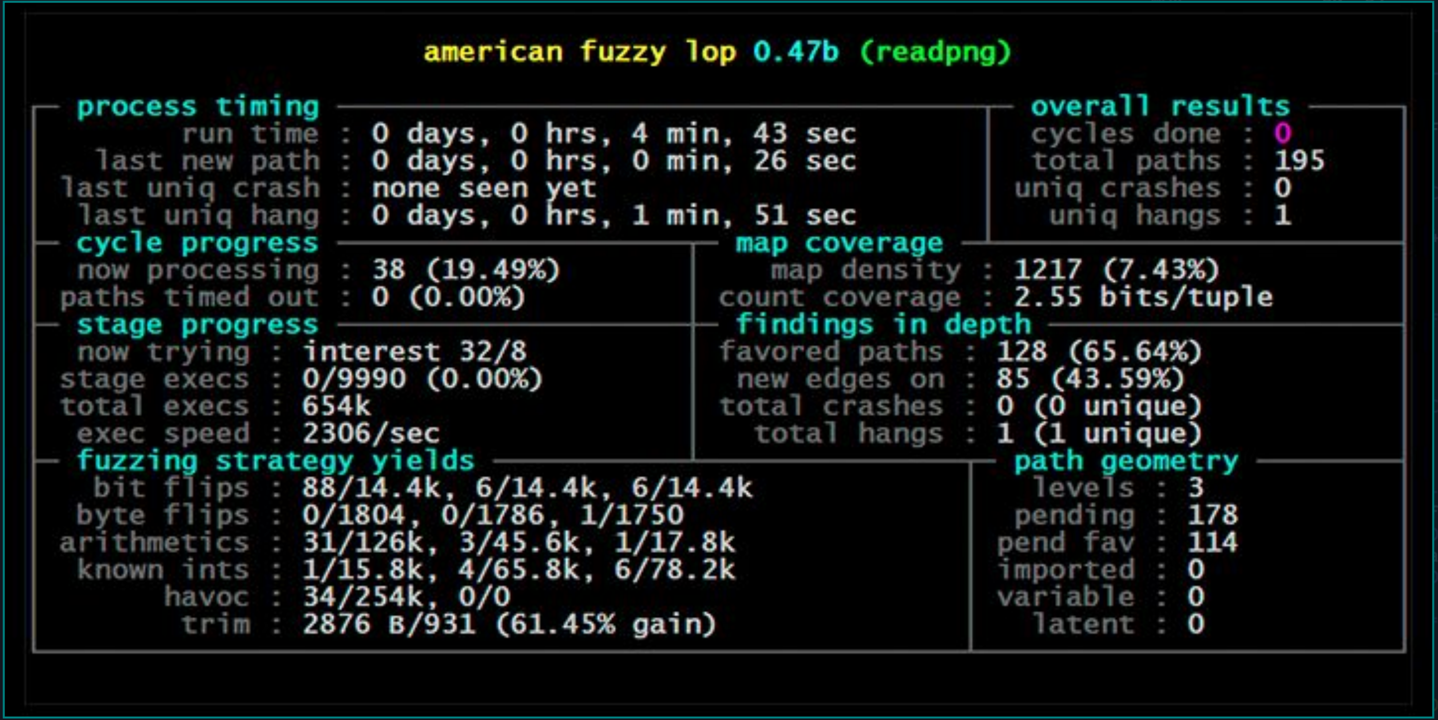
```
piazzt@DESKTOP-IERJHAA:~/radamsa$ wget -q https://www.w3schools.com/XML/simple.xml -O- | radamsa
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
  <name><food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
    <calories>650</calories>
  </food></name>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>Light Belgian waffles covered with strawberries and whipped cream</description>
    <calories>900</calories>
  </food>
```

Case Study: Radamsa

- More sophisticated mutations, XML-like inputs aware
- No harness
- No feedback

```
$ gzip -c /bin/bash > sample.gz
$ while true; do radamsa sample.gz | gzip -d > /dev/null; done
```

Case Study: AFL



Case Study: AFL

- Huge emphasis on file format fuzzing
- Heavy binary-based mutations
  - Sequential bit/byte flips, random stepover
  - Incs/decs of varying bitwidths
  - Boundary values

Case Study: AFL

- Probably unhelpful for *most* menu-based CTF challenges...

Case Study: AFL

- "Coverage-guided Fuzzing"
    - Associate each input with its corresponding coverage bitmap
    - Save inputs that have unique coverage to corpus
    - Mutator draws from corpus

Case Study: AFL

- Employs compiler or Qemu instrumentation to compute “path coverage”
- Saves coverage as 4k bitmap

```
cur_location = (block_address >> 4) ^ (block_address << 8);
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Case Study: AFL

- In general: coverage can be anything from edge or basic blocker counters, to wrapping compare / memcmp instructions

Case Study: AFL

- Coverage informs the mutator about inputs interesting to this target
  - Typically Block/Path coverage
- We probably haven't scratched the surface of instrumentation

Source Instrumentation

- Other kinds of instrumentation to aid fuzzing:
  - *ASAN*
  - *UBSAN*
  - *TSAN*
- Source pretty much required for these

Binary Instrumentation

- AFL-Qemu - As accurate as qemu, as slow as qemu
- AFL-Pin/honggfuzz-pin - As fast as your CPU, as slow as your disassembler
- WinAFL - DynamoRIO, Windows only
- These are just off the shelf tools

Binary Instrumentation

- mesos-style[1] - A mix of both; Temporary breakpoints that are removed after each hit.
  - Block coverage only (currently)
- Snapshot fuzzing - Emulated

More non-source fuzzing

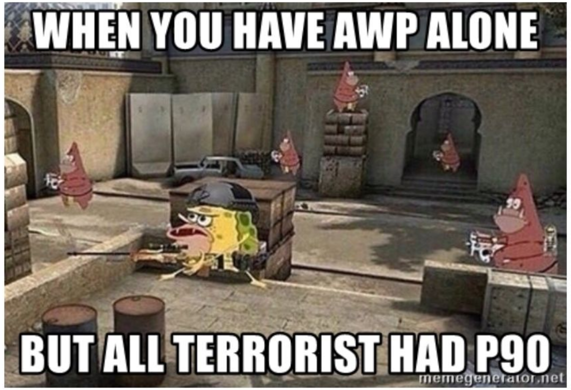
- If a bug doesn't crash does it make a sound?
- Poor man's alternative to clang sanitizers:
  - Page Heap
  - Application Verifier
  - LibDislocator
  - DrMemory
- Binpatching…? :'(

A Case Study: RealWorldCTF

Fuzzing Counter-Strike: Global Offensive maps files with AFL

Aug 26, 2018 • By niklasb

RealWorldCTF 2018 had a really fun challenge called “P90 Rush B”, an allusion to a desparate tactic employed in the Valve game “Counter-Strike: Global Offensive”. It was about finding and exploiting a bug in the map file loader used by a CS:GO server. During the CTF, I exploited a stack buffer overflow that was later described well in a writeup by another team.



A Case Study: RealWorldCTF

- CSGO Map File Format Fuzzing (niklasb) - *https://phoenhex.re/2018-08-26/csgo-fuzzing-bsp*
- Literally dlopen'd csgo engine and repeatedly called the CModelLoader::GetModelForName and had AFL+Qemu do the rest

A Case Study: RealWorldCTF

- Not always that easy, sometimes the input is not just a binary blob
  - Prefer not to waste mutations...
  - Write a generator or stamp in magic values by hand after some REing

Modern Techniques

The Future is *Now*

Emulation

- Environment can be completely specified
  - In-vitro
  - *You* specify the environment (hooks)
- Can instrument programs via hooks

```
cur_location = (block_address >> 4) ^ (block_address << 8);
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Emulation

- Qemu linux i386 user:
  - Library dependencies (need correct ld.so)
  - Unsupported system calls (sys_memfd_create, etc)
- Emulating from the first instruction is a pain
  - Hold this thought...

Emulation

Snapshot Fuzzing

- Start emulation from a “snapshot” in time
  - Completely repeatable continued executions
- Perfect state restoration
  - *cough* AFL forking-server mode
  - *cough* WinAFL
- No source required

Snapshot Fuzzing

- "Human in the loop"
  a. Fuzz for a while
  b. Analyze coverage
     i. Take new snapshot
     ii. Or update mutator/generator
  c. Repeat

Snapshot Restore

- Optimized snapshot restore ensures we spend more time fuzzing and less time loading all program state into the emulator

Snapshot Restore

1. Dirty page tracking - only restore pages that have been written to
2. Abuse COW pages using OS facilities
   a. AFL-forkserver mode
   b. Can easily do this with file-backed pages and unicorn

Snapshotting (usermode)

- Userland snapshots don't have a kernel (core dump, minidump)
  - Sockets
  - Files
  - Registry
- Implement a "mock OS"

Snapshotting (reduced kernel)

- Full kernel+usermode memory snapshot, but no device state
- Able to run a few syscalls accurately

lunixbochs commented on Jan 16, 2016 · Author · Contributor

@aquynh yes, I have a legitimate reason to run a reduced Linux kernel under Unicorn. I've already confirmed I seem to be able to to switch between user/kernel mode as well.

Snapshotting (full system)

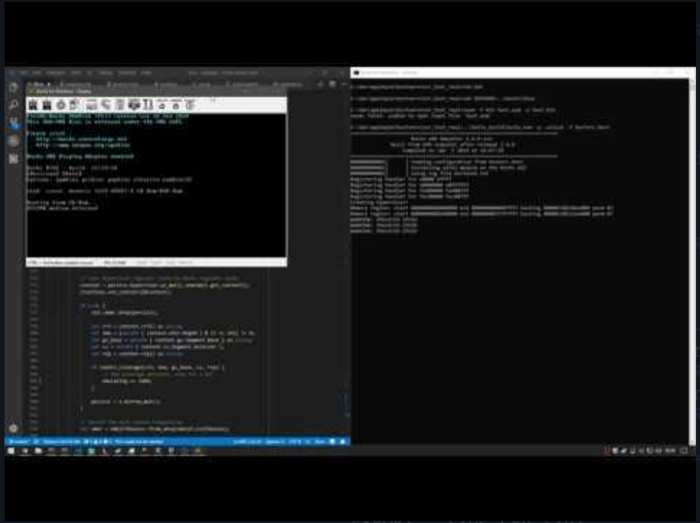
- Entire CPU and device state is captured
- a. la. VMWare/Bochs snapshots
  - Cache reads, throw away writes to disk
  - State of networking stack?

Snapshotting (full system)

- Device restoration is expensive and hard to do deterministically
- Emulating a full system (i.e. up to the snapshot point) takes forever

Snapshotting (full system)

- Make full-system snapshotting fast: throw a hypervisor into bochs?



Snapshot Fuzzing

- (Almost) no tools for this
- Can somewhat replicate with off-the-shelf tools
  - AFL and honggfuzz fork-server
  - LibFuzzer
  - Need to walk up to the desired state often

Snapshot Fuzzing

- Hopefully some are released soon ;-)

Justin Campbell
@metr0
Following

We've built tools for fuzzing based on emulation of a process snapshot captured via minidump. We're considering open sourcing the tool, and I'm curious about interest level from the rest of the world. (1/3)

9:29 AM - 14 Feb 2019

"These go to 11"

Hypervisor Snapshot Fuzzing

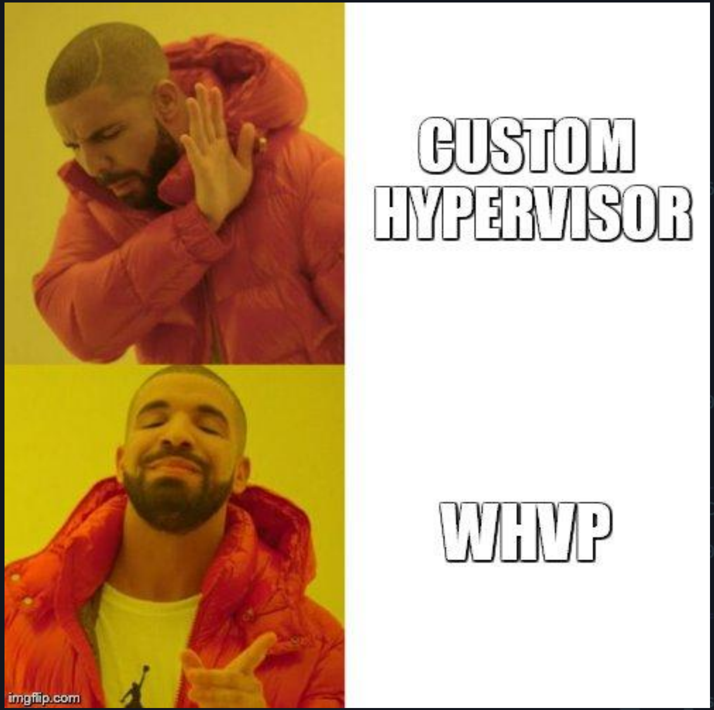
- Not for the faint of heart
  - Maximum performance
  - Extreme effort
- Fun application of snapshot fuzzing!

https://github.com/gamozolabs/falkervisor_grilled_cheese

Hypervisor Snapshot Fuzzing

- Load full memory dump into address space of VM
  - + mesos-style coverage using vmexits
  - + fast snapshot restore using COW memory

Hypervisor Snapshot Fuzzing



Hypervisor Snapshot Fuzzing

```rust
// Run the hypervisor until exit, returning the exit context
pub fn run(&mut self) -> WHV_RUN_VP_EXIT_CONTEXT {
    let mut context: WHV_RUN_VP_EXIT_CONTEXT =
        unsafe { std::mem::zeroed() };
    let res = unsafe { WHvRunVirtualProcessor(self.partition, 0,
        &mut context as *mut WHV_RUN_VP_EXIT_CONTEXT as *mut c_void,
        std::mem::size_of_val(&context) as u32) };
    assert!(res == 0, "WHvRunVirtualProcessor() error: {:#x}", res);

    // Mark that memory may be dirty
    self.memory_dirty = true;

    context
}
```

Source-Based Fuzzing

For Real People Doing Real Things and Not CTFs

Source-Based Fuzzing

- The compiler is your friend, let it help you
- So step 1: Instrumentation
  - GCC has gcov
  - LLVM+Clang have SanitizerCoverage
  - AFL has a similar compiler pass
  - Build your own birdfeeder

Source-Based Fuzzing

- Step 2: Harness the code you want to target
  - Either embed a harness in the codebase, and behave like AFL's forkserver
  - Simple python harness will do to delivery fuzz test case to program

Source-Based Fuzzing

- Step 3: Work on your mutator / fuzz test case generator
  - Do this as appropriate when coverage shows deficiencies
  - Will find *much* deeper bugs

Source-Based Fuzzing

```
try { someTypedArray1.reduce(function(acc, cval, c_index, c_array) { try{ c_array[c_index] = c_array.reduce((acc, cval, c_index, c_array) => { try{
y.lastIndexOf(new Object(), -32760);c_array[c_index] = c_array.filter((arg) => { return (arg == new Object()) }); } catch(e){ } });c_array.reverse()
try { someRegex1[Symbol.search]("WUtazcQunqxEnKAbPkeIfNoQnSpOwQULMUoDVf") } catch (e) { }
try { someSet1.entries() } catch (e) { }
try { someWeakSet1.delete(function() {}) } catch (e) { }
try { Object.getOwnPropertyNames(someWeakSet1) } catch (e) { }
try { someString1.hasOwnProperty("toString") } catch (e) { }
try { for (var element in someObject1) { try{ someObject1[element] = someObject1[element].concat(); } catch(e) { } } } catch (e) { }
try { someTypedArray1 = new Uint16Array(someArrayBuffer1, 114) } catch (e) { }
try { someIntlNumberFormat1.formatToParts(+17064) } catch (e) { }
try { Math.sinh(11582) } catch (e) { }
try { someWeakSet1.add(someTypedArray1) } catch (e) { }
try { someDataView1.getFloat64(250, false) } catch (e) { }
try { Intl.NumberFormat.supportedLocalesOf("fi-FI") } catch (e) { }
try { someArray1[0] = someRegex1.test(String.fromCodePoint(669014) + "prAmHEKKXgdQBgytdTnyQnd") } catch (e) { }
try { someWeakSet1.delete(someObject1) } catch (e) { }
try { Intl.DateTimeFormat.supportedLocalesOf("ar-LB-u-hc-h11-nu-beng") } catch (e) { }
try { Intl.NumberFormat.supportedLocalesOf("es-PA-u-nu-kali") } catch (e) { }
try { for(var index=0; index < 7; index++){ someArray1[index] = someArray1.entries(); } } catch (e) { }
try { for(var index=0; index < 8; index++){ someArray1[index] = someArray1.join("iRq"); } } catch (e) { }
```

Effective Mutation

- Don't need neural networks or genetic mutations really
  - Just take an input that you know has caused coverage, and add it to your corpus
  - If a mutation on that test case causes new coverage, add it to corpus

LibFuzzer

- LLVM Fuzzer
- Can combine the LLVM Sanitizers for an excellent in-memory coverage based fuzzer
- No processes spawning, no recv() calls, less transitions away from the code you want to target

LibFuzzer

- Implements techniques already described
  - SanitizerCoverage module increments into a per-module bitmap (edge coverage)
  - Table of recent comparisons to record literal cmp’s
- Ultimately very fast, and very precise

LibFuzzer Example

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  uint8_t Uncompressed[100];
  size_t UncompressedLen = sizeof(Uncompressed);
  if (Z_OK != uncompress(Uncompressed, &UncompressedLen, Data, Size))
    return 0;
  if (UncompressedLen < 2) return 0;
  if (Uncompressed[0] == 'F' && Uncompressed[1] == 'U')
    abort();  // Boom
  return 0;
}
```

LibFuzzer Example

```
extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
                                          size_t MaxSize, unsigned int Seed) {
  uint8_t Uncompressed[100];
  size_t UncompressedLen = sizeof(Uncompressed);
  size_t CompressedLen = MaxSize;
  if (Z_OK != uncompress(Uncompressed, &UncompressedLen, Data, Size)) {
    // The data didn't uncompress. Return a dummy...
  }
  UncompressedLen =
      LLVMFuzzerMutate(Uncompressed, UncompressedLen, sizeof(Uncompressed));
  if (Z_OK != compress(Data, &CompressedLen, Uncompressed, UncompressedLen))
    return 0;
  return CompressedLen;
}
```

If all you have is a hammer…

- LibFuzzer has mostly a binary-file format esque mutator (understandable)
- To get around this, generate a grammar that maps raw bytes -> your input structure and mutate that
- LibProtobuf-Mutator makes this faster

Case Study: Fuzzing Clang

- [Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator]()

Case Study: Fuzzing Clang

- Protobuf syntax:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

Case Study: Fuzzing Clang

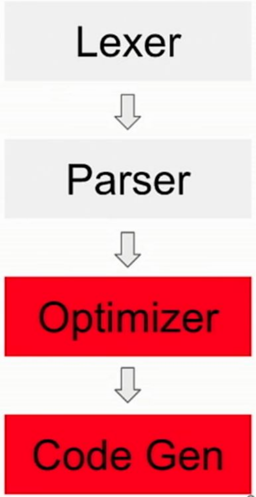
1. Define proto that “looks like” C/C++ AST
2. Map arbitrary byte array -> libprotobuf
3. Mutate protobuffer object with LPM
4. Turn protobuf into C-like syntax
5. Run test case

Case Study: Fuzzing Clang

clang-proto-fuzzer trophies

null deref in llvm::ScalarEvolution::getMulExpr

```
void foo(int *a) {
  while (1) {
    a[60] = ((1 + a[60]) + a[0]);
    while ((a[60] + a[0])) {
      a[0] = (a[0] + 1);
    }
  }
}
```



30

Case Study: Fuzzing Clang

- Related: [Attacking Chrome IPC: Reliably finding bugs to escape the Chrome sandbox]

Sanitizers

- ASAN, UBSAN, MSAN, TSAN, COVSAN
- Runtime checking for memory corruption, undefined behavior, race conditions, uninitialized memory, etc
- You want these.

Address Sanitizer

- Catches buffer-out-of-bounds accesses (reads and writes)
- Usually the go-to as the bare minimum for the bugs you probably want anyways

Ripping Out Code

- May be useful to just rip out the function
  - Definitely perf to gain
- Don’t jump through hurdles to get to vulnerable functions
- Make sure to keep the same assumptions as it would have in the original program

Documenting the source(??)

- Add assumptions that aren't enforced as runtime asserts
- Log state to check for invalid transitions
- Harden the source as you fuzz it

Big Questions (1)

- How do I improve fuzzing performance?
  - Remove syscalls? Seriously, all of them
  - Skip code that you don’t care about… (easier said than done of course)

Big Questions (2)

- At what point does my mutator output this?
  - Useful for scenarios like JS engines
  - Can design a fuzzer after-the-fact to find this bug, and hope it finds new bugs. Chicken-And-Egg problem

Fuzzing as Exploit Assistance

- Can’t find a good primitive? Search space is relatively small?
- Let fuzzer run with crashing input and analyze the different crashes
- Cross fingers and pray for rip = 0x41414141

Fuzzing as Exploit Assistance

- Chainsawing is smarter:
  - Write a harness to trigger different allocations
  - Groom LFH, different bucket sizes, different objects, etc
  - Trigger bug at end
  - Cross fingers and pray for deref *0x41414141

Fuzzing Take-Away

- Make every mutation count
- Make every iteration fast, only run the code you want to target, don’t waste cpu cycles
- But before you hyper-optimize, get your dumb fuzzer up and running.
  - It doesn’t cost you anything!