GUSTAVE: Fuzz It Like It's App

(feat. QEMU & AFL)

Stéphane Duverger, Anaïs Gantet

THC - March 8, 2019

AIRBUS

Outline

Outline

What we'll talk about

Some basics about

- Fuzzing
- OS system calls
- AFL/QEMU

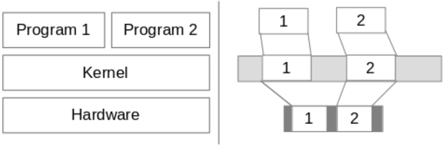
The challenges of fuzzing kernels as simple user programs

- Input translation
- Target instrumentation
- Target behavior monitoring
- *Crash* detection and classification

Target

What?

- Embedded OS in charge of space partitioning
  - kernel/user isolation
  - memory segregation
  - process partitioning through *address spaces*
  - etc.



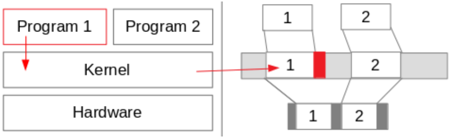
Security considerations

- Problem: Serious security consequences on segregation bypass
- Question: Is this space partitioning correctly implemented? not breakable?

Attack playground

Context

- attack vector: from an unprivileged program
- attack surface: kernel services via system calls
- aim: try to bypass the memory segregation



How?

- Build "malicious" user programs performing system calls
- Craft weird system call arguments
  - to trigger security vulnerabilities
  - to try to run/cover the maximum of OS existing code

Toward full automation

Expected workflow

1. Prepare a platform and its OS environment
2. Save full system state
3. Inject the code of a "malicious" user program
4. Run the attack
5. Analyze the impact
6. Restore full system state
7. Goto 3

Outline

AIRBUS

Vulnerability discovery methods

Static analysis

- Manual code review (white box)
- Reverse code engineering (black box)
- Automation (formal methods, model checking)

Runtime analysis

- Concrete/symbolic execution (concolic testing)
- Program tracing/instrumentation
- Fuzzing (chosen one)

Fuzzing methods

Did you say random ?

- Basic fuzzing: the children and keyboard paradigm
- Catalog-guided/model-based: classification, prior knowledge of API
- Coverage-guided: maximize target code coverage

Fuzzing methods

Did you say random ?

- Basic fuzzing: the children and keyboard paradigm
- Catalog-guided/model-based: classification, prior knowledge of API
- Coverage-guided: maximize target code coverage

Mix coverage-guided/behavior monitoring

- No previous knowledge of target
- Try to cover as much as possible from entries (system calls)
- Classify fuzzed input from target behavior upon execution
- Adapt/evolve *faulting* inputs to trigger more crashes

Fuzzing methods

Did you say random ?

- Basic fuzzing: the children and keyboard paradigm
- Catalog-guided/model-based: classification, prior knowledge of API
- Coverage-guided: maximize target code coverage

Mix coverage-guided/behavior monitoring

- No previous knowledge of target
- Try to cover as much as possible from entries (system calls)
- Classify fuzzed input from target behavior upon execution
- Adapt/evolve *faulting* inputs to trigger more crashes

Solid candidate

AFL: American Fuzzy Lop, *Google Inc.*

AFL in a nutshell

One of the best fuzzer out there

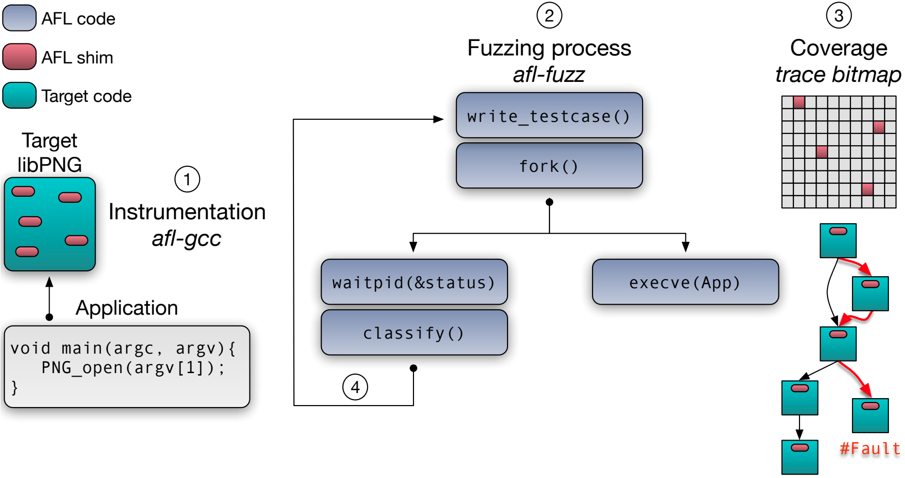
- Free & open-source software: `http://lcamtuf.coredump.cx/afl/`
- A lot of discovered vulnerabilities (mainly applications, libs)
- Advanced fuzzing technology based on *evolutionary algorithms*

AFL workflow

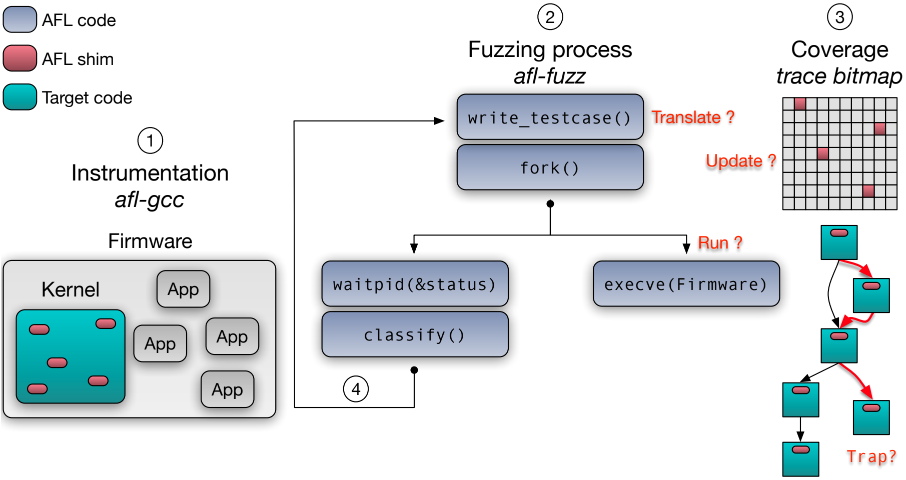
- Phase 1: instrumentation
  - Rebuild target with instrumentation[a]
  - Inject shims at every target basic block
  - The shims will update an execution coverage trace bitmap (shim)
- Phase 2: fuzzing
  - Generate inputs to maximize target code coverage
  - Spawn target process and monitor its execution
  - Classify inputs based on exit status and trace bitmap

[a] need source code, binary mode possible

AFL against libPNG



AFL against OS kernel?



State-of-the-art tools

Objectives

- Try to reuse available softwares as building blocks
- Choose the most flexible/versatile technologies
- *evicted* syzkaller/MWRlabs

Interesting candidates to fuzz kernels with AFL?

- kAFL, Intel centric, OS agnostic
- Triforce-AFL, arch/OS agnostic (almost)
- Unicorn-AFL, CPU only

State-of-the-art tools

Objectives

- Try to reuse available softwares as building blocks
- Choose the most flexible/versatile technologies
- *evicted* syzkaller/MWRlabs

Interesting candidates to fuzz kernels with AFL?

- kAFL, Intel centric, OS agnostic
- Triforce-AFL, arch/OS agnostic (almost)
- Unicorn-AFL, CPU only

Conclusion: nobody's perfect

- Inappropriate design choices
- . . . ok build our own :)

Assemble and extend existing building blocks

Selected technologies

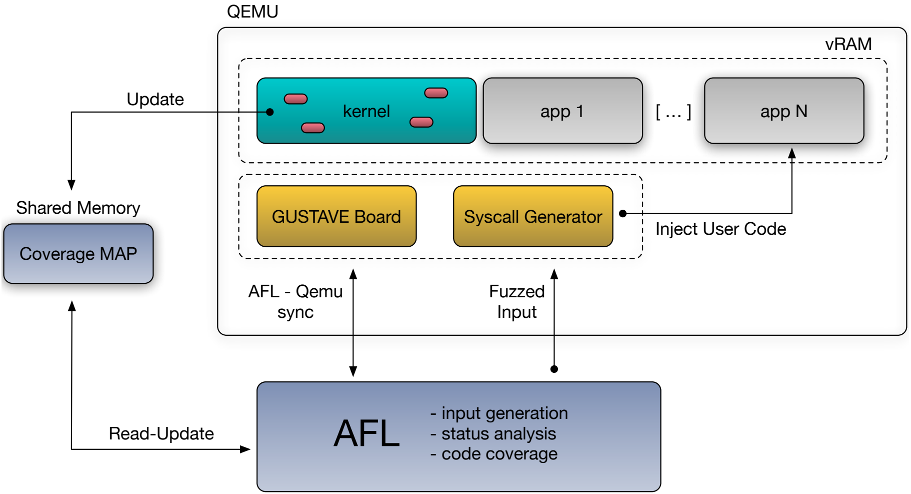
- Fuzzing with AFL
- Simulation environment with QEMU

Extend the best tools

- No heavy modifications (internals) allowed !
- Build glue to make AFL/QEMU interact seamlessly

Outline

GUSTAVE architecture



GUSTAVE answer to challenges

How to run?

- Implement an AFL-QEMU board
- Synchronize with AFL

GUSTAVE answer to challenges

How to run?

- Implement an AFL-QEMU board
- Synchronize with AFL

How to translate?

- Requires to define an *input logic*
- Idea is to translate them either as:
  - Sequences of system calls (ID and arguments)
  - Fixed system call ID with fuzzed arguments

GUSTAVE answer to challenges (2)

How to trap?

- Timeout and normal exits are easy to trap
- Faulty behaviors are tricky
- We are trying to crash an OS
- Should we monitor the CPU itself?

No *SegFault* for OS

- This is an application paradigm
- Need to hook on *controlled failures*: `panic`, `reboot`, etc.
- Requires to define partitioning bypass oracles:
  - memory region boundary checks
  - internal CPU state/fault interception

QEMU board details

How to update? *(trace bitmap)*

- Target kernel will hit bitmap through arbitrary `mm i/o`
- Map host bitmap SHM physical pages to VM `mm i/o` area
- Zero overhead (like it's app)

QEMU board details

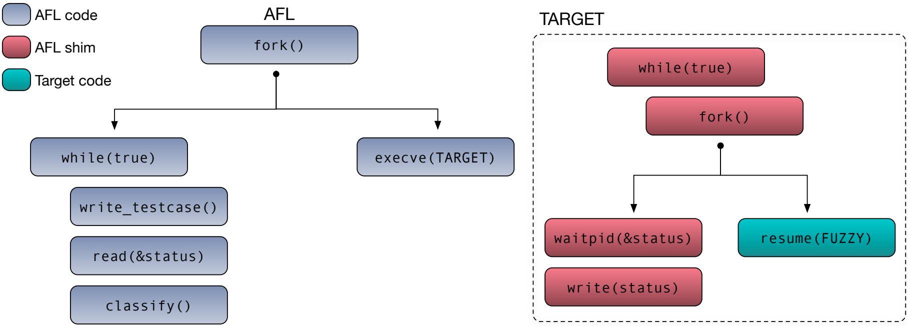
How to update? *(trace bitmap)*

- Target kernel will hit bitmap through arbitrary `mm i/o`
- Map host bitmap SHM physical pages to VM `mm i/o` area
- Zero overhead (like it's app)

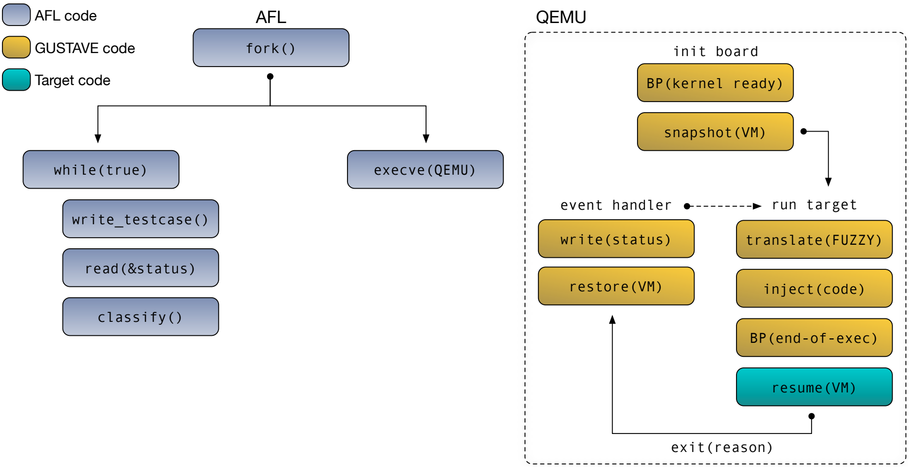
Core features/optimizations

- Snapshot API to save/restore VM state
- Internal breakpoints subversion (no gdb :)
- Fix CPU state (paging), intercept exceptions
- No TCG modification (can use KVM)

AFL fork-server mode



QEMU board fork-server



Outline

What is POK?

"POK, a real-time kernel for secure embedded systems"

- A small OS, open-source
- Implements memory partitioning
- 90% formally verified (according to the website[a])

[a] https://pok-kernel.github.io/

What is POK?

"POK, a real-time kernel for secure embedded systems"

- A small OS, open-source
- Implements memory partitioning
- 90% formally verified (according to the website[a])

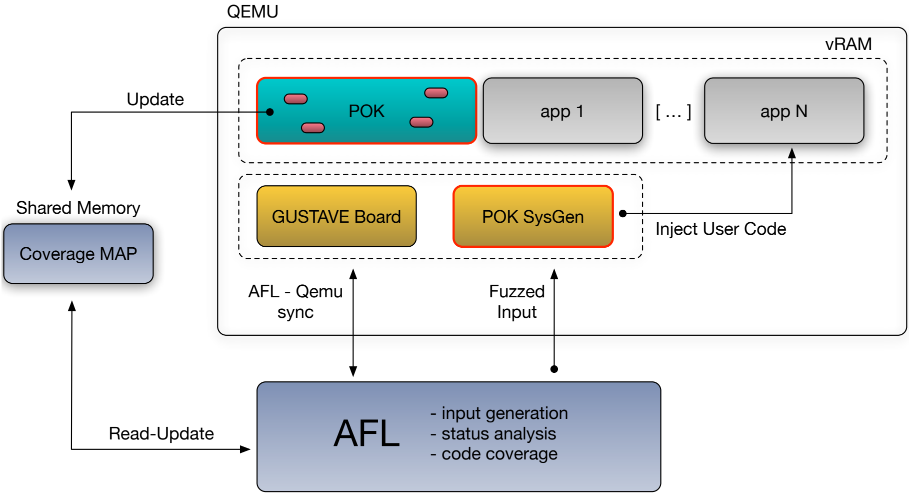
[a] https://pok-kernel.github.io/

You said *"secure"*?

- Still contains vulnerabilities we discovered by reading the OS code manually
- The best target to validate the first prototype of our proposed tool
- Aim: rediscover the known vulnerabilities with AFL

GUSTAVE and POK: architecture

(POK partially recompiled with AFL-GCC)

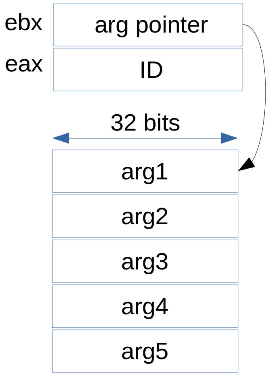


AIRBUS

GUSTAVE and POK: attack surface

POK syscall API

- About 50 kernel functions
  - Thread management
  - Partition information
  - Port send/receive
  - etc.
- Callable from the user program with
  - The corresponding syscall ID
  - 1 to 5 arguments as input
- Various argument types
  - Pointer to structures
  - Integer
  - String
  - etc.

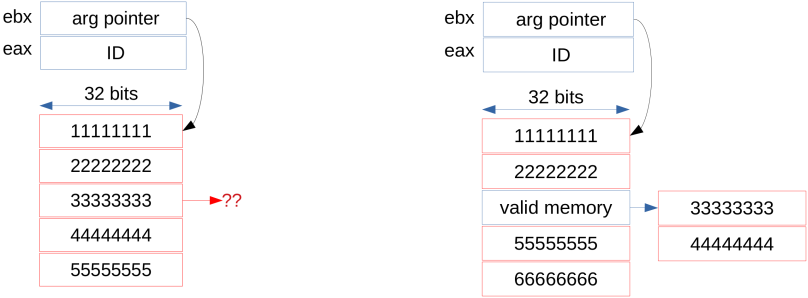


AIRBUS

GUSTAVE and POK: fuzzing strategies

2 different versions for POK SysGen

- Totally random inputs (including pointer values)
- Controlled pointers and random pointed content



GUSTAVE and POK: memory vulnerability detection

POK memory management

- Based on Intel x86 segmentation
- 1 code/data segment for each user program
- 1 code/data segment for the kernel (FLAT!!)

GUSTAVE memory oracles

- Relies on Intel x86 paging (not used by POK)
- Mimics POK memory layout (kernel / user programs)
- Unmaps the rest of the memory
- Traps Page Faults in QEMU board
- Notifies AFL when Page Faults occur

GUSTAVE against POK

```
                 american fuzzy lop 2.52b (qemu-system-i386)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 9      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec        │ uniq crashes : 5      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 0.03% / 0.20%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 1 (11.11%)             │
│ stage execs : 372/4096 (9.08%)      │  new edges on : 7 (77.78%)             │
│ total execs : 402                   │ total crashes : 14 (5 unique)          │
│  exec speed : 346.4/sec             │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                         │    levels : 2          │
│  byte flips : n/a, n/a, n/a                         │   pending : 9          │
│ arithmetics : n/a, n/a, n/a                         │  pend fav : 1          │
│  known ints : n/a, n/a, n/a                         │ own finds : 8          │
│  dictionary : n/a, n/a, n/a                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │ stability : 100.00%    │
│        trim : 11.11%/2, n/a                         ├────────────────────────┘
^C──────────────────────────────────────────────────┘          [cpu:168%]
```


AIRBUS

GUSTAVE and POK: results

It works! :)

- First valid proof of concept against a real OS
- Expected vulnerabilities detected by GUSTAVE

Performances

- Reach $\sim$ 350 tests/second on a single core/thread
- Several optimizations
  - Single-threaded execution
  - Optimize scheduling (time frames)

Crash analysis

- 25 new *write-everywhere* vulnerabilities found in a couple of seconds
- more time needed to analyze the further crash cases

Outline

Takeaways

GUSTAVE usage

1. Preliminarily, reverse some kernel parts
   - System call operation (ABI)
   - Memory segregation strategy
2. Implement the syscall generator specific to the target
3. Define and add vulnerability detection strategies
4. Run GUSTAVE
5. Analyze the detected vulnerabilities, report, exploit, enjoy :)

Conclusion and future outlook

GUSTAVE and state-of-the-art advances

- Capable to fuzz all syscalls (not *mount* only)
- Uses AFL and QEMU without internals modification
- Finds vulnerabilities not caught by the OS itself
- Run with acceptable performances (hardware-virtualization when supported)

Next steps?

- Open-source the tool
- Play with other kernel targets
- Make the tool more user-friendly (target specificities via config file)

Thanks for your attention. Any questions ?

stephane.duverger@airbus.com
anais.gantet@airbus.com

@AirbusSecLab
(https://airbus-seclab.github.io)

Appliquez-vous à développer un progrès aussi minime soit-il. Vous en ferez un bien général.

Gustave Eiffel

Fuzz it like it's app, fuzz it like it's app.

Gustave AFL