An Introduction to Fuzzing and a Direct Application to the Real World

Leonardo Galli

flagbot (ctf@vis.ethz.ch)

October 15, 2021

What does this ... have to do with this?

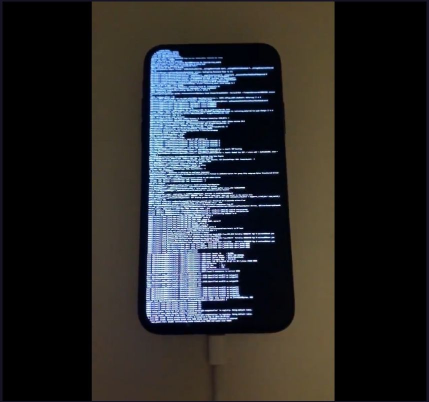


Table of Contents

About Me

- Finished my Bachelor of Computer Science at ETH
- Member of flagbot since over three years
- President of flagbot since over two years
- Lead `organizer` since half a year

Leonardo Galli
leonardo.galli@vis.ethz.ch

About flagbot

- VIS committee and ETH's Capture the Flag team
  - CTFs are team-based cybersecurity competitions, often involving real-world attacks
- Ranked 1^{st} place in Switzerland in 2019 and 2020[1]
- Playing CTFs on weekends
- Weekly meetings on Monday at 19:00 over Zoom and in person at CAB H52, open to anyone
  - Discussion of challenges and lectures aimed at beginners (recordings available on `flagbot.ch/material`)

Contact: ctf@vis.ethz.ch

More Information: `flagbot.ch`

[1] According to `ctftime.org`

About organizers

- Joint team between flagbot, polygl0ts (EPFL), cr0wn (UK) and secret.club
- Team up together for larger events
- Currently ranked 7th worldwide[2]
- Multiple big wins, such as best European team at DEF CON and #1 at Tencent CTF 2021

```
     ,aooooa,
   ,oY"'  '"Yo,
  ,oY00      Yo,
  oo  ORG     oo
  oo   ANI    oo
  oo   {S|Z}  oo
  oo     ERS  oo
  `ob      00do'
   `oba,  ,ado'
     "YooooY"
```

Contact: org@anize.rs
Website: `org.anize.rs`

[2] According to `ctftime.org`

Introduction

Motivation

- Imagine you are company REDACTED

Motivation

- Imagine you are company REDACTED
- Many security flaws are discovered
  - "REDACTED Issues Emergency Security Updates to Close a Spyware Flaw" [7]
  - "REDACTED zero-day let SolarWinds hackers compromise fully updated REDACTED" [3]
  - "New REDACTED 'Zero Day' Hack Has Existed For Months" [4]

Motivation

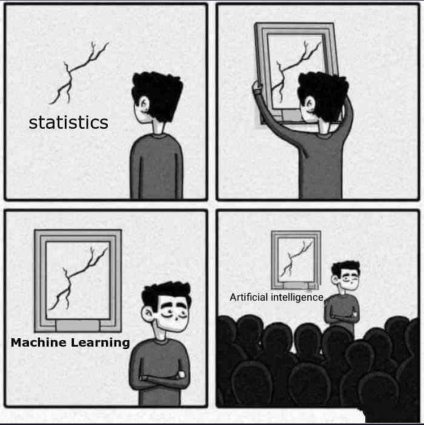
- Imagine you are company REDACTED
- Many security flaws are discovered
  - "REDACTED Issues Emergency Security Updates to Close a Spyware Flaw" [7]
  - "REDACTED zero-day let SolarWinds hackers compromise fully updated REDACTED" [3]
  - "New REDACTED 'Zero Day' Hack Has Existed For Months" [4]
- **Problem:** Want to reduce the number of security issues
  - There are just too many

Motivation

- Imagine you are company REDACTED
- Many security flaws are discovered
  - "REDACTED Issues Emergency Security Updates to Close a Spyware Flaw" [7]
  - "REDACTED zero-day let SolarWinds hackers compromise fully updated REDACTED" [3]
  - "New REDACTED 'Zero Day' Hack Has Existed For Months" [4]
- **Problem:** Want to reduce the number of security issues
  - There are just too many
- **Problem:** Security experts are costly

Automatic Vulnerability Discovery

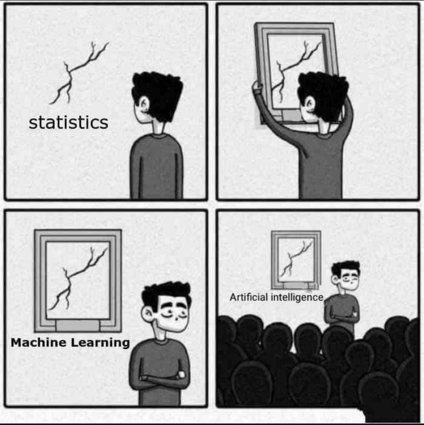
- **Solution:** Use modern automation to find vulnerabilities



Not this kind of automation

Automatic Vulnerability Discovery

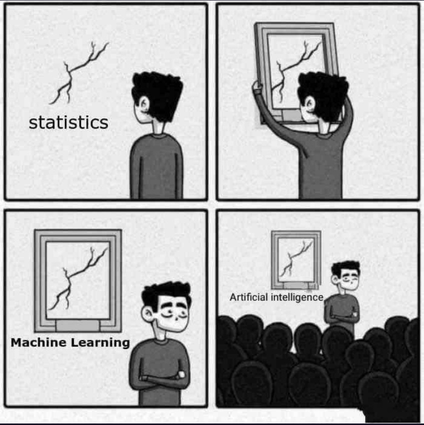
- **Solution:** Use modern automation to find vulnerabilities
- Introducing "fuzzing"
  - Automatically find vulnerabilities
  - Can be run unsupervised
  - Great track record



Not this kind of automation

Automatic Vulnerability Discovery

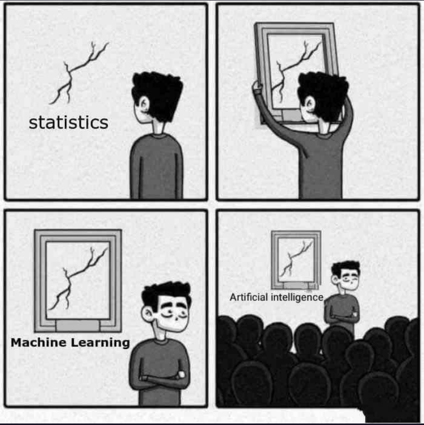
- **Solution:** Use modern automation to find vulnerabilities
- Introducing "fuzzing"
  - Automatically find vulnerabilities
  - Can be run unsupervised
  - Great track record
- Becoming more and more popular and useful



Not this kind of automation

Automatic Vulnerability Discovery

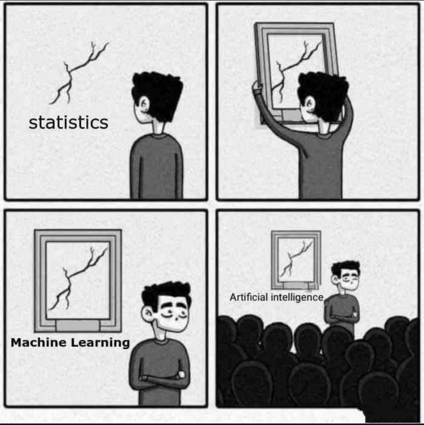
- **Solution:** Use modern automation to find vulnerabilities
- Introducing "fuzzing"
  - Automatically find vulnerabilities
  - Can be run unsupervised
  - Great track record
- Becoming more and more popular and useful
- Native fuzzing support in go 1.18



Not this kind of automation

Automatic Vulnerability Discovery

- **Solution:** Use modern automation to find vulnerabilities
- Introducing "fuzzing"
  - Automatically find vulnerabilities
  - Can be run unsupervised
  - Great track record
- Becoming more and more popular and useful
- Native fuzzing support in go 1.18
- OSS-Fuzz provides continuous fuzzing for OSS
  - "As of February 2021, 26,000+ bugs found in over 400 open source projects integrated with OSS-Fuzz." [2]



Not this kind of automation

Fuzzing

Basic Idea

Almost the right kind of fuzzy[3].

[3] Depending on the context.

Basic Idea

- Basic fuzzing loop:

Basic Idea

- Basic fuzzing loop:

1. Generate (random) input

Basic Idea

- Basic fuzzing loop:

1. Generate (random) input
2. Run application with generated input

Basic Idea

- Basic fuzzing loop:

1. Generate (random) input
2. Run application with generated input
3. Observe application behaviour

Basic Idea

- Basic fuzzing loop:

1. Generate (random) input
2. Run application with generated input
3. Observe application behaviour

$\Longrightarrow$ **Any observed crashes indicate presence of bugs**

Basic Idea

- Basic fuzzing loop:

1. Generate (random) input
2. Run application with generated input
3. Observe application behaviour

$\Longrightarrow$ **Any observed crashes indicate presence of bugs**

- Not necessarily any vulnerabilities yet, more on that later

Fuzzing
Types of Fuzzing

Overview

- Fuzzing encompasses broad spectrum of techniques
- Three important ways of categorizing fuzzers

1. How input is generated
   - Mutate existing input
   - Generate from scratch
   - Usually mutation based
2. Awareness of input structure
3. Awareness of application structure

Awareness of Input Structure

- Fuzzing input can be anything, not just text
- Inputs should have certain structures
  - Structure distinguishes valid from invalid input
  - Example of structure is a file format

Awareness of Input Structure

- Fuzzing input can be anything, not just text
- Inputs should have certain structures
  - Structure distinguishes valid from invalid input
  - Example of structure is a file format
- **Question:** Why do we care about valid input when fuzzing?

Awareness of Input Structure

- Fuzzing input can be anything, not just text
- Inputs should have certain structures
  - Structure distinguishes valid from invalid input
  - Example of structure is a file format
- **Question:** Why do we care about valid input when fuzzing?

$\Longrightarrow$ Fuzzer should tow fine line between valid and invalid input

Awareness of Input Structure

- Fuzzing input can be anything, not just text
- Inputs should have certain structures
  - Structure distinguishes valid from invalid input
  - Example of structure is a file format
- **Question:** Why do we care about valid input when fuzzing?

$\Longrightarrow$ Fuzzer should tow fine line between valid and invalid input

- A "smart" (model-based or grammar-based) fuzzer uses pre-existing knowledge about the input structure to generate "valid enough" inputs.
- A "dumb" fuzzer does no such thing

Awareness of Input Structure

- Fuzzing input can be anything, not just text
- Inputs should have certain structures
  - Structure distinguishes valid from invalid input
  - Example of structure is a file format
- **Question:** Why do we care about valid input when fuzzing?

$\Longrightarrow$ Fuzzer should tow fine line between valid and invalid input

- A "smart" (model-based or grammar-based) fuzzer uses pre-existing knowledge about the input structure to generate "valid enough" inputs.
- A "dumb" fuzzer does no such thing
- Modern fuzzers usually use a combination of both

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize
  - Quick setup for any program

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize
  - Quick setup for any program
- A “white-box” fuzzer is fully aware and uses program analysis to reach high coverage and critical points.

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize
  - Quick setup for any program
- A “white-box” fuzzer is fully aware and uses program analysis to reach high coverage and critical points.
  - For example, symbolic execution or taint analysis

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A “black-box” fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize
  - Quick setup for any program
- A “white-box” fuzzer is fully aware and uses program analysis to reach high coverage and critical points.
  - For example, symbolic execution or taint analysis
  - Heavyweight analysis, slow and difficult to scale

Awareness of Application Structure

- Fuzzing effective if high degree of coverage achieved
- A "black-box" fuzzer is unaware of internal program structure.
  - Blindly test inputs and hope for crashes
  - Very fast and easy to parallelize
  - Quick setup for any program
- A "white-box" fuzzer is fully aware and uses program analysis to reach high coverage and critical points.
  - For example, symbolic execution or taint analysis
  - Heavyweight analysis, slow and difficult to scale
  - Cannot be applied to every application without significant effort

A Middle Ground

- A "grey-box" fuzzer uses lightweight instrumentation to learn information about program structure.

A Middle Ground

- A "grey-box" fuzzer uses lightweight instrumentation to learn information about program structure.
  - Usually tracing basic block transitions or standard code coverage

A Middle Ground

- A “grey-box” fuzzer uses lightweight instrumentation to learn information about program structure.
  - Usually tracing basic block transitions or standard code coverage
  - Recent advancements include the use of sanitizers [6]

A Middle Ground

- A "grey-box" fuzzer uses lightweight instrumentation to learn information about program structure.
  - Usually tracing basic block transitions or standard code coverage
  - Recent advancements include the use of sanitizers [6]
- Almost as fast and scalable as black-box fuzzing

A Middle Ground

- A "grey-box" fuzzer uses lightweight instrumentation to learn information about program structure.
  - Usually tracing basic block transitions or standard code coverage
  - Recent advancements include the use of sanitizers [6]
- Almost as fast and scalable as black-box fuzzing
- Most popular approach by far

A Middle Ground

- A “grey-box” fuzzer uses lightweight instrumentation to learn information about program structure.
  - Usually tracing basic block transitions or standard code coverage
  - Recent advancements include the use of sanitizers [6]
- Almost as fast and scalable as black-box fuzzing
- Most popular approach by far
- Support for most program configurations

Fuzzing
Getting Started with Fuzzing

Choosing a Fuzzer

- Look for language support

Choosing a Fuzzer

- Look for language support
- Otherwise, start with AFL++

Choosing a Fuzzer

- Look for language support
- Otherwise, start with AFL++
  - Supports many configurations

Choosing a Fuzzer

- Look for language support
- Otherwise, start with AFL++
  - Supports many configurations
  - Continuously updated

Choosing a Fuzzer

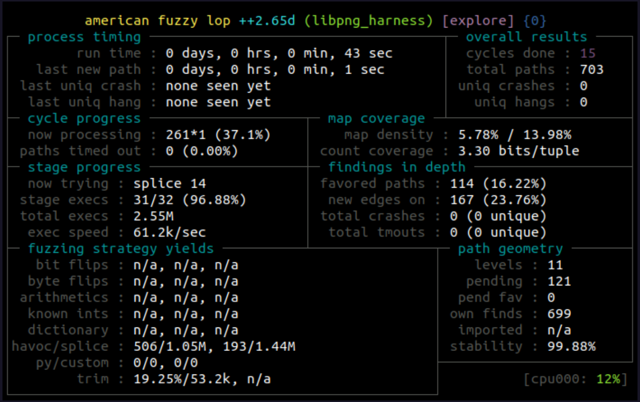
- Look for language support
- Otherwise, start with AFL++
  - Supports many configurations
  - Continuously updated
  - (simple) grammar and advanced instrumentation supported

Choosing a Fuzzer

- Look for language support
- Otherwise, start with AFL++
  - Supports many configurations
  - Continuously updated
  - (simple) grammar and advanced instrumentation supported
- **Here:** assume AFL++ used

AFL++

- Based on American Fuzzy Lop (AFL)



The famous AFL TUI

AFL++

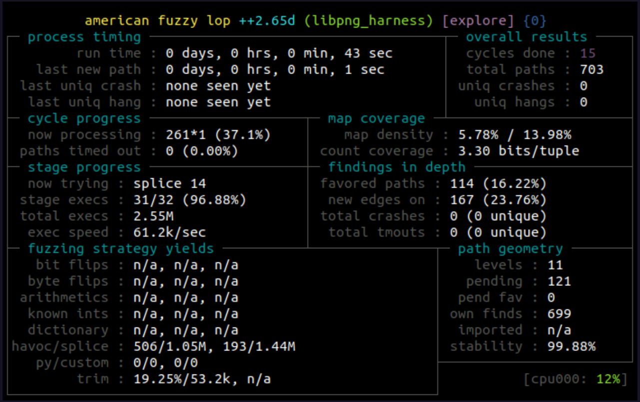
- Based on American Fuzzy Lop (AFL)
- Most well known coverage-guided grey-box fuzzer

```
         american fuzzy lop ++2.65d (libpng_harness) [explore] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 43 sec      │  cycles done : 15    │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec       │  total paths : 703   │
│ last uniq crash : none seen yet                     │ uniq crashes : 0     │
│  last uniq hang : none seen yet                     │   uniq hangs : 0     │
├─ cycle progress ───────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 261*1 (37.1%)    │    map density : 5.78% / 13.98%       │
│ paths timed out : 0 (0.00%)        │ count coverage : 3.30 bits/tuple      │
├─ stage progress ───────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 14            │ favored paths : 114 (16.22%)          │
│ stage execs : 31/32 (96.88%)       │  new edges on : 167 (23.76%)          │
│ total execs : 2.55M                │ total crashes : 0 (0 unique)          │
│  exec speed : 61.2k/sec            │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ──────────┴─────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 11         │
│  byte flips : n/a, n/a, n/a                      │   pending : 121        │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 0          │
│  known ints : n/a, n/a, n/a                      │ own finds : 699        │
│  dictionary : n/a, n/a, n/a                      │  imported : n/a        │
│havoc/splice : 506/1.05M, 193/1.44M               │ stability : 99.88%     │
│   py/custom : 0/0, 0/0                           ├────────────────────────┘
│        trim : 19.25%/53.2k, n/a                  │          [cpu000: 12%]
└──────────────────────────────────────────────────┘
```

The famous AFL TUI

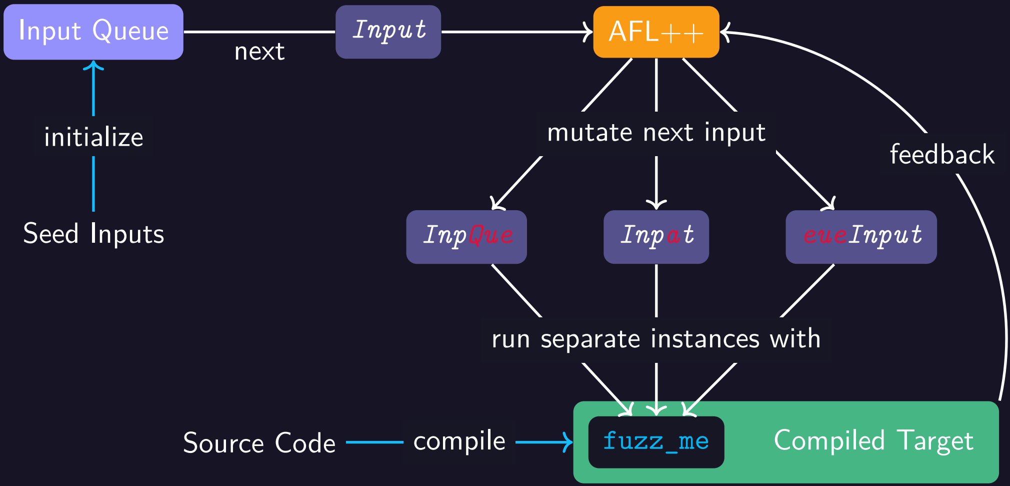
AFL++

- Based on American Fuzzy Lop (AFL)
- Most well known coverage-guided grey-box fuzzer
- Uses execution tracing, comparison coverage and simple constraint solving to mutate input



The famous AFL TUI

AFL++ Schematic



Seed Inputs

- Need to provide initial inputs to AFL++
  - Often called “seed inputs” or just seeds
  - Provide basis for mutation

Seed Inputs

- Need to provide initial inputs to AFL++
  - Often called “seed inputs” or just seeds
  - Provide basis for mutation
- Some considerations:

Seed Inputs

- Need to provide initial inputs to AFL++
  - Often called “seed inputs” or just seeds
  - Provide basis for mutation
- Some considerations:
  - The smaller, the better

Seed Inputs

- Need to provide initial inputs to AFL++
  - Often called "seed inputs" or just seeds
  - Provide basis for mutation
- Some considerations:
  - The smaller, the better
  - No crashing inputs

Seed Inputs

- Need to provide initial inputs to AFL++
  - Often called "seed inputs" or just seeds
  - Provide basis for mutation
- Some considerations:
  - The smaller, the better
  - No crashing inputs
  - Wide range, no inputs should be very similar

Setup Fuzzing

- Select good target functions
  - Complex parsing, many corner cases, etc.
  - Often makes sense to throw fuzzing at only parts of the program

Setup Fuzzing

- Select good target functions
  - Complex parsing, many corner cases, etc.
  - Often makes sense to throw fuzzing at only parts of the program
- Remove potentially difficult-to-fuzz features
  - Checksums, cryptography, etc. lead to many invalid inputs
  - Usually also slow down fuzzing
  - Better to fully remove, to speed up fuzzing

Compiling your Program

- Follow instructions of fuzzer
  - Usually compile with specialized compiler

Compiling your Program

- Follow instructions of fuzzer
  - Usually compile with specialized compiler
- Adds necessary instrumentation

Compiling your Program

- Follow instructions of fuzzer
  - Usually compile with specialized compiler
- Adds necessary instrumentation
- Sanitizers help by crashing when common security issues occur
  - Increases chances that crashes correspond to vulnerabilities
  - Still not guaranteed, hence manual triaging is always required

Fuzzing

Binary-only Fuzzing

Oh no, I "Lost" my Source Code

- **Question:** What if you "lost" access to your source code?[3]

[3] This happens constantly to US military agencies.

Oh no, I "Lost" my Source Code

- **Question:** What if you "lost" access to your source code?[3]
- **Solution:** AFL++ supports fuzzing binary-only targets
  - Uses QEMU (a CPU emulator)
  - Inserts instrumentation on the fly
  - Can be used to fuzz "cross-architecture"

[3] This happens constantly to US military agencies.

Oh no, I "Lost" my Source Code

- **Question:** What if you "lost" access to your source code?[3]
- **Solution:** AFL++ supports fuzzing binary-only targets
  - Uses QEMU (a CPU emulator)
  - Inserts instrumentation on the fly
  - Can be used to fuzz "cross-architecture"
- **Solution:** Can use tools like RetroWrite to statically rewrite binary with instrumentation
  - Results in faster fuzzing
  - Much more tricky to do
  - Still active area of research

[3] This happens constantly to US military agencies.

Fuzzing the iPhone Boot Loader

Motivation

- iPhone security major talking point in the press
  - "Apple Issues Emergency Security Updates to Close a Spyware Flaw" [7]
  - "iOS zero-day let SolarWinds hackers compromise fully updated iPhones" [3]
  - "New iPhone 'Zero Day' Hack Has Existed For Months" [4]
- Boot loader very important for security guarantees

$\Longrightarrow$ **Vulnerabilities in the iPhone boot loader are highly sought after.**

Motivation

- iPhone security major talking point in the press
  - "Apple Issues Emergency Security Updates to Close a Spyware Flaw" [7]
  - "iOS zero-day let SolarWinds hackers compromise fully updated iPhones" [3]
  - "New iPhone 'Zero Day' Hack Has Existed For Months" [4]
- Boot loader very important for security guarantees

$\Longrightarrow$ **Vulnerabilities in the iPhone boot loader are highly sought after.**

- **Goal:** Apply state-of-the-art fuzzing to iPhone boot loader

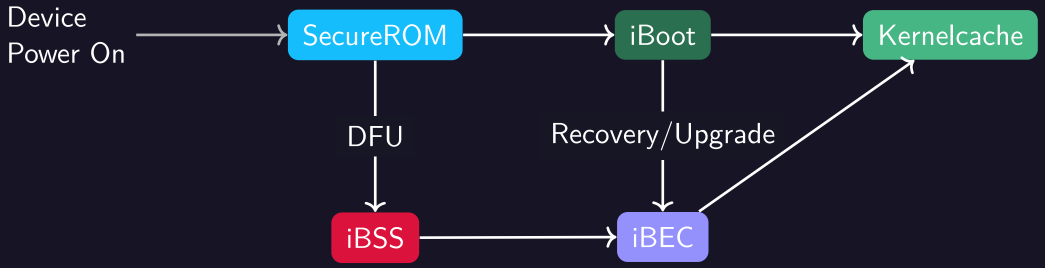
Fuzzing the iPhone Boot Loader

Background

iPhone Boot Sequence

- Boot loader responsible for initializing hardware and setting up everything for the main OS to run
- Consists of multiple stages on iPhones
- Stages form a secure boot chain
  - Every stage loads, verifies and runs next one
  - Verification uses standard X.509 certificate chains, RSA signatures
  - Every stage is stored in a custom format, called IMG4

Schematic Boot Diagram



Schematic view of the iOS boot sequence and its boot loader stages adapted from [5].

Fuzzing the iPhone Boot Loader
Threat Model

Attacking the Secure Boot Chain

- **Question:** Why could attacking SecureROM be interesting?

Attacking the Secure Boot Chain

- **Question:** Why could attacking SecureROM be interesting?
- Exploit in SecureROM very powerful:
  - Getting kernel code execution is trivial
  - Can lead to larger attack surface for Secure Enclave Processor (SEP) [9]
  - Cannot be patched
  - Might lead to persistence

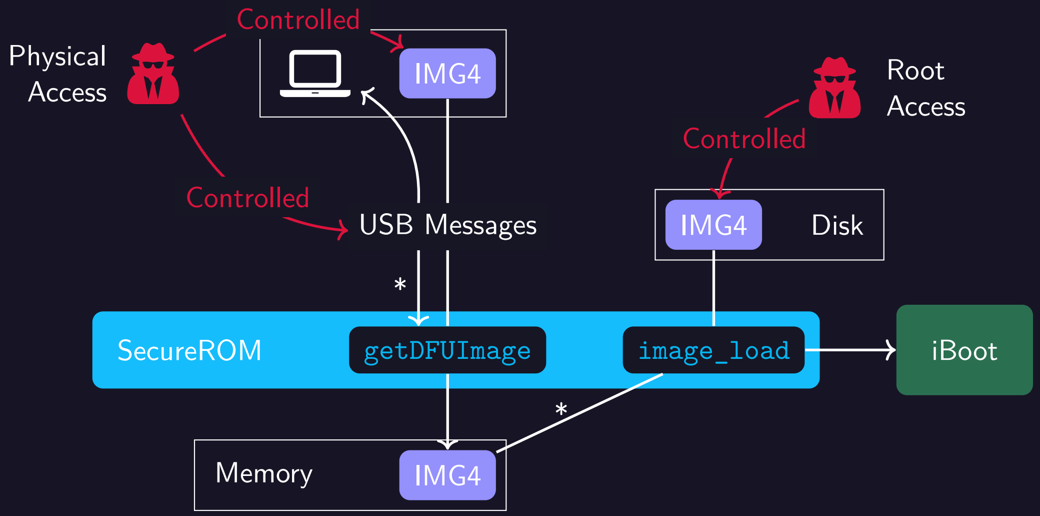
Attacking the Secure Boot Chain

- **Question:** Why could attacking SecureROM be interesting?
- Exploit in SecureROM very powerful:
  - Getting kernel code execution is trivial
  - Can lead to larger attack surface for Secure Enclave Processor (SEP) [9]
  - Cannot be patched
  - Might lead to persistence
- Two major threat models:
  - **Physical access:** Attacker can interface with USB DFU protocol
  - **Root on device:** Attacker can write malformed IMG4 file to disk

Attacking the Secure Boot Chain

- **Question:** Why could attacking SecureROM be interesting?
- Exploit in SecureROM very powerful:
  - Getting kernel code execution is trivial
  - Can lead to larger attack surface for Secure Enclave Processor (SEP) [9]
  - Cannot be patched
  - Might lead to persistence
- Two major threat models:
  - **Physical access:** Attacker can interface with USB DFU protocol
  - **Root on device:** Attacker can write malformed IMG4 file to disk
- We assume the physical access threat model

Schematic Threat Model



Fuzzing the iPhone Boot Loader

Building a Fuzzable Binary

Challenges

- **Normally:** Build from source with instrumentation

Challenges

- **Normally:** Build from source with instrumentation
- Binary blob without symbols

Challenges

- **Normally:** Build from source with instrumentation
- Binary blob without symbols
- Designed for Apple processors

Challenges

- **Normally:** Build from source with instrumentation
- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Static Analysis

```
__int64 sub_100009BCC(char *a1)
{
  sub_1000127BC();
  if (a1 == aKsat || a1 == &unk_19C0107C0)
    sub_100008F90();
  if (*((_QWORD *)a1 + 3) || *((_QWORD *)a1 + 4))
    sub_100009C50(a1 + 24);
  sub_100009C50(a1 + 8);
  v3 = sub_100001C14(a1);
  sub_100012810(v3);
  return sub_10000FEF4(a1);
}
```

Solution: Static Analysis

```c
void task_destroy(struct task *a1)
{
  enter_critical_section();
  if (a1 == &bootstrap_task || a1 == &idle_task)
    panic();
  if (a1->queue_node.prev || a1->queue_node.next)
    list_delete(&a1->queue_node);
  list_delete(&a1->task_list_node);
  arch_task_destroy(a1);
  exit_critical_section();
  heap_free(a1);
}
```

Solution: Static Analysis

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Static Analysis

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Static Analysis

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Static Analysis

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Binary Patching

```
// Convert PAC branch to normal branch
r.PatchInstruction("blraaz ").Patch(r.PatchTmpl("blr {{(index .Args 0)}}"))
// Force bzero to never use dca
symb.rom__bzero.PatchOffset(0x18).Patch(r.PatchASM("cmp x2, #0x40000"))
// Override USB driver with custom one
symb.rom_synopsys_otg_controller_init.PatchOffset(0).Patch(
        r.PatchFunctionNoLink("emmutaler_controller_init")
)
// Patch in custom root certificate
certPath := filepath.Join(filepath.Dir(r.inputPath), "..", "certs", "root_ca.der")
certData, _ := os.ReadFile(certPath)
r.RawPatch(symb.rom_root_ca.Start, len(certData),
        fmt.Sprintf(`.incbin "%s"`, certPath))
```

Solution: Binary Patching

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Binary Patching

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Binary Patching

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: Binary Patching

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: SecureROM as a Library

- **Main idea:** Create normal Linux program calling into SecureROM as necessary.

Solution: SecureROM as a Library

- **Main idea:** Create normal Linux program calling into SecureROM as necessary.
- Can use existing fuzzers without modifications
- Functions interesting to fuzz do not need low-level access
- Can fuzz selectively
- Easy to debug without complicated fuzzing harness

Solution: SecureROM as a Library

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: SecureROM as a Library

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

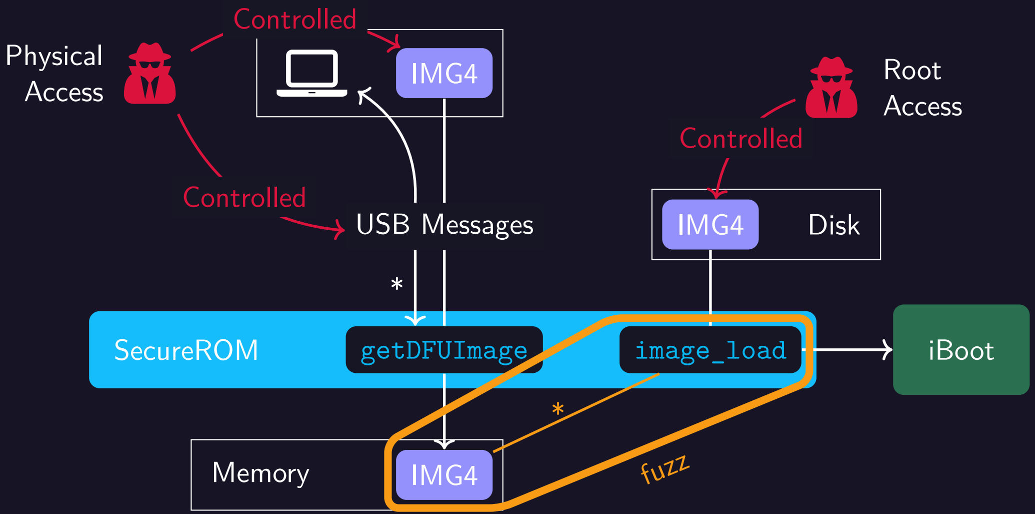
Solution: SecureROM as a Library

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

Solution: SecureROM as a Library

- Binary blob without symbols
- Designed for Apple processors
- Bare metal or bust

IMG4 Fuzzing



IMG4 Schematic

Schematic view of the high-level fuzzing design for IMG4 parsing.

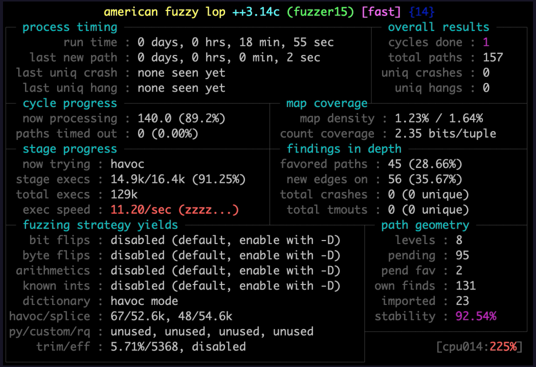
Results

- Ran for one week
- No interesting crashes
- Interesting results with respect to speed

IMG4 Fuzzing
Fuzzing Speed

Fuzzing Speed

- **Problem:** Fuzzing speed is much lower than expected



Fuzzing Speed

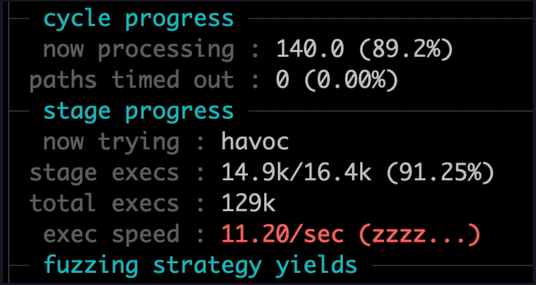
- **Problem:** Fuzzing speed is much lower than expected

```
─ cycle progress ──────────────────────
  now processing : 140.0 (89.2%)
 paths timed out : 0 (0.00%)
─ stage progress ──────────────────────
  now trying : havoc
 stage execs : 14.9k/16.4k (91.25%)
 total execs : 129k
  exec speed : 11.20/sec (zzzz...)
─ fuzzing strategy yields ─────────────
```

We get it, it is slow

Fuzzing Speed

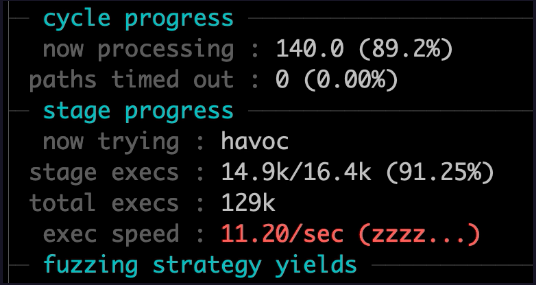
- **Problem:** Fuzzing speed is much lower than expected
- Multiple factors:
  - Target restarted for every run



We get it, it is slow

Fuzzing Speed

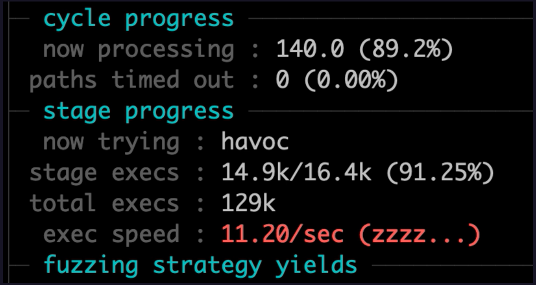
- **Problem:** Fuzzing speed is much lower than expected
- Multiple factors:
  - Target restarted for every run
  - PAC instructions are slow in software



We get it, it is slow

Fuzzing Speed

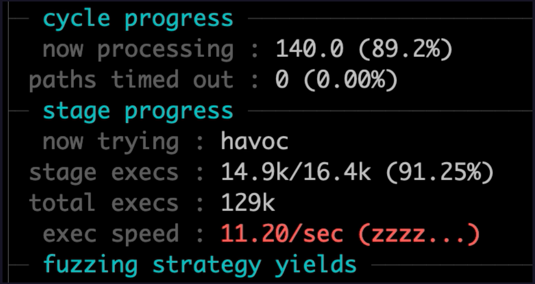
- **Problem:** Fuzzing speed is much lower than expected
- Multiple factors:
  - Target restarted for every run
  - PAC instructions are slow in software
- **Solution:** Patch QEMU to ignore PAC



We get it, it is slow

Fuzzing Speed

- **Problem:** Fuzzing speed is much lower than expected
- Multiple factors:
  - Target restarted for every run
  - PAC instructions are slow in software
- **Solution:** Patch QEMU to ignore PAC
- **Solution:** Use persistent mode for *better* performance



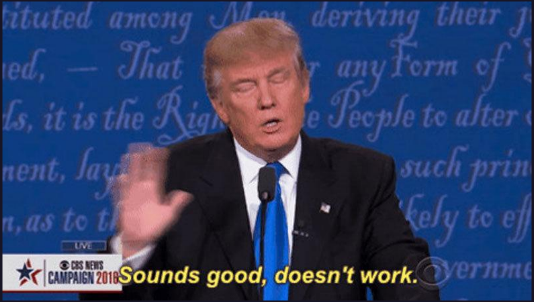
We get it, it is slow

Persistent Mode

- AFL++ creates "fake" loop by going back to fuzzing function after exit
- Should provide great speedup

Persistent Mode

- AFL++ creates "fake" loop by going back to fuzzing function after exit
- Should provide great speedup
- **Problem:** Fuzzing is now actually slower



Persistent Mode

- AFL++ creates "fake" loop by going back to fuzzing function after exit
- Should provide great speedup
- **Problem:** Fuzzing is now actually slower
- AFL++ snapshots all writable memory pages

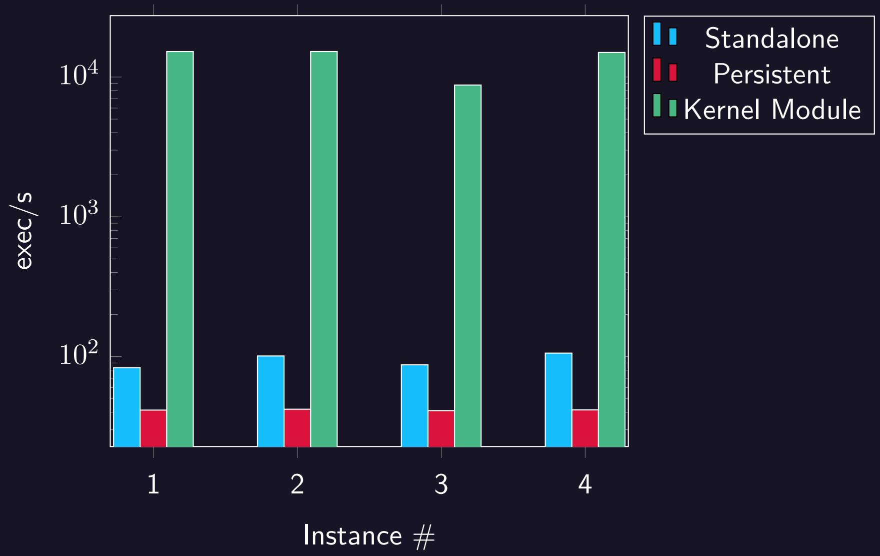
Persistent Mode

- AFL++ creates "fake" loop by going back to fuzzing function after exit
- Should provide great speedup
- **Problem:** Fuzzing is now actually slower
- AFL++ snapshots all writable memory pages
- **Solution:** Patch AFL++ to only snapshot certain pages

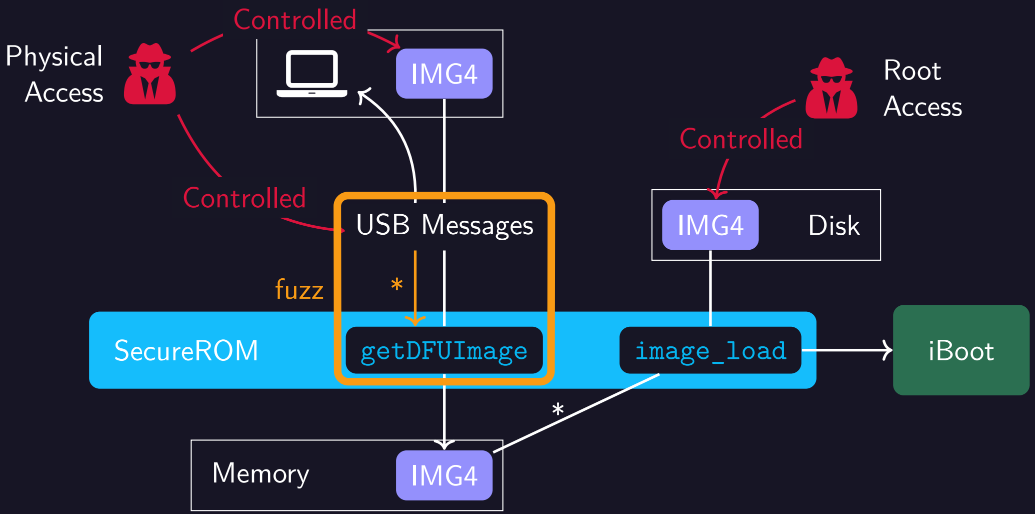
Persistent Mode

- AFL++ creates “fake” loop by going back to fuzzing function after exit
- Should provide great speedup
- **Problem:** Fuzzing is now actually slower
- AFL++ snapshots all writable memory pages
- **Solution:** Patch AFL++ to only snapshot certain pages
- **Solution:** Use kernel module for copy-on-write snapshotting [10]

IMG4 Fuzzing Speed Results

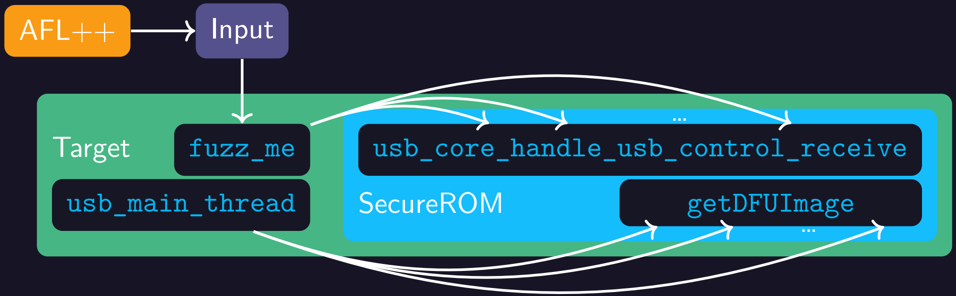


USB Fuzzing



Controlled
Physical
Access
IMG4
Root
Access
Controlled
Controlled
USB Messages
IMG4
Disk
fuzz
*
SecureROM
getDFUImage
image_load
iBoot
*
Memory
IMG4

USB Schematic



Schematic view of the high-level fuzzing design for USB messages.

Short Refresher on the Heap

- Provides mechanism for dynamic memory allocation

Short Refresher on the Heap

- Provides mechanism for dynamic memory allocation
- Used by most programs, but might not be directly visible

Short Refresher on the Heap

- Provides mechanism for dynamic memory allocation
- Used by most programs, but might not be directly visible
- `void* ptr = malloc(100)` : allocate 100 bytes
  - Starting at `ptr` , 100 bytes of memory available for anything
  - Usually called dynamically allocated buffer

Short Refresher on the Heap

- Provides mechanism for dynamic memory allocation
- Used by most programs, but might not be directly visible
- `void* ptr = malloc(100)` : allocate 100 bytes
  - Starting at `ptr` , 100 bytes of memory available for anything
  - Usually called dynamically allocated buffer
- `free(ptr)` : release memory previously allocated to be used elsewhere
  - `ptr` should not be used afterwards
  - Needed, since memory management is manual
  - Everything allocated must be freed by programmer

checkm8

Use-After-Free (UAF)

A use-after-free occurs when a pointer to a buffer on the heap is used, after said buffer has already been freed.

- Previously found vulnerability in DFU protocol titled “checkm8” [1]
- Core bug exploited: use-after-free (UAF) in DFU protocol handling
- Before this thesis, iPhone 4S to X were publicly known to exhibit the UAF bug [1].

checkm8

Use-After-Free (UAF)

A use-after-free occurs when a pointer to a buffer on the heap is used, after said buffer has already been freed.

- Previously found vulnerability in DFU protocol titled “checkm8” [1]
- Core bug exploited: use-after-free (UAF) in DFU protocol handling
- Before this thesis, iPhone 4S to X were publicly known to exhibit the UAF bug [1].
- **Goal:** Our fuzzing finds the same UAF bug
- Shows that the fuzzing is successful, since it can find bugs

The Quest for checkm8

- **Problem:** Fuzzer does not find any crashes
- Surprising at first

The Quest for checkm8

- **Problem:** Fuzzer does not find any crashes
- Surprising at first
- Multiple possible reasons:
  - Fuzzing does not work correctly

The Quest for checkm8

- **Problem:** Fuzzer does not find any crashes
- Surprising at first
- Multiple possible reasons:
  - Fuzzing does not work correctly
  - Fuzzing the wrong parts

The Quest for checkm8

- **Problem:** Fuzzer does not find any crashes
- Surprising at first
- Multiple possible reasons:
  - Fuzzing does not work correctly
  - Fuzzing the wrong parts
  - Fuzzing finds the bug, but does not crash

The Quest for checkm8

- **Problem:** Fuzzer does not find any crashes
- Surprising at first
- Multiple possible reasons:
  - Fuzzing does not work correctly
  - Fuzzing the wrong parts
  - Fuzzing finds the bug, but does not crash

Heap Feng Shui

Heap Feng Shui [8]

The process of carefully manipulating the heap, allowing exploitation. It is also sometimes called "heap grooming". Usually, it consists of allocating and freeing very specific sizes in a specific order to get the heap into a very specific state.

- checkm8 performs complicated "heap feng shui" before actual exploit
- Otherwise, exploited buffer is allocated at the same place
- Not exclusive to SecureROM

Heap Feng Shui

Heap Feng Shui [8]

The process of carefully manipulating the heap, allowing exploitation. It is also sometimes called “heap grooming”. Usually, it consists of allocating and freeing very specific sizes in a specific order to get the heap into a very specific state.

- checkm8 performs complicated “heap feng shui” before actual exploit
- Otherwise, exploited buffer is allocated at the same place
- Not exclusive to SecureROM
- **Solution:** Custom allocator tailored to find heap bugs that depend on specific state

USB Fuzzing

Fuzzing-Enabling Thread-safe Allocator (FETA)

FETA Overview

- Drop-in replacement for any code using `malloc` and `free`
- Thread-safe
- Can detect and crash on:
  - heap overflows, both read and write
  - use-after-free, both read and write

FETA Overview

- Drop-in replacement for any code using `malloc` and `free`
- Thread-safe
- Can detect and crash on:
  - heap overflows, both read and write
  - use-after-free, both read and write
- Basic idea:
  - Want to crash as soon as bug happens

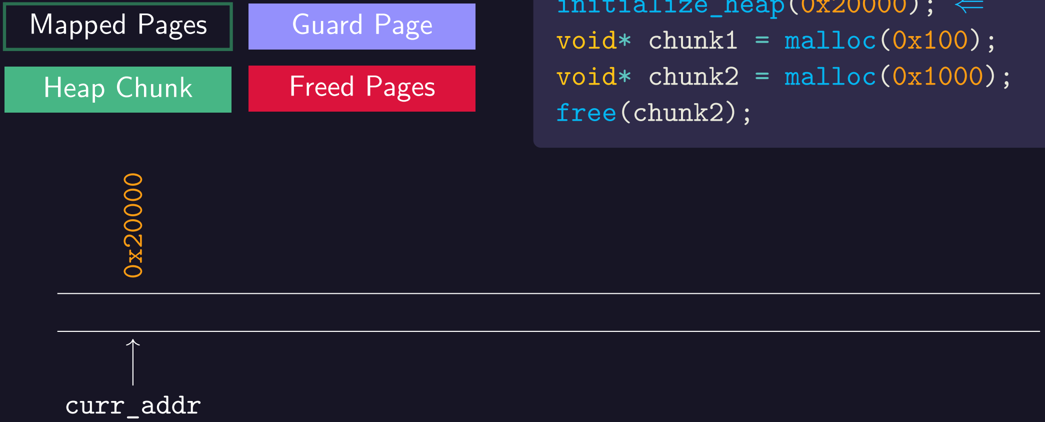
FETA Overview

- Drop-in replacement for any code using `malloc` and `free`
- Thread-safe
- Can detect and crash on:
  - heap overflows, both read and write
  - use-after-free, both read and write
- Basic idea:
  - Want to crash as soon as bug happens
  - Access to unmapped page causes immediate crash, for both read and write

FETA Overview

- Drop-in replacement for any code using `malloc` and `free`
- Thread-safe
- Can detect and crash on:
  - heap overflows, both read and write
  - use-after-free, both read and write
- Basic idea:
  - Want to crash as soon as bug happens
  - Access to unmapped page causes immediate crash, for both read and write
  - **Solution:** "isolate" every heap chunk to its own set of pages

Example Allocations with FETA

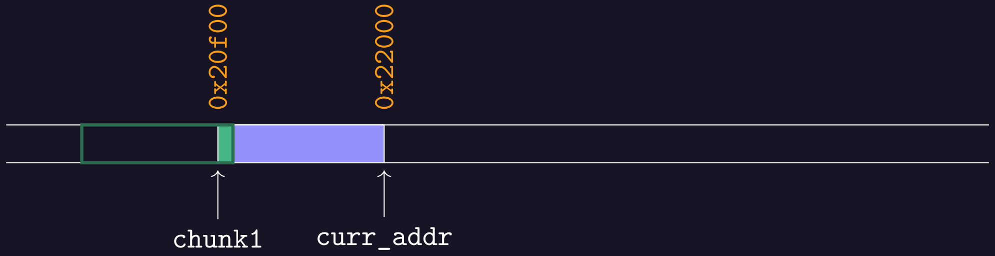



```
initialize_heap(0x20000); ⇐
void* chunk1 = malloc(0x100);
void* chunk2 = malloc(0x1000);
free(chunk2);
```

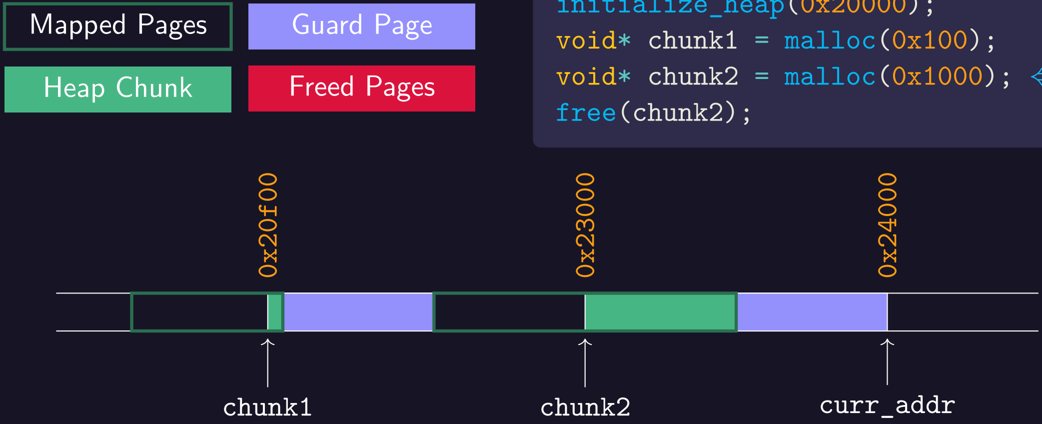
Example Allocations with FETA

Mapped Pages | Guard Page
Heap Chunk | Freed Pages

```c
initialize_heap(0x20000);
void* chunk1 = malloc(0x100); ⇐
void* chunk2 = malloc(0x1000);
free(chunk2);
```

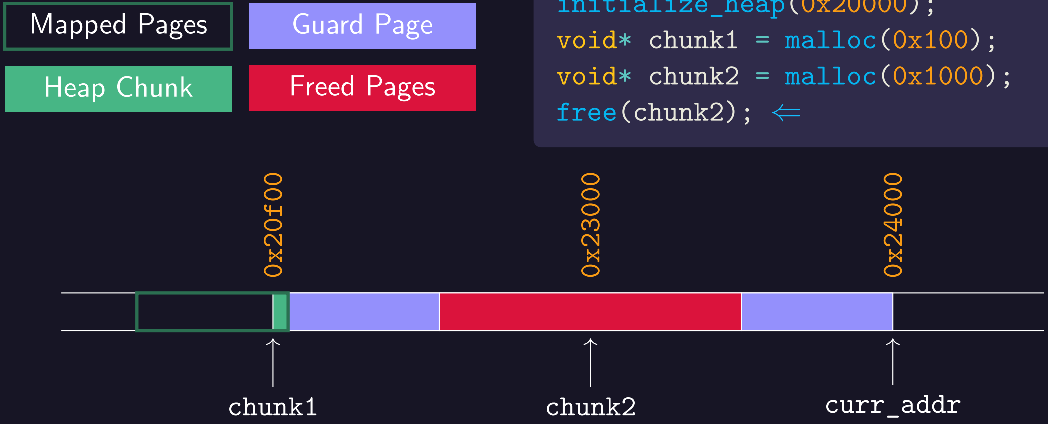


Example Allocations with FETA




```
initialize_heap(0x20000);
void* chunk1 = malloc(0x100);
void* chunk2 = malloc(0x1000); ⇐
free(chunk2);
```

Example Allocations with FETA

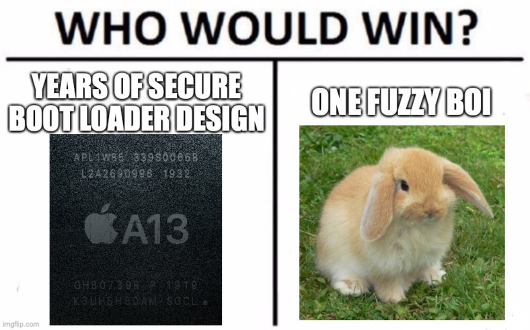



```
initialize_heap(0x20000);
void* chunk1 = malloc(0x100);
void* chunk2 = malloc(0x1000);
free(chunk2); ⇐
```

USB Fuzzing
Results

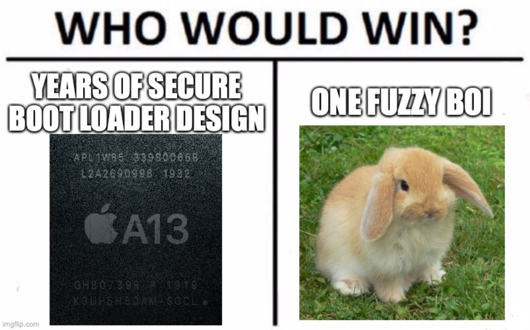
Overall Results

- checkm8 could be found using FETA
- Time-to-exposure (TTE) very short
- Confirms that checkm8 still present on iPhone 11



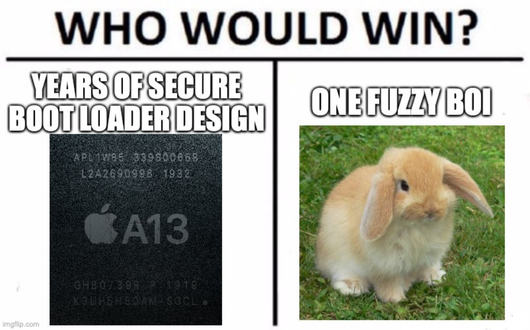
Overall Results

- checkm8 could be found using FETA
- Time-to-exposure (TTE) very short
- Confirms that checkm8 still present on iPhone 11
- **Question:** What other bugs are we missing?



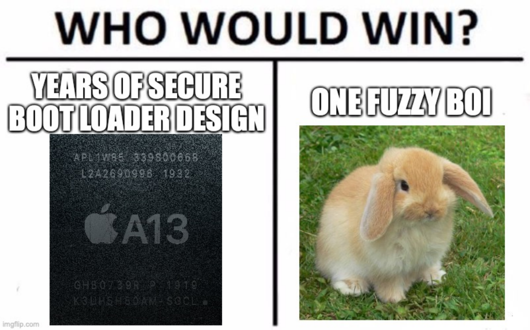
Overall Results

- checkm8 could be found using FETA
- Time-to-exposure (TTE) very short
- Confirms that checkm8 still present on iPhone 11
- **Question:** What other bugs are we missing?
  - Interesting future research possibilities



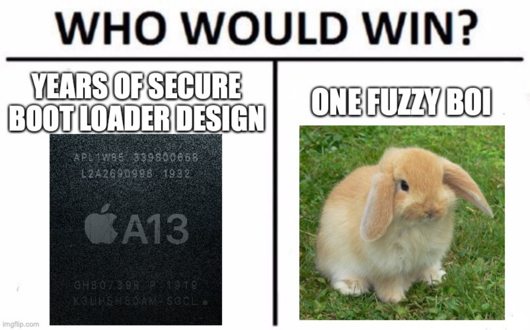
Overall Results

- checkm8 could be found using FETA
- Time-to-exposure (TTE) very short
- Confirms that checkm8 still present on iPhone 11
- **Question:** What other bugs are we missing?
  - Interesting future research possibilities
  - Expand FETA to also detect memory leaks via fuzzing



Overall Results

- checkm8 could be found using FETA
- Time-to-exposure (TTE) very short
- Confirms that checkm8 still present on iPhone 11
- **Question:** What other bugs are we missing?
  - Interesting future research possibilities
  - Expand FETA to also detect memory leaks via fuzzing
  - Threading library to expose race conditions?



Conclusion

Conclusion

- Fuzzing becoming more and more important

Conclusion

- Fuzzing becoming more and more important
- Setup can be quite quick with modern tooling
  - No more excuses

Conclusion

- Fuzzing becoming more and more important
- Setup can be quite quick with modern tooling
  - No more excuses
- Does not replace security engineers

Conclusion

- Fuzzing becoming more and more important
- Setup can be quite quick with modern tooling
  - No more excuses
- Does not replace security engineers

- iPhone boot loader fuzzing was successful
  - Confirmed existence of checkm8 on iPhone 11

Conclusion

- Fuzzing becoming more and more important
- Setup can be quite quick with modern tooling
  - No more excuses
- Does not replace security engineers

- iPhone boot loader fuzzing was successful
  - Confirmed existence of checkm8 on iPhone 11
- FETA performs great and raises interesting questions

Useful Links

Fuzzing

- Fuzzing in Go 1.18: go.dev/blog/fuzz-beta
- AFL++ documentation: aflplus.plus
- Fuzzing-101: github.com/antonio-morales/Fuzzing101
- Awesome Fuzzing Discord: discord.gg/vmAGPuUUvn

Other

- Source code for iPhone boot loader fuzzing: github.com/galli-leo/emmutaler
- flagbot homepage: `flagbot.ch`
- These slides: flagbot.ch/material

Questions?

Bibliography

[1] axi0mX. EPIC JAILBREAK: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of iOS devices. Most generations of iPhones and iPads are vulnerable: from iPhone 4S (A5 chip) to iPhone 8 and iPhone X (A11 chip). Sept. 27, 2019. URL: https://twitter.com/axi0mX/status/1177542201670168576. https://twitter.com/axi0mX.

[2] Clusterfuzz. URL: https://github.com/google/clusterfuzz (visited on 10/14/2021).

[3] D. Goodin. Ios zero-day let solarwinds hackers compromise fully updated iphones. 2021. URL: https://arstechnica.com/gadgets/2021/07/solarwinds-hackers-used-an-ios-0-day-to-steal-google-and-microsoft-credentials/ (visited on 09/21/2021).

[4] G. Kelly. New iphone 'zero day' hack has existed for months. 2021. URL: https://www.forbes.com/sites/gordonkelly/2021/07/31/apple-iphone-12-pro-max-memory-hack-warning-ios-update-iphones-ipads/?sh=2e15b393f8e5 (visited on 09/21/2021).

[5] J. Levin. **OS Internals Volume II: Kernel Mode.* Technologeeks.com, 2020.

[6] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*, Aug. 2020. URL: https://comsec.ethz.ch/wp-content/files/parmesan_sec20.pdf.

[7] N. Perlroth. Apple issues emergency security updates to close a spyware flaw. URL: https://www.nytimes.com/2021/09/13/technology/apple-software-update-spyware-nso-group.html (visited on 09/18/2021).

[8] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007:11–20, 2007.

[9] H. Xu. Attack Secure Boot of SEP. Paper presented at MOSEC 2020, Shanghai, China. 2020.

[10] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2313–2328, Dallas, Texas, USA. Association for Computing Machinery, 2017. ISBN: 9781450349468. DOI: 10.1145/3133956.3134046. URL: https://doi.org/10.1145/3133956.3134046.