Fuzzing the Phone in the iPhone

D-d-d-di-d-di-d-d-d-d-dimm!

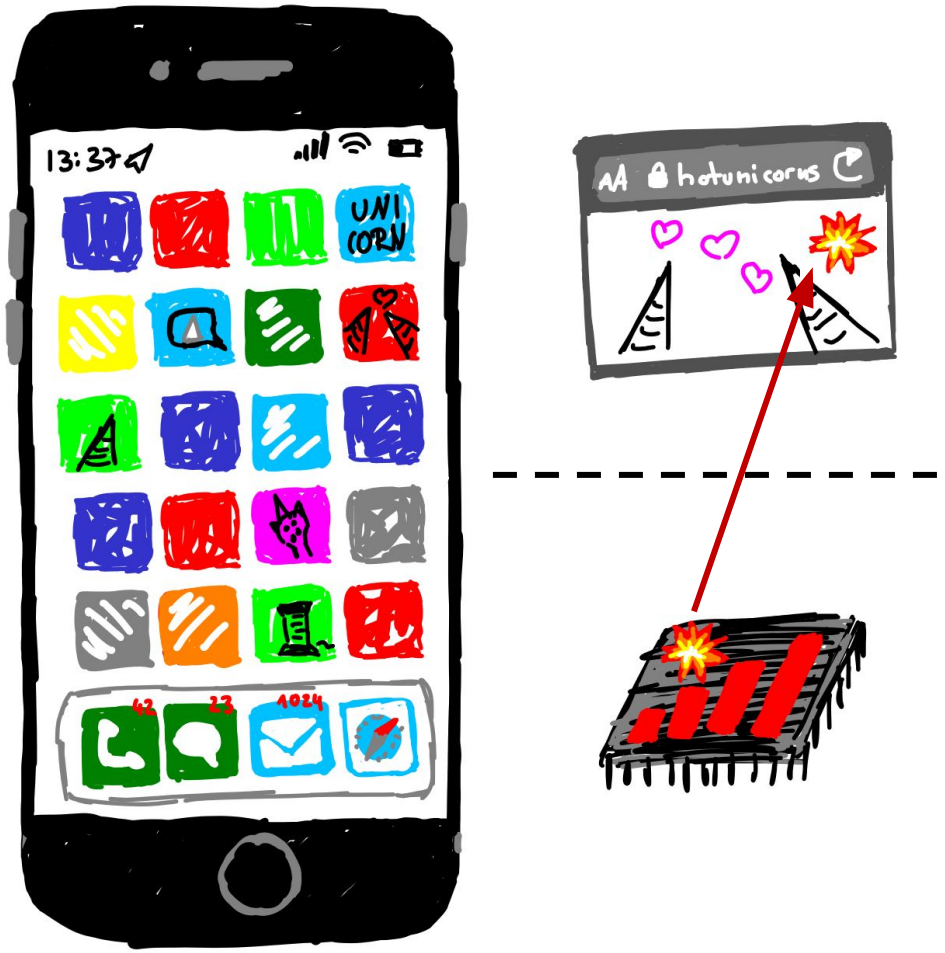
Jiska Classen
Secure Mobile Networking Lab - SEEMOO
Technische Universität Darmstadt, Germany

Video that I already posted on Twitter

In this video I’m fuzzing SMS. I like to take SMS as an example since they’re probably the most tested functionality in the iPhone baseband.

Motivation

Baseband Security Assumptions



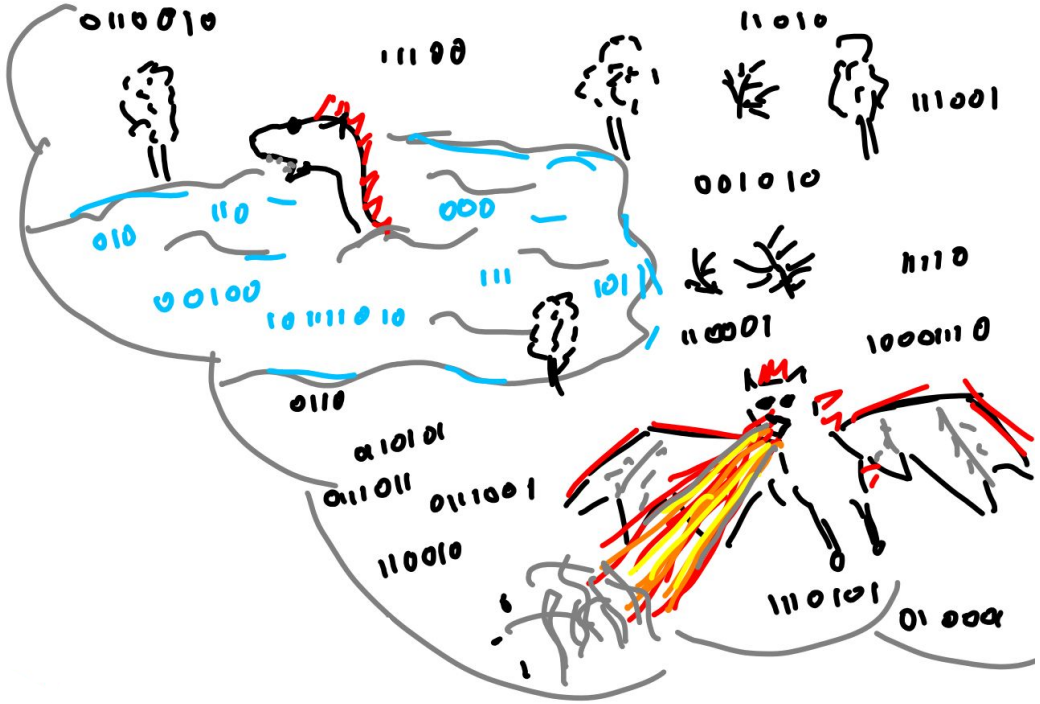
"Once the attacker is in the baseband they can further escalate using a browser exploit."

Zerodium lists Baseband RCE+LPE at $200k but Safari/Chrome RCE+LPE at $500k.

RCE: Remote Code Execution, LPE: Local Privilege Escalation, Zerodium: 💩

Baseband Security Assumptions

Traffic might be manipulated anywhere on the path to a smartphone, no need for a baseband exploit.



Attacks like IMP4GT allow over-the-air traffic manipulation, see https://imp4gt-attacks.net by David Rupprecht (@pily112).

SS7 Attacks

See 31C3 talks for reference: "SS7: Locate. Track. Manipulate." by Tobias Engel (@2b_as) & "Mobile self-defense" by Karsten Nohl.

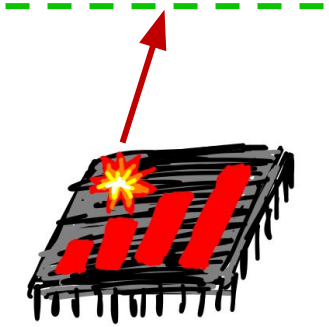
Baseband exploits solely exist to stealthily escalate into the operating system.

Baseband RCE+LPE Strategies?

- There is interest in baseband exploits.
- LPE is very likely not via browsers.
  - Requires user interaction with the browser, not instant.
  - Could be achieved using traffic manipulation anywhere else.
  - Too expensive.

Breaking Local Privilege Escalation (LPE) paths makes baseband exploitation harder.

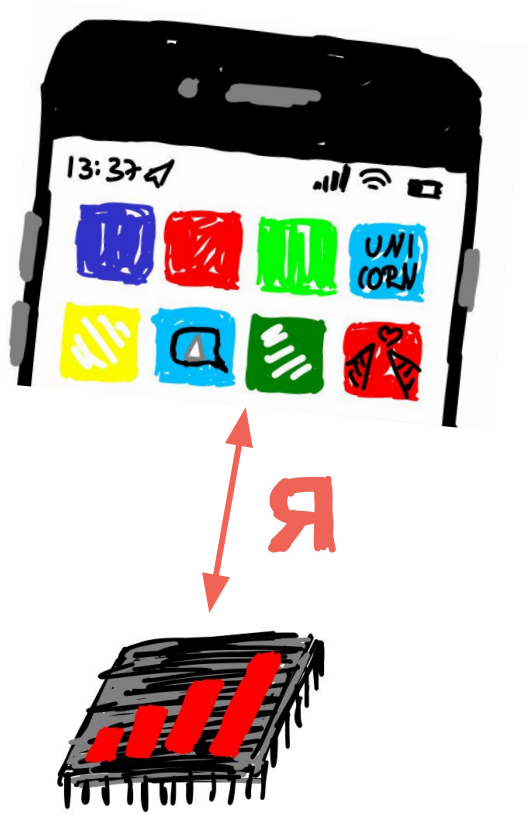
Attackers with on-chip Remote Code Execution (RCE) need to escalate.



A Reverse Engineer's Perspective 👩‍💻

- Observe baseband modem interaction on iOS.
  Only requires a baseband debug profile.

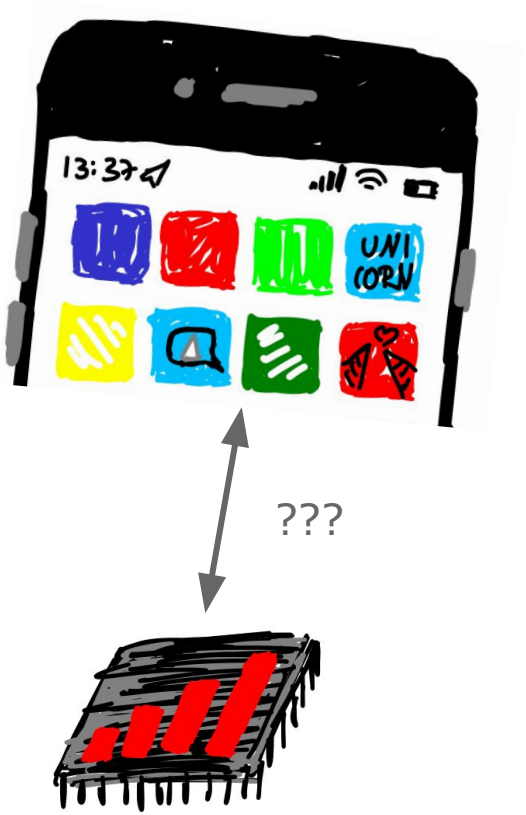
- Interact with the baseband modem.
  Requires a jailbreak and FЯIDA.



https://developer.apple.com/bug-reporting/profiles-and-logs/

Let’s fuzz this!

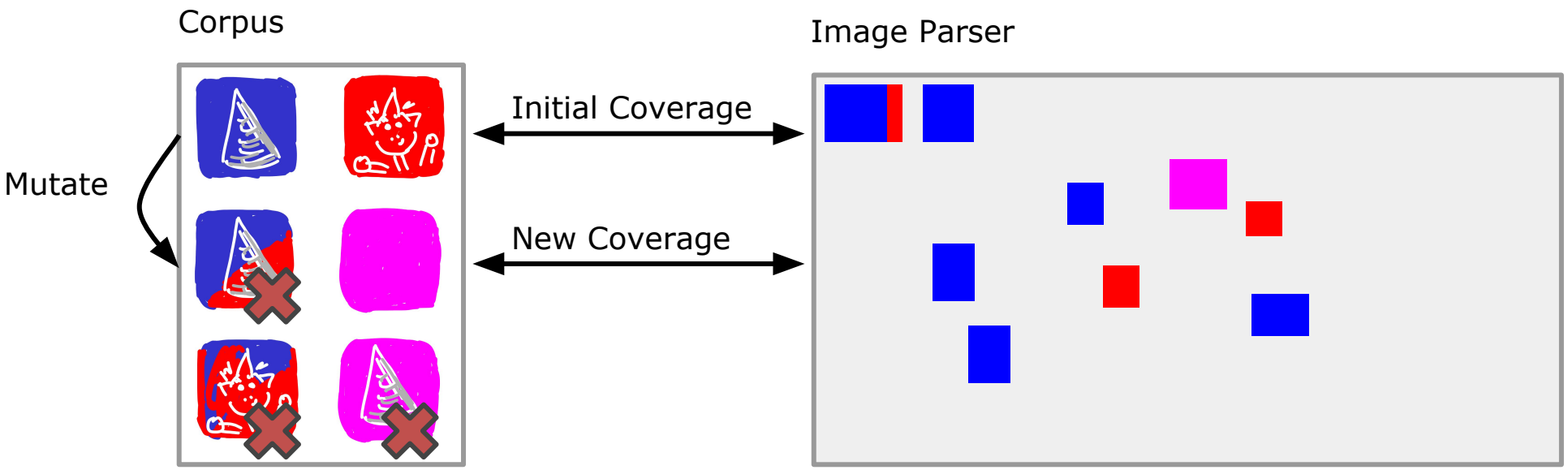
- Did I identify the baseband↔iPhone interface correctly?
- How powerful is this interface?
- Has it been designed with security in mind?



Wireless Protocol Fuzzing

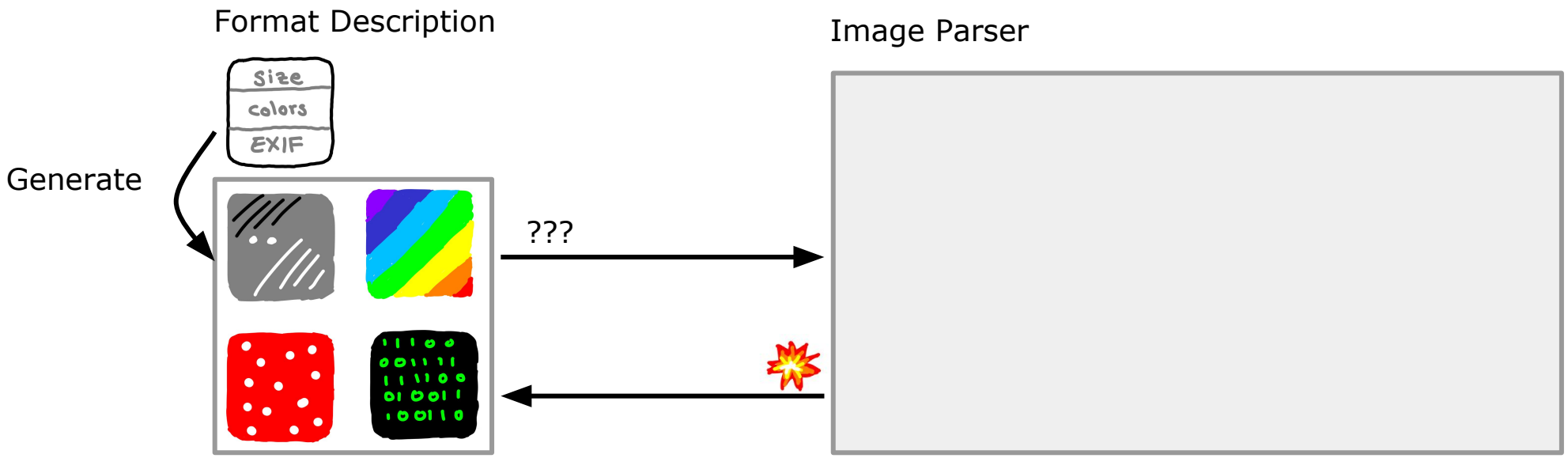
A SHORT RANT!!!! AAARGH!

Mutation-based Fuzzing + Coverage



Mutate corpus, input set should have high quality.
Ideally combined with coverage feedback.

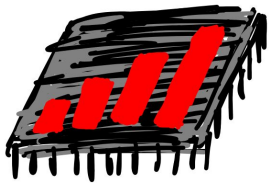
Generation-based Fuzzing + Crash Feedback



Generate new inputs based on knowledge about the input format.
Can also be combined with mutations, coverage, etc.

Modifying vs. Injecting Packets

Modification
Wait for payload and change contents.
Slow but preserves complex states.



SMS from +1234567: 🐴🐴🐴

When fuzzing wireless protocols, the state difference is significant.

Some states cannot be reached solely by injecting packets.

Injection
Generate arbitrary payloads and inject them.
Fast but likely screws up complex states.

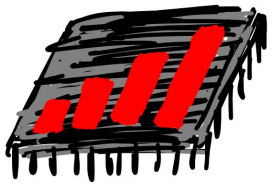
Current date: 1.1.1970

SMS from +1337: Secure Message Secondfactor

Call from +414141414141414141414141414141414141

D-d-d-di-d-di-d-d-d-d-dimm!

Injection Example: Trace Replay



Packet 1

Packet 2

Packet 3

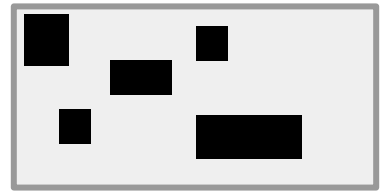
Packet 4

Packet 1

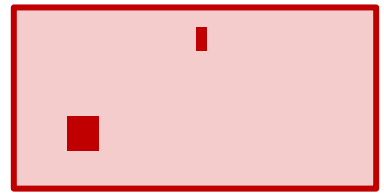
Packet 3

Replaying a recorded message results in less or inconsistent coverage.

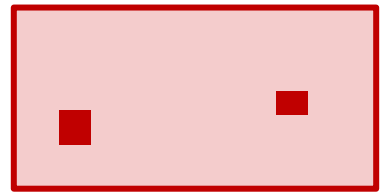
Original Coverage



Replay Coverage



Replay Coverage #2



Video of Injection-based Fuzzing

In this video you can see how I fuzzed based on existing packets but exchanged a couple of strings. I promise the ‘Unicorn<3’ provider also brings 5G to the iPhone 8.

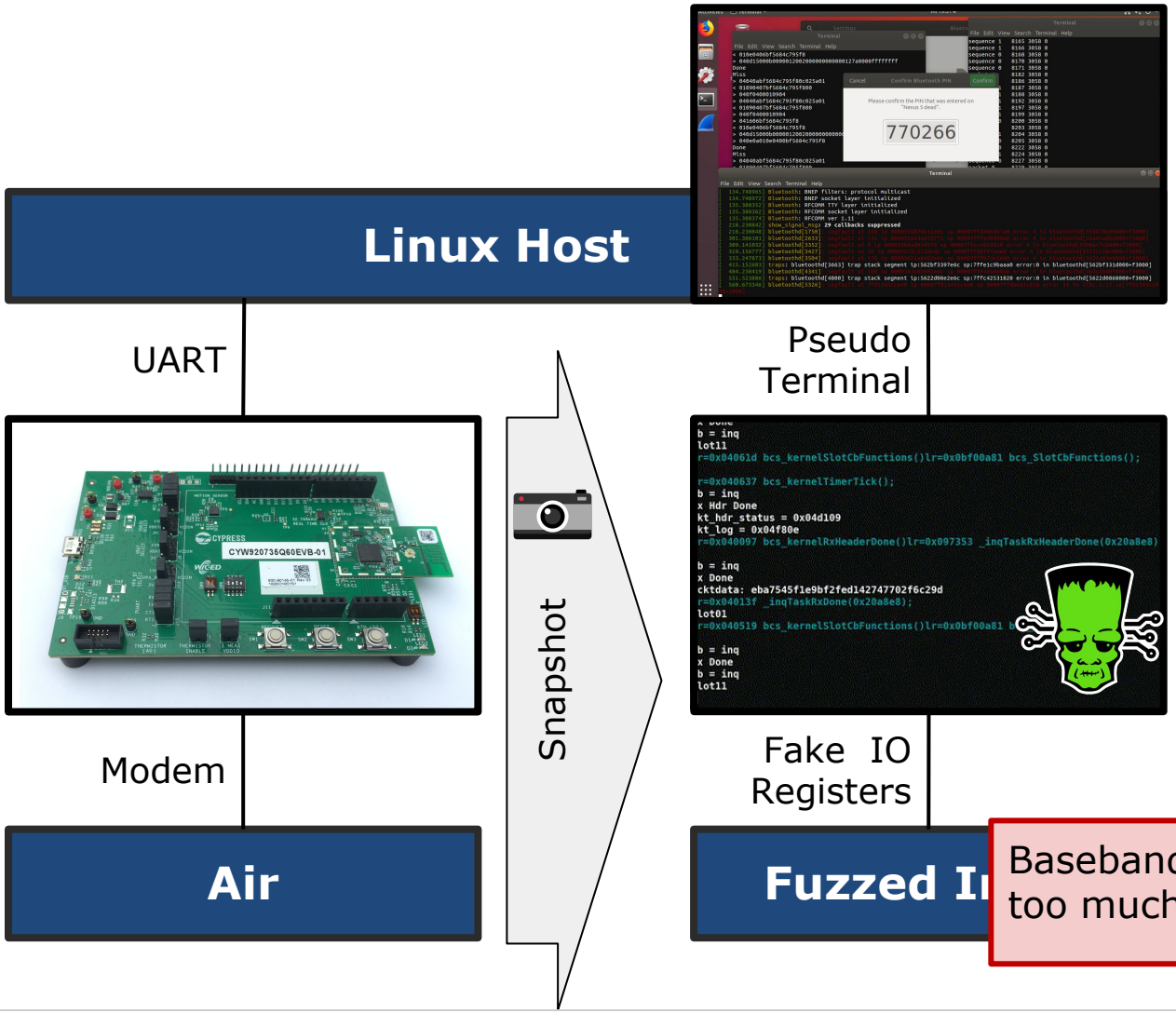
Some states cannot be reached solely by injecting packets.

Fuzz this 💩.
We’re going to miss 80% of the 💩 anyway.

Previously...

Fuzzing at SEMC

Frankenstein



Frankenstein: Emulate Bluetooth firmware with the same speed as in hardware for realistic **full-stack fuzzing**.

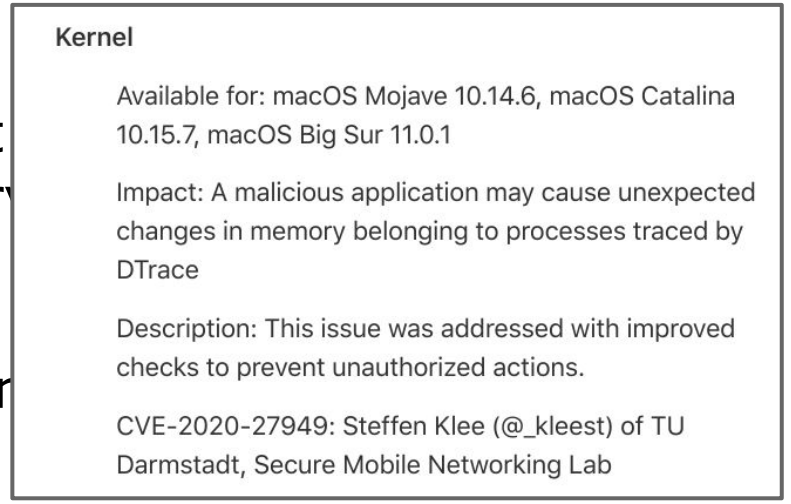
- The Linux host can run a full Bluetooth stack on a desktop setup.
- Add an xmit_state hook to the
- Bluetooth firmware function of interest, e.g., device scanning, active connection, …
- Reattach emulated snapshot with btattach, enter a similar state on the desktop, and start fuzzing.

Baseband firmware is 10x the size of Bluetooth firmware, too much work to customize this, and still no LPE.

By Jan Ruge, https://github.com/seemoo-lab/frankenstein

DTrace & AFL

- DTrace can collect function-level coverage in the **macOS kernel & user space**
- AFL requires basic block coverage…
- DTrace scripts:
  - Use disassembler to determine basic blocks in target
  - Set additional tracepoints to collect coverage at ever
  - Report coverage to AFL(++)
- Slightly faster than FЯIDA
- **1k fcps in Wi-Fi, 3-5k fcps in Bluetooth** 🔥🔥🔥 on ar



Kernel

Available for: macOS Mojave 10.14.6, macOS Catalina 10.15.7, macOS Big Sur 11.0.1

Impact: A malicious application may cause unexpected changes in memory belonging to processes traced by DTrace

Description: This issue was addressed with improved checks to prevent unauthorized actions.

CVE-2020-27949: Steffen Klee (@_kleest) of TU Darmstadt, Secure Mobile Networking Lab

- Complexity of wireless protocols means that no bugs were found with AFL 🙃
- No arbitrary traps in kernel space

DTrace is not supported by iOS (might work with some tweaks) and everything in iOS user space can be targeted with FЯIDA.

Wang Yu (@keenjoy95) found multiple issues in macOS Bluetooth and Wi-Fi drivers while we were fuzzing as well.

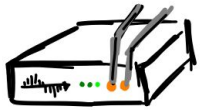
Emulation

Implementing Specifications

Firmware Modification

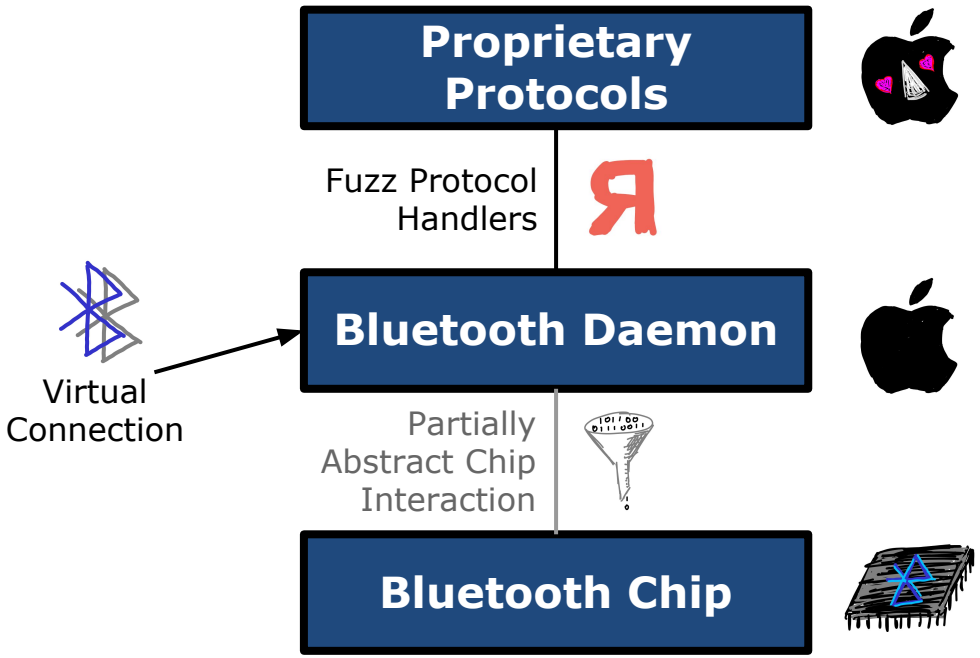
Protocol Reverse Engineering

Software-defined Radios



Fuzzing with FRIDA

ToothPicker



Bluetooth

Available for: iPhone 6s and later, iPad Air 2 and later, iPad mini 4 and later, and iPod touch 7th generation

Impact: A remote attacker may be able to cause arbitrary code execution

Description: An out-of-bounds read was addressed with improved bounds checking.

CVE-2020-9838: Dennis Heinze (@ttdennis) of TU Darmstadt, Secure Mobile Networking Lab

Description: A denial of service issue was addressed with improved input validation.

CVE-2020-9931: Dennis Heinze (@ttdennis) of TU Darmstadt, Secure Mobile Networking Lab

We would like to acknowledge Dennis Heinze (@ttdennis) of TU Darmstadt, Secure Mobile Networking Lab for their assistance.

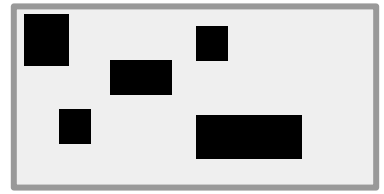
Wait what we can find Bluetooth vulns with 20 fcps?!

Why can we find Bluetooth bugs with 20 fcps???

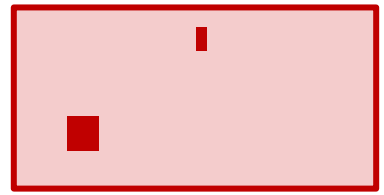
- Over-the-air connections are terminated after a few invalid packets.
- Virtual connections double the coverage during replay.
  - We only miss 50% during replay+fuzzing.
  - Virtual connections introduce behavior that cannot be triggered over-the-air 🙃

- Inconsistent coverage is fixed by replaying each packet 5x in a row.
  - The mutator is not aware of packet sequences 🙃

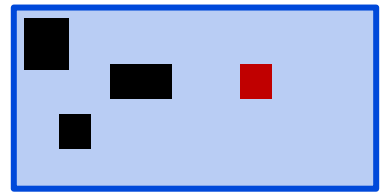
Original Coverage



Replay Coverage



Replay+Fuzzing with Virtual Connection



Video of Bluetooth Inplace Fuzzer

So, after seeing the BlackHat talk, I asked myself how much we missed due to state within bluetoothd. As a simple PoC I wrote an in-place fuzzer that replaced single bytes every few packets in HCI, ACL, and SCO packets.

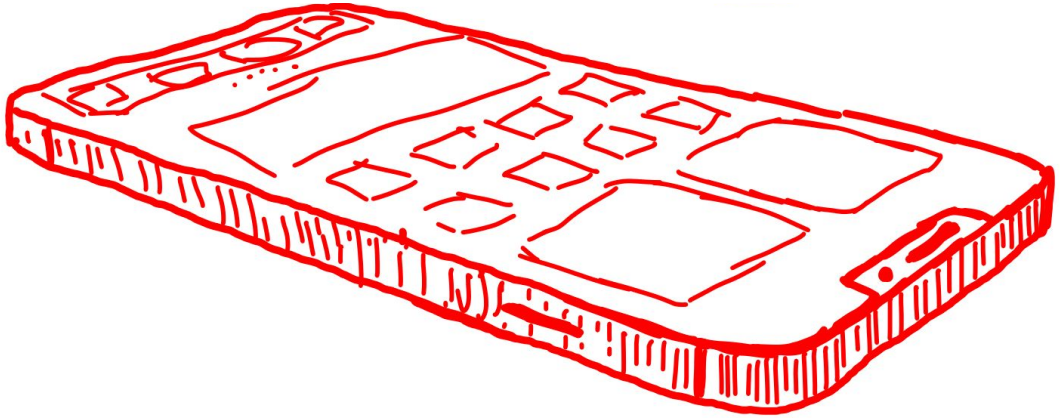
It uncovered that the Bluetooth chip itself is missing length checks (CVE-2020-24362), but except from that, I let it run many many days per payload type and didn’t get any crash we knew before.

Bluetooth Inplace Modification

- Fuzzed multiple active connections with different protocols an a few pairings etc.
- No new `bluetoothd` bugs found.
- Might have harmed my AirPods' sound quality…

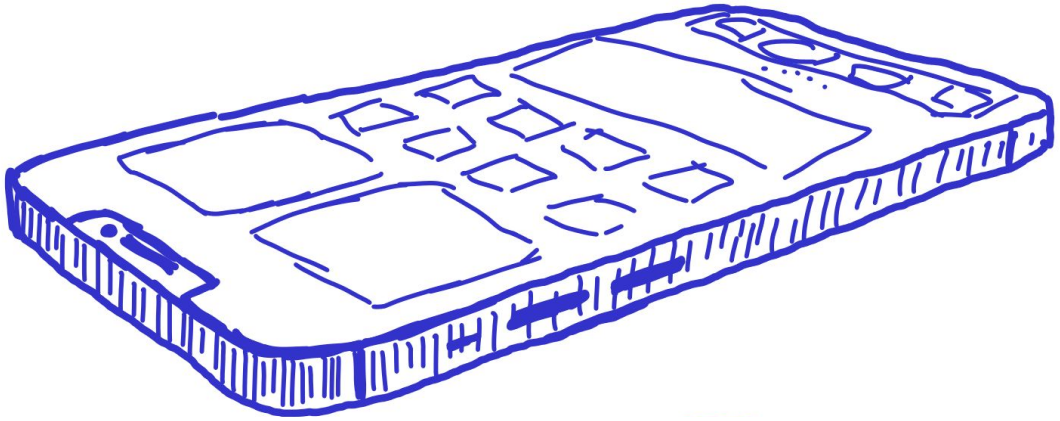
- Broadcom chips don't check UART lengths (CVE-2020-24362).

iPhone Basebands



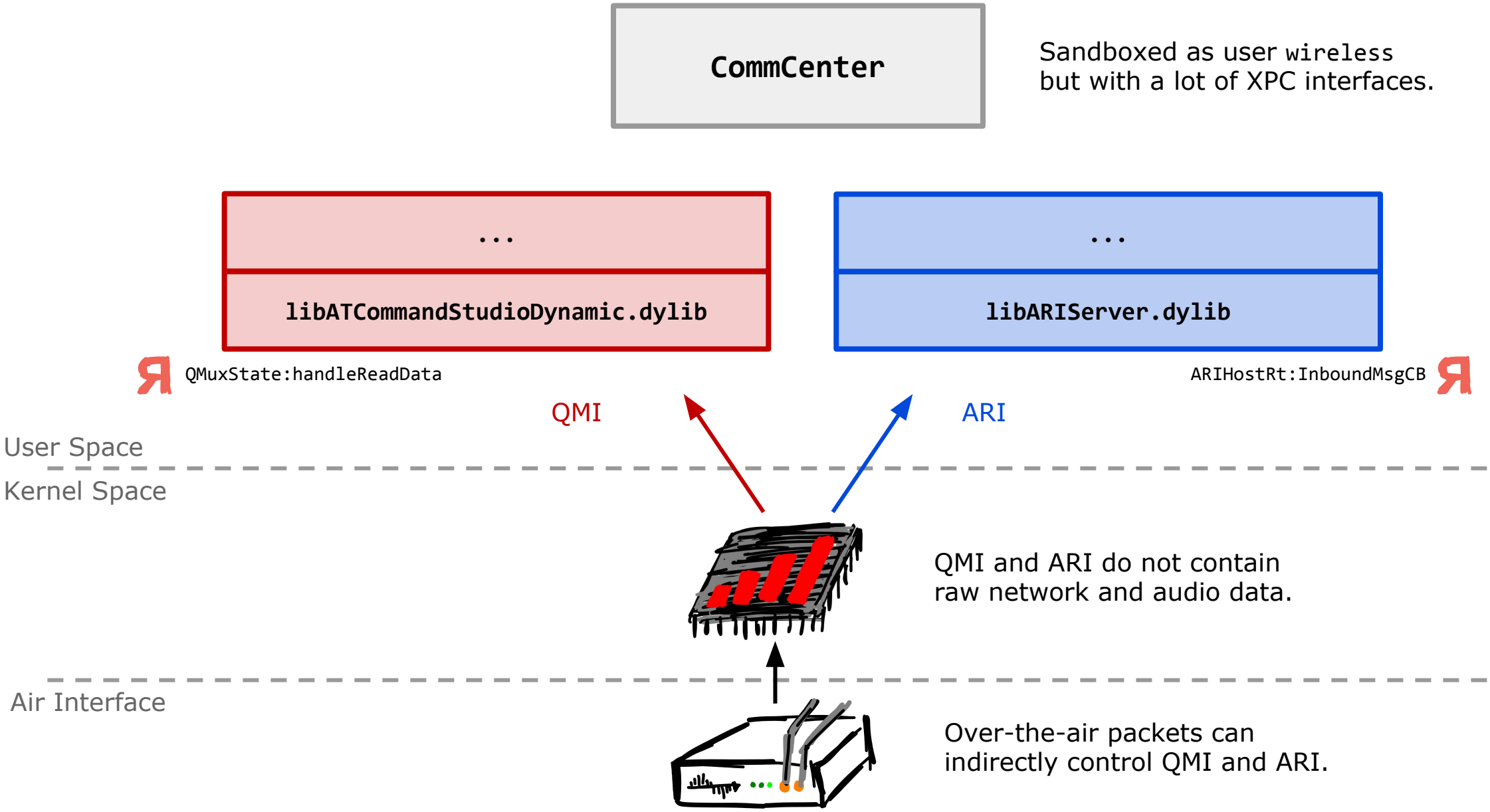
Qualcomm

- US devices
- Qualcomm MSM Interface (QMI) with some documentation
- Open-source implementations exist



~~Intel~~ Apple

- EU devices
- Apple Remote Invocation (ARI) has not been researched before
- Custom interface not used within Android etc.


CommCenter
Sandboxed as user wireless
but with a lot of XPC interfaces.
...
libATCommandStudioDynamic.dylib
...
libARIServer.dylib
QMuxState:handleReadData
ARIHostRt:InboundMsgCB
QMI
ARI
User Space
Kernel Space
QMI and ARI do not contain
raw network and audio data.
Air Interface
Over-the-air packets can
indirectly control QMI and ARI.

QMI

ASSERT ALL THE THINGS

Terminate `CommCenter` process on every invalid payload.

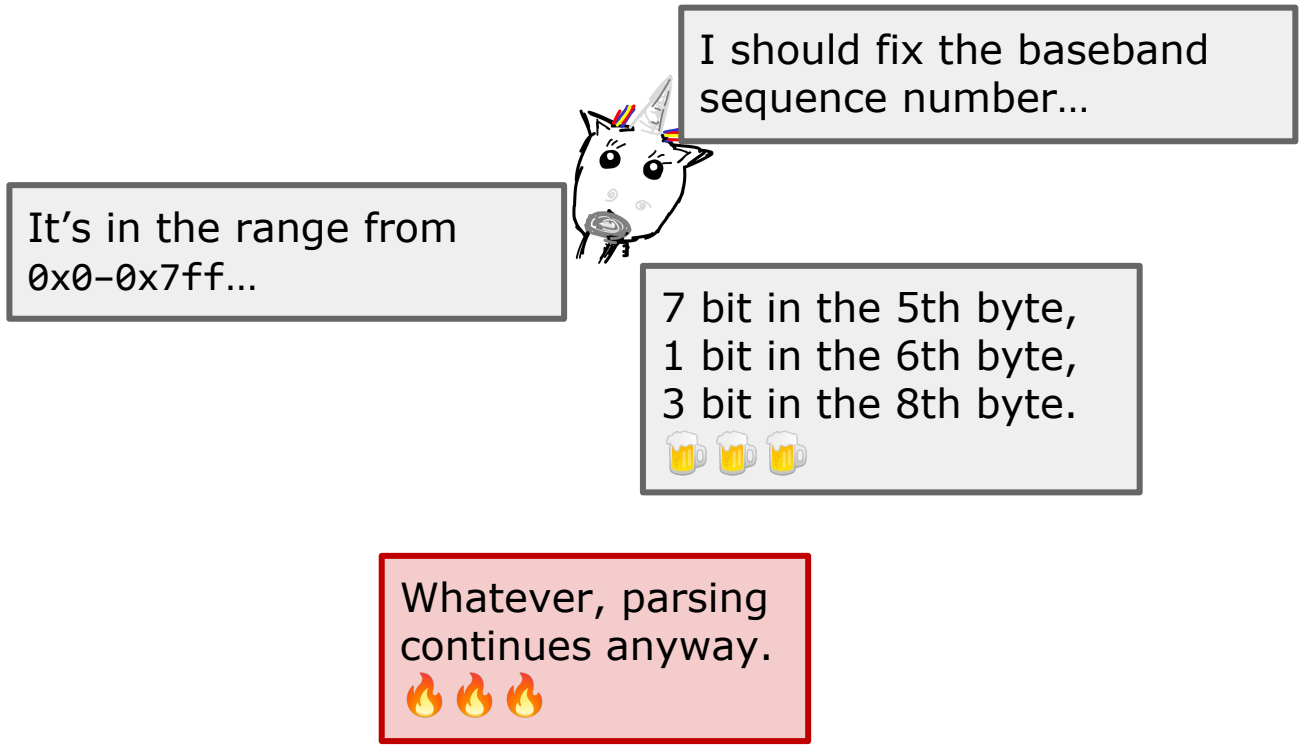
ARI

IT'S PARSING SHIP IT

Parsing is magic <3

Apple Remote Invocation Format

```
CommCenter(libARI.dylib)[1337] <Notice>:
 ari: (blk@ari_host_rt.cpp:1061) Missing expected BB.SEQ(0x42), got BB.SEQ(0x41)
```




JAILBREAK
ADVISORY
DON'T FUZZ THIS
TU Darmstadt

I know what I’m doing!!!

Actually I don’t know what I’m doing, so I only fuzz BB→AP.

Packet 1

Packet 2

Packet 3

Packet 4

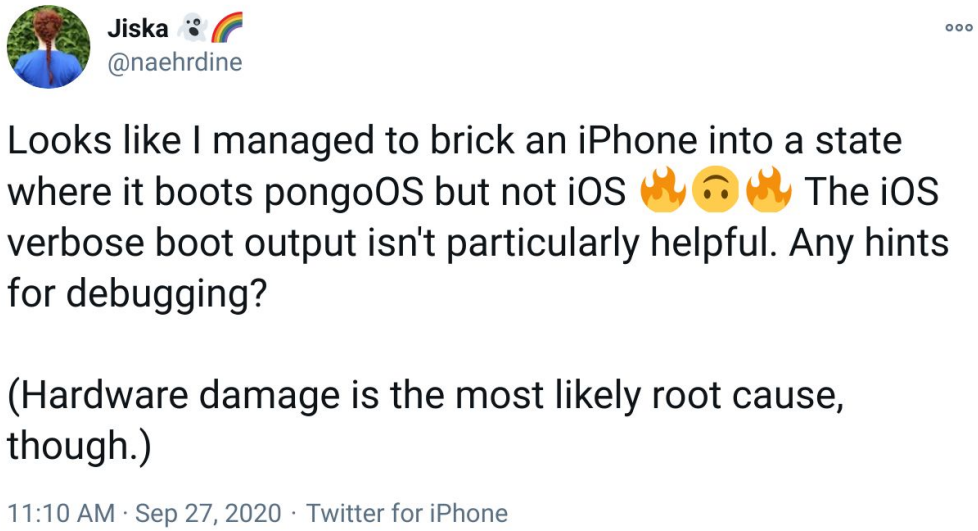
The AP might still reply with stuff that’s confusing to the BB.

F1: Calling for help…

iPhone 8, Qualcomm
Tends to boot loop for a couple of hours.

iPhone 8, Intel
Creates 500 MB of `istp` files during fuzzing, including ARI messages, every few minutes. Make a script that deletes these regularly!

All iPhones
Who am I? Where am I? Activate me, plz!



Activation Screenshot/Video

Flash SMS Video

When you see inconsistent UI like this, like black font on dark gray background, you know that this probably wasn’t tested for a while. So we really want to continue fuzzing it.

Am I locked?! Video

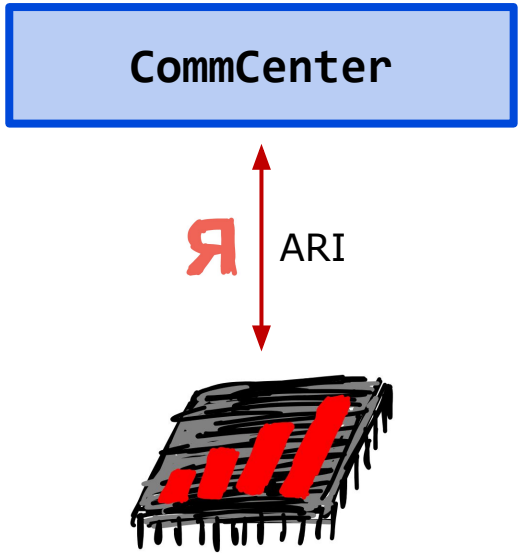
And another inconsistent UI on top of the lock screen xD


JAILBREAK
ADVISORY
DON'T FUZZ THIS
TU Darmstadt

Building Fuzzers

Fuzzer #1: Modifying Existing Packets

- High state awareness!
- Testing requires user interaction like phone calls, flight mode, SMS, MMS, ...
- Finds **fancy bugs** the other fuzzers couldn't repro yet 🐞



“Hey Apple,
I wrote this ugly 10 lines of code fuzzer Я
and see what it just found. Crash logs are
attached. Should be very simple to reproduce,
found all of this multiple times within only
3 days of fuzzing.”

Actually not that simple to reproduce!

- ARI & fuzzing script are so generic that this version runs on all iPhones with an Intel chip.
- **Pre-A12** (iPhone 7+8) crash at completely different locations than **A12+** (iPhone 11+SE2).
- Fuzzing results depend a lot on the **SIM**, even on a “passive” iPhone.

Video of AudioController Null Pointer

Let’s listen to a Null Pointer in the AudioController that was fixed in iOS 14.2 <3

Fuzzer #2 (ICEPicker 🧊⛏️): Local AFL++

```
git clone latest-toothpicker-alpha-🦄-very-very-💥-unstable
```

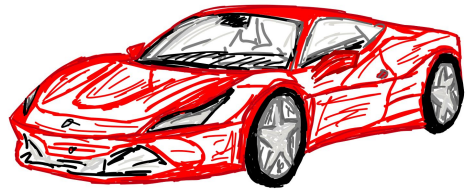
- AFL++ tries to shorten inputs & brute-force magic bytes
- No awareness/logging of packet order
- Faster than legacy ToothPicker (250 fcps on an iPhone 7)
- Speed drops in ARI when packets are invalid (<10 fcps)
- Ignores ARI host state (interface shut down)
- Did not find any new coverage for >6 h

- … wanted to recompile with different AFL++ settings but had my Xcode beta in a weird state.

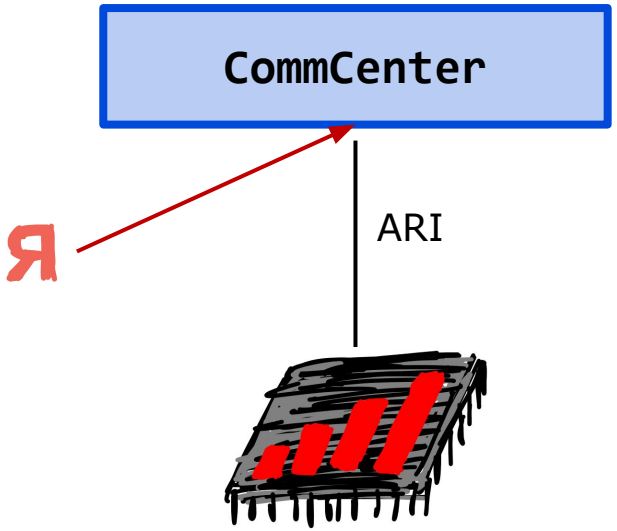
Uuuuh during fuzzer initialization it’s actually doing a replay that seems to work even across multiple iPhone versions.

Fuzzer #3: External Blind Corpus-based Injection

- Can’t reproduce all states & crashes…
- … but most crashes can be reproduced 💥



- No coverage. Blind injection. Suuuper fast! (400 fcps on iPhone 7)
- Can correct length field, sequence number, etc.
- Collect a good corpus. Use all the SIMs. Make all the calls…

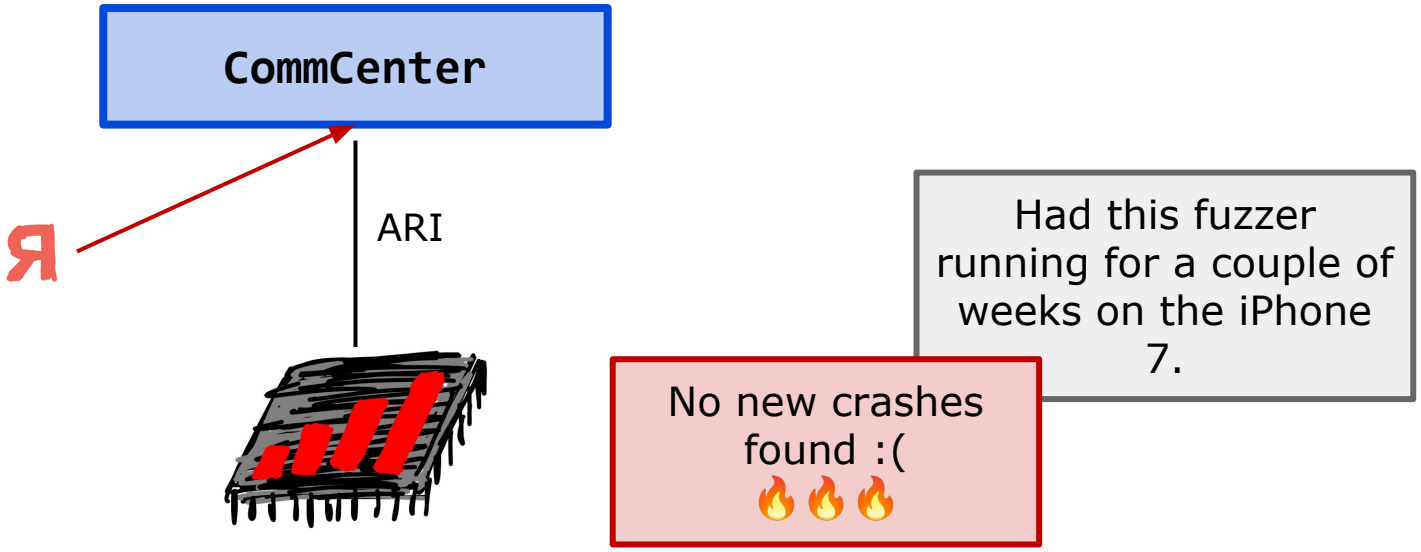


Had this fuzzer running for a couple of weeks on multiple iPhones in parallel.

Fuzzer #4: External `radamsa`

```
git clone publicly-released-toothpicker
```

- Finds new coverage.
- Only aware of single packets.
- Slooooooooooow. (10–25 fcps on iPhone 7)
- Can’t catch a lot of `CommCenter` crashes and then the whole fuzzer crashes. 💥



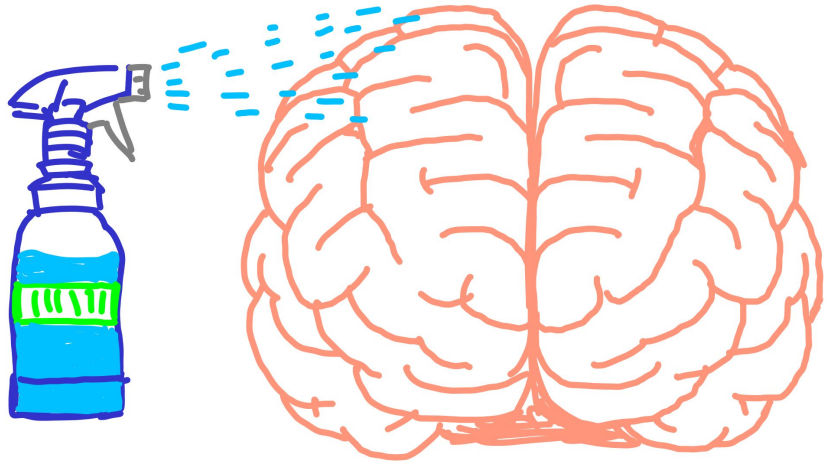
Video of Scrolling Phone Number

At least radamsa creates some smart mutations :)

Memory Sanitization

- `MallocStackLogging` framework exists on iOS.
- Got this running!
- Even when increasing `Jetsam` limits `CommCenter` is killed immediately. 💥

- No `libgmalloc` for iOS, just Xcode.
- Got one of these running on one of my iPhones.
- Smaller programs execute.
- `CommCenter` crashes during startup when parsing some config files… 💥



What the Fuzz...

gpsd 1/0/0

2/0/1 misd 4/0/1

mediaserverd 2/0/0

suggestd 1/0/0

WirelessRadioManagerd 1/1/1

3/2/0 locationd 5/1/0

9/3/0 CommCenter 42/14/4

Qualcomm Libraries

Intel Libraries

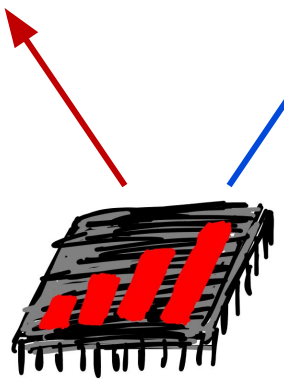
abmlite 1/0/0

1/0/0 sharingd

awdd 1/0/0

photoanalysisd 2/0/0

1/1/0 abm-helper



imagent 1/0/0

Individual crashes found:
Total / **Replayable** / **Fixed in iOS 14.3**

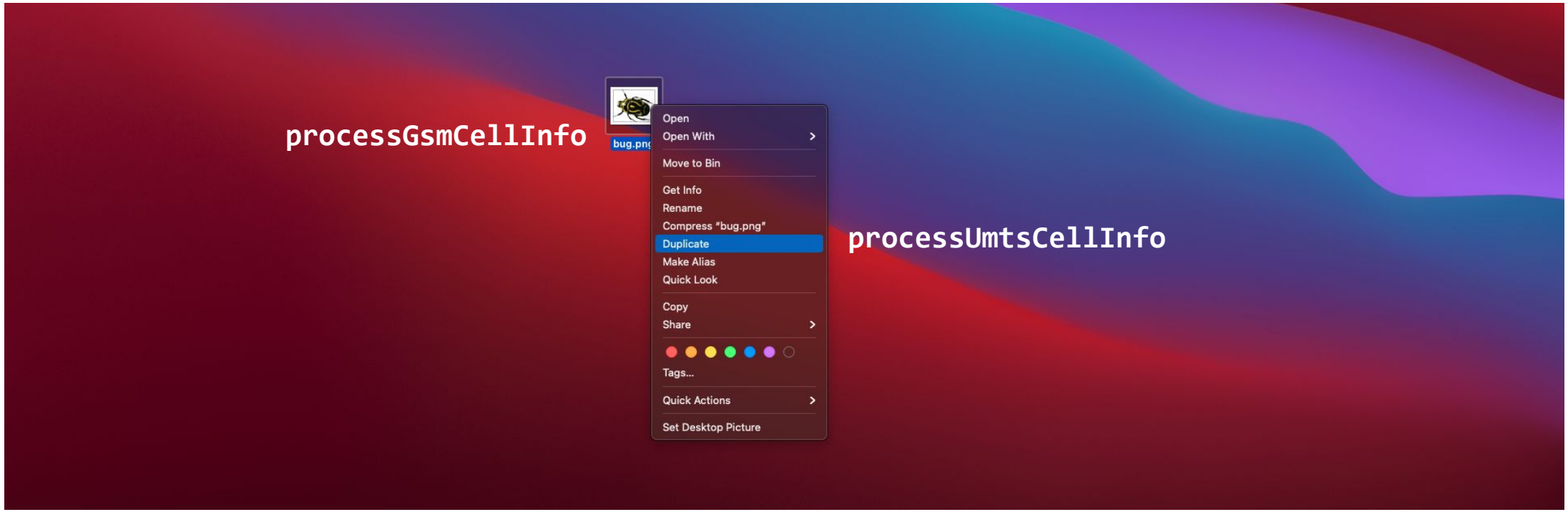
misd – Mobile Internet Sharing Daemon

```
Hardware Model:      iPhone10,4
Process:             misd [44]
Coalition:           com.apple.MobileInternetSharing [31]
OS Version:          iPhone OS 13.6 (17G68)

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Subtype: EXC_ARM_DA_ALIGN at 0x3e676e6972747344

Thread 1 name:  Dispatch queue: com.apple.main-thread
Thread 1 Crashed:
0   CoreFoundation                0x000000019a6613e8 CFRelease + 28
1   misd                          0x00000001048d7aa4 0x1048c8000 + 64164
2   CoreTelephony                 0x000000019efaebac invocation function for block in CTServerState::sendNotification_sync+ 486316
                                                     (CTEvent, __CFString const*, __CFDictionary const*) const + 32
3   libdispatch.dylib             0x000000019a44e9a8 _dispatch_call_block_and_release + 24
4   libdispatch.dylib             0x000000019a44f524 _dispatch_client_callout + 16
5   libdispatch.dylib             0x000000019a42cb3c _dispatch_lane_serial_drain$VARIANT$armv81 + 564
6   libdispatch.dylib             0x000000019a42d580 _dispatch_lane_invoke$VARIANT$armv81 + 448
7   libdispatch.dylib             0x000000019a4361c0 _dispatch_kevent_worker_thread + 1192
8   libsystem_pthread.dylib       0x000000019a4a0bac _pthread_wqthread + 328
9   libsystem_pthread.dylib       0x000000019a4a3740 start_wqthread + 8
```

> “We fixed this in a beta prior to your report.”

ARI `CellMonitor`

- Reported as part of the initial replacement fuzzer on August 24 2020.
- Doesn’t require user interaction or active SIM data plan.
- Didn’t test yet if triggerable over-the-air only via the chip…
- Fixed in iOS 14.2 🙂 … still 3 more `CellMonitor` crash-only bugs remaining 💥💥💥



More bugs to be fixed…

… but primarily stability improvements.

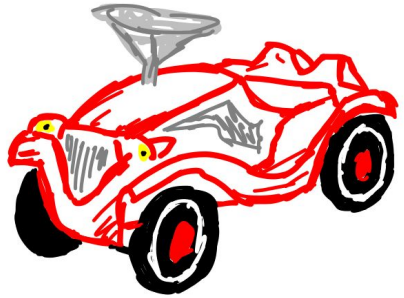
FRIDA Performance

How to replicate some of these bugs within a day…

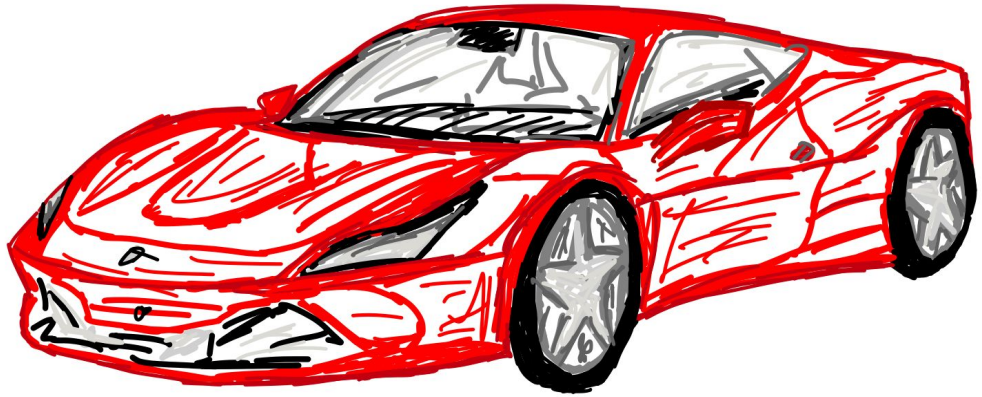
🔥 xxx hot iPhone research 🔥

On processing-intense payloads, fuzzing drains more power than my laptop’s USB port provides to the iPhone.

Identifying Bottlenecks



20 fcps



20000 fcps

Driving so fast that you can no longer read the traffic signs...

FЯIDA with External Script

JavaScript injected into iOS target process

```
// send a message to the Python script
send("some message");

// this can be called by the Python script
rpc.exports = {
    callme: function() {
        # do something
    };
};
```

JSON serialization & deserialization, USB

Python script running on Linux/macOS

```
def on_message(message, data):
   # handle message from script

script = frida_session.create_script("...")
script.on("message", on_message)
script.load()

# call function in script
script.exports.callme()
```

Serialization and deserialization via JSON also takes place if you use FЯIDA C bindings to inject JavaScript locally on the iPhone, but it seems to be faster than Python :)

Fuzzing with FЯIDA

JavaScript calling the ARI message handler

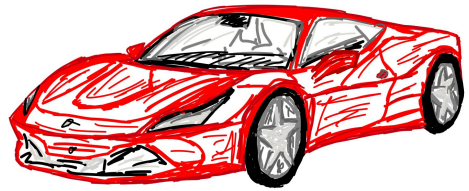
```
// define hook for ARI messages chip (BB) -> iOS (AP)
var InboundMsgCb_addr = Module.getExportByName('libARIServer.dylib', '_ZN9AriHostRt12InboundMsgCBEPhm');
var InboundMsgCb = new NativeFunction(InboundMsgCb_addr, "int64", ["pointer", "int64"]);

// optional hook customization
Interceptor.attach(InboundMsgCb_addr, {
    onEnter: function(args) {
        // fix ARI sequence number, collect basic block coverage, etc.
    }
});

// call the target once
InboundMsgCb(payload, length);
```

- Fuzz via external Python script by calling the RPC exports: 500 fcps
- Fuzz within the same JavaScript in a loop: 22000 fcps 🔥🔥🔥

On an iPhone 8 on iOS 14.3b without collecting coverage, SMS payload.

🔥 Statistics 🔥

Scenario	iPhone 7	iPhone 8
Local JavaScript SMS bit flipper	**17000 fcps**	22000 fcps
Local JavaScript SMS bit flipper with additional array copy	11000 fcps	13000 fcps
Local JavaScript SMS bit flipper printing each payload in hex while fuzzing	800 fcps	1500 fcps
Local JavaScript SMS bit flipper collecting basic block coverage	**250 fcps**	—
Remote Python SMS injection	400 fcps	500 fcps
Remote Python SMS injection collecting basic block coverage	100 fcps	—
ToothPicker/Frizzer with `radamsa` mutator	**20 fcps**	—



- 15–22× slowdown due to hex printing
- ~68× slowdown due to coverage collection

Deleting SMS Video

...no longer able to delete 200k SMS /o\

Q&A

Twitter: @naehrdine, @seemoolab

jiska@bluetooth.lol