Virtualizing IoT with Code Coverage Guided Fuzzing

HITB2018DXB, November 2018

NGUYEN Anh Quynh, aquynh -at- gmail.com
KaiJern LAU, kj -at- theshepherd.io
About NGUYEN Anh Quynh

- Nanyang Technological University, Singapore
- PhD in Computer Science
- Operating System, Virtual Machine, Binary analysis, etc
- Usenix, ACM, IEEE, LNCS, etc
- Blackhat USA/EU/Asia, DEFCON, Recon, HackInTheBox, Syscan, etc
- Capstone disassembler: http://capstone-engine.org
- Unicorn emulator: http://unicorn-engine.org
- Keystone assembler: http://keystone-engine.org
About kaijern.xwings.L

**Founder**
Stays in the lab 24/7 by hoping making the world a better place
- IoT Research
- Blockchain Research
- Fun Security Research

**Badge Maker**
Electronic fan boy, making toys from hacker to hacker
- Reversing Binary
- Reversing IoT Devices
- Part Time CTF player

**Broker**
Crew since 2008, from Kuala Lumpur till now AMS, SG, BEIJING and DXB
- 2006 (ctf) till end of time
- Core Crew
- Review Board

2005, HITB CTF, Malaysia, First Place /w 20+ Intl. Team
- 2010, Hack In The Box, Malaysia, Speaker
- 2012, Codegate, Korean, Speaker
- 2015, VXRL, Hong Kong, Speaker
- 2015, HITCON Pre Qual, Taiwan, Top 10 /w 4K+ Intl. Team
- 2016, Codegate PreQual, Korean, Top 5 /w 3K+ Intl. Team
- 2016, Qcon, Beijing, Speaker
- 2016, Kcon, Beijing, Speaker
- 2016, Intl. Antivirus Conference, Tianjin, Speaker
- 2017, Kcon, Beijing, Trainer
- 2017, DC852, Hong Kong, Speaker
- 2018, KCON, Beijing, Trainer
- 2018, DC010, Beijing, Speaker
- 2018, Brucon, Brussel, Speaker
- 2018, H2HC, San Paolo, Brazil, Speaker
- 2018, HITB, Beijing/Dubai, Speaker
- 2018, beVX, Hong Kong, Speaker
- MacOS SMC, Buffer Overflow, suid
- GDB, PE File Parser Buffer Overflow
- Metasploit Module, Snort Back Orifice
- Linux ASLR bypass, Return to EDX
Agenda

Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
Fuzzing

- Automated software testing technique to find bugs
  - Feed craft input data to the program under test
  - Monitor for errors like crash/hang/memory leaking
  - Focus more on exploitable errors like memory corruption, info leaking
  - Maximize code coverage to find bugs
  - Blackbox fuzzing
  - Whitebox fuzzing
  - Graybox fuzzing, or **Coverage Guided Fuzzing**
Coverage-guided Fuzzer

- Instrument target binary to collect coverage info
- Mutate the input to maximize the coverage
- Repeat above steps to find bugs
  - Proved to be very effective
    - Easier to use/setup & found a lot of bugs
  - Trending in fuzzing technology
    - American Fuzzy Lop (AFL) really changed the game
Guided Fuzzer for Embedded

- Guided fuzzer was introduced for powerful PC systems
- Bring over to embedded world?
  - No support for introducing new tools
  - Not open source
  - Lack support for embedded hardware
Issues

**Restricted System**

- Without built-in shell access for user interaction
- Without development facilities required for building new tools
  - Compiler
  - Debugger
  - Analysis tools

**Closed System**

- Binary only - without source code
- Existing guided fuzzers rely on source code available
- Source code is needed for branch instrumentation to feedback fuzzing progress
- Emulation such as QEMU mode support in AFL is slow & limited in capability
- Same issue for other tools based on Dynamic Binary Instrumentation

**Lack Support for Embedded**

- Most fuzzers are built for X86 only
- Embedded systems based on Arm, Arm64, Mips, PPC
- Existing DBIs are poor for non-X86 CPU
  - Pin: Intel only
  - DynamoRio: experimental support for Arm
Agenda

Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
The SoC

- Scale Down from PC
- System on Chip
- A chip with all the PCI-e slot and card in it
- Pinout to different parts
- Wifi, Lan, Bluetooth and etc
- Low power device
Hardware + GNU Command
also
love hardware and not only hardware hacking
Lets Get Started
Device Limited Bug

If all_model_one_firmware

In The Beginning:
We Need Firmware
Getting Firmware
Firmware and Hardware

Extract From Flash, Extract From APK, Traffic Sniffing or Just Download

Technically 1. Download 2. Patch with Backdoor 3. Flash 4. pwned

If we need more?
1. RCE 2. Fuzz
The Easy Way
Complete Kit to Success

**MIPS**
How Many Dev Board

**ARM**
Classic LIBC Issue

**AARCH64**

Hardware is not "downgradable"
Assembly Instruction Compatibility

```
get> get config context.layout "code stack"
get> Break *0x8001004c
Breakpoint 1 at 0x8103c
get> run
Starting program: /home/azeria/exp/stack

0x010408 <main+0> sub sp, sp, r11
0x010410 <main+12> str r6, [r11, 8-16]
0x010412 <main+16> str r11, [r11, 412]
0x01042c <main+20> mov r0, r11
0x010438 <main+24> mov r0, r3
0x01043c <main+28> b 0x1024 <getopt>

0x1003c <main+30> mov r0, r3
0x100440 <main+36> mov sp, r11, #4
0x100444 <main+40> pop (r11, pc)
0x10044c <__libc_csu_init+0> push (r3, r4, r5, r6, r7, r8, r9, lr)
0x100450 <__libc_csu_init+4> mov r7, r0

```

---

**ARM**

**AARCH64**
Current Work Around
QEMU-Static is good for binary execution without additional software or hardware interection.
Current Primitive Firmware Emulation

Leaving squashfs and going into a unknown world

Its not easy after 2016
Why Firmware Emulation
More Resources = More Power

Processor
Normally 1-2 Core

RAM
Normally 256MB/512MB

FLASH
Normally 8MB/16MB/32MB/256MB

Most Important, we got apt-get
Objectives
Only One Process with Interaction

most of the devices comes with one big binary

Hunt for the one that spawn listener port
Booting Up
Distro and Kernel Mix and Match

argument: running new or old distro + kernel

script to boot arm

```bash
#!/bin/bash
sudo tuncitl -d tap0
sudo screen -dm /opt/qemu/bin/qemu-system-arm -n 2048 -M Virt -cpu cortex-a15 -spi cpus=4,maxcpu=4 -kernel boot.stretch.armhf/vmlinux-4.9.0-6.6.1-qcow2 -append 'nographic'
debian -udeb devlin-stretch-armhf/vmlinux-4.9.0-6.6.1-qcow2 -device virtio-blk-device,driver=sd
sudo sysct1 -w net.ipv4.ip_forward=1
echo "Stopping firewall and allowing everyone..."
sudo iptables -F
sudo iptables -t nat -F
sudo iptables -t mangle -F
sudo iptables -t arp -F
sudo iptables -t lattice -F
sudo iptables -t nat -X
sudo iptables -t mangle -X
sudo iptables -t lattice -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
sudo iptables -t nat -A POSTROUTING -o ens33 -j MASQUERADE
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 10.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 10.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 10.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 10.253.253.10:88
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.10:88
echo "Boot the VM, etc 10 seconds"
sleep 10
sudo ifconfig tap0 19.253.253.254 netmask 255.255.255.0
```

script to boot mips

```bash
#!/bin/bash
sudo screen -dm /opt/qemu/bin/qemu-system-mipsel -n 512 -M malta -kernel boot.stretch.mipsel/vmlinux-4.9.0-4-kc-malta -initrd boot.stretch.mipsel/initrd.img-4.9.0-4-kc-malta -append "root=/dev/vda1 net.ifnames=0 blockdevname=区块mfs"
debian -udeb devlin-stretch.mipsel/qcow2 -device virtio-blk-device,driver=sd
sudo sysct1 -w net.ipv4.ip_forward=1
echo "Stopping firewall and allowing everyone..."
sudo iptables -F
sudo iptables -t nat -F
sudo iptables -t mangle -F
sudo iptables -t lattice -F
sudo iptables -t nat -X
sudo iptables -t mangle -X
sudo iptables -t lattice -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
sudo iptables -t nat -A POSTROUTING -o ens33 -j MASQUERADE
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 19.253.253.11:22
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 10.253.253.11:22
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 19.253.253.11:22
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 10.253.253.11:22
sudo iptables -t nat -A POSTROUTING -o ens33 -j DNAT --to-destination 19.253.253.11:22
sudo iptables -t nat -A POSTROUTING -o eth0 -j DNAT --to-destination 10.253.253.11:22
echo "Boot the VM, etc 10 seconds"
sleep 10
sudo ifconfig tap0 19.253.253.254 netmask 255.255.255.0
```
chroot
Easy Way Out, chroot

chroot is easy (still hardware dependent), but we will have issue with tools

Running without chroot
Classic Case: File Not Found
The File Missing Trick

We Missed You

```bash
cdhin("/")
execve("/bin/bash", ["/bin/bash", "-i"], 0xffffca14f650 /* 18 vars */) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/lib/aarch64-linux-gnu/charset.alias", O_RDONLY|O_NOFOLLOW) = -1 ENOENT (No such file or directory)
write(2, "chroot: ", 8chroot: )
write(2, "failed to run command "/bin/bash"", 33failed to run command "/bin/bash") = 33
write(2, ": No such file or directory", 27: No such file or directory) = 27
write(2, "\n", 1
) = 1
close(1)
```

We found you

```bash
root@rpi3:/opt/
```

```bash
./lib64# file ./bin/bash
./bin/bash: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter 
/lib64/ld-linux-aarch64.so.1, for GNU/Linux 3.14.0, BuildID[sha1]=22e2854c58b1814825b95cba103ac6580 
371f5b0, stripped
```
The missing .SO and binary Issue
Out from chroot, we need feeding

Feeding all the required so and binary with “ln –s”
Out from chroot, we need feeding

bash-3.2# /usr/bin/appmainprog
<appmain>******************************************************
<appmain>child process id is 3931
<appmain>Appcliation Init Begin
<appmain>Audio Mas process Init
[Aud][PPC] AudioPPControl constructor
[Aud][PPC] AudioPPControl getInstanceOf
[Aud][PPC] AudioPPControl freeInstance
[Aud][PPC] AudioPPControl destructor
[Aud][PPC][deInit] PPC deinit begin.
[Aud][PPC][ppcStructUnalloc] ppc_destroy_info begin.
Segmentation fault
bash-3.2#

Classical file not found error

“segfault” without clear error. strace come to rescue
The Secretive NVRAM
Dark side of NVRAM

Relationship between main binary is so intimate, but in actual fact. Is just a hit and run
Dark Side of NVRAM

Dark Side of the main process, we ignore and can't to next step

Relationship between main binary is so intimate, but in actual fact, it's just a hit and run
A fake NVRAM

IF interactor is the medium, can we fake it?

reply with nvram info
A fake NVRAM

If interactor is the medium, can we fake it?

Custom Interactor
The bridge trick

The switch looking device
Wireless Device
Faking wpa_supplicant

-making eth0 looks like wlan0 works too
Everything Things Else Fail
BL, BNE, BEQ and friends

Original BIN

```
LDR R0, [R11,#fd]: fd
MOV R1, #6: level
MOV R2, #4: optname
BL setsockopt
LDR R0, [R11,#fd]: fd
MOV R1, R2: cmd
MOV R2, R1

MOV R3, R0
CPN R3, R1
BEQ loc_1BE88
```
Issues

> Binary only - without source code
  > Existing guided fuzzers rely on source code available
  > Source code is needed for branch instrumentation to feedback fuzzing progress
  > Emulation such as QEMU mode support in AFL is slow & limited in capability
  > Same issue for other tools based on Dynamic Binary Instrumentation

> Without built-in shell access for user interaction
> Without development facilities required for building new tools
  > Compiler
  > Debugger
  > Analysis tools

> Most fuzzers are built for X86 only
  > Embedded systems based on Arm, Arm64, Mips, PPC
  > Existing DBIs are poor for non-X86 CPU
  > Pin: Intel only
  > DynamoRIO: experimental support for Arm

Firmware Emulation Closed System Lack Support for Embedded
Dynamic Binary Instrumentation (DBI)

**Definition**

- A method of analyzing a binary application at runtime through injection of instrumentation code.
  - Extra code executed as a part of original instruction stream
  - No change to the original behavior
- Framework to build apps on top of it

**Applications**

- Code tracing/logging
- Debugging
- Profiling
- Security enhancement/mitigation
DBI Illustration

Original code

Inline instrumentation
DBI Techniques

- **Just-in-Time translation**
  - Transparently translate & execute code at runtime
    - Perform on IR: Valgrind
    - Perform directly on native code: DynamoRio
  - Better control on code executed
  - Heavy, super complicated in design & implementation

- **Hooking**
  - Lightweight, much simpler to design & implement
  - Less control on code executed & need to know in advance where to instrument
Hooking Mechanisms - Inline

- Inline code injection
  - Put instrumented code inline with original code
  - Can instrument anywhere & unlimited in extra code injected
  - Require complicated code rewrite
Hooking Mechanisms - Detour

- Detour injection
  - Branch to external instrumentation code
    - User-defined **CALLBACK** as instrumented code
    - **TRAMPOLINE** memory as a step-stone buffer
  - Limited on where to hook
    - Basic block too small?
  - Easier to design & implement

![Diagram of Detour injection](image)
Detour Injection Mechanisms

- Branch from original instruction to instrumented code
- Branch to trampoline, or directly to callback
  - Jump-trampoline technique
  - Jump-callback technique
  - Call-trampoline technique
  - Call-callback technique
Jump-trampoline Technique

- instruction
  ...
- save context
  JUMP
  ...
- restore context
  save context
  CALL
  restore context
  reloc instruction
  JUMP
- callback
Jump-callback Technique

- Instruction
- Save context
- Jump
- Restore context
- Reloc instruction
- Ret

Original vs Instrumented

Callback
Call-trampoline Technique

- Instruction
  - ...
- Original
- Instrumented
- Trampoline
  - Save context
  - Call
  - Restore context
  - Reloc instruction
  - RET
  - Callback
Call-callback Technique

Original

Instrumented

Instruction

Save context

CALL

ReLoc instruction

RET

Callback
Problems of Existing DBI

- Limited on platform support
- Limited on architecture support
- Limited on instrumentation techniques
- Limited on optimization
SKORPIO Framework

- Low level framework to build applications on top
  - App typically designed as dynamic libraries (DLL/SO/DYLIB)
- Cross-platform-architecture
  - Windows, MacOS, Linux, BSD, etc
  - X86, Arm, Arm64, Mips, Sparc, PowerPC
- Allow all kind of instrumentations
  - Arbitrary address, in any privilege level
- Designed to be easy to use, but support all kind of optimization
  - Super fast (100x) compared to other frameworks, with proper setup
- Support static instrumentation, too!
SKORPIO Architecture

Making eth0 looks like wlan0 works too
Cross Platform - Memory

- Thin layer to abstract away platform details
- Different OS supported in separate plugin
  - Posix vs Windows
- Trampoline buffer
  - Allocate memory: malloc() vs VirtualAlloc()
  - Memory privilege RWX: mprotect() vs VirtualAlloc()
  - Trampoline buffer as close as possible to code to reduce branch distance
- Patch code in memory
  - Unprotect -> Patch -> Re-protect
  - mprotect() vs VirtualProtect()
Save memory/registers modified by initial branch & callback

Keep the code size as small as possible

Depend on architecture + mode
  - X86-32: PUSHAD; PUSHFD & POPFD; POPAD
  - X86-64 & other CPUs: no simple instruction to save all registers :-(
    - Calling convention: cdecl, optlink, pascal, stdcall, fastcall, safecall, thiscall, vectorcall, Borland, Watcom
    - SystemV ABI vs Windows ABI

Special API to customize code to save/restore context
Cross Architecture - Callback argument

- Pass user argument to user-defined callback
- Depend on architecture + mode & calling convention
  - SysV/Windows x86-32 vs x86-64
    - Windows: cdecl, optlink, pascal, stdcall, fastcall, safecall, thiscall, vectorcall, Borland, Watcom
  - X86-64: "mov rcx, <value>" or "mov rdi, <value>". Encoding depends on data value
  - Arm: "ldr r0, [pc, 0]; b .+8; <4-byte-value>"
  - Arm64: "movz x0, <lo16>; movk x0, <hi16>, lsl 16"
  - Mips: "li $a0, <value>"
  - PPC: "lis %r3, <hi16>; ori %r3, %r3, <lo16>"
Distance from hooking place to callback cause nightmare :-(

- Some architectures have no explicit support for far branching
  - X86-64 JUMP: "push <addr>; ret" or "push 0; mov dword ptr [rsp+4], <addr>" or "jmp [rip]"
  - X86-64 CALL: "push <next-addr>; push <target>; ret"
  - Arm JUMP: "b <addr>" or "ldr pc, [pc, #-4]"
  - Arm CALL: "bl <addr>" or "add lr, pc, #4; ldr pc, [pc, #-4]"
  - Arm64 JUMP: "b <addr>" or "ldr x16, .+8; br x16"
  - Arm64 CALL: "bl <addr>" or "ldr x16, .+12; blr x16; b .+12"
  - Mips JUMP: "li $t0, <addr>; jr $t0"
  - Mips CALL: "li $t0, <addr>; move $t9, $t0; jalr $t0"
  - Sparc JUMP: "set <addr>, %l4; jmp %l4; nop"
  - Sparc CALL: "set <addr>, %l4; call %l4; nop"
Cross Architecture - Branch for PPC

- PPC has no far jump instruction :-(
  - copy LR to r23, save target address to r24, then copy to LR for BLR
  - restore LR from r23 after jumping back from trampoline
  - "mflr %r23; lis %r24, <hi16>; ori %r24, %r24, <lo16>; mtlr %r24; blr"

- PPC has no far call instruction :-(
  - save r24 with target address, then copy r24 to LR
  - point r24 to instruction after BLR, so later BLR go back there from callback
  - "lis %r24, <target-hi16>; ori %r24, %r24, <target-lo16>; mtlr %r24; lis %r24, <ret-hi16>; ori %r24, %r24, <ret-lo16>; blr"

```c
SK_INLINE_NO static void bbb_hook(size_t v)
{
    // restore LR from R24
    __asm__("mtlr %r24");
    printf("== in callback, userdata = %zu\n", v);
    return;
}
```
Scratch registers used in initial branching

- Arm64, Mips, Sparc & PPC do not allow branch to indirect target in memory
- Calculate branch target, or used as branch target
- Need scratch register(s) that are unused in local context
  - Specified by user via API, or discovered automatically by engine
• Code patching need to be reflected in i-cache
• Depend on architecture
  ▶ X86: no need
  ▶ Arm, Arm64, Mips, PowrPC, Sparc: special syscalls/instructions to flush/invalidate i-cache
  ▶ Linux/GCC has special function: cacheflush(begin, end)
Code Boundary & Relocation

- Need to extract instructions overwritten at instrumentation point
  - Determine instruction boundary for X86
  - Use Capstone disassembler
- Need to rewrite instructions to work at relocated place (trampoline)
  - Relative instructions (branch, memory access)
  - Use Capstone disassembler to detect instruction type
  - Use Keystone assembler to recompile
Avoid overflow to next basic block
  ▶ Analysis to detect if basic block is too small for patching
Reduce number of registers saved before callback
Registers to be chosen as scratch registers
API to setup calling convention
User-defined callback
User-defined trampoline
User-defined scratch registers
User-defined save-restore context
User-defined code to setup callback ars
Patch hooks in batch, or individual
User decide when to write/unwrite memory protect
Sample for Skorpio engine

--- Original code
BBB code = 0x400ca0, callback = 0x400c80

Hook info:
Hook type: 2
Hook address: 0x400ca0
Hook callback: 0x400c80
Hook user_data: 0x7b
Hook trampoline addr: 0x7f1aa7911000
Hook trampoline size: 86
Hook trampoline code:
505351525755541504151241549c48c7c77b0000006a00c70424321091a7c74424041a7f00006a00c70424800c4000c39d415c415a41594158d5e5f5a595b584883ec08b9800c4000baa0c400068ae0c4000c3
Patch size: 14
Patched code: ff25000000000001091a71a7f0000
Hook original code size: 14
Hook original code: 4883ec08b9800c4000baa0c4000

--- Functions with instrumentation now
== inside callback, userdata = 123
BBB code = 0x400ca0, callback = 0x400c80

--- Restored original code, now without instrumentation
BBB code = 0x400ca0, callback = 0x400c80
Agenda

Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
Issues

24K Core Architecture

Firmware Emulation

Skorpio DBI

Lack Support for Embedded

- Binary only - without source code
  - Existing guided fuzzers rely on source code available
    - Source code is needed for branch instrumentation to feedback fuzzing progress
    - Emulation such as QEMU mode support in AFL is slow & limited in capability
    - Same issue for other tools based on Dynamic Binary Instrumentation

- Without built-in shell access for user interaction
- Without development facilities required for building new tools
  - Compiler
  - Debugger
  - Analysis tools

- Most fuzzers are built for X86 only
  - Embedded systems based on Arm, Arm64, Mips, PPC

- Existing DBIs are poor for non-X86 CPU
  - Pin: Intel only
  - DynamoRio: experimental support for Arm
Fuzzer Features

- Built on top of AFL fuzzer
- Support closed-source binary for all platforms & architectures
  - Use Skorpio DBI to support all popular embedded CPUs
- Support selective binary fuzzing
- Support persistent mode
- Other enhanced techniques
  - Symbolic Execution to guide fuzzer forward
  - Combine with static analysis for smarter/deeper penetration
Fuzzer Design

- Pure software-based
- Cross-platform/architecture
  - Native compiled on embedded systems
- Binary support
  - Full & selected binary fuzzing + Persistent mode
- Fast & stable
  - Stable & support all kind of binaries
  - Order of magnitude faster than DBI/Emulation approaches
Fuzzer Implementation

- Reuse AFL fuzzer - without changing its core design
- AFL-compatible instrumentation
- Static analysis on target binary beforehand
- Inject Skorpio hooks into selected area in target binary at runtime
- At runtime, hook callbacks update execution context in shared memory, like how source-code based instrumentation do
- Near native execution speed, ASLR / threading compatible
Fuzzer Instrumentation

- LD_PRELOAD to dynamically inject instrumentation
  - Take place before main program runs
  - Linux: shared object file (.so)
- Inject hooks at SO initialisation time
  - Can be 10k hooks, so must do as quickly as possible
- Inject forkserver at program entry-point, or at user-defined point
Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
*bug disclosed in geekpwn 2018, shanghai*
Web Cam Buffer Overflow

Pre Authentication Bug

Buffer Overflow

Address Overwritten

Debug is almost Impossible *watchdog*

Emulation comes into play
IoT with UDP Access

Web Cam with Motor
Command Execution Injection

Chinese based WiFi Router
Agenda

Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
Issues

- Binary only - without source code
  - Existing guided fuzzers rely on source code available
  - Source code is needed for branch instrumentation to feedback fuzzing progress
  - Emulation such as QEMU mode support in AFL is slow & limited in capability
  - Same issue for other tools based on Dynamic Binary Instrumentation

- Without built-in shell access for user interaction
- Without development facilities required for building new tools
  - Compiler
  - Debugger
  - Analysis tools

- Most fuzzers are built for X86 only
  - Embedded systems based on Arm, Arm64, Mips, PPC
  - Existing DBIs are poor for non-X86 CPU
    - Pin: Intel only
    - DynamoRio: experimental support for Arm

Firmware Emulation

24K Core Architecture

- Without built-in shell access for user interaction
- Without development facilities required for building new tools
  - Compiler
  - Debugger
  - Analysis tools

Skorpio DBI

- Binary only - without source code
- Existing guided fuzzers rely on source code available
- Source code is needed for branch instrumentation to feedback fuzzing progress
- Emulation such as QEMU mode support in AFL is slow & limited in capability
- Same issue for other tools based on Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

- Most fuzzers are built for X86 only
  - Embedded systems based on Arm, Arm64, Mips, PPC
  - Existing DBIs are poor for non-X86 CPU
    - Pin: Intel only
    - DynamoRio: experimental support for Arm
Conclusions

- We built our smart guided fuzzer for embedded systems
  - Emulate firmware
  - Cross platforms/architectures
  - Binary-only support
  - Fast + stable
  - Found real impactful bugs in complicated software
Agenda

Coverage Guided Fuzzer vs Embedded Systems

Emulating Firmware

Skorpio Dynamic Binary Instrumentation

Guided Fuzzer for Embedded

DEMO

Conclusions

Secret Menu
Capstone 4.0

- Started 2013
- ~160 Contributors
- World Class Disassembler, Industrial Standard
- Used by almost all reverse engineering tools
- Foundation for 400+ opensource/public projects
- Current Release 3.0.5
- In version 3 since 2014
- Dec 2018, Capstone 4.0
- Why take us so long
Questions

Virtualizing IoT
with Code Coverage Guided Fuzzing

NGUYEN Anh Quynh, aquynh -at- gmail.com
KaiJern LAU, kj -at- theshepherd.io