

Rehosting and coverage guided fuzzing of embedded network services

Marc Heuse <marc@srlabs.de>



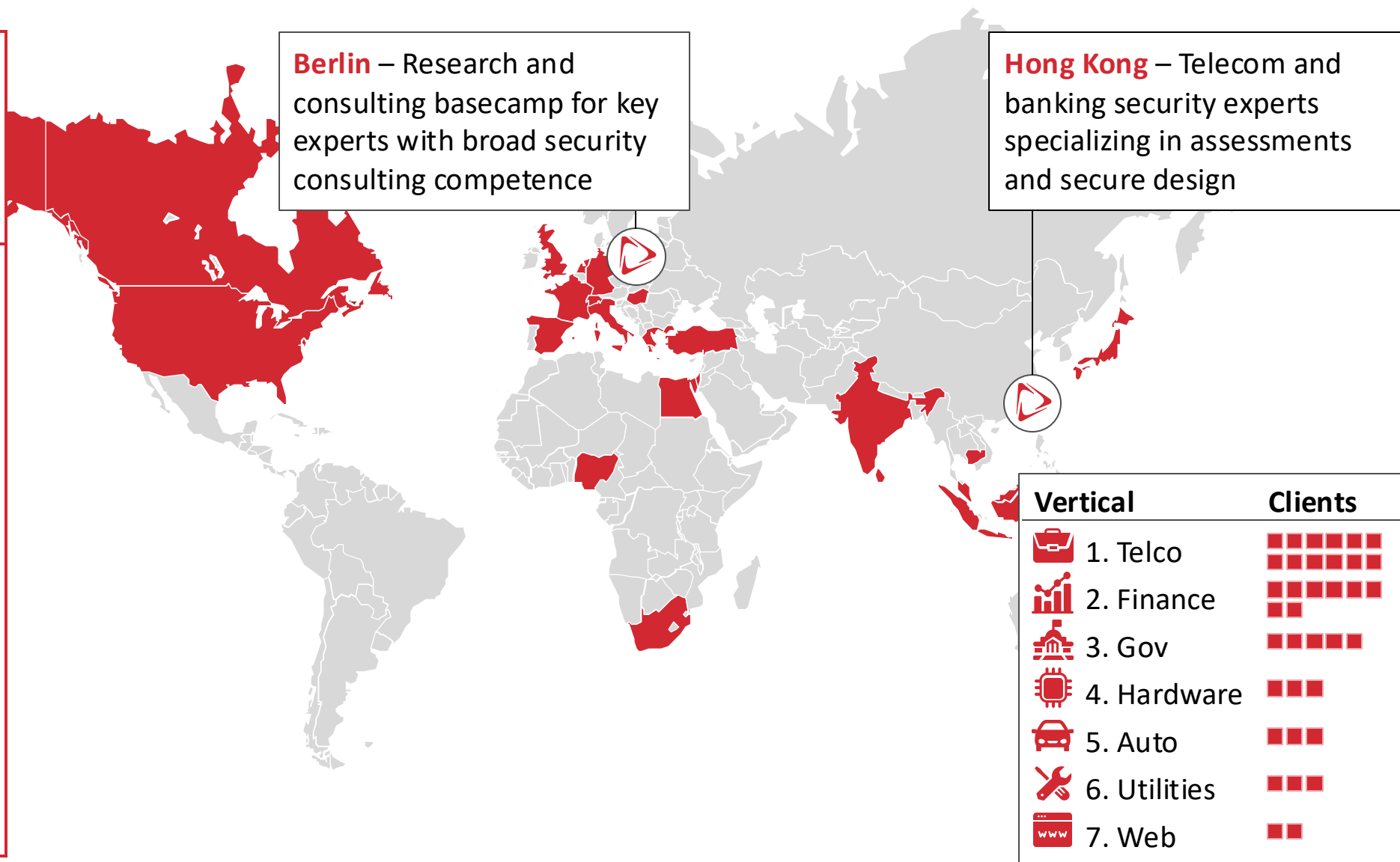
Security
Research
Labs















I lead complex code audits at Security Research Labs



Marc
“vanHauser”
Heuse
Team Lead

- 25 years IT security industry in-depth experience
- Author of AFL++, hydra, thc-ipv6, amap, THC-Scan, SuSEfirewall2, etc.
- Focus on complex system analysis, embedded device analysis, reverse engineering, fuzzing, source code audits



Vertical		Clients
 1. Telco		
 2. Finance		
 3. Gov		
 4. Hardware		
 5. Auto		
 6. Utilities		
 7. Web		

Agenda

-
- A** Workshop introduction
 - B** Workshop preparation
 - C** From firmware to fuzzing
-

Continuously providing randomly malformed data to a target
is a highly efficient way to find bugs

Definition

fuzz testing

noun /fʌz 'tɛstɪŋ/

A software testing technique that involves automatically feeding a program with large volumes of invalid, unexpected, or random data (“fuzz”) to detect bugs, crashes, memory leaks, or security vulnerabilities.

Impact

Google’s free fuzzing platform OSS-Fuzz has found

- more than 8,800 vulnerabilities and
- 28,000 bugs

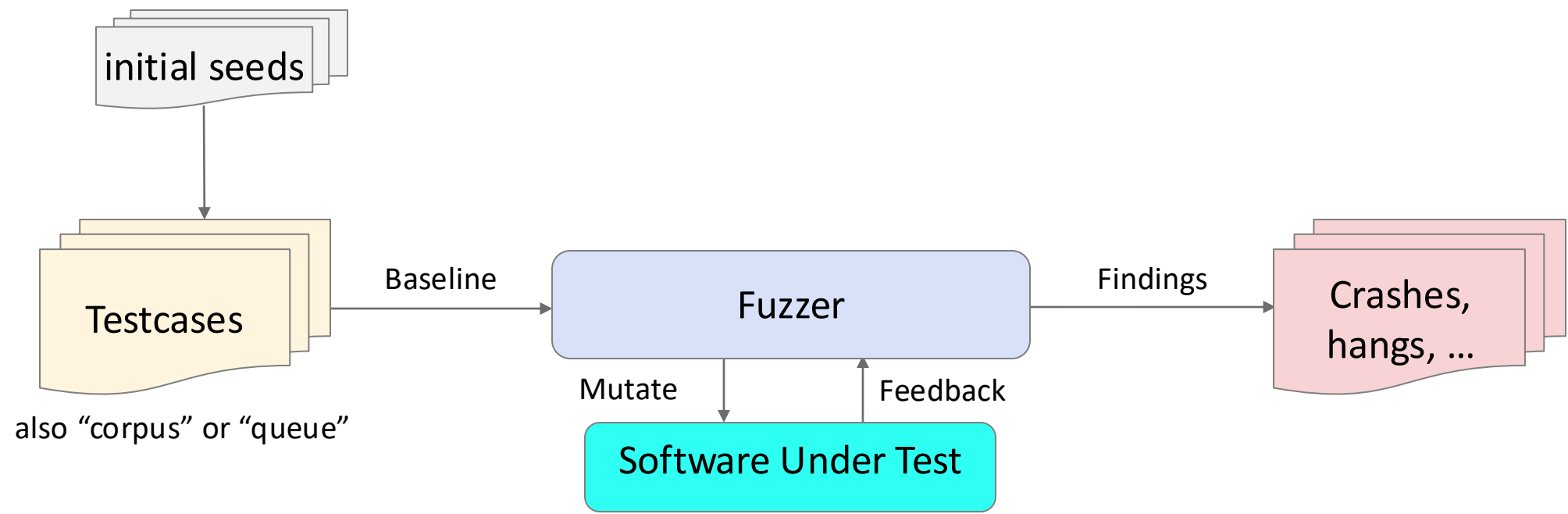
in 850 critical open-source projects since 2016

In this workshop you learn how to rehost network services from embedded systems for fuzzing



- We assume basic fuzzing knowledge
- Some AFL++ experience is a prerequisite
- Use Linux (Ubuntu 24.04 preferred) – MacOS at your own risk only

There are not many components to a fuzzer



Fuzzing network services on embedded devices is important, effective – and difficult

Fuzzing embedded devices is important

- Besides manual testing and reverse engineering binaries, fuzzing is one of the most effective techniques to identify vulnerabilities in an embedded device.
- It can uncover memory corruption issues that an attacker could exploit to compromise the target.

Embedded fuzzing is often very difficult though

To be able to fuzz an embedded services we face many issues:

- Target service and its execution environment required
- On-device testing difficult due CPU & RAM limits and security restrictions
- Target service has often complex device and process dependencies

We show you an approach that often works – and is “easy”

We successfully fuzz embedded network services by:

1. Copying over the embedded filesystem (rehosting)
2. Running the service rehosted and on the device to identify required process and device interactions
3. Mock/Patch process and device interactions
4. Fuzz the service!

Fuzzing a rehosted embedded network service on a dedicated machine has multiple advantages

Speed

- A laptop/server has more CPU and RAM, leading to faster executions and more parallel fuzzing instances

Coverage

- Through emulation we obtain coverage information on our fuzzing efforts helping to perform in-depth testing

Mocking

- By preloading, patching or emulation modifications, limitations on fuzzing can be circumnavigated

Agenda

-
- A Workshop introduction
 - B Workshop preparation**
 - C From firmware to fuzzing
-

Prerequisite 1/2: Get the docker container that has all required files and tools

**Get the
workshop
docker
container**

```
docker pull vanhauser/workshop
```

**Run the
workshop
docker
container**

```
docker run -ti -v /tmp:/share --privileged vanhauser/workshop
```

Prerequisite 2/2: Have a binary reverse engineering tool of your choice available

Ghidra

```
wget  
https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_11.4_build/ghidra_11.4_PUBLIC_20250620.zip
```

Binary Ninja

```
firefox https://binary.ninja/free/
```

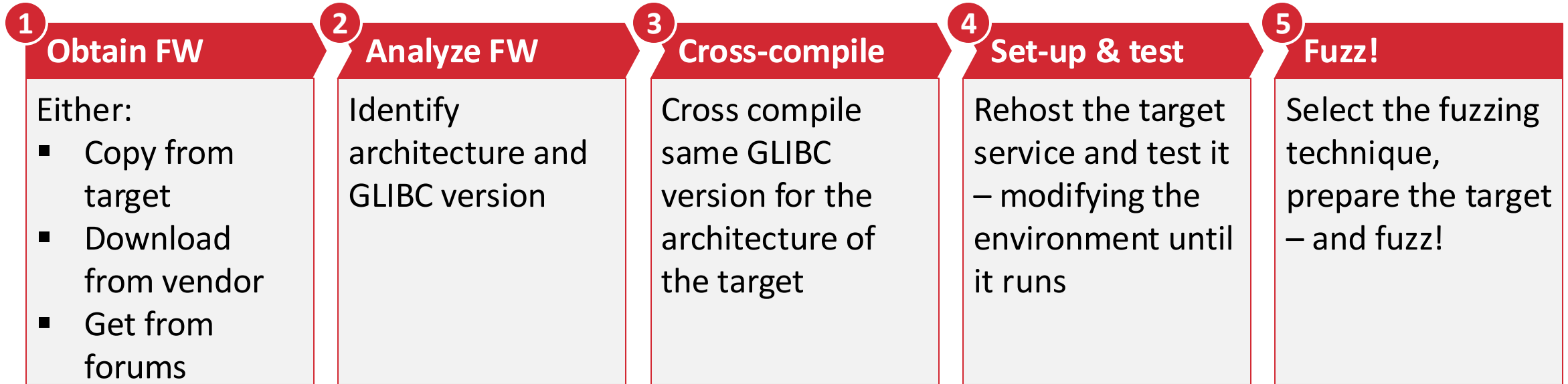
Agenda

-
- A Workshop introduction
 - B Workshop preparation
 - C From firmware to fuzzing**
-

Our target for this workshop: ASUS ExpertWiFi EBM68



We fuzz our target in five steps



1 Obtain the firmware

Browse to https://www.asus.com/de/supportonly/ebm68/helpdesk_bios/ and select a firmware update to download.

Utilities

ASUS Firmware Restoration version 2.1.0.3

Version 2.1.0.3 1.25 MB 2023/08/28

Please verify the checksum with the zip file.

SHA256: 474e81da30e3ccf419e2ffab27f7c2309017bbf1472ada85bf15c8faa411a30f

OS support: Windows XP/7/8/8.1/10/11

Firmware Restoration is used on an ASUS Wireless Router that failed during its firmware upgrading...

[MEHR BESCHREIBUNG ANZEIGEN](#) ▾

Download

`https://dlcdnets.asus.com/pub/ASUS/wireless/EBM68/FW_EBM68_300610244384.zip?model=EBM68`

You can find it in the workshop docker container as **FW_EBM68_300610244384.zip**

2 Unpack the firmware, then identify the architecture and GLIBC version

Unzip FW

```
unzip FW_EBM68_300610244384.zip
```

Binwalk extracts FW binary data

```
binwalk -e --run-as=root EBM68_3.0.0.6_102_44384_-g304340a_370-  
g24e51_sec_nand_squashfs.pkgtb
```

Go to the extracted file system

```
cd _EBM68_3.0.0.6_102_44384_-g304340a_370-  
g24e51_sec_nand_squashfs.pkgtb.extracted  
cd squashfs-root
```

Identify arch and GLIBC

```
file lib/libc.so*  
strings lib/libc.so* | grep GLIBC | tail
```

```
lib/libc.so.6: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), ...  
...  
GLIBC_2.30
```


If in your future research binwalk fails – well bad for you :-)



3 Cross-compile a compatible GLIBC to the target architecture

Unpack GLIBC

```
# wget https://ftp.gnu.org/gnu/glibc/glibc-2.30.tar.gz #  
tar xzf glibc-2.30.tar.gz  
cd glibc-2.30
```

Create a build directory and configure for ARM

```
mkdir build  
cd build  
../configure --host=arm-linux-gnueabi \  
--build=x86_64-linux-gnu CC=arm-linux-gnueabi-gcc-9 \  
CXX=arm-linux-gnueabi-g++-9 AR=arm-linux-gnueabi-ar \  
AS=arm-linux-gnueabi-as LD=arm-linux-gnueabi-ld \  
RANLIB=arm-linux-gnueabi-ranlib \  
STRIP=arm-linux-gnueabi-strip \  
--prefix=`pwd`/local_install
```

Build GLIBC for ARM

```
make -j4
```

Install locally

```
make install # this is installed to $PWD/local_install
```

3 Cross-compile a compatible GLIBC – compiling for the target

Use the
repository
script

```
cross-compile.sh -o target.so target.c
```

Or cross-
compile by
hand for the
ARM target
with GLIBC
2.30

```
arm-linux-gnueabi-gcc-9 -mfloat-abi=soft -nostdlib \  
-Wl,--dynamic-linker=/lib/ld-linux.so.3 \  
-Wl,-rpath=/lib \  
-I./glibc-2.30/build/local_install/include \  
-L./glibc-2.30/build/local_install/lib \  
-shared -fPIC \  
-o target.so target.c
```

4 Obtain the binary, its libraries and other necessary files

In our setup we do not need this step as we have a full copy of the filesystem (preferred!).

Identify all libraries

```
objdump -p usr/sbin/httpd | grep NEEDED  
# grab ./lib/ld*.so* too!
```

Check for file references

```
strings usr/sbin/httpd | grep -E ' ^/'
```

Check for runtime references

```
strace -o strace.log -f -v qemu-arm -L `pwd` \  
    ./usr/sbin/httpd  
grep -E 'open|stat' strace.log | grep -w -- -1
```

4 Obtain the binary, its libraries and other necessary files – then test the target

Our target is **usr/sbin/httpd** – a self written HTTP server by Asus.

**Enter the
target root**

```
cd /path/to/squashfs-root
```

**Be root
(port bind!)**

```
sudo bash
```

**Test the
target – fix
any issues!**

```
qemu-arm -L `pwd` usr/sbin/httpd
```

5 Carefully choose how to fuzz the target

Fuzzing technique	Advantages	Disadvantages
Blind via TCP/IP (we do not care for this basic approach in this workshop 😊)	<ul style="list-style-type: none">▪ Easiest setup▪ Many tools available	<ul style="list-style-type: none">▪ Unknown coverage▪ Hard to find bugs
a Coverage-guided via TCP/IP	<ul style="list-style-type: none">▪ Coverage▪ Minimal target modification	<ul style="list-style-type: none">▪ Slow
b Coverage-guided with desocketing	<ul style="list-style-type: none">▪ Coverage▪ Minimal target modification▪ Fast	<ul style="list-style-type: none">▪ Desocketing can be very difficult
c Coverage-guided persistent	<ul style="list-style-type: none">▪ Coverage▪ Super fast	<ul style="list-style-type: none">▪ Some/huge target modification required▪ No state allowed in the fuzzed functionality▪ Sometimes impossible

5 Coverage guided fuzzing considerations – fuzzing happens in a loop

The fuzzing process in the target	What we need to do
<ol style="list-style-type: none">1. The fuzzing loop starts from forkserver2. The target reads the fuzzing input3. The targets processes the input4. The target must exit after processing the input	<p>Find the optimal entry address TCP/IP, desocketing, in-process patch in <code>_exit()</code>/return</p>

5a Reverse engineer the target – where does it accept() and close()/shutdown()?

CodeBrowser: ghidra/httpd

File Edit Analysis Graph Navigation Search Select Tools Window Help

Program Trees

- httpd
 - .bss
 - .data
 - .got
 - .dynamic
 - .data.rel.ro
 - .fini_array
 - .init_array

Symbol Tree

- Imports
 - <EXTERNAL>
 - libasuslog.so
 - libbcm_boardctl.so
 - libbcm_flashutil.so
 - libbcm_util.so
 - libbwdni.so

Data Type Manager

- Data Types
 - BuiltInTypes
 - httpd
 - generic_clib

Listing: httpd

```
000194c8 00 40 80 e5 str r4, #0x0
000194cc 00 00 8f e0 add r0, pc, r0
000194d0 5f ff ff ea b LAB_00019254

LAB_000194d4
000194d4 28 04 9f e5 ldr r0, [DAT_00019904]
000194d8 00 00 8f e0 add r0=>s_HTTPD_DBG_0009226
000194dc b6 f8 ff eb bl <EXTERNAL>::nvram_get_i
000194e0 00 00 50 e3 cmp r0, #0x0
000194e4 e6 ff ff ca bgt LAB_00019484
000194e8 f5 ff ff ea b LAB_000194c4

LAB_000194ec
000194ec 05 20 a0 e1 cpy r2, r5
000194f0 20 10 a0 e3 mov r1, #0x20
000194f4 28 10 a2 e5 str r1, [r2, #0x28]!
000194f8 0c 10 85 e2 add r1, r5, #0xc

LAB_000194fc
000194fc 0b 00 a0 e1 cpy r0, r11
00019500 2c 20 8d e5 str r2, [sp, #local_164]
00019504 28 10 8d e5 str r1, [sp, #local_168]
00019508 14 fc ff eb bl <EXTERNAL>::accept
0001950c 00 00 50 e3 cmp r0, #0x0
00019510 08 00 85 e5 str r0, [r5, #0x8]
00019514 76 00 00 aa bge LAB_000196f4
00019518 00 30 96 e5 ldr r3, [r6, #0x0]
0001951c 28 10 9d e5 ldr r1, [sp, #local_168]
00019520 04 00 53 e3 cmp r3, #0x4
00019524 2c 20 9d e5 ldr r2, [sp, #local_164]
00019528 1c 30 8d e5 str r3, [sp, #local_174]
0001952c f2 ff ff 0a beq LAB_000194fc
00019530 d0 03 9f e5 ldr r0, [DAT_00019908]
00019534 00 00 8f e0 add r0=>s_/tmp/HTTPD_DEBUG_
```

Functions - 6 items (of 1646)

Name	Function Signat...	Function Size
strspn	00... thunk size_...	12
strpbrk	00... thunk char ...	12
accept	00... thunk int a...	12
strspn	00... thunk size_...	1
strpbrk	00... thunk char ...	1
accept	00... thunk int a...	1

Filter: accept

Decompile: FUN_00018a88 x Defined Strings x Functions x

Console - Scripting

00019508 FUN_00018a88 bl 0x00018560

5 How to make the target exit

How to force an exit

1. **Binary patch.** Use your binary reverser to modify the import for shutdown() to _exit()
2. **LD_PRELOAD.** Preload a library that hooks shutdown() and calls _exit()
3. **AFL_EXITPOINT.** Define the address for qemu afl when to exit.
(features needs still to be implemented 😊)

Key takeaway: Always use the fastest/easiest solution.
Time to fuzz is important, not a perfect solution!

Input assembly

Arch and syntax: ARM - little endian

```
mov r0, #0  
bl #-2620
```

Output code

☒ C String ☐ C Array ☐ Python Array ☐ Hex

```
"\x00\x00\xa0\xe3" // mov r0, #0  
"\x6f\xfd\xff\xeb" // bl #-2620
```

5a Fuzz coverage-guided via the TCP/IP

We use AFL++'s custom mutator for TCP send (that is already compiled).

Setup environment for our TCP module

```
export CUSTOM_SEND_IP=127.0.0.1
export CUSTOM_SEND_PORT=80
export AFL_CUSTOM_MUTATOR_LIBRARY=
    /afl++/custom_mutators/custom_send_tcp/custom_send_tcp.so
export AFL_CUSTOM_MUTATOR_LATE_SEND=1
export QEMU_LD_PREFIX=`pwd`
```

Preload shutdown()

```
export AFL_PRELOAD=./lib-fuzz-tcp.so
```

Fuzz!

```
afl-fuzz -Q -i in -o out -c 0 -- usr/sbin/httpd
```

5a Reverse engineer the target – where should we set the forkserver? (for more speed)

CodeBrowser: ghidra/httpd

File Edit Analysis Graph Navigation Search Select Tools Window Help

Program Trees

- httpd
 - .bss
 - .data
 - .got
 - .dynamic
 - .data.rel.ro
 - .fini_array
 - .init_array

Symbol Tree

- Imports
 - <EXTERNAL>
 - libasuslog.so
 - libbcm_boardctl.so
 - libbcm_flashutil.so
 - libbcm_util.so
 - libbwdni.so

Data Type Manager

- Data Types
 - BuiltInTypes
 - httpd
 - generic_clib

Listing: httpd

```
000194c8 00 40 80 e5 str r4, #0x0
000194cc 00 00 8f e0 add r0, pc, r0
000194d0 5f ff ff ea b LAB_00019254

LAB_000194d4
000194d4 28 04 9f e5 ldr r0, [DAT_00019904]
000194d8 00 00 8f e0 add r0=>s_HTTPD_DBG_0009226
000194dc b6 f8 ff eb bl <EXTERNAL>::nvram_get_i
000194e0 00 00 50 e3 cmp r0, #0x0
000194e4 e6 ff ff ca bgt LAB_00019484
000194e8 f5 ff ff ea b LAB_000194c4

LAB_000194ec
000194ec 05 20 a0 e1 cpy r2, r5
000194f0 20 10 a0 e3 mov r1, #0x20
000194f4 28 10 a2 e5 str r1, [r2, #0x28]!
000194f8 0c 10 85 e2 add r1, r5, #0xc

LAB_000194fc
000194fc 0b 00 a0 e1 cpy r0, r11
00019500 2c 20 8d e5 str r2, [sp, #local_164]
00019504 28 10 8d e5 str r1, [sp, #local_168]
00019508 14 fc ff eb bl <EXTERNAL>::accept
0001950c 00 00 50 e3 cmp r0, #0x0
00019510 08 00 85 e5 str r0, [r5, #0x8]
00019514 76 00 00 aa bge LAB_000196f4
00019518 00 30 96 e5 ldr r3, [r6, #0x0]
0001951c 28 10 9d e5 ldr r1, [sp, #local_168]
00019520 04 00 53 e3 cmp r3, #0x4
00019524 2c 20 9d e5 ldr r2, [sp, #local_164]
00019528 1c 30 8d e5 str r3, [sp, #local_174]
0001952c f2 ff ff 0a beq LAB_000194fc
00019530 d0 03 9f e5 ldr r0, [DAT_00019908]
00019534 00 00 8f e0 add r0=>s_/tmp/HTTPD_DEBUG_
```

Functions - 6 items (of 1646)

Name	Function Signat...	Function Size
strspn	00... thunk size_...	12
strpbrk	00... thunk char ...	12
accept	00... thunk int a...	12
strspn	00... thunk size_...	1
strpbrk	00... thunk char ...	1
accept	00... thunk int a...	1

Filter: accept

Decompile: FUN_00018a88 x Defined Strings x Functions x

Console - Scripting

00019508 FUN_00018a88 bl 0x00018560

5a Fuzz coverage-guided via the TCP/IP

Add for
speed

```
export AFL_ENTRYPOINT=0x190c4
```

Speed without
forkserver

1230 exec/s

Speed with
forkserver

1885 exec/s **+50%!**

5a Run as many instances as you want – with network namespaces (if run not in docker)

```
#!/bin/bash

set -e

NS_NAME="$1"
ip netns add "$NS_NAME"
ip netns exec "$NS_NAME" ip link set lo up

ip netns exec "$NS_NAME" -- afl-fuzz -S "$NS_NAME" -Q ...

ip netns delete "$NS_NAME"
```

5b Reverse engineer the target – how does it interact with the TCP socket?

The screenshot displays the CodeBrowser interface for the ghidra:/httpd target. The main window shows the assembly code for the function FUN_0001bb18, with the instruction at address 0001bb44 (add r6,sp,#0x380) selected. The left sidebar contains the Program Trees, Symbol Tree, and Data Type Manager. The right sidebar shows the decompiled code for FUN_0001bb18, which includes local variables and a call to fgets. The bottom status bar indicates the current instruction is at address 0001bb44.

CodeBrowser: ghidra:/httpd

File Edit Analysis Graph Navigation Search Select Tools Window Help

Program Trees

- httpd
 - .bss
 - .data
 - .got
 - .dynamic
 - .data.rel.ro
 - .fini_array
 - .init_array

Symbol Tree

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type Manager

- Data Types
 - BuiltInTypes
 - httpd
 - generic_clib

Listing: httpd - (4 addresses selected)

```
0001bb18 f0 4e 2d e9 stmdb    sp!,{r4,r5,r6,r7,r9,r10}
0001bb1c aa dd 4d e2 sub     sp,sp,#0x2a80
0001bb20 78 af 9f e5 ldr     r10,[DAT_0001caa0]
0001bb24 10 d0 4d e2 sub     sp,sp,#0x10
0001bb28 74 3f 9f e5 ldr     r3,[DAT_0001caa4]
0001bb2c a9 4e 8d e2 add     r4,sp,#0xa90
0001bb30 0a a0 8f e0 add     r10,pc,r10
0001bb34 00 10 a0 e3 mov     r1,#0x0
0001bb38 3c 1a 04 e5 str     r1,[r4,#local_2a5c]
0001bb3c 01 20 a0 e3 mov     r2,#0x1
0001bb40 03 30 9a e7 ldr     r3,[r10,r3]==>PTR_DAT_0001caa8
0001bb44 0e 6d 8d e2 add     r6,sp,#0x380
0001bb48 06 00 a0 e1 cpy     r0,r6
0001bb4c 00 20 83 e5 str     r2,[r3,#0x0]==>DAT_0001bee2
0001bb50 10 27 02 e3 movw    r2,#0x2710
0001bb54 4c 3f 9f e5 ldr     r3,[DAT_0001caa8]
0001bb58 03 30 9a e7 ldr     r3,[r10,r3]==>PTR_DAT_0001caa8
0001bb5c 00 10 83 e5 str     r1,[r3,#0x0]==>DAT_0001bee2
0001bb60 44 3f 9f e5 ldr     r3,[DAT_0001caac]
0001bb64 03 30 9a e7 ldr     r3==>host_name,[r10,r3]==>PTR_DAT_0001caa8
0001bb68 00 10 c3 e5 strb    r1,[r3,#0x0]==>host_name
0001bb6c e1 ef ff eb bl      <EXTERNAL>::memset
0001bb70 38 3f 9f e5 ldr     r3,[DAT_0001cab0]
0001bb74 10 17 02 e3 movw    r1,#0x2710
0001bb78 06 00 a0 e1 cpy     r0,r6
0001bb7c 03 30 8f e0 add     r3==>DAT_0001bee8,pc,r3
0001bb80 04 20 93 e5 ldr     r2,[r3,#0x4]==>DAT_0001bee8
```

Decompile: FUN_0001bb18 - (httpd)

```
34  undefined4 local_2a5c;
35  char *local_2a58;
36  char *local_2a54;
37  undefined1 auStack_2a50 [32];
38  uint local_2a30;
39  undefined1 auStack_2a2c [252];
40  char local_2930 [512];
41  byte local_2730 [10000];
42
43  local_2a5c = 0;
44  DAT_000b69ac = 1;
45  DAT_000bee2c = 0;
46  host_name[0] = 0;
47  memset(local_2730,0,10000);
48  pcVar1 = fgets((char *)local_2730,10000,DAT_000bee2c);
49  if (pcVar1 == (char *)0x0) {
50      return;
51  }
52  local_2a68 = local_2730;
53  strsep((char **)&local_2a68," ");
54  for (; (local_2a68 != (byte *)0x0 && (*local_2a68 == 0x20)); local_2a68 = local_2a68 + 1) {
55  }
56  local_2a64 = local_2a68;
57  strsep((char **)&local_2a64," ");
58  for (; (local_2a64 != (byte *)0x0 && (*local_2a64 == 0x20)); local_2a64 = local_2a64 + 1) {
59  }
60  local_2a60 = local_2a64;
61  strsep((char **)&local_2a60," ");
62  pbVar1 = local_2a64;
63  if ((local_2a68 == (byte *)0x0) || (local_2a64 == (byte *)0x0)) {
64      pcVar1 = "Can't parse request.";
65  }
```

Decompile: FUN_0001bb18 x 0101 DAT Defined Strings x Functions x

Console - Scripting

0001bb44 FUN_0001bb18 add r6,sp,#0x380

5b Fuzz coverage-guided via desocketing

Desocketing explained

Transform the TCP/IP messaging of the target applicated to ***stdin*** reading/writing by modifying the target. The usual solution is to AFL_PRELOAD a library that intercepts the accept() call and return file descriptor 0.

- <https://github.com/zardus/preeny> => desock and desock2
- <https://github.com/fkie-cad/libdesock>
- <https://github.com/zyingp/desockmulti>
- <https://github.com/vanhauser-thc/network-emulator> (fork)

5b Fuzz coverage-guided via desocketing can be very difficult

Common issue	Solution path	Tool hint
Statically compiled targets	Binary modification (e.g. with Ghidra) of the function that handles <code>accept()</code>	Ghidra <ul style="list-style-type: none">▪ https://github.com/NationalSecurityAgency/ghidra/releases
Complex network libraries (e.g. boost, Rust)	AFL_PRELOAD the necessary exported functions (which is difficult and messy)	---
Potential solution is not feasible	Modify qemuaf1 to intercept the right <code>accept/read/write</code> syscalls	QEMU AFL <ul style="list-style-type: none">▪ https://github.com/AFLplusplus/qemuaf1

5b Fuzz coverage-guided via desocketing

Find the right desocket library and modify it for your target (e.g. handling getsockopt/setsockopt etc.)

Preload desocketing

```
export AFL_PRELOAD=./lib-fuzz-desock.so  
export QEMU_LD_PREFIX=`pwd`
```

Fuzz!

```
afl-fuzz -Q -i in -o out -c 0 -- usr/sbin/httpd
```

5a Fuzz coverage-guided via the desocketing

Add for
speed

```
export AFL_ENTRYPOINT=0x1bb84
```

Speed without
forkserver

1125 exec/s

Speed with
forkserver

1930 exec/s **+70%!**

5c Fuzz persistent – how could we fuzz persistently? (start/end loop!)

The screenshot displays the CodeBrowser application window for the target `ghidra/httpd`. The interface is divided into several panes:

- Program Tree:** Shows the file structure of the target, including `.bss`, `.data`, `.got`, `.dynamic`, `.data.rel.ro`, `.fini_array`, and `.init_array`.
- Symbol Tree:** Displays the symbol table, including imports, exports, functions, labels, classes, and namespaces.
- Data Type Manager:** Shows the data types used in the code, including built-in types and user-defined types like `httpd` and `generic_clib`.
- Listing:** Shows the assembly code for the selected function `FUN_0001bb18`. The code is listed with addresses and disassembled instructions.
- Decompile:** Shows the decompiled C code for the selected function. The code includes local variables, memory management, and a loop that processes input data.
- Console - Scripting:** A window at the bottom for running scripts.

The decompiled code for `FUN_0001bb18` is as follows:

```
34 undefined4 local_2a5c;
35 char *local_2a58;
36 char *local_2a54;
37 undefined1 auStack_2a50 [32];
38 uint local_2a30;
39 undefined1 auStack_2a2c [252];
40 char local_2930 [512];
41 byte local_2730 [10000];
42
43 local_2a5c = 0;
44 DAT_000b69ac = 1;
45 DAT_000bee2c = 0;
46 host_name[0] = 0;
47 memset(local_2730, 0, 10000);
48 pcVar1 = fgets((char *)local_2730, 10000, DAT_000bebec);
49 if (pcVar1 == (char *)0x0) {
50     return;
51 }
52 local_2a68 = local_2730;
53 strsep((char **)&local_2a68, " ");
54 for (; (local_2a68 != (byte *)0x0 && (*local_2a68 == 0x20)); local_2a68 = local_2a68 + 1) {
55 }
56 local_2a64 = local_2a68;
57 strsep((char **)&local_2a64, " ");
58 for (; (local_2a64 != (byte *)0x0 && (*local_2a64 == 0x20)); local_2a64 = local_2a64 + 1) {
59 }
60 local_2a60 = local_2a64;
61 strsep((char **)&local_2a60, " ");
62 pbVar11 = local_2a64;
63 if ((local_2a68 == (byte *)0x0) || (local_2a64 == (byte *)0x0)) {
64     pcVar1 = "Can't parse request.";
65 }
```

5c Fuzz persistent – what we need

What we need

Start & end point

of target functionality in one function

- Can be in the middle
- Can call other functions

Stateless functionality:

Our target should not keep state

Input path:

We need to be able to easily provide our fuzz input

What we do

- Follow normal program flow functionality if initialization is required
- export function with LIEF if no setup required

We look for globals being set in the call tree

Examine how the data is read. Use afl++/utils/qemu_persistent_hook to inject data if it is possible

How it is in our target

We can do both:

- export FUN_0001bb18
- patch the binary for select/accept/...
- Use the desocketing library 😊

Looks good!

Multiple calls to fgets(), with the FILE pointer being set in main(). Solvable with fmemopen() and overwriting the FILE pointer => super messy.
=> Keep stdin solution!

5c Fuzz persistent

Set the persistent loop

```
export AFL_QEMU_PERSISTENT_ADDR=0x195dc
export AFL_QEMU_PERSISTENT_RET=0x19688
export AFL_QEMU_PERSISTENT_GPR=1
export QEMU_LD_PREFIX=`pwd`
export AFL_PRELOAD=./lib-fuzz-desock.so
```

Fuzz!

```
afl-fuzz -Q -i in -o out -c 0 -- usr/sbin/httpd
```

Speed!!! 😊

14235 exec/s

Add a target dictionary for better coverage

**Create a
dictionary
for this
target**

```
for i in `strings usr/sbin/httpd | grep -E '^[A-Z][a-zA-Z-]*:$'\`;  
do  
    echo \"$i\  
done > target.dic
```

Fuzz!

```
afl-fuzz -Q -x target.dic -i in -o out -c 0 -- usr/sbin/httpd
```

Workshop material: You can find everything (including slides) in the repository

**Clone
workshop
repository**

```
git clone https://github.com/srlabs/rehosting-fuzzing-workshop/  
cd rehosting-fuzzing-workshop/
```

**Copy-paste
commands**

```
less COMMANDS.md
```

Any final questions?

