Rehosting and coverage guided fuzzing of embedded network services

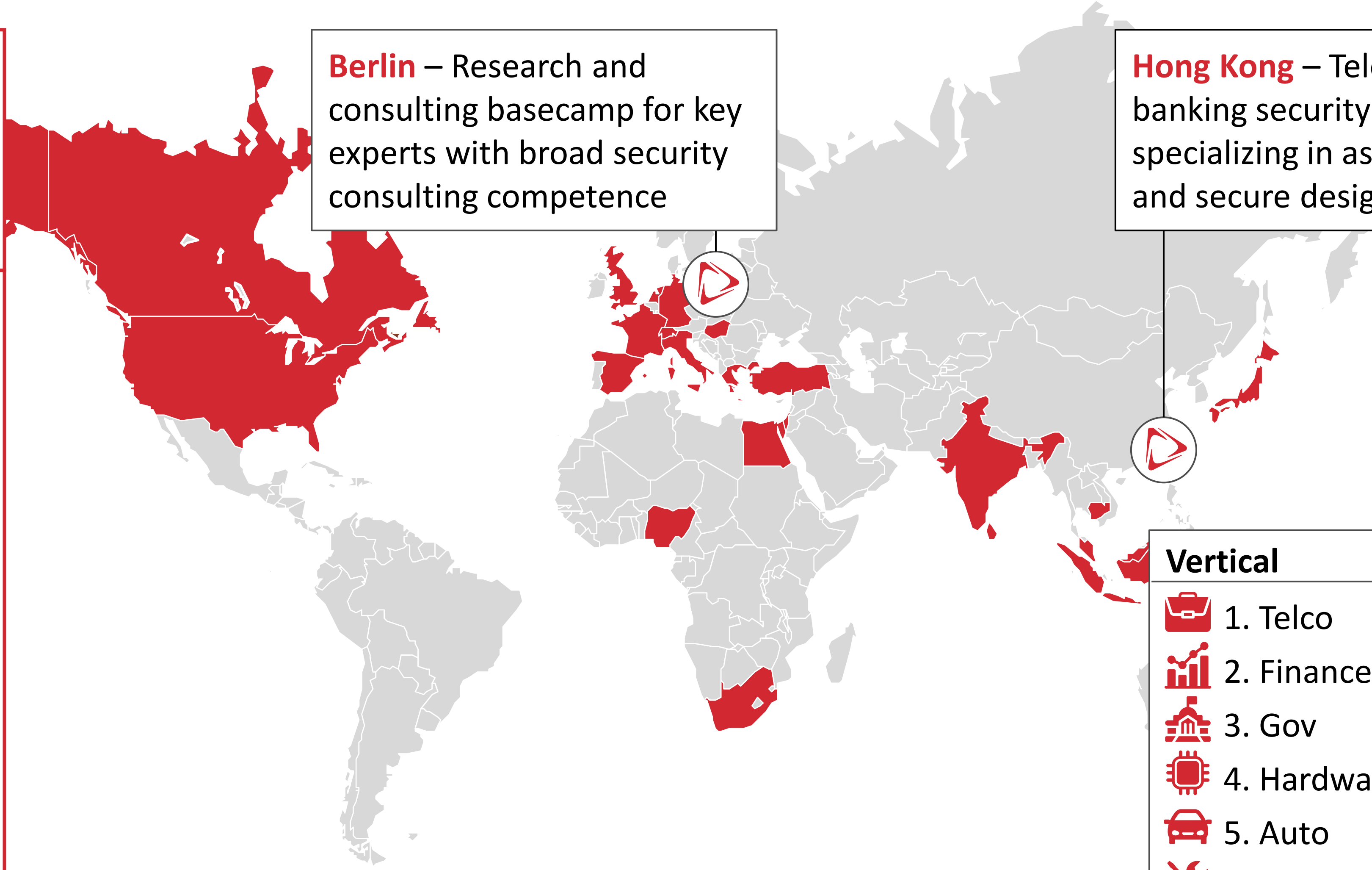
Marc Heuse <marc@srlabs.de>

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



Berlin – Research and consulting basecamp for key experts with broad security consulting competence

Hong Kong – Telecom and banking security experts specializing in assessments and secure design

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 ˈtɛstɪŋ/

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

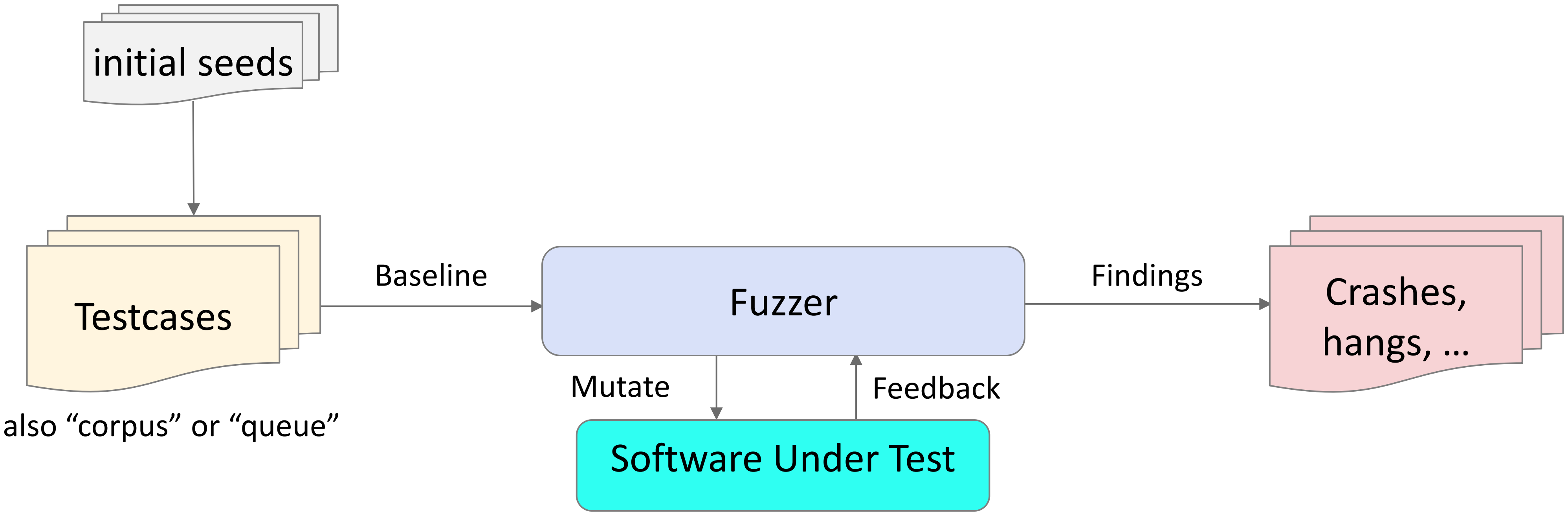
- more than 8,800 vulnerabilities and
- 28,000 bugs

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`

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

Ghidra	`wget https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_11.3.2_build/ghidra_11.3.2_PUBLIC_20250415.zip`
Binary Ninja	`firefox https://binary.ninja/free/`

Agenda

Our target for this workshop: ASUS ExpertWiFi EBM68

We fuzz our target in five steps

1 Obtain FW

Either:
- Copy from target
- Download from vendor
- Get from forums

2 Analyze FW

Identify architecture and GLIBC version

3 Cross-compile

Cross compile same GLIBC version for the architecture of the target

4 Set-up & test

Rehost the target service and test it – modifying the environment until it runs

5 Fuzz!

Select the fuzzing technique, prepare the target – and fuzz!

❶ Obtain the firmware

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

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

Unzip FW	`unzip FW_EBM68_300610244384.zip`
Binwalk extracts FW binary data	`binwalk -e -M 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`

❹ 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``

❺ Carefully choose how to fuzz the target

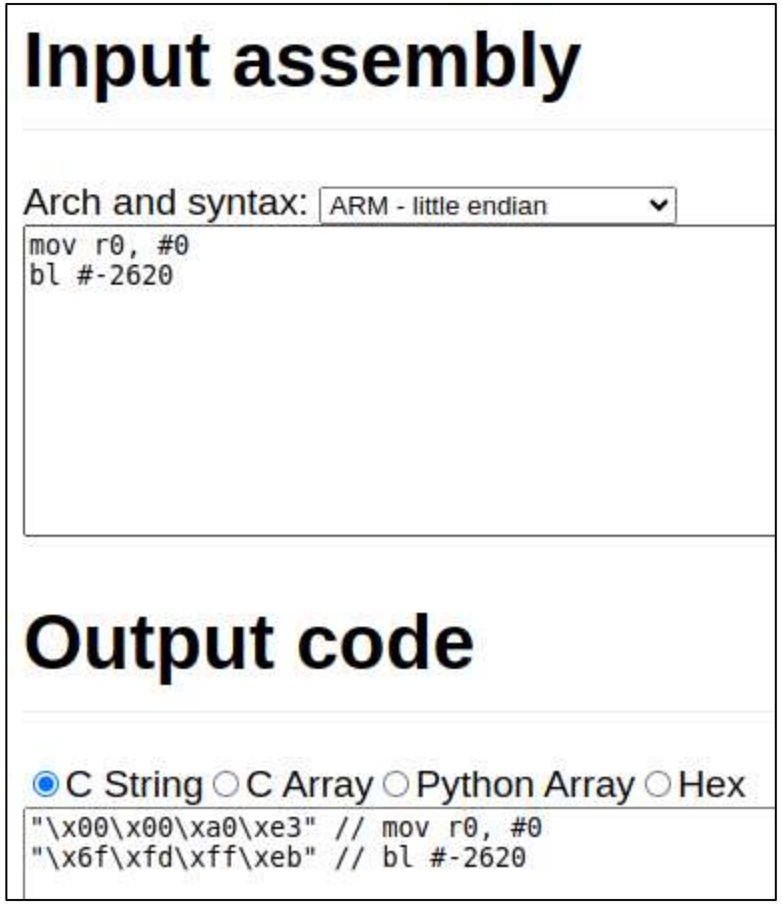
Fuzzing technique	Advantages	Disadvantages
Blind via TCP/IP (we do not care for this basic approach in this workshop ☺)	▪ Easiest setup ▪ Many tools available	▪ Unknown coverage ▪ Hard to find bugs
ⓐ **Coverage-guided via TCP/IP**	▪ Coverage ▪ Minimal target modification	▪ Slow
ⓑ **Coverage-guided with desocketing**	▪ Coverage ▪ Minimal target modification ▪ Fast	▪ Desocketing can be very difficult
ⓒ **Coverage-guided persistent**	▪ Coverage ▪ Super fast	▪ Some/huge target modification required ▪ No state allowed in the fuzzed functionality ▪ Sometimes impossible

❺ Coverage guided fuzzing considerations – fuzzing happens in a loop

The fuzzing process in the target	What we need to do
1. The fuzzing loop starts from forkserver	Find the optimal entry address
2. The target reads the fuzzing input	TCP/IP, desocketing, in-process
3. The targets processes the input	
4. The target must exit after processing the input	patch in _exit()/return

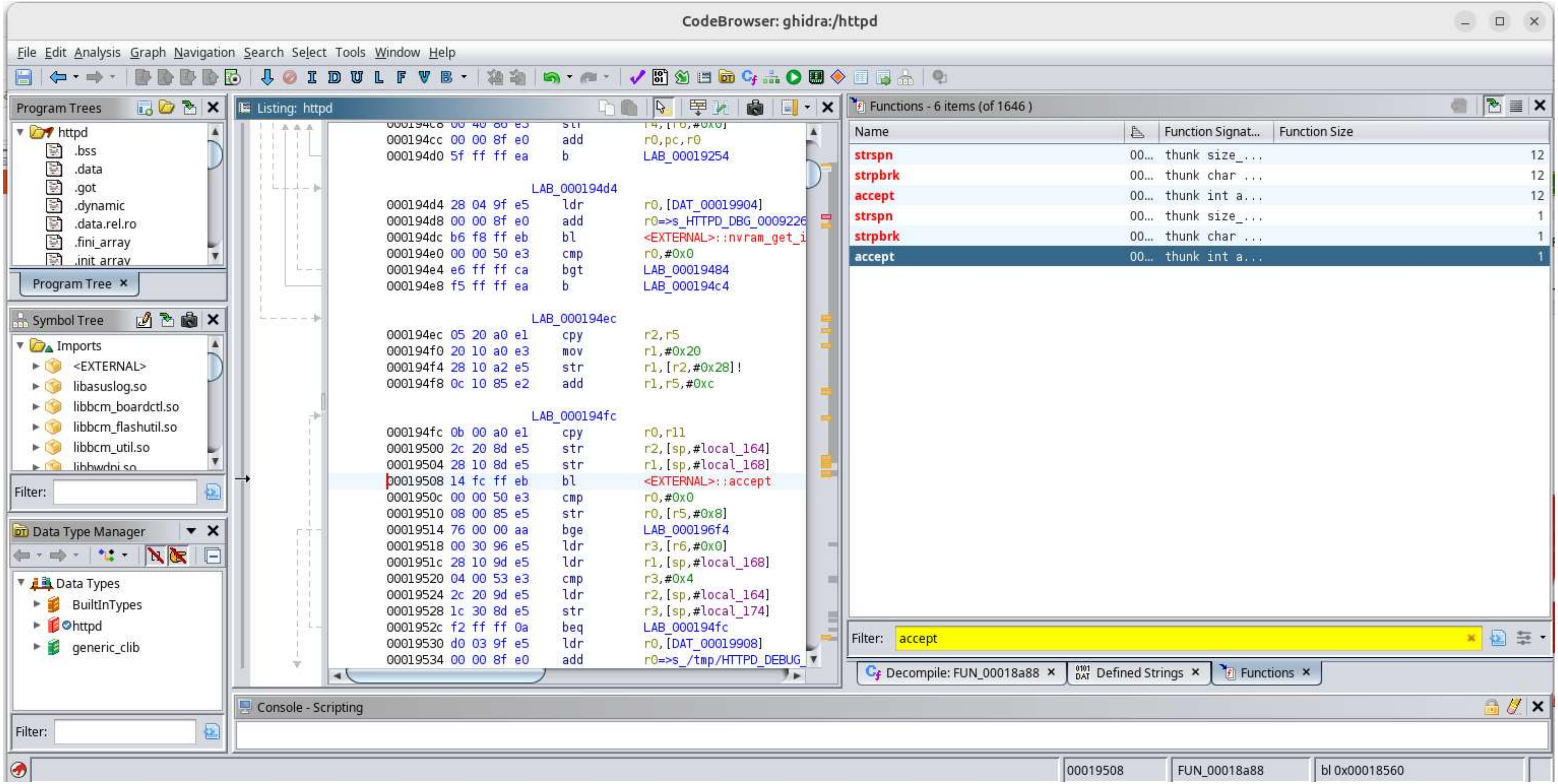
❺ How to make the target exit

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 qemuafl 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!

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



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`
```

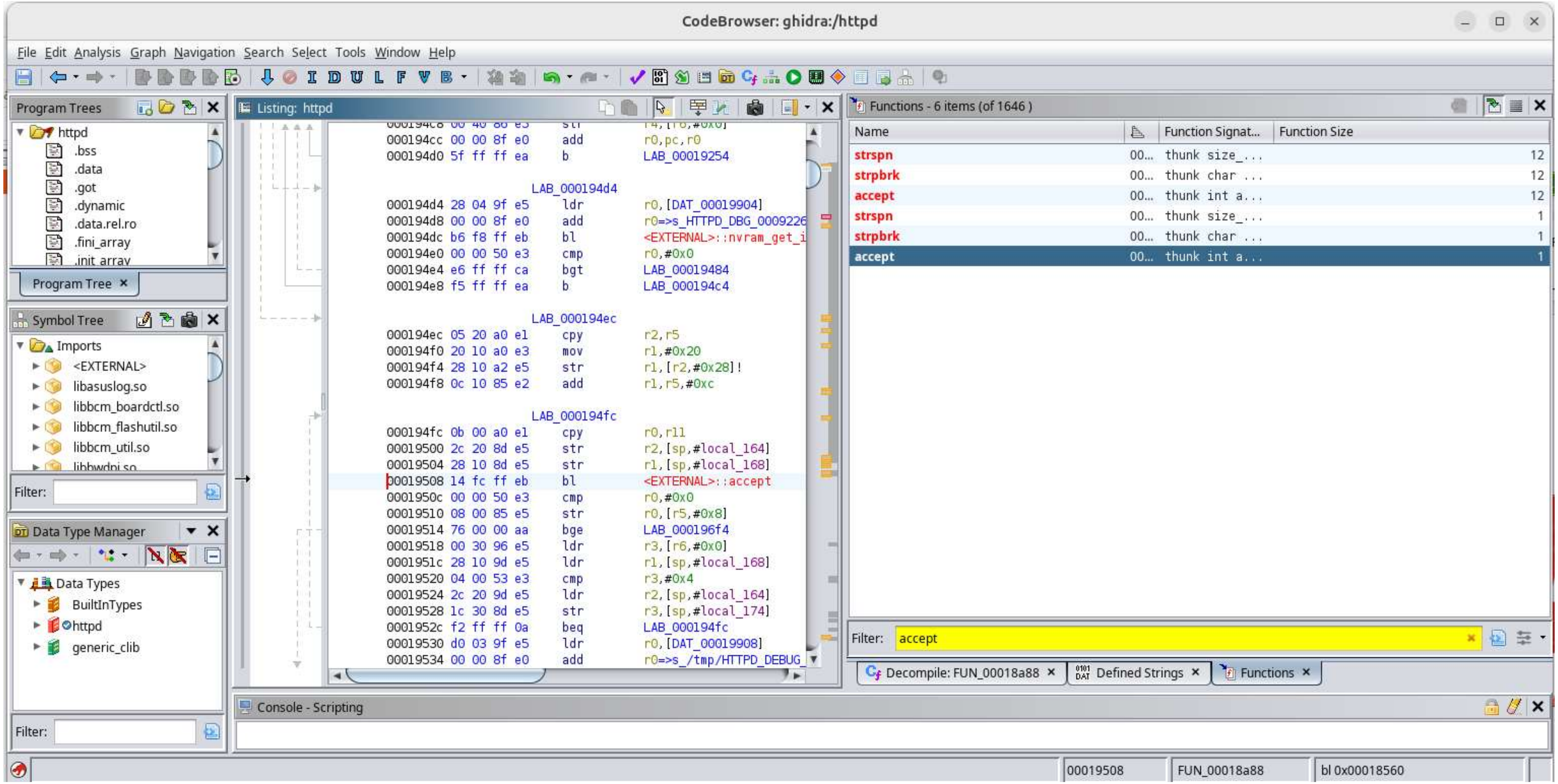
Preload shutdown()

```
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)



5a Fuzz coverage-guided via the TCP/IP

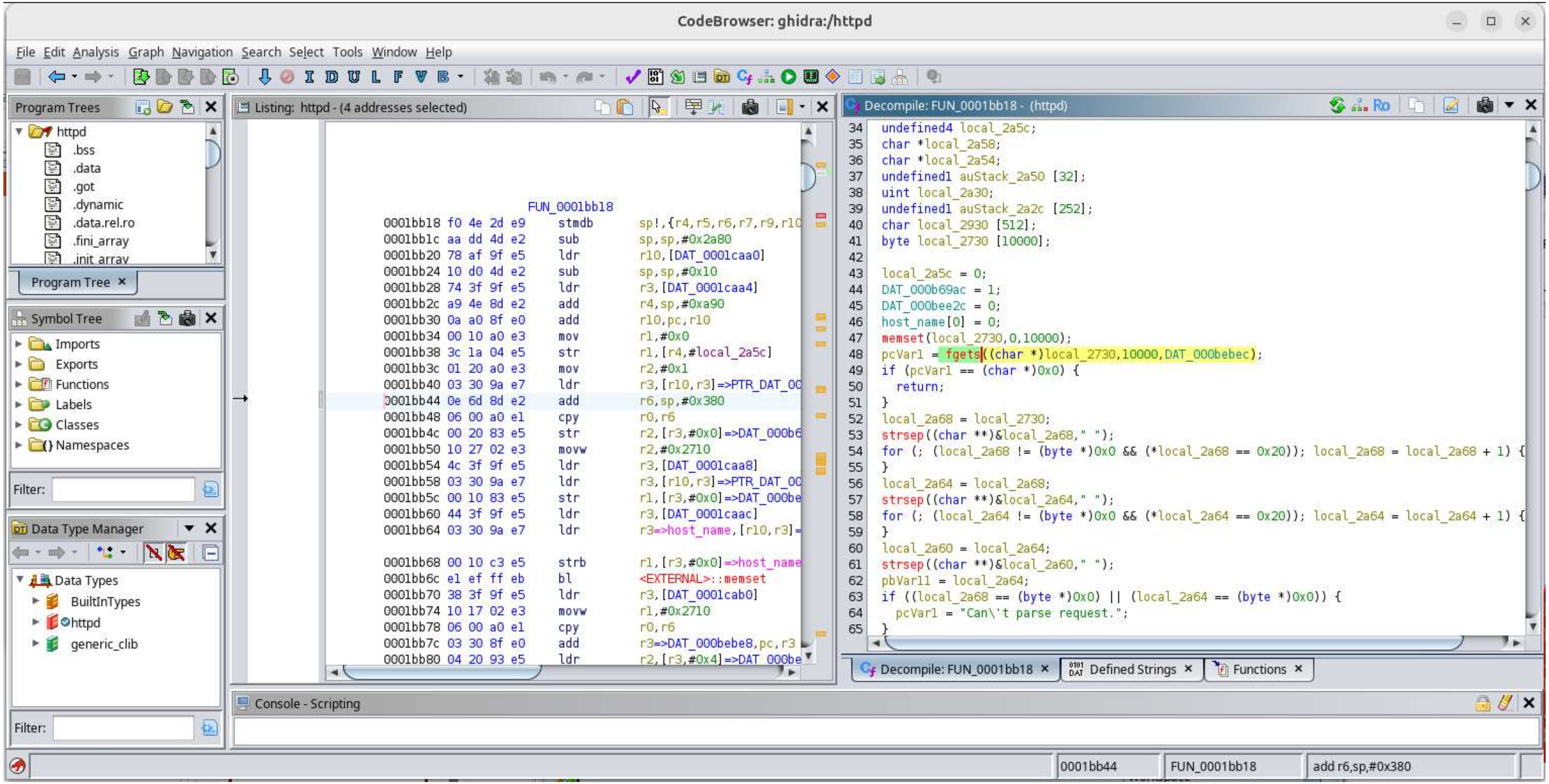
Add for speed	`export AFL_ENTRYPOINT=0x190c4`

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?



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 `accept()`	Ghidra ▪ 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 qemuafl to intercept the right accept/read/write syscalls	QEMUAFL ▪ https://github.com/AFLplusplus/qemuafl

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!

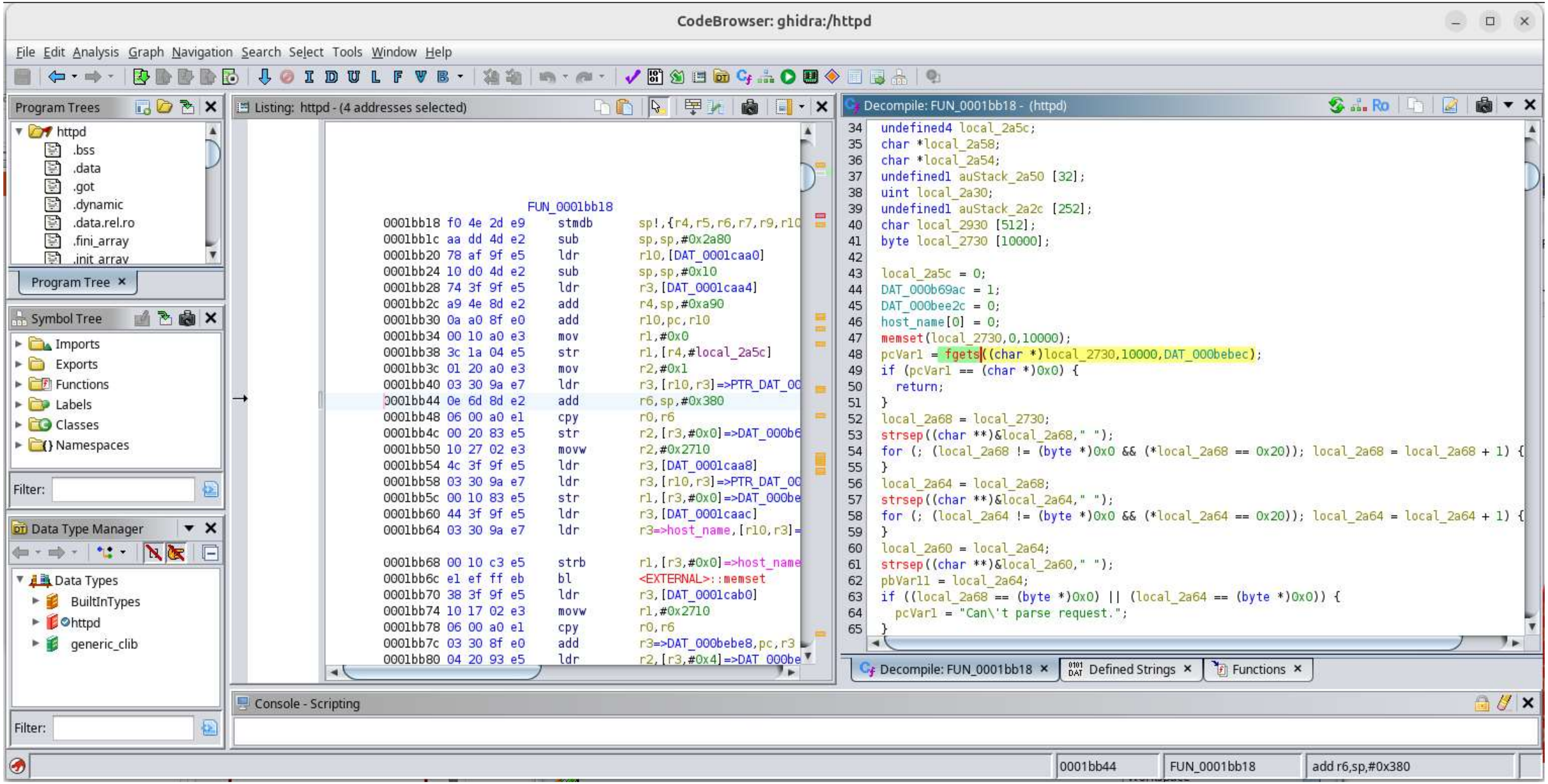
```
afl-fuzz -Q -i in -o out -c 0 -- usr/sbin/httpd
```

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!)



5c Fuzz persistent – what we need

What we need	What we do	How it is in our target
Start & end point of target functionality in one function ▪ Can be in the middle ▪ Can call other functions	▪ Follow normal program flow functionality if initialization is required ▪ export function with LIEF if no setup required	We can do both: • export FUN_0001bb18 • patch the binary for select/accept/… • Use the desocketing library ☺
Stateless functionality: Our target should not keep state	We look for globals being set in the call tree	Looks good!
Input path: We need to be able to easily provide our fuzz input	Examine how the data is read. Use afl++/utils/qemu_persistent_hook to inject data if it is possible	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!

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`

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

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?