fuzz系列之libfuzzer

前言

本文以 libfuzzer-workshop 为基础 介绍 **libFuzzer** 的使用。

libFuzzer简介

libFuzzer 是一个`in-process`，`coverage-guided`，`evolutionary` 的 `fuzz` 引擎，是 `LLVM` 项目的一部分。

`libFuzzer` 和 要被测试的库 链接在一起，通过一个模糊测试入口点（**目标函数**），把测试用例喂给要被测试的库。

`fuzzer`会跟踪哪些代码区域已经测试过，然后在输入数据的语料库上进行**变异**，来使代码覆盖率最大化。代码覆盖率的信息由 `LLVM` 的 `SanitizerCoverage` 插桩提供。

一些概念

`fuzz` 的种类

- `Generation Based`： 通过对目标协议或文件格式建模的方法，从零开始产生测试用例，没有先前的状态
- `Mutation Based`： 基于一些规则，从已有的数据样本或存在的状态变异而来
- `Evolutionary`： 包含了上述两种，同时会根据代码覆盖率的回馈进行变异。

target (被 fuzz 的目标)

基本上所有的程序的主要功能都是对一些 **字节序列** 进行操作，`libfuzzer` 就是基于这一个事实（`libfuzzer` 生成 随机的 字节序列 ， 扔给 待 `fuzz` 的程序，然后检测是否有异常出现） 所以在 `libfuzzer` 看来，`fuzz` 的目标 其实就是一个 以 **字节序列** 为输入的 **函数**。

fuzzer

一个 **生成 测试用例， 交给目标程序测试，然后检测程序是否出现异常** 的程序

corpus（语料库）

给目标程序的各种各样的输入

> 以图片处理程序为例:
>
> 语料库就是各种各样的图片文件，这些图片文件可以是正常图片也可以不是。

传统Fuzz

介绍

传统的 `fuzz` 大多通过对已有的样本 **按照预先设置好的规则** 进行变异产生测试用例，然后喂给 目标程序同时监控目标程序的运行状态。

这类 `fuzz` 有很多，比如: `peach` , `FileFuzz` 等

实战

生成测试用例

本节使用 `radamsa` 作为 变异样本生成引擎，对 pdfium 进行 `fuzz` 。

相关文件位于

`https://github.com/Dor1s/libfuzzer-workshop/tree/master/lessons/02`

`radamsa` 是一个 测试用例生成引擎，它是通过对已有的样本进行变异来生成新的测试用例。

首先看看 测试样本的生成

generate_testcases.py

```
#!/usr/bin/env python2
import os
import random

WORK_DIR = 'work'

# Create work `directory` and `corpus` subdirectory.
if not os.path.exists(WORK_DIR):
  os.mkdir(WORK_DIR)

corpus_dir = os.path.join(WORK_DIR, 'corpus')
if not os.path.exists(corpus_dir):
```

```
  os.mkdir(corpus_dir)

seed_corpus_filenames = os.listdir('seed_corpus')

for i in xrange(1000):
  random_seed_filename = random.choice(seed_corpus_filenames)
  random_seed_filename = os.path.join('seed_corpus', random_seed_filename)
  output_filename = os.path.join(WORK_DIR, 'corpus', 'testcase-%06d' % i)
  cmd = 'bin/radamsa "%s" > "%s"' % (random_seed_filename, output_filename)
  os.popen(cmd)
```

就是调用 `radamsa` ,然后随机选取 `seed_corpus` 目录中的文件名作为参数，传递给 `radamsa` 进行变异，然后把生成的测试用例，放到 `work/corpus` 。

开始fuzz

这样测试样本就生成好了，下面看看 用于 fuzz 的脚本

run_fuzzing.py

```
#!/usr/bin/env python2
import os
import subprocess

WORK_DIR = 'work'

def checkOutput(s):
  if 'Segmentation fault' in s or 'error' in s.lower():
    return False
  else:
    return True

corpus_dir = os.path.join(WORK_DIR, 'corpus')
corpus_filenames = os.listdir(corpus_dir)
for f in corpus_filenames:
  testcase_path = os.path.join(corpus_dir, f)
  cmd = ['bin/asan/pdfium_test', testcase_path]
  process = subprocess.Popen(cmd, stdin=subprocess.PIPE, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
  output = process.communicate()[0]
  if not checkOutput(output):
    print testcase_path
    print output
    print '-' * 80
```

就是不断调用 程序 处理刚刚生成的测试用例，根据执行的输出结果中 是否有 `Segmentation fault` 和 `error`来判断是否触发了漏洞。

ps: 由于用于变异样本的选取 和 样本的变异方式是随机的，可能需要重复多次 **样本生成 && fuzz** 才能找到 `crash`

写个 `bash` 脚本，不断重复即可

```
#!/bin/bash
while [ "0" -lt "1" ]
do
  rm -rf ./work/
  ./generate_testcases.py
  ./run_fuzzing.py
done
```

Helloworld-For-libFuzzer

安装

本节相关资源文件位于:

https://github.com/Dor1s/libfuzzer-workshop/tree/master/lessons/04

首先先把 `libFuzzer` 安装一下

首先

```
git clone https://github.com/Dor1s/libfuzzer-workshop.git
sudo ln -s /usr/include/asm-generic /usr/include/asm
apt-get install gcc-multilib
```

然后进入 `libfuzzer-workshop/` ， 执行 `checkout_build_install_llvm.sh` 安装好 `llvm`.

然后进入 `libfuzzer-workshop/libFuzzer/Fuzzer/` ，执行 `build.sh` 编译好 `libFuzzer`。

如果编译成功，会生成 `libfuzzer-workshop/libFuzzer/Fuzzer/libFuzzer.a`

实战

这一节中主要使用 `libFuzzer` 对 `vulnerable_functions.h` 中实现的几个有漏洞的 **函数** 进行 `fuzz`

VulnerableFunction1

```
bool VulnerableFunction1(const uint8_t* data, size_t size) {
  bool result = false;
  if (size >= 3) {
    result = data[0] == 'F' &&
             data[1] == 'U' &&
             data[2] == 'Z' &&
             data[3] == 'Z';
  }

  return result;
}
```

这个函数有两个参数，第一个参数 `data` 是 `uint8_t*` 类型的，说明 `data` 应该是指向了一个缓冲区， `size` 应该是缓冲区的大小，如果 `size >=3` , 会访问 `data[3]` , 越界访问了。

进行 fuzz 的第一步是 实现一个 入口点，用来接收 libFuzzer 生成的 测试用例（比特序列）。

示例

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

对于 `LLVMFuzzerTestOneInput` 有一些要注意的 `tips`

- `data` 是 `libFuzzer` 生成的 测试数据， `size` 是数据的长度
- `fuzz` 引擎会在一个进程中进行多次 `fuzz`， 所以其效率非常高
- 要能处理各种各样的输入 （空数据， 大量的 或者 畸形的数据...）
- 内部不会调用 `exit()`
- 如果使用多线程的话，在函数末尾要把 线程 `join`

对于 `VulnerableFunction1` ， 直接把 `libFuzzer` 传过来的数据，传给 `VulnerableFunction1` 即可

first_fuzzer.cc

```
// Copyright 2016 Google Inc. All Rights Reserved.
// Licensed under the Apache License, Version 2.0 (the "License");

#include <stdint.h>
#include <stddef.h>

#include "vulnerable_functions.h"

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  VulnerableFunction1(data, size);
  return 0;
}
```

然后用 `clang++` 编译

```
clang++ -g -std=c++11 -fsanitize=address -fsanitize-coverage=trace-pc-guard \
    first_fuzzer.cc ../../libFuzzer/Fuzzer/libFuzzer.a \
    -o first_fuzzer
```

- `-fsanitize=address`： 表示使用 AddressSanitizer
- `-fsanitize-coverage=trace-pc-guard`: 为 `libfuzzer` 提供代码覆盖率信息

然后运行

```
haclh@ubuntu:~/vmdk_kernel/libfuzzer-workshop-master/lessons/04$ ./first_fuzzer
INFO: Seed: 1608565063
INFO: Loaded 1 modules (37 guards): [0x788ec0, 0x788f54),
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
```

```
#0  READ units: 1
#1  INITED cov: 3 ft: 3 corp: 1/1b exec/s: 0 rss: 11Mb
#3  NEW    cov: 4 ft: 4 corp: 2/4b exec/s: 0 rss: 12Mb L: 3 MS: 2 InsertByte-InsertByte-
#3348   NEW    cov: 5 ft: 5 corp: 3/65b exec/s: 0 rss: 12Mb L: 61 MS: 2 ChangeByte-InsertRepeatedBytes-
#468765 NEW    cov: 6 ft: 6 corp: 4/78b exec/s: 0 rss: 49Mb L: 13 MS: 4 CrossOver-ChangeBit-EraseBytes-ChangeByte-
#564131 NEW    cov: 7 ft: 7 corp: 5/97b exec/s: 0 rss: 56Mb L: 19 MS: 5 InsertRepeatedBytes-InsertByte-ChangeByte-InsertByte-InsertByte-
=================================================================
==32049==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200072bb93 at pc 0x000000528540 bp 0x7ffdb3439100 sp 0x7ffdb34390f8
READ of size 1 at 0x60200072bb93 thread T0
    ....................................................
    ....................................................
    ....................................................

0x60200072bb93 is located 0 bytes to the right of 3-byte region [0x60200072bb90,0x60200072bb93)
allocated by thread T0 here:
   ....................................................
   ....................................................
   ....................................................

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/hac1h/vmdk_kernel/libfuzzer-workshop-master/lessons/04/./vulnerable_functions.h:22:14 in VulnerableFunction1(unsigned char
Shadow bytes around the buggy address:
  0x0c04800dd720: fa fa fd fd fa fa fd fa fa fa fd fa fa fa fd fa
  0x0c04800dd730: fa fa fd fd fa fa fd fd fa fa fd fd fa fa fd fd
  0x0c04800dd740: fa fa fd fa fa fa fd fa fa fa fd fa fa fa fd fa
  0x0c04800dd750: fa fa fd fa fa fa fd fa fa fa fd fa fa fa fd fa
  0x0c04800dd760: fa fa fd fa fa fa fd fd fa fa fd fd fa fa fd fd
=>0x0c04800dd770: fa fa[03]fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c04800dd780: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c04800dd790: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c04800dd7a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c04800dd7b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c04800dd7c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
....................................................
....................................................
==32049==ABORTING
MS: 1 CrossOver-; base unit: 38a223b0988bd9576fb17f5947af80b80203f0ef
0x46,0x55,0x5a,
FUZ
artifact_prefix='./'; Test unit written to ./crash-0eb8e4ed029b774d80f2b66408203801cb982a60
Base64: R1Va
```

正常的话应该可以看到类似上面的输出，这里对其中的一些信息解析一下

- `Seed: 1608565063` 说明这次的种子数据
- `-max_len is not provided, using 64` ， `-max_len` 用于设置最大的数据长度，默认为 `64`
- 接下来 `#` 开头的行是 `fuzz` 过程中找到的路径信息
- 倒数第二行是触发漏洞的测试用例

使用

```
ASAN_OPTIONS=symbolize=1 ./first_fuzzer ./crash-0eb8e4ed029b774d80f2b66408203801
# ASAN_OPTIONS=symbolize=1 用于显示 栈的符号信息
```

重现 `crash`.

VulnerableFunction2

```
constexpr auto kMagicHeader = "ZN_2016";
constexpr std::size_t kMaxPacketLen = 1024;
constexpr std::size_t kMaxBodyLength = 1024 - sizeof(kMagicHeader);

bool VulnerableFunction2(const uint8_t* data, size_t size, bool verify_hash) {
  if (size < sizeof(kMagicHeader))
    return false;

  std::string header(reinterpret_cast<const char*>(data), sizeof(kMagicHeader));

  std::array<uint8_t, kMaxBodyLength> body;
```

```
  if (strcmp(kMagicHeader, header.c_str()))
    return false;

  auto target_hash = data[--size];

  if (size > kMaxPacketLen)
    return false;

  if (!verify_hash)
    return true;

  std::copy(data, data + size, body.data());
  auto real_hash = DummyHash(body);
  return real_hash == target_hash;
}
```

代码量比第一个要复杂了一些，不管那么多先 `fuzz`， 首先看看这个函数的参数， 比 `VulnerableFunction1` 多了一个 `bool` 类型的参数，所以 fuzz 脚本比 `VulnerableFunction1` 中的加一点修改即可。

```
#include "vulnerable_functions.h"

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  bool verify_hash_flags[] = { false, true };

  for (auto flag : verify_hash_flags)
    VulnerableFunction2(data, size, flag);
  return 0;
}
```

`fuzz` 测试的目标就是尽可能的测试所有代码路径，又 `VulnerableFunction2` 的第 3个参数是 `bool` 型的（有两种可能）， 那我们就对每一个测试用例都用 `{false, true}`测试一遍以获取更多的测试路径。

首先编译一下

```
clang++ -g -std=c++11 -fsanitize=address -fsanitize-coverage=trace-pc-guard \
    second_fuzzer.cc ../../libFuzzer/Fuzzer/libFuzzer.a \
    -o second_fuzzer
```

直接执行会得到类似下面的结果

```
haclh@ubuntu:~/vmdk_kernel/libfuzzer-workshop-master/lessons/04$ ./second_fuzzer
INFO: Seed: 4141499174
INFO: Loaded 1 modules (39 guards): [0x788f00, 0x788f9c),
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
#0  READ units: 1
#1  INITED cov: 5 ft: 5 corp: 1/1b exec/s: 0 rss: 11Mb
#26 NEW    cov: 6 ft: 6 corp: 2/29b exec/s: 0 rss: 12Mb L: 28 MS: 5 ChangeByte-ChangeBit-ShuffleBytes-ChangeByte-InsertRepeatedBytes-
#1840   NEW    cov: 26 ft: 26 corp: 3/70b exec/s: 0 rss: 12Mb L: 41 MS: 4 InsertRepeatedBytes-ShuffleBytes-CMP-CMP- DE: "\x00\x00\x00\x00\x00\x00\x00\x00"-"ZN_2016"-
#1048576    pulse  cov: 26 ft: 26 corp: 3/70b exec/s: 174762 rss: 103Mb
#2097152    pulse  cov: 26 ft: 26 corp: 3/70b exec/s: 149796 rss: 194Mb
^C==32554== libFuzzer: run interrupted; exiting
```

可以看到 `libfuzzer` 到后面基本找不到新的路径了，一直 `pulse` 。回到 `VulnerableFunction2` 我们发现函数可以处理的最大数据长度是 1024，所以给 fuzzer 加个参数设置一下最大数据长度。

```
./second_fuzzer -max_len=1024
```

```
haclh@ubuntu:~/vmdk_kernel/libfuzzer-workshop-master/lessons/04$ ./second_fuzzer -max_len=1024
INFO: Seed: 1482916821
INFO: Loaded 1 modules (39 guards): [0x788f00, 0x788f9c),
INFO: A corpus is not provided, starting from an empty corpus
#0      READ units: 1
#1      INITED cov: 5 ft: 5 corp: 1/1b exec/s: 0 rss: 11Mb
#4      NEW    cov: 6 ft: 6 corp: 2/56b exec/s: 0 rss: 12Mb L: 55 MS: 3 ChangeByte-ShuffleBytes-InsertRepeatedBytes-
#22085  NEW    cov: 26 ft: 26 corp: 3/68b exec/s: 0 rss: 19Mb L: 12 MS: 4 ChangeBit-ChangeByte-CMP-CMP- DE: "\x00\x00\x00\x00"-"ZN_2016"-
==========================================================
==32569==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffcd304f4a8 at pc 0x0000004e93d1 bp 0x7ffcd304eef0 sp 0x7ffcd304e6a0
WRITE of size 1023 at 0x7ffcd304f4a8 thread T0
    #0 0x4e93d0 in __asan_memmove /home/haclh/vmdk_kernel/libfuzzer-workshop-master/src/llvm/projects/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cc:31
    #1 0x529320 in unsigned char* std::__copy_move<false, true, std::random_access_iterator_tag>::__copy_m<unsigned char>(unsigned char const*, unsigned char const*, unsigned char*) /usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/bits/stl_algobase.h:384:6
    #2 0x5291f2 in unsigned char* std::__copy_move_a<false, unsigned char const*, unsigned char*>(unsigned char const*, unsigned char const*, unsigned char*) /usr/lib/gcc/x86_64-linux-gnu/5.4.0/include/c++/5.4.0/bits/stl_algobase.h:401:14
    #3 0x529150 in unsigned char* std::__copy_move_a2<false, unsigned char const*, unsigned char*>(unsigned char const*, unsigned char const*, unsigned char*) /usr/lib/gcc/x86_64-linux-gnu/5.4.0/include/c++/5.4.0/bits/stl_algobase.h:438:18
    #4 0x528f63 in unsigned char* std::copy<unsigned char const*, unsigned char*>(unsigned char const*, unsigned char const*, unsigned char*) /usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/bits/stl_algobase.h:470:15
    #5 0x528979 in VulnerableFunction2(unsigned char const*, unsigned long, bool) /home/haclh/vmdk_kernel/libfuzzer-workshop-master/lessons/04/./vulnerable_functions.h:61:3
    #6 0x528e10 in LLVMFuzzerTestOneInput /home/haclh/vmdk_kernel/libfuzzer-workshop-master/lessons/04/second_fuzzer.cc:7:5
    #7 0x533864 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerLoop.cpp:451:13
    #8 0x533a91 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerLoop.cpp:408:3
    #9 0x53445c in fuzzer::Fuzzer::MutateAndTestOne() /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerLoop.cpp:587:30
    #10 0x5346c7 in fuzzer::Fuzzer::Loop() /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerLoop.cpp:615:5
    #11 0x52c994 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerDriver.cpp:644:6
    #12 0x5294b0 in main /home/haclh/vmdk_kernel/libfuzzer-workshop-master/libFuzzer/Fuzzer/./FuzzerMain.cpp:20:10
    #13 0x7f94b55e082f in __libc_start_main /build/glibc-CL5G7W/glibc-2.23/csu/../csu/libc-start.c:291
    #14 0x41d518 in _start (/home/haclh/vmdk_kernel/libfuzzer-workshop-master/lessons/04/second_fuzzer+0x41d518)
```

可以看到检测到了 栈溢出，触发漏洞的指令位于

vulnerable_functions.h:61:3

跟到该文件内查看，发现是

```
std::copy(data, data + size, body.data());
```

触发了漏洞，漏洞产生的原因在于， body 的 buf 的大小为 kMaxBodyLength ， 而这里可以往 body 的 buf 里面写入最多 kMaxPacketLen 字节。

```
constexpr std::size_t kMaxPacketLen = 1024;
constexpr std::size_t kMaxBodyLength = 1024 - sizeof(kMagicHeader);
```

溢出。

VulnerableFunction3

```
constexpr std::size_t kZn2016VerifyHashFlag = 0x0001000;

bool VulnerableFunction3(const uint8_t* data, size_t size, std::size_t flags) {
  bool verify_hash = flags & kZn2016VerifyHashFlag;
  return VulnerableFunction2(data, size, verify_hash);
}
```

就是动态生成了 verify_hash 的值，然后传给 了 VulnerableFunction2。

我们也照着来就行

```
#include "vulnerable_functions.h"

#include <functional>
#include <string>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {

    VulnerableFunction3(data, size, 0x0001000); // 触发 flag = true
    VulnerableFunction3(data, size, 0x1000000); // 触发 flag = false
    return 0;
}
```

编译

```
clang++ -g -std=c++11 -fsanitize=address -fsanitize-coverage=trace-pc-guard \
    third_fuzzer.cc ../../libFuzzer/Fuzzer/libFuzzer.a \
    -o third_fuzzer
```

运行

```
haclh@ubuntu:~/vmdk_kernel/libfuzzer-workshop-master/lessons/04$ ./third_fuzzer -max_len=1024
INFO: Seed: 4085125045
INFO: Loaded 1 modules (37 guards): [0x788ec0, 0x788f54),
INFO: A corpus is not provided, starting from an empty corpus
#0      READ units: 1
#1      INITED cov: 4 ft: 4 corp: 1/1b exec/s: 0 rss: 11Mb
#3      NEW    cov: 5 ft: 5 corp: 2/77b exec/s: 0 rss: 12Mb L: 76 MS: 2 ChangeByte-InsertRepeatedBytes-
#25571  NEW    cov: 25 ft: 25 corp: 3/93b exec/s: 0 rss: 21Mb L: 16 MS: 5 ChangeBit-CMP-CMP-ChangeASCIIInt-CMP- DE: "ZN_2016"-"\x00\x00\x00\x00\x00\x00\x00\x00"-"ZN_2016"-
==========================================================
==32645==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd53cfae88 at pc 0x0000004e93d1 bp 0x7ffd53cfa8d0 sp 0x7ffd53cfa080
WRITE of size 1023 at 0x7ffd53cfae88 thread T0
    #0 0x4e93d0 in __asan_memmove /home/haclh/vmdk_kernel/libfuzzer-workshop-master/src/llvm/projects/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cc:31
    #1 0x529120 in unsigned char* std::__copy_move<false, true, std::random_access_iterator_tag>::__copy_m<unsigned char>(unsigned char const*, unsigned char const*, unsigned char*) /usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/bits/stl_algobase.h:384:6
```

总结

简单理解 libfuzzer 。如果我们要 fuzz 一个程序，找到一个入口函数，然后利用

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    .......
    .......
}
```

接口，我们可以拿到 libfuzzer 生成的 **测试数据以及测试数据的长度**，我们的任务就是**把这些生成的测试数据 传入到目标程序中 让程序来处理 测试数据， 同时要尽可能的触发更多的代码逻辑。**

实战两个简单的CVE

CVE-2014-0160 (openssl 心脏滴血漏洞)

libfuzzer 是用于 fuzz 某个函数的，所以我们先编译好目标代码库， 然后在根据要 fuzz 的功能编写 fuzzer 函数。

本节资源文件位于

https://github.com/Dor1s/libfuzzer-workshop/tree/master/lessons/05

首先用 clang 编译 openssl.

```
./config
make clean
make CC="clang -O2 -fno-omit-frame-pointer -g -fsanitize=address -fsanitize-coverage=trace-pc-guard,trace-cmp,trace-gep,trace-div" -j$(nproc)
```

主要是为了加上 `AddressSanitizer` ，用于检测程序中出现的异常（uaf, 堆溢出，栈溢出等漏洞)

常用内存错误检测工具

`AddressSanitizer`: 检测 `uaf`, 缓冲区溢出， `stack-use-after-return`, `container-overflow`

`MemorySanitizer`： 检测未初始化内存的访问

`UndefinedBehaviorSanitizer`： 检测一些其他的漏洞，整数溢出，类型混淆等

然后写 fuzzer 的逻辑

```
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <assert.h>
#include <stdint.h>
#include <stddef.h>

#ifndef CERT_PATH
# define CERT_PATH
#endif

SSL_CTX *Init() {
  SSL_library_init();
  SSL_load_error_strings();
  ERR_load_BIO_strings();
  OpenSSL_add_all_algorithms();
  SSL_CTX *sctx;
  assert (sctx = SSL_CTX_new(TLSv1_method()));
  /* These two file were created with this command:
      openssl req -x509 -newkey rsa:512 -keyout server.key \
     -out server.pem -days 9999 -nodes -subj /CN=a/
  */
  assert(SSL_CTX_use_certificate_file(sctx, CERT_PATH "server.pem",
                                      SSL_FILETYPE_PEM));
  assert(SSL_CTX_use_PrivateKey_file(sctx, CERT_PATH "server.key",
                                     SSL_FILETYPE_PEM));
  return sctx;
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  static SSL_CTX *sctx = Init();
  SSL *server = SSL_new(sctx);
  BIO *sinbio = BIO_new(BIO_s_mem());
  BIO *soutbio = BIO_new(BIO_s_mem());
  SSL_set_bio(server, sinbio, soutbio);
  SSL_set_accept_state(server);
  BIO_write(sinbio, data, size);
  SSL_do_handshake(server);
  SSL_free(server);
  return 0;
}
```

感觉用 `libfuzzer` 的话，**我们需要做的工作就是根据目标程序的逻辑，把 `libfuzzer`生成的 测试数据 传递 给 目标程序去处理， 然后在编译时采取合适的 `Sanitizer` 用于检测运行时出现的内存错误。**

比如上面就是模拟了 SSL 握手的逻辑，然后把 `libfuzzer` 生成的 测试数据作为握手包传递给 `openssl` 。

编译之

```
clang++ -g openssl_fuzzer.cc -O2 -fno-omit-frame-pointer -fsanitize=address      -fsanitize-coverage=trace-pc-guard,trace-cmp,trace-gep,trace-div      -Iopenssl1.0.1f/include open
```

运行然后就会出现 `crash` 信息了

```
haclh@ubuntu:~/vmdk_kernel/libfuzzer-workshop-master/lessons/05$ ./openssl_fuzzer ./corpus/
INFO: Seed: 1472290074
INFO: Loaded 1 modules (33464 guards): [0xc459b0, 0xc66490),
Loading corpus dir: ./corpus/
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
#0  READ units: 1
#1  INITED cov: 1513 ft: 396 corp: 1/1b exec/s: 0 rss: 22Mb
```

```
.............................
.............................
.............................
#68027  NEW    cov: 1592 ft: 703 corp: 28/1208b exec/s: 13605 rss: 363Mb L: 35 MS: 1 EraseBytes-
=================================================================
==35462==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x0000004e8f7d bp 0x7ffd58180520 sp 0x7ffd5817fcd0
READ of size 65535 at 0x629000009748 thread T0
    #0 0x4e8f7c in __asan_memcpy /home/haclh/vmdk_kernel/libfuzzer-workshop-master/src/llvm/projects/compiler-rt/lib/asan/asan_interceptors_memintrinsics.cc:23
    #1 0x5353f6 in tls1_process_heartbeat /home/haclh/vmdk_kernel/libfuzzer-workshop-master/lessons/05/openssl1.0.1f/ssl/t1_lib.c:2586:3
```

可以看到在 `tls1_process_heartbeat` 中 触发了堆溢出 （`heap-buffer-overflow`）

CVE-2016-5180 （c-ares 堆溢出）

本节相关文件位于

https://github.com/Dor1s/libfuzzer-workshop/tree/master/lessons/06

和前面一样，首先编译一下这个库。

```
tar xzvf c-ares.tgz
cd c-ares

./buildconf
./configure CC="clang -O2 -fno-omit-frame-pointer -g -fsanitize=address -fsanitize-coverage=trace-pc-guard,trace-cmp,trace-gep,trace-div"
make CFLAGS=
```

然后 `fuzzer` 的逻辑就非常简单了

```
#include <ares.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  unsigned char *buf;
  int buflen;
  std::string s(reinterpret_cast<const char *>(data), size);
  ares_create_query(s.c_str(), ns_c_in, ns_t_a, 0x1234, 0, &buf, &buflen, 0);
  ares_free_string(buf);
  return 0;
}
```

把 `libfuzzer` 传过来的数据，转成 `char *`，然后扔给 `ares_create_query` 进行处理。

运行就有 `crash` 了。

总结

感觉`libfuzzer` 已经把 一个 `fuzzer` 的核心（样本生成引擎和异常检测系统） 给做好了， 我们需要做的是根据目标程序的逻辑，把 `libfuzzer` 生成的数据，交给目标程序处理。

libFuzzer进阶

前面介绍了 `libFuzzer` 的一些简单的使用方法，下面以 `fuzz libxml2` 为例，介绍一些 `libFuzzer` 的高级用法。

相关文件位于

https://github.com/Dor1s/libfuzzer-workshop/tree/master/lessons/08

Start fuzz

首先把 `libxml2` 用 `clang` 编译

```
tar xzf libxml2.tgz
cd libxml2

./autogen.sh

export FUZZ_CXXFLAGS="-O2 -fno-omit-frame-pointer -g -fsanitize=address \
    -fsanitize-coverage=edge,indirect-calls,trace-cmp,trace-div,trace-gep,trace-pc-guard"

CXX="clang++ $FUZZ_CXXFLAGS" CC="clang $FUZZ_CXXFLAGS" \
    CCLD="clang++ $FUZZ_CXXFLAGS"  ./configure
make -j$(nproc)
```

然后写个 `fuzzer`，这里选择 测试 `xmlReadMemory`

```
#include "libxml/parser.h"
```

```
void ignore (void* ctx, const char* msg, ...) {
  // Error handler to avoid spam of error messages from libxml parser.
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  xmlSetGenericErrorFunc(NULL, &ignore);

  if (auto doc = xmlReadMemory(reinterpret_cast<const char*>(data),
                               static_cast<int>(size), "noname.xml", NULL, 0)) {
    xmlFreeDoc(doc);
  }

  return 0;
}
```

然后编译之。

```
export FUZZ_CXXFLAGS="-O2 -fno-omit-frame-pointer -g -fsanitize=address \
    -fsanitize-coverage=edge,indirect-calls,trace-cmp,trace-div,trace-gep,trace-pc-guard"
clang++ -std=c++11 xml_read_memory_fuzzer.cc $FUZZ_CXXFLAGS -Ilibxml2/include \
    libxml2/.libs/libxml2.a /usr/lib/x86_64-linux-gnu/liblzma.a ../../libFuzzer/Fuzzer/libFuzzer.a -lz \
    -o xml_read_memory_fuzzer
```

ps: `/usr/lib/x86_64-linux-gnu/liblzma.a` 是 `liblzma.a` 的路径，需要安装 `liblzma-dev`:

```
apt-get install liblzma-dev
```

编译好 fuzzer 后，我们运行它:

```
mkdir corpus1
./xml_read_memory_fuzzer -max_total_time=300 -print_final_stats=1 corpus1
```

其中的一些参数做个解释

- `-max_total_time`：设置最长的运行时间， 单位是 秒， 这里是 `300s`，也就是 `5` 分钟
- `-print_final_stats`: 执行完 `fuzz` 后 打印统计信息
- `corpus1`： `fuzzer` 程序可以有多个目录作为参数，此时 `fuzzer` 会递归遍历所有目录，把目录中的文件读入最为样本数据传给测试函数，同时会把那些可以产生新的的代码路径的样本保存到第一个目录里面。

运行完后会得到类似下面的结果

```
###### Recommended dictionary. ######
"X\x00\x00\x00\x00\x00\x00\x00" # Uses: 1228
"prin" # Uses: 1353
.........................
.........................
.........................
"U</UTrri\x09</UTD" # Uses: 61
###### End of recommended dictionary. ######
Done 1464491 runs in 301 second(s)
stat::number_of_executed_units: 1464491
stat::average_exec_per_sec:     4865
stat::new_units_added:          1407
stat::slowest_unit_time_sec:    0
stat::peak_rss_mb:              407
```

开始由 `####` 夹着的是 `libfuzzer` 在 `fuzz` 过程中挑选出来的 `dictionary`， 同时还给出了使用的次数，这些 `dictionary` 可以在以后 `fuzz` 同类型程序时 节省 `fuzz` 的时间。

然后以 `stat:` 开头的是一些 fuzz 的统计信息， 主要看 `stat::new_units_added` 表示整个 `fuzz` 过程中触发了多少个代码单元。

可以看到直接 `fuzz`，5分钟 触发了 `1407` 个代码单元

使用 Dictionary 提升性能

`Dictionary` 貌似是 `afl` 中提出的， 具体可以看下面这篇文章

https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

我们知道基本上所有的程序都是**处理具有一定格式的数据**，比如 `xml` 文档， `png`图片等等。 这些数据中会有一些**特殊字符序列 （或者说关键字）**， 比如 在 `xml` 文档中 就有 `CDATA`, `<!ATTLIST` 等，**png图片** 就有 **png 图片头。**

如果我们事先就把这些 **字符序列** 列举出来， `fuzz` 直接使用这些关键字去 组合，就会就可以减少很多没有意义的 尝试，同时还有可能会走到更深的程序分支中去。

`Dictionary` 就是实现了这种思路。 `libfuzzer` 和 `afl` 使用的 `dictionary` 文件的语法是一样的， 所以可以直接拿 afl 里面的 `dictionary` 文件来给 `libfuzzer` 使用。

下面这个网址里面就有一些 `afl` 的 `dictionary` 文件

https://github.com/rc0r/afl-fuzz/tree/master/dictionaries

以 libfuzzer 官网 的示例介绍一下语法

```
# Lines starting with '#' and empty lines are ignored.
# Adds "blah" (w/o quotes) to the dictionary.
kw1="blah"
# Use \\ for backslash and \" for quotes.
kw2="\"ac\\dc\""
# Use \xAB for hex values
kw3="\xF7\xF8"
# the name of the keyword followed by '=' may be omitted:
"foo\x0Abar"
```

- `#` 开头的行 和 空行会被忽略
- `kw1=` 这些就类似于注释， 没有意义
- 真正有用的是由 `"` 包裹的**字串**，这些 **字串** 就会作为一个个的关键字， `libfuzzer` 会用它们进行组合来生成样本。

`libfuzzer` 使用 `-dict` 指定 `dict` 文件，下面使用 `xml.dict` 为 `dictionary`文件，进行 `fuzz`。

```
./xml_read_memory_fuzzer -dict=./xml.dict -max_total_time=300     -print_final_stats=1 corpus2
```

最终这种方式可以探测到 `2200+` 的代码单元， 效率提升还是很明显的。

```
###### End of recommended dictionary. ######
Done 1379646 runs in 301 second(s)
stat::number_of_executed_units: 1379646
stat::average_exec_per_sec:     4583
stat::new_units_added:          2215
stat::slowest_unit_time_sec:    0
stat::peak_rss_mb:              415
```

精简样本集

在上一节里面我们获得了很多的样本，其中有很多其实是重复的，可以使用 `libfuzzer` 把样本集进行精简。

```
mkdir corpus1_min
./xml_read_memory_fuzzer -merge=1 corpus1_min corpus1
```

`corpus1_min`: 精简后的样本集存放的位置

`corpus1`: 原始样本集存放的位置

可以得到类似的结果

```
03:20 haclh@ubuntu:08 $ ls corpus1 | wc -l
1403
03:20 haclh@ubuntu:08 $ ./xml_read_memory_fuzzer -merge=1 corpus1_min corpus1
INFO: Seed: 3775041129
.....................
.....................
.....................
#1024   pulse  cov: 1569 exec/s: 0 rss: 108Mb
MERGE-OUTER: succesfull in 1 attempt(s)
MERGE-OUTER: the control file has 4150228 bytes
MERGE-OUTER: consumed 2Mb (24Mb rss) to parse the control file
MERGE-OUTER: 928 new files with 5307 new features added
```

可以看到 `1403` 个样本被精简成了 `928` 个样本

生成代码覆盖率报告

使用 `-dump_coverage` 参数。

```
03:28 haclh@ubuntu:08 $ ./xml_read_memory_fuzzer corpus1_min -runs=0 -dump_coverage=1
INFO: Seed: 2354910000
..............
..............
./xml_read_memory_fuzzer.69494.sancov: 1603 PCs written
```

然后会生成一个 *.sancov 文件

```
03:26 haclh@ubuntu:08 $ ls *.sancov
xml_read_memory_fuzzer.69494.sancov
```

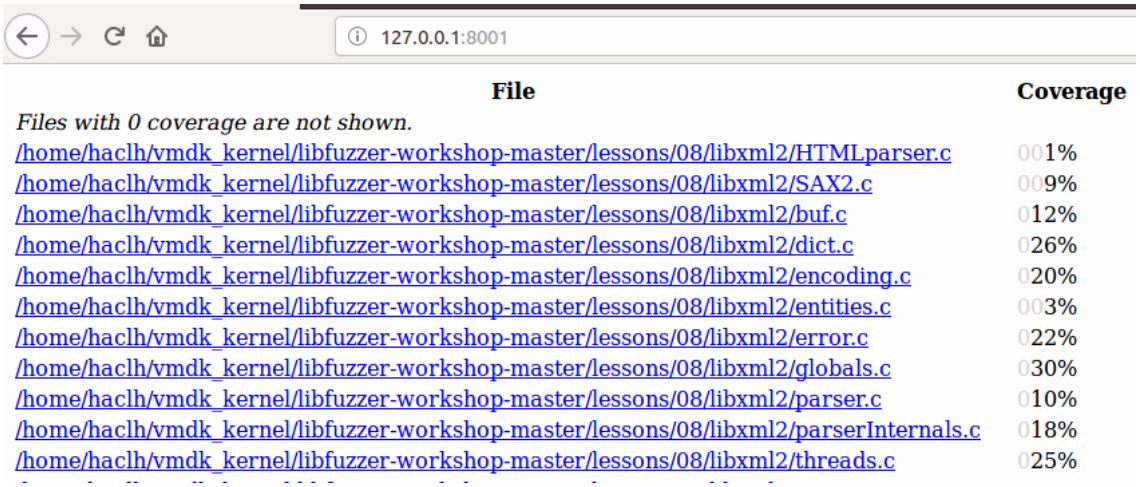
然后把它转换成 `.symcov`

```
sancov -symbolize xml_read_memory_fuzzer ./xml_read_memory_fuzzer.69494.sancov > xml_read_memory_fuzzer.symcov
```

然后使用 coverage-report-server 解析这个文件。

```
03:29 haclh@ubuntu:08 $ python3 coverage-report-server.py --symcov xml_read_memory_fuzzer.symcov     --srcpath libxml2
Loading coverage...
Serving at 127.0.0.1:8001
```

用浏览器访问之



通过这个功能，我们可以非常直观的看到每个源文件的覆盖率。

Fuzz xmlRegexpCompile

前面已经 `fuzz` 了 `xmlReadMemory`，这里 `fuzz` 另外一个函数 `xmlRegexpCompile`

看看 `fuzz` 代码

```
#include "libxml/parser.h"
#include "libxml/tree.h"
#include "libxml/xmlversion.h"

void ignore (void * ctx, const char * msg, ...) {
  // Error handler to avoid spam of error messages from libxml parser.
}
// Entry point for LibFuzzer.
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  xmlSetGenericErrorFunc(NULL, &ignore);
  std::vector<uint8_t> buffer(size + 1, 0);
  std::copy(data, data + size, buffer.data());
  xmlRegexpPtr x = xmlRegexpCompile(buffer.data());
  if (x)
    xmlRegFreeRegexp(x);
  return 0;
}
```

就是把 数据用 `std::vector` 做了个中转，喂给 `xmlRegexpCompile` 函数。

编译之

```
export FUZZ_CXXFLAGS="-O2 -fno-omit-frame-pointer -g -fsanitize=address \
    -fsanitize-coverage=edge,indirect-calls,trace-cmp,trace-div,trace-gep,trace-pc-guard"
clang++ -std=c++11 xml_compile_regexp_fuzzer.cc $FUZZ_CXXFLAGS -Ilibxml2/include \
    libxml2/.libs/libxml2.a /usr/lib/x86_64-linux-gnu/liblzma.a ../../libFuzzer/Fuzzer/libFuzzer.a -lz \
    -o xml_compile_regexp_fuzzer
```

运行

```
mkdir corpus3
./xml_compile_regexp_fuzzer -dict=./xml.dict corpus3
```

然后过一会就会出现 `crash` 了。

参考

https://github.com/Dor1s/libfuzzer-workshop/