Finding Security Vulnerabilities with Fuzzing

Hardik Shah

@hardik05

McAfee™

Agenda

- Introduction
- Vulnerabilities
  - Different Types of Vulnerabilities.
  - Manually Identifying the vulnerabilities in C Program.
- Fuzzing and Bug Hunting
  - Different types of Fuzzer
  - Basic Block, Code Coverage and Instrumentation
- What is AFL?
  - How it works?
  - Fork Server vs Persistent Mode
  - Fuzzing Strategies
- Sanitizers – ASAN,UBSAN, MSAN,TSAN
- Corpus Collection & Crash Triage
- Using AFL & Honggfuzz
  - Hands on: How to install AFL and HonggFuzz
  - Hands on: Fuzzing sample C program with AFL & HonggFuzz
  - Hands on: Root cause analysis using GDB
  - Hands on: Crash Triage using crashwalk
  - Fuzzing real world programs – TcpDump
- Conclusion

About Me

- Security Researcher @ McAfee
  - Vulnerability, exploit, malware analysis.
- Fuzzing and bug hunting.
  - Have around 24 CVEs in my name.
  - MSRC 2018-19 Most Valuable Researcher.
  - MSRC Q1 2020 Top Contributing Researcher.
- Blogs:
  - https://www.mcafee.com/blogs/author/hardik-shah/
- Twitter:
  - @hardik05

Vulnerability

- Bug in the software.
  - Ex: if you send get request where uri length is more then 1000 bytes of data to a web server, it will crash.
- Can be used to perform various unwanted activities:
  - Remote code execution – someone can execute malicious code.
  - Denial of service – can crash the software or entire system.
  - Privilege Escalation – from local account to admin account.

Can be converted to Exploits

- How they can be used in malicious activity?
  - Leads to system compromise, ransomware, trojan, botnet, bitcoin miners, data theft etc.
- Common types of vulnerabilities
  - Integer overflow/underflow, stack/heap overflow, out of bound read/write, use after free, double free

Different Types of Vulnerabilities

Integer overflow/Underflow
Stack/Heap Overflow
Out of bound Read/Write
Use after free/Double free

Integer Overflow

- What it is?
- Vulnerability in integer data types, they way they store data.
- Example:
  - unsigned int j;
  - Int i;
  - Size of integer = 4 bytes
  - Max Value = 11111111111111111111111111111111
  - 2^32
  - Signed vs unsigned?
    - MSB is used for signedness.
    - 1= 00000000000000000000000000000001
    - -1 = 10000000000000000000000000000001
    - Max value for signed int = 0x7FFFFFFF
    - Max value for unsigned int = 0xFFFFFFFF
- What happens in this case?
  - Int i;
  - Unsigned int j;
  - j = 0xFFFFFFFF + 1
    - Result will become 0, carry 1 bit will be truncated.
  - i = 0x7FFFFFFF + 1
    - Result will become -0x8000000 (negative number)

- 0XFFFFFFFF + 1 =

11111111111111111111111111111111

\+ 1

\---

100000000000000000000000000000000

- Integer overflow, very small number as carry will be truncated.
- Will become 0 in this case.

```
int var1,var2;
int size1 =  var1+ var2;
char* buff1=(char*)malloc(size1);
memcpy(buff1,data,sizeof(data));
```

```
    uint32_t i_len;
    MP4_READBOX_ENTER( MP4_Box_data_rdrf_t );

    MP4_GETVERSIONFLAGS( p_box->data.p_rdrf );
    MP4_GETFOURCC( p_box->data.p_rdrf->i_ref_type );
    MP4_GET4BYTES( i_len );
    if( i_len > 0 )
    {
        uint32_t i;
        p_box->data.p_rdrf->psz_ref = malloc( i_len  + 1);
        for( i = 0; i < i_len; i++ )
        {
            MP4_GET1BYTE( p_box->data.p_rdrf->psz_ref[i] );
        }
        p_box->data.p_rdrf->psz_ref[i_len] = '\0';
```

Ref: https://mailman.videolan.org/pipermail/vlc/2008-March/015488.html

Integer Underflow

- What it is?
  - Size of integer = 4 bytes
  - Signed vs unsigned?
    - Range for signed int= -0x80000000 to 0x7FFFFFFF
    - Range for unsigned int = 0 to 0xFFFFFFFF
- What happens in this case?
  - Int i;
  - i = -0x80000000 – 1 = 0x7FFFFFFF
  - i = highest possible positive number.

- -0x80000000 - 1 =
- 10000000000000000000000000000000

  - 1

01111111111111111111111111111111

- integer underflow, very large number.
- Change in signedness. (-) to (+)

```
int var1,var2;
int size1 =  var1 - var2; → integer underflow
char* buff1=(char*)malloc(size1);
memcpy(buff1,data,sizeof(data));
```

Stack overflow/Heap Overflow

- Stack Overflow
  - Local variable are stored in Stack
  - Finite size
  - Overflow in local variable, can corrupt other data on stack.
  - Example:
    - Function foo(){

      char var1[8];

      char var2[100];

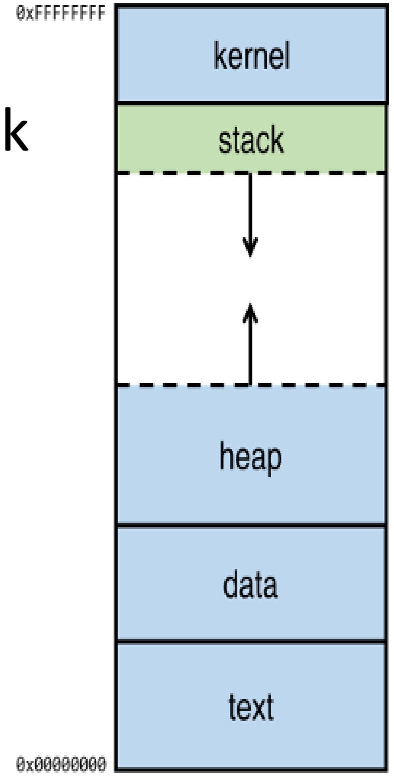
      memcpy(var1,var2,sizeof(var2)); → stack overflow

      }

```
/** Sender name is set to max length of MAX_CNAME (128), line: 446 **/
char new_sender_name[MAX_CNAME];

/** name_length is read from the RTSP header, line: 489 **/
int8_t name_length = rtcp_sdes_get_name_length(buf);

/** memcpy new_sender_name with name_length bytes, line: 525 **/
memcpy(new_sender_name, buf + RTCP_SDES_SIZE, name_length);
```



- Heap Overflow
  - Dynamic memory allocation
  - Allocated from heap
  - Overflow in heap can corrupt other data in heap.
  - Example:

    Char *var1 = (char*) malloc(8);

    Char var2[100];

    memcpy(var1,var2,sizeof(var2));

    → heap overflow

Ref and Img credits:
https://www.coengoedegebure.com/buffer-overflow-attacks-explained/
https://hackerone.com/reports/489102

Out of bound Read/Write

- Stack Out of Bound Read/Write
  - Memory access or write operation at beyond the allowed limits of Stack memory.
  - Can cause access violation.
  - Example:
    - char a[10];
    - char b;
    - b=a[100]; →OOB Read
    - a[100] = ‘c’; →OOB Write

```
                    buffer_caret++;

-                   for (i = 0; i < encoded_pixels; i++) {
-                       for( j = 0; j < pixel_block_size; j++, bitmap_caret++ ) {
-                           tga->bitmap[ bitmap_caret ] = decompression_buffer[
buffer_caret + j ];
```

Ref: https://github.com/libgd/libgd

- Heap Out of Bound Read/Write
  - Memory access or write operation at beyond the allowed limits of heap memory.
  - Can cause access violation.
  - Example:
    - char* a = (char*)malloc(10);
    - char b;
    - b=a[100]; → OOB read
    - a[100] =‘c’; →OOB Write

Use After Free/Double Free

- Use After Free
  - Using a memory after it has been freed.
  - Can cause program crash or unexpected behavior.
  - Example:

    char *buff = (char*)malloc(10);

    free(buff);

    buff[0]='c'; → use after free

```
TIFFFileName(input));
t2p->t2p_error = T2P_ERR_ERROR;
_TIFFfree(buffer);
} else {
buffer=samplebuffer;
t2p->tiff_datasize *= t2p->tiff_samplesperpixel;
}
t2p_sample_realize_palette(t2p, buffer);
```

Ref: https://www.asmail.be/msg0055359936.html

- Double Free
  - Freeing allocated memory multiple time.
  - Can cause program to crash.
  - Example:

    char *buff = (char*)malloc(10);

    free(buff);

    free(buff); → double free!

McAfee

What we have learned So far?

Different types of vulnerabilities

Integer overflow/underflow, stack/heap buffer overflow, use after free, double free

Hands on: Manually Identify Vulnerabilities!

```
struct Image
{
    char header[4];
    int width;
    int height;
    char data[10];
};

int size1 = img.width + img.height;
char* buff1=(char*)malloc(size1);

memcpy(buff1,img.data,sizeof(img.data));
free(buff1);

int size2 = img.width - img.height;
char* buff2=(char*)malloc(size2);
memcpy(buff2,img.data,sizeof(img.data));

if (size1/2==0){
    free(buff1);
    }
else{
if(size1 == 123456){
    buff1[0]='a';
    }
```

Integer Overflow

Integer underflow

Double Free

Use After Free

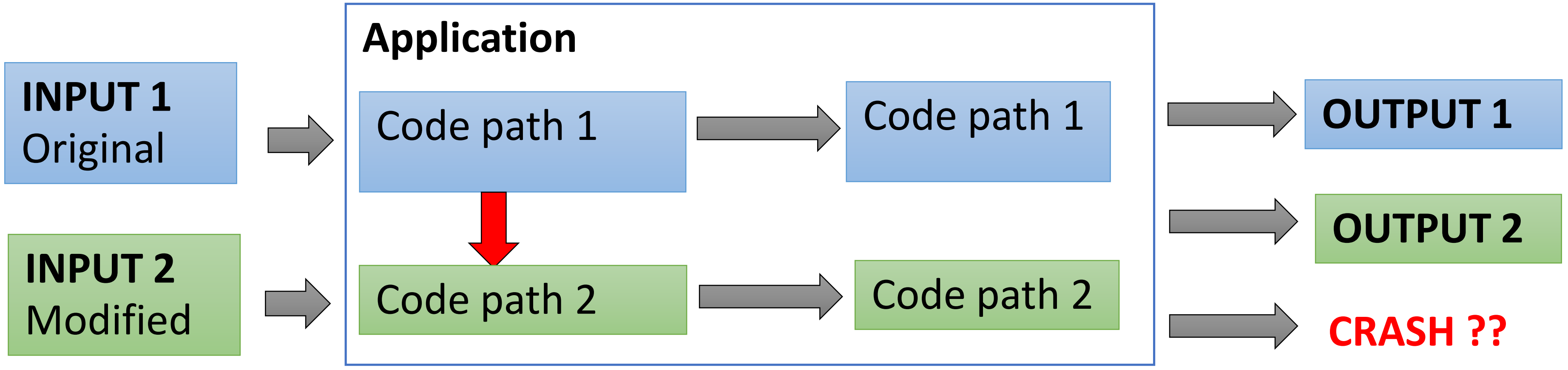
Fuzzing and Bug Hunting

Bug hunting.

- Manual code audit.
  - Takes lot of time. Very slow.
  - Not possible to cover all the code paths.
  - Large code base, not possible for a single person to do audit.
  - Not very productive.
  - Things can be missed.
  - Can not cover all the scenarios.
- Automated
  - Automate bug finding. Very fast.
  - Can cover most of the code paths.
  - No need to worry about size of the code.
  - Can be done by an individual.
  - Can be automated further to notify about crashes, issues.

What is fuzzing?

- Process of automated bug finding.
    1. Feed input to program.
    2. Monitor for crashes.
    3. Save crashing test case.
    4. Generate new input.
    5. Go to 1.



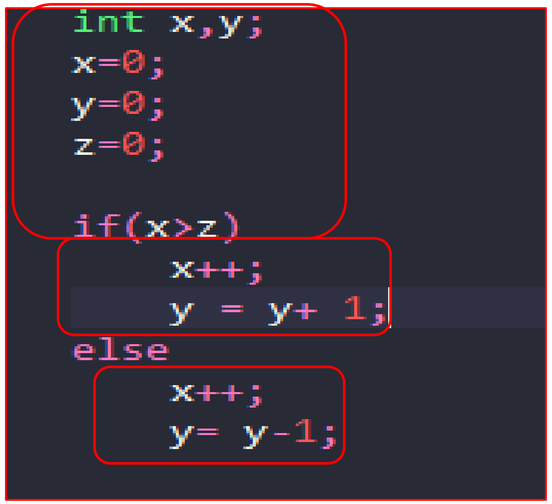
McAfee

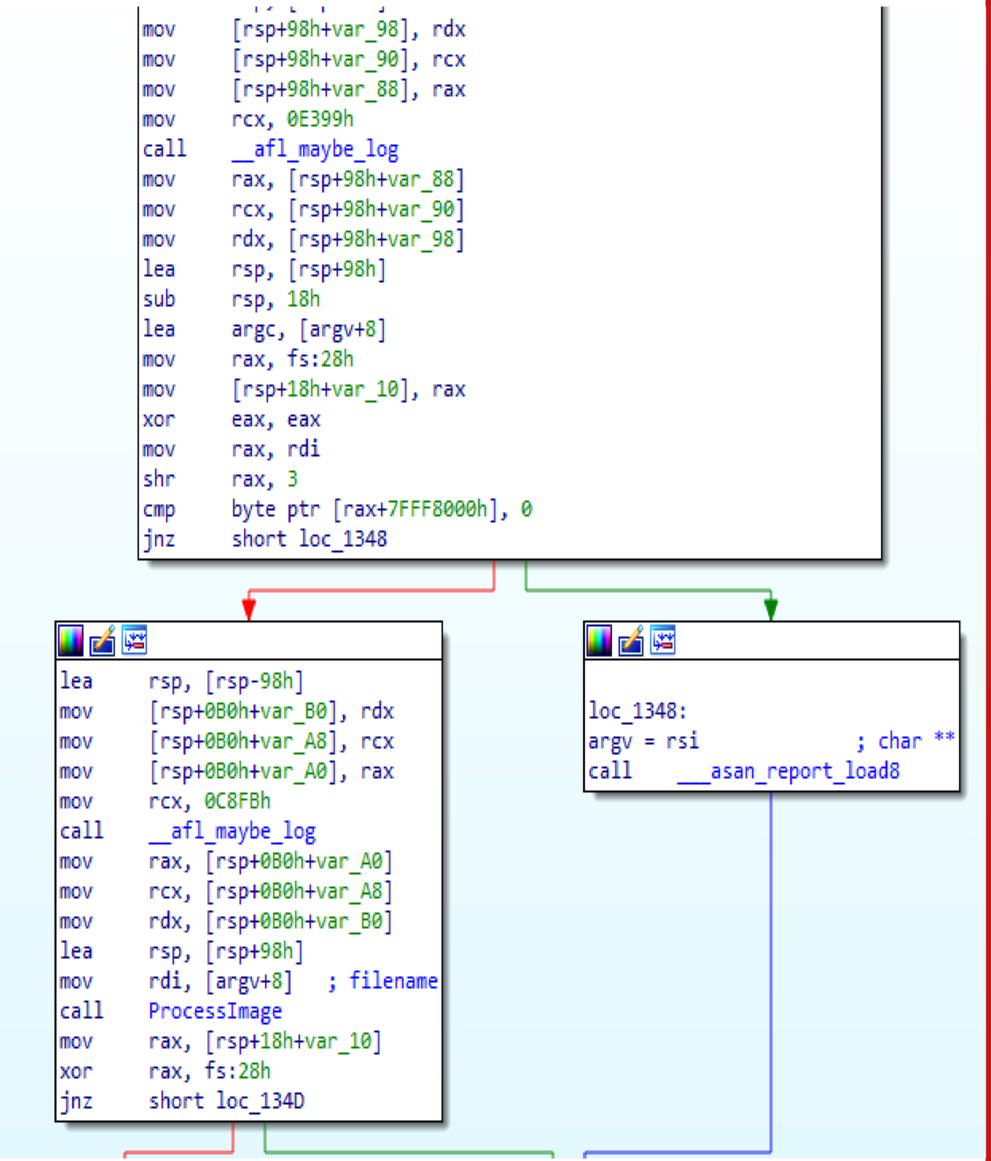
Types Of Fuzzers

- Dumb Fuzzers
  - Random inputs.
  - No idea of fuzzed file.
  - No idea of Program flow.
  - Example: Radmasa
- Generation Fuzzers
  - Generate input based on templates.
  - Need to know file format.
  - No idea of program flow.
  - Example: Peach, Sulley
- Coverage Guided Fuzzers.
  - Monitors program execution by instrumentation.
  - No need to know file format.
  - Mutates file and check for new code path coverage/crash
    - New Code path -> Add to Queue
    - Crash -> Save the input ☺
  - Example: AFL, WinAFL, HonggFuzz, libfuzzer
  - pulling jpeg out of thin air:
  - https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html
    - $ mkdir in_dir
    - $ echo 'hello' >in_dir/hello
    - $ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg

Basic blocks and Coverage

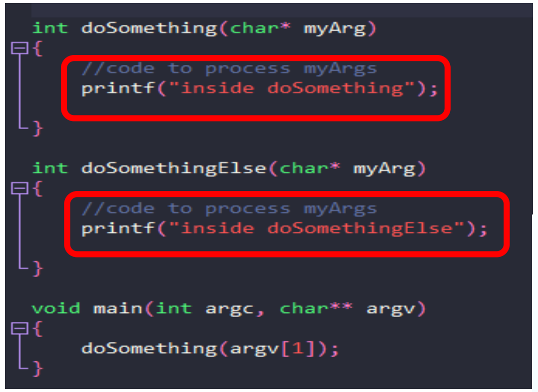
- Basic block
  - consecutive lines of code with no branches.
  - Entry point – control comes to this basic block.
  - Exit point – control goes to another basic block.
- Code Coverage

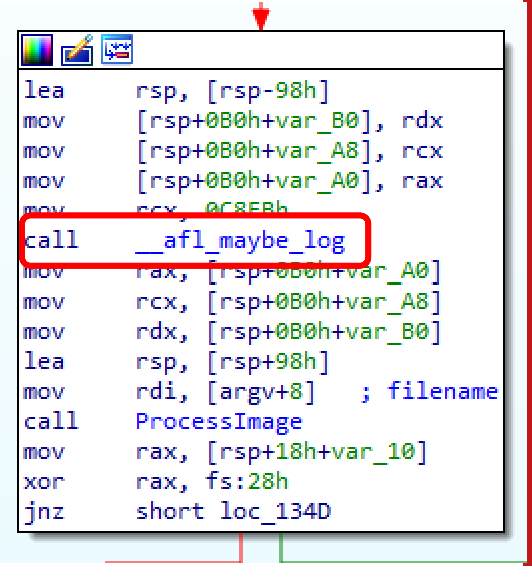


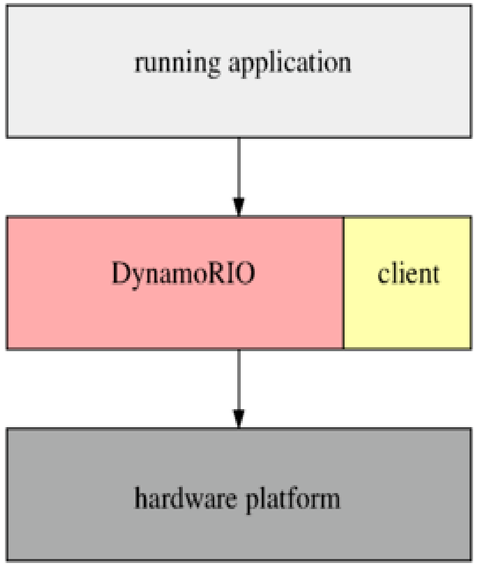


Instrumentation?

- How to trace the program execution at runtime?
  - Basic - add printf in the code and debug.
  - Doesn’t provide much data
  - Need to do manual work.
- If source code is available.
  - Compile time instrumentation
  - Adds instrumentation code at compile time.
  - Can automate things like coverage measurement, Removes manual efforts.
- If source code is not available.
  - Runtime instrumentation
  - Add instrumentation code at runtime.







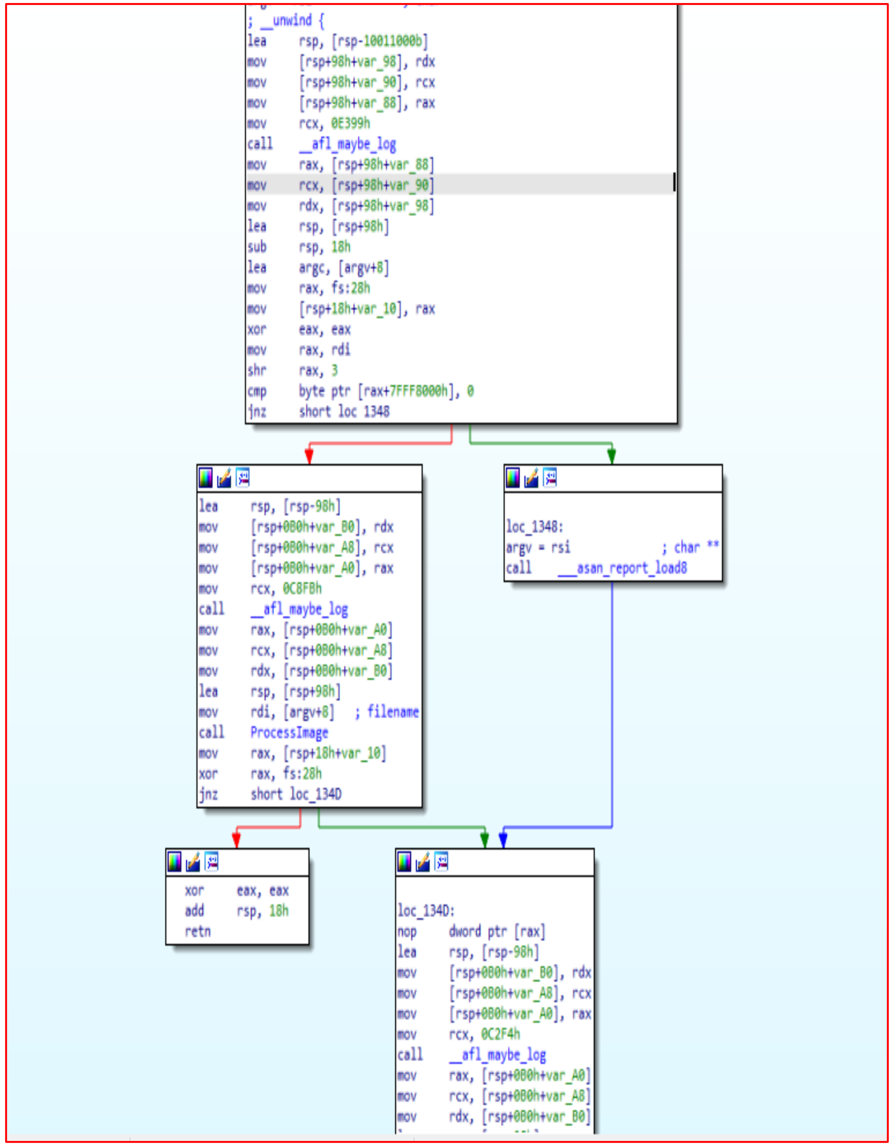
AFL – American Fuzzy Lop

What Is AFL?

- Created by Michael Zelwaski
- Fuzzer with instrumentation-guided genetic algorithm.
- Comes with set of utilities:
  - afl-fuzz, afl-cmin, afl-tmin, afl-showmap etc..
- Fork server/Persistent mode.
- Mutate the files based on various strategies.

How it works?

- Adds Compile time instrumentation.
- Provides compiler wrappers
    - afl-gcc,afl-g++, afl-clang, afl-clang++, afl-clang-fast, afl-clang-fast++
- uses binary rewriting technique.
    - Add instrumentation at each basic block
        - Each basic block will have a unique random id.
    - Done by assembly equivalent of the following pseudo code:
        - cur_location = <COMPILE_TIME_RANDOM >;
        - shared_mem[cur_location ^ prev_location]++;
        - prev_location = cur_location >> 1;
    - A → B →C → D → E vs A → B → D → C → E



Fork Server Vs Persistent Mode

- Fork Server Mode
  - Stop at main().
  - Uses fork to create clone of the program.
  - Process input and create another clone.
  - Saves time in initializing program and thus offer speed improvements.

Ref: https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

- Persistent Mode
  - Fork is still costly.
  - Don’t really need to kill child process after each run.
  - Uses in process Fuzzing.
  - Need to write a harness program.
  - Ex:

```
int main(int argc, char** argv) {

  while (__AFL_LOOP(1000)) {

    /* Reset state. */
    memset(buf, 0, 100);

    /* Read input data. */
    read(0, buf, 100);

    /* Parse it in some vulnerable way.
       You'd normally call a library here. */
    if (buf[0] != 'p') puts("error 1"); else
    if (buf[1] != 'w') puts("error 2"); else
    if (buf[2] != 'n') puts("error 3"); else
      abort();

  }

}
```

- Ref: https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html

Fuzzing Strategies

- **Bitflip** – flips a bit i.e. 1 becomes 0, 0 becomes 1
  - 1/1,2/1,4/1,8/8 ….32/8
- **Byte Flip** – flips a byte
- **Arithmetic** –random arithmetic like plus/minus
- **Havoc** – random things with bit/bytes/addition/subtraction
- **Dictionary** – user provided dictionary or auto discovered tokens.
  - Over/insert/over(autodetected)
- **Interest** - replace content in original file with interesting values
  - 0xff,0x7f etc – 8/8,16/8..
- **Splice** – split and combine two or more files to get a new file.
- Ref: https://github.com/google/AFL/blob/master/docs/technical_details.txt

Sanitizers

Sanitizers

- Tools based on compiler instrumentation.
- Helpful for identifying bugs.
- Can discover bugs like large memory allocations, heap overflow, use after free etc.
- Different types of sanitizers
  - ASAN (**-fsanitize=address)**
  - MSAN (**-fsanitize=memory**)
  - UBSAN (**-fsanitize=undefined**)
  - TSAN (**-fsanitize=thread**)

- Ref:
- https://clang.llvm.org/docs/AddressSanitizer.html
- https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- https://clang.llvm.org/docs/MemorySanitizer.html
- https://clang.llvm.org/docs/ThreadSanitizer.html

Use After Free vulnerabilities

Heap Buffer Overflows

Stack Buffer Overflows

Initialization order bugs

Memory Leaks

Use after scope

Null Pointer Dereferences

Signed Integer Overflows

Typecast Overflows

Divide by Zero errors

Corpus Collection and Crash Triage

Corpus Collection & Minimization

- A good file corpus will help to discover paths in short amount of time.
  - Use regression/test case corpus if available for the software/libs.
  - Use availble corpus files.
    - Ex:
      - https://lcamtuf.coredump.cx/afl/demo/
      - http://samples.ffmpeg.org/
  - Search github/google
- Need to Minimize input corpus
  - Filter out the files which doesn't result in new path.
  - Filter out large files.
- How?
  - afl-cmin –i input –o mininput -- ./program @@

McAfee

Crashes->root cause->Vulnerability

- Root cause analysis
  - We found a crash – now what?
    - Which field in file?
    - What value in the field?
    - Which condition in program?

Vulnerability!!

- 1-2 crashes
  - Manual sorting
- Hundred or Thousands of crashes?
  - How to Triage them?
    - Crashwalk, atriage, afl-collect

McAfee

What we have learned So far?

What is AFL, How it works?
Fuzzing strategies.
Different sanitizers and how to enable them.
Crashes and root cause.

Hands on

Hands on : Compiling and installing AFL

- **git clone https://github.com/google/AFL.git**
- **make**
- **cd llvm_mode**
- **make** → need clang installed
- **cd ..**
- **sudo make install**

Hands on: How to compile program with AFL?

- afl-clang -fsanitize=address imgRead.c -g -o imgReadafl
  - afl-clang -> compiler wrapper for gcc, this will compile and instrument the binary.
  - -fsanitize address -> enables asan[can also use AFL_USE_ASAN=1 and –fsanitize=undefined]
  - -g -> debugging symbols support
  - imgRead.c -> source file.
  - imgReadafl -> generated executable file which will be fuzzed.
  - AFL_DONT_OPTIMIZE=1 -> will disable compiler optimizations.

Hands on: Fuzzing Sample C program with AFL

- Generate Input
  - **echo “IMG” > input/1.img**
- **afl-fuzz –i input –o output –m none -- ./imgRead @@**
  - **afl-fuzz** -> fuzzer binary.
  - **-i** -> directory containing input seed files.
  - **-o** -> directory containing output data from fuzzer
    - Crashes -> contains input files which crashes target program.
    - Hangs -> contains input files which causes hangs for target program.
  - **-m** -> memory limit, if ASAN and 64 bit, set it to none
    - Else compile it in 32 bit using compiler flag –m32 for afl-gcc
    - Set memory limit as –m 800
    - Find more crashes.
  - **-M** -> Master instance, in case you have multiple CPU core.
  - **-S** -> Slave instance, can be n- number depending on the cores you have.

McAfee

Hands on: Root Cause analysis

- Let's use GDB to analyze crashes.
- Commands:
  - **gdb <exe file name>**
  - **r** -> run the program
  - **s** -> step over
  - **next/fi** -> execute till return
  - **b** <filename.c:linenumber> -> puts breakpoint in filename.c at linenumber
- In our case **gdb ./imgread**
  - r <output/crashes/filename>

Hands on: Crash Triage

- Crashwalk is a useful tool to triage crashes if you get lot of crashes.
- Installing crashwalk
  - **sudo apt-get install golang**
  - **go get -u github.com/bnagy/crashwalk/cmd/...**
  - **~/go/bin**
- Installing exploitable
  - **~/src/exploitable/exploitable/exploitable.py**
  - **mkdir ~/src**
  - **cd ~/src**
  - **git clone https://github.com/jfoote/exploitable.git**

Hands on: Crash Triage

- Cwtriage – utility to triage crashes
  - **ASAN_OPTIONS="abort_on_error=1:symbolize=0" Cwtriage –afl –root output**
  - Analyzes each crash file and saves results in crashwalk.db.
  - Run with ASAN else crash will not get replicate.
- Cwdump – utility to dump crash info from crashwalk.db
  - Cwdump crashwalk.db

McAfee

Hands on : Compiling and installing HongFuzz

- **git clone https://github.com/google/honggfuzz.git**
- **make**
- **sudo apt install binutils-dev libunwind-dev**
- **sudo make install**

Hands on: How to compile program with HonggFuzz?

- hfuzz-clang -fsanitize=address imgRead.c -g –O0 -o imgReadhfuzz
  - hfuzz-clang -> compiler wrapper, this will compile and instrument the binary.
  - -fsanitize address -> enables asan
  - -g -> debugging symbols support
  - -O0 -> disable optimization
  - imgRead.c -> source file
  - imgReadhfuzz -> generated executable file which will be fuzzed.

Hands on: Fuzzing Sample C program with HonggFuzz

- Generate Input
  - **echo “IMG” > input/1.img**
- **honggfuzz -i input –workspace output -- ./imgRead ___FILE___**
  - honggfuzz -> fuzzer binary.
  - **-i** -> directory containing input seed files.
  - **--workspace** -> directory containing output data from fuzzer

Fuzzing open source softwares

Hands on: Fuzzing tcpdump

- Get the source code of tcpdump and libpcap.
  - **git clone https://github.com/the-tcpdump-group/tcpdump.git**
  - **cd tcpdump**
  - **git clone https://github.com/the-tcpdump-group/libpcap.git**
  - **cd libpcap**
- Compile it using AFL
  - **CC=afl-gcc CFLAGS="-g -fsanitize=address -fno-omit-frame-pointer" LDFLAGS="-g -fsanitize=address -fno-omit-frame-pointer" ./configure**
  - **sudo make && make install**
- Corpus?
  - Check tests folder ☺
  - Minimise it: **afl-cmin –i tests –o mincorpus –m none -- ./tcpdump –vv –ee –nnr @@**
- Fuzz it
  - **afl-fuzz –i mincorpus –o fuzzoutput –m none -- ./tcpdump –vv –ee –nnr @@**

McAfee

What we have learned So far?

How to compile and install AFL and Honggfuzz

How to fuzz programs using AFL, Honggfuzz

Root cause analysis, crash triage

Reporting to Vendors/Bug Bounty

- Report to vendor first.
- Vendors have a security@vendor.com email address.
- Do not publicly disclose your finding.
- You may get rewarded for your crashes.

Conclusion

- Fuzzing is helpful in overall understanding of software security.
  - Helps to write better code.
  - Can help to find issues which can not be found using normal testing.
  - Helps in securing software.
  - Part of software development life cycle.
- Requires lot of hard work
  - Broken and non working stuff
  - Countless hours analyzing issues and fixing them
  - Multiple vendor follow-ups
  - rejections

But in the end its worth it ☺

McAfee

Thank you!

Comments/Suggestions/Feedback are welcome!

Follow me on Twitter: **@hardik05**