GRAYHAT

Fuzzing and finding vulnerabilities using AFL/WinAFL

Hardik Shah

@hardik05

Overall Agenda

- Introduction
- Part – I
  - Vulnerabilities, Fuzzing process, crash triage, root cause analysis.
- Part - II
  - Fuzzing Using AFL on Linux
- Part – III
  - Fuzzing Using WinAFL on Windows
- 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

Agenda for Part I

- Vulnerabilities
  - Different Types of Vulnerabilities.
    - Integer overflow/Underflow
    - Stack/Heap Overflow
    - OOB Read/Write
    - Use After Free/ Double Free
  - Manually Identifying the vulnerabilities in C Program.
- What is Fuzzing?
  - Need for Fuzzing
  - Types of Fuzzers
  - Fuzzing a program - Process
- Crash analysis .
  - Crash Triage
  - Root cause
- Reporting issues to vendor/Bug Bounty

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

Can be converted to Exploits

Different Types of Vulnerabilities

Integer Overflow

- What it is?
- Vulnerability in integer data types, the way in which 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 = 0x7FFFFFFFF + 1
    - Result will become -0x80000000 (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));
```

```
1954  static int MP4_ReadBox_rdrf( stream_t *p_stream, MP4_Box_t *p_box )
1955  {
1956    uint32_t i_len;
1961    MP4GET4BYTES(i_len);
1962    if( i_len > 0 )
1963    {
1964      uint32_t i;
1965      p_box->data.p_rdrf->psz_ref = malloc( i_len  + 1);
1966      for( i = 0; i < i_len; i++ )
1967      {
1968        MP4_GET1BYTE( p_box->data.p_rdrf->psz_ref[i] );
```

- **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

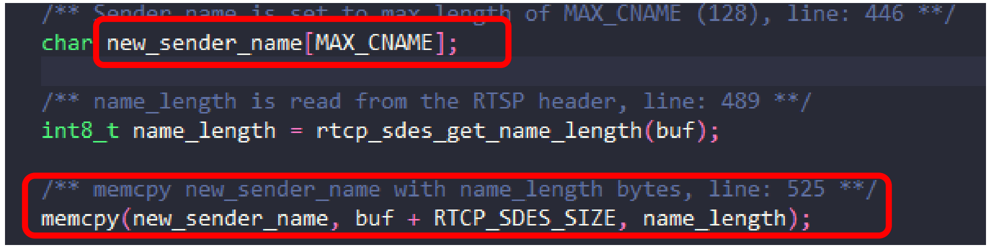
- Stack Overflow
  - Local variable are stored in Stack
  - Finite size
  - Overflow in local variable, can corrupt other data on stack.
  - Example:
    - Function foo(){

      char var1[8];

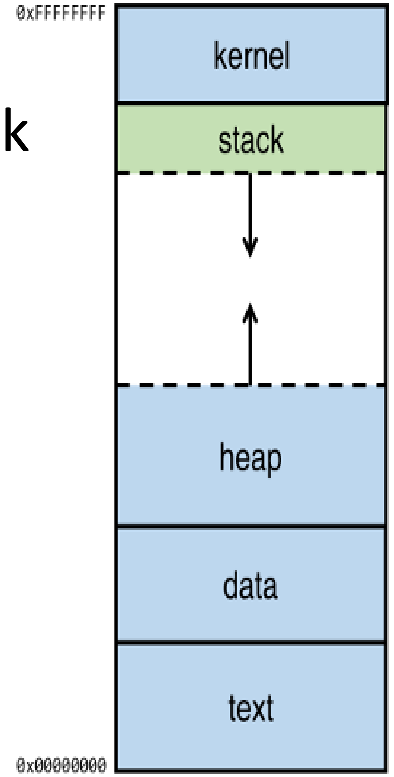
      char var2[100];

      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];

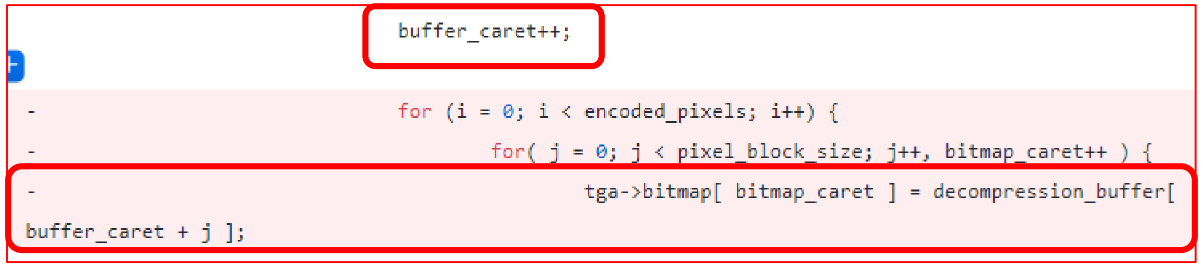
    memcpy(var1,var2,sizeof(var2));
    → heap overflow

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

- 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



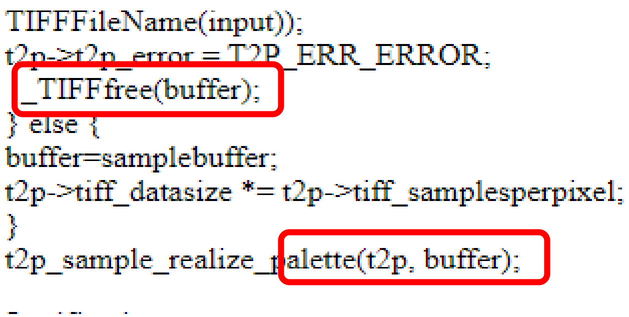
```
                                      buffer_caret++;

-                                     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

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:

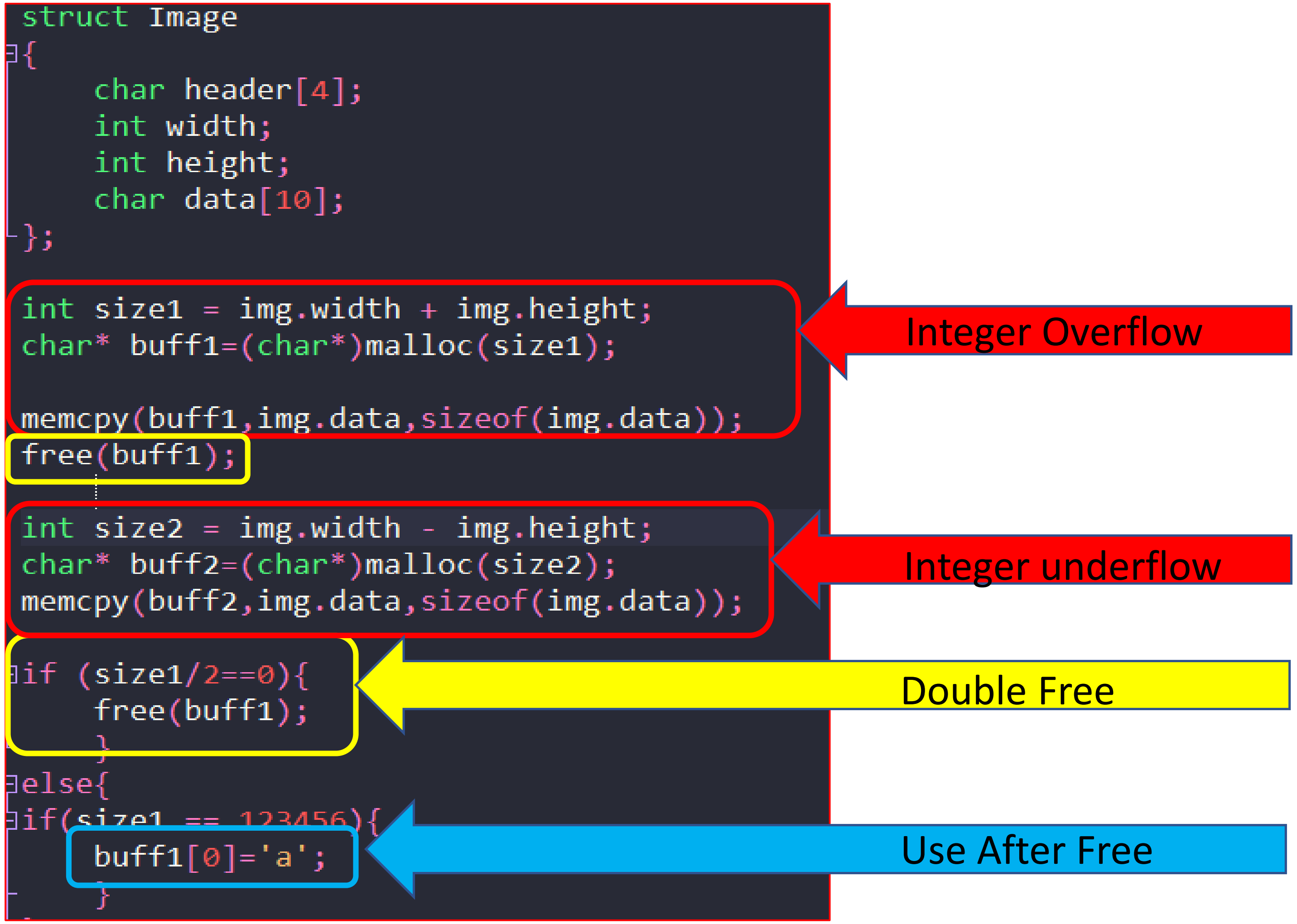
```
char *buff = (char*)malloc(10);
free(buff);
free(buff); → double free!
```

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

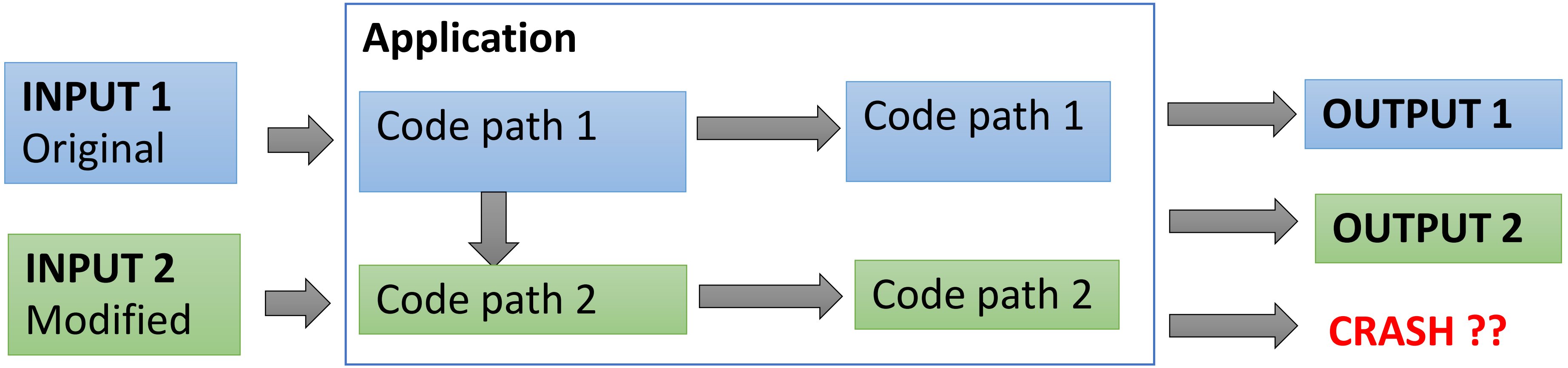
Bug Hunting and Fuzzing

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 in program.
  1. Feed input to program.
  2. Monitor for crashes.
  3. Save crashing test case.
  4. Generate new test case.
  5. Go to 1.



Types of Fuzzers

Dumb Fuzzers

- Random input
- No understanding of file format/network protocol is required.
- Can take lot of time (depending up on your luck).
- Example: radmasa

Generation Fuzzer

- Create input based on predefined structure.
- Requires understanding of file format.
- Requires understanding of network protocol.
- Example: peach,sulley

Coverage Guided Fuzzer

- Monitors program flow by using instrumentation
- No knowledge of file format is required.
- 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

What we have learned So far?

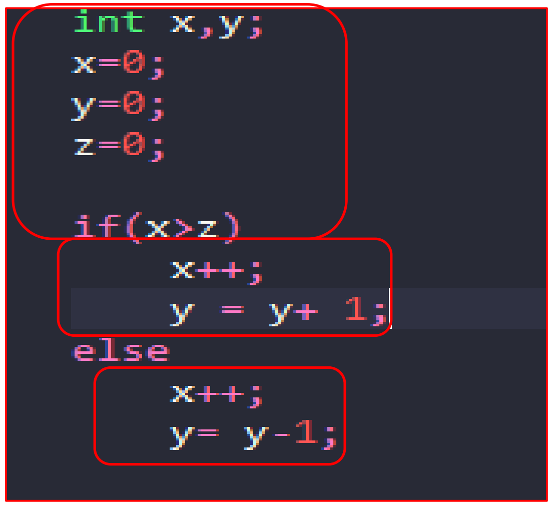
Fuzzing and bug hunting

Different types of fuzzers – dumb, generation, coverage guided

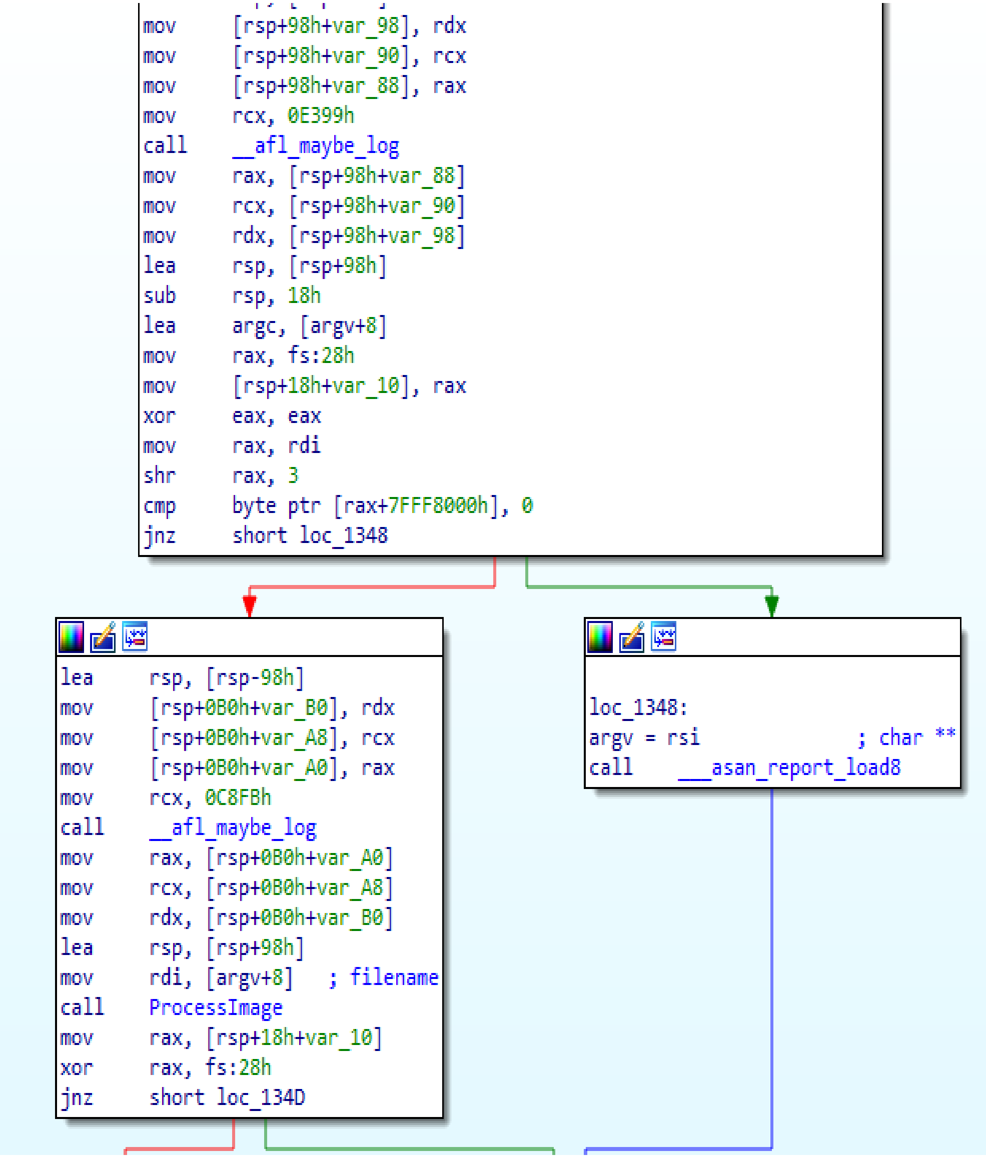
Coverage & Instrumentation

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



```
int x,y;
x=0;
y=0;
z=0;

if(x>z)
    x++;
    y = y+ 1;
else
    x++;
    y= y-1;
```



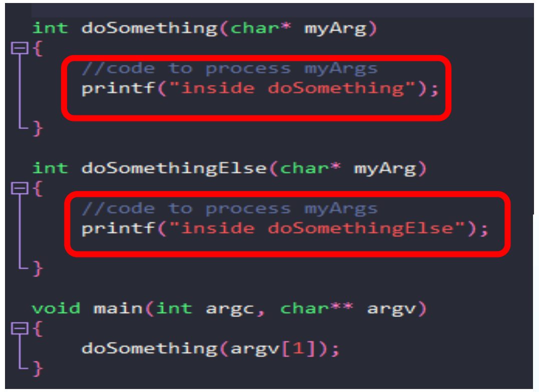
```
mov     [rsp+98h+var_98], rdx
mov     [rsp+98h+var_90], rcx
mov     [rsp+98h+var_88], rax
mov     rcx, 0E399h
call    __afl_maybe_log
mov     rax, [rsp+98h+var_88]
mov     rcx, [rsp+98h+var_90]
mov     rdx, [rsp+98h+var_98]
lea     rsp, [rsp+98h]
sub     rsp, 18h
lea     argc, [argv+8]
mov     rax, fs:28h
mov     [rsp+18h+var_10], rax
xor     eax, eax
mov     rax, rdi
shr     rax, 3
cmp     byte ptr [rax+7FFF8000h], 0
jnz     short loc_1348
```

```
lea     rsp, [rsp-98h]
mov     [rsp+0B0h+var_B0], rdx
mov     [rsp+0B0h+var_A8], rcx
mov     [rsp+0B0h+var_A0], rax
mov     rcx, 0C8FBh
call    __afl_maybe_log
mov     rax, [rsp+0B0h+var_A0]
mov     rcx, [rsp+0B0h+var_A8]
mov     rdx, [rsp+0B0h+var_B0]
lea     rsp, [rsp+98h]
mov     rdi, [argv+8]    ; filename
call    ProcessImage
mov     rax, [rsp+18h+var_10]
xor     rax, fs:28h
jnz     short loc_134D
```

```
loc_1348:
argv = rsi                  ; char **
call    ___asan_report_load8
```

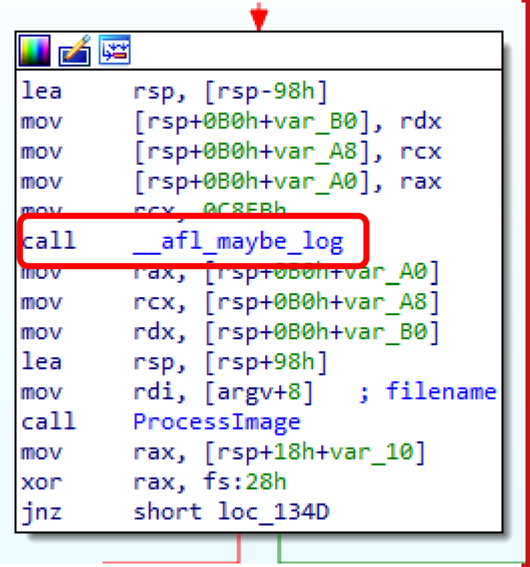
Instrumentation?

- 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.



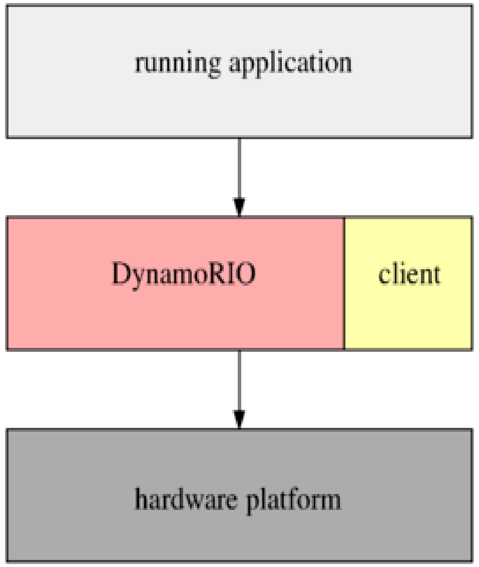
```
int doSomething(char* myArg)
{
    //code to process myArgs
    printf("inside doSomething");
}

int doSomethingElse(char* myArg)
{
    //code to process myArgs
    printf("inside doSomethingElse");
}

void main(int argc, char** argv)
{
    doSomething(argv[1]);
}
```

- If source code is available.
  - Compile time instrumentation
  - Adds instrumentation code at compile time.
  - Can automate things like coverage measurement, Removes manual efforts.

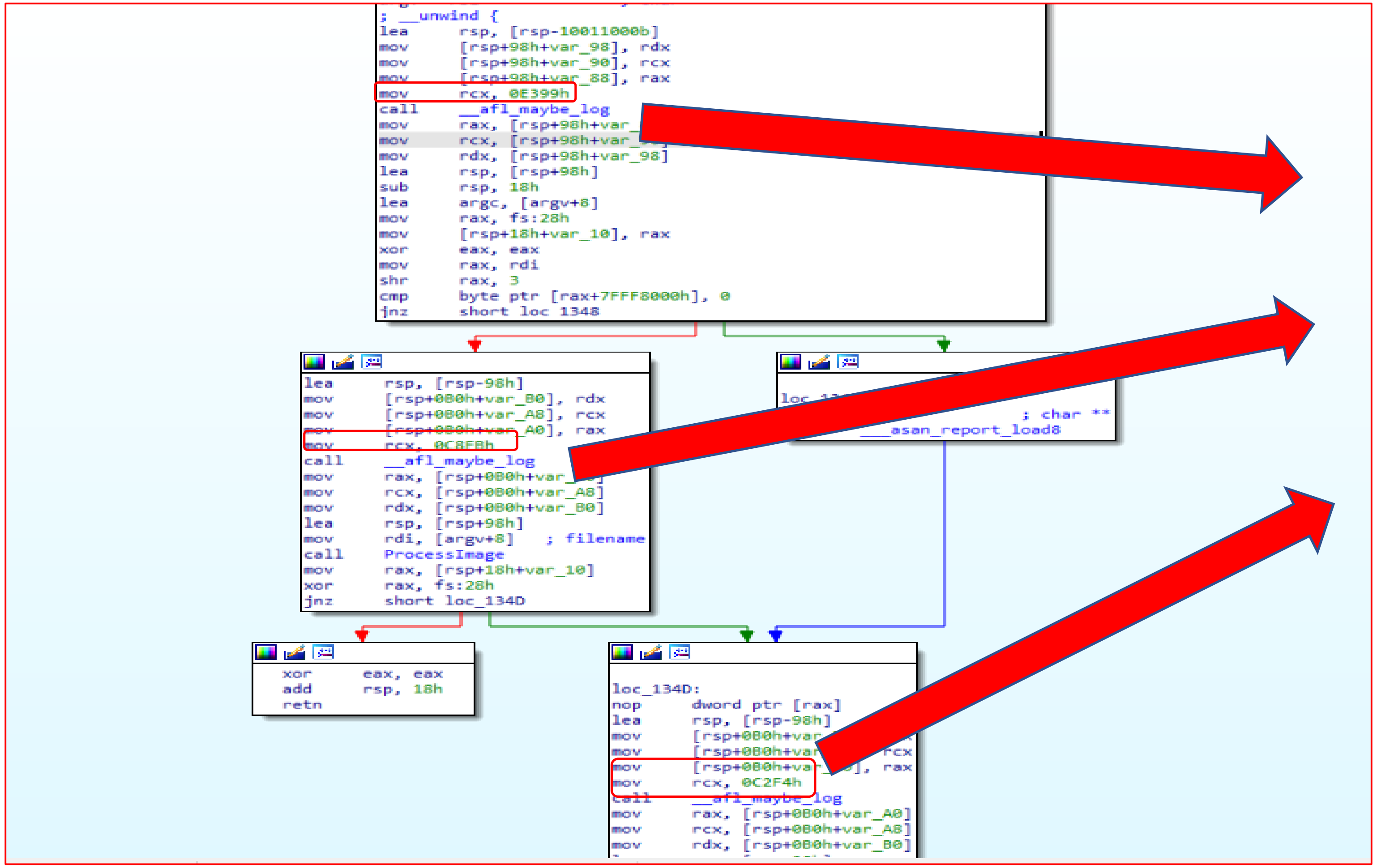


```
lea     rsp, [rsp-98h]
mov     [rsp+0B0h+var_B0], rdx
mov     [rsp+0B0h+var_A8], rcx
mov     [rsp+0B0h+var_A0], rax
mov     rcx, 0C8EBh
call    __afl_maybe_log
mov     rax, [rsp+0B0h+var_A0]
mov     rcx, [rsp+0B0h+var_A8]
mov     rdx, [rsp+0B0h+var_B0]
lea     rsp, [rsp+98h]
mov     rdi, [argv+8]    ; filename
call    ProcessImage
mov     rax, [rsp+18h+var_10]
xor     rax, fs:28h
jnz     short loc_134D
```

- If source code is not available.
  - Runtime instrumentation
  - Add instrumentation code at runtime.



AFL Binary Instrumentation



```
; __unwind {
lea     rsp, [rsp-10011000b]
mov     [rsp+98h+var_98], rdx
mov     [rsp+98h+var_90], rcx
mov     [rsp+98h+var_88], rax
mov     rcx, 0E399h
call    __afl_maybe_log
mov     rax, [rsp+98h+var_
mov     rcx, [rsp+98h+var_
mov     rdx, [rsp+98h+var_98]
lea     rsp, [rsp+98h]
sub     rsp, 18h
lea     argc, [argv+8]
mov     rax, fs:28h
mov     [rsp+18h+var_10], rax
xor     eax, eax
mov     rax, rdi
shr     rax, 3
cmp     byte ptr [rax+7FFF8000h], 0
jnz     short loc 1348
```

```
lea     rsp, [rsp-98h]
mov     [rsp+0B0h+var_B0], rdx
mov     [rsp+0B0h+var_A8], rcx
mov     [rsp+0B0h+var_A0], rax
mov     rcx, 0C8FBh
call    __afl_maybe_log
mov     rax, [rsp+0B0h+var_
mov     rcx, [rsp+0B0h+var_A8]
mov     rdx, [rsp+0B0h+var_B0]
lea     rsp, [rsp+98h]
mov     rdi, [argv+8]    ; filename
call    ProcessImage
mov     rax, [rsp+18h+var_10]
xor     rax, fs:28h
jnz     short loc_134D
```

```
loc_
                          ; char **
        ___asan_report_load8
```

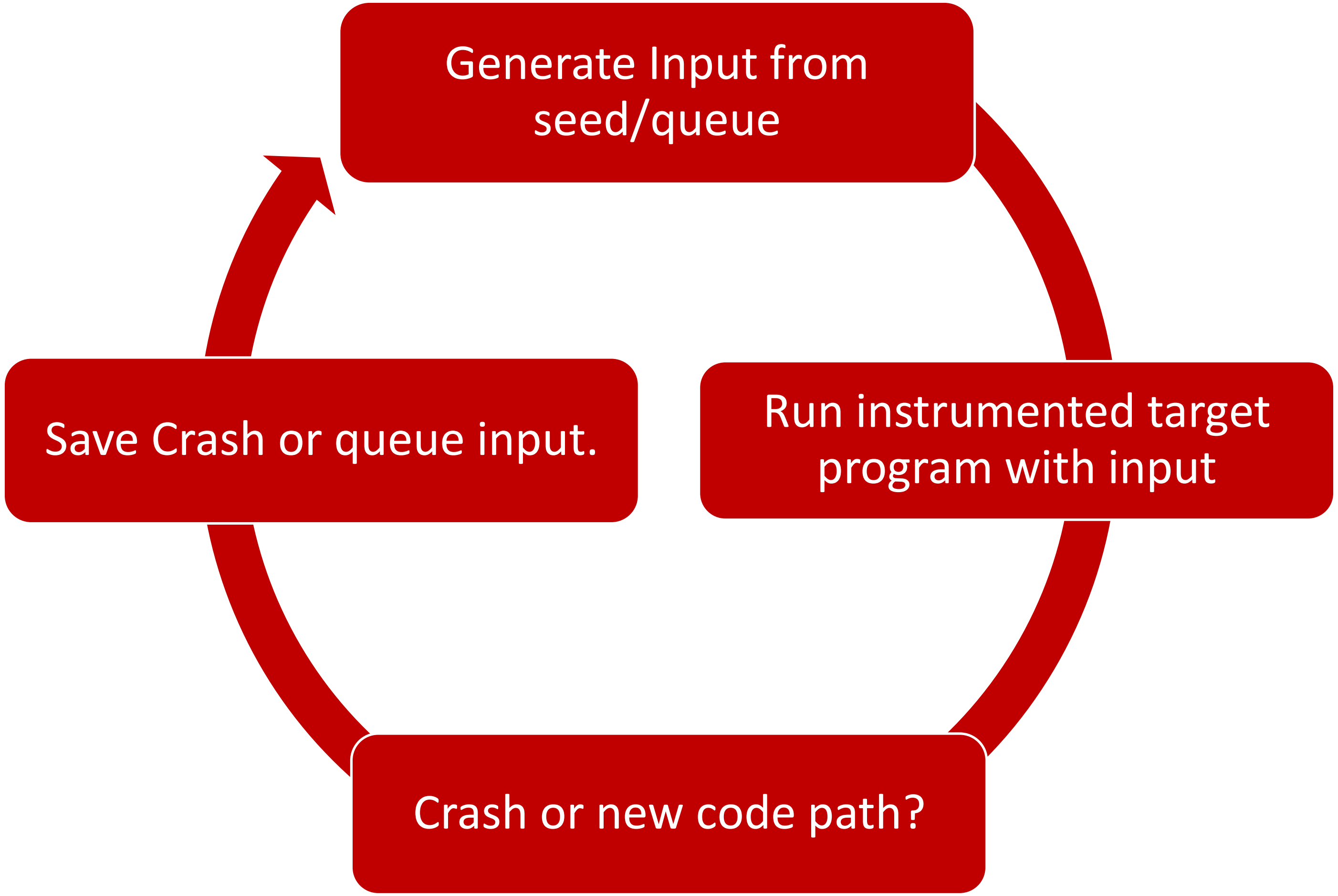
```
    xor     eax, eax
    add     rsp, 18h
    retn
```

```
loc_134D:
nop     dword ptr [rax]
lea     rsp, [rsp-98h]
mov     [rsp+0B0h+var
mov     [rsp+0B0h+var           rcx
mov     [rsp+0B0h+va      ], rax
mov     rcx, 0C2F4h
call    __afl_maybe_log
mov     rax, [rsp+0B0h+var_A0]
mov     rcx, [rsp+0B0h+var_A8]
mov     rdx, [rsp+0B0h+var_B0]
```

1. random, unique id for each block.
2. __afl_maybe_log
3. Afl will maintain coverage bitmap based on this.

Fuzzing Process

Coverage Guided Fuzzing



Generate Input from seed/queue

Run instrumented target program with input

Crash or new code path?

Save Crash or queue input.

Corpus Collection

- 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
- Search google

Corpus Minimization

- Having a large corpus is good or bad?
  - What is file size is too large?
    - Bitflip/byteflip will take lot of time.
    - 10MB = **10485760** Bytes
  - What If many files trigger same code path?
    - Fuzzer will spent unnecessary cycles on going through them.
- 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 @@

Crashes->rootcause->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

What we have learned So far?

Instrumentation, fuzzing process, corpus collection, root cause

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.

GRAYHAT