When virtualization encounter AFL

- A Portable virtual device fuzzing framework with AFL

Jack Tang, Moony Li

Twitter: @jacktang310, @Flyic

TrendMicro, Security Researcher

Jack_tang@trendmicro.com, moony_li@trendmicro.com

1. Abstract

Along with virtualization technology adopted by both enterprise and customer popularly, virtual machines escape attacking become more and more critical which could NOT be ignored. Because of virtual devices' nature character (virtual device emulation is in host level, guest can access virtual devices with arbitrary data), they are a big attack surface to achieve virtual machine escaping. In fact, among those reported virtual machines escape attacking, the virtual device attacking takes a big ratio. For example, the VENOM attacking (Reference 5.1).

Several fuzzing methods towards virtual devices have been released including dump I/O traces and replay in guest OS, conformance fuzzing to constraint virtual device in proper internal state and so forth. However, rare of them considered calculating and controlling code coverage and control in intension. And also, rare of them consider keep their fuzzing framework portable for difference virtualization software.

So what happens when virtualization fuzzing encounter with AFL? We would like give you one possible answer. Our portable virtual device fuzzing framework with AFL could solve both of the two challenges—code coverage feedback and portability.

2. Outline

2.1 Problem Statement:

Security researchers have released several good methods to hunt vulnerabilities in virtual devices (Reference 5.2, 5.4). We summarize these methods into several basic types in general.

I. Passive or active fuzzing I/O requests from guest OS

In guest OS (for example Linux or Windows), pouring fuzzing I/O requests traffic into virtual device and monitor the virtual device's status from host side. The fabricated I/O requests are usually encapsulated with system call related to virtual devices or direct I/O access towards virtual devices. If the virtualization software panic, analyzing the panic and find vulnerabilities.

Usually, real I/O towards specific devices would be dumped and replay the I/O requests in the guest OS.

a. Advantage:

A. Simple to implement.

B. No need to know the hardware protocol in detail.

b. Disadvantage:

A. No code coverage feedback and control.

Hard to know testing status (for example: code coverage) and base on the status to adjust and optimize the input data.

B. Incomplete fuzzing by design

For there exist multiple life-time stages (e.g. recognition, initialization, configuration, R/W, de-initialization) and functions for virtual devices, fuzzing in runtime guest OS could usually only cover some part of the whole stages (e.g. only normal R/W) and function by design.

II. Conformance fuzzing test for virtual devices

In fact, there exists so many failure testing cases because of in-conformant internal state of registers and memories for virtual devices in I/O dump and trace replay solution mentioned before. One possible solution is to introduce Symbolic Execution to retrieve the constraint input data to keep virtual devices in right state.

a. Advantage:

A. It could really exclude much of meaningless or duplicated input data for fuzzing.

This method could make the fuzzing scope deeper.

b. Disadvantage:

A. Time consuming to find input constraints which are impractical

B. Also lack of code coverage statistic and control

III. Code reviewing on the virtual device code.

a. Advantage: flexible.
b. Disadvantage:
   Efficiency is low.
   Hard to be scalable automatically.

So we introduce our solution to make virtual device testing more controllable and efficient: A Portable virtual device fuzzing framework with AFL (American Fuzzy Lop, see reference 5.3).

Our approach has 2 key word: “portable”, “AFL”.

“Portable” is got from light-weight BIOS which we describe in the next part.

“AFL” is in order to get fuzzing status feedback. As you know, AFL is a popular and effective fuzzing framework which has branch coverage feedback and is based on the testing coverage feedback to mutate and tune input test data automatically to cover more code branch. But how to adjust AFL to fuzzing various virtual devices and meanwhile can support various virtualization software (for example: Qemu+KVM, virtual box …) is a challenging work.

2.2 Our Approach

As known, one of the most essential attack surface of devices is a series of I/O communication. Our approach is from the idea that we take communicating with a device as a sequence of accessing specified memory or IO space address with specified data. The specified data is our fuzzing test’s input data.

In brief, our solution is composed of three parts: a customized BIOS system (named CBS) and device control clients (named DCC) and AFL integration.

CBS (Customized BIOS System). CBS runs in the virtualization software. The CBS mainly does following jobs:

1. Discovering attached target devices
2. Initialize serial port device as communication channel
3. Run a MIOPS (Memory and IO space Operation Proxy Server). The server receives operation instructions from serial port and access specified address according to the received instructions.

DCC (Device Control Client). DCC runs in the host side. The DCC does following jobs:

1. Reading input data.
2. Fork virtualization software process which runs CBS in it.
3. Initialize target devices by sending memory or IO access instruction sequence to MIOPS via CBS' serial port channel.
4. Sending memory or IO access instruction with input data to MIOPS via serial port channel.

Integrating AFL is to get fuzzing coverage feedback and mutate the input data automatically. DCC would read in mutated data and deliver it to CBS. CBS would execute the data as I/O requests, and also CBS execution status (e.g. crash) would be monitored and notify to AFL.

The approach gains many advantages which are listed as follow:

1. Portable

Our solution has few requirements to target virtualization software. The only requirement is the target virtualization software can start customized BIOS in a guest machine with a serial port device. So our approach can support a number of virtualization software.

2. Good performance

The fuzzing of our solution does not depend on guest operation system which is usually time-consuming to bootstrap and run.

Because the customized BIOS system is not a full operating system, it only focus direct I/O with devices, the speed to launch the BIOS system is very fast.

3. Direct

The solution directly communicates with virtual device without communicating with device drivers. In most case, device driver's code is much complex and vulnerable than virtual device

code. By communicating directly with virtual device, we can avoid interference from the device driver.

4. Code coverage feedback & control

This is why we introduce AFL. AFL's function let our solution get the capabilities.

2.3 Solution implementation

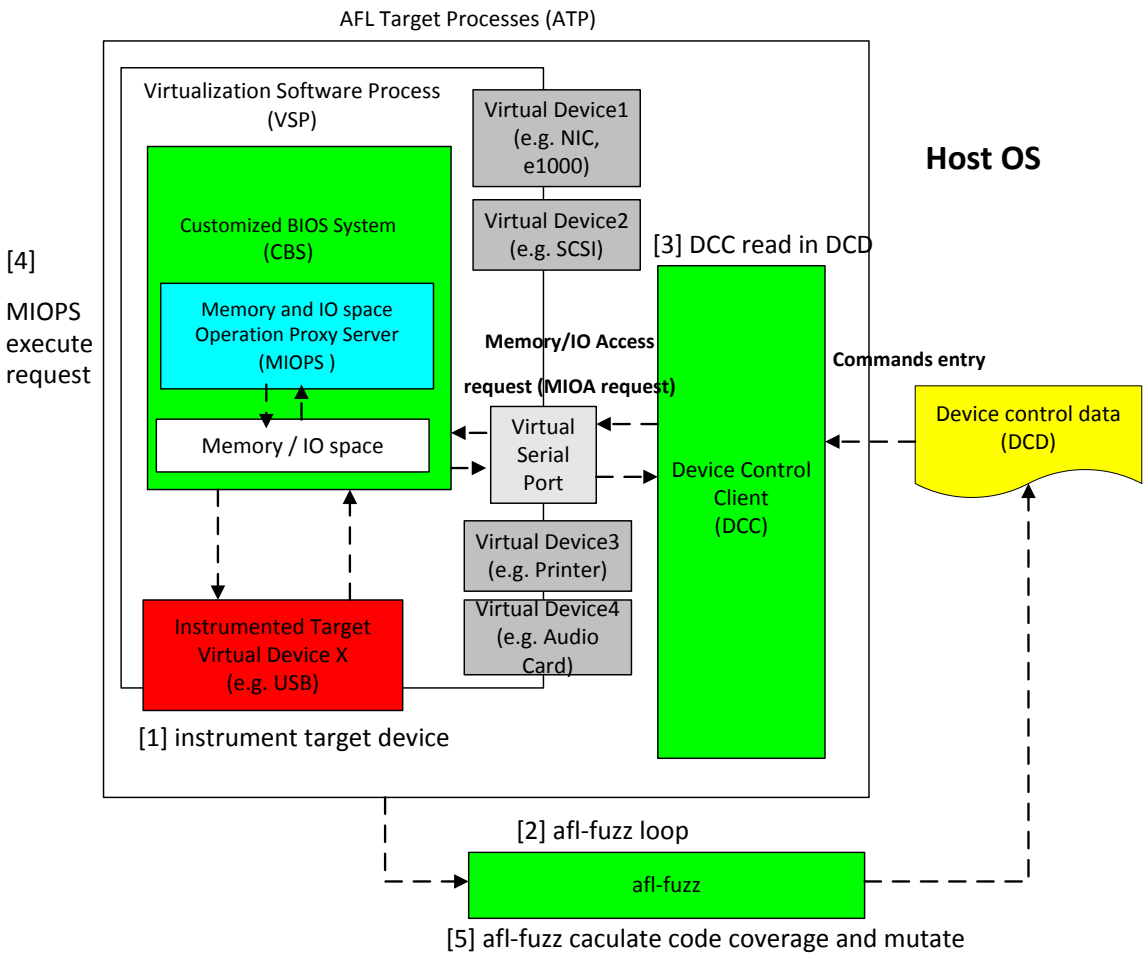

Figure 1 Overall Architecture and workflow

The figure above shows the architecture of our solution architecture. We will explain every part in the architecture in detail.

In brief, target device in Virtual Software Process (VSP) would be instrumented for execution trace as step 1. Then afl-fuzz would launch AFL Target Process(ATP) which contains Device Control Client(DCC) and VSP in a loop as step 2. DCC would read in Device Control Data (DCD) , translate it to MIOA (Memory / IO accessing) requests and transfer the request to Memory and IO space Operation Proxy Server(MIOPS) in Customized BIOS System (CBS)as step 3. MIOPS in CBS would execute the request so as to hit the execution trace in target virtual devices as step 4. By design, afl-fuzz would calculate code coverage and mutate the DCD to generate new DCD in the whole loop as step 5.

2.3.1 Customized BIOS System(CBS)

CBS (Customized BIOS System) is a simple BIOS running in the virtualization software which would initiate virtual devices and prepare basic I/O execution environment for fuzzing. Actually, vulnerabilities of virtual devices are more suitable to be hunt in BIOS than heavy-weight OS(e.g. Linux) because I/O operations are almost the same for the two environments but BIOS is usually more “light” than OS.

The CBS mainly does following jobs:

1. Discovering attached target devices
2. Initialize serial port device
3. Run a MIOPS (Memory and IO space Operation Proxy Server).

2.3.1.1 Seabios customization

The CBS is a simple BIOS system which is customized with Seabios project (reference5.7). So why we select it as code base?

1. Seabios is very light weight.

2. Seabios provides basic device/bus support (for example: basic PCI bus). This saves effort for testing PCI device.

3. Seabios is open source which code is easy to understand.

In view of traditional BIOS, Seabios ‘s code execution flow can be divided into 3 phases (refer 5.8):

1. POST (Power On Self Test) phase:

The goal of the phase is to initialize internal state, initialize external interfaces, detect and setup hardware, and to then start the boot phase.

2. Boot phase:

The goal of the boot phase is to load the first sectors which contain boot loader into memory and start execution of that boot loader. After boot loader is executed, operation system will be launched. As we do not need boot to operation system, the boot phase code is removed in CBS.

3. BIOS runtime service phase:

The goal of this phase is to support basic and legacy I/O system service for runtime operation system. The BIOS runtime service usually provides legacy calling interfaces which are compatible with BIOS standards specification. Also, this part is not needed in CBS.

Actually, CBS would only keep the POST phase and customized it as follow:

1. Detect physical memory
2. Setup platform hardware (for example: PCI bus, clock…)
3. Setup necessary device hardware (for example: serial port device which is used as communication channel in our solution)
4. Recognize the devices which would be fuzzed.
5. Start to run MIOPS (Memory and IO space Operation Proxy Server) handle loop. The specific serial port is a communication channel which we can send request from DCC to MIOPS.

2.3.1.2 Memory and IO space Operation Proxy Server (MIOPS)

MIOPS is a memory/IO access proxy server is responsible for handling MIOA (Memory/IO Access) request which is sent from DCC. MIOPS' goal is to parse MIOA request and execute it in CBS.

I. What is MIOA request (Memory/IO Access request)

MIOA request is a very simple protocol which describes low level IO access request which is usually described in pseudo disassembly code. For example:
"inb <address>"
"inw <address>"

“inl <address>”
“outb <address> <value>”
“outw <address> <value>”
“outl <address> <value>”
“write <address> <value> <length>”
“read <address> <length>”

II. Execute MIOA request

When MIOPS receives MIOA requests from DCC, it parses the request and does real IN/OUT instruction or memory space access operation. The handling process is very like QEMU ‘s QTest framework (refer 5.9). The code snippet is as following:

```
struct request_MIOA* process_MIOA_request(char* request_string)
{
    struct request_MIOA* pReq = NULL;
    pReq = parse_MIOA_request(request_string);
    if(!pReq)
        return NULL;
    switch(pReq->command_id)
    {
        case e_inb:
            do_inb(pReq);
            break;
        case e_inw:
            do_inw(pReq);
            break;
        case e_inb:
            do_inl(pReq);
            break;
        case e_outb:
            do_outb(pReq);
            break;
        case e_outw:
            do_outw(pReq);
            break;
        case e_outl:
            do_outl(pReq);
            break;
        case e_read:
            do_read(pReq);
            break;
        case e_write:
            do_write(pReq);
            break;
        default:
            return NULL;
    } ? end switch pReq->command_id ?
    return pReq;
} ? end process_MIOA_request ?
```

Figure 2 Process function code snippet of MIOA request

As we wish, the execution of fuzzed rquests in MIOPS would trigger potential vulnerability of virtual devices. Finally, the virtual device bug could cause VSP to crash.

III. Poll device status instead of using interruption

In order to simplifying our CBS, we do not support interrupt mechanism which is usually common design in guest OS for target virtual device status notification. To get the result of virtual device request's result, we use another solution to replace interruption: Polling.

In order to get target virtual device request's result, we follow RFC(Requests For Comments) documents to poll specific registers or memory space address before timeout.

Following is an example for waiting for USB XHCI device's request to set specific bits:

```
static int wait_bit(XHCIQState *xhci ,u32 *reg, u32 mask, int value, u32 timeout)
{
    u32 waste_time_ms = 0;
    while ((qpci_io_readl(xhci->dev, reg) & mask) != value) {
        if ( waste_time_ms > timeout ) {
            return -1;
        }
        usleep(2*1000);
        waste_time_ms+=2*1000;
    }
    return 0;
}
```

Figure 3 Code snippet of poll result of USB XHCI device's request to set bits

2.3.1.3 Virtual Serial Port

Actually, serial port device is a very common device which is supported by virtualization software. In our solution, it is designed to bridge the connection of DCC and CBS. Why we choosing virtual serial port as the connection channel is based on consideration as follow:

1. No need to modify virtual machine code
   To make solution portable, the less modification to your target virtual machine, the better the solution is. For CBS or DCC, what they just should do is to open the device and read/write data stream from/to it.
2. General device for virtual machine
   Among all virtual devices supported by virtual machine, Virtual serial port may be one of the most basic and common device.

2.3.2 Device Control Client (DCC)

DCC is running in host side which goal is to generate device control request base on tester's requirement and communicate requests with MIOPS in CBS.

The workflow of DCC can be divided into 4 parts:

1. Launch VSP to load CBS.
2. Ping MIOPS in CBS.
3. Initialize target virtual device.
4. Parsing DCD and translating to MIOA request.

2.3.2.1 Launch VSP to load CBS

Actually, VSP is the run-time process instance of your target virtual machine. Usually, CBS could be loaded in VSP's guest OS. DCC forks VSP directly.

For example, on KVM+QEMU, we use following command:

mkfifo jack_pipe

mkfifo jack_pipe1

qemu-system-x86_64 -bios out/bios.bin -serial pipe:jack_pipe -serial pipe:jack_pipe1

This means the bios.bin (which contains CBS) would be launched with 2 serial ports which link respective pipes on host.

2.3.2.2 Ping MIOPS in CBS.

DCC will open the pipe and write "hello". If MIOPS returns "ok", this means MIOPS is ready for receiving MIOA requests.

2.3.2.3 Initialize target virtual device

Communicating MIOPS with MIOA requests, we can initialize target virtual device. If we want to test the initialization code, this step could be ignored. For many devices, initialization is necessary in order to handle requests to it properly. For example, USB XHCI devices initialization would be as following steps:

1. Find XHCI controller device from PCI bus base on device ID and function ID.

2. Read PCI capability configuration of XHCI controller device, for example: operational registers address, doorbell address …

3. Map specific memory space to XHCI controller device memory for specific function. For example: command queue, event queue.

4. Find USB device attached to the controller.

5. Initialize device basing on USB device type

In essence, these initialization steps are a series of IOMA requests. Taking USB XHCI device initialization for example, the MIOA requests snippet is as follow:

```
outl 0xcf8 0x8000e800
ECHO
inw 0xcfc
ECHO 0x1033
outl 0xcf8 0x8000e800
ECHO
inl 0xcfc
ECHO 0x1941033
outl 0xcf8 0x8000e804
ECHO
inw 0xcfc
ECHO 0x0000
outl 0xcf8 0x8000e804
ECHO
outw 0xcfc 0x7
ECHO
outl 0xcf8 0x8000e804
ECHO
inw 0xcfc
ECHO 0x0007
outl 0xcf8 0x8000e810
ECHO
outl 0xcfc 0xffffffff
ECHO
outl 0xcf8 0x8000e810
ECHO
inl 0xcfc
ECHO 0xffffc004
outl 0xcf8 0x8000e810
ECHO
outl 0xcfc 0xe0000000
ECHO
read 0xe0000000 1
ECHO 0x0000000000000040
read 0xe0000004 4
ECHO 0x0000000008001040
```

Figure 4 MIOA requests snippet for USB XHCI device initialization

2.3.2.4 Parse DCD and translate to MIOA request

DCD is Device Control Data which is essentially sequence of MIOA requests. Usually we format DCD suitable for AFL fuzz and put it in a test case file.

Taking USB XHCI device for example, the DCD looks like this:

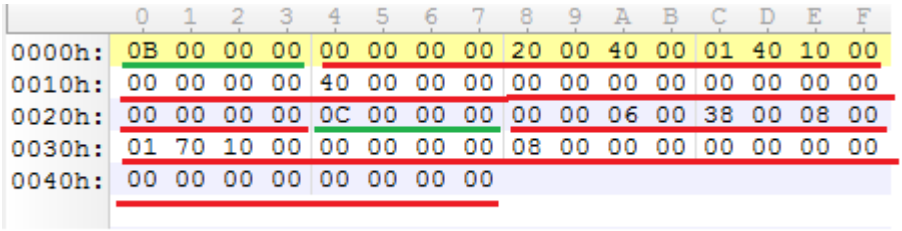

Figure 5 DCD for testing USB XHCI device

```
typedef struct _command_entry
{
  u32 _slotid;
  u32 _command_id;
  void* _inctx;
  u8 _input_buf[32];

}command_entry_t;
```

This DCD contains 2 USB XHCI commands as defined above. The first_command_id (the first 4 bytes is green underlined) is “Address Device”. Next part which is red underlined is the parameter (actually it is _input_buf) for “Address Device” command. The second command id (the second 4 bytes is green underlined) is “Configure Endpoint”. Next part which is red underlined is the “Configure Endpoint” command’s parameter. (USB XHCI spec refer 5.10)

Similarly, we implement DCD format for floppy device as following:

```
struct fdc_command

{

unsigned char cid;

unsigned int args_count;

unsigned int args[0];
```

};

The DCD for testing floppy device contains several fdc_command structures. The cid field in fdc_command structure represents floppy device control command id . For example,

0x6: FD_CMD_READ,

0x5: FD_CMD_WRITE. (refer 5.11)

For DCC translating the DCD content to MIOA request, one fdc_command for testing floppy device in DCD looks like this:

```
struct fdc_command
{
        cid = 0x8e
        args_count = 0x5
        args[] = [0x45, 0x12, 0x34, 0x7f, 0x98]
};
```

The command structure will be translated to following MIOA requests

“outb 0x3f5 0x8e”

“outb 0x3f5 0x45”

“outb 0x3f5 0x12”

“outb 0x3f5 0x34”

“outb 0x3f5 0x7f”

“outb 0x3f5 0x98”

Actually, the 0x3f5 is floppy controller’s FIFO port. (Refer 5.11)

The snippet above is very simple. Some translation from command structure to MIOA requests would be complex. Taking USB XHCI device for example, we need follow USB XHCI specification to generate MIOA requests. Some MIOA request may need special attributes (e.g. XHCI command queue mapping physical memory address, XHCI event queue mapping physical memory address) which should be gotten from previous USB XHCI device initialization.

2.3.3 AFL integration

In this part, you will see why traditional AFL fuzzing is integrated into our whole solution. What is more, you will know how we tailor AFL to be suitable for virtual devices fuzzing with detail in practice.

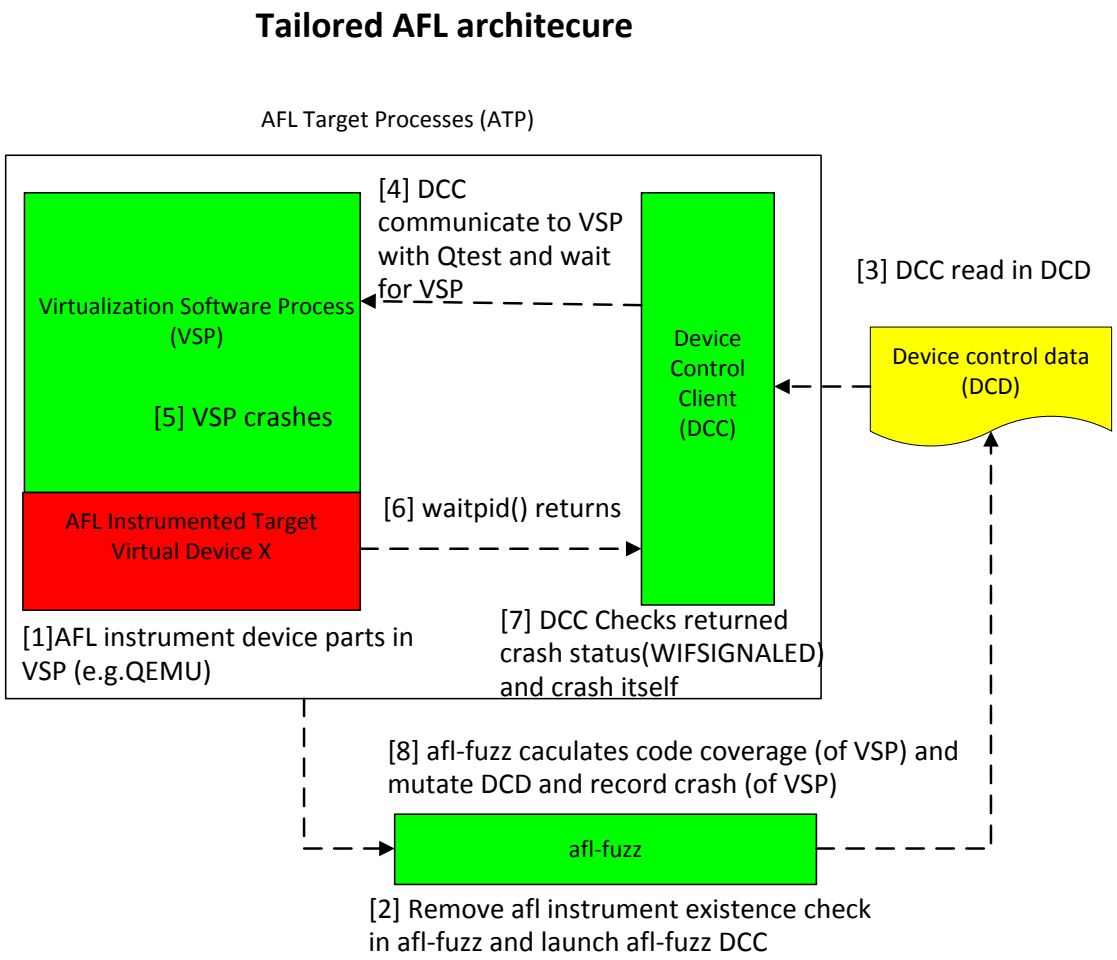

Figure 6 Tailored AFL fuzzing system

The figure above shows the tailored AFL fuzzing system in detail in practice.

2.3.3.1 AFL introduction

American fuzzy lop (i.e. AFL) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.

Actually, AFL is perfect generic fuzzing method towards open-source applications (e.g. media player, pdf reader) which handle structured data in file without complex semantic and grammars. Also AFL provides distinct code coverage feedback and mutation strategy to increase code coverage gradually.

Based on the perfect compile-time instrumentation and code coverage mechanism, AFL has been enhanced to adjust to non-traditional scenario. For example, fuzzing kernel file system (i.e. fuzzing ext4 file system type for file read/write/mount, referring to 5.5), Linux system call fuzzing in QEMU (e.g. Triforce, Referring to 5.6).

2.3.3.2 Basic Workflow with AFL integrated

2.3.3.2.1 Setup instrumentation for target devices

Instrumentation at compile time using AFL-gcc to compile the target virtual device source code (for example: hcd-xhci.c in QEMU if your target device is USB) is an efficient way for code coverage trace.

If the target virtual device is NOT open source (for example: vmware), we reverse the binary code(i.e. the executable file of VSP process) to determine the target virtual device ‘s process address scope, and patch binary instruction into the binary code within the scope.

By instrument the code of target devices in VSP process leaving the other part un-touched at all, the code coverage calculation and feedback could be more accurate. And also the whole fuzzing performance could be higher.

2.3.3.2.2 Setup trace and feedback with AFL

Using AFL-fuzz to launch DCC with DCD as input argument, then DCC will fork VSP attaching with target virtual device. The VSP would then load up CBS which would visit and fuzz target virtual devices.

Internally, afl-fuzz would mutate the DCD according to its mutation strategies (e.g. bit flip, insert, delete, splicing) and trigger ATP(i.e. VSP and DCC) running. Whenever ATP is finished, new code coverage is calculated and new DCD is mutated for next try. Here below is the glance of afl-fuzz GUI which target device is USB.

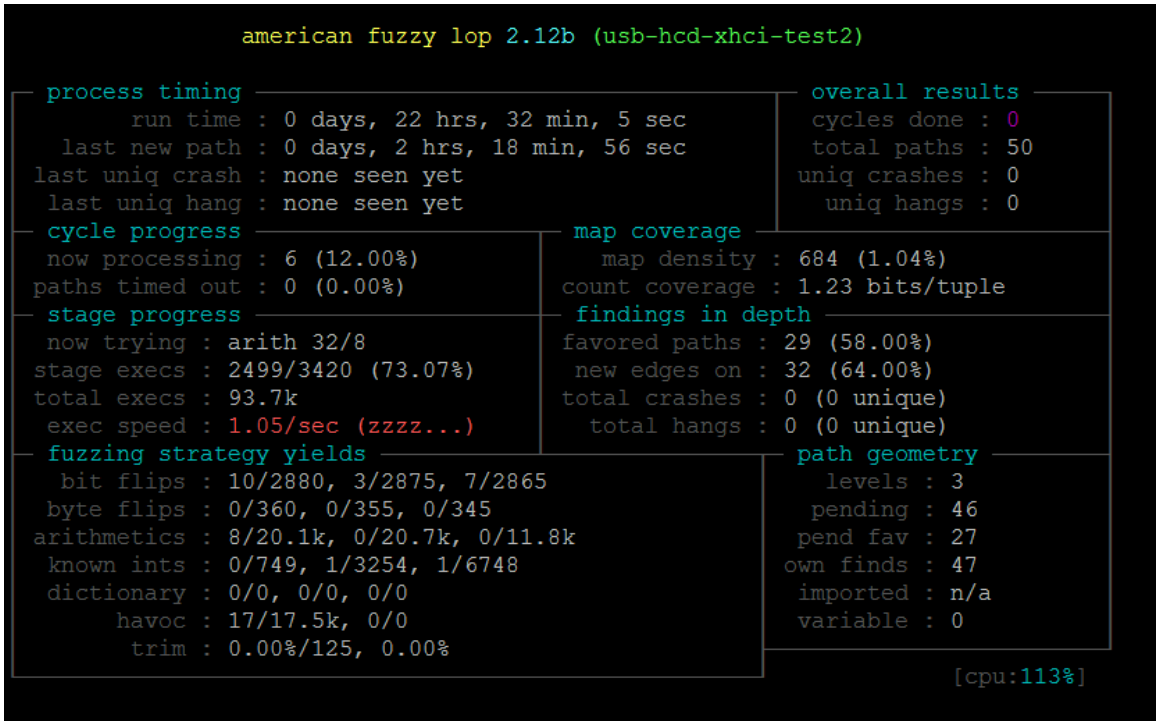

Figure 7 Fuzzing UI glance

2.3.3.2.3 Handle device controls internally in AFL loop

MIOPS in CBS is a memory/IO access proxy server is responsible for handle MIOA request and execute it in CBS.

DCC is the bridge between DCD and MIOPS. DCC would read DCD as device control command and then convert it to MIOA request to MIOPS.

I. Encoding MIOA requests to DCD

All the MIOA requests should be encoded to DCD which is suitable for AFL fuzz. Basically, every field in MIOA is encoded into special Bits/Bytes. AFL would naturally mutate(e.g. Bit/Bytes flip) this kind of data structure and generate new DCD for further fuzz.

II. Devices communication loop

Step 1: VSP starts up

VSP would load up CBS as its guest VM which is relatively light weight. CBS in guest VM would start and discover the attached device, and then start MIOPS.

Step 2: After MIOPS is ready, DCC would communicate with MIOPS to initialize the target devices. For example, if the target virtual device is USB XHCI device, the basic information would include doorbell address, command ring address, event address.

Step 3: After target device is ready, DCC will read in DCD as sequence of commands, translate the command into MIOA request and send the request to CBS via virtual serial port finally.

Step 4: MIOPS would receive the MIOA request through virtual serial port, will execute corresponding request. For example, if MIOPS receives request “out 0xf000 0x8” in MIOA request, it will use ” out” instruction to access port address 0xf000 with data 0x8.

2.3.3.3 Why to tailor AFL

Just like the other non-traditional AFL fuzzing, virtual device fuzzing in virtual machine (e.g. QEMU) is NOT naturally suitable to be fuzzed by traditional AFL.

The major obstacles which are different from traditional AFL fuzzing could be listed as below:

I. Encoding DCD into structured data file

Traditionally, there exists an input file which is handled by target process during afl-fuzz, and the data structure of the input file should be formatted suitable for mutation (e.g. flip-bytes, flip-bit, insert, delete, spice). So we decide to encode the bus/device protocol in DCD and extract the CDC as input file.

II. AFL fuzz multiple processes

Actually there exists two processes (VSP and DCC)using in our solution. How to synchronize the two processes including crash, start/end synchronization to “cheat” afl-fuzz as if it is fuzzing one process seems a potential headache.

III. Instrument target process in part

In general, we only care about virtualization device part instead of the whole one in VSP during afl-fuzz. Because unnecessary part except the virtualization device (e.g. QEMU initialization, hypervisor, binary translation) would not only drop the fuzzing performance but also introduce unexpected crash/hang/abortion/failure for AFL fuzzing.

So we should find tricks to instrument part of the source code when AFL compile or instrument part of the binary code for the executable file.

2.3.3.4 Tips in practice

These are key tricks which we found out in practice for your reference.

2.3.3.4.1 Part instrumentation during compile for open-source software

Actually, during AFL compilation, afl-gcc would compile the source code into the ASM code(i.e. assemble code) using gcc , parse all branch instruction in ASM code , insert extra ASM code for instrumentation and compile the modified ASM code to object file finally.

Hence, we introduce ticks which we call “conditional compilation”. Actually, we could afl-gcc the interesting source file leaving the other source file compiled by gcc instead of afl-gcc. The afl-instrumented object file could be linked into executable file without any side effect.

Here below is the sample code in make file for conditional compilation. It means when any source file (*.c) related with target (*.o) hcd-xhci is found, then we would use afl-gcc to compile. Any other source file would be compiled by gcc.

```
%.o: %.c
        $(call quiet-command,\
        if [ $@ = "hw/usb/hcd-xhci.o" -o $@ = "hw/usb/dev-storage.o" ] ;\
        then \
                echo afl-gcc... $< $@    ;\
                afl-gcc $(QEMU_INCLUDES) $(QEMU_CFLAGS) $(QEMU_DGFLAGS) $(CFLAGS)
$($@-cflags) -c -o $@ $< ;\
        else \
```

```
		echo cc... $<  $@ ;  \
		$(CC) $(QEMU_INCLUDES) $(QEMU_CFLAGS) $(QEMU_DGFLAGS) $(CFLAGS)
$($@-cflags) -c -o $@  $< ;\
```

2.3.3.4.2 Part instrumentation executable file for close-source software

Actually, some virtualization software such as Vmware WorkStation is close-sourced. The best practice for setting up trace towards target software is to instrument its executable file. Usually, you should detect the (virtual devices) code scope for instrument by reverse engineering. Like instrumentation in compile time, we should modify and patch(e.g. inline hook) all the branch instruction using trampoline code like _afl_maybe_log(). The trampline code would look like this:

```
static const u8* trampoline_fmt_32 =

"/* --- AFL TRAMPOLINE (32-BIT) --- */\n"

".align 4\n"

"leal -16(%%esp), %%esp\n"

"movl %%edi,  0(%%esp)\n"

"movl %%edx,  4(%%esp)\n"

"movl %%ecx,  8(%%esp)\n"

"movl %%eax, 12(%%esp)\n"

"movl $0x%08x, %%ecx\n"

"call __afl_maybe_log\n"

"movl 12(%%esp), %%eax\n"

"movl  8(%%esp), %%ecx\n"

"movl  4(%%esp), %%edx\n"

"movl  0(%%esp), %%edi\n"
```

"leal 16(%%esp), %%esp\n"

"/ --- END --- */\n"*

We could add extra executable segment to contain trampoline code and initialization code. At last, we should modify the entry point of the executable file to our initialization code.

2.3.3.4.3 Multiple process fuzzing in afl-fuzz

Actually, our aim is to afl-fuzz DCC process, and at the same time DCC would launch VSP process which is instrumented and traced for code coverage control. Internally, VSP process would synchronize its important events(e.g. crash, start/exit process)in its life cycle with DCC process.

I. AFL naturally shares instrumentation trace info crossing processes using shared memory

This is good news for fuzzing two-process targets. In AFL, actually, all branch instructions in target process would be instrumented with API like

static inline void afl_maybe_log(abi_ulong cur_loc)

and the cur_loc means offset address during compilation.

The instrumentation info (i.e. cur_loc variable) is hashed and recorded into system shared memory (i.e. Linux shmat) which is cross processes if the instrumented target process is launched.

AFL would NOT care whether the target process is single process or multiple processes if only the instrumentation info is generated in shared memory.

II. Remove afl instrumentation existence check in afl-fuzz

When afl-fuzz launch the target process (DCC in our design), it would check afl instrument existence before processing on. So we should remove this kind of code in afl-fuzz because DCC has not any afl instrument code in its process.

Actually, the command for afl fuzzing is like this:

${test_prog} is DCC.

${test_root}/IN/ contains structured data file of DCD.

afl-fuzz -t 90 -m 1024 -i ${test_root}/IN/ -o ${test_root}/OUT/ ${test_prog} @@

(@@ means file path of every file in ${test_root}/IN/ folder.)

III. Check crash status for VSP

Another obstacle is that: traditionally afl-fuzz would just monitor and collect crashes from target process which is DCC process in our solution, however, what interesting crashes we really care about is VSP process. So the crash of VSP should be delivered to DCC as one practical trick in our design.

Actually, there exist many ways you can check crash status of VPS in practice.

1. Fork and Waitpid()

The basic principle is that: DCC would fork and wait for VSP's crash signal and crash DCC itself if any crash happens in VSP. In this way, the crash info of VSP is "delivered" to DCC.

The pseudo- code for crash delivery in DCC would be like this:

```
fpid = fork();
if (fpid == 0)
{
    //In child process
    launchVSP(argv);
    exit(0);
}
else
{
    //In parent process (DCC)
    waitpid(fpid, &status, 0);
    if (WIFEXITED(status))
    {
    //Child process normal exit, bypass
    }
    else if (WIFSIGNALED(status))
```

```
    {
      //Child process crash signal, crash self
      crashMyself();
    }
  }
```

2. Check serial port status

   As one of the basic virtual devices, serial port is usually bond to VSP process life cycle. If the VSP process exits (because of crash), you could check and detect corresponding status change in virtual port. If such status change is found in virtual port, you should also crash DCC process.

3. Real case study

With our solution, we fuzz floppy disk controller and reproduce one critical vulnerability (CVE-2015-3456, Venom) which is reported by Jason Geffner.

The vulnerability exists in QEMU floppy disk controller handling control command code. Floppy disk controller will put command id and its arguments into a buffer which named FIFO buffer. The buffer size is 512. When handling some specific command, floppy disk controller will overflow the buffer.

Our fuzzing strategy is taking floppy disk controller command as attacking surface.

Step 1: Using AFL-gcc to compile fdc.c

We modified the make file (rules.mak) as following:

```
%.o: %.c
    $(call quiet-command,\
    if [ $@ = "hw/block/fdc.o" ] ;\
    then \
        echo afl-gcc... $< $@    ;\
        afl-gcc $(QEMU_INCLUDES) $(QEMU_CFLAGS) $(QEMU_DGFLAGS) $(CFLAGS) $($@-cflags) -c -o $@ $< ;\
    else \
        echo cc... $<  $@ ;  \
        $(CC) $(QEMU_INCLUDES) $(QEMU_CFLAGS) $(QEMU_DGFLAGS) $(CFLAGS) $($@-cflags) -c -o $@  $< ;\
    fi)
```

Figure 8 make file modification for testing fdc

It means that we use afl-gcc to compile source file if fdc.c file is encountered during compiling so that afl instrumentation is set.

Step 2: Designing input case file format (DCD)

One input case file contains several commands like this:

```
struct fdc_command
{
    unsigned char cid;
    unsigned int args_count;
    unsigned int args[0];
};
```

The field cid in fdc_command means the floppy disk controller command ID. As I mentioned above, the structures will be translated to MIOA requests.

Step 3. We prepare 30 input case file (one case input file for each floppy disk control command) for afl mutation.

Step 4: Preparing DCC

1. Parsing input case file and translating to MIOA requests.
2. Fork VSP using following commands:
   mkfifo jack_pipe
   mkfifo jack_pipe1
   qemu-system-x86_64 -bios out/bios.bin -serial pipe:jack_pipe -serial pipe:jack_pipe1
   The bios.bin is CBS.

3. Open pipe, and write MIOA requests to pipe and read response from pipe.
4. Wait the VSP ‘s pid.

Step 5: Using commands to start fuzzing.

afl-fuzz -t 99000 -m 2048 -i IN/ -o OUT/ <DCC command> @@

In our testing environment (8G ram, 4 cores CPU), we reproduce the Venom vulnerability on the QEMU 2.3 after running the solution about 1.5 hours.

4. Conclusion

Virtualization security would keep increasingly “hot” because more and more enterprises have embraced virtualization infrastructure for cloud computing. For the virtual device bug hunting, we have designed novel fuzzing approach which is light-weight with good performance, easy to be ported and also provide fuzzing code coverage feedback and control.

We have introduced the implementation of our solution in detail, and also best practice we have ever encountered.

As to prove and verify our approach, using the solution we have fuzzed floppy disk controller and **reproduce** one critical vulnerability (CVE-2015-3456) which is reported by Jason Geffner.

5. Reference

5.1 http://venom.crowdstrike.com/

5.2 http://www.slideshare.net/CanSecWest/csw2016-tang-virtualizationdevice-emulator-testing-technology

5.3 http://lcamtuf.coredump.cx/AFL/

5.4 https://www.youtube.com/watch?v=O9P7kXm5WSg

5.5 Filesystem Fuzzing with American Fuzzy Lop

https://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf

5.6 Triforce-run-afl-on-everything

https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/

5.7 https://www.seabios.org/SeaBIOS

5.8 https://www.seabios.org/Execution_and_code_flow

5.9 http://wiki.qemu.org/Features/QTest

5.10 http://www.intel.com/content/www/us/en/io/universal-serial-bus/extensible-host-controler-interface-usb-xhci.html

5.11 http://wiki.osdev.org/Floppy_Disk_Controller