When virtualization encounters AFL

Jack Tang, @jacktang310
Moony Li, @Flyic
What We Will Cover

- Who We Are
- Approach
- Implementation
- Case Study
- Demo
Who We Are
Jack Tang

- @jacktang310
- 10+ years security
- Browser
- Document
- Mac/Windows Kernel
- Virtualization Vulnerability
<table>
<thead>
<tr>
<th>Approach</th>
<th>Notes</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformance fuzzing test</td>
<td>1.Symbol Execution</td>
<td>1.Fuzz deeper</td>
<td>1.No code coverage 2.Academical</td>
</tr>
</tbody>
</table>
Target: Virtual Devices

Hypervisor

Guest OS
- Kernel Driver
- Virtual Devices

Guest OS
- Kernel Driver
- Virtual Devices

Hardware Devices
Our Approach

- **Portable**
  - Customized BIOS as common format Virtual Disk
  - Serial port
- **Performance**
  - sizeof(BIOS)<sizeof(OS).
- **Direct**
  - Bypass device driver layer
- **Code coverage feedback & control**
  - AFL integration
Implementation

- Architecture
- Modules
- Work Flow
- AFL Tips
<table>
<thead>
<tr>
<th>Fuzzing UI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>process timing</strong></td>
</tr>
<tr>
<td>run time : 0 days, 22 hrs, 32 min, 5 sec</td>
</tr>
<tr>
<td>last new path : 0 days, 2 hrs, 18 min, 56 sec</td>
</tr>
<tr>
<td>last uniq crash : none seen yet</td>
</tr>
<tr>
<td>last uniq hang : none seen yet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing : 6 (12.00%)</td>
<td>map density : 684 (1.04%)</td>
</tr>
<tr>
<td>paths timed out : 0 (0.00%)</td>
<td>count coverage : 1.23 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying : arith 32/8</td>
<td>favored paths : 29 (58.00%)</td>
</tr>
<tr>
<td>stage execs : 2499/3420 (73.07%)</td>
<td>new edges on : 32 (64.00%)</td>
</tr>
<tr>
<td>total execs : 93.7k</td>
<td>total crashes : 0 (0 unique)</td>
</tr>
<tr>
<td>exec speed : 1.05/sec (zzzz...)</td>
<td>total hangs : 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips : 10/2880, 3/2875, 7/2865</td>
<td>levels : 3</td>
</tr>
<tr>
<td>byte flips : 0/360, 0/355, 0/345</td>
<td>pending : 46</td>
</tr>
<tr>
<td>arithmetics : 8/20.1k, 0/20.7k, 0/11.8k</td>
<td>pend fav : 27</td>
</tr>
<tr>
<td>known ints : 0/749, 1/3254, 1/6748</td>
<td>own finds : 47</td>
</tr>
<tr>
<td>dictionary : 0/0, 0/0, 0/0</td>
<td>imported : n/a</td>
</tr>
<tr>
<td>havoc : 17/17.5k, 0/0</td>
<td>variable : 0</td>
</tr>
<tr>
<td>trim : 0.00%/125, 0.00%</td>
<td>[cpu: 113%]</td>
</tr>
</tbody>
</table>
• Customized BIOS System (Seabios)
  • Decision principle:
    • Portability
  • Contains MIOPS (Memory and IO space Operation Proxy Server)
  • VSP (Virtual Software Process) is its runtime instance

• Virtual serial port
  • Communication bridge
Modules- How To Customize CBS 2/3

• Traditional BIOS vs Customized BIOS

  POST phase
  (Power On Self Test)

  Boot phase
  (Boot Operating System)

  BIOS runtime service phase
  (BIOS service for runtime OS)
POST phase
(Power On Self Test)

- Detect physical memory
- Platform hardware setup (for example: PCI bus, clock…)
- Necessary device hardware setup (for example: serial port device for communication)
- Recognize the devices
- Start to run MIOPS (Memory and IO space Operation Proxy Server) handle loop.
• Memory and IO space Operation Proxy Server
• MIOA Request- (Memory I/O Access) request
• Tips:
  • Remove interruption in CBS
  • MIOPS polls virtual device result of request with timeout
Job:
  - Receive MIOA
  - Execution it

MIOA example:
  “inb <address>”
  “inw <address>”
  “inl <address>”
  “outb <address> <value>”
  “outw <address> <value>”
  “outl <address> <value>”
  “write <address> <value> <length>”
  “read <address> <length>”

```c
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_inl:
        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;
    }
    return pReq;
}
```
• Device Control Client
• DCD – Device Control Data
  • Formatted suitable for AFL
• Steps in Job:
  • 1. Launch VSP to load CBS
    
    mkfifo jack_pipe
    mkfifo jack_pipe1
    qemu-system-x86_64 -bios out/bios.bin -serial
    pipe:jack_pipe -serial pipe:jack_pipe1
  
• 2. Ping MIOPS in CBS
• Steps in Job:

  • 3. Initialize target virtual device
     • e.g. USB XHCI device initialization

  • 4. Parse DCD to MIOA
Step 1: Instrument the target devices

- Source code is available
  - Afl-gcc compile source code in part

- Source code NOT available
  - Instrument executable file statically + restricted instrumentation range
Step 2: Launch Afl-fuzz loop

- `afl-fuzz -t 90 -m 2048 -i ${test_root}/IN/ -o ${test_root}/OUT/ ${DCC} @@`

- Afl-fuzz will automatically mutate DCD and calculate code coverage in the loop
Workflow Overview - Step 2

AFL Target Processes (ATP)

- Virtualization Software Process (VSP)
  - Customized BIOS System (CBS)
  - Memory and IO space Operation Proxy Server (MIOPS)
  - Memory / IO space
- Virtual Device1 (e.g. NIC, e1000)
- Virtual Device2 (e.g. SCSI)
- Memory/IO Access request (MIOA request)
- Virtual Device3 (e.g. Printer)
- Virtual Device4 (e.g. Audio Card)

[1] instrument target device
[2] afl-fuzz loop
[3] DCC read in DCD
[4] MIOPS execute request
[5] afl-fuzz calculate code coverage and mutate

Host OS

Device control data (DCD)
Work Flow -
Encode DCD and read in – Step 3

- Step 3: Encode MIOA Requests to DCD and read into DCC

```c
typedef struct _command_entry {
  u32 _slotid;
  u32 _command_id;
  void* _inctx;
  u8 _input_buf[32];
} command_entry_t;
```
Workflow Overview – Step 3

1. Instrument target device
2. AFL fuzz loop
3. DCC read in DCD
4. MIOPS execute request
5. AFL fuzz calculate code coverage and mutate
Work Flow - Control Devices Communicate –Step 4/4

VSP

CBS

MIOPS

Memory / IO space

In/out/DMA

Instrumented Target Virtual Device X

Initialize

Communicate

Finalize

Initialize

Communicate

Finalize

DCC

DCD
Step 4: Communication Loop

- CBS starts up
  - Start & discover devices
  - Start MIOPS in it

- MIOPS starts up
  - Initiate target devices
    - E.g. doorbell/command/event address of USB XHCI device

- Communicate between MIOPS and DCC until crash
  - DCC read in DCD
  - MIOPS execute data
Work Flow - In view of AFL

AFL Target Processes (ATP)

[1] AFL instrument device parts in VSP (e.g. QEMU)

Virtualization Software Process (VSP)


AFL Instrumented Target Virtual Device X

[3] DCC read in DCD

Device control data (DCD)

[4] DCC communicate to VSP with Qtest and wait for VSP

Device Control Client (DCC)

[5] VSP crashes

[6] waitpid() returns

[7] DCC Checks returned crash status (WIFSIGNALED) and crash itself

[8] afl-fuzz calculates code coverage (of VSP) and mutate DCD and record crash (of VSP)
Tips: Compile the source code in part using afl-gcc, for example:

Makefile:

%.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 $@ $< ; |

Tips: Instrument the executable file for close-source software

```
static const u8* trampoline_f =
".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"
```

**AFL Tips - Part Instrumentation – Instrument PE 2/2**
AFL Tips - Multiple Processes 1/3

Target Process (e.g. VSP)

AFL Instrument

Launcher (e.g. DCC)

Input Data (e.g. DCD)

afl-fuzz
AFL Tips-
Multiple Processes 2/3

- Trace info is shared across processes in nature
  - \texttt{static inline void afl\_maybe\_log(abi\_ulong cur\_loc)}
  - Trace \texttt{cur\_loc} in shared memory mechanism

- Remove instrumentation check
  - In Afl-fuzz
  - Because Launcher (e.g. DCC) is not instrumented

- Deliver crash info from Target to Launcher
  - fork & waitpid
  - Check connection (e.g. serial port status)
AFL Tips -
Multiple Processes 3/3

```c
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();
    }
}
```
AFL Tips - Misc

• Adjust parameter for afl-fuzz to avoid start failure

  • -t, set more time for fuzz cycle
  • -m, set more memory for virtual machine

  • E.g. afl-fuzz -t 9 -m 2048 -i ${test_root}/IN/ -o ${test_root}/OUT/ ${DCC} @@

• …
What We Will Cover

• Case Study
  • Fuzz FDC and Reproduce Venom Vulnerability (CVE-2015-3456)

• Demo
1. afl-gcc  fdc.c

```bash
%.o: %.c

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

2. Designing input case file(i.e. DCD) format

```c
struct fdc_command
{
  unsigned char cid;
  unsigned int args_count;
  unsigned int args[0];
};
```
3. Prepare 30 input case file
   • each case for one floppy disk controller command
     • For example: FD_CMD_READ, FD_CMD_WRITE, FD_CMD_SEEK...

4. Preparing DCC
   • Parsing input case file and translating to MIOA

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

```
"outb 0x3f5 0x8e"
"outb 0x3f5 0x45"
"outb 0x3f5 0x12"
"outb 0x3f5 0x34"
"outb 0x3f5 0x7f"
"outb 0x3f5 0x98"
```
5. Using commands to start testing
afl-fuzz -t 99000 -m 2048 -i IN/ -o OUT/ <DCC command> @@
Reference

- http://venom.crowdstrike.com/
- http://www.slideshare.net/CanSecWest/csw2016-tang-virtualizationdevice-emulator-testing-technology
- http://lcamtuf.coredump.cx/AFL/
- https://www.youtube.com/watch?v=O9P7kXm5WSg
- Filesystem Fuzzing with American Fuzzy Lop
  https://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf
- Triforce-run-afl-on-everything
- https://www.seabios.org/SeaBIOS
- https://www.seabios.org/Execution_and_code_flow
- http://wiki.qemu.org/Features/QTest
- http://wiki.osdev.org/Floppy_Disk_Controller
Thanks very much