

【PWN】 AWD 技巧

开始之前：在了解 AWD 的 PWN 之前，我们需要考虑 AWD 中 PWN 的位置，在大部分 AWD 比赛中，WEB 的得分都会变得非常困难，其涉及知识面广，难度大，调试时间长，并且简单的修复也可能会需要很长的时间，但 PWN 相对没有那么复杂。

就算是这两年很火的 LLVM 等题型，其实也就是加大了逆向难度，而 PWN 本身难度依然不大，因此，大部分的比赛中，PWN 都会成为拉开差距的点，虽然攻击依旧困难，但是修复是相对很简单的（只要出题人别瞎出。。）

PWN 题攻击

这个其实就没什么好说的了，和常规 CTF 那是一模一样的，比较大的区别应该就是 AWD 中的 PWN 一般很少会出堆题，主要是因为太好修了。。。为了应对这个问题，出题人也经常会将堆和栈结合在一起出，比较经典的就是 setcontext、打 environ 指针之类的方法，具体怎么修我们下文中会提。

PWN 题修复

关于 pwn 的修复这个东西我们这里考虑两种场景，第一个给了源码时的修复，第二个是没给源码时的修复

有源码时的基础修复

先考虑有源码时的修复，这个其实很好操作，主要是熟练 gcc g++ 的各种指令，还有就是要能快速发现漏洞点，由于 pwn 题中经常是溢出类题型和 UAF 类题型偏多，所以要分别考虑这两种情况，先看有码的溢出类题型（要非常注意 strcat strcpy 等会造成 off by null 的函数

```
int main(){
    char buf[24];
    scanf("%s", buf);
    return 0;
}
```

像这种那肯定是改 %s 为 %23s，不能有更长的了，或者适当增扩 buf 的大小，而且它方法也是类似的，有源码的情况下那可太好修复了

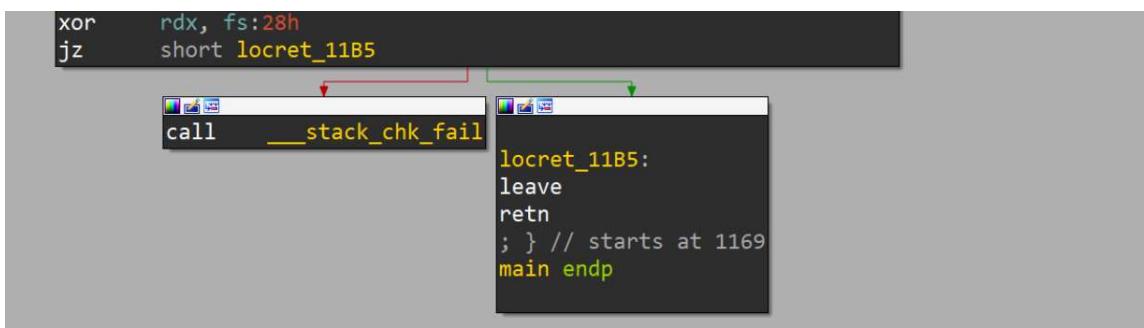
无源码时的程序修改

那么无源码应该怎么修复？这似乎就是 pwn 里面的 patch 了

先列出来几张表，下面是 jmp 的相关指令，对于负数等情况需要用到

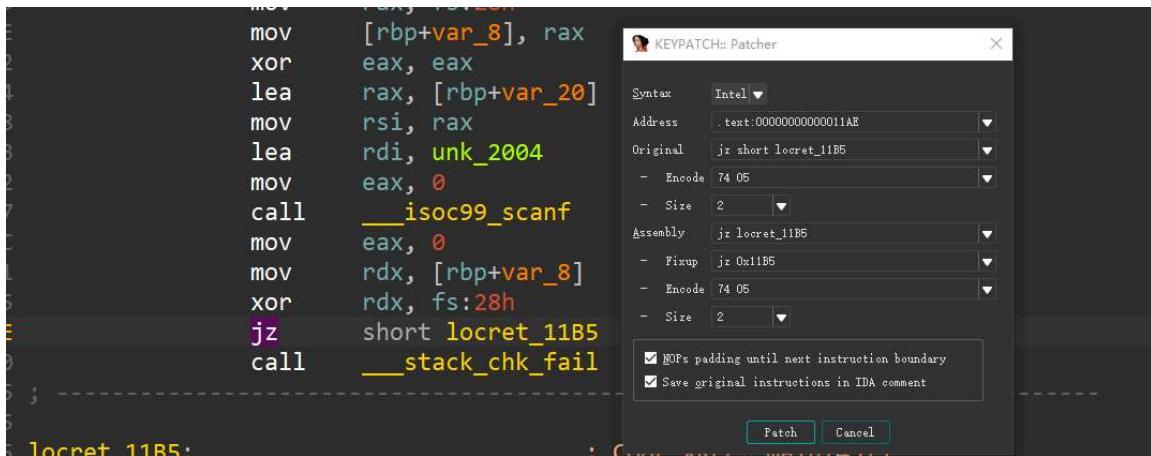
指令	机器码	指令	机器码
jmp	EB XX	jz	74 XX
je	74 XX	jne	75 XX
jg	7F XX	jge	7D XX
jl	7C XX	jle	7E XX
ja	77 XX	jae	73 XX
jb	72 XX	jbe	76 XX
jna	76 XX	jnb	73 XX
jnae	72 XX	jnc	73 XX
jnb	73 XX	jng	7E XX
jnge	7C XX	jnl	7D XX

现在我们以单纯的篡改程序的思想来看下面这个逻辑（注意不是修复哦，只是试一试改程序）

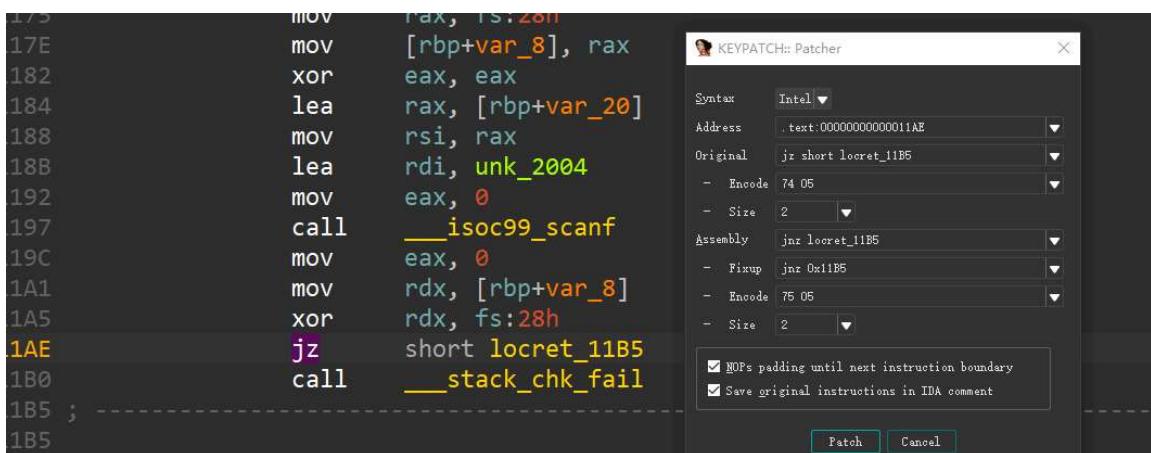


这是一个很明显的 canary，那么我们想让这个 canary 反过来应该怎么做呢？即不溢出的时候调用 `_stack_chk_fail`，溢出的时候不调用

选择 IDA 中的 key -> patcher



可以发现 jz 的指令是 74 05，即 jz \$.+5，我们将其修改为 jnz



指令就变成了 75 05，当然，也可以使用 pwntools 的 asm，也可以帮助我们快速找 opcode

然后选择 IDA 中的 edit ->patch program -> apply patches to input file 即可保存文件，然后去跑一跑就会发现雀食达到了我们要的效果

```
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./stackoverflow
123
*** stack smashing detected ***: terminated
Aborted (core dumped)
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./stackoverflow
2222222222222222222222222222222222222222
yeuoly@yeuoly-virtual-machine:~/Desktop/test$
```

常规补丁

好这是上面是第一个最简单的 patch 修改指令，下面我们来一个难点的，对于 scanf("%s") 我们应该如何 patch？

我搜了很多其他师傅的思路，第一个是利用 eh_frame 段，这个段在程序正常运行的时候一般用不到，但是它是被赋予了 X 权限的，也就是说可执行，你比如说对于这个

```
● 6  v5 = __readfsqword(0x28u);
● 7  __isoc99_scanf(&unk_2004, v4, envp);
● 8  return b;
● 9 }
```

这里 call 了 __isoc99_scanf 的 PLT，那么我们在 eh_frame 上自己写一个函数，让它 call 到我们自己的函数上去，而我们自己的函数我们就严格限制输入大小，注意这里传入的第二个参数为缓冲区地址

我们这里非常简单暴力，自己通过 syscall 来输入，然后我们去修改 eh_frame，编写如下脚本用于生成字节码

```
from pwn import *

context.arch = 'amd64'

code = '''
    xor rdi, rdi
    mov rdx, 23
    xor rax, rax
    syscall
    ret
'''

for i in list(asm(code, arch='amd64')):
    print(hex(i)[2:]).rjust(2, '0'), end = ' ')
print()
```

然后去修改 **call scanf**

```
.eh_frame:0000000000002050 unk_2050 db 48h ; H
.eh_frame:0000000000002050
.eh_frame:0000000000002051 db 31h ; 1
.eh_frame:0000000000002052 db 0FFh
.eh_frame:0000000000002053 db 48h ; H
.eh_frame:0000000000002054 db 0C7h
.eh_frame:0000000000002055 db 0C2h
.eh_frame:0000000000002056 db 17h
.eh_frame:0000000000002057 db 0
.eh_frame:0000000000002058 db 0
.eh_frame:0000000000002059 db 0
.eh_frame:000000000000205A db 48h ; H
.eh_frame:000000000000205B db 31h ; 1
.eh_frame:000000000000205C db 0C0h
.eh_frame:000000000000205D db 0Fh
.eh_frame:000000000000205E db 5
.eh_frame:000000000000205F db 0C3h
```

最后，修改 call scanf 到 call eh_frame 上来，由于这里我们开了 PIE，所以需要 call 一个相对坐标

先看看原本的地方

```
.text:0000000000001192          mov    eax, 0
.text:0000000000001197          call   __isoc99_scanf
.text:000000000000119C          mov    eax, 0
```

我们需要把这个函数除了对我们有用的部分全部劫持到我们在 eh_frame 上的代码，多余的部分全部改为 nop

```
mov    [rbp+var_8], rsi
xor    eax, eax
lea    r15, unk_2050
lea    rsi, [rbp+var_20]
call   r15 ; unk_2050
nop
```

这是我改完以后的，主要就是 lea r15, qword_2050 这个点，我们需要使用相对寻址来计算目标代码的位置，生成 opcode 的代码如下，当然也可以直接用 keypatch

```
from pwn import *

context.arch = 'amd64'

code = '''
    lea r15, [rip + 0xec5]
    lea rsi, [rbp - 0x20]
```

```

    call r15
    ...

for i in list(asm(code, arch='amd64')):
    print(hex(i)[2:].rjust(2, '0'), end = ' ')
print()

```

至于这个 **0xec5** 怎么来的，首先，lea 的长度为 7，所以 lea 的下一条指令的地址是 0x118b，使用 0x2050 - 0x118b 得到 0xec5

好了现在我们将程序打包到 gdb 里调试看看

```

Registers:
RBX 0x55555555551c0 (<__tbc_csu_init>) ← 0x8d4c5741fa1e0ff3
RCX 0x55555555551c0 (<__tbc_csu_init>) ← 0x8d4c5741fa1e0ff3
RDX 0x7fffffffded8 → 0x7fffffff2e253 ← 'SHELL=/bin/bash'
RDI 0x1
RSI 0x7fffffffdec8 → 0x7fffffff2e22b ← '/home/yeuoly/Desktop/test/stackoverflow'
R8 0x0
R9 0x7fffff7fe0d50 ← endbr64
R10 0x7fffff7ffcf68 ← 0x6fffffff
R11 0x206
R12 0x5555555555080 (_start) ← 0x8949ed31fa1e0ff3
R13 0x7fffffffdec0 ← 0x1
R14 0x0
R15 0x555555556050 ← 0xc748ff3148fe8948
RBP 0x7fffffffdd0 ← 0x0
RSP 0x7fffffffdda8 → 0x5555555518e (main+37) ← 0x9090909090909090
RIP 0x555555556050 ← 0xc748ff3148fe8948

[ DISASM ]
▶ 0x555555556050  mov   rsi, rdi
0x555555556053  xor   rdi, rdi
0x555555556056  mov   rdx, 0x17
0x55555555605d  xor   rax, rax
0x555555556060  syscall
0x555555556062  ret

0x555555556063  or    byte ptr [rax + 0x14000001], dl
0x555555556069  add   byte ptr [rax], al
0x55555555606b  add   byte ptr [rax + rax], bl
0x55555555606e  add   byte ptr [rax], al
0x555555556070  adc   al, dh

[ STACK ]

```

现在雀食是可以跳转过来执行了，但是还有最后的问题，即 eh_frame 没有执行权限，这个我们通过 IDA 修改 ELF 的头来实现

```
dd 1 ; Type: LOAD
dd 4 ; Flags
dq 0 ; File offset
dq 0 ; Virtual address
dq 0 ; Physical address
dq 680h ; Size in file image
dq 680h ; Size in memory image
dq 1000h ; Alignment

; PHT Entry 3
dd 1 ; Type: LOAD
dd 5 ; Flags
dq 1000h ; File offset
dq offset _init_proc ; Virtual address
dq 1000h ; Physical address
dq 245h ; Size in file image
dq 245h ; Size in memory image
dq 1000h ; Alignment

; PHT Entry 4
dd 1 ; Type: LOAD
dd 7 ; Flags
dq 2000h ; File offset
dq offset _IO_stdin_used ; Virtual address
dq 2000h ; Physical address
dq 158h ; Size in file image
dq 158h ; Size in memory image
dq 1000h ; Alignment

; PHT Entry 5
dd 1 ; Type: LOAD
dd 6 ; Flags
dq 2DB0h ; File offset
dq offset __frame_dummy_init_array_entry ; Virtual address
dq 3DB0h ; Physical address
dq 260h ; Size in file image
```

Type 一定得是 LOAD，而我们的补丁打在第三个 LOAD 里，所以我将 Flags 修改为了 7，当然啊，这样做其实是非常不负责的，但是做题嘛，怎么快怎么来，如果想要负责的话，那就得加一个 LOAD，并且还需要处理好 PIE，至少现在是正常跑起来了也不会溢出

替换 PLT|GOT 修复

如果说程序东西不多而且敏感函数就调用那么一两次的话我们也可以通过修改重定位表来修复

先来看源码

```
#include<stdio.h>

int main(){
    char buf[24];
    scanf("%s", buf);
    printf(buf);
    return 0;
}
```

这里我们只考虑修复 printf 而不考虑修复 scanf，修 scanf 参考上文的方案即可

编译

```
| gcc replace_plt.c -o replace_plt -fno-stack-protector -z lazy
```

修 printf，这里我们通过篡改 printf 为 puts 来实现，但是源程序里并没有 puts，所以需要更改 dyn

我们找到 printf 的 dysym

```
; ELF Symbol Table
Elf64_Sym <0>
Elf64_Sym <offset aITmDeregistert - offset unk_488, 20h, 0, 0, \ ; "_ITM_deregisterTMCloneTable"
    offset dword_0, 0>
Elf64_Sym <offset aPrintf - offset unk_488, 12h, 0, 0, offset dword_0,\ ; "printf"
    0>
Elf64_Sym <offset aLibcStartMain - offset unk_488, 12h, 0, 0, \ ; "__libc_start_main"
    offset dword_0, 0>
Elf64_Sym <offset aGmonStart - offset unk_488, 20h, 0, 0, \ ; "__gmon_start__"
    offset dword_0, 0>
Elf64_Sym <offset aIsoc99Scanf - offset unk_488, 12h, 0, 0, \ ; "__isoc99_scanf"
    offset dword_0, 0>
Elf64_Sym <offset aITmRegisterTmc - offset unk_488, 20h, 0, 0, \ ; "_ITM_registerTMCloneTable"
    offset dword_0, 0>
Elf64_Sym <offset aCxaFinalize - offset unk_488, 22h, 0, 0, \ ; "__cxa_finalize"
    offset dword_0, 0>
```

我们去修改 printf 为 puts

```
; ELF Symbol Table
Elf64_Sym <0>
Elf64_Sym <offset aItmDeregistert - offset unk_488, 20h, 0, 0, \ ; "_ITM_deregisterTMC"
    offset dword_0, 0>
Elf64_Sym <offset aPrintf - offset unk_488, 12h, 0, 0, offset dword_0,\ ; "puts\x00f"
    0>
Elf64_Sym <offset aLibcStartMain - offset unk_488, 12h, 0, 0, \ ; "__libc_start_main"
    offset dword_0, 0>
Elf64_Sym <offset aGmonStart - offset unk_488, 20h, 0, 0, \ ; "__gmon_start__"
    offset dword_0, 0>
Elf64_Sym <offset aIsoc99Scanf - offset unk_488, 12h, 0, 0, \ ; "__isoc99_scanf"
    offset dword_0, 0>
Elf64_Sym <offset aItmRegistertmc - offset unk_488, 20h, 0, 0, \ ; "_ITM_registerTMCloneTable"
    offset dword_0, 0>
Elf64_Sym <offset aCxaFinalize - offset unk_488, 22h, 0, 0, \ ; "__cxa_finalize"
    offset dword_0, 0>

; ELF String Table
unk_488      db 0          ; DATA XREF: LOAD:00000000000003E0↑o
                ; LOAD:00000000000003F8↑o ...
aLibcSo6      db 'libc.so.6',0 ; DATA XREF: LOAD:0000000000000538↓o
aIsoc99Scanf db '_isoc99_scanf',0 ; DATA XREF: LOAD:0000000000000440↓o
aPrintf       db 'puts',0,'f',0 ; DATA XREF: LOAD:00000000000003F8↑o
aCxaFinalize db '__cxa_finalize',0 ; DATA XREF: LOAD:0000000000000470↓o
aLibcStartMain db '__libc_start_main',0 ; DATA XREF: LOAD:0000000000000410↓o
aGlibc27      db 'GLIBC_2.7',0 ; DATA XREF: LOAD:0000000000000548↓o
aGlibc225     db 'GLIBC_2.2.5',0 ; DATA XREF: LOAD:0000000000000558↓o
aItmDeregistert db '_ITM_deregisterTMCloneTable',0

(Synchronized with Hex View-1)
```

然后 patch，再次打开程序的时候可以发现就变成 puts 了

```
nop
lea    rax, [rbp+s]
mov    rdi, rax        ; s
mov    eax, 0
call   _puts
mov    eax, 0
leave
```

这里涉及到的其实是一个 ret2resolve 里的小知识，我们简单提一嘴，就是 Libc 动态链接的过程其实是依靠函数名的，虽然我们平时看到的 PLT GOT 等表中都没有包含函数名，但实际上它储存了一些索引，方便动态链接函数通过 PLT 找到这个函数名，最后再通过对 libc 中对比函数名寻找真实的函数地址，因此修改函数名的修复是有效的，玩的花一点我们还可以把 free 的函数名修改为 atoi 之类的东西，直接 pass 掉大部分堆题。。

不过这样其实还是有问题，因为我们这里使用的是 puts，它比 printf 短，那如果需要替换为 __printf_chk 呢？

还是使用到了 eh_frame 段，主要是因为这个段一般真用不到，不会影响程序正常运行，不过如果想的话也可以自己再加点东西进来

.eh_frame:0000000000002060	db 5Fh ; _
.eh_frame:0000000000002061	db 5Fh ; _
.eh_frame:0000000000002062	db 70h ; p
.eh_frame:0000000000002063	db 72h ; r
.eh_frame:0000000000002064	db 69h ; i
.eh_frame:0000000000002065	db 6Eh ; n
.eh_frame:0000000000002066	db 74h ; t
.eh_frame:0000000000002067	db 66h ; f
.eh_frame:0000000000002068	db 5Fh ; _
.eh_frame:0000000000002069	db 63h ; c
.eh_frame:000000000000206A	db 68h ; h
.eh_frame:000000000000206B	db 6Bh ; k
.eh_frame:000000000000206C	db 0

然后修改 printf 到这里来 $0x2060 - 0x488 = 0x1bd8$

```
    offset dword_0, 0>
Elf64_Sym <offset unk_2060 - offset unk_488, 12h, 0, 0, \
           offset dword_0, 0>
Elf64_Sym <offset aLibcStartMain - offset unk_488, 12h, 0,
```

最后我们执行

```
yeuoly@yeuoly-Virtual-Machine:~/Desktop/test$ chmod +x replace_plt
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./replace_plt
123
./replace_plt: symbol lookup error: ./replace_plt: undefined symbol: __printf_chk, version GLIBC_2.2.5
yeuoly@yeuoly-virtual-machine:~/Desktop/test$
```

虽然说报错了，但是可以发现报错信息也只是说找不到这个符号，我们的替换还是正确的，换成别的函数是没问题的，只是 IDA 这个地方会报错，但是不影响

堆题的一些通用修复

我们都知道堆题高度依赖于 free 函数，如果说 malloc 等函数的不正确使用是造成漏洞的主要原因（比如说溢出就属于长度没控制好），那 free 就是触发漏洞的关键，当然了，还有别的触发方式，我们这里不深入

那么我们如何修复 free 呢？很简单啊，nop 掉就完事了，这样的做法有时候在比赛中比较管用，但也不是啥时候都管用，因此我们还有另外几种方案

因为现在的 PWN 题运行过程动不动就有个 5s 以上（比如说要打 IO 爆破），而 checker 的运行只有短短 1~5 秒，那么为什么不从 alarm 上下手呢？

alarm 函数会在时间到了以后强制结束进程，那么我们可以 patch 程序最开始的 alarm(60) 为 alarm(3)，这样就有可能造成 checker 通过而 exp 不通过，从而通过 check 逻辑

通防

EVILPATCHER

这个东西是我在某次比赛中看到的，基本上通杀。。神挡杀神佛挡杀佛，那次比赛吃了大亏，全场貌似就包括我在内的少数几个人不知道这东西，我们就简单给出链接吧

<https://github.com/TTY-flag/evilPatcher>

自制通防思路

写在前面：这种方法对于 Docker 环境不适用，因为需要 Docker 开启 **CAP_SYS_PTRACE** 权限，但是这玩意会导致 Docker 逃逸，不过当然了，如果是纯黑盒的话，想逃还是很难的，360 等平台使用的似乎就是 Docker 的环境，而永信至诚的平台似乎是 VM 环境，这种方法可以使用

开始吧：要拼速度的话，慢慢找洞慢慢替换肯定是不行的，所以要提前准备好通防操作，在 pwn 中，通防肯定就是不让你开 shell 不让你读文件了，主要就是要关掉 open openat execve 等系统调用了

这里的我的思路是将注入代码写入 eh_frame 中，并且劫持程序入口到这个地方来，先执行注入逻辑，将原本的进程作为子进程启动，父进程通过 ptrace 等函数来监控子进程的行为，如果发现子进程调用了 open openat execve 等函数，那么就 kill 掉子进程

shellcode 的 C 版本如下，为了避免外部引入的库函数用到了 plt 等结构（用到了的话就 g 了），如 ptrace 等函数我都换成了用汇编编写的 syscall

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <stdio.h>

void d(){
    register pid_t child;
    int status;

    struct user_regs_struct regs;

    asm(
        "movq $57, %%rax;"      // Set RAX to 57 (SIGKILL)
        "syscall;"              // Call syscall
        "movl %%eax,%0": "=r"(child)
    );
    if(child == 0) {
        asm(
            "movl $101, %eax;"  // Set EAX to 101 (SIGKILL)
            "xorq %rsi, %rsi;" // XOR RSI with itself
            "xorq %rdx, %rdx;" // XOR RDX with itself
        );
    }
}
```

```

    "xorq %r10, %r10;" 
    "xorq %rdi, %rdi;" 
    "syscall;" //ptrace(PTRACE_TRACEME)
    "movl $39, %eax;" 
    "syscall;" 
    "movl %eax, %edi;" 
    "movl $19, %esi;" 
    "movl $62, %eax;" 
    "syscall;" 
);
//kill(getpid(), 19);
goto end;
} else {
asm(
    "movl $61, %eax;" 
    "xorq %rdi, %rdi;" 
    "syscall;" 
);
//wait(NULL);
while(1){
    asm(
        "movl %0, %%esi"::"r"(child):"%esi"
    );
    asm(
        "movl $101, %eax;" 
        "movl $24, %edi;" 
        "xorq %rdx, %rdx;" 
        "xorq %r10, %r10;" 
        "syscall;" 
    );
    //ptrace(PTRACE_SYSCALL, child, NULL, NULL);
    void *p = (void *)&status;
    asm(
        "movl %0, %%edi"::"r"(child):"%edi"
    );
    asm(
        "movq %0, %%rsi"::"r"(p):"%rsi"
    );
    asm(
        "xorq %rdx, %rdx;" 
        "movl $61, %eax;" 
        "syscall;" //waitpid(pid)
    );
    void *regs_addr = (void *)&regs;
    asm(
        "movq %0, %%r10"::"r"(regs_addr):"%r10"
    );
    asm(
        "movl %0, %%esi"::"r"(child):"%esi"
    );
    asm(
        "movl $12, %edi;" 
        "xorq %rdx, %rdx;" 
        "movl $101, %eax;" 
        "syscall;" //ptrace(PTRACE_GETREGS)
    );
}

```

```

        if((regs.orig_rax >= 56 && regs.orig_rax <= 59 || 
            regs.orig_rax == 231)){
        asm(
            "movl %0, %%edi"::"r"(child):"%edi"
        );
        asm(
            "movl $9, %esi;" 
            "movl $62, %eax;" 
            "syscall;"
        );
        //kill(child, 9);
        break;
    }
}
asm(
    "movl $555, %edi;" 
    "movl $60, %eax;" 
    "syscall;"
);
}
end:
return;
}

int main(){
d();
//execl("/bin/ls", ".");
system("ls");
return 0;
}

```

现在只要调用到了 execve clone vfork fork 等系统调用就会被拦截然后强制终止程序的执行，限制执行了 system，如果没有 d 函数执行的话应该是会执行 ls 指令的，但是现在 d 函数开启了沙盒，那么按道理来说会直接退出

yeuoly@yeuoly-virtual-machine:~/Desktop/test\$./syscall
yeuoly@yeuoly-virtual-machine:~/Desktop/test\$ |

可以发现去雀食没有执行了

后面我又加了几段代码用来平衡栈和保存环境

生成完的 shellcode 如下（没有包含 jmp start，64 位，这个 shellcode 不包含 open 和 openat）

```

57 56 52 51 55 48 89 E5 53 48 83 EC 78 48 C7 C0
39 00 00 00 0F 05 89 C0 89 C3 85 DB 75 2D B8 65
00 00 00 48 31 F6 48 31 D2 4D 31 D2 48 31 FF 0F
05 B8 27 00 00 00 0F 05 89 C7 BE 13 00 00 00 B8
3E 00 00 00 0F 05 E9 97 00 00 00 B8 3D 00 00 00

```

```

48 31 FF 0F 05 89 DE B8 65 00 00 00 BF 18 00 00
00 48 31 D2 4D 31 D2 0F 05 48 8D 45 E4 48 89 45
F0 89 DF 48 8B 45 F0 48 89 C6 48 31 D2 B8 3D 00
00 00 0F 05 48 8D 85 08 FF FF FF 48 89 45 E8 48
8B 45 E8 49 89 C2 89 DE BF 0C 00 00 00 48 31 D2
B8 65 00 00 00 0F 05 48 8B 45 80 48 83 F8 37 76
0A 48 8B 45 80 48 83 F8 3B 76 0C 48 8B 45 80 48
3D E7 00 00 00 75 8E 89 DF BE 09 00 00 00 B8 3E
00 00 00 0F 05 90 BF 2B 02 00 00 B8 3C 00 00 00
0F 05 90 48 83 C4 78 5B 5D 59 5a 5e 5f

```

包含 open 和 openat 的 shellcode

```

57 56 52 51 55 48 89 E5 53 48 83 EC 78 48 C7 C0
39 00 00 00 0F 05 89 C0 89 C3 85 DB 75 2D B8 65
00 00 00 48 31 F6 48 31 D2 4D 31 D2 48 31 FF 0F
05 B8 27 00 00 00 0F 05 89 C7 BE 13 00 00 00 B8
3E 00 00 00 0F 05 E9 B1 00 00 00 B8 3D 00 00 00
48 31 FF 0F 05 89 DE B8 65 00 00 00 BF 18 00 00
00 48 31 D2 4D 31 D2 0F 05 48 8D 45 E4 48 89 45
F0 89 DF 48 8B 45 F0 48 89 C6 48 31 D2 B8 3D 00
00 00 0F 05 48 8D 85 08 FF FF FF 48 89 45 E8 48
8B 45 E8 49 89 C2 89 DE BF 0C 00 00 00 48 31 D2
B8 65 00 00 00 0F 05 48 8B 45 80 48 83 F8 37 76
0A 48 8B 45 80 48 83 F8 3B 76 26 48 8B 45 80 48
3D E7 00 00 00 74 1A 48 8B 45 80 48 83 F8 02 74
10 48 8B 45 80 48 3D 01 01 00 00 0F 85 74 FF FF
FF 89 DF BE 09 00 00 00 B8 3E 00 00 00 0F 05 90
BF 2B 02 00 00 B8 3C 00 00 00 0F 05 90 48 83 C4
78 5B 5D 59 5a 5e 5f

```

32 位的 C 代码和 shellcode 如下，编译的时候要关闭位置无关代码，否则 gcc 在 32 位下会引入 __x86.get_pc_thunk.ax 函数，从而用到了 GOT，导致生成的 shellcode 不能移植

```

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/conf.h>
#include <sys/user.h>
#include <sys/param.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/user.h>
#include <sys/param.h>

void d(){
    pid_t child;
    int status;

    struct user_regs_struct regs;

    asm(
        "movl $2, %%eax;" //fork
        "int $0x80;" //execve
        "movl %%eax,%0": "=r"(child)
    );
}

```

```

);
if(child == 0) {
asm(
    "movl $26, %eax;" // ptrace
    "xorl %ecx, %ecx;" //2
    "xorl %edx, %edx;" //3
    "xorl %esi, %esi;" //4
    "xorl %ebx, %ebx;" //1
    "int $0x80;" //ptrace(PTRACE_TRACEME)
    "movl $20, %eax;" // getpid
    "int $0x80;"
    "movl %eax, %ebx;" // 0 1
    "movl $19, %ecx;" // 2
    "movl $37, %eax;" // 0 kill
    "int $0x80;"
);
//kill(getpid(), 19);
goto end;
} else {
asm(
    "movl $114, %eax;" // 0 wait4
    "xorl %ebx, %ebx;" // 1
    "int $0x80;"
);
//wait(NULL);
while(1){
asm(
    "movl %0, %%ecx"::"r"(child):"%ecx" // 0 2
);
asm(
    "movl $26, %eax;" // 0 ptrace
    "movl $24, %ebx;" // 1
    "xorl %edx, %edx;" // 3
    "xorl %esi, %esi;" // 4
    "int $0x80;"
);           //ptrace(PTRACE_SYSCALL, child, NULL, NULL);
void *p = (void *)&status;
asm(
    "movl %0, %%ebx"::"r"(child):"%ebx" // 0 1
);
asm(
    "movl %0, %%ecx"::"r"(p):"%ecx" // 0 2 ...
);
asm(
    "xorl %edx, %edx;" // 3
    "movl $114, %eax;" // 0 0 wait4
    "int $0x80;" //waitpid(pid)
);
void *regs_addr = (void *)&regs;
asm(
    "movl %0, %%esi"::"r"(regs_addr):"%esi" // 0 4
);
asm(
    "movl %0, %%ecx"::"r"(child):"%ecx" // 2
);
}
}

```

```

asm(
    "movl $12, %ebx;" // 0 1
    "xorl %edx, %edx;" // 3
    "movl $26, %eax;" // 0 0 ptrace
    "int $0x80;"      //ptrace(PTRACE_GETREGS)
);
if((regs.orig_eax == 2 || regs.orig_eax == 11 ||
    regs.orig_eax == 190 || regs.orig_eax == 120 ||
    regs.orig_eax == 252)){
    asm(
        "movl %0, %%ebx":r"(child):%ebx" // 0 1
    );
    asm(
        "movl $9, %ecx;" // 0 2
        "movl $37, %eax;" // 0 0 kill
        "int $0x80;" //kill(child, 9);
        break;
    }
}
asm(
    "movl $555, %ebx;" // 0 1
    "movl $1, %eax;" // 0 0 exit
    "int $0x80;" //exit(1);
);
}

end:
return;
}

int main(){
d();
execl("/bin/ls", ".");
//system("ls");
return 0;
}

```

shellcode (加入了 pusha 和 popa) :

```

60 55 89 E5 56 53 83 EC 60 B8 02 00 00 00 00 CD 80
89 C0 89 45 F4 83 7D F4 00 75 29 B8 1A 00 00 00
31 C9 31 D2 31 F6 31 DB CD 80 B8 14 00 00 00 CD
80 89 C3 B9 13 00 00 00 B8 25 00 00 00 CD 80 E9
9F 00 00 00 B8 72 00 00 00 31 DB CD 80 8B 45 F4
89 C1 B8 1A 00 00 00 BB 18 00 00 00 31 D2 31 F6
CD 80 8D 45 E8 89 45 F0 8B 45 F4 89 C3 8B 45 F0
89 C1 31 D2 B8 72 00 00 00 CD 80 8D 45 A4 89 45
EC 8B 45 EC 89 C6 8B 45 F4 89 C1 BB 0C 00 00 00
31 D2 B8 1A 00 00 00 CD 80 8B 45 D0 83 F8 02 74
24 8B 45 D0 83 F8 0B 74 1C 8B 45 D0 3D BE 00 00
00 74 12 8B 45 D0 83 F8 78 74 0A 8B 45 D0 3D FC
00 00 00 75 88 8B 45 F4 89 C3 B9 09 00 00 00 B8

```

```
25 00 00 00 CD 80 90 BB 2B 02 00 00 B8 01 00 00
00 CD 80 90 83 C4 60 5B 5E 5D 61
```

那么接下来我们实践看一下，编写程序如下

```
#include<stdio.h>

int main(){
    system("ls");
}
```

执行效果如下

```
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./system
add.c b.txt      fgets  fmtest.c fork.c   main.c   malloc32       overflow     p
add.s clean      fgets.c for     fpr      main.i   malloc32.c    overflow.c   p
a.out c.txt      fgj.py   for.c    gdblink  main.s   malloc.c    overflow.py  r
a.txt dshiled.txt fmtest   fork     gdblink.c Makefile  opcodepatch.py overflow.s  r
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ chmod +x system
```

然后开始植入代码到 eh_frame

篡改程序入口到 eh_frame

```

        db 2          ; File class: 64-bit
        db 1          ; Data encoding: little-endian
        db 1          ; File version
        db 0          ; OS/ABI: UNIX System V ABI
        db 0          ; ABI Version
        db 7 dup(0)   ; Padding
        dw 3          ; File type: Shared object
        dw 3Eh        ; Machine: x86-64
        dd 1          ; File version
dq offset unk_2050 ; Entry point
        dq 40h        ; PHT file offset
        dq 3978h      ; SHT file offset
        dd 0          ; Processor-specific flags
        dw 40h        ; ELF header size
        dw 38h        ; PHT entry size
        dw 0Dh        ; Number of entries in PHT
        dw 40h        ; SHT entry size
        dw 1Fh        ; Number of entries in SHT
        dw 1Eh        ; SHT entry index for string table

```

修改 eh_frame 的权限为 7

```

; PHT Entry 4
        dd 1          ; Type: LOAD
        dd 7          ; Flags
dq 2000h          ; File offset
dq offset _IO_stdin_used ; Virtual address
dq 2000h          ; Physical address
dq 158h          ; Size in file image
dq 158h          ; Size in memory image
dq 1000h         ; Alignment

```

计算 jmp 到 start 上去的偏移，lea 的指令长度为 7，我们 shellcode 结束的地方是 0x213c，所以 lea 的地址是 0x213d，所以这个时候的 rip 是 0x2144，计算 start 函数到这里的偏移（如果没有开启 PIE 的话就直接 jmp 绝对地址）

$$2144 - 1060 =$$

10E4

生成 shellcode

```

from pwn import *

context.arch = 'amd64'

code = '''
    lea r15, [rip-0x10e4]
    jmp r15
'''

for i in list(asm(code, arch='amd64')):

```

```
print(hex(i)[2:]).rjust(2, '0'), end = ' ')
print()
```

```
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ python3 opcodepatch.py
4c 8d 3d 1c ef ff ff 41 ff e7
```

植入在之前通防壳的后面

现在我们再去执行一下 system 看看

```
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ chmod +x system
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./system
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./system
yeuoly@yeuoly-virtual-machine:~/Desktop/test$ ./system
```

另外说一句，对于 32 位的程序，由于 pwntools 在生成代码的时候很蛋疼，对重定位的检测很离谱，所以我们最好直接用 opcode 来写，jmp 指令的 opcode 为 E9，操作数为 4 字节，指令长度为 5 字节，4 字节的操作数即为相对偏移，支持负数，所以我在 32 位程序上就是直接写 jmp 指令，当然，64 位下也是一样的，区别不大，大概改成下面这样即可

[上一页](#)

[下一页](#)

← [3. 【WEB】AWD 技巧](#) [AI | 人工智能](#) →

评论

0 个表情



0 条评论