Linux exploit development tutorial

Sn0rt@abc.shop.edu.cn

2016 年 10 月 10 日

前　　言

发现 Linux 下二进制学习曲线陡峭, 而套路零散, 于是整理编著这篇文章, 来帮助感兴趣的人学习, 还想结识更多对 Linux 二进制感兴趣的人. 万事开头难, 首先要感谢本文原来的的作者 sploitfun, 他开始做了这件事并写出了思路, 我在他的基础上进行了补充和翻译.

还要要感谢 phrack, 乌云知识库, 各种 wiki 上面文章的作者, 这些作者和安全研究人员讲解了很多关于 exploit 相关技术, 是大家的无私分享使很多东西变的可能, 我也想把这样的分享精神学习来.

为了防止文章过于臃肿, 我们讲分享讨论的话题尽量限制在 Linux,x86,MIPS,ipv4 范围内, 我们假设读者能正常使用 Linux, 熟悉 C 语言, 了解汇编语言, 认识计算机专业词汇, 基本体系结构知识 (栈, 堆, 内存之类的). 如果不能因为知识储备不够, 推荐 0day 安全 [2], 不建议特为了某个事情把所有预先条件都修好, 需要什么要用到在去学, 因为不用都会忘的. 我认为技术人员的学习能力比现实技术重要.

如果关于本文有什么疑问可以联系我.

目录

0x01 预备

在这个 level 我将要花点时间给大家介绍基本的漏洞类型和安全机制, 然后关闭全部的安全保护机制, 学习如何在 Linux 下面编写最基本的 exp.

1.1 安全机制

分为两大类: 编译相关 (elf 加固), 部分编译选项控制着生成更安全的代码 (损失部分性能或者空间), 还有就说运行时的安全, 都是为增加了漏洞利用的难度, 不能从本质上去除软件的漏洞.

1.1.1 STACK CANARY

Canary 是放置在缓冲区和控制数据之间的一个 words 被用来检测缓冲区溢出, 如果发生缓冲区溢出那么第一个被修改的数据通常是 canary, 当其验证失败通常说明发生了栈溢出, 更多信息参考这里 [1].

```
1   gcc -fstack-protector
```

[1] https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries

1.1.2 NX

在早期, 指令是数据, 数据也是数据, 当 PC 指向哪里, 那里的数据就会被当成指令被 cpu 执行, 后来 NX 标志位被引入来区分指令和数据. 更多信息参考这里 [1] [2] [3].

```
1    gcc -z execstack
```

1.1.3 FORTIFY

在编译和运行时候保护 glibc:

- expand unbounded calls to "sprintf", "strcpy" into their "n" length-limited cousins when the size of a destination buffer is known (protects against memory overflows).
- stop format string "%n" attacks when the format string is in a writable % memory segment.
- require checking various important function return codes and arguments (e.g.system, write, open).
- require explicit file mask when creating new files.

```
1    gcc -D_FORTIFY_SOURCE=2 -O
```

1.1.4 PIE

-fPIC: 类似于-fpic 不过克服了部分平台对偏移表尺寸的限制. 生成可用于共享库的位置独立代码。所有的内部寻址均通过全局偏移表 (GOT) 完成. 要确定一个地址, 需要将代码自身的内存位置作为表中一项插入. 该选项需要操作系统支持, 因此并不是在所有系统上均有效. 该选项产生可以在共享库中存放并从中

[2] «Intel® 64 and IA-32 Architectures Software Developer′ s Manual» volumes 3 section 4.6

[3] https://en.wikipedia.org/wiki/NX_bit

加载的目标模块. 参考链接 [4].

-fPIE: 这选项类似于-fpic 与-fPIC, 但生成的位置无关代码只可以链接为可执行文件, 它通常的链接选项是-pie.

```
1    gcc -pie -fPIE
```

1.1.5 RELRO

Hardens ELF programs against loader memory area overwrites by having the loader mark any areas of the relocation table as read-only for any symbols resolved at load-time ("read-only relocations"). This reduces the area of possible GOT-overwrite-style memory corruption attacks [5].

ASLR

[6]

[4] https://en.wikipedia.org/wiki/Position-independent_code#PIE

[5] http://blog.isis.poly.edu/exploitation%20mitigation%20techniques/exploitation%20techniques/2011/06/02/relro-relocation-read-only/

[6] https://en.wikipedia.org/wiki/Address_space_layout_randomization

1.2 漏洞类型

1.2.1 栈溢出

1.2.2 整数溢出

1.2.3 off-by-one(stack base)

1.2.4 格式化字符串

%h(短写) %n$d(直接参数访问) %n(任意内存写) %s(任意内存读)

1.3 Exp 开发

1.3.1 rop

nop seld + shellcode + ret

1.3.2 .dtors(废弃)

```
1   static void cleanup() __attribute__((destructor))
```

1.3.3 覆写 GOT

0x02 stack

这个阶段可能要花点时间了, 需要学习主流的 bypass 安全机制的部分手段 (base stack).

ret2any: 返回到任何可以执行的地方, 已知的地方

- stack
- data/heap
- text
- library (libc)
- code chunk (ROP)

2.1 CANARY

2.1.1 overwriting TLS

2.2 NX

2.2.1 return-to-libc

[1].

2.2.2 chained return-to-libc

[2].

2.3 ASLR

2.3.1 return-to-plt

[3].

[1] https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-return-to-libc/

[2] https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-chained-return-to-libc/

[3] https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-i/

2.3.2 brute-force

[4].

2.3.3 overwriting GOT

[5].

[4] https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-ii/

[5] https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-iii/

0x03 内核

这个阶段归档了 kernel 安全相关的文档 (安全保护, 利用).

3.1 安全机制

早期 kernel 可以随意访问用户态代码,ret2usr 技术可以让内核执行用户态的代码, 不过随着 Linux 的发展 SMAP(禁止 kernel 随意访问用户态,RFLAGE.AC 标志位置位可以),SMEP 禁止 kernel 态直接执行用户态代码.

3.1.1 SMAP

现代 Linux 默认启用.

3.1.2 SMEP/PXN

现代 Linux 默认启用.

3.1.3 kaslr

ubuntu 14.04 desktop 默认还没有启用, 更多信息参考 Ubuntu security Features [1]

3.2 利用方法

3.2.1 rop-2-usr(废弃)

早期能工作

3.2.2 rop

3.2.3 vDSO overwriting

SEMP using vDSO overwrites(CSAW Fianl 2015 string IPC)

[1] https://wiki.ubuntu.com/Security/Features#Userspace_Hardening

0x04 漏洞挖掘

漏洞挖掘的重要性不言而喻, 打个比喻上面写的如何啃肉, 漏洞挖掘就是肉在哪里.

4.1 fuzz

4.1.1 why fuzz?

- 容易实现 – 覆盖面广 – 低投入高产出

4.1.2 why not fuzz?

– 分析困难 (无法调试) – Panic 多 / Exploitable 少 – 欠缺精度

4.1.3 where to fuzz?

– ioctl – sysctl – File system – Network

4.1.4 how to fuzz?

4.2 代码审计

4.2.1 source

- - Heap Overflow
- - Integer Overflow
- - Type Confusion
- - Use after Free
- - Logical Error
- - Kernel Information Leak

参考文献

[1] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 5 2012.

[2] 王清. *0day* 安全*:* 软件漏洞分析技术. 电子工业出版社, 2 edition, 2011.