LAB3：缓冲区溢出攻击

实验提交（请认真阅读以下内容）

截止时间: 见本科教学平台上的相应作业发布信息。

请你在实验截止前务必确认你提交的内容符合要求(格式、相关内容等)，建议你在提交后下载你提交的内容进行确认。如果由于你的提交格式错误造成评分程序扣分，责任自负。

实验介绍

本实验的目的在于加深对IA-32函数调用规则和栈结构的具体理解。实验的主要内容是对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，例如将给定的字节序列插入到其本不应出现的内存位置。

实验中你需要对目标可执行程序bufbomb分别完成5个难度递增的缓冲区溢出攻击。5个难度级分别命名为smoke (level 0)、fizz (level 1)、bang (level 2)、boom (level 3)和kaboom (level 4)，其中smoke级最简单而kaboom级最困难。

实验语言：C；实验环境：Linux i386

实验数据

在本实验中，每位同学会得到一个包含以下二进制可执行程序的TAR文件，可从(http://114.212.86.190/course/ics17a/getbufbomb.htm)下载， 下载该文件到本地目录中并使用“tar xvf xxxxxxxx.tar”命令将其中包含的文件提取出来：

- bufbomb：实验需要攻击的目标buffer bomb程序。
- makecookie：该程序基于命令行参数（例如你的学号）产生一个唯一的由8个16进制数字组成的字节序列（例如0x1005b2b7），称为“cookie”，用作实验中需要置入栈中的数据之一。
- hex2raw：字符串格式转换程序。

目标程序**bufbomb**说明

bufbomb程序接受下列命令行参数：

- -u userid：以给定的用户ID“userid”（例如学号）运行程序。你应该在运行程序时总指定该参数，因为bufbomb程序将基于userid决定你使用的cookie值（同makecookie程序的输出），并且在bufbomb程序内部，你需要使用的一些关键的栈地址取决于你的userid所对应的cookie值。
- -h：打印可用命令行参数列表.
- -n：以“Nitro”模式运行，用于下述Level 4的实验阶段。

bufbomb目标程序在运行时使用如下getbuf函数从标准输入读入一个字符串:

```
/* Buffer size for getbuf */
int getbuf()
{
    other variables ...;
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

其中，函数Gets类似于标准库函数gets，它从标准输入读入一个字符串（以换行‘\n’或文件结束end-of-file字符结尾)，并将字符串（以null空字符结尾）存入指定的目标内存位置。在getbuf函数代码中，目标内存位置是具有NORMAL_BUFFER_SIZE字符存储空间的数组buf，而NORMAL_BUFFER_SIZE是大于等于32的一个常数。 但是，函数Gets()并不判断buf数组是否足够大而只是简单地向目标地址复制全部输入字符串，因此有可能超出预先分配的存储空间边界，即缓冲区溢出。 如果用户输入给getbuf()的字符串不超过(NORMAL_BUFFER_SIZE-1)个字符长度的话，很明显getbuf()将正常返回1，如下列运行示例所示：

```
unix> ./bufbomb -u 123456789
Type string: I love ICS.
Dud: getbuf returned 0x1
```

如果输入一个更长的字符串，通常会发生类似下列的错误：

```
unix> ./bufbomb -u 123456789
Type string: It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!
```

正如上面的错误信息所指，缓冲区溢出通常导致程序状态被破坏，产生存储器访问错误。（想想：为什么会产生一个段错误？x86的栈结构是怎么组成的？） 本实验的任务就是精心设计输入给bufbomb的字符串，（通过缓冲区溢出）使其完成一些有趣的事情，这样的字符串称为“exploit string”（攻击字符串），关键是你应该以栈中的哪些数据条目做为攻击目标。

假设将你的攻击字符串包含于一文件Solution.txt中，则可使用如下命令测试攻击字符串在bufbomb上的运行结果，并与相应难度级的期望输出对比，以验证通过与否。

```
unix> cat solution.txt | ./hex2raw | ./bufbomb -u [userid]
```

辅助程序hex2raw说明

由于攻击字符串（exploit string）可能需要包含不属于ASCII可打印字符集合的字节取值（因而无法直接编辑输入），为此程序hex2raw可帮助你构造这样的字符串，它从标准输入接收一个十六进制格式的字符串（即每一目标字节用两个十六进制数字表示其值，不同字节之间用空格或换行等空白字符分隔），再将每个目标字节转为二进制数值表示的单字节逐一送往标准输出。 建议用换行分隔攻击字符串的不同部分，这并不会影响字符串的解释。 hex2raw程序还支持C语言风格的块注释以便为攻击字符串添加注释（如下例），这同样不影响字符串的解释与使用。

```
bf 66 7b 32 78 /* mov $0x78327b66,%edi */
```

注意务必要在开始与结束注释字符串（“/*”和“*/”）前后保留空白字符以便注释部分被程序正确忽略。

辅助程序makecookie说明

如前所述，本实验各阶段的正确解答基于进行实验的学生userid生成的cookie值。 一个cookie是一个由8个16进制数字组成的字节序列（例如0x1005b2b7），对每一个userid是唯一的。你可以如下使用makecookie程序生成你的cookie，其中将你的userid作为makecookie程序的唯一参数。

```
unix> ./makecookie 133333333
0x1005b2b7
```

0x1005b2b7即为学号为123456789学生的cookie值。

实验结果提交

做为实验结果，你需要提交最多5个solution文件（一起打包为“学号.tar”文件提交），分别包含完成5种级别攻击的攻击字符串（exploit string），文件命名方式为“学号-级别.txt”（例如“123456789-smoke.txt”），其中级别务必采取全部小写的形式（例如“smoke”、“fizz”、“bang”等），以便测试评分程序处理。 每个文件中应包含一段代码字符串序列，序列格式为：两个16进制值作为一个16进制对，每个16进制对代表一个字节，每个16进制对之间用空格分开，例如“68 ef cd ab 00 83 c0 11 98 ba dc fe”。 如前所述，程序hex2raw可用于将代码字符串转化为字节序列，供输入bufbomb程序中执行（使用一系列管道操作符），用法示例如下：

```
unix> cat [userid]-[level].txt | ./hex2raw | ./bufbomb -u [userid]
```

除上述方式以外，你还可以将攻击字符串对应的raw字节序列存于一个文件中，并使用I/O重定向将其输入给bufbomb：

```
unix> ./hex2raw < [userid]-[level].txt > [userid]-[level]-raw.txt
unix> ./bufbomb -u [userid] < [userid]-[level]-raw.txt
```

该方法也可用于在GDB中运行bufbomb的情况：

```
unix> gdb bufbomb
(gdb) run -u 123456789 < exploit-raw.txt
```

重要提示：

- 攻击字符串绝不能在任何中间位置包含值为0x0A的字节（因为该ASCII代码对应换行符‘\n’），因为当Gets函数遇到该字节时将认为你意图结束字符串。
- 由于hex2raw期望字节由两个十六进制格式的数字表示，因此如果你想构造一个值为0的字节，你需要指定00。

当你成功完成了某一级别的攻击，例如Level 0（smoke），程序将输出类似如下的信息：

```
./hex2raw < 123456789-smoke.txt | ./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x1005b2b7
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

实验内容

下面针对5种不同级别的攻击，分别说明实验需要达到的目标。

Level 0: smoke

在bufbomb程序中，函数getbuf被一个test函数调用，代码如下：

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
            printf("Boom!: getbuf returned 0x%x\n", val);
            validate(3);
        } else {
            printf("Dud: getbuf returned 0x%x\n", val);
        }
}
```

在getbuf执行完其返回语句（getbuf函数第5行），程序正常情况下应该从test函数的第7行开始继续执行。现在我们要改变该行为。在bufbomb程序中有一个对应如下C代码的函数smoke：

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

本实验级别的任务是当getbuf函数执行它的return语句后，使bufbomb程序执行smoke函数的代码，而不是返回到test函数继续执行。（注意：你的攻击字符串可能会同时破坏了与本阶段无关的栈结构部分，但这不会造成问题，因为smoke函数会使程序直接结束。）

一些建议

1. 在本级别中，你用来推断攻击字符串的所有信息都可从检查bufbomb的反汇编代码中获得（使用objdump –d命令）。
2. 注意字符串和代码中的字节顺序。
3. 可使用GDB工具单步跟踪getbuf函数的最后几条指令，以了解程序的运行情况。

Level 1: fizz

在bufbomb程序中有一个fizz函数，其代码如下：

```
void fizz(int val)
{
    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
        validate(1);
    } else
        printf("Misfire: You called fizz(0x%x)\n", val);
```

```
    exit(0);
}
```

与Level 0类似，本实验级别的任务是让bufbomb程序在其中的getbuf函数执行return语句后转而执行fizz函数的代码，而不是返回到test函数。不过，与Level 0的smoke函数不同，fizz函数需要一个输入参数，本级别要求设法将使用makecookie得到的cookie值作为参数传递给fizz函数。

建议

- 程序不会真的调用fizz——程序只是执行fizz函数的语句代码。因此需要仔细考虑将cookie放置在栈中什么位置。

Level 2: bang

更复杂的缓冲区攻击将在攻击字符串中包含实际的机器指令，进而攻击字符串将原返回地址指针改写为位于栈上的攻击机器指令的开始地址。这样，当调用函数（这里是getbuf）执行ret指令时，程序将开始执行攻击代码而不是返回上层函数。 使用这种攻击方式，你可以使被攻击程序做任何事。通过攻击字符串放置到栈上的代码称为攻击代码（exploit code）。 然而，此类攻击具有一定难度，因为你必须设法将（攻击）机器代码置入栈中且将返回地址指针指向代码起始位置。

在bufbomb程序中，有一个bang函数，代码如下：

```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

与Level 0和Level 1类似，本实验级别的任务是让bufbomb执行bang函数中的代码而不是返回到test函数继续执行。具体来讲，你的攻击代码应首先将全局变量global_value设置为对应你userid的cookie值，再将bang函数的地址压入栈中，然后执行一条ret指令从而跳至bang函数的代码继续执行。

一些建议

1. 可以使用GDB获得构造攻击字符串所需的信息。例如，在getbuf函数里设置一个断点并执行到该断点处，进而确定global_value和缓冲区等变量的地址。
2. 手工进行指令的字节编码枯燥且容易出错。相反，你可以使用一些工具来完成该工作——首先编写一个汇编代码文件包含你希望置于栈上的指令和数据，然后，使用“gcc –m32 –c”命令将该文件汇编成机器代码，再使用“objdump –d”命令将其反汇编，从中你可获得与在命令行上输入的一样的字节序列（见后面的简要示例）。
3. 不要试图利用jmp或者call指令跳到bang函数的代码中，这些指令使用相对PC的寻址，很难正确达到前述目标。相反，你应向栈中压入地址并使用ret指令。

Level 3: boom

本实验的前几个级别实现的攻击都是使得程序跳转到不同于正常返回地址的其他函数中，进而结束整个程序的运行。因此，使用攻击字符串破坏、改写栈中原有记录值的方式是可

接受的。然而，更高明的缓冲区溢出攻击除执行攻击代码来改变程序的寄存器或内存中的值外，仍然使得程序能够返回到原来的调用函数（例如test）继续执行——即调用函数感觉不到攻击行为。然而，这种攻击方式的难度相对更高，因为攻击者必须：1)将攻击机器代码置入栈中，2）设置return指针指向该代码的起始地址，3）还原（清除）对栈状态的任何破坏。

本实验级别的任务是构造一个攻击字符串，使得getbuf函数将cookie值返回给test函数，而不是返回值1。除此之外，你的攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行ret指令从而真正返回到test函数。

一些建议

- 同上一级别。例如，可使用GDB确定保存的返回地址等参数。

Level 4: kaboom

首先注意：你需要使用“-n”命令行开关运行bufbomb程序（”Notro”模式），以便进行本级别实验。

通常，一个给定函数的栈的确切内存地址随程序运行实例（特别是运行用户）的不同而不同。其中一个原因是当程序开始执行时，所有环境变量的值所在内存位置靠近栈的基地址，而环境变量的值是做为字符串存储的，视值的不同需要不同数量的存储空间。因此，为一特定运行用户分配的栈空间取决于其环境变量的设置。此外，当在GDB中运行程序时，程序的栈地址也会存在差异，因为GDB使用栈空间保存其自己的状态。之前实验中，在bufbomb调用getbuf的代码中通过一定措施获得了稳定的栈地址，因此不同运行实例中，getbuf函数的栈帧地址保持不变。这使得你在之前实验中能够基于buf的已知确切起始地址构造攻击字符串。但是，如果你尝试将这样的攻击用于一般的程序，你会发现你的攻击有时奏效，有时却导致段错误（segmentation fault）。

然而，本实验级别（”Notro”）采取相反的栈策略，即栈帧的地址相比一般情况将更不固定————程序在调用testn函数前会在栈上分配一随机大小的内存块，因此testn函数及其所调用的getbufn函数的栈帧起始地址在每次运行程序时呈现一个随机的、不再固定的值。
另一方面，在该模式下程序调用的getbufn函数区别于getbuf函数之处在于——getbufn函数使用如下不小于512字节的缓冲区，以方便你利用更大的存储空间构造可靠的攻击代码：

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE 一个大于等于512的整数常量
```

本级别实验的任务与前一级别相同，即构造一攻击字符串使得getbufn函数返回cookie值至testn函数，而不是返回值1。 具体来说，你的攻击字符串应将函数返回值设为cookie值，复原/清除所有被破坏的状态，将正确的返回位置压入栈中，并执行ret指令以返回testn函数。然而，在Nitro模式下运行时，bufbomb使用你的攻击字符串执行5次getbufn函数，每次采用不同的栈偏移位置。你的攻击字符串必须使程序每次均能返回cookie值。

一些建议

1. 本实验的技巧在于合理使用nop指令，该指令的机器代码只有一个字节（0x90）。
2. 你可以如下使用hex2raw程序生成并传送攻击字符串的多个拷贝给bufbomb程序（假设123456789-kaboom.txt文件中保存了攻击字符串的一个拷贝）：

```
unix> cat 123456789-kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 123456789
```

实验工具

生成字节代码

使用GCC做为汇编器并用OBJDUMP做为反汇编器，这将方便生成指令序列的字节编码表示。 例如，编写一个example.S文件包含如下汇编代码：

```
# Example of hand-generated assembly code
push $0xabcdef          # Push value onto stack
add $17,%eax            # Add 17 to %eax
.align 4                # Following will be aligned on multiple of 4
.long 0xfedcba98        # A 4-byte constant
```

我们现在可以如下汇编和反汇编该文件：

```
unix> gcc -m32 -c example.S
unix> objdump -d example.o > example.d
```

生成的example.d文件包含如下代码行：

```
0: 68 ef cd ab 00               push $0xabcdef
5: 83 c0 11                     add $0x11,%eax
8: 98                           cwtl
9: ba                           .byte 0xba
a: dc fe                        fdivr %st,%st(6)
```

其中，每行显示一个单独的指令。左边的数字表示指令的起始地址（从0开始），”:”之后的16进制数字给出指令的字节编码。例如，指令”push $0xABCDEF“对应的16进制字节编码为”68 ef cd ab 00“。 然而，注意从地址“8”开始，反汇编器错误地将本来对应数据的字节解释成了指令（cwtl）。实际上，从该地址起的4个字节“98 ba dc fe”对应于前述example.S文件中最后的数据0xFEDCBA98的小端字节表示。

如上确定了字节序列“68 ef cd ab 00 83 c0 11 98 ba dc fe”对应的机器指令组成后：我们可以把该十六进制格式字符串输入hex2raw函数以产生一个准备输入到bufbomb程序的攻击字符串。或者，回忆hex2raw程序支持在输入字符串中包含C语言块注释（以便使用者更好地理解其中字符串对应的指令），因此我们可以编辑修改example.d文件为如下形式（将指令说明变为注释）：

```
68 ef cd ab 00 /* push $0xabcdef */
83 c0 11 /* add $0x11,%eax */
98 ba dc fe
```

从而可将该文件做为hex2raw程序的合法输入：

```
unix> cat example.d | ./hex2raw | ./bufbomb -u [userid]
```