缓冲区溢出攻击实验

（苏丰，南京大学，修改自 CMU:CSAPP2/3 实验）

一、实验介绍

本实验的目的在于加深对 IA-32 过程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程序的运行内存映像（例如将专门设计的字节序列插入到栈中特定内存位置）和行为，以实现实验预定的目标。

实验中需要针对目标可执行程序 bufbomb，分别完成多个难度递增的缓冲区溢出攻击（完成的顺序没有固定要求）。按从易到难的顺序，这些难度级分别命名为 smoke (level 0)、fizz (level 1)、bang (level 2)、rumble (level 3)、boom (level 4)和 kaboom (level 5)。

- 实验环境：Linux i386
- 实验语言：汇编

二、实验数据

在本实验中，每位学生可从 Lab 实验服务器下载包含本实验相关文件的一个 tar 文件。可在 Linux 实验环境中使用命令“**tar xvf <tar 文件名>**”将其中包含的文件提取到当前目录中。该 tar 文件中包含如下实验所需文件：

- 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给出的 ID，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”，用作实验中可能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”（**本实验中应设为学号**）运行程序。在每次运行程序时均应指定该参数，因为 bufbomb 程序将基于该 ID 决定应该使用的 **cookie 值**（与 makecookie 程序的输出相同），而 bufbomb 程序运行时的一些关键栈地址取决于该 cookie 值。
- -h：打印可用命令行参数列表
- -n：以“Nitro”模式运行，用于 kaboom 实验阶段

bufbomb 目标程序在运行时使用如下 getbuf 过程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，过程 Gets 类似于标准库过程 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 过程代码中，目标内存位置是具有 NORMAL_BUFFER_SIZE 个字节存储空间的数组 buf，而 NORMAL_BUFFER_SIZE 是大于等于 32 的一个常数。

注意，过程 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：

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

但是，如果输入一个更长的字符串，则可能会发生类似下列的错误：

```
Linux>./bufbomb -u 123456789
Type string: It is easier to love this class when you are a TA. For a student, the class should be interesting too, though some hard work may be needed. Anyway, trying it remains a happy thing. Right?
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。(思考：为什么会产生一个段错误？Linux x86 的栈结构组成是什么样的？)

本实验的任务就是精心设计输入给 bufbomb 的字符串，通过造成缓冲区溢出达成预定的实验目标，这样的字符串称为"exploit string"（攻击字符串），实验的关键是确定栈中哪些数据条目做为攻击的目标。

工具程序 hex2raw 说明

由于攻击字符串（exploit string）可能包含不属于 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，实验提供了工具程序 hex2raw 帮助构造这样的字符串。该程序从标准输入接收一个采用十六进制格式编码的字符串(其中使用两个十六进制数字对攻击字符串中每一字节的值进行编码表示，不同目标字节的编码之间用空格或换行等空白字符分隔)，进一步将输入的每对编码数字转为二进制数表示的单个目标字节并逐一送往标准输出。

注意，为方便理解攻击字符串的组成和内容，可以用换行分隔攻击字符串的编码表示中的不同部分，这并不会影响字符串的解释和转换。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。

另外，注意：

- 攻击字符串中不能包含值为 0x0A 的字节，因为该字符对应换行符'\n'，当 Gets 过程遇到该字符时将认为该位置为字符串的结束，从而忽略其后的字符串内容
- 由于 hex2raw 期望字节由两个十六进制格式的数字表示，因此如果想构造一个值为 0 的字节，应指定 00

进一步，可将上述十六进制数字对序列形式的攻击字符串（例如"68 ef cd ab 00 83 c0 11 98 ba dc fe"）保存于一文本文件中，用于测试等（见后面说明）。

工具程序 makecookie 说明

如前所述，本实验部分阶段的正确解答基于从 bufbomb 命令行选项 userid（**本实验中应设为学号**）计算生成的 cookie 值。一个 cookie 是由 8 个 16 进制数字组成的一个字节序列（例如 0x1005b2b7），对每一个 userid 是唯一的。可以如下使用 makecookie 程序生成对应特定 userid 的 cookie，即将 userid 作为 makecookie 程序的唯一参数。

```
linux>./makecookie 123456789
0x25e1304b
```

0x25e1304b 即为学号 123456789 对应的 cookie 值。

三、测试攻击字符串

可将攻击字符串保存在一文件 solution.txt 中，使用如下命令（**将参数[userid]替换为学号**）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，以验证相应实验阶段通过与否。

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

上述命令使用一系列管道操作符将程序 hex2raw 从编码字符串转换得到的目标攻击字节序列输入 bufbomb 程序中进行测试。

除上述方式以外，还可以如下将攻击字符串的二进制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：

```
linux>./hex2raw < solution.txt > solution-raw.txt
linux>./bufbomb -u [userid] < solution-raw.txt
```

该方法也可用于在 GDB 中运行 bufbomb 的情况：

```
linux>gdb bufbomb
(gdb) run -u [userid] < solution-raw.txt
```

当设计的攻击字符串成功完成了预定的缓冲区溢出攻击目标，例如实验 Level 0（smoke），程序将输出类似如下的信息，提示攻击字符串（此例中保存于文件 smoke.txt 中）设计正确：

```
./hex2raw < smoke.txt | ./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

四、实验结果提交

1）为每个实验级别编写一个文本格式的 solution 文件，包含对应实验级别的攻击字符串（exploit string，表示为如下所述编码数字对序列的形式）。注意文件命名方式必须为“级别.txt”（例如“smoke.txt”），其中级别名应统一采取**小写**的形式（例如“smoke”、“fizz”、“bang”等），以方便评分程序处理。

注意：

- ✓ 每个文件中应包含对应攻击字符串的一个编码数字对序列，序列格式为：两个 16 进制数作为一个 16 进制数值对，对应攻击字符串中的一个字节的值。各 16 进制数值对之间用空格分隔，例如“68 ef cd ab 00 83 c0 11 98 ba dc fe”。

- ✓ 攻击字符串中不能包含值为 0x0A 的字节，因为该字符对应换行符'\n'，当 Gets 函数遇到该字符时将认为该位置为字符串的结束，从而忽略其后的字符串内容。
- ✓ 由于 hex2raw 期望每个字节由两个十六进制数字表示，因此如果想构造一个值为 0 的字节，应写为 00。

2）把所有 solution 文件打包为"学号.tar"文件提交（其中务必不能包含任何目录结构）。

五、实验内容

下面针对不同级别的攻击，分别说明实验需要达到的目标。

Level 0: smoke

在 bufbomb 程序中，过程 getbuf 被一个 test 过程调用，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 过程执行完其返回语句后，程序正常情况下应该从上列 test 过程中第 8 行的 if 语句继续执行，现在应设法改变该行为。在 bufbomb 程序中有一个对应如下 C 代码的过程 smoke：

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

本实验级别的任务是当 getbuf 过程执行它的 return 语句后，使 bufbomb 程序执行 smoke 过程的代码，而不是返回到 test 过程继续执行。（注意：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，但在本级别中这没有问题，因为 smoke 过程会使程序直接结束）

建议：

- ✓ 在本级别中，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）
- ✓ 注意字符串和代码中的字节顺序
- ✓ 可使用 GDB 工具单步跟踪 getbuf 过程的最后几条指令，以了解程序的运行情况

Level 1: fizz

在 bufbomb 程序中有一个 fizz 过程，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 过程执行 return 语句后转而执行 fizz 过程的代码，而不是返回到 test 过程。不过，与 Level 0 的 smoke 过程不同，fizz 过程需要一个输入参数，如上列代码所示，本级别要求设法使该参数的值等于使用 makecookie 得到的 cookie 值。

建议：

- ✓ 程序无需且不会真的调用 fizz——程序只是执行 fizz 过程的语句代码，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，并通过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，当调用过程（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过程。

使用这种攻击方式可以使被攻击程序执行任何操作。随攻击字符串被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，因为必须设法将攻击机器代码置入栈中，并且将返回地址指向攻击代码的起始位置。

在 bufbomb 程序中，有一个 bang 过程，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 过程中的代码而不是返回到 test 过程继续执行。具体来讲，攻击代码应首先将全局变量 global_value 设置为对应 userid（即**学号**）的 cookie 值，再将 bang 过程的地址压入栈中，然后**执行一条 ret 指令从而跳至 bang 过程的代码继续执行**。

建议：

- ✓ 可以使用 GDB 获得构造攻击字符串所需的信息。例如，在 getbuf 过程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
- ✓ 手工进行指令的字节编码枯燥且容易出错。相反，可以使用一些工具来完成该工作，具体可参考本文档最后的示例说明
- ✓ 不要试图利用 jmp 或者 call 指令跳到 bang 过程的代码中，这些指令使用相对 PC 的寻址，很难正确达到前述目标。相反，**应向栈中压入地址并使用 ret 指令**实现跳转

Level 3: rumble

与前一级别相同，本级别实验需要在攻击字符串中包含实际的机器指令以实现改写原返回地址指针、执行特定的攻击行为等目标。与前一级别不同之处是，本级别需要在攻击字符串中包含准备和传递过程调用参数的指令，注意所传递参数与攻击字符串一样将占用栈中的存储空间。

在 bufbomb 程序中，有一个 rumble 过程，代码如下：

```
void rumble(char *str)
{
    if (eval2equal(str, cookie)) {
        printf("Rumble!: You called rumble(\"%s\")\n", str);
        validate(3);
    } else
        printf("Misfire: You called rumble(\"%s\")\n", str);
    exit(0);
}
```

该过程将进一步调用一个原型为"int eval2equal(char *strval, unsigned val)"的过程，并当 eval2equal 过程的返回值为真时，提示正确地通过了本实验级别。

与前面级别类似，本实验级别的任务是让 bufbomb 执行 rumble 过程中的代码而不是返回到 test 过程继续执行。具体来讲，攻击代码应首先在栈上准备正确的调用参数，再将 rumble 过程的地址压入栈中，然后**执行一条 ret 指令从而跳至 rumble 过程的代码继续执行**。

建议：

- ✓ 可以使用 OBJDUMP 工具分析被调用过程 eval2equal 的执行逻辑（可集中注意力于自 sprintf 过程调用及其参数压栈开始的指令，其前的多数指令与随机数的生成和使用有关——与本实验级别的主要目的关系不大），以获得传递参数的正确取值，从而构造满足实验目标的攻击字符串
- ✓ 如前一级别实验，可以使用一些工具来帮助进行指令的字节编码（可参考本文档最后的示例说明），并通过向栈中压入地址并使用 ret 指令来实现跳转
- ✓ 在栈中合理安排调用参数的值的存储空间

Level 4: 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过程中，进而结束整个程序的运行。因此，使用攻击字符串破坏、改写栈中原来保存的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，还设法使程序能够返回到原来的调用过程（例如 test）继续执行——即调用过程感觉不到攻击行为。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）对栈状态的破坏

本实验级别的任务是构造一个攻击字符串，使得 getbuf 过程将 cookie 值返回给 test 过程，而不是返回值 1。除此之外，攻击代码应还原必要但被破坏的栈状态，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 过程。

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 5: kaboom

首先注意：要进行本级别实验，运行 bufbomb 程序（以及 hex2raw 程序）时必须加上“-n”命令行选项，从而使程序进入”Nitro”模式。

通常，一个过程的栈帧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，所有环境变量的字符串形式的值存储在栈中接近栈基地址的内存位置中，视环境变量字符串的不同占用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 使用栈空间保存自己的状态。

之前级别的实验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 过程的栈帧地址保持不变。这使得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，如果尝试将这样的攻击用于一般的程序，会发现攻击有时奏效，有时却导致段错误（segmentation fault）。

不同于之前级别，本实验级别（”Nitro”）中栈帧的地址不再固定————程序在调用 testn 过程前会在栈上分配一随机大小的内存块，因此 testn 过程及其所调用的 getbufn 过程的栈帧起始地址在每次运行程序时是一个随机、不固定的值。

另一方面，在该模式下程序调用的 getbufn 过程区别于 getbuf 过程之处在于——getbufn 过程使用的缓冲区长度在 512 字节以上，以方便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 过程返回 cookie 值至 testn 过程，而不是返回值 1。具体来说，攻击字符串应将过程返回值设为 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 过程。然而，在 Nitro 模式下运行时，bufbomb 使用输入的同一攻击字符串连续执行 5 次 getbufn 过程，每次采用不同的栈偏移位置，而攻击字符串必须使程序每次均能返回 cookie 值。

建议：

- ✓ 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）
- ✓ 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 kaboom.txt 文件中保存了攻击字符串的一个拷贝）：

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

六、实验提示：生成对应汇编指令序列的字节代码

为方便生成指令序列的字节编码表示（例如用于 Level 2-4），可以依次使用 GCC 和 OBJDUMP 对所设计完成特定攻击目标的汇编指令序列进行汇编并再反汇编，从而得到指令序列的字节编码表示。

例如，可编写一个 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
```

然后，可如下汇编再反汇编该文件：

```
linux>gcc -m32 -c example.S
linux>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（即实验所需的编码后的攻击字符串内容）。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程序中静态数据的多个字节解释成了指令（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。

按上述步骤确定了所设计机器指令对应的字节序列"68 ef cd ab 00 83 c0 11 98 ba dc fe"后，就可以把该十六进制格式字符串输入 hex2raw 程序以产生一个用于输入到 bufbomb 程序的攻击字符串。更方便的方法是，由于 hex2raw 程序支持在输入字符串中包含 C 语言块注释（以方便用户理解其中字符串对应的指令），可以编辑修改 example.d 文件为如下形式（将反汇编结果中的指令说明变为注释）：

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

然后就可将该文件做为 hex2raw 程序的输入进行实验（**参数[userid]应替换为学号**）：

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