缓冲区溢出攻击实验

（苏丰，南京大学）

一、实验介绍

本实验的目的在于加深对 IA-32/x86-64 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组可执行目标程序 phase1~5 实施一系列不同复杂度的缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变程序的运行数据状态（例如使用专门设计的数据替换程序栈中特定位置上的内容）和行为，实现实验预定的目标。

实验包含以下五个阶段：

阶段 1：获得对程序的控制

阶段 2：植入攻击代码

阶段 3：模拟过程调用

阶段 4：应对栈地址随机化

阶段 5：ROP 攻击

- 实验环境：IA32/x86-64、Linux
- 实验语言：汇编/C

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：

- phase1, …, phase5：实验各阶段中被攻击的二进制可执行目标文件，其中存在缓冲区溢出漏洞。

目标程序 phase1~5 运行时可接受一个解答文件的路径作为命令行参数，该文件中包含用于实施缓冲区溢出攻击的字符串（注意如果该字符串应作为文件中唯一一行信息）。当未指定解答文件时，程序默认将从标准输入设备（stdin）读入攻击字符串。

各目标程序 phase*n*（*n*=1~5）中包含一个以输入的攻击字符串为参数的名为 phase 的函数并被 main 函数间接调用，该函数原型为“int phase(char *inputs)”，它通常进一步调用一个名为 do_phase 的函数以及其它可能函数。该 do_phase 函数在不同阶段中的定义可能略有不同，共同点是都如下定义了一个字符缓冲区数组，并将其作为参数之一（另一参数是输入的攻击字符串）调用一个名为 digit2hex 的函数。

```
int do_phase( const char *inputs )
{
    unsigned char buffer[BUFLEN];  /* BUFLEN 是预定义的缓冲区大小 */
    digit2hex(inputs, buffer);
    return 0;
}
```

函数 digit2hex 的定义如下，它使用预定义的字符映射数组_hexmap，将作为第一个参数的十六进制字符对表示的攻击字符串转换为攻击代码，存入第二个参数指出的缓冲区中（对上例即为位于 do_phase 过程栈帧中的字符缓冲区 buffer）。

```
unsigned char* digit2hex( const char *ip, unsigned char *op )
{
```

```
    const char *ipp = 0;   int comment = 0;
    for( ; *ip != '\0'; ip ++ )  {
        ……
        if( _hexmap_[*ip & 0x7f] == 0xff )   continue;

        if( ipp )  {
            *op = (_hexmap_[*ipp & 0x7f] << 4) | (_hexmap_[*ip & 0x7f] & 0xf);
            op += 1;   ipp = 0;
        }
        else
            ipp = ip;
    }
    return op;
}
```

该函数将字符指针 ipp 和 ip 所指向的一对十六进制数字（例如'6'和'E'）转换为具有对应数值的一个字节（例如 0x6E）存于字符指针参数 op 所指的地址处，然后递增 ip（及 ipp）和 op 指针继续转换下一对数字，直到 ip 指向标志字符串结尾的'\0'字符才停止处理。可见，当 ip 参数所指向的输入字符串转换得到的字节序列的长度超过 op 参数所指向的缓冲区的长度时，向 op 指向的地址写入转换结果字节最终将导致缓冲区溢出，所写入字节将改写比缓冲区更高地址上保存的内容。例如，当缓冲区位于栈中时，溢出的字节将破坏栈中记录的重要信息如返回地址、由被调用过程保存于栈中的寄存器值等。

在本实验中，用户输入的字符串称为攻击字符串（exploit string），其表示为一个编码数字对的序列，其中每个编码数字对由两个十六进制数字组成，表示攻击代码中一个字节的值。其中，前一个十六进制数字是字节的高 4 位的数值，后一个十六进制数字是字节的低 4 位的数值。例如，攻击代码中的一个字节 0x7a 应表示为"7a"两个十六进制数字。各数字对之间可以用空格、换行等不属于'0-9'、'a-z'、'A-Z'中的字符加以分隔（这些字符将被 digit2hex 函数忽略，亦可以不分隔），例如"68 ef cd ab 00 83 c0 11 98 ba dc fe"（注意值为 0 的攻击字节应写为"00"）。此外，攻击字符串中可加入成对的"#"字符并在其间放置除了"#"字符以外的任意注释文字以便于理解（digit2hex 函数将忽略所读入攻击字符串中一对"#"字符之间的内容），例如可编写如下分为两行并包含注释的攻击字符串：

```
68 20 30 40 00          # push $0x403020 #
c3                      # ret #
```

编写完成的攻击字符串可保存在一个文本文件（例如 solution.txt）中并作为目标程序的命令行参数，以验证相应实验阶段通过与否。当攻击字符串成功完成了预定的缓冲区溢出攻击目标，程序将输出类似如下的信息（不同阶段的输出可能有所不同），提示保存于文件 solution.txt 中的攻击字符串设计正确。

```
./phase1 solution.txt
Welcome to the buffer overflow attack lab.
Reading from your solution file ...
Task succeeded.
```

为设计合适的攻击字符串完成实验任务，需要分析程序中主函数 phase 及其调用函数的反汇编代码（C 标准库函数的功能可查阅相关文献），理解函数的功能和执行逻辑，需要时也可在程序的相关函数中设置断点并使用 gdb 调试工具跟踪函数的执行，以获知函数对栈帧内容的访问和修改，相应构造满足测试要求的攻击字符串。

三、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。

Phase 1: 获得对程序的控制

在二进制目标程序 phase1 中，phase 过程调用了一个如下定义的 do_phase 过程：

```
int do_phase( char *inputs )
{
    unsigned char buffer[BUF_LEN];
    digit2hex(inputs, buffer);
    return 0;
}
```

其中具有预定义长度 BUF_LEN 的局部字符数组 buffer 被用作缓冲区，接收从参数 inputs 对应的输入字符串通过 digit2hex 过程转换得到的字节序列。

过程 phase 的 C 代码如下所示：

```
int phase( char *inputs )
{
    int result = 0;
    result = do_phase(inputs);
    message(result);
    return result;
}
```

过程 message 的 C 代码如下所示：

```
ivoid message( int result )
{
    char success[] = "Task succeeded.";
    char failure[] = "Task failed.";

    if ( result )
        puts(success);
    else
        puts(failure);
}
```

由于 do_phase 过程总是向 phase 过程返回 0，后者进一步将 0 作为参数传递给 message 过程，导致本阶段实验程序正常运行时总是输出实验失败的提示。

本实验的目标是构造有效的攻击字符串，通过造成缓冲区溢出来改变程序的运行行为。具体来说，当攻击字符串被目标程序读入后，在 do_phase 过程执行 ret 指令返回时，不是返回到调用过程 phase 中的正常返回地址继续执行，而应转而执行程序中特定地址处的指令序列，使得 phase 过程向 message 过程传递非零值从而通过本实验。

提示：

- ✓ 在本级别中，用来推断攻击字符串的所有信息都可从检查 phase1 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 工具跟踪、了解程序的运行情况。

Level 2: 植入攻击代码

本阶段实验的目标是构造有效的攻击字符串，使其被目标程序读入后，在 do_phase 过程执行 ret 指令返回时，转而执行有效的指令代码，使程序输出实验成功的提示信息并正常结束运行（要求不出现任何程序运行错误的提示）。

在本实验的目标程序 phase2 中，do_phase 函数在比字符数组缓冲区 buffer 的存储地址更高的栈地址上分配了 4 个字节的存储空间，并在过程开始时向该存储空间中存入了一个 int 型全局变量 secret_number 的值。在过程结束前，该存储空间中保存的值将与 secret_number 的值进行比较。当两者不一致时，将把一个初始值为 0 的全局变量 buffer_overflowed 的值设置为 1，并调用 abort()函数结束程序的运行。

函数 phase 的 C 代码如下所示：

```
int phase( char *inputs )
{
    register int result = 0;
    result = do_phase(inputs);
    message(result);
    return result;
}
```

注意本阶段实验的 phase 过程中保存 do_phase 过程返回值的局部变量 result 分配在通用寄存器中。message 函数当输入参数的值和全局变量 buffer_overflowed 的值均为非零时，将输出实验成功的提示，否则提示实验失败。

为实现本实验攻击目标，需要在攻击字符串中包含机器指令（称为攻击代码，exploit code），通过指令设置程序中的数据值以获得期望的攻击结果。并且，为使植入到缓冲区中的攻击指令得到执行，还需要通过攻击字符串改写过程原来的返回地址，使其指向栈中的攻击指令的开始处，从而在过程返回时转去执行攻击代码。

提示：

- ✓ 为简化实验任务，本阶段目标程序每次运行时栈中缓冲区的地址保持不变。因此，可在实施攻击前预先使用 GDB 工具跟踪目标程序的运行，以获得缓冲区的运行时地址，并在构造攻击字符串时加以使用；
- ✓ 在攻击代码中，可将某目标地址压入栈中，然后执行一条 ret 指令从而跳转至该地址处的指令执行。
- ✓ 针对所设计的实现攻击目标的汇编指令序列，可以依次使用 gcc（或 as）和 objdump 工具对汇编指令序列进行汇编并再反汇编，从而得到指令序列的对应字节编码。

Level 3: 模拟过程调用

本阶段实验的目标是构造有效的攻击字符串，使得其被目标程序读入并在 do_phase 过程中转换、写入缓冲区后，do_phase 过程在执行返回指令时不是正常返回到调用过程 phase 继续执行，而是转而执行程序中的一个名为 target 的过程并输出期望的字符串。本实验需要在攻击代码中包含准备和传递过程调用参数的指令。注意与攻击字符串一样，所传递参数、被调用的过程的栈帧等都将占用栈中的存储空间。

过程 target 的 C 代码框架如下（其中略去了部分数值，需要从反汇编代码中推测出）：

```
void target( const int vec[], int len )
{
    register int result = 0;
```

```
    if( len == …… )
        result = verify( vec, len, …… );

    if( result )
        printf("You passed the correct arguments starting at address %p with a first value of %d.\n", vec, vec[0]);
    else
        printf("You passed wrong arguments.\n");

    exit(0);
}
```

注意过程 target 需要一个 int 数组类型的入口参数 vec 和一个包含数组元素个数的 int 类型的入口参数 len 并将它们传递给过程 verify，过程 verify 另外将一个数值作为第三个入口参数。仅当参数 len 的值等于某数值且过程 verify 返回非零值时，过程 target 才输出提示实验成功的字符串，并显示关于数组 vec 的若干数值。

提示：

- ✓ 可以使用 objdump 工具分析被调用过程 verify 和 target 的执行逻辑，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；
- ✓ 攻击代码应首先在栈上准备/构建调用参数，再将 target 过程的地址压入栈中，然后执行一条 ret 指令从而跳至 target 过程的代码执行；
- ✓ 注意在栈中合理安排调用参数、target/verify 过程栈帧等的存储位置。

Level 4: 应对栈地址随机化

通常情况下，某过程栈帧的确切地址在程序的不同运行实例中往往是不同的。在之前两个实验阶段中通过一定措施获得了固定不变的运行时栈地址，使得能够基于缓冲区的确定起始地址构造攻击字符串。在本实验阶段中，栈帧的地址不再固定，目标程序在调用 do_phase 过程前会在栈上分配一随机大小的内存块，使得 do_phase 过程的栈帧起始地址在每次程序运行时是一个随机、不固定的值。由于攻击字符串中需要包含缓冲区中攻击代码的地址，如果该地址随每次程序的运行而改变，就无法在程序运行前预先知道该地址的确切值，加大了实施缓冲区溢出攻击的难度。

本阶段实验的目标是构造有效的攻击字符串，使得其被目标程序读入并在 do_phase 过程中转换、写入缓冲区后，do_phase 过程在返回时不是正常返回到 phase 过程，而是先跳转到一个名为 target 的过程执行并输出期望的字符串，然后再返回 phase 过程继续执行。与前面实验阶段不同，本阶段中目标程序使用同一输入攻击字符串连续调用多次（例如 6 次）do_phase 过程，并且每次采用不同的过程栈帧地址，要求攻击字符串能够使 do_phase 过程每次执行返回时均成功跳转到 target 过程输出期望的字符串。注意本阶段实验中 do_phase 过程使用了更大的缓冲区，以方便构造可靠的攻击代码。

提示：

- ✓ 本实验的技巧在于合理使用 nop 指令（机器代码为 0x90）构造称为空操作雪橇（nop sled）的攻击代码结构。

Level 5: ROP 攻击

本阶段实验中，目标程序 phase5 的运行时栈受到数据执行保护机制的保护，即栈所在存储区域不具有执行属性，因此即使向栈中缓冲区植入了攻击代码，代码也不能被执行以实

现攻击目标。为此，本阶段实验将探索使用 Return Oriented Programming（ROP）技术实现缓冲区溢出攻击。ROP 技术的核心是寻找并串接目标程序中已有的指令序列片段（称为 ROP gadget）以构造攻击代码来实现攻击的目标。这些 gadget 可以是目标程序中的一个过程或只是其代码段中的一段字节序列，其共同的特点是通常以 ret 指令对应的机器字节代码 0xc3 结尾。在 ROP 攻击中，通过缓冲区溢出依次将构成攻击代码的各 gadget 的起始地址置于栈中，并且使用第一个将被执行的 gadget 的起始地址替换保存于栈中的目标程序中某过程的返回地址。这样，当该过程执行 ret 指令时，将跳转到第一个 gadget 开始执行，而当其执行 ret 指令返回时，将跳转到第二个 gadget 继续执行……，如此实现各 gadget 的依次执行以实现攻击的目标。

本阶段实验的目标是构造有效的攻击字符串，使得其被目标程序读入并在 do_phase 过程中转换、写入缓冲区后，程序不是正常输出字符串“Hello, world!”，而是输出一特定字符串“Hi，学号”，其中“学号”是实验者的学号中前 9 个字符（如果学号长度少于 9 个字符，则为实际学号）。

提示：

- ✓ 在目标程序的反汇编代码中定位修改输出字符串内容的 gadget 和参数处理相关的 gadget，并设计合适的 gadget 调用顺序。
- ✓ 设计各 gadget 所需参数值在栈中的合适布局。

四、实验结果提交

1）为每个实验阶段编写一个扩展名为“txt“的文本文件 phase*n*.txt（*n*=1~5，注意采用小写形式），其中包含该实验阶段的攻击字符串（该字符串应作为文件中唯一的一行，且自行首开始存放）。

2）把所有文本文件使用 tar 工具打包为一个 tar 文件（其中不能包含任何目录结构），并命名为“学号.tar” 提交。