Introduction

这个实验包括对两个具有不同安全漏洞的程序生成总共五次攻击。你将从这个实验中获得的成果包括：
- 你将学习攻击者如何利用安全漏洞，特别是当程序未能充分防范缓冲区溢出时。
- 通过这个过程，你将更好地理解如何编写更安全的程序，以及编译器和操作系统提供的一些特性，以减少程序的脆弱性。
- 你将更深入地理解x86-64机器代码的栈和参数传递机制。
- 你将更深入地理解x86-64指令是如何编码的。
- 你将获得更多使用调试工具（如 *GDB* 和 *OBJDUMP*）的经验。

注意：在这个实验中，你将亲手体验利用操作系统和网络服务器的安全弱点的方法。我们的目的是帮助你了解程序的运行时操作，并理解这些安全弱点的本质，以便在编写系统代码时避免它们。我们不支持使用任何其他形式的攻击来未经授权地访问任何系统资源。
你将需要研究 *CS:APP3e*书籍的*3.10.3*节和*3.10.4*节作为这个实验的参考资料。

Preparation

像往常一样，这是一个个人项目。你将为专门为你生成的目标程序生成攻击。

获取文件

在你的主目录中有一个名为 *targetk.tar* 的文件，其中k是你的目标程序的唯一编号。然后输入命令：*tar -xvf targetk.tar*。这将解压缩一个名为 *targetk* 的目录，其中包含下面描述的文件。
targetk 中的文件包括：
README.txt：一个描述目录内容的文件
ctarget：一个易受代码注入攻击的可执行程序
rtarget：一个易受返回导向编程攻击的可执行程序
cookie.txt：一个8位十六进制代码，你将在攻击中使用它作为一个独特的标识符。
farm.c：你的目标的“小工具农场”的源代码，你将在生成返回导向编程攻击时使用它。
hex2raw：一个生成攻击字符串的实用工具。
在以下说明中，我们假设你已经将文件复制到了一个受保护的本地目录，并且你正在该本地目录中执行这些程序。

重要点

以下是一些关于此实验室有效解决方案的重要规则的总结。当你第一次阅读此文档时，这些点可能没有多大意义。一旦你开始后，它们将作为规则的中心参考。
ↄ 你必须在与生成你的目标类似的机器上完成作业。
ↄ 你的解决方案不能使用攻击来绕过程序中的验证代码。具体来说，你在攻击字符串中包含的用于ret指令的任何地址都应该是以下目的地之一：
– 函数touch1、touch2或touch3的地址。
– 你注入代码的地址
– 你从小工具农场中的一个小工具的地址。
ↄ 你只能使用rtarget文件中的地址构造小工具，这些地址的范围应该在函数start_farm和end_farm之间。

Target Programs

CTARGET 和 RTARGET 都从标准输入读取字符串。它们通过下面定义的函数 getbuf 来实现：

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

函数 Gets 类似于标准库函数 gets——它从标准输入读取一个字符串（以‘\n’或文件结束符终止），并将其（连同一个空终止符）存储在指定的目的地。在这段代码中，你可以看到目的地是一个声明为具有 BUFFER_SIZE 字节的数组 buf。在生成你的目标程序时，BUFFER_SIZE 是一个特定于你程序版本的编译时常数。
函数 Gets() 和 gets() 无法确定它们的目的缓冲区是否足够大，可以存储它们读取的字符串。它们只是简单地复制字节序列，可能会超出目的地分配的存储范围。
如果用户键入的字符串并被 getbuf 读取的足够短，很明显 getbuf 将返回 1，如以下执行示例所示：

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit. Getbuf returned 0x1
Normal return
```

如果你输入一个长字符串，通常会发生错误：

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

（请注意，显示的 cookie 值将与你的不同。）程序 RTARGET 也将有相同的行为。
正如错误消息所示，溢出缓冲区通常会导致程序状态被破坏，从而导致内存访问错误。你的任务是更巧妙地使用你提供给 CTARGET 和 RTARGET 的字符串，使它们做更有趣的事情。这些被称为利用字符串。
CTARGET 和 RTARGET 都接受几个不同的命令行参数：
-h：打印可能的命令行参数列表
-q：不将结果发送到评分服务器
-i FILE：从文件而不是标准输入提供输入
你的利用字符串通常会包含不对应于打印字符的 ASCII 值的字节值。程序 HEX2RAW 将使你能够生成这些原始字符串。有关如何使用 HEX2RAW 的更多信息，请参见附录 A。

重要点：

- 你的利用字符串在任何中间位置都不能包含字节值 0x0a，因为这是换行符（‘\n’）的 ASCII 码。当 Gets 遇到这个字节时，它会认为你打算终止字符串。
- HEX2RAW 期望两位十六进制值由一个或多个空格分隔。所以如果你想创建一个十六进制值为 0 的字节，你需要将其写为 00。要创建单词 0xdeadbeef，你应该将 “ef be ad de” 传递给 HEX2RAW（注意小端字节顺序所需的反转）。

当你正确解决其中一个阶段时，你的目标程序将自动向评分服务器发送通知。例如：

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

服务器将测试你的利用字符串以确保它确实有效，并将更新 Attacklab 计分板页面，表明你的 userid 已经完成了这个阶段。
你可以通过将 Web 浏览器指向以下地址来查看计分板：

http://10.120.11.13:19330/scoreboard （对于 309-312 班）
http://10.120.11.13:19340/scoreboard （对于 313-314、320-321 班）
与 Bomb Lab 不同，在这个实验室中犯错没有任何惩罚。随意向 CTARGET 和 RTARGET 发射任何你喜欢的字符串。
图 1 总结了实验室的五个阶段。可以看出，前三个阶段涉及对 CTARGET 的代码注入（CI）攻击，而最后两个阶段涉及对 RTARGET 的返回导向编程（ROP）攻击。

第一部分：代码注入攻击

在前三个阶段，你的利用字符串将攻击 CTARGET。这个程序的设置方式是，堆栈位置将在一次运行到下一次运行时保持一致，并且堆栈上的数据可以被视为可执行代码。这些特性使程序容易受到利用字符串包含可执行代码的字节编码的攻击。

第一阶段

在第一阶段，你不会注入新代码。相反，你的利用字符串将重定向程序以执行一个现有的过程。

函数 getbuf 在 CTARGET 中被一个名为 test 的函数调用，该函数具有以下 C 代码：

```
1 void test()
2 {
3 int val;
4 val = getbuf();
5 printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

当 getbuf 执行其返回语句（getbuf 的第5行）时，程序通常会在 test 函数中（此函数的第5行）恢复执行。我们想要改变这种行为。在 ctarget 文件中，有一个名为 touch1 的函数的代码，其 C 表示如下：

```
1 void touch1()
2 {
3 vlevel = 1; /* 验证协议的一部分 */
4 printf("Touch1!: You called touch1()\n");
5 validate(1);
6 exit(0);
7 }
```

你的任务是在 getbuf 执行其返回语句时，让 CTARGET 执行 touch1 的代码，而不是返回到 test。请注意，你的利用字符串可能还会破坏与此阶段不直接相关的堆栈的部分，但这不会引起问题，因为 touch1 会使程序直接退出。

一些建议：
- 你需要制定此级别的利用字符串的所有信息，都可以通过检查 CTARGET 的反汇编版本来确定。使用 objdump -d 获取这个反汇编版本。
- 目标是将 touch1 的起始地址的字节表示放置在一个位置，以便 getbuf 代码末尾的 ret 指令将控制权转移给 touch1。
- 注意字节顺序。
- 你可能想要使用 GDB 来逐步执行 getbuf 的最后几条指令，以确保它在做正确的事情。
- buf 在 getbuf 的堆栈帧中的位置取决于编译时常量 BUFFER_SIZE 的值，以及 GCC 的分配策略。你将需要检查反汇编代码以确定其位置。

第二阶段

第二阶段涉及作为你的利用字符串的一部分注入少量代码。

在 ctarget 文件中，有一个名为 touch2 的函数的代码，其 C 表示如下：

```
1 void touch2(unsigned val)
2 {
3 vlevel = 2; /* 验证协议的一部分 */
4 if (val == cookie) {
5 printf("Touch2!: You called touch2(0x%.8x)\n", val);
6 validate(2);
7 } else {
8 printf("Misfire: You called touch2(0x%.8x)\n", val);
9 fail(2);
10 }
11 exit(0);
12 }
```

你的任务是让 CTARGET 执行 touch2 的代码，而不是返回到 test。然而，在这种情况下，你必须让 touch2 认为你已经将你的 cookie 作为其参数传递。

一些建议：
- 你将需要将你注入代码的地址的字节表示放置在一个位置，以便 getbuf 代码末尾的 ret 指令将控制权转移给它。
- 回想一下，函数的第一个参数是通过寄存器 %rdi 传递的。

ↄ 你注入的代码应该将寄存器设置为你的 cookie，然后使用 ret 指令将控制权转移到 touch2 的第一个指令。
ↄ 不要尝试在你的利用代码中使用 jmp 或 call 指令。这些指令的目标地址的编码很难制定。即使你没有从调用中返回，也要使用 ret 指令进行所有控制权的转移。
ↄ 参见附录 B 中关于如何使用工具生成指令序列的字节级表示的讨论。

第三阶段

第三阶段也涉及代码注入攻击，但传递的是一个字符串作为参数。

在 ctarget 文件中，有函数 hexmatch 和 touch3 的代码，其 C 表示如下：

```
1 /* 将字符串与无符号值的十六进制表示进行比较 */
2 int hexmatch(unsigned val, char *sval)
3 {
4 char cbuf[110];
5 / 使检查字符串的位置不可预测 */
6 char *s = cbuf + random() % 100;
7 sprintf(s, "%.8x", val);
8 return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char sval)
12 {
13 vlevel = 3; / 验证协议的一部分 */
14 if (hexmatch(cookie, sval)) {
15 printf("Touch3!: You called touch3("%s")\n", sval);
16 validate(3);
17 } else {
18 printf("Misfire: You called touch3("%s")\n", sval);
19 fail(3);
20 }
21 exit(0);
22 }
```

你的任务是让 CTARGET 执行 touch3 的代码，而不是返回到 test。你必须让 touch3 认为你已经将你的 cookie 的字符串表示作为其参数传递。

一些建议：
ↄ 你需要在你的利用字符串中包含你的 cookie 的字符串表示。字符串应该由八个十六进制数字组成（从最高有效位到最低有效位），没有前导的 “0x”。

ↄ 回想一下，在 C 中，字符串表示为一系列字节，后面跟着一个值为 0 的字节。在任何 Linux 机器上键入 “man ascii” 可以看到你需要的字符的字节表示。
ↄ 你注入的代码应该将寄存器 %rdi 设置为此字符串的地址。
ↄ 当调用函数 hexmatch 和 strncmp 时，它们会将数据推送到堆栈上，覆盖 getbuf 使用的缓冲区所占据的内存部分。因此，你需要小心放置你的 cookie 的字符串表示。

Part II: Return-Oriented Programming

对 RTARGET 程序进行代码注入攻击比对 CTARGET 进行攻击要困难得多，因为它使用了两种技术来阻止此类攻击：

它使用随机化，使得堆栈位置在每次运行时都不同。这使得无法确定注入代码的位置。
它将存放堆栈的内存区域标记为不可执行，所以即使你能将程序计数器设置到注入代码的开始，程序也会因为段错误而失败。
幸运的是，有人已经设计了一种策略，通过执行现有代码而不是注入新代码来在程序中完成有用的操作。这种最通用的形式被称为返回导向编程（ROP）。ROP 的策略是在现有程序中识别由一个或多个指令后跟 ret 指令组成的字节序列。这样的段称为小工具（gadget）。图 2 说明了如何设置堆栈来执行一系列 n 个小工具。在这个图中，堆栈包含一系列小工具地址。每个小工具由一系列指令字节组成，最后一个是 0xc3，编码了 ret 指令。当程序以这种配置执行 ret 指令时，它将启动一连串小工具的执行，每个小工具末尾的 ret 指令将使程序跳转到下一个小工具的开始。

小工具可以利用由编译器生成的汇编语言语句对应的代码，特别是函数末尾的代码。在实践中，可能有一些有用的小工具，但不足以实现许多重要的操作。例如，编译后的函数不太可能在 ret 之前有 popq %rdi 作为其最后一条指令。幸运的是，对于像 x86-64 这样的字节导向的指令集，可以通过从指令字节序列的其他部分提取模式来找到小工具。

例如，rtarget 的一个版本包含以下 C 函数生成的代码：

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

这个函数对于攻击系统来说似乎用处不大。但是，这个函数的反汇编机器代码显示了一个有趣的字节序列：

```
0000000000400f15 <setval_210>:
400f15: c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)
400f1b: c3 retq
```

字节序列 48 89 c7 编码了指令 movq %rax, %rdi。这个序列后面是字节值 c3，编码了 ret 指令。函数从地址 0x400f15 开始，序列从函数的第四个字节开始。因此，这段代码包含了一个从 0x400f18 开始的小工具，它将寄存器 %rax 中的 64 位值复制到 %rdi。

你的 RTARGET 代码包含许多类似于上面所示的 setval_210 函数的函数，它们位于我们称之为小工具农场（gadget farm）的区域。你的任务将是识别小工具农场中的有用小工具，并使用它们来执行类似于第 2 和第 3 阶段的攻击。

重要提示：小工具农场由你的 rtarget 副本中的 start_farm 和 end_farm 函数界定。不要尝试从程序代码的其他部分构造小工具。