15-213, Fall 20xx

The Attack Lab: Understanding Buffer Overflow Bugs

Assigned: Tue, Sept. 29

Due: Thu, Oct. 8, 11:59PM EDT

Last Possible Time to Turn in: Sun, Oct. 11, 11:59PM EDT

1 Introduction

This assignment involves generating a total of five attacks on two programs having different security vulnerabilities. Outcomes you will gain from this lab include:

- You will learn different ways that attackers can exploit security vulnerabilities when programs do not safeguard themselves well enough against buffer overflows.
- Through this, you will get a better understanding of how to write programs that are more secure, as well as some of the features provided by compilers and operating systems to make programs less vulnerable.
- You will gain a deeper understanding of the stack and parameter-passing mechanisms of x86-64 machine code.
- You will gain a deeper understanding of how x86-64 instructions are encoded.
- You will gain more experience with debugging tools such as GDB and OBJDUMP.

Note: In this lab, you will gain firsthand experience with methods used to exploit security weaknesses in operating systems and network servers. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of these security weaknesses so that you can avoid them when you write system code. We do not condone the use of any other form of attack to gain unauthorized access to any system resources.

You will want to study Sections 3.10.3 and 3.10.4 of the CS:APP3e book as reference material for this lab.

15-213,20xx 年秋季 攻击实验室：了解缓冲区溢出错
误分配时间：9 月 29 日，星期二

截止日期：美国东部时间 10 月 8 日星期四晚上 11：59
最后可能上交的时间：美国东部时间 10 月 11 日星期日晚上 11：59

1 介绍

此任务涉及对具有不同安全漏洞的两个程序总共生成五次攻击。您将从本实验室获得的成果包括：

- 您将了解当程序不能很好地保护自己免受缓冲区溢出时，攻击者可以利用安全漏洞的不同方式。
- 通过这个，您将更好地了解如何编写更安全的程序，以及编译器和作系统提供的一些功能，以降低程序的脆弱性。
- 您将更深入地了解 x86-64 机器代码的堆栈和参数传递机制。
- 您将更深入地了解 x86-64 指令的编码方式。
- 您将获得更多使用 GDB 和 OBJDUMP 等调试工具的经验。

注意：在本实验中，您将获得用于利用作系统和网络服务器中安全漏洞的方法的第一手经验。我们的目的是帮助您了解程序的运行时作，并了解这些安全漏洞的性质，以便在编写系统代码时避免它们。我们不容忍使用任何其他形式的攻击来未经授权访问任何系统资源。

您需要学习 CS：APP3e 书的第 3.10.3 和 3.10.4 节作为本实验的参考资料。

2 Logistics

As usual, this is an individual project. You will generate attacks for target programs that are custom generated for you.

2.1 Getting Files

You can obtain your files by pointing your Web browser at:

```
    http://$Attacklab::SERVER_NAME:15513/

INSTRUCTOR: $Attacklab::SERVER_NAME is the machine that runs the
 attacklab servers. You define it in attacklab/Attacklab.pm and in
 attacklab/src/build/driverhdrs.h
```

The server will build your files and return them to your browser in a `tar` file called `target`k`.tar`, where k is the unique number of your target programs.

Note: It takes a few seconds to build and download your target, so please be patient.

Save the `target`k`.tar` file in a (protected) Linux directory in which you plan to do your work. Then give the command: `tar -xvf target`k`.tar`. This will extract a directory `target`k containing the files described below.

You should only download one set of files. If for some reason you download multiple targets, choose one target to work on and delete the rest.

Warning: If you expand your `target`k`.tar` on a PC, by using a utility such as Winzip, or letting your browser do the extraction, you'll risk resetting permission bits on the executable files.

The files in `target`k include:

`README.txt`: A file describing the contents of the directory

`ctarget`: An executable program vulnerable to *code-injection* attacks

`rtarget`: An executable program vulnerable to *return-oriented-programming* attacks

`cookie.txt`: An 8-digit hex code that you will use as a unique identifier in your attacks.

`farm.c`: The source code of your target's "gadget farm," which you will use in generating return-oriented programming attacks.

`hex2raw`: A utility to generate attack strings.

In the following instructions, we will assume that you have copied the files to a protected local directory, and that you are executing the programs in that local directory.

2 物流

像往常一样，这是一个单独的项目。您将为自定义生成的目标程序生成攻击。

2.1 获取文件

您可以通过将 Web 浏览器指向以下位置来获取文件：

网址：http：//$Attacklab：：SERVER_NAME：15513/

INSTRUCTOR：$Attacklab：：SERVER_NAME 是运行 attacklab 服务器的机器。您可以在 attacklab/Attacklab.pm 和 attacklab/src/build/driverhdrs.h 中定义它

服务器将构建您的文件，并将它们以名为 targetk.tar 的 tar 文件形式返回到您的浏览器，其中 k 是目标程序的唯一编号。

注意：构建和下载目标需要几秒钟的时间，因此请耐心等待。

将 targetk.tar 文件保存在您计划在其中工作的（受保护的）Linux 目录中。然后给出命令：tar -xvf targetk.tar。这将提取包含下述文件的目录 targetk。
您应该只下载一组文件。如果由于某种原因下载了多个目标，请选择一个要处理的目标并删除其余目标。
警告：如果您在 PC 上扩展 targetk.tar，使用 Winzip 等实用程序或让您的浏览器进行解压，您将面临重置可执行文件的权限位的风险。

targetk 中的文件包括：

README.txt：描述目录内容的文件

ctarget：易受代码注入攻击的可执行程序

rtarget：易受面向返回编程攻击的可执行程序

cookie.txt：一个 8 位十六进制代码，您将用作攻击中的唯一标识符。

farm.c：目标“小工具农场”的源代码，您将使用它来生成面向返回的编程攻击。

hex2raw：生成攻击字符串的实用程序。

在以下说明中，我们将假设您已将文件复制到受保护的本地目录，并且您正在执行该本地目录中的程序。

2.2 Important Points

Here is a summary of some important rules regarding valid solutions for this lab. These points will not make much sense when you read this document for the first time. They are presented here as a central reference of rules once you get started.

- You must do the assignment on a machine that is similar to the one that generated your targets.
- Your solutions may not use attacks to circumvent the validation code in the programs. Specifically, any address you incorporate into an attack string for use by a `ret` instruction should be to one of the following destinations:
  - The addresses for functions `touch1`, `touch2`, or `touch3`.
  - The address of your injected code
  - The address of one of your gadgets from the gadget farm.
- You may only construct gadgets from file `rtarget` with addresses ranging between those for functions `start_farm` and `end_farm`.

3 Target Programs

Both CTARGET and RTARGET read strings from standard input. They do so with the function `getbuf` defined below:

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

The function `Gets` is similar to the standard library function `gets`—it reads a string from standard input (terminated by '`\n`' or end-of-file) and stores it (along with a null terminator) at the specified destination. In this code, you can see that the destination is an array `buf`, declared as having `BUFFER_SIZE` bytes. At the time your targets were generated, `BUFFER_SIZE` was a compile-time constant specific to your version of the programs.

Functions `Gets()` and `gets()` have no way to determine whether their destination buffers are large enough to store the string they read. They simply copy sequences of bytes, possibly overrunning the bounds of the storage allocated at the destinations.

If the string typed by the user and read by `getbuf` is sufficiently short, it is clear that `getbuf` will return 1, as shown by the following execution examples:

```
unix> ./ctarget
```

2.2 要点

下面是有关本实验室有效解决方案的一些重要规则的摘要。当您第一次阅读本文档时，这些要点将没有多大意义。一旦您开始，它们将作为规则的中心参考呈现在此处。

- 您必须在与生成目标的计算机类似的计算机上执行分配。
- 您的解决方案不得使用攻击来规避程序中的验证代码。具体而言，您合并到攻击字符串中以供 ret 指令使用的任何地址都应指向以下目标之一：
  – 功能 touch1、touch2 或 touch3 的地址。
  – 您注入的代码的地址– 来自小工具群的某个小工具的地址。
- 您只能从文件 rtarget 构建地址介于 start_farm 和 end_farm 之间的小工具。

3 目标项目

CTARGET 和 RTARGET 都从标准输入中读取字符串。他们使用下面定义的函数 getbuf 来做到这一点：

```
1 个无符号 getbuf（）
2 {
3 oxf[BUFFER_SIZE];
4 Gets（buf）；
5 返回 1;
6 }
```

函数 Gets 类似于标准库函数 gets——它从标准输入（以 '\n' 或文件末尾结尾）读取字符串，并将其（连同空终止符）存储在指定的目标位置。在此代码中，您可以看到目标是一个数组 buf，声明为具有 BUFFER_SIZE 个字节。在生成目标时，BUFFER_SIZE 是特定于程序版本的编译时常量。

函数 Gets（） 和 gets（） 无法确定它们的目标缓冲区是否足够大以存储它们读取的字符串。它们只是复制字节序列，可能会超出在目标处分配的存储边界。

如果用户键入并由 getbuf 读取的字符串足够短，则很明显 getbuf 将返回 1，如以下执行示例所示：

```
统一的> ./ctarget
```

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

Typically an error occurs if you type a long string:

```
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
```

(Note that the value of the cookie shown will differ from yours.) Program RTARGET will have the same behavior. As the error message indicates, overrunning the buffer typically causes the program state to be corrupted, leading to a memory access error. Your task is to be more clever with the strings you feed CTARGET and RTARGET so that they do more interesting things. These are called *exploit* strings.

Both CTARGET and RTARGET take several different command line arguments:

`-h`**:** Print list of possible command line arguments

`-q`**:** Don't send results to the grading server

`-i FILE`**:** Supply input from a file, rather than from standard input

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program HEX2RAW will enable you to generate these *raw* strings. See Appendix A for more information on how to use HEX2RAW.

Important points:

- Your exploit string must not contain byte value `0x0a` at any intermediate position, since this is the ASCII code for newline ('`\n`'). When `Gets` encounters this byte, it will assume you intended to terminate the string.
- HEX2RAW expects two-digit hex values separated by one or more white spaces. So if you want to create a byte with a hex value of 0, you need to write it as `00`. To create the word `0xdeadbeef` you should pass "`ef be ad de`" to HEX2RAW (note the reversal required for little-endian byte ordering).

When you have correctly solved one of the levels, your target program will automatically send a notification to the grading server. For example:

```
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!
```

```
Cookie：0x1a7dd803 类型字符串：保持简短！没
有漏洞利用。Getbuf 返回 0x1 正常返回
```

通常，如果键入长字符串，则会出现错误：

```
统一的> ./ctarget
Cookie：0x1a7dd803
类型字符串：这不是一个非常有趣的字符串，但它具有以下属性 ...
哎哟！：你造成了分段错误！下次好运
```

（请注意，显示的 cookie 的值将与您的值不同。程序 RTARGET 将具有相同的行为。如错误消息所示，溢出缓冲区通常会导致程序状态损坏，从而导致内存访问错误。你的任务是更聪明地使用你提供给 CTARGET 和 RTARGET 的字符串，以便它们做更多有趣的事情。这些称为漏洞利用字符串。

CTARGET 和 RTARGET 都采用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

您的漏洞利用字符串通常包含与打印字符的 ASCII 值不对应的字节值。程序 HEX2RAW 将使您能够生成这些原始字符串。有关如何使用 HEX2RAW 的更多信息，请参阅附录 A。

要点：

- 您的漏洞利用字符串不得在任何中间位置包含字节值 0x0a，因为这是换行符 （'\n'） 的 ASCII 代码。当 Gets 遇到此字节时，它将假定您打算终止字符串。

- HEX2RAW 需要两位数的十六进制值，由一个或多个空格分隔。因此，如果您想创建一个十六进制值为 0 的字节，则需要将其写为 00。要创建单词 0xdeadbeef 您应该将 “ef be ad de” 传递给 HEX2RAW（请注意小端字节排序所需的反转）。

当您正确解决其中一个级别后，您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 类型 string：Touch2！：您调用了 touch2（0x1a7dd803）
具有目标 ctarget 的 2 级的有效解决方案 PASSED：已将漏洞利用字符串发送到服
务器进行验证。干得好！
```

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	25
3	CTARGET	3	CI	`touch3`	25
4	RTARGET	2	ROP	`touch2`	35
5	RTARGET	3	ROP	`touch3`	5

CI: Code injection
ROP: Return-oriented programming

Figure 1: Summary of attack lab phases

The server will test your exploit string to make sure it really works, and it will update the Attacklab scoreboard page indicating that your userid (listed by your target number for anonymity) has completed this phase.

You can view the scoreboard by pointing your Web browser at

```
http://$Attacklab::SERVER_NAME:15513/scoreboard
```

Unlike the Bomb Lab, there is no penalty for making mistakes in this lab. Feel free to fire away at CTARGET and RTARGET with any strings you like.

IMPORTANT NOTE: You can work on your solution on any Linux machine, but in order to submit your solution, you will need to be running on one of the following machines:

```
INSTRUCTOR: Insert the list of the legal domain names that you
 established in buflab/src/config.c.
```

Figure 1 summarizes the five phases of the lab. As can be seen, the first three involve code-injection (CI) attacks on CTARGET, while the last two involve return-oriented-programming (ROP) attacks on RTARGET.

4 Part I: Code Injection Attacks

For the first three phases, your exploit strings will attack CTARGET. This program is set up in a way that the stack positions will be consistent from one run to the next and so that data on the stack can be treated as executable code. These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

4.1 Level 1

For Phase 1, you will not inject new code. Instead, your exploit string will redirect the program to execute an existing procedure.

Function `getbuf` is called within CTARGET by a function `test` having the following C code:

阶段	程序	级别	方法	功能点
1	CTARGET	1	CI	触摸 1 10 2 CTARGET 2 CI 触摸 2 25 3 CTARGET 3 CI 触摸 3 25
4	RTARGET	2	ROP	触摸 2 35
5	RTARGET	3	ROP	touch3 5 CI：代码注入 ROP：面向返回的编程

图 1：攻击实验室阶段摘要

服务器将测试您的漏洞利用字符串以确保它确实有效，并将更新 Attacklab 记分牌页面，指示您的用户 ID（按您的目标编号列出以示匿名）已完成此阶段。

您可以通过将 Web 浏览器指向

```
http：//$Attacklab：：SERVER_NAME：15513/scoreboard
```

与炸弹实验室不同，在这个实验室中犯错不会受到惩罚。随意用任何您喜欢的字符串向 CTARGET 和 RTARGET 开火。

重要说明：您可以在任何 Linux 计算机上处理您的解决方案，但为了提交您的解决方案，您需要在以下计算机之一上运行：

INSTRUCTOR：插入您在 buflab/src/config.c 中建立的合法域名列表。

图 1 总结了实验室的五个阶段。可以看出，前三个涉及对 CTARGET 的代码注入 （CI） 攻击，而后两个涉及对 RTARGET 的面向返回编程 （ROP） 的攻击。

4 第一部分：代码注入攻击

在前三个阶段，您的漏洞利用字符串将攻击 CTARGET。该程序的设置方式是，堆栈位置在一次运行到下一次运行之间保持一致，以便堆栈上的数据可以被视为可执行代码。这些功能使程序容易受到攻击，其中漏洞利用字符串包含可执行代码的字节编码。

4.1 1 级

对于第 1 阶段，您不会注入新代码。相反，您的漏洞利用字符串将重定向程序以执行现有过程。

函数 getbuf 由具有以下 C 代码的函数测试在 CTARGET 中调用：

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

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within function `test` (at line 5 of this function). We want to change this behavior. Within the file `ctarget`, there is code for a function `touch1` having the following C representation:

```
1 void touch1()
2 {
3     vlevel = 1;        /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

Your task is to get CTARGET to execute the code for `touch1` when `getbuf` executes its return statement, rather than returning to `test`. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since `touch1` causes the program to exit directly.

Some Advice:

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of CTARGET. Use `objdump -d` to get this dissembled version.
- The idea is to position a byte representation of the starting address for `touch1` so that the `ret` instruction at the end of the code for `getbuf` will transfer control to `touch1`.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of `getbuf` to make sure it is doing the right thing.
- The placement of `buf` within the stack frame for `getbuf` depends on the value of compile-time constant `BUFFER_SIZE`, as well the allocation strategy used by GCC. You will need to examine the disassembled code to determine its position.

4.2 Level 2

Phase 2 involves injecting a small amount of code as part of your exploit string.

Within the file `ctarget` there is code for a function `touch2` having the following C representation:

```
1 void touch2(unsigned val)
```

```
1 个无效测试（）
2 {
3 int val;
4       val = getbuf（）;
5 printf（“没有漏洞。Getbuf 返回 0x%x\n”， val）;
6 }
```

当 getbuf 执行其 return 语句（getbuf 的第 5 行）时，程序通常会在函数 test 中恢复执行（在此函数的第 5 行）。我们希望改变这种行为。在文件 ctarget 中，有一个函数 touch1 的代码具有以下 C 表示：

```
1 void touch1（）
2 {
3 vlevel = 1;/* 验证协议的一部分 */
4       printf（“Touch1！：你调用了 touch1（）\n”）;
5       验证（1）;
6       出口（0）;
7 }
```

您的任务是让 CTARGET 在 getbuf 执行其 return 语句时执行 touch1 的代码，而不是返回测试。请注意，您的漏洞利用字符串还可能损坏与此阶段没有直接关系的堆栈部分，但这不会导致问题，因为 touch1 会导致程序直接退出。

一些建议：

- 设计此级别的漏洞利用字符串所需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 获取此拆解版本。
- 这个想法是定位 touch1 起始地址的字节表示，以便 getbuf 代码末尾的 ret 指令将控制权转移到 touch1。
- 小心字节排序。
- 您可能希望使用 GDB 对程序执行 getbuf 的最后几条指令，以确保它正在做正确的事情。
- 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 使用的分配策略。您需要检查反汇编的代码以确定其位置。

4.2 2 级

第 2 阶段涉及注入少量代码作为漏洞利用字符串的一部分。

在文件 ctarget 中，有一个函数 touch2 的代码，具有以下 C 表示：

```
1 void touch2（无符号 val）
```

```
2 {
3     vlevel = 2;        /* Part of validation protocol */
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 }
```

Your task is to get CTARGET to execute the code for `touch2` rather than returning to `test`. In this case, however, you must make it appear to `touch2` as if you have passed your cookie as its argument.

Some Advice:

- You will want to position a byte representation of the address of your injected code in such a way that `ret` instruction at the end of the code for `getbuf` will transfer control to it.
- Recall that the first argument to a function is passed in register `%rdi`.
- Your injected code should set the register to your cookie, and then use a `ret` instruction to transfer control to the first instruction in `touch2`.
- Do not attempt to use `jmp` or `call` instructions in your exploit code. The encodings of destination addresses for these instructions are difficult to formulate. Use `ret` instructions for all transfers of control, even when you are not returning from a call.
- See the discussion in Appendix B on how to use tools to generate the byte-level representations of instruction sequences.

4.3 Level 3

Phase 3 also involves a code injection attack, but passing a string as argument.

Within the file `ctarget` there is code for functions `hexmatch` and `touch3` having the following C representations:

```
1 /* Compare string to hex represention of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
```

```
2 {
3 vlevel = 2;/* 验证协议的一部分 */ 4 if （val == cookie） {

5            printf（“Touch2！： 你叫 touch2（0x%.8x）\n”， val）；
6 验证（2）；
7 } 否则 {
8 printf（“失火：您调用了 touch2（0x%.8x）\n”， val）；
9 失败（2）；
10 }
11       出口（0）；
12  }
```

您的任务是让 CTARGET 执行 touch2 的代码，而不是返回测试。但是，在这种情况下，您必须使其看起来像 touch2 一样，就好像您已将 cookie 作为其参数传递一样。

一些建议：

- 您需要定位注入代码地址的字节表示，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在寄存器 %rdi 中传递的。
- 注入的代码应将寄存器设置为 cookie，然后使用 ret 指令将控制权转移到 touch2 中的第一个指令。
- 不要尝试在漏洞利用代码中使用 jmp 或调用指令。这些指令的目标地址的编码很难制定。对所有控制权转移使用 ret 指令，即使您没有从呼叫中返回。
- 请参阅附录 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。

4.3 3 级

第 3 阶段还涉及代码注入攻击，但将字符串作为参数传递。

在文件 ctarget 中，有函数 hexmatch 和 touch3 的代码，具有以下 C 表示：

```
1 /* 将字符串与无符号值的十六进制表示进行比较 */
2 int hexmatch（无符号 val， char *sval）
3 {
4 char cbuf[110];
5 /* 使校验字符串的位置不可预测 */
6       字符 *s = cbuf + random（） % 100;
7       sprintf（s， “%.8x”， val）；
8       返回 strncmp（sval， s， 9） == 0;
9 }
```

```
10
11 void touch3(char *sval)
12 {
13     vlevel = 3;        /* Part of validation protocol */
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 }
```

Your task is to get CTARGET to execute the code for `touch3` rather than returning to `test`. You must make it appear to `touch3` as if you have passed a string representation of your cookie as its argument.

Some Advice:

- You will need to include a string representation of your cookie in your exploit string. The string should consist of the eight hexadecimal digits (ordered from most to least significant) without a leading "`0x`."
- Recall that a string is represented in C as a sequence of bytes followed by a byte with value 0. Type "`man ascii`" on any Linux machine to see the byte representations of the characters you need.
- Your injected code should set register `%rdi` to the address of this string.
- When functions `hexmatch` and `strncmp` are called, they push data onto the stack, overwriting portions of memory that held the buffer used by `getbuf`. As a result, you will need to be careful where you place the string representation of your cookie.

5 Part II: Return-Oriented Programming

Performing code-injection attacks on program RTARGET is much more difficult than it is for CTARGET, because it uses two techniques to thwart such attacks:

- It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located.
- It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail with a segmentation fault.

Fortunately, clever people have devised strategies for getting useful things done in a program by executing existing code, rather than injecting new code. The most general form of this is referred to as *return-oriented programming* (ROP) [1, 2]. The strategy with ROP is to identify byte sequences within an existing program that consist of one or more instructions followed by the instruction `ret`. Such a segment is referred to as a

```
10
11 void touch3（char *sval）
12 {
13       vlevel = 3;/* 验证协议的一部分 */
14 如果 （hexmatch（cookie， sval））｛
15 printf（“Touch3!： 你叫 touch3（\”%s\“）\n”， sval）;
16 验证（3）;
17 } else {
18 printf（“失火：您调用了 touch3（\”%s\“）\n”， sval）;
19            失败（3）;
20       }
21       出口（0）;
22 }
```

您的任务是让 CTARGET 执行 touch3 的代码，而不是返回测试。您必须使其看起来像 touch3 一样，就好像您已将 cookie 的字符串表示形式作为其参数传递一样。

一些建议：

- 您需要在漏洞利用字符串中包含 cookie 的字符串表示形式。字符串应由八个十六进制数字组成（从最重要到最不重要排序），不带前导“0x”。
- 回想一下，字符串在 C 中表示为字节序列，后跟值为 0 的字节。在任何 Linux 机器上键入“man ascii”以查看所需字符的字节表示。
- 注入的代码应将 register %rdi 设置为此字符串的地址。
- 当调用函数 hexmatch 和 strncmp 时，它们会将数据推送到堆栈上，覆盖保存 getbuf 使用的缓冲区的内存部分。因此，您需要小心放置 cookie 的字符串表示形式。

5 第二部分：面向回报的编程

对程序 RTARGET 执行代码注入攻击比对 CTARGET 要困难得多，因为它使用两种技术来阻止此类攻击：

- 它使用随机化，因此堆栈位置因一次运行而异。这使得无法确定注入的代码将位于何处。
- 它将保存堆栈的内存部分标记为不可执行，因此即使您可以将程序计数器设置为注入代码的开头，程序也会因分段错误而失败。

幸运的是，聪明的人已经设计了策略，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事情。最通用的形式称为面向回报的编程（ROP）[1,2]。ROP 的策略是识别现有程序中的字节序列，该程序由一条或多条指令后跟指令 ret 组成。这样的段称为

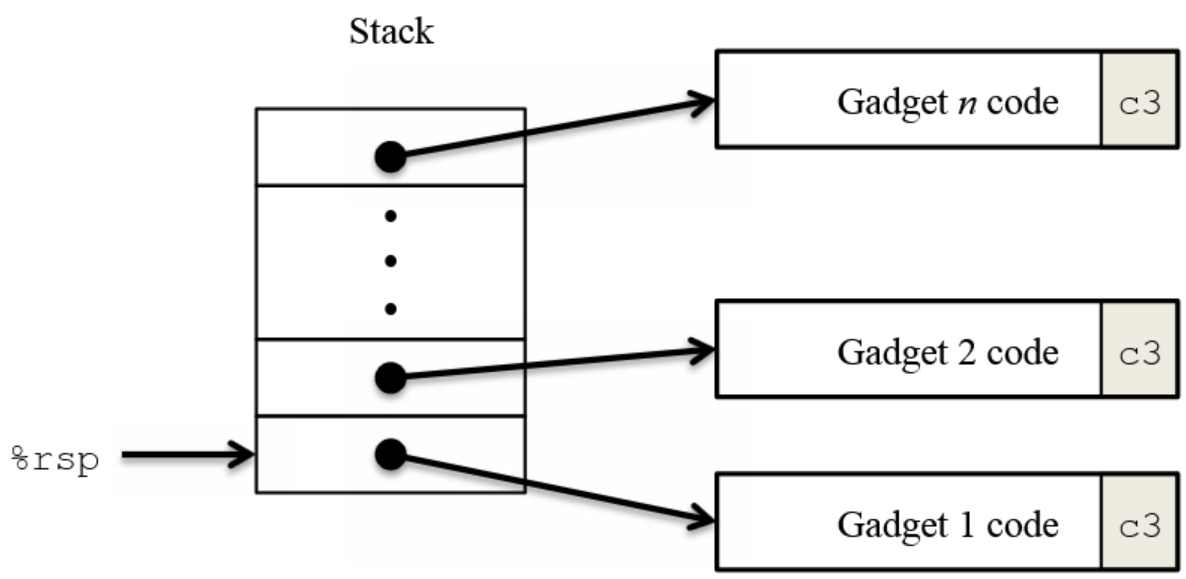

Figure 2: Setting up sequence of gadgets for execution. Byte value `0xc3` encodes the `ret` instruction.

gadget. Figure 2 illustrates how the stack can be set up to execute a sequence of n gadgets. In this figure, the stack contains a sequence of gadget addresses. Each gadget consists of a series of instruction bytes, with the final one being `0xc3`, encoding the `ret` instruction. When the program executes a `ret` instruction starting with this configuration, it will initiate a chain of gadget executions, with the `ret` instruction at the end of each gadget causing the program to jump to the beginning of the next.

A gadget can make use of code corresponding to assembly-language statements generated by the compiler, especially ones at the ends of functions. In practice, there may be some useful gadgets of this form, but not enough to implement many important operations. For example, it is highly unlikely that a compiled function would have `popq %rdi` as its last instruction before `ret`. Fortunately, with a byte-oriented instruction set, such as x86-64, a gadget can often be found by extracting patterns from other parts of the instruction byte sequence.

For example, one version of `rtarget` contains code generated for the following C function:

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

The chances of this function being useful for attacking a system seem pretty slim. But, the disassembled machine code for this function shows an interesting byte sequence:

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

The byte sequence `48 89 c7` encodes the instruction `movq %rax, %rdi`. (See Figure 3A for the encodings of useful `movq` instructions.) This sequence is followed by byte value `c3`, which encodes the `ret` instruction. The function starts at address `0x400f15`, and the sequence starts on the fourth byte of the function. Thus, this code contains a gadget, having a starting address of `0x400f18`, that will copy the 64-bit value in register `%rax` to register `%rdi`.

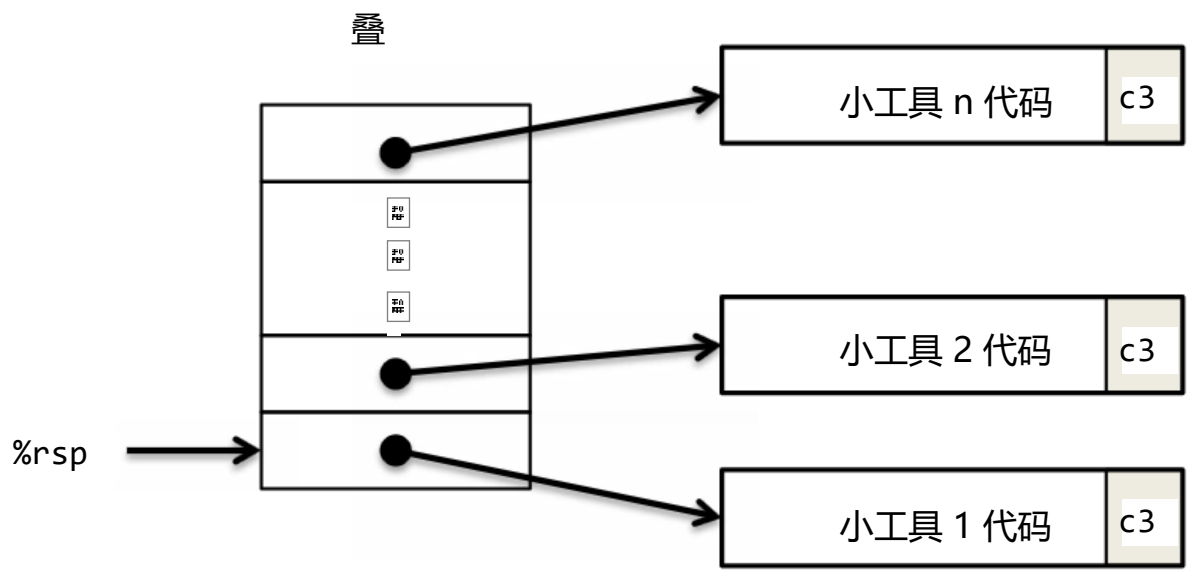

图 2：设置要执行的小工具序列。字节值 0xc3 对 ret 指令进行编码。

最通用的形式称为面向返回的编程 （ROP） [1 个小工具。图 2 说明了如何设置堆栈以执行 n 个小工具序列。在此图中，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，最后一个字节是 0xc3，对 ret 指令进行编码。当程序执行从此配置开始的 ret 指令时，它将启动一系列小工具执行，每个小工具末尾的 ret 指令会导致程序跳转到下一个小工具的开头。

小工具可以使用与编译器生成的汇编语言语句相对应的代码，尤其是函数末尾的语句。在实践中，可能有一些这种形式的有用小工具，但不足以实现许多重要的作。例如，编译后的函数不太可能将 popq %rdi 作为其 ret 之前的最后一条指令。幸运的是，对于面向字节的指令集（例如 x86-64），通常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，一个版本的 rtarget 包含为以下 C 函数生成的代码：

```
无效 setval_210（无符号 *p） {
     *p = 3347663060U;
}
```

此功能对攻击系统有用的可能性似乎很小。但是，此函数的反汇编机器代码显示了一个有趣的字节序列：

```
00000000000400f15 ： 400f15： c7 07 d4 48 89 c7 movl $0xc78948d4，（%rdi） 400f1b： c3
retq
```

字节序列 48 89 c7 对指令 movq %rax， %rdi 进行编码。 （有关有用的 movq 指令的编码，请参见图 3A。此序列后面跟着字节值 c3，它对 ret 指令进行编码。该函数从地址 0x400f15 开始，序列从函数的第四个字节开始。因此，此代码包含一个起始地址为 0x400f18 的小工具，它会将寄存器 %rax 中的 64 位值复制到寄存器 %rdi。

Your code for RTARGET contains a number of functions similar to the `setval_210` function shown above in a region we refer to as the *gadget farm*. Your job will be to identify useful gadgets in the gadget farm and use these to perform attacks similar to those you did in Phases 2 and 3.

Important: The gadget farm is demarcated by functions `start_farm` and `end_farm` in your copy of `rtarget`. Do not attempt to construct gadgets from other portions of the program code.

5.1 Level 2

For Phase 4, you will repeat the attack of Phase 2, but do so on program RTARGET using gadgets from your gadget farm. You can construct your solution using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (`%rax`–`%rdi`).

`movq` : The codes for these are shown in Figure 3A.

`popq` : The codes for these are shown in Figure 3B.

`ret` : This instruction is encoded by the single byte `0xc3`.

`nop` : This instruction (pronounced "no op," which is short for "no operation") is encoded by the single byte `0x90`. Its only effect is to cause the program counter to be incremented by 1.

Some Advice:

- All the gadgets you need can be found in the region of the code for `rtarget` demarcated by the functions `start_farm` and `mid_farm`.
- You can do this attack with just two gadgets.
- When a gadget uses a `popq` instruction, it will pop data from the stack. As a result, your exploit string will contain a combination of gadget addresses and data.

5.2 Level 3

Before you take on the Phase 5, pause to consider what you have accomplished so far. In Phases 2 and 3, you caused a program to execute machine code of your own design. If CTARGET had been a network server, you could have injected your own code into a distant machine. In Phase 4, you circumvented two of the main devices modern systems use to thwart buffer overflow attacks. Although you did not inject your own code, you were able inject a type of program that operates by stitching together sequences of existing code. You have also gotten 95/100 points for the lab. That's a good score. If you have other pressing obligations consider stopping right now.

Phase 5 requires you to do an ROP attack on RTARGET to invoke function `touch3` with a pointer to a string representation of your cookie. That may not seem significantly more difficult than using an ROP attack to invoke `touch2`, except that we have made it so. Moreover, Phase 5 counts for only 5 points, which is not a true measure of the effort it will require. Think of it as more an extra credit problem for those who want to go beyond the normal expectations for the course.

RTARGET 的代码包含许多类似于上面所示的 setval_210 函数的函数，该区域称为小工具场。你的工作是识别小工具农场中有用的小工具，并使用它们来执行类似于您在第 2 阶段和第 3 阶段中所做的攻击。
重要提示：小工具场由 rtarget 副本中的函数 start_farm 和 end_farm 划分。不要尝试从程序代码的其他部分构造小工具。

5.1 2 级

对于第 4 阶段，您将重复第 2 阶段的攻击，但在程序 RTARGET 上使用小工具场中的小工具进行攻击。您可以使用由以下指令类型组成的小工具来构造解决方案，并且仅使用前八个 x86-64 寄存器 （%rax–%rdi）。

movq ： 这些代码如图 3A 所示。
popq ： 这些代码如图 3B 所示。

ret ： 此指令由单字节 0xc3 编码。

nop ： 此指令（发音为“no op”，是“no operation”的缩写）由单字节 0x90 编码。它的唯一作用是使程序计数器递增 1。

一些建议：

- 您需要的所有小工具都可以在由函数 start_farm 和 mid_farm 划定的 rtarget 代码区域中找到。

- 您只需使用两个小工具即可进行此攻击。
- 当小工具使用 popq 指令时，它将从堆栈中弹出数据。因此，您的漏洞利用字符串将包含小工具地址和数据的组合。

5.2 3 级

在进行第 5 阶段之前，请停下来考虑一下您迄今为止所取得的成就。在第 2 阶段和第 3 阶段，您使程序执行您自己设计的机器代码。如果 CTARGET 是一台网络服务器，您可以将自己的代码注入到远处的机器中。在第 4 阶段中，您规避了现代系统用来阻止缓冲区溢出攻击的两个主要设备。尽管您没有注入自己的代码，但您可以注入一种程序，该程序通过将现有代码序列拼接在一起来运行。您还获得了 95/100 分的实验室积分。这是一个不错的分数。如果您有其他紧迫的义务，请考虑立即停止。
第 5 阶段要求您对 RTARGET 进行 ROP 攻击，以调用函数 touch3，并指向 cookie 的字符串表示。这似乎并不比使用 ROP 攻击调用 touch2 困难得多，只不过我们已经做到了这一点。此外，第 5 阶段只计 5 分，这并不是衡量其所需努力的真实指标。对于那些想要超越课程正常期望的人来说，这更像是一个额外的学分问题。

A. Encodings of `movq` instructions

`movq` S, D

Source S	Destination D							
	`%rax`	`%rcx`	`%rdx`	`%rbx`	`%rsp`	`%rbp`	`%rsi`	`%rdi`
`%rax`	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
`%rcx`	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
`%rdx`	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
`%rbx`	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
`%rsp`	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
`%rbp`	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
`%rsi`	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
`%rdi`	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register R							
	`%rax`	`%rcx`	`%rdx`	`%rbx`	`%rsp`	`%rbp`	`%rsi`	`%rdi`
`popq` R	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` S, D

Source S	Destination D							
	`%eax`	`%ecx`	`%edx`	`%ebx`	`%esp`	`%ebp`	`%esi`	`%edi`
`%eax`	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
`%ecx`	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
`%edx`	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
`%ebx`	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
`%esp`	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
`%ebp`	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
`%esi`	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
`%edi`	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register R			
	`%al`	`%cl`	`%dl`	`%bl`
`andb` R, R	20 c0	20 c9	20 d2	20 db
`orb` R, R	08 c0	08 c9	08 d2	08 db
`cmpb` R, R	38 c0	38 c9	38 d2	38 db
`testb` R, R	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

A. movq 指令的编码

movq S、D

源目标 D																
S	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi	%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48
89 c7	%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf	%rdx	48 89 d0	48 89 d1	48 89				
d2 48 89	d3 48 89	d448 89	d5 48 89	d6 48 89	d7	%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89				
de 48 89	df %rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7	%rbp	48 89 e8	48 89 e9					
48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef	%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5					
48 89 f6	48 89 f7	%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff							

B. popq 指令的编码

作寄存器 R								
%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi	popq R 58 59 5a
5b 5c 5d 5e 5f								

C. movl 指令的编码

移动 S、D

源目标 D																
S	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi	%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89		
c5 89	c6 89	c7	%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf	%edx	89 d0	89			
d1 89	d2 89	d3 89	d4 89	d5 89	d6 89	d7	%ebx	89 d8	89 d9	d9 89	da 89	db 89	dc 89				
dd 89	de89	df	%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7	%ebp	89 e8	89			
e9 89	ea 89	eb 89	ec 89	ed 89	ee 89	ef	%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5				
89 f6	89 f7	%edi	89 f8	89 f9	f9 89	fa 89	fb 89	fc 89	fd 89	fe 89	ff						

D. 2 字节函数 nop 指令的编码

作寄存器 R																
%al	%cl	%dl	%bl	和	b R,	R	20 c0	20	c9	20 d2	20	db	球			
体 R,	R	08 c0	08	c9	08 d2	08 db	cmpb R,	R	38 c0	38						
c9 38 d2 38 db	testb R,	R	84 c0	84 c9	84 d2	84 db										

图 3：指令的字节编码。所有值均以十六进制显示。

To solve Phase 5, you can use gadgets in the region of the code in `rtarget` demarcated by functions `start_farm` and `end_farm`. In addition to the gadgets used in Phase 4, this expanded farm includes the encodings of different `movl` instructions, as shown in Figure 3C. The byte sequences in this part of the farm also contain 2-byte instructions that serve as *functional nops*, i.e., they do not change any register or memory values. These include instructions, shown in Figure 3D, such as `andb %al,%al`, that operate on the low-order bytes of some of the registers but do not change their values.

Some Advice:

- You'll want to review the effect a `movl` instruction has on the upper 4 bytes of a register, as is described on page 183 of the text.
- The official solution requires eight gadgets (not all of which are unique).

Good luck and have fun!

A Using HEX2RAW

HEX2RAW takes as input a *hex-formatted* string. In this format, each byte value is represented by two hex digits. For example, the string "`012345`" could be entered in hex format as "`30 31 32 33 34 35 00`." (Recall that the ASCII code for decimal digit x is `0x3`x, and that the end of a string is indicated by a null byte.)

The hex characters you pass to HEX2RAW should be separated by whitespace (blanks or newlines). We recommend separating different parts of your exploit string with newlines while you're working on it. HEX2RAW supports C-style block comments, so you can mark off sections of your exploit string. For example:

```
48 c7 c1 f0 11 40 00 /* mov    $0x40011f0,%rcx */
```

Be sure to leave space around both the starting and ending comment strings ("`/*`", "`*/`"), so that the comments will be properly ignored.

If you generate a hex-formatted exploit string in the file `exploit.txt`, you can apply the raw string to CTARGET or RTARGET in several different ways:

1. You can set up a series of pipes to pass the string through HEX2RAW.

   ```
   unix> cat exploit.txt | ./hex2raw | ./ctarget
   ```

2. You can store the raw string in a file and use I/O redirection:

   ```
   unix> ./hex2raw < exploit.txt > exploit-raw.txt
   unix> ./ctarget < exploit-raw.txt
   ```

   This approach can also be used when running from within GDB:

要解决第 5 阶段，您可以在 rtarget 中由函数 start_farm 和 end_farm 划定的代码区域中使用小工具。除了第 4 阶段中使用的小工具外，这个扩展的农场还包括不同 movl 指令的编码，如图 3C 所示。服务器场这一部分中的字节序列还包含用作功能 nop 的 2 字节指令，即它们不更改任何寄存器或内存值。其中包括图 3D 所示的指令，例如 andb %al，%al，它们对某些寄存器的低阶字节进行作，但不会改变它们的值。

一些建议：

- 您需要查看 movl 指令对寄存器上 4 个字节的影响，如本文第 183 页所述。
- 官方解决方案需要八个小工具（并非所有小工具都是唯一的）。

祝你好运，玩得开心！

A 使用 HEX2RAW

HEX2RAW 将十六进制格式的字符串作为输入。在这种格式中，每个字节值由两个十六进制数字表示。例如，字符串“012345”可以以十六进制格式输入为“30 31 32 33 34 35 00”。（回想一下，十进制数字 x 的 ASCII 代码是 0x3x，字符串的末尾由空字节表示。）传递给 HEX2RAW 的十六进制字符应用空格（空格或换行符）分隔。我们建议在处理漏洞利用字符串时用换行符分隔漏洞利用字符串的不同部分。HEX2RAW 支持 C 样式块注释，因此您可以标记漏洞利用字符串的各个部分。 例如：

```
48 c7 c1 f0 11 40 00 /* mov $0x40011f0, %rcx */
```

请务必在开始和结束注释字符串（“/*”、“*/”）周围留出空格，以便正确忽略注释。
如果在文件 exploit.txt 中生成十六进制格式的漏洞利用字符串，则可以将原始字符串应用于

CTARGET 或 RTARGET 以几种不同的方式：

1. 您可以设置一系列管道来将绳子穿过 HEX2RAW。UNIX> cat exploit.txt | ./hex2raw | ./ctarget

2. 您可以将原始字符串存储在文件中并使用 I/O 重定向：unix> ./hex2raw < exploit.txt > exploit-raw.txt

```
unix> ./ctarget < exploit-raw.txt
```

在 GDB 中运行时也可以使用此方法：

```
unix> gdb ctarget
(gdb) run < exploit-raw.txt
```

3. You can store the raw string in a file and provide the file name as a command-line argument:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -i exploit-raw.txt
```

This approach also can be used when running from within GDB.

B Generating Byte Codes

Using GCC as an assembler and OBJDUMP as a disassembler makes it convenient to generate the byte codes for instruction sequences. For example, suppose you write a file `example.s` containing the following assembly code:

```
# Example of hand-generated assembly code
        pushq   $0xabcdef           # Push value onto stack
        addq    $17,%rax            # Add 17 to %rax
        movl    %eax,%edx           # Copy lower 32 bits to %edx
```

The code can contain a mixture of instructions and data. Anything to the right of a '#' character is a comment.

You can now assemble and disassemble this file:

```
unix> gcc -c example.s
unix> objdump -d example.o > example.d
```

The generated file `example.d` contains the following:

```
example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 68 ef cd ab 00        pushq  $0xabcdef
   5: 48 83 c0 11           add    $0x11,%rax
   9: 89 c2                 mov    %eax,%edx
```

The lines at the bottom show the machine code generated from the assembly language instructions. Each line has a hexadecimal number on the left indicating the instruction's starting address (starting with 0), while

```
UNIX> gdb ctarget （gdb） 运行<
exploit-raw.txt
```

3. 您可以将原始字符串存储在文件中，并将文件名作为命令行参数提供：unix> ./hex2raw < exploit.txt > exploit-raw.txt

```
unix> ./ctarget -i exploit-raw.txt
```

在 GDB 中运行时也可以使用此方法。

B 生成字节码

使用 GCC 作为汇编器，使用 OBJDUMP 作为反汇编器可以方便地生成指令序列的字节码。例如，假设您编写了一个包含以下程序集代码的文件 example.s：

```
# 手工生成的汇编代码示例
        pushq $0xabcdef # 将值推送到堆栈上 addq $17，%rax # 将 17 添加到 %rax movl
        %eax，%edx # 将较低的 32 位复制到 %edx
```

代码可以包含指令和数据的混合。“#”字符右侧的任何内容都是注释。您现在可以汇编和反汇编此文件：

```
unix> gcc -c example.s unix> objdump -d
example.o > example.d
```

生成的文件 example.d 包含以下内容：

```
example.o：文件格式 elf64-x86-64


.text 部分的反汇编：

0000000000000000000000 <.text>： 0： 68 EF CD AB 00 Pushq
$0xabcdef
    5： 48 83 c0 11 添加 $0x11，%rax 9： 89 c2 mov %eax，%edx
```

底部的行显示了从汇编语言指令生成的机器代码。每行左侧都有一个十六进制数字，表示指令的起始地址（从 0 开头），而

the hex digits after the '`:`' character indicate the byte codes for the instruction. Thus, we can see that the instruction `push $0xABCDEF` has hex-formatted byte code `68 ef cd ab 00`.

From this file, you can get the byte sequence for the code:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

This string can then be passed through HEX2RAW to generate an input string for the target programs.. Alternatively, you can edit example.d to omit extraneous values and to contain C-style comments for readability, yielding:

```
68 ef cd ab 00   /* pushq  $0xabcdef  */
48 83 c0 11      /* add    $0x11,%rax */
89 c2            /* mov    %eax,%edx  */
```

This is also a valid input you can pass through HEX2RAW before sending to one of the target programs.

References

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15(1):2:1–2:34, March 2012.

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

而“：”字符后面的十六进制数字表示指令的字节码。因此，我们可以看到指令推送 $0xABCDEF 具有十六进制格式的字节码 68 ef cd ab 00。

从这个文件中，您可以获取代码的字节序列：

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

然后可以将此字符串传递给 HEX2RAW，为目标程序生成输入字符串。或者，您可以编辑 example.d 以省略无关值并包含 C 样式注释以提高可读性，从而产生：

```
68 ef cd ab 00 /* pushq $0xabcdef */ 48 83 c0 11 /* 加
法 $0x11，%rax */ 89 c2 /* mov %eax，%edx */
```

这也是在发送到目标程序之一之前可以通过 HEX2RAW 传递的有效输入。

引用

[1] R. Roemer、E. Buchanan、H. Shacham 和 S. Savage。面向返回的编程：系统、语言和应用程序。ACM 信息系统安全汇刊，15（1）：2：1–2：34,2012 年 3 月。

[2] EJ Schwartz、T. Avgerinos 和 D. Brumley。问：漏洞利用强化变得简单。2011 年 USENIX 安全研讨会。