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

* Introduction

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

这项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：

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.

您将了解到，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法。

通过这些，您将更好地了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功能，以使程序不易受到攻击。

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指令是如何编码的。

您将获得更多使用GDB和OBJDUMP等调试工具的经验。

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.

注意：在这个实验中，您将获得利用操作系统和网络服务器中的安全弱点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操作，并了解这些安全弱点的本质，以便您在编写系统代码时能够避免它们。我们不容忍使用任何其他形式的攻击来获得对任何系统资源的未经授权的访问。

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

你需要学习CS:APP3e书中的3.10.3和3.10.4节作为本实验室的参考资料。

1

* Logistics

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

和往常一样，这是一个单独的项目。您将为为您自定义生成的目标程序生成攻击。

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

您可以通过将Web浏览器指向以下位置来获取文件：

http://$Attacklab:：服务器\_姓名：15513/

讲师：$Attacklab:：SERVER\_NAME是运行Attacklab服务器的机器。你在attacklab里定义它/攻击实验室.pm在attacklab/src/build/driverhdrs.h中

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

服务器将生成您的文件，并将它们返回到您的浏览器中，该tar文件名为targetk.tar公司，其中k是目标程序的唯一编号。

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

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

注意：构建和下载目标需要几秒钟的时间，所以请耐心等待。

保存targetk.tar公司文件放在（受保护的）Linux目录中，您计划在其中执行工作。然后给出命令：tar-xvftargetk.tar公司. 这将提取包含以下文件的目录targetk。

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.

你应该只下载一组文件。如果出于某种原因下载了多个目标，请选择一个目标进行处理，然后删除其余目标。

Warning: If you expand your targetk.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.

警告：如果您扩展targetk.tar公司在PC上，通过使用Winzip等实用程序，或者让浏览器执行提取，您将有可能重置可执行文件的权限位。

The files in targetk 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.

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位的十六进制代码，您将在攻击中用作唯一标识符。

farm.c：目标“gadget farm”的源代码，您将使用该源代码生成面向返回的编程攻击。

hex2raw：生成攻击字符串的实用程序。

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.

在下面的说明中，我们将假定您已将文件复制到受保护的本地目录，并且您正在该本地目录中执行程序。

2

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.

以下是关于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，这些要点没有多大意义。一旦您开始使用，它们将作为规则的中心参考。

You must do the assignment on a machine that is similar to the one that generated your targets.

您必须在与生成目标的计算机相似的计算机上执行分配。

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:

您的解决方案不能使用攻击来规避程序中的验证代码。具体地说，您合并到攻击字符串中以供ret指令使用的任何地址都应该指向以下目标之一：

– 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.

–函数touch1、touch2或touch3的地址。

–注入代码的地址

–gadget farm中某个小工具的地址。

You may only construct gadgets from file rtarget with addresses ranging between those for func-tions start\_farm and end\_farm.

您只能从file rtarget构造gadget，其地址介于start\_farm和end\_farm函数的地址之间。

* Target Programs

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

CTARGET和RTARGET都从标准输入读取字符串。它们使用下面定义的函数getbuf执行此操作：

1 unsigned getbuf()

* {
* char buf[BUFFER\_SIZE];
* Gets(buf);

5return 1;

* }

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.

函数get类似于标准库函数Gets，它从标准输入（以'\n'或文件结尾结尾）读取字符串并将其（连同空终止符）存储在指定的目标。在这段代码中，您可以看到目标是一个数组buf，声明为具有缓冲区大小字节。在生成目标时，缓冲区大小是特定于程序版本的编译时常量。

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.

它们的get（）函数是否足够大来确定它们的destination（）是否足够大。它们只是复制字节序列，可能会超出在目的地分配的存储的界限。

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:

如果用户键入并由getbuf读取的字符串足够短，则getbuf显然将返回1，如下执行示例所示：

unix> ./ctarget

3

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.

（请注意，显示的cookie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。您的任务是更巧妙地使用您提供给CTARGET和RTARGET的字符串，以便它们做更有趣的事情。这些被称为攻击字符串。

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

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： 不将结果发送到分级服务器

-i文件：从文件提供输入，而不是从标准输入

报错

拼音 双语对照

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.

攻击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，请参见附录A。

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).

您的攻击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，用一个或多个空格隔开。所以如果你想创建一个十六进制值为0的字节，你需要把它写成00。要创建单词0xdeadbeef，您应该将“ef be ad de”传递给HEX2RAW（注意，little-endian字节排序需要反转）。

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

正确求解其中一个级别后，目标程序将自动向放坡服务器发送通知。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!

4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 score-board page indicating that your userid (listed by your target number for anonymity) has completed this phase.

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记分板页面，指示您的userid（按匿名目标号码列出）已完成此阶段。

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.

与炸弹实验室不同，在这个实验室里犯错误是没有惩罚的。你可以随意用你喜欢的任何字符串来攻击CTARGET和RTARGET。

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.

重要提示：您可以在任何Linux计算机上处理解决方案，但要提交解决方案，您需要在以下计算机之一上运行：

讲师：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，前三个阶段涉及对CTARGET的代码注入（CI）攻击，而后两个阶段涉及对RTARGET的面向返回编程（ROP）攻击。

* 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.

在前三个阶段，利用漏洞字符串将攻击CTARGET。此程序的设置方式使堆栈位置在每次运行时保持一致，因此堆栈上的数据可以作为可执行代码处理。这些特性使程序容易受到攻击，攻击字符串包含可执行代码的字节编码。

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:

对于阶段1，您不会注入新代码。相反，利用漏洞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：

5

1 void test()

* {
* int val;
* val = getbuf();
* printf("No exploit. Getbuf returned 0x%x\n", val);
* }

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:

当getbuf执行它的return语句（getbuf的第5行）时，程序通常在函数test中（在这个函数的第5行）继续执行。我们想改变这种行为。在文件ctarget中，函数touch1的代码具有以下C表示：

1 void touch1()

* {
* vlevel = 1;/\* Part of validation protocol \*/

4printf("Touch1!: You called touch1()\n");

* validate(1);
* exit(0);
* }

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.

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test。请注意，您的漏洞利用字符串也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。

Some Advice:

All the information you need to devise your exploit string for this level can be determined by exam-ining 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.

设计此级别的攻击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来获取这个被分解的版本。

其思想是定位touch1的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。

注意字节排序。

您可能希望使用GDB逐步执行getbuf的最后几条指令，以确保它执行的是正确的操作。

buf在getbuf的堆栈帧中的位置取决于编译时常量缓冲区大小的值，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 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:

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:

第2阶段涉及注入少量代码作为攻击字符串的一部分。

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

1 void touch2(unsigned val)

6

* {
* vlevel = 2;/\* Part of validation protocol \*/

4if (val == cookie) {

5printf("Touch2!: You called touch2(0x%.8x)\n", val);

* validate(2);

7} else {

8printf("Misfire: You called touch2(0x%.8x)\n", val);

* fail(2);

1. }
2. exit(0);
3. }

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.

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，您必须使它看起来像touch2，就好像您已经将cookie作为参数传递了一样。

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.

您需要定位注入代码地址的字节表示形式，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。

注入的代码应该将寄存器设置为cookie，然后使用ret指令将控制权转移到touch2中的第一条指令。

不要试图在攻击代码中使用jmp或call指令。这些指令的目的地址编码很难制定。对所有的控制权转移使用ret指令，即使您没有从通话中返回。

关于如何使用工具生成指令序列的字节级表示，请参阅附录B中的讨论。

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:

第3阶段还涉及代码注入攻击，但传递字符串作为参数。

在ctarget文件中，hexmatch和touch3函数的代码具有以下C表示：

1 /\* Compare string to hex represention of unsigned value 比较字符串与无符号值的十六进制表示\*/

2 int hexmatch(unsigned val, char \*sval)

* {
* char cbuf[110];
* /\* Make position of check string unpredictable 使检查字符串的位置不可预测\*/

6char \*s = cbuf + random() % 100;

7sprintf(s, "%.8x", val);

8return strncmp(sval, s, 9) == 0;

* }

7

10

11 void touch3(char \*sval)

1. {
2. vlevel = 3;/\* Part of validation protocol \*/
3. if (hexmatch(cookie, sval)) {
4. printf("Touch3!: You called touch3(\"%s\")\n", sval);
5. validate(3);
6. } else {
7. printf("Misfire: You called touch3(\"%s\")\n", sval);
8. fail(3);
9. }
10. exit(0);
11. }

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:

您的任务是让CTARGET执行touch3的代码，而不是返回测试。您必须使它看起来像touch3，就好像您传递了cookie的字符串表示作为参数一样。

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.

您需要在漏洞利用字符串中包含cookie的字符串表示形式。字符串应该由八个十六进制数字组成（从最高到最低有效位排序），不带前导“0x”

回想一下，字符串在C语言中表示为一个字节序列，后跟一个值为0的字节。在任何Linux机器上输入“manascii”以查看所需字符的字节表示形式。

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.

注入的代码应将寄存器%rdi设置为此字符串的地址。

当函数hexmatch和strncmp被调用时，它们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的字符串表示形式的位置。

* 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:

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

It uses randomization so that the stack positions differ from one run to another. This makes it impos-sible 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.

它使用随机化，以便堆栈位置在不同的运行中有所不同。这使得无法确定注入代码的位置。

它将保存堆栈的内存部分标记为不可执行的，因此即使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。

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

幸运的是，聪明的人设计了一些策略，通过执行现有代码而不是注入新代码来完成程序中有用的事情。最常见的形式是面向返回的编程（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

8

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Stack |  |  |  |  |
|  |  |  |  |  | Gadget *n* code | c3 |  |
|  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Gadget 2 code | c3 |  |
|  |  |  |  |  |  |
| %rsp |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  | Gadget 1 code | c3 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

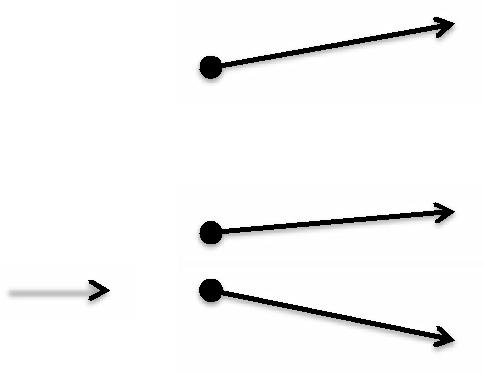


Figure 2: Setting up sequence of gadgets for execution. Byte value 0xc3 encodes the ret instruction.

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.

图2：设置小工具的执行序列。字节值0xc3对ret指令进行编码。

小玩意。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gadget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，编码ret指令。当程序从这个配置开始执行ret指令时，它将启动一系列gadget执行，每个gadget末尾的ret指令会导致程序跳转到下一个gadget的开头。

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.

gadget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操作。例如，编译后的函数不太可能将popq%rdi作为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一个小工具。

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

void setval\_210(unsigned \*p)

{

\*p = 3347663060U;

}

例如，rtarget的一个版本包含为以下C函数生成的代码：

void setval\_210（无符号\*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:

这个函数用于攻击系统的可能性很小。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：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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.

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。）此序列后面是字节值c3，它对ret指令进行编码。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gadget，其起始地址为0x400f18，它将把寄存器%rax中的64位值复制到寄存器%rdi。

9

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.

RTARGET的代码包含许多与上面显示的setval\_210函数类似的函数，这些函数位于我们称为gadget farm的区域中。您的任务是识别gadget场中有用的小工具，并使用这些小工具执行与第2阶段和第3阶段类似的攻击。

要点：gadget场由rtarget副本中的start\_farm和end\_farm函数划分。不要试图从程序代码的其他部分构造小工具。

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.

在你进入第五阶段之前，停下来考虑一下你目前已经完成了什么。在第2和第3阶段，您让一个程序执行您自己设计的机器代码。如果CTARGET是一个网络服务器，那么您可以将自己的代码注入到远程计算机中。在第4阶段中，您绕过了现代系统用来阻止缓冲区溢出攻击的两个主要设备。虽然您没有注入自己的代码，但您可以注入一种程序类型，该程序通过将现有代码的序列缝合在一起来操作。你的实验室也得了95/100分。这是一个很好的分数。如果你有其他紧迫的义务，考虑马上停止。

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.

第5阶段要求您对RTARGET执行ROP攻击，以使用指向cookie字符串表示的指针调用函数touch3。这似乎并不比使用ROP攻击来调用touch2困难得多，但我们已经做到了。此外，第5阶段只计算5个点，这不是一个真正的措施，它将需要努力。对于那些想超出课程正常预期的人来说，这更像是一个额外的学分问题。

10

A. Encodings of movq instructions

movq S, D

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Source |  |  |  |  |  |  |  |  |  | Destination 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 | 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 |  |  |  |  |  |  | Destination 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 | 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.

11

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 0x3x, 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:

12

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.

* 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

13

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.. Alter-natively, 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, lan-guages, 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.

14