第 7 章实验报告

|  |  |  |
| --- | --- | --- |
| 学号： | 姓名： 杜恩俊 | 班级： 12012222 |
| 实验题目：简单栈溢出 | | |
| 实验日期：2024.3.28 | | |
| 实验目的：了解栈溢出攻击原理，并实现简单栈溢出攻击实验 | | |
| 硬件环境：  磁盘驱动器：NVMe KIOXIA- EXCERIA G2 SSD  NVMe Micron 3400 MTFDKBA1TOTFH  显示器：NVIDIA GeForce RTX 3070 Ti Laptop GPU  系统型号 ROG Strix G533ZW\_G533ZW  系统类型 基于 x64 的电脑  处理器 12th Gen Intel(R) Core(TM) i9-12900H，2500 Mhz，14 个内核，20 个逻辑处理器  BIOS 版本/日期 American Megatrends International, LLC. G533ZW.324, 2023/2/21  BIOS 模式 UEFI  主板产品 G533ZW  操作系统名称 Microsoft Windows 11 家庭中文版 | | |
| 软件环境：  VMware Workstation Pro  Ubuntu 18.04.6 LTS  Red Panda Dev-C++ | | |
| 实验步骤与分析：   |  | | --- | | 1. #include <stdio.h> 2. #include <string.h> 3. #include <stdlib.h> 4. char input[] = "11112222333344445555666677778888"; 5. void inject(){ 6. printf("\*\*\*\*\*inject success\*\*\*\*\*\n"); 7. } 9. void func\_call(){ 10. char param[16]; 11. strcpy(param,input); 12. } 13. int main(int argc,char\*\* argv){ 14. func\_call(); 15. printf("main exit...\n"); 16. return 0; 17. } |  1. 首先在dev上编辑C语言代码StackOverflow.c如上 2. 接下来为了成功执行攻击，需要关闭Ubuntu系统中的一系列保护机制：    1. 在命令行运行cd /home/sky/Desktop将地址定位在目标文件层    2. 在终端输入sudo apt-get install gcc-multilib，代表支持交叉编译cross-compiling，例如可以在64位处理器上处理32位程序    3. 在终端输入sudo sysctl -w kernel.randomize\_va\_space=0，代表关闭进程空间地址随机化功能    4. 使用 gcc -Wall -g -o StackOverflow StackOverflow.c -fno-stack-protector -z execstack -m32 编译程序，其中-g代表关闭所有优化机制，-fno-stack-protector代表关闭Stack Canary保护，-z execstack代表禁用NX（No-eXecute protect）保护，-m32代表在编译阶段将编译目标指定为32 位   3.对程序分析可知，其将在第15行执行输出操作，故我们将断点设置在15行。  接下来使用gdb程序对该代码进行调试：  3.1 首先使用如下代码安装gdb   |  | | --- | | 1. sudo apt update 2. sudo apt install gdb |   3.2 在终端输入 gdb StackOverflow 开启调试。    4根据分析代码，我们发现关键代码在13行，故在13行设置断点。  我们输入命令：break 13。    5 然后输入命令run 代表运行程序至断点处。    6 输入disassemble后回车  结果如下：    =>这一行即为断点（13行）的步骤行。  7 输入info registers ebp esp来查看寄存器里的栈顶指针和栈底指针    8然后输入 print&param来查看param数组的首地址。     1. 输入 x/2xw 0xffffd1a8来查看该func\_call函数的返回地址，在这个命令中，"x" 是一个 GDB 命令，它是 "examine memory" 的缩写，用于检查内存中的内容。"/2xw" 是一个格式化参数，它告诉 GDB 以十六进制格式显示两个字（32位）的内容，并将其解释为一个有符号整数。"0xffffd1a8" 是内存地址，表示我们要查看的内存位置。      1. 然后输入disassemble main来反汇编main函数。      1. 可以看到 func\_cal函数的返回地址0x565556c6指向main函数里的<+31>，其and所指的位置是<+4> 2. 输入print&inject来查看inject函数的地址     现在我们分析所获取到的信息：func\_cal函数的返回地址为0xffffd1ac（0xffffd1a8+0x4），。param 的首地址为0xffffd190,两者相差28个字节。查看到inject函数地址为0x5655554d,因此可以将input输入更改为“28个字节+4位inject地址”。故将input修改为char input[] = "AAAAABBBBBCCCCCDDDDDEEEEEFFF\x4d\x55\x55\x56";后重新编译程序。   1. 在代码中修改input后重新编译程序      1. 如上图所示，攻击完成。 | | |
| 结论与体会：  在本次实验中，我深入了解并实践了栈溢出攻击的原理与技术。通过设计和实施一个简单的栈溢出攻击，我不仅加深了对程序内存布局和操作系统安全机制的理解，还学会了如何在实际环境中利用软件漏洞。  实验的过程中，我首先在Ubuntu系统下编写了一个简单的C程序，该程序包含了易受栈溢出攻击的漏洞。通过精心构造输入数据，我成功引导程序执行了未授权的inject函数，从而实现了攻击目标。实验过程中，我关闭了操作系统的几项安全保护机制，如地址空间布局随机化（ASLR）、栈保护等，以模拟一个容易受到攻击的环境。  通过本次实验，我学习到了几个重要的技术和概念：  1.栈溢出的原理：了解了栈溢出是如何通过覆盖函数的返回地址来控制程序流程的。  2.安全保护机制的重要性：实验中需要关闭的安全保护机制说明了这些机制在防御栈溢出攻击中的重要作用。  3.调试和分析工具的应用：通过使用gdb调试工具和其它命令行工具，我学会了如何分析程序的内存布局和识别潜在的安全漏洞。  这次实验不仅加强了我的理论知识，也提高了我的实践技能，让我对计算机安全领域有了更深刻的理解。我认识到，编写安全的代码需要程序员具备深厚的安全意识和技能，以及对各种攻击技术和防御策略的熟悉。未来，我希望能够继续深入研究这一领域，为创建更安全的软件环境做出贡献。 | | |