数据驱动攻击主要包括缓冲区溢出攻击与输入验证攻击。其定义就是通过数据的发送，使得目标活动产生了意外的结果。还有格式化字符串攻击、同步漏洞攻击、信任漏洞攻击、

输入验证攻击产生的原因就是因为没有对输入进行严格的验证，程序对特殊输入没有多多少处理能力。比如说telnet的时候加了个xxxroot，然后一连接就是root。

最简单的缓冲区攻击命令：echo "vrfy `perl -e 'print "a" x 1000'`" | nc www.example.com 25 通过一行perl产生1000个a并结合了echo，再由管道输出到nc，转发至其他地方。 python –c "print('a'\*3)" | nc xxx xx

1.栈金丝雀保护Stack Canaries是已知的放置在缓冲器和控制数据之间的一个随机值。当缓冲器溢出时，最先被破坏通常是金丝雀值。因此当金丝雀的数据的验证失败的时候，就表示出现了缓冲区溢出，从而触发保护机制，并使程序停止运行。这种编译是技术叫做GS。不过对应的就有了不通过改写返回地址的缓冲区溢出攻击。

缓冲区溢出常见的是栈与堆缓冲区溢出。

2.从前的windows没有采用地址空间布局随机化（Address Space Layout Randomization，ASLR）。XP SP1、SP2以及2003 Server没有使用。使得系统的数据执行保护（Data Execution Prevention，DEP）容易受到攻击。windows的1NETAPI32.dll中存在的标准化漏洞使得攻击者可以绕过系统的DEP（阻止代码从数据也被执行），并修改返回地址和各类寄存器的内容。

3.创建一个服务器消息快（Server Message Block，SMB）登录连接通过特意设计的远程过程调用（RPC）请求，将栈的返回地址改写为入侵者希望的值。我们的ShellCode就放在这个值代表的地址中。

4.解析HTTP 请求头的时候，在Connection添加大量无用数据，直接导致了缓冲区的溢出，从而将自定义数据写入寄存器。通过EIP寄存器的跳转到ESP（这里放着我们代码中的静态字符串），EIP与ESP之间有一个不确定区域，并不是连着的，我们需要自己判断出来区域大小然后随便填充掉这块区域，我们会获得目标系统的全部权限。所以缓冲区溢出攻击得到的权限都很大。

5.SHE结构化异常处理，就是一种用来处理软硬件异常的机制。在Immunity Debugger的SEH Chains功能中可以看到SEH消息。比如说缓冲区溢出攻击可能就会被SEH阻止。

6.msf的tools目录下有个pattern\_offset.rb和pattern\_start.rb可以用来进行程序缓冲区溢出fuzz测试。结合调试器找到特定的溢出位置。

在可执行模块列表中选择的时候，不要选00开头地址的模块，他可能会被看做坏字符（因为不同类型的程序、协议，这些字符会导致我们代码失效，常见的就是转义字符。对于POP3的邮件服务器，0x00表示对字符串的终止，0x0D表示回车也就是命令输入完成，这两个字符不能出现在对缓冲区的给予中，出现了也会被过滤掉而不会到缓冲区中，简单的方法是自己发送0x00-0xff 查看到底有哪些坏字符，发现还有0x0A）而不能被执行。

绕过SEH：用fuzz测试出来位置在220，那么就把溢出字符串差分成216+4+剩余部分。

7.mona.py

典型的缓冲区溢出！看到这里如何写一个利用工具相信你已经轻车熟路了，最简单的办法是找一个指向溢出段数据区的寄存器，然后在加载的模块中搜索类似于call esp,jmp esp指令的地址，手动计算溢出后覆盖返回地址的偏移位置，然后寻找合适的shellcode，再写利用程序….但是等等，如果有坏字符怎么办，会导致溢出失败或者shellcode不起作用，你有什么简单的办法确定有哪些坏字符吗？如果目标系统开启了DEP和ASLR呢？\x00和\x0a和\0x0d，这是已知的坏字符，如果出现在cwd命令中会导致命令被截断。

这一切mona都能自动帮你完成!

把mona.py放到Immunity Debugger的PyCommands文件夹下，然后通过底部的命令执行选项卡执行“!mona skeleton”-》选择网络协议-》选择端口号码，然后就会在Immunity Debugger的工作目录下生成一个叫做msfskeleton.rb的缓冲区溢出攻击模块框架文件。

msf中的make\_nops(number)可以创造一个number个\x90的字符串，number可以使用target['Offset']，target可以在initialize的super中设定。

mona bytearray \x00到\xff组成的字符串。 看一下剩余的空间能不能转下egg hunter

通过mona.py可以很方便地产生egg hunter：mona egg -t w00t

8.通过缓冲区溢出，绕过了密码的安全检验逻辑。

上述check\_result和pass两个变量由于是局部变量，故申请的时候是在栈上分配。由于栈是从高地址往低地址，所以从地址由低到高，check\_result分配在pass后面，当进行strcpy操作的时候，因为没有检测输入长度，所以pass溢出了，会覆盖后面的内存区域。由于check\_result在pass后面，所以当溢出足够的时候，check\_result被改写了。

PS：check\_result是否是仅挨着pass分配那不得而知，这个取决于编译器的优化，但是溢出就面临数据被异常改写的风险。但是肯定可以溢出。不安全的strcpy

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

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

{

if (argc != 2)

{

printf("Invalid params\n");

exit(1);

}

bool check\_result = false;

char pass[10];

memset(pass,0,10);

strcpy(pass, argv[1]);

if (0 == strcmp("xuzeshui", pass))

{

check\_result = true;

}

if (check\_result)

{

printf("Check password succ!\n");

}

else

{

printf("Check password failed!\n");

}

return 0;

}

9.对于安装有SLMail的windows的POP3溢出攻击：

关键是使用PASS命令进行数据发送，如果使用nc来进行连接的话，那么就先使用USER test来进行登录，之后就要输入密码，但是“PASS 溢出攻击字符串”

使用带有SOCKET的脚本来进行测试，因为fuzz测试的量可能很大。

对于协议，去看RFC，对于数据输入的指令，就可以进行溢出1fuzz测试；如果协议没有RFC（就像公司私有协议），就需要自己抓包分析。

10.使用Immunity Debugger可以对静态的exe文件或者已经运行着的动态进程进行调试，当你在fuzz测试的时候。

11.确定溢出位置：二分法、唯一字符串法（/usr/share/metasploit-framework/tools/pattern\_create.rb 2700 就会创建2700个字符的唯一字符串，每n个字符不会出现重复，EIP不是要装4个字节或者8个字节么。配合同样tools目录下的pattern\_offset.rb 这个ruby脚本后面跟上EIP中的十六进制；如果自己去找的话需要注意可能是小端字序）。

12.shellcode需要300字节左右。msfvenom可以产生105字节的shellcode，具体看你选择的payload咯。

13.linux的内存保护机制： 不同发行版不同

DEP ASLR 堆栈cookies 堆栈粉碎

直接对linux端口上开着的程序发送它意想不到的接收内容

14.0x90 nop

15.在写溢出代码的时候会将EIP赋予某个寄存器的值，这个寄存器上的内容是输入的地址。/usr/share/metasploit-framework/tools/nasm\_shell.rb 交互模式，可以将汇编指令转换为二进制。windows下可以使用老系统中的debug.exe

16.Heap Spray是在shellcode的前面加上大量的slide code（滑板指令），组成一个注入代码段。传统slide code（滑板指令）一般是NOP指令，但是随着一些新的攻击技术的出现，逐渐开始使用更多的类NOP指令，譬如0x0C（0x0C0C代表的x86指令是OR AL 0x0C），0x0D等等，不管是NOP还是0C，他们的共同特点就是不会影响shellcode的执行。使用slide code的原因下面还会详细讲到。

**16.5.汇总：**



现有常用缓冲区溢出技术

Return-to-libc攻击 因为任意代码都禁止注入到进程的地址空间中，所以就返回到标准C库。而不是返回到堆栈。传统观点认为这种攻击是可以管理的，但是“面向返回的编程ROP”证明这些假设是错误的，利用任意的没有函数调用的计算程序也可能实现这种攻击

17.SEH

曾说一名黑客可以通过用控制值改写异常处理程序并获得代码执行权。

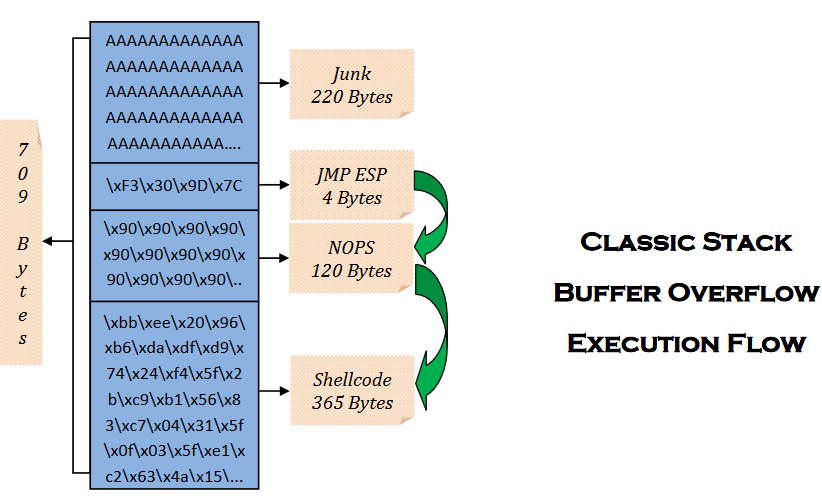
18.Egg 解决可用缓冲区太短装不了所有的shellcode的问题，可能是因为之前的junk code太多了。

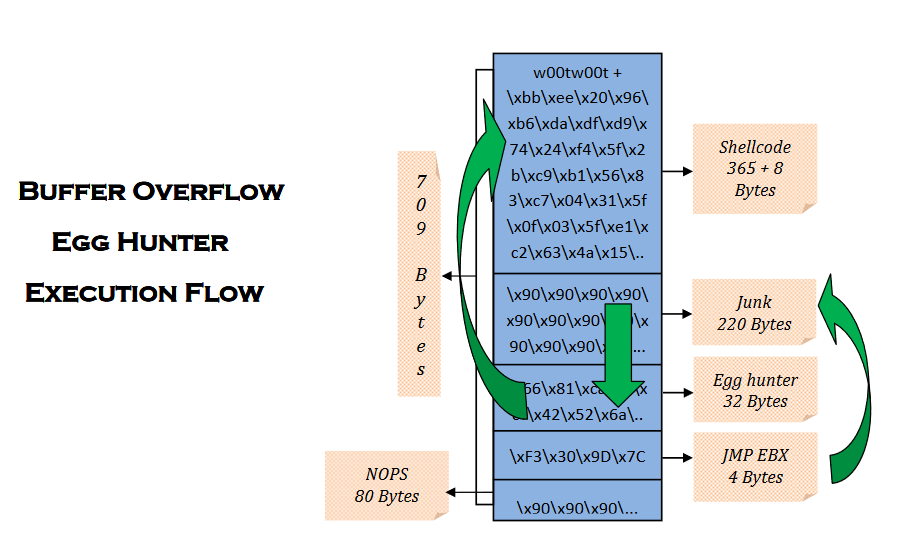
可能会因为系统和系统架构的不同而不同，因为都是直接写机器码。

<https://www.exploit-db.com/docs/18482.pdf> 英文详细版

<http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>

<http://www.pediy.com/kssd/pediy11/120392.html> 中文简化版





shellcode安放的位置或多或少是静态的或是可以利用寄存器来引用的。Egg hunting这种技巧可以被归为“分级shellcode”，它主要可以支持你用一小段特制的shellcode来找到你的实际的（更大的）shellcode。

shellcode都是在写成机器码。

你必须要用一个独特的记号标记最终的shellcode。最初的shellcode（短小的“鸡蛋猎人”）会遍历内存空间，寻找这个小标记（就是Egg）。找到它后，用一个jmp或call 指令跳转到小标记后面的代码中执行。这就意味着你需要在鸡蛋猎人代码中定义这个标记，并且把标记写到实际shellcode的前面。常常见到Egg会重复两次，这是为了优化搜索，这样每次就能够跳过4个字节再进行比较了。

但是写入内存是可以知道地址的，这样就不用这么麻烦了。网页中给的方法就是真的把从某一个地方开始的内存都找了一遍，但是代码可以改一下，直接倒着找就好。如果遇到无效内存就将指针跳过本页（or dx,0xfff），然后inc edx。之所以分成两行指令，是因为一般都要用到inc edx，所以就分成两个不同的分支。

Windows：

通过call 0x2来执行系统调用

SEH(结构化异常处理)的方法需要60字节，IsBadReadPtr需啊哟37字节，NtDispalyString方法需要32字节。（最后一种方法只适用于NT架构，其他的在Windows 9x下也可以工作。）NtAccessCheckAndAuditAlarm函数也是可以使用的。在汇编里面都是通过中断配合参数来调用指定的系统函数。需要处理的关键就是不可访内存（有的内存还没有分配）才能异常：

cmp al,0x5 ; check if access violation occurs

; (0xc0000005 == ACCESS\_VIOLATION) 5

SHE方法：自己新增一个异常处理，处理不可读异常（就是不可访问）Access Violation

NTSYSAPI NTSTATUS NTAPI NtDisplayString( IN PUNICODE\_STRING String ); 这个函数参与在蓝屏的时候在屏幕上输出字。它对于参数只会读取，不会对参数指针写入，最终使用scads命令来判断。我们通过这个函数进行异常处理判断，判断这块内存是否可读。<http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>中关于这一段汇编代码十分神奇！如果抛出异常就会进行ret操作，会自动释放栈回到上一次的cal么。0x23开始到最后的ret以及0x02到0xD 都是对中断的初始化，0x10的命令就是push handler函数到stack。

BOOL IsBadReadPtr(const VOID\* lp, UINT\_PTR ucb); 这个函数会验证8个连续字节内存的有效性，两个参数刚好8个字节。Egg hunter的size能只有37字节这么大。

Linux：

访问无效内存会产生SIGSEGV信号，那么就要注册一个这个信号的handler（自己编写SEH）或者使用系统调用接口来方便我们，遇到无效内存返回EFAULT这个错误代码，同时减少代码字节数量。Linux中暴露给用户模式程序的系统调用接口是靠软中断0x80。使用函数int access(const char \*pathname, int mode); 整个egg hunters只需要39字节，使用scasd汇编指令进行进一步优化可以缩小到35字节。int sigaction(int signum, const struct sigaction \*act, struct

sigaction \*oldact);

struct sigaction 每一个成员都是4字节

{

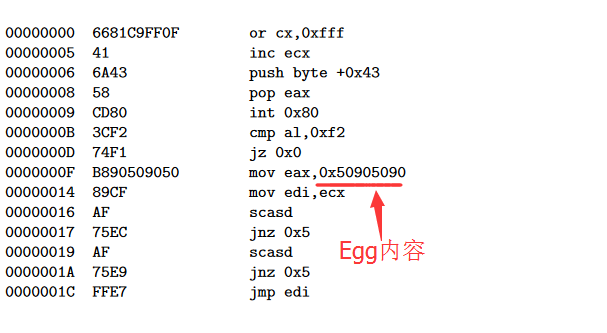
\_\_sighandler\_t sa\_handler;

\_\_sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer) (void);

};这个函数被调用的时候1，它将确保有16个字节的有效地址。于是产生了更短更快速的汇编egg hunter：



**解析：**

**首先不要忘了其实jump短跳转最终使用的是相对位移**

**jz 0x0 遇到了无效的16个字节，打算跳过这个页**

**通过两次scasd，将edi内容和eax中的内容比较。其实一次比较就够了，其余的就拿来用作跳过加速好了。**

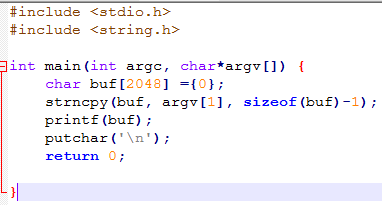
通过mona.py可以很方便地产生egg hunter：mona egg -t w00t

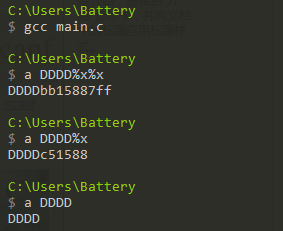
<https://www.exploit-db.com/shellcode/> shellcode合集网页

19.格式化字符串函数

最常见的一个函数就是printf，如果我们格式化字符串被作为变量，那么传入%x就能将整个进程空间的数据都读出来

当只有一个参数的时候: 如果在格式化字符串中提供了足够的额外指令，格式化字符串本身将被用作格式化指令的后续参数。对于C语言的printf，这个数值是2。





后来实验知道第一个%x的输出都是变化的，但是第二个固定在7ff。

20.

**进程环境块（PEB）：**PEB是Windows NT操作系统家族中的一个数据结构。它是由操作系统内部使用的一个不透明的数据结构，它的大部分字段不适用于操作系统之外。[1]微软MSDN文档（其中只包含了部分字段）说明这个结构可能随着操作系统版本不同而变化。[2]PEB包含全局上下文、启动参数、程序映像加载器的数据结构、映像基址和进程内提供互斥的同步对象。

**导入地址表（IAT）：**当应用程序在不同模块中调用一个函数时，地址表被用来作为一个查询表。它包括两种导入形式（序号和名字）。因为一个编译好的程序无法知道依赖库的内存位置，当调用API时需要间接跳转。因为动态链接器加载模块并将它们连接在一起，它将真实的地址写入IAT, 因此它们指向相应库函数的内存位置。

**数据执行保护（DEP）：**DEP是用来校验内存来帮助阻止恶意代码执行的一组硬件和软件的技术。在微软Windows XP SP2和Windows XP Tablet PC Edition 2005版本中，DEP通过硬件和软件实现。DEP的好处是阻止代码从数据页执行。典型的，代码不能在默认堆和栈中执行。硬件增强的DEP检测这些位置的代码运行，当执行发生时抛出异常。软件增强的DEP帮助阻止恶意代码利用Windows的异常处理机制。

**地址空间布局随机化（ASLR）：**它是一种避免缓冲区溢出攻击的保护措施。为了阻止攻击者固定的跳转，例如，一个特别的内存漏洞利用，ASLR能随机分配进程内关键区域的地址，包括可执行文件的基地址和栈、堆、动态库的位置。

**stdcall调用约定：**stdcall调用约定由pascal约定演变而来，被调用者负责清理栈，但是参数从右向左的顺序压栈（和\_cdecl调用约定一样）。寄存器EAX，ECX，EDX在函数中使用。返回结果保存在EAX中。stdcall是微软win32 API和open Watcom C++的标准调用约定。

21.为不同的操作系统编写shellcode需要不同的方法，不像Windows，基于Unix的操作系统提供了一种通过int 0x80接口与内核通信的方式，基于Unix的操作系统的所有的系统调用都有一个唯一的调用号，通过80中断来调用（int 0x80），内核通过被提供的调用号和参数执行系统调用，但是这里有个问题，Windows没有一个直接调用内核的接口，意味着不得不有精准的函数指针（内存地址）以便调用它们。不幸的是，硬编码函数地址不能完全解决问题，Windows中的每个函数地址在每个版本中都会改变，使用硬编码的shellcode高度依赖版本，在Windows上编写不依赖版本的shellcode是可能的，只要解决地址问题，这个能通过在运行时动态获取地址解决。

22.程序崩溃后自动调用gdb并加载自己，fork与信号Hook。