Internet Security [1]

VU 184.216

Language Security
Buffer Overflows

Paolo Milani Comparetti	pmilani@seclab.tuwien.ac.at
Clemens Kolbitsch	ck@seclab.tuwien.ac.at

News from the Lab

- *Challenge 5 – Man-in-the-Middle*
  - 31 people solved it
  - BERNHARD "Mind Mole" URBAN was first to steal some money
    - congrats – completes your hat-trick!!
    - 1h 54min, great time for a coding challenge
- *Challenge 6* will be announced today
  - you will have to do a stack buffer overflow
  - 2 weeks to solve
  - this will be the last challenge in Inetsec1
  - have fun!!

News from the Lab

- Some words about the exam
  - 16. Juni 2010, 12:00
  - EI 10
  - that is, same place, same time in 3 weeks
- Closed book
  - mixture of theory and practical questions
- Topics:
  - lecture content
  - challenges
  - shouldn't be too tough if you understood challenges and followed the lecture

Outline

- Overview: Buffer Overflows
- Memory Management & *x86* Process Layout
- Stack / Frames / *x86* Function Calls
- Buffer Overflow Exploitation
- Shell Code 101

Buffer Overflows

- One of the most widely used attack methods of compromising a machine
  - if successful, allows execution of *arbitrary code* in the context of the *exploited program*

- Goal / Steps
  1) inject instructions into memory of vulnerable program
  2) exploit program vulnerability to change control flow (flow of execution), and
  3) execute (arbitrary) injected code

Buffer Overflows

- 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 system call functions
  - some guess work involved (correct addressing)

Memory Management

- Many modern languages (Java, python, etc.) provide *automatic buffer size checks* when accessing memory

```
try {
  byte data[] = new byte[16];
  for (int i=0; true; i++) {
    byte b = readbyte();
    if (b == 0) break;
    data[i] = b;
  }
} catch (Exception e) {
  ...
```

Statically sized buffer. If user provides more than 16 bytes of input, the buffer will overflow!

- Here, the language will catch the overrun and throw an exception (that can be handled by the program)
  - if program ignores the exception, execution terminates (i.e., without causing any harm)

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
  - program must make sure that only the allocated number of bytes are written to the buffer
  - if it fails to do so, adjacent buffers in memory will be overwritten
  - overwritten buffers may contain sensitive information (such as passwords, memory management information, or control flow data)

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
  - program must make sure that only the allocated number of bytes are written to the buffer

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         “secret”);

for (int i=0; 1; i++) {
  char b = readbyte();
  if (b == 0) break;
  data[i] = b;
}
```

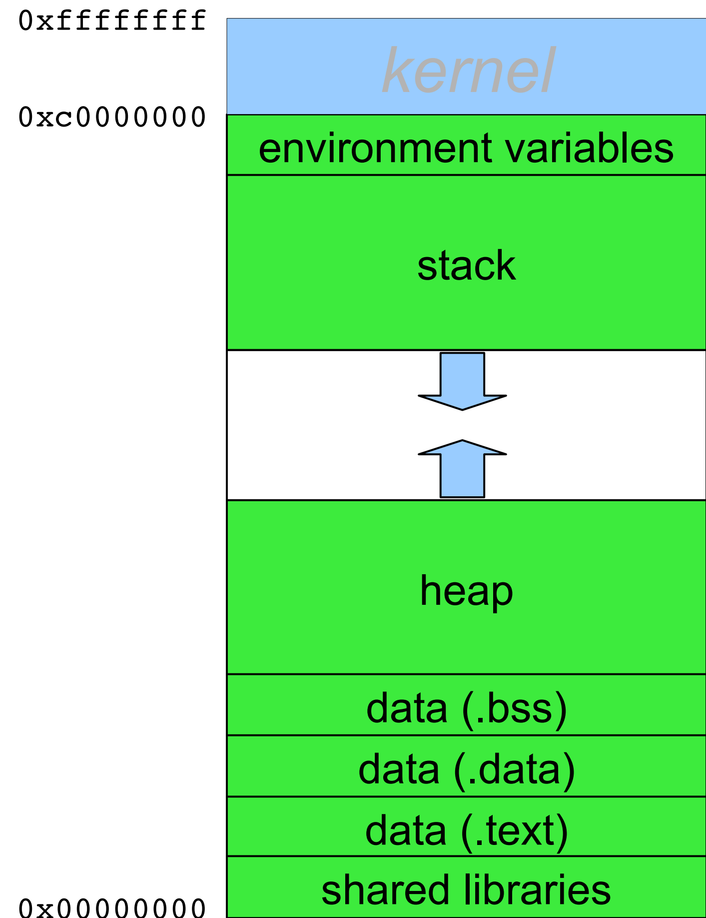
00 00 00 00
00 00 00 00
'e' 't' '\0' 00
41 41 41 41

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

Memory Layout

- Stack segment
  - local variables
  - procedure activation records
- Data segment
  - global uninitialized variables (.bss)
  - global initialized variables (.data)
  - dynamic variables (heap)
- Code (text) segment
  - program instructions
  - usually read-only
- In Linux, under the `proc` filesystem

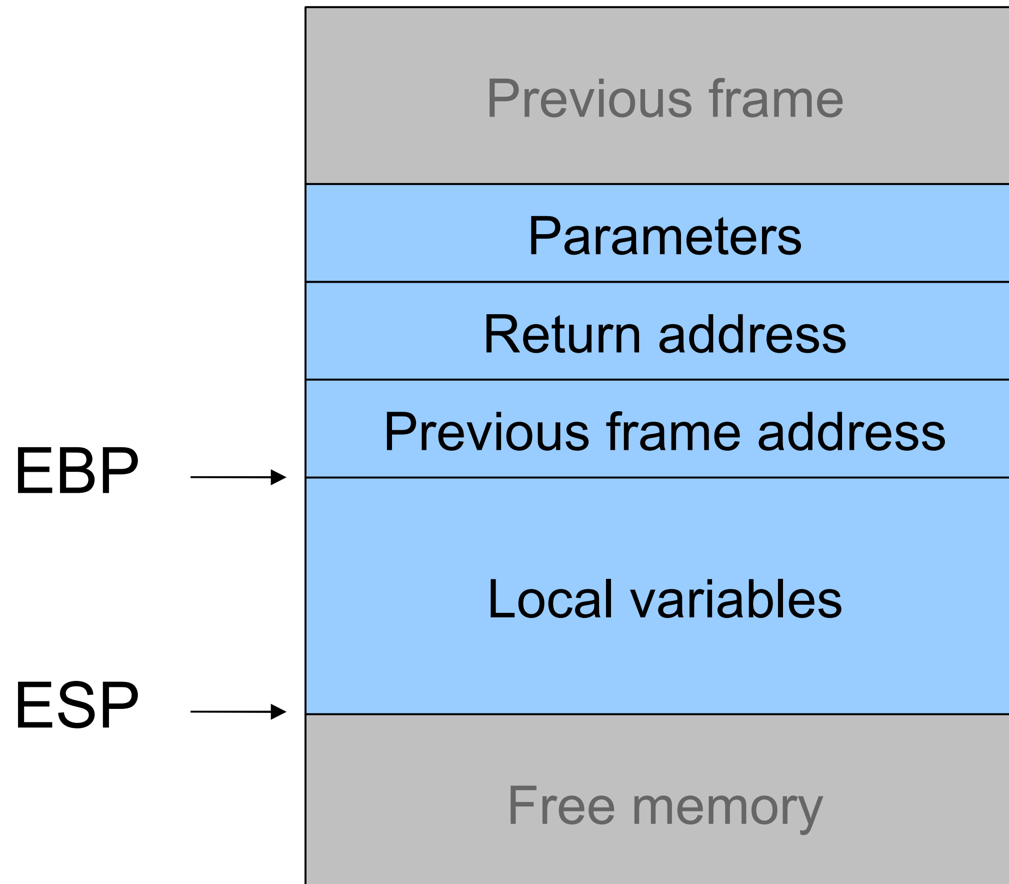
```
$ cat /proc/<pid>/maps
```



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 – FP`

Frame

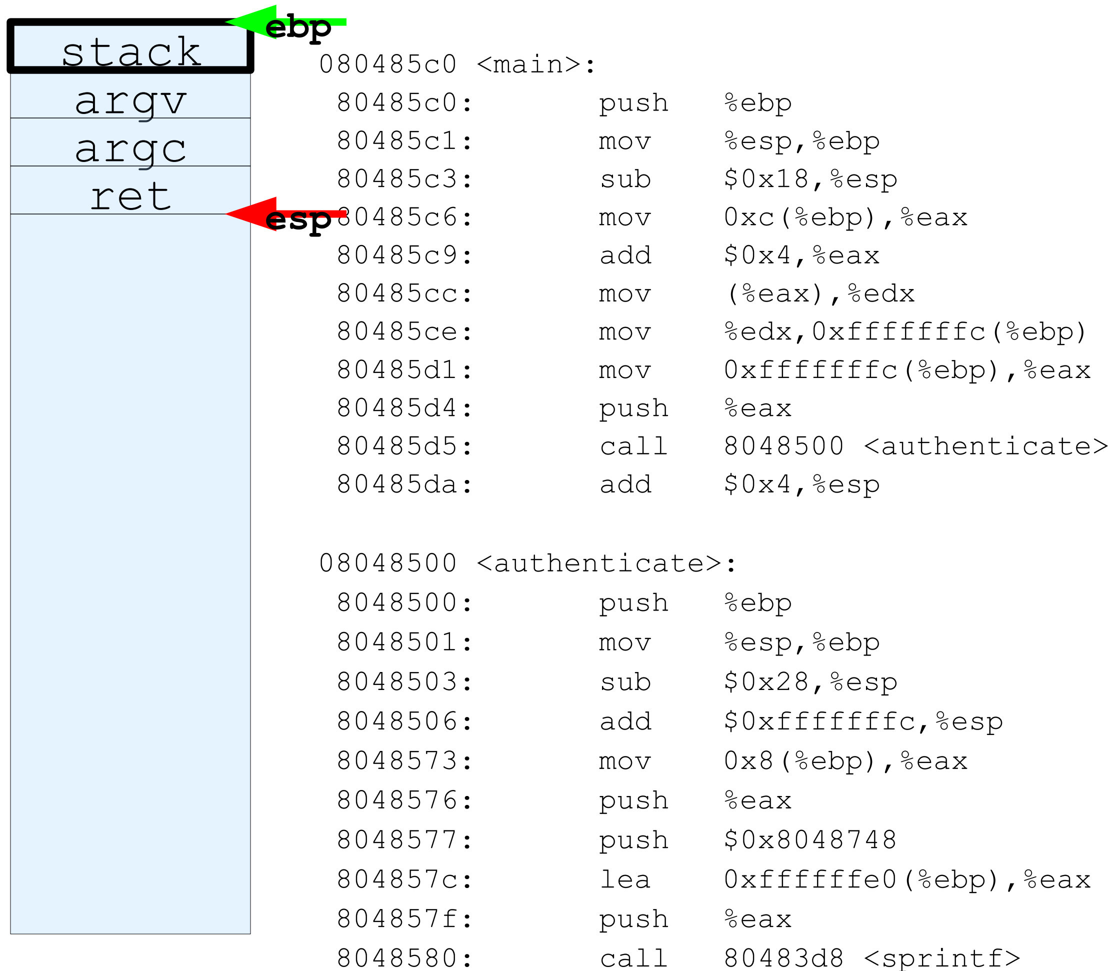


Stack – Function Call


```
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,0xfffffffc(%ebp)
 80485d1:       mov    0xfffffffc(%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    $0xfffffffc,%esp
 8048573:       mov    0x8(%ebp),%eax
 8048576:       push   %eax
 8048577:       push   $0x8048748
 804857c:       lea    0xffffffe0(%ebp),%eax
 804857f:       push   %eax
 8048580:       call   80483d8 <sprintf>
```

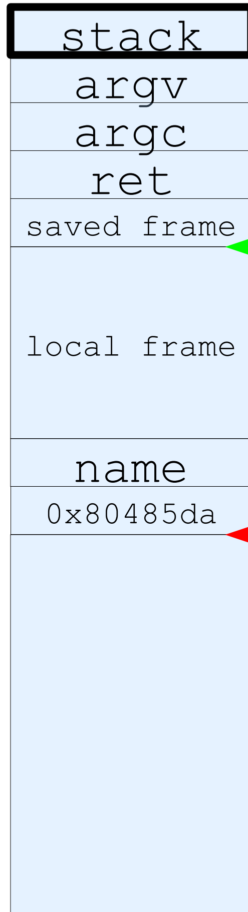
save frame and allocate new one

func params are above frame border

local vars are below frame border

Stack – Function Call

Int. Secure Systems Lab
Vienna University of Technology

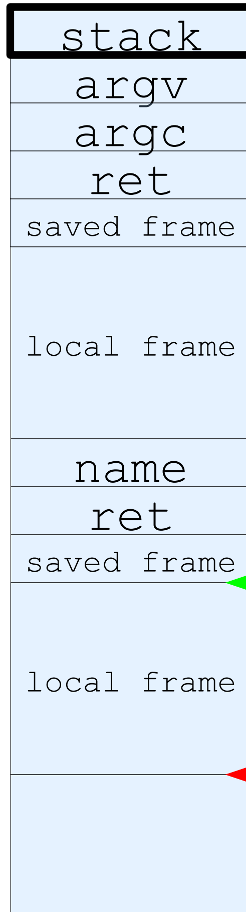



```
080485c0 <main>:
 80485c0:       push   %ebp
 80485c1:       mov    %esp,%ebp
 80485c3:       sub    $0x18,%esp
 80485c6:       mov    0xc(%ebp),%eax
 80485c9:       add    $0x4,%eax
 80485cc:       mov    (%eax),%edx
 80485ce:       mov    %edx,0xfffffffc(%ebp)
 80485d1:       mov    0xfffffffc(%ebp),%eax
 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    $0xfffffffc,%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
remove params from stack

Stack – Function Call

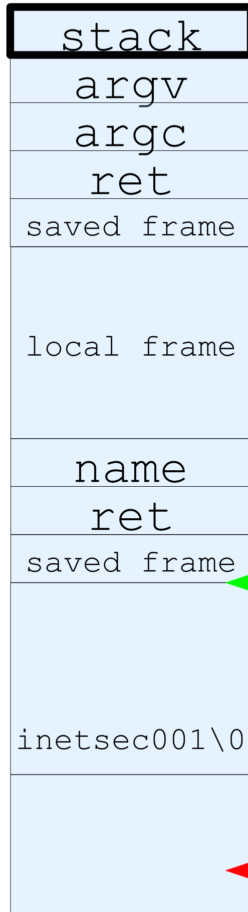



```
080485c0 <main>:
 80485c0:       push   %ebp
 80485c1:       mov    %esp,%ebp
 80485c3:       sub    $0x18,%esp
 80485c6:       mov    0xc(%ebp),%eax
 80485c9:       add    $0x4,%eax
 80485cc:       mov    (%eax),%edx
 80485ce:       mov    %edx,0xfffffffc(%ebp)
 80485d1:       mov    0xfffffffc(%ebp),%eax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

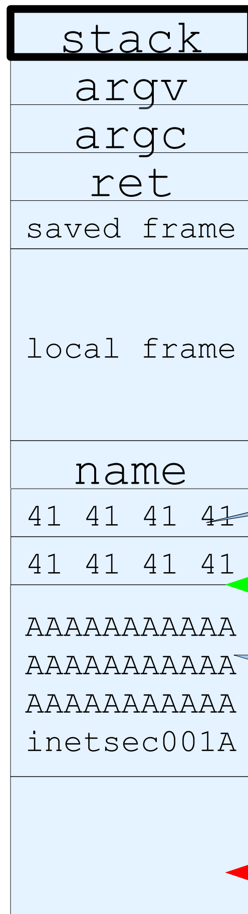



```
080485c0 <main>:
 80485c0:       push   %ebp
 80485c1:       mov    %esp,%ebp
 80485c3:       sub    $0x18,%esp
 80485c6:       mov    0xc(%ebp),%eax
 80485c9:       add    $0x4,%eax
 80485cc:       mov    (%eax),%edx
 80485ce:       mov    %edx,0xfffffffc(%ebp)
 80485d1:       mov    0xfffffffc(%ebp),%eax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

Stack – Function Call




```
080485c0 <main>:
 80485c0:        push   %ebp
 80485c1:
 80485c3:
 80485c6:
 80485c9:
 80485cc:
 80485ce:
 80485d1:
 80485d4:
 80485d5:              8048500 <authenticate>
 80485d                add    $0x4,%esp

08048500 <
 8048500:
 8048501:

 80485
 8048573:
 8048576:        push   %eax
 8048577:        push   $0x8048748
 804857c:        lea    0xffffffe0(%ebp),%eax
 804857f:        push   %eax
 8048580:        call   80483d8 <sprintf>
```

Instead of injecting AAAA (0x41414141) into the return address, the attacker could inject an arbitrary address and force the program to "return" to the given function instead of main!

Instead of injecting "inetsec001AAAA" into the buffer, the attacker could inject arbitrary assembler statements. Thus, all she needs to do is jump to this memory area to be able to execute her malicious code!

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

Buffer Overflow

- Short recap
  - code (or parameters) get injected running process
  - 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
  - normally, this just crashes the program (e.g., `sigsegv`)

Buffer Overflow

- Simple buffer overflow
  1) create executable content, and
  2) set code pointer to point to this content

- Code pointer
  – most often, the *return address* to the calling function
  – alternatives: function pointers or base-pointer modification

- Effect
  – causes a jump to code under our control
  – successfully modifies execution flow
  – have this code executed with privileges of running process

Buffer Overflow

- Advanced buffer overflow
  1) set up function *parameters*, and
  2) set code pointer to point to *existing code*
- Effect
  - causes a jump to existing code with chosen arguments
  - also successfully modifies execution flow, but
  - cannot execute arbitrary code
- Alternative name: *return-into-libc* exploits
- More details in Inetsec2 ;-)

Shell Code

- 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

- System calls
  - mechanism to ask operating system for services
  - transition from user mode to kernel mode
  - different implementations

- Linux system calls are invoked by
  - passing arguments in registers and
  - calling `0x80` interrupt

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`

Shell Code

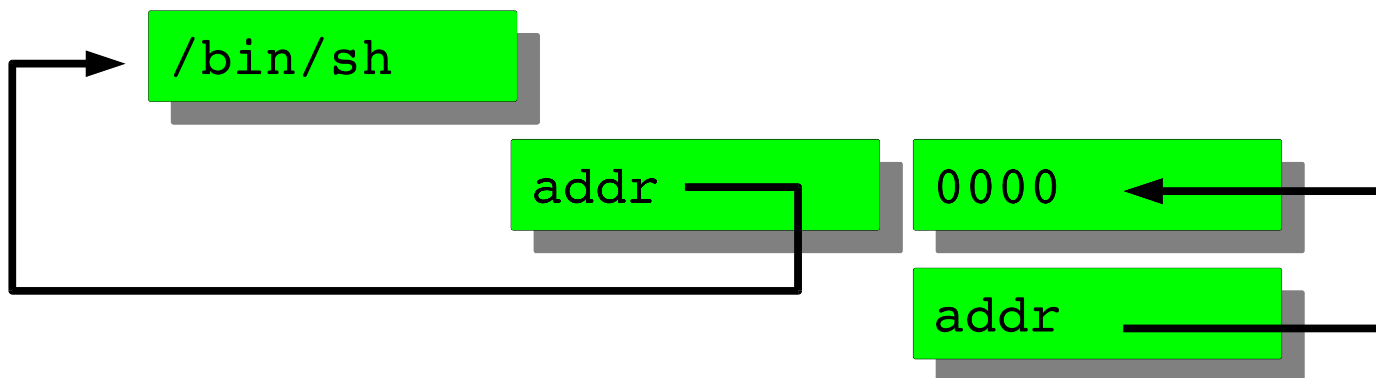
- `file` parameter
  - we need the null terminated string `/bin/sh` somewhere in memory

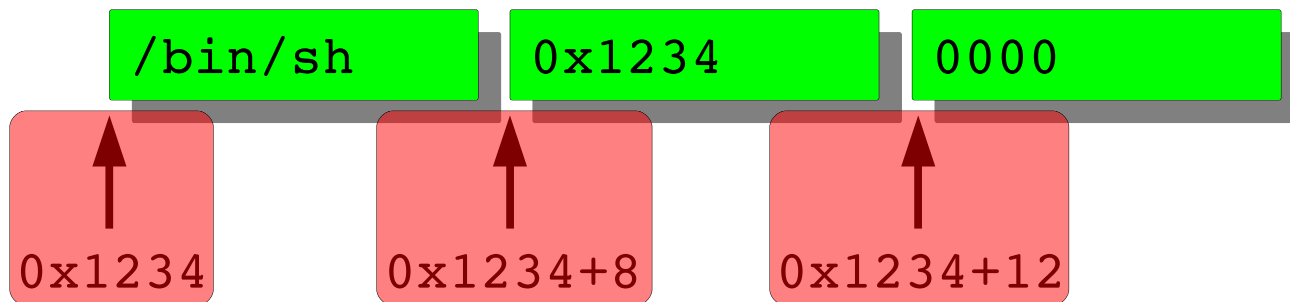
- `argv` parameter
  - we need the address of the string `/bin/sh` somewhere in memory,
  - followed by a NULL word

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

Shell Code


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





Shell Code

- Spawning the shell in assembly

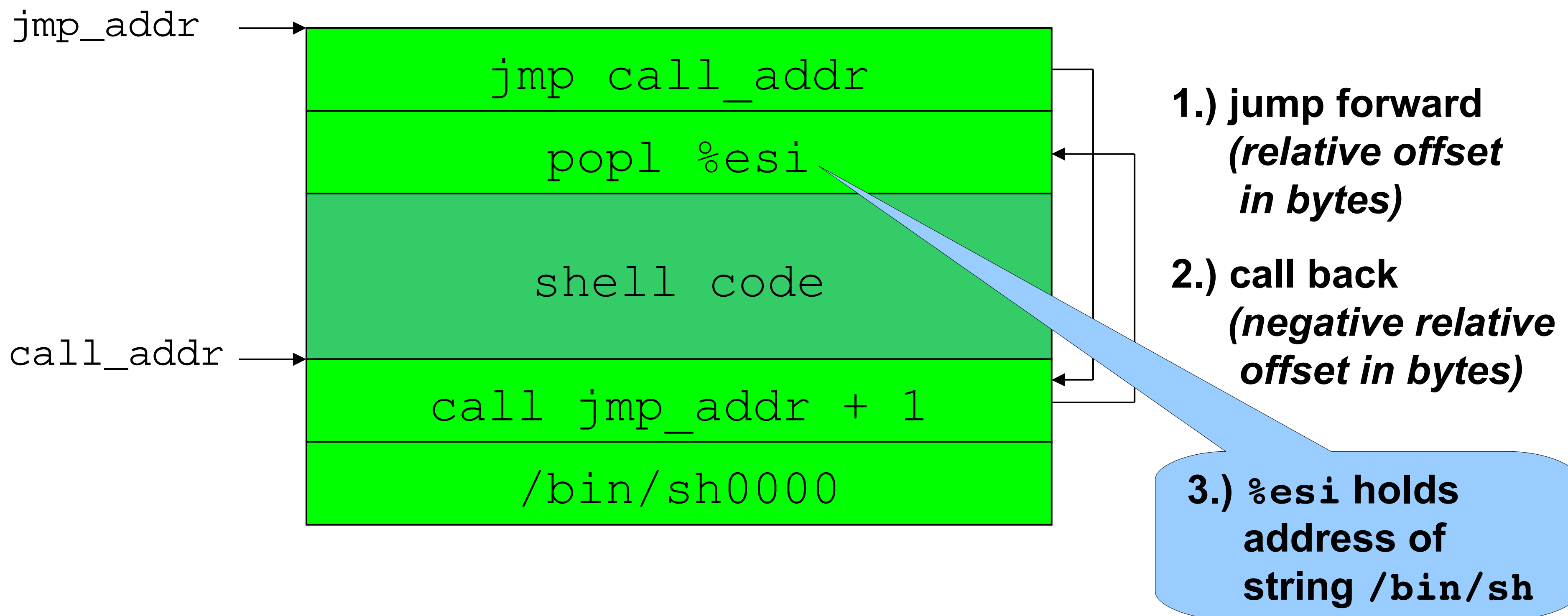
1) move system call number (`0x0b`) into `%eax`
2) move *address* of string `/bin/sh` into `%ebx`
3) move *address of the address* of `/bin/sh` into `%ecx` (using `lea`)
4) move *address* of null word into `%edx`
5) execute the interrupt `0x80` instruction

Shell Code

- Problem – position of shell code in memory is unknown
  - how do we determine the *address of string* ?
- Make use of instructions using *relative* addressing
  - `jmp` and `call` variants for relative and absolute targets
- `call` instruction saves IP of next instruction on the stack and then 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

Int. Secure Systems Lab
Vienna University of Technology



Shell Code

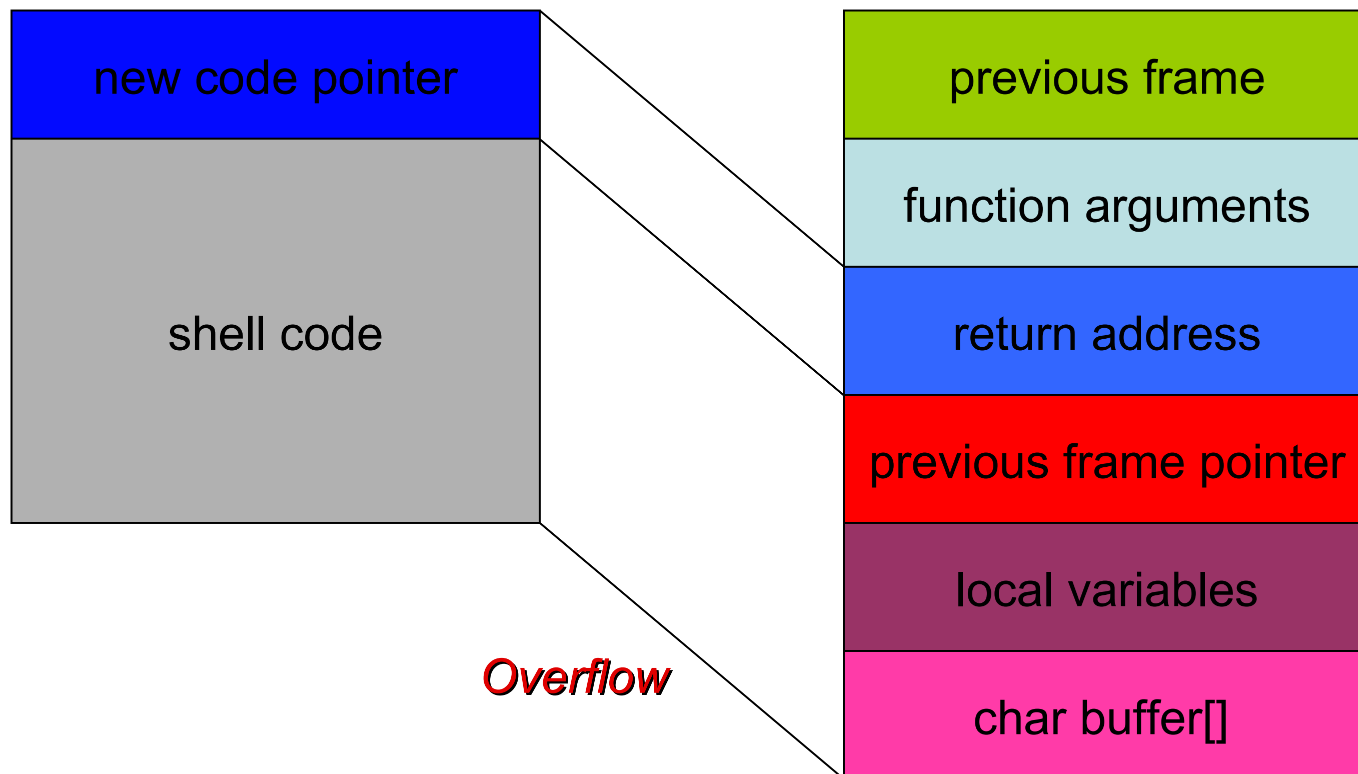
- Shell code is usually copied into a string buffer
- Problem
  - any null byte would stop copying
  - → null bytes must be eliminated

➢ Substitution

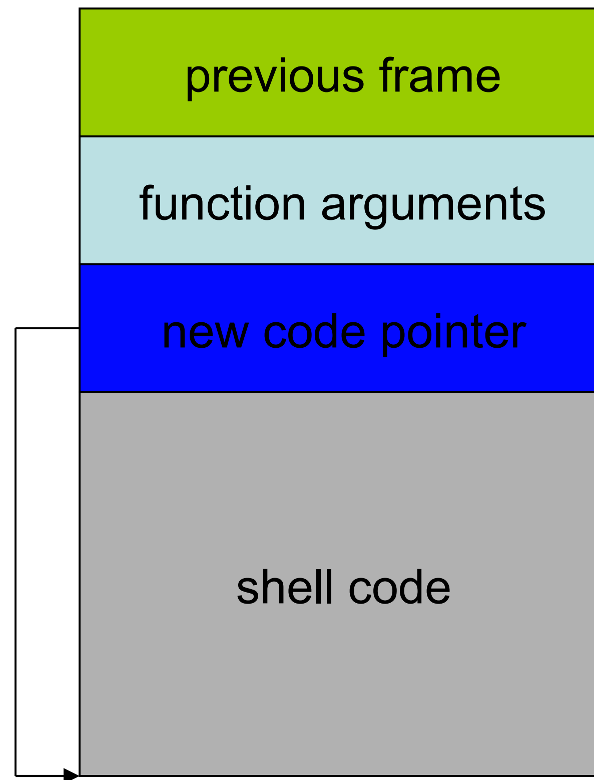
```
      mov 0x0, reg     →     xor reg, reg
      mov 0x1, reg     →     xor reg, reg
                             inc reg

e.g.  movl 0x0, %eax   →     xor %eax, %eax
```

Pulling It All Together



Pulling It All Together



Code Pointer

- Code pointer
  - e.g., return address in stack frame
  - must be overwritten with correct value
  - start of exploit code (`jmp`)
  - it has to be guessed (must be very precise)

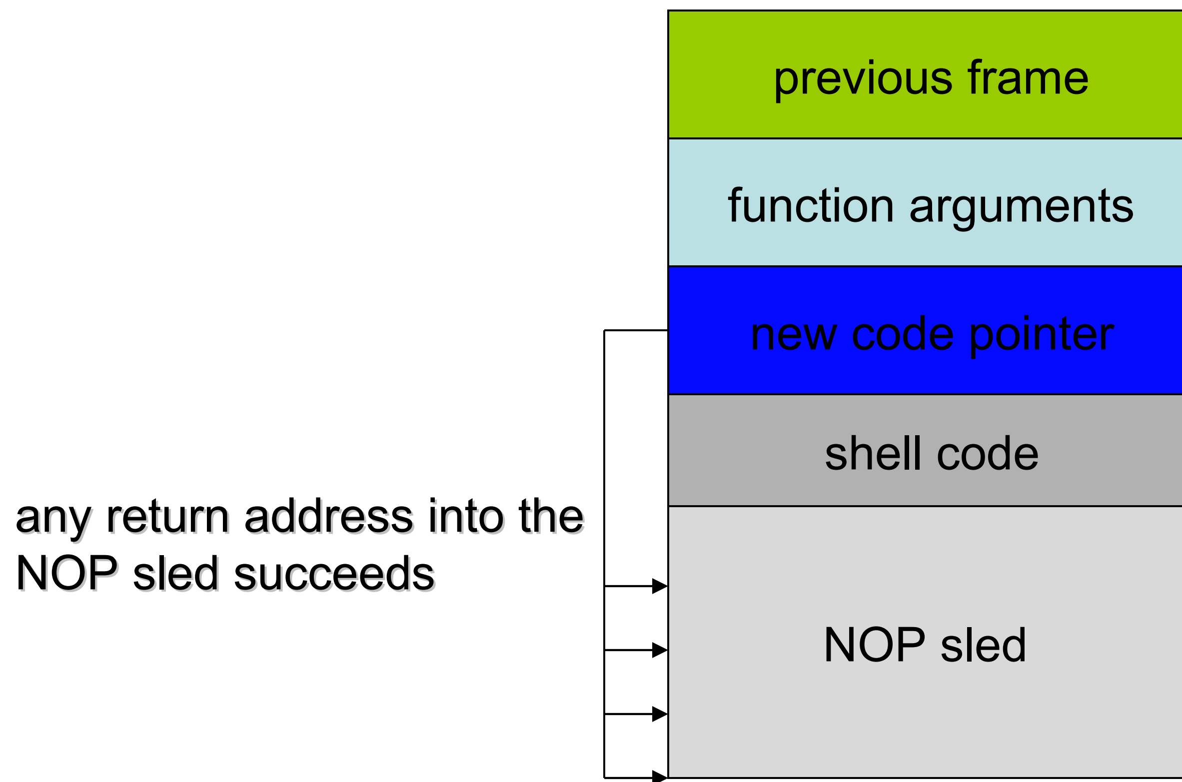
- Hints
  - stack starts at same address for every program (depending on the security settings, OS might explicitly modify this)
  - can be obtained by function

```
unsigned long get_sp(void) {
 __asm__("movl %esp, %eax");
}
```

Code Pointer

- NOP (no operation) sled
  - series of NOP (`0x90`) (no operation) instructions at the beginning of exploit code
  - return address must not be as precise anymore
  - it is enough to hit the NOP sled
  - can also be obfuscated via instruction substitution to make detection more difficult (e.g., `ADMmutate`)

Code Pointer



SetUID 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 (typically only for local exploits)

Articles

- 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)

Conclusion

- Buffer overflows
  - implementation flaw
  - occur when an application receives more input than there is space allocated for this input

- Exploit steps
  - inject shell code or parameters
    - practical issues: locate shell code in memory, NULL bytes, NOP sled
  - change code pointer

- Code pointer
  - various possibilities to change
    - return address, frame pointer, jump buffer, function pointer

Outlook

- Two weeks to prove your 1337ness on buffer overflows
- Next week
  - lecture on language security in 'safe' languages
  - Java
- After that: one week break
- After that: written exam :-)