CSc 466/566

Computer Security

8 : Buffer Overflow I

Version: 2019/09/23 11:14:34

Department of Computer Science
University of Arizona

collberg@gmail.com

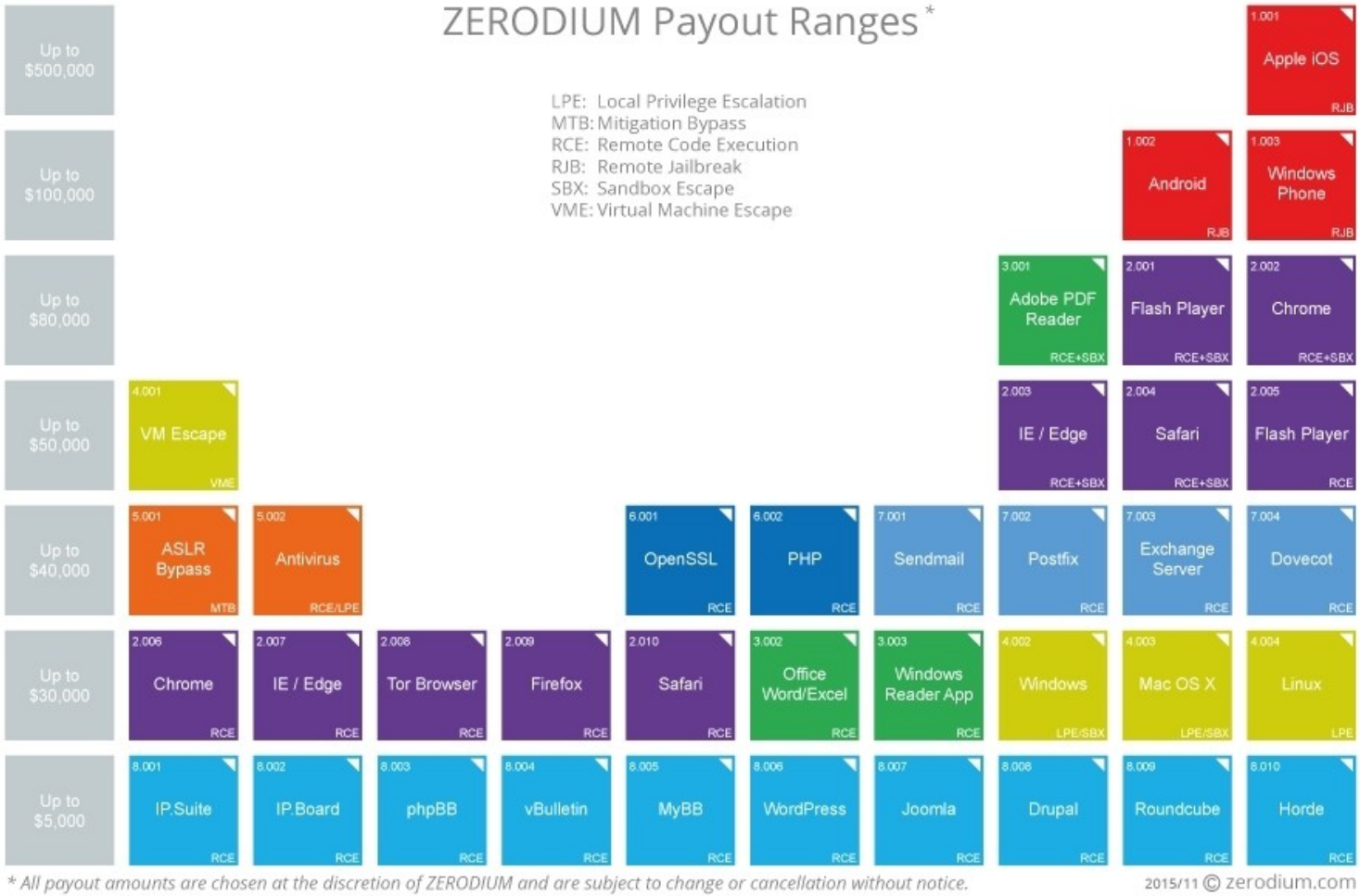
Christian Collberg

Outline

This Lecture

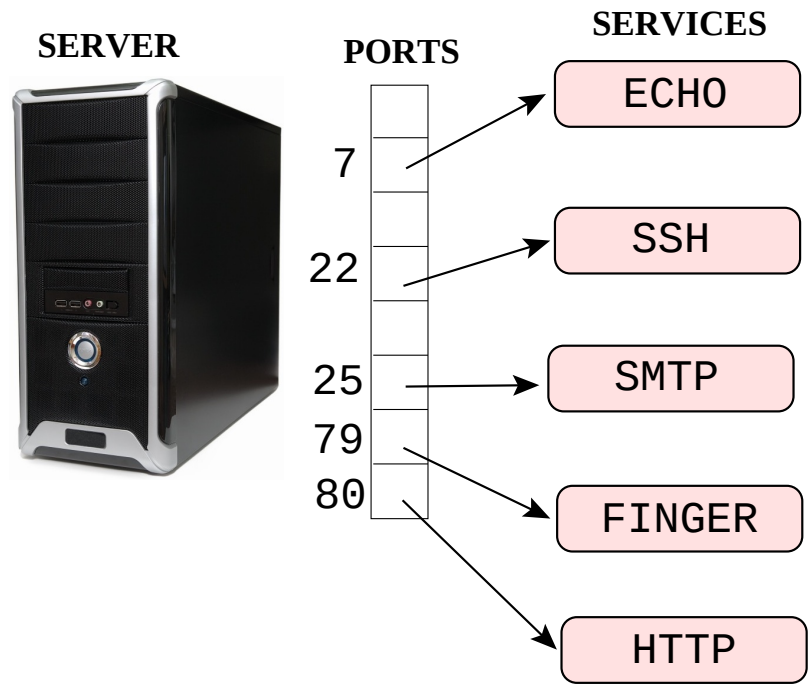
- How do viruses and worms get into a system to do their dirty work?
- In this lecture we are going to examine how certain types of bugs can be exploited to run arbitrary code on a victim computer.
- These bugs occur because important programs are written in languages with weak type systems, i.e. C and C++.
- In a future lecture we will discuss how malware (viruses, worms, trojans) make us of these bugs.

Cost of buying vulnerabilities

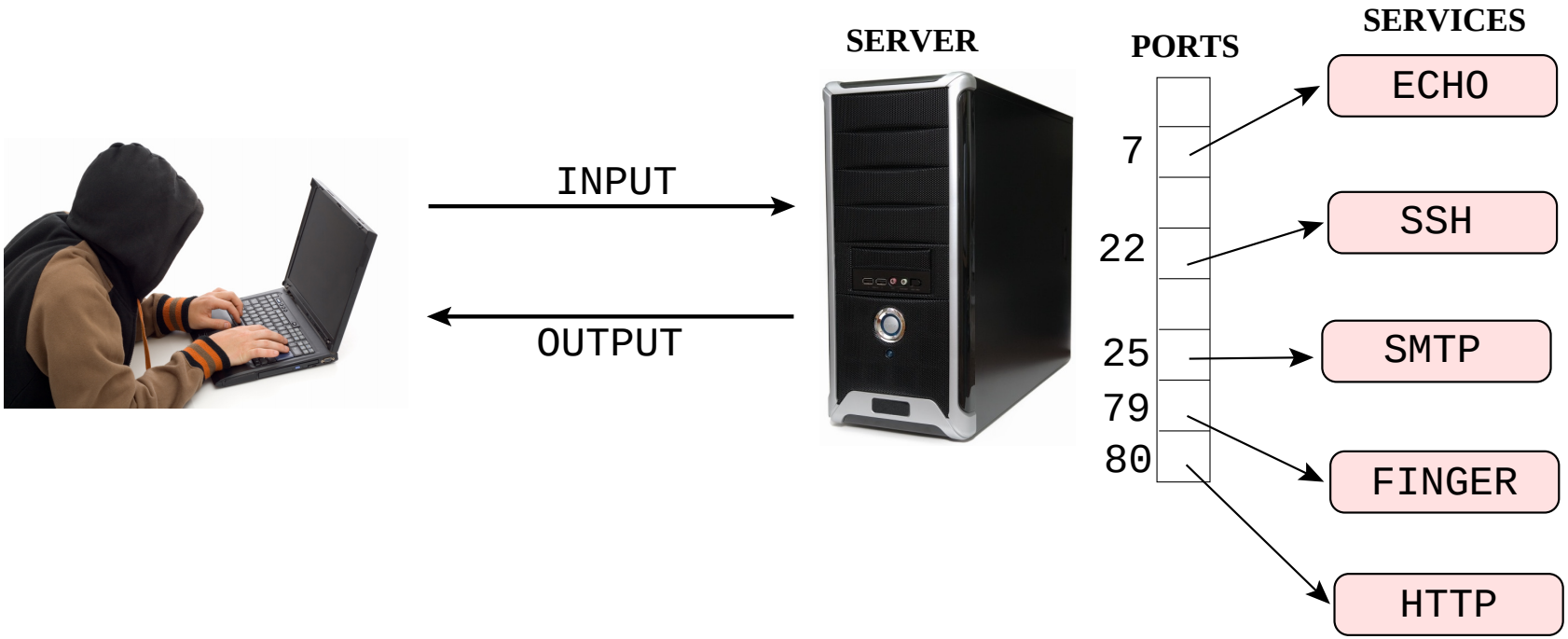


https://news-cdn.softpedia.com/images/news2/exploit-vendor-publishes-price-list-ios-valued-above-android-496449-2.jpg

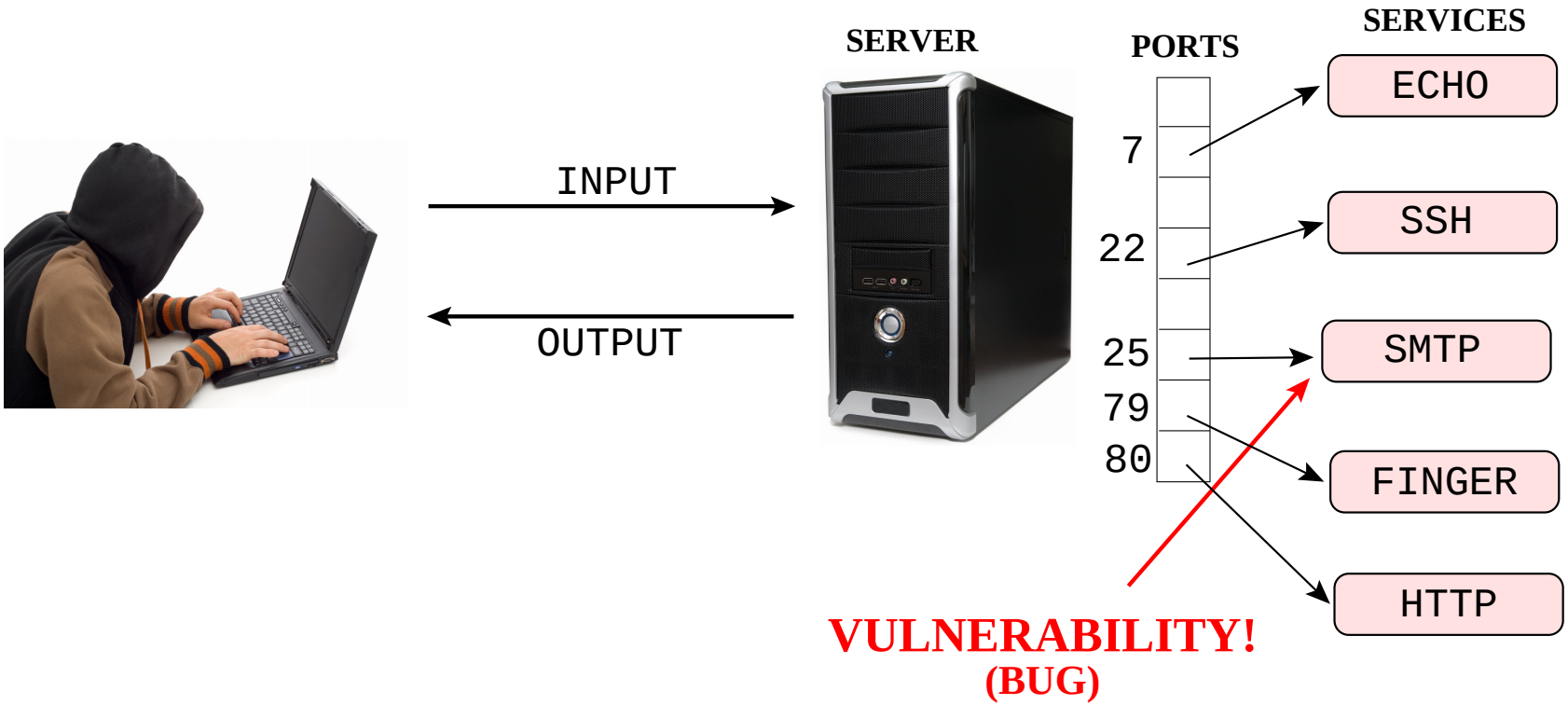
Attacking from the network



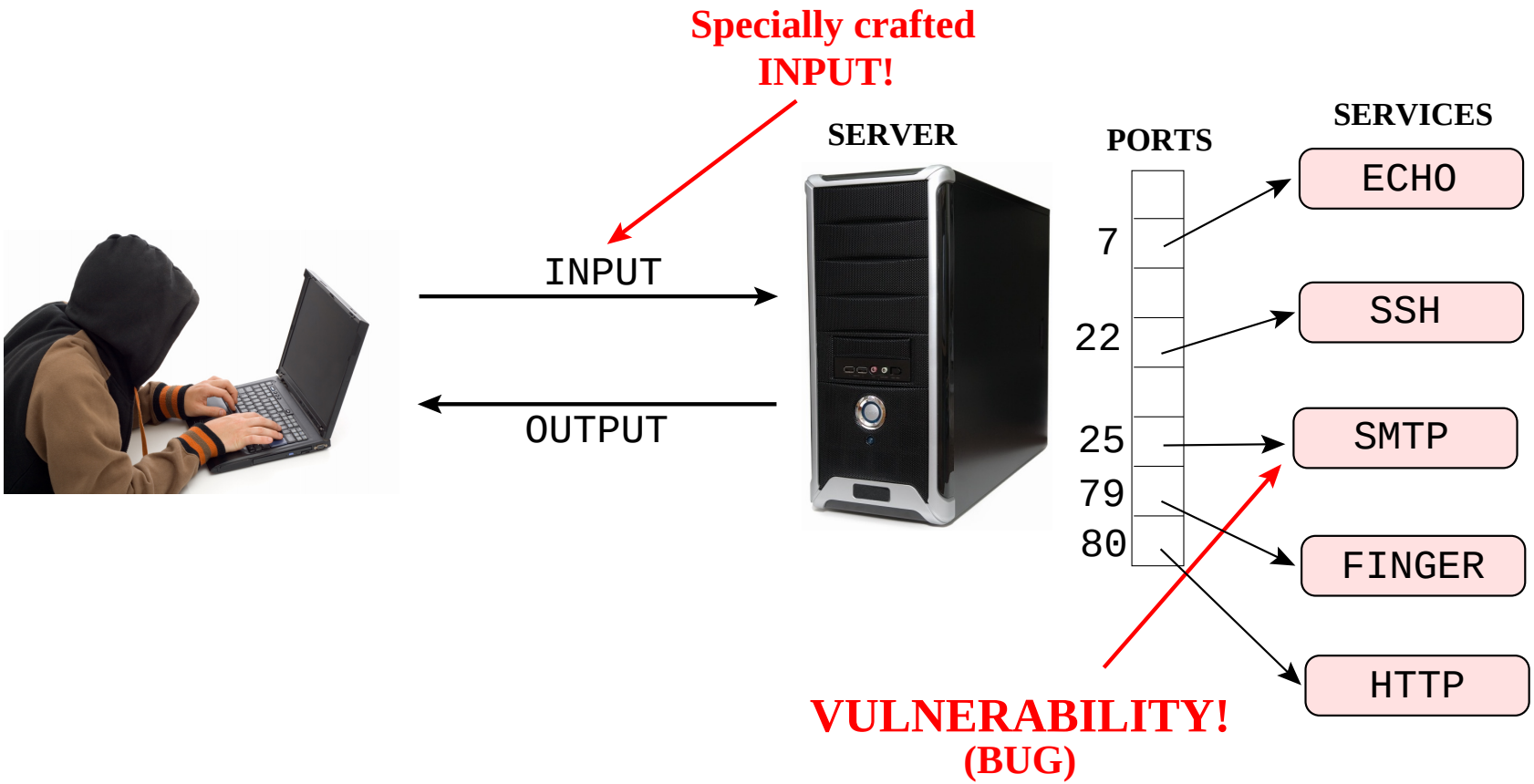
Attacking from the network



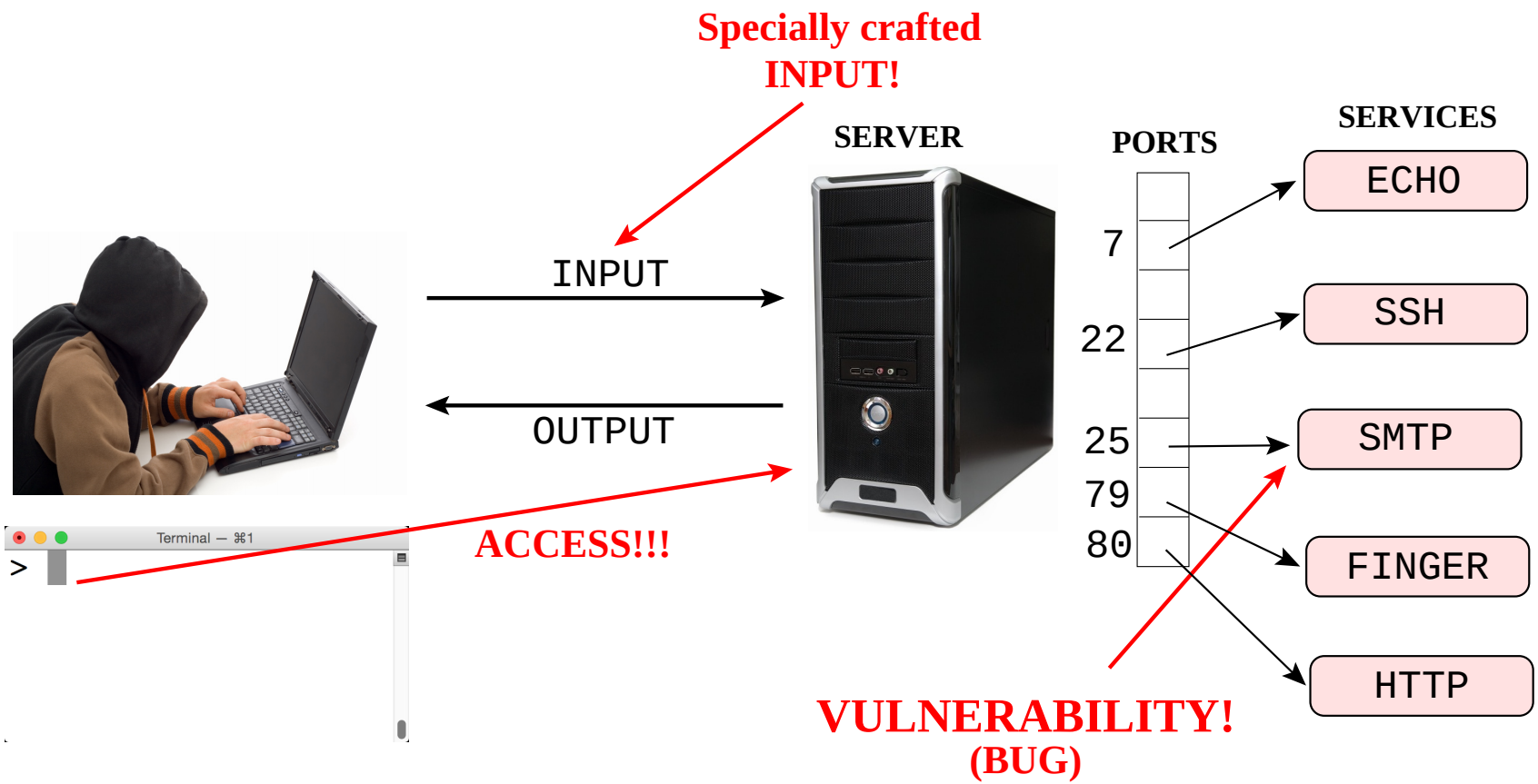
Attacking from the network



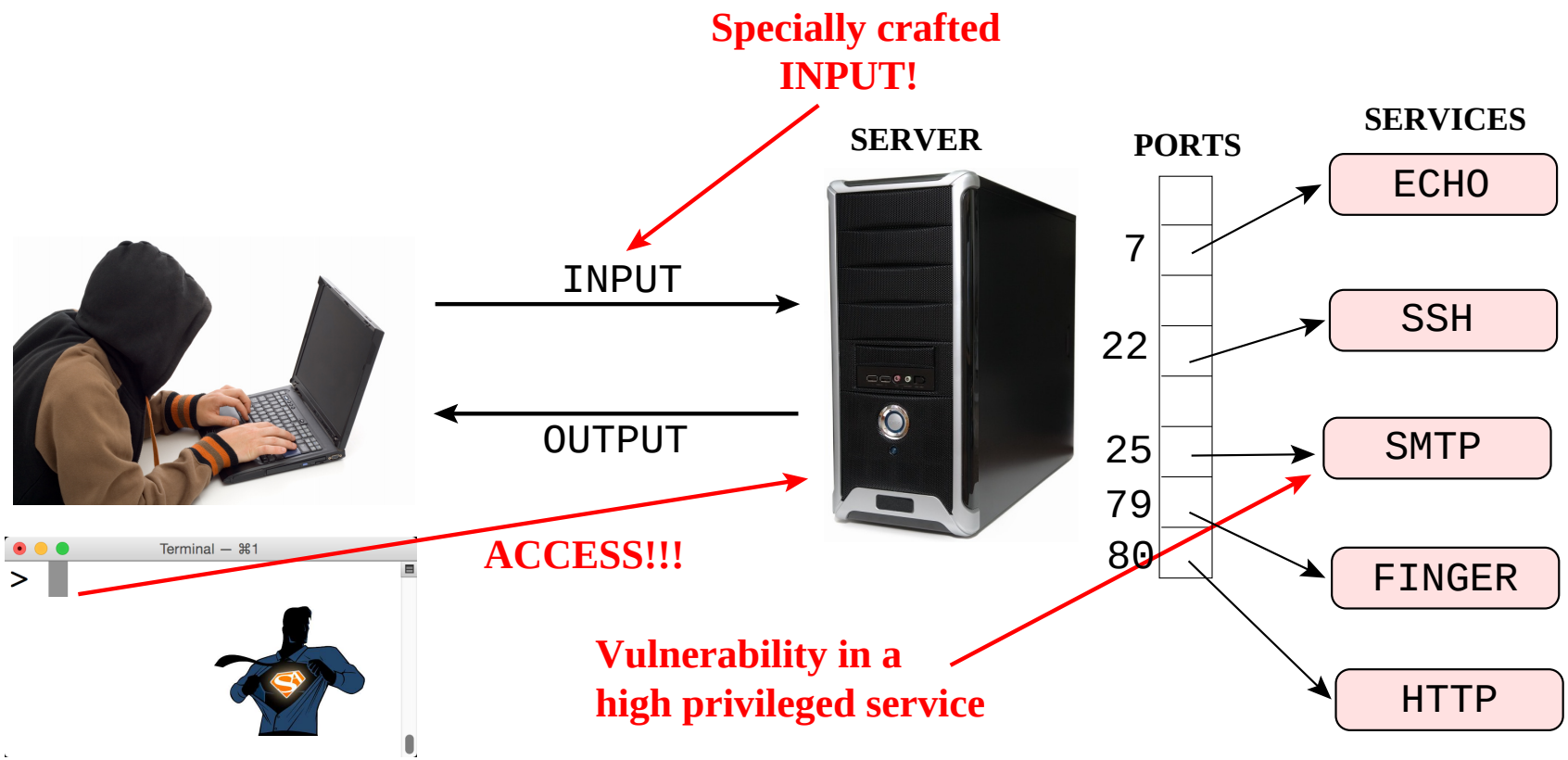
Attacking from the network



Attacking from the network



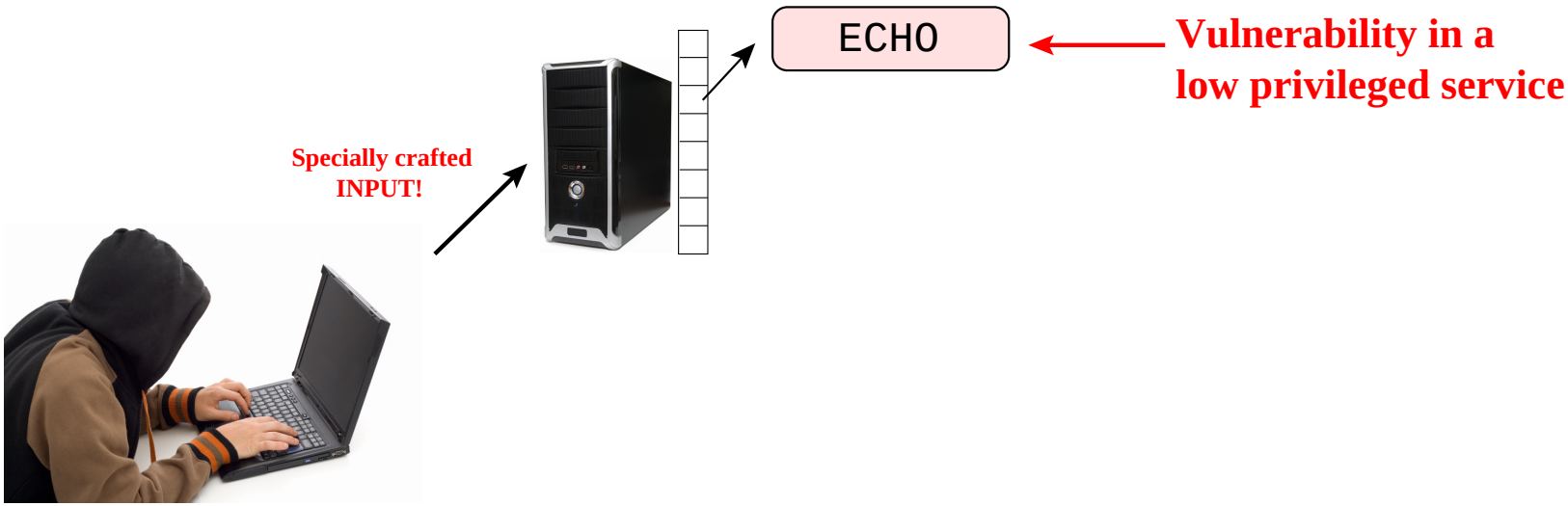
Attacking from the network



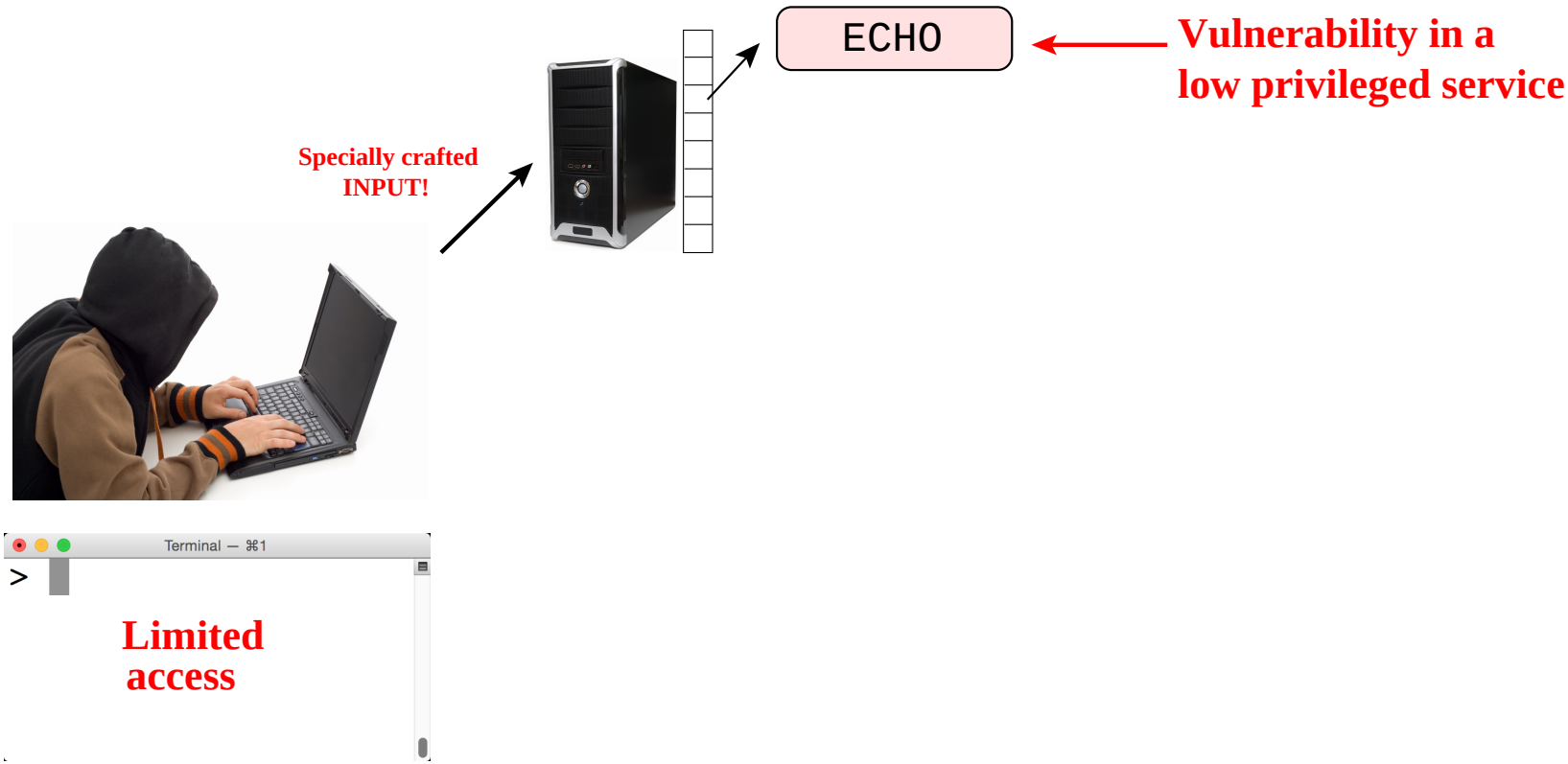
Attacking from the network

- A server provides services, such as an HTTP server to serve up web pages.
- Each of these services are programs, connected to a port. Port 80 accepts HTTP traffic, for example.
- The adversary tries to exploit a vulnerability in one of the services to get access to the machine.
- If the vulnerable service runs with high privilege, say as root, then the adversary who gets access also runs as root!

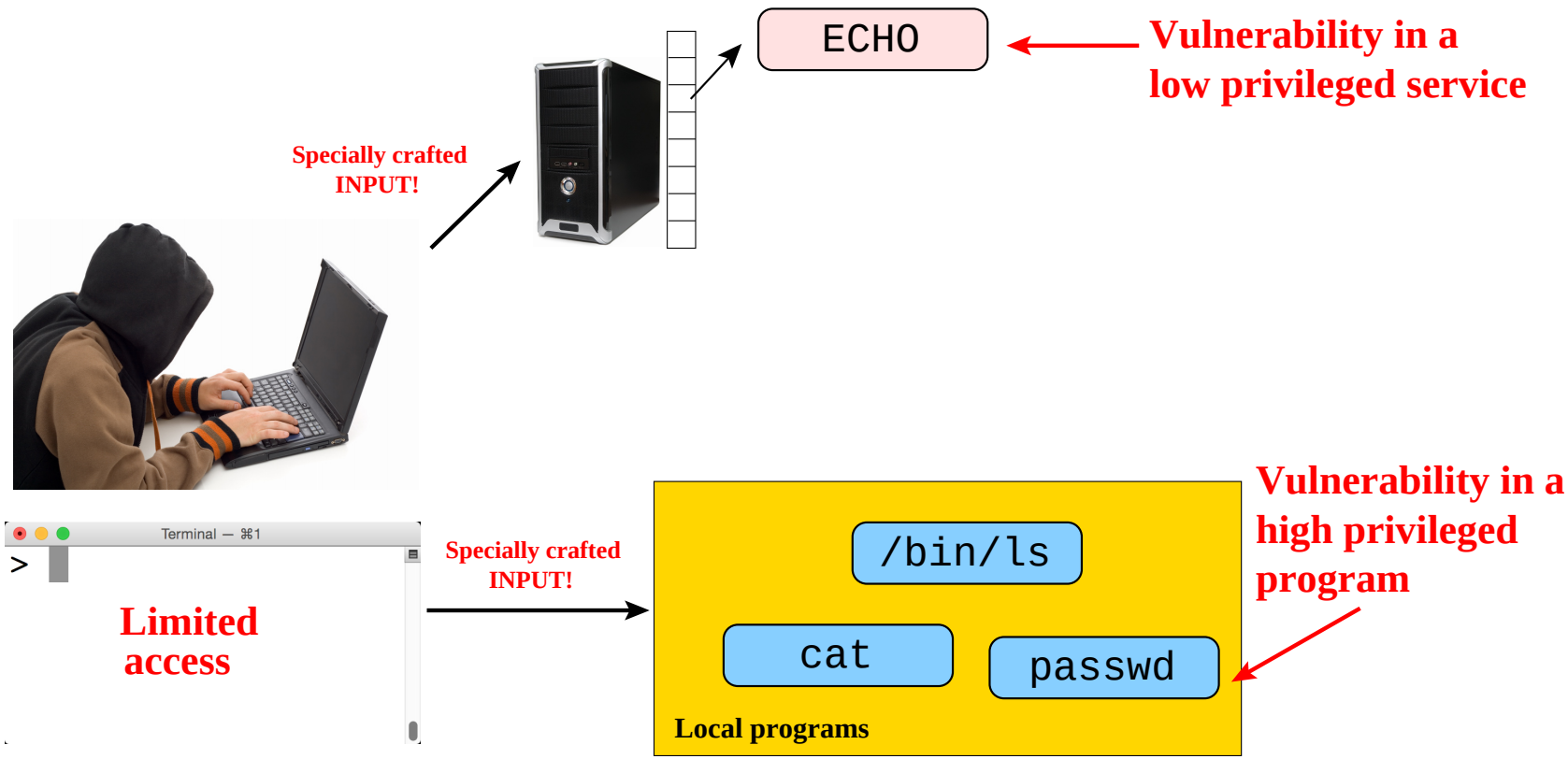
Privilege escalation



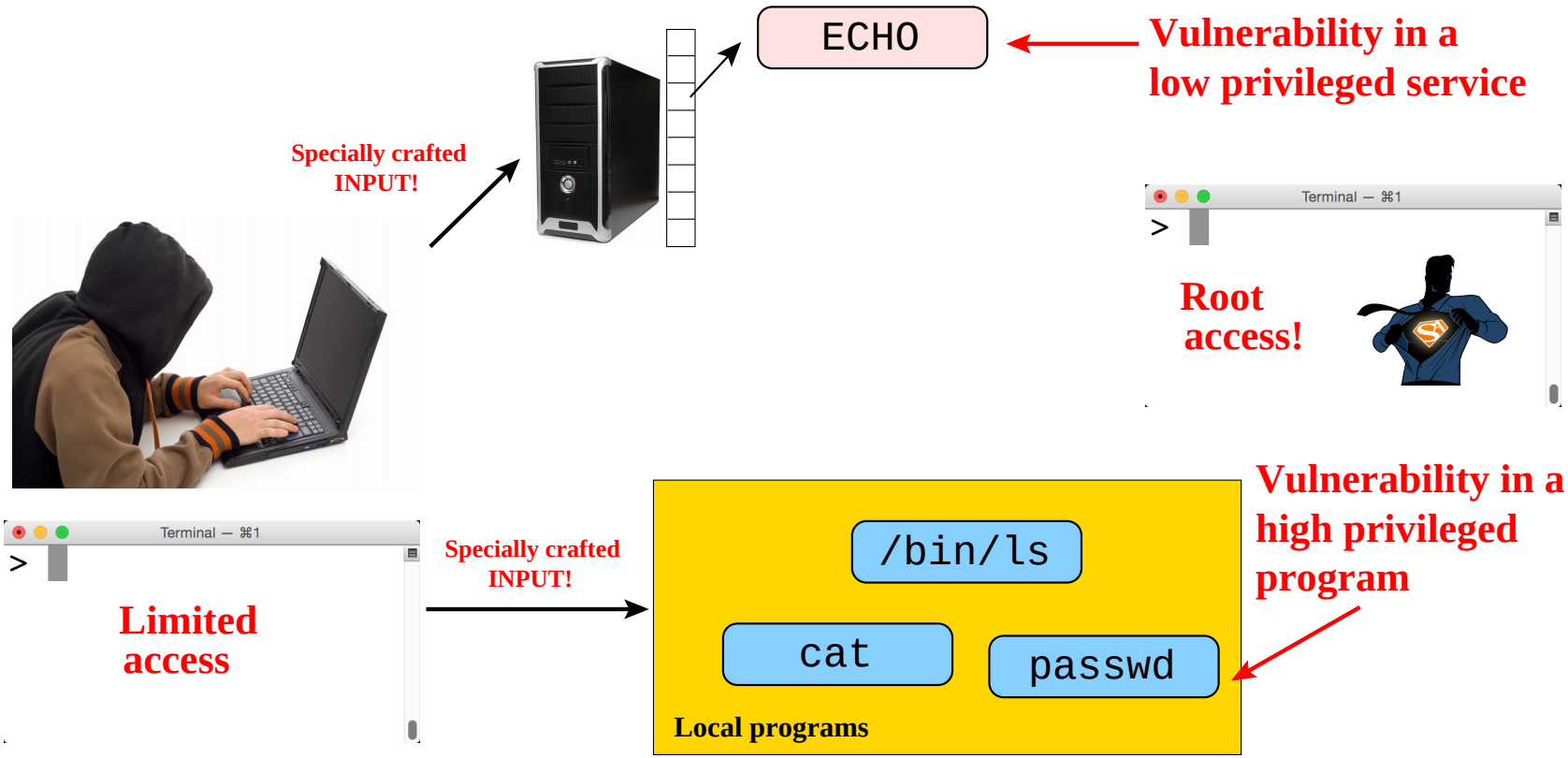
Privilege escalation



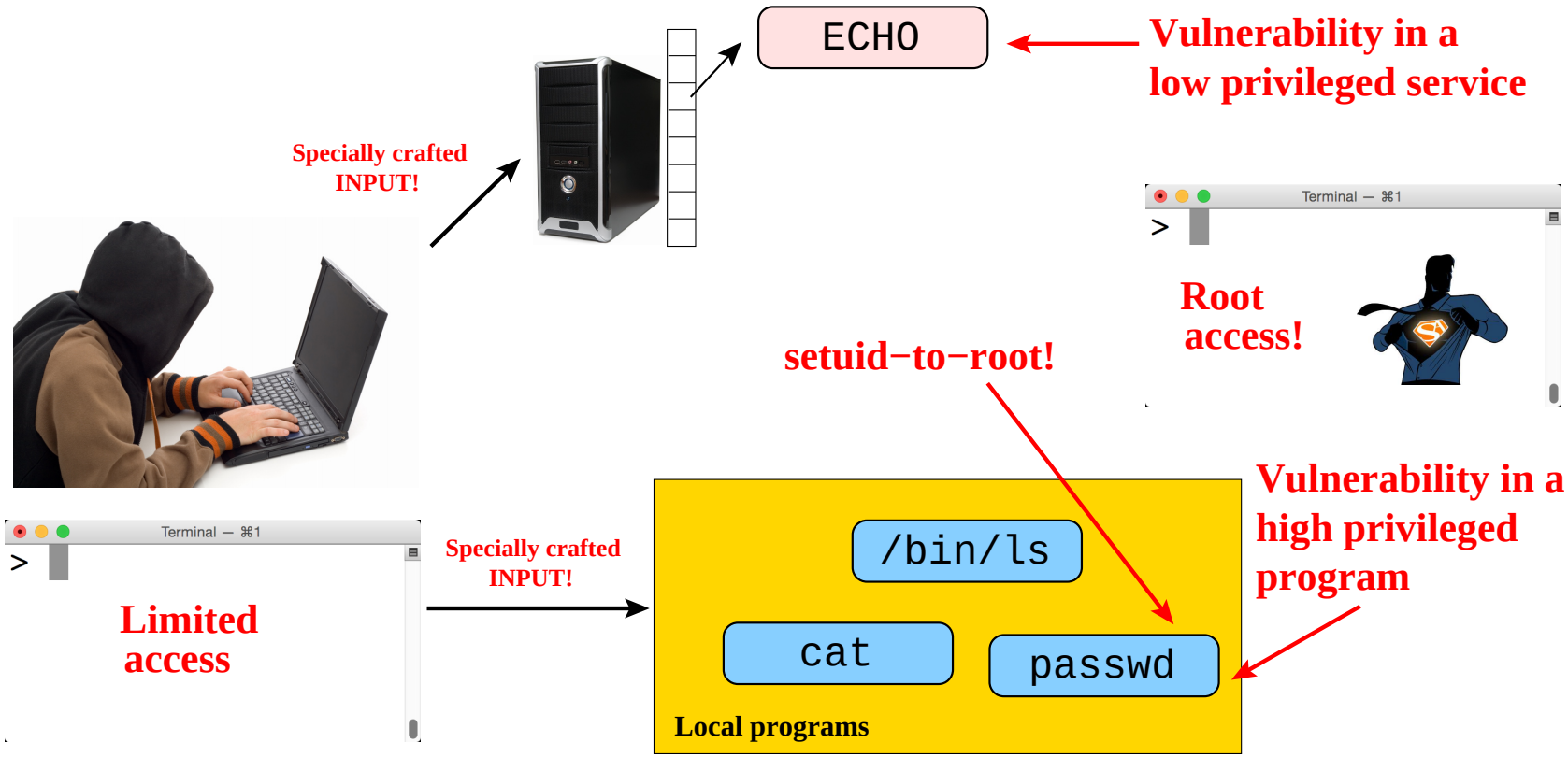
Privilege escalation



Privilege escalation



Privilege escalation



Privilege escalation

Definition (Privilege escalation)

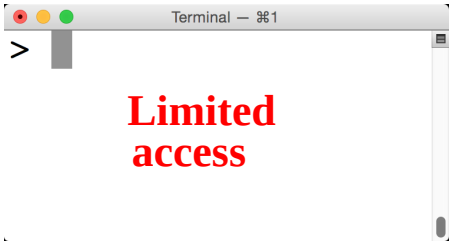
Privilege escalation is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. The result is that an application with more privileges than intended by the application developer or system administrator can perform unauthorized actions.

Wikipedia: https://en.wikipedia.org/wiki/Privilege_escalation

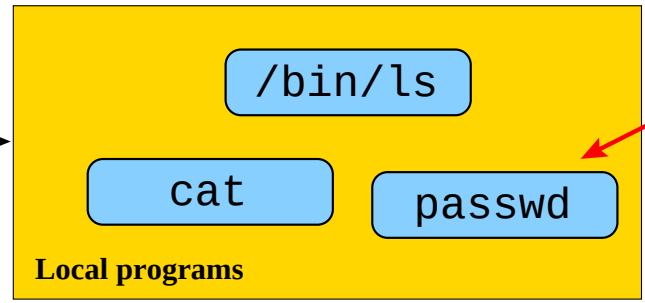
Privilege escalation

- If the service has low privilege, the attacker will only get limited access once he gets in.
- If there are high privilege local programs with vulnerabilities on the attacked machines, then the attacker can exploit *those*, to get higher privilege.
- This is called a privilege escalation attack.

Insider attack



Specially crafted
INPUT!

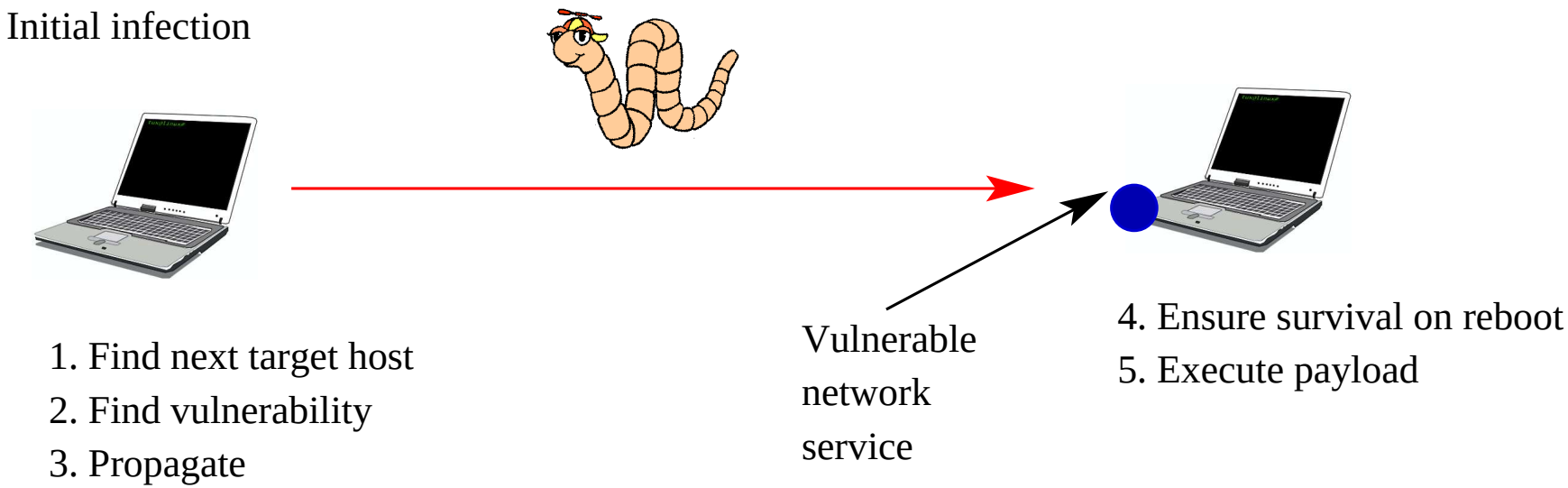


Vulnerability in a
high privileged
program

Insider attack

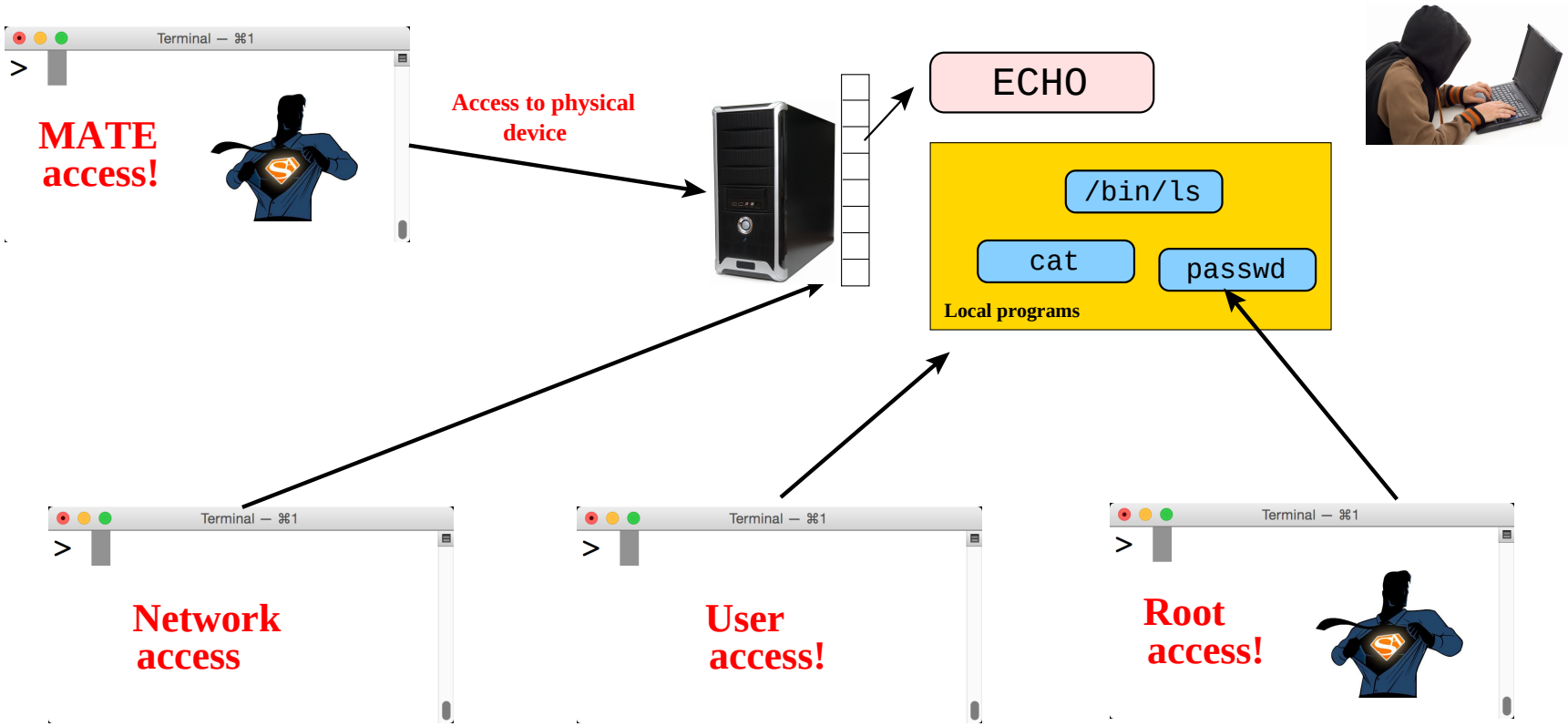
- An insider (someone who already has *some* access to a system) can also exploit a vulnerability in a program on a local machine to escalate his privileges.
- Privilege escalation attacks often target programs with setuid-to-root.

Malware attacks: Worms



- A worm is a malicious program that infects a victim machine.
- It does so by exploiting a vulnerability in a network service exported by the machine.

Levels of access



Levels of access

- An adversary is always looking to increase their privileges.
- From network access to a user account.
- From a user account to a root account.
- From a root account to access to the physical machine (MATE access).

What is a vulnerability?

- A vulnerability is a bug that can be exploited to do harm.
- Programmers tend to avoid
  - checking for error conditions that rarely happen;
  - checking for boundary conditions to save time;
  - checking user input to make sure it's valid.
- How do we avoid writing vulnerable code? We need defensive programming practices!

Exercise: Examples of programmers. . .

- not checking for rare error conditions:

Exercise: Examples of programmers. . .

- not checking for rare error conditions: `if (malloc(24) == NULL) abort()`
- not checking for boundary conditions to save time:

Exercise: Examples of programmers. . .

- not checking for rare error conditions: `if (malloc(24) == NULL) abort()`
- not checking for boundary conditions to save time: `x = x+1; assert (not_overflow(x))`
- not checking that user input is valid.

Exercise: Examples of programmers. . .

- not checking for rare error conditions: `if (malloc(24) == NULL) abort()`
- not checking for boundary conditions to save time: `x = x+1; assert (not_overflow(x))`
- not checking that user input is valid. `read(x); assert(isNumberString(x))`

Exercise: Summary

- What's the difference between a bug and a vulnerability?

Exercise: Summary

- What's the difference between a bug and a vulnerability? A vulnerability is a a bug that can be exploited to do harm
- Explain privilege escalation:

Exercise: Summary

- What's the difference between a bug and a vulnerability? A vulnerability is a a bug that can be exploited to do harm
- Explain privilege escalation: To gain elevated access to resources that are normally protected from an application or user

Outline

Intel x86

Definition (x86)

x86 is a family of backward-compatible instruction set architectures based on the Intel 8086 CPU . . . introduced in 1978 . . . Many additions and extensions have been added . . . over the years, almost consistently with full backward compatibility. . . . Intel, AMD, and VIA . . . are producing modern 64-bit designs. https://en.wikipedia.org/wiki/X86

The x86 architecture really isn't that complicated, it just doesn't make any sense. – Anonymous

https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf

CISC vs. RISC

- x86 is a CISC architecture: lots of complex instructions, as opposed to MIPS (a RISC architecture) which has few, simple, instructions.
- In MIPS and other RISCs, every instruction is 4 bytes.
- n x86, instrucions are variable length, from 1 to 15 bytes long.

Intel x86 — Registers

- Registers are named `%eax`, `%ecx`, ...
- Register `%esp` holds SP (the stack pointer).
- Register `%ebp` holds FP (the frame pointer).

Intel x86 — Addressing Modes

- The addressing mode *offset(register)* means *the memory cell pointed to by register*.
- The addressing mode *offset1(reg1,reg2,offset2)* means *the memory cell pointed to by (offset1 + reg1 + (reg2*offset2)*. It is useful to index into arrays.

Intel x86 — Instructions Examples

Instruction	Meaning
`movl` *src*, *dst*	*dst* ← *src*
`subl` *src*, *dst*	*dst* ← *dst* - *src*
`addl` *src*, *dst*	*dst* ← *dst* + *src*
`push` *src*	*%esp* ← *%esp* - 4 `movl` *src*, (*%esp*)
`pop` *dst*	`movl` (*%esp*), *dst* *%esp* ← *%esp* + 4
`cdq`	all bits of %edx ← sign bit of %eax
`int` *num*	generate a software interrupt # *num*

Gcc — Compiling to Assembly Code

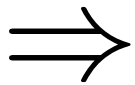
- Use the `-S` option to generate assembly code:

```
> gcc -m32 -O0 -g -S -o layout1.s layout1.c
```

- This is useful when you want to know what code the compiler emits for a particular C construct!

x86 Example 1 – Output from gcc -S

```
int main (){
    int x;
    x++;
}
```



```
_main:
    pushq %rbp
    movq  %rsp, %rbp
    xorl  %eax, %eax
    movl  -4(%rbp), %ecx
    addl  $1, %ecx
    movl  %ecx, -4(%rbp)
    popq  %rbp
    retq
```

- `gcc -S example1.c -o example1.s`
- Note how the `addl` instruction takes 2 arguments, not 3 as on the MIPS.

x86 Example – Output from objdump

```
100000fa0: 55           push   %rbp
100000fa1: 48 89 e5     mov    %rsp,%rbp
100000fa4: 31 c0        xor    %eax,%eax
100000fa6: 8b 4d fc     mov    -0x4(%rbp),%ecx
100000fa9: 83 c1 01     add    $0x1,%ecx
100000fac: 89 4d fc     mov    %ecx,-0x4(%rbp)
100000faf: 5d           pop    %rbp
100000fb0: c3           retq
```

- `gcc example1.c -o example1; objdump -d example1`
- Note how some instructions are only 1 byte (`retq`) but some are 3 (`addl`).

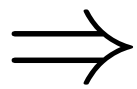
X86 Example - Shellcode

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- The goal of a buffer overflow attack is to launch a shell.
- Ideally, we want a root shell so we can completely own the machine.
- So, we want to trick a vulnerable program to execute the code above, the shellcode.

X86 Example - Running the Shellcode

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0],name,NULL
        );
}
```



```
> gcc shellcode.c
> a.out
sh$ echo yay
yay
```

- Of course, we can't give someone the source code above and expect them to compile and run it for us!
- Instead, we have to take the binary instruction bytes of shellcode.c and try to force a vulnerable program to execute them!

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  "\x99"          /* Line 9: cdq                 */
  "\xb0\x0b"      /* Line A: movb    $0x0b,%al   */
  "\xcd\x80"      /* Line B: int     $0x80       */;
```

- Note that instructions are of different lengths.

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  "\x99"          /* Line 9: cdq                 */
  "\xb0\x0b"      /* Line A: movb    $0x0b,%al   */
  "\xcd\x80"      /* Line B: int     $0x80       */;

int main(int argc, char **argv){
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

- This code does the same as the code in `shellcode.c` above.

X86 Example - Compiling

```
gcc -m32 -z execstack -o call_shellcode
    call_shellcode.c
```

- The code in the previous slide copies the x86 instructions (the `code` array) onto the stack.
- It then jumps to the beginning of the code (on the stack).
- Normally, you can't execute code on the stack, so we have to compile with the `-z execstack` flag.
- `-m32` compiles for a 32-bit architecture.

Execve system call

```
#include <unistd.h>
int execve(const char *path,
           char *const argv[],
           char *const envp[]);

DESCRIPTION
  Execve() transforms the calling process into a
  new process. ... constructed from an ordinary
  file, whose name is pointed to by path,...

  argv is an array of argument strings passed
  to the new program, the first of  these
  strings should contain the filename
  associated with the file being executed.

  envp is an array of strings, ... passed as
  environment to the new program.
```

Execve: System call number

```
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
```

Execve: Creating a shell

```
int execve(const char *path,
           char *const argv[],
           char *const envp[]);
```

- To set up a system call to create a shell, we need to do the following:
  1. %ebx (*path*) should hold the address of "/bin/sh" followed by a NULL byte.
  2. %ecx (*argv*) should hold the address of "/bin/sh" followed by 0x0000000.
  3. %edx (*envp*) should hold 0x0000000.
- Set register %al to 11, the system call number for execve.
- Issue the instruction int 0x80: this makes the system call.

Exercise: Explain Shellcode (Lines 1-2)

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  ...
```

- Line 1:

Exercise: Explain Shellcode (Lines 1-2)

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  ...
```

- Line 1: Zeros out register %eax
- Line 2:

Exercise: Explain Shellcode (Lines 1-2)

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  ...
```

- Line 1: Zeros out register %eax
- Line 2: Pushes 0x00000000

Exercise: Explain Shellcode (Lines 3-4)

```
const char code[] =
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  ...
```

- Line 3:

Exercise: Explain Shellcode (Lines 3-4)

```
const char code[] =
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  ...
```

- Line 3: Pushes the 4 chars "\\sh"
- Line 4:

Exercise: Explain Shellcode (Lines 3-4)

```
const char code[] =
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  ...
```

- Line 3: Pushes the 4 chars "\\sh"
- Line 4: Pushes the 4 chars "\bin"

Exercise: Explain Shellcode (Lines 1-4)

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  ...
```

- Lines 1-4:

Exercise: Explain Shellcode (Lines 1-4)

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  ...
```

- Lines 1-4: Pushes "\bin\\"sh followed by 0x00000000

Exercise: Explain Shellcode (Lines 5-8)

```
const char code[] =
  ...
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  ...
```

- Line 5:

Exercise: Explain Shellcode (Lines 5-8)

```
const char code[] =
  ...
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  ...
```

- Line 5: Moves contents of stack pointer to %ebx, i.e. %ebx (*path*) now holds the address of "/bin//sh" followed by 0x00
- Line 8:

Exercise: Explain Shellcode (Lines 5-8)

```
const char code[] =
  ...
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  ...
```

- Line 5: Moves contents of stack pointer to %ebx, i.e. %ebx (*path*) now holds the address of "/bin//sh" followed by 0x00
- Line 8: Moves contents of stack pointer to %ecx, i.e. %ecx (*argv*) now holds the address of "/bin//sh" followed by 0x00000000

Exercise: Explain Shellcode (Lines 9, A, B)

```
const char code[] =
  ...
  "\x99"          /* Line 9: cdq                  */
  "\xb0\x0b"      /* Line A: movb   $0x0b,%al     */
  "\xcd\x80"      /* Line B: int    $0x80         */;
```

- Line 9:

Exercise: Explain Shellcode (Lines 9, A, B)

```
const char code[] =
  ...
  "\x99"          /* Line 9: cdq                    */
  "\xb0\x0b"      /* Line A: movb    $0x0b,%al      */
  "\xcd\x80"      /* Line B: int     $0x80          */;
```

- Line 9: cdq copies the sign of %eax into every bit position of %edx. Since %eax=0, line 9 sets %edx to 0!
- Lines A-B:

Exercise: Explain Shellcode (Lines 9, A, B)

```
const char code[] =
  ...
  "\x99"          /* Line 9: cdq                    */
  "\xb0\x0b"      /* Line A: movb    $0x0b,%al      */
  "\xcd\x80"      /* Line B: int     $0x80          */;
```

- Line 9: cdq copies the sign of %eax into every bit position of %edx. Since %eax=0, line 9 sets %edx to 0!
- Lines A-B: Invokes system call #11, i.e. execve.

Exercise: Explain Shellcode

```
const char code[] =
  "\x31\xc0"      /* Line 1: xorl    %eax,%eax   */
  "\x50"          /* Line 2: pushl   %eax        */
  "\x68""//sh"    /* Line 3: pushl   $0x68732f2f */
  "\x68""/bin"    /* Line 4: pushl   $0x6e69622f */
  "\x89\xe3"      /* Line 5: movl    %esp,%ebx   */
  "\x50"          /* Line 6: pushl   %eax        */
  "\x53"          /* Line 7: pushl   %ebx        */
  "\x89\xe1"      /* Line 8: movl    %esp,%ecx   */
  "\x99"          /* Line 9: cdq                 */
  "\xb0\x0b"      /* Line A: movb    $0x0b,%al   */
  "\xcd\x80"      /* Line B: int     $0x80       */;
int main(int argc, char **argv){
   char buf[sizeof(code)]; strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Now explain the whole thing!

Outline

Stack layout

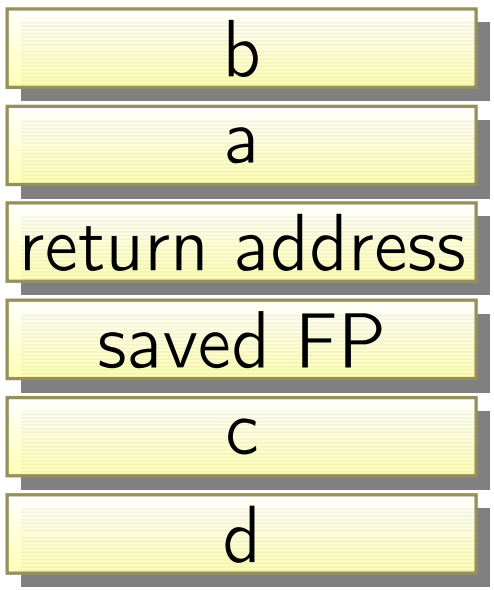
- The execution stack of a program (on an x86 machine) grows downward (to lower memory addresses) as procedures are called.
- Information is placed in stack frames.
- Among the things stored on the stack are
  - the local and formal variables,
  - the return address, and
  - the frame pointer of the procedure.
- The positions of these values in memory are shown on the next slide.

Stack layout...

- The Stack Pointer (SP) points to the top of the stack.
- The Frame Pointer (FP) points in the middle of the current stack frame.
- The frame pointer is used to reference arguments and local variables.

Stack layout...

```
void foo(int a, int b){
    int c;
    int d;
}
```



High addresses, bottom of stack

Low addresses, top of stack

Example I

```
void foo(int a, int b){
    int c;
    int d;
}
```

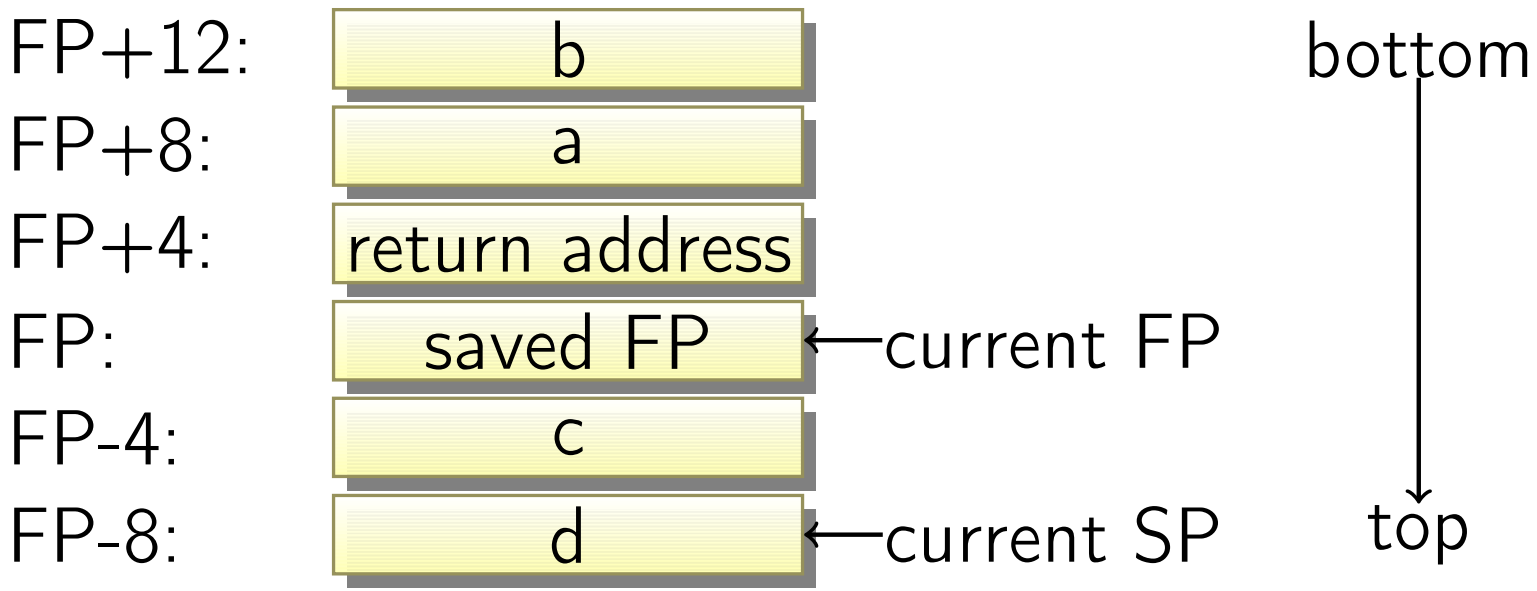
```
foo:
    pushl %ebp          # push frame pointer (FP)
    movl  %esp, %ebp    # set FP to stack-top (SP)
    subl  $16, %esp     # create space for locals
        ....
    addl $16, %esp      # remove space for locals
    popl %ebp           # reset FP
    retl                # return
```

- Here is function `main` calling `foo`:

```
int main(){
    foo(6,7);
}
```

```
main:
    pushl %ebp          # push frame pointer (FP)
    movl  %esp, %ebp    # set FP to stack-top (SP)
    subl  $24, %esp     # create space for locals
    movl  $6, (%esp)    # push 6
    movl  $7, 4(%esp)   # push 7
    calll foo           # push return address
                        # then jump to foo
    addl  $24, %esp     # remove space for locals
    popl  %ebp          # reset FP
    retl                # return
```

```
void foo(int a, int b){
    int c;
    int d;
}
```



Example II

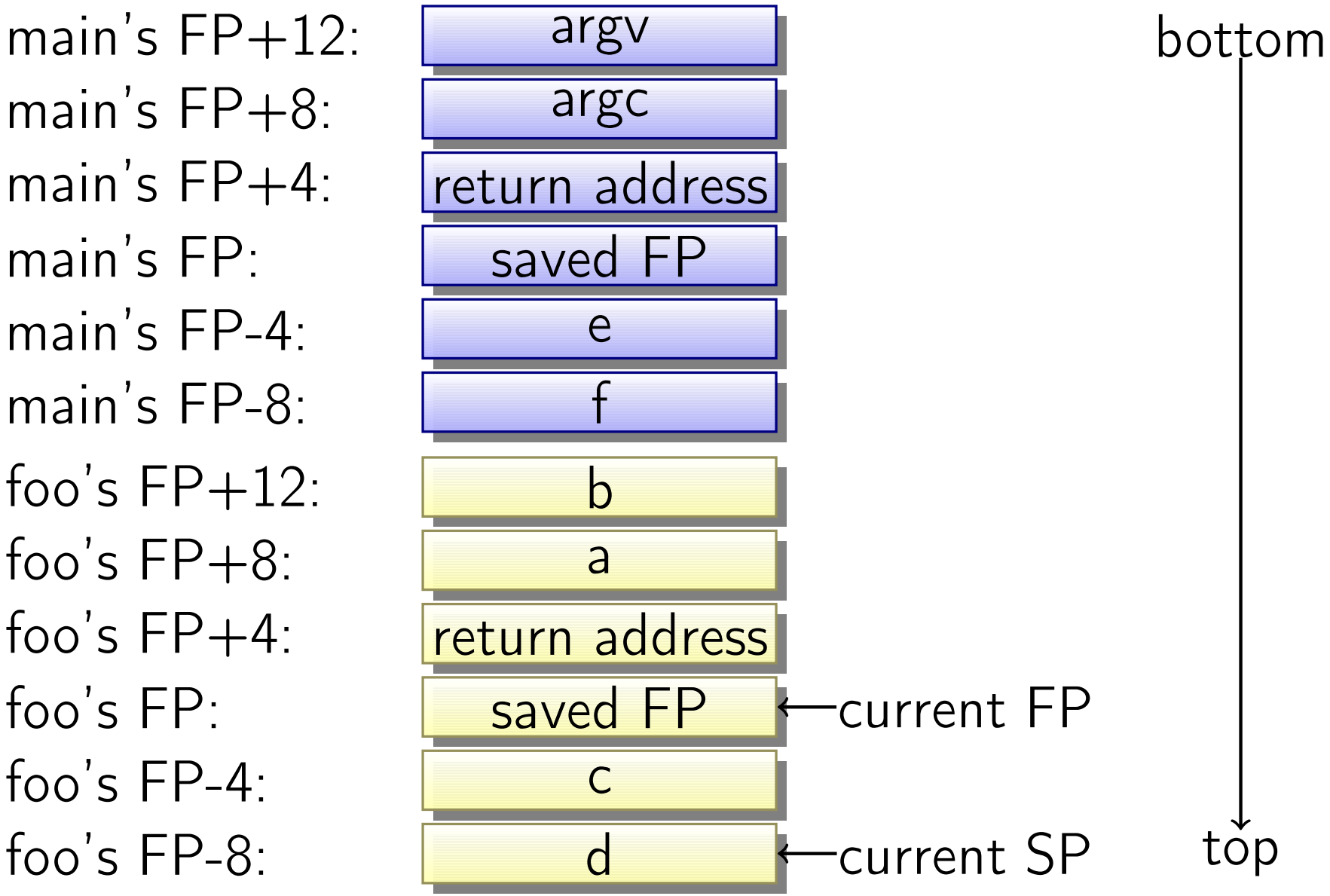
```
void foo(int a, int b){
    int c;
    int d;
    d = a + 1;
}
```

```
foo:
    pushl %ebp
    movl  %esp, %ebp
    subl  $16, %esp

    movl 8(%ebp), %eax      # eax = a
    addl $1, %eax           # eax++
    movl %eax, -8(%ebp)     # d = eax

    addl $16, %esp
    popl %ebp
    retl
```

Example III

```
void foo(int a, int b){
    int c;
    int d;
}

int main(argc,argv){
    int e=6;
    int f=7;
    foo(e,f);
}
```

Example III


main's FP+12:
argv
main's FP+8:
argc
main's FP+4:
return address
main's FP:
saved FP
main's FP-4:
e
main's FP-8:
f
foo's FP+12:
b
foo's FP+8:
a
foo's FP+4:
return address
foo's FP:
saved FP
current FP
foo's FP-4:
c
foo's FP-8:
d
current SP
bottom
top

Exercise I

1. Draw the stack (as in the previous slide) right before `foo's` returns.
2. Assume that when `main` starts executing SP=100. Show the *actual* addresses for all arguments, return addresses, and local variables on the stack.

```
void foo(int a, int b, int c){
    int d;
    d = a + b;
    return;
}
int main(argc,argv){
    int e=6;
    foo(e,3,4);
}
```

Exercise I

Exercise II

1. Draw the stack (as in the example in the previous slide) right before `foo` returns.
2. Give the address of every memory location in `foo`'s stack frame.

```
void foo(int i, int b){
    int arr[5];
    arr[i] = b;
}
```

Exercise II...

- Here is the generated code:

```
pushl %ebp
movl %esp, %ebp
subl $28, %esp

movl 12(%ebp), %eax
movl 8(%ebp), %ecx
movl %eax, -28(%ebp,%ecx,4) #[-28+ebp+ecx*4]=eax

addl $28, %esp
popl %ebp
retl
```

Exercise II

Outline

What is a buffer overflow?

- Buffer overflow attacks explained with beer!

  http://www.youtube.com/watch?v=7LDdd90aq5Y
- Another video: https://www.youtube.com/watch?v=1S0aBV-Waeo
- What is a buffer overflow attack?
- Why are they possible?
- How do I perform a buffer overflow attack?
- How do I prevent a buffer overflow attack?

Buffer overflow example

```
int foo(int n){
  char buffer[100];
  buffer[n] = 'c';
}
int main(){
  foo(200);
}
```

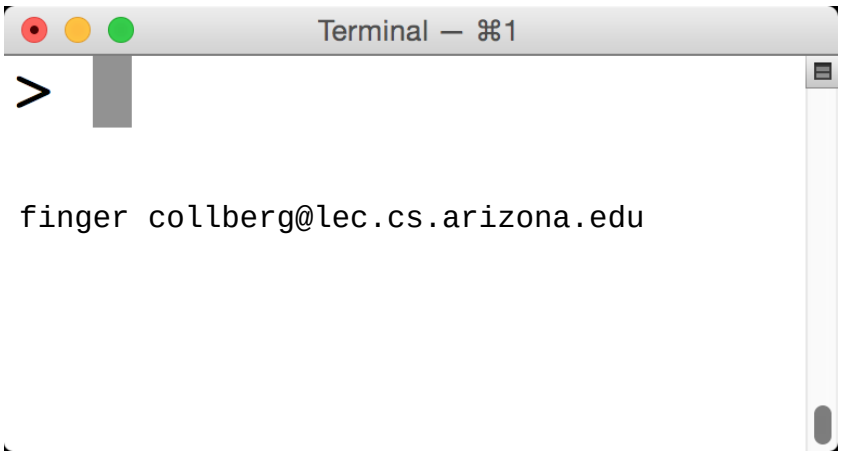
Buffer overflow example

```
#include <stdio.h>

int main(){
   char buffer[100];
   gets(buffer);
}
```

Buffer overflow example

FINGER

```
int main(){

   char BUFFER[100];

   gets(BUFFER);

}
```



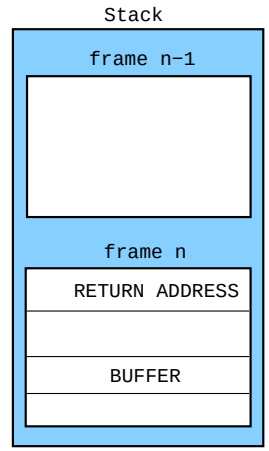
Buffer overflow example

FINGER

```
int main(){
   char BUFFER[100];
   gets(BUFFER);
}
```

Terminal — ⌘1

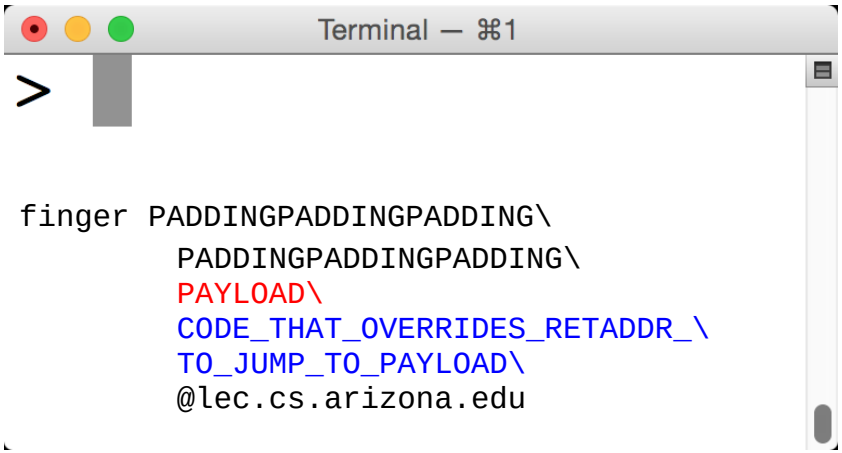
```
>

finger PADDINGPADDINGPADDING\
        PADDINGPADDINGPADDING\
        PAYLOAD\
        CODE_THAT_OVERRIDES_RETADDR_\
        TO_JUMP_TO_PAYLOAD\
        @lec.cs.arizona.edu
```

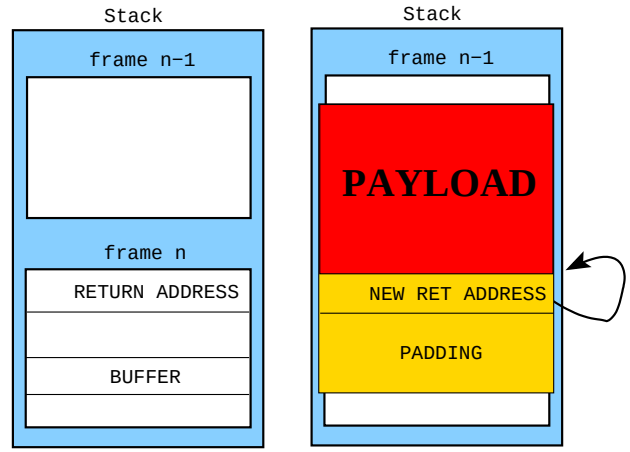


Buffer overflow example

FINGER

```
int main(){
   char BUFFER[100];
   gets(BUFFER);
}
```





Definitions

- buffer: A span of contiguous writable memory.
- stack frame: The space on the stack alotted to a particular procedure.
- buffer overflow: Writing past the declared bounds of a buffer.
- buffer overflow attack:
  - A method of gaining control of a system by executing some program/procedure with more data than it is prepared to handle.
  - The extra data is designed to cause malicious side effects.

Buffer overflow attack — Basic Idea

1. Inject some code into the memory of an executing program. The code consists of
   - Malicious payload that we want to execute;
   - Some way to get this code to execute.
2. Overwrite a memory address with the address of the payload.
   - This is often the return address of a function.
3. When the program jumps to this address, it instead jumps to our payload!