Programação Segura

Semana 2: Validação de Entrada – Buffer Overflow, String de Formatação

Profª Denise Goya
Denise.goya@ufabc.edu.br – UFABC - CMCC

Classes de Erros de Segurança

- Validação e representação dos dados de entrada
- Abuso de API
- Características de segurança
- Tempo e estado
- Tratamento de exceções
- Qualidade do código
- Encapsulamento
- Ambiente

Validação e Representação da Entrada

- Problemas relacionados a essa classe de erros são causados pelo mal tratamento de:
  - metacaracteres
  - alternância de codificações ou de representações numéricas
- Incluem uma variedade de problemas:
  - buffer overflow
  - string de formatação
  - ataques de Cross-site scripting

Buffer Overflow

- Transbordo de memória
- Tipo mais famoso e mais explorado de vulnerabilidade
- Causa:
  - O programador não limita a quantidade de informação que pode ser escrita em uma determinada área de memória (buffer: string, array, etc)

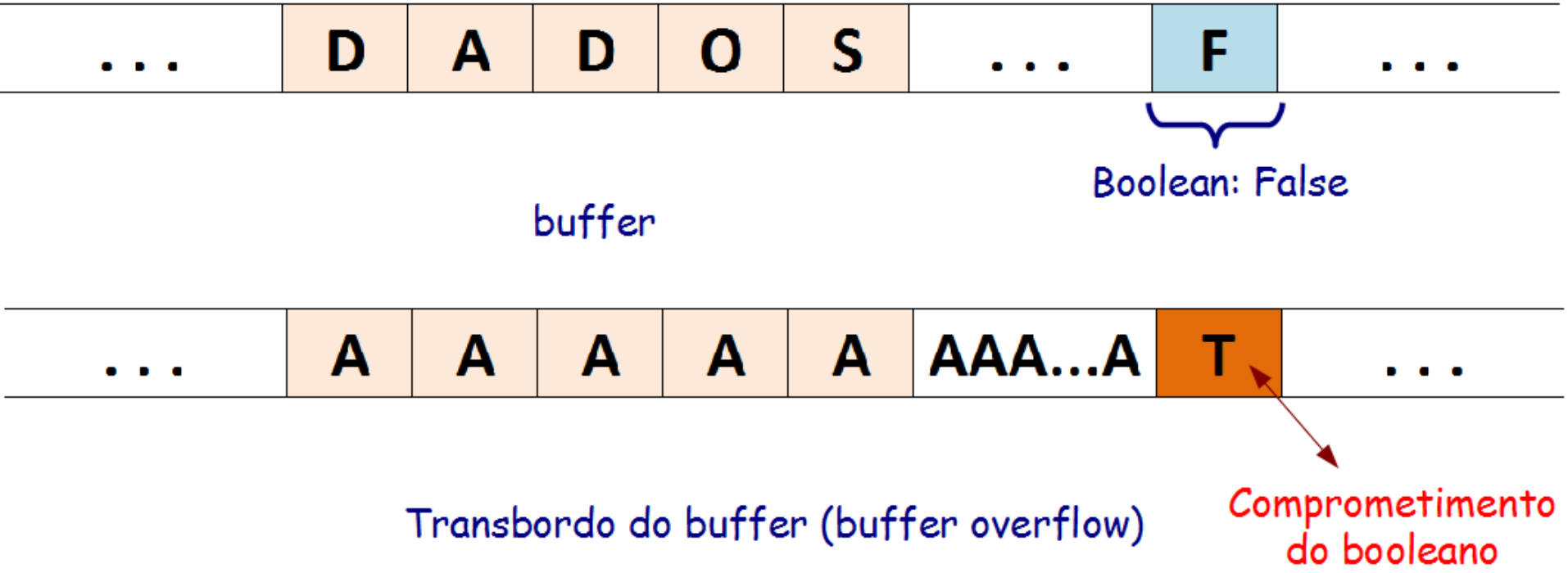
Buffer Overflow

- O transbordo da memória ocorre quando o programa copia os dados de entrada para o buffer sem verificar o seu tamanho

- Metade das vulnerabilidades descobertas nos últimos anos são de buffer overflow (CERT)

Buffer Overflow: exemplo

- Variáveis na memória:
  - Array de 5 posições (ex: char buffer[5])



Buffer Overflow – Ex. de exploração

- Descoberta (exemplo de um chat)
- O chat usa um string de **tamanho 50** para nomes
- O cara malvado acessa servidor através de telnet
  - Ex: telnet 192.168.0.4 1234
- O servidor pede o nome do usuário
- O malvado fornece um nome com **100** “A” repetidos
- O servidor aborta (buffer overflow leva a um estado inconsistente)
- ...

Buffer Overflow – Ex. de exploração

- ...
- O malvado roda o servidor com um debugger
- O malvado fornece de novo os “A” repetidos
- O servidor mostra um erro no endereço 0x39E8765A
- O malvado consegue fazer **DoS** em vários servidores
- Se o código fonte for disponível, ele pode implementar “stack smashing” e obter uma shell (ou sessão de linha de comando)

Buffer Overflow - Requisitos

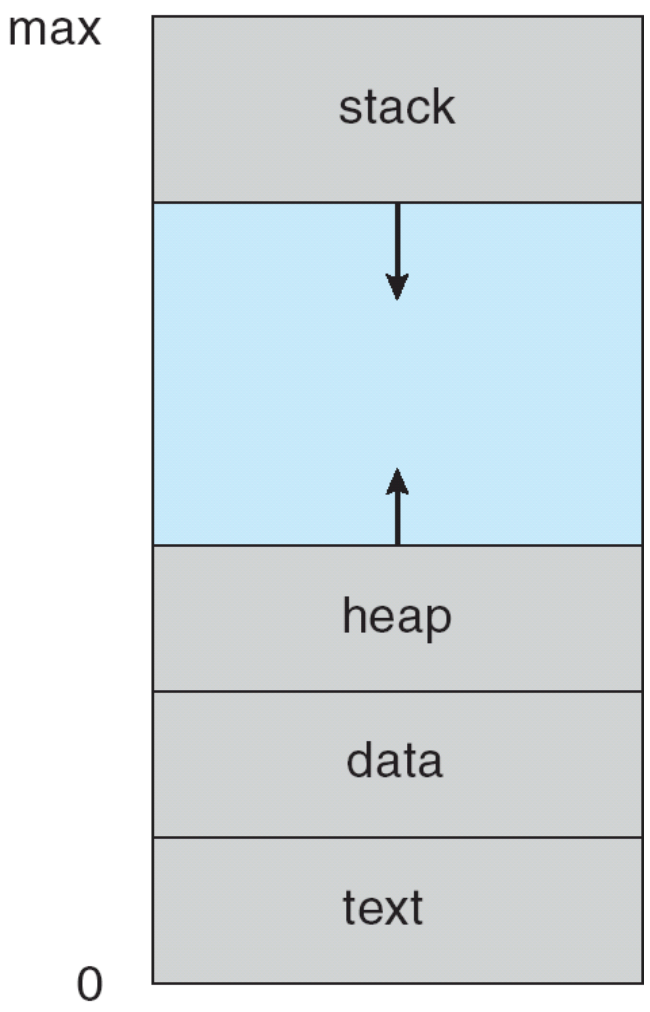
- Para implementar o ataque:
  - Entender funções em Linguagem C e a pilha
  - Alguma familiaridade com código de máquina
  - Conhecer como *systems calls* são realizadas
  - A chamada exec( )
  - Os invasores precisam saber qual CPU e SO são executados na máquina a ser atacada

Buffer Overflow - SO

- Veremos, resumidamente, como o Sistema Operacional organiza a memória internamente
  - Algumas particularidades de implementação são dependentes do SO

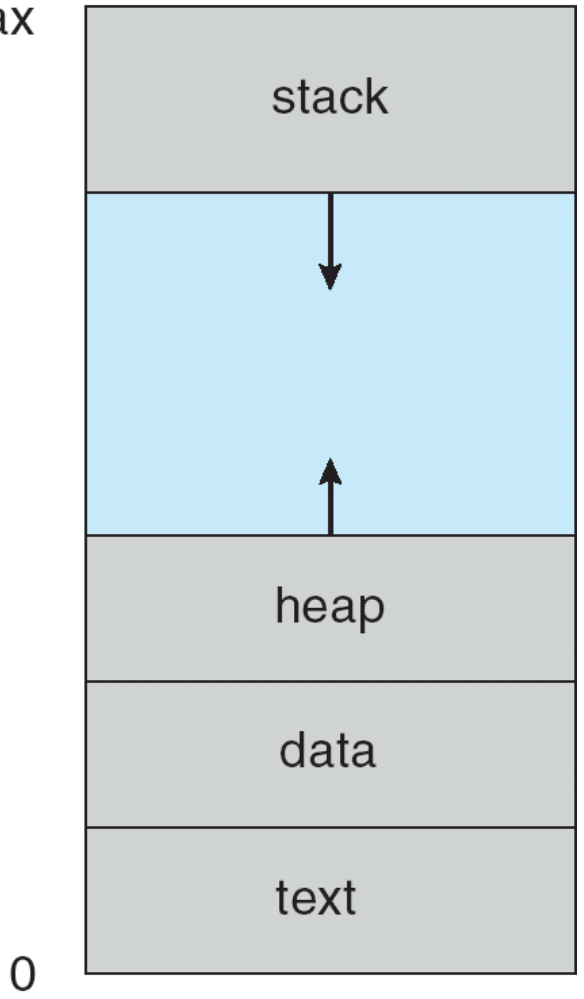
Memória

- O sistema operacional cria um processo que aloca memória e outros recursos
- Grandes componentes da memória (organização lógica):
  - Stack
  - Heap
  - Dados
  - Códigos



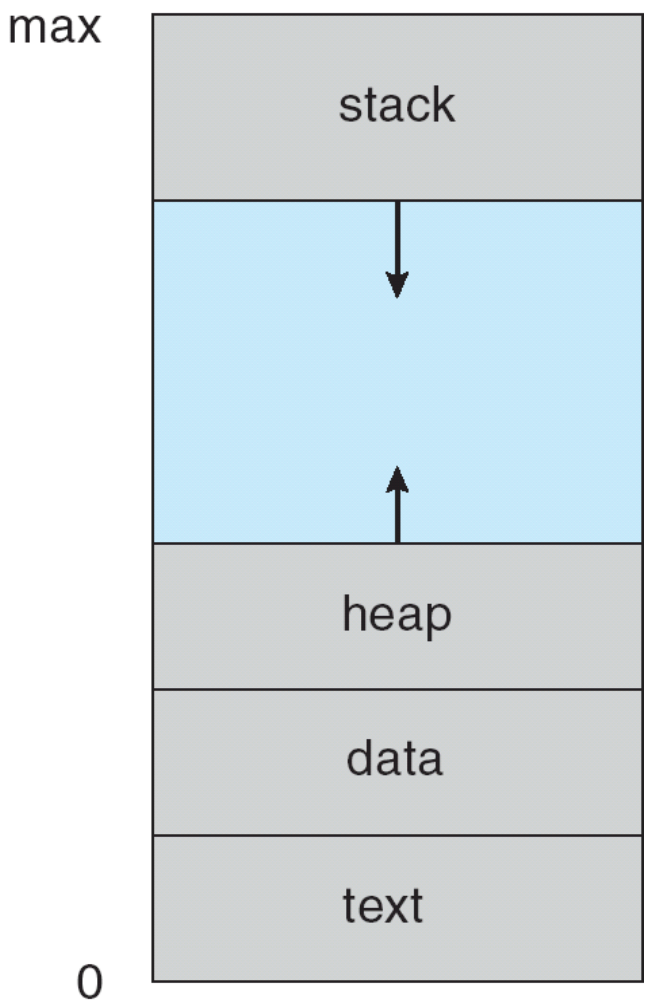
Stack (pilha de execução)

- guarda o ponto em que cada subrotina ativa deve retornar o controle quando terminar a execução;
- **armazena as variáveis que são locais** para as funções;



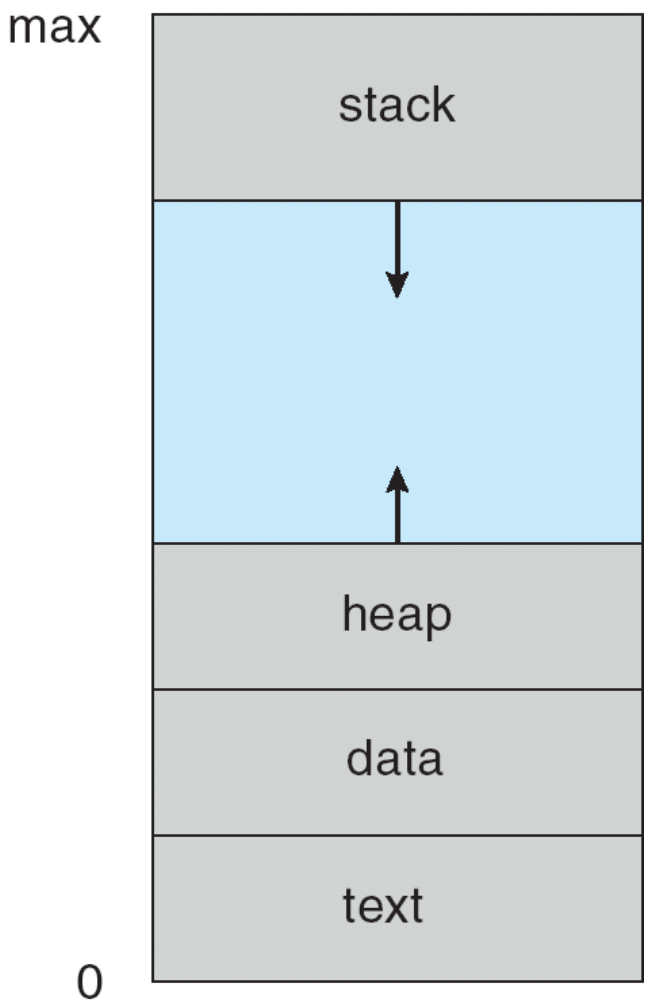
Heap

- memória dinâmica para variáveis criadas com **malloc**, calloc, realloc e liberadas com free



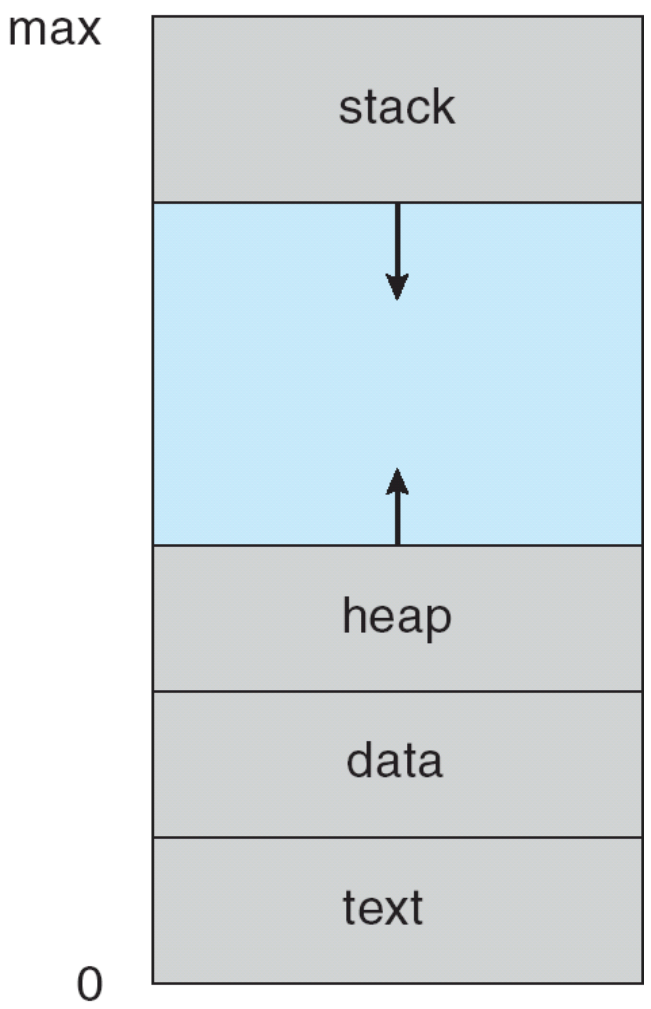
Data (Dados)

- Área em que são armazenadas as variáveis, incluindo variáveis **globais e estáticas**, ou não inicializadas



Text (Códigos)

- Área com os códigos dos processos do usuário, as instruções do programa que serão executadas



Processo na Memória

- Chamamos de **Processo** um programa em execução
- Veremos, resumidamente, como é o controle de execução de um processo na memória

Contador de Programa – PC

- O contador de programa (program counter – PC) aponta para a próxima instrução a ser executada (ou a instrução em execução)

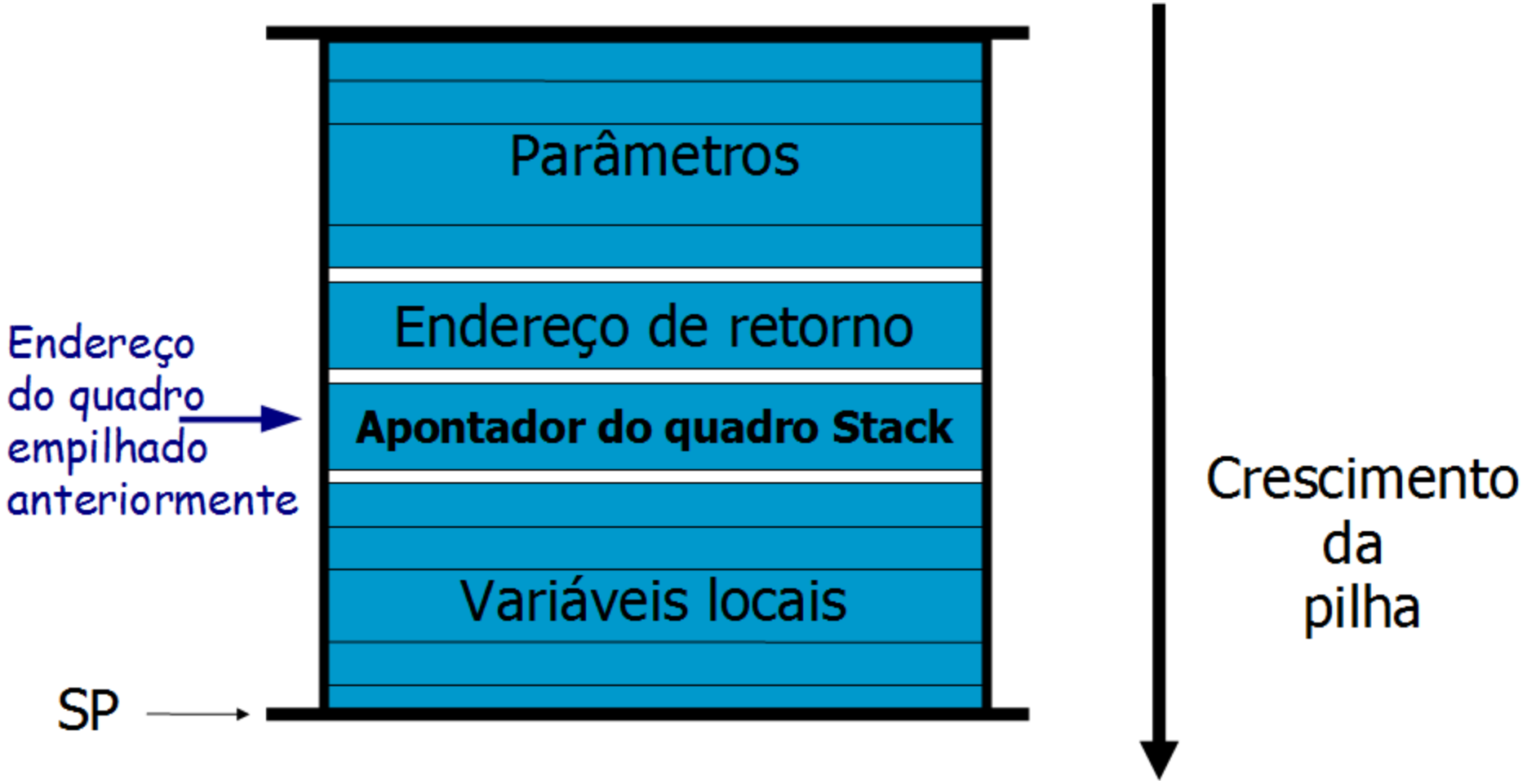
Chamada de Função

- Procedimento de chamada de função:
  - Prepara os parâmetros
  - Salva o estado e aloca na pilha as variáveis locais
  - Aponta para o começo do procedimento solicitado

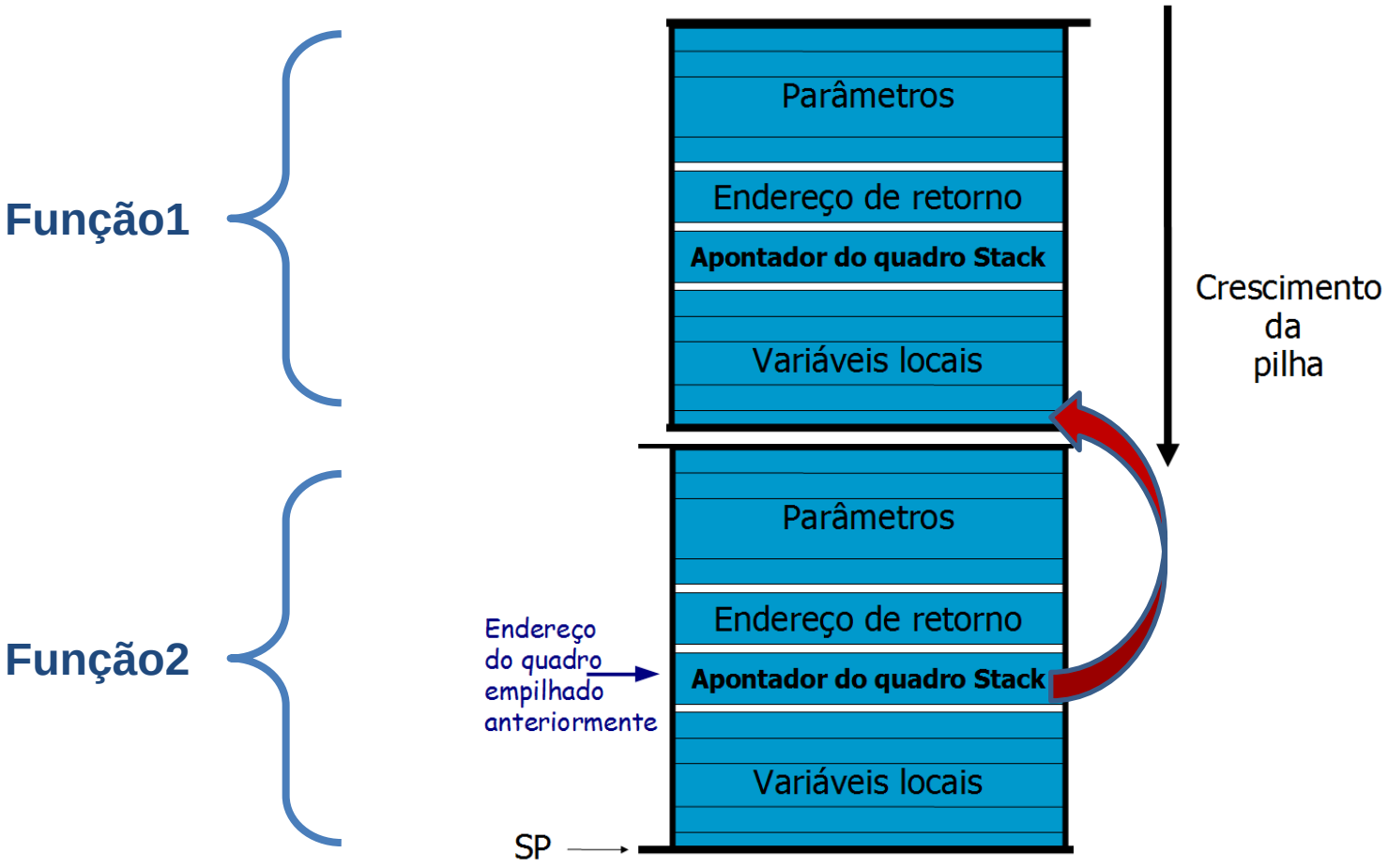
Retorno de uma Função Chamada

- Procedimento de retorno:
  - Grava o estado atual da pilha e ajusta a pilha
  - Execução continua a partir do endereço retornado

Pilha de Execução



Pilha de Execução

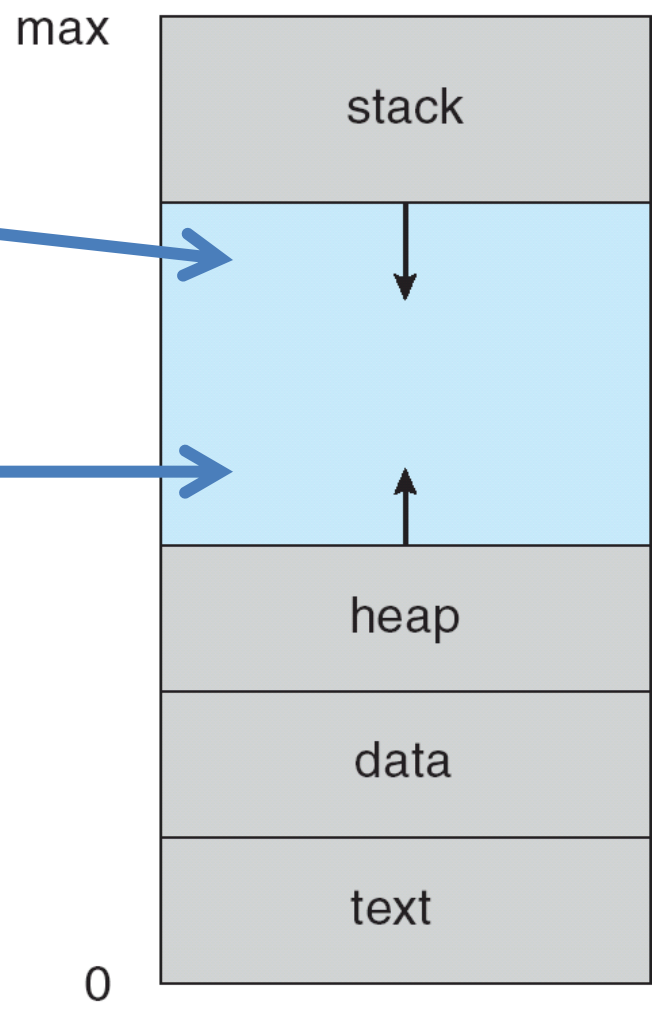


Parâmetros na Pilha

- Passos para tratar os parâmetros:
  - Salva o PC na stack (endereço de retorno)
  - Salva o valor de SP na stack (função empilhada anteriormente)
  - Aloca o espaço da stack para variáveis locais

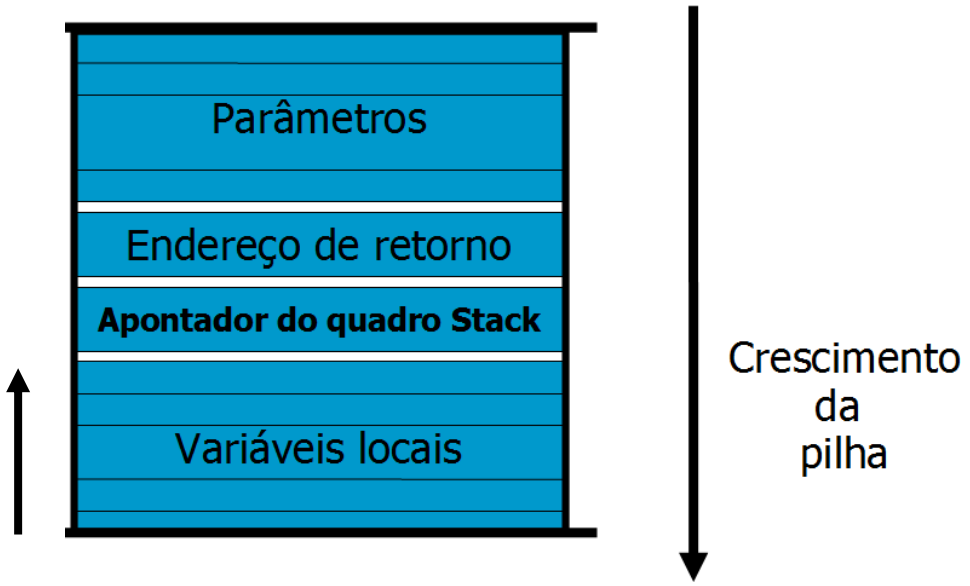
Ordem de Alocação

- A pilha de execução cresce num sentido, enquanto os demais dados, crescem no sentido oposto

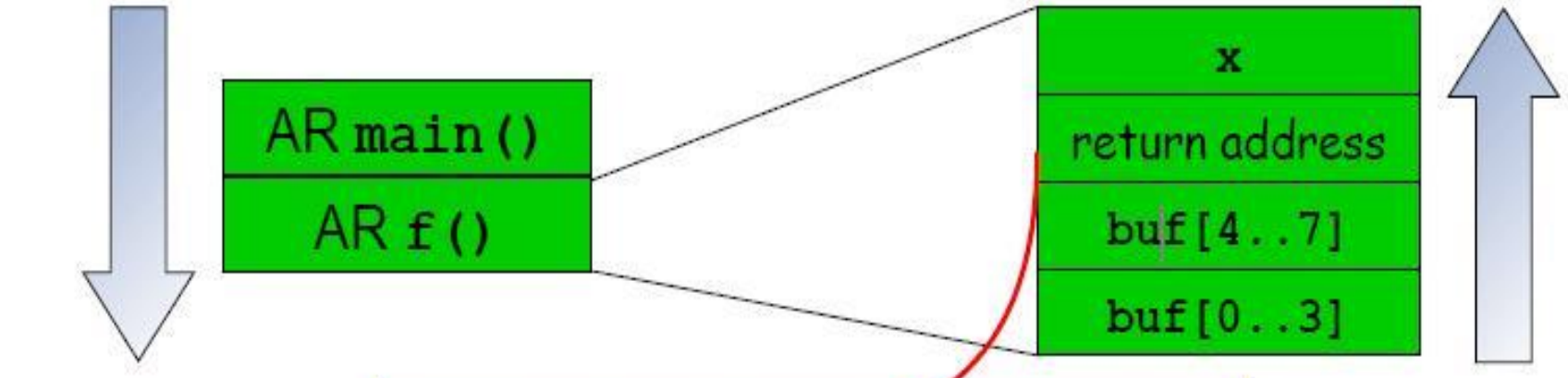


Ordem de Alocação

- A pilha de execução cresce num sentido, enquanto os dados locais à função, crescem no sentido oposto

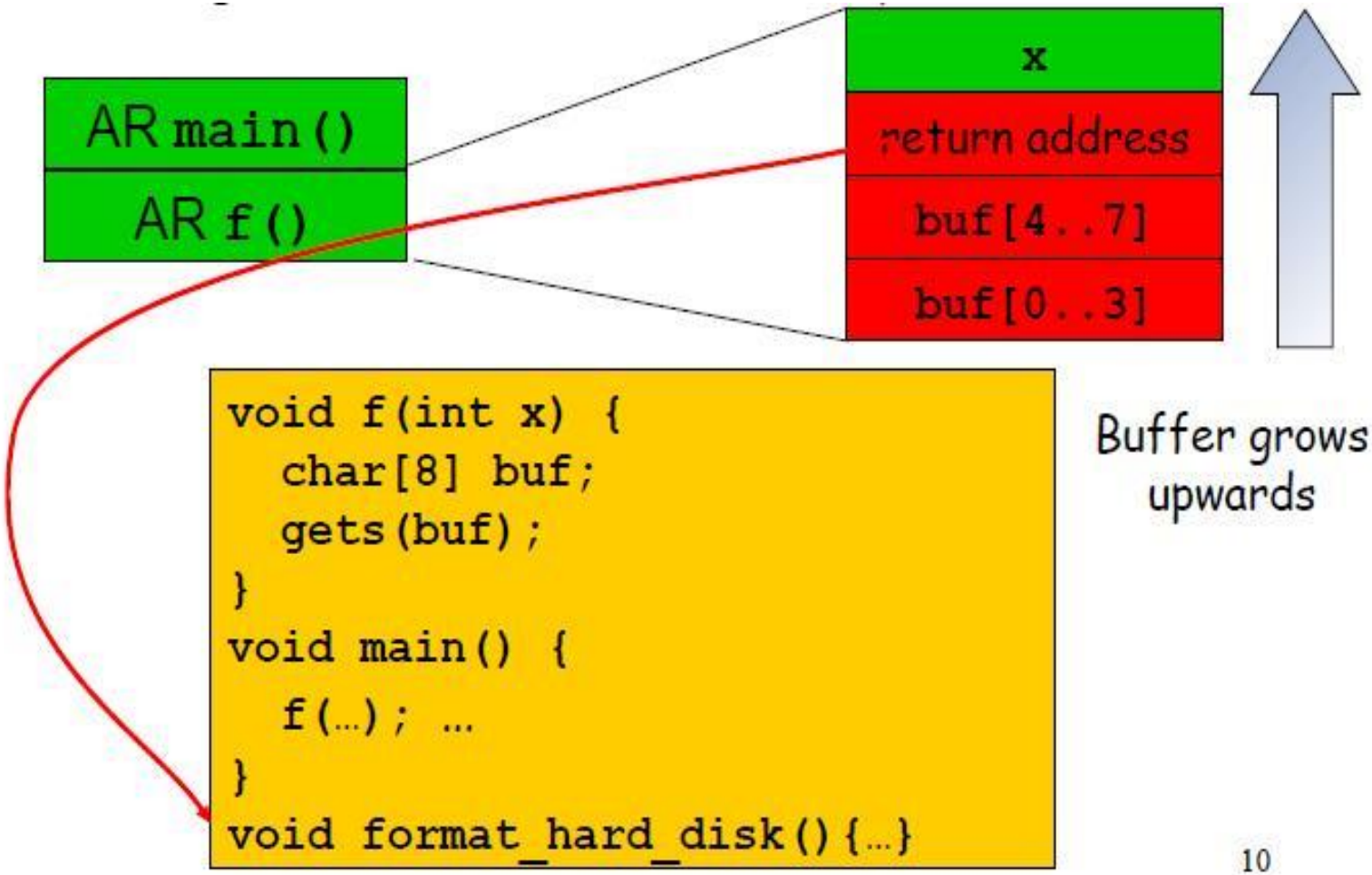


Buffer Overflow: Dinâmica




```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); …
}
void format_hard_disk(){…}
```

Buffer Overflow: Dinâmica



Buffer Overflow

- Estouro de stack:
  - é estouro de um buffer alocado na pilha (conhecido como **stack smashing**)
- Estouro de heap:
  - estouro do buffer alocado no heap

Buffer Overflow: Causas Comuns

- Programação fraca em cadeias de strings
- Especificação de bibliotecas de funções para término de strings
- Problemas com strings de formatação

Funções em C Problemáticas

- Algumas funções de bibliotecas em ANSI-C que podem ser fonte de buffer overflow quando mal usadas:
  - strcpy (char *dest, const char *src)
  - strcat (char *dest, const char *src)
  - gets (char *s)
  - scanf (const char *format, …)
  - printf (conts char *format, …)
  - . . .

Disseminação da Linguagem C

- A linguagem C é largamente usada em
  - Sistemas Operacionais
  - funções de baixo nível mesmo em outras linguagens de programação
  - codificação de drivers (controladores de dispositivos de hardware)
- Aplicações (no alto nível) fazem acesso a funções mais elementares (muitas vezes codificadas em C)

Exemplos de Funções de Biblioteca C

- **gets**

```
char buf[20];
gets(buf);    // lê a entrada do usuário até
              // primeiro caractere EoF
```

- Nunca use gets
- Use **fgets(buf, size, stdin)**

Exemplos de Funções de Biblioteca C

- **strcpy**

```
char dest[20];
strcpy(dest, src); // copia string src para dest
```

- strcpy assume: que *dest* é longo o bastante e que *src* possui um **‘\0’** que encerra a string
- Use **strncpy(dest, src, size)**

Exemplos de Funções de Biblioteca C

function

strcpy

<cstring>

```
char * strcpy ( char * destination, const char * source );
```

Copy string

Copies the C string pointed by *source* into the array pointed by *destination*, including the terminating null character (and stopping at that point).

To avoid overflows, the size of the array pointed by *destination* shall be long enough to contain the same C string as *source* (including the terminating null character), and should not overlap in memory with *source*.

function

strncpy

<cstring>

```
char * strncpy ( char * destination, const char * source, size_t num );
```

Copy characters from string

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

destination and *source* shall not overlap (see memmove for a safer alternative when overlapping).

Exemplos de Funções de Biblioteca C

- **strcpy:** cadê o erro abaixo?

```
char buf[20];
char prefix[] = “http://”;
...
strcpy(buf, prefix); // copia a string prefix para  buf

strncat(buf, path, sizeof(buf)); // concatena path
                                 // na string buf
```

Dangerous C system calls - Building secure software, J. Viega & G. McGraw, 2002

Extreme risk

- gets

High risk

- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsscanf

High risk (cntd)

- streadd
- strecpy
- strtrns
- realpath
- syslog
- getenv
- getopt
- getopt_long
- getpass

Moderate risk

- getchar
- fgetc
- getc
- read
- bcopy

Low risk

- fgets
- memcpy
- snprintf
- strccpy
- strcadd
- strncpy
- strncat
- vsnprintf

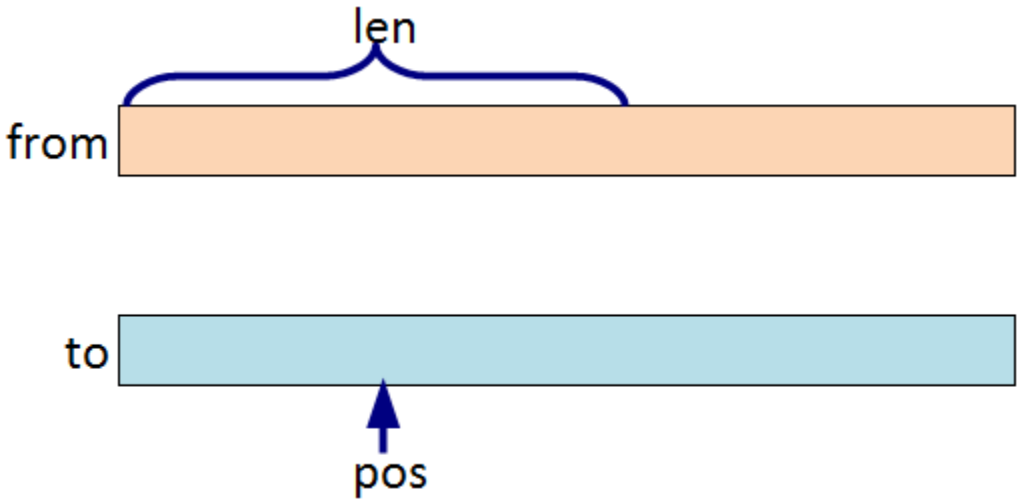
Buffer Overflow por Más Práticas

- No entanto, na prática, o problema maior **não** é causado pelas funções inseguras
  - os efeitos de Bibliotecas com rotinas problemáticas tendem a ser o menor dos problemas
- Más Práticas de Programação (Inseguras) são mais comuns

Buffer Overflow por Más Práticas

- Exemplo: por que o código abaixo é vulnerável a buffer overflow?

```
int copy_buf (char *to, int pos, char *from, int len) {
int i;
  for (i=0; i<len; i++) {
    to[pos] = from[i];
    pos++;
  }
  return pos;
}
```



Buffer Overflow: Exploração

- Estouro do buffer pode ser explorado para abortar um programa ou serviço (ataque à **disponibilidade**)

Buffer Overflow: Exploração

Shellcode:

- Caso mais sofisticado de exploração de overflow, em que o endereço de retorno é modificado para uma região na **própria pilha**, na qual o atacante colocou um código executável de seu interesse

Buffer Overflow: Exploração

Shellcode mais comum:

- Acionar um shell que dá acesso à **linha de comando**, com privilégios do programa atacado.
- Consequências das mais diversas
- Grande poder ao atacante

Buffer Overflow: Prevenção

- Programação criteriosa
- Uso de bibliotecas e compiladores mais recentes
- Compilação com mecanismos de **proteção da pilha**

Buffer Overflow: Prevenção

- Compilação com mecanismos de **proteção da pilha**:
  - Deteção de modificação do endereço de retorno de funções na pilha (**stackguard)**:
    - **Canário**: valor gravado na pilha para indicar modificações
  - Deteção de escritas indevidas na pilha (stackshield)
- Disponível em qualquer compilador mais recente

Buffer Overflow: Prevenção

- Proteção do espaço de endereçamento executável
  - Pode ser usada para impedir que código seja executado a partir do heap ou stack
  - Requer suporte da unidade de gerenciamento de memória (MMU) do processador
- Aleatorização do espaço de endereçamento
  - Aleatoriza os endereços de início de stack e heap
  - Torna mais difícil um atacante adivinhar os endereços para efetivar ataques

STRINGS DE FORMATAÇÃO

Strings de Formatação

printf (“uma string de formação %d %s ”, ...)

- Permite que um invasor passe como parâmetro especificadores de conversão (ex: “%d”, “%s”) e faça com que sejam processados mais dados do que o programador considerou originalmente
- Permite que endereços de memória sejam sobrescritos e código malicioso seja executado
- O atacante precisa ter muito conhecimento, ou seja, precisa ser um “hacker de verdade”

Strings de Formatação

- Problema com linguagens C e C++
- Não é possível contar argumentos passados para uma função em C, de modo que argumentos faltando não são identificados
  - Quantidade variável de argumentos
- O que vair ocorrer se o seguinte código for executado?

```
int main () {
        printf("Juliana tem %d gatos");
}
```

a intenção do programador era:

```
int main () {
        printf(nomeUsuario);
}
```

Strings de Formatação

- Resultado possível do programa:

  Juliana tem -1073742416 gatos

- O programa lê o argumento que está faltando da pilha do processo
- E pega lixo
  - ou coisas interessantes se o atacante deseja vasculhar a pilha

Strings de Formatação

Principais parâmetros de formatação:

- %d - decimal (int) – valor
- %u - unsigned decimal (unsigned int) – valor
- %x - hexadecimal (unsigned int) – valor
- %s - string ((const) (unsigned) char *) - referência
- %n – número de bytes escritos até então, (* int) - referência

Strings de Formatação

Ataque: Crashing

- printf ("%s%s%s%s%s%s%s%s%s%s%s%s%s");
- Para cada %s, printf() buscará um número na pilha, considerando esse número como endereço, e imprimirá o conteúdo da memória apontado por esse endereço como uma string até um caractere NULL encontrado
- Uma vez o número carregado por printf() pode não ser um endereço, a memória aponta para esse número que pode não existir e o programa irá falhar

Strings de Formatação

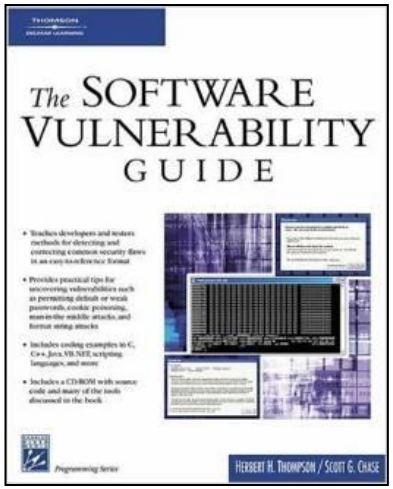
Ataques mais sofisticados

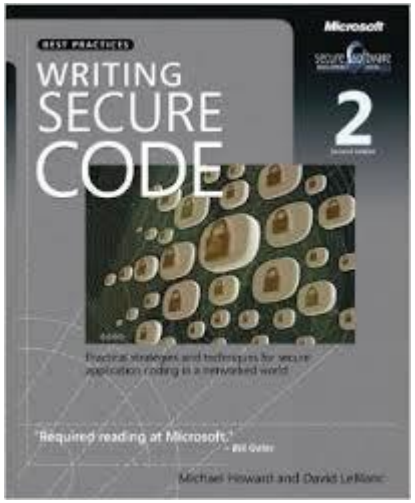
- Requer maior experiência em programação
- Pode-se ter acesso a certas áreas de memória
- Pode-se imprimir dados críticos (valor de chaves, por exemplo)
- Pode-se alterar dados de variáveis quaisquer

ESTUDO INDIVIDUAL

Leitura Recomendada

- THOMPSON, H.; CHASE, SCOTT G.: "**The Software Vulnerability Guide**" Charles River Media, 2005
- HOWARD, M.; LEBLANC, D.: "Writing Secure Code" Microsoft Press, 2a edição, 2002.
- WEELER, D.: "Secure Programming for Linux and Unix HOWTO – Creating Secure Software". Ebook disponível em http://www.dwheeler.com/secure-programs/





Exercício

Dos problemas estudados hoje, cite quais podem afetar os requisitos de segurança abaixo e justifique:

- confidencialidade
- integridade
- disponibilidade