

Buffer Overflow – Uma introdução

Teórica

Raphael Duarte Paiva

Equipe de Pesquisa e Desenvolvimento



GRIS – Grupo de Resposta a Incidentes de Segurança

DCC – IM - UFRJ

Buffer Overflow – Uma introdução

Teórica

- Duas partes:
- Introdução
- Dissecando



Buffer Overflow – Uma introdução

Teórica

- Um breve histórico
- Buffer – o que é?
- Overflow
- Buffer Overflow – Uma visão geral
- Exploits

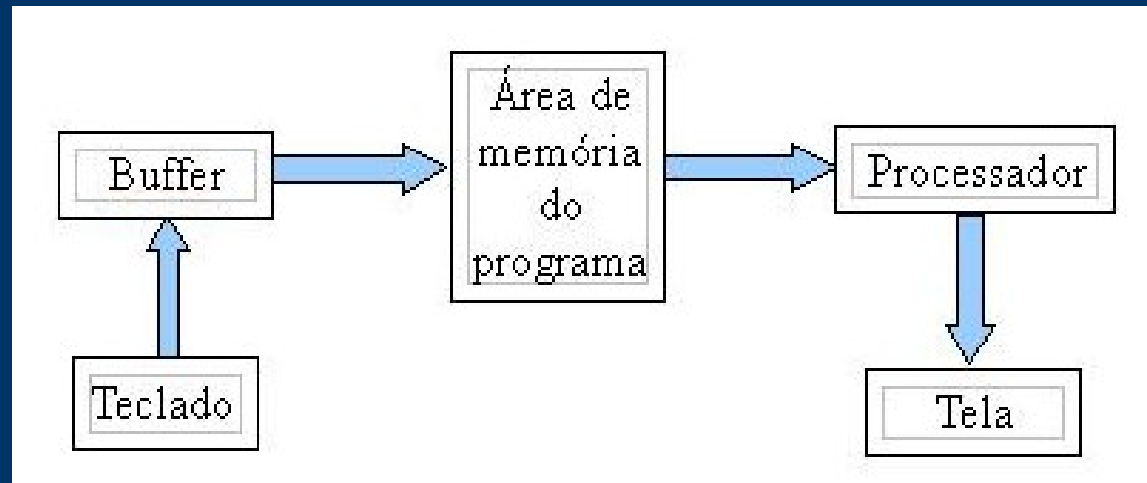


Buffer Overflow – Um breve histórico

- Morris Worm - 1988
 - Segundo o autor, o worm tinha o objetivo de medir o tamanho da internet.
 - Utilizou uma falha no finger (uma chamada à gets()) para se copiar para outras máquinas.
 - Um critério muito rigoroso para decidir se iria se copiar para um host ou não foi o que o tornou maléfico.
 - Vários processos rodando = consumo de recursos, tanto da própria máquina quanto da rede.
 - Teve um impacto devastador na época, atingindo 6000 computadores – aproximadamente 10% da internet da época.
 - Robert Morris, o autor do worm, foi sentenciado à um período probatório de 3 anos, 400 horas de serviço comunitário e multado em US\$10.500,00

Buffer Overflow – Buffer – o que é?

- Uma região temporária de memória utilizada para guardar dados que serão transportados de um lugar para o outro
- Exemplos:
 - Um programa que recebe dados direto do teclado e os imprime na tela, i.e. Terminal.
 - Uma pessoa copiando a matéria de um quadro-negro.



Buffer Overflow - Overflow

- Transbordamento



Buffer Overflow – Uma visão geral

- Tecnicamente, o buffer overflow consiste em armazenar em um buffer de tamanho fixo, dados maiores que o seu tamanho.



Buffer Overflow – Uma visão geral

- Exemplo:

- Duas estruturas de dados adjacentes: A (vetor de caracteres de 8 bytes) e B (inteiro de 2 bytes)

A:

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

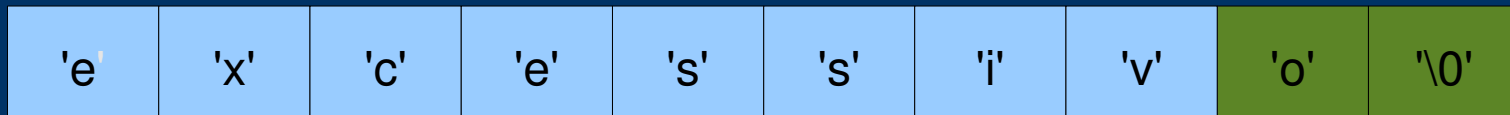
B:

0	9
---	---

- O que aconteceria se tentássemos inserir a palavra "excessivo" em A?

Buffer Overflow – Uma visão geral

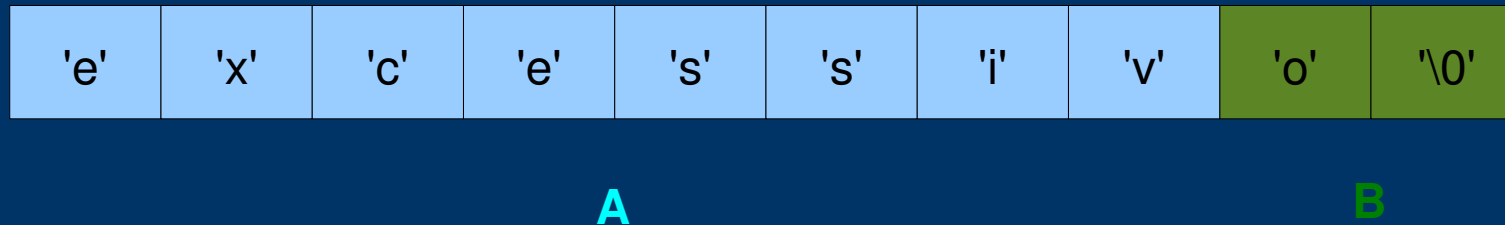
- "excessivo" ocupa 10 caracteres – 'e', 'x', 'c', 'e', 's', 's', 'i', 'v', 'o', '\0', para o computador.
- '\0' indica o final da string, que é um vetor de caracteres.
- Cada caractere ocupa 1 byte em memória.
- Logo, A, que tem 8 bytes, será transbordado e os dados excedentes serão armazenados em uma posição adjacente na memória:



A

B

Buffer Overflow – Uma visão geral



- A "transbordou" e os 2 bytes excedentes foram armazenados em B.
- Quando B for lido como um inteiro em um sistema little-endian, seu valor será igual a concatenação dos valores binários de '\0' e 'o', nesta ordem, ou seja: B = 000000001101111, que corresponde ao inteiro 111.

Buffer Overflow – Uma visão geral

- Quais seriam as consequências?
 - Funcionamento errôneo do programa
 - Uma string maior resultaria na tentativa de escrita de uma região de memória que não pertence ao programa, resultando numa falha de segmentação.
 - Erros deste tipo em programas maiores podem levar à bugs desastrosos, horas de trabalho perdidas com depuração e falhas de segurança que podem levar ao surgimento de exploits, possivelmente causando grandes transtornos à todos que usam o programa.

Buffer Overflow - Exploits

- Exploits
 - Pedacos de código com o objetivo de se aproveitar de bugs ou falhas de segurança em softwares.
 - Podem causar comportamento errôneo, inesperado de um programa.
 - Podem ter o objetivo de prover acesso privilegiado ao sistema, local ou remotamente.
- Tipos mais comuns de ataques baseados em buffer overflow:
 - Stack-based buffer overflow
 - Heap-based buffer overflow
 - Return-to-libc attack

Buffer Overflow - Dissecando

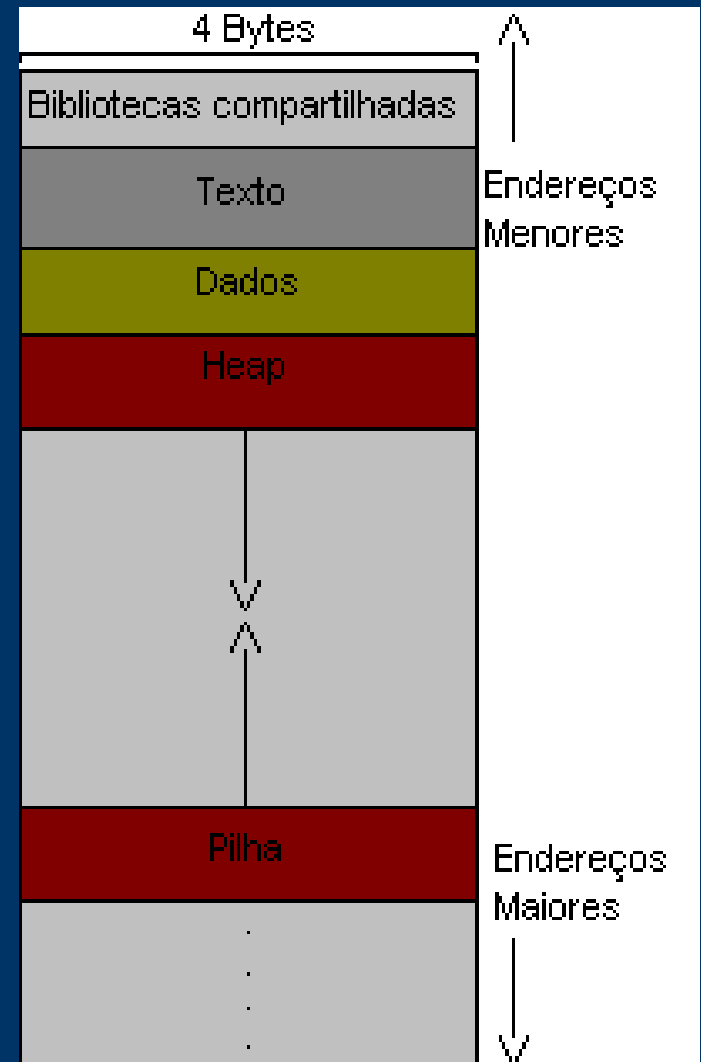
- A Estrutura de um programa na memória
- Conhecendo melhor a pilha
- Uma visão mais detalhada
- Técnicas de buffer overflow



Buffer Overflow – A estrutura de um programa na memória

- A memória se divide em 4 partes:

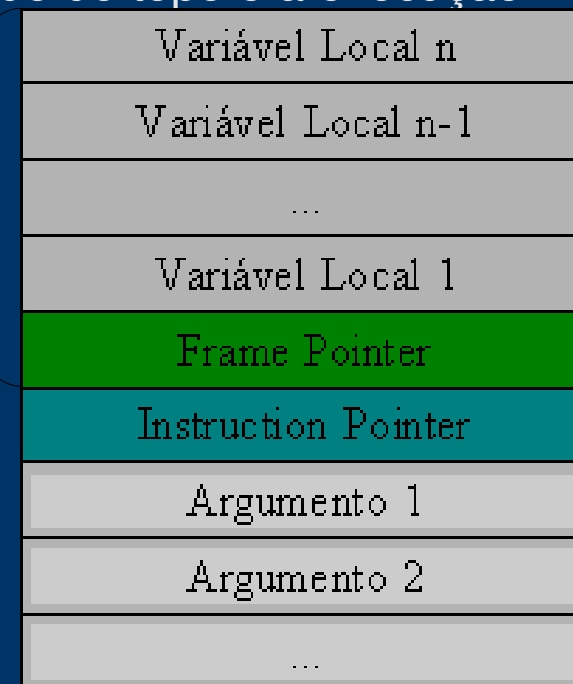
- Texto
- Dados
- Heap
- Pilha



Buffer Overflow – Conhecendo melhor a pilha

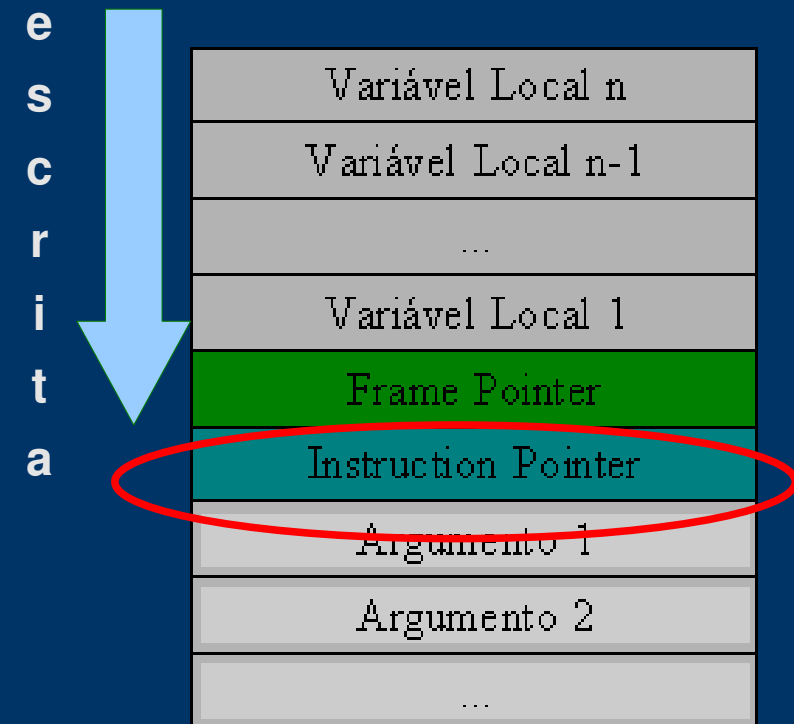
- A pilha consiste de frames empilhados
- Para cada subrotina (função) de um programa que é chamada, cria-se um frame, que é colocado no topo da pilha para a execução da mesma.
- Ao término da execução de uma subrotina, o frame é retirado do topo e a execução do programa continua do frame anterior.
- Um frame contém as seguintes informações sobre a subrotina associada:
 - Variáveis locais
 - Frame Pointer
 - Instruction Pointer
 - Argumentos (parâmetros) da subrotina

Payload
Space



Buffer Overflow – Uma visão mais detalhada

- Em um escopo mais específico, podemos agora definir um "alvo" para um ataque.
- Só podemos escrever em uma direção na memória
- Alterando o Instruction Pointer, podemos direcionar a execução do programa para onde quisermos!



Buffer Overflow - Exemplo

- Qual seria a saída do seguinte programa?

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x;
```

```
    x = 0;
```

```
    x = 1;
```

```
    printf("%d\n", x);
```

```
    return 0;
```

```
}
```



Buffer Overflow - Exemplo

- Existe uma maneira de "pular" uma instrução deste programa?
 - Sim! O exemplo a seguir mostrará o procedimento para pularmos a instrução "x = 1", para que a saída do programa seja "0".
- Precisamos então redirecionar a execução do programa logo após a instrução "x = 0".
- Vamos adicionar a chamada à uma função logo após "x = 0":

Buffer Overflow - Exemplo

```
int main(void)
{
    int x;

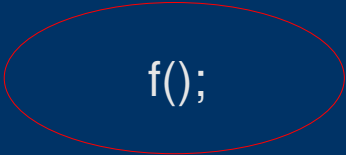
    x = 0;

    f();

    x = 1;

    printf("%d\n", x);

    return 0;
}
```



Buffer Overflow - Exemplo

- Qual deve ser o conteúdo de f()?
 - Quando f() for chamada será criado um frame que conterà:
 - Um "início" de f():

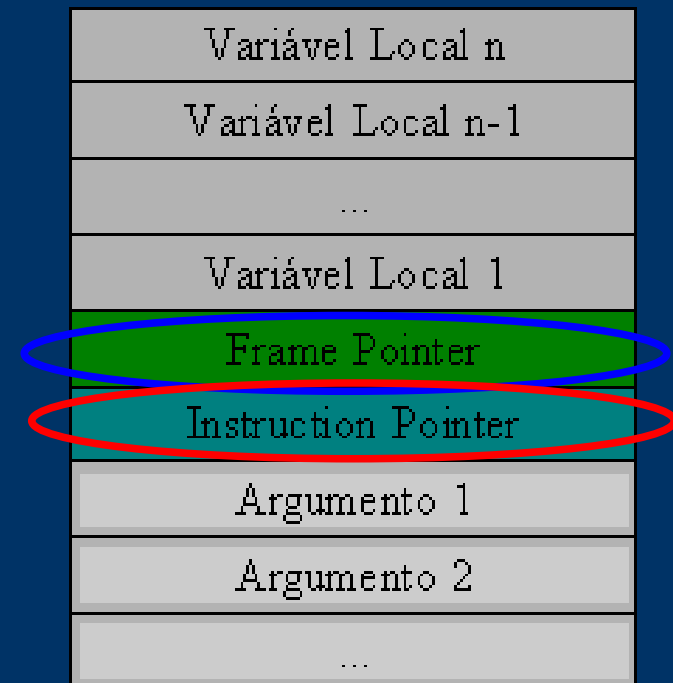
```
void f(void)
{
    char buffer[8];
}
```

- Na memória, buffer[8] ficaria:
 - buffer[0]
 - buffer[1]
 - (...)
 - buffer[6]
 - buffer[7]

Variável Local n
Variável Local n-1
...
Variável Local 1
Frame Pointer
Instruction Pointer
Argumento 1
Argumento 2
...

Buffer Overflow - Exemplo

- Como buffer[8] foi a primeira estrutura de dados declarada, logo após o espaço reservado para ela, está o frame pointer.
- 4 bytes depois está o nosso alvo: o Instruction Pointer.
- Precisamos agora reescrever o Instruction pointer.
 - Em C, podemos criar um ponteiro e fazê-lo apontar para o Instruction Pointer.
 - Assim, poderemos mudar o valor armazenado no Instruction Pointer.



Buffer Overflow - Exemplo

```
void f(void)
{
    char buffer[8];
    int *ret;

    ret = 0/*endereço do IP*/;

    (*ret) = 0/*endereço da instrução desejada*/;
}
```

- Criamos o ponteiro *ret, mas agora, como vamos descobrir o endereço do IP e o endereço da instrução desejada?

Buffer Overflow - Exemplo

- Os endereços do IP e das instruções de um programa, em certos ambientes ficam em um mesmo espaço durante uma mesma sessão.
- Como queremos que o programa funcione em sessões diferentes, vamos utilizar valores relativos:



Buffer Overflow - Exemplo

- Temos um buffer de caracteres de 8 posições (8 bytes) e um espaço (frame pointer) de 4 bytes antes do Instruction Pointer.
- Então concluimos que a distância entre o primeiro endereço de buffer[8] e o IP é de $8 + 4 = 12$ bytes.
- `ret = buffer + 12; // ret agora guarda o endereço do IP`

Buffer Overflow - Exemplo

- Para descobrirmos o endereço para reescrever o IP, precisamos olhar o código assembly do programa, então compilemos o seguinte código:

```
#include <stdio.h>

void f(void)
{
    char buffer[8];
    int *ret;

    ret = buffer + 12;

    (*ret) += 0;
}

int main(void) {
    int x;

    x = 0;

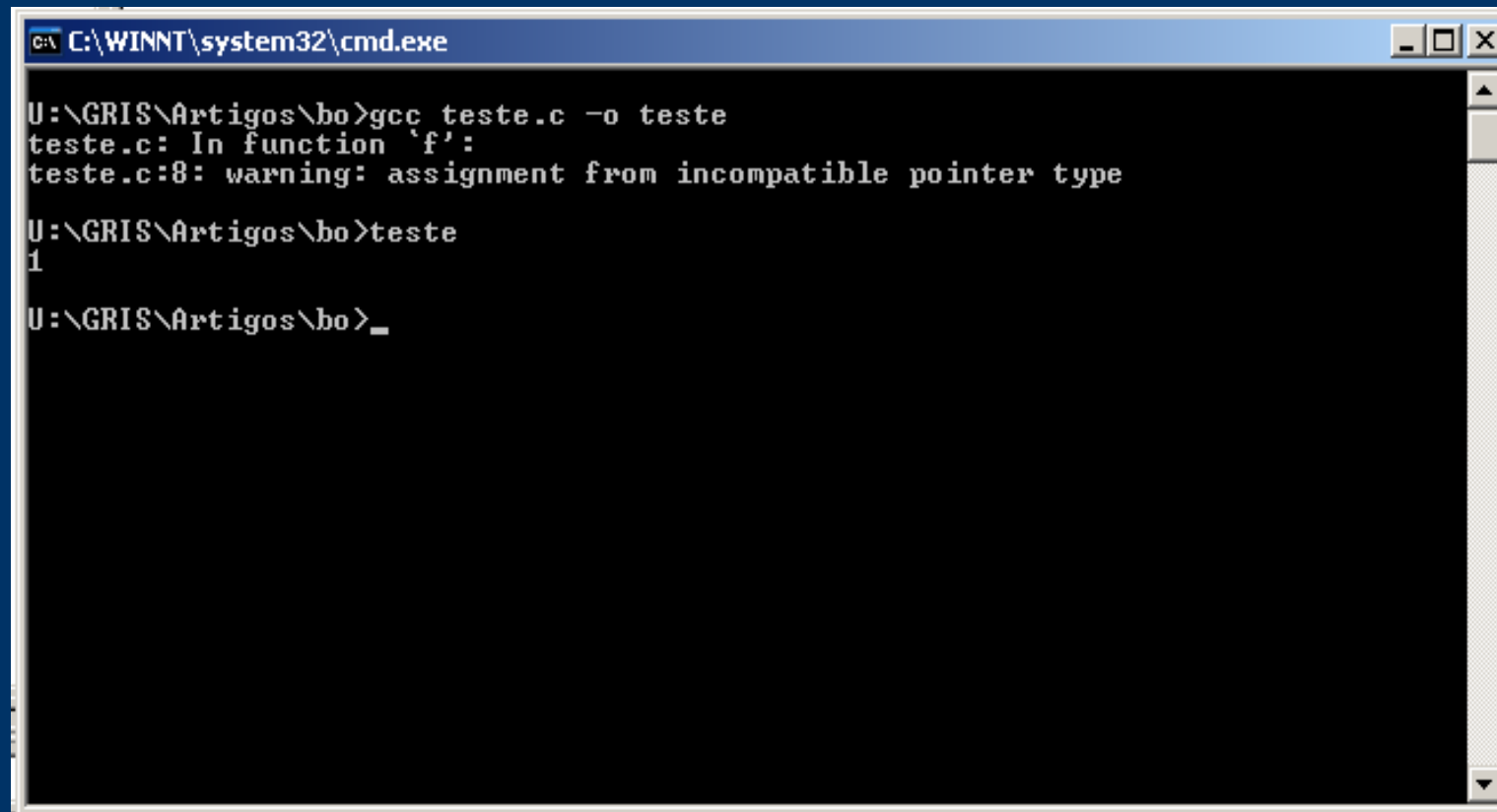
    f();

    x = 1;

    printf("%d\n", x);

    return 0;
}
```

Buffer Overflow - Exemplo



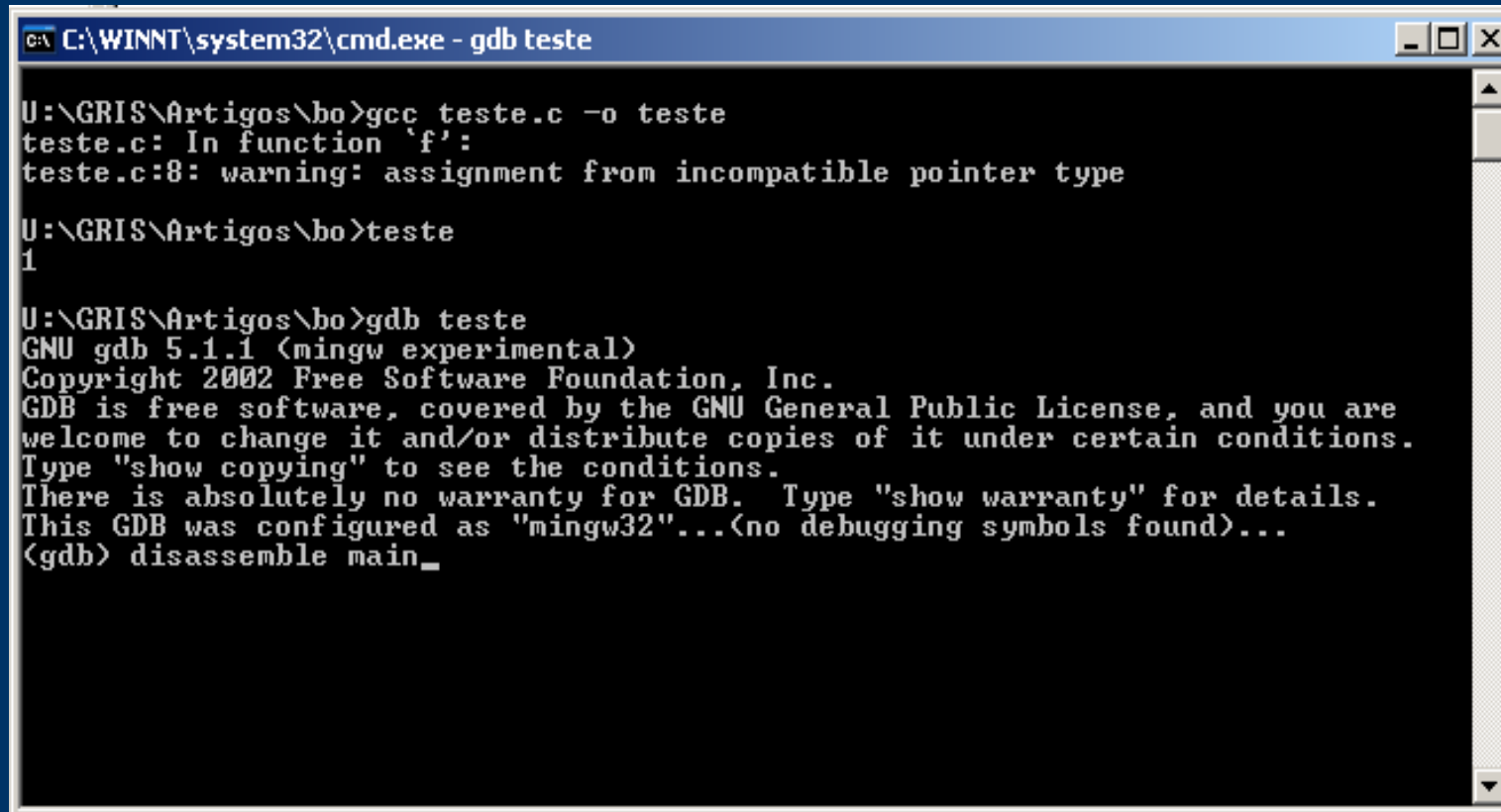
```
C:\WINNT\system32\cmd.exe

U:\GRIS\Artigos\bo>gcc teste.c -o teste
teste.c: In function 'f':
teste.c:8: warning: assignment from incompatible pointer type

U:\GRIS\Artigos\bo>teste
1

U:\GRIS\Artigos\bo>_
```

Buffer Overflow - Exemplo



```
C:\WINNT\system32\cmd.exe - gdb teste

U:\GRIS\Artigos\bo>gcc teste.c -o teste
teste.c: In function 'f':
teste.c:8: warning: assignment from incompatible pointer type

U:\GRIS\Artigos\bo>teste
1

U:\GRIS\Artigos\bo>gdb teste
GNU gdb 5.1.1 (mingw experimental)
Copyright 2002 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 "mingw32"...<no debugging symbols found>...
(gdb) disassemble main_
```

Buffer Overflow - Exemplo

```
C:\WINNT\system32\cmd.exe - gdb teste
(gdb) disassemble main
Dump of assembler code for function main:
0x4012a0 <main>:      push    %ebp
0x4012a1 <main+1>:      mov     %esp,%ebp
0x4012a3 <main+3>:      sub     $0x8,%esp
0x4012a6 <main+6>:      and     $0xffffffff0,%esp
0x4012a9 <main+9>:      mov     $0x0,%eax
0x4012ae <main+14>:     mov     %eax,0xffffffff8(%ebp)
0x4012b1 <main+17>:     mov     0xffffffff8(%ebp),%eax
0x4012b4 <main+20>:     call    0x402980 <_alloca>
0x4012b9 <main+25>:     call    0x4013c0 <__main>
0x4012be <main+30>:     movl    $0x0,0xffffffffc(%ebp)
0x4012c5 <main+37>:     call    0x401280 <f>
0x4012ca <main+42>:     movl    $0x1,0xffffffffc(%ebp)
0x4012d1 <main+49>:     sub     $0x8,%esp
0x4012d4 <main+52>:     pushl   0xffffffffc(%ebp)
0x4012d7 <main+55>:     push    $0x40129b
0x4012dc <main+60>:     call    0x402a30 <printf>
0x4012e1 <main+65>:     add     $0x10,%esp
0x4012e4 <main+68>:     mov     $0x0,%eax
0x4012e9 <main+73>:     leave
0x4012ea <main+74>:     ret
0x4012eb <main+75>:     nop
0x4012ec <main+76>:     nop
---Type <return> to continue, or q <return> to quit---
```

Buffer Overflow - Exemplo

- Temos então que o endereço de retorno original é 0x4012ca e o endereço do início da printf() é 0x4012d4.
- Então se somarmos ao endereço de retorno original a diferença dele mesmo com o endereço do início do printf(), teremos f() retornando sempre para o printf().
- $0x4012d4 - 0x4012ca = 0xA$;
- Com isso, podemos completar o código:



Buffer Overflow - Exemplo

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
void f(void)
```

```
int x;
```

```
{
```

```
char buffer[8];
```

```
x = 0;
```

```
int *ret;
```

```
f();
```

```
ret = buffer + 12;
```

```
x = 1;
```

```
(*ret) += 0xA;
```

```
printf("%d\n", x);
```

```
}
```

```
return 0;
```

```
}
```

Buffer Overflow - Exemplo

```
C:\WINNT\system32\cmd.exe
0x4012b1 <main+17>:  mov     0xffffffff8(%ebp),%eax
0x4012b4 <main+20>:  call    0x402980 <_alloca>
0x4012b9 <main+25>:  call    0x4013c0 <__main>
0x4012be <main+30>:  movl    $0x0,0xffffffffc(%ebp)
0x4012c5 <main+37>:  call    0x401280 <f>
0x4012ca <main+42>:  movl    $0x1,0xffffffffc(%ebp)
0x4012d1 <main+49>:  sub     $0x8,%esp
0x4012d4 <main+52>:  pushl   0xffffffffc(%ebp)
0x4012d7 <main+55>:  push    $0x40129b
0x4012dc <main+60>:  call    0x402a30 <printf>
0x4012e1 <main+65>:  add     $0x10,%esp
0x4012e4 <main+68>:  mov     $0x0,%eax
0x4012e9 <main+73>:  leave
0x4012ea <main+74>:  ret
0x4012eb <main+75>:  nop
0x4012ec <main+76>:  nop
---Type <return> to continue, or q <return> to quit---q
Quit (expect signal SIGINT when the program is resumed)
(gdb) q

U:\GRIS\Artigos\bo>gcc teste.c -o teste
teste.c: In function 'f':
teste.c:8: warning: assignment from incompatible pointer type

U:\GRIS\Artigos\bo>_
```

Buffer Overflow - Exemplo

```
C:\WINNT\system32\cmd.exe
0x4012be <main+30>:    movl    $0x0,0xffffffff(%ebp)
0x4012c5 <main+37>:    call   0x401280 <f>
0x4012ca <main+42>:    movl    $0x1,0xffffffff(%ebp)
0x4012d1 <main+49>:    sub     $0x8,%esp
0x4012d4 <main+52>:    pushl   0xffffffff(%ebp)
0x4012d7 <main+55>:    push    $0x40129b
0x4012dc <main+60>:    call    0x402a30 <printf>
0x4012e1 <main+65>:    add     $0x10,%esp
0x4012e4 <main+68>:    mov     $0x0,%eax
0x4012e9 <main+73>:    leave
0x4012ea <main+74>:    ret
0x4012eb <main+75>:    nop
0x4012ec <main+76>:    nop
---Type <return> to continue, or q <return> to quit---q
Quit <expect signal SIGINT when the program is resumed>
(gdb) q

U:\GRIS\Artigos\bo>gcc teste.c -o teste
teste.c: In function 'f':
teste.c:8: warning: assignment from incompatible pointer type

U:\GRIS\Artigos\bo>teste
0

U:\GRIS\Artigos\bo>
```


Buffer Overflow – Técnicas e shellcodes

- Stack-based buffer overflow
- A estrutura de um shellcode
- Conhecendo melhor a Heap
- Heap-based buffer overflow
- Return-to-libc attack



Buffer Overflow – Stack-based buffer overflow

- As informações necessárias:
 - O tamanho do payload space
 - O intervalo na memória em que o payload space se encontra
- Com estas informações podemos executar um buffer overflow baseado em pilha:
 - Devemos inserir o código no formato de bytes no payload space dado.
 - Após o payload space, temos o nosso alvo: o Instruction Pointer.
 - Devemos reescrever o Instruction Pointer com o endereço do início do código inserido.
- Deste modo, quando a subrotina terminar, o programa irá continuar sua execução no endereço guardado pelo Instruction Pointer, ou seja: o código inserido.

Buffer Overflow - ShellCodes

- São códigos legíveis ao processador
 - Seu formato é em bytes
 - Geralmente são passados a um programa na forma de String
 - Na forma de String, cada byte é representado do seguinte modo : "\xNN", onde:
 - '\' significa que estamos passando um byte (não um caractere)
 - 'x' significa que o valor do byte está na base hexadecimal
 - NN é o valor do byte, na base hexadecimal
-
-

Buffer Overflow – ShellCodes:

Cuidados

- Como o shellcode é uma string, quando é passado para um programa neste formato, aquele será processado como uma string, então alguns cuidados em relação a bytes especiais devem ser tomados:
 - '\x00' ou simplesmente '\0': caractere terminador de string. Ao chegar em um byte nulo, o processamento da string pára, pois as funções que trabalham com string interpretam o '\0' como o seu fim.
 - '\x0A' e '\x0B' (\n): este byte, linefeed, pode quebrar o shellcode em duas linhas, o que representa outro problema.
 - Dentre outros...
- Para se contornar estes problemas, torna-se necessário o conhecimento da linguagem de máquina.

Buffer Overflow – ShellCodes:

Exemplo

- No caso de querermos construir um shellcode que imite a função `system()`, devemos seguir os seguintes passos:
 - Construir um programa que chame a função `system` com o argumento referente ao comando de sistema que desejamos executar
 - Analisar o código assembly do programa, por meio de uma ferramenta, como o `gdb`, conseguindo assim acesso ao código assembly da função `system()`
 - Com o `gdb`, podemos ter acesso aos bytes referentes a cada instrução pelo comando `"x/bx <nome_da_função>"`
- Com os bytes em mão, podemos contruir o ShellCode, concatenando os bytes em uma string, tomando os cuidados e adaptações necessários.

Buffer Overflow – ShellCodes:

Exemplo

- Com o shellcode pronto, devemos preparar a string maliciosa na seguinte organização:
 - Dados para encher o buffer até o início do shellcode
 - O shellcode em si
 - O valor para reescrever o Instruction Pointer com o endereço do início do shellcode
- Um problema comum: aleatoriedade da memória
 - Este problema pode ser contornado utilizando shellcodes bem menores que o payload space e precedidos de bytes NOP ('`\x90`')

Buffer Overflow – Conhecendo melhor a Heap

- A Heap é usada para alocação dinâmica de memória.
 - Seu tamanho é ajustável.
 - Binários possuem o Optional Header, que dá informações ao Sistema Operacional sobre o quão grande a Heap é esperada.
 - Cresce na direção oposta da pilha.
 - Além do espaço para dados do programa, um bloco na Heap possui outros dados de organização, como:
 - Tamanho do bloco
 - Espaço utilizado
 - Próximo bloco
 - Bloco anterior
 - Tais dados de organização são dependentes de implementação da Heap
 - Não há como definir limites para o espaço da Heap em um programa, pois este é variável, nem o processador nem o kernel são capazes de detectar overflows na heap.
-
-

Buffer Overflow – Heap-based

buffer overflow

- Esta é uma técnica muito mais difícil de se explorar
- Dificuldades:
 - Na heap não existem valores os quais o processador usa para saber qual a próxima operação.
 - A heap cresce na mesma direção em que escrevemos na memória, então não é possível reescrever dados de organização do bloco em que o programa grava, apenas blocos posteriores
 - Como a organização do Heap pode variar, o atacante deve conhecer a implementação utilizada pelo programa

Buffer Overflow – Heap-based buffer overflow

- Para explorar um overflow na heap, é necessário achar uma vulnerabilidade em alguma função interna de administração da heap, tal que esta função processe os dados em um bloco da heap.
 - Precisamos explorar a heap para sobrescrever os valores de administração de um bloco alvo e então inserir o shellcode
 - Os valores de administração devem ser alterados de forma a que este bloco seja processado diretamente por alguma subrotina, como a `free()`, ou de um modo em que a `free()` a processe de um meio especial, chamando uma outra subrotina.
 - Além do passo anterior, os dados no bloco devem ser reescritos de modo a causar um overflow no frame da subrotina, para assim direcionar a execução do programa para o shellcode inserido.
-
-

Buffer Overflow – Algumas questões

- Mas por que um shellcode na heap, se também temos que explorar uma vulnerabilidade na pilha?
 - Questão do espaço.
- Questão da pilha não-executável.
- Não importa onde você esteja, a libc estará sempre lá!
- A libc encontra-se em uma região restrita da memória.

Buffer Overflow – Return-to-libc attack

- A idéia aqui é basicamente a mesma do buffer overflow baseado em pilha.
 - Devemos, porém, reescrever o Instruction Pointer com o endereço de uma função da libc.
 - O endereço de uma função da libc pode ser descoberto com um programa teste.
 - Devemos então escrever na memória no seguinte esquema:
 - Reescrever o Instruction Pointer do frame alvo
 - O endereço de retorno da função da libc
 - Os argumentos da função alvo da libc
-
-

Buffer Overflow - Conclusões

- É um erro de implementação
 - Anti-vírus não apresentam soluções diretas aos problemas.
 - Existem programas que dependem de pilhas executáveis, como o JRE, por exemplo.
-
- Algumas dicas desenvolvimento sobre como se evitar transtornos com buffer overflows:
 - Usar bibliotecas seguras
 - Sempre fazer verificação de fronteiras, quando fazendo uso de strings
 - Utilizar uma implementação própria de heap

Buffer Overflow – Fim!

Dúvidas?

Obrigado!

raphaelpaiva@gris.dcc.ufrj.br

www.gris.dcc.ufrj.br

Raphael Duarte Paiva

Equipe de Pesquisa e Desenvolvimento

GRIS – DCC – IM - UFRJ
