UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PAULO ESTIMA MELLO

Uma Ferramenta Multiplataforma para Prevenção de Buffer Overflow

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, Julho de 2009.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Mello, Paulo Estima

Uma ferramenta multiplataforma para prevenção de buffer overflow / Paulo Estima Mello – Porto Alegre: Programa de Pós-Graduação em Computação, 2009.

55 pg.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2009. Orientador: Raul Fernando Weber.

1.Segurança. 2.Instrumentação 3.Buffer overflow 4.Erros de programação. I. Weber, Raul Fernando. II. Titulo

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço a todas as pessoas que contribuíram direta ou indiretamente para que esse trabalho se tornasse realidade. Em especial aos meus pais (Fernando e Sandra), irmãos (Eduardo e Fernando) e parentes próximos que sempre me estimularam mesmo nos momentos em que pensava ser impossível continuar. Um agradecimento especial à Laura que vem me acompanhando desde o início da jornada.

Por último, mas certamente não menos importante agradeço ao professor Raul F Weber pela orientação não somente neste trabalho como em todo meu percurso acadêmico.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASLR	Address Space Layout Randomization
ELF	Executable and Linkable File
PE	Portable Executable
PIN	Process INstrumentation
SEH	Structured Exception Handling
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta um método para prevenir as vulnerabilidades causadas por erros de programação insegura que, normalmente, é resultado da solução de um problema proposto ou do desenvolvimento de funcionalidade sem levar em consideração a segurança do sistema como um todo. Os erros de programação (no contexto da segurança de um sistema e não apenas da sua funcionalidade) são normalmente frutos da ignorância do programador sobre as vulnerabilidades apresentadas pelas suas ferramentas para construção de programas.

O estado da arte é brevemente apresentado demonstrando as soluções atuais em termos de proteção contra ataques de buffer overflow baseado em pilha. Soluções em tempo de compilação e pós-compilação por parte do sistema operacional são as mais comuns.

Neste escopo é demonstrada a solução proposta por um protótipo funcional que valida o modelo para uma série de aplicações em duas plataformas diferentes (Windows e Linux). A solução converge a instrumentação de aplicações com o uso de um repositório de endereços de retorno para prevenir o retorno de funções a endereços não legalmente especificados.

Testes do protótipo foram realizados em ambas as plataformas e mostraram a eficácia do protótipo prevenindo falhas em casos reais de buffer overflow baseado em pilha.

Palavras-Chave: segurança, instrumentação, buffer overflow, erros de programação.

A MULTIPLATFORM TOOL TO PREVENT BUFFER OVERFLOWS

ABSTRACT

This paper presents a method to prevent the vulnerabilities caused by insecure programming which, usually, is an outcome of taking into account only the solution of a proposed problem or the development of new functionalities disregarding security on development of the system as a whole. The programming mistakes (in the context of the system security despite the system's functionality) are usually a result of the unawareness of the programmed about the vulnerabilities contained on the tools they use to develop software.

The state of the art is briefly presented showing the current solutions related to preventing buffer overflows based on stack. Both compile time and post-compilation solutions (usually as part of the operating system) are the most widely used.

In this work the proposed solution is demonstrated by a functional prototype which validates the model for a set of applications in two different platforms (Windows and Linux). The solution converges process instrumentation with a return address repository to prevent a function from returning to an address not legally specified.

Testes of the prototype were performed in both platforms previously mentioned and have proved the correctness of the prototype by actually preventing exploitation on real case scenarios of real world applications.

Keywords: security, instrumentation, buffer overflow, programming error.

1 INTRODUÇÃO

A segurança na computação, atualmente, tem um papel fundamental no dia a dia das pessoas. Uma série de atividades são desempenhadas dependendo de sistemas seguros – ou o mais próximo que se possa chegar disso. Exemplos de sistemas que demandam segurança seriam bancos de dados governamentais, de dados bancários, de finanças pessoais, de documentos confidenciais, de chamadas telefônicas e tantos outros.

Existem inúmero sistemas que demandam segurança, tanto a nível comercial quanto pessoal. Uma empresa que deixa de lado a segurança em seus sistemas pode vir a ter sua imagem deturpada por pessoas mal intencionadas que se aproveitam dessa fragilidade para modificar, digamos, o layout de um site na web.

Um usuário que deixa de lado a segurança pode vir a ter sua senha bancária capturada, pois alguma pessoa pode ter se aproveitado desse desleixo para inserir serviços não desejados no ambiente do usuário como um interceptador de digitação no teclado ou de imagens em torno de lugares clicados com o mouse.

A segurança de sistemas é uma área que abrange tanto o comportamento do usuário em relação à sua interação com o computador quanto a qualidade do software utilizado no que tange a segurança. Se um usuário for descuidado é plausível que ele mesmo instale aplicações indesejadas no seu sistema ou que ele mesmo deixe o sistema vulnerável por alguma configuração mal feita. Por outro lado, as empresas desenvolvedoras de software podem vir a ser descuidadas e não aplicarem as políticas de segurança padrão, como disponibilizar configurações mínimas (que instalam somente os módulos essenciais do pacote) no seu software para reduzir a área de ataque ao software.

Adicionalmente, se um desenvolvedor não primar pela segurança, mesmo que não intencionalmente, seu software pode conter falhas que podem ser exploradas por pessoas que originalmente não deveriam e não teriam essa possibilidade caso técnicas de segurança de sistemas fossem aplicadas no ciclo de desenvolvimento do software. O ato de não visar segurança no desenvolvimento é conhecido por programação insegura (HOWARD, 2003).

São muitos pontos a serem analisados, porém, este trabalho se restringe ao estudo do último citado: programação insegura. Esta pode ocorrer por diversas razões, porém dois grandes potencializadores são, em primeiro lugar, os desenvolvedores não atentos aos aspectos de segurança e às vulnerabilidades das ferramentas utilizadas para desenvolvimento e o uso de ferramentas reconhecidamente inseguras.

Um grande exemplo de ferramenta reconhecidamente insegura é a linguagem C. Há vários aspectos inseguros na linguagem, sendo um dos mais conhecidos não ser

fortemente tipada. Isso permite que o usuário consiga interpretar um tipo de dado de várias formas, virtualmente, sem restrições. Além desse, há um pacote de funções padrões da linguagem que não primam pela segurança, como a cópia de arrays de bytes – strcpy -, a leitura da entrada padrão – fgets – entre outras.

Esses pacotes de funções foram extensivamente utilizados em muitos softwares, indicando que possivelmente nestes haja alguma falha de segurança. Para esse tipo de problema, há três soluções possíveis: descontinuar o uso do software, fazer uma remodelagem do código fonte aplicando técnicas para ampliar a sua segurança ou analisar seu código binário buscando possíveis falhas e tratando as mesmas de alguma forma.

A proposta é analisar e desenvolver uma técnica que dê apoio à segurança dos sistemas quanto à programação insegura ao maior número de falhas possível de forma automatizada. O maior atrativo dessa proposta seria a não alteração do sistema operacional, a possibilidade de utilização em software legado, a não alteração das aplicações protegidas e, obviamente, a proteção frente a falhas de segurança ocasionadas por programação insegura. Algumas dessas características estão presentes em algumas ferramentas, como StackGuard (COWAN, 1998), extensões do GCC (IBM, 2005), DEP (MICROSOFT, 2006) e PaX (PAX) porém não todas elas como é possível ver em (SILBERMAN, 2004).

Essas ferramentas citadas oferecem diferentes tipos de proteção, podendo algumas prevenir execução de código na pilha (fora do segmento de código), inserir marcas de proteção na pilha (canários), sobrescrever o ponto de retorno com o valor correto após um overflow e etc. As formas como cada ferramenta age depende da proposta da mesma, podendo ser visto dois tipos mais claros de ação: em tempo de compilação e em tempo de execução.

As ferramentas StackGuard e as extensões do GCC agem na compilação inserindo proteções antes do código de máquina ser gerado enquanto DEP (Data Execution Prevention) e PaX são embutidos no sistema operacional e podem vir a agir durante a execução do programa, prevenindo, por exemplo, a execução de código a partir da pilha.

A ferramenta proposta prevê ação em código já compilado – podendo proteger softwares legado ou que aplicações em que o código fonte não esteja mais disponível – e também sem alterações tanto na aplicação em si quanto no sistema operacional.

A falha de programação que este trabalho visa evitar é o buffer overflow baseado em pilha de forma a impedir o desvio da execução do programa original para algo não esperado como algum código malicioso que abre uma porta dos fundos no sistema atacado.

Os conceitos relacionados a esta falha serão apresentados na íntegra, assim como algumas abordagens já existentes e por fim a abordagem que fundamenta o trabalho: proteção via replicação dos endereços de retorno com utilização de instrumentação de aplicações.

No segundo capítulo será discutido o problema base a ser tratado pelo modelo proposto neste trabalho: buffer overflows baseados na pilha. No terceiro capítulo serão discutidas as proteções mais convencionais consolidadas no mercado e as alternativas estudadas antes de projetar o modelo final. No quarto capítulo a solução proposta é descrita, assim como o protótipo desenvolvido e as considerações relacionadas a ele. No

quinto capítulo considerações sobre o modelo, protótipo e os resultados dos experimentos são apresentados.

2 BUFFER OVERFLOW

Dentre os erros de programação mais conhecidos, o buffer overflow possui enorme destaque, pois é uma das formas de ataque mais praticadas e se mantém como estado da arte, em termos de ataques, até hoje. O primeiro ataque utilizando buffer overflows conhecido e registrado data de 1988. Nos primórdios da internet um worm (KEHOE, 1992) conhecido por "Morris worm" infectou aproximadamente 6000 computadores (cerca de 10% da internet na época) causando sérios problemas, sendo o pior deles o uso massivo de recursos (tanto de processamento quanto de banda, que na época tinha alto custo).

De acordo com (ARANHA, 2003), apesar de ser uma falha bem conhecida e bastante séria, que se origina exclusivamente na incompetência do programador durante a implementação do programa, o erro repete-se sistematicamente a cada nova versão ou produto liberados. Alguns programas já são famosos por freqüentemente apresentarem a falha, como o Sendmail, módulos do Apache, e boa parte dos produtos da Microsoft, incluindo o Internet Information Services (IIS). Mesmo software considerado seguro, como o OpenSSH, já apresentou o problema. Para se ter uma idéia, das vulnerabilidades já encontradas no ano 2003 e cadastradas no banco de dados da ICAT, 37% corresponde à buffer overflow explorável localmente ou remotamente (num total de 19 falhas). Segundo a mesma fonte, durante o ano de 2002, foram comunicadas 288 falhas também locais ou remotas, totalizando 22% das falhas reportadas naquele ano.

Mais recentemente o IE7 foi atacado com um buffer overflow em 2008 (MS, 2008). Isso mostra que apesar dessa falha de programação ter começado a se disseminar em 1988 o tema ainda é atual, e dado o volume de ocorrências dessa mesma falha muitos estudos se iniciaram para prevenir esse problema de forma automática, sem depender do programador diretamente.

Tecnicamente, um overflow ocorre quando algum dado é colocado num espaço menor do que o necessário. O overflow primário que se vê nos cursos de computação é conhecido por integer overflow. Um overflow de inteiro acontece quando, por exemplo, tentamos colocar numa variável inteira, sem sinal, de 16 bits (ou seja, tem capacidade para guardar números até 65535) o número 70000. Seria necessário uma variável de 17 bits para guardar o valor, então o bit que sobra é truncado e resta somente o excedente (4465). Nesse caso, um integer overflow aconteceu.

Da mesma forma, é possível expandir o conceito para strings. Se temos um espaço de 20 bytes para uma string em ASCII 8 bits, poderemos guardar nessa variável uma frase como: "Segurança é cara", mas não uma como: "Segurança é cara mas faz a

diferença". A última frase requer mais que os 20 bytes alocados inicialmente, resultando, então, em um excesso que deveria ser ignorado.

Como vimos no último exemplo, há situações em que o programador deve especificar claramente o tamanho das suas variáveis e também a forma como elas serão preenchidas para que não ocorram overflows e o programa se comporte da maneira esperada. Porém, há alguns programadores que não têm essa prática ou o conhecimento necessário para fazer isso; nesse ponto um fator determinante é a linguagem de programação utilizada.

A tabela 1, a seguir, apresenta as linguagens e os fatores de risco relacionados à segurança ou não no desenvolvimento utilizando a linguagem.

Tabela 2.1: Lista de linguagens e suas características no que tange segurança

Linguagem/ Ambiente	Compilada ou Interpretada	Fortemente tipada	Acesso direto à memória	Segura
Java	Ambos	Sim	Não	Sim
.NET	Ambos	Sim	Não	Sim
Perl	Ambos	Sim	Não	Sim
Python	Interpretada	Sim	Não	Sim
Ruby	Interpretada	Sim	Não	Sim
C/C++	Compilada	Não	Sim	Não
Assembly	Compilada	Não	Sim	Não
COBOL	Compilada	Sim	Não	Sim

Fonte: OWASP

De acordo com a tabela 2.1 as linguagens fortemente tipadas e que não permitem acesso direto à memória são mais seguras (ou seja, são menos vulneráveis a erros de programação).

Duas das linguagens mais comuns para programação hoje são C/C++ e Java, e a mais popular, para sistemas de grande porte, é C/C++. Essa última possui uma característica muito interessante: liberdade no desenvolvimento, mesmo que isso implique em erros de programação. Essa liberdade é o que causa a maioria das falhas de programação.

Em se tratando de C/C++, há uma confiança do compilador no programador; ou seja, o código desenvolvido é compilado em instruções de máquina mesmo que eventualmente seja usada uma construção propensa a falhas, como veremos nas próximas seções.

Um buffer overflow normalmente resulta em falta de segmentação ou as variantes dessa mesma falta nos diferentes sistemas operacionais. O ataque propriamente dito, baseado em buffer overflow, faz uso dessa falha para afetar o sistema alvo do ataque.

Para ilustrar a idéia o seguinte código é apresentado (IZIK, 2006):

```
/*
* vuln.c, Classical strcpy() buffer overflow
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[256];
    strcpy(buf, argv[1]);

    return 1;
}
```

O código acima é o exemplo clássico de buffer overflow causado pela fraca tipagem da linguagem C. Este programa copia o primeiro argumento passado ao executável gerado por um compilador para este código para a variável buf.

Apesar de parecer simples e convencional a programação dessa aplicação está incorreta. Não há checagem de limites no tamanho da variável buf ao copiar o conteúdo de argv[1], ou seja, se argv[1] possuir 300 caracteres, os 256 bytes de buf serão preenchidos e, também, mais 44 adicionais adjacentes à buf. Caso um inteiro tivesse sido declarado acima de buf o conteúdo de argv[1] teria sobrescrito também este inteiro.

Um buffer overflow não planejado resulta em potencial falha na aplicação. Um buffer overflow planejado pode resultar em execução de código arbitrário. Para facilitar a compreensão do processo de exploração desta falha é interessante analisar a estrutura de um processo em memória na arquitetura x86.

2.1 Conceitos básicos da arquitetura x86

O processador x86 utiliza palavras de 32 bits em sua arquitetura. Atualmente existem novas arquiteturas com palavras maiores, como o x86-64 ou o Itanium, ambos de 64 bits. Essas arquiteturas se tornarão bastantes populares no futuro, mas a maior parte do mercado ainda pertence às arquiteturas de 32 bits. A arquitetura predominante hoje é a x86.

2.1.1 Registradores

A arquitetura x86 é constituída de unidades aritméticas e lógicas e unidades de execução (entre outras partes) que controlam a execução de programas e operam valores dentro do processador. Esses valores são guardados em uma memória interna de rápido acesso (muito mais que a cachê L1) chamada registrador. Estes registradores são divididos em: registrador de propósito geral, registradores de segmento e registradores de índice. Para este estudo são fundamentais os registradores de propósito geral e três registradores de índice em especial (ESP, EBP e ESP). A tabela 2.2 apresenta os registradores de interesse:

Tabela 2.2: Principais registradores da arquitetura x86

32 bits	16 bits	8 bits
EAX	AX	AH, AL
EBX	BX	BH, BL
ECX	CX	CH, CL
EDX	DX	DH, DL
ESP	SP	
EBP	BP	
EIP	IP	

Fonte: Adaptação de Rosielo (2004). p. 3

Os registradores de 32 bits podem ter seus conjuntos de bytes acessados independentemente de acordo com a tabela abaixo. O registrador EAX possui 32 bits sendo que o registrador AX representa seus últimos 16 bits, o registrador AH representa os primeiros 8 bits do registrador AX e o AL os últimos 8 bits do registrador AX.

Tabela 2.3: Posição dos registradores com EAX como base

8bits	8bits	8bits	8bits	
				EAX
				AX
				AH
				AL

Os registradores ESP, EBP e EIP possuem funções bastante importantes no fluxo de execução de um programa. ESP é o registrador que controla a pilha de execução das aplicações e como ela funciona será apresentado nas próximas seções. EBP é o registrador que controla o segmento da pilha utilizado por uma função em um momento. Ele é ajustado a cada execução e sua função principal é servir de ponto fixo para identificação dos parâmetros da função e das variáveis locais. EIP controla o fluxo de execução da aplicação, contendo sempre a posição na memória da próxima instrução a ser executada.

2.1.2 Instruções

Os primeiros modelos baseados na arquitetura x86 foram o 80186, 80286 e o 80386. Todos os modelos baseados na arquitetura x86 possuem praticamente 100% de compatibilidade retroativa. Isso quer diz que, em tese, um programa desenvolvido para o processador 80386 deve rodar no modelo mais novo do Core i7, por exemplo. Para este estudo é interessante um subconjunto das instruções presentes no primeiro modelo x86 (8086), como mov, xor, *push*, *pop*, *int*, *call* e *ret*.

Para apresentar as instruções será usada a notação AT&T que tem como padrão:

- O operando destino é o último;
- % referenciam registradores;
- $ são utilizados para constantes inteiras.

A tabela 4, a seguir, apresentada as instruções de interesse deste trabalho no que tange tanto shellcode quanto chamada e retorno de funções:

Tabela 2.4: Instruções da arquitetura x86

Instrução	Exemplo	Ação
mov	mov $0x80, %eax	Move um valor ($0x80) para %eax
xor	xor %eax, %eax	Opera um XOR entre %eax e %eax
push	push %eax	Empilha %eax (decrementando %esp)
pop	pop %eax	Desempilha o topo da pilha para %eax
int	int $0x80	Sinaliza interrupção de software $0x80
call	call $0x1	Chama a função localizada em $0x1
ret	ret	Retorna da função corrente

Existe uma infinidade de outras instruções na arquitetura x86 que foram adicionadas a cada modelo novo de processador que chegou ao mercado. Adicionalmente, existem alguns tipos de endereçamento que não faz parte da análise aqui proposta e por essa razão serão deixados de lado, mas é possível encontrar uma vasta documentação sobre o assunto em Intel (b).

A base deste trabalho encontra-se no funcionamento de duas instruções mencionadas acima: *call* e *ret*. Tais instruções estão fortemente envolvidas na forma como buffer overflows são explorados como forma de ataque a aplicações vulneráveis. Para um melhor entendimento de como essas instruções interagem possibilitando a exploração da vulnerabilidade nas próximas seções serão apresentadas tanto a estrutura do processo em memória quanto as convenções de chamada de função mais comuns.

2.1.3 Estrutura dos processos em memória

Por Aranha (2003), os processos em execução são divididos em quatro regiões: texto, dados, pilha e heap.

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

A região de dados contém as variáveis globais e estáticas do programa.

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

A pilha consiste em uma seqüência de frames que são colocados no topo quando uma função é chamada e retirados ao final da execução. Um frame contém os parâmetros para a função, suas variáveis locais, e os dados necessários para recuperar o frame anterior, incluindo o valor do ponteiro de instrução no momento da chamada de função.

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

para os endereços baixos, com o ponteiro de pilha (registrador ESP) apontando para o último endereço da pilha.

A heap permite a alocação dinâmica de memória por meio de chamadas da família malloc(3). A área de heap cresce em sentido oposto à pilha e em direção a esta.

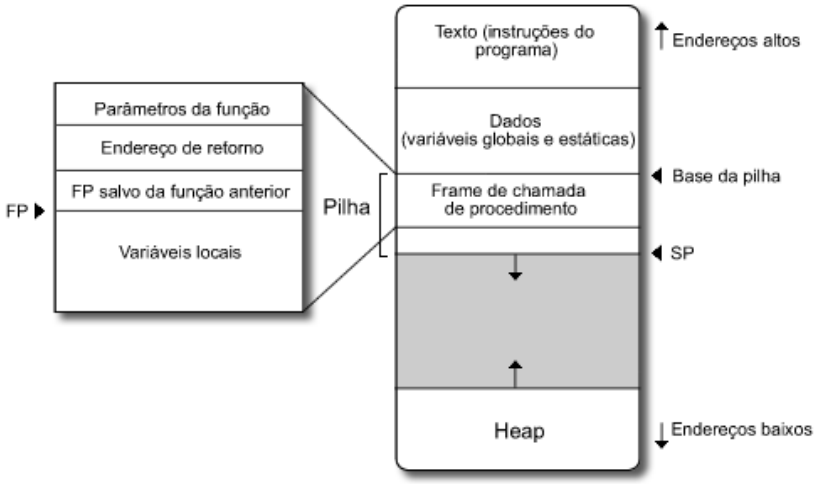

Figura 2.1: Estrutura de um processo em memória (Aranha, 2003)

2.1.4 Convenções de chamada de função

Nos primórdios da computação não havia a diversidade de fornecedores tanto de hardware quanto de software que existem na atualidade e por isso não havia múltiplas formas de pensar no mesmo problema no que tange chamadas de função. O mesmo fornecedor do hardware provia o software base a ser utilizado com o mesmo (sistema operacional e compiladores).

Esse modelo de negócio não reforçou a utilização de padrões gerando, por assim dizer, uma quantidade grande de formas de chamada de função não havendo então nenhum padrão estabelecido.

Atualmente há múltiplos fornecedores tanto de hardware (HP, IBM, Dell, Apple, Intel, AMD entre outros) quanto de software (Microsoft, Oracle, Borland, comunidades de software livre, HP, IBM, Dell, Apple entre vários outros). Todas essas empresas procuram manter uma certa compatibilidade entre suas tecnologias (tanto de hardware quanto de software), ou seja, aplicações da Oracle compiladas com o compilador da Borland deve executar no sistema operacional da Microsoft que roda sobre o processador da Intel em um servidor da HP. O que faz com que toda essa diversidade funcione em razoável harmonia são os padrões, protocolos e convenções. Muitos detalhes sobre interoperabilidade podem ser vistos em Fog (2009).

As convenções de chamada de função suprem uma necessidade relativa à forma como as funções são chamadas tanto dentro da mesma aplicação quanto à integração entre camadas de software com as bibliotecas dinâmicas.

È possível dividir as convenções de chamadas de funções em dois grandes grupos (X86-a): **caller clean-up** e **callee clean-up**. Na primeira a função que chamou outra faz a limpeza necessária para continuar sua execução. Na última a função chamada faz a limpeza necessária para que a função que a chamou continue executando.

Para que o conceito de limpeza fique claro é necessário entender o que acontece quanto uma função é chamada. A forma como uma função é chamada determina, também, a forma como ela deverá retornar ao chamador.

2.1.4.1 A chamada de uma função: call

A instrução *call* toma um endereço como parâmetro (que é para onde a instrução irá desviar) e, basicamente, empilha o endereço seguinte à instrução call (endereço conhecido como **endereço de retorno**) e desvia para o parâmetro da instrução.

Para elucidar o conceito é necessário comentar sobre a arquitetura x86. Os processadores que seguem a arquitetura x86 implementam uma organização de memória para chamadas de procedimento como a mostrada na figura 2.2, a seguir:

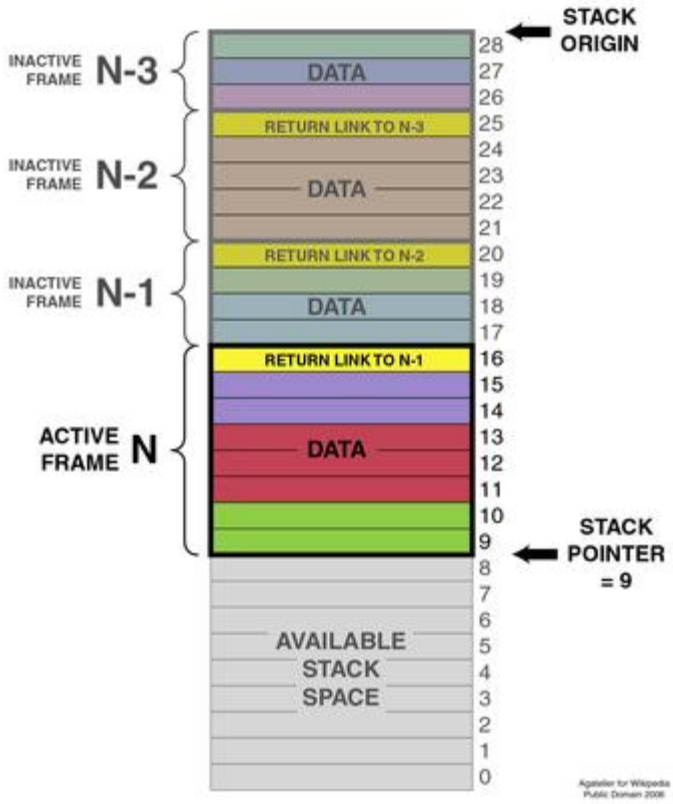

Figura 2.2: Estrutura da pilha em memória (STACK-b)

Na figura 2.2 são mostradas quatro funções aninhadas, ou seja, a função que está utilizando o frame N-3 fez uma chamada à função que está utilizando o frame N-2 que fez uma chamada à função que está utilizando o frame N-1 que enfim faz uma chamada a função que está com o controle do processo no momento: a que está utilizando o frame N.

É possível notar que sempre no início de um segmento de pilha da função é salvo o ponto de retorno para que o processador saiba para onde voltar, ou seja, qual instrução executar após a chamada da função que está no momento usando o frame.

2.1.4.2 O retorno de uma chamada: ret

A instrução *ret* desempilha o endereço de retorno empilhado pela instrução *call* e desvia para o mesmo retornando à execução da função que originalmente a chamou. É

necessário observar que dada a forma de funcionamento da pilha para que *ret* funcione apropriadamente o endereço de retorno deve estar no topo de pilha.

Isso implica no fato de que alguma função, seja a chamadora ou a chamada, deverá desempilhar os valores contidos na pilha. No caso do frame N da figura 2 alguma função deverá desempilhar tudo desde o endereço 9 até o 16.

2.1.4.3 Organização dos parâmetros da função

Normalmente funções tomam parâmetros (ou argumentos) que são utilizados na sua computação, gerando um resultado útil para o programa que a utiliza. Algumas linguagens (como pascal) chamam funções que não retornam diretamente um valor por procedimentos, mas por simplicidade nos referiremos a todas as funções ou procedimentos por funções.

Conforme visto na seção anterior, todo processo possui uma pilha que é utilizada para guardar os parâmetros das funções e as variáveis locais das mesmas; adicionalmente conta-se com um "ponteiro de frame" (registrador EBP) que aponta para um endereço fixo no frame. A princípio, variáveis locais podem ser referenciadas fornecendo-se seus deslocamentos em relação ao ponteiro de pilha (registrador ESP). Entretanto (IBM, 2005), quando palavras são inseridas e retiradas da pilha, estes deslocamentos mudam. Apesar de em alguns casos o compilador poder corrigir os deslocamentos observando o número de palavras na pilha, essa gerência é cara. O acesso a variáveis locais a distâncias conhecidas do ponteiro de pilha também iria requerer múltiplas instruções. Desta forma, a maioria dos compiladores utiliza um segundo registrador que aponta para o topo da pilha no início da execução da função, para referenciar tanto variáveis locais como parâmetros, já que suas distâncias não se alteram em relação a este endereço com chamadas a PUSH e POP. Na arquitetura Intel x86, o registrador EBP é utilizado para esse propósito. Por causa da disciplina de crescimento da pilha, parâmetros reais têm deslocamentos positivos e variáveis locais tem deslocamentos negativos a partir de FP.

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

2.1.4.4 Caller clean-up

Neste modelo quem chama uma função deve limpar a pilha após o retorno da mesma. Este modelo é interessante para funções que possuem argumentos variáveis (funções que fazem uso de varargs) como as variações de printf.

A convenção de chamada de função mais representativa deste modelo é a **cdecl**. Neste modelo os parâmetros de funções são empilhados da direita para a esquerda de suas declarações. Os valores de retorno de função (sejam ponteiros ou valores significativos) são retornados no registrador EAX. As formas de chamada de função que representam esse grupo são: cdecl, syscall e optlink.

2.1.4.5 Callee clean-up

Neste modelo a função chamada faz a limpeza da pilha. Normalmente funções com número fixo de argumentos utilizam este modelo. As chamadas e retornos de funções que aplicam este modelo são facilmente reconhecidas ao analisar o assembly das mesmas por utilizarem o parâmetro da instrução *ret* que indica quantos bytes da pilha devem ser liberados. Isso só é possível quando a função sabe de antemão o número de parâmetros que possui (ou seja, não utiliza suporte a número variável de argumentos) conseguindo assim calcular a quantidade de posições da pilha que devem ser liberadas.

Uma vez sendo fixo o número de parâmetros é possível não somente utilizar a pilha para passagem dos mesmos como utilizar também os registradores. Como representantes dos padrões gerados ao longo do tempo onde a limpeza da função é feita pela função chamada são citadas as formas: pascal, register, stdcall, fastcall e safecall.

Cada convenção citada modifica a forma como os registradores são usados no sentido de determinar algum registrador específico para o valor retornado pela função ou quais registradores estão livres para terem seus valores alterados.

2.1.4.6 Prólogo padrão de funções

Mesmo havendo uma ampla gama de convenções para chamada de funções sempre houve um padrão para o prólogo de chamada de funções. O procedimento padrão é empilhar o registrador EBP (que contém o ponteiro de segmento da função que chamadora), mover o registrador ESP para EBP (passar o topo da pilha para o registrador de ponteiro de segmento) e reservar espaço para as variáveis locais (somente quando existem).

O seguinte código assembly representa o prólogo de uma função, assumindo X como o número de bytes necessários para acomodar as variáveis locais (mantendo um alinhamento de 4 bytes, ou seja, um array de 5 bytes possui alocação de 8 bytes):

```
Push %ebp

Mov %esp, %ebp

Sub X, %esp
```

Para simplificar o prólogo das funções existe a instrução *enter,* que dados alguns parâmetros, desempenha a função das três instruções acima. Há relatos de problemas de performance com a instrução e provavelmente por isso ela não é extensivamente utilizada.

2.1.4.7 Epilogo padrão de funções

Os epílogos de funções, diferentemente do prólogo, são múltiplos. Uma função pode conter um epílogo por possibilidade de saída (em um código C, um epílogo por return). O seguinte código assembly representa o epílogo de uma função, assumindo X como o número de bytes necessários para acomodar as variáveis locais:

```
Mov %ebp, %esp

Pop %ebp

Ret X
```

Analogamente ao prólogo, a instrução *leave* desempenha a função das duas primeiras instruções (fazer o ajuste do segmento da função para a qual está sendo retornado).

2.1.4.8 Otimizações e como o código é afetado

É importante ressaltar que mesmo que os compiladores têm liberdade de alterar o código sendo gerado. Os compiladores para linguagem Delphi, em geral, utilizam as instruções *enter* e *leave* enquanto os compiladores GNU GCC e Microsoft Visual Studio não fazem uso.

A forma de entrada funções deve ser padronizada para que componentes de aplicações diferentes consigam utilizar funções, porém, partes internas da função podem ser otimizadas dependendo do processador de acordo com a vontade do distribuidor do compilador.

2.2 Exploração de uma falha do tipo buffer overflow

Conforme visto nas últimas seções os ataques via buffer overflow ocorrem por um conjunto de fatores:

- inaptidão do programador desenvolvendo o programa;
- arquitetura do processador x86;
- fraca tipagem e checagem de limites nos vetores nas linguagens de programação;

A seguir os tipos interessantes de ataques via buffer overflow relacionados a este trabalho serão apresentados. A forma como a junção da arquitetura do processador x86, de uma linguagem fracamente tipada e um programador mal orientado são combinadas para formar uma falha explorável na aplicação será apresentado.

2.2.1 Stack based overflow

Este tipo de ataque utiliza o conjunto de fatores citados na seção anterior que resultam na possibilidade de sobrescrever o endereço de retorno de uma função com um valor arbitrário podendo, eventualmente, escrever no endereço de retorno da função uma posição de memória conhecida que contem código executável escrito pelo atacante.

Para elucidar a situação, o seguinte código de Stack (a) esclarece o que acontece na pilha durante um overflow:

```
#include <string.h>

void foo (char *bar)
{
    char  c[12];

    memcpy(c, bar, strlen(bar));  // sem checagem do tamanho de c
}

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

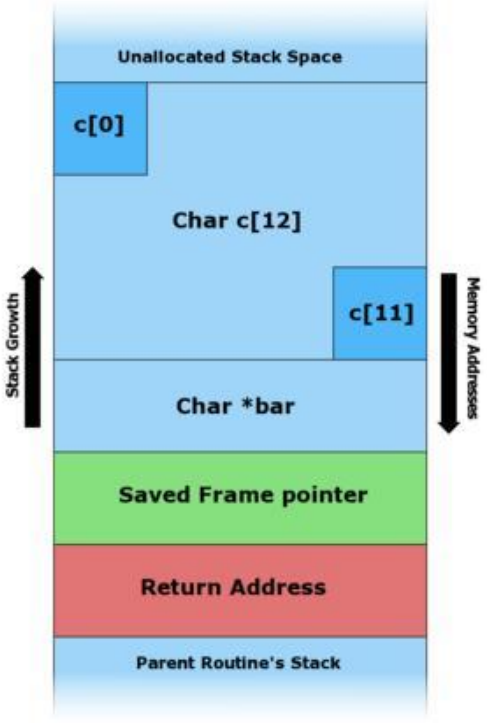

Figura 2.3: Estado da pilha para o código anteriormente citado (STACK-a)

È uma característica da arquitetura x86 o fato de que a pilha "cresce" num sentido oposto ao do aumento das posições de memória. Ou seja, a pilha começa num endereço alto da memória e cresce em direção aos menores.

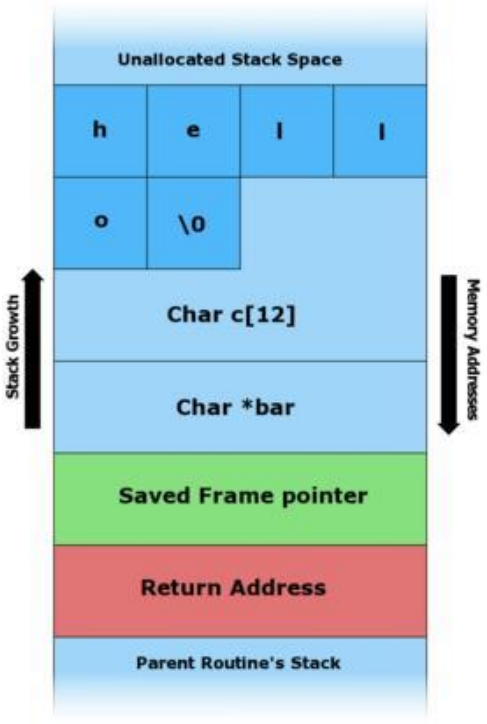

Figura 2.4: Estado da pilha ao executar foo("hello") (STACK-a)

De acordo com a figura 2.4, os endereços de memória crescem em sentido oposto à pilha e é aqui que reside o maior problema nesse tipo de ataque: caso tenhamos algum

dado numa posição x + 1 da pilha e tentemos usar mais espaço do que o alocado para a posição x + 1, o excesso de dados será colocado na posição x. O tamanho desse excesso determina quantas posições da pilha serão sobrescritas.

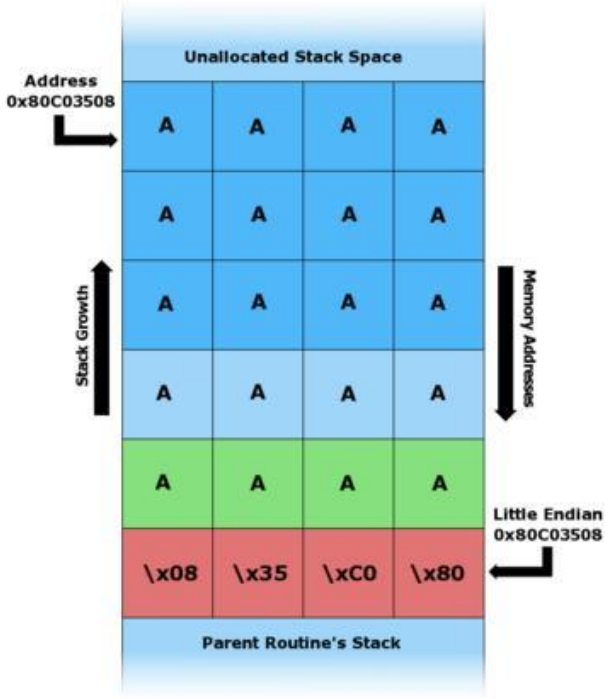

Figura 2.5: Estado da pilha ao executar foo com uma string maior que o buffer (STACK-a)

A figura 2.5 demonstra a estrutura básica do problema. Dada a estrutura de memória fica claro ser possível executar um trecho de código arbitrário. Para isso basta se valer de que, antes das declarações de variáveis locais na pilha estará o endereço de retorno da função. Dessa forma, na figura 2.5, ao executar foo com uma string grande o suficiente – no caso foo("AAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80") - o buffer sofreu um overflow sobrescrevendo a pilha desde o início do buffer até o endereço de retorno da função. A string é constituída por caracteres A e o endereço de retorno pelo endereço inicial do buffer - 0x80C03508 -. Ao terminar a execução da função o programa pulará para o endereço de buffer e executará o que estiver lá, neste caso A que no processador x86 não corresponde a uma instrução válida.

Este tipo de ataque é conhecido por "stack smashing" ou "stack based overflow" (ONE). Essa classe de ataques faz uso de técnicas de criação de códigos executáveis que podem ser usados para preencher algum buffer do sistema - shellcode (ROSIELO, 2004) -, a ser usado, posteriormente, como endereço de retorno. Caso um buffer venha a ser preenchido por um shellcode (string bem formada contendo instruções de máquina com a finalidade de abrir um shell ou executar qualquer outra função desejada pelo atacante) e uma função tenha seu endereço de retorno alterado para esse buffer o código contido nele será executado como uma instrução normal do programa atacado. Exemplos e a técnica para montagem de shellcode será mostrada numa seção deste mesmo capítulo.

Eventualmente alguns sistemas podem implementar defesas (serão vistas no próximo capítulo) que não permitem que sejam executadas instruções contidas na pilha

(onde ficam as variáveis locais das funções). Por essa razão surgiu uma nova subclasse de ataques conhecidos por “return to libc” (SHACHAN, 2004) que chamam funções contidas na libc (biblioteca que é linkada na maioria dos programas compilados utilizando a linguagem C) ou alguma outra função já carregada em memória para executar funções não desejadas como será mostrado na seção correspondente a este ataque.

De acordo com o visto até então, para um ataque ser bem sucedido é necessário:

1. Encontrar um buffer que possa sofrer overflow
2. Montar um shellcode para executar alguma função (no sentido geral, não necessariamente uma função ou procedimento)
3. Encontrar o endereço do buffer (que contém o shellcode) para que o endereço de retorno da função (uma vez sobrescrito) desvie para o shellcode em vez do endereço esperado

Originalmente os sistemas operacionais eram menos preparados para resistir a esse tipo de ataque. Essa falta de preparação os deixava altamente suscetíveis tornando viáveis ataques de baixa complexidade.

Para 1) ser evitado os compiladores usam técnicas que reduzem a incidência de má programação com buffers, como canários (será discutido no próximo capítulo) que são essencialmente valores alocados no início da função que são verificados ao final da execução da função. Caso o valor tenha sido alterado houve um overflow em algum buffer alocado na função.

O ponto 2) possui algumas dificuldades inerentes, como a própria montagem e transformação do código em uma string, técnicas de mascaramento de zeros (zero corresponde ao byte que encerra strings na linguagem C) e restrições de tamanho.

Nas versões mais antigas de sistemas operacionais 3) era obtido uma única vez por tentativa e erro, ou seja, dentro de um mesmo processo em todas as execuções o buffer era alocado no mesmo endereço por que a pilha iniciava sempre no mesmo endereço. Ao descobrir o endereço do buffer por tentativa e erro o ataque era executado posteriormente com 100% de sucesso. Com a evolução dos mecanismos de defesa dos sistemas operacionais uma série de melhorias foram implementadas, mas no que tange este tópico uma proteção em especial aumentou a dificuldade deste ataque: Address Space Layout Randomization (ASLR).

A técnica ASLR altera a organização de algumas áreas de memória no sentido de introduzir aleatoriedade na mesma. No que se refere à pilha o endereço inicial desta é alterado a cada execução da aplicação. Isso reduz drasticamente a probabilidade de um endereço de buffer se repetir entre execuções da aplicação.

2.2.2 Retorno à libc

Ataques de retorno à libc são alternativas de exploração de buffer overflow que não requer injeção de código na pilha. Apesar do nome o ataque não se restringe a funções da libc podendo ser aplicado com qualquer função que esteja em memória.

A base do ataque é a mesma do baseado em pilha, porém, a instrução *ret* é utilizada para chamar uma função já presente em memória. O formato da string sendo injetada difere bastante da utilizada quando código é injetado:

- Não há shellcode envolvido;

- Os parâmetros de função a ser chamada via *ret* são informados;

O valor que sobrescreve o endereço de retorno da função, em vez de ser o endereço de início do buffer sobrescrito (que contém o shellcode), se torna o endereço de alguma função em memória. O nome retorno à libc se consolidou pelo uso das chamadas contidas na libc, como *system* ou *execve*. A chamada à *system* é bastante simples pelo fato da função tomar somente um parâmetro. Uma vez que esse parâmetro esteja no lugar correto (normalmente EBP substraído algum deslocamento) a chamada vai ser bem sucedida.

No caso da chamada à função *system* (que executa um programa determinado pelo único parâmetro passado – uma string que representa o caminho completo ao programa), em especial, somente um parâmetro é passado, facilitando a exploração da falha.

O grande problema é que essas funções normalmente tomam ponteiros como parâmetro às funções. No caso de *system* um ponteiro a uma string é passado tornando, então, necessário carregar uma string na memória contendo o caminho do programa a ser executado e o endereço da mesma precisa ser conhecido. Adicionalmente, esse endereço precisa ser visível ao programa em execução. Essa é a principal razão pela qual esse ataque normalmente é executado localmente na máquina. Para resolver esse problema as variáveis de ambiente são idéias possibilitando que um valor na memória contenha uma string, e mais, permitindo que esta seja vista pelos programas em execução (somente) após sua criação.

2.3 Shellcode

Os códigos feitos especialmente, baseados tanto na arquitetura do processador (x86) quanto no sistema operacional, para explorar falhas de programação são chamados shellcodes (ou bytecodes).

Uma vez localizado um ponto de falha num aplicativo é necessário produzir código assembly a ser executado para que a falha seja explorada. O ponto de falha localizado pode ter um tamanho que varia de alguns bytes a vários kilobytes então os códigos produzidos para este fim tendem ser a tão pequenos quanto possível para se ajustar a qualquer tamanho disponível.

Os passos comuns para a produção de um shellcode envolvem:

1. Definir o que o código irá fazer
2. Projetar essa função utilizando uma linguagem de programação
3. Produzir o assembly referente ao código (como um compilador)
4. Transformar o código assembly em código de máquina
5. Transformar esse código de máquina em uma string **sem** zeros

Tais passos soam muito com a tarefa que um compilador faria e de certa forma o são. Alguns desenvolvedores de shellcode utilizam código gerado por compiladores e apenas os ajustam para que sirvam para esse fim. Estes ajustem são necessários por razões relacionadas à forma como esse código é utilizado. Estes códigos são utilizados por aplicações que já foram inicializadas pelo sistema operacional, ou seja, não é possível definir novos dados para a seção de dados do processo. Strings definidas em programas compilados normalmente são armazenadas na seção de dados. Outra limitação é causada pela necessidade do código não conter o terminador null (ou seja,

0). Como strings são injetadas num processo em execução terminadores devem ser evitados para que toda string seja processada pela aplicação.

Para ilustrar o método de desenvolvimento de shellcodes, um shellcode será produzido visando um ataque à uma aplicação rodando no sistema operação Linux.

2.3.1.1 Projetando a solução

O desenvolvimento de um shellcode normalmente começa com o projeto da solução para um determinado problema. Assumindo que o objetivo deste shellcode seja abrir uma shell - linha de comando - no Linux o seguinte código em C representa o shellcode em linguagem inteligível:

```
int main(int argc, char *argv[]) {
    write(0, "You got hacked!", 15);
    exit(0); // Terminar silenciosamente
}
```

O programa acima escreve no descritor de arquivo apontado pelo primeiro parâmetro de *write* a frase "You got hacked!". O último parâmetro indica a quantidade de caracteres a serem escritas. A última chamada, *exit*, resulta num fim normal para a aplicação em vez de uma falha de segmentação caso deixássemos o código seguir a execução após a chamada *write*. Essa falha de segmentação aconteceria pois o shellcode terminaria e o processador iria tentar interpretar algum dado da pilha como instrução.

O código possui duas chamadas de sistema: *write* e *exit*. No sistema operacional Linux as chamadas de sistema são feitas através de interrupções de software. No registrador %al é armazenado o código da interrupção. Para *write* o código é 4 e para *exit* o código é 1.

Em termos de código assembly, para gerar um shellcode a partir do código C anteriormente mencionado, devemos:

1. Executar a escrita - *write*
2. Executar a saída - *exit*

Para executar um *write*:

1. Mover o número do descritor de arquivo para %ebx
2. Mover um ponteiro para o que *write* deverá escrever para %ecx
3. Mover o tamanho do conteúdo – 15 - a ser escrito para %edx
4. Mover o código - 4 - da interrupção de software para %eax
5. Executar a interrupção $0x80 - interrupção de software

Para executar um *exit*:

1. Mover o código de saída - o que o main da aplicação retorna - para %ebx
2. Mover o código – 1 - da interrupção de software para %eax
3. Executar a interrupção 0x80 - interrupção de software

Através desta especificação o código assembly deve ser montado. Durante a montagem zeros devem ser evitados. Portanto, para finalizar uma string, em vez de mover um zero para uma determinada posição, a instrução xor deve ser utilizada. Ela tem o mesmo efeito de um "mov $0x0, X". Outro ponto especialmente importante a ser levado em consideração é a largura dos registradores sendo operados. Uma instrução que lida com registradores de 32 bits, como %eax, possuirá zeros em seu código final

caso o valor sendo movido o possua. Para ilustrar a idéia o código e sua representação final (no shellcode) são apresentados:

mov $0x4, %eax → \xb8\x04\x00\x00\x00

Como é possível ver, o final \x00\x00 resulta em dois bytes zero, ou seja, terminadores da string. Para resolver o problema um conjunto de instruções que produzem o mesmo resultado sem a representação final com zeros deve ser usado. Um exemplo seria:

xor %eax, %eax → \x31\xc0

mov $0x4, %al → \xb0\x04

2.3.1.2 Codificando a solução

Para executar a função *write*, devemos primeiramente, os registradores envolvidos devem ser zerados:

Xor %eax, %eax

Xor %ebx, %ebx

Xor %ecx, %ecx

Xor %edx, %edx

A string a ser escrita deve ser alocada na pilha – conforme visto na seção passada, dada a forma como a pilha funciona a string deve ser empilhada de trás para frente:

Push %eax

Push $0x216465 # “!dek”

Push $0x6b636168 # “kcah”

Push $0x20746f67 # “ tog”

Push $0x20756f59 # “ uoY”

O registrador que aponta para o topo da pilha %esp agora serve como ponteiro para a string a ser utilizada pela chamada a *write*:

Mov %esp, %ecx

Após zerar o registrador %edx, podemos mover o tamanho da string – 15 - para os bits mais baixos dele através do registrador %dl que representa os bits mais baidos – low - de %edx:

Mov $0xF, %dl

Após zerar o registrador %eax podemos mover o identificador - 4 - da chamada write para %al:

Mov $0x4, %al

Uma vez com os registradores possuindo os parâmetros para a chamada da interrupção podemos sinalizar para o sistema operacional a interrupção de software:

Int $0x80

Para executar a função *exit* devemos zerar os registradores envolvidos:

Xor %eax, %eax

Xor %ebx, %ebx

Mover o identificador da interrupção – 1 - para %al:

Mov $0x1, %al

E então sinalizar para o sistema operacional a interrupção de software:

Int %0x80

2.3.1.3 Transformando em código de máquina

O código programado em assembly deve ser compilado como tal para gerar um arquivo executável contendo código de máquina. Existem inúmeras maneiras de gerar código de máquina, mas uma comum e disponível em qualquer sistema de desenvolvimento Linux é utilizar o c++ em conjunto com a diretiva __asm__. Um exemplo disso, levando em consideração o código anteriormente apresentado é mostrado na figura 2.6:

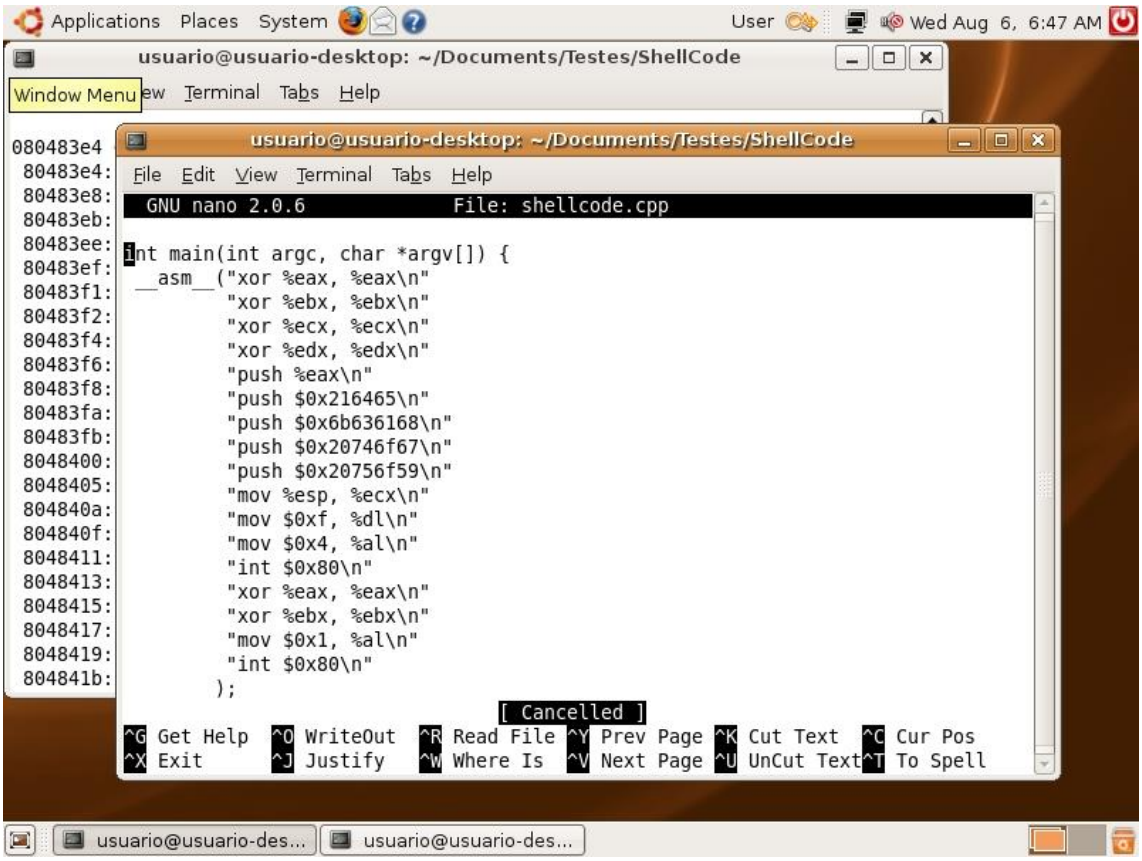

Figura 2.6: Ambiente de desenvolvimento de shellcode: NANO e C++ com __asm__

Após gerar o executável contendo o código de máquina baseado no código assembly para o shellcode é preciso extrair uma string que possa ser utilizada para explorar um buffer overflow. Nesse ponto do desenvolvimento do shellcode entra o objdump. Na figura 2.7, a seguir, é apresentado, no ambiente de desenvolvimento, a saída do comando objdump:

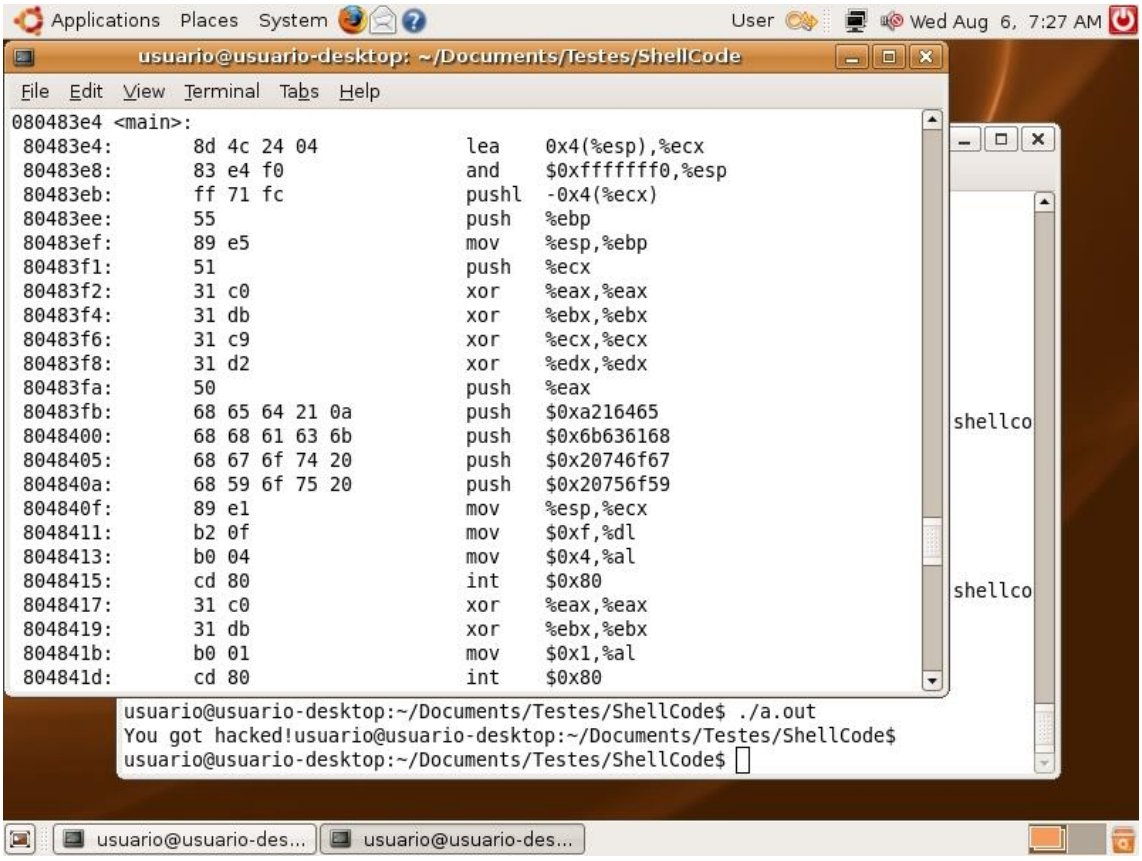

Figura 2.7: Ambiente de desenvolvimento de shellcode: objdump

Na figura 2.7, as instruções anteriores à 0x80483f2 correspondem á inicialização da função main, depois de 0x80483f2 até 0x804841d está o código assembly projetado anteriormente. Entre os endereços e o código assembly das instruções está a representação em hexadecimal do código binário da aplicação. Este é o código a ser injetado numa aplicação vulnerável.

Para formar a string a ser injetada deve-se concatenar os números hexadecimais em uma única string, em seqüência de acordo com a saída do objdump. Portanto, a string a ser injetada no código vulnerável é:

"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x65\x64\x21\x0a\x68\x68\x61\x63\x6b\x68\x67\x6f\x74\x20\x68\x59\x6f\x75\x20\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0\x01\xcd\x80"

A string acima, se injetada corretamente executará num processador x86 causando a impressão na tela da frase "You got hacked!" e uma saída normal com código 0 da aplicação.

Para garantir um funcionamento correto do shellcode normalmente ele é testado com uma aplicação simulando um overflow básico, como a seguinte:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
/*
 80483f2:        31 c0                   xor    %eax,%eax
 80483f4:        31 db                   xor    %ebx,%ebx
 80483f6:        31 c9                   xor    %ecx,%ecx
 80483f8:        31 d2                   xor    %edx,%edx
```

```
 80483fa:       50                      push   %eax
 80483fb:       68 65 64 21 0a          push   $0xa216465
 8048400:       68 68 61 63 6b          push   $0x6b636168
 8048405:       68 67 6f 74 20          push   $0x20746f67
 804840a:       68 59 6f 75 20          push   $0x20756f59
 804840f:       89 e1                   mov    %esp,%ecx
 8048411:       b2 0f                   mov    $0xf,%dl
 8048413:       b0 04                   mov    $0x4,%al
 8048415:       cd 80                   int    $0x80
 8048417:       31 c0                   xor    %eax,%eax
 8048419:       31 db                   xor    %ebx,%ebx
 804841b:       b0 01                   mov    $0x1,%al
 804841d:       cd 80                   int    $0x80
*/

char code[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x65\x64\x21"
              "\x0a\x68\x68\x61\x63\x6b\x68\x67\x6f\x74\x20\x68\x59"
              "\x6f\x75\x20\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0"
              "\x31\xdb\xb0\x01\xcd\x80";

int overflow() {
    int *ret;

    ret = ((int *) &ret) + 2;
    (*ret) = (int) &code;
    return 0;
}

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

A aplicação acima possui uma função chamada overflow que, na prática, declara um inteiro que é usado para encontrar o endereço da pilha - através da invocação da referência &ret -, para então acrescentar 2 - avançando o ponteiro em 2 posições, ou seja, 8 bytes adiante na pilha - para pular tanto a variável quanto o ponteiro do frame da função que também é alocado na pilha. O último passo do programa é sobrescrever o endereço de retorno com o endereço do shellcode declarado global.

Ao longo deste capítulo foi discutido como executar um ataque via buffer overflow baseado em pilha. Para facilitar o entendimento, um conjunto pequeno e poderoso de instruções assembly x86 foi apresentado. Os conceitos básicos de buffer overflow foram analisados: tanto como e por que ele acontece quanto a forma para projetar códigos especiais – shellcodes - a serem utilizados no mesmo.

Este tipo de ataque se vale da estrutura dos processos em memória e da forma como uma falha de programação provê os meios para que um ataque possa ser executado. Ao longo dos próximos capítulos serão discutidas as proteções atuais contra estas falhas, suas limitações e a solução proposta por este trabalho.

3 ALTERNATIVAS PARA PREVENÇÃO

Dada a definição do problema, o primeiro passo foi definir as alternativas possíveis para a resolução do mesmo. As alternativas foram divididas entre três grupos: alternativas pré-compilação, em tempo de compilação e pós-compilação.

3.1 Alternativa pré-compilação

Nesta linha as ferramentas mais comuns são as de análise estática de código. São consideradas estáticas, pois não levam em conta o funcionamento do programa e isso faz com que sejam menos confiáveis. É possível que a programação com códigos mais obscuros possam vir a camuflar falhas assim como o inverso - falhas não existentes podem ser apontadas.

Existem inúmeras ferramentas de análise estática de código no mercado. Por análise estática entende-se um programa que recebe código fonte como entrada e tem como saída uma análise indicando as falhas encontradas, como: uso de memória anteriormente liberada, vazamento de memória, acesso ilegal de memória - como a posição 26 de um vetor com 25 posições - entre outras. Essas ferramentas comumente apresentam a opção para verificação de uso de funções inseguras, como strcpy ou gets ou até overflows de buffer menos óbvios. Este tipo de análise é bastante comum e há várias ferramentas do tipo no mercado. Uma lista das ferramentas separada por linguagem de programação pode ser vista em List of tools for static code analysis. Um exemplo de ferramenta para análise de código C++ é o Cppcheck (CPPCHECK).

3.2 Alternativa em tempo de compilação

Esta alternativa envolve modificar o código produzido para uma função em tempo de compilação. Nesse ponto o que é alterado é o código binário e não mais o código fonte em texto legível. Portanto, esse compilador geraria um código binário não somente com funcionalidade como o originalmente planejado, mas também com um incremento na segurança.

A falta de habilidade do programador em levar em consideração a segurança seria amenizada por esse compilador. Essa alternativa - com injeção de código em tempo de compilação - é explorada amplamente pelos compiladores mais modernos. A Microsoft mantém habilitado por default um parâmetro de compilação no Visual Studio - ambiente de desenvolvimento da mesma - para que a pilha de execução seja protegida nos programas compilados por ele. O GCC, comumente presente nas distribuições baseadas

em Linux, possui uma série de pacotes de extensão para proteger a pilha, servindo como exemplo o StackGuard (COWAN, 1998), entre outros.

A prática de proteger o software no momento da compilação é interessante e é muito usada hoje em dia, especialmente nos sistemas Microsoft. Porém, há uma gama de softwares compilados anteriormente à padronização do uso da proteção da pilha. Esses softwares ainda estão vulneráveis em sistemas operacionais mais antigos, porém estão mais – mas não totalmente - protegidos nos sistemas operacionais atuais. As alternativas pós-compilação são muito usadas nos sistemas operacionais modernos.

3.3 Alternativas pós-compilação

Esse conjunto é constituído por duas alternativas: proteção via sistema operacional e proteção da imagem do executável.

3.3.1 Proteção via sistema operacional

Hoje em dia é comum nos sistemas operacionais o bloqueio de execução de código a partir da pilha. Esse tipo de proteção pode ser encontrada de duas maneiras distintas: via hardware ou via software.

Na proteção via hardware o padrão é utilizar um bit para indicar a permissão para execução de código a partir da pilha. Os processadores Intel e AMD mais novos possuem um bit - conhecido como XD bit nos processadores Intel (INTEL-a) e NX bit nos processadores AMD - com essa funcionalidade. Esse bit, quando ativado, bloqueia execução de código e normalmente é ativado para executar procedimentos do sistema, ou, para ativar os modos de compatibilidade com aplicações mais antigas que requer este tipo de acesso.

Esta proteção consiste em marcar determinadas páginas de memória como não executáveis. No caso, um bit na tabela de páginas - PTE ou Page Table Entry - é ativado para identificar a página como não executável. Este bit pode ser suportado via hardware ou software.

Na proteção via software a funcionalidade dos bits de proteção - seja NX ou XD - é simulada pelo kernel do sistema operacional. O módulo PaX para o Linux provê uma implementação completa dos bits - mesmo incorrendo em redução do desempenho - enquanto o Windows apenas provê proteção contra ataques ao mecanismo de gerenciamento de exceções - SEH ou Structured Exception Handling. Esta classe de ataques – buffer overflow para sobrescrita do gerenciador de exceções (JOHNDAS) - consiste em sobrescrever o endereço que aponta para o gerenciador de uma determinada exceção de forma a desviar para algum código arbitrário. Esta é uma variação de ataque que não necessita sobrescrever o endereço de retorno de uma função, mas sim do gerenciador de exceções.

Os sistemas operacionais também utilizam uma técnica chamada ASLR - Address Space Layout Randomization (SHACHAM, 2004) – que consiste em dificultar, para o atacante, o encontro do buffer no qual o código arbitrário foi escrito através da introdução da aleatoriedade no início da pilha. Como a pilha é alocada em um endereço aleatório a cada execução das aplicações, pelo sistema operacional, o atacante fica impossibilitado de reproduzir consistentemente o ataque, pois ao sobrescrever o endereço de retorno ele precisa retornar para o seu código e o mesmo estará potencialmente em um endereço diferente.

Essa proteção adicional não impede que os ataques aconteçam, ela apenas dificulta a execução dos mesmos. Adicionalmente, para contornar esse problema, caso o atacante tenha acesso a uma linha de comando na máquina atacada, uma variável de ambiente pode ser definida contendo o shellcode. O endereço das variáveis de ambiente é fixo enquanto o sistema operacional estiver em funcionamento e isso proporciona um local fora da pilha para que o código malicioso seja colocado. Este tipo de ataque é utilizado normalmente para criação de linha de comando como administrador ao atacar serviços que executam com alto privilégio no sistema. Isso não é possível ao atacar sistemas remotos onde o atacante não possui acesso à linha de comando.

Tanto o sistema Windows quanto os sistemas baseados em Linux possuem essas proteções, ou um subconjunto delas, implementadas. Essas defesas que são independentes das aplicações são cada vez mais comuns, pois desoneram os desenvolvedores da necessidade de focar tempo, esforços e recursos nesse quesito.

3.3.2 Proteção da imagem do executável

Este tipo de proteção é feita na imagem do software já compilado. A complexidade de um software já compilado é grande, pois cada sistema operacional faz uso de algum formato específico: sistemas operacionais baseados em Linux utilizam ELF, Windows usa PE - Portable Executable - e o MacOS utiliza PEF - Preferred Executable File.

Essa diversidade impõe sérias limitações no desenvolvimento de uma solução multiplataforma. Cada formato citado possui uma estrutura de arquivo específica para agrupar as seções, tabelas de símbolos, funções, descritores de dados e outras estruturas específicas de cada formato. Uma solução multiplataforma, na prática, requer a leitura e correta montagem da imagem do software - a imagem consiste nos dados contidos no arquivo do software - em memória para posterior alteração - aplicando alguma regra para prevenção de ataques - para então remontar a imagem alterada em disco. Para desenvolver uma ferramenta multiplataforma seria possível utilizar uma tecnologia como Java que é capaz de rodar o mesmo código em diferentes máquinas, produzindo leitores e editores para cada tipo de imagem. Porém, esse processo se mostrou muito complexo e, apesar de considerado, foi descartado dadas as implicações anteriormente citadas.

Para contornar esse problema optou-se pela busca de uma ferramenta de instrumentação de imagens de aplicativos. Uma ferramenta de instrumentação permite ao usuário colocar condições dependentes do fluxo de execução de um software, assim como analisar o mesmo. É um ponto forte a possibilidade de analisar a construção do executável, ou seja, os dados do aplicativo seguindo o padrão do sistema operacional onde foi compilado.

Essa última alternativa compreendeu o escopo do trabalho permitindo que o mesmo pudesse ser executado em tempo. A próxima seção comentará mais essa alternativa assim como apresentará a linha de trabalho seguida.

3.4 Abordagens aventadas

Uma vez entendido o problema e como ele é aplicado ficou claro que o ponto a ser atacado era o overflow. Sem a ocorrência de um overflow nenhum ataque desta linha é possível tornando os programas blindados.

3.4.1 Módulo em tempo de compilação

A primeira alternativa aventada foi a criação de um módulo para compiladores que tornasse os programas mais seguros uma vez compilados. Atualmente existem boas alternativas nessa linha como o StackGuard para o GCC e o GS para o Visual Studio no Windows.

Essa alternativa tem um ponto muito positivo que é o de reorganizar o executável produzido a partir do código (em formato texto em uma linguagem como C) introduzindo canários ou clonando porções da pilha para validação posterior. Essa reorganização normalmente resulta em baixo impacto no desempenho da aplicação - salvo casos extremos como aplicações com funções curtas e com muitas chamadas de função. As aplicações produzidas com esta técnica ficam blindadas para os ataques baseados em overflow na pilha.

O ponto mais fraco é que essa proteção não é retroativa. Aplicações compiladas antes da introdução dessa modificação nos compiladores não estão protegidas e elas mesmas não permitem a correção de aplicações já compiladas, deixando então a responsabilidade para o produtor da aplicação.

Um usuário final que possui versões anteriores continuará vulnerável a menos que possua algum tipo de contrato de suporte que preveja liberação de patches de segurança como os liberados pelos mantenedores de sistemas operacionais.

Essa alternativa não foi selecionada, pois já havia boas abordagens disponíveis e em uso no mercado.

3.4.2 Reescrita do código binário da aplicação

Esta abordagem visava tomar um aplicativo como entrada (ou seja, código binário) e retornar um aplicativo mais seguro. A segurança seria atingida através da modificação dos prólogos e epílogos das funções de forma a validar aplicando alguma técnica convencional, como canário, que um overflow de buffer ocorreu. No caso de um overflow a aplicação poderia ser encerrada apontando um erro de segmentação ou uma potencial falha de segurança.

A idéia original seria uma ótima solução para o problema mesmo para aplicação legadas, contudo após alguns meses de pesquisa ela se mostrou não aplicável dadas as várias restrições, como:

- A inserção de novo código em um aplicativo já compilado implica em reajuste de endereços no próprio executável;
- Cada sistema operacional utiliza um formato de executável diferente: PE para Windows e ELF para Linux, por exemplo;
- Mudanças de código implicam em uma mudança do hash do executável, o que pode invalidar assinaturas;
- Antivírus poderiam apontar contaminação na aplicação através de suas heurísticas que levam em conta a estrutura do arquivo e da sua imagem em memória;

Independente do formato da aplicação - seja PE, ELF ou qualquer outro - ele normalmente é dividido em seções - cada uma devidamente estruturada contendo header próprio - que descrevem os dados alocados (como variáveis globais e strings), que contém o código a ser executado, que contém a tabela de símbolos entre outras. Todas essas seções precisam ser corretamente mapeadas e reescritas respeitando seu

funcionamento sem introduzir regressões (mudanças que quebram uma funcionalidade já existente).

A despeito desses detalhes, mesmo que o código binário a ser injetado na aplicação pertença à mesma arquitetura (x86) a montagem dos arquivos executáveis difere de sistema operacional para sistema operacional. Isso é um forte impeditivo para aproveitar a padronização do assembly e desenvolver uma solução multiplataforma, por exemplo. Hoje existem linguagens de programação (como Java) que quando corretamente programadas conseguem executar programas em diferentes plataformas.

Uma arquitetura inicial foi pensada para que essa solução fosse aplicada. Um framework em Java para modificações de aplicações do processador x86 seria desenvolvido. Esse framework permitiria a remontagem de arquivos executáveis e seria tão multiplataforma quanto possível.

Ao longo da pesquisa ficou claro que essa abordagem já seria um sucesso caso fosse desenvolvida para apenas uma plataforma. Durante a pesquisa dessa abordagem, um artigo (Prasad, 2003) foi publicado com a finalidade de reescrever o código binário das aplicações gerando aplicações mais seguras (mas não totalmente). A abordagem do artigo é fortemente direcionada somente a um sistema operacional (no caso, Windows). Diferentemente das outras ferramentas citadas anteriormente, esta não encontrou aplicação comercial e por isso acabou não sendo amplamente divulgada.

Dadas as limitações relacionadas acima foi pensado que talvez o problema de entender e reorganizar arquivos executáveis em múltiplas plataformas já houvesse sido resolvido por algum grupo de pesquisa. No decorrer da pesquisa ficou claro que o que era necessário era uma ferramenta de instrumentação de código binário.

Nessa linha uma busca por ferramentas de instrumentação de aplicações foi feita resultando na escolha de uma em especial que mudou o rumo do trabalho. No capítulo seguinte as ferramentas serão apresentadas e um estudo sobre a abordagem escolhida.

4 PREVENINDO BUFFER OVERFLOWS

Nos últimos tempos uma série de softwares de instrumentação de código vem sendo desenvolvidos para analise de fluxo de execução, depuração e teste de software. Normalmente o foco delas se encontra em áreas de suporte ao desenvolvimento de sistemas e não em uso prático em campo. Apesar deste fato, é possível utilizar a infra-estrutura que essas ferramentas possuem para subverter o conceito que temos delas hoje em dia e utilizar como defesa pró-ativa em termos de segurança.

Tais ferramentas permitem a um desenvolvedor inspecionar a aplicação em diversos aspectos, dentre eles:

- Seções da imagem da aplicação em memória;
- Funções e símbolos vinculados à aplicação;
- Instruções da seção de texto;
- A memória da aplicação;
- E outros.

A forma como elas fazem isso é criando abstrações em código que representam partes de executáveis de acordo com os padrões de formato de arquivos executáveis vigentes no sistema operacional em questão. Atualmente, boa parte das ferramentas suporta ao menos dois sistemas operacionais (que englobam a maior parte do mercado): os baseados em Linux e o Windows.

Tais abstrações são vantajosas, pois refletem em facilidade de programar módulos para tais ferramentas. Adicionalmente, são módulos que funcionam em ambos os sistemas operacionais com poucas - ou eventualmente nenhuma - alteração no código do mesmo. O benefício disso é que o desenvolvedor não precisa conhecer profundamente os padrões vigentes (no caso do Windows é o PE – Portable Executable - e dos sistemas operacionais baseados em Linux é o ELF – Executable and Linkable File) para desenvolver tais módulos de instrumentação.

A abstração usada pelas ferramentas de instrumentação mais conhecidas é, em geral, um modelo de classes que generaliza as partes que compõem ambos os padrões, como:

- Imagem;
- Seções;
- Rotinas/Funções;
- Instruções;
- E outros.

Estas classes são utilizadas para encontrar pontos de interceptação da aplicação sendo instrumentada. Os pontos de interceptação são utilizados para inserir código contido no módulo acoplado à ferramenta de instrumentação. Um desenvolvedor pode desenvolver um módulo que intercepte uma chamada a determinada função da aplicação sendo instrumentada. Uma vez que esta função seja chamada no código o módulo de instrumentação desenvolvido é notificado e toma uma ação programada, como contar o número de chamadas à função, verificar os parâmetros da mesma ou validar o retorno da mesma.

O exemplo da interceptação de funções é bem simples quando comparado à flexibilidade permitida por estas ferramentas. Todas as abstrações disponíveis permitem análise e inclusão de código adicional na aplicação instrumentada. A memória da aplicação a ser instrumentada fica disponível para o módulo de instrumentação avaliar e este é o ponto chave que tornou estas ferramentas tão interessantes para este trabalho.

A maior parte das ferramentas de instrumentação disponíveis permite a instrumentação do processo em execução (ou seja, já como uma imagem da aplicação do disco na memória). Esse fato teve grandes implicações no desenvolvimento do trabalho, pois o objetivo original era instrumentar as aplicações (o próprio código binário) em disco gerando uma nova aplicação mais segura para que posteriormente fossem executadas. Essa abordagem visava manter uma compatibilidade retroativa apenas inserindo código assembly x86 compatível com processadores Intel 80386 (e outros baseados no mesmo conjunto de instruções).

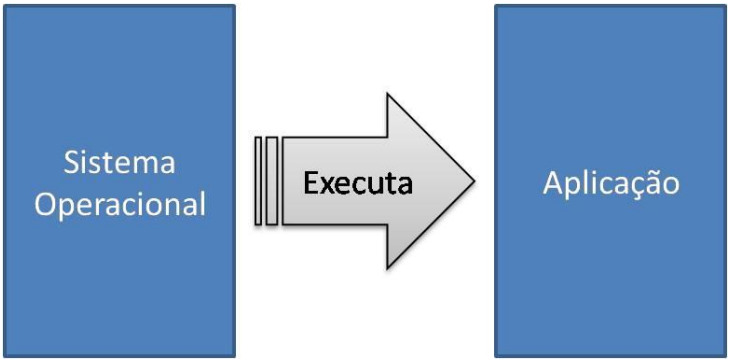

Figura 4.1: Interação do sistema operacional com a aplicação

Na figura 4.1 é mostrada a interação do sistema operacional com a aplicação. Para demonstrar o funcionamento geral das ferramentas de instrumentação, na figura 4.2, será apresentada a interação do sistema operacional no modo instrumentado.

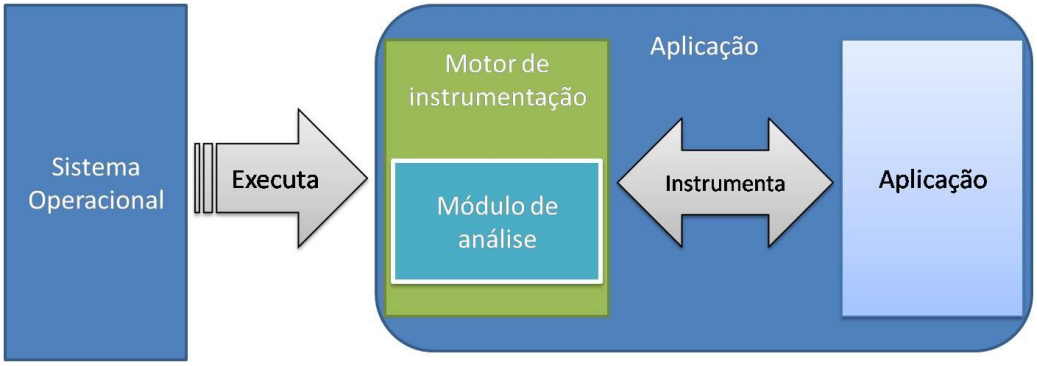

Figura 4.2: Interação do sistema operacional com o framework de instrumentação

Na figura 4.2 a interação do sistema operacional com o motor de instrumentação é apresentada. Através da figura fica claro que o sistema operacional interage com todo o complexo de aplicações como se fosse uma única aplicação. Na prática o sistema operacional vê somente o motor de instrumentação como aplicação. Esse motor é iniciado utilizando um módulo de análise para verificar algum aspecto da aplicação sendo instrumentada. O último componente envolvido é a aplicação real.

A figura 4.2 também deixa clara uma forte limitação da instrumentação: desempenho. As camadas adicionais que a diferenciam da figura 8 evidenciam a quantidade adicional de instruções a serem executadas para que o mesmo programa da figura 8 execute. Tradicionalmente, ferramentas de instrumentação funcionam adicionando funções para prólogos e epílogos de instruções. Como as ferramentas em geral permitem a instrumentação até este nível se faz necessário que isso seja feito. As limitações de desempenho serão discutidas mais nas próximas seções.

As vantagens e a simplicidade da instrumentação em tempo de execução apontaram uma possibilidade só vislumbrada no encontro da limitação (no que tange instrumentação estática – reescrita do código binário). Neste momento os pontos positivos e negativos desse modo de instrumentação foram levantados e serão discutidos ao longo deste capítulo.

4.1 Escolhendo a ferramenta de instrumentação

Muitas dessas ferramentas possuem características especiais dependentes de plataforma. Algumas são extensões para o núcleo do sistema operacional, outras conseguem rodar sem alterar o núcleo do sistema. A ferramenta mais interessante para esse caso teria como características:

- Ser multiplataforma;
- Não alterar o sistema hospedeiro de forma alguma;
- Ter suporte a sistemas operacionais mais antigos;

A possibilidade de alterar o fluxo de execução do software instrumentado é necessária, pois eventualmente um software atacado deverá ser terminado abruptamente gerando algum registro ou ser encerrado silenciosamente. Uma última qualidade interessante é permitir instrumentar um software que já está rodando. Isso é uma característica relevante, pois eliminaria a necessidade de reiniciar algum serviço vital caso desejássemos proteger o mesmo.

Uma série de ferramentas foi encontrada que reunia, no mínimo, um subconjunto das características. Dentre elas as que se encaixaram nos requisitos: PIN (LUK, 2005) e Dyninst (BUCK, 2000).

Ambas possuem funcionalidades semelhantes no sentido de prover ao desenvolvedor uma abstração facilmente programável dos processos. Comparativamente, o modelo de dados da ferramenta PIN é mais simples de programar (guardadas as complexidades de ambas). Uma diferença para o desenvolvimento de ferramentas na linguagem é que PIN tem como base a linguagem C, já Dyninst utiliza C++.

No que tange o funcionamento, ambas permitem instrumentação tanto de aplicações que já estão em execução (conhecido por “attach to process”) quanto de aplicações sendo iniciadas. Ambas as ferramentas também permitem alteração do fluxo de execução da aplicação, o que é fator chave para o protótipo desenvolvido para validação da abordagem proposta neste trabalho.

O fator multiplataforma é mais amplo para a ferramenta PIN que suporta, também, a arquitetura IA-32, IA64 e x86-64. Tanto PIN quanto Dyninst são portáveis para Windows e Linux, porém cada uma é portável, também, para plataformas diferentes, como MacOS (PIN) e Solaris (Dyninst).

A ferramenta PIN se destaca por ser suportada oficialmente pela Intel. E esse foi o fator de desempate na decisão final sobre qual ferramenta utilizar. Uma vez selecionada a ferramenta um estudo sobre a funcionalidade foi feito de forma a identificar como aplicar a abordagem escolhida para resolução do problema.

A possibilidade de desenvolver utilizando uma ferramenta de instrumentação que possui versões para os sistemas operacionais mais conhecidos torna possível escrever ferramentas que rodem nas diversas plataformas e façam as verificações necessárias. A maneira encontrada para garantir portabilidade para os diferentes sistemas é utilizar para o desenvolvimento da ferramenta uma linguagem que possui compiladores nas diferentes plataformas alvo. Isso foi atingido através do uso da linguagem C++ em conjunto com a STL. Isso garante que podemos compilar o mesmo código, quando aplicável, nas diversas plataformas e ainda assim gerar um software usável.

Ainda que seja possível que isso seja feito, vale frisar que para cada plataforma adicional um período de validação se faz necessário. Os testes com o protótipo mostraram que cada plataforma possui suas vicissitudes que tornam uma generalização completa difícil.

Neste estudo não consideraremos as vulnerabilidades passíveis de serem encontradas nas ferramentas de instrumentação de aplicações. As ferramentas podem possuir falhas de programação que podem ser exploradas através da criação de um processo adequadamente preparado para explorar uma vulnerabilidade específica; portanto, trabalhamos com a hipótese de que o processo é genuíno, ou seja, ele por si só não representa um ataque.

Adicionalmente, assumindo que realmente haja um problema na ferramenta de instrumentação, o benefício que ela traz quando utilizada com o módulo de proteção em conjunto com a facilidade de correção de vulnerabilidades em uma única aplicação (ou seja, na ferramenta de instrumentação) compensa. O custo final de corrigir somente a ferramenta de instrumentação acaba sendo menor que não utilizá-la e deixar todas as vulnerabilidades sem proteção.

4.2 Abordagem para prevenir ataques

De acordo com a análise do problema feita no capítulo 2, vimos que existem alguns passos para que um ataque via buffer overflow na pilha se concretize:

1. Entrada maliciosa que provoca o overflow do buffer sobrescrevendo o endereço de retorno
2. Desvio da execução normal do programa
3. Execução de código não planejado originalmente pela aplicação

Para a classe de ataques baseados em buffer overflow na pilha fica claro que ao deter o primeiro passo, evita-se que os outros dois aconteçam.

No capítulo 2 vimos que há duas classes maiores de ataques baseados em buffer overflow na pilha:

- Execução de shellcode injetado (ataque convencional)

- Retorno à libc (execução de função existente não planejada)

Ambos os ataques requerem que o endereço de retorno seja modificado para pular tanto para o shellcode injetado no buffer sobrescrito quanto para uma função existente como system ou exec.

Se uma camada de software monitorar os endereços de retorno realizando validações para garantir que ele é intencional podemos assegurar a segurança da aplicação. O método empregado é chamado pela literatura de RAD (Return Address Repository).

Ele consiste basicamente da criação de um repositório de endereços de retorno para a aplicação de forma a assegurar nele que somente endereços de retorno intencionais sejam salvos.

A cada chamada de função o endereço de retorno (ou seja, o endereço da instrução seguinte à chamada da instrução *call*) é colocado no repositório. A cada retorno (chamada da instrução *ret*) a validade do endereço é verificada no repositório. Caso o endereço de retorno seja legítimo a execução do programa segue, do contrário o programa é encerrado. No ponto de encerramento é possível programar alguma espécie de aviso qualquer, no caso um aviso é registrado em um arquivo de log.

Este aviso poderia ser, também, um e-mail para o administrador do sistema para que ele seja informado em tempo real do ataque conforme sugerido em Prasad (2003).

Em Hsu (2000) há uma prova formal do método. A nomenclatura utilizada é RAR (Return Address Repository). No mesmo trabalho uma extensão para o compilador GCC é implementada utilizando a técnica RAR. Em Prasad (2003) os pesquisadores evoluíram o método para que não somente funcionasse em tempo de compilação como também pós compilação. Neste último caso o arquivo binário é reescrito para implementar a técnica RAR. A abordagem implementada nesse trabalho tem a vantagem adicional de prevenir mais casos de buffer overflow baseados em pilha que a proposta em Prasad (2003), pois no trabalho em questão existem algumas funções cujo prólogo não possui espaço suficiente para que uma chamada de função de validação do repositório seja inserida, permitindo, então, que uma função permaneça insegura.

Adicionalmente, a abordagem utilizando instrumentação da aplicação possibilita proteger não somente a aplicação como todo código ligado a ela. Os códigos das bibliotecas dinâmicas, por exemplo, também são instrumentados utilizando a ferramenta. Esse é um diferencial bastante interessante em relação à abordagem de reescrita do arquivo binário.

4.3 Prototipando a solução

Uma vez delineada a proposta para realizar a prevenção de buffer overflows baseados em pilha e escolhida a ferramenta de instrumentação uma análise das APIs providas pela ferramenta foi feita para mapear o trabalho necessário.

Os primeiros testes da ferramenta envolveram executar simulações já prontas fornecidas pela mesma. O PIN encontrou mais utilidade nas simulações para coleta de dados de desempenho e depuração de aplicações que propriamente em segurança.

A modelagem da solução foi feita paralelamente ao design do protótipo, pois era necessário avaliar a forma como os códigos dos aplicativos convencionais de campo (java, notepad, ...) eram montados. Para essa análise, muito convenientemente, a ferramenta de instrumentação teve papel fundamental.

No ambiente Windows o Visual Studio foi usado em conjunto com o Cygwin que possui tanto ferramentas de debug quanto de desmontagem de código binário para assembly. Há também o pacote Windbg, da Microsoft, que permite depuração em nível de código assembly.

No ambiente Linux há um pacote completo de desenvolvimento que vai desde compiladores a depuradores e desmontadores de código binário para assembly.

As primeiras experiências com o PIN foram executadas usando o contador de instruções executadas por uma aplicação. Foi possível perceber uma compatibilidade bastante grande pois alguns aplicativos padrão como o notepad e o wbemtest (ambos empacotados com o sistema operacional Windows) funcionaram corretamente e tiveram suas instruções contadas.

O objetivo do protótipo é validar que aplicações podem ser executadas aplicando os conceitos de RAR sem alterar a aplicação binária inicial. Adicionalmente, o protótipo deve funcionar pelo menos nas plataformas Windows e Linux para aplicações x86 compiladas a partir de código C ou C++.

Apesar das restrições uma ampla gama de aplicações é coberta. Algumas linguagens, como Delphi, não respeitam integralmente as convenções de chamada de função como descrito na seção 2.1.4. Essa limitação foi considerada aceitável.

4.3.1 O desenvolvimento do protótipo

O protótipo desenvolvido funciona tanto na plataforma Windows quando nos sistemas baseados em Linux. Durante os testes do mesmo houve alguns percalços que foram contornados à medida que apareciam.

Cada plataforma tem suas nuâncias que causam falsos positivos na análise. Esses problemas serão descritos mais adiante neste trabalho.

O protótipo foi desenvolvido utilizando a linguagem C++ com o auxílio da ferramenta de instrumentação PIN (desenvolvida pela Intel com suporte a grande maioria de suas arquiteturas).

Este protótipo fez uso da instrumentação de instruções x86 provida por PIN. É possível inspecionar o estado do processador no momento da execução de qualquer passo do programa, seja na execução de instruções, de funções ou da carga de uma imagem em memória (programas ou bibliotecas dinâmicas resultam em imagens).

Ao instrumentar uma aplicação utilizando o protótipo a memória de ambos compartilha o mesmo espaço, ou seja, tanto a aplicação instrumentada quanto a aplicação de defesa compartilham o mesmo espaço de endereçamento. Esse fato não é um problema, pois o repositório de endereços de retorno fica alocado na HEAP. Como a HEAP está localizada na direção oposta à qual o overflow do buffer ocorre não há perigo dessa memória ser sobrescrita via buffer overflow baseado em pilha.

Um espaço para o repositório de endereços de retorno foi alocado de forma a criar uma zone de endereços válidos. A ferramenta faz uso da característica de instrumentação por instrução para verificar quando uma chamada de função é feita. Ao detectar a mesma o repositório recebe um novo endereço válido. Quando a ferramenta detecta um retorno de função o endereço de retorno é validado contra o repositório.

O fato de poder inspecionar o estado tanto do processador quanto da memória da aplicação torna possível a criação dessa ferramenta. A abstração criada por PIN permite que o mesmo código utilizado no Windows execute nos sistemas baseados em Linux.

Em uma chamada de função a ferramenta registra o endereço de retorno utilizando uma chamada de função similar a:

```
INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) registerRet, IARG_UINT32,
INS_NextAddress(ins), IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
IARG_END);
```

Esta função insere uma chamada a outra função antes da execução da instrução apontada por `ins`. Ela é inserida antes pelo uso da constante `IPOINT_BEFORE` indica que a função deve ser chamada antes da instrução.

Os parâmetros seguintes descrevem os parâmetros de **registerRet**, no caso o endereço da instrução imediatamente seguinte a **ins** e o endereço alvo da chamada de função.

A verificação do endereço de retorno é feita através da seguinte chamada:

```
INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) auditRet, IARG_REG_VALUE,
REG_STACK_PTR, IARG_INST_PTR, IARG_END);
```

O formato do código é relativamente similar à chamada anterior pois isso faz parte da abstração proporcionada pela ferramenta PIN. Os dois primeiros parâmetros tem funcionalidade idêntica a anteriormente citada e desta vez a função **auditRet** é chamada para validar o endereço passado por parâmetro a ela contra o repositório de endereços de retorno. Os parâmetros seguintes são o ponteiro para o topo da pilha e o endereço da instrução *ret* na imagem da aplicação na memória.

PIN consegue prover o endereço do topo da pilha (na prática, o registrador ESP comentado na seção 2.1.1) a função parâmetro da instrumentação. De posse de ESP e sabendo que no momento da execução da instrução *ret* o topo da pilha deve apontar para o endereço a ser retornado é fácil obter na memória o endereço de retorno que a aplicação irá retornar.

De posse desse endereço **auditRet** valida e interrompe a execução da aplicação **antes** que algum código possa ser executado. Essa é a forma de proteção mais poderosa contra falhas de buffer overflow baseados em pilha, pois impede o passo essencial do ataque (o desvio para o endereço de retorno modificado).

Alguns problemas advindos da forma como a solução foi desenvolvida foram previstos:

- Conflito com camadas de proteção já existentes
- Código assembly em não conformidade com as convenções ou feito à mão
- Instabilidade da ferramenta de instrumentação (não funcionar corretamente com todas as aplicações)
- Potenciais situações inesperadas decorrente de toda solução multiplataforma

A abstração da solução para o problema é relativamente simples, porém a aplicação prática dessa regra provou não ser simples, pois três dos quatro complicadores citados anteriormente se tornaram reais.

4.3.2 Código assembly feito à mão ou sem conformidade com convenções

Hoje em dia são raros os casos que exigem programação assembly diretamente. A maior parte do código, em geral, é gerado a partir de um código produzido em uma

linguagem de programação mais abstrata, como C, C++, Pascal, Delphi entre várias outras. Essas linguagens abstratas são compiladas por um compilador em código assembly. Após esse passo o código assembly é montado por um assembler (montador) em código binário no formato de arquivo de objeto do sistema operacional em questão.

No caso das linguagens de programação abstratas os compiladores são programados para seguir certas convenções, ou seja, adotar padrões de código assembly gerado para manter conformidade com alguma regra de ligação entre aplicações (por exemplo, ao chamar uma função de uma biblioteca dinâmica os argumentos devem ser colocados na pilha na ordem certa).

Adicionalmente, a geração do código assembly é mecânica. Uma vez estabelecido o método para chamadas de função todas as chamadas serão feitas da mesma forma a menos que haja uma razão para que o compilador modifique essa forma.

Contudo, caso o programador opte por programar código assembly diretamente ele tem a opção de modificar a forma como determinadas coisas são feitas podendo, por exemplo, traduzir diretamente uma instrução *call* para um *push* do endereço da próxima instrução e um *jump* para a função desejada. Caso um *ret* seja utilizado ao fim da função chamada ela retornará corretamente, pois o endereço estará na pilha.

Como a construção citada o mesmo pode ser feito para modificar a instrução *ret* realizando um *pop* do endereço que está no topo da pilha e após um *jump* para o mesmo. Caso um programador desça até o nível assembly de programação a aplicação não terá como identificar esses casos sem demandar uma boa quantidade de processamento. Mesmo que eventualmente buscássemos por um *pop* seguido de um *jump* no código sempre será possível inserir *nop* (instrução que não faz nada, apenas passa à próxima instrução, muito usada para aumentar a área de acesso em um shell code injetado) entre ambas instruções e isso enganaria a ferramenta. Há muitos casos em que seria possível simular um *ret* colocando código útil entre o *pop* e o *jump*. Por essa razão a ferramenta pode não ser útil em código assembly feito à mão.

As convenções de chamada de função são seguidas nas aplicações compiladas por compiladores C e C++. Uma série de testes foi feita (serão apresentados mais tarde) validando esse comportamento, contudo, as convenções não são seguidas em código compilado por compiladores Delphi, tornando, então, o protótipo não útil neste caso (muito embora a linguagem Delphi seja mais resiliente a ataques via buffer overflow baseado em pilha).

A limitação a código gerado com compiladores C e C++ não é considerada problemática, pois isso não reduz em muito a cobertura desejada para o protótipo.

4.3.3 Proteções conflitantes em diferentes sistemas operacionais

Recentemente os sistemas operacionais têm incorporado ao seu núcleo proteções contra ataques que são, normalmente, não intrusivas. Isso quer dizer que o sistema operacional modifica o ambiente do software e não sua estrutura do mesmo. A mais comum proteção até então é o bloqueio de execução de instruções provenientes do espaço de endereçamento da pilha. Como exemplo disso temos DEP no sistema operacional Windows e PAX nos sistemas baseados em Linux.

O maior problema encontrado em virtude dessa proteção é a forma como ela é ativada e desativada pelos núcleos dos sistemas operacionais para determinadas funções dificultando a detecção de chamada de função. Essa ativação e desativação é necessária,

pois algumas funções ainda necessitam deste tipo de acesso ativado. As razões pela qual a ativação da execução de instruções na pilha é necessária é totalmente dependente de implementação.

Um caso conhecido em que a execução de código na pilha é necessária são os trampolins. Um trampolim é uma construção de compiladores para permitir que ponteiros de uma função aninhada a outra (a definição de uma função é feita dentro de outra permitindo acesso às variáveis locais da função dentro da função aninhada) sejam passados adiante.

Apesar de parecer um caso simples, o fato de uma função aninhada ter acesso às variáveis locais da função em que ela está contida torna a tradução para o código de máquina complicada, pois, relembrando as convenções de chamada de função na seção 2.4.1, um registrador é usado para conter o ponteiro do quadro em que a função está executando e a função aninhada usa este registrador para acessar as variáveis locais da função em que ela foi definida. Maiores detalhes podem ser encontrados em Wienand (2006) e GNU.

Existem algumas propostas para resolver o problema da passagem de ponteiros para funções aninhadas, mas o mais convencional é o uso de trampolins. Os trampolins são pedaços de código montados em tempo de execução na pilha que fazem os ajustes necessários para a correta execução da função aninhada quando passada por ponteiros. O fato desse código ser montado na pilha e executado conflita com as proteções via marcação do espaço de memória da pilha como não executável.

Durante o desenvolvimento deste trabalho os trampolins foram uma preocupação constante, porém, não houve problemas durante a validação relacionados com este assunto.

4.3.4 Instabilidade da ferramenta de instrumentação e situações inesperadas

Apesar de certa instabilidade na ferramenta de instrumentação ter sido esperada a ferramenta escolhida surpreendeu por permitir tanto a execução de módulos já implementados publicados com a mesma quanto o módulo desenvolvido para o protótipo deste trabalho. Nenhum problema de compatibilidade foi encontrado exceto pequenas complicações para compilação no Ubuntu (SO baseado no Linux).

No que tange situações não esperadas nada ocorreu. Alguns falsos positivos realmente causaram problemas mas uma vez investigadas acabavam se revelando pertencentes ou a conflitos com as proteções já existentes ou à código assembly não conforme com as restrições do protótipo (como no caso das aplicações compiladas a partir de código em Delphi).

4.4 Validações da solução

Uma vez desenvolvido o protótipo, uma série de testes foram aplicados para validar a solução. Um dos benefícios de utilizar a ferramenta PIN, como mencionado anteriormente, foi o fato de ela ser multiplataforma. Existiam versões tanto para Linux, quanto para Windows e Mac. Isso possibilitou que fossem executadas validações tanto no Linux (no caso, Ubuntu 8.04 e 7.2) quanto no Windows (Windows XP).

4.4.1 Validação em ambiente livre de ataques

Os primeiros testes executados foram para validar que o protótipo não interferia na execução das aplicações num ambiente livre de ataques. O propósito disso era tanto

garantir que a ferramenta não iria causar efeitos colaterais quanto manteria a funcionalidade original da aplicação monitorada. Os testes de regressão foram importantes para garantir que aplicações não atacadas não viessem a deixar de funcionar por um falso positivo.

Estes testes foram feitos no Windows XP executando primeiramente aplicações simples, como notepad, wordpad e listagem de diretórios, passando em seguida para aplicações mais robustas como java.

Neste ponto foi encontrada a primeira limitação do protótipo: o funcionamento dele depende da forma como o programa é montado. As montagens do código binário mudam dependendo do compilador e linguagem. No caso da falha encontrada no teste, um programa compilado a partir de código Delphi resultou em falha.

O falso positivo reportado no teste com programas Delphi é ocasionado pela forma como as chamadas e retornos de função são feitas. Os programas Delphi não respeitam as convenções de chamadas de função para montagem de código que são padrões para os programas compilados em C. Isso resultou numa restrição no escopo do trabalho: somente programas compilados a partir de código fonte C serão validados.

Uma vez tendo verificado a limitação do protótipo foi desenvolvida uma aplicação teste que, intencionalmente, sobrescreve o endereço de retorno com o endereço de um shellcode foi desenvolvida (esta aplicação foi desenvolvida no capítulo de shellcode na seção 2.3). Uma variação desta aplicação teste foi desenvolvida para Windows de forma a provocar a sobrescrita do endereço de retorno. O mesmo código exatamente não serve pois a montagem do código pelo compilador utilizado no Windows colocava o endereço de retorno mais distante do que o compilador GCC no Linux o faz. O protótipo detectou com sucesso as situações em que o overflow ocorreria.

```
char code[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x65\x64\x21"
              "\x0a\x68\x68\x61\x63\x6b\x68\x67\x6f\x74\x20\x68\x59"
              "\x6f\x75\x20\x89\xe1\xb2\x0f\xb0\x04\xcd\x80\x31\xc0"
              "\x31\xdb\xb0\x01\xcd\x80";

int overflow() {
    int *ret;

    ret = ((int *) &ret) + 2;
    (*ret) = (int) &code;
    return 0;
}

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

Uma vez validado o protótipo com algumas sobrescritas de endereço de retorno padrões – o fluxo de trabalho envolveu algumas iterações para resolver os problemas citados ao longo dessa seção – mais testes avançados em casos reais de buffer overflow baseado em pilha foram feitos. Os resultados foram bastante interessantes.

Uma vez tendo o protótipo passado nas validações básicas (tanto de regressão quanto funcional) houve uma busca por falhas atuais de buffer overflow que afetassem diretamente o sistema onde as aplicações defeituosas eram executadas. No quesito validação a característica multiplataforma da ferramenta de instrumentação PIN foi

bastante valiosa. Ela permitiu que o mesmo protótipo (escrito em ANSI C++) pudesse ser compilado para ambos ambientes assim permitindo que uma maior gama de aplicações pudesse ser testada.

A busca de aplicações para teste do protótipo foi feita no Google em busca por sites contendo índices de falhas recentes separadas por classe (buffer overflow, heap overflow, SQL injection entre outras). Destes, as mais recentes e verificadas à época foram escolhidas e testadas. Dentre as atuais viáveis para teste, duas foram selecionadas: freeSSHd e 3proxy.

A aplicação 3proxy, portada tanto para Linux quanto para Windows, possuía uma falha de buffer overflow em um de seus logs internos. Uma mensagem perfeitamente montada para a aplicação levando em conta as chamadas específicas do sistema operacional que hospeda a aplicação pode abrir uma backdoor no servidor provendo acesso a um console remoto. Caso o proxy seja executado com permissões de administrador (root) este console remoto criado à partir da aplicação terá as mesmas permissões.

A aplicação freeSSHd possuía uma falha com resultados similares aos da falha do 3proxy. Ambas as aplicações são distribuídas livremente na internet na forma de software livre.

Os testes foram executados num notebook HP modelo 6910p com processador Intel core 2 duo rodando um Windows XP Pro. O processador é um fator importante nesse teste para mostrar que não há proteção em hardware para esse tipo de falha. Esse modelo é bastante recente e perdurará no mercado por alguns poucos anos ainda.

Para permitir a execução de múltiplos sistemas operacionais com agilidade foi utilizado o VMWare Server que possibilita a criação de máquinas virtuais contendo sistemas operacionais diversos mesmo sob um servidor Windows XP.

Um total de 3 máquinas virtuais foram criadas: uma Windows XP Pro, uma Ubuntu 8.04 e outra Ubuntu 7.1. As máquinas virtuais contendo Windows XP Pro e Ubuntu 8.04 foram os alvos do ataque e a máquina virtual contendo Ubuntu 7.1 foi a máquina atacante.

4.4.2 Prevenindo ataque no Windows XP Pro executando freeSSHd

Para simular o ataque foi necessário identificar a versão da ferramenta que possuía a vulnerabilidade. Adicionalmente, uma forma de explorar a vulnerabilidade era necessária. O escopo deste trabalho está em não permitir a exploração da vulnerabilidade buffer overflow e não em, efetivamente, encontrar falhas de buffer overflow.

Para tanto, a versão com defeito da aplicação foi encontrada e instalada em uma máquina virtual com o sistema operacional Windows XP Pro. Adicionalmente, scripts e aplicações de ataque foram descobertos e testados contra a aplicação em questão.

O freeSSHd é um serviço similar ao sshd no Linux, que permite conexões remotas através do protocolo SSH, permitindo conexões seguras através de verificação de credenciais como usuário e senha ou chaves assimétricas.

As versões anteriores a Windows 2003 contendo o último SP não possuíam um serviço SSH nativo e, portanto era necessária a instalação de um serviço de terceiros. Esse era um procedimento comum nas versões do Windows até a 2008 que possui

nativamente um set de ferramentas muito similar ao das distribuições mais conhecidas de Linux.

O ataque explora uma vulnerabilidade de buffer overflow nos comandos ligados a operação de diretórios via SFTP. Uma falha de programação resultando na não checagem de limites ao sobrescrever em um buffer o nome do diretório sendo aberto permite a sobrescrita do endereço de retorno.

Neste ataque a desativação do ASLR não foi necessária, pois a técnica empregada neste ataque é mais avançada. Neste caso, em vez de procurar por força bruta o endereço de retorno uma instrução já presente na aplicação que pula para um endereço de retorno interessante ao atacante é usada. Para exemplificar pode-se dizer que caso a aplicação ou as bibliotecas dinâmicas ligadas a ela contenha uma instrução do tipo "jump %esp" e o endereço de retorno da função seja sobrescrito com o endereço dessa instrução ao retornar a instrução será executada e irá executar o código presente no topo da pilha.

Estes tipos de falha de programação são altamente perigosas e tratadas com maior importância, pois conseguem inutilizar a ASLR efetivamente permitindo que o ataque seja repetido sem erros caso nenhuma outra proteção desative a execução de instruções na pilha ou verifique o endereço de retorno como o protótipo desenvolvido o faz.

Uma vez verificado e executado o ataque, ou seja, tenha sido possível acessar o servidor rodando a aplicação sem autorização prévia, a aplicação foi submetida ao mesmo ataque, porém, desta vez, sendo executada sob a plataforma do protótipo. Uma degradação no desempenho da aplicação foi notada e esta será mencionada no capítulo dedicado a teste tema.

A aplicação atacada não executou o código arbitrário que rodara no teste de execução sem proteção do ataque. Nenhuma alteração no código binário ou fonte da aplicação alvo foi feito para que isso fosse possível e a mesma aplicação foi compilada pelos próprios desenvolvedores. Isso é um ponto crucial para destacar a importância deste trabalho, o código binário original se mantém o mesmo.

O protótipo detectou o ataque e encerrou abruptamente a execução do programa apresentando uma mensagem que informa que um ataque via buffer overflow foi feito, porém a execução do código arbitrário foi impedida.

4.4.3 Prevenindo ataque no Ubuntu 8.04 executando 3proxy

A aplicação 3proxy é um proxy multiprotocolos (HTTP, HTTPS, FTP, SOCKS, POP3, SMTP entre outros) bastante pequeno e portável. Ela é freeware, permitindo que empresas pequenas e usuários domésticos façam uso do mesmo. Os desenvolvedores validam a aplicação em diversas plataformas, como: Windows 98/NT/2000/2003/XP/x64, FreeBSD/i386, NetBSD/i386, OpenBSD/i386, Linux/i386, Linux/PPC, Linux/Alpha, Mac OS X/PPC e Solaris 10/i386.

O interessante de um ataque a uma aplicação destas é o fato de que essa aplicação potencialmente estará rodando em máquinas de usuários domésticos. Isso é especialmente atrativo, pois é possível montar uma rede de zumbis (máquinas controladas por uma terceira pessoa utilizada para processamento distribuído de ataques ou qualquer funções que o valha).

Este teste foi executado utilizando um Ubuntu 7.10 para atacar um Ubuntu 8.04. Por simplicidade, para a execução deste ataque a proteção padrão do sistema operacional

ASLR foi desligada para que o endereço de retorno desejado para a execução correta do buffer overflow fosse acertado múltiplas vezes mais facilmente.

A falha de buffer overflow nesta aplicação fazia uso de um erro de programação na função de log do proxy que não checava os limites na escrita do buffer que continha a mensagem do log. Como o log registrava as URLs passadas para o proxy via um GET do protocolo HTTP, por exemplo, caso o esmo tivesse o tamanho correto iria sobrescrever o endereço de retorno. Conforme explicado acima, para evitar força bruta para encontrar o endereço correto para o retorno a proteção ASLR foi removida.

Uma vez executado o ataque com sucesso múltiplas vezes (resultando na abertura de um Shell remoto com a permissão com a qual o proxy foi iniciado) a ferramenta de proteção foi utilizada para verificar se a mesma falha repetidamente explorada ocorreria da mesma forma.

O proxy foi iniciado sob a proteção do protótipo desenvolvido e o mesmo ataque anteriormente bem sucedido foi feito sem sucesso. O proxy foi encerrado abruptamente deixando uma mensagem de log informando sobre a tentativa de intrusão.

4.5 Considerações quanto ao desempenho

Com o objetivo de entender o impacto da execução da aplicação sob a proteção do protótipo algumas medidas básicas de desempenho foram feitas. A ferramenta de instrumentação de aplicações permitia o uso recursivo da mesma, isso permitiu que aplicações simples (tanto sem buffer overflows quanto com) fossem mensuradas no que tange o número de instruções executado sem qualquer instrumentação por parte da ferramenta, o número de instruções com instrumentação sem o uso da ferramenta para detectar o buffer overflow (para medir a carga que o motor de instrumentação aplicava sobre as aplicações) e também o número de instruções com a utilização do protótipo.

Posto isso, as medidas de desempenho mostraram que em média, a execução de uma instrução da aplicação instrumentada resulta na execução de mais de 10 instruções reais pelo processador.

A abordagem escolhida para permitir o desenvolvimento do protótipo da solução fez uso de uma ferramenta de instrumentação de aplicações genérica. Por se tratar de uma ferramenta de instrumentação de propósitos gerais, ela possui uma série de passos e tratamentos em seu funcionamento que não necessariamente se aplicam à solução desenvolvida. Relembrando o estudo sobre ferramentas de instrumentação apresentados no início deste capítulo, a cada instrução da aplicação instrumentada que é executada uma série de instruções da ferramenta de instrumentação é executada.

Um objetivo informal de atingir 1% de overhead máximo foi inicialmente estimado, porém, este mesmo mostrou ser não atingível uma vez que em Prasad (2003), utilizando um método que teoricamente reduziria o overhead ao mínimo possível (a reescrita do binário não utiliza instrumentação sendo, portanto, executada diretamente pelo sistema operacional), o overhead máximo calculado chega a 45%. A cada instrução executada pela aplicação original mais 0,4 instruções adicionais são executadas pela aplicação reescrita.

A partir destes resultados um projeto inicial de modificações no protótipo foi feito para tentar aproximar o overhead do protótipo produzido por este trabalho ao overhead do protótipo em Prasad (2003). No decorrer da modelagem destas modificações ficou

claro que um grande volume de trabalho seria necessário, em especial na substituição da ferramenta de instrumentação por uma versão mais leve.

Idealmente, uma solução mista entre este trabalho e Prasad (2003) seria o ideal. As duas soluções são complementares, pois uma troca o desempenho pela maior cobertura (pode proteger até bibliotecas dinâmicas ligadas à aplicação principal) e diversidade (permitir fácil portabilidade entre plataformas) enquanto a outra mantém o desempenho abrindo mão de proteger a aplicação em alguns casos específicos e se restringe unicamente aos binários reescritos. Adicionalmente, o código binário é alterado em Prasad (2003), podendo produzir falhas no cálculo do hash da aplicação e, também, resultar em falsos positivos nas análises dos antivírus.

A solução ideal inseriria os pontos de checagem diretamente na imagem dos objetos (seja aplicações ou bibliotecas ligadas) diretamente em memória antes da execução da mesma. Em um único passo a aplicação teria seu fluxo de execução alterado através da modificação do código nas imagens em memória. Isso permitiria que a aplicação protegida fosse executada diretamente pelo sistema operacional sendo a instrumentação feita num passo pré-execução da aplicação. Esse método se chama instrumentação estática (a forma atual é a dinâmica). Até o momento deste trabalho não há um framework estabelecido e consolidado na literatura que permita a instrumentação estática em múltiplas plataformas, apesar de haver um movimento nessa direção pelo time de desenvolvedores da ferramenta DynInst (mencionada no princípio deste capítulo).

5 CONCLUSÕES

Os ataques à aplicações via buffer overflow baseado em pilha são uma realidade até hoje no mercado de aplicações. As falhas de programação que viabilizam os ataques ainda são inseridas nas aplicações através do uso de técnicas de programação simplistas que focam apenas no desenvolvimento de funcionalidade em detrimento dos aspectos relacionados à segurança. Tanto as aplicações comerciais quanto as aplicações de código aberto possuem estas falhas de programação, e mesmo com todos os boletins de segurança expedidos em nome dessas falhas ainda existe uma cultura de que é menos custoso corrigir uma falha de segurança após sua descoberta do que prevenir a mesma durante os ciclos de desenvolvimento.

O foco deste trabalho foi desenvolver uma ferramenta que desse suporte à prática da programação não orientada a segurança, ao menos no que tange as falhas de buffer overflow baseado em pilha. Assumindo uma postura pessimista ficou claro que era necessário construir uma camada de proteção independente que pudesse eventualmente até ser estendida para abranger mais classes de falhas conhecidas.

Para resolver o problema uma camada de proteção entre o sistema operacional e a aplicação foi inserida que monitora o fluxo de execução da aplicação armazenando os endereços de retorno válidos em um repositório para posterior validação. A cada retorno de função (instrução que, em conjunto com a sobrescrita do endereço de retorno permite o ataque) o endereço a ser retornado é validado contra o repositório.

Os resultados obtidos ao longo do trabalho foram promissores no que tange a segurança. Sabendo ser impossível validar a solução para todas as aplicações a estratégia foi restringir o escopo a uma classe clara de problemas (programas que seguem as convenções, compilados a partir de código C e que não possuem código assembly feito à mão). Como esperado, não houveram casos nas aplicações testadas em que ocorresse um falso positivo, porém, os testes no Ubuntu (baseado em Linux) resultaram em falsos positivos quando todo o fluxo de execução era monitorado. A proteção de execução de código na pilha utilizada por essa distribuição (PAX) utiliza código assembly com construções que invalidam a proposta.

Isso gerou a criação de uma seleção para que fosse possível escolher se toda aplicação (incluindo bibliotecas dinâmicas) seriam monitoradas ou não. A despeito deste problema os testes foram executados com sucesso em ambas as plataformas (Windows e Linux) e tanto nos testes de regressão do conjunto de aplicações (verificar se aplicações sabidamente seguras funcionariam normalmente) quanto nos testes de funcionalidade (prevenir ataques ativamente) os resultados foram positivos.

Durante os testes do protótipo foi notada uma degradação no desempenho das aplicações devido ao uso da instrumentação dinâmica. O fato das validações não serem estaticamente inseridas nos pontos de interesse (entrada e saída das funções) e depois executados com o fluxo normal do programa causa uma degradação significativa no desempenho que, dependendo da finalidade do programa sendo monitorado, pode se tornar fator impeditivo.

Testes aprofundados de desempenho não foram executados, mas uma medição de número de instruções executadas em algumas aplicações teste apontou um overhead de 9 instruções por cada instrução da aplicação monitorada.

Para endereçar este problema uma investigação foi feita levando em consideração as medições de aplicações sem instrumentação, com instrumentação de código, mas sem proteção e com proteção. Tais testes mostraram que das 9 instruções adicionais aproximadamente 7 eram executadas por parte da ferramenta de instrumentação.

A solução para o problema de desempenho seria utilizar uma ferramenta que permitisse instrumentação estática de código, porém, a única iniciativa disponível nesse sentido é bem recente e potencialmente instável. O método ideal seria desenvolver um módulo de proteção inteiro da instrumentação à criação do repositório de endereços de retorno e este é o trabalho sugerido para dar seqüência a esta pesquisa.

Adicionalmente, outras variações deste trabalho são propostas como trabalho futuro: monitoramento do frame pointer das funções – uma vez que ele antecede o endereço de retorno teria um funcionamento similar aos canários – e alterações na estrutura de dados utilizada como repositório de endereços de retorno para ampliar a eficiência do processo de prevenção. Estas alternativas são bastante promissoras e devem ser investigadas em momento oportuno.

Um aspecto bem importante do conhecimento obtido com este trabalho é que existe uma imensa variedade de códigos assembly nas aplicações oferecidas pelo mercado. Em especial, os sistemas operacionais e os antivírus precisam executar operações que freqüentemente não seguem os padrões de chamada de função na montagem do código binário. Qualquer trabalho que se aventure nesta área requer uma cuidadosa identificação destas construções irregulares para que sejam tratadas de acordo.

REFERÊNCIAS

ADDRESS space layout randomization. **Wikipedia**. Julho 2009. Disponível em: <http://en.wikipedia.org/wiki/Address_space_layout_randomization>. Acesso em: Julho 2009.

ARANHA, D. F.. **Tomando o controle de programas vulneráveis a *buffer overflow***. Fevereiro, 2003. Brasília, DF. Disponível em: <http://www.cic.unb.br/docentes/pedro/trabs/buffer_overflow.htm>. Acesso em: Julho, 2009.

BREUEL, T. M.. **Lexical closures for C++**. Cambridge, MA, [s.n], 1988

BUCK, B.; HOLLINGSWORTH, J. K.. An API for runtime code patching. In: **The International Journal of High Performance Computing Applications**. 2000, p. 317-329.

BUFFER overflow. **Wikipedia**, Julho 2009. Disponível em: <http://en.wikipedia.org/wiki/Buffer_overrun>. Acesso em: Julho 2009.

OWASP. **Buffer Overflows**. Disponível em <http://www.owasp.org/index.php/Buffer_Overflows>. Acesso em: Julho 2009.

CARTER, P.. PC Assembly Language**.** Disponível em: <http://www.drpaulcarter.com/pcasm/>. Acesso em: Julho, 2009.

COWAN, C. et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: **7th USENIX Security Symposium**, Santo Antonio, Texas. January, 1998.

CPPCHECK. **WIKIPEDIA.** Julho 2009. Disponível em <http://en.wikipedia.org/wiki/Cppcheck>. Acesso em: Julho 2009.

FOG, A.. **Calling conventions for different C++ compilers and operating systems**. 2009

GNU. **Trampolines for Nested Functions**. Disponível em: <http://gcc.gnu.org/onlinedocs/gccint/Trampolines.html>. Acesso em: Julho 2009.

HOWARD, M.; LEBLANC, D.. **Writing secure code:** practical strategies and proven techniques for building secure applications in a networked world. 2. ed. Redmond: Microsoft, 2003.

HSU, F.; CHIUEH, T.. **RAD: A Compile-Time Solution to Buffer Overflow Attacks**. Stony Brook, NY, [s.n], 2000.

IBM. **GCC extension for protecting applications from stack-smashing attacks**. Agosto 2005. Disponível em: <http://www.research.ibm.com/trl/projects/security/ssp/>. Acesso em: Julho, 2009.

INTEL. **Execute Disable Bit and Enterprise Security**. Disponível em: <http://www.intel.com/technology/xdbit/>. Acesso em: Julho, 2009.

INTEL. Intel® **64 and IA-32 Architectures Software Developer's Manuals**. Disponível em: <http://www.intel.com/products/processor/manuals/index.htm>. Acesso em: Julho 2009.

IZIK. **Smack the Stack**: Advanced Buffer Overflow Methods. Março 2006. Disponível em: < http://www.milw0rm.com/papers/7>. Acesso em: Julho, 2009.

JOHNDAS, R.. Steps Involved in Exploiting a Buffer Overflow Vulnerability using a SHE Handler. In: **Information Security Writers**. [S.l:s.n], 2009.

KAEMPF, M.. **Smashing The Heap For Fun And Profit**. Disponível em: <http://doc.bughunter.net/buffer-overflow/heap-corruption.html>. Acesso em: Julho, 2009.

KEHOE, B. P.. **Zen and the Art of the Internet**: The Robert Morris Internet Worm. Janeiro 1992. Disponível em: <http://groups.csail.mit.edu/mac/classes/6.805/articles/morris-worm.html>. Acesso em Julho 2009.

LIST of tools for static code analysis. **Wikipedia**. Julho 2009. Disponível em <http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis>. Acesso em: Julho 2009.

LUK, C. et al. **Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation**. Chicago, IL, 2005. pp. 190-200.

MICROSOFT. **A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003**. Setembro 2006. Disponível em: <http://support.microsoft.com/kb/875352/en-us/>. Acesso em: Julho 2009.

MIKHALENKO, P.. **How Shellcodes Work**. Maio 2006. Disponível em: <http://linuxdevcenter.com/pub/a/linux/2006/05/18/how-shellcodes-work.html>. Acesso em: Julho 2009.

MS Internet Explorer XML Parsing Buffer Overflow Exploit (vista) 0day. December, 2008. Disponível em: <http://exp.syue.com/exploits/7410>. Acesso em: Julho 2009.

ONE, A.. **Smashing The Stack For Fun And Profit**. Disponível em: <http://doc.bughunter.net/buffer-overflow/smash-stack.html>. Acesso em: Julho, 2009.

PRASAD, M., CHIUEH, T.. **A Binary Rewriting Defense against Stack based Buffer Overflow Attacks**. Stony Brook, NY, [s.n], 2003.

PAX. **Wikipedia**. Julho 2009. Disponível em <http://en.wikipedia.org/wiki/PaX>. Acesso em: Julho 2009.

PAX Team. **PaX**. Disponível em: <http://pax.grsecurity.net/docs/pax.txt>. Acesso em: Julho 2009.

ROSIELO, A.. The basics of Shellcoding. In: **Information Security Writers**. Italy, 2004. Disponível em: <http://www.infosecwriters.com/texts.php?op=display&id=143>. Acesso em: Julho, 2009.

SHACHAM, H. et al. On the effectiveness of address-space randomization. In: **ACM CCS'04**, Washington, DC. 2004.

SILBERMAN, P.; JOHNSON, R.. **A Comparison of Buffer Overflow Prevention Implementations and Weaknesses**. Reston, VA, [s.n], 2004

STACK buffer overflow. **Wikipedia**, Julho 2009. Disponível em: <http://en.wikipedia.org/wiki/Stack_buffer_overflow>. Acesso em: Julho 2009.

STACK (data structure). **Wikipedia**. Julho 2009. Disponível em: <http://en.wikipedia.org/wiki/Stack_(data_structure)>. Acesso em: Julho 2009.

WIENAND, I.. **Technovelty**. GCC Trampolines. Maio 2006. Disponível em: <http://www.technovelty.org/code/c/trampoline.html>. Acesso em: Julho 2009.

X86 calling conventions. **Wikipedia**. Julho 2009. Disponível em: <http://en.wikipedia.org/wiki/X86_calling_conventions>. Acesso em: Julho 2009.

X86 Registers. Disponível em: <http://www.pouet.free.fr/spip/IMG/html/x86reg.html>. Acesso em: Julho 2009.

ZILLION. **Writing shellcode**. Abril 2002. Disponível em: <http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html>. Acesso em: Julho 2009.