UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DANIEL RIEGER BECKERT

Estudo comparativo sobre ataques de *buffer overflow* em sistemas de 32 e 64 bits.

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Raul Fernando Weber

Porto Alegre
2015

AGRADECIMENTOS

A finalização desse trabalho não foi um esforço de uma única pessoa, por esse motivo não poderia deixar de lado o agradecimento a todos aqueles que me apoiaram durante a etapa de desenvolvimento desta obra. Em primeiro lugar e acima de tudo devo agradecer a Deus por conduzir minha vida da melhor maneira possível, sempre me abençoando e me dando forças para continuar, mesmo nos momentos mais difíceis.

Em segundo lugar, um agradecimento especial às pessoas que me conduziram e me mostraram os caminhos corretos: minha família. Em especial, a meus pais JONES BECKERT e MÁRCIA RIEGER BECKERT e meu irmão LUCAS RIEGER BECKERT, sem o apoio dos quais esse trabalho não seria possível.

Minha querida amiga e amada namorada KETHELIN KLAGENBERG, por ter me incentivado e apoiado em todos os momentos desde que nos conhecemos. Agradeço por ter aturado os momentos de estresse durante esse período.

Agradeço também aos amigos que sempre estiveram presentes durante esta etapa, um especial ELMER LINK, que em diversos momentos precisou chamar a atenção quanto à importância da finalização desta etapa. Além de amigo, sinto-me muito abençoado por poder chamá-lo de Pastor.

Por fim, gostaria de agradecer à Universidade Federal do Rio Grande do Sul, aqui representada na figura do professor RAUL WEBER, que me orientou nos passos para a conclusão deste trabalho.

RESUMO

Vulnerabilidades do tipo *buffer overflow* têm sido, ao longo da história moderna da computação, uma das formas mais comuns de falhas na área de segurança. Ataques desse tipo tornaram-se possíveis, principalmente, pelo desenvolvimento de uma cultura de programação voltada exclusivamente ao desempenho dos sistemas, onde, quaiquer ações que gerem um maior tempo de desenvolvimentou ou execução são colocadas em segundo plano. Questões relacionadas à área de segurança foram, por muito tempo, deixadas de lado e o resultado disso é um alto número de sistema vulneráveis que necessitam de correções. A mudança de 32 bits para 64 bits é um fato novo. Essa migração pode influenciar na efetividade dos ataques já existentes e dificultar a criação de novos. O estudo das características de novas tecnologias e ataques é importante tanto para proteção como para observar as consequências futuras da evolução tecnológica.

Palavras-chave: buffer overflow, shellcode, 64 bits, 32 bits, segurança.

A comparative study on buffer overflow attacks in 32 and 64 bits systems

ABSTRACT

Buffer overflow vulnerabilities have been, throughout the modern history of computing, one of the most common forms of faults in the security area. Attacks of this kind were made possible mainly by the development of a culture of programming oriented only by systems performance. Issues related to security area were, for a long time, overlooked and the result is a high number of vulnerable systems that require corrections. The shift from 32-bit to 64-bit is a new fact. This migration can influence the effectiveness of existing attacks and hinder the creation of new ones. The study on features of new technologies and attacks is important both for protection and to comply with the future consequences of technological change.

Keywords: *buffer overflow*, shellcode, 64 bits, 32 bits, system security.

LISTA DE FIGURAS

SUMÁRIO

1 MOTIVAÇÃO

Informações pessoais, financeiras e acadêmicas são cada vez mais armazenadas e utilizadas através do uso de computadores. A segurança dessas informações é requisito essencial para sua adoção por um número maior de pessoas. Infelizmente, uma grande parte dos sistemas contém falhas graves de segurança, muitas vezes devidas à falta de conhecimento dos desenvolvedores sobre o assunto. Em especial, a falta de prevenção contra falhas do tipo *buffer overflow* tornam a tarefa de um atacante muito mais simples, ao permitir o acesso e execução de código arbitrário no sistema de suas vítimas.

Embora não seja novo, sendo algumas das primeiras explorações dessas vulnerabilidades datadas do final da década de 1980, como, por exemplo, o verme da internet desenvolvido por Morris no ano de 1988 (FISHER, 1988) , ainda hoje vulnerabilidades muito semelhantes ainda são relatadas (CVE-2015-0235, CVE-2015-8126).

A crescente adoção de sistemas de 64 bits, assim como a adoção de qualquer nova tecnologia nos motivas a estudá-las. De maneira especial, a possibilidade de geração de endereços inválidos, pela maneira como foi realizada a implementação atual desses sistemas, nos leva ao questionamento: serão esses novos sistemas mais seguros que seus antecessores?

Procuramos, ao longo deste trabalho, estudar algumas características que poderão facilitar ou dificultar a maneira como os ataques são executados. O capítulo 2 traz uma introdução sobre o assunto, indicando alguns conceitos e contextos históricos. O capítulo 3 tem como foco principal a análise da estrutura de uma sub-rotina, pois é através dos elementos envolvidos na chamada de sub-rotinas que a exploração de um ataque pode ser efetuada. O capítulo 4 faz uma análise mais detalhada de um ataque de *buffer overflow* e as principais diferenças existentes entre as arquiteturas de 32 e 64 bits. O capítulo 5 traz uma lista de métodos utilizados atualmente para tentar impedir ou, ao menos, dificultar a ação de um atacante. Por fim, no capítulo 6 as conclusões sobre a eficácia dos métodos mencionados no capítulo anterior, bem como uma breve análise sobre as mudanças de arquitetura, são apresentadas. Ainda deixamos como incentivo para trabalhos futuros a possibilidade de um estudo mais aprofundado em relação aos métodos de compatibilidade existentes e não abordados nesse trabalho.

2 INTRODUÇÃO

O avanço da tecnologia nos últimos anos vem ocorrendo de maneira cada vez mais acelerada. Na área da computação, em especial, a cada nova descoberta a utilização dos recursos atuais, ou a inserção de novos recursos, faz com que a capacidade de processamento, transmissão e interpretação de dados ganhe novo fôlego. Juntamente com esses avanços, novas ameaças às informações que trafegam através desses sistemas computacionais, também estão sendo desenvolvidas e aprimoradas.

Apesar dos significativos avanços na área de segurança da computação, dados e informações mantidos e utilizados por computadores estão mais vulneráveis do que nunca. Cada novo grande avanço tecnológico na computação traz consigo o desenvolvimento de novas ameaças, que necessitam de novas soluções em relação à segurança. Além disso, o avanço da tecnologia acontece em um ritmo mais rápido do que essas soluções podem ser desenvolvidas (GASSER, 1988).

Existe uma grande dificuldade em desenvolver sistemas capazes de proteger corretamente informações neles contidas, apesar do avanço na área de desenvolvimento de sistemas, que torna corriqueira a produção de novos sistemas com milhões de linhas de código. Apesar disso, nenhum sistema foi capaz de garantir seu funcionamento sem nenhum tipo de falhas. A presença de erros é inerente ao desenvolvimento e, apesar disso, a maioria dos sistemas realiza de forma adequada a função para a qual foi projetado.

De forma geral, questões de segurança são vistas como um problema operacional da área relacionada à tecnologia da informação, focada em proteger computadores e redes de atacantes mal intencionados e contra brechas de segurança. Ou, ainda, pensamos em segurança da informação como sendo unicamente a preocupação em proteger informações relevantes e pessoais armazenadas em um formato digital. Apesar disso, cada vez mais a falta de uma correta abordagem em relação à segurança de software tem se tornado a fonte de diversas vulnerabilidades para muitas organizações (ALLEN, 2008).

Quando falamos em segurança de sistemas computacionais, o termo vulnerabilidade pode possuir diversos significados. Entre eles, os mais aceitos são:

1. “Vulnerabilidade é a intersecção de três elementos: Um sistema suscetível a falhas, o acesso de atacantes à falha e a capacidade de exploração da falha” (SPI, 2009).

2. “Uma falha ou ponto fraco no projeto, implementação, operação ou gerenciamento de um sistema que pode ser exercitado (desencadeado acidentalmente ou explorado

intencionalmente) e resulta em uma brecha de segurança ou em uma violação da politica de segurança do sistema ou organização" (NIST, 2008).

3. "A existência de uma fraqueza, erro de projeto ou implementação, que pode levar a um evento inesperado e indesejável, comprometendo a segurança do sistema computacional, rede, aplicação ou protocolo envolvido" (ENISA, 2012).

Assim, vulnerabilidade pode ser definida como um erro ou falha existente no processo de projeto, desenvolvimento, implantação ou utilização de um software que permita a um agente (interno ou externo) o acesso indevido a documentos ou áreas do sistema em geral que não condizem com o nível de acesso que lhe é autorizado.

Podemos, ainda, classificar as diferentes vulnerabilidades encontradas durante a evolução dos sistemas computacionais em diversas categorias. Essas categorias podem ser definidas utilizando estratégias baseadas nas causas, severidades, impactos e fontes das vulnerabilidades. A seguir, são apresentadas oito categorias de vulnerabilidades. A classificação utilizada é a mesma empregada pelo "*National Vulnerability Database of the National Institute of Standards and Technology"*. Essa classificação usa como base as causas encontradas para que as vulnerabilidades existam (ALHAZMI, 2006; , NVD 2005; SECURITY FOCUS 2002).

1. Erro de validação de entrada, erro de condição de fronteira, *buffer overflow*: estes tipos de vulnerabilidades são caracterizados pela falha na verificação e identificação de entradas incorretas e na verificação de leitura ou escrita em endereços de memória inválidos.

2. Erro de validação de acesso: estas vulnerabilidades decorrem de falhas ao aplicar os privilégios de acesso corretos aos usuários que estão utilizando o sistema.

3. Erro no gerenciamento de condições excepcionais: estas vulnerabilidades aparecem devido a falhas em resposta a dados ou condições inesperadas.

4. Erro de ambiente: estas vulnerabilidades são desencadeadas por condições especificas do ambiente computacional em que o sistema é executado.

5. Erro de configuração: estas vulnerabilidades são decorrem de uma configuração inadequada do sistema.

6. Condição de corrida: estas vulnerabilidades são causadas pela serialização inadequada das sequências de ações dos processos que fazem parte do sistema.

7. Erro de projeto: estas vulnerabilidades são causadas por erros durante o projeto do sistema desenvolvido.

8. Outras: incluem vulnerabilidades que não pertencem aos tipos listados acima.

Embora existam diversos outros modelos de classificação de vulnerabilidades (SEACORD, 2005; MEUNIER, 2008), e de esta não ser uma classificação exaustiva, podemos ter uma ideia das diversas razões que levam um sistema computacional a conter falhas que possam ser exploradas, dando acesso indevido a atacantes mal intencionados.

Apesar de conhecidas, as diversas classificações de vulnerabilidades ainda existem, e muitas outras estão surgindo. Ao longo da evolução da ciência da computação, diversas tentativas foram feitas para tentar acabar com essas falhas. O desenvolvimento de novas linguagens de programação, bem como o incentivo a programadores para que utilizassem técnicas mais seguras de programação, demonstrou-se produtivo[1], porém ainda temos que lidar com código escrito antes desses avanços. Grande parte dos sistemas comerciais utilizados atualmente por empresas possui alguma característica ou mesmo sua completa funcionalidade vinculada a modelos antigos de desenvolvimento e de linguagens de programação.

Linguagens de programação são notações utilizadas por desenvolvedores para instruir um computador sobre seu funcionamento. Antes da metade da década de 1940, operadores de computadores realizavam a programação manualmente, sendo necessária a configuração de interruptores para ajustar as ligações internas do computador, para que este realizasse a tarefa desejada (LOUDEN, 2011). Utilizando essa técnica, era possível realizar efetivamente a comunicação necessária entre o ser humano e a máquina, porém desenvolver programas dessa forma consistia na tarefa de desmontar e reconfigurar todo o hardware do computador, que era cara e propensa a erros.

Com o objetivo de facilitar essa tarefa e diminuir o risco de erros gerados por falha humana, foram desenvolvidas ferramentas que facilitassem a comunicação na passagem de instruções aos computadores. Essas ferramentas são chamadas linguagens de programação.

Um dos primeiros grandes avanços nesta área ocorreu ao final da década de 1940. Em 1946, um cientista chamado John Von Neumann, propôs um novo projeto de computador baseado em um modelo conhecido como "*computador com programa armazenado"* (BURKS, 1982). Von Neumann propôs que as instruções que controlam a operação do computador fossem codificadas de forma binária e armazenadas na própria memória do computador, juntamente com os dados necessários para a realização da computação. Assim, quando fosse necessária a solução de novos problemas, ao invés de realizar a reconfiguração de todo sistema elétrico dos computadores, seria preciso, apenas, reescrever algumas

sequências de instruções (CENCAGEBRAIN, 2012). Dessa forma, para realizar a programação o programador escreveria sequências de 1's e 0's, cada sequência poderia representar um instrução ou um dado na realização da computação. O computador então, interpretaria as sequências de instruções, e realizaria a computação com os dados fornecidos. Esse tipo de linguagem é conhecido como "*linguagem de máquina"*.

Apesar da grande revolução produzida pelo uso deste sistema em pouco tempo os programadores começaram a identificar dificuldades que poderiam facilmente levar a falhas no desenvolvimento de programas. Entre elas, podemos destacar a utilização de códigos numéricos para especificar operações a serem realizadas no processo de computação. A primeira grande melhoria em relação à linguagem de máquina foi a chamada "*linguagem de montagem"*. A linguagem de montagem é a linguagem de programação simbólica que mais se aproxima da linguagem de máquina (IBM). Nesse tipo de linguagem, programadores utilizam símbolos para representar as operações que devem ser realizadas pelo computador. O nome dado a esse tipo de linguagem vem da necessidade da utilização de um "*montador"* para traduzir os símbolos utilizados em linguagem de máquina. Devido à proximidade com a linguagem de máquina essa tradução pode ser feita de maneira direta, onde um símbolo representa uma operação a ser realizada. Assim, um montador pode ser definido como:

"*Um montador é um tradutor capaz de traduzir instruções (escritas em linguagem simbólica) em instruções para uma dada arquitetura alvo (em linguagem de máquina), em uma relação de um para um"* (SALOMON, 2004)

Por volta da década de 1950, surgiram as primeiras linguagens de programação conhecidas como linguagens de alto nível (UEW, 2014). As linguagens desse tipo se diferenciam dos modelos anteriores por estarem mais próximas das linguagens utilizadas para comunicação entre seres humanos, tanto em questões gramaticais como em questões sintáticas. Essas linguagens são ditas de alto nível por suportarem um sistema de desenvolvimento com um nível elevado de abstração, permitindo ao programador manter seu foco no problema a ser solucionado (CUNNINGHAM, 2004). Uma única instrução nesse tipo de linguagem pode ser mapeada para diversas instruções em linguagens de nível mais baixo.

Para que um computador possa realizar as operações descritas em uma linguagem de alto nível, é necessário que primeiro estas operações sejas traduzidas para um conjunto de instruções que possam ser processadas e corretamente interpretadas pelo computador. A ferramenta responsável por realizar essa correta tradução é chamada de compilador.

Um compilador pode ser definido com um programa que lê uma sequência de instruções escritas em uma linguagem e traduz estas instruções para uma sequência equivalente em outra linguagem. A linguagem original, também chamada de linguagem fonte, é, normalmente, uma linguagem de alto nível, e para que o computador possa realizar a tarefa que foi descrita, a linguagem para a qual a tradução é realizada deve ser a linguagem de máquina. O compilador se diferencia de um montador em diversos aspectos, dentre eles a possibilidade de realizar a tradução no formato "um para muitos", onde uma única instrução na linguagem original pode ser traduzida para muitas instruções na linguagem destino.

Para realizar a compilação, ou a tradução de um código fonte para um código que possa ser executado por um computador, o compilador deve passar por diversas fases. Em cada fase o compilador transforma o programa fonte de uma representação para outra (AHO, 1986). Tipicamente, podemos dividir o trabalho do compilador nas seguintes fases.

1. **Análise léxica**: A primeira fase de um compilador. Seu objetivo principal é ler os caracteres de entrada e produzir como saída, uma sequência de *tokens* que um *parser* (analisador léxico) possa utilizar na próxima fase (CSE, 2012).

2. **Análise sintática**: A fase de análise sintática tem como objetivo determinar se uma sequência de *tokens* está ou não no formato adequado para aquela linguagem. Geralmente, utiliza-se uma representação em forma de árvore, denominada árvore de sintaxe, para representar a estrutura das sentenças (NORTHWOOD, 2009).

3. **Análise semântica**: As fases anteriores verificam se um programa é composto por uma sequência de *tokens* escritos em uma combinação sintaticamente válida (STANDFORD, 2012). A fase de análise semântica realiza a verificação do programa fonte em busca de erros semânticos. Ela utiliza é utilizada a estrutura definida na fase de análise sintática para identificar operandos e operadores em expressões. É nesta fase, também, que são identificadas e verificadas as informações referentes aos tipos dos dados que serão utilizados na fase de geração de código (AHO, 1986).

4. **Geração de código intermediário**: Após as fases anteriores, alguns compiladores geram uma representação intermediária do programa fonte (AHO, 1986). Essa representação deve ser gerada facilmente e deve ter como característica a fácil tradução para a linguagem de destino. Essa representação intermediária pode ser utilizada para otimização do código de forma independente da arquitetura alvo, bem como ser reutilizada para a geração da tradução final para diversas arquiteturas.

5. **Otimização**: O código intermediário gerado contém trechos ineficientes, devido à maneira individual como cada expressão é analisada nas fases anteriores. O objetivo da fase de otimização é aplicar um conjunto de regras para detectar estes trechos ineficientes e substituí-los por trechos mais eficientes. Essas otimizações podem ser aplicadas tanto no código intermediário, de forma independente da arquitetura alvo, bem como no código final (RICARTE, 2008)

6. **Geração de código**: É a última etapa do processo de compilação. Nesta fase, é gerado o código final para a máquina alvo. É necessário que o compilador conheça a estrutura de dados que deve ser utilizada, a fim de poder selecionar localizações de memória para as variáveis do programa. Após, as instruções intermediarias geradas anteriormente são traduzidas para uma sequência de instruções, que realizam a mesma tarefa em nível de máquina (AHO, 1986)

Após o término das 6 fases descritas acima, o código gerado está pronto para ser executado em um computador com a arquitetura escolhida no momento da compilação. A arquitetura escolhida tem grande importância na execução do programa. A arquitetura é um conjunto de regras e métodos que descrevem a funcionalidade, organização e implementação de um sistema computacional. Ela descreve de forma abstrata as capacidades e modelos de programação de um computador, sem levar em conta uma implementação específica (CLEMENTS, 2000)

O primeiro trabalho de destaque na área de arquitetura de computadores, foi resultado de um esforço conjunto entre John Von Neumann e os desenvolvedores do "*Electronic Numerical Integrator and Calculator (ENIAC)"*, na Escola de Engenharia Elétrica da Universidade da Pensilvânia. O ENIAC foi a primeira máquina de calcular eletrônica de propósito geral com alguma importância (SMITH, 1988).

Com o término do projeto ENIAC, dois pesquisadores, Eckert e Mauchly, começaram a desenvolver a ideia de uma máquina ainda mais poderosa, chamada EDVAC ("*Electronic Discrete Variable Automatic Computer"*). Um dos principais objetivos deste projeto era evitar a difícil tarefa de configurar externamente, através de conexões por fios, a resolução de problemas, como no ENIAC. A solução encontrada foi fornecer ao EDVAC uma memória capaz de armazenar as instruções que deveriam ser executadas. Assim, o computador poderia ser configurado para solucionar outro problema, com a colocação de um conjunto diferente de instruções na memória. Esse tipo de abordagem é um conceito conhecido como "*programa armazenado"*.

Em 1946, John Von Neumann em colaboração com Arthur Burks e Herman Goldstine projetaram um novo computador, no "*Institute for Advanced Study"* da universidade de Princeton, conhecido posteriormente como máquina IAS. Em junho daquele ano foi publicado então o primeiro relatório oficialmente distribuído, que descrevia em detalhes os fundamentos da arquitetura de computadores. O relatório obteve tamanha relevância, que os computadores produzidos posteriormente, utilizando o conceito de programa armazenado, foram chamados de máquinas com arquitetura de Von Neumann.

A assim chamada arquitetura de Von Neumann descreve que um computador de propósito geral deve conter partes principais relacionadas a aritmética, armazenamento de memória, controle e uma conexão para comunicação com um operador humano. Também se espera que, idealmente, após o inicio da computação, a máquina seja totalmente independente do operador (BURKS, 1982).

Segundo a descrição de Von Neumann, um computador deveria ser capaz de armazenar não apenas os dados necessários para o cálculo desejado e seus valores intermediários, mas também as instruções que regem a tarefa a ser executada sobre estes dados. A arquitetura de Von Neumann propôs então que estas instruções fossem codificadas em formato numérico, permitindo assim que tanto dados como instruções residissem na mesma memória.

Uma vez que a memória é utilizada apenas como unidade de armazenamento, tanto de dados como de instruções, é necessário que haja um componente capaz de executar automaticamente estas instruções. A unidade de controle é responsável pela execução das instruções contidas na memória. Para isso é necessário que a unidade tenha conhecimento da localização das instruções na memória. Cabe à unidade de controle a tarefa de comandar a memória na transmissão das instruções e dos dados para os registradores adequados, possibilitando assim que a computação possa ser realizada.

Quando os dados necessários para a computação foram devidamente carregados nos registradores, é função da unidade aritmética realizar as operações necessárias para que os cálculos sejam executados de forma correta. Depois de concluída a etapa de operações aritméticas, a unidade controle direciona os resultados para os locais apropriados na memória e procede para a próxima instrução. Ao fim da computação, o operador humano deve ser informado do resultado.

São as instruções armazenadas na memória que descrevem para o computador a tarefa que deve ser realizada. O programador, através de uma linguagem de programação, informa as ações que devem ser tomadas para que o resultado desejado seja atingido. Caso a

linguagem utilizada seja de alto nível, o compilador traduz essas ações em instruções para que possam ser executadas pelo computador.

Cada arquitetura possui um conjunto de instruções próprio, projetado para atender os propósitos da máquina sendo desenvolvida. O conjunto de instruções é a parte da arquitetura de computadores relacionada diretamente à programação.

A utilização de 32 bits como tamanho de palavra no conjunto de instruções IA32 tornou-se um fator limitante para a capacidade dos microprocessadores modernos. O tamanho da palavra em uma máquina define a quantidade de memória virtual que pode ser endereçada e utilizada pelos programas. Para uma palavra de 32 bits, o espaço máximo possível de ser utilizado é de 4 gigabytes (BRYANT, 2005). Com a constante evolução das tecnologias de produção de memória, e a diminuição dos preços (MCCALLUM, 2015), tornou-se viável a aquisição de quantidades de memória superiores às que podem ser efetivamente utilizadas por esses sistemas.

A necessidade de uma arquitetura capaz de endereçar quantidades maiores de memória surgiu a partir de aplicações que fazem uso de grandes quantidades de dados, como, por exemplo, aplicações em áreas científicas, bancos de dados, mineração de dados e servidores de alta performance. Para que estas pudessem realizar a manipulação dessas grandes quantidades de dados, era necessário realizar acessos a dispositivos de memória auxiliar, como discos magnéticos, para efetivamente carregar os dados para a memória principal do computador à medida que estes eram necessários (BRYANT, 2005; AMD64, 2013).

3 ANÁLISE DE SUB-ROTINAS

Vulnerabilidades do tipo *buffer overflow* têm sido, ao longo da história moderna da computação, uma das formas mais comuns de falhas na área de segurança (COWAN, 2000). Pela grande quantidade de falhas desse tipo, existente em diversos softwares (CRABB, 1997; CA, 2003; CVE-2009-2550), os ataques contra essa classe de vulnerabilidade representam uma porção substancial de todos os ataques na área de segurança. A grande quantidade de material encontrado na internet (ONE, 1996; UDEL, 20015; MAD 2012; INSECURE 2006), permite que um atacante mal intencionado possa, facilmente, planejar, codificar e executar um software capaz de explorar essas vulnerabilidades em sistemas desprotegidos.

Para entendermos o funcionamento de um ataque desse tipo, primeiro precisamos defini-lo:

"*Um ataque de buffer overflow utiliza operações de manipulação de memória com o objetivo de ultrapassar os limites de um buffer, resultando na modificação de um endereço de desvio para que este aponte para um código malicioso ou inesperado."* (PIROMSOPA, 2006)

De maneira geral, um ataque de *buffer overflow* tem por objetivo corromper a função de um programa que está sendo executado com um alto nível de privilégios. Dessa forma, o atacante pode controlar o programa de tal forma que seja possível, através dele, controlar o computador alvo. Para atingir seu objetivo, um atacante deve conseguir realizar duas tarefas (COWAN, 2000):

1. Organizar o código necessário para o ataque e torná-lo disponível para acesso ao programa alvo dentro de seu espaço de endereçamento.
2. Fazer com que o programa execute o código injetato, com os dados necessários corretamente ajustados.

O funcionamento de um ataque de *buffer overflow* é possível devido ao modelo de organização adotado nos sistemas atuais, devido a falhas tanto no planejamento quanto na implementação de algumas linguagens de programação e devido à má utilização de ferramentas ou à falta de conhecimento dos desenvolvedores no momento da codificação do software.

Quando um programa é executado em um sistema, esse sistema irá realizar a alocação de uma área de memória contínua, para armazenar os diversos tipos de dados necessários. O

espaço de memória onde os dados são armazenados é chamado de *buffer*. Quando a cópia de dados de um string com tamanho superior para um *buffer* de tamanho inferior é permitida, devido à falta de verificação dos limites do *buffer* de tamanho inferior, ocorre o "*estouro"* e a área de memória adjacente ao *buffer* de destino dos dados será sobrescrita (FU, 2012).

3.1 Sub-rotinas

Sub-rotinas podem ser descritas como uma sequência de instruções, autocontidas, desenvolvidas para realizar de forma independente uma tarefa especifica. Elas são, por si só, partes de programas capazes de serem reutilizadas por diversos outros programas.

Um programa não precisa utilizar sub-rotinas para executar a tarefa para a qual foi projetado. Porém, organizar o programa em sub-rotinas costuma ser vantajoso. Quando um programa é desenvolvido a partir de um conjunto de sub-rotinas, as diversas atividades necessárias para que a computação seja concluída podem ser melhor organizadas, permitindo que o programador se concentre nos detalhes de cada atividade de maneira independente. Outro ponto importante é que cada sub-rotina pode ser testada de maneira independente. Uma vez testada uma sub-rotina, sua utilização pode ser considerada segura e seu teste não é mais necessário, evitando que novos testes devam ser efetuados sobre todo o programa desenvolvido (WHEELER, 1952).

Quando uma sub-rotina é chamada, o fluxo de execução do programa é transferido para o local da memória onde se encontra armazenado o trecho de código que deve ser executado. Ao terminar a execução da sub-rotina, o programa deve ser capaz de continuar suas operações a partir do ponto posterior à invocação da sub-rotina. Com o objetivo de manter essas informações salvas e, ao mesmo tempo, permitir a invocação subsequente de sub-rotinas por outras sub-rotinas, os sistemas costumam utilizar uma estrutura denominada "pilha".

Quando um compilador realiza a tradução do código de uma sub-rotina, algumas instruções são adicionadas no inicio e fim do código traduzido. O conjunto de instruções adicionadas no início de uma sub-rotina é conhecido como prólogo e tem por objetivo preparar a pilha e os registradores necessários para execução da sub-rotina. O código adicionado ao fim da sub-rotina, denominado epílogo, tem por objetivo restaurar a pilha e os registradores para o estado anterior à invocação da sub-rotina.

3.2 A Pilha

A pilha, em termos de computação, é uma estrutura capaz de armazenar dados. Esses dados devem ser inseridos e removidos seguindo como princípio uma ordem pré-estabelecida: o último elemento inserido na pilha deve, obrigatoriamente, ser o primeiro elemento a ser removido da pilha. A pilha é uma estrutura com acesso limitado, apenas duas operações são permitidas: "*push*", que insere um novo elemento no topo da pilha e "*pop*", que remove o elemento no topo da pilha (CMU, 2015).

Estruturas que utilizam o conceito de pilha são comuns em diversas áreas dentro da computação. Em especial, quando falamos em termos de sistemas operacionais e arquiteturas e organização de computadores, uma de suas principais funções está relacionada ao armazenamento de informações relacionadas às sub-rotinas que estão sendo utilizadas por um programa. Quando utilizada dessa forma, esse tipo de estrutura é denominada "pilha de execução" ou "pilha de chamadas".

A pilha de execução recebe este nome porque, toda vez que uma sub-rotina é executada ou chamada pelo sistema, uma nova entrada, denominada quadro de pilha ou registro de ativação, é inserida no topo dessa estrutura. Quando a execução da sub-rotina termina, seu registro de ativação é removido da pilha e o espaço antes ocupado por ele, pode ser utilizado para a execução de novas sub-rotinas

Nos modelos de organização atuais, a pilha de execução é normalmente implementada como uma área de memória contínua. Se o crescimento da pilha ocorre dos endereços mais baixos da memória em direção aos endereços mais altos, ou vice-versa, é uma decisão tomada durante o projeto do sistema. Em muitas das arquiteturas atuais os projetistas optaram por manter a base da pilha em endereços de memória mais elevados. Com isso, à medida que novas informações são inseridas na pilha, o topo da pilha migra para endereços de memória cada vez mais baixos.

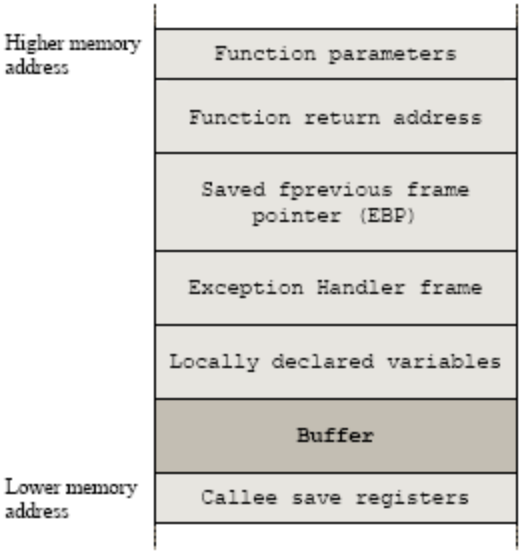
3.3 Registro de ativação

A pilha de execução é dividida em porções contínuas. Essas porções, conhecidas como quadros ou registros de ativação, armazenam informações associadas às sub-rotinas em execução e do programa que as contém (GNU, GDB). Quando a execução de um programa é

iniciada, a pilha de chamadas é formada por apenas pelo registro referente a função principal daquele programa. À medida que novas sub-rotinas são chamadas pelo programa que está sendo executado, novos quadros são criados e armazenados no topo da pilha. Ao término da execução de uma sub-rotina, cujo quadro encontra-se no topo da pilha, o registro de ativação associado à mesma é removido e o fluxo de execução prossegue a partir do ponto, no trecho do código chamador, em que a chamada para a sub-rotina foi realizada.

A fim de realizar a tarefa para a qual foi projetada, a sub-rotina necessita de dados que forneçam as informações requeridas para a computação, denominados parâmetros. Da mesma maneira, é necessário armazenar o endereço para o qual o o fluxo de execução deve retornar, após o término da execução da sub-rotina. Essas informações são partes integrantes de um registro de ativação (figura 1).

Figura 1. Registro de Ativação



Fonte: tenouk.com/Bufferoverflowc

Um registro de ativação é formado por diferentes seções. Cada seção é responsável pelo armazenamento de um tipo de dado que, juntos, compõem o quadro. Os dados que devem ser armazenados, o local onde devem ser armazenados e a ordem na qual esse armazenamento deve ser feito, variam de acordo com a arquitetura do sistema e a convenção de chamada utilizada. Quando nos referimos à linguagem C, as seções e dados presentes no registro, normalmente, são os seguintes:

- Parâmetros da sub-rotina: para que uma sub-rotina possa efetuar as operações necessárias para a conclusão de sua tarefa, é fundamental que esta tenha disponíveis todas as informações necessárias. Quando ocorre a chamada de uma sub-rotina, algumas dessas informações são fornecidas como parâmetros. Esses parâmetros são, então, armazenados na pilha de execução para posterior uso.

- Endereço de retorno: quando uma sub-rotina é chamada, o fluxo de execução do programa é alterado. O código que deve ser executado encontra-se em um endereço de memória fora da sequência normal. Antes de iniciar a execução deste trecho de código, é preciso armazenar o endereço a partir do qual a execução deve prosseguir após o término da execução da sub-rotina, denominado "endereço de retorno".

- Registro de ativação anterior: da maneira como são implementadas, sub-rotinas têm a capacidade de realizar a chamada de novas sub-rotinas. Quando esse encadeamento de chamadas ocorre, novos registros de ativação vão sendo criados no topo da pilha. Assim, para manter a ordem correta neste encadeamento de chamadas, é necessário armazenar as informações referentes a sucessivas chamadas. As informações contidas no "registro de ativação anterior" possibilitam esse encadeamento e manter a consistência entre a ordem de chamada e de retorno das sub-rotinas.

- Gerência de exceções: se uma sub-rotina possui construções para o gerenciamento de exceções, como por exemplo blocos "*try/catch"*, essas informações também são armazenadas na pilha de execução.

- Variáveis locais: assim como o programa principal, sub-rotinas também são capazes de declarar espaços de memória onde informações e resultados podem ser armazenados. Essas áreas podem ser acessadas apenas dentro do contexto da sub-rotina e são armazenadas na pilha juntamente com as demais informações.

- Buffer: diversas vezes, uma sub-rotina necessita reservar um espaço de memória durante sua execução. Juntamente com as demais seções, o registro de ativação, possui uma área disponível para essa finalidade.

Todas essas seções desempenham um papel importante para garantir que as sub-rotinas executem de forma correta suas operações. No decorrer da história da computação, com o desenvolvimento constante de novas linguagens e arquiteturas, diferentes informações foram sendo adicionadas ou removidas no processo de montagem da pilha. Embora existam diferenças, o objetivo final é sempre o mesmo: permitir a utilização de sub-rotinas no desenvolvimento de aplicativos.

3.4 Convenções de chamada

Quando uma sub-rotina é chamada e sua execução é iniciada, é necessário que estejam bem definidas as informações necessárias para as operações (operandos), as operações a realizar, os locais onde devem ser armazenados os dados e a maneira como o resultado deve ser apresentado. Caso contrário, a execução de uma sub-rotina pode não ocorrer como esperado ou até mesmo não ocorrer.

Convenções de chamada especificam como os argumentos são fornecidos às sub-rotinas, como os valores gerados pela execução destas sub-rotinas são devolvidos, como é realizada a invocação de uma sub-rotina e como deve ser feita a administração da pilha e do registro de ativação (WIKIBOOKS, 2011). Somente seguindo essas convenções, é possível que os compiladores traduzam códigos de chamadas de funções escritas em linguagens de alto nível, como C e C++, para chamadas em linguagens de baixo nível, de maneira que possam interagir com programas gerados por outros compiladores.

Arquiteturas como a x86 da Intel não usam o mesmo conceito de chamada de sub-rotinas que as linguagens de alto nível. Considerando um nível menos elevado de abstração, uma chamada a uma sub-rotina nada mais é do que uma série de desvios entre diversos blocos de código. Portanto, é necessário que tanto o trecho de código que realiza a chamada de uma sub-rotina quanto a própria sub-rotina saibam que passos devem ser tomados para a conclusão de suas tarefas. Ao longo do desenvolvimento da computação, diversas convenções de chamada foram criadas, sendo que, na linguagem C, três delas são predominantes: **`_cdecl`**, **`_stdcall`** e **`_fastcall`.**

3.4.1 CDECL

`_cdecl` é a convenção de chamada padrão usada nos programas escritos em C e para chamada de funções de escopo global em programas escritos em C++ (MSDN, 2015). Quando esta convenção é utilizada, a administração da pilha, incluindo a reserva e liberação de espaço na mesma, é de responsabilidade do trecho de código que realiza a chamada da sub-rotina. Os argumentos necessários para execução da sub-rotina são armazenados na pilha seguindo a ordem inversa de sua declaração, de forma que o primeiro argumento declarado fica no topo da pilha. Assim, a ordem na qual os argumentos são acessados pela sub-rotina é a

mesma usada em sua declaração. Ao terminar a execução da sub-rotina, é necessário que ocorra a limpeza da pilha. Segundo esta convenção, o código chamador é também responsável por esse processo.

Pela maneira como é implementada a convenção, o primeiro parâmetro necessário para a execução da sub-rotina sempre estará no topo da pilha, ficando facilmente acessível não importando o número de parâmetros esperados pela função. Essa caracteristica da passagem de parâmetros torna o **`cdecl`** a convenção ideal para ser utilizada em sub-rotinas que apresentam um número variado de argumentos (EAGLE, 2011).

Devido à necessidade de realizar a remoção dos parâmetros armazenados na pilha, é comum a inclusão, por parte do compilador, de trechos de código que façam os ajustes necessário para que o programa siga o fluxo de execução corretamente. Em se tratando de sub-rotinas que possuem um número variável de argumentos, o código que realiza a chamada é o ideal para realizar as operações de limpeza e ajuste necessárias, uma vez que ele tem informações sobre a quantidade de parâmetros que foram armazenados. Por sua vez, a sub-rotina chamada não sabe quantos argumentos irá receber, o que dificultaria a realização destas tarefas por ela.

3.4.2 STDCALL

Criada pela empresa Microsoft, **`_stdcall`** (MSDN, 2014) é uma convenção de chamada com características muito semelhantes às de **`_cdecl`**. Essa convenção é utilizada para chamada de funções presentes na API Win32. Essa convenção utiliza a mesma técnica para passagem de parâmetros da anterior: o primeiro argumento armazenado na pilha é o último declarado, de modo que o primeiro parâmetro necessário para operação da sub-rotina sempre fica no topo da pilha.

A principal diferença de **`_stdcall`** em relação à convenção **`_cdecl`** é a divisão de responsabilidades no momento de realizar as ações de limpeza da pilha. Enquanto na convenção descrita anteriormente essa tarefa era realizada pelo código chamador da sub-rotina, em **`_stdcall`** a tarefa fica sob responsabilidade da sub-rotina chamada. Para que possa realizar esse processo, a sub-rotina necessita conhecer a quantidade de parâmetros armazenados na pilha. Essa informação está disponível apenas para funções que administram um número fixo de parâmetros, pois tendo essa informação, a realização de ajustes e liberação da área ocupada pelos parâmetros se torna uma tarefa trivial.

Como consequência das escolhas realizadas pela convenção **`_stdcall`**, não é possível implementar com ela chamadas a sub-rotinas que utilizam um número variável de argumentos. A principal vantagem na utilização dessa convenção é a simplicidade com que é feita a limpeza dos parâmetros da pilha, dispensando a necessidade de código extra para esta tarefa. Assim, por evitar a codificação e execução de código extra para essa tarefa, programas que utilizam essa convenção, tendem a ser levemente menores e mais rápidos (EAGLE, 2011).

3.4.3 FASTCALL

A convenção de chamada conhecida por **`_fastcall`** se diferencia das anteriores, entre outros motivos, por utilizar passagem de parâmetros fora da pilha. Enquanto as convenções citadas anteriormente utilizam exclusivamente a pilha para armazenar os parâmetros passados para a sub-rotina chamada, a **`_fastcall`** procura armazenar esses parâmetros, sempre que possível, em registradores. O resultado desse tipo de técnica é a redução do número de operações realizadas com a memória, aumentando assim a velocidade de execução de uma chamada a sub-rotina (SENSEPOST, 2014).

A convenção **`_fastcall`** estabelece que os primeiros parâmetros sejam alocados em registradores, acelerando assim o acesso a eles pela sub-rotina chamada. Como esta não é uma convenção padronizada, a definição de quis registradores serão utilizados varia conforme o compilador e a arquitetura. Normalmente, o compilador seleciona alguns dos registradores de propósito geral disponíveis na arquitetura alvo e passa os argumentos para as funções através desses (KASPERSKY, 2003). Quando o número de parâmetros a passar é superior à quantidade de registradores disponíveis, é utilizada a pilha. Nesse caso, é responsabilidade da sub-rotina chamada realizar a liberação da área de memória ocupada pelos parâmetros passados através da pilha (GCC, 2015).

A **`_fastcall`** é, em geral, uma convenção associada à arquitetura Intel x86. A comparação entre as versões da **`_fastcall`** que utilizam arquitetura de 32 bits e 64 bits mostra algumas diferenças entre elas. Na primeira, o número de parâmetros passados através de registradores é, geralmente, limitado a dois, podendo variar de acordo com o tipo de parâmetro. Além do aumento da capacidade de endereçamento, a arquitetura x86-64 conta com novos registradores, permitindo assim que mais parâmetros sejam passados desta forma

(MSDN, 2015). Atualmente, até quatro parâmetros são passados às funções utilizando registradores.

Convenções de chamadas são "*contratos"* estabelecidos entre rotinas chamadoras e chamadas para que possam realizar corretamente a comunicação necessária, de maneira a concluir suas tarefas. Além das três convenções citadas anteriormente, muitas outras foram desenvolvidas ao longo da evolução da computação, cada uma buscando melhorar algum aspecto do processo de chamada e retorno de sub-rotinas.

4 BUFFER OVERFLOW

Durante o processo de computação, diversas operações envolvendo a transferência de dados são efetuadas. Essas operações, em sua grande maioria, não causam problemas. Porém, quando mal executadas, tendem a provocar o término da execução da instância atual do programa. Um dos motivos para a ocorrência de erros é a cópia de dados de maneira indiscriminada entre dois *buffers*. Quando a quantidade de dados transferidos excede a capacidade de armazenamento do *buffer* de destino, uma parte desses dados acaba sobrescrevendo inadvertidamente áreas contíguas da memória, o que pode causar falhas que comprometem o funcionamento do sistema.

O preenchimento de um *buffer* além de seus limites não é o suficiente para caracterizar um ataque de *buffer overflow*. Regiões de memória sobrescritas acidentalmente, com dados aleatórios, tendem a causar erros do tipo Falha de Segmentação e não são caracterizados como uma exploração da falha, na medida em que não proporcionam vantagens ao "atacante". Porém, quando um usuário malicioso assume o controle dos dados que serão inseridos, a escrita além dos limites de um *buffer* torna-se uma falha de segurança que poderá ser explorada, garantindo ao usuário permissões e acessos indevidos (FU, 2012).

Para que tenha sucesso na exploração de uma falha do tipo *buffer overflow*, um atacante deve utilizar trechos de código que lhe forneçam acesso e controle da máquina alvo. O código utilizado pode ser injetado durante a exploração da falha ou ter sido previamente carregado através de outros programas. Normalmente, o código escolhido irá garantir ao atacante acesso a um terminal com o mesmo nível de privilégios do programa através do qual a falha foi explorada. Quando o código escolhido se encontra residente na máquina alvo, o usuário deve encontrar um meio de provocar a execução do mesmo.

Quando o usuário obtém sucesso na implantação e execução do código malicioso, a exploração da falha acontece de fato. Com o controle do computador através de um programa com níveis de privilégio elevados, o atacante possui a habilidade de executar tarefas proibidas para os usuários comuns e acessar informações sigilosas, comprometendo a integridade e confidencialidade do sistema.

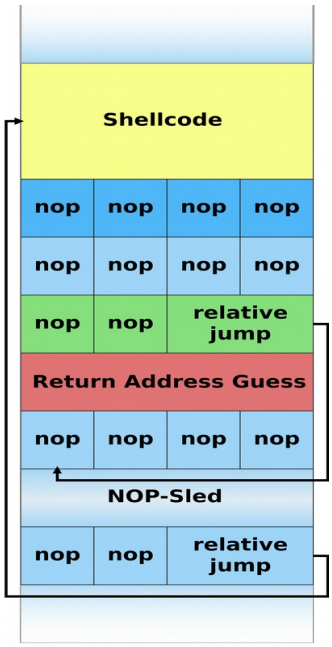
4.1 Shellcode

Um *shellcode* pode ser descrito como um trecho de código capaz de ativar uma interface de linha de comando em um sistema, permitindo que o atacante execute operações

através desta (VIRUSBTN, 2008). Quando nos referimos a aspectos mais específicos de segurança, podemos definir um *shellcode* como um trecho de código executado após ser injetado em um sistema, geralmente por meio de vulnerabilidades do tipo *buffer overflow*. O termo *shellcode* é utilizado por suas origens históricas, onde os primeiros ataques a sistemas tinham por objetivo executar uma interface de terminal através da qual o acesso ao sistema e a exploração da vulnerabilidade eram efetuadas (MCDERMOTT, 2011).

O desenvolvimento de um *shellcode* envolve um profundo conhecimento sobre a linguagem de montagem para a arquitetura na qual se deseja efetuar o ataque. De maneira geral, é necessário o desenvolvimento de um código diferente para explorar diferentes versões de sistemas operacionais em cada uma das arquiteturas existentes. Para realizar as ações necessárias, um *shellcode* costuma utilizar chamadas de sistema. Em consequência, a maior parte das implementações encontradas na literatura depende de um sistema operacional específico, uma vez que diferentes sistemas, utilizam chamadas e parâmetros próprios para executar uma dada tarefa (JASON, 2005).

Figura 2: Divisão de um shellcode



Fonte: commons.wikimedia.org/wiki/File:Nopsled.svg

Podemos dividir a construção de um *shellcode* em três partes distintas (figura 2), a saber: trilha de NOPs, *payload* e a zona de retorno. A zona de retorno é composta unicamente pelo endereço para o qual se deseja que a execução seja desviada. Devido a fatores de imprecisão, esse endereço é repetido diversas vezes, tentando garantir, desta forma, que uma

dessas repetições seja inserida no local correto da memória. Na seção denominada *payload* o atacante costuma inserir o código que deseja executar (*shellcode*, na figura 2). O objetivo do atacante é então sobrescrever o endereço de retorno de uma sub-rotina de modo que a execução seja desviada para o início deste segmento de código. Para garantir que a execução ocorra com sucesso, o atacante deve ter certeza de que seu código seja completamente executado. A função da trilha de NOPs é proporcionar ao usuário mal intencionado uma pequena margem para erros no desvio para o *shellcode*. Essa seção é composta por uma sequência de instruções NOP posicionada, classicamente, anteriormente ao código que deve ser executado, podendo haver variações. Assim, mesmo que o desvio ocorra para uma dessas posições, após a execução da sequência de NOPs o código existente na área de *payload* também será executado posteriormente (SONG, 2010). A sequência de valores mostrada abaixo é um exemplo de *shellcode*.

"\x48\x31\xd2\x52\xb0\x3b\x48\xb9\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x51\x48
\x8d\x3c\x24\x0f\x05" (EXPLOITDB-37384)

Através da execução do *shellcode* acima, o atacante ativa um terminal para comunicação com o sistema operacional. A partir desse terminal, com privilégios de execução elevados, o usuário malicioso é capaz de executar comandos no sistema invadido. A ativação do terminal é realizada através da chamada de sistema *execve*. A versão do mesmo código, em linguagem de montagem, pode ser vista a seguir.

```
section .text
    global start
start:
    xor rdx, rdx
    push rdx
    mov al, 0x3b
    mov rcx, 0x68732f2f6e69622f
    push rcx
    lea rdi, [rsp]
  syscall
```

A explicação detalhada de cada instrução executada foge do escopo deste trabalho. Em resumo, o código acima carrega o trecho de código responsável pela execução do terminal no

registrador requerido e, em seguida, efetua a chamada de sistema que irá se utilizar do registrador mencionado para iniciar a execução.

À medida que foram sendo desenvolvidos sistemas e ferramentas com o objetivo de aumentar o nível de segurança das aplicações e impedir que usuários mal intencionados obtivessem acessos e privilégios indevidos, o grau de sofisticação das técnicas utilizadas para efetuar a exploração de falhas também teve uma evolução. Uma das técnicas mais utilizadas atualmente é o ofuscamento, que busca descaracterizar o código do *shellcode* para que este não seja reconhecido por ferramentas de proteção. Um exemplo desta técnica é criptografar o código do *shellcode* utilizando uma chave escolhida aleatoriamente. Para que essa técnica seja executada corretamente, é necessário que um pequeno trecho de código seja executado antes da execução do próprio *shellcode*. Esse trecho é injetado juntamente com o *shellcode* e é capaz de realizar a decodificação dinamicamente, durante a execução (SONG, 2010).

A utilização de *shellcodes* eficientes é parte fundamental para que um ataque seja bem sucedido. É possível encontrar exemplos de códigos utilizados para exploração de vulnerabilidades na internet (EXPLOITDB, SHELLSTORM). Embora muitos desses não realizem mais a função para a qual foram projetados, devido à implementação de novas rotinas de segurança, é possível perceber que constantemente novas falhas, passíveis de exploração, são encontradas e em pouco tempo o desenvolvimento de ferramentas capazes de realizar o ataque é concluído.

O estudo aprofundado da maneira como são desenvolvidas essas ferramentas está além do escopo deste trabalho. Material de apoio para o leitor que deseja mais informações sobre o assunto pode ser encontrado na internet e em livros especializados sobre o tema (ANLEY, 2007).

4.2 *Buffer overflow* – 32 bits

Para exemplificar como ocorre um ataque de *buffer overflow* vamos tomar como base um sistema operacional de 32 bits. Atualmente esses sistemas representam grande parte dos existentes no mercado e sua grande base de software nos estimula a estudá-los.

Como citado anteriormente, um sistema de 32 bits é capaz de endereçar cerca de 4 gigabytes de memória. Essa quantidade de memória é suficiente para a maioria dos usuários comuns e, atualmente, apenas aplicações com taxas mais elevadas de manipulação de dados necessitam quantidades de memória maiores. Ataques de *buffer overflow* foram primeiramente descritos no artigo "*Smashing The Stack For Fun and Profit*" onde um *hacker*

conhecido como Aleph One expôs a fragilidade dos sistemas e demonstrou como sua exploração era possível. Diversos tipos de ataque baseado em *overflow* são possíveis, sendo que, na literatura, as principais referências encontradas demonstram ataques dos tipos *stack overflow* (ONE, 1996), *heap overflow* (CONOVER, 1999) e *string format* (LHEE, 2003). Como demonstração, nos concentraremos em um ataque à estrutura de pilha utilizando como base a linguagem C.

Ataques desse tipo tornaram-se possíveis, principalmente, pelo desenvolvimento de uma cultura de programação voltada exclusivamente ao desempenho dos sistemas. Questões relacionadas à área de segurança foram, por muito tempo, deixadas de lado e o resultado disso é um alto número de sistema vulneráveis que necessitam de correções. O alto número de atualizações e pacotes fornecidos diariamente por desenvolvedores de sistemas demonstra que, com o passar do tempo, a preocupação com questões de segurança vem aumentando, mas ainda estamos longe de vencer essa batalha.

O primeiro passo para entendermos o funcionamento de um ataque de *buffer overflow* é a identificação de um programa vulnerável.

O trecho de código mostrado na figura 3, apesar de simples, é capaz de demonstrar as vulnerabilidades encontradas em diversos sistemas. Os detalhes de operação do trecho citado serão apresentados a seguir.

Duas funções compõem o trecho de código apresentado, *main* e *func*. O ponto de inicio para a execução de programas escritos em C é tradicionalmente a função *main* e através dela as operações e chamadas das demais rotinas devem ser efetuadas. No trecho mostrado, o programa recebe dois argumentos: *argc*, através do qual é informada a quantidade de parâmetros que seguem e *argv*, um *array* no qual os valores dos parâmetros estão armazenados.

Em linguagens como C, Java ou Python, a indexação das posições dos elementos de um array é feita com a adição de um "deslocamento" em relação à sua base (endereço de memória onde inicia o array). Assim, para acessar o primeiro elemento utilizamos o índice 0, ex.: argv[0], indicando que não devemos adicionar nenhum valor de deslocamento ao endereço da base. O segundo elemento deve ser acessado utilizando-se o índice 1, indicando que o elemento está posicionado após um deslocamento a partir de sua base. Os demais elementos são acessados utilizando a mesma lógica.

Figura 3: Código Vulnerável.

```
void func(char *string)
{
        char buffer[32];
        strcpy(buffer, string);
}

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

Após ser iniciada sua execução, o programa realiza uma chamada à sub-rotina *func*. Como declarada, essa sub-rotina recebe um único argumento, uma sequência de caracteres, realiza a cópia desse argumento para a variável local *buffer*, com capacidade de armazenar até 32 caracteres, utilizando a função *strcpy*. Completado os passos anteriores. a execução da sub-rotina é encerrada e o controle da execução retorna para o ponto posterior à chamada da sub-rotina, onde a execução do programa, por fim, é encerrada.

4.2.1 A Falha

A realização de operações como as descritas no programa de exemplo são comuns em diversos sistemas. O trecho apresentado poderia, por exemplo, fazer parte de uma rotina de verificação de identidade de um usuário, onde a sequência de caracteres passada como argumento poderia representar o nome de acesso ou uma senha.

Embora aparentemente inofensivo, o trecho mencionado apresenta uma falha grave de segurança, permitindo a um usuário mal intencionado a exploração da mesma. A cópia de uma sequência de caracteres efetuada de maneira displicente, por falta de conhecimento ou descaso do programador, juntamente com o uso de uma linguagem permissiva como é a linguagem C, provoca a vulnerabilidade.

Diversas funções implementadas na linguagem C, especialmente as relacionadas à manipulação de sequências de caracteres, apresentam falhas (CERN, 2010). Na maioria dos

casos, essas falhas resultariam em erros do tipo *falha de segmentação,* quando um segmento de memória é acessado de maneira indevida, devido à sobreposição do código com dados inválidos que impossibilitam a continuação da execução do programa. Porém, quando um usuário malicioso se utiliza dessa falha através do uso de uma sequência de caracteres pensada para a exploração da vulnerabilidade, o resultado pode levar à execução de trechos arbitrários de código.

A função *strcpy* é uma entre várias funções implementadas e largamente utilizadas que apresentam falhas. Seu objetivo é realizar a cópia de uma sequência de caracteres para um local de destino a partir de uma dada origem. A falha dessa função está na falta de verificação da capacidade de armazenamento do destino da cópia. Assim, dada uma sequência de 33 caracteres que devem ser copiados para um *buffer* com capacidade de armazenamento de 32 caracteres, a função não irá se limitar a copiar uma quantidade de caracteres igual ao tamanho do *buffer*, sobrescrevendo o espaço de memória contíguo ao local de destino.

Quando o espaço sobrescrito pela utilização dessa função possui informações necessárias à execução do programa, sua sobrescrita poderá permitir a manipulação do fluxo de operação e das ações executadas pelo programa.

4.2.2 O Ataque

Quando um usuário mal intencionado identifica a presença de uma vulnerabilidade, ele pode começar o desenvolvimento de um ataque. Para isso, é necessário que haja um amplo conhecimento da arquitetura alvo, da linguagem de programação utilizada e da organização da memória do sistema. Essas informações podem ser encontradas na internet, livros, manuais técnicos e muitas delas podem ser deduzidas através da execução do mesmo sistema em um ambiente com características semelhantes de processador e memória..

Para começar a exploração de uma vulnerabilidade do tipo *buffer overflow*, um atacante deve primeiramente descobrir a quantidade de dados necessários para que a falha seja desencadeada. Uma maneira simples de realizar esse primeiro passo é através de uma sequência de tentativas. Quando uma entrada de dados é solicitada pelo programa, o atacante pode sistematicamente incrementar a quantidade de caracteres fornecidos em cada tentativa. Enquanto o programa continua sua execução de maneira consistente após a entrada de dados, podemos supor que o limite do *buffer* de entrada não foi ultrapassado. Caso ocorra um desvio do comportamento esperado do sistema, ou até mesmo um erro do tipo *falha de segmentação*, é possível concluir que posições de memória subsequentes ao *buffer* foram corrompidas pela

sobreposição dos dados de entrada. A partir de então, o desenvolvimento de um *shellcode* adequado para o ataque pode começar.

A principal dificuldade no desenvolvimento do *shellcode* que irá fornecer ao usuário a capacidade de acesso ao sistema encontra-se no cálculo correto do ponto a partir do qual a sobrescrita do endereço de retorno ocorre e o local para o qual o controle da execução deve ser desviado. De posse dessas duas informações, o atacante pode então sobrescrever o endereço de retorno na pilha com o endereço no qual se encontra o trecho de código malicioso que ele deseja executar.

Se um atacante possui acesso ao ambiente no qual o sistema alvo está sendo executado ou consegue reproduzi-lo localmente, a utilização de ferramentas de depuração de código que permitem a inspeção dos elementos e organização da memória pode auxiliá-lo na tarefa. Com a capacidade de uma análise em mais baixo nível, o usuário mal intencionado é capaz de realizar os cálculos necessários.

4.3 *Buffer overflow* – 64 bits

Recentemente, o avanço da tecnologia e o aumento da complexidade dos programas vem levando desenvolvedores à troca de uma arquitetura baseada em 32 bits por uma baseada em 64 bits. O impacto dessa mudança, quando relacionada a questões de segurança, ainda não é clara, mas alguns pontos merecem atenção e estudo.

A principal mudança nas características entre os dois sistemas é o aumento da capacidade de endereçamento. Enquanto, em um sistema de 32 bits, os endereços de memória estão limitados a 4 gigabytes (2^{32} bytes), os sistemas de 64 bits são capazes de endereçar, teoricamente, até 16 exabytes (2^{64} bytes) de memória. Atualmente, nenhum sistema é capaz de utilizar toda a capacidade de endereçamento dos 64 bits. Além disso, o aumento no espaço de endereçamento resulta no aumento significativo da estrutura de páginas utilizada pelos sistemas operacionais para gerenciar os elementos alocados. Por esses motivos, os projetistas dos processadores hoje disponíveis no mercado optaram pela não utilização de todo o espaço de endereçamento disponível.

Uma grande variedade de ataques do tipo *buffer overflow* tem como base a habilidade de um atacante descobrir endereços importantes dentro de um processo e modificá-los a seu favor. Algumas técnicas de proteção buscam adicionar dados à memória, de maneira aleatória,

com o objetivo de confundir o atacante. O simples aumento no espaço de endereçamento permite a essas técnicas inserir uma quantidade maior desses dados, aumentando sua efetividade, uma vez que, comparativamente, o espaço de endereçamento possível em um sistema de 64 bits é mais de 4 bilhões de vezes maior do que o de um sistema de 32 bits.

Os modelos atuais de processadores disponíveis no mercado utilizam apenas 48 bits para o endereçamento da memória. Dessa maneira, nem todos os endereços possíveis quando se usa 64 bits são válidos nesses sistemas. A arquitetura implementada pela desenvolvedora de processadores AMD, por exemplo, somente utiliza os 48 bits menos significativos, sendo os bits restantes (48 a 63) uma cópia do bit 47. Assim, os endereços válidos variam de 0x0 a 0x00007FFFFFFFFFFF e de 0xFFFF800000000000 a 0xFFFFFFFFFFFFFFFF. Essa faixa permite um endereçamento total de 254 terabytes

A existência de endereços inválidos é um fato novo e que pode contribuir para a segurança dos sistemas. Mesmo sem intenção, a não utilização de todos os endereços possíveis acrescentou um grau de dificuldade extra no desenvolvimento de programas capazes de realizar a exploração de falhas.

O desenvolvimento de programas capazes de explorar esses novos sistemas, deve levar em consideração, além das dificuldades já existentes, a necessidade de sobrescrever o endereço de retorno com um endereço de memória válido. Caso o endereço não seja adequadamente calculado, mesmo o deslocamento de um bit poderá comprometer a efetividade do ataque. Pela restrição imposta para que um endereço seja considerado válido, a grande maioria dos *exploits* desenvolvidos para as antigas arquiteturas de 32 bits, senão todos, se tornou obsoleta, se considerarmos apenas os modos padrões do sistema.

Apesar das mudanças encontradas na migração entre as duas arquiteturas já mencionadas, a exploração de falhas em sistemas de 64 bits ainda é possível (RINGZER0). O desenvolvimento de *shellcodes* para essa nova arquitetura, levando em consideração as mudanças citadas, já é uma realidade.

"\x48\xb9\x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe1\x08\x48\xc1\xe9\x08\x51
\x48\x8d\x3c\x24\x48\x31\xd2\xb0\x3b\x0f\x05" (EXPLOITDB-37362, 20015)

O *shellcode* acima foi desenvolvido para execução em um ambiente com um sistema operacional Linux x86-64 sendo executado em um processador Intel. O objetivo do código é a abertura de um terminal de comandos, *shell*, que permita ao usuário a execução de comandos com privilégios elevados. A chamada de sistema utilizada para isso é a que segue.

*int execve(const char *filename, char *const argv[], char *const envp[])*

Através da versão em linguagem de montagem do código, é possível identificar elementos inseridos na tentativa de contornar as limitações de endereçamento impostas pela utilização de apenas 48 dos 64 bits.

```
section .text
        global start
    start:
        mov rcx, 0x1168732f6e69622f
        shl rcx, 0x08
        shr rcx, 0x08
        push rcx
        lea rdi, [rsp]
        xor rdx, rdx
        mov al, 0x3b
        syscall
```

As linhas destacadas permitem ao atacante realizar a validação do endereço, levando em conta a existência de endereços inválidos. De forma semelhante à da arquitetura desenvolvida pela AMD, o processador Intel exige que os 16 bits não utilizados contenham valor 0. Dessa forma, ao realizar uma operação de deslocamento para a esquerda *shl* o atacante elimina esses bits. Para que o endereço fique correto a operação reversa, deslocamento para a direita, *shr*, deve ser realizada. Assim, no caso dos processadores Intel, os 16 bits não utilizados ficam com os valores adequados. Realizada a ação de validação dos endereços, o ataque pode proceder normalmente.

Embora a geração de endereços inválidos possa dificultar a utilização de ferramentas de ataques antigas e exigir a criação e novas ferramentas específicas para o momento atual, é de se esperar que no futuro todo o espaço de endereçamento possa ser utilizado, tornando válido todo e qualquer endereço de 64 bits. Portanto, enquanto o estado atual ainda se mostra minimamente mais favorável aos defensores, medidas de proteção devem ser desenvolvidas para reduzir esse risco no futuro.

5 MÉTODOS DE PREVENÇÃO

Durante a constante batalha entre desenvolvedores e atacantes, diversas técnicas (THOMAS, 2013; SHANECK, 2003) foram desenvolvidas para tentar impedir que a exploração de vulnerabilidades do tipo *buffer overflow* obtivesse êxito. Os métodos atuais podem ser divididos em três categorias: *análise estática, soluções dinâmicas e isolamento* (PIROMPOSA, 2006).

5.1 Análise Estática

Técnicas que utilizam esquemas baseados em análise estática têm por objetivo encontrar possíveis vulnerabilidades antes do lançamento do programa no mercado(?), assim o código vulnerável pode ser corrigido, evitando a falha. A metodologia utilizada parte da premissa de que, com o passar do tempo, acumulamos conhecimento de padrões de código ou sub-rotinas utilizadas que tornam o código suscetível a ataques. Assim, é possível analisar um código fonte ou um arquivo binário em busca desses padrões e substituí-los por código mais seguro. Por exemplo, sabendo que a sub-rotina "*strcpy"* da linguagem C realiza cópia de textos sem realizar a verificação das fronteiras, tornando-a um possível ponto de vulnerabilidade, podemos criar um perfil de busca capaz de encontrar qualquer utilização dessa sub-rotina, gerando um alerta ao desenvolvedor sobre cada possível ameaça encontrada (PIROMPOSA, 2006; PIROMPOSA, 2011).

5.1.1 Análise Léxica

Uma das formas de análise estática utilizada para encontrar vulnerabilidades que possam resultar em ataques do tipo *buffer overflow* é a utilizada pelas chamadas ferramentas de análise léxica. Esse tipo de análise é baseada na busca de *tokens*, tendo como referência uma base de sub-rotinas consideradas vulneráveis. Sempre que uma dessa sub-rotinas é encontrada no código em análise, um alerta é gerado. A base de dados desse tipo de ferramenta parte da ideia de que grande parte dos problemas relacionados a segurança são causados pelo uso incorreto de sub-rotinas presentes em bibliotecas. Muitas sub-rotinas são consideradas sempre vulneráveis, devido à sua implementação, e devem ser evitadas. Outras, levam à criação de falhas devido ao uso inadequado. Algumas ferramentas utilizam heurísticas para determinar se a maneira como a sub-rotina está sendo utilizada apresenta

algum padrão que pode levar a falhas que comprometam a segurança (POZZA, 2006). Técnicas que utilizam análise léxica são geralmente simples e rápidas, porém sua eficácia é limitada pelo fato de não utilizarem informações sintáticas e semânticas do programa para refinar suas buscas (LAROCHELLE, 2001).

5.1.2 Análise Semântica

Tentando melhorar o desempenho das ferramentas de análise estática, buscando reduzir a quantidade de falsos positivos existentes quando apenas técnicas de análise léxica são utilizadas, foram implementadas técnicas mais avançadas, comumente utilizadas em compiladores, como a criação de árvores de sintaxe abstratas e tabelas de símbolos. Com o uso dessas técnicas, conceitos básicos de semântica podem ser levados em consideração no momento da análise (KARIN, 2015).

Ferramentas que utilizam este tipo de técnica não buscam apenas casos triviais na utilização de sub-rotinas. Ao invés disso, levam em consideração todo o contexto, que envolve tanto variáveis como sub-rotinas. Assim, por exemplo, um espaço de memória declarado com um dado tamanho pode ser verificado quanto a possíveis falhas de acesso a endereços fora de suas fronteiras (FOSTER, 2005).

Ferramentas que utilizem técnicas de análise estática podem auxiliar desenvolvedores na busca por erros que possam comprometer a segurança da aplicação desenvolvida. Infelizmente, realizar a verificação completa do código desenvolvido é uma tarefa difícil e de maneira geral indecidível (CHESS, 2004) segundo o teorema Rice (HOBBY, 1965; PINKUS, 1976), podendo ser reduzida ao problema da parada de Turing (TURING, 1936). Assim, não podemos garantir, para questões não triviais como a não existência de vulnerabilidade, que a verificação abrange a totalidade dos casos.

5.2 Soluções Dinâmicas

Técnicas que utilizam soluções dinâmicas buscam proteger informações sensíveis do programa. Sendo um dos principais objetivos de um atacante, ao explorar vulnerabilidade de *buffer overflow*, controlar e corromper dados que transitam através do programa, realizar a verificação de integridade nos permite detectar ataques, ou erros, que possam levar à exploração da falha.

Para realizar a validação de integridade de informações críticas, algumas pesquisas sugerem que tenhamos metadados associados aos dados sendo protegidos. Assim, para verificar a integridade seria realizada a validação dos metadados com os próprios dados (PIROMPOSA, 2011). Podemos agrupar as técnicas que utilizam soluções dinâmicas em quatro grupos distintos: *proteção de endereço*, *proteção de entrada*, *validação de fronteiras* e *ofuscamento*. Cada grupo é diferenciado pela maneira como gerencia os metadados envolvidos.

5.2.1 Proteção de Endereço

Soluções que pertencem a este grupo pressupõem que as informações referentes a endereços são sensíveis e, quando corrompidas, podem facilmente levar à exploração da vulnerabilidade. Esse tipo de solução modifica as funções responsáveis pela criação dos endereços para que realizem a criação dos metadados. Com os metadados criados, é necessário que ocorra a verificação de sua integridade no momento em que for necessário o acesso ao endereço, assim, as funções que utilizam esses dados também são modificadas. Como parte deste grupo, podemos citar técnicas como utilização de *canário* (COWAN, 1998), *codificação de endereço* (COWAN, 2003), *cópia de endereço* (FRANTZEN, 2001).

5.2.1.1 Canário

Um compilador, ao realizar a tradução de um código para um dada arquitetura, possui conhecimento suficiente para modificar as informações que devem ser inseridas na pilha. "*Canários"* são valores especiais inseridos em diversas posições da memória a fim de detectar quando uma estrutura de controle é corrompida. Ataques de *buffer overflow*, tendem a modificar o endereço de retorno através da escrita além dos limites de um dados espaço de memória. Em virtude da maneira como são estruturados, muitos ataques acabam modificando também as áreas adjacentes ao *buffer* e ao local onde é armazenado o endereço de retorno.

Para realizar a proteção, o "*canário"* é colocado de tal maneira que, caso ocorra o estouro do buffer, este seja modificado antes da modificação do endereço de retorno. O valor do "*canário"* pode ser verificado durante o *epílogo* de uma sub-rotina, antes que o endereço de retorno seja restaurado (SILBERMAN, 2004). Caso a integridade do dado não seja comprovada, ações para impedir a possibilidade de uma exploração podem ser tomadas.

Durante os anos de aprimoramento da técnica, foram utilizados quatro tipos de "*canários"* distintos (SILBERMAN, 2004):

Random Canary: Conceito original utilizado na criação do *canário.* O *canário* é um valor pseudorrandômico de 32-bits gerado pelas funções /dev/random ou /dev/urandom no sistema operacional Linux.

Random XOR Canary: O conceito de valor randômico foi extendido para fornecer um pouco mais de proteção. Assim, além do valor gerado, uma operação XOR é realizada com o dado que deve ser protegido.

Null Canary: O *canário* recebe o valor de 0x00000000. Em linguagens como C, a maior parte de funções que lidam com entradas textuais terminam quando encontram um valor *null*. Assim, não seria possível alterar endereços de retorno se o buffer sobre o qual está ocorrendo a operação, armazenasse valores nulos.

Terminator Canary: O valor do canário é escolhido como uma combinação de *Null, CR, LF* e *0xFF.* Esses valores são considerados como fim de entradas textuais, assim, é possível proteger os dados, mesmo para funções que não terminam ao encontrar valores *null*.

Ao inserir novos elementos, que devem ser gerenciados e verificados, o sistema sofre uma redução em sua performance. Quando o gerenciamento de proteção é feito através da utilização de *canários*, dois pontos, durante o processo de execução, sofrem com o aumento de operações que devem ser realizadas. *Prólogos de funções* sofrem uma pequena perda de desempenho pela necessidade de alocar o *canário* na pilha. *Epílogos de funções* sofrem uma perda moderada de desempenho pois necessitam verificar a integridade do *canário* antes de realizar o retorno da sub-rotina.

Para programas que utilizam muitas invocações de sub-rotinas, a verificação constante dos metadados que tentam garantir a integridade dos dados, pode resultar em uma perda significativa de desempenho, podendo chegar a um aumento de até 60% (COWAN, 1998) para funções simples. Embora essa perda de desempenho torne-se menos significativa à medida que ocorre o aumento na complexidade das funções, muitos desenvolvedores optam pelo desempenho em detrimento da segurança, esquecendo que uma vulnerabilidade explorada pode causar ainda mais prejuízos.

5.2.1.2 Codificação de Endereço

Uma das principais caracteristicas de um ataque de *buffer overflow* é a capacidade de um atacante corromper a memória, de maneira controlada, a fim de realizar a modificação do endereço de retorno de uma sub-rotina para um endereço no qual a execução do código irá fornecer vantagens, acessos ou permissões indevidas ao atacante.

Métodos de prevenção que utilizam técnicas de codificação de endereço não procuram identificar quando dados sensíveis foram corrompidos. Ao invés de tentar impedir ou detectar o estouro do *buffer*, esse tipo de técnica visa dificultar a ação de um atacante, realizando outras operações sobre os dados protegidos antes que estes sejam utilizados.

Sendo o endereço de retorno de sub-rotinas alvo constante de ataques, a proteção deste torna-se muito importante. A capacidade de um atacante organizar o código necessário para realizar um ataque é apenas um dos passos necessários para o sucesso da operação. Tornar esse código disponível dentro do sistema e executá-lo com os parâmetros corretos é o que faz com que o atacante seja bem sucedido. A fim de efetuar a execução do código, o atacante busca sobrescrever o endereço de retorno de forma que, ao terminar a execução da sub-rotina, a execução continue a partir de um ponto escolhido pelo próprio atacante.

A técnica de codificação de endereço codifica o endereço de retorno através de uma função. Assim, mesmo quando o atacante obtém sucesso em sobrescrever o endereço de retorno, sua efetividade fica comprometida, na medida em que o valor armazenado será transformado antes de ser utilizado. O compilador, ao traduzir o código para a arquitetura alvo, adiciona os trechos necessários para codificação e decodificação dos valores adequados (PYO, 2002).

5.2.1.3 Cópia de Endereço

Quando um segmento de memória é corrompido, dados anteriormente armazenados nas posições sobrescritas são perdidos e, assim, é impossível para um computador saber quando um elemento de controle, como por exemplo o endereço de retorno de uma sub-rotina, foi corrompido. Técnicas como a cópia de endereço, visam identificar inconsistências com os valores originais destes elementos de controle e tomar as ações necessárias para evitar a exploração da vulnerabilidade.

O conceito utilizado quando nos referimos à cópia de endereço é a manutenção do endereço de retorno original de uma sub-rotina. A cópia desse endereço é armazenada no inicio do ciclo de execução da função chamada. Antes de desviar o controle da execução para a posição estabelecida pelo endereço presente na pilha, é realizada uma comparação deste com o valor salvo anteriormente. Caso exista consistência entre os dois dados, é suposto que não houve corrupção na memória e a execução pode seguir seu fluxo. Se, ao contrário, ocorrer a detecção de incoerência entre os dois valores, ações apropriadas podem ser tomadas para evitar a continuidade do ataque (GUPTA, 2006).

Apesar da similaridade com técnicas como o Canário, que verifica se um valor armazenado anteriormente continua sem modificação, a cópia de endereço traz uma diferença fundamental. Enquanto o Canário é armazenado na própria pilha onde o ataque é executado, devendo ser sobrescrito para a identificação de uma falha, a cópia do endereço é salva em uma estrutura auxiliar, inacessível ao atacante. A manutenção dessa estrutura auxiliar é realizada durante o prólogo, fase na qual o endereço original é armazenado no local escolhido, e o epílogo, fase onde ocorre a verificação de integridade dos valores e a posterior limpeza da área de armazenamento.

Mais de uma maneira de gerenciar essa estrutura auxiliar foram desenvolvidas (PARK, 2004; THOMAS 2013). Apesar da eficiência dessa técnica, capaz de detectar qualquer corrupção no endereço de retorno, existem alguma desvantagens que, por vezes, levam à não utilização da mesma. Há perda de desempenho, semelhante ao que acontece quando utilizamos o Canário como solução. Da mesma maneira, essa perda é constante para cada função e, à medida que funções mais complexas são executadas, a perda torna-se cada vez menos significativa. Outro fator que por vezes influencia na decisão é a perda de espaço. Como é necessário armazenar dados redundantes, além do armazenamento e sincronização corretos da pilha com a estrutura auxiliar o utilizador dessa técnica também deve se preocupar com questões de segurança relacionadas ao local de armazenamento, muitas vezes tão complexas quanto evitar o próprio ataque.

5.2.2 Proteção de Entrada

A interação entre o usuário de um sistema e o próprio sistema é parte fundamental no processo de computação. À medida que um programa necessita de informações, é solicitado ao usuário que as informe para que o processo possa continuar. Operações de entrada de dados estão presentes nos mais diversos ambientes dentro da computação, seja para efetuar o

acesso a um sistema através do fornecimento de credenciais, como senhas, ou mesmo através de dispositivos externos, como microfones, quando utilizamos programas capazes de interpretá-los.

A utilização de um programa por parte de um usuário comum não causa grandes problemas porque, apesar da ocorrência de erros na entrada de dados, esses erros não afetam a segurança do sistema. Por outro lado, quando um usuário mal intencionado utiliza um sistema vulnerável, a possibilidade de fornecer dados que serão processados torna-se o meio ideal para a execução de um ataque. Por não saber as intenções reais de um usuário e para garantir a segurança de seus dados, um programa não deve processar dados de entrada de forma indiscriminada (HOWARD, 1965). É necessário que ocorra a validação das informações à medida que essas transitam de um ambiente inseguro para um ambiente seguro.

Esse tipo de proteção não se aplica apenas a ataques de *buffer overflow*. Toda aplicação que possui alguma forma de interação com o meio externo deve tomar precauções em relação à segurança. Essa análise de entrada pode ser feita através da verificação da existência de elementos estranhos ou proibidos na entrada, como palavras chave e comandos de execução, ou caracteres inválidos. Como medida para tentar diminuir o risco, pode ser feita a troca desses elementos por outros seguros ou mesmo a invalidação completa da entrada.

Muitas vezes, esse tipo de verificação acaba sendo deixada de lado pelos desenvolvedores por exigir dedicação a um aspecto do sistema que não impacta diretamente na corretude da aplicação. Entretanto, de uma maneira geral, supor que o sistema não será alvo de usuários mal intencionados é um erro que poderá custar a dedicação de muito mais tempo para sua solução do que para a prevenção.

5.2.3 Validação de Fronteiras

A validação de fronteiras é o processo de manter o registro dos endereços de fronteira de objetos, *buffers* e *arrays* e de efetuar a verificação das operações que acessam e utilizam essas estruturas a fim de garantir que toda a manipulação de dados efetuada se restrinja a uma região válida (CHUANG, 2007).

Para implementar essa técnica, os compiladores modificam o código gerado para a utilização de uma estrutura que contenha os atributos como tamanho, posição e tempo de vida das estruturas que devem ser protegidas. Validar uma operação executada sobre essas estruturas implica na verificação dos atributos das mesmas. Caso os limites de espaço e tempo

da estrutura sejam observados, a operação é permitida. Caso contrário, um erro é gerado e as ações necessárias podem ser tomadas (AUSTIN, 1994).

Quando nos referimos a falhas geradas por corrupção da memória, em especial *buffer overflow*, a validação de fronteiras é considerada um dos métodos mais eficientes. Sua utilização, entretanto, acarreta alguma perda de desempenho, tanto em questões de velocidade quanto tamanho. A fim de aumentar sua performance, otimizações em nível de software e hardware foram sugeridas (SHAO, 2005; RUWASE, 2004).

5.2.4 Ofucasmento

Técnicas de ofuscamento têm por objetivo realizar reorganizações em pontos sensíveis seguindo algumas heurísticas para dificultar a realização dos ataques, uma vez que esses dependem fortemente no conhecimento aprofundado da maneira como dados e endereços estão organizados.

O primeiro ponto sensível que podemos analisar se refere ao código fonte dos programas. A ideia de realizar modificações em programas a fim de que sua lógica seja de difícil compreensão não é nova. Programas capazes de realizar essa tarefa são chamados de ofuscadores (COLLBERG, 1998). Dado um programa P e um chave secreta K um ofuscador tem por objetivo produzir um código semanticamente equivalente a P, porém com uma implementação diferente (PUCELLA, 2006). O objetivo de tal técnica é dificultar o aprendizado, por parte do atacante, dos detalhes de implementação do programa original.

A eficiência de do uso de ofuscadores para prevenir a compreensão dos detalhes de implementação de um dado programa mostrou-se pouco eficiente e passível de reversão (BARAK, 2001; GOLDWASSE, 2005). Outros autores sugerem a utilização de técnicas de ofuscamento em código fonte como meio de atingir diversidade entre as cópias (FORREST, 1997). Tal diversidade torna mais difícil o desenvolvimento de código para exploração de falhas, uma vez que diferentes ataques devem ser desenvolvidos para diferentes cópias do mesmo programa.

O ataque de *buffer overflow* utilizando a sobrescrita do endereço de retorno pressupõe o conhecimento da organização da memória de um computador. Com o objetivo de impedir que o atacante se beneficie deste conhecimento, comum a diversas aplicações, foi proposta a utilização de um elemento capaz de alterar os endereços nos quais os programas são armazenados a cada execução.

A randomização do espaço de memória é uma técnica que visa reforçar a segurança dos sistemas através do aumento de possíveis alvos. Ao invés de remover vulnerabilidades existentes, este método tem por objetivo dificultar a exploração dessas vulnerabilidades (WHITEHOUSE, 2007). A técnica proposta (TEAM, 2003) busca realocar a pilha e outras regiões de memória como *heap* e a biblioteca de funções compartilhadas, adicionando um deslocamento aleatório à base de cada uma dessas regiões. Porém, ela falha ao realocar de forma eficiente os segmentos de código e de dados (KIL, 2006).

Para que esta técnica seja eficiente, é necessário que todos os elementos da memória sejam randomicamente alocados, pois uma vez sabendo o local exato de uma dada estrutura, o atacante poderá deduzir o das demais (SHACHAM, 2004).

5.3 Isolamento

A maneira mais tradicional para execução de um ataque de *buffer overflow* é a injeção de código malicioso em um registro de execução e a alteração do endereço de retorno de uma sub-rotina para a posição onde se localiza o código injetado. Métodos que utilizam isolamento como medida de proteção buscam impedir que um atacante seja capaz de executar o trecho de código requerido para o ataque.

Quando um atacante obtém sucesso em sua tentativa de explorar uma falha de vulnerabilidade, o trecho de código inserido indevidamente encontra-se em um segmento de memória onde, em teoria, apenas dados, ou seja, informações que não têm o objetivo de ser executadas pelo processador, estão presentes. Essa característica nos dá a oportunidade de isolar esses segmentos de maneira que, onde somente dados devem estar presentes, fica desabilitada a execução de código.

Na tentativa de garantir maior proteção aos sistemas, tanto desenvolvedores de hardware como software (MICROSOFT-DEP) buscaram implementar essa técnica. A adição de um *bit* extra por palavra de memória em alguns dos novos processadores (DAILEY, 2004) possibilita a marcação de uma seção de memória como não executável, esses *bits* são denominados NX ou XD dependendo do processador utilizado. Assim, em toda a área da pilha, onde apenas dados devem ser armazenados fica desabilitada a execução, impedindo que o atacante possa explorar a falha executando o código injetado.

6 CONCLUSÃO E TRABALHOS FUTUROS

Ataques que tiram proveito de vulnerabilidades do tipo *buffer overflow* ainda são uma realidade no momento atual da tecnologia. A necessidade de computadores com maiores capacidades de processamento fez com que novas tecnologias fossem desenvolvidas, de maneira especial arquiteturas de 64 bits, cuja capacidade de endereçamento de memória ultrapassa em até 4 bilhões de vezes às das arquiteturas de 32 bits.

Para tentar diminuir o aumento da sobrecarga gerada pelo aumento massivo dessa capacidade, em especial em relação à estrutura de páginas, e por nenhum sistema possuir a necessidade real de endereçar até 16 exabytes, desenvolvedores de processadores optaram pela utilização de apenas 48 dos 64 bits disponíveis para endereçamento.

Essa escolha resultou na criação de endereços inválidos no espaço de endereçamento da memória e, mesmo sem intenção, estabeleceu uma dificuldade extra no momento da criação de códigos para a exploração de falhas. Toda uma base de *shellcodes* existentes deixou de realizar as tarefas para as quais foram projetados, necessitando de adaptação para o novo ambiente. Infelizmente, isso não foi o suficiente para diminuir a capacidade dos atacantes. Atuamente, já é possível encontrar uma ampla base de *shellcodes* desenvolvidos especialmente para lidar com essa nova característica. Embora pareça promissora, por estes ainda serem vitimas de menos ataques, a substituição de sistemas de 32 bits por outros de 64 bits não mostra nenhuma melhoria significativa em questões de segurança. À medida que seu uso vai ficando mais comum, a probabilidade de que novas falhas e vulnerabilidades sejam encontradas também aumenta. É apenas uma questão de tempo para que atacantes tenham a mesma facilidade em explorar esses novos sistemas.

Em relação ao métodos de prevenção e de detecção existentes, a mudança de arquitetura não alterou seu comportamento, embora alguns métodos possam possuir um desempenho melhor nesse ambiente, como o que usa a randomização do espaço de memória. Diversos artigos podem ser encontrados colocando à prova sua eficiência e, até mesmo, mostrando como ultrapassar as barreiras impostas por eles.

Além de não apresentar melhorias significativas, é possível levantar alguns questionamentos que poderão ser observados em trabalhos futuros. Um desses questionamentos vai de encontro a uma característica muito comum no mercado atual. Pela grande base de código desenvolvido, não é interessante que a compatibilidade seja quebrada durante o processo de evolução da tecnologia. Por esse motivo, os sistemas atuais oferecem suporte a um método de compatibilidade no qual a execução de programas compilados para

sistemas de 32 bits ainda é possível. Se, por conta dessa característica, ainda seria possível explorar as vulnerabilidades nesses novos sistemas através de *shellcodes* antigos é um assunto que merece atenção poderá fornecer bases para o desenvolvimento de pesquisas futuras.

A prevenção continua sendo o melhor caminho para atingirmos um patamar de segurança adequado em nossos sistemas. Ao mesmo tempo que ainda temos que conviver com código desenvolvido no passado e que não levava em consideração a possível criação de vulnerabilidades, ainda hoje deixamos de alertar os jovens desenvolvedores, colocando em risco a segurança de nossas informações em favor de um sistema com um nível de desempenho um pouco maior. Os desenvolvedores devem conhecer profundamente a organização e arquiteturas dos sistemas para os quais estão desenvolvendo e não apenas aprender sobre a utilização de ferramentas. Caso os desenvolvedores do passado tivessem levado em consideração esses aspectos, a simples tarefa de realizar, por exemplo a checagem dos limites durante um processo de cópia, teria impedido a exploração de uma diversidade de falhas.

REFERÊNCIAS

FISHER, D., FINGER, H., KRAMER, W., STANLEY, J. **Report Of Computer Virus Incident at AMES**. November, 1988.

CVE-2015-8126 - NVD – Detail., 2015. Disponível em: <https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-8126> Acesso em: 07 dez. 2015.

CVE-2015-0235 - Common Vulnerabilities and Exposures., 2015. Disponível em: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0235> Acesso em: 07 dez. 2015

GASSER, M. **Building a secure computer system,** 1988. New York, NY: Van Nostrand Reinhold Company. p.85

ALLEN, Julia H. **Tackling Software Security: An Increasing Threat** | CIO., 2008. Disponível em: <http://www.cio.com/article/2434636/enterprise-software/tackling-software-security—an-increasing-threat.html>

SPI, **Software Protection Initiative - Three Tenets**. 2009. Disponível em: <http://www.spi.dod.mil/tenets.htm>

NIST, SP. **Guide for Assessing the Security Controls in Federal Information Systems: Building Effective Security Assessment Plans,** 2008, p 9-10.

ENISA, **Glossary** , 2012. Disponível em: <https://www.enisa.europa.eu/activities/risk-management/current-risk/risk-management-inventory/glossary>

ALHAZMI, O. H., WOO, S., & MALAIYA, Y. K. (2006). **Security vulnerability categories in major software systems,** 2006 Communication, Network, and Information Security.

NVD – Home, 2005. Disponível em: <https://nvd.nist.gov/>

SECURITY FOCUS, 2002. Disponível em <http://www.securityfocus.com/.>

SEACORD, R. C., & HOUSEHOLDER, A. D. **A structured approach to classifying security vulnerabilities,** 2005 CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

MEUNIER, P. **Classes of Vulnerabilities and Attacks**, 2008 - Purdue University. Disponível em: <http://homes.cerias.purdue.edu/~pmeunier/aboutme/classes_vulnerabilities.pdf>

LOUDEN, Kenneth. **Programming languages: principles and practices**. Cengage Learning, 2011.

CENGAGEBRAIN, **Invitation to Computer Science**, 6th ed. 2012. Disponível em: <http://www.cengagebrain.com.au/content/9781133994152.pdf>

IBM – Knowledge Center. Disponível em: <http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm>

SALOMON, David, **Assemblers and Loaders,** 2004. Disponível em: <http://www.davidsalomon.name/assem.advertis/asl.pdf>

UEW, **A Brief History of Programming Languages.** 2014. Disponível em: <hfttp://www.uew.edu.gh/sites/default/files/teaching %20materials/A_BRIEF_HISTORY_OF_PROGRAMMING_LANGUAGES_Sem2_2012.pdf>

CUNNINGHAM, **High Level Language**, 2004 Disponível em: <http://c2.com/cgi/wiki? HighLevelLanguage>

AHO, Alfred V, RAVI Sethi, and JEFFREY D Ullman. **Compilers, Principles, Techniques**. Addison wesley, 1986.

CSE, **Basic Compiling Theory,** 2012. Disponível em: <http://www.cse.msu.edu/rgroups/sens/Software/Telelogic-3.5/locale/english/help/htmlhlp/comptheory.html>

NORTHWOOD, Chris, **Lexical and Syntax Analysis of Programming Languages**, 2009. Disponível em: <http://www.pling.org.uk/cs/lsa.html>

STANDFORD, Handout CS. **Semantic Analysis**. 2012. Disponível em: <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/180%20Semantic %20Analysis.pdf>

RICARTE, Ivan. **Introdução à Compilação**, 2008. Elsevier Brasil

CLEMENTS, A. **The Principles of Computer Hardware - ACM Digital Library**. 2000. http://dl.acm.org/citation.cfm?id=518138

SMITH, Richard E. **A historical overview of computer architecture**. Annals of the History of Computing 10.4, 1988. p. 277-303.

BURKS, Arthur W, Herman H Goldstine, and John Von Neumann. **Preliminary discussion of the logical design of an electronic computing instrument**. Springer Berlin Heidelberg, 1982.

BRYANT, Randal E, and David R O'hallaron. **x86-64 Machine-Level Programming,** 2005

MCCALLUM, John C. **Memory Prices 1957 to 2015**, 2015. Disponível em: <http://www.jcmit.com/memoryprice.htm>

AMD64, **AMD64 Architecture Programmer's Manual Volume 1: Application Programming** 2013, AMD.com. Disponível em: <http://support.amd.com/TechDocs/24592.pdf>

COWAN, Crispin et al. **Buffer overflows: Attacks and defenses for the vulnerability of the decade**. DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings 2000. p. 119-129.

CRABB, Michele Curmudgeon's Executive Summary. In Michele Crabb, editor, The SANS Network Security Digest. SANS, 1997.

CA, **CA-2003-16**, 2014. Disponível em: <https://www.cert.org/historical/advisories/CA-2003-16.cfm>

CVE, **CVE-2009-2550 - Common Vulnerabilities and Exposures**. 2009. Disponível em: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2550>

ONE, Aleph. **Smashing the stack for fun and profit.** Phrack magazine 7.49, 1996. p. 14-16.

UDEL, **Writing buffer overflow exploits - a tutorial for beginners**. 2015. Disponível em: <https://www.eecis.udel.edu/~bmiller/cis459/2007s/readings/buff-overflow.html>

MAD, Irish. **Writing Buffer Overflows - Mad Irish**, 2012. Disponível em: <http://www.madirish.net/142>

INSECURE. **How to write Buffer Overflows**. 2006. Disponível em: <http://insecure.org/stf/mudge_buffer_overflow_tutorial.html>

PIROMPOSE, Krerk, and Richard J Enbody. **Buffer-overflow protection: the theory**. Electro/information Technology, 2006 IEEE International Conference on 7 May. 2006: p. 454-458.

FU, Desheng, and Feiyue Shi. **Buffer Overflow Exploit and Defensive Techniques.** Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on 2 Nov. 2012: 87-90.

CMU, **Stacks and Queues**. 2015. Disponível em: <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html>

GNU, **Frame Layout - GCC – Gnu**. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc-3.1/gccint/Frame-Layout.html>

GDB, **Debugging with GDB - Frames** Disponível em: <ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_41.html>

PIROMPOSA, Krerk, and Richard J Enbody. **Arbitrary Copy: Bypassing Buffer-Overflow Protections**. Electro/information Technology, 2006 IEEE International Conference on 7 May. 2006: p. 580-584.

PIROMPOSA, Krerk, and Richard J Enbody. **Survey of Protections from Buffer-Overflow Attacks**., 2011 Engineering Journal 15.2. p 31-52.

WHEELER, D. J. **The use of sub-routines in programmes**. Proceedings of the 1952 ACM national meeting (Pittsburgh) on - ACM '52. 1952. p. 235.

POZZA, Davide et al. **Comparing lexical analysis tools for buffer overflow detection in network software.** Communication System Software and Middleware, 2006. Comsware 2006. p 1-7.

LAROCHELLE, David, and David Evans. **Statically Detecting Likely Buffer Overflow Vulnerabilities.** USENIX Security Symposium 13 Aug. 2001.

KARIM, MD Monjurul. **Source Code base Buffer Overflow Detection Technology.** 2015. Dissertação (Mestrado em Ciência da Computação e Tecnologia) - Northwestern Polytechnical University, 2015.

FOSTER , James C., Vitaly Osipov, Nish Bhalla, Niels Heinen and Dave Aitel, **Chapter 5 - Stack Overflows, In Buffer Overflow Attacks.** Syngress, Burlington. 2005. p. 161-228.

TURING, Alan Mathison. **On computable numbers, with an application to the Entscheidungsproblem**. J. of Math 58.345-363.1936. p 5.

HOBBY, Charles R, and John R Rice. **A moment problem in L1 approximation**. Proceedings of the American Mathematical Society. 1965. p 665-670.

PINKUS, Allan. **A simple proof of the Hobby-Rice theorem.** Proceedings of the American Mathematical Society 60.1. 1976. p. 82-84.

CHESS, Brian, and Gary McGraw. "**tatic analysis for security.** IEEE Security & Privacy 6 . 2004. p. 76-79.

COWAN, Crispin et al. **Pointguard TM: protecting pointers from buffer overflow vulnerabilities.** Proceedings of the 12th conference on USENIX Security Symposium 4 Aug. 2003. p 91-104.

FRANTZEN, Michael, and Michael Shuey. **StackGhost: Hardware Facilitated Stack Protection.** USENIX Security Symposium 13 Aug. 2001.

SILBERMAN, Peter, and Richard Johnson. **A comparison of buffer overflow prevention implementations and weaknesses**. IDEFENSE, August, 2004.

COWAN, Crispan et al. **StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks**. Usenix Security 26 Jan. 1998. p 63-78.

PYO, Changwoo, and Gyungho Lee. **Encoding function pointers and memory arrangement checking against buffer overflow attack**. Information and Communications Security. 2002. p 25-36.

MADAN, Bharat B, Shashi Phoha, and Kishor S Trivedi. **StackOFFence: a technique for defending against buffer overflow attacks.** Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on 4 Apr. 2005. p. 656-661.

WIKIBOOKS, **x86 Disassembly - Wikibooks, open books for an open world.** 2011. Disponível em: <https://en.wikibooks.org/wiki/X86_Disassembly>

MSDN, **__cdecl - MSDN – Microsoft.** 2015. Disponível em: <https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx>

EAGLE, Chris. **The IDA pro book: the unofficial guide to the world's most popular disassembler**. No Starch Press, 2011.

MSDN, **__stdcall - MSDN – Microsoft.** 2014. Disponível em: <https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx>

SENSEPOST. **A Crash Course in x86 Assembly for Reverse Engineers**. 2014. p19

KASPERSKY, Kris.Hacker **Disassembling Uncovered: Powerful Techniques To Safeguard Your Programming**. A-List Publishing. April, 2003. p 230-276

GCC, **x86 Function Attributes - Using the GNU Compiler Collection**. 2015. Disponível em: <https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html>

MSDN, **Overview of x64 Calling Conventions - MSDN - Microsoft**. 2015. Disponível em: <https://msdn.microsoft.com/en-us/library/ms235286.aspx>

VIRUSBTN, **Virus Bulletin : Glossary - Shellcode**. 2008. Disponível em: <https://www.virusbtn.com/resources/glossary/shellcode.xml>

MCDERMOTT, **Windows x64 Shellcode | McDermott Cybersecurity**. 2011. Disponível em: <http://mcdermottcybersecurity.com/articles/windows-x64-shellcode>

JASON, Deckard. **Buffer Overflow Attacks: Detect, Exploit, Prevent.** Syngress. 2005.p 25 – 52

SONG, Yingbo et al. "On the infeasibility of modeling polymorphic shellcode." Machine learning 81.2. 2010. p 179-205.

EXPLOITDB, **Shellcode - Exploit Database.** 2015. Disponível em: <https://www.exploit-db.com/shellcode/>

SHELLSTORM, **shell-storm | Shellcodes Database**. 2009. Disponível em: <http://shell-storm.org/shellcode/>

GUPTA, Suhas et al. **Dynamic code instrumentation to detect and recover from return address corruption.** Proceedings of the 2006 international workshop on Dynamic systems analysis 23 May. 2006. p. 65-72.

PARK, Yong-Joon, and Gyungho Lee. "**Repairing return address stack for buffer overflow protection.**" Proceedings of the 1st conference on Computing frontiers 14 Apr. 2004. p. 335-342.

THOMAS D, Jamal S, S J. **A Categorized Survey on Buffer Overflow Countermeasures** . International Journal of Advanced Research in Computer and Communication Engineering. 2013

DAILEY, L. **New chips stop buffer overflow attacks**. Computer 37.10 .2004.p. 28-28.

MICROSOFT-DEP, **A detailed description of the Data Execution Prevention (DEP)**. Disponível em: <https://support.microsoft.com/en-us/kb/875352>

COLLBERG C. S., C. Thomborson, and D. Low. **Breaking ab-stractions and unstructuring data structures**. InProc. 1998 International Conference on Computer Languages, IEEE computer Society Press, 1998. p 28-38.

PUCELLA, Riccardo, and Fred B Schneider. **Independence from obfuscation: A semantic framework for diversity.** Computer Security Foundations Workshop, 2006. 19th IEEE 30 Jan. 2006. p. 241.

BARAK, Boaz et al. **On the (im) possibility of obfuscating programs.** Advances in cryptology—CRYPTO 2001 1 Jan. 2001.p. 1-18.

GOLDWASSER, Shafi, and Yael Tauman Kalai. **On the impossibility of obfuscation with auxiliary input**. *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on* 23 Oct. 2005. p. 553-562.

FORREST, Stephanie, Anil Somayaji, and David H. Ackley. **Building diverse computer systems**. In 6th Workshop on Hot Topics in Operating Systems, Los Alamitos, CA, 1997. IEEE Computer Society Press. p. 67-72

SHANECK, Mark. **An Overview of Buffer Overflow Vulnerabilities and Internet Worms**. CSCI. 2003.

WHITEHOUSE, Ollie. **An analysis of address space layout randomization on Windows Vista.** Symantec advanced threat research. 2007. p. 1-14.

TEAM, PaX. **PaX address space layout randomization (ASLR).** Mar. 2003. p. 42.

KIL, Chongkyung et al. **Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software.** Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual Dec. 2006. p. 339-348.

SHACHAM, Hovav et al. **On the effectiveness of address-space randomization**. Proceedings of the 11th ACM conference on Computer and communications security 25 Oct. 2004. p. 298-307.

CHUANG, Weihaw et al. **Bounds checking with taint-based analysis**. High Performance Embedded Architectures and Compilers. 2007. p. 71-86.

SHAO, Zili et al. **Efficient array & pointer bound checking against buffer overflow attacks via hardware/software**. Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on 4 Apr. 2005. p. 780-785.

AUSTIN, Todd M, Scott E Breach, and Gurindar S Sohi. **Efficient detection of all pointer and array access errors**. ACM, 1994.

RUWASE, Olatunji, and Monica S Lam. **A Practical Dynamic Buffer Overflow Detector.** NDSS 4 Feb. 2004.

HOWARD, M. and Leblanc D., **Chapter 10: all input is evil!** in Writing Source Code, 2nd ed: Microsoft Press, 1965.

LHEE, Kyung-Suk, and Steve J Chapin. "Buffer overflow and format string overflow vulnerabilities." Software: Practice and Experience 33.5. 2003. p. 423-460.

CONOVER, Matt. **w00w00 on heap overflows**. Jan. 1999.

CERN, **CERN Computer Security Information**. 2010. Disponível em: <https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml>

INTEL, **Introduction to x64 Assembly | Intel® Developer Zone**. 2013. Disponível em: <https://software.intel.com/en-us/articles/introduction-to-x64-asse

RINGZER0 Team , Mr.Un1k0d3r . "64 Bits Linux Stack Based Buffer Overflow." Disponível em: <https://www.exploit-db.com/docs/33698.pdf>

EXPLOITDB-37362, **linux/x86-64 execve/bin/sh 30 bytes - Exploits Database.** 2015. Disponível em: <https://www.exploit-db.com/exploits/37362/>

EXPLOITDB-37384, **Linux x86 - execve /bin/sh 23 Bytes - Exploits Database.** 2015. Disponível em: <https://www.exploit-db.com/exploits/37384/>

ANLEY, Chris, John Heasman, Felix “FX” Linder, Gerardo Richarte. **The Shellcoder's Handbook: Discovering and Exploiting Security Holes 2nd edition**. Wiley, 2007.