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

CÉSAR MALERBA

Vulnerabilidades e Exploits: técnicas, detecção e prevenção

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, Junho de 2010

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

AGRADECIMENTOS

Não poderia, em hipótese alguma, deixar de agradecer minha família por tudo que alcancei em meus estudos. Sobretudo o apoio de meus pais. Neles sempre tive o incentivo e a estrutura necessárias para o meu desenvolvimento.

Agradeço também a todos que contribuem para o excelente funcionamento do Instituto de Informática da UFRGS. Notável pelo interesse de seus professores e funcionários. Provando que o ensino público de qualidade é alcançável e oferece a nossa sociedade enormes benefícios. Em especial ao meu orientador Raul Fernando Weber; professor de enorme qualidade que muito contribuiu pelo meu interesse na área de segurança da computação e que me auxiliou no desenvolvimento desse trabalho.

Deixo também um agradecimento especial a uma pessoa que nunca deixou de me lembrar do meu potencial e me auxiliou durante períodos difíceis. Ela que me acompanhou durante parte da minha jornada no curso: minha namorada Luciana.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASLR	Address Space Layout Randomization
EBP	Extended Base Pointer
ESP	Extended Stack Pointer
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DNS	Domain Name Service
DoS	Denial of Service
DPL	Descriptor Privelege Level
NUMA	Non-Uniform Memory Access
RAM	Random Access Memory
RPC	Remote Procedure Call
SQL	Structered Query Language
SSP	Stack-Smashing Protector
SWF	Shockwave Flash

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A importância do software em nossa sociedade aumenta a cada dia. Entretanto, ele ainda sofre com vários problemas de segurança - a maioria deles relacionados ao seu desenvolvimento. Para auxiliar os desenvolvedores no entendimento dessa matéria, esse trabalho trata de pontos chave na segurança do software: vulnerabilidades e *exploits*.

Dois temas principais relacionados a vulnerabilidades são apresentados: métodos de classificação e detecção através da técnica *fuzzing*. Ambos objetivando fornecer ao leitor conceitos básicos para o alcance de um desenvolvimento mais seguro e consciente do software. Dado o enorme valor da técnica *fuzzing* na descoberta de vulnerabilidades, ela é proposta como nova ferramenta aos desenvolvedores para que eles a apliquem antes dos atacantes.

Considerando que o conhecimento de técnicas de *exploits* é indispensável na busca por software mais seguro, esse trabalho também tenta oferecer ao leitor uma visão a partir da perspectiva dos atacantes. Para isso, uma visão geral dos *exploits* é apresentada. Após uma ideia mais ampla nos métodos dos atacantes, um deles é detalhado: o *NULL pointer exploit*.

Palavras-chave: Segurança, teste de software, exploits.

Exploits: técnicas, detecção e prevenção

ABSTRACT

Software becomes more important in our society everyday . However, it still suffers from many security problems - most of them related to its development. To help developers in understanding the subject, this paper addresses key points in software security: vulnerabilities and exploits.

Two main issues related to vulnerabilities are presented: classification methods and detection through fuzzing technique. Both aim at offering the reader basic concepts to achieve safer and more conscious software development. Given the great value of fuzzing technique in vulnerabilities discovery, it is proposed as a new tool to developers so that they can apply it before the attackers.

Considering the knowledge of exploit techniques indispensable in the pursuit of building safer software, this paper also tries to offer a view from the attackers perspective. To achieve it, an overview of exploits is presented. After giving a broad idea on the attackers methods, one of them is detailed: the NULL pointer exploit.

Keywords: security, exploits, testing.

1 INTRODUÇÃO

A Internet...foi projetada no espírito da confiança. Nem os protocolos de rede de comunicações nem o software que comanda os sistemas computacionais(nodos) conectados a rede foram arquitetados para operação num ambiente no qual estão sob ataque.(COMMITTEE, 2005). Esse trecho[1] foi extraído de um *report* criado pelo comitê consultivo sobre tecnologia da informação do presidente dos Estados Unidos. Sua intenção é chamar atenção para um problema que ameça todos os países do mundo e que, sem o devido tratamento, pode trazer graves consequências às nações.

Esse mesmo documento vai além e adverte para as consequências da insegurança: *Apesar dos recentes esforços para adicionar componentes de segurança aos sistemas computacionais, às redes e ao software, os atos hostis se propagam largamente causando danos em escala nacional e internacional.* Conforme os consultores, o elo mais fraco da cadeia é o software. *Os métodos de desenvolvimento de software...falham em prover a alta qualidade, a confiabilidade e a segurança das quais os ambientes de tecnologia da informação necessitam.*

Logo, de acordo com os estudos estratégicos de defesa da nação mais poderosa do mundo, um dos seus pontos mais vulneráveis é o software que compõe seus sistemas. E mais, o processo de desenvolvimento é considerado como fator predominante para a existência dessa ameaça. E, é claro, diante da crescente ubiquidade dos sistemas de computação e, por consequência, do software, aumenta ainda mais a responsabilidade dos projetistas e dos desenvolvedores em garantir que seu trabalho não será utilizado em prejuízo dos usuários.

Por isso é fundamental que todos aqueles envolvidos no processo de produção do software tenham o devido conhecimento das implicações relativas à segurança. Conhecer as vulnerabilidades e saber como detectá-las acabam sendo habilidades necessárias para garantir projetos seguros.

Outra forma de contribuir para atenuação dessa ameaça, é conhecer as táticas aplicadas pelos atacantes. Pois: *quanto mais se sabe sobre o que o seu inimigo é capaz de fazer, maior é a sua condição de discernir sobre quais mecanismos são necessários para sua própria defesa* - (HARRIS, 2008). O conhecimento das técnicas de ataque constitui, portanto, mais uma das dimensões que compõem uma estratégia de defesa.

O foco do presente trabalho gira em torno de dois conceitos abordados anteriormente: vulnerabilidade e *exploits*. Ambos são apresentados e sob a ótica do desenvolvedor de sistemas e do atacante num esforço para contextualizar o leitor sobre os caminhos para tornar o software mais seguro.

[1] Extratos de (COMMITTEE, 2005) foram traduzidos livremente pelo autor do presente trabalho.

1.1 Organização do trabalho

Para alcançar o objetivo proposto, esse trabalho está organizado da seguinte forma. Inicia com conceitos básicos que servem de suporte aos capítulos seguintes. São definições de termos usados e informações básicas sobre o funcionamento do software - como organização da memória dos processos, virtualização da memória, entre outros.

No terceiro capítulo, é abordado o tópico de classificação de vulnerabilidades. Nele, o leitor terá contato com propostas e projetos que auxiliam o estudo dessa matéria.

O quarto capítulo traz uma visão geral sobre técnicas de exploração de vulnerabilidades: os *exploits*. São trazidos tipos que possuem boa representatividade.

Para ilustrar com maior precisão os conceitos de vulnerabilidade e de exploit na prática, é apresentado em maior detalhes o *NULL pointer exploit*. Tópico que assume bastante relevância uma vez que uma série de falhas desse gênero foram recentemente descobertas - muitas delas no Kernel do Linux, onde permaneceram por mais de 8 anos.

O último capítulo aborda uma técnica para detecção de vulnerabilidades: *Fuzzing*. Foi escolhida entre as demais para constar nesse trabalho pois possui excelente relação custo benefício. É destacada com o intuito de mostrar aos desenvolvedores os eficientes métodos empregados pelos atacantes para encontrar problemas no software.

2 CONCEITOS INICIAIS

2.1 Vulnerabilidade/Exploit

O primeiro termo que devemos definir neste trabalho é *exploit*. Mas antes dele, trataremos de vulnerabilidade - pois eles têm uma ligação estreita. Podemos definir vulnerabilidade como uma falha em um sistema que permite a um atacante usá-lo de uma forma não prevista pelo projetista (ANLEY, 2007). Ou seja, uma vulnerabilidade implica a possibilidade de uso indevido de um sistema. Os passos necessários para explorar essa fraqueza, ou mesmo o código (programa) que pode tirar proveito da vulnerabilidade é descrito como *exploit*. Um *exploit* surge apenas quando há uma vulnerabilidade - mas podem existir vulnerabilidades para as quais não exista *exploit*.

De maneira simplificada, podemos dizer que vulnerabilidades surgem devido a 3 causas básicas:

- Erros de implementação
- Falhas de *design*
- Erros de configuração ou de infra-estrutura do sistema

Na sequência desse trabalho, serão aprofundados melhor esses dois conceitos. Também serão examinadas com mais detalhe as causas dos problemas no software.

2.2 Pressupostos básicos

Neste trabalho iremos tratar principalmente de *exploits* na arquitetura x86 de 32 bits. Trata-se da arquitetura de computadores pessoais mais difundida nos dias de hoje. Mas boa parte do estudo realizado pode ser aplicada a praticamente qualquer outra arquitetura.

Dessa forma, principalmente quando tratarmos de erros de corrupção de memória, lembramos que as discussões serão feitas com a arquitetura x86 de 32 bits em mente. Sendo devidamente sinalizado quando forem examinados aspectos de outros casos.

2.3 Memória Virtual

Para facilitar o entendimento de questões discutidas nesse trabalho, é importante esclarecer pontos básicos sobre uso do esquema de memória virtual na arquitetura x86. Isso nos permitirá entender melhor as alternativas existentes para a implementações do sistemas bem como as relações com as vulnerabilidades e *exploits*. Primeiro repassamos as motivações e o funcionamento básico da memória virtual. Entre suas vantagens, conforme (BOVET, 2005), encontramos:

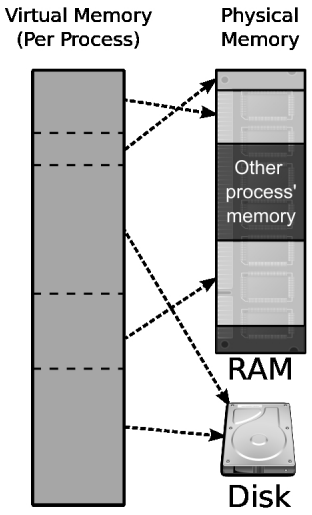

Figura 2.1: Esquema de memória virtual. Fonte: Wikepedia - Junho 2010.

- É possível executar aplicações que necessitam mais memória que a disponível fisicamente
- Processos podem compartilhar uma única imagem na memória de uma biblioteca
- As aplicações podem ser realocadas na memória física em qualquer lugar
- Um processo pode ser executado com apenas parte de seu código carregado em memória física
- Cada processo tem a disposição seu próprio espaço de endereçamento como se fosse a única aplicação no sistema

De forma geral, podemos que dizer que há um melhor uso dos recursos disponíveis e a abstração criada pela virtualização pode diminuir a preocupação dos programadores sobre como o processo é organizado na memória física.

A figura 2.1 apresenta uma visão global do funcionamento da memória virtual. Nela, é mostrado como um processo pode possuir regiões diferentes de memória mapeadas arbitrariamente para memória física; nesse caso composta de memória RAM e de um disco de armazenamento.

2.3.1 Como é usada na arquitetura x86

Na arquitetura x86, a virtualização da memória é implementada com segmentação e paginação. Na segmentação, a memória é dividida em segmentos de tamanho variável; sendo o endereço virtual um identificador de segmento e um *offset.* No caso da paginação, a divisão da memória é feita em páginas de tamanho fixo e, analogamente, o endereço virtual é composto de um identificador para a página e um *offset.* Os dois modelos não são excludentes bem como podem ser usados independentemente. Nesse caso, foi optado pela presença de ambos. Assim, é possível dividir a memória em segmentos que por sua vez podem ser particionados em diferentes páginas.

Figura 2.2: Processamento de endereço na arquitetura x86. Fonte: (BOVET, 2005)

Chamamos de endereço lógico aquele acessado pela aplicação - que seria, portanto o nível mais alto na virtualização. Já o endereço linear, é resultado do processamento da segmentação quando é fornecido um endereço lógico. Para produzir o endereço físico, ainda é necessário que a unidade de paginação processe o endereço linear. A figura 2.2 ilustra esse processo.

Assim, máquinas x86 possuem 2 níveis, segmentação e paginação na ordem, a serem tratados para que um endereço físico possa ser encontrado a partir de um endereço lógico. Cada um dos processos possui suas particularidades e implementa suas próprias proteções conforme passamos mais detalhes a seguir.

2.3.2 Segmentação

Para a segmentação, a memória é organizada em segmentos e cada um deles possui diferentes atributos. Os principais segmentos são: o de código(mantido no registrador cs), o de dados(no registrador ds) e o da pilha(no registrador ss). Para os fins desse trabalho, cabe destacar os seguintes atributos de segmentos:

Base Endereço linear inicial do segmento.

Limite Endereço linear final do segmento.

Tipo Os tipos definem as propriedades de acesso como leitura, escrita e execução.

DPL *Descriptor Privelege Level.* Define o privilégio mínimo da CPU para que ele possa ser acessado. O máximo privilégio é identificado pelo valor 0; o menor em 3. Normalmente o sistema operacional se reserva o DPL 0, enquanto os demais processos ficam com 3.

Naturalmente, podem existir segmentos distintos para o sistema operacional e para as aplicações do usuário. Assim, aqueles pertencentes ao sistema ficam com DPL em zero, enquanto os demais ficam marcados como 3.

2.3.3 Paginação

Separando a memória em páginas de um tamanho fixo (normalmente 4Kb), a paginação opera criando um mapeamento entre aquelas existentes na memória física e aquelas do espaço de endereçamento das aplicações. Assim, endereços contíguos dentro do endereço linear são mapeados para endereços contínuos dentro da página física.

Através das tabelas de páginas, é realizado esse mapeamento. São estruturas de dados mantidas pelo sistema operacional com o suporte do hardware. Elas mantém os dados referentes às páginas. Devemos destacar as seguintes propriedades:

Leitura/Escrita Indica se a página possui permissão de leitra e/ou escrita.

User/Supervisor flag De forma análoga ao DPL na segmentação, define o privilégio mínimo exigido para o acesso.

Segmento	Base	Limite	Tipo	DPL
Código do usuário	0x00000000	0xfffff	Leitura	3
Dados do usuário	0x00000000	0xfffff	Escrita/Leitura	3
Código do kernel	0x00000000	0xfffff	Leitura	0
Dados do kernel	0x00000000	0xfffff	Escrita/Leitura	0

Tabela 2.1: Quatro segmentos principais no Linux com seus descritores

Ao ocorrer algum problema no acesso a alguma página, como falta de privilégios ou mesmo porque uma página não está mapeada fisicamente, o sistema operacional é chamado através de uma interrupção de hardware. Dessa forma, ele pode tomar as medidas necessárias como: cancelar a aplicação que realizou um acesso ilegítimo ou buscar fazer o mapeamento da página que falta à aplicação. Com a cooperação do hardware e do sistema, fica possível implantar a memória virtual com paginação.

2.3.4 Caso abordado: memória virtual no sistema Linux x86

Conforme veremos na seção 5.3.2, a escolha no uso da memória virtual no sistema Linux trouxe impactos sobre sua segurança. Nesse sistema, o uso da segmentação é extremamente limitado. A paginação foi escolhida em detrimento à segmentação - sendo a última usada apenas por obrigatoriedade da arquitetura. Os segmentos de dados e código para o sistema operacional e para as aplicações do usuário são praticamente os mesmos. A tabela 2.1 apresenta os principais segmentos.

Embora isso tenha vantagens em termos de desempenho, pois facilita a troca de contexto entre modo usuário e modo do sistema bem como a troca de dados entre ambos, há uma desvantagem de segurança. Tanto o kernel as aplicações do usuário podem usar os mesmos endereços lógicos. Certas restrições de segurança acabam recaindo, portanto, apenas para a paginação.

2.4 Gerência de memória

O controle da memória é um ponto crítico. Falhas nele acabam resultando em vulnerabilidades gravíssimas. Faremos uma breve abordagem sobre o gerenciamento de memória sobre o ponto de vista dos *exploits*.

Um primeiro ponto a destacar sobre a memória é um princípio básico que norteia quase todas as arquiteturas modernas. Dados e instruções não são diferenciados na memória. Ou seja, não há uma separação rígida entre instruções que compõem um programa e os dados sobre os quais opera. Essa característica foi herdada da arquitetura básica de von Neumann. Como veremos a seguir, essa decisão de design, com origem nos anos 1940, embora tenha facilitado a evolução dos computadores, abriu caminhos para os *exploits* que conhecemos hoje.

Abaixo descrevemos o layout básico da memória de um processo em um sistema UNIX. Ele pode ser separado em 6 partes fundamentais:

text A parte que contém as instruções do programa - seu código propriamente dito. Seu tamanho é fixo durante a execução e ela não deve possibilitar escrita.

data Contém variáveis globais já inicializadas. Seu tamanho é fixo durante a execução.

bss Nome de origem história significando Block Started by Symbol. Área da memória

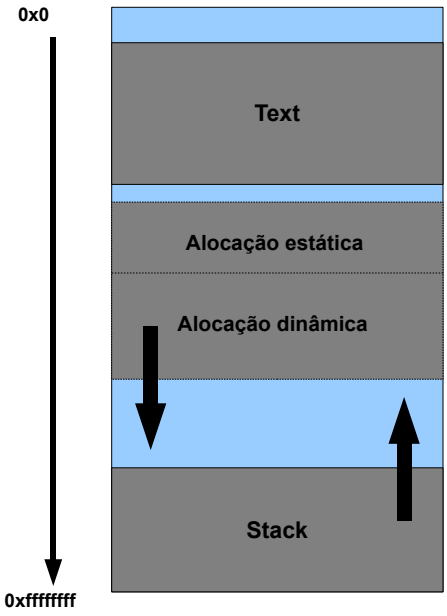

Figura 2.3: Regiões de memória em um processo.

responsável por armazenar variáveis globais não inicializadas. Como text e data, bss também tem tamanho fixo conhecido desde o início do processo.

heap Espaço para variáveis alocadas dinamicamente. A chamada de sistema sbrk é responsável pelo controle do crescimento/encolhimento desse espaço. Bibliotecas geralmente facilitam a vida do programador disponibilizando interfaces mais amigáveis como malloc() e free(). Assim a biblioteca se encarrega de chamar sbrk() para diminuir/aumentar o heap. Ela cresce do endereço mais baixo para o mais alto.

stack Mantém controle das chamadas de funções. Possibilita a recursividade. Logo, possui tamanho variável - crescendo do endereço mais alto para o mais baixo (sendo antagonista do heap - ver figura 2.3). Esse crescimento é que torna possível que uma chamada de função que tenha seus dados sobrescritos influencie numa chamada de função anterior. Esse característica é explorada pelo *Buffer Overflow* - tratado no capítulo 4.

enviroment A última porção de memória do processo guarda uma cópia das variáveis de ambiente do sistema. Essa seção possui permissão de escrita, mas como bss, data e text, possui tamanho fixo.

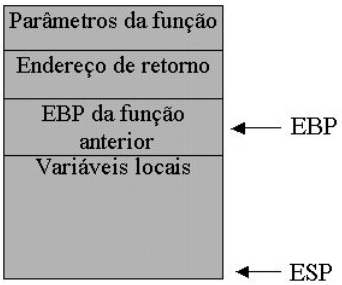

Figura 2.4: Organização do *frame* na pilha. Fonte: (FURLAN, 2005) pg. 17.

2.5 Funcionamento mais detalhado do Stack

A pilha é uma região contínua com base fixa e tamanho variável. Na arquitetura abordada por esse trabalho, x86 (bem como em muitas outras), a pilha cresce em direção ao endereço mais baixo. É organizada em *frames* que são os blocos alocados quando ocorrem chamadas a funções. Cada *frame* contém(ver figura 2.4):

- parâmetros
- variáveis locais
- endereço de retorno da função anterior
- endereço do *frame* da função que a chamou

2.5.1 Chamada de funções

Quando uma função é chamada, seus parâmetros são empilhadas e posteriormente o endereço do retorno. Isso fica a encargo da função que faz a chamada. Para completar o *frame*, aquela que é chamada, empilha o endereço do frame da função chamadora (EBP) e posteriormente aloca na pilha o espaço correspondente a suas variáveis locais. É importante ressaltar que, caso o endereço de retorno, empilhado por quem chama, seja alterado, o fluxo de execução é mudado. Pois é justamente este o princípio do *Buffer Overflow*. Ele será abordado em maior detalhes na Seção 4.2.1.

2.6 Funcionamento mais detalhado do heap

A porção de memória correspondente ao heap possibilita ao programador alocar dinamicamente memória que fica disponível durante toda a execução para qualquer chamada de função. Existem diversas formas de administrar a memória do heap, mas o mais encontrado, conforme (LOVE, 2007), é dividir o todo em partições de potências de 2. Chamadas à função malloc()/free(), em última análise, correspondem a operações de alocar/liberar partições internas do heap. Normalmente a organização das partições se dá como forma de uma lista encadeada. Assim, para efetuar o controle dos blocos, são mantidas meta-informações que determinam tamanho, endereço do próximo bloco livre - entre outros - para que a gerência da memória dinâmica seja eficiente. Uma chamada a free(), por exemplo, pode implicar acerto de diversos ponteiros que existem dentro das partições no heap.

Havendo uma validação incorreta no software, pode ocorrer um *overflow* na área do heap. Se os dados escritos sobrepuserem os valores dos ponteiros de controle interno dos blocos, fica aberto um caminho para que um atacante consiga uma escrita em um endereço arbitrário. Pois é esse o princípio básico de funcionamento de *Heap Overflow*. Ele pressupõe o conhecimento aprofundado da gerência do heap; pois só dessa forma é possível prever exatamente como os blocos são mantidos.

2.7 Mapemamento de memória anônimo

Como visto no funcionamento básico do heap, a fragmentação é um problema a ser considerado. Uma forma alternativa de alocar memória dinamicamente que não usa o heap são os mapeamentos anônimos. No Linux por exemplo, através da função mmap() é possível alocar um bloco de memória contínuo fora do heap que não está sujeito aos problemas de fragmentação. Conforme, (LOVE, 2007), é possível considerar esse espaço de memória como um novo heap vindo de apenas uma alocação. No exemplo a seguir, é requisitado uma porção de memória de 64 bytes iniciando em NULL.

Listing 2.1: Mapeamento de memória anônimo.

```
1    mmap(NULL, 64, PROT_READ | PROT_WRITE,
2            MAP_FIXED | MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
```

Na técnica *NULL pointer exploit*, conforme veremos mais adiante, essa possibilidade de alocar um bloco com o início pré-determinado é muito útil. No caso da chamada de mmap() do exemplo anterior, estamos obtendo um bloco de 64 bytes começando no endereço zero(NULL). Isso nos permite colocar código executável nessa porção da memória e, na presença de uma vulnerabilidade, desviar a execução para esse ponto.

2.8 Registradores de controle

Uma parte fundamental da arquitetura que deve ser mencionada são os registradores que possuem relação direta com o gerenciamento da memória. Talvez o mais importante (na arquitetura base do estudo IA32) seja o EIP(Extended Instruction Pointer). Ele indica o endereço da próxima instrução. Sobrescrevê-lo equivale obter o controle do fluxo de um processo. Além dele, destacamos EBP(Extended Base Pointer) e ESP(Extended Stack Pointer). ESP indica o endereço do último valor inserido na pilha. O EBP indica o início da pilha para aquela chamada de função. É usado para referenciar variáveis locais da função.

2.9 *Shellcode*

Outra parte fundamental de muitos *exploits* é o chamado *shellcode*. Podemos definí-lo, segundo (ANLEY, 2007), como um conjunto de instruções que são injetados e executados por um programa através de um *exploit*. Normalmente, é escrito em linguagem *assembly* por manipular diretamente os registradores e depois transformado em *opcodes* em hexadecimal.

A palavra *shell* contida em seu nome tem origem no fato de, normalmente, ele ser usado para abrir um *shell* na máquina atacada. Sendo aberto com permissões de *root*, o atacante assume controle absoluto do sistema. Ainda que isso seja o mais comum, o *shellcode* não se restringe a isso. Como qualquer outro programa, ele, em muitos casos,

só é limitado pela imaginação do seu construtor. Sua criação envolve certas dificuldades, como a ausência de bytes em zero; já que normalmente, ele é armazenado na memória como uma string na linguagem C - que termina com um zero.

Atualmente, existem diversos tipos de *shellcode* disponíveis para os mais variados sistemas e arquiteturas. Dificilmente um atacante terá a necessidade de produzir seu próprio dada a abundância de alternativas prontas. Alguns, por exemplo, possuem até estratégias para enganar sistemas de detecção de invasão. Para as necessidades desse trabalho, não iremos nos aprofundar nesse tema, mas no capítulo 2 de (ANLEY, 2007), há informações muito úteis para um aprendizado de *shellcode*.

3 CLASSIFICAÇÃO DE VULNERABILIDADES

A classificação de vulnerabilidades representa enorme desafio. Nos dias de hoje, não existe nenhum padrão aceito globalmente para essa tarefa. Ainda assim, já houve vários avanços na área. Existem padrões para enumerar e catalogar vulnerabilidades, bem como propostas que podem criar bases para uma classificação que venha a ser aceita pela comunidade. Métricas, relativas à gravidade e ao impacto, também estão disponíveis e são empregadas no auxílio às instituições nas tomadas de decisões.

No trabalho de Seacord e Householder, (SEACORD, 2005), temos os fatores que motivam a busca pela organização das vulnerabilidades em classes:

- O entendimento das ameaças que elas representam;
- Correlacionamento de incidentes, de *exploits* e de artefatos;
- Avaliação da efetividade das ações de defesa;
- Descoberta de tendências de vulnerabilidades;

Vemos, portanto, que a taxonomia[1] das vulnerabilidades pode trazer uma série de benefícios para seu entendimento, tratamento e prevenção. Nesse capítulo, nosso intuito é abordar a dificuldade nesse processo e apresentar os avanços já obtidos nesse sentido.

3.1 A dificuldade em classificar; estágio já alcançado: enumeração

Antes de entrarmos no mérito das vulnerabilidades, é preciso definir com precisão dois termos que utilizaremos por todo o capítulo: classificar e enumerar. Como veremos, a taxonomia é mais custosa que a enumeração.

3.1.1 Classificar

Como podemos encontrar em (HOLANDA FERREIRA, 1975), classificar implica "distribuir em classes e/ou grupos segundo um sistema". Logo, para a classificação, é preciso haver uma metodologia que possa separar os itens em estudo em diferentes grupos. A ciência que estuda esse processo é chamada taxonomia. Ela é guiada, conforme (GRéGIO, 2005a), pelos princípios taxonômicos. São eles:

Exclusão mútua Um item não podem ser categorizado simultaneamente em dois grupos.

Exaustividade Os grupos, unidos, incluem todas as possibilidades.

[1] Ciência da classificação.

Repetibilidade Diferentes pessoas extraindo a mesma característica do objeto devem concordar com o valor observado.

Aceitabilidade Os critérios devem ser lógicos e intuitivos para serem aceitos pela comunidade.

Utilidade A classificação pode ser utilizada na obtenção de conhecimento na área de pesquisa.

Vemos que os critérios para a taxonomia são exigentes e pressupõem uma metodologia cuidadosamente gerada para atendê-los.

3.1.2 Enumerar

A enumeração é um processo semelhante a "indicar por números; relacionar metodicamente"; como encontramos em (HOLANDA FERREIRA, 1975). Trata-se, portanto, de algo muito mais simples que a classificação. Mesmo sendo mais simples, é extremamente importante pois permite que os itens enumerados sejam facilmente apontados e diferenciados entre si.

Sem um procedimento de enumeração dos objetos de estudo, adotado de comum acordo, não é possível que duas partes se comuniquem sem risco de cometerem enganos. Quem garante que estão tratando exatamente da mesma coisa naquele momento? Logo a enumeração é essencial para o devido entendimento sobre os objetos de estudo.

3.1.3 Da enumeração à classificação

No trabalho de Mann, (MANN, 1999), há um excelente paralelo entre a questão abordada nesse capítulo e o advento da tabela periódica[2] na Química. A organização dos elementos da forma como conhecemos hoje na tabela periódica foi um processo longo que culminou com as ideias de Dimitri Mendeleev. Outros químicos que o precederam foram responsáveis pela identificação e listagem dos elementos. Isso possibilitou um melhor estudo e uma maior troca de informação precisa entre os pesquisadores.

Segundo Mann, a tabela periódica só pode ser efetivamente criada graças aos esforços daqueles que enumeraram os elementos de forma mais simples antes de Mendeleev. O trabalho deles permitiu a interoperabilidade necessária para o surgimento da tabela periódica. Da mesma forma, nos anos antecedentes a 2000, a comunidade que estudava e acompanhava as vulnerabilidades estava num patamar semelhante àqueles que precederam Mendeleev. Ou seja, sequer havia uma enumeração mais amplamente aceita e reconhecida das vulnerabilidades que permitisse avanços suficientes para uma taxonomia.

Citamos o ano de 2000 como parâmetro, pois nessa época, 1999, surgiria um projeto que se tornaria referência para a criação de uma padronização da enumeração de vulnerabilidades. Não seria ainda um evento comparável à criação da tabela periódica para Química (pois não trouxe a taxonomia) mas certamente lançaria as bases para a interoperabilidade exigida para estudos mais aprofundados na área. Estamos falando da criação do CVE(Common Vulnerabilities and Exposures)[3][4] pelo MITRE. A seção 3.2 traz mais detalhes.

[2]Dispõe sistematicamente os elementos de acordo com suas propriedades permitindo uma análise multidimensional.

[3]http://cve.mitre.org

[4]Na época de sua criação era originalmente conhecido por Common Vulnerabilities Enumeration - vide (MEUNIER, 2006) pg. 9.

Organização	Como se referia à vulnerabilidade
CERT	CA-96.06.cgi_example_code
Cisco Systems	http - cgi-phf
DARPA	0x00000025 = http PHF attack
IBM ERS	ERS-SVA-E01-1996:002.1
Security Focus	#629 - phf Remote Command Execution Vulnerability

Tabela 3.1: Uma vulnerabilidade: diversos nomes e nenhum entendimento

Podemos dizer, portanto, que atualmente, embora não tenhamos uma taxonomia amplamente aceita pela comunidade, já foi atingido o estágio de enumeração. Projetos como o CVE podem ser considerados como marcos dessa etapa. A seguir, iremos abordar em mais detalhes o surgimento e o funcionamento dele. Isso nos possibilitará compreender melhor a complexidade da classificação das vulnerabilidades bem como irá facilitar o entendimento dos capítulos seguintes que abordam *exploits*.

3.2 CVE

3.2.1 Surgimento e objetivos

Para deixar mais nítida a dificuldade de interoperabilidade das organizações no que se refere a ameaças de segurança na época que antecede o CVE, temos a tabela 3.1, extraída de (MARTIN, 2001). Ela mostra como diferentes organizações se referiam à mesma vulnerabilidade em 1998. Trata-se de um verdadeira Torre de Babel.

O CVE, como dito anteriormente, surge em 1999 e seu maior objetivo, como podemos ler em sua FAQ, (CVE, 2010), é tornar mais fácil o compartilhamento de informações sobre vulnerabilidades utilizando uma enumeração comum. Essa enumeração é realizada através da manutenção de uma lista na qual, conforme encontramos em (SANTOS BRANDãO, 2004), valem os seguintes princípios:

- Atribuição de um nome padronizado e único a cada vulnerabilidade.
- Independência das diferentes perspectivas em que a vulnerabilidade ocorre.
- Abertura total voltada ao compartilhamento pleno das informações.

Segundo a própria organização, vide (CVE, 2010), o CVE não possui um objetivo inicial de conter alguma espécie de taxonomia. Essa é considerada uma área de pesquisa ainda em desenvolvimento. É esperado que, com o auxílio prestado pela catalogação das vulnerabilidades já constitua um importante passo para que isso ocorra.

Na figura 3.1, podemos visualizar um histórico da quantidade de vulnerabilidades adicionadas. Nos últimos anos podemos perceber que os incidentes registrados ficam na média de 7000. Isso mostra relevância que o projeto do CVE alcançou.

3.2.2 Funcionamento

O CVE é formado por uma junta de especialistas em segurança dos meios acadêmico, comercial e governamental. Eles são responsáveis por analisar e definir o que será feito dos reports passados pela comunidade - se eles devem ou não se integrar àqueles já pertencentes à lista. Cabe a eles definir nome, descrição e referências para cada nova ameaça.

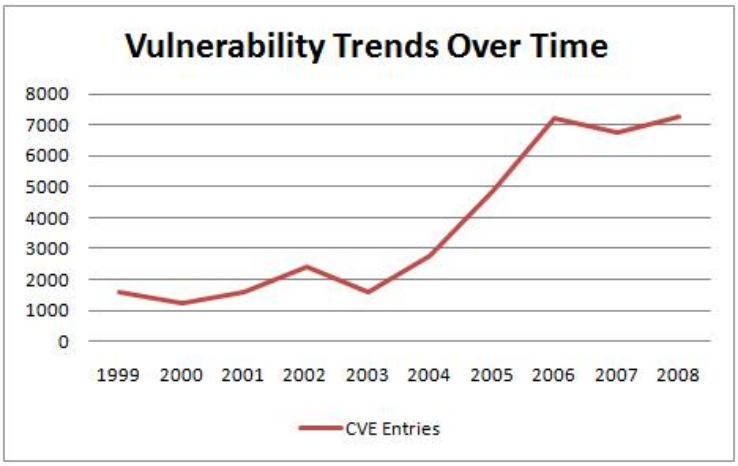

Figura 3.1: Vulnerabilidades registradas no CVE a cada ano entre 1999 e 2008. Fonte: (FLORIAN, 2009)

Esse processo inicia quando uma vulnerabilidade é reportada. Ela assume um CAN (Candidate Number), número de candidata. Até que ela seja adicionada à lista, ele permanece com um CAN que a identificará. Apenas após o devido estudo e aprovação do caso pela junta responsável, é que ela assume um identificador CVE.

Os identificadores CVE são definidos conforme o padrão: CVE-2010-0021. Onde, separados por '-', há 3 partes. A primeira é fixa: CVE. A segunda refere-se ao ano de surgimento; enquanto a terceira indica o número sequencial daquela vulnerabilidade entre todas aquelas que foram adicionadas naquele ano. Logo, no exemplo fornecido, essa seria a vigésima primeira de 2010.

Uma vez integrada, a vulnerabilidade passa a estar publicamente disponível. Essa abertura pode servir de auxílio aos atacantes - pois informações sobre possíveis furos de segurança são sempre bem vindas a eles. Porém, conforme podemos verificar na FAQ do CVE, (CVE, 2010), há uma série de motivos pelos quais a disponibilidade desses dados supera o risco oferecido pela exposição. São eles:

- O CVE está restrito a publicar vulnerabilidades já conhecidas.
- Por diversas razões, a comunidade de segurança de informação sofre mais para compartilhar dados sobre as ameças que os atacantes.
- É muito mais custo a uma organização proteger toda sua rede contra as ameças que a um atacante descobrir e explorar uma delas para comprometer alguma das redes.

3.3 Propostas taxonômicas

Nessa Seção, apresentaremos taxonomias para vulnerabilidades e um projeto, análogo ao CVE, que reúne esforços para a padronização da classificação: o CWE. Primeiramente, faremos um breve histórico das propostas já criadas com esse fim. A seguir, apresentaremos algumas das classificações que consideramos de maior relevância e, por fim, trataremos do projeto CWE - que assume importante papel no contexto atual na catalogação dos tipos de vulnerabilidades existentes.

3.3.1 Histórico das propostas

Em (GRéGIO, 2005b), encontramos um levantamento das dessas alternativas. Mas, como veremos, nenhuma delas atinge os objetivos de uma taxonomia plena - apresentados na seção 3.1.1. Assim, buscaremos discutir as ideias para as metodologias de classificação com o intuito de apontar suas vantagens e fraquezas.

Em 1976, surge o primeiro estudo, chamado *Research Into Secure Operating Systems*(RISOS). Ele objetivava auxiliar na compreensão das falhas encontradas nos sistemas operacionais como MUTICS, GECOS, IBM OS. Foram propostas 7 classes, pág. 328 de (GRéGIO, 2005b):

- Validação incompleta de parâmetros;
- Validação inconsistente de parâmetros;
- Compartilhamento implícito de privilégios ou dados confidenciais;
- Validação assíncrona ou serialização inadequada;
- Autorização, autenticação ou identificação inadequadas;
- Violação de proibição de limite;
- Erro de lógica explorável;

Esse estudo teve importância pelo pioneirismo, mas se limitava a problemas de sistemas operacionais bem como não atendia a todos os princípios taxonômicos.

Dois anos após o projeto RISOS, em 1978, seria criado o *Protection Analysis*(PA). Seu objetivo principal era permitir que qualquer pessoa, mesmo sem conhecimento específico sobre falhas de segurança, utilizando um padrão dirigido, pudesse encontrar vulnerabilidades nos sistemas - (TSIPENYUK, 2005), pág. 2. O PA, separava as falhas em 4 grandes classes - (GRéGIO, 2005b), pág. 329:

- Reforço e inicialização do domínio da proteção;
- Validação de operandos / dependências no gerenciamento das filas;
- Sincronização imprópria;
- Erros de seleção de operadores críticos;

Embora a ideia inicial do PA também incluísse a detecção automática de vulnerabilidades, sendo pioneiro nesse ponto, a classificação proposta não era intuitiva e de difícil aplicação - conforme consta em (TSIPENYUK, 2005). Logo, a aplicação prática não foi levada adiante, mas a base da proposta adicionou conhecimento na área.

Segundo (GRéGIO, 2005b), apenas no ano de 1992, teríamos uma nova proposta de classificação que trouxesse nova perspectiva ao estudo em questão. Trata-se do trabalho de Landwher: *A Taxonomy of Computer Security Flaws.* Seu foco estava no auxílio aos projetistas no desenvolvimento mais seguro do software. Sua classificação tinha por base 3 critérios:

- Como o defeito entra no sistema(gênese);
- Quando o defeito entrou no sistema(tempo de introdução);

- Onde ele se manifesta(localização);

De acordo com Grégio, seu principal problema era a ambiguidade no processo de classificação. A dependência na visão do classificador tem do sistema impede a objetividade necessária a uma boa taxonomia. Outro problema nessa proposição, abordado por Katrina, em (TSIPENYUK, 2005), está na dificuldade que pode surgir caso, por exemplo, se desconheça a forma como a vulnerabilidade adentrou o sistema. Em tal situação, não seria possível identificar a gênese.

No ano de 1996, a taxonomia proposta por Aslam, em *Use of a Taxonomy of Security Faults*, traria nova acréscimo às pesquisas na área. Segundo, (TSIPENYUK, 2005)(pág. 3), o esquema proposto por Aslam é bastante preciso; consistindo numa série de perguntas para cada categoria de vulnerabilidade. Em (GRéGIO, 2005b), pág. 329, encontramos as classes criadas por Aslam, com suas subdivisões:

1. Falhas de codificação;

   Erros de sincronização;

   Erros na validação de condição;

2. Falhas emergentes;

   Erros de configuração;

   Falhas no ambiente;

Embora seja uma taxonomia precisa, conforme ressaltado anteriormente, ela sofre por estar focada excessivamente em sistemais UNIX - como indicado em (TSIPENYUK, 2005).

3.3.2 Taxonomias e classificações mais recentes

Agora trataremos das propostas para classificação de vulnerabilidades que surgiram mais recentemente (após 2005) e que merecem uma análise mais apurada. São eles:

- *Preliminary List of Vulnerability Examples for Researchers*(PLOVER);
- *Comprehesive, Lightweight Application Security Process*(CLASP);
- *Seven Pernicious Kingdoms*;

São taxonomias que não passam pelo rigor científico, pois não atendem a todos os princípios taxonômicos, mas que ainda assim acrescentam bastante sobre o entendimento dos problemas que as vulnerabilidades representam. É o que diz Meunier em (MEUNIER, 2006) ao tratar das classificações populares.

O PLOVER, criado em 2005 pelo MITRE em colaboração com o DHS(*US. Departement of Homeland Security*)[5] e o o NIST(*National Institute of Technology*) é um esquema de classificação que possui dezenas de categorias principais e, naturalmente, possui ainda mais precisão do que a proposição de Aslam. Um de seus principais idealizadores foi Steve Christey. Trata-se de um trabalho com sólidas fundações, pois apresenta um *Framework* conceitual que permite discutir as vulnerabilidades em diversos níveis. Nele são definidos uma série de conceitos essenciais para o estudo da área. Pode ser encontrado em detalhes em (CHRISTEY, 2006).

[5] http://www.dhs.gov/index.shtm

Dentre as suas contribuições, destacam-se o caráter prático; mais de 1400 vulnerabilidades identificadas no CVE foram devidamente classificadas utilizando esse sistema. Foi uma taxonomia construída de baixa para cima(*bottom-up*). Essa experiência foi muito útil para a definição dos critérios.

Abaixo, seguem algumas categorias de mais alto nível existentes no PLOVER (existem 30 no total):

BUFF Contém erros como *Buffer Overflow* e *Heap Overflow*.

SPECTS(*Technology-Specific Special Elements*) Abrange erros que possibilitam ataques de Injeção de SQL e XSS.

RACE Erros advindos de condições de corrida.

CRYPTO Falhas relacionadas o uso inadequado ou problemas na criptografia.

Conforme será abordado a seguir, o PLOVER serviu de base para a criação do projeto CWE.

Do trabalho de John Viega e outros colaboradores, temos o CLASP. É um conjunto de atividades que busca melhorar a segurança dos sistemas. Embora trate também da classificação das falhas, ele vai muito além. Possui uma formalização de boas práticas para a construção de software seguro através de ciclos de desenvolvimento estruturados, repetíveis e mensuráveis - (SECURE SOFTWARE, 2006).

No que se refere à classificação, sua contribuição tem origem no trabalho proposto por Landwher(que utiliza os critérios de gênese, tempo de introdução e localização). O CLASP adiciona outro eixo classificatório: a consequência. As classes mais básicas, tipo do problema, são:

- Erros de *range* e de tipo;
- Problemas no ambiente;
- Erros de sincronização e de temporização;
- Erros de protocolo;
- Erros de lógica;
- *Malware*;

Para exemplificar, consideremos uma falha que permita um *Buffer overflow*. Segundo o CLASP, trata-se de um erro de tipo e de *range* - já que é permitida a escrita além do permitido no *buffer*. A injeção de SQL também cai na mesma categoria, pois os dados passados pelo usuário são utilizados incorretamente; permitindo que assumam um tipo inesperado. Já um erro no qual é ignorado o valor de retorno de uma função é considerado como erro de lógica - como, por exemplo, uma chamada à função malloc que não avalia se a alocação de memória foi bem sucedida.

O *Seven Pernicious Kingdoms*, de autoria de Katrina Tsipenyuk et alem, conforme (MCGRAW, 2006), capítulo 12, é uma taxonomia que, mesmo sendo imperfeita possibilita um bom entendimento por parte dos desenvolvedores; auxiliando na prevenção de problemas. É estruturada em dois conjuntos emprestados da Biologia: Filo e Reino. O Reino é a classificação mais ampla - enquanto o Filo é uma subdivisão do Reino. Possui

8 reinos; procurando respeitar a famosa regra de George Miller do "sete mais ou menos dois"[6]. São eles:

Erro de validação e de representação Resultam da confiança indevida nos dados de entrada. Caso do *Buffer Overflow*, injeção de SQL, XSS.

Abuso de API Sendo a API um contrato entre quem chama a rotina e aquela que é chamada, uma quebra das regras pode resultar em problemas de segurança. Quando quem chama uma função assume certas condições que não estão garantidas pela rotina chamada, temos um caso de abuso de API.

***Features* de segurança** Trata do uso correto de peças chave na garantia da segurança do software como: criptografia, autenticação, gerenciamento de privilégios, entre outros.

Tempo e estado Relativo a problemas advindos do paralelismo. Como erros na sincronização que expõem o sistema.

Gerenciamento de erros Vulnerabilidades desse reino surgem quando os erros não tratados corretamente. Quando um erro expõe informações do sistema ao atacante desnecessariamente, já estamos diante de um exemplo.

Qualidade de código Se a qualidade é baixa, o comportamento é imprevisível. Tendo em vista essa regra, problemas na codificação acabam levando a erros que permitem a subversão do sistema.

Encapsulamento Falhas relacionadas ao não estabelecimento de limites entre os componentes do sistema.

Ambiente Trata de problemas surgidos com questões externas ao software. Não estão relacionados ao código, mas influenciam diretamente na segurança.

Como subdivisões dos reinos, encontramos, por exemplo, o filo correspondente ao *Buffer Overflow* - no reino do Erro de Validação e de Representação. Ainda nele, também encontramos o filo de Injeção de Comandos. Já no reino de *Features de segurança*, está o filo da Randomização Insegura - que trata da randomização incorreta que pode ser uma fraqueza para a criptografia. O erro correspondente a NULL *pointer* pertence ao filo *Null Dereference* - que por sua vez é englobado pelo reino Qualidade de código.

Essa classificação foi desenvolvida com a projeção da adição de novos filos conforme a necessidade. O intuito foi de criar reinos amplos o suficiente para que os devidos filos fossem incorporados com o tempo.

3.3.3 O projeto CWE

Após o surgimento do CVE, uma padronização para identificação das vulnerabilidades foi sendo alcançada. Entretanto, a classificação, que não era objetivo direto do CVE, foi deixada de lado pelo projeto. Em 2005, mais de 5 anos depois da criação do CVE, após o estudo de uma série de vulnerabilidades catalogadas, foi gerado o PLOVER - abordado na Seção anterior.

A partir do estudo e da classificação resultante do PLOVER, surgiu a possibilidade de se estabelecer descrições comuns a comunidade para os tipos de vulnerabilidades. Desse

[6] Artigo *The Magic Number Seven, Plus or Minus Two* de George Miller.

esforço, surge o CWE: *Common Weakness Enumeration*[7]. Embora esteja fundamentada na classificação proposta pelo PLOVER, o CWE não se limita a ela. Também opera com outras como: *Seven Pernicious Kingdoms* e o CLASP.

Conforme encontramos em (CWE, 2007), ele é uma resposta à necessidade das instituições e das organizações em utilizarem os mesmos termos e taxonomias no tratamento dos problemas de segurança que enfrentam. Como saber quais as classes de vulnerabilidades que uma ferramenta de detecção é capaz de encontrar? Perguntas como essas caem justamente no escopo do CWE.

Entre os objetivos e impactos diretos do CWE, encontrados em (CWE, 2009), temos:

- Providencia uma linguagem comum para as discussões relativas às fraquezas encontradas no software e nos sistemas;
- Permite aos fabricantes de ferramentas de segurança fazer afirmações claras e consistentes sobre quais tipos de falhas elas cobrem;
- Permite aos compradores de ferramentas de segurança comparar com melhor qualidade as alternativas em virtude da discriminação da cobertura delas encontradas no CWE.
- Habilita governos, instituições e a indústria a utilizar a padronização fornecida pelo CWE para estabelecer contratos, termos e condições.

Mesmo tendo sido criado recentemente, tendo menos de 5 anos completos, o projeto do CWE certamente já está trazendo contribuições para a padronização na área de classificação de vulnerabilidades. A esperança é que ele se fortalece e possibilite uma referência de grande valor assim como foi estabelecido com o CVE.

3.4 Métricas para vulnerabilidades: CVSS

Comparar objetivamente vulnerabilidades de acordo com sua criticidade é algo muito útil para as organizações. Isso possibilita que os gestores mensurem o grau de urgência com que devem ser tratadas as ameaças. Podemos considerar esse procedimento como um tipo rudimentar de classificação. Ainda que não seja uma taxonomia, assume um papel de destaque por permitir um padrão para distinguir vulnerabilidades mais graves das demais.

3.4.1 Surgimento do CVSS

Para essa finalidade existe uma alternativa relativamente recente, o CVSS (Common Vulnerability Scoring System), criado em 2005. Trata-se de um *framework* aberto para atribuição de escore a vulnerabilidades. Ele oferece as seguintes vantagens, encontradas em (MELL, 2007) - pg. 3:

Padronização de escore de vulnerabilidades Quando uma organização normaliza os escores de vulnerabilidades em todas suas plataformas de hardware e software, ela pode instituir uma política comum de gerenciamento das ameaças.

***Framework* aberto** A abertura permite que os usuários tenham livre acesso para compreenderem as razões das vulnerabilidades assumirem esse ou aquele escore.

[7] http://cwe.mitre.org/

Priorização de riscos Quando o escore ambiental é calculado, a vulnerabilidade passa a possuir contexto. De tal forma que o risco real que ela representa para a organização possa ser mensurado.

A organização responsável pelo CVSS é a *Forum of Incident Response and Security Teams* (FIRST)[8]. Além do FIRST, as seguintes organizações também cooperaram para seu surgimento:

- CERT/CC
- Cisco
- DHS/MITRE
- eBay
- IBM Internet Security Systems
- Microsoft
- Qualys
- Symantec

Sua primeira versão data de 2005. Desde 2007, já se encontra na segunda versão; tratada em (MELL, 2007). Nesse trabalho, abordaremos apenas a versão atual do CVSS.

3.4.2 As métricas usadas

Para o cálculo do escore de uma vulnerabilidade, o CVSS, na sua versão 2, possui diversas métricas que são divididas em 3 grupos principais. em (MELL, 2007)[9]:

Métricas básicas Representam as características fundamentais da vulnerabilidade e são constantes com relação ao tempo e ao ambiente.

Métricas temporais Mudam com o transcorrer do tempo, mas não são suscetíveis a fatores ambientais.

Métricas ambientais Estão relacionadas unicamente ao ambiente em que a vulnerabilidade é analisada. Por isso, são absolutamente dependentes das particularidades de cada caso.

Na tabela 3.2, são mostradas as métricas usadas subdivididas nos seus respectivos grupos.

A seguir, faremos breve explicação de cada uma das métricas dos três grupos - vide tabela 3.2. Para o grupo **básico**, existem seis critérios. São eles:

Vetor de acesso Diz respeito ao nível de acesso necessário para explorar a vulnerabilidade. Pode assumir três valores: **Local**(exige acesso físico ou uma conta *shell*), **Rede adjacente**(é preciso ter acesso à rede local) ou **Rede**(indica a chamada vulnerabilidade *remota* - pode ser disparada de qualquer ponto da Internet).

[8] www.first.org

[9] Termos em inglês traduzidos livremente pelo autor.

CVSS		
Métricas básicas	**Métricas Temporais**	**Métricas Ambientais**
Vetor de acesso	Facilidade de exploração	Dano colateral potencial
Complexidade de acesso	Confiabilidade no *report*	Abundância de alvos
Necessidade de autenticação	Nível de remediação	Importância da confidencialidade
Impacto na confidencialidade		Importância da integridade
Impacto na integridade		Importância da disponibilidade
Impacto na disponibilidade		

Tabela 3.2: Métricas CVSS por grupo

Complexidade de acesso Indica a complexidade a ser enfrentada pelo atacante para que ele, uma vez que tenha obtido acesso ao sistema alvo, possa explorar a vulnerabilidade. Assume um dos valores **alto, médio ou baixo**. A complexidade é considerada alta, por exemplo, se o ataque exige alguma técnica de engenharia social mais sofisticada ou se existe uma condição de corrida com janela muito exígua que deve ser vencida.

Necessidade de autenticação Mede a quantidade de vezes que o atacante é obrigado a se autenticar durante o ataque - mesmo que seja usada a mesma credencial. É um dos valores: **nenhuma, uma, várias**.

Impacto na confidencialidade Mede o impacto causado na abertura de dados confidenciais gerados pelo ataque. Se nenhuma informação, em princípio protegida, é comprometida a medida assume valor **nenhum**. Havendo acesso a alguma informação, é considerado **parcial**. É dito **completo** caso o atacante tenha total acesso de leitura aos dados confidenciais.

Impacto na integridade Avalia a possibilidade que o atacante possui de alterar os dados quando o ataque é bem sucedido. Se não é mais possível confiar na integridade dos dados após o ataque, pois qualquer arquivo pode ter sido modificado, é considerada **completa**. Não havendo possibilidade de alteração, assume o valor **nenhuma**. É denominada **parcial** quando apenas parte dos dados pode ter sido comprometidos.

Impacto na disponibilidade Indica o quanto a disponibilidade do sistema pode ser afetada pelo ataque. É dita **completa** caso o sistema possa ser totalmente desligado ou inutilizado pelo atacante. Assume o valor **nenhuma** quando não pode haver alteração na disponibilidade e **parcial** se o serviço ainda puder estar disponível mas não plenamente.

As métricas do grupo **temporal** são opcionais. Ou seja, podem ser desconsideradas no cálculo do escore conforme a vontade nos analistas. Por isso, cada uma delas pode assumir o valor **não definido** indicando que ela não deve participar do escore. São 3 os critérios que são suscetíveis a alterações com o passar do tempo:

Facilidade de exploração Mede o estado atual das técnicas e do código disponível para exploração da vulnerabilidade. Seus valores são, em ordem crescente de facilidade: **não comprovada**, **prova de conceito**, **funcional** e **alta**. O primeiro indica que um *exploit* é meramente teórico e não há código disponível que comprove como explorar a falha. Havendo código facilmente acessível de *exploit* ou mesmo se operações manuais são suficientes, estamos diante de alta facilidade de exploração.

Confiabilidade no *report* Mede o grau de confiança na existência da vulnerabilidade bem como a credibilidade dos detalhes técnicos fornecidos quando ela foi reportada. Seus possíveis valores são: **não confirmada**(quando há apenas um rumor de uma origem sem credibilidade), **não corroborada**(há fontes não oficiais com possíveis incoerência em seus *reports*) e **confirmada**(o autor ou o fabricante admitem o problema ou ele já é amplamente conhecido existindo até *exploits* facilmente encontrados).

Nível de remediação Determina o quão longe se está de uma medida definitiva para estancamento da vulnerabilidade. Logo que o problema surge, assume o valor **indisponível**. Se houver alguma forma, não oficial, de mitigar a vulnerabilidade, é dito que a remediação está no estágio de ***workaround***. Se existe alguma medida oficial, mas ainda não definitiva, seu valor é **conserto temporário**. O nível é máximo, portanto assumindo o escore mínimo, **conserto definitivo**, caso exista uma remediação de caráter oficial definitiva.

Os fatores relativos à influência do ambiente, são medidos na **métricas ambientais**. Cada organização pode sofrer diferentemente o impacto de uma vulnerabilidade dada a heterogeneidade com que podem se organizar em termos do software e hardware utilizados para desempenhar suas funções. Exemplificando, caso uma empresa preste algum tipo de serviço de *backup* de dados, a integridade e a confidencialidade da informação que ela mantém possuem importância máxima. Em contrapartida, se a atividade desempenhada pela empresa estiver relacionada à hospedagem de projetos de código fonte aberto, a disponibilidade assume muito maior importância que a confidencialidade. Da mesma forma como os critérios temporais, eles podem assumir o valor não definido; indicando que ele não é utilizado no cálculo do escore final. Abaixo, são explicados os 5 critérios que compõem a métrica temporal:

Dano colateral potencial Mede o potencial do estrago que a vulnerabilidade pode causar à organização. Podem ser danos patrimoniais, pessoais ou relativos a ganhos financeiros. Assume os valores(do menor para o maior dano potencial): **nenhum**, **baixo**, **baixo-médio**, **médio-alto** e **alto**.

Abundância de alvos Mensura a proporção dos possíveis alvos sobre o contingente de sistemas da organização. Pode ser **nenhuma**, **baixa**(1 a 25%), **média**(26 a 75%) e **alta**(76 a 100%) .

Importância da confidencialidade Indica a relevância da confidencialidade dos dados mantidos pela empresa. Assume os valores **baixo**, **médio** e **alto**.

Importância da integridade Análogo à importância da confidencialidade.

Importância da disponibilidade Análogo à importância da confidencialidade.

3.4.3 Cálculo do escore

Tendo sido apresentadas as métricas, faremos breve explicação do funcionamento do cálculo do escore, que varia de 0 a 10, para uma vulnerabilidade. A figura 3.2 apresenta uma visão geral desse processo. Passos necessários:

1. Para cada um dos critérios descritos na Seção 3.4.2, atribuir um valor válido.

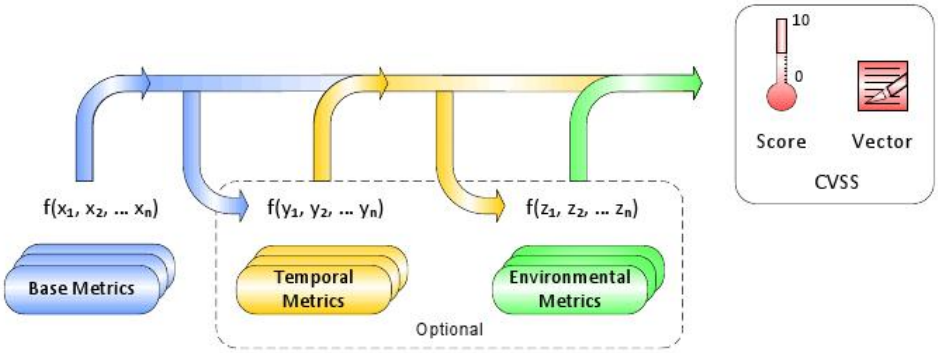

Figura 3.2: Aplicação das métricas e equações do CVSS. Fonte: (MELL, 2007)

2. Consultar as tabelas, no apêndice A, para definir um valor numérico a partir do valor nominal escolhido no passo anterior.

3. Fazer o cálculo do escore básico usando a equação A.1. Para isso, é necessário resolver antes as equações A.3 e A.2 antes.

4. Fazer o cálculo do escore temporal usando sua respectiva equação - A.4 - e o escore básico. Passo opcional. É possível manter apenas o escore básico como o final.

5. Calcular o do escore final usando a equação ambiental, A.5, a partir do escore temporal. Também é opcional; pois os critérios ambientais podem ser desconsiderados.

Ao final do cálculo, a vulnerabilidade recebe um escore de 0 a 10. Sendo 10 o valor da mais crítica possível. É importante destacar, que a atribuição dos valores, feita no passo 1, deve ser realizada por especialistas na área seguindo um critérios padronizados.

4 *EXPLOITS*

No presente capítulo, será feita uma breve análise sobre *exploits*. É importante salientar que, apenas conhecendo as técnicas usadas pelos atacantes torna-se possível criar defesas efetivas contra elas. Portanto, o estudo dessa matéria não constitui, de forma alguma, uma apologia ao ataque. Essa questão é muito bem abordada na parte I de (HARRIS, 2008); deixando claro que o conhecimento é uma arma importantíssima para aqueles que buscam uma melhoria na segurança do software.

Como ponto de partida, será aprofundado o conceito de *exploit*. De forma a mostrar sua amplitude e sua intrínseca relação com as vulnerabilidades. A seguir, serão explicadas algumas técnicas que são representativas para uma visão ampla do assunto. Na sequência, serão abordados princípios básicos de programação que visam prevenir a aplicação de *exploits* no software - combatem as falhas na origem e pontos de apoio usados pelas técnicas dos atacantes. Por fim, serão apresentadas algumas das proteções já existentes para barrar os ataques; em certos casos, também serão mostradas as formas de escape que os atacantes já desenvolveram como reação.

Como não será detalhada nenhuma técnica em particular nesse capítulo, para se obter um exemplo mais aprofundado de *exploit*, o *NULL pointer exploit* será o tema do capítulo seguinte. Assim, após um acompanhamento mais amplo do tema, será possível compreender melhor um caso específico.

4.1 Definição

Conforme foi tratado na Seção 2.1, o *exploit* é um conjunto de passos, muitas vezes materializado em um programa, capaz de tirar proveito de uma vulnerabilidade. Para muitos, entretanto, *exploit* é sinônimo de um código em C escrito por um *hacker* que tem o potencial de atacar um sistema. Essa visão, todavia, é muito limitada. Assim como existem diversos tipos de vulnerabilidades, há muitos meios de tirar vantagem delas. Por vezes, basta conhecer uma série de passos, como cliques na interface da aplicação alvo, para para explorar uma falha.

Em (HOGLUND, 2004), encontramos a seguinte lista de possíveis consequências para um *exploit* bem sucedido:

- Parada parcial ou completa do sistema(DoS);
- Exposição de dados confidenciais;
- Escalada de privilégios;
- Execução de código injetado pelo atacante;

Logo, ao explorar uma vulnerabilidade, podem ser gerados impactos na integridade, na confidencialidade ou na disponibilidade de um sistema.

De modo geral, o grande objetivo de um atacante é conseguir executar código arbitrário em seu alvo. Isso, porém, nem sempre é possível. Cada vulnerabilidade, conforme analisado no capítulo anterior, determina um universo de possibilidades para um *exploit* que a ataque.

Uma interessante forma de entender os *exploits*, sob a ótica de sua operação, está na separação deles em ***control-data*** e ***non-control-data***. Conforme (CHEN, 2005), ataques do tipo *control-data* são aqueles que alteram dados de controle do programa alvo (como endereço de retorno da função ou ponteiros) para executar código injetado ou desviar para outras bibliotecas. Os do tipo *non-control-data*, em contraponto, são aqueles que não alteram nenhum dado de controle do programa e não desviam seu fluxo de execução, mas conseguem alguma vantagem para o atacante - como autenticação ilegítima, elevação de privilégio, leitura de dados confidenciais, etc. Ao apresentar os tipos de *exploits*, será feito uso desse critério de classificação. Normalmente, os ataques que alteram estruturas de controle são aqueles que possibilitam execução arbitrária de código, enquanto os demais usam o próprio código da aplicação, explorando alguma falha de lógica ou de verificação.

4.2 Tipos

Nessa Seção, será feita uma breve explicação sobre alguns tipos de *exploits* existentes. Isso para que o leitor possa ter uma noção geral sobre as técnicas usadas pelos atacantes para explorar as vulnerabilidades no software. Não é, de forma alguma, uma lista exaustiva, mas contém muitos exemplos significativos.

Abaixo, lista dos tipos abordados:

- *Buffer overflow*;
- *Heap overflow*;
- Injeção de SQL;
- XSS(*Cross Site Scripting*);

4.2.1 *Buffer Overflow*

Um dos tipos mais bem conhecidos e um dos mais explorados. Tem um impacto enorme pois possibilita ao atacante a execução de código arbitrário no sistema atacado. O famoso artigo *Smashing the Stack for Fun and Profit* de 1996, por Aleph One, foi o primeiro a tratar em detalhes dessa técnica. Mas conforme, (ANLEY, 2007), essa estratégia já vinha sendo aplicada com sucesso por mais de 20 anos antes da publicação do artigo de Aleph One.

Ocorre quando a aplicação guarda dados a serem lidos dos usuário(ou de qualquer fonte externa) em um *buffer* alocado na pilha sem verificar se o que foi fornecido está dentro do limite aceitável(tamanho do *buffer*. Isso acaba resultando na grave falha que será apresentada abaixo.

Conforme explicado na Seção 2.4, no *stack frame* existem valores que controlam o fluxo de execução de uma aplicação. Dentre eles, está o valor de retorno de uma rotina. Qualquer chamada de função coloca na pilha o endereço para o qual ela deve retornar após seu fim. Se esse valor for alterado, é possível mudar o fluxo da aplicação - fazendo com que ele seja desviado para outro ponto qualquer.

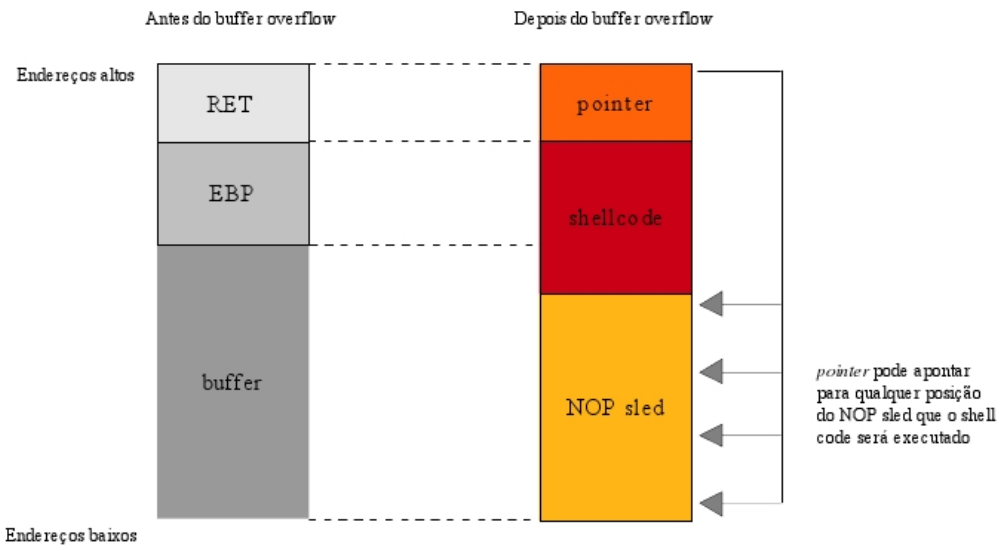

Figura 4.1: Esquema da pilha no *buffer overflow*. Fonte: (MARTINS, 2009).

Essa técnica tira proveito desse fato. Caso a aplicação possua alguma falha que permita que o usuário forneça dados maiores que o espaço alocado na pilha para armazená-los, o excedente acaba sobrescrevendo o endereço de retorno da função.

É o exemplo claro de ataque *control-data*. A mudança em um dado de controle do *stack frame* permite a colocação de um endereço forjado pelo atacante para mudar o fluxo de execução do programa atacado.

Na versão "clássica"desse *exploit*, o atacante fornece código executável, *shellcode*, que vai além do *buffer* criado para armazená-lo. No final dos dados enviados, também é inserido o endereço de início do *buffer*, que agora contém o código do atacante, para substituir no *stack frame* o valor de retorno da função. Assim, no retorno o fluxo é desviado para o *buffer* com o *shellcode*. A figura 4.1 mostra uma visão simplificada da pilha antes e depois do ataque.

A execução bem sucedida desse ataque depende de uma série de acertos. Um deles é a descoberta do endereço do *shellcode* inserido no *buffer*. Isso porque a execução deve ser desviada para lá; por isso esse valor deve substituir o endereço de retorno da função. Esses e outros desafios são pontos cruciais para a técnica. Como nessa seção desejamos fornecer apenas um visão geral, aconselhamos a busca de boas abordagens para o assunto em (ANLEY, 2007) e (FURLAN, 2005).

4.2.2 *Heap Overflow*

Semelhante ao *buffer overflow* quanto à falha que o provoca. Diferencia-se, entretanto, pelo fato do *buffer* a sofrer o *overflow* estar no heap e não na pilha. Pode ser de muito mais complexa execução que muitos outras técnicas - isso porque, conforme veremos, não possui o caráter mais genérico que seu equivalente para a pilha.

Está diretamente relacionado à implementação feita para manejar o heap no sistema afetado. Isso geralmente é atribuição da biblioteca C. Logo, além da vulnerabilidade de *overflow*, deve estar presente uma versão de biblioteca C que faça alguma gerência incorreta do heap para que um ataque desse tipo seja possível.

Assim, um *exploit* de *heap overflow* é totalmente focado em uma determinada versão de biblioteca C de um sistema, pois normalmente as aplicações não fazem sua própria gerência do heap. Na Seção 2.6, há uma explicação do funcionamento do heap que auxilia na compreensão desse tipo de ataque.

Para ilustrarmos melhor esse tema, iremos focar em um sistema específico para mostrar a sistemática e a potencialidade de um *heap overflow*. Será a implementação de gerência do heap do Linux originalmente escrito por Doug Lee. A tarefa de controle da memória dinâmica é extremamente complexa e desafiadora, pois está condicionada a otimização temporal e espacial. Como muitas aplicações fazem uso intensivo de chamadas a malloc, free e mmap - todas para controle do heap - é preciso um enorme cuidado para que os recursos de CPU e memória sejam bem utilizados de forma a não prejudicar o desempenho da aplicação e do sistema como um todo.

Para manter controle do heap, nos blocos alocados e fornecidos às aplicações, são também postos dados de manutenção. São meta informações que visam auxiliar na administração dos blocos de memória. Assim, por exemplo, ao alocarmos uma porção de memória utilizando malloc, escondido no bloco, teremos dados que a biblioteca mantém. Para a referida versão da biblioteca C do Linux, havia uma falha na qual, uma vez que os metadados dos blocos fossem alterados (via *overflow*) de uma determinada forma, o atacante poderia conseguir uma escrita arbitrária e um endereço arbitrário. Conforme já abordado anteriormente, uma falha dessa magnitude implica a possibilidade de alteração do fluxo da aplicação caso seja sobrescrita alguma estrutura de dados de controle. Uma alternativa seria um ponteiro para um função - pois uma vez sobrescrito, bastaria que ele passasse a apontar para o código injetado pelo atacante.

Como a intenção desse capítulo é fornecer uma visão geral, não será detalhada a construção do ataque. É possível, porém, descrever de forma o contexto de atuação do atacante. Sendo possível o *overflow* no heap, através da construção de um bloco de memória cuidadosamente montado[1], o atacante pode inserir dados que irão explorar uma falha na gerência dos blocos. Quando ocorrer uma chamada à função free() na aplicação, será possível obter uma escrita em endereço arbitrário - graças a atualização incorreta da lista encadeada que mantém os blocos. Isso porque o atacante, através do *overflow*, terá alterado maliciosamente as meta informações de controle do heap. Para uma visão mais completa, é aconselhada a leitura de (ANLEY, 2007) capítulo 5.

4.2.3 Injeção de SQL

Diferentemente dos *exploits* anteriores, não se trata de um erro de corrupção de memória. Serve como boa forma de contraponto para mostrar que um sistema pode ter sua confidencialidade e integridade afetados de outra forma. Ocorre na camada de banco de dados de uma aplicação em virtude de uma filtragem inadequada dos dados usados para gerar *queries* SQL.

Ainda que seja uma classe muito diferente, quando comparado aos 2 tipos descritos anteriormente, cabe destacar que seria identificado como *non-control-data*. Não é necessária nenhuma alteração no fluxo do programa explorado.

Sua potencialidade é enorme. Como implica a possibilidade do atacante injetar *queries* no banco de dados do sistema alvo, significa dizer que ele terá todos os privilégios de acesso que a aplicação possuir. Pode ser possível expor informações sigilosas, alterá-las ou mesmo destruir toda a base de dados.

[1] Alocado via malloc()

A técnica desse *exploit*, portanto, consiste em utilizar os comandos SQL previstos na aplicação para executar ações de interesse do atacante - expondo ou alterando dados de forma não prevista. Assim, basta que o atacante possua bons conhecimentos da linguagem SQL, para que ele possa alterar a semântica das *queries* e obter vantagens.

4.2.4 *XSS*(*Cross Site Scripting*)

Um dos ataques mais difundidos na web. Conforme (DHANJANI, 2009), é o meio mais comum de ataques a clientes web - constituindo poderosa arma contra a rede interna das corporações. Trata-se de um ataque voltado para o lado do cliente - diferentemente daqueles expostos anteriormente - que buscam explorar o servidor.

Seu funcionamento básico se dá através da injeção de código malicioso por atacantes em páginas web. Esse código acaba sendo executado por clientes sem seu conhecimento. Isso possibilita aos atacantes obter acesso a dados restritos mantidos pelos clientes nos *browsers*. Uma das possíveis implicações é o roubo de sessões web - tornando o atacante capaz de acessar o servidor, indistintamente, com os mesmos privilégios do usuário legítimo.

É um problema semelhante à injeção de SQL - já que a validação imprópria(ou mesmo inexistente) permite que código malicioso seja processado pelo servidor e posto no conteúdo de suas páginas para ser entregue a outros usuários. Isso confirma, novamente, a premente necessidade de validação de todo e qualquer dado de entrada em uma aplicação.

Uma das técnicas mais utilizadas para roubo de sessões, descrita em (DHANJANI, 2009), é a injeção de código Javascript no servidor para repassar ao atacante todos os dados da sessão do cliente que acesse a página. Para isso, o atacante mantém um servidor que é acionado toda vez que um cliente processa o script que ele injetou no servidor vulnerável. Esse script executado no cliente, vítima, fornece ao servidor do atacante toda informação necessária para que seja possível assumir a identidade dela.

Outro possível ataque, também apresentado em (DHANJANI, 2009) é o roubo de senhas armazenadas nos *browsers* dos clientes. Isso ocorrer quando alguma vítima utiliza o recurso de armazenamento de senhas. Muito embora isso constitua uma comodidade, uma vez que o servidor esteja vulnerável a XSS, os atacantes podem, através de script forjado para fingir um login injetado no servidor, recuperar as senhas armazenadas *browser*.

Para aprofundamento nas técnicas de XSS e para maior conhecimento nas formas de prevenção, é aconselhável a leitura do capítulo 2 de (DHANJANI, 2009). Há riqueza de exemplos e derivações do XSS que constituem nova geração dessa forma de ataque.

4.3 Prevenção de ataques

Para prevenir as indesejáveis consequências dos *exploits* apresentados anteriormente, mas não se restringindo a eles, serão discutidos princípios básicos para o desenvolvimento do software. São meios de trazer maiores garantias contra os ataques na origem. Será demonstrado que a validação dos dados usados pelas aplicações, bem como o uso de ferramentas de análise de código e de testes são exigências que não podem ser desconsideradas.

4.3.1 Validação de dados de entrada

Um dos pontos primordiais para a defesa contra os ataques é a validação dos dados de entrada. Sendo esse procedimento capaz de deter uma série de ameaças. Uma aplicação

que não verifique devidamente os dados que lhe são fornecidos é séria candidata a ser explorada. Não é possível confiar em nada que advém de qualquer ponto externo ao sistema. Conforme visto anteriormente, ataques como o de *buffer overflow* ou de *heap overflow* estão diretamente ligados a uma validação incorreta(ou mesmo ausente) de dados de entrada. O mesmo ocorrendo para injeção de SQL ou XSS.

Para que essa prática seja bem aplicada, é essencial que sejam levantados todos os vetores de entrada de uma aplicação. Por vezes, alguns deles podem ser esquecidos. No ambiente UNIX, por exemplo, variáveis de ambiente também devem ser consideradas dados de entrada. Entretanto, nem sempre são devidamente validadas. Nesse aspecto, toda uma preocupação com a entrada do sistema pode ser perdida se restar apenas um ponto não verificado. Por isso a exigência de uma avaliação dos pontos que devem ser protegidos.

Para ilustrar ainda melhor, podemos tomar como exemplo um sistema que faça uso de DNS reverso[2]. Se, para um dado IP, não for validado o nome retornado pelo DNS reverso, um atacante pode, uma vez que tenha comprometido parte da rede, forçar a aplicação a utilizar dados impróprios. Se a aplicação do exemplo usar diretamente o resultado, ela corre sérios riscos de sofrer algum tipo de exploração - como um *buffer overflow*.

É, fundamental, portanto, que os pontos de entrada sejam identificados e sejam definidas formas de validação. Em (SECURE SOFTWARE, 2006), anexo B, há detalhes sobre esse tópico - definindo diretivas para a validação.

4.3.2 Ferramentas de análise estática e auditoria de código

Uma das melhores formas de prevenção a ataques é auditar o código. A busca por falhas não precisa ser um procedimento manual; há uma série de ferramentas, algumas delas sofisticadas e focadas nessa tarefa, que podem facilitar muito a vida dos desenvolvedores. Nessa Seção, iremos abordar essa estratégia na busca por problemas que possam ser eliminados já na fase de desenvolvimento - procurando deixar o mínimo possível de brechas para os atacantes.

Conforme (TAKANEM, 2008), a auditoria de código cai na categoria de teste estrutural caixa-branca. Isso porque parte do código fonte para desempenhar sua tarefa. O mesmo autor também destaca que esse processo, assim como os testes fuzzing(vide capítulo 6), não é *capaz de comprovadamente encontrar todos os bugs ou erros possíveis*. Ainda assim, ele recomenda fortemente seu uso em complementação a outras técnicas de testes(como o fuzzing ou outros tipos de teste caixa-preta).

Muito embora o uso de ferramentas estáticas não possa substituir um auditor experiente, conforme ressalta (ANLEY, 2007), elas podem servir de base para a tarefa. Em sua maioria, elas possuem uma base de dados de padrões de código perigoso. É o caso do uso da função strcpy para a cópia de strings. Uma linha de que contenha esse tipo de chamada será encontrada e reportada como problema a ser tratado - dado o risco que ela representa. Como exemplos de ferramentas para auxílio na busca por falhas no software, temos:

Splint Faz análise de falhas de código C. Segundo (ANLEY, 2007), é capaz de realizar algumas verificações bem complexas.

RATS Busca por falhas já bem conhecidas em linguagens com C, C++, Perl e Python. Não possui a mesma profundidade nas verificações que Splint, mas é uma ótima forma de garantir a ausência de problemas já superados.

[2]Processo de descoberta do nome associado a um dado IP.

Flawfinder Semelhante a RATS. Ambas surgem simultaneamente e cobrem uma mesma gama de falhas em suas verificações.

Como forma de prevenção aos ataques, principalmente em se tratando de projetos construídos em linguagens como C e C++, o uso dos tipos de ferramentas descritas acima constitui quase uma obrigação. Seu uso é simples e pode fornecer o ponto de partida para uma auditoria manual do código - que, naturalmente, também é fortemente aconselhável.

4.4 Proteções e contra-proteções

Existem diversas proteções para impedir um *exploit*. São recursos dos compiladores, das bibliotecas, do hardware e dos sistemas operacionais que servem de contra ponto às mais variadas técnicas que os atacantes já criaram. Seu principal objetivo é resguardar os sistemas mesmo que os desenvolvedores não tenham seguido as recomendações de segurança. De forma que, mesmo na presença de uma vulnerabilidade, um ataque não seja possível ou seus efeitos sejam minimizados ao máximo.

Conforme é possível encontrar em (ANLEY, 2007), destacamos os seguintes mecanismos de proteção:

1. Pilha não executável;
2. W ^ X(permissão de escrita ou de execução - nunca ambas);
3. Canário para pilha;
4. Reordenamento das variáveis na pilha;
5. ASLR - Randomização do espaço de endereços;

A seguir, cada uma será explicada em seus aspectos fundamentais.

4.4.1 Pilha não executável

A primeira, pilha não executável, é uma reação natural a um dos ataques mais comuns: o *buffer overflow*. Há registros de propostas de pilha não executável desde 1996 - conforme (ANLEY, 2007)(pg. 376). O *exploit* clássico sendo baseado na cópia de *shell code* para o buffer e posterior execução dele ficaria impraticável. Mas não demorou muito para os atacantes reagirem. Surgiram novas técnicas que funcionam mesmo quando não é possível executar o código injetado na pilha. Sua estratégia básica era: a partir do controle do *stack frame*, criar uma chamada válida para biblioteca C ou chamadas de sistema. Inicialmente, ela foi denominada **return-into-libc**.

Essa nova técnica de *exploit* abriria caminho para uma série de outras. Todas elas conseguindo desviar a execução para algum código já existente e, portanto, válido, evitando a necessidade de uma pilha executável. Para citar algumas delas: *ret2plt, ret2strcpy, ret2gets, ret2syscall, ret2data, ret2text*. Em (ANLEY, 2007), capítulo 14, há detalhes sobre elas.

4.4.2 W^ X

Impedir que memória com proteção de escrita seja executável e, bloquear a escrita para aquela que é executável é uma das melhores formas de proteção. Ataca justamente

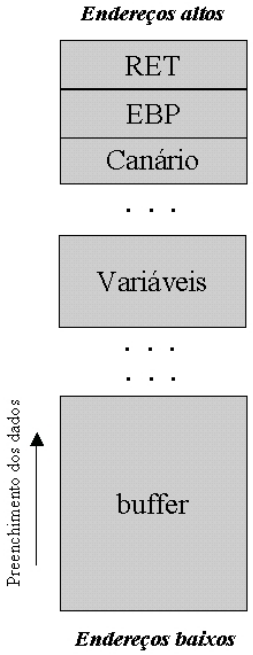

Figura 4.2: *Stack frame* protegido por canário. Fonte: (FURLAN, 2005).

um princípio fundamental da maioria das técnicas de ataque: injetar código (escrever) e executá-lo.

Embora essa técnica seja hoje em dia conhecida pelo batismo de Theo Raadt, desenvolvedor e líder do projeto do OpenBSD, ela tem sua origem na década de 1970. Em (ANLEY, 2007), é mencionado que o sistema Multics teria sido um dos pioneiros a contar com esse tipo de proteção. Para facilitar essa estratégia de defesa na arquitetura x86, em 2003, a AMD criaria o **NX(Non-eXecutable)**. Um suporte no hardware que identificasse uma página de memória que não pudesse ser executada. O equivalente da Intel seria o *ED(Execute Disable).*

Mesmo sendo uma excelente forma de impedir ataques, isolada, essa defesa não é capaz suficiente. Algumas técnicas derivadas de *return-into-libc* são imunes.

4.4.3 Canário para a pilha

Outra forma de proteção para a pilha é colocação de um canário. Trata-se de um valor(normalmente de 32 bits) que é posto no *stack frame* para identificar se houve um *overflow* na pilha. A figura 4.2 ilustra essa proteção. O canário é posto de forma a proteger o endereço de retorno. Ao término da chamada da função, ele é verificado e, caso não seja o valor esperado, a aplicação é terminada.

Sua primeira implementação foi o *StackGuard* em 1998, vindo a fazer parte do compilador GCC(GNU Compiler Collection) - posteriormente sendo substituído pelo SSP(*Stack-Smashing Protector*) (MARTINS, 2009). O SSP além de implementar proteção por canário, também atua reordenando as variáveis da pilha para aumentar a segurança - conforme explicado na Seção 4.4.4.

Atualmente é uma proteção padrão em quase todos os sistemais operacionais e certamente contribui muito para frear *exploits* de *buffer overflow*. Sua proteção é ainda maior quando combinada com o reordenamento da pilha.

Figura 4.3: Modelo de pilha ideal para o SSP. Fonte: (MARTINS, 2009).

4.4.4 Reordenamento de variáveis na pilha

É aplicada pelo SSP e complementa a proteção oferecida pelo canário. É uma barreira extra para que um *overflow* nos *buffers* - que só é detectado após o término da função - não seja usado para afetar outras variáveis.

Seu objetivo é, conforme (MARTINS, 2009): "isolar os arrays que podem vir a vazar dados, para que seu estouro não afete as outras variáveis locais da função. Isso garante a integridade das variáveis automáticas no decorrer da função, e evita o seu possível uso para a injeção de *shellcode*".

É baseada em um modelo ideal de pilha no qual as variáveis locais que não são *buffers* são melhor protegidas contra possíveis *overflows*. O modelo é melhor compreendido através da visualização da figura 4.3.

4.4.5 ASLR

O ***Address Space Layout Randomization*** implementa uma randomização dos endereços de forma a dificultar enormemente a vida dos atacantes. Bibliotecas e rotinas passam a ter endereços aleatórios e os saltos necessários para esses endereços ficam muito mais complexos de serem realizados.

Conforme explicado anteriormente, vários *exploits* dependem de um conhecimento prévio dos endereços. Portanto, essa aleatoriedade é muito interessante como forma de proteção genérica. Sua fraqueza, porém, conforme (ANLEY, 2007), está no fato de bastar algum endereço fixo para que ela não tenha efeito algum. Mas nem sempre é necessário que haja algo fixo; há uma técnica chamada ***heap spraying*** que é capaz de driblar o ASLR. Ela injeta várias porções de código executável na aplicação alvo para que, mesmo desconhecendo um endereço preciso, a chance de que ele seja encontrado venha a ser muito maior.

Há mais detalhes sobre ***heap spraying*** em (RATANAWORABHAN, 2008). No referido trabalho, inclusive, é sugerido um verificador de *heap* que procura impedir que esse tipo de ataque seja aplicado. Isso é feito através da detecção do padrão imposto pela

técnica de *spraying*, já que ela cria objetos na memória contendo código executável.

5 NULL POINTER EXPLOIT

Dentre as várias técnicas de exploits existentes, uma que certamente merece destaque, é o NULL pointer exploit. Sua disseminação é recente, sendo fruto da crescente dificuldade em aplicar técnicas que exploram vulnerabilidades de corrupção de memória.

Um marco para esse tipo de exploit certamente foi o artigo de Mark Dowd (DOWD, 2008). A forma como ele trouxe à luz uma falha na máquina virtual do ActionScript chamou a atenção de diversos especialistas na área. Isso porque, para muitos, o NULL pointer era apenas sinônimo de um *bug* que resultaria, no máximo, em uma negação de serviço. Por isso, o raciocínio empregado por ele serviria de base para encontrar muitos outros problemas.

O ano de 2009 chegou a ser considerado o ano do "kernel NULL pointer deference"em virtude da grande quantidade de falhas desse gênero encontradas no kernel do Linux. Como podemos encontrar em, (COX, 2010), a lista de problemas causados por esse tipo de vulnerabilidade foi extensa. Para sistemas Linux Red Hat, por exemplo, ainda conforme (COX, 2010), o NULL pointer foi considerado o grande vilão de 2009 com 6 vulnerabilidades.

Nesse capítulo, nossa intenção é apresentar esse tipo de vulnerabilidade e seu correspondente exploit. Assim como identificar os meios de detecção e prevenção.

5.1 O que é um NULL pointer

O primeiro ponto a ser abordado é o NULL pointer. Na linguagem de programação C, podemos considerar um ponteiro como um valor inteiro que referencia uma posição de memória. Ou seja, trata-se de um valor que aponta para o início de uma determinada região de memória. Quando um ponteiro é deferenciado, passamos a acessar o valor presente na posição de memória para o qual ele aponta. Ilustrando, segue pequeno trecho de código C.

Listing 5.1: Ponteiro em C

```
1 int val = 10;
2 int *pointer = &val;
3 /* pointer has the address of val */
4
5 int x = *pointer;
6 /* *pointer returns 10 */
```

No Linux, o arquivo stddef.h contém a definição de NULL, que por convenção, denomina um ponteiro com valor zero. Um ponteiro nulo, então, aponta para a posição

zero de memória. Como, em regra geral, os sistemas utilizam o esquema de memória virtual, na prática, esse endereço zero deve ser considerado tão somente no espaço de endereçamento do processo em questão. Como normalmente ele não constitui um mapeamento válido, pois os processos não iniciam com aquela porção mapeada, os acessos a essa região implicam violação às regras do esquema de memória virtual. Erros como esse resultam no término da aplicação. Por isso, na maioria dos casos, um acesso a um ponteiro nulo é apenas sinônimo de uma DoS (negação de serviço).

Diversas falhas em uma aplicação real podem levar à presença de um ponteiro zerado. Falhas ao inicializar uma estrutura de dados pode deixar ponteiros nulos inadvertidamente. Outro possível problema pode ocorrer quando o sistema tem sua memória esgotada e, a chamada responsável por alocar mais espaço retorna NULL, mas como essa possibilidade não é considerada pelo programador, o ponteiro a receber esse bloco de memória acaba ficando zerado e a aplicação segue normalmente.

Vemos, portanto, que um ponteiro nulo é um caso particular no qual a região de memória referenciada é aquela que inicia no endereço zero (no contexto de endereçamento do processo em questão - considerado o uso de memória virtual). Exceto em casos especiais, essa situação leva a erros na aplicação que resultam em seu término. Conforme trataremos a seguir, há casos em que um ponteiro nulo irá possibilitar um ataque.

5.2 Como funciona a técnica

A técnica de exploração desse tipo de vulnerabilidade irá variar conforme o contexto em que surge e como é utilizado o ponteiro nulo. Conforme exposto anteriormente, esse método não é tão genérico como as falhas de *buffer overflow*. São ataques mais focados que exigem ajustes muito maiores em função das especifidades da aplicação alvo. Como em outros gêneros de exploits, o objetivo desejado é a escrita de dados fornecidos pelo usuário em endereços arbitrários. Pois isso possibilita, por exemplo, a cópia de um *shellcode* para ser executado. Mas isso não é uma regra, há falhas de NULL pointer que envolvem ponteiros para funções que possuem um caminho mais simples para exploração.

Para fins de simplificação, vamos dividir os tipos de ataques com essa técnica em duas famílias. Como podem existir várias formas de exploração de aplicações em que surgem ponteiros nulos, para facilitar a compreensão, vamos tomar dois tipos representativos que são capazes de passar a ideia fundamental. Numa delas, um endereço que define a localização de uma escrita depende de um ponteiro zerado. Em outra, esse ponteiro define uma função a ser executada. A primeira chamaremos de **ponteiro nulo de escrita** e a segunda de **ponteiro nulo de função**.

5.2.1 Ponteiro nulo de escrita

Nessa situação, por algum motivo, um ponteiro que define um endereço de escrita fica nulo. Seja porque a memória alocada foi retornada em NULL e não foi verificada ou mesmo porque a aplicação não validou corretamente a entrada e o calculou indevidamente. O artigo de Mark Dowd, (DOWD, 2008), trata com riqueza de detalhes esse gênero de falha.

Para que uma falha de NULL pointer desse tipo possa resultar em um ataque, podemos elencar dois pré-requisitos:

- O ponteiro nulo é utilizado para calcular o endereço de uma escrita

- A escrita depende de algo fornecido pelo usuário além do NULL pointer
- Os dados a serem gravados podem ser controlados de alguma forma pelo usuário

Abaixo, ilustrando o que foi exposto, um pequeno trecho de código em linguagem C. Nele, o usuário fornece dados, mas como o endereço base de destino de uma cópia está zerado, é possível influenciar diretamente na escolha de onde são gravados. Essa vulnerabilidade implica a condição do atacante de gravar em um endereço arbitrário dados que ele pode controlar - que pode ser um *shellcode*.

Listing 5.2: Ponteiro em C

```
1 /* user input at user_data */
2 write_address = null_pointer + offset_influenced_by_user;
3 /* the address has been 'chosen' by the user */
4
5 memcpy(write_address, user_data, certain_size);
6 /* data is copied from one point to
7     another according to user's will */
```

5.2.2 Ponteiro nulo de função

Ocorre quando, por necessidade de dinamismo, uma função que deve cumprir determinado papel, é definida por um ponteiro. Normalmente, ele deve conter um valor válido de um endereço de memória que contenha código que cumpra com as ações desejadas. Mas isso pode, na prática, não se confirmar. Um valor NULL pode estar no ponteiro no momento em que a função é chamada.

Se o endereço zero não constituir uma região válida, a aplicação terminará com um erro. Mas e, se pusermos algo nessa região para ser executado? Imagine que um atacante tenha posto um *shellcode* justamente nesse ponto e provocou a chamada função definida pelo ponteiro nulo. Aí, certamente, poderíamos estar frente a um ataque com grandes chances de ser bem sucedido.

Como pré-requisitos, podemos elencar, portanto:

- Um ponteiro nulo define o endereço de uma função a ser chamada
- O usuário pode provocar a chamada dessa função
- É possível mapear para o endereço zero uma região válida de memória contendo dados do usuário

Com esses pontos básicos atendidos, há condições para o emprego da técnica. Como exemplo maior, mostraremos um bug no Kernel do Linux na seção 5.3.

5.3 Exemplos reais de NULL pointer exploit

Nessa seção, apresentamos vulnerabilidades reais que exemplificam o exploit em estudo. Aquele que não poderia faltar, sem dúvida, é a falha tratada por Mark Dowd. Seu artigo é rico em detalhes e mostra todas as etapas que tornam possível um ataque.

Também não poderíamos deixar de analisar os erros encontrados no Kernel do Linux em 2009. Isso porque problemas encontrados recentemente demonstraram que o sistema estava exposto fazia oito anos. "Eight Years of Linux Kernel Vulnerable": chegou a

ser o título de matérias divulgadas na Internet como encontrado em (CUNNINGHAM, 2009). Não demorou muito para que uma caçada a NULL pointer fosse realizada para que diversas falhas fossem encontradas. Abordaremos duas delas.

5.3.1 Falha na máquina virtual do ActionScript

Trata-se de uma vulnerabilidade que se enquadra no que denominamos **ponteiro nulo de escrita** (em 5.2.1). Consta no CVE como CVE-2007-0071. Foi objeto do estudo do artigo (DOWD, 2008). A falha ocorre na leitura de arquivos SWF(Shockwave Flash). Dados no arquivo são usados como parâmetros de alocação de memória. Se for passado um valor muito alto, como 2 gigabytes, a alocação não é bem sucedida e, por consequência, um ponteiro nulo é retornado. A aplicação realiza uma escrita na memória usando como parâmetros do cálculo do endereço de destino o ponteiro nulo com outro valor lido do arquivo (escolhido pelo usuário). Na página 7 de (DOWD, 2008), temos uma versão alto nível desse trecho de código em que ocorre o cálculo do endereço de destino e a escrita na memória.

O destino da escrita é escolhido pelo usuário quase de forma arbitrária. Existem algumas restrições como divisibilidade por 12 quando somado a 4. Mas isso não impede que um ataque seja realizado. Como é possível acompanhar em (DOWD, 2008), criando um arquivo do tipo SWF da forma correta e manipulando detalhes da máquina virtual do ActionScript, o atacante torna-se capaz de executar seu *shellcode* na máquina alvo. Isso é feito através da construção de *bytecode* nativo para a máquina virtual ActionScript que permite a injeção do *shellcode*. Após a execução do último, a aplicação retorna normalmente ao seu fluxo criando a impressão que nada demais ocorreu.

Pela enorme base de usuários que utilizam o Flash Player afetado, podemos dizer que o impacto da exploração dessa vulnerabilidade foi enorme. Principalmente porque a esmagadora maioria dos usuários jamais consideraria um uma apresentação em Flash como um potencial vetor de ataque.

Vários fatores foram necessários para que um exploit fosse possível nesse caso. Falhas na validação de dados fornecidos pelo usuário foram os mais graves. Mas não foram os únicos. A aplicação também não soube lidar corretamente com erros na alocação de memória. Essa vulnerabilidade, como tantas outras, portanto, surge apenas pela combinação de uma série de problemas que são devidamente concatenados por uma mente criativa e obstinada de um atacante.

5.3.2 Falhas no kernel do Linux

Existem diversas falhas documentadas no kernel do Linux relacionadas a NULL pointer. Desde problemas na inicialização de estruturas de dados, condições de corrida inesperadas e até erros na compilação. São falhas que surpreenderam pelo tempo que permaneceram escondidas e algumas até pela relativa facilidade de exploração conforme veremos a seguir.

Iniciaremos pelo erro conhecido no CVE como CVE-2009-2692. Trata-se de uma vulnerabilidade muito grave que possibilita uma escalada de privilégios no sistema. Versões desde 2.6.0 a 2.6.30.4 e 2.4.4 a 2.4.37.4 estavam suscetíveis a esse *bug*; nada menos que 8 anos de *releases* do kernel.

Sua origem encontra-se na inicialização incorreta de ponteiros de funções em estruturas de dados do kernel; nesse caso, *proto_ops_structures*. Um *bug* em uma macro (SOCKOPS_WRAP) acabava deixando não inicializadas funções responsáveis, por exemplo, de assumir o controle quando uma operação não disponível fosse requisitada.

Enquadra-se, portanto, no que convencionamos como **ponteiro nulo de função**.

Mais especificamente, quando um socket fosse usado e, fosse chamada a função sock_sendpage, e não fosse possível enviar a página, a função sock_no_sendpage deveria ser despertada para tratar a situação. Mas, conforme explicamos, o valor NULL estaria ocupando o devido local do endereço da função sock_no_sendpage. Logo, o contexto da execução seria transferido para a região de memória iniciada em zero. Por isso, sendo injetado um código nesse bloco, ele seria executado com os privilégios do kernel. Em (TINNES, 2009), é possível obter mais detalhes sobre a questão.

A seguir, segue o código que tira proveito dessa vulnerabilidade e possibilita ao atacante a execução de código com privilégio máximo no sistema.

Listing 5.3: Exploit para CVE-2009-2692

```
1  #include <stdio.h>
2  #include <sys/socket.h>
3  #include <sys/mman.h>
4
5  #define PAGE_SIZE 4096
6
7  int main() {
8    void *mem;
9    char template[] = "/tmp/fileXXXXXX";
10   int fdin = mkstemp(template);
11   int fdout = socket(PF_PPPOX, SOCK_DGRAM, 0);
12   int i;
13
14   mem = mmap(NULL, 64, PROT_READ | PROT_WRITE,
15         MAP_FIXED | MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
16
17   printf("mmap=%x\n", mem);
18
19   char *shellcode[] =
20        "\x31\xdb\xf7\xe3\xb0\x04\x43\xeb\x0a\x59"
21        "\xb2\x1d\xcd\x80\xb0\x01\x4b\xcd\x80\xe8"
22        "\xf1\xff\xff\xffgot_kernel!\a\n";
23
24   for(i=0; i<36; i++)
25     (char*)mem[i] = shellcode[i];
26
27   printf("fdin=%d\n", fdin);
28   printf("fdout=%d\n", fdout);
29   printf("%d\n", unlink(template));
30   printf("%d\n", ftruncate(fdin, PAGE_SIZE));
31   printf("%d\n", sendfile(fdout, fdin, NULL, PAGE_SIZE));
32
33   return 0;
34 }
```

Na linha 14, é feito uma alocação de um bloco de memória iniciado no endereço zero; nele é posto o *shellcode*. Já a operação que desencadeia o erro encontra-se na linha 31.

A chamada a sendfile irá exigir, no contexto criado de erro de envio, uma chamada a sock_no_sendpage. Como o endereço dela estará em zero, conseguimos desviar para o *shellcode* no contexto do kernel.

A segunda vulnerabilidade que utilizaremos como exemplo é conhecida no CVE como CVE-2009-3547. Qualquer versão anterior a 2.6.32-rc6 é vulnerável. Como o exemplo anterior, também trata-se de uma falha que possibilita escalada de privilégios. Mas é um *bug* de mais difícil compreensão e exploração. Ainda assim, vários *exploits* foram divulgados por diferentes autores; provando sua viabilidade.

Ocorre devido a uma condição de corrida que pode deixar um ponteiro nulo. De forma simplificada, podemos apontar a falha como uma não proteção de uma região crítica. As funções pipe_read_open(), bem como pipe_write_open() e pipe_rdwr_open(), relacionadas à intercomunicação de processos, podem não proteger corretamente o acesso ao ponteiro que controla o total de processos leitores/escritores. Podemos ver a dificuldade de reprodução dessa condição de corrida pelas palavras, (CHEW, 2009), do desenvolvedor do kernel do Linux Earl Chew : "Note that the failure window is quite small and I could only reliably reproduce the defect by inserting a small delay". Segundo Earl, apenas uma pequena janela de tempo surgia para que o problema fosse detectado.

Podemos considerar essa falha muito mais sutil em sua manifestação que a primeira. Ainda assim, pelo *diff* apresentado por Earl Chew para tratar o problema, vemos que, novamente, um caminho possível para evitar o pior foi a validação do ponteiro. Abaixo, apresentamos *patch* por Earl Chew, encontrado em (CHEW, 2009), que impede a exploração do erro.

Listing 5.4: Patch para CVE-2009-3547

```
1  index 52c4151..ae17d02 100644 (file)
2  --- a/fs/pipe.c
3  +++ b/fs/pipe.c
4  @@ -777,36 +777,55 @@ pipe_rdwr_release(struct inode *inode,
5                          struct file *filp)
6   static int
7   pipe_read_open(struct inode *inode, struct file *filp)
8   {
9  -        /* We could have perhaps used atomic_t,
10 -        but this and friends below are the
11 -        only places.  So it doesn't seem worthwhile.  */
12 +        int ret = -ENOENT;
13 +
14          mutex_lock(&inode->i_mutex);
15 -        inode->i_pipe->readers++;
16 +
17 +        if (inode->i_pipe) {
18 +                ret = 0;
19 +                inode->i_pipe->readers++;
20 +        }
21 +
22          mutex_unlock(&inode->i_mutex);
23
24 -        return 0;
25 +        return ret;
```

```
26   }
27
28   static int
29   pipe_write_open(struct inode *inode, struct file *filp)
30   {
31  +        int ret = -ENOENT;
32  +
33           mutex_lock(&inode->i_mutex);
34  -        inode->i_pipe->writers++;
35  +
36  +        if (inode->i_pipe) {
37  +                ret = 0;
38  +                inode->i_pipe->writers++;
39  +        }
40  +
41           mutex_unlock(&inode->i_mutex);
42
43  -        return 0;
44  +        return ret;
45   }
46
47   static int
48   pipe_rdwr_open(struct inode *inode, struct file *filp)
49   {
50  +        int ret = -ENOENT;
51  +
52           mutex_lock(&inode->i_mutex);
53  -        if (filp->f_mode & FMODE_READ)
54  -                inode->i_pipe->readers++;
55  -        if (filp->f_mode & FMODE_WRITE)
56  -                inode->i_pipe->writers++;
57  +
58  +        if (inode->i_pipe) {
59  +                ret = 0;
60  +                if (filp->f_mode & FMODE_READ)
61  +                        inode->i_pipe->readers++;
62  +                if (filp->f_mode & FMODE_WRITE)
63  +                        inode->i_pipe->writers++;
64  +        }
65  +
66           mutex_unlock(&inode->i_mutex);
67
68  -        return 0;
69  +        return ret;
70   }
71
72   /*
```

Vemos que nas linhas 16 a 21 temos a inserção de uma verificação do ponteiro i_pipe.

Logo, mesmo na presença da concorrência, não há chance de que i_pipe seja deferenciado com valor NULL.

Ambas as vulnerabilidades analisadas, ao nosso ver, apontam dois problemas graves. Primeiramente, a aparente falta de rigor nos testes. Como foi possível que uma vulnerabilidade como CVE-2009-2692 tenha permanecido por 8 anos sem ser descoberta pelos desenvolvedores? Como se trata de um problema de inicialização de variáveis, não seria tão complexo detectá-lo.

Já a segunda questão que gostaríamos de apontar, é o problema do mapeamento do endereço zero por parte da aplicação do usuário permitindo que o kernel o acesse e execute código nele. Na seção 2.3.4, foi demonstrado como kernel e aplicações em modo usuário compartilham o mesmo espaço de endereçamento lógico. Essa decisão tem impacto direto sobre a possibilidade dos *exploits* abordados. Houvesse uma devida separação entre os segmentos, usando os atributos de base e limite por exemplo, isso jamais ocorreria. Pois o mapeamento do endereço zero em modo usuário não seria acessível pelo kernel.

Como forma de tratar esse problema, foi usado o parâmetro mmap_min_addr. Ele define no sistema qual o endereço mais baixo que pode ser requisitado mapeamento de memória via mmap. Quando seu valor é diferente de zero, como 4Kb ou 64Kb, o mapeamento para endereço zero, até o valor escolhido, fica vedado. Por padrão, seu valor era zero, mas foi elevado para 4096 a partir de junho de 2009, como resposta às vulnerabilidades de NULL pointer. O autor do *patch* foi Christoph Lameter - encontrado em (LAMETER, 2009)

5.3.3 NULL pointer em ARM e XScale

Embora o foco do presente trabalho recaia sobre a arquitetura x86, é válido identificar a repercussão de um acesso a posição zero de memória em outros casos. Existem arquiteturas nas quais esse endereço já é mapeado inicialmente. Podemos apontar o caso da ARM e da XScale; ambas para sistemas embarcados. Nelas, o vetor de exceções se encontra nessa posição. Ele contém, por exemplo, o endereço que define o vetor para o tratamento das interrupções de software.

Essa vulnerabilidade, é tratada por Barnaby Jack, pesquisador de segurança da Juniper, em (JACK, 2007). Conforme Jack, caso alguma aplicação nas arquiteturas em questão possua alguma falha na qual o endereço de destino de uma escrita seja um ponteiro nulo, o vetor de exceções acaba sendo sobrescrito. Isso potencializa enormemente um erro de NULL pointer.

Como exemplo, em (JACK, 2007), é apresentada uma falha na biblioteca libpng. Um tratamento inadequado da alocação de memória para imagens, que retornava NULL, permitia que os dados de uma imagem fossem copiados via memcpy() para o endereço zero. Por esse caminho, um atacante seria capaz de sobrescrever a tabela de endereços de interrupções de software. Assim, bastaria uma chamada do sistema em virtude de uma interrupção, para que o código injetado pudesse ser executado.

Segundo avaliação de Jack, uma das formas de prevenir esse tipo de ataque é não permitir a escrita na área do vetor de exceções. Outra medida sugerida, e existente em versões posteriores das arquiteturas, como ARM9, é a possibilidade de mapeamento do vetor de exceções para endereços mais altos - como 0xFFFF00000. De qualquer forma, não resta dúvida que os projetistas cometeram sério equívoco nas escolhas envolvidas no vetor de exceções.

5.4 Como evitar o problema

Há vários caminhos que podem convergir para que não existam vulnerabilidades causadas por ponteiros nulos. Nas seções anteriores, foram abordados diversos aspectos que demonstraram porque o problema existe e os pré-requisitos para que ele seja explorado.

Podemos elencar três pontos principais sobre os quais podem se assentar as defesas e medidas de precaução contra os exploits de NULL pointer. A prevenção passa diretamente por:

- Boas escolhas arquiteturais
- Um desenvolvimento consciente da ameaça do ponteiro nulo
- A aplicação contínua de testes

5.4.1 Decisões estruturais

A forma como o sistema é concebido em termos de endereçamento e possibilidades de mapeamento das regiões de memória deve estar ciente do riscos impostos por ponteiros nulos. Conforme tratado na seção referente a exemplos de exploits, vide 5.3.2, não é aceitável que o sistema operacional divida com a aplicação do usuário o mesmo espaço de endereçamento. Considerando que

Da mesma forma, de acordo com o exposto na seção 5.3.3, qualquer decisão arquitetural que coloque importantes estruturas de controle do sistema no endereço zero, é extremamente temerária. Isso porque havendo uma referência a um ponteiro nulo que possa ser explorada, o atacante tem seu trabalho facilitado uma vez que haja um alvo direto na região de memória zero. Arquiteturas como a XScale, que deixam a tabela de rotina de interrupções iniciando no endereço zero, assumem um risco desnecessário.

Assim, desde a concepção de uma arquitetura ou da organização de um sistema operacional, existem fatores que devem ser relevados sobre a ótica de uma vulnerabilidade de ponteiro nulo. Um design preventivo contribui decisivamente para que esse tipo de problema sequer seja possível. Mesmo que isso resulte em problemas de compatibilidade ou perdas de desempenho, as escolhas mais seguras devem ter prioridade.

5.4.2 Programação consciente

Como em todas as vulnerabilidades, o papel do programador é primordial. Sua incapacidade de vislumbrar situações de risco e tratá-las devidamente faz total diferença. É absolutamente necessário que essa forma de falha de segurança seja considerada. Um cuidado especial deve ser tomado para que não surjam ponteiros nulos durante a execução de uma aplicação e, mesmo que isso ocorra, que seja devidamente detectado e tratado.

Certos procedimentos, portanto, são indispensáveis. Como, por exemplo, verificar se, ao alocar memória dinâmica, não recebemos um NULL. Não é aceitável que os ponteiros sejam manipulados sem a devida verificação. O programador não pode simplesmente assumir que receberá um endereço válido. Ambos os exploits reais apresentados, vide 5.3, poderiam ter sido evitados caso houvesse uma conferência correta dos valores manipulados.

Logo, o conhecimento dos riscos que a falha de NULL pointer apresenta é essencial para que os desenvolvedores construam aplicações mais seguras. Padrões para validar ponteiros antes de seu uso devem ser apresentados e cobrados. Nesse aspecto, ferramentas

automáticas para verificação da qualidade do código podem ser um grande diferencial. Assim, práticas de programação não seguras são detectadas desde a origem.

5.4.3 Testes

Toda e qualquer forma de teste contribui direta ou indiretamente para a detecção desse tipo de falha. Mas é essencial que a aplicação seja examinada sob a ótica de testes de requisitos negativos. No capítulo 6, que trata de Fuzzing, são apresentadas diversas formas de testes que podem auxiliar.

É possível, por exemplo, testar a aplicação simulando falhas na alocação de memória. De tal forma que, certas requisições de memória propositalmente retornem NULL. Com esse tipo de cenário, situações inusitadas podem ser criadas com facilidade. Analogamente, outras bibliotecas também podem ser substituídas por versões de teste que gerem contextos nos quais a aplicação é forçada a tratar ponteiros nulos.

No caso da vulnerabilidade CVE-2009-2692, analisada em 5.3.2, situações que simulassem uma falha no envio de um arquivo, como no *exploit* apresentado para CVE-2009-2692, seriam suficientes para detectar o problema. Isto porque ocorreria a falha na chamada a função cujo ponteiro estaria nulo. Por isso a enorme necessidade de testes, notoriamente aqueles que criem contextos em que falhas sejam inseridas.

6 FUZZING: DETECÇÃO DE VULNERABILIDADES

Dentre muitas alternativas na busca por vulnerabilidades no software, a abordagem fuzzing, sem dúvida, deve ser destacada. Constitui um meio que pode resultar em ótima relação custo benefício, pois pode, em muitos casos, oferecer uma resposta rápida e de baixo custo. Vem se tornando cada vez mais sofisticada e já assume papel importante em grandes desenvolvedores de software.

6.1 O que é fuzzing?

Enquanto as técnicas padrão de teste de software se concentram em testes positivos (também conhecidos como testes de conformidade), a técnica fuzzing é voltada para os requisitos negativos. Não busca testar as features, mas visa verificar o comportamento do software nos casos em que o sistema recebe entradas mal formadas ou fora do padrão esperado.

Essa característica é extremamente interessante no que se refere à detecção de vulnerabilidades. Isso porque elas geralmente são descobertas quando se busca combinações de entradas não testadas originalmente pelo desenvolvedor.

Podemos compará-la à técnica injeção de falhas - muito embora essa seja mais conhecida por testes em hardware. O princípio, porém, é muito semelhante. Entradas mal formadas são fornecidas ao hardware de forma a sabermos com exatidão as possíveis reações do sistema.

Considerando as observações introduzidas acima, definimos fuzzing, conforme (TAKANEM, 2008), como **um método de descoberta de falhas no software que fornece entradas inesperadas ao sistema e o monitora esperando por exceções** .

Por essa definição, vemos que nenhum conhecimento do funcionamento interno da aplicação é exigido. Nesse sentido, fuzzing é considerado um tipo de teste caixa preta (Black Box Testing) - mas é importante ressaltar que muito embora o código fonte não seja necessário, ele pode ser de grande ajuda na aplicação do método. Há, ainda, novas formas de fuzzing que partem do somente código fonte para geração dos testes. A denominada fuzzing de caixa branca (Whitebox Fuzz Testing) busca aplicar o conceito central de variação nas entradas aliada ao conhecimento interno da aplicação visando superar barreiras intrínsecas aos testes caixa preta.

Vemos, portanto, que esse é um campo extremamente amplo dentro da área de testes de software. Nosso intuito é fornecer uma visão ampla que permita demonstrar seu valor no contexto da busca por vulnerabilidades.

6.2 Origens e breve histórico

6.2.1 Uso do conceito antes do surgimento oficial

Essa metodologia de teste é relativamente recente. Surge na década de 1980. A primeira ferramenta com conceitos fuzzing teria surgido com o *The Monkey*. Não era um software, mas um gerador de cliques e movimentos de mouse que visavam simular um macaco utilizando o Macintosh das maneiras mais inesperadas possíveis. Os desenvolvedores a consideraram uma excelente ajuda pois através dela puderam descobrir uma série de bugs e conseguiram aumentar a robustez do sistema. (HERTZFELD, 1983).

6.2.2 O surgimento oficial

A experiência acima seria uma das primeiras formas de uso do conceito de fuzzing. Mas como marco oficial do nascimento, podemos considerar a pesquisa feita por Barton Miller no final da década de 1980 e início dos anos 90. Como fruto do seu trabalho, surgiu a primeira ferramenta fuzzing em software chamada Fuzz. Miller e sua equipe a utilizaram para gerar entradas randômicas para testar ferramentas básicas dos sistemas UNIX. Sua surpresa foi enorme em perceber como foi possível derrubar boa parte das aplicações sem muito esforço. Ficava nítido o enorme potencial de um novo conceito a ser explorado.

6.2.3 O desenvolvimento da técnica

A partir do final da década de 1990, os pesquisadores, entusiasmados com os resultados iniciais, criaram o projeto PROTOS. Seu objetivo estava na geração de suítes de teste capazes de simplificar a análise de protocolos - como HTTP, DNS e SNMP. Nesse ponto, a simples geração de entrada randômica já havia evoluído para ferramentas que modelavam os protocolos. No início dos anos 2000, a Microsoft chegou a investir no projeto PROTOS. Esse gigante do software viria apostar fortemente nesse caminho, pois como veremos mais adiante, seção 6.7, ela será responsável pela criação de importantes ferramentas na área.

6.3 Conceitos importantes

A seguir são discutidos conceitos que são peças de grande relevância para o melhor entendimento da técnica fuzzing.

6.3.1 Cobertura

Quais partes do código da aplicação testado são testadas. Esse conceito retrata um dos objetivos básicos de qualquer tipo de teste. A necessidade de cobrir o máximo possível o código da aplicação alvo.

6.3.2 Superfície de ataque

Muito embora a intenção seja alcançar cobertura máxima, muitas vezes certo trechos do código simplesmente são inacessíveis a fatores externos. De forma que, nenhum tipo de entrada possa alterar em nada seu comportamento. A superfície de ataque é justamente todo o código possível de ser coberto - pois, em contraste ao exposto acima, é influenciável por ações do usuário.

6.4 Etapas Fuzzing

Segundo (TAKANEM, 2008), podemos dividir a aplicação da metodologia fuzzing de teste em 5 etapas.

- Identificação das entradas
- Geração das entradas
- Envio das entradas
- Monitoramento do alvo
- Análise dos resultados

6.4.1 Identificação das entradas

Etapa que corresponde à busca por interfaces ao sistema alvo. Podem ser sockets de rede, variáveis de ambiente, arquivos, argumentos da linha de comando, interfaces de chamada remota(RPC), entre outros. Toda e qualquer forma de comunicação que possa influir na execução deve ser considerada. Como um exemplo mais surpreendente, podemos citar a memória compartilhada.

6.4.2 Geração das entradas

É o ponto crítico da metodologia. Saber como criar os dados a serem passados ao alvo. Podem ser completamente randômicos, mutações de dados pré-existentes ou mesmo fruto de uma completa modelagem de um protocolo. Se estamos testando um servidor web por exemplo, podemos gerar requisições totalmente aleatórias, alterar sessões web legítimas gravadas para inserir possíveis falhas ou até modelarmos o protocolo HTTP para criação de sessões semi-válidas.

6.4.3 Envio das entradas

Consiste no fornecimento das entradas criadas ao sistema alvo. Implica o contato com o sistema através de suas interfaces. Seja ela a linha de comando, o sistema de arquivos ou conexões ao um servidor. Apenas alterar uma variável de ambiente antes de iniciar a aplicação testada já pode ser considerada um envio de entrada.

6.4.4 Monitoramento do alvo

Pouco adianta interagir com sistema testado de todas as formas possíveis sem acompanhar criteriosamente sua execução. Suas manifestações devem ser observadas e possíveis falhas ou problemas, objetivos do teste, não podem passar despercebidas. Naturalmente, quanto mais qualificada a técnica fuzzing, maior a capacidade de identificação de problemas no sistema alvo. Por isso, saber identificar, por exemplo, falhas de corrupção de memória, negações de serviço, acessos não permitidos, constitui o grande diferencial de um fuzzer. O uso de um debugger na aplicação alvo é uma das alternativas.

6.4.5 Análise dos resultados

Com as informações coletadas pelo monitoramento, torna-se necessário identificar se existem ou não falhas de segurança. Muitas vezes os problemas manifestados constituem

bugs que não implicam possibilidade de exploit. Por isso a necessidade de uma análise detalhada e qualificada para que as reais aberturas no sistema avaliado sejam encontradas.

6.5 Tipos de fuzzers

Para classificar os fuzzers podemos seguir dois critérios básicos. Eles determinam a área de atuação e o tipo de entradas geradas.

- Tipo de vetor de ataque
- Complexidade dos casos de teste

6.5.1 Tipos por vetor de ataque

De acordo com o tipo de aplicação ao qual o fuzzer se dirige, ele possui um determinado vetor de ataque. Pode ser voltado, por exemplo, para testes de clientes web (como browsers). Nesse caso, seu vetor de ataque está no protocolo HTTP. De forma análoga, se for voltado para testes de leitores de pdf, seu vetor de ataque estará na geração dos arquivos.

6.5.2 Tipos por complexidade de casos de teste

A complexidade com que o fuzzer cria suas entradas constitui outro meio de classificação. Alguns podem ser muito simples pois apenas randomizam certos parâmetros antes de fornecê-los ao sistema alvo. Outros, porém, podem conter todo um modelo de um protocolo; sendo capazes de gerar complexas interações semi-válidas em que apenas certos parâmetros sofrem algum tipo de alteração visando disparar algum erro.

Geralmente, os mais simples, meramente randômicos, acabam possuindo baixa cobertura do código testado. Isso porque eles não alcançam grande profundidade na aplicação testada. Logo na superfície, algum parâmetro gerado acaba não sendo aceito e mudanças em outros pontos da entrada sequer são considerados. É o caso, por exemplo, de um testador de um servidor HTTP que, sendo totalmente randômico, cria apenas requisições mal formadas que sequer chegam a disparar alguma rotina de geração de resposta pelo servidor.

Por esse critério, podemos citar as seguintes famílias de fuzzer:

Estáticos ou randômicos Os mais simples. Não possuem qualquer noção de protocolo. Testam aplicações baseadas em requisição/resposta sem controle de estado.

Baseados em bloco Implementam estruturas básicas de requisição/resposta e são capazes de validar as entradas de forma a respeitar parâmetros como checksums.

De geração dinâmica ou baseados em evolução Não compreendem o protocolo a ser testado, mas baseado nas respostas do sistema podem, dinamicamente, gerar entradas.

Baseados em modelos ou simuladores Podem constituir a implementação completa de um protocolo. Permitem gerar entradas que em sequência de acordo com um estado. Por isso, podem, por exemplo, interagir com aplicações que trabalham com sessões.

6.6 Monitoramento da aplicação

A interação com a aplicação testada torna possível examiná-la de forma a revelar os erros; ainda assim, isso de nada é útil caso não saibamos identificar no sistema alvo os sintomas das falhas. O monitoramento é o responsável por passar os alertas de problemas. Logo a pergunta que se impõe é: o que pode ocorrer no sistema indicando uma falha? Entre as manifestações que devem ser reconhecidas pelo monitoramento, podemos citar:

- Negações de serviço (DoS) (o sistema deixa de responder)
- Problemas relacionados à memória (segfault)
- Injeção de metadados (como injeção de SQL)
- Permissão de acesso a áreas proibidas

As formas de monitoramento variam, naturalmente, tanto quanto os próprios sistemas testados. Acompanhar uma aplicação escrita em C e um portal web em PHP são tarefas bem distintas - exigindo, naturalmente, técnicas diferenciadas.

Uma forma genérica é garantir que o alvo está sempre respondendo corretamente a certas interações. Assim, é possível identificar se ocorre ou não uma negação de serviço. Logo, manter requisições bem formadas com respostas conhecidas intercaladas aos demais testes, auxilia o reconhecimento de falhas.

Outro meio, talvez o mais natural, é acompanhar a saída gerada pelo alvo na suas mais variadas formas. É o caso da saída padrão, dos logs, de arquivos temporários, entre outros. O fuzzer pode procurar por padrões que revelem as falhas - como, por exemplo, avisos de erros de corrupção de memória.

O uso de um depurador (debugger) também constitui outra forma de monitoramento. Com esse recurso, podemos esperar por determinadas exceções e descobrí-las tão prontamente os casos de teste as gerem.

6.6.1 Meios intrusivos

Para auxiliar na descoberta das falhas o mais na origem possível, existem métodos ainda mais intrusivos que os expostos anteriormente. Fazer com que a aplicação testada carregue bibliotecas diferentes das originais é um dos caminhos.

Trocando a biblioteca que faz o gerenciamento de memória dinâmica, responsável pelo chamadas como malloc(), é possível devolver devolver à aplicação blocos de memória que permitam a fácil identificação de overflows. Assim, muito antes que um erro de corrupção de memória fosse gerado, ele já teria sido detectado.

Numa mesma abordagem, a técnica chamada simulação binária, (TAKANEM, 2008) pg. 181, também visa acompanhar com enorme proximidade o sistema alvo. Esse é o caso da ferramenta Valgrind (encontrada em http://valgrind.org). Nela, é usada uma CPU sintética que recebe todas as instruções e pode analisá-las na busca por problemas antes de serem repassadas à CPU real. Todos os acessos à memória são controlados. Com esse tipo de acompanhamento, as vulnerabilidades podem ser encontradas em tempo real e informações valiosas sobre sua possibilidade de exploração já são conhecidas.

6.7 White Fuzz Testing: execução simbólica e fuzzing

Nessa seção, apresentamos uma nova abordagem do ramo fuzzing. Fruto da pesquisa de Patrice Godefroid e associados descrita em (GODEFROID, 2008). Originalmente,

a técnica fuzzing sequer se apoiava no código fonte para busca de qualquer tipo de auxílio no aumento de sua efetividade. Com o tempo, porém, foi possível perceber que, partindo de certas informações do funcionamento interno da aplicação, os resultados obtidos poderiam ser melhores. Vindo do outro extremo, o White Fuzz Testing não apenas faz uso do código fonte, mas o executa simbolicamente - sendo totalmente caixa branca.

6.7.1 Deficiências do método caixa preta

Antes de apresentarmos a técnica de fuzzing caixa branca desenvolvida por pesquisadores da Microsoft, é necessário expor certas deficiências naturais dos testes de caixa preta.

Tomemos como exemplo uma aplicação que possua 3 parâmetros de entrada (de 32 bits): x, y e z. No seu código fonte, existe uma condição na qual, a menos que valor de y seja 25, apenas um primeiro bloco da aplicação é executado. Logo percebemos que, a cobertura de um teste caixa preta nesse caso fica seriamente prejudicada. Dificilmente teremos entradas geradas com essa particularidade a ponto de explorar com efetividade possíveis erros. Essa dificuldade de percorrer todos os caminhos de execução possíveis é um fator que limita muito a capacidade da maioria das abordagens fuzzing de caixa preta.

6.7.2 Funcionamento básico

A técnica opera fornecendo, primeiramente, entradas válidas à aplicação. Então ela é executada simbolicamente com todas as condições sendo registradas. Cada um dos blocos condicionais acaba escolhendo um caminho distinto de execução dadas as entradas iniciais. Assim, é possível saber que certos valores de entradas implicam a exploração de certos caminhos. Numa próxima execução, usando um resolvedor lógico, as condições são negadas de forma a descobrir novos valores de entrada que possibilitem que a aplicação siga outros caminhos.

Operando iterativamente, o fuzzer aliado à execução simbólica acaba explorando os mais variados caminhos existentes. O intuito é garantir que situações inesperadas sejam encontradas graças a combinações de entradas escolhidas justamente para adentrar os blocos de código que podem não ter sido testados adequadamente.

Para ilustrar, simplificadamente, apresentamos abaixo um pequeno trecho de código.

Listing 6.1: Código de teste para ilustrar técnica

```
1 void test(int a, int b) {
2     if(a > 10) {
3       if(b == 5)
4           error();
5       ok();
6     } else {
7       ok();
8     }
9 }
```

No caso acima, a poderíamos fornecer, inicialmente, os valores 0 e 1 para a e b respectivamente. O primeiro bloco condicional, avaliado pela execução simbólica, assumiria falso e cairíamos em ok().

Na segunda etapa, o resolvedor dos blocos condicionais, buscando negar a primeira condição, descobriria, que o valor de a deveria ser 11. Mantido o valor inicial de b, conseguiríamos, nessa nova execução simbólica, chegar em um novo bloco condicional.

Dessa vez, b sendo 1, caímos novamente em ok(). Na iteração seguinte, porém, o algoritmo seria capaz de identificar que, para negar o segundo bloco condicional, seria preciso que b assumisse o valor 5. Assim, com a valendo 11 e b com valor 5, numa última execução encontraríamos error().

Logo, o Whitebox Fuzz necessita de um sistema muito sofisticado de execução simbólica bem como um resolvedor de condições suficientemente inteligente para encontrar os valores corretos de entradas que explorem toda aplicação.

6.7.3 Limitações

Teoricamente, com a aplicação do Whitebox Fuzz, é possível alcançar O alcance de uma cobertura completa fica limitado, segundo os autores, por dois fatores:

- Explosão combinatorial de caminhos
- Imperfeições da execução simbólica

Devido a enorme quantidade de possíveis caminhos de uma aplicação de grande porte, pode não ser factível explorar a todos. Isso pode ser contornado examinando certas funções em isolado através de sumários que identificam pré-condições e pós-condições de cada uma.

Além da dificuldade da quantidade dos caminhos, as imperfeições na execução simbólica podem apresentar sérias restrições. Instruções muito complexas bem como chamadas de sistema e de certas bibliotecas podem ser extremamente difíceis de prever. Nesses casos, a randomização pode ser usada mas gerando prejuízos à precisão.

6.7.4 SAGE: implementação da técnica

Como resultado da pesquisa, foi implementado na Microsoft, a ferramenta SAGE (Scalable, Automated, Guided Execution). Embora o acesso a SAGE seja restrito a pessoal da empresa, a abordagem aplicada é de domínio público. Pela experiência relatada em (GODEFROID, 2008), é possível dizer que a SAGE foi bem sucedida em encontrar erros de segurança até então não descobertos por outras ferramentas. Os autores, observam, porém, que uma das suas grandes dificuldades é a lentidão imposta pela execução simbólica.

7 CONCLUSÃO

Nesse trabalho foram abordados aspectos essenciais relacionados à segurança do software. Ao tratar de vulnerabilidades e técnicas de *exploits*, ele objetivou trazer ao leitor um contexto fundamental para um entendimento da área. Dada a relevância que o software atingiu nos dias de hoje, não é mais admissível que qualquer desenvolvimento sério desconsidere princípios de segurança. Sendo eles: as possíveis vulnerabilidades, as formas de ataque e, naturalmente, as formas de prevenção.

No que se refere às vulnerabilidades, esse trabalho, ao tratar de sua classificação, pode identificar que esse tópico ainda não é pacífico no meio acadêmico ou industrial. Mesmo que tenham sidos feitos avanços, a comunidade carece de um padrão aceito uniformemente. Ficou nítido que a complexidade dessa tarefa é enorme. O próprio caráter multifacetado das vulnerabilidades explica um pouco essa barreira; elas podem ser analisados por diversos ângulos e estamos longe de encontrar uma visão unificadora que traga sentido a todas suas faces. Apenas assim seria alcançada uma taxonomia em sentido estrito.

No campo dos *exploits*, pode ser visto que, com a evolução natural das técnicas de ataque e de defesa, a vida dos especialistas na área torna-se cada vez mais árdua. As formas mais simples de explorar vulnerabilidades já não são mais efetivas; seja porque as falhas que as tornam possíveis ficaram menos frequentes no desenvolvimento, seja porque proteções mais bem concebidas foram sendo habilitadas por padrão nos sistemas. Isso vai obrigando os atacantes a encontrarem novos métodos cada vez mais sofisticados que, naturalmente, vão exigindo conhecimento ainda mais específico. Em alguns momentos, porém, ainda será possível surgir alguma espécie de reviravolta - como na descoberta dos diversos erros de *NULL pointer* no kernel do Linux. Episódio que demonstrou a existência de uma série de falhas por vários anos em um dos sistemas mais utilizados - surpreendentemente, algumas delas de fácil exploração.

Para a prevenção de problemas de segurança no software, uma das principais propostas apresentadas foi o uso do testes fuzzing. Sendo uma técnica extremamente eficiente e que já vem sendo usada por atacantes para a descoberta de problemas nos sistemas, esse trabalho buscou demonstrar seu valor e indicá-la como arma a ser utilizada pelos próprios desenvolvedores. Por que já não conceber, desde o princípio, um projeto considerando essa alternativa de teste se os atacantes certamente irão utilizá-la? Conforme foi visto, gigantes da área, como a Microsoft, já perceberam seu enorme valor e investem fortemente nela. É necessário, portanto, que, ao menos, consideremos essa possibilidade. Isso porque não é aceitável correr o risco de deixar apenas para os atacantes utilizarem e aperfeiçoarem uma técnica que possa desequilibrar em favor deles.

REFERÊNCIAS

ANLEY, C. **The shellcoder's Handbook**: discovering and exploring security holes. 2ª.ed. [S.l.]: Wiley Publishing, Inc., 2007.

BOVET, D. P. **Understanding the Linux Kernel, 3rd Edition**. [S.l.]: O'Reilly, 2005.

CHEN, S. **Non-Control-Data Attacks Are Realistic Threats**. Junho, 2005. Disponível em: http://research.microsoft.com/en-us/um/people/shuochen/papers/usenix05data_attack.pdf. Acessado em Junho 2010.

CHEW, E. **Linux kernel git commit**: pipe.c null pointer deference. Outubro, 2009. Disponível em: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=ad3960243e55320d74195fb85c975e0a8cc4466c. Acesso em: junho 2010.

CHRISTEY, S. **PLOVER**. Março, 2006. Disponível em: http://cwe.mitre.org/documents/sources/PLOVER.pdf. Acesso em: junho 2010.

COMMITTEE, P. I. T. A. **Cyber Security**: a crisis of prioritization. Disponível em: http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf. Acessado em: junho 2010.

COX, M. J. **Red Hat's Top 11 Most Serious Flaw Types for 2009**. Feveiro, 2010. Disponível em: http://www.awe.com/mark/blog/20100216.html . Acesso em: junho 2010.

CUNNINGHAM, J. S. **Eight Years of Linux Kernel Vulnerable**. Agosto, 2009. Disponível em: http://www.osnews.com/story/21993/Eight_Years_of_Linux_Kernel_Vulnerable. Acesso em: junho 2010.

CVE. **CVE FAQ**. Janeiro, 2010. Disponível em: http://cve.mitre.org/about/faqs.html. Acesso em: maio 2010.

CWE. **About CWE**. Setembro, 2007. Disponível em: http://cwe.mitre.org/about/index.html. Acessado em: junho 2010.

CWE. **Process**. Agosto, 2009. Disponível em: http://cwe.mitre.org/about/process.html. Acessado em: junho 2010.

DHANJANI, N. **Hacking**: the next generation. [S.l.]: O'Reilly, 2009.

DOWD, M. Application-Specific Attacks: leveraging the actionscript virtual machine. **IBM Global Technology Services**, [S.l.], p.25, 2008.

FLORIAN, C. **Vulnerability Related Standards**. Outubro 2009. Disponível em: http://www.gfi.com/blog/vulnerability-related-standards/. Acesso em: maio 2010.

FURLAN, L. H. **Estudo sobre Estouros de Buffer**. 2005. Trabalho de conclusão de curso — Instituto de Informártica da UFRGS.

GODEFROID, P. Automated Whitebox Fuzz Testing. **Network Distributed Security Symposium (NDSS)**, [S.l.], p.16, 2008.

GRéGIO, A. R. A. **Um Estudo sobre Taxonomias de Vulnerabilidades**. Disponível em http://mtc-m18.sid.inpe.br/dpi.inpe.br/hermes2@1905/2005/10.04.04.11. Acessado em: Junho 2010.

GRéGIO, A. R. A. Taxonomias de Vulnerabilidades: situação atual. **V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**, [S.l.], 2005.

HARRIS, S. **Gray Hat Hacking**: the ethical hacker's handbook. [S.l.]: McGraw, 2008.

HERTZFELD, A. **Folklore story about Monkey**. Outubro, 1983. Disponível em: www.folklore.org/StoryView.py?story=Monkey_Lives.txt. Acesso em: junho 2010.

HOGLUND, G. **Exploiting Software**: how to break code. [S.l.]: Addison-Wesley Professional, 2004.

HOLANDA FERREIRA, A. B. de. **Novo Dicionário Aurélio da Língua portuguesa**. 1ª.ed. [S.l.: s.n.], 1975.

JACK, B. Vector Rewrite Attack: exploitable null pointer vulnerabilities on arm and xscale architectures. **Juniper**, [S.l.], 2007.

LAMETER, C. **Linux kernel git commit**: use mmap_min_addr independently of security models. Junho, 2009. Disponível em: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=e0a94c2a63f26448260690446496 69b5e7ca75d3. Acessado em: Junho 2010.

LOVE, R. **Linux System Programming**. 1ª.ed. [S.l.]: O'Reilly Media, Inc., 2007.

MANN, D. E. Towards a Common Enumeration of Vulnerabilities. **The MITRE Corporation**, [S.l.], 1999.

MARTIN, R. A. **The Vulnerabilities of Developing on the Net**. Janeiro, 2001. Disponível em: http://cve.mitre.org/docs/docs-2001/DevelopingOnNet.html. Acessado em Junho 2010.

MARTINS, H. G. **Estudo sobre a exploração de vulnerabilidades via estouros de buffer, sobre mecanismos de proteção e suas fraquezas**. 2009. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informártica da UFRGS.

MCGRAW, G. **Software Security**: building security in. [S.l.: s.n.], 2006.

MELL, P. **CVSS**: a complete guide to the common vulnerability scoring system version 2.0. Disponível em: http://www.first.org/cvss/cvss-guide.pdf. Acessado em: Junho 2010.

MEUNIER, P. **Classes of Vulnerabilities and Attacks**. [S.l.]: Wiley Handbook of Science and Technology for Homeland Security, 2006.

RATANAWORABHAN, P. **Nozzle**: a defense against heap-spraying code injection attacks. Novembro, 2008. Disponível em: http://research.microsoft.com/pubs/76528/tr-2008-176.pdf. Acessado em: Junho 2010.

SANTOS BRANDãO, A. J. dos. O Uso de Ontologia em Alertas de Vulnerabilidades. **IV Workshop em Segurança de Sistemas Computacionais**, [S.l.], 2004.

SEACORD, R. C. **A structured approach to classifying security vulnerabilities**. [S.l.]: CMU/SEI, 2005.

SECURE SOFTWARE, I. **The CLASP Application Security Process**. Janeiro, 2006. Disponível em: http://searchappsecurity.techtarget.com/searchAppSecurity/downloads/clasp_v20.pdf . Acessado em: Junho 2010.

TAKANEM, A. **Fuzzing for Software Security Testing and Quality Assurance**. [S.l.]: Artech House, INC., 2008.

TINNES, J. **Linux NULL pointer deference due to incorrect proto_ops initializations**. Agosto, 2009. Disponível em: http://blog.cr0.org/2009/08/linux-null-pointer-dereference-due-to.html. Acesso em: maio 2010.

TSIPENYUK, K. Seven Pernicious Kingdoms: a taxonomy of software security errors. **NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics**, [S.l.], 2005.

APÊNDICE A EQUAÇÕES CVSS 2.0

A.1 Equações do escore básico

$$EscoreBasico = ((0.6 * Impacto) + (0.4 * Explorabilidade) - 1.5) * algo \quad \text{(A.1)}$$

$$Impacto = 10.41 * (1 - (1 - ImactoConf) * (1 - ImpactoInt) * (1 - ImpactoDisp)) \quad \text{(A.2)}$$

$$Explorabilidade = 20 * VetorAcesso * ComplexidadeAcesso * NecessidadeAut \quad \text{(A.3)}$$

A.2 Equações do escore temporal

Utiliza o escore básico.

$$EscoreTemporal = EscoreBasico * FacExploracao * NivelRemed * ConfReport \quad \text{(A.4)}$$

A.3 Equações do escore ambiental

Utiliza o escore temporal.

$$EscoreTemporal = EscoreBasico * FacExploracao * NivelRemed * ConfReport \quad \text{(A.5)}$$

Métricas básicas		
Métrica	**Valor nominal**	**Valor numérico**
Vetor de acesso	local	0.395
	rede adjacente	0.646
	rede	1.0
Complexidade de acesso	alta	0.35
	média	0.61
	baixa	0.71
Necessidade de autenticação	várias	0.45
	uma	0.56
	nenhuma	0.704
Impacto na confidencialidade	nenhum	0.0
	parcial	0.275
	completo	0.660
Impacto na integridade	nenhum	0.0
	parcial	0.275
	completo	0.660
Impacto na disponibilidade	nenhum	0.0
	parcial	0.275
	completo	0.660

Métricas temporais		
Métrica	**Valor nominal**	**Valor numérico**
Facilidade de exploração	não comprovada	0.85
	prova de conceito	0.9
	funcional	0.95
	alta	1.0
	não definida	1.0
Nível de remediação	conserto definitivo	0.87
	conserto temporário	0.90
	workaround	0.95
	indisponível	1.0
	não definido	1.0
Confiabilidade no *report*	não confirmada	0.9
	não corroborada	0.95
	confirmada	1.0
	não disponível	1.0

Métricas ambientais		
Métrica	**Valor nominal**	**Valor numérico**
Dano colateral potencial	nenhum	0.0
	baixo	0.1
	baixo-médio	0.3
	médio-alto	0.4
	alto	0.5
	não definido	0
Abundância de alvos	nenhuma	0
	baixa	0.25
	média	0.75
	alta	1.0
	não definida	1.0
Importância da confiabilidade	baixa	0.5
	média	1.0
	alta	1.51
	não definida	1.0
Importância da integridade	baixa	0.5
	média	1.0
	alta	1.51
	não definida	1.0
Importância da disponibilidade	baixa	0.5
	média	1.0
	alta	1.51
	não definida	1.0