FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Segurança da Informação

Ivan Edson dos Santos Junior

SEGURANÇA EM SOFTWARE

Americana, SP
2013

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Tecnologia em Segurança da Informação

Ivan Edson dos Santos Junior

SEGURANÇA EM SOFTWARE

Trabalho monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação da Fatec Americana, sob orientação do Prof. Me. Rossano Pablo Pinto.
Área de concentração: Segurança da Informação.

Americana, SP
2013

Ivan Edson dos Santos Junior

SEGURANÇA EM SOFTWARE

Trabalho de conclusão de curso apresentado à Faculdade de Tecnologia de Americana como parte dos requisitos para obtenção do título de Tecnólogo em Segurança da Informação.
Área de concentração: Segurança da Informação.

Americana, 04 de dezembro de 2013.

Banca Examinadora:

_____
Rossano Pablo Pinto (Presidente)
Mestre
Fatec Americana

_____
Gabriel de Souza Fedel (Membro)
Mestre
Fatec Americana

_____
Rogério Nunes de Freitas (Membro)
Especialista
Fatec Americana

AGRADECIMENTOS

Ao professor Rossano Pablo Pinto, pela orientação, compreensão, atenção, suporte no aprendizado e incentivo ao desenvolvimento deste trabalho.

A Anja Ariel pela ajuda, compreensão e motivação.

A Carolina dos Santos pela ajuda e incentivo.

A You Chwen pelo companheirismo e incentivo.

DEDICATÓRIA

À minha família, pai, mãe e irmã que incentivaram, à minha amada pela ajuda, compreensão, conforto e motivação, aos meus amigos pela ajuda e incentivo, e a todos que acreditaram e transmitiram sua energia positiva.

RESUMO

O presente texto diz respeito à área de segurança da informação, enfatizando o ramo de segurança em *software*. O objetivo principal deste trabalho é apresentar uma visão geral sobre algumas ameaças, proteções e prevenções que abrangem o referido tema. Em relação a isso, vale ressaltar o grau de importância da segurança em *software* perante a interconectividade atual de sistemas e a relevância da informação, tanto de organizações quanto de usuários. O tema abordado, de forma geral, é válido a qualquer tipo de *software*: local – em um sistema operacional – e remoto – em uma aplicação distribuída em rede. O motivo de empregar segurança no desenvolvimento de um *software* é evitar a introdução de vulnerabilidades que podem acarretar impactos futuros, tais como no sistema/ambiente, em outros ativos envolvidos, e até mesmo na própria aplicação. De fato, um programa com determinadas vulnerabilidades expostas pode ser a raiz do problema. Portanto, com base nas teorias pesquisadas, a importância e a necessidade de empregar desenvolvimento seguro evita estas fraquezas antecipadamente.

Palavras Chave: segurança em *software*; vulnerabilidades; desenvolvimento seguro.

ABSTRACT

This monograph concerns about the information security, emphasizing the security software area. The main objective of this paper is to present an overview of some threats, protections and preventions which comprehends this related topic. Thus, it is worth noting the software security importance before the current system interconnectivity and the information relevance, rather organizations or users. The topic is generally legitimated to any type of software: local - in an operating system - and remote - in a distributed application network. The reason to employ security for developing software is to avoid vulnerability introduction that can lead to future impacts, such as through the system / environment in other uncommitted assets, and even to the application itself. In fact, a certain exposed vulnerability program may be the root of the problem. Therefore, based on some researched theories, the importance and necessity of employing safe development avoids these weaknesses in advance.

Keywords: *security software; vulnerability; secure development.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ASLR: *Address space layout randomization*

BD: Banco de dados

bss: *Block started by symbol*

CC: *Common Criteria*

DMCA: *Digital millennium copyright act*

EULA: *End-user licence agreement*

gcc: GNU *compiler collection*

GNU: *Gnu is not* Unix

HTML: *Hyper text markup language*

ISO: *International standardization organization*

LIFO: *Last-in, first-out*

OpenSAMM: *Open Software assurance maturity model*

OSDEV: *Operating system development*

OWASP: *Open web application security project*

SDL: *Secure development lifecycle*

SI: Segurança da informação

SO: Sistema operacional

TI: Tecnologia da informação

XP: *Extreme programming*

SUMÁRIO

1 Introdução

Atualmente, estamos vivenciando uma era da tecnologia bastante acelerada e em processo de constante atualização. Em meio a isto, esta alta velocidade de avanço, de certa forma, se torna essencial na vida das pessoas e também na existência de instituições em geral, proporcionando grandes passos direcionados ao auto desenvolvimento.

No ato de almejar ou exigir fatores como funcionalidade, facilidade de uso e principalmente rapidez, um detalhe muito importante que acaba ficando de fora é a segurança, às vezes julgada como incômodo e algo sem importância. Em meio a isto, é evidente que a informação deve ser processada sem falhas e sem ser danificada ou acessada por um sujeito malicioso. Segurança em *software* carrega uma parcela relacionada à maneira segura em que as informações devem ser processadas, além do objetivo mais temido que é conseguir a segurança necessária para proteger dados e sistemas em geral.

O objetivo principal é apresentar uma visão geral sobre a importância da segurança em *software*, tomando como exemplo algumas ameaças ainda existentes, proteções e prevenções que estão inclusas no tema referido.

Os objetivos específicos deste trabalho consistem em: alertar os desenvolvedores sobre os perigos que a linguagem de programação pode oferecer, em especial a linguagem C, além de preveni-los para não confiarem totalmente em mecanismos de proteção dinâmica; apresentar o uso de uma ferramenta de análise estática de código e o uso de uma ferramenta de análise dinâmica; disponibilizar um mini *checklist* para contribuir com o desenvolvimento seguro.

O trabalho está estruturado em seis capítulos, sendo que o segundo aborda um levantamento bibliográfico sobre conceitos básicos para o entendimento a diante. Com base nas teorias capítulo 2, o capítulo 3 enfatiza vulnerabilidades e ataques mais amplamente, selecionando como exemplos algumas vulnerabilidades ainda existentes. Dados os conceitos até então, é possível compreender com mais clareza a necessidade de empregar proteções e boas práticas voltadas à segurança ao longo do processo de desenvolvimento, sendo justamente este o assunto do capítulo 4. A partir de todo este estudo sobre a segurança em aplicações, para contribuir com o a segurança em *software,* o capítulo 5 expõe dois testes com ferramentas que

detectam vulnerabilidades, além de apresentar uma *checklist* para contribuir minimamente com a programação segura. Por fim, no capítulo 6 são levantadas algumas considerações finais sobre o tema e o que foi tratado neste trabalho.

2 Revisão bibliográfica

Este capítulo apresentada uma série de conceitos computacionais e termos técnicos da tecnologia da informação (TI), em especial segurança da informação no processo de desenvolvimento de *software*.

O referido assunto é baseado na plataforma Linux, na organização padrão dos programas compilados com o compilador *GNU Compiler Collection* (gcc) versão 4.4.3, e na linguagem de programação C, embora também seja compatível com diversas outras plataformas e outros compiladores.

2.1 Sistema computacional

Um sistema computacional de acordo com Machado e Maia (2007) é constituído por uma série de dispositivos físicos (*hardware*, como por exemplo, teclado, monitor, mouse, processador, memória física, entre outros), e pela parte lógica (*softwares*, instruções, bloco de memórias, entre outros). Todos os dispositivos físicos manipulam informações, até suas limitações e capacidades que cada um oferece. Informações essas que se tornam legíveis aos humanos por meio dos mecanismos lógicos, ou seja, para que tenha utilidade, o *hardware* é dependente de *software*, e vice versa (TANENBAUM, 2010).

2.2 *Software*

Software, também conhecido como “programa de computador”, é composto por um conjunto de instruções programadas, que ao passar pelo processo de compilação, é convertido em linguagem de máquina (instruções binárias, “zeros e uns”), possibilitando sua operação e comunicação com os dispositivos físicos (*hardware*) e cumprindo suas funções de acordo com o que foi especificado, estando presente hoje praticamente em todos os dispositivos tecnológicos.

O autor Foster (2005) traz uma breve definição de *software*:

> “Um programa é uma coleção de comandos que pode ser compreendida por um sistema operacional. Os programas podem ser escritos em uma linguagem de alto nível, como Java ou C, ou em linguagem de montagem de baixo nível.”

Dependendo de quem o cria, para qual finalidade, ou para quem são destinados, os *softwares* podem possuir diferentes licenças para o uso. O uso do *software* proprietário (que também é conhecido como *software* de código fechado) pode ser gratuito, parcialmente gratuito, ou totalmente pago mediante uma licença estabelecida; Já o *software* livre, de código aberto (*Open source)*, além da disponibilidade de uso gratuito, suas funcionalidades originais podem ser modificadas a gosto de quem o utiliza (DEITEL; DEITEL; CHOFFNES, 2005).

2.2.1 Processo de criação de um programa

Na construção de *softwares* é primordial o uso de linguagens de programação para que as instruções sejam descritas passo a passo segundo um algoritmo. Normalmente são sujeitas a determinadas regras e sintaxes variando de linguagem para linguagem. Em geral, estas linguagens de programação podem ser classificadas como de alto, médio ou baixo nível. Quanto mais alto o nível, mais fácil é a compreensão de leitura humana, porém, quanto mais baixo, se torna mais ilegível ao ser humano e mais próximo de como um computador realmente lê e processa as informações (no caso da linguagem de montagem *assembly*), segundo os autores Deitel, Deitel e Choffnes (2005). Para que haja esta comunicação com a máquina, é necessário que a linguagem de programação/montagem passe por uma espécie de tradução até se tornarem elegíveis a serem processadas.

Conforme a Figura 1, cada linguagem possuem seus compiladores apropriados, que por finalidade irão traduzir o código fonte (sintaxe, dados e endereços) em arquivo binário, também conhecido como arquivo objeto. Estando pronto, este arquivo objeto, será manipulado por um ligador (*linker*) na seguinte etapa. Entretanto, em alguns sistemas o compilador pode realizar além de suas funções básicas também as tarefas de um ligador (JANDL, 1999).

O *linker*, na definição do mesmo autor Jandl (1999), basicamente é o responsável por fazer a ligação de chamadas para um ou diversos módulos

(subprogramas independentes), ou seja, fazer referência de bibliotecas ao arquivo objeto e por fim gerar o arquivo executável. Além disso, os módulos também determinam qual a região de memória que o programa será carregar para sua execução. Os módulos, seguindo o exemplo do autor Jandl (1999) podem ser funções e rotinas já existentes, pré-compiladas e contidas em bibliotecas utilizadas para a manipulação de entrada e saída de informações ao programa. O seguinte passo será responsabilidade de uma entidade denominada de carregador.

Ainda segundo o autor Jandl (1999), os carregadores normalmente fazem parte do Sistema Operacional por terem o papel de transportar e preparar os arquivos já gerados até agora (armazenados em memória secundaria) para a memória primária, para assim entrarem em processamento.

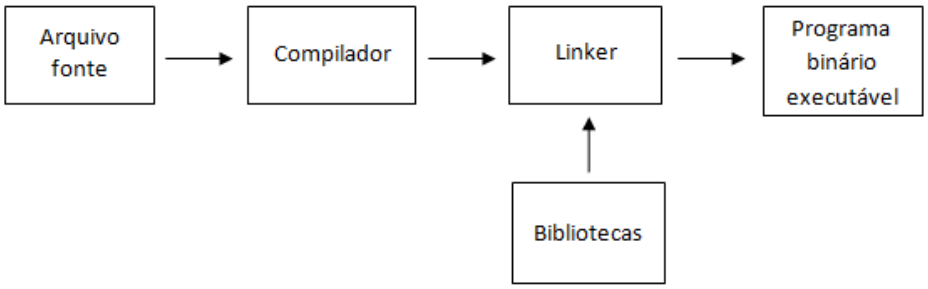

Figura 1 - Representação básica da criação de um executável. Adaptada de Jandl (1999).

2.2.2 Execução e organização na memória

Após o entendimento de como um programa é criado vale ressaltar a sua execução no sistema. Deitel, Deitel e Choffnes (2005), definem que um programa é uma entidade inanimada em um sistema, e a partir do momento em que ele começa a ser executado ele se torna uma entidade ativa conhecida como processo.

No processo de execução, o *software* (que passará a ser mais um processo no sistema), é primeiramente carregado na memória física pelo sistema operacional, que em seguida irá organizá-lo em blocos de memória - também conhecidos como segmentos - (JANDL, 1999). Estes segmentos normalmente são a região de código, dados, *bss*, *heap* e pilha (*text*, *data*, *bss*, *heap* e *stack*), conforme Figura 2.

O segmento de código armazena o próprio código em binário do programa, instruções que serão executadas pelo processador de forma não linear (DEITEL;

DEITEL; CHOFFNES, 2005). A permissão de escrita neste segmento é desabilitada, portanto, seu tamanho e conteúdo são estáticos.

Em sequência, o segmento de dados, armazena variáveis globais, estáticas e a memória alocada dinamicamente que venha a ser útil no processo de execução. Apesar de ser um segmento com permissão de escrita, ele também possui um tamanho fixo. Assim como o segmento de dados, o próximo segmento, bss, também é utilizado para armazenar variáveis estáticas e globais, porém, quando ainda não foram inicializadas. O bss possui escrita habilitada e seu tamanho é fixo (ERICKSON, 2008).

O quarto segmento é o de *heap*, utilizada para alocações e liberação de reservas na memória, sendo diretamente controlada pelo programador. Seus blocos reservados de memória podem ser utilizados para qualquer necessidade do processo. Seu tamanho não é fixo, variando de acordo com a necessidade. A *heap* cresce para baixo avançando em direção aos endereços de memória mais altos (ERICKSON, 2008).

O último segmento é a pilha (*stack*), a área de memória que armazena as variáveis locais e os endereços para chamadas (decisão do fluxo de execução após a invocação de uma função). Seu tamanho é dinâmico e segue uma ordem LIFO (*last-in, first-out*), ou seja, o último item alocado na pilha é o primeiro a ser liberado. A pilha cresce para cima e conforme aumenta, ela tende seguir em direção aos endereços mais baixos do processo (ERICKSON, 2008).

Cada um dos segmentos possui um tamanho independente dos outros segmentos. Com isso, a alocação de cada processo pode aumentar ou diminuir de acordo com suas necessidades. Contudo, os segmentos facilitam e simplificam a organização da estrutura de dados ainda mais quando se trata de variações em tempo de execução. Outra vantagem obtida neste modelo é a possibilidade de proteção da memória quando se trata de compartilhamento (JANDL, 1999). Alguns desses segmentos, com certas vulnerabilidades, podem sofrer ataques. Este trabalho aborda sobre vulnerabilidade de *buffer overflow* presentes nos dois últimos segmentos: *heap* e pilha, seção 3.2.2 e seção 3.2.3, respectivamente. Na próxima seção, é exposto uma visão geral sobre vulnerabilidades em *software*.

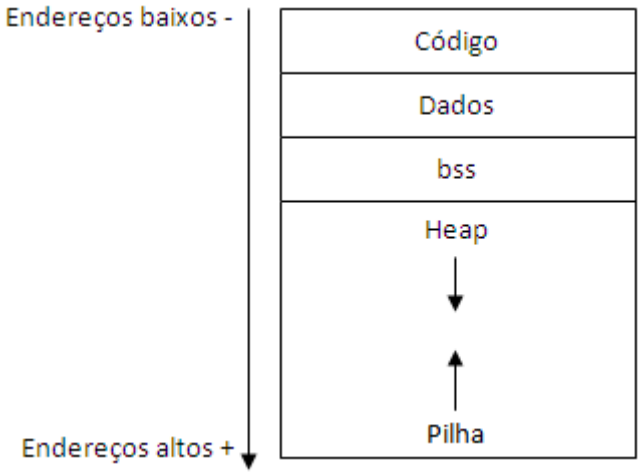

Figura 2 - Representação da organização dos segmentos de memória.

.

2.3 Vulnerabilidades de *software*

Uma vulnerabilidade pode ser entendida como um defeito de sistema (presente em qualquer ramo de TI) relevante a segurança, ou seja, um ponto fraco que pode ser explorado por algo ou alguém com o objetivo de violar as regras da política de segurança estabelecida (CORREIA; SOUSA, 2008).

Dowd, McDonald e Schuh (2006) apresentam uma breve definição de vulnerabilidade:

> "[...] as vulnerabilidades são falhas específicas ou descuidos em uma parte do software que permitem que atacantes possam realizar algo malicioso, expor ou alterar informações sensíveis, interromper ou destruir um sistema, ou assumir o controle de um sistema computacional ou programa".

Para os autores, Correia e Sousa (2008), em torno de um alvo, sempre haverá pontos vulneráveis, não só no *software* em si ou nos elementos inclusos na superfície de ataque (como visto na seção 2.5.1), mas também outras entidades que dependam do programa. Como exemplo tomado pelo autor Peixoto (2006): os usuários com falta de educação e cultura à segurança sempre são vulneráveis a engenharia social.

Atualmente, existem inúmeras vulnerabilidades, e como breve exemplo de algumas populares, citadas por *Open Web Application Security Project* (OWASP)[1], podem ser a falta de verificação de limites de alocação de memória (seção 3.2.2),

[1] OWASP. Disponível em: <https://www.owasp.org/index.php/Category:Vulnerability>. Acesso em: 04 ago 2013.

falta de validação na entrada de dados (seção 3.2.6), falha na abertura de arquivos (ou más manipulações), finalização incorreta de uma conexão com um banco de dados, entre muitas outras.

Apesar das vulnerabilidades poderem vir a existir por meio de falhas, descuidos, erros e *bugs*, no ramo de segurança de *software*, existem diversos modelos para auxiliar a classificação de tipo de vulnerabilidades. Porém, não há uma taxonomia simples e determinada a ser seguida por padrão internacional. Mas quando utilizado tais modelos existentes, podem oferecer vantagens como organização, compreensão e comunicação dos problemas (DOWD; MCDONALD; SCHUH, 2006). Determinadas vulnerabilidades podem se encaixar em mais de uma classe. O capítulo 3 aborda brevemente sobre três tipos (CORREIA; SOUSA, 2008): vulnerabilidade de projeto, vulnerabilidade operacional e vulnerabilidade de programação, com ênfase nesta última.

As vulnerabilidades encontradas em um *software* devem receber respostas, as eliminando ou reduzindo até um nível de segurança aceitável. Para isto, é aplicada e distribuída correções para o *software* vulnerável.

2.4 Correções de falhas

Vulnerabilidades existentes em um *software* estão sempre sujeitas a serem descobertas, ainda mais quando se trata de *softwares* mais populares. Quando é descoberta alguma vulnerabilidade no *software* produzido por determinada empresa, é criada uma correção (*patch*) e em seguida disponibilizada para que seus utilizadores a instalem, a fim de remover a falha. Apesar disso, a tática de disponibilizar atualizações e o próprio usuário ter que permitir a instalação não obtém tanto sucesso devido à falta de compreensão, descuido ou ignorância dos usuários sobre a segurança, e consequentemente muitos desses usuários ou administradores de sistemas não instalam as devidas correções (CORREIA; SOUSA, 2008).

Em meio a isso, outro problema pode presenciar na própria atualização quando instalada, pois, a correção de determinado erro pode estar sujeita em gerar outros problemas, como instabilidade no sistema ou até mesmo inserção de novas vulnerabilidades. No entanto, quando *patches* de segurança são ignorados, os

sistemas permanecem com vulnerabilidades já conhecidas, o que é grave, pois a probabilidade de ocorrer um ataque é ainda maior devido à disseminação de informações na *internet* a respeito da mesma (CORREIA; SOUSA, 2008).

Entretanto, o fato de uma vulnerabilidade ser desconhecida da comunidade de segurança da informação, não quer dizer que não seja de conhecimento especifico por alguém ou algum meio restrito, como por exemplo, um grupo de *crackers* (CORREIA; SOUSA, 2008). Tal tipo de vulnerabilidade também é conhecido como vulnerabilidade "*zero-day*" (FOSTER, 2005).

2.5 Ataques

Em meio aos defeitos de sistemas, em destaque os que geram algum tipo de vulnerabilidade, incidentes podem ocorrer, afetando os princípios de segurança da informação. Tal incidente pode ser resultado de um ataque. Um ataque é uma ação maliciosa que, explorando determinada vulnerabilidade, tem a visão de contornar os mecanismos de segurança e se beneficiar de alguma forma no sistema, seja capturando, modificando ou excluindo informações de forma não autorizada (CORREIA; SOUSA, 2008).

Shirey (2008) define o que pode ser um ataque aos sistemas computacionais:

> "*An assault on system security that derives from an intelligent threat, i.e., an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system*."

Entretanto, os ataques podem ser executados manualmente por alguém, ou por meio de um processo automatizado através de programas maliciosos destinados a exploração, conhecidos como *exploits* (FOSTER, 2005).

2.5.1 Superfície de ataque

Em segurança de *softwares*, existe o conceito de superfície de ataque, que é a interface que pode dar origem a um ataque contra o aplicativo alvo (CORREIA; SOUSA, 2008). Pode ser representada em três principais camadas, como ilustrado na Figura 3.

A primeira camada (externa) tem relação com a infraestrutura lógica como, por exemplo, a rede de computadores, interfaces utilitárias, sistema de arquivos e outros *softwares* no sistema. Esses meios podem ser atacados quando possuem alguma vulnerabilidade e abrir caminho à camada seguinte ou diretamente ao *software* alvo. A segunda camada é considerada o sistema operacional que provê suporte básico ao funcionamento do "*software* alvo", possuindo ligações com a camada mais externa. O sistema operacional também pode sofrer ataques abrindo caminho até a aplicação alvo. Por fim, no centro, a terceira camada é representada pelo *software* que pode vir a ser o alvo mediante a alguma vulnerabilidade que ele possua, ou gerada por ataques das camadas à volta (CORREIA; SOUSA, 2008).

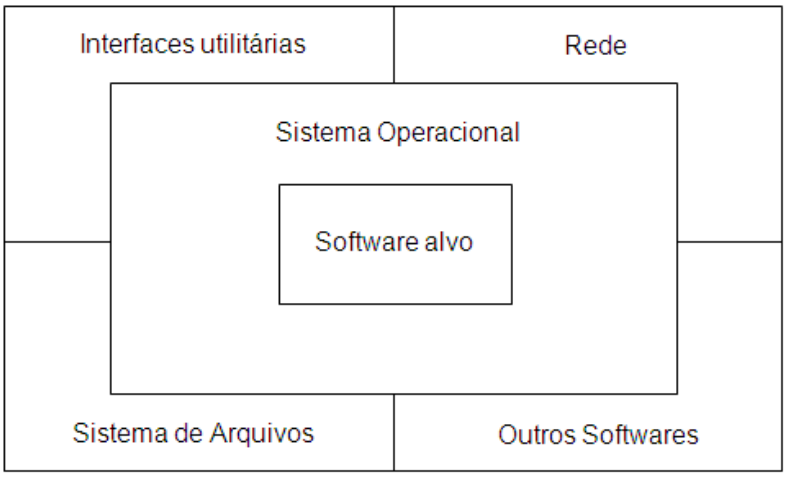

Figura 3 - Representação da superfície de ataque.
Adaptação de Whittaker e Thompson (2004) apud Correia e Sousa (2008).

2.6 Segurança da informação

Para Peixoto (2006), a informação pode ser vista como o bem mais precioso das empresas, já que hoje sem isto nenhum negócio de nenhuma empresa consegue existir e caminhar, ou seja, estão dependentes da informação. Em meio a isto, há casos em que a informação é confidencial, não podendo ser modificada, excluída, ou simplesmente acessada por qualquer pessoa não autorizada, pois, se dispostas a qualquer um, pode haver um comprometimento do negócio, gerando prejuízos em escalas variantes de baixo a catastróficos impactos.

Além disso, não só o ramo corporativo necessita de precauções, mas também outras instituições como, por exemplo, governamentais, de ensino e até mesmo

usuários domésticos. Entretanto, devido a esta necessidade de proteção surgiu o ramo de segurança da informação, com uma ampla abrangência envolvendo *hardware*, sistemas operacionais, redes, *softwares,* banco de dados (BD), entre outros. Portanto, além da informação, a segurança da informação também possui foco em ativos de TI, levando em conta três conceitos básicos interligados (PEIXOTO, 2006):

- **Confidencialidade**, que visa tornar a informação algo confidencial, sem que ela possa sofrer danos ou acesso não autorizado;
- **Integridade**, tornando as informações íntegras, com a certeza de que ao serem solicitadas (por sujeitos autorizados) estejam em estado legível para o entendimento esperado, e sem alteração;
- **Disponibilidade**, garantindo a disposição das informações, com meios seguros em que pessoas certas possam ter acesso, no momento desejado e permitido.

Contudo, os diversos ambientes que dependem da informação e de seus ativos, podem contar com sua própria política de segurança da informação, contendo normas e regras que devem ser seguidas de acordo com a necessidade e anseio de proteção.

Com os conceitos anteriores sobre vulnerabilidade, ataque, e necessidade de proteção da informação, a seção 2.6.1 apresenta a importância de segurança nas aplicações.

2.6.1 Segurança em *software*

Com a difusão da internet pelo mundo, em todos os meios tecnológicos o *software* está presente fazendo a diferença, tornando o sistema ao todo operável de várias maneiras (VIEGA; MCGRAW, 2001). A segurança no *software* é um requisito fundamental para os fornecedores de *software* por se tratar de uma necessidade crítica proteger e preservar a confiança do ambiente computacional.

Os autores Viega e McGraw (2001) nos trazem um bom parâmetro para refletir melhor sobre o tema deste trabalho:

> “Não teríamos que gastar tanto tempo, dinheiro e esforço em segurança de redes se não tivéssemos uma tão má segurança em *software*.”

O desenvolvimento de sistemas é considerado, segundo Mariotti (2004), uma das áreas mais afetadas em segurança devido a erros na produção ou de arquiteturas. Muitas vezes, a pressa de desenvolver o *software* o quanto antes acaba comprometendo a segurança do próprio, deixando-a de lado. Poucos analistas se preocupam em especificar com mais detalhe os requisitos de segurança. A segurança nas aplicações sempre foi importante, e em meio à vasta conectividade mundial necessita de mais atenção, pois a raiz de toda a funcionalidade por traz disso é o *software*. Por exemplo, falhas de segurança em um cenário de sistemas interconectados podem perfeitamente facilitar acessos remotos indevidos. Hoglund e McGraw (2004) nos alertam: “A invasão de uma máquina quase sempre envolve a exploração de *software*.”

Contudo, a segurança nas aplicações é algo delicado por ser possível executar outros procedimentos que não foram especificados originalmente no *software* (MARIOTTI, 2004). Portanto, segundo Hoglund e McGraw (2004), o *software* pode ser considerado como "calcanhar de aquiles", “a raiz do problema”, ou ainda “*software* defeituoso é onipresente”.

2.6.2 Segurança proprietária e de código aberto

Em um comparativo a estas duas linhas de desenvolvimento, os autores Deitel, Deitel e Choffnes (2005) apresentam uma visão geral sobre o assunto. As soluções proprietárias garantem apenas segurança por obscuridade (superficial/simples/insuficiente). Seu código fonte não é disponibilizado e por conta dessa limitação, o número de usuários colaborativos que procuram por falhas de segurança é reduzido, ao contrário de códigos fontes abertos. Os *softwares* proprietários ao serem concebidos, são testados somente pela equipe privada, e nisso, sempre haverá falhas que passam despercebidas.

Os mesmos autores, Deitel, Deitel, Choffnes (2005) argumentam que uma vantagem primordial de código aberto é a possibilidade de modificação e adaptação. Usuários com conhecimento podem personalizar o nível de segurança como desejarem. Muito útil para administradores de sistemas que desejam adaptar

determinadas ações do *software* com a política de segurança da entidade. Já os usuários de *software* proprietário podem ficar limitados somente a funcionalidades que o fabricante oferece no produto.

Na condição de utilizar *software* proprietário, também há os riscos do término de suporte ao cliente, como exemplo, o fim de suporte ao sistema operacional Windows XP da Microsoft[2], não havendo mais atualizações automáticas para proteger o usuário/entidade das vulnerabilidades que vão sendo descobertas neste sistema.

Uma situação polêmica envolvendo *software* proprietário, é a questão do *software* estar sujeito a leis do país, ao contrário do *software* livre. Nisso, perante leis, questões de segurança que envolve este *software* (como a descoberta de novas vulnerabilidades) pode estar dependente de organizações maiores, como por exemplo, a situação da Microsoft perante a NSA (*National Security Agency*). Quando descoberta uma nova vulnerabilidade em sistemas Microsoft, esta informação é primeiramente encaminhada a NSA para fins de uso da vulnerabilidade, ou caso não for útil a eles, são publicadas correções de segurança para a mesma.

Por fim Deitel, Deitel e Choffnes (2005) ressaltam que cada método pode oferecer armadilhas e méritos, e exigem atenção em segurança. Contudo, Correia e Sousa (2008) reforçam que sempre há vulnerabilidades descobertas em *software* proprietário e aberto (como exemplo, as vulnerabilidades diárias publicadas no site *National Vulnerability Database*[3]), e independente da disposição, o que faz diferença no desenvolvimento seguro está na qualidade e atenção de toda a equipe/comunidade em relação à segurança que deve ser aplicada desde o início do projeto até sua conclusão em diante.

Com base no levantamento das informações deste capítulo, o capítulo 3 irá se especificar em vulnerabilidades e fraquezas que existem por conta de descuidos.

[2] MICROSOFT. Disponível em: <http://www.microsoft.com/business/pt-pt/a-par-e-passo/paginas/fimsuporte.aspx>. Acesso em: 10 dez 2013.
[3] NIST. Disponível em: <http://nvd.nist.gov/>. Acesso em: 10 dez 2013.

3 Vulnerabilidades

Este capítulo aborda um assunto mais específico sobre algumas vulnerabilidades, contando com exemplos explicativos de vulnerabilidades e breves descrições de formas de ataque ou consequências, entre outros detalhes. As vulnerabilidades selecionadas para este capítulo prevalecem até hoje no contexto de desenvolvimento de aplicações.

3.1 Tipos de vulnerabilidades

Como mencionado na seção 2.3, devido à grande variedade de vulnerabilidades existentes hoje, é possível classificá-las de acordo com alguns modelos (KUNST; RIBEIRO, 2005), porém nesta monografia, de forma geral, não é o foco classificar vulnerabilidades, portanto, é apresentado brevemente três possíveis classificações gerais: vulnerabilidade de projeto, de programação e operacional. A ênfase neste capitulo é em vulnerabilidade criada na programação.

A vulnerabilidade de projeto pode ter origem durante a construção do projeto para o *software*. Como um exemplo, a possibilidade de utilizar um mecanismo de autenticação fraco que possibilite os usuários escolherem senhas fáceis ou muito curtas, ou não ser considerado a existência de espionagem na rede em que trafegam as informações, assim estando sujeitas a captura. O tipo de vulnerabilidade de programação é introduzida durante a concretização do sistema, ou seja, quando está sendo desenvolvido o que foi levantado no projeto. Descuidos durante a programação contribuem para a introdução de fraquezas na aplicação. Já a vulnerabilidade operacional pode ser causada pelo ambiente no qual o sistema (*software*) é executado ou por meio de más configurações. Isso ocorre durante o tempo de execução, como problemas de gerenciamento de memória, conflitos no SO, entre outros (CORREIA; SOUSA, 2008).

3.2 Vulnerabilidades em programação

As demonstrações nesta seção têm base na linguagem de programação C. Uma linguagem de alto nível, tendo como característica sua boa velocidade e eficiência, visando simplicidade. O autor Erickson (2008) explica que estas características são possíveis graças à falta de checagem de integridade dos dados. Caso contrário, se houvessem todas as checagens necessárias para a segurança, o desempenho do programa gerado seria mais carregado que o normal. O programador que utiliza a linguagem C deve estar ciente desses descuidos, do quanto esta linguagem por padrão pode facilmente resultar em programas vulneráveis. Portanto, o próprio programador deve se informar e aprimorar suas técnicas de segurança para contornar e evitar a introdução de vulnerabilidades.

Segundo relatórios da empresa TIOBE Software[4], atualmente C é a linguagem de programação mais utilizada no mundo. No entanto, além de desempenho rápido, outro ponto que justifica a grande utilidade da linguagem C é a possibilidade de interação com o baixo nível (por exemplo, *assembly*), provendo funcionalidades diferenciais[5], ao contrário de outras linguagens que não permitem.

Para Correia e Sousa (2008), estima-se que qualquer pacote de *software* possui entre cinco a cinquenta *bugs* a cada mil linhas e código. Em meio a isso, alguns sendo vulnerabilidades. Portanto, quanto mais rigoroso os métodos para projetar e concretizar o *software* menos *bugs* ele terá.

3.2.1 *Buffer overflow*

Uma das vulnerabilidades mais clássicas da programação, tomada sempre como parâmetro, possuindo diversos níveis de risco, ainda existente e provavelmente permanente para muitos anos, segundo Hoglund e McGraw (2004). O *buffer overflow* ocorre quando é armazenada uma quantidade de dados maior do que a capacidade de um espaço de memória (*buffer*). Este problema é bem comum em programas escritos em linguagem C/C++ pela falta de verificação de limites

[4] TIOBE Software. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 07 dez 2013.
[5] WIKIQUOTE. "With great power comes great responsibility". Disponível em: <http://en.wikiquote.org/wiki/Stan_Lee>. Acesso em: 09 dez 2013.

antes de armazenar conteúdo em *buffer*, ao contrário de linguagens mais robustas como Java ou C# - linguagens estas que ainda assim não são totalmente imunes a esta vulnerabilidade (HOGLUND; MCGRAW, 2004).

Ataques destinados a estas vulnerabilidades de *overflow*, dependendo, podem ser bem perigosas, possibilitando a execução de códigos arbitrários (com os mesmos privilégios do *software* atacado) modificando a forma em que o programa irá se comportar. Consequentemente, o ataque pode corromper ou sobrescrever dados já existentes, causar parada crítica do programa ou obter controle total do ambiente (DEITEL; DEITEL; CHOFFNES, 2005).

3.2.2 *Heap overflow*

O segmento da *heap*, por armazenar dados dinâmicos e alocações de memória, está sujeito a *buffer overflow*. A vulnerabilidade neste segmento pode existir devido ao tamanho mal calculado de *buffer*. Um *overflow* neste segmento pode substituir regiões de memória reservadas para determinados fins. Normalmente isso ocorre quando a variável do endereço mais baixo excede seu tamanho, passando a alocar a parte transbordada para o próximo *buffer* de endereço mais alto. Isso pode levar a violação de acesso. Em um ataque, o sujeito pode causar *overflow* intencionalmente e substituir locais de memória por instruções controladas (*shellcodes*). Certos tipos de ataques procuram substituir ponteiros de funções, fazendo-os apontarem para o código arbitrário inserido (OWASP)[6].

Com base em explicações dos autores Correia e Sousa (2008) a Figura 4 é um simples exemplo da inserção desta vulnerabilidade na *heap*. Nas linhas 3 e 4 é alocado memória em `reserva1` e `reserva2` respectivamente com a função `malloc`. Na linha 6 a função `printf` irá imprimir o endereço de memória em que se encontra `reserva1` e na linha 7 é feito o mesmo, porém, com `reserva2`. Estas reservas armazenam dados caracteres. A `reserva2` armazenará a palavra `"abacate"` especificada no próprio código (linha 9) e `reserva1` armazenará dados informados pelo usuário (linha 10). Após os armazenamentos, na linha 12 é

[6] OWASP. Disponível em: <https://www.owasp.org/index.php/Testing_for_Heap_Overflow>. Acesso em: 19 set 2013.

impresso o que `reserva1` contém, e na linha 13 o que `reserva2` contém. Neste programa as atribuições às duas reservas são realizadas por meio da vulnerável função `strcpy` (linha 9 e 10). A função `strcpy` da biblioteca C padrão não faz verificação de limites antes de copiar algum conteúdo ao *buffer* caso haja espaço suficiente para a cópia.

```
1  int main(int argc, char **argv) {
2
3  »       char *reserva1 = (char *) malloc(sizeof(char)*10);
4  »       char *reserva2 = (char *) malloc(sizeof(char)*10);
5
6  »       printf("Endereco de reserva1 = %p\n", reserva1);
7  »       printf("Endereco de reserva2 = %p\n", reserva2);
8
9  »       strcpy(reserva2, "abacate");
10 »       strcpy(reserva1, argv[1]);
11
12 »       printf("Valor de reserva1 = %s\n", reserva1);
13 »       printf("Valor de reserva2 = %s\n", reserva2);
14
15 }
```

Figura 4 - Trecho de código com vulnerabilidade de *heap overflow*. Adaptação de Correia e Sousa (2008).

A Figura 5 demonstra a saída normal do programa. Foi informado como argumento a palavra "`teste`" (linha 1), que, pelo seu tamanho (cinco caracteres), pôde ser alocada normalmente em `reserva1`. O conteúdo de `reserva2` permanece "`abacate`". Nas duas primeiras impressões do programa (linha 2 e 3), percebe-se que o endereço de memória de `reserva1` é menor que o endereço de `reserva2`. Como `reserva1` foi criada primeiro, possui um endereço mais baixo, e `reserva2` em seguida, possui um endereço mais alto. Se observarmos a diferença do maior endereço para o menor é de 0x10, que em decimal representa 16 *bytes.* Portanto, dezesseis caracteres é a quantidade normal que `reserva1` pode alocar, sendo também a distância até `reserva2`. Na linha 4 e 5 são impressos o conteúdo das reservas.

```
1 ./OverflowHeap.out teste
2 Endereco de reserva1 = 0x804a008
3 Endereco de reserva2 = 0x804a018
4 Valor de reserva1 = teste
5 Valor de reserva2 = abacate
```

Figura 5 - Execução do programa OverflowHeap.out com uma saída normal.

Com a informação de quantos caracteres `reserva1` pode comportar, se for inserido um argumento com mais de dezesseis caracteres o transbordamento dos dados irá sobrescrever `reserva2`. Como exemplo, na Figura 6 é apresentado o resultado de uma saída inesperada, o excesso de `reserva1` sobrescrevendo `reserva2`. Foi inserido primeiramente dezesseis caracteres quaisquer e juntamente o conteúdo que irá alterar `reserva2` (linha 1), alterando `"abacate"` para `"cajamanga"`, como visto na impressão (linha 5). Este simples exemplo apenas ocorre uma alteração imprevista de valores, podendo ser caracterizada como violação de acesso. Porém, é perfeitamente possível injetar códigos arbitrários (*shellcodes*) que modifiquem e controlem o funcionamento do programa, embora atacar não seja o foco deste trabalho.

```
1 ./OverflowHeap.out 1234567890123456cajamanga
2 Endereco de reserva1 = 0x804a008
3 Endereco de reserva2 = 0x804a018
4 Valor de reserva1 = 1234567890123456cajamanga
5 Valor de reserva2 = cajamanga
```

Figura 6 - Execução do programa OverflowHeap.out com uma saída inesperada.

3.2.3 *Stack overflow*

Stack overflow é a condição de *buffer overflow* no segmento de memória *stack*. Um dos principais motivos da existência de *stack overflow* é a má implantação de tratamento de *string* que se encontram nas bibliotecas C padrão, segundo Hoglund e McGraw (2001), e também pode ser resultado de erros de lógica.

Um ataque clássico é o *smashing stack* (esmagamento de pilha), que basicamente corrompe a pilha de execução. Como consequência apresentadas por OWASP[7], ataques em geral de *overflow* na pilha pode afetar a disponibilidade induzindo o programa a entrar em *loop* infinito, em estado de *deadlock*, ou obrigatoriamente ser encerrado, além da possibilidade de um sujeito poder controlar a funcionalidade do binário. Neste ataque visando controlar o programa a partir deste segmento, também pode ser explorado com a execução de códigos arbitrários

[7] OWASP. Disponível em: <https://www.owasp.org/index.php/Stack_overflow>. Acesso em: 20 set 2013.

injetados, sendo possível alterar o fluxo do programa para executar códigos ou funcionalidades fora do fluxo de execução.

Como exemplo de código vulnerável a isto, temos na Figura 7. Primeiramente, na linha 12 é impresso o endereço de memória da função `secreto`. A próxima linha faz chamada para a função `store`, passando como argumento o que o usuário informar. Na execução da função `store`, é criado um *buffer* de dez posições (linha 6) e é copiado para ele o conteúdo passado como argumento a função (linha 7), e o exibe (linha 8). A vulnerabilidade se encontra na linha 7, novamente, devido a falta de precauções da função `strcpy`. Observa-se que as três primeiras linhas fazem parte da função `secreto` que não é utilizada durante a execução do programa. Ela apenas está preparada para imprimir "`FRASE SECRETA!`", porém observa-se que não está no fluxo de execução normal do programa. Esta função está presente no código para exemplificar em teoria que um ataque a esta vulnerabilidade deste segmento pode alterar o fluxo normal de execução do programa, induzindo a execução da função `secreto`. Para injetar código em um *buffer* vulnerável, o tamanho deste código não pode exceder o tamanho ainda livre do *buffer* e a pilha (também no caso da *heap*), deve estar em modo executável (seção 4.4.1.3).

Embora o endereço de `secreto` seja impresso no exemplo, por meio de técnicas de engenharia reversa (seção 3.3) é possível conseguir informações de memória do binário.

```
1  void secreto() {
2    printf("\n\nFRASE SECRETA!\n\n");
3  }
4
5  void store(char *valor) {
6    char buf[10];
7    strcpy(buf, valor);
8    printf("Valor no buffer = %s\n", buf);
9  }
10
11 int main(int argc, char **argv) {
12   printf("Endereco da funcao secreto: %p\n", &secreto);
13   store(argv[1]);
14 }
```

Figura 7 - Trecho de código com vulnerabilidade *stack overflow*. Adaptação de Correia e Sousa (2008).

3.2.4 Erro *off-by-one*

Para Erickson (2008) este erro de programação (também conhecido como *fencepost)* é tratado como uma má interpretação de lógica por parte do programador. Ele ocorre em casos em que o programador confunde operadores de comparação ou quando faz estimativa errada por um dígito. Basta estar presente em apenas em um ponto do programa para comprometer todo o resto da lógica em sequência.

Este erro pode ser considerado uma espécie de *buffer overflow* ocasionado por um *byte*. Em relação a reservas, tal problema pode ocorrer quando o programador se esquece que na linguagem C (assim como em diversas outras linguagens) o índice de um vetor começa sempre em zero e não em um. Também ocorre quando outro detalhe é esquecido: toda *string* deve terminar com o caracter de terminação "`\0`" (CORREIA; SOUSA, 2008).

Como exemplo citado por Erickson (2008), o OpenSSH, mesmo com a visão de ser um sistema de comunicação segura, possuía um erro *off-by-one* em determinado trecho de código que trata uma condição sobre canais alocados. Tal erro permitia a exploração em nível de usuário normal, que ao efetuar autenticação poderia obter direitos administrativos do sistema.

No exemplo da Figura 8, é apresentado um erro de lógica caracterizado como *off-by-one*. Na linha 3 é declarado um *buffer* que pode conter oito posições. Na linha 5, apesar da condição verificar se o tamanho do argumento passado pelo usuário é maior que o tamanho da variável `buf`, esta condição está falha. Isso significa que se o usuário informar um argumento maior que oito caracteres será acusado um valor longo (linha 6). Se o argumento for menor que oito caracteres ele é copiado para `buf` (linha 9). Porém, se o argumento possuir o mesmo tamanho que `buf` pode conter, a condição retorna valor falso, e em sequência é feita uma cópia errônea, preenchendo o *buffer* sem o necessário caracter de terminação "`\0`" (linha 9). Ao invés do operador "maior" (`>`) deveria ser utilizado o operador "maior ou igual" (`>=`) para apenas acusar um argumento longo.

```
1  int main(int argc, char **argv) {
2
3  »       char buf[8];
4
5  »       if ( strlen(argv[1]) > sizeof(buf) ) {
6  »       »       printf("O valor inserido e muito longo\n");
7  »       }
8  »       else {
9  »       »       strcpy(buf, argv[1]);
10 »       }
11
12 }
```

Figura 8 - Trecho de código contendo o erro *off-by-one.*
Adaptação de Correia e Sousa (2008).

3.2.5 *Integer underflow*

O *underflow*, ao "contrário" do *overflow*, é um erro de uma expressão inteira quando seu resultado é menor do que o valor mínimo permitido. Esta vulnerabilidade ocorre em casos quando a lógica trabalha com a subtração de valores. É o caso de variáveis declaradas para trabalhar com valores sem sinal – *unsigned* (CORREIA; SOUSA, 2008), não sendo "permitido" em condições normais conter valores negativos.

Como exemplo tomado por Medeiros (2007), uma variável inteira sem sinal de 16 *bits* que recebe a expressão 0-1 não armazenará o valor -1, e sim 2^16-1, que resulta em 65535 *bytes*, ocupando toda a capacidade desta variável.

Baseando-se em Medeiros (2007), a Figura 9 representa uma função que recebe como primeiro argumento um inteiro sem sinal (`len`), e como segundo argumento uma cadeia de caracteres (linha 1). Na linha 3, variável `size` é declarada como um inteiro sem sinal, portanto, teoricamente não deve receber ou ser induzida a valor negativo. Na linha 4 `size` recebe o valor da subtração de `len` por duas unidades. A vulnerabilidade começa nesta linha se `len` estiver com valor 0 ou 1, pois haverá um *underflow* na tentativa de subtrair duas unidades, deixando `size` com um valor muito alto. Na linha 5 será reservado com a função `malloc` um espaço de memória de acordo com o valor de `size` com a soma de um *byte*. Na linha 6 a função `memcpy` copiará para `comm` a origem (`src`) e número de *bytes* a serem copiados (`size`). Observa-se o prejuízo nas linhas 5 e 6, respectivamente

podendo criar uma reserva enorme de *bytes* e copiá-la. A consequência disso pode resultar em negação de serviço, ou a criação de reserva de 0 *bytes*, possibilitando um atacante causar *buffer overflow* por meio da função `memcpy` e explorar.

```
1 void getComm(unsigned int len, char *src) {
2
3       unsigned int size;
4       size = len - 2;
5       char *comm = (char *) malloc(size + 1);
6       memcpy(comm, src, size);
7
8 }
```

Figura 9 - Trecho de código com *integer underflow* presente. Adaptação de Medeiros (2007).

3.2.6 Falta de validação em entradas

De acordo com o OWASP[8], as entradas de um programa (*input*) que não realizam validação de dados, ou seja, que não determinam o que não pode (ou o que apenas pode) ser inserido como valor, pode comprometer a aplicação/sistema a ataques de injeção de comando/código.

Segundo a mesma fonte (OWASP), o objetivo deste ataque é executar comandos que o próprio sujeito determina. Normalmente, comandos que vão além do que é esperado ou já preparado internamente pela aplicação. Quando esta vulnerabilidade é explorada a partir de um terminal, os comandos do sujeito vão ser executados com os mesmos privilégios que a aplicação possui no sistema. Isso pode oferecer um grande risco ao próprio ambiente operacional se o sujeito obtiver privilégios administrativos.

A Figura 10 toma como exemplo um trecho de código adaptado de OWASP. Dado como parâmetro o nome de um arquivo texto existente, este código, após ser compilado, tem o objetivo de ler o arquivo texto especificado pelo usuário. Na linha 3 e 4 é criado uma *string* contendo o comando `cat` (comando nativo do Linux) e um ponteiro para o primeiro argumento (destinado ao nome do arquivo texto), respectivamente. Na linha 5 `tamanhoComando` é destinada ao controle de limite de

[8] OWASP. Disponível em: <https://www.owasp.org/index.php/Command_Injection>. Acesso em: 29 out 2013.

reserva logo abaixo na linha 7 a 10. Na linha 7 é calculado o tamanho em *bytes* para que na próxima linha seja destinado a função `malloc` o tamanho de alocação necessária a `comando`. Em seguinte, na linha 9 é copiado a `comando` "`cat `" e na próxima linha é concatenado a "`cat `" o primeiro argumento da linha de comando. Por fim, na linha 12 é efetuado a chamada para a vulnerável função `system` para executar o comando.

```
1  int main(int argc, char **argv) {
2
3      char cat[] = "cat ";
4      char *comando;
5      size_t tamanhoComando;
6
7      tamanhoComando = strlen(cat) + strlen(argv[1]) + 1;
8      comando = (char *) malloc(tamanhoComando);
9      strncpy(comando, cat, tamanhoComando);
10     strncat(comando, argv[1], (tamanhoComando - strlen(cat)) );
11
12     system(comando);
13     printf("\n");
14     return (0);
15
16 }
```

Figura 10 - Código vulnerável a ataque de injeção de comandos. Adaptada de OWASP.

A Figura 11 apresenta um resultado esperado (linha 2) após informar a `input.out` (resultado da compilação do código da Figura 10) o nome do arquivo texto `leia.txt` (linha 1).

```
1 ./input.out "leia.txt"
2 Lendo este arquivo texto..........x
```

Figura 11 - Execução de input.out. Leitura do arquivo leia.txt, saída normal. Adaptada de OWASP.

A Figura 12 demonstra a exploração com injeção de comando na entrada deste programa (*input*). Primeiramente, na linha 1, é passado o nome do arquivo texto, porém, se adicionado entre aspas, e ser inserido o caracter delimitador de comandos do *shell* (ponto e vírgula) após o nome do arquivo texto, é possível executar outros comandos que não estavam previstos por este programa, como no exemplo, `echo` (final da linha 2), e listando o diretório atual com `ls` (linha 3 à diante).

```
./input.out "leia.txt; echo INJECTION; ls -l"
 Lendo este arquivo texto..........xINJECTION
total 32
-rw-r--r-- 1 root root    2 Oct 30 02:14 arquivo1
-rw-r--r-- 1 root root    2 Oct 30 02:14 arquivo2
-rw-r--r-- 1 root root    2 Oct 30 02:14 cm.sh
-rw-r--r-- 1 root root  485 Oct 30 02:10 input.c
-rw-r--r-- 1 root root  485 Oct 30 02:10 input.c~
-rwxr-xr-x 1 root root 6200 Oct 30 02:10 input.out
-rw-r--r-- 1 root root   37 Oct 30 02:04 leia.txt
```

Figura 12 - Leitura do arquivo leia.txt e injeção de outros comandos. Adaptada de OWASP.

Um dos maiores problemas das entradas são os metacaracteres, que podem modificar a funcionalidade do programa ou obter informações de memória que normalmente não deveriam ser expostas (CORREIA; SOUSA, 2008).

3.3 Engenharia reversa

Como visto anteriormente, algumas simples vulnerabilidades podem causar impacto em diversos níveis à segurança. Uma prática que também pode ser considerada um ataque ao *software* é a engenharia reversa.

A engenharia reversa é considerada pelos autores Peikari e Chuvakin (2004) a arte de obter e dissecar o código fonte de programas binários em que seu código é fechado. Ao desmontar um programa binário, é possível ver como o mesmo funciona em seu mais baixo nível. Porém, mesmo com as proteções que a equipe de desenvolvimento implantam para evitar isso, apesar de reduzir uma parcela de atacantes, pode não ser o suficiente quando praticado por um sujeito habilidoso, que certamente também irá quebrar as proteções. O caso de proteções ao código fonte trata-se de segurança por obscuridade, ou seja, uma forma superficial de segurança, pois, por um lado a proteção “protege” a aplicação, mas por outro lado a própria “proteção” possui a mesma fraqueza de uma aplicação sem proteção, também sendo sujeita a ataques.

Segundo os mesmos autores, Peikaei e Chuvakin (2004) a maioria das empresas desenvolvedoras de *software* contam com um acordo quando o usuário o instala. Este acordo é o *end-user licence agreement* (EULA), é um contrato de licença com o usuário final sobre os devidos usos, ações autorizadas e não

autorizadas. A maioria dessas licenças especificam uma cláusula para prevenir estas práticas de engenharia reversa, além disso, contam com a *Digital Millennium Copyright Act* (DMCA).

Algumas ferramentas da engenharia reversa são conhecidas como: Desmontadores (*disassemblers*), que transformam o código binário em código *assembly;* Depuradores (*debuggers*), que além de fundamentais no desenvolvimento de *software* podem executar o código binário de forma controlada; Descompiladores, usados para a tentativa de traduzir o código compilado a código fonte; Descarregadores (*dumper*), usados como atalhos para “burlar” a técnica de cifras mencionada na seção 4.5; E programas editores que modificam o funcionamento original da aplicação (CORREIA; SOUSA, 2008).

Esta breve abordagem sobre engenharia reversa está presente neste capítulo não por ser considerada uma vulnerabilidade, mas sim por ser uma prática em que todo *software* de código fechado, ou depois de compilado, está sujeito a ser alvo.

Diante de todos estes perigos e fraquezas como na própria linguagem C, é indispensável valorizar a importância deste assunto. Contudo, diante destas poucas ameaças estudadas, o próximo capítulo apresenta proteções e prevenções que podem ser tomadas para deixar a aplicação desenvolvida mais segura.

4 Desenvolvimento seguro e proteções

Como visto, as vulnerabilidades podem trazer consequências variadas à aplicação, e também para o ambiente operacional, quando introduzidas por meio da programação. Este capítulo trata diretamente sobre meios de se obter segurança, de mitigar vulnerabilidades, de diminuir chances de ataques, ou de pelo menos dificultar alguns ataques já citados no capítulo 3. O capítulo apresenta alguns mecanismos e técnicas que auxiliam a "proteção" dos programas, além de suas vantagens, desvantagens e normas relacionadas a segurança.

4.1 Normas ISO

O grupo de consultoria Reavis - Reavis Consulting Group LLC[9] - (2013), alerta as organizações que os *softwares* que são produzidos sem a orientação de normas e metodologias de desenvolvimento seguro provavelmente estarão sujeitos a inserção de riscos de segurança para a própria corporação e para quem utiliza o produto.

4.1.1 ISO 27002

A ISO 27002[10] (*International Standardization Organization*) visa auxiliar as organizações a iniciarem, implementarem ou/e manterem seus sistemas de segurança da informação, mediante diretrizes gerais padronizadas sobre segurança da informação (SI), técnicas de segurança e melhores práticas sobre gestão de segurança da informação providas pela própria ISO. Os controles de segurança que esta norma apresenta são para atender aos requisitos levantados com base nas análises de riscos previamente realizadas.

Uma parte desta ISO trata-se de segurança no desenvolvimento de *software* e alguns controles de segurança que a aplicação deve possuir. Essa abordagem tem o objetivo de prevenir erros, perdas, mau uso, e modificação não autorizada de informações que são manipuladas durante o processamento da aplicação. Alguns controles específicos apresentados pela norma são a validação dos dados de entrada, controle do processamento interno e validação de dados de saída, entre outros.

---

[9] REAVIS. Disponível em: <http://www.reavis.org>. Acesso em: 19 out 2013.
[10] ISO. Disponível em: <http://www.iso.org/>. Acesso em: 07 dez 2013.

4.1.2 ISO 15408 – *Common Criteria*

A ISO 15408, também conhecida como *Common Criteria* (CC), basicamente determina os requisitos de segurança que as aplicações devem ter para serem consideradas e reconhecidas como seguras internacionalmente. Apesar desta ISO mencionar as exigidas funcionalidades de segurança como necessárias para obter a certificação, ela não orienta como implementar tais funcionalidades de segurança nas aplicações (MARIOTTI, 2004).

A *Common Criteria* é dividida em três partes, 15408-1:2009, 15408-2:2008 e 15408-3:2008, publicadas no site da ISO[11]:

- **ISO 15408-1:2009**: Estabelece os conceitos e princípios gerais de avaliação de segurança em TI e especifica o modelo de avaliação da segurança nas aplicações de forma geral. Nas descrições são definidos vários termos que são utilizados pela ISO, além de estabelecer o alvo, contexto e critérios de avaliação;
- **ISO 15408-2:2008**: Define o conteúdo e a apresentação dos requisitos funcionais de segurança a serem avaliados, contando com um abrangente catálogo destes componentes de acordo com as necessidades mais comuns do mercado;
- **ISO 15408-3:2008**: Define os requisitos de garantia dos critérios de avaliação juntamente com uma escala para medir as metas e os critérios de avaliação.

4.1.3 ISO 27034

A ISO 27034 oferece orientações sobre segurança da informação especificadamente em projetos, programação, implementação e utilização de sistemas aplicativos. O objetivo desta norma é garantir que as aplicações informáticas atinjam um nível de segurança necessário ou desejado, ajudando as organizações a integrarem a segurança durante todo o ciclo de vida de seu *software*. Portanto, de acordo com o grupo Reavis (2013), esta ISO não descreve controles de segurança específicos ou padrões de tecnologia, pois foi projetada para ser compatível com o ciclo de desenvolvimento seguro de modelos existentes.

O grupo de consultoria Reavis (2013) descreve brevemente sobre a primeira parte desta ISO, a 27034-1:2011, que é a parte que pode ser utilizada independente das outras seguintes. Ela visa proporcionar uma compreensão de consenso,

[11] ISO. Disponível em: <http://www.iso.org/>. Acesso em: 22 out 2013.

apresentando definições, conceitos e princípios envolvidos sobre segurança em aplicações (ISO, 2011)[12].

Atualmente as seguintes cinco partes encontram-se em desenvolvimento e apesar de não estarem prontas e disponibilizadas, provavelmente abordarão o assunto com mais profundidade.

4.2 Modelos de desenvolvimento seguro

As normas ISO abordadas até então, proporcionam orientações e vantagens quando adotadas em um projeto. No desenvolvimento de *software*, é primordial optar e seguir um modelo de desenvolvimento. Alguns dos modelos tradicionais como o modelo ágil, sequencial e espiral, visam instruir as equipes por meio de várias fases, maneiras de executar o projeto até resultar no produto. Porém, segurança no desenvolvimento não é o foco nestes modelos. Atualmente existem modelos de desenvolvimento de *software* que são focados inteiramente em segurança. Dois modelos abordados neste trabalho são o *Secure Development Lifecycle* (SDL)[13] desenvolvido pela *Microsoft* e o livre *Open Software Assurance Maturity Model* (*OpenSAMM*)[14].

4.2.1 *Secure Development Lifecycle*

O modelo SDL tem como objetivos reduzir o número de defeitos que possam comprometer a segurança computacional e reduzir a gravidade de todos os outros defeitos restantes. Este modelo adota uma série de atividades relacionadas à segurança, como por exemplo, análises, testes e revisões. Howard (2013) afirma que este modelo é adaptável e flexível a qualquer metodologia tradicional de desenvolvimento. Este modelo possui sete fases – Figura 13 – (LIPNER; HOWARD, 2005):

- **Fase de treinamento**: a primeira fase é fundamental para a implantação do SDL. Nesta fase, são apresentados conceitos fundamentais sobre segurança para o projeto, modelagem de ameaças, programação segura, testes e melhores práticas.

[12] ISO. Disponível em <http://www.iso.org/iso/>. Acesso em: 21 out 2013.
[13] MICROSOFT. Disponível em: <http://www.microsoft.com/security/>. Acesso em: 17 out 2013.
[14] OPENSAMM. Disponível em: <http://www.opensamm.org>. Acesso em: 17 out 2013.

- **Fase de requerimentos**: esta fase visa identificar os objetivos principais da segurança. Define a documentação (sujeita a alteração ao longo do projeto), níveis de privilégios e níveis mínimos de segurança aceitáveis.
- **Fase de design**: identifica a estrutura e os requisitos gerais de *software*. Esta fase define as diretivas de *design* e arquitetura de segurança, organizando os processos. Os elementos da superfície de ataque são documentados e é feita a modelagem das ameaças.
- **Fase de implementação**: nesta fase é programado o código e efetuado testes direcionados nos resultados da modelagem de ameaças. Para isto, devem ser aplicados padrões de programação que ajudam a evitar a introdução de falhas de segurança. Os testes são efetuados com ferramentas, entre eles, ferramentas de análise estática de código (seção 4.3.1). Os testes manuais completam esta fase (seção 4.3.3).
- **Fase de verificação**: é o ponto em que o *software* está funcionalmente concluído. Durante esta fase é realizado um "esforço de segurança" que inclui revisões do código (estática e manual) além das já concluídas na fase anterior. Isso objetiva que a versão final do *software* venha a ser segura. Em a cada nova implementação no *software* é importante revisá-la juntamente com o código já herdado. Uma nova vulnerabilidade neste cenário estará isolada, facilitando sua remoção. Em meio a isto, é realizada análise dinâmica (seção 4.3.2) monitorando o comportamento do *software* enquanto é executado.
- **Fase de lançamento**: garante e certifica que o *software* cumpre todos os requisitos de segurança e avalia se a documentação foi atendida com sucesso. Nesta fase começa o planejamento de respostas a incidentes que venham a acontecer. Também é realizada uma revisão final de todos os componentes e atividades de segurança que foram inseridos no *software*.
- **Fase de resposta**: ajuda a adotar medidas para as futuras vulnerabilidades, conforme necessário. Para isso, é feita a avaliação dos relatórios de vulnerabilidades. Quando uma vulnerabilidade é corrigida, é lançado um *patche* de segurança para corrigir a falha.

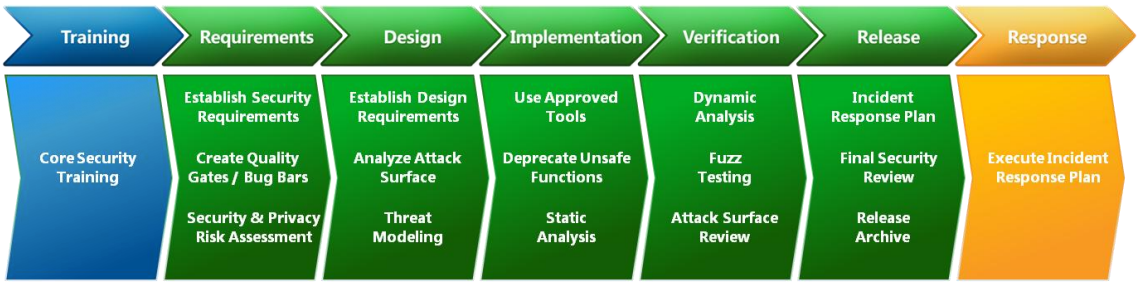

Figura 13 - Visão geral do modelo SDL da Microsoft.
Fonte: Microsoft.

4.2.2 *Open Software Assurance Maturity Model*

Este é um modelo aberto para ajudar as organizações a formular e implementar segurança em *software* alinhando-se com a estratégia de negócio. Este modelo foi projetado para ser bastante flexível, para empresas de qualquer porte, sendo possível utilizar com qualquer modelo de desenvolvimento tradicional. Segundo o site do OpenSAMM (2012)[15], seu modelo é dividido em quatro funções de negócio (Figura 14), e cada função possui três práticas de segurança:

- **Função de governança**: é centrada nas atividades e processos de negócio estabelecidos a na organização, tendo relação com o próprio desenvolvimento de *software*. As práticas de segurança envolvem estratégia e métricas, educação e orientação, política e conformidade.
- **Função de construção**: que define metas, adota padrões reconhecidos para o desenvolvimento, design e implementação (CONVISO, 2012)[16]. As práticas seguras desta etapa é o levantamento de requisitos de segurança, a modelagem de ameaças e arquitetura segura.
- **Função de verificação**: É focada nas verificações e testes do que é produzido durante o desenvolvimento de *software*. Com isso, inclui a garantia de qualidade. As práticas de segurança para esta função envolvem a revisão de arquitetura, testes de segurança, e revisão de código.
- **Função de desenvolvimento**: tem o objetivo de gerenciar o lançamento do *software* que foi desenvolvido, verificando se o produto estará de acordo com as especificações (CONVISO, 2012). Esta última etapa conta com práticas para gerenciamento de vulnerabilidades, proteção do ambiente, e capacitação operacional.

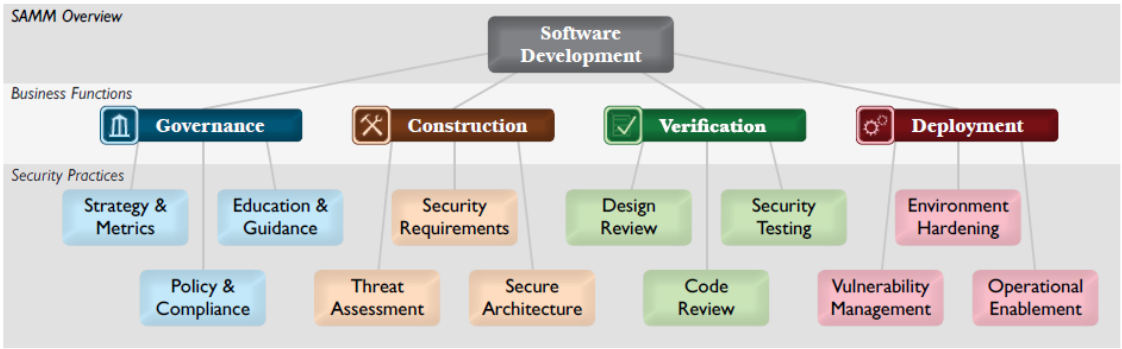

Figura 14 - Visão geral do modelo OpenSAMM.
Fonte: OpenSAMM.

[15] OPENSAMM. Disponível em: <http://www.opensamm.org>. Acesso em: 18 out 2013.
[16] CONVISO. Disponível em: <http://blog.conviso.com.br/2012/11/opensamm-o-que-e-e-para-que-serve.html>. 18 out 2013.

4.3 Métodos de análises

Como visto nos dois modelos de desenvolvimento seguro apresentados, ambos possuem análises em algumas de suas fases. O objetivo de analisar é detectar erros de programação existentes no código, erros gerados durante o tempo de execução ou por conta de falha no próprio projeto. As ferramentas de análise são bem vindas no ciclo de desenvolvimento de *software*, pois além de contribuir com a segurança detectando e/ou eliminando vulnerabilidades, poupam tempo e custo (TEIXEIRA, 2007). Geralmente, três tipos de análises mais comuns são: análise estática (CHESS; MCGRAW, 2004), análise dinâmica e análise manual.

4.3.1 Análise estática de código

A análise estática examina o código fonte do programa (embora também seja possível analisar o código compilado). Normalmente é realizada em tempo de compilação (CHESS; MCGRAW, 2004). Esta análise pode melhorar consideravelmente a qualidade e confiabilidade do *software*, apresentando redução significativa de falhas (SCHILLING; ALAM, 2006)[17].

Algumas ferramentas podem apresentar análises de forma simples, apenas buscando por funções vulneráveis no código (por exemplo, `strcpy`, `gets`, entre outras encontradas no Anexo I), outras, buscam vulnerabilidades por meio de técnicas avançadas na varredura. Basicamente, as ferramentas irão identificar somente vulnerabilidades que foram destinadas a detectar, sendo necessária atenção na escolha da ferramenta apropriada (TEIXEIRA, 2007).

Algumas ferramentas de análise estática para linguagem C são comentadas por Chess e McGraw (2004):

- ITS4[18]: Esta é uma ferramenta simples de usar. Suas análises consistem em regras que identificam possíveis vulnerabilidades.
- Flawfinder: Também utiliza o método de análise de sintaxe (ferramenta demonstrada no Estudo de caso).

Uma ferramenta online para este tipo de análise é disponibilizada por Gimpel *Software*[19]. No site *spinroot*[20] encontrasse diversas ferramentas desta análise destinadas à linguagem C

[17] EMBEDDED. Disponível em: <www.embedded.com>. Acesso em: 27 out 2013.
[18] CIGITAL. Disponível em: <www.cigital.com/its4>. Acesso em: 27 out 2013.
[19] GIMPEL. Disponível em: <http://www.gimpel-online.com/OnlineTesting.html>. Acesso em: 26 out 2013.
[20] SPINROOT. Disponível em: <http://www.spinroot.com/static/>. Acesso em: 26 out 2013.

4.3.2 Análise dinâmica

Ao invés de tentar determinar a presença de vulnerabilidade no código fonte, este tipo de análise verifica o comportamento do *software* na memória, ou seja, em tempo de execução. As ferramentas para esta análise, por exemplo, podem controlar o acesso à memória verificando por vulnerabilidade de memória sobrelotada, detectar problemas de acesso fora de limites, *buffer overflows*, entre outros (TEIXEIRA, 2007). Útil para detectar *bugs* que surgem devido à incompatibilidades com o sistema operacional. Portanto, é necessário ter atenção no foco da ferramenta de análise dinâmica escolhida, pois, se as análises não forem suficientes ou abrangentes, partes da aplicação correm o risco de não serem analisadas. Esta ferramenta pode ser utilizada em conjunto com um depurador no processo de desenvolvimento. Algumas ferramentas são mencionadas por Teixeira (2007):

- FormatGuard: Esta ferramenta foca em vulnerabilidades relacionadas a formatação de *strings*.
- Safe C: Uma ferramenta que verifica problemas de transbordamento da memória e possíveis acessos não autorizados a endereços.
- Valgrind: Detecta problemas relacionados com a memória em geral. (ferramenta demonstrada no Estudo de Caso).

4.3.3 Análise manual

A análise manual (ou auditoria de código) se destina a descobrir vulnerabilidades de programação e também é útil para detectar vulnerabilidades de projeto (CORREIA; SOUSA, 2008). Como conceituado anteriormente, algumas vulnerabilidades de programação podem ser detectadas com ferramentas sofisticadas de análise estática ou dinâmica, porém o método manual também possui seu valor. Entretanto, Chmielewski, Clift, Fonrobert e Ostwald (2007), afirmam que certos erros que induzem à vulnerabilidades, especialmente as de projeto, não são detectados com ferramentas, pois normalmente envolvem erros durante especificações do projeto, ou lógica de programação.

Este tipo de análise está sujeita a limitações. Pode levar muito tempo até sua conclusão, além de correr o risco de não ser feita de forma eficaz ou adequada devido a erros humanos. Quem analisa os códigos desta forma deve ter um bom conhecimento sobre vulnerabilidades e estar sempre atualizado às novas (CHESS; MCGRAW, 2004). Além disso, é preciso conhecer a documentação do *software*.

Contudo, neste tipo de análise pode-se envolver ferramentas que auxiliam o profissional na leitura e buscas, como por exemplo, o comando `grep`, nativo do Linux (CORREIA; SOUSA, 2008).

4.4 Evitando vulnerabilidades e dificultando exploração

Na programação, para evitar as vulnerabilidades apresentadas no capítulo 3, bastam algumas medidas e sugestões (como no mini *checklist*, seção 5.4) a fim de minimizá-las. Para dificultar alguns ataques à vulnerabilidades presentes no programa, existem alguns mecanismos de proteção dinâmica. Esta seção apresenta algumas medidas, e mecanismos de defesa relacionados às vulnerabilidades do capítulo 3.

4.4.1 Proteções contra exploração de *buffer overflow*

Para evitar a inserção de *buffer overflow*, basicamente podem ser especificados limites à quantidade de informação que o usuário pode informar, além de implantar verificações testando se um *buffer* pode conter determinado conteúdo em relação ao seu tamanho. Também é valido sempre evitar o uso de funções vulneráveis que induzem a esta vulnerabilidade.

Para as vulnerabilidades de *overflow* na pilha e *overflow* na *heap* existem mecanismos de proteção dinâmica (em tempo de execução) e outros mecanismos que são inseridos na aplicação durante a programação. Algumas técnicas simples apresentadas a seguir procuram minimizar as chances de um ataque à *buffer overflow* ser executado com êxito.

4.4.1.1 Aleatoriedade de memória

A técnica de aplicar aleatoriedade nos endereços de memória, *Address Space Layout Randomization* (ASLR), basicamente modifica a organização de memória a cada reinicialização do SO (CUGLIARI; GRAZIANO, 2010). Atualmente está presente no Linux[21] e nos sistemas Windows[22] mais recentes. Ela reage em tempo

[21] Disponível em: <http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems>. Acesso em: 28 out 2013.
[22] MICROSOFT. Disponível em: <http://www.microsoft.com/security/sir/strategy/default.aspx#_!section_3_3>. Acesso em: 28 out 2013.

de execução, modificando os endereços onde reside o código, contando com certa aleatoriedade. Os elementos que podem sofrer este embaralhamento são os endereços do código do programa, endereços de bibliotecas dinâmicas, endereços bases da pilha e da *heap* (CORREIA; SOUSA, 2008). Com isso, quem está atacando não sabe inicialmente os endereços necessários para seguir o ataque, pois a probabilidade de acerto em tentativa e erro é pequena.

Este mecanismo apresenta certo grau de dificuldade. Mover código e dados pela memória pode trazer limitações a compartilhamentos entre as aplicações (CORREIA; SOUSA, 2008), pois há casos em que a localização de memória de certos componentes do SO é fixa.

Mesmo este mecanismo dificultando alguns tipos de ataques à *buffer overflow*, pode estar sujeito à força bruta, ou antes, o sujeito pode descobrir de alguma forma a organização da memória e obter o endereço desejado.

4.4.1.2 Aleatoriedade de instruções

A prioridade de defesa da técnica de aleatoriedade de instruções é combater injeção de códigos nas entradas da aplicação. Basicamente, ela propõe através de um emulador, ofuscar todo o código da aplicação (através de um único trecho de código), operando o ou-exclusivo (*XOR*) de cada *byte* com uma chave aleatória. Por exemplo, o resultado do *XOR* da injeção de um código arbitrário com o um número aleatório, não fará sentido à aplicação se o atacante não souber o número aleatório. Porém, esta técnica prejudica seriamente o desempenho do *software* (CORREIA; SOUSA, 2008).

4.4.1.3 Pilha/*heap* não executável

Para a pilha que armazena dados como endereços de retorno, parâmetros de funções, variáveis locais, não há razão tão comum para ter permissão de execução habilitada. No entanto, é possível que a pilha esteja em modo somente de escrita ou de execução. Desabilitar a execução nestes segmentos apenas evita os ataques envolvendo injeção de códigos que as vulnerabilidades de *buffer overflow* estão sujeitas, como citados na seção 3.2.3 e na seção 3.2.2*,* embora não ser o foco deste trabalho dificultar as outras inúmeras formas de ataque. Basicamente, desabilitando a execução, as páginas de memória que contém a pilha são marcadas para modo não executável. Portanto, ela passa a ser somente em modo escrita e de fato, instruções controladas injetadas passam a não ter efeito (CORREIA; SOUSA, 2008).

4.4.1.4 Proteção contra esmagamento de pilha

O site de pesquisas da empresa *International Business Machines* (IBM)[23] informa que o compilador *gcc,* a partir da versão 3 inclui uma "proteção" contra os ataques de *Smashing Stack*. Esta funcionalidade é conhecida por *GCC-Stack-Smashing Protector* (ou *ProPolice*), que utiliza uma técnica conhecida por "canário". Ou seja, qualquer código fonte compilado atualmente num sistema Linux por padrão terá esta proteção adicional.

De acordo com o site *Operating System Development* (OSDEV)[24] a técnica de canário consiste em colocar um ponto de verificação no início das funções presentes na pilha. No final da função, antes do retorno, é feita verificação se o canário não foi corrompido. Se o valor original do canário for alterado indica que houve uma corrupção, e a execução do programa é encerrada. Apesar de evitar ataques, o fato deste mecanismo encerrar a aplicação pode não ser tão viável, dependendo do grau de importância da mesma.

4.4.2 Medidas ao erro *off-by-one*

Como visto na seção 3.2.4 os erros *off-by-one* podem surgir por meio de má interpretação de lógica. Uma boa prática para evitar este erro é sempre incluir um *byte* a mais além do necessário, e na última posição inserir o caracter de terminação "`\0`". Em testes lógicos do programa, sempre testar todas as possibilidades e avaliar os resultados se são consistentes com o esperado. Estes testes podem ter o auxilio de um fluxograma. Além disso, o programador não deve se esquecer que o índice de um vetor sempre se inicia na posição 0, e não na posição 1 (HOWARD; LEBLANC, 2002).

Em meio a isto, é recomendado utilizar funções mais "seguras" que tratam estes problemas de falta de caracter de terminação, cópia excessiva no *buffer*, entre outros (CORREIA; SOUSA, 2008).

4.4.3 Medidas ao *integer underflow*

Apesar desta vulnerabilidade também ser difícil de detectar, a análise estática e dinâmica podem ser bastante útil para a verificação de possíveis condições de

[23] IBM. Disponível em: <http://www.research.ibm.com/trl/projects/security/ssp>. Acesso em: 10 out 2013.
[24] OSDEV. Disponível em: <http://wiki.osdev.org/GCC_Stack_Smashing_Protector>. Acesso em: 10 out 2013.

underflow. Para evitá-la, é recomendado nunca atribuir expressões variáveis aritméticas à variáveis `unsigned int` sem antes aplicar verificações adequadas às condições que podem oferecer *underflow* – subtrações, ou adição de variáveis que podem assumir valor negativo (OWASP)[25].

4.4.4 Validação nas entradas

A validação de dados visa garantir que os dados a serem manipulados estão de acordo com o tipo aceito, e da forma correta. Por exemplo, uma entrada designada para receber apenas números inteiros sem sinal, tecnicamente deve receber um valor na faixa de 0 a 65535 *bytes* (tamanho mínimo e máximo que `unsigned int` comporta). Para tratar as entradas, é possível implementar uma lista branca, aceitando somente dados que estão presentes nela, ou implementar lista negra, contendo informações que não podem ser aceitas. Além disso, é possível implementar a técnica de higienização, que consistem em tratar o que o usuário inseriu, removendo caracteres perigosos – metacaracter (CORREIA; SOUSA, 2008).

4.4.5 Ligação dinâmica

A ligação dinâmica consiste em uma técnica capaz de informar ao sistema a biblioteca que deseja utilizar para a execução de um programa. Tal programa pode já estar compilado. Esta prática está sujeita a algumas regras para seu funcionamento. A função personalizada deve possuir o mesmo nome e escopo (mesmos parâmetros e mesmos tipos) que a função da biblioteca C padrão. No Linux, é necessário informar à variável de ambiente `LD_PRELOAD` a biblioteca personalizada que deseja utilizar para a execução. Com isso, é substituída as funcionalidades padrões já implantadas. Com isto, é possível modificar funções C inseguras (Anexo I) para versões mais seguras, por exemplo, implementando verificações de limite de *buffer* e outras regras ausentes (LEVINE, 1999).

[25] OWASP. Disponível em: <https://www.owasp.org/index.php/OWASP_Periodic_Table_of_Vulnerabilities_-_Integer_Overflow/Underflow>. Acesso em: 30 out 2013.

4.5 Mecanismos para proteger/dificultar engenharia reversa

Diante das diversas técnicas que tentam ao menos dificultar as práticas de engenharia reversa, cada uma possui seus pontos fracos a ser explorados, e desvantagens, tanto na utilização do produto quanto na concretização (PEIKARI; CHUVAKIN, 2004).

As técnicas utilizadas para dificultar são conhecidas como ofuscamento (COLLBERG; THOMBORSON, 2002). Algumas técnicas simples que podem dificultar a leitura consistem em apagar ou misturar nomes de funções e variáveis, evitar explicações em comentários, adicionar código nunca executado, embaralhamento do código, codificar os dados, entre outros (CORREIA; SOUSA, 2008). Uma solução um tanto eficaz, porém difícil de concretizar, consiste em cifrar algumas partes do código em memória utilizando a operação ou-exclusívo (*XOR*) com um número aleatório de 1 *byte*.

Outro mecanismo que dificulta é chamado de metamorfismo (DUBE, 2006). Ele é operado dinamicamente, fazendo com que o *software* se auto modifique durante sua execução. Essas modificações podem contar com: *overflow* intencional na pilha com a intenção de alterar o endereço de retorno de funções; Escrita por cima de instruções de código, escondendo chamadas a interfaces; Reorganização das sub-rotinas na memória no tempo de execução (CORREIA; SOUSA, 2008). Com este meio, a compreensão do código se torna bastante difícil, assim como a tradução dos *disassemblers*.

Após uma visão sobre vulnerabilidades no capítulo 3, e estudado neste capítulo 4 sobre normas e modelos de desenvolvimento seguro e análises, o próximo capítulo traz alguns testes relacionados às vulnerabilidades e aos métodos de análise estática e dinâmica referidos nas seções 4.3.1 e 4.3.2 respectivamente.

5 Estudo de caso

A proposta de estudo de caso deste trabalho visa demonstrar o uso de ferramentas de análises, suas funcionalidades e opções. Para isto, foi selecionado o analisador estático Flawfinder[26], e examinado um código para focar na funcionalidade da ferramenta. Outra ferramenta também demonstrada é a de análise dinâmica, o Valgrind[27], sob um arquivo binário. Além disso, no fim deste estudo de caso é proposto um mini *checklist* para condições mínimas a seguir no ato de programar, a fim de evitar as vulnerabilidades apresentadas no capítulo 3.

5.1 Ambiente de execução

Os seguintes testes de análise estática de código e análise dinâmica foram executados em um sistema Linux, mais especificadamente na distribuição BackTrack 5 R3 de 32 *bits*. Para os testes das análises, os códigos-fontes foram compilados pelo gcc, versão 4.4.3. Na análise estática, foi utilizada a ferramenta Flawfinder em sua versão 1.27. No teste da análise do Flawfinder sob o Mozilla Firefox[28], a versão do navegador é a 10.0.12. Para a análise dinâmica, foi utilizado o analisador Valgrind, versão 3.6.0.

5.2 Análise estática com o Flawfinder

Para os seguintes testes com o Flawfinder, é examinado o código fonte presente na Figura 15. Para o uso, a sintaxe desta ferramenta é simples, sendo: `flawfinder -flags arquivo/diretório`. Conforme a Figura 16, ao analisar um arquivo fonte, por padrão o Flawfinder exibe um cabeçalho contendo informações da ferramenta, como sua versão atual e seu autor - David A. Wheeler – (linha 2), além de apresentar a quantidade de funções da linguagem C consideradas perigosas (linha 3).

---

[26] Disponível em: <http://www.dwheeler.com/flawfinder/>. Acesso em: 31 out 2013.
[27] Disponível em: <http://valgrind.org/>. Acesso em: 2nov 2013.
[28] MOZILLA. Disponível para download em: <ftp://archive.mozilla.org/pub/mozilla.org/firefox/releases/10.0.12esr/source/>. Acesso em: 31 out 2013.

```
 5 int main(int argc, char **argv) {
 6
 7         char *reserva1 = (char *) malloc(sizeof(char)*10);
 8         char *reserva2 = (char *) malloc(sizeof(char)*10);
 9         char *reserva3 = (char *) malloc(sizeof(char)*10);
10         char variavel[] = "hey";
11
12         strcpy(reserva2, "abacate");
13         strcpy(reserva1, argv[1]);
14         gets(reserva3);
15
16         printf("Valor de reserva1 = %s\n", reserva1);
17         printf("Valor de reserva2 = %s\n", reserva2);
18         printf(variavel);
19
20 }
```

Figura 15 - Código (vulnb.c) submetido a análise do Flawfinder. Adaptação de Correia e Sousa (2008).

Na Figura 16, os resultados da análise são expostos a partir da linha 4. De forma geral, o resultado contém o nome do arquivo analisado, localização das supostas vulnerabilidades no código fonte, nível de risco e uma pequena explicação das vulnerabilidades encontradas. Por fim, é impresso um resumo geral apresentando a quantidade de alertas (*hits*), linhas analisadas, tempo decorrido, entre outras informações calculadas.

```
root@bt:~/codigos# flawfinder vulnb.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining vulnb.c
vulnb.c:14:  [5] (buffer) gets:
  Does not check for buffer overflows. Use fgets() instead.
vulnb.c:13:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
vulnb.c:18:  [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
vulnb.c:12:  [2] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused). Risk
  is low because the source is a constant string.

Hits = 4
Lines analyzed = 21 in 0.51 seconds (1411 lines/second)
Physical Source Lines of Code (SLOC) = 15
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   2 [5]   1
Hits@level+ = [0+]   4 [1+]   4 [2+]   4 [3+]   3 [4+]   3 [5+]   1
Hits/KSLOC@level+ = [0+] 266.667 [1+] 266.667 [2+] 266.667 [3+] 200 [4+] 200 [5+
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Figura 16 - Análise estática sob um arquivo fonte (vulnb.c).

Seguindo a Figura 16, na linha 5 é mostrado o nome do arquivo analisado (`vulnb.c`) e logo a frente dos dois pontos, o número da linha no código fonte onde se encontra o problema (linha 14 do código fonte). Após isso, a frente é mostrado entre colchetes o nível de risco à segurança (`[5]`), e em seguida uma classificação do problema entre parênteses (`buffer`). No final da linha 5 é impresso a função vulnerável presente neste alerta (`gets`). Abaixo, na linha 6 é informado qual o problema cometido pelo programador, e às vezes uma breve explicação do que isto pode causar à segurança, neste caso "`Does not check for buffer overflows`". Dependendo da função, o Flawfinder apresenta uma sugestão de função alternativa "mais segura" que possa substituir a perigosa (minimizando o impacto), neste caso da linha 6, "`Use fgets() instead.`".

Na linha 7 e 13 foram encontradas mais duas vulnerabilidades classificadas como problema de *buffer*. Em ambas as situações é utilizada a função `strcpy` e nas linhas 8 e 14 é alertada a falta de verificação de limites na cópia de conteúdo ao *buffer*, embora a vulnerabilidade da linha 13 (que copia uma constante) não ofereça tanto risco quanto a vulnerabilidade da da linha 7 (que copia um conteúdo de dimensão variável), conforme a Figura 16.

Outra vulnerabilidade encontrada (linha 10 da Figura 16) está relacionada à formatação de informações de saída (`format`) com a má utilização da função `printf`. Neste caso, na função `printf` não foi especificado o formato de saída das informações. Isso pode ocasionar uma vulnerabilidade que permite um atacante especificar caracteres de formatação de dados a fim de obter informações sensíveis da memória que não devem ser expostas.

Na Figura 16, com as vulnerabilidades exibidas nas linhas 5, 7, 10 e 13, é possível observar que são impressas em ordem decrescente em relação ao nível de risco, sendo primeiramente exibidos os perigos de nível mais alto (linha 5) até os de nível mais baixo (linha 13). No fim da análise, na linha 18 é mostrada a quantidade de supostas vulnerabilidades encontradas no código fonte analisado.

Conforme a Figura 17, com a opção `-D` ou `--dataonly` (linha 1) são mostrados apenas os resultados da análise estática. Ela omite as informações de cabeçalho e rodapé. Pode ser usada em conjunto com a opção `-Q` ou `--quiet` que omite o nome do arquivo fonte analisado.

Para mais detalhes na análise, ao incluir a *flag* `--columns` (linha 1 da Figura 17), é impresso o ponto específico (linha e coluna) da localidade da vulnerabilidade no código. A impressão da coluna se encontra logo após o número da linha, por exemplo, na linha 2 da Figura 17, a vulnerabilidade se encontra na linha 14 do código fonte, na segunda coluna. O primeiro caracter da linha (contando com espaço em branco) é representado pela coluna número 1. Isso pode ser útil quando se quer a localização exata do início da vulnerabilidade.

Para melhorar visualmente os resultados, a opção `-c` ou `--context` (linha 1 da Figura 17) traz juntamente com os alertas de risco o código presente na linha, facilitando a interpretação rápida ao ler o alerta junto com a falha de segurança (linhas 4, 8, 12 e 17 da Figura 17).

Com essas omissões e apresentação de detalhes, o profissional pode focar melhor nos problemas de segurança apontados pela ferramenta.

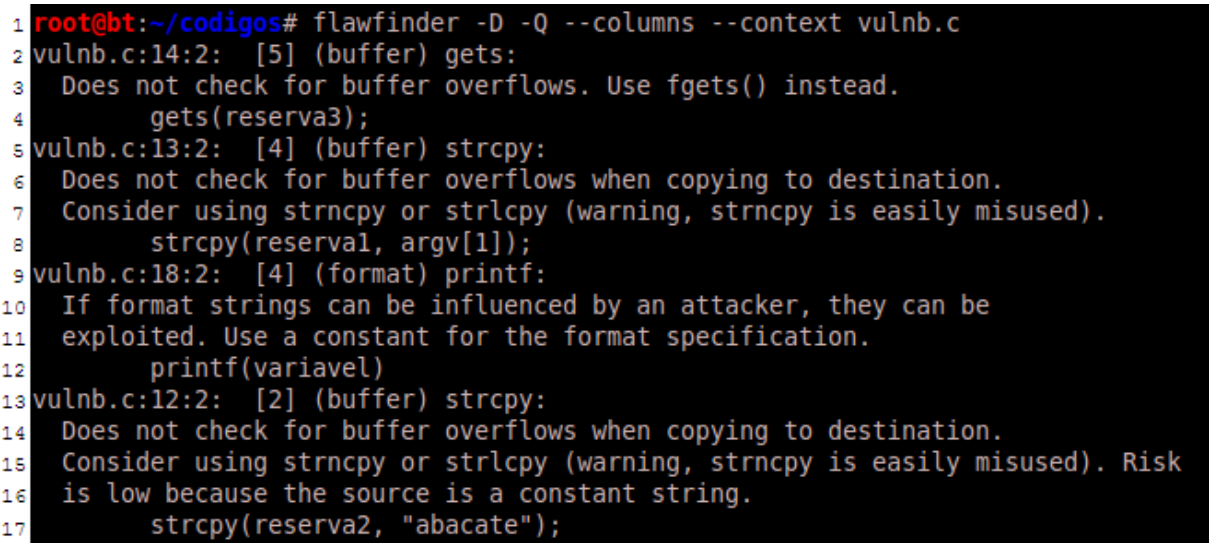

Figura 17 - Parâmetros para "lapidar" a visualização dos resultados.

O Flawfinder tem a capacidade também de examinar todos os códigos fontes de um diretório inteiro, incluindo sub-diretórios. Para isso, basta informar na linha de comando o nome do diretório que deseja. Para incluir arquivos fontes ocultos, é preciso especificar a *flag* `--followdotdir`. Para este exemplo, na Figura 18, são analisados todos os arquivos fontes do navegador Firefox, incluindo os ocultos.

No entanto, quando o diretório possui uma quantidade muito grande de arquivos fontes, e supondo que cada arquivo fonte gere diversos erros, não seria possível visualizar tudo simplesmente pelo *shell*. Portanto, neste caso, é

aconselhável utilizar também a *flag* `--html` (linha 1 da Figura 18). Com isso, os resultados são gerados em arquivo HTML (*Hyper Text Markup Language*), para serem visualizados em um navegador *web*. Ao especificar esta opção é preciso redirecionar a saída para um arquivo HTML.

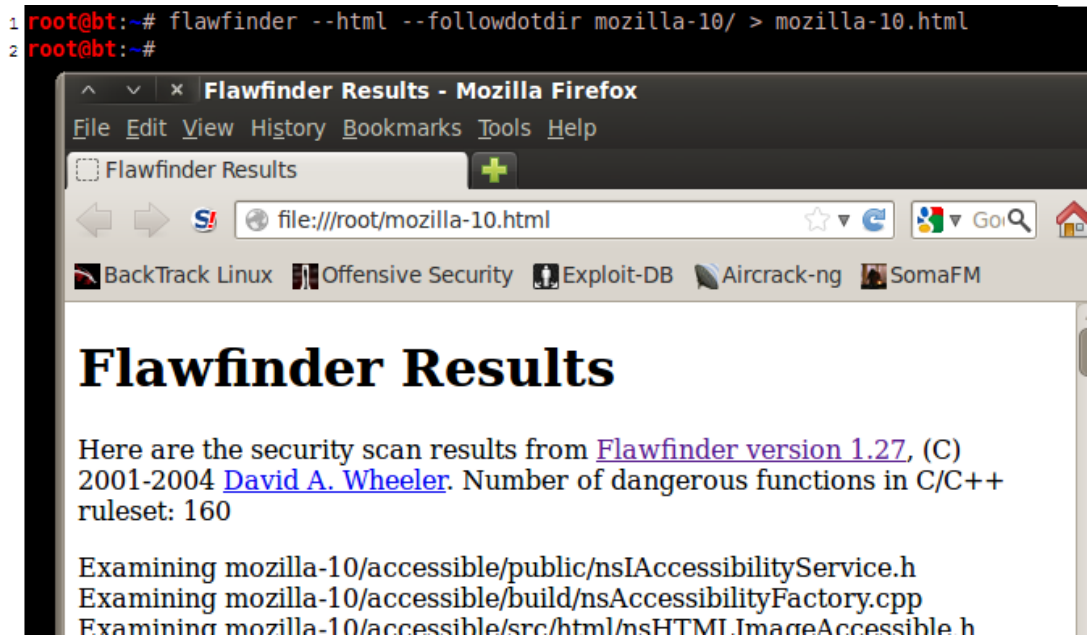

Figura 18 - Resultados de uma análise estática gerada em arquivo HTML.

Os níveis de riscos do programa podem variar de 1 a 5. A classificação 1 representa risco baixo, enquanto que o nível 5 são os riscos mais perigosos segundo a ferramenta. Com a opção `-m` ou `--minlevel=X` é possível exibir somente os riscos que são maiores ou iguais ao nível especificado, como exibido na Figura 19, configurado para mostrar os riscos a partir do nível 3. Além disso, vale ressaltar que ao especificar o nível de risco 0 torna a analise muito mais rigorosa, mostrando todos as possibilidades de riscos à segurança.

```
1  root@bt:~/codigos# flawfinder -D -Q --context --minlevel=3 vulnb.c
2  vulnb.c:14:  [5] (buffer) gets:
3    Does not check for buffer overflows. Use fgets() instead.
4          gets(reserva3);
5  vulnb.c:13:  [4] (buffer) strcpy:
6    Does not check for buffer overflows when copying to destination.
7    Consider using strncpy or strlcpy (warning, strncpy is easily misused).
8          strcpy(reserva1, argv[1]);
9  vulnb.c:18:  [4] (format) printf:
10   If format strings can be influenced by an attacker, they can be
11   exploited. Use a constant for the format specification.
12         printf(variavel)
```

Figura 19 - Analisando os riscos do código a partir do nível 3.

Outra opção que o Flawfinder oferece, demonstrada na Figura 20, é a *flag* `-I` ou `--inputs`, com o intuito de exibir apenas as linhas de código que contenham funções perigosas dependentes de dados externos (por exemplo, informações que o usuário deve informar), ou seja, funções *input* (linha 2). Bastante útil para separar as entradas de outros alertas, e assim, tratá-las separadamente.

```
1 root@bt:~/codigos# flawfinder -D -Q --context --inputs vulnb.c
2 vulnb.c:14:  [5] (buffer) gets:
3   Does not check for buffer overflows. Use fgets() instead.
4         gets(reserva3);
5 root@bt:~/codigos#
```

Figura 20 - Resultado revelando apenas funções *input* perigosas.

Conforme a Figura 21, outra funcionalidade disposta é a possibilidade de salvar listas com o parâmetro `--savehitlist` (linha 1). Criar listas pode ser bastante útil para futuras comparações com outras versões conforme novas implementações, e assim, detectar as novas vulnerabilidades inseridas. A linha 18 confirma a lista de perigos salva desta versão do código fonte.

```
1  root@bt:~/codigos# flawfinder -D -Q --context --savehitlist vulnb.hits vulnb.c
2  vulnb.c:14:  [5] (buffer) gets:
3    Does not check for buffer overflows. Use fgets() instead.
4          gets(reserva3);
5  vulnb.c:13:  [4] (buffer) strcpy:
6    Does not check for buffer overflows when copying to destination.
7    Consider using strncpy or strlcpy (warning, strncpy is easily misused).
8          strcpy(reserva1, argv[1]);
9  vulnb.c:18:  [4] (format) printf:
10   If format strings can be influenced by an attacker, they can be
11   exploited. Use a constant for the format specification.
12         printf(variavel);
13 vulnb.c:12:  [2] (buffer) strcpy:
14   Does not check for buffer overflows when copying to destination.
15   Consider using strncpy or strlcpy (warning, strncpy is easily misused). Risk
16   is low because the source is a constant string.
17         strcpy(reserva2, "abacate");
18 Saving hitlist to vulnb.hits
```

Figura 21 - Criando uma lista de riscos (vulnb.hits) do arquivo fonte vulnb.c.

Após salvar uma lista, para melhor exemplificar, no final do próprio código fonte foi adicionado mais uma simples linha contendo mais uma vulnerabilidade. Após esta nova implementação, é possível comparar os riscos do código mais recente (alterado) com a lista de vulnerabilidades anteriormente criada com a opção `--savehitlist` na Figura 21.

Para comparar, é usada a *flag* `--diffhitlist`, tornando mais fácil identificar em qual nova implementação foi introduzido vulnerabilidades. O algoritmo desta opção irá considerar como nova ameaça a partir do momento em que qualquer ameaça (já salva na lista) teve sua linha, coluna, nome de função ou nível de risco alterado. No exemplo da Figura 22, na linha 5 é mostrada a nova vulnerabilidade inserida no código fonte e na linha 10 é emitida a quantidade de novos riscos em comparação com a lista anterior.

```
1  root@bt:~/codigos# flawfinder -Q --context --diffhitlist vulnb.hits vulnb.c
2  Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
3  Showing hits not in vulnb.hits
4  Number of dangerous functions in C/C++ ruleset: 160
5  vulnb.c:20:  [4] (buffer) scanf:
6    The scanf() family's %s operation, without a limit specification,
7    permits buffer overflows. Specify a limit to %s, or use a different input
8    function.
9          scanf("%s", variavel); //nova implementação
10 Hits not in original histlist = 1
11 Lines analyzed = 23 in 0.51 seconds (1642 lines/second)
12 Physical Source Lines of Code (SLOC) = 16
13 Hits@level = [0]   0 [1]   0 [2]   0 [3]   0 [4]   1 [5]   0
```

Figura 22 - Detecção de vulnerabilidade após a nova implementação.

Também é possível visualizar a lista completa das funções presentes no banco de dados do Flawfinder com o parâmetro `--listrules`. Esta lista consiste em funções da biblioteca padrão da linguagem C que proporcionam riscos a aplicação. O número na frente de cada nome das funções representa o nível de risco (de 0 a 5) associado a função. Porém, este nível pode ser variável para algumas funções dependendo da maneira que ela é manipulada no código fonte.

5.3 Análise dinâmica com o Valgrind

Para apresentar um pequeno teste de análise dinâmica, é utilizado um binário, gerado a partir do código fonte `codigo_erros.c` (Figura 23). Este binário é analisado sob opção padrão do Valgrind, que monitora a gestão de memória do programa (memckeck). O programa analisado basicamente cria uma reserva de memória e a preenche com os caracteres de A em diante, por fim imprime.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5
6          char *vetor = (char *) malloc(10);
7          int cont = 0;
8
9          for ( ; cont<10; cont++)
10                 vetor[cont] = 65+cont;
11
12         vetor[cont] = '\0';
13         printf("Valor de vetor: %s\n", vetor);
14
15         free(vetor);
16
17 }
```

Figura 23 - Código para o teste do Valgrind.

Conforme a Figura 24, na compilação com o gcc (linha 1), a opção `-g` é necessária apenas para manter um vinculo do binário com o código fonte, e com isso, disponibilizar informações de depuração (linha que ocorre o problema no código fonte, por exemplo as linhas 4 e 12 desta Figura 24 apontam). No teste, foi utilizada a *flag* `--quiet` para não serem exibidos cabeçalho e rodapé padrões.

Na Figura 24, o Valgrind conseguiu encontrar dois problemas relacionados à memória: escrita e leitura inválida de 1 *byte*. Isso ocorre devido à atribuição do caracter "`\0`" em um *byte* além da capacidade do *buffer* (Figura 23, linha 12), e tentativa de leitura da *string* sem seu caracter de terminação "`\0`" no seu último *byte* (Figura 23, linha 13). Este erro pode ser caracterizado como *off-by-one*.

```
1  root@bt:~/codigos/dynamic# gcc -g codigo_erros.c -o teste.out
2  root@bt:~/codigos/dynamic# valgrind --quiet ./teste.out
3  ==9977== Invalid write of size 1
4  ==9977==    at 0x4005F7: main (codigo_erros.c:12)
5  ==9977==  Address 0x51b004a is 0 bytes after a block of size 10 alloc'd
6  ==9977==    at 0x4C274A8: malloc (vg_replace_malloc.c:236)
7  ==9977==    by 0x4005C5: main (codigo_erros.c:6)
8  ==9977==
9  ==9977== Invalid read of size 1
10 ==9977==    at 0x4E7635E: vfprintf (vfprintf.c:1614)
11 ==9977==    by 0x4E7D539: printf (printf.c:35)
12 ==9977==    by 0x400612: main (codigo_erros.c:13)
13 ==9977==  Address 0x51b004a is 0 bytes after a block of size 10 alloc'd
14 ==9977==    at 0x4C274A8: malloc (vg_replace_malloc.c:236)
15 ==9977==    by 0x4005C5: main (codigo_erros.c:6)
16 ==9977==
17 Valor de vetor: ABCDEFGHIJ
```

Figura 24 - Resultados do Valgrind. Dois problemas detectados em tempo de execução.

Este é um tipo de erro que um analisador estático com técnica de procura por padrões não está capacitado a encontrar (como por exemplo, o Flawfinder). Graças à análise dinâmica é possível monitorar os maus comportamentos no tempo de execução do binário, detectando problemas da gestão de memória, e possíveis vulnerabilidades. Como visto no exemplo, muito útil para resolver problemas que surgem segundo a má interpretação lógica do programador.

5.4 *Checklist* de apoio a programação segura

Um programa por si só não é seguro. A princípio, ele não reconhece o que é segurança ou meios para se auto proteger. Portanto, é preciso evitar vulnerabilidades. Para programar de maneira segura é necessário entender diversos conceitos e deve haver muita atenção no que é especificado para a aplicação fazer.

De maneira resumida, a Tabela 1 apresenta um total de 24 medidas ao programador para obter o mínimo de segurança, e medidas de como evitar pequenos desconfortos, sendo uma mini *checklist* de apoio que o programador pode seguir.

Com base nos autores Correia e Sousa (2008), Dowd, McDonald e Schuh (2006), Erickson (2008), Hoglund e McGraw (2004), Howard e Leblanc (2002), Medeiros (2007), Viega e McGraw (2001), e nos sites OSDEV (2011) e OWASP (2013), foram elaboradas algumas medidas (organizadas em uma *checklist*) para contribuir com a programação defensiva. Esta *checklist* está relacionada às vulnerabilidades enfatizadas no capítulo 3, abrangendo boas práticas, gerenciamento de memória, entrada de dados, cadeia de caracteres, limites, saída de dados e operadores lógicos.

Tabela 1 – *Checklist* de apoio a programação segura.

Medida	**Exemplos**	
	Usar:	**Ao invés de:**

Mantenha uma boa estrutura para o código. Utilize espaçamento, divisões.	`for (int i; i<10; i++) {`
	`  vetor[i] = i+1;`
	`  printf("%d", i);`
	`}`
		`if(vetor != NULL){`
		`for(int i; i<10;i++)`
		`{vetor[i]=i+1;`
		`printf("%d", i); }}`

Medida	Exemplos	
	Usar:	**Ao invés de:**

Indente o código o melhor possível.	`//Comentário`
	`/*`
	`  Comentários`
	`*/`
		-

Evite complexidade além do necessário.	-	-
Evite funções perigosas aparentemente sem solução.	-	-
Se a escolha de funções seguras não for viável, saiba lidar com as perigosas.	-	-
Calcule adequadamente o espaço necessário para qualquer tipo de *buffer*.	-	-

Cheque se não está sendo reservado 0 *bytes* na memória	`int valor = 0;`
	`...`
	`if (valor > 0)`
	`  char *buffer = (char *)`
	`malloc(valor);`
		`int valor = 0;`
		`...`
		`char *buffer = (char *)`
		`malloc(valor);`

Certifique-se de não reservar os *bytes* com número negativo.	`if (valor > 0)`
	`  char *buffer = (char *)`
	`malloc(valor);`
		`int valor = -10;`
		`...`
		`char *buffer = (char *)`
		`malloc(valor);`

Cheque se a reserva dinâmica obteve sucesso.	`if ( ! buffer ) {`
	`  printf("Erro ao`
	`reservar!\n");`
	`  return 0;`
	`}`
		-

Alocações de memória devem ser liberadas adequadamente após não serem mais necessárias.	`free(buffer);`	-

Certifique-se de não liberar espaço de memória que não foi alocado anteriormente.	`char *buffer=(char`
	`*)malloc(10);`
	`fPreencher(buffer);`
	`free(buffer);`
		`char *buffer = (char *)`
		`malloc(10);`
		`free(buffer);`

Atribua `NULL` ao ponteiro quando o mesmo não for mais utilizado.	`buffer = NULL;`	-
Limitar a quantidade de caracteres que o usuário pode informar.	`char buffer[10];` `scanf("%9s", buffer);`	`char buffer[10];` `scanf("%s", buffer);`

Medida	Exemplos	
	Usar:	**Ao invés de:**
Limitar o tipo de caracter de acordo com o tipo de dado do *input*.	-	-
Evitar a inserção de determinados caracteres no *input*.	-	-
Tratar caracteres diferentes após inseridos no *input*.	-	-

O índice dos vetores sempre começam na posição `[0]`.	`char buffer[3];`
	`buffer[0] = 'a';`
	`buffer[1] = 'b';`
	`buffer[2] = '\0';`
		`char buffer[3];`
		`buffer[1] = 'a';`
		`buffer[2] = 'b';`
		`buffer[3] = '\0';`

Sempre inclua um *byte* além do necessário.	-	-

Sempre incluir o caracter de terminação `\0` no final da *string*.	`int MAX = 10;`
	`char *buffer = (char *)`
	`malloc(MAX);`
	`buffer[MAX-1] = '\0';`
		-

Evite funções inseguras que manipulam *string*.	`strlcat(buffer, "joinnnn",` `sizeof(buffer));`	`strcat(buffer, "concatenando");`

Cheque se inteiros `unsigned` não receberão resultados negativos.	`unsigned int buffer;`
	`if (valor > 2)`
	`  buffer = valor - 2;`
		`unsigned int buffer;`
		
		`buffer = valor - 2;`

Tratar os dados de saída antes de disponibilizá-los.	-	-
Sempre formatar adequadamente as informações de saída.	`printf("%s", buffer);`	`printf(buffer);`
Atenção aos operadores lógicos. Utilizar corretamente nas condições e repetições.	-	-

6 Considerações finais

Com base no conteúdo sobre possíveis vulnerabilidades e suas implicações, normas reconhecidas internacionalmente, modelos de desenvolvimento focados em segurança e algumas proteções, é claramente visível a importância de evitar ameaças, especialmente quando elas são expostas em *softwares* cujo grau de importância é relevante. Com isso, é perceptível a necessidade de investir esforço em programação defensiva para evitar introduções de vulnerabilidades que causam impactos a sistemas.

Mesmo que o programador (na pior das hipóteses) não se preocupe em empregar segurança em seu código, algumas proteções adicionais no sistema operacional e no próprio programa podem no mínimo dificultar as tentativas de ataque às vulnerabilidades. Contudo, essas proteções estão sujeitas a serem "burladas" com técnicas mais avançadas de ataque (que não é o foco deste trabalho). Em meio a isso, quando essas proteções dinâmicas entram em ação, elas podem tomar decisões indesejáveis (por exemplo, encerrando a aplicação). Decisões estas que talvez possam não ser a melhor medida no momento em que a aplicação está em operação. Com isso, conclui-se que em relação às proteções dinâmicas, é recomendado evitar estas vulnerabilidades no próprio ato de programar ao invés de deixar a aplicação exposta dependente das ações desses mecanismos que por um lado protegem contra certos ataques, porém podem trazer impactos a operabilidade.

Com base nos testes realizados, é possível concluir que ao utilizar uma ferramenta de análise estática de código como apoio, é poupado muito tempo, e por se tratar de um processo automatizado, dependendo da situação, está menos sujeito a erros imperceptíveis quando comparado com análise manual. A mesma vantagem de encontrar erros também é percebida com a análise dinâmica, detectando falhas de segurança que passam despercebidos aos olhos humanos. A análise manual também possui seu grau de importância, portanto, pode-se dizer que um método de análise complementa o outro em direção ao objetivo principal: detectar erros num *software*.

Por fim, com base referente às vulnerabilidades apresentadas, erros no código fonte e a maneira que as ferramentas trabalham, são reunidas diversas

medidas em uma *checklist* para contribuir minimamente com a programação segura. Com isso, o programador pode prevenir-se de pequenos detalhes que implicam em vulnerabilidades que podem passar despercebidos dependendo do grau de complexidade do código e aprimorar suas técnicas de segurança.

6.1 Trabalhos futuros

Neste trabalho, embora algumas vulnerabilidades bastante comuns tenham sido abordadas, se faz necessário expor outras inúmeras vulnerabilidades e seus diversos riscos à segurança que as diversas linguagens de programação existentes podem oferecer, cabendo até mesmo um estudo comparativo entre as linguagens. Também se faz proveitoso estudar vulnerabilidades dos diversos elementos da superfície de ataque, que podem criar “pontes” de ataque ao *software* alvo. Além de vulnerabilidades, as diversas formas de ataques e exploração das diversas vulnerabilidades podem ser proveitosas, pois é compreendendo o funcionamento de um ataque é que se consegue explorar, e para aprimorar o conhecimento sobre segurança atacar é essencial.

O conteúdo da abordagem sobre engenharia reversa neste trabalho se encontra bastante teórico. A prática é essencial para o desenvolvimento do conhecimento. Todo *software*, depois de desenvolvido, possui fragilidades diante da engenharia reversa com seus diversos tipos de ferramentas para as práticas. O estudo de ferramentas destes tipos se faz necessário, tanto a fim de entender o funcionamento de ataques, quanto o funcionamento de possíveis técnicas de segurança para dificultar estas ferramentas.

Referente às normas, e metodologias focadas no desenvolvimento seguro, é necessário um estudo mais aprofundado sobre seus objetivos, etapas, funcionamentos e práticas. Dentre os modelos de desenvolvimento seguro, atividades relacionadas à segurança como os métodos de análises podem ser estudados com mais profundidade por meio de apresentação de diversas outras ferramentas, e funcionamento mais detalhado de cada método.

Para aplicações já desenvolvidas com vulnerabilidades, é possível realizar um estudo mais amplo sobre os diversos mecanismos de proteção dinâmica a fim de obter resultados em relação à segurança de cada um, suas vantagens e também desvantagens à operação e aplicação/ambiente.

7 Referências bibliográficas

CHESS, B.; MCGRAW, G. ***Static analysis for security***. 2004 - Disponível em: <https://buildsecurityin.us-cert.gov/sites/default/files/bsi5-static_0.pdf>. Acesso em: 29 out 2013.

CHMIELEWSKI, M. et al. **Revisões de código**: encontre e corrija vulnerabilidades antes de lançar seu aplicativo. 2007 – Disponivel em: <http:// msdn.microsoft.com/pt-br/magazine/cc163312.aspx>. Acesso em: 24 out 2013.

COLLBERG, C. S.; THOMBORSON, C. ***Watermarking, tamper-proofing, and obfuscation***: *tools for software protection*. 2002 - Disponivel em: <http://www. cs.auckland.ac.nz/~cthombor/Pubs/01027797a.pdf>. Acesso em: 30 out 2013.

CORREIA, M. P.; SOUSA, P. J. **Segurança no *software*.** Lisboa/PT: FCA, 2008.

CUGLIARI, A.;GRAZIANO, M. ***Smashing the stack in 2010***. *Report for the Computer Security exam at the Politecnico di Torino*, 2010.

DEITEL, Harvey M.; DEITEL, Paul J.; CHOFFNES, David R. **Sistemas operacionais.** 3 Ed. São Paulo: Pearson Education do Brasil, 2005.

DOWD, M.;MCDONALD, J.;SCHUH, J.***The art of software security assessment****: identifying and preventing software vulnerabilities*. New York City/USA: Pearson Education, 2006.

DUBE, T. E. ***Metamorphism as a Software Protection for Non-Malicious Code*** – *Masters thesis*. Air Force Institute of Technology. Ohio/USA, 2006.

ERICKSON, J. ***Hacking****: the art of exploitation*. 2 Ed. San Francisco/USA: No Starch Press, 2008.

FOSTER, J. C. ***Buffer overflow attacks****: detect, exploit, prevent*. Boston/USA: 2005.

HOGLUND, G.; MCGRAW, G. **Como quebrar códigos**: a arte de explorar (e proteger) *software*. São Paulo/BR: Pearson Education do Brasil, 2004.

HOWARD, M. **Uma análise do ciclo de vida do desenvolvimento da segurança na Microsoft**. 2005 - Disponivel em: http://www.microsoft.com/ brasil/msdn/Tecnologias/Seguranca/SDLdefault_US.mspx. Acesso em: 17 out 2013.

HOWARD, M.; LEBLANC, D. ***Writing secure code***. 2 Ed. Redmond/USA: Microsoft Press, 2002.

HOWARD, M.; LIPNER, S. ***The security development lifecycle***. 1 Ed. Redmond/USA: Microsoft Press, 2006.

IBM. ***GCC extension for protecting applications from stack-smashing attacks*** - Disponível em: http://www.research.ibm.com/trl/projects/security/ssp>. Acesso em: 10 out 2013.

JANDL, P. J. **Sistemas operacionais.** USF - Faculdade de Engenharia. Notas de aula, 1999.

KUNST, R.; RIBEIRO, V. G. **Um estudo multicritério para classificação de vulnerabilidades de *software***. 2005 – Disponível em: <http://www.academia. edu/3157688/Um_estudo_multicriterio_para_classificacao_de_vulnerabilidades_de_ software>. Acesso em: 11 set 2013.

LEVINE, J. R. ***Linkers and loaders***. 1 Ed. Massachusetts/USA: Morgan Kaufmann, 1999.

LIPNER, S.; HOWARD, M. **O ciclo de vida do desenvolvimento da segurança de computação confiável**. 2005 - Disponivel em: <http://msdn. microsoft.com/pt-br/library/ms995349.aspx>. Acesso em: 17 out 2013.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de sistemas operacionais**. 4 Ed. Rio de Janeiro/BR: LTC, 2007.

MARIOTTI, F. S.Conhecendo a ISO 15408. **Engenharia de *software* Magazine**. 45 Ed, 2004.

MEDEIROS, I. V. S. **Detecção de vulnerabilidades de inteiros na adaptação de *software* de 32 para 64 *bits***. Dissertação de mestrado em informática. Universidade de Lisboa. Lisboa, 2007.

MICROSOFT. **Central de proteção e segurança**. 2013 - Disponível em: <http:// www.microsoft.com/security/>. Acesso em: 17 out 2013.

MICROSOFT. ***Protecting your software***. 2013 - Disponível em: <http://www. microsoft.com/security/sir/strategy/default.aspx#!section_3_3>. Acesso em 28 out 2013.

MICROSOFT. **Windows XP SP3 e ao Office 2003: suporte termina a 8 de Abril de 2014**. 2013 - Disponível em: <http://www.microsoft.com/business/pt-pt/a-par-e-passo/paginas/ fimsuporte.aspx>. Acesso em: 10 dez 2013.

NIST. ***National vulnerability database version* 2.2**. 2013 - Disponível em: <http://nvd.nist.gov/>. Acesso em: 10 dez 2013.

OSDEV. ***GCC stack smashing protector***. 2011 - Disponível em: <http://wiki.osdev .org/GCC_Stack_Smashing_Protector>. Acesso em: 10 out 2013.

OWASP. ***Command injection***. 2012 - Disponível em: <https://www.owasp.org/index .php/Command_Injection>. Acesso em: 29 out 2013.

OWASP. ***OWASP periodic table of vulnerabilities - integer overflow/underflow***. 2013 - Disponível em: <https://www.owasp.org/index.php/OWASP_Periodic_Table_ of_Vulnerabilities_-_Integer_Overflow/ Underflow> . Acesso em: 30 out 2013.

OWASP. ***Stack overflow***. 2009 - Disponível em: <https://www.owasp.org/ index.php/Stack_overflow>. Acesso em: 20 set 2013.

OWASP. ***Testing for heap overflow***. 2012 - Disponível em: <https://www.owasp. org/ index.php/Testing_for_Heap_Overflow>. Acesso em: 19 set 2013.

OWASP. ***Vulnerability***. 2011 - Disponível em: <https://www.owasp.org/index .php/Category:Vulnerability>. Acesso em: 04 ago 2013.

PEIKARI, C.; CHUVAKIN, A. ***Security warrior***. California/USA: O'Reilly Media, 2004.

PEIXOTO, M. C. P. **Engenharia social e segurança da informação na gestão corporativa**. Rio de Janeiro/BR: Brasport Livros, 2006.

SCHILLING, W. W.; ALAM, M. ***Integrate static analysis into a software development process***. 2006 - Disponivel em: <http://www.embedded.com/design/prototyping-and-development/4006735/Integrate-static-analysis-into-a-software-development-process>. Acesso em: 27 out 2013.

SHIREY, R. **The internet society**. 2000 - Disponivel em: <http://www.ietf.org/rfc/rfc2828.txt?number=2828>. Acesso em: 03 set. 2013.

TANENBAUM, A. S. **Sistemas operacionais modernos**. 3 Ed. São Paulo/BR: Pearson Education do Brasil, 2010.

TEIXEIRA, E. P. L. T. **Ferramenta de análise de código para detecção de vulnerabilidades**. Dissertação de mestrado em engenharia informática. Universidade de Lisboa. Lisboa, 2007.

TIOBE. **TIOBE *Index for december 2013***. 2013 - Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 07 dez 2013

VIEGA, J.; MCGRAW, G. ***Building secure software****: how to avoid security problems the right way*. 1 Ed. New York City/USA: Pearson Education, 2001.

WIKIQUOTE. **Stan Lee**. 2013 - Disponível em: <http://en.wikiquote.org/ wiki/Stan_Lee>. Acesso em: 09 dez 2013.

Anexos

Anexo I – Lista de algumas funções vulneráveis da linguagem C segundo a ferramenta Flawfinder versão 1.27.

access
atoi
atol
bcopy
char
chgrp
chmod
chown
chroot
crypt
curl_getenv
cuserid
drand48
erand48
execl
execle
execlp
execv
execvp
fgetc
fopen
fprintf
fread
fscanf
getc
getchar
getenv
getlogin
getopt
getopt_long
getpass
getpw
gets
getwd
gsignal
jrand48
lcong48
lrand48
lstrcat
lstrcatn
lstrcpy
lstrcpyn
memalign
memcpy
mkstemp
mktemp
mrand48
nrand48
open
popen
printf
random
read
readlink
readv
realpath
recv
recvfrom
recvmsg
scanf
seed48
setstate
snprintf
sprintf
srand
srandom
sscanf
ssignal
strcadd
strcat
strccpy
strcpy
streadd
strecpy
strfry
strlen
strncat
strncpy
strtrns
swprintf
syslog
system
tempnam
tmpfile
tmpnam
ulimit
umask
usleep
vfork
vfprintf
vfscanf
vfwprintf
vprintf
vscanf
vsnprintf
vsprintf
vsscanf
vswprintf
vwprintf
wchar_t
wcscat
wcscpy
wcslen
wcsncat
wcsncpy