UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Ferramenta de Análise de Código para Detecção de Vulnerabilidades

Emanuel Pedro Loureiro Teixeira

MESTRADO EM ENGENHARIA INFORMÁTICA

Setembro de 2007

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Ferramenta de Análise de Código para Detecção de Vulnerabilidades

Emanuel Pedro Loureiro Teixeira

Projecto orientado pelo Prof. Dr. Nuno Fuentecilla Maia Ferreira Neves

MESTRADO EM ENGENHARIA INFORMÁTICA

Setembro de 2007

Aos Alunos de Ensino Especial

Agradecimentos

Em primeiro gostava de agradecer ao Professor Doutor Nuno Ferreira Neves pelos seus ensinamentos e comentários durante o desenrolar do projecto de investigação e escrita desta tese. Assim como, a sua amabilidade em representar de forma desconhecida o aluno Emanuel Teixeira.

Agradeço também ao Professor Doutor Dinis Pestana pelos seus concelhos e orientações, como também na revisão desta tese.

Ao Mestre João Antunes pelo seu tempo em auxiliar-me na construção do teste padronizado.

Aos colegas Mestre João Leitão e Inês Fragata pelos comentários de revisão da tese e apoio ao trabalho realizado.

A todos os Professores que me apoiaram durante a minha formação escolar, tendo em consideração as seguintes escolas: Escola Secundária da Amadora, Escola Secundária Seomara da Costa Primo (Amadora) e Escola Básica 2+3 Dom Francisco Manuel de Melo (Amadora), Obrigado pelos apoios que me foram concedidos. Como também, estimo os Professores da Universidade de Lisboa da Faculdade de Ciências, em particular ao Departamento de Informática, Obrigado.

Um obrigado muito especial para os meus Pais António & Florinda Teixeira, ao qual despenderam o seu tempo das suas vidas, na educação de um ser autodidacta, e um apreço especial ao meu Irmão Rodrigo Teixeira.

Resumo

A análise de código é um mecanismo de apoio à detecção de anomalias ou erros de concretização existentes num ambiente de programação. Diversos factores podem levar ao aparecimento destes erros, destacando-se entre eles a programação inadequada, o conhecimento limitado das interfaces e outros factores inerentemente humanos, como o esquecimento. As ferramentas de análise podem então ter um impacto positivo no ciclo de desenvolvimento de um produto, permitindo poupar tempo, dinheiro e contribuindo para que a aplicação seja construída sem vulnerabilidades de segurança. A análise de código pode ser efectuada em três modos: *estática*, *dinâmica* ou *manual*; cada um destes modos têm as suas vantagens e desvantagens, sendo os mais usados os *estáticos* e os *dinâmicos*.

Nesta tese aborda-se o problema de detecção de vulnerabilidades de segurança através de ferramentas de análise estática, focadas para a linguagem de programação C. A tese propõe um teste que permite avaliar e comparar o desempenho de diversas ferramentas de análise estática, nomeadamente em relação ao número de falsos alarmes por elas despoletadas. A tese descreve também uma nova ferramenta designada por Mute, que tem como objectivo uma melhor eficácia e precisão na detecção de um conjunto alargado de vulnerabilidades. Genericamente, o Mute utiliza um mecanismo de agregação de resultados produzidos por várias ferramentas existentes, para decidir da existência ou não de uma vulnerabilidade. A tese inclui ainda uma avaliação comparativa do Mute e de diversas outras ferramentas.

Palavras Chave: Análise Estática de Código Fonte, Detecção de Vulnerabilidades, Teste Padronizado, Classes de Vulnerabilidades.

Abstract

Code analysis is a tool aiding the detection of anomalies created during the development of software by programmers. These errors can appear due to various factors, namely bad programing skills, limited knowledge of the interfaces, and human errors. This important tool brings in a new alignment in the development cycle, the main goals being to save time and expenditure, since it improves the final quality, by allowing the product to be built with less security vulnerabilities, ideally without security vulnerabilities. Code analysis can be worked out in three ways: *static*, *dynamic*, or *manual*. Each of those choices has advantages and disadvantages; the two most widely used are *static* and *dynamic* analysis.

In this dissertation we focus on vulnerabilities detection using static code analysers, and we choose to analyse C code. The thesis approach with a benchmark that allows to evaluate and compare the performance of some static analyzers, in concern to the number of false positives. The tool we shall develop will give a new and better perspective of their usefulness, with recall and precision. Our goal, in developing this new tool, is to decrease the number of false alerts, which is the main problem of static code analysers used so far; this tool, called Mute *"Multiple Tools"* uses several techniques to archive this goal.

Keywords: Static Analysis in Source Code, Vulnerabilities Detection, Benchmark, Taxonomy of Vulnerabilities.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Excertos de Código

Capítulo 1

Introdução

O desenvolvimento da ARPANET (Roberts, 1986) proporcionou o aparecimento dos sistemas distribuídos[1], e com estes a possibilidade de diversos indivíduos dispersos fisicamente poderem comunicar e cooperar.

Inicialmente a segurança era um elemento em falta, não só porque não era idealizada, mas devido a más concretizações, neste sentido levou ao aparecimento de inúmeras vulnerabilidades no código fonte (Chess & McGraw, 2004). Por conseguinte, os sistemas vulneráveis tornavam-se alvos fáceis de explorar, tanto que os adversários somente precisavam de escolher o tipo de ataque a realizar no sistema alvo.

As falhas na concretização poderiam dever-se tanto às capacidades limitadas dos compiladores, em detectar a totalidade das vulnerabilidades presentes no código fonte, ou mesmo devido há falta de diligência dos programadores (por exemplo, o uso de funções vulneráveis da biblioteca *libc* — ver Apêndice C e Stallman *et al.* (2006)). Assim, tornou-se imprescindível o aparecimento de mecanismos para rever eficazmente o código fonte.

Um dos primeiros passos para a eliminação de vulnerabilidades, passou pela revisão manual de código fonte. Nesta técnica, utilizada há mais de duas décadas, um conjunto de pessoas com um vasto conhecimento sobre a aplicação a desenvolver, examinam linha a linha o código da aplicação, de modo a encontrar possíveis falhas de segurança. Este processo é muito moroso e caro, daí surgirem dois tipos de revisão por meio computacional — a análise estática e dinâmica.

[1] Dando origem há *Internet*.

Este trabalho focou-se nas ferramentas de análise estática, em detrimento das de análise dinâmica, porque estas permitem potencialmente a detecção de um maior número de vulnerabilidades, de forma exaustiva, precisa (não eliminando nenhuma linha de código) e rápida.

Este tipo de ferramentas têm uma grande utilidade na fase de testes, altura em que normalmente são usadas para detecção de erros na aplicação. Elas também podem ser empregues durante a concretização, onde o programador encontra a vulnerabilidade revelada pela ferramenta, removendo-a imediatamente. Por isso, convém que os programadores tenham acesso a uma ferramenta capaz de reportar todo o tipo de vulnerabilidades num só processo.

Actualmente, a maioria das ferramentas de análise estática disponíveis na *Internet* estão vocacionadas para a detecção de certos tipos específicos de vulnerabilidades. Por outro lado, não existe uma maneira eficiente de se compararem as capacidades efectivas de cada ferramenta, tornando difícil a escolha entre elas. Assim, com este trabalho pretende-se realizar uma análise comparativa das várias ferramentas existentes[1], e produzir uma ferramenta teoricamente melhor, mas que ao mesmo tempo utilize o que há disponível. Afim de avaliar as ferramentas, estabeleceu-se uma versão preliminar de um teste padronizado (do inglês, *benchmark*) com vista a apreciar cada uma de igual modo. Depois, desenvolveu-se uma nova ferramenta, Mute, que tem mostrado alguma superioridade em alguns testes, nomeadamente em relação à eficácia e precisão.

1.1 Motivação

Uma vez que as vulnerabilidades aumentam ano após ano (ver Secção 2.6.1), é importante produzir mecanismos que possam de alguma maneira contribuir para a diminuição destes erros nas aplicações.

Ao longo da última década, a análise estática tem evoluído consideravelmente. Algumas das ferramentas têm surgido de um processo de investigação e evoluído para um produto industrial, como é o caso do *Coverity* (Coverity, 2006). Porém, também existem ferramentas para uso interno a uma instituição, em conjunto

[1] Ferramentas para análise estática de código fonte com o seu código fonte disponível na *Internet*.

com outros produtos de desenvolvimento de aplicações, por exemplo *PREFast* da Microsoft (Larus *et al.*, 2004).

A análise estática de código fonte é uma solução apelativa, dado que, as aplicações muitas vezes não são construídas por base em especificações de segurança, além de que, as instituições pensam em "Concretizar a aplicação em primeiro, depois vem a segurança (...)" (Haffner *et al.*, 1998).

As ferramentas de análise estática de código fonte conseguem detectar vulnerabilidades, mostrando ao utilizador a localização destas. No entanto, correntemente não há ferramentas capazes de detectar um grande número de diferentes tipos de vulnerabilidades, daí, um indivíduo tem de recorrer a várias ferramentas para analisar o seu código fonte. Adicionalmente, as ferramentas costumam exibir falsos alertas, tornando-se assim importante criar métodos que de certa forma possam reduzir o seu número (a investigação de um falso alerta acaba por consumir tempo que poderia estar a ser usado em alguma actividade útil).

Nesta tese optou-se por se focar na linguagem C, porque esta é uma das linguagens mais usadas[1], especialmente ao nível do sistema. Por outro lado, como o C permite uma grande flexibilidade, facilita a introdução de vulnerabilidades no código.

1.2 Contribuição do Trabalho

A análise estática de código fonte é utilizada para se detectarem vulnerabilidades. Este método de análise é interessante, dado que é capaz de encontrar diversos tipos de vulnerabilidades, pesquisa em todas as partes do código, além de que fornece resultados com uma elevada rapidez.

Actualmente, existem várias ferramentas de análise estática disponíveis na *Internet*, que tipicamente se concentram na localização de certas classes de vulnerabilidades. Estas ferramentas utilizam técnicas de análise distintas, e como seria de esperar, têm características diferentes em termos de detecção.

[1] Classificação das linguagens mais utilizadas em, `http://www.tiobe.com/tpci.htm` (visitado em Setembro de 2007).

Infelizmente, neste momento não existem bons métodos que possibilitem a comparação das ferramentas, o que dificulta a decisão por parte dos programadores em relação a que ferramentas devem empregar. Neste sentido, a primeira parte do trabalho foi desenvolver um teste padronizado que permitisse avaliar as ferramentas em relação às suas capacidades. Em particular, foram considerados dois aspectos fundamentais dos resultados produzidos pelas ferramentas: o grau de falsos positivos (ou falsos alertas) que são gerados, isto é, o número de vulnerabilidades que são indicadas mas que na realidade não existem; a grandeza de falsos negativos que são observados, isto é, o número de vulnerabilidades que a ferramenta é incapaz de reportar. Como é de esperar, uma ferramenta será tanto melhor (ou exibirá um comportamento superior) quanto menores forem ambos os factores avaliados.

A segunda parte do trabalho residiu na avaliação de várias ferramentas existentes, e na criação de uma nova ferramenta mais abrangente, chamada de Mute, capaz de reportar mais vulnerabilidades e de tipos distintos que as anteriores. A solução empregue na construção do Mute baseou-se em se juntar os dados resultantes da análise de um mesmo código fonte, por um conjunto de ferramentas existentes. É de notar, no entanto, que a agregação de resultados acaba por ser um problema com alguma complexidade, porque o objectivo seria que a nova ferramenta tivesse um menor número de falsos positivos, mantendo os verdadeiros positivos. A avaliação do Mute através do teste padronizado demonstrou que este objectivo era passível de ser atingido.

1.3 Metodologias e Calendarização do Trabalho

Esta secção explica o processo de desenvolvimento adoptado, e como é que o trabalho foi organizado e escalonado. A escolha do processo de desenvolvimento acaba por ser muito importante porque determina a qualidade de concepção, para além da sua escolha ter implicações no planeamento (Braude, 2001).

Para este projecto também foi realizado uma análise de riscos, para prevenir perturbações no desenrolar do trabalho, ver Apêndice A.

1.3.1 Processo de Desenvolvimento

O processo de desenvolvimento serve para organizar as diversas etapas que levam ao produto final. Essa organização faz com que as pessoas envolvidas no projecto se sintam confortáveis, orientadas e não tenham quebras de produtividade. Existem diversas etapas que um projecto deve seguir (Braude, 2001):

1. Compreender a natureza e propósito da aplicação;
2. Seleccionar o processo de desenvolvimento e criar um plano;
3. Recolha e análise de requisitos;
4. Desenho e construção da aplicação;
5. Testar a aplicação;
6. Entregar e manter a aplicação.

Estas etapas enunciadas podem ser vistas na Secção 1.3.2, onde estão detalhadas para o trabalho a realizar. De seguida, descrevem-se vários processos de desenvolvimento, que foram considerados no início deste trabalho (Dix *et al.*, 2004) (Braude, 2001):

Cascata O ciclo de desenvolvimento passa por: análise de requisitos, desenho, concretização, testes e manutenção. O processo de desenvolvimento é feito em modo linear e unidireccional. Os resultados de cada fase são analisados com o requerido. A maioria dos projectos reais raramente se adaptam ao modelo cascata porque é difícil de capturar todos os requisitos de uma vez. É aplicado a pequenos projectos com duração aproximada de 5 a 12 meses.

Espiral O processo passa pela análise de requisitos, desenho, concretização e testes. Depois de uma etapa, volta-se ao início, iterativamente até à conclusão do produto final ou há não existência de novos requisitos. Consegue-se obter um produto rápido, embora necessite de mais gestão que no modelo cascata, porque a documentação tem que ser consistente após cada iteração. Normalmente, usa-se este modelo porque se consegue controlar melhor as anomalias e é difícil começar uma iteração sem acabar outra.

Incremental As fases do processo são: análise de requisitos, desenho, concretização, teste das componentes, integração e teste do sistema. É um processo iterativo, quando se acaba uma fase volta-se para outra, tipicamente uma fase pode corresponder a uma ou duas semanas (fases muito curtas com pequenos incrementos). Teoricamente, os incrementos podem ser trabalhados em paralelo, embora, na prática seja difícil dos sincronizar. Este modelo é muito utilizado em grandes empresas, onde um projecto pode levar 2 ou mais anos. O cliente tem sempre conhecimento da fase actual e próxima. Neste método é possível antecipar alterações de forma rápida.

Prototipagem O seu ciclo consiste em obter os requisitos do utilizador, construir, rever o protótipo, testar e ensaiar com o utilizador. É um processo iterativo, que se baseia em criar ilusões no utilizador. Fazer protótipos implica gastos, devido às constantes alterações até se concretizar o sistema.

RAD(2) Neste processo as etapas são: modelação de negócio, dados, processos, geração da aplicação, testes e entrega. Utilizado para desenvolver rapidamente uma aplicação num modo sequencial. Direccionado para projectos que possam durar entre 60 a 90 dias.

O trabalho desenvolvido nesta tese não tem uma grande dimensão, nem uma duração significativa. Como não é um trabalho em grupo, onde cada elemento produz a sua parte e todos têm que se sincronizar, leva a que a documentação seja gerida por uma única pessoa e o mesmo se passa com a análise de requisitos. Assim, não requer uma coordenação precisa, intervindo apenas o coordenador e orientando, deste modo, basta ao próprio aluno seguir as várias etapas do processo de desenvolvimento e chegar ao produto final.

Este projecto é visto de uma forma sequencial, ou seja, define-se o trabalho a desenvolver, concretiza-se e testa-se. Portanto, dos vários processos de desenvolvimento optou-se por o modelo Cascata. Este é o mais adequado ao projecto, pois os requisitos não sofreram grandes alterações durante o desenvolvimento, a documentação é controlada por uma única pessoa, e além disso a descrição das várias etapas a desenvolver assemelha-se ao modelo Cascata.

1.3.2 Calendarização

Este trabalho foi desenvolvido no grupo de investigação Navigators[1], no Laboratório de Sistemas Informáticos de Grande Escala — LASIGE, na Faculdade de Ciências da Universidade de Lisboa. A entidade acolhedora é responsável pela supervisão e também pela orientação do candidato a Mestre. Do acordo entre as duas entidades estabeleceu o seguinte planeamento:

1. Estudo de diversas tecnologias e ferramentas de permitem a análise estática de código fonte com o fim de se detectarem vulnerabilidades de segurança; com selecção de uma ou mais ferramentas (ou seus componentes) para testes;

2. Concretização de uma ferramenta que integre as tecnologias avaliadas que forem consideradas mais promissoras;

3. Teste e avaliação experimental da ferramenta desenvolvida e eventualmente adição de melhoramentos;

4. Elaboração de um relatório sobre o trabalho realizado.

O projecto teve início em 1 de Outubro de 2006 e foi concluído em 1 de Julho de 2007.

Como se pode observar no planeamento para o trabalho a realizar, tudo se encontra bem especificado e delineado. Primeiro, efectuou-se uma análise da temática em questão, antes de concretizar elegeu-se a arquitectura da ferramenta a desenvolver. Após a concretização, realizou-se os testes necessários para avaliar o trabalho desenvolvido. Por fim, tendo em conta o objectivo do trabalho, descreveu-se como foi concretizado, e que conclusões se pode tirar na sua contribuição para a comunidade.

A elaboração de um plano de estudo é muito importante, à vista disso dá sentido de orientação e fortifica o trabalho a realizar. O plano descrito em cima pode ser visto no Apêndice B em mais detalhe. No Apêndice B encontram-se dois mapas de Gant, cujas propriedades ilustram o trabalho desenvolvido ao longo dos

[1] Mais informação sobre o grupo em, `http://www.navigators.di.fc.ul.pt` (visitado em Setembro de 2007).

nove meses de mestrado. No entanto, durante o decorrer do projecto foram feitos alguns ajustes, para tornar as tarefas mais explicitas e exactas temporalmente, ou seja, sem haver alterações/prejuízos temporais, detalhou-se a tarefa de concretização da ferramenta sempre mantendo uma coerência com o planeado.

1.4 Organização da Tese

O documento encontra-se estruturado da seguinte forma:

Capítulo 2 Apresenta as vantagens da utilização de análise estática de código e aborda o contexto onde esta é aplicada;

Capítulo 3 Descreve a arquitectura Mute e a interacção entre os módulos, é apresentado o teste padronizado e as propriedades que o definem e também descreve as classes de vulnerabilidades;

Capítulo 4 Apresenta e discute os resultados obtidos por avaliação da ferramenta Mute, utilizando resultados comparativos com outras ferramentas existentes, por forma a auxiliar o leitor na asserção das vantagens do Mute;

Capítulo 5 Conclui o trabalho e apresenta algumas linhas orientadoras para trabalho futuro.

Capítulo 2

Objectivos e Contexto do Trabalho

Neste capítulo é apresentado o ambiente geral onde está enquadrado este trabalho, evidenciando a influência da análise estática durante o desenvolvimento de aplicações. É confrontada a necessidade de utilizar ferramentas de análise estática de código fonte, com a utilidade dos compiladores para a detecção de vulnerabilidades. São apresentados também alguns aspectos relativos à gestão da memória, uma vez que estes são essenciais para a compreensão da vulnerabilidade de memória sobrelotada. Por fim, descreve-se brevemente as vulnerabilidades mais comuns, visto que, estão por base de grande parte dos erros existentes.

2.1 Importância da Análise Estática no Processo de Desenvolvimento

Uma aplicação tem diversas fases pelas quais passa durante a sua construção. A programação de código fonte é uma das fases em que existe maior probabilidade de se inserirem vulnerabilidades (outra seria na configuração da aplicação no ambiente de produção). Os programadores cometem pequenos erros com alguma frequência, mas muitos dos lapsos que são cometidos não têm consequências maiores. Muitas vezes os programadores confiam no compilador para os detectar, possibilitando depois a sua correcção. Porém, o compilador não detecta tudo, e as vulnerabilidades de segurança podem continuar no código durante muito tempo

até serem descobertas. Tipicamente, quanto mais tarde for feita essa detecção, mais cara fica a sua correcção (Chess & McGraw, 2004).

Há pelo menos duas maneiras de construir código seguro:

1. criar directivas para uma boa concretização e/ou;
2. recorrer a uma ferramenta de análise estática.

As duas maneiras são complementares, sendo desejável a utilização de ambas. Idealmente as directrizes devem ser utilizadas durante a fase de concretização, diminuindo o tempo que se gasta com correcções à posteriori. A análise também deve ser empregue durante a concretização para a eliminação dos erros. Caso não se usarem estes métodos, será conveniente na fase de testes criar testes específicos de segurança. Todavia, segundo Heffley & Meunier, mesmo que um sistema tenha sido desenvolvido com toda a preocupação de segurança, alguns problemas podem permanecer ocultos. No entanto, vale sempre apena reduzir o número de vulnerabilidades existentes no código, porque se diminui o risco do sistema ser alvo de um ataque bem sucedido (Heffley & Meunier, 2004).

Na fase de testes a maior causa de insegurança deve-se há má especificação, porque se descartam situações que se pensam impossíveis de acontecer, ou com fraca probabilidade de acontecerem (Viega & McGraw, 2006). Existem muitos problemas em delinear testes, mas a ideia geral é construí-los para validar a funcionalidade da aplicação, isto é, verificar se faz o pretendido. A segurança pode ser vista como um módulo individual de desenvolvimento no produto final e como tal, também precisa de ser testada (ArKin *et al.*, 2005). Na realidade não é o que acontece, primeiro faz-se testes para validar os vários módulos da aplicação e se a segurança não faz parte, não se testa ou não se testa devidamente.

Testes relacionados com segurança são testes muito particulares e muito trabalhosos de se especificar, uma vez que têm a tendência em trabalharem a um alto nível de abstracção. Para ArKin, Stender & McGraw, as ferramentas de análise deveriam fazer parte do processo de testes, pois têm características muito ricas e são capazes de eliminar a maioria dos erros, diminuindo também o tempo de testes e de correcção (ArKin *et al.*, 2005).

2.2 Qualidade numa Aplicação

A qualidade de uma aplicação depende de diversos factores, desde logística a culturais. Há dois conceitos que influenciam a qualidade numa aplicação, a *validação* e a *verificação* (V&V).

O grau de qualidade da aplicação pode ser determinado através de testes. Os testes a executar são muito específicos e dependem do conceito V&V a aplicar (Kit, 1995). Na validação pergunta-se: *"Estamos a construir a aplicação desejada?"*, na verificação: *"Estamos a construir a aplicação correctamente?"*. Isto é, a *validação* permite avaliar o correcto funcionamento da aplicação para um determinado conjunto específico de testes, esperando-se que os resultados obtidos sejam os correctos; enquanto que a *verificação* analisa os mecanismos utilizados durante o funcionamento da aplicação, com vista à satisfação das necessidades do utilizador e dos requisitos previamente especificados (Braude, 2001). Este processo irá determinar o correcto funcionamento da aplicação e consequentemente a sua qualidade.

Existem várias razões para se realizar testes às aplicações, podendo estes serem categorizados de diversas formas, tais como: aceitação, conformidade, usabilidade, desempenho, confiabilidade e robustez (Kit, 1995) (Dix *et al.*, 2004). Contudo, as metodologias usadas nos testes das aplicações em geral, são correctas se os testes efectuados estiverem de acordo, com a especificação dos requisitos da funcionalidade. Isto faz com que uma falsa sensação do correcto funcionamento seja transmitida, tanto aos analistas como aos utilizadores, uma vez que uma aplicação pode estar completamente compatível com o definido nos seus requisitos e ainda assim executar outras acções inesperadas e inseguras.

Assim sendo, o comportamento adequado de uma aplicação deve ser analisado também de outra maneira, de forma a abranger falhas de segurança. Pode-se diferenciar os *testes de uma aplicação*, que tratam da execução da aplicação para determinar a qualidade em conformidade com requisitos e/ou especificações, e os *testes de segurança*, que abrangem as falhas causadas por defeitos inerentes (exemplo, de programação ou configuração) (Whittaker, 2002). Portanto, dizer que uma aplicação é correcta pode não significar que seja segura (Whittaker & Thompson, 2003).

Por isso, para se garantir a segurança da aplicação é necessário desenvolver testes específicos, e complementá-los com a análise de outras ferramentas. Por exemplo, a análise de código mesmo que não sendo perfeita, ajuda porque ao se detectar um número de vulnerabilidades, reduz-se o esforço a despender nos testes.

2.3 Compiladores

Antes de surgir o compilador, existia o analisador sintáctico, ou seja, havia uma ferramenta que validava a sintaxe usada num programa de acordo com a linguagem usada na sua programação. O primeiro compilador surgiu na década de 50, para a linguagem Fortran, pela equipa de John Bacus[1], dando origem a um longo estudo sobre compiladores.

A definição típica de um compilador diz que, "*É um programa que traduz uma representação de alto nível para uma representação de baixo nível.*" (Langlois, 2003). Por outras palavras, o compilador é um programa capaz de traduzir um conjunto de ficheiros que tipicamente se encontram escritos numa linguagem de programação (alto nível[2]) para uma linguagem máquina (baixo nível[3]), convertendo uma representação para outra, e gerando um executável.

Um exemplo de um compilador conhecido é o GCC[4] da GNU. Este suporta diversas linguagens de programação, tais como: Ada, C, C++, Fortran, Java e Objective-C.

2.3.1 Estrutura do Compilador

É fundamental compreender os mecanismos subjacentes a um compilador, uma vez que estes traduzem o código fonte para um código executável (Azul, 1998). Os compiladores modernos têm um conjunto de fases consecutivas, onde cada

[1] Mais informação sobre a história dos compiladores em, `http://www.economicexpert.com/a/Compiler.html` (visitado em Setembro 2007).

[2] Linguagem de alto nível - linguagem de programação com um nível de abstracção relativamente elevado, longe do código máquina e mais próximo à linguagem humana.

[3] Linguagem de baixo nível - trata-se de uma linguagem de programação mais próxima ao código da máquina e portanto, menos abstracta.

[4] Sobre o compilador consulte, `http://gcc.gnu.org/` (visitado em Setembro de 2007).

uma analisa ou transforma o código do programa, e cede o seu resultado para a fase seguinte.

A maioria dos compiladores pressupõe as seguintes fases (Aho *et al.*, 1996):

1. **Fases de Análise (Lexical, Sintáctica e Semântica)** Têm como principal objectivo detectar se algo no código fonte se encontra mal especificado. Na **Análise Lexical** os caracteres do código fonte são lidos e agrupados num fluxo de cadeia de caracteres com um significado lógico, e remove os espaços em branco e comentários. O resultado desta filtragem é transmitida à fase seguinte, a **Análise Sintáctica**, que transforma os dados recebidos numa árvore abstracta de sintaxe (estrutura sintáctica do programa), que é construída por meio de regras gramaticais da linguagem. Depois, na **Análise Semântica** é verificada a semântica do código fonte, há procura de incoerências (exemplo, se as variáveis são declaradas antes de serem usadas ou a consistência entre tipos, etc), por meio da árvore criada. Todas estas fases de análise são efectuadas sequencialmente.

2. **Geração de Código Intermédio** Basea-se em traduzir a árvore de sintaxe da fase anterior para uma representação intermédia (antes do código executável), independente da máquina que se pretende criar o executável.

3. **Optimização** É a penúltima fase da compilação do código fonte (nesta fase, o código fonte já tem uma representação). É a fase mais complicada e a mais importante, pois visa melhorar o código intermédio, com vista a tornar o futuro código executável mais rápido e seguro. Existem diversos tipos de optimizações como: detectar código que nunca é utilizado, eliminar atribuições repetidas, etc. Uma das consequências da optimização é a redução do código intermédio.

4. **Geração de Código** Nesta fase tem-se uma representação do código fonte sem erros e melhorada, que irá servir para converter em código executável. Esta conversão consiste em três etapas: na primeira converte-se o código intermédio para código assembley; na segunda, há medida que se obtém o código assembley, gera-se o ficheiro objecto; por último, utiliza uma ferramenta de ligação de ficheiros objectos, com o intuído de criar um único

ficheiro objecto que corresponde ao executável[1]. A fase de optimização pode participar na construção do executável.

Este processo é o mais utilizado pelos diversos compiladores modernos, embora, alguns não pressupõem a fase de optimização, já que esta não é trivial (Ramalingam, 1994). As fases da compilação podem ser classificadas em duas partes (Langlois, 2003): as fases de análise (*font end*) e as fases de geração de código (*back end*). A primeira parte corresponde às duas primeiras fases descritas anteriormente, a segunda às restantes fases.

2.3.2 Exemplo de Compilador: GCC

O GCC é um compilador que tem o seu código fonte disponível, suporta múltiplas linguagens de programação e mantém uma boa eficiência com grandes projectos. O primeiro GCC (versão 1.0) foi apresentado em 1987 como parte do projecto da GNU e da Free Software Foundation. O seu primeiro autor foi Richard Stallman que começou a desenvolvê-lo em 1985. Mais tarde, o compilador foi reescrito por Len Tower e Stallman (Puzo, 1986), e foi apresentado em 1987[2], tornando-se um produto livre de consumo. Desde essa altura, o GCC tem sofrido melhoramentos e neste momento encontra-se na versão 4.2. Ao longo das diversas versões, o GCC foi-se tornando mais sofisticado, devido ao facto de ser uma aplicação livre e um compilador muito utilizado na área da investigação científica.

2.3.2.1 Estrutura

O GCC possui uma arquitectura de encadeamento (do inglês, *pipeline*), como se pode ver na Figura 2.1. Esta arquitectura foi usada nas versões 1 a 3 do GCC, tendo sido alterada significativamente na versão 4.0, em Julho de 2005.

A extremidade inicial do compilador interpreta os ficheiros e constrói uma representação intermédia do código fonte (Merrill, 2003), correspondendo às duas primeiras fases enunciadas na Secção 2.3.1. Os dados obtidos desta etapa são

[1] Informação sobre o conteúdo de um ficheiro objecto em, `http://www.iecc.com/linker/linker03.html` (visitado em Setembro de 2007).

[2] É possível ver todas as versões e suas datas de lançamento em, `http://gcc.gnu.org/develop.html#timeline` (visitado em Setembro de 2007).

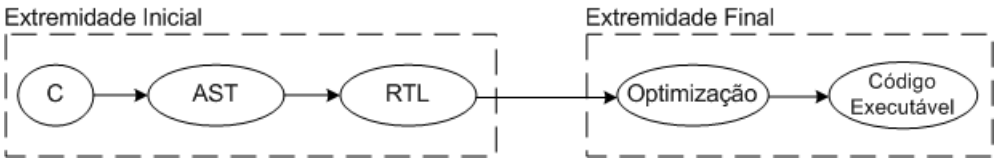


Figura 2.1: Arquitectura das versões anteriores à 4 do GCC.

passados para a fase de optimização, que aplica um número de optimizações, por forma a melhorar o desempenho do código executável gerado.

Nas versões anteriores à 4.0, não existia uma representação intermédia em comum, e cada extremidade inicial utilizava a sua própria representação numa árvore. Esta era depois convertida numa representação de baixo nível, designada por Linguagem de Transferência de Registos (RTL), que é a representação utilizada na passagem para a extremidade final do compilador. Estas versões sofriam de um problema durante a fase da optimização, onde muitas das optimizações requeriam informação de alto nível, o que era difícil de obter com o RTL. Deste modo era difícil concretizar optimizações mais complexas e/ou avançadas (Sotirov, 2005). Sendo assim, decidiu-se desenvolver uma nova arquitectura, com passos mais genéricos para auxiliar as futuras optimizações.

2.3.2.2 Projecto da Árvore Estática de Atribuições Únicas

Este projecto foi uma consequência do mau desempenho das versões anteriores à 4.0 do GCC. O problema encontrava-se no RTL, que não tinha toda a informação do programa, informação esta que se encontra numa representação de alto nível e que o RTL não conseguia reproduzir (Sotirov, 2005).

Uma vez que o problema não podia permanecer, arranjou-se uma solução que passava por uma nova arquitectura para o compilador GCC, como se pode observar na Figura 2.2. Na nova arquitectura, surge a plataforma de suporte Árvore SSA, no qual, irá resolver a maioria dos problemas, relacionados com a optimização.

A Árvore SSA começou a ser desenvolvida em 2000 e consiste numa plataforma de suporte, utilizada entre o GIMPLE e o RTL, onde suporta uma linguagem independente (SSA) para uma optimização de alto nível (Novillo, 2003).

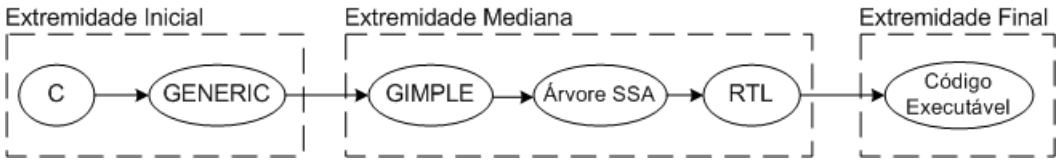


Figura 2.2: Nova arquitectura do GCC versão 4.

As grandes vantagens de criar a nova plataforma de suporte são: a simplificação da representação intermédia, as infra-estruturas comuns para uma análise de alto nível e a possibilidade de criar novos métodos de optimização mais complexos.

2.3.2.3 Opções do Compilador

Existem inúmeras opções no GCC para os efeitos mais diversos, desde o dialecto até ao código a gerar (Stallman & Community, 2005). Desse conjunto de opções as que têm maior relevância para este trabalho, são as que envolvem problemas de segurança. Cada opção, força o compilador à procura de problemas específicos, para protecção do código fonte contra possíveis vulnerabilidades. Alguns dos exemplos mais significativos são:

-Wformat=2 Famílias do *printf()* ou *scanf()* que não têm a formatação esperada;

-Wuninitialized Variáveis que não foram inicializadas. Esta opção só é activada com -O ou -O -Wall;

-Wdiv-by-zero Há divisão por zero;

-Wconversion Atribuições incorrectas (*unsigned sort* a = -1);

-Wunreachable-code Código que já mais será executado durante a execução do programa;

-Wdeclaration-after-statement Quando é declarado algo depois de uma instrução;

-Wpointer-arith Operações que dependem do tamanho "*size of*" do conteúdo;

-Wbad-function-cast Quando uma função chama outra e o valor a receber não é o especificado;

-Wcast-qual Um tipo qualificativo (*const*) é convertido para um tipo não qualificativo;

-Wcast-align Quando um apontador é convertido para um tipo de dados, e o espaço em memória destino é maior que o original (exemplo, *char** para *int**);

-Wsign-compare Quando se comparam tipos de dados com sinais diferentes (exemplo, *signed* com *unsigned*).

A opção do GCC mais conhecida é o **-Wall**. Esta opção engloba várias opções úteis, como a detecção de código que não é utilizado, problemas de conversão entre tipos de dados (*cast*), etc. Porém, não inclui as opções descritas anteriormente. Existe ainda uma outra opção relevante, a **-Wextra**, que apesar de não alertar sobre problemas de segurança, esta opção activa opções que não pertencem à opção *-Wall*. Estas opções auxiliam o programador a rever o seu código, levando a que este possa ser mais optimizado ou coerente.

Existe uma outra opção muito apelativa: o FORTIFY_SOURCE. Apesar de ser pouco conhecida, esta opção é uma característica adquirida pelo GCC e pela *glibc* que permite detectar problemas relacionados com memória sobrelotada, quer em tempo de compilação como em tempo de execução (Oliveira, 2006).

2.3.3 Compiladores vs. Ferramentas de Análise Estática

Os compiladores foram desenvolvidos para efectuarem análise sintáctica e semântica da linguagem de programação e converter código fonte para código executável. Daí, o compilador não tem obrigação de detectar problemas de segurança no código fonte, mas esta actividade pode ser vista como uma extensão ou complemento. Visto que utiliza técnicas sofisticadas (exemplo, análise de fluxo de dados (Sotirov, 2005) e propagação de intervalos de valor (Patterson, 1995)) para detectar diversos problemas, seria útil que se aproveitasse a sua execução, para

se detectarem vulnerabilidades de segurança no código fonte. Infelizmente, os compiladores muitas vezes não o fazem.

As ferramentas de análise estática utilizam técnicas semelhantes aos compiladores, e outras que os compiladores não possuem, como por exemplo, controlo do fluxo por grafos (Sotirov, 2005). O facto destas ferramentas serem independentes do compilador, permite-lhes efectuar testes sobre o código fonte sem que este necessite de ser compilado. Um exemplo de uma plataforma de desenvolvimento que contém um compilador e uma ferramenta independente de análise estática de vulnerabilidades é o Microsoft Visual Studio[1], que compila o código fonte e contém uma ferramenta para detecção de vulnerabilidades estaticamente, e que é independente do compilador.

2.4 Organização da Memória

Compreender a estrutura de memória é importante para uma melhor percepção de uma das vulnerabilidades mais comuns, memória sobrelotada (do inglês, *buffer overflow* ou *underflow*). Esta vulnerabilidade só ocorre quando o programa se encontra a trabalhar/aceder em zonas de memória ilegais, violando o paradigma de protecção de memória (One, 1996).

Durante a execução de um programa este vai utilizar duas zonas de memória, a *pilha* e o *segmento*. A *pilha* é usada para armazenar variáveis locais, parâmetros e os endereços de retorno da função. Enquanto que no *segmento* são armazenadas atribuições dinâmicas de memória (por exemplo, *malloc()*) e variáveis locais ao processo (Aho *et al.*, 1996).

O espaço de endereçamento de um processo[2] pode ser observado na Figura 2.3. O ponteiro BRK indica a quantidade de memória dinamicamente reservada e o *ponteiro da pilha* (ESP) aponta para o topo da pilha ou a próxima posição livre. O tamanho do código gerado é estabelecido em tempo de compilação, e dessa forma, o compilador pode colocá-lo numa área estaticamente determinada,

[1] Mais informação em, `http://www.microsoft.com/brasil/msdn/Tecnologias/vs2005/vs05security_US.mspx` (visitado em Setembro de 2007).

[2] Informação destinada a compiladores de linguagem de programação C e C++ para uma arquitectura Intel x86.

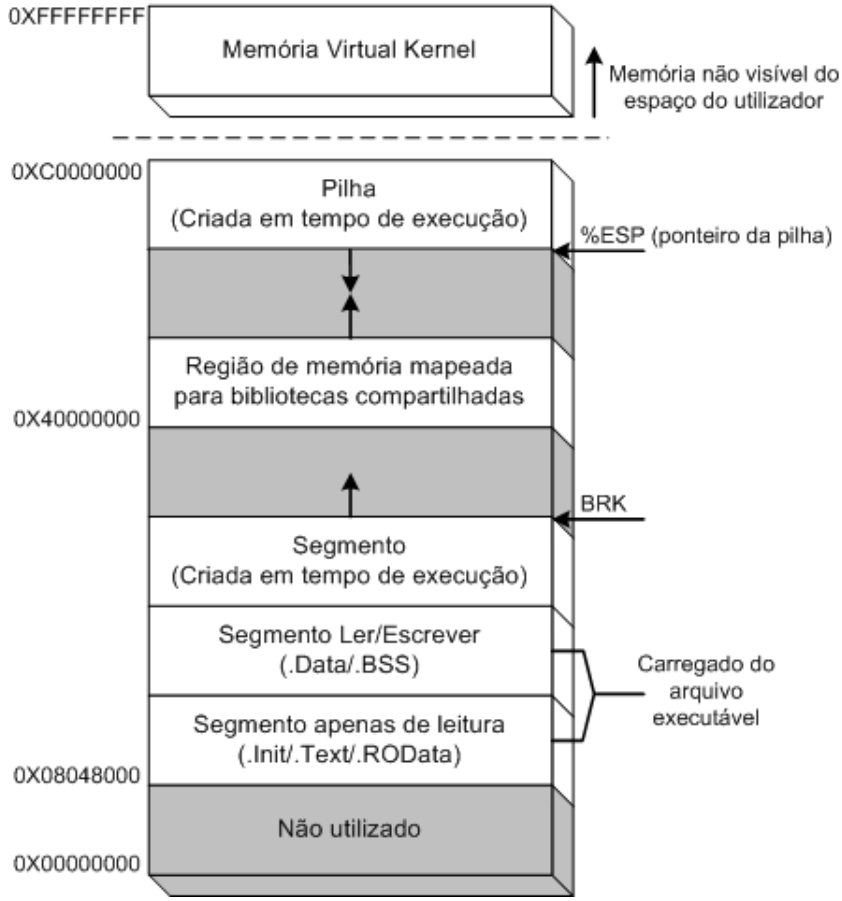


Figura 2.3: Espaço de endereçamento de um processo em Linux.

normalmente na parte mais baixa da memória. Analogamente, os tamanhos de alguns objectos de dados já podem ser conhecidos em tempo de compilação e estes dados também têm um espaço reservado antes da execução do programa (exemplo, variáveis globais e estáticas).

Como é possível observar na Figura 2.3, existe um espaço livre de memória por cima do *segmento* e em baixo da *pilha*. Esse espaço é utilizado quando o processo deseja mais espaço em memória, do que aquele que foi previamente estabelecido.

A pilha consiste em inúmeras molduras (do inglês, *frames*) de pilha, sendo cada moldura inserida na memória quando uma função é chamada, e é retirada quando a função retorna (através do apontador FP). A moldura é utilizada para referenciar variáveis locais e parâmetros na pilha, já que a distância entre va-

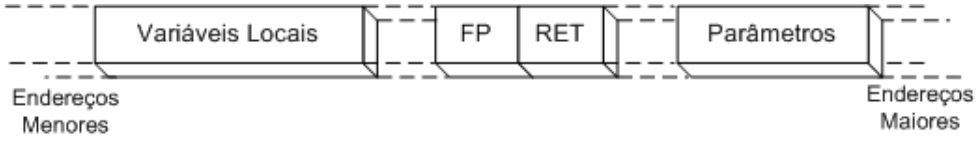


Figura 2.4: Estrutura de uma moldura de pilha.

riáveis e parâmetros não se altera durante a execução. Outro dado armazenado numa moldura é o endereço de retorno (RET), que é o endereço que deve ser executado depois do epílogo da função. Em muitas concretizações, a pilha cresce dos endereços maiores de memória para os menores.

A estrutura básica de uma moldura de pilha num sistema Linux é apresentada na Figura 2.4. Quando a moldura é inserida os parâmetros da função são os primeiros a serem colocados, sendo a inserção feita da direita para a esquerda, começando a partir do último parâmetro da função. Para além disso, são inseridos o *endereço de retorno* (RET) e o *apontador da moldura* (FP). Por fim, são inseridas as *variáveis locais*, pela ordem que estas variáveis são declaradas na função (Grégio *et al.*, 2005).

2.5 Métodos de Análise

As ferramentas de análise devem ser utilizadas em programas onde o grau de segurança desejado é elevado, independentemente do seu tamanho. O que interessa é se o programa está ou não susceptível a conter vulnerabilidades, quer tenha ligação a uma rede global ou não.

É aconselhado para as pequenas, médias e grandes empresas o uso de normas de concretização e ferramentas de análise. No entanto, para as pequenas empresas, é aconselhável a utilização de ferramentas de análise gratuitas. É de notar que uma ferramenta de análise não substitui o processo de validação e verificação (V&V), porque esta não interpreta se a aplicação faz o que é pretendido ou desejado. Um dos objectivos da análise[1] é encontrar deficiências no código e diminuir

[1] A análise de código é uma operação muito complexa, uma vez que existem diferentes modos de programar e além da informação que as ferramentas precisam é vasta.

os riscos de segurança, podendo fazer parte do ciclo de desenvolvimento de uma aplicação.

2.5.1 Análise Manual

A análise manual[1] é o método mais antigo para detecção de vulnerabilidades e/ou problemas na programação da aplicação. Para proceder a uma análise manual é preciso que haja suporte para a auditoria, ou seja, é preciso ter noção da arquitectura, o código deve estar comentado e conter um sumário da funcionalidade de cada método (Kruegel, 2004).

Dado ser uma tarefa difícil, cansativa e morosa, a análise manual não é o melhor método para rever e detectar anomalias no código fonte. Por exemplo, Crispin Cowan disse[2]:

"*Reviewing old code is tedious and boring and no one wants to do it.*".

Um indivíduo para realizar uma auditoria deste tipo precisa de ter as condições para o fazer, como já foram referenciadas, e para o fazer de forma eficiente, o auditor necessita de ter conhecimento sobre vulnerabilidades de segurança. Normalmente, os auditores já têm técnicas de detecção de vulnerabilidades bem definidas, por exemplo, procurar por funções que recebem informação de um utilizador local ou remoto, verificar se há funções denominadas perigosas e se os parâmetros estão a ser validados (Chess & McGraw, 2004).

No entanto, pode-se concluir que este método tem limitações fortes porque envolve questões sociológicas e psicológicas para realizar a auditoria, uma vez que o auditor depois de analisar algumas linhas de código tende ficar cansado, o que potencialmente causará falhas na análise.

2.5.2 Análise Estática

A análise estática consiste numa verificação automática do código fonte, normalmente em tempo de compilação. Existem diversos analisadores estáticos, que

[1] O sistema operativo OpenBSD é um bom exemplo de análise manual, Kruegel (2004).

[2] Citação de Crispin Cowan em, `http://news.com.com/2100-1001-864236.html` (visitado em Setembro de 2007).

processam o código fonte em cooperação com o compilador (Younan *et al.*, 2004). Há outros que estão preparados para funcionar independentemente do compilador, e ainda existem analisadores que requerem que o código fonte seja compilado antes de o tratar (Chess & McGraw, 2004).

O objectivo desta análise é encontrar vulnerabilidades, algumas mais comuns e outras mais complexas, através de um conjunto de técnicas como, o controlo de fluxo e procura de padrões (ver em Sotirov (2005) e Antunes (2006)).

Embora a análise estática utilize técnicas sofisticadas para captura de informação sobre a execução do programa, muitas vezes recorre a *anotações*. As anotações permitem ao programador especificar por exemplo que uma variável tem um máximo e um mínimo, ou que o seu valor provém de fora da aplicação. Estas anotações vão possibilitar à ferramenta obter invariantes sobre o código fonte, e eliminar outras invariantes. Logo, pode-se ter uma maior precisão na localização das vulnerabilidades e qualidade nos resultados. A desvantagem desta abordagem é a necessidade de o programador ter de conhecer a sintaxe das anotações e escreve-las.

A análise estática também tem as suas limitações porque as ferramentas estão restringidas a um conhecimento à priori. Basicamente, elas só reportam vulnerabilidades para as quais foram programadas para detectar. Como consequência, uma pessoa mal intencionada pode recolher as limitações de várias ferramentas, e assim, explorar vulnerabilidades que estas não conseguem abranger.

Adicionalmente as vulnerabilidades podem estar camufladas, ou seja, é possível haver problemas que envolvam *variable aliasing* ou que o erro resida na função chamada e não na que chama, o que impede a sua localização por causa da complexidade adjacente (Horwitz, 1997).

Algumas ferramentas conhecidas a nível académico são:

- **BOON**, detecção de vulnerabilidades de memória sobrelotada por avaliação de intervalos de valores (memória obtida e memória em uso) (Wagner *et al.*, 2000);

- **CQual**, ferramenta de análise estática de código fonte por tipos qualificativos (Foster *et al.*, 1999);

- **Crystal**, plataforma de suporte capaz de detectar vulnerabilidades de gestão de memória (Orlovich & Rugina, 2006);
- **Deputy**, baseia-se em dependências entre tipos de dados e acesso à memória (Condit *et al.*, 2006);
- **Flawfinder**, detecta vulnerabilidades de diferentes tipos, encontrando-as através de análise de padrões (Wheeler, 2007);
- **ITS4**, procura funções denominadas perigosas e reporta a perigosidade do seu uso (Viega *et al.*, 2000);
- **Locksmith**, ferramenta para detecção de problemas em processos multitarefa (*Pthreads*) (Pratikakis *et al.*, 2006);
- **Meta-compilation**, uma extensão ao compilador para detectar vulnerabilidades de espaço de endereçamento de um inteiro excedido (Ashcraft & Engler, 2002);
- **MOPS**, analisa código fonte em tempo de compilação por meio de um conjunto de propriedades pré-definidas (Chen *et al.*, 2004);
- **PScan**, detecta funções que causam formatação de cadeias de caracteres e memória sobrelotada (DeKok, 2007);
- **RATS**, ferramenta semelhante ao Flawfinder e ITS4, com a diferença de que contém mais conhecimento e avalia diferentes linguagens de programação (Secure Software, 2007);
- **Splint** ou **LCLint** ou **Lint**, ferramenta utilizada em tempo de compilação baseada em anotações (Larochelle & Evans, 2001);
- **UNO**, detecta vulnerabilidades, como não inicialização de variáveis, acesso indevido à memória e referência nula de apontadores (Holzmann, 2007);
- **Vulncheck**, é muito semelhante ao Meta-compilation, mas pode também detectar memória sobrelotada, formatação de cadeias de caracteres e acessos fora do limite de uma lista (Sotirov, 2005).

A utilização do método de análise estática tem diversas vantagens, como a simplicidade e rapidez no processamento do código e produção de resultados. Este método pode no entanto gerar alguns falsos positivos, ou seja, as ferramentas podem reportar vulnerabilidades que não existem no código fonte. Estes erros na detecção acabam por ser inconvenientes porque obrigam os programadores a desperdiçarem tempo a tentarem solucionar problemas que na realidade não existem. É por isso que neste trabalho se pretende desenvolver uma solução que diminua o número de falsos positivos e mantenha os verdadeiros positivos.

2.5.3 Análise Dinâmica

Neste método analisa-se o comportamento da aplicação em tempo de execução. As ferramentas de análise dinâmica também detectam diversas vulnerabilidades e utilizam algumas técnicas de análise estática.

Para algumas ferramentas de análise dinâmica é necessário conhecer as funcionalidades da aplicação, por exemplo, para se realizarem testes específicos para detectar problemas de acesso fora dos limites de uma lista (do ingês, *array out of bound*). Há também ferramentas que não necessitam destes testes e que simplesmente controlam o acesso à memória, evitando vulnerabilidades de memória sobrelotada.

O grande problema da análise dinâmica reside na dificuldade da especificação dos casos de teste (Haugh & Bishop, 2003). O problema baseia-se em que se os casos de teste não forem suficientes e/ou abrangentes, então ficam partes do código por visitar, permanecendo assim vulnerabilidades por encontrar.

Claramente, a análise dinâmica não é muito útil durante a concretização da aplicação, mas pode ser utilizada durante a fase de testes. Enquanto se testa, verifica-se o comportamento do sistema ou aplicação.

Há diversos tipos de ferramentas com diferentes técnicas para proteger a memória da aplicação em tempo de execução (Younan *et al.*, 2004). As ferramentas apresentadas são algumas das mais relevantes:

- **FormatGuard**, detecta dinamicamente vulnerabilidades de formatação de cadeias de caracteres (Cowan *et al.*, 2001);

- **PointGuard**, protege a memória durante a execução da aplicação, cifrando o conteúdo dos apontadores e decifrando-os quando utilizados (Cowan *et al.*, 2003);

- **Purify**, ferramenta utilizada na detecção de vulnerabilidades de gestão de memória e acessos ilegais à memória (Hastings & Joyce, 1992);

- **RAD**, guarda os endereços de retorno num repositório de endereços de retorno, detectando problemas de memória sobrelotada na pilha (Chiueh & Hsu, 2001);

- **Safe C**, verifica dinamicamente problemas de memória sobrelotada e acessos ilegais à memória (Austin *et al.*, 1994);

- **StackGuard**, protege o endereço de retorno na pilha, colocando um número aleatório (*canary*) depois das variáveis da moldura de pilha (Cowan *et al.*, 1999);

- **StackShield**, é também utilizado com o objectivo de proteger o endereço de retorno, utilizando dois métodos: *Global Ret Stack* e *Ret Range Check* (Younan, 2003);

- **STOBO**, ferramenta que detecta dinamicamente memória sobrelotada (Haugh & Bishop, 2003);

- **Valgrind**, detecta problemas associados à gestão de memória (Oliveira, 2006).

Desta lista pode-se observar que existe uma preponderância de ferramentas para detectar anomalias na memória, do que outro tipo de vulnerabilidades como acesso a ficheiros, condições de disputa, etc.

Portanto, as ferramentas de análise dinâmica analisam o comportamento da aplicação durante a sua execução, e para testar a aplicação demora-se potencialmente muito tempo, pois depende do número de testes e também da dimensão da aplicação.

2.5.4 Comparação entre os Métodos

Uma auditoria manual é muito semelhante a uma análise estática, com a diferença que este requer pessoas especializadas (os auditores) e a outra não. A análise estática é mais rápida, ou seja, permite realizar auditorias com maior frequência e não se precisa de conhecer a totalidade da arquitectura da aplicação.

A auditoria manual é difícil de ser comparada com a análise dinâmica, uma vez que uma analisa o código fonte estaticamente e a outra baseada em testes sobre a aplicação em tempo de execução.

Em geral, a análise manual é um método que requer muito esforço e persistência, e consequentemente, é ineficiente na análise, apesar de que pode ter um custo razoável, dependendo do valor do ordenado do auditor[1].

As ferramentas de análise estática detectam aproximadamente o mesmo tipo de vulnerabilidades que a análise dinâmica. A análise estática pode ser introduzida durante a fase de concretização, o que diminuí a propagação de erros para outras fases, contribuindo para diminuir os custos.

As ferramentas de análise dinâmica não beneficiam uma empresa durante a fase de concretização, pois perde-se muito tempo a testar a aplicação, mas, são um recurso relevante podendo ser utilizado durante a fase de testes. O seu problema é a geração ou criação de testes, ou seja, se estes não percorrem todos os caminhos podem ficar vulnerabilidades por encontrar. Há porém ferramentas de análise dinâmica que protegem a memória e reportam se algo ocorreu mal, o que já é significativo.

Em conclusão, a análise estática é mais rápida e tem um maior impacto no ciclo de desenvolvimento de um produto final. Como tal, é de aconselhar que se utilize um analisador estático. Tem um problema que é a emissão de falsos positivos, existindo ferramentas que reportam mais que outras. Além disso, é considerado um método de prevenção, isto é, previne vulnerabilidades antes que estas sejam exploradas pelos adversários.

[1] A utilização de uma ferramenta como o *Coverity* pode custar 50 mil dólares por 500 mil linhas de código, (Binstock & Streng, 2006).

2.6 Vulnerabilidades

Existe um conjunto substancial de tipos distintos de vulnerabilidades possíveis de descrever e categorizar. As vulnerabilidades tipicamente encontram-se divididas em duas categorias: as de concretização e as de desenho (Lavenhar, 2006); porém, aquelas que se irá focar são as de concretização. As vulnerabilidades a descrever posteriormente são as mais conhecidas[1]: *Memória Sobrelotada*, *Formatação de Cadeias de Caracteres*, *Controlo de Acesso*, *Condições de Disputa*, *Exceder o Espaço de Representação do Inteiro* e *Números Aleatórios*.

Um erro num programa só corresponde a uma vulnerabilidade, se esta poder ser explorada por um agente malicioso, então temos um problema. Assim, uma lição a ter sobre vulnerabilidades, e que é aplicada a qualquer linguagem de programação, consiste em: *"Não confiar nos dados introduzidos por um qualquer utilizador e nunca presumir nada sobre os parâmetros a receber."* (Correia, 2006).

Alongo deste trabalho, constatou-se que existe um vasto conjunto de funções das bibliotecas do C, que potencialmente podem introduzir vulnerabilidades quando usadas num programa. Algumas destas funções encontram-se descritas no Apêndice C.

2.6.1 Estatísticas

As vulnerabilidades são um quebra-cabeças para as instituições que desenvolvem aplicações, que posteriormente vendem aos potenciais consumidores. Como tal, convém que o produto final não tenha vulnerabilidades, para que o seu nome não apareça associado ao conjunto de empresas que desenvolvem produtos com falhas de segurança.

A Figura 2.5 mostra a evolução quanto ao número de vulnerabilidades reportadas ao longo dos últimos cinco anos[2]. Assim, em 2001 obteve-se 1672 vulnerabilidades reportadas, enquanto que 2006 se obteve 6600. A estes valores falta o número de vulnerabilidades que foram encontradas mas que não foram reportadas,

[1] Para mais informação consulte, Koziol *et al.* (2004), Viega & McGraw (2006) e Basirico (2004).

[2] Dados obtidos ao CVE e CCE (US-CERT) em, `http://nvd.nist.gov/statistics.cfm` (visitado em Setembro de 2007).

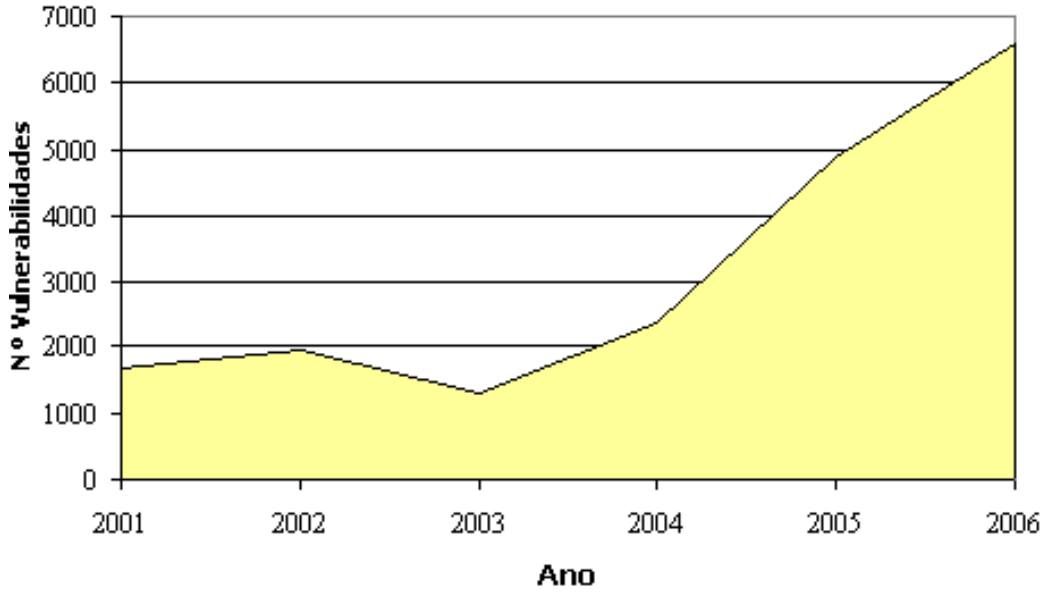


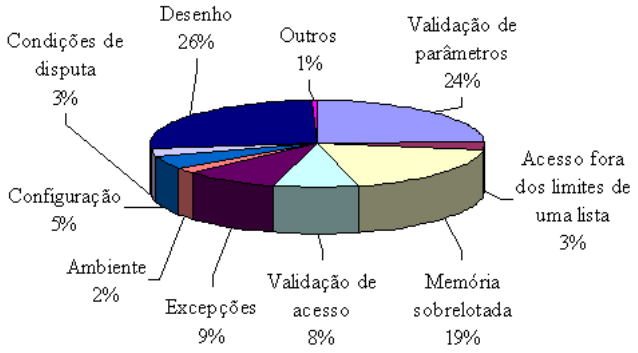
Figura 2.5: Evolução das vulnerabilidades reportadas entre 2001 a 2006.

valor este que é muito difícil de obter, mas possível de imaginar. Porém, observando bem, consegue-se obter a seguinte informação: a partir de 2004 o número de vulnerabilidades reportadas começaram a duplicar. Isto pode significar que as pessoas começaram a ganhar mais coragem para reportar tais vulnerabilidades e/ou os meios de comunicação passaram a ficar mais fáceis de aceder.

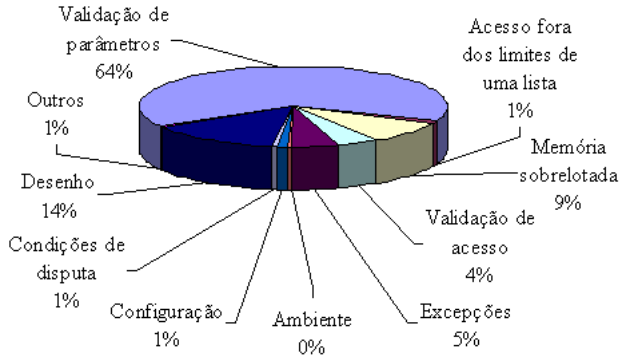
Também é curioso observar a distribuição do conjunto de vulnerabilidades[1] pelas várias classes para o ano de 2001 e 2006 (ver Figura 2.6). Note-se que a validação de parâmetros tem ganho vantagem ao longo dos anos. Este tipo de vulnerabilidade, quase triplicou, deixando antever a existência de graves problemas nos módulos do código que são responsáveis por processar os dados fornecidos pelos utilizadores.

Pode-se concluir que as vulnerabilidades estão a aumentar e que alguns tipos de vulnerabilidades perderam importância. O que se pode explicar é que com uma maior utilização de linguagens de programação como o Java, certas

[1] Os valores para construir os gráficos foram alcançados na página *web*, `http://nvd.nist.gov/statistics.cfm` (visitado em Setembro de 2007). No entanto, em `http://cwe.mitre.org/documents/vuln-trends.html` (visitado em Setembro de 2007), se encontra um relatório mais exaustivo de cada vulnerabilidade.



(a) Ano 2001



(b) Ano 2006

Figura 2.6: Evolução dos tipos de vulnerabilidades entre 2001 e 2006.

vulnerabilidades têm tendência a continuar a diminuir (exemplo, memória sobrelotada). O problema mais preocupante é a validação de parâmetros, necessitando de cuidados específicos durante o processo de codificação.

2.6.2 Memória Sobrelotada

Esta secção descreve a vulnerabilidade mais conhecida de todas — a memória sobrelotada (Alexander, 2005). Esta vulnerabilidade ocorre especificamente na memória, mais concretamente na pilha e/ou no segmento (ver Secção 2.4). A vulnerabilidade ocorre quando é permitido escrever numa estrutura de dados para além do seu limite de memória.

Genericamente, a pilha e o segmento, como referido na Secção 2.4, têm um espaço específico na memória que é partilhado, ou seja, quando se escreve para

além do que é devido, pode-se alterar valores de outras estruturas de dados do programa.

As principais razões para o aparecimento da vulnerabilidade são: a falta de cuidado com os índices das listas ou ciclos; o carácter "\0" não está presente na cadeia de caracteres a copiar; ou mesmo em erros com operações aritméticas de ponteiros.

Memória sobrelotada na pilha faz com que se possa escrever por cima do valor de retorno de uma função, ou outros dados na memória, levando um programa a comportar-se de forma anómalo. Também se pode sobrelotar a memória no segmento onde se escreve por cima de tipos diferentes de dados, como ponteiros para funções.

```
1 char str_A[50];
2 char str_B[50];
3 scanf(``%s'',    str_A);
4 scanf(``%49s'', str_B);
```

Listagem 2.1: Código vulnerável.

Na Listagem 2.1, é possível observar uma codificação muito frequente que não faz validação dos parâmetros de entrada, e que tem a consequência de criar uma vulnerabilidade de memória sobrelotada.

Na linha 3 é possível haver memória sobrelotada porque se se introduzir algo com mais de 50 caracteres existe uma escrita para além do espaço guardado na memória para str_A. Porém, a linha 4 já não é vulnerável porque usa uma propriedade nos formatadores de cadeias de caracteres, que permite indicar o número máximo de *bytes* a ler (neste caso 49 caracteres). Assim, pode-se usar o *scanf()* de forma segura, mesmo que um utilizador insira mais do que os 49 *bytes*, estes não são gravados para memória.

Como se viu através do exemplo, o problema de memória sobrelotada pode ser em grande parte evitado se utilizar funções mais seguras (ver Apêndice C), como o *strncpy()*, mas acima de tudo deve-se validar os dados que são recebidos e/ou restringir o tamanho a ler (Viega & McGraw, 2006).

2.6.3 Formatação de Cadeias de Caracteres

A vulnerabilidade de formatação de cadeias de caracteres (do inglês, *format string*) começou a ser explorada recentemente, por volta do ano 2000 (Sotirov, 2005). Esta vulnerabilidade surge quando uma função não utiliza formatadores de cadeias de caracteres "%s".

A vulnerabilidade está de certa forma relacionada com a validação de parâmetros, no entanto, é simples de perceber e corrigir uma vez que ocorre em funções que trabalham com formatação de cadeias de caracteres (por exemplo, o *printf()* e sua família ou mesmo o *syslog()*) (Shankar *et al.*, 2001).

Estas funções têm tipicamente um descritor de formatação do tipo "%d" ou "%s", que indica como os argumentos devem ser tratados, quer para ler ou escrever. Se uma aplicação está susceptível a uma vulnerabilidade de formatação de cadeias de caracteres, então pode sofrer ataques para obter informação de zonas de memória. Através desta informação podem-se localizar os endereços de retorno e nestes inserir código malicioso para ser executado (Bhatkar *et al.*, 2003).

A solução passa por limitar o tamanho do conteúdo a imprimir e/ou utilizar o formatador "%s" (consultar Apêndice C).

2.6.4 Controlo de Acesso

A vulnerabilidade de controlo de acesso (do inglês, *access control*) tem como causa um abuso no acesso a recursos do sistema. Um utilizador uma vez autenticado para entrar no sistema, tem associado um certo número de permissões. A vulnerabilidade surge quando o atacante deixa de estar limitado nas acções que pode realizar, ou seja, passa a ter mais permissões do que deveria ter (Viega & McGraw, 2006).

Há diversos tipos de ataques que se podem explorar, sendo o mais conhecido a obtenção de acesso a contas de outros utilizadores; tendo-se conseguido esse acesso, pode-se ler por exemplo informação confidencial, e usá-la para actos ilícitos.

Na programação das aplicações é necessário limitar o acesso aos recursos, e evitar que um atacante possa obter os mesmos privilégios que estão associados ao utilizador da aplicação. Assim, sempre que a aplicação execute uma chamada

ao sistema deve-se diminuir os privilégios (de leitura, escrita e execução sobre qualquer recurso) e depois da operação actualizam-se os privilégios.

2.6.5 Condições de Disputa

A condição de disputa (do inglês, *race condition*) só é possível ocorrer num ambiente multi-tarefa, ou seja, entre processos concorrentes (como acesso de leitura, escrita, etc). Esta vulnerabilidade é limitada, e como só acontece em determinados ambientes, torna-se complexa e subtil. Normalmente é uma vulnerabilidade muito problemática porque é difícil de ser detectada e de resolver.

Uma variante muito conhecida desta vulnerabilidade é o TOCTOU (*Time of check, Time of use*). Este aparece quando são executadas duas operações, uma para verificar se um determinado recurso pode ser utilizado, e outra então para usar o recurso caso o acesso seja válido (Viega & McGraw, 2006). Entre o tempo de validar e utilizar há um pequeno intervalo de tempo possível de alterar o conteúdo do recurso a usar (utilização *pseudo-simultânea*), permitindo assim que se execute uma operação diferente do que se esperava. Por outras palavras, as condições de disputa acontecem quando dois ou mais processos utilizam a mesma variável, arquivo ou recursos simultaneamente (Heffley & Meunier, 2004).

Um exemplo muito prático desta vulnerabilidade ocorre durante a utilização de ficheiros temporários. Um atacante consegue detectar qual o ficheiro temporário gerado, e poderá tentar modificar as informações contidas nesse arquivo, alterando a execução habitual do programa.

As soluções para este tipo de vulnerabilidades dependem de caso a caso, embora a solução mais usual seja garantir que sempre que se usa recursos partilhados, deve-se criar todos os mecanismos (por exemplo, trancar um recurso) necessários para evitar os acessos concorrentes aos mesmos, durante o período critico de validação/utilização.

2.6.6 Exceder o Espaço de Representação do Inteiro

Quando se pretende guardar um valor num tipo numérico e o valor a guardar não cabe nesse tipo (não fica com o valor correcto de origem), significa que o espaço

de representação do tipo numérico foi excedido (do inglês, *integer overflow* ou *underflow*) (Younan, 2003).

Na linguagem de programação C existem limites, ou seja, há um valor máximo e mínimo, para cada tipo numérico (que pode ser consultado no Apêndice D). Ao ultrapassar o valor máximo, o seu valor volta a zero mais o valor que foi excedido; no caso do valor mínimo é ao contrário, excede-se o valor mínimo e volta ao valor máximo menos o valor excedido (isto, no caso de tipos *unsigned*) (Grégio *et al.*, 2005) (Koziol *et al.*, 2004).

A vulnerabilidade pode acontecer em dois tipos de operações: conversão entre tipos numéricos ou operações aritméticas. Destas duas, a conversão de tipos numéricos é a que mais ocorre, e consiste em converter um tipo numérico com sinal para um sem sinal ou entre diferentes tipos numéricos; as operações aritméticas são baseadas em somas, subtracções ou multiplicações, entre dois ou mais operandos.

Esta vulnerabilidade é mais difícil de ser explorada uma vez que consiste num ataque muito específico e requer atenção do próprio atacante, onde este depois de realizar consecutivos ataques, observa um comportamento irregular da aplicação. Este tipo de vulnerabilidade já foi reportada, no *OpenSSH*, *Internet Explorer* e *Linux Kernel* (Sotirov, 2005).

As causas que levam à ocorrência deste tipo de vulnerabilidades são a falta de validação, quer internamente no programa ou por dados externos, ou erros humanos durante a concretização. A solução passa por validar o valor do tipo numérico, em relação aos seus limites pré-definidos pelo sistema (ver Apêndice D).

2.6.7 Números Aleatórios

Os computadores são deterministas, deste modo, o que torna complicada a tarefa de gerar valores aleatórios perfeitos. Assim, sempre que são necessários este tipo de números recorre-se normalmente a Geradores de Números Pseudo-Aleatórios (PRNGs) (Viega & McGraw, 2006). A utilização de números aleatórios é muito usada na criptografia, por exemplo para se obterem chaves de sessão.

Existe uma vulnerabilidade na utilização de números aleatórios quando o número gerado é fraco, isto é, quando o adversário consegue descobrir o número, por

exemplo, a partir de observação da sequência de números gerados. A ferramenta de análise estática deveria ser capaz de avaliar a forma como os números aleatórios estão a ser obtidos, e garantir que eles têm um nível de segurança adequado.

Como forma de solucionar esta vulnerabilidade, deve-se utilizar algoritmos capazes de gerar números de forma aleatória, ou utilizar *seeds* variáveis em algoritmos deterministas.

Capítulo 3

Trabalho Realizado

Ao longo deste capítulo descreve-se as ferramentas de análise estática estudadas para localizar diversas vulnerabilidades em código fonte. Apresenta-se também uma primeira versão de um teste padronizado para comparar e avaliar as ferramentas em relação às suas capacidades de detecção de vulnerabilidades. Por fim, detalha-se a arquitectura e funcionamento da ferramenta Mute.

3.1 Objectivos

O trabalho consistiu em criar uma ferramenta capaz de agregar os resultados de outras ferramentas, que usam diferentes técnicas de análise, com o intuito de detectar mais vulnerabilidades. No entanto, durante a avaliação das ferramentas, verificou-se que estas além de encontrarem vulnerabilidades reais, também indicavam um número de vulnerabilidades que na realidade não existem — os falsos positivos. Assim, o desejável é que a nova ferramenta consiga reportar mais vulnerabilidades que cada ferramenta individualmente, reduzindo ao mesmo tempo o número de falsos positivos.

Porém, durante a fase de análise das ferramentas existentes, percebeu-se que era muito complicado fazer-se uma comparação entre elas (por exemplo, cada autor descrevia a sua ferramenta como a melhor do grupo). Tornou-se assim necessário desenvolver um teste que fornecesse alguma indicação da qualidade de cada ferramenta. Este teste baseia-se num programa que contém diversas vulnerabilidades bem identificadas, que é entregue a cada ferramenta para análise.

O resultado dessa análise permite obter valores comparáveis sobre as capacidades de detecção de cada ferramenta.

A forma como o programa se encontra organizado é com excertos de código ou funções, cada uma delas contendo tipos distintos de vulnerabilidades. Ao longo do trabalho, o programa foi sendo refinado de maneira a ter representantes das vulnerabilidades mais comuns (muitas vezes usando os exemplos de testes que eram distribuídos com as ferramentas).

Portanto, o trabalho realizado evoluiu para três fases distintas: o desenvolvimento de uma primeira versão de um teste padronizado que possibilitasse a comparação de ferramentas de análise estática; a utilização do teste para avaliar uma selecção de nove ferramentas; a criação de uma ferramenta capaz de detectar mais vulnerabilidades que as anteriores, com um número de falsos positivos reduzido.

3.2 Estudo das Ferramentas

Existem várias ferramentas de análise estática de código fonte, quer comerciais como associadas a projectos de investigação (Younan *et al.*, 2004). Para este trabalho só foram utilizadas e testadas ferramentas disponíveis na *Internet*, pois estas são gratuitas. Esta opção mostrou-se acertada, para se poder trabalhar e colocar em prática o pretendido.

Ferramentas distintas conseguem encontrar diferentes vulnerabilidades, isto devido à técnica de análise utilizada e também aos tipos de vulnerabilidades para os quais as ferramentas foram desenvolvidas. No entanto, existem ferramentas com técnicas diferentes, mas dirigidas para o mesmo tipo de vulnerabilidades, que encontram vulnerabilidades reais diversas, para além das em comum.

Na maioria dos casos estudados, as ferramentas demoram a analisar o código fonte pouco mais de um segundo, até menos — testado para uma aplicação aproximadamente com 5 mil linhas de código. A ferramenta MOPS pode levar substancialmente mais tempo, consoante a propriedade a analisar.

Nesta secção apresentam-se brevemente as ferramentas que foram analisadas e mais tarde utilizadas para construir a ferramenta final, **Mute**. É possível encon-

trar no Apêndice E, um resumo das ferramentas e mais algumas características de cada ferramenta.

3.2.1 Crystal

A ferramenta Crystal[1] é a mais recente de todas as que foram testadas e só foi disponibilizada no fim do ano de 2006. Foi desenvolvida na Universidade de Cornell (Orlovich & Rugina, 2006).

O Crystal é uma plataforma de suporte com diversas extensões, no entanto, até ao momento só está concretizada a detecção de vulnerabilidades relacionadas com a gestão de memória (encontram-se em desenvolvimento duas novas extensões). Esta ferramenta trata da vulnerabilidade de gestão de memória, por exemplo, reservar espaço na memória n vezes sem libertar o que foi obtido anteriormente (consulte o Apêndice E e F).

A ferramenta foi programada na linguagem Java para detectar vulnerabilidades em código fonte C e utiliza diferentes técnicas de análise de código, como árvores abstractas de sintaxe, controlo do fluxo por grafos e a análise de fluxo de dados (Aho *et al.*, 1996). Além destas técnicas, consegue ainda obter invariantes do código fonte a partir da análise dentro e entre procedimentos (potencialmente localizados em ficheiros diversos).

Esta necessita que o código fonte seja preprocessado, ou seja, o código fonte tem que passar pelo preprocessador do compilador C, de maneira a gerar código com directivas de preprocessamento. Após esta fase, os ficheiros gerados são dados ao Crystal, que constrói grafos com o fluxo do programa e a partir daqui retira conclusões.

É uma ferramenta muito eficiente porque durante os testes que se realizaram não houve nenhum falso positivo reportado.

[1] A ferramenta encontra-se em, `http://www.cs.cornell.edu/projects/crystal/` (visitado em Setembro de 2007).

3.2.2 Deputy

O Deputy[1] é baseado em anotações, embora, consiga detectar alguns problemas, como memória sobrelotada, sem necessitar das anotações (Condit *et al.*, 2007). Foi desenvolvida por Condit, Harren, Anderson, Gay & Necula na Universidade de Berkeley. A ferramenta usa vinte e uma anotações[2] a adicionar no código fonte C. Utiliza uma linguagem intermédia de alto nível, o CIL[3], para analisar e transformar código fonte.

O Deputy tem um número razoável de anotações para variáveis, apontadores e estruturas de dados (*struct* ou *union*). Para os autores, o uso de anotações é uma mais valia da ferramenta porque são poucas e fáceis de usar, o que facilita o trabalho dos programadores.

A vulnerabilidade que a ferramenta detecta mais é a memória sobrelotada, embora também detecte problemas de formatação de cadeias de caracteres e erros comuns de programação.

3.2.3 Flawfinder

O Flawfinder[4] é uma ferramenta de análise de padrões, desenvolvida por David Wheeler na linguagem de programação Python, sendo muito semelhante às ferramentas ITS4 e RATS (ver Secção 3.2.4 e 3.2.7). A ferramenta suporta código nas linguagens C e C++.

A ferramenta à medida que pesquisa o código fonte, guarda informação sobre as possíveis vulnerabilidades para depois reportar, com uma descrição do problema e solução, assim como a severidade e o tipo de vulnerabilidade. As vulnerabilidades reportadas encontram-se sempre ordenadas pelo grau de severidade. O grau de severidade depende não só das funções, mas também dos valores dos parâmetros da função, por exemplo, uma cadeia de caracteres constante tem menor risco que uma cadeia de caracteres variável.

[1] A ferramenta Deputy tem uma página *web* em, `http://deputy.cs.berkeley.edu/` (visitado em Setembro de 2007).

[2] As anotações são tipos qualificativos, muito semelhantes ao *const* em C.

[3] Mais informação sobre o CIL consulte, `http://hal.cs.berkeley.edu/cil/` (visitado em Setembro de 2007).

[4] Para obter a ferramenta e informações em, `http://www.dwheeler.com/flawfinder/` (visitado em Setembro de 2007).

Uma vez que esta ferramenta analisa o código fonte em busca de padrões, ela contém uma base de dados com funções potencialmente vulneráveis. Estas funções correspondem a vulnerabilidades do tipo memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso, condições de disputa e números aleatórios.

Das diversas ferramentas, o Flawfinder é aquela que detalha melhor o problema de cada vulnerabilidade, possibilitando a diminuição do número de falsos positivos apresentados através da inserção de comentários no código (ver Apêndice F).

3.2.4 ITS4

O ITS4[1] foi uma das primeiras ferramentas para detectar vulnerabilidades através da análise de padrões (Viega *et al.*, 2000). A ferramenta foi criada para substituir o *grep*[2], tornando a busca mais fácil e rápida de vulnerabilidades. O ITS4 foi desenvolvido por um conjunto de investigadores da Reliable Software Technologies (Cigital) e concretizado em C++.

A ferramenta analisa programas em C e C++, e utiliza uma base de dados com uma lista de funções vulneráveis. À medida que encontra uma função vulnerável guarda a ocorrência para no fim reportar. A cada função está associada uma descrição do problema e solução, assim como o grau de severidade.

A única técnica de análise é a busca de padrões e não utiliza outros conceitos como uma árvore abstracta de sintaxe, isto porque, para os autores, era importante tornar a ferramenta rápida, simples e útil no desenvolvimento de uma aplicação (Viega *et al.*, 2002).

O ITS4 pesquisa por diversas vulnerabilidades, tais como, memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso, condições de disputa e números aleatórios.

Existem diversas diferenças entre o ITS4 e outras ferramentas de análise de padrão, por exemplo, ele usa heurísticas para eliminar determinados falsos posi-

[1] Sobre a ferramenta ITS4 em, `http://www.cigital.com/services/its4` (visitado em Setembro de 2007).

[2] Comando Unix, que dada uma expressão, procura-a num ou mais ficheiros, `http://www.gnu.org/software/grep/` (visitado em Setembro de 2007).

tivos no problema TOCTOU. Neste caso, o ITS4 sabe quais as funções que fazem a validação e as que são usadas a seguir a uma validação, e só quando as duas estão presentes é que reporta a vulnerabilidade.

O ITS4 e a maioria dos analisadores por busca de padrões, têm algumas limitações como falta de capacidade para verificarem que uma função potencialmente vulnerável está a ser utilizada seguramente, ou seja, se anteriormente os seus parâmetros já foram validados. Assim como a existência de falsos negativos devido a uma vulnerabilidade existir no código fonte e não na base de dados.

3.2.5 MOPS

A ferramenta MOPS[1] foi desenvolvida por Hao Chen e David Wagner, e até ao momento ainda continua numa fase inicial de desenvolvimento (recentemente saiu a nova versão 0.9.2) (Chen *et al.*, 2004).

O MOPS foi construído para a linguagem de programação C, e utiliza duas técnicas de análise sofisticadas, a análise por fluxo de dados e o controlo de fluxo por grafos. Detecta as vulnerabilidades a partir de um conjunto de propriedades, onde cada propriedade corresponde a um tipo particular de vulnerabilidade (exemplo, a validação do *strcpy()* é diferente do *strncpy()*) e são estas que verificam se há alguma sequência de operações que mostram um comportamento inseguro. Até ao momento, estão disponíveis vinte e duas propriedades.

Cada propriedade é avaliada de forma individual, ou seja, o código é examinado só para uma propriedade, depois se for necessário examina-se para outra propriedade, executando-o outra vez.

As vulnerabilidades que o MOPS tenta avaliar são de controlo de acesso, condições de disputa e formatação de cadeias de caracteres. Procura também por vulnerabilidades muito específicas de segurança, por exemplo, assume que se a aplicação estiver com privilégios *root*, então deve diminuir os privilégios do utilizador em determinados casos (Schwarz *et al.*, 2005).

Uma vantagem deste mecanismo de análise é a sua modularidade, isto é, permite a adição de novas propriedades porque estas são independentes do modo

[1] O *site* oficial da ferramenta em, `http://www.cs.ucdavis.edu/~hchen/mops/` (visitado em Setembro de 2007).

como a ferramenta está concretizada.

3.2.6 PScan

PScan[1] é uma ferramenta de análise estática de código fonte para detecção de vulnerabilidades de formatação de cadeias de caracteres e memória sobrelotada (ou seja, família *printf()* e *scanf()*). É constituída por uma base de dados com lista de funções possivelmente vulneráveis para a linguagem de programação C.

O PScan sempre que encontra problemas no código fonte reporta-os como erros ou avisos, seguido de uma descrição do problema. Os erros tipicamente correspondem a problemas mais graves que os avisos. A técnica de análise utilizada pelo PScan é a análise lexical, que procura por funções vulneráveis e examina os seus argumentos.

A desvantagem desta ferramenta é o número limitado de vulnerabilidade que consegue detectar, o que nem sempre é útil quando se pretende analisar código fonte com diferentes tipos de vulnerabilidades. No entanto, é uma ferramenta rápida e com um bom grau de precisão.

3.2.7 RATS

RATS [2] é uma ferramenta com semelhanças ao Flawfinder e ITS4 (ver Secção 3.2.3 e 3.2.4), sendo considerada um sucessor do ITS4 (Secure Software, 2007).

A ferramenta foi desenvolvida na linguagem de programação C e utiliza a análise por padrões. Detecta vulnerabilidades de memória sobrelotada, formatação de cadeias de caracteres, controlo de acesso, condições de disputa e números aleatórios. A análise do código fonte pode ser feita sobre as seguintes linguagens de programação: C, C++, Perl, PHP e Python.

Esta ferramenta também requer uma base de dados com funções vulneráveis, no entanto, comparativamente a outras, possui uma lista maior de funções vulneráveis. Depois de analisar o código fonte reporta as vulnerabilidades encontradas,

[1] A ferramenta encontrada-se em, `http://www.striker.ottawa.on.ca/~aland/pscan/` (visitado em Novembro de 2006).

[2] A ferramenta pode ser encontrada em, `http://www.fortifysoftware.com/security-resources/rats.jsp` (visitado em Setembro de 2007).

classificando-as em três graus de severidade. No relatório de vulnerabilidades, estas são ordenadas pelo tipo de vulnerabilidade e grau de severidade, sendo também indicado o ficheiro onde se encontra a vulnerabilidade e linha correspondente, assim como, uma breve descrição do problema e solução.

Uma limitação que a ferramenta tem é reportar como vulnerabilidade as variáveis que possuem o mesmo nome de uma função vulnerável, com a consequência de aumentar o número de falsos positivos.

3.2.8 Sparse

O Sparse[1] é um analisador semântico para a linguagem de programação C na norma ANSI[2]. É muito semelhante aos analisadores semânticos utilizados nos compiladores. O Sparse reporta problemas existentes no código fonte que vão contra a norma ANSI, detectando também problemas entre tipos de dados e acesso fora do limite de uma lista.

O Sparse foi desenvolvido por Linus Torvalds a partir de 2003, e tinha como objectivo inicial detectar falhas com apontadores. Neste momento, a ferramenta continua em desenvolvimento, agora por Josh Triplett, e encontra-se na versão 0.3, que foi disponibilizada em Maio de 2007. O seu desenvolvimento está muito condicionado porque se pretende que jovens estudantes possam aplicar as suas ideias neste pequeno projecto.

3.2.9 UNO

O UNO[3] é um analisador estático para código C na norma ANSI (Holzmann, 2002). Começou a ser desenvolvido em 2001, por Holzmann; a última versão (2.12) foi lançada em Agosto de 2007.

O UNO usa um conjunto de propriedades pré-definidas e sobre elas analisa a aplicação. Dá oportunidade aos utilizadores de definirem as suas propriedades e de estas serem utilizadas durante a análise, funcionando de maneira semelhante

[1] Mais informação consulte, `http://www.kernel.org/pub/software/devel/sparse/` (visitado em Setembro de 2007).

[2] Sobre a norma ANSI consulte, Kernighan & Ritchie (1988).

[3] É possível encontra a ferramenta e mais informação em, `http://spinroot.com/uno/` (visitado em Setembro de 2007).

ao MOPS. Permite também escolher a forma de análise, global ou local, isto é, analisa localmente ficheiro a ficheiro ou todos os ficheiros.

A ferramenta consegue detectar três tipos de vulnerabilidades: a não inicialização de uma variável, referência nula de um apontador e acesso fora dos limites de uma lista. Nas três vulnerabilidades que consegue detectar fornece bons resultados, com um elevado grau de precisão.

O UNO foi desenvolvido para detectar vulnerabilidades muito específicas, deste modo, consegue concentrar-se em problemas importantes, obtendo um maior número de verdadeiros positivos do que falsos positivos, garantindo alguma qualidade aos resultados.

3.3 Teste Padronizado para Análise Estática

Como foi possível constatar a partir da amostra apresentada na secção anterior, existem neste momento várias ferramentas de análise estática de código fonte disponíveis na *Internet*. No entanto, como em muitos casos, estas ferramentas resultam de projectos de investigação, normalmente têm um suporte limitado e a documentação é incompleta. Torna-se assim difícil a identificação real das suas verdadeiras capacidades de detecção. Por outro lado, actualmente não se encontram disponíveis bons métodos que permitam avaliar e comparar as ferramentas quanto ao seu funcionamento, o que complica a escolha das ferramentas a usar.

Os testes padronizados permitem avaliar e comparar sistemas ou componentes de acordo com determinadas características (Vieira, 2005). Genericamente, as características a avaliar nas ferramentas de análise estática são o número e tipo de vulnerabilidades que são detectadas e o número de falsos alertas reportados.

A vantagem de se utilizarem testes padronizados consiste em se identificar a aptidão de cada ferramenta através de um teste simples e reproduzível, e assim, por exemplo desmistificar ferramentas que garantem a localização de 100% de vulnerabilidades. Com a utilização do teste ir-se-á confirmar (ou descrebilizar) citações como a ferramenta x tem menos falsos positivos que a y.

Para que o teste padronizado se torne num meio confiável de comparação, deve haver uma validação adequada do teste, caso contrário, as conclusões podem ser

erradas. Na secção seguinte é apresentado um conjunto de propriedades que devem ser satisfeitas pelo teste padronizado.

3.3.1 Propriedades do Teste Padronizado

Um teste padronizado deve satisfazer as seguintes propriedades: representatividade, portabilidade, reprodutibilidade, não-intrusividade e simplicidade de utilização (Vieira, 2005). Nas subsecções seguintes descreve-se como um teste padronizado satisfaz cada uma destas propriedades.

3.3.1.1 Representatividade

O teste padronizado só pode devolver resultados interessantes se representar os tipos de problemas que são observados na realidade. Caso contrário, os resultados não caracterizam uma utilização real das ferramentas do teste.

No nosso caso, o teste padronizado deve-se basear num conjunto de vulnerabilidades a detectar, de maneira adequada e útil, ou seja, as medidas devem reflectir as características de um conjunto de vulnerabilidades de maneira a permitir uma comparação correcta entre diferentes ferramentas.

Os métodos de avaliação estão sempre relacionados com o conjunto de medidas especificadas e uma vez definidos podem ser reajustados ou redefinidos, mas sempre com a mesma equidade para com todas as ferramentas.

3.3.1.2 Portabilidade

A portabilidade num teste padronizado significa que este deve permitir a comparação equitativa de entre diferentes ferramentas, pertencentes ao mesmo domínio de aplicação.

A melhor forma de avaliar a portabilidade consiste em concretizar e executar o teste padronizado num conjunto de ambientes de execução, com características diferentes. Através desta avaliação consegue-se verificar a aplicabilidade do teste padronizado em sistemas muito diferentes.

3.3.1.3 Reprodutibilidade

Entende-se por reprodutibilidade de um teste padronizado à capacidade de produzir resultados similares, quando executado mais do que uma vez no mesmo ambiente. Uma vez que as ferramentas de análise estática que iremos considerar não têm variações quanto ao ambiente de execução, somente o desempenho, os resultados obtidos serão tipicamente iguais.

É possível comprovar que um teste padronizado é reprodutível através de várias execuções no mesmo sistema, sendo que, os resultados deverão ser analisados e as potenciais variações identificadas. Se o ambiente de execução for diferente, então dependendo da ferramenta é possível que a sua execução não seja reprodutível. Sempre que houver variações devem-se anotar para futura análise, de maneira a que se possa tornar o teste tão imune às características do ambiente quanto possível.

3.3.1.4 Escalabilidade

Um teste padronizado deve ser capaz de avaliar uma ferramenta com cargas (ou programas) de diferentes dimensões.

A forma como a escala pode ser alterada é através do aumento de linhas de código a processar, com ou sem novas vulnerabilidades inseridas.

3.3.1.5 Não-intrusividade

Esta propriedade exige que um teste padronizado não obrigue a alterações nas ferramentas, sendo considerado intrusivo quando há modificações para que ele possa ser executado. Por outro lado, caso se tenha que modificar o teste padronizado, existe uma redução na portabilidade do teste padronizado, além de que pode alterar as características dos resultados.

A propriedade não-intrusividade é normalmente fácil de verificar, quando se aplica o teste padronizado sobre uma ferramenta e esta necessita de uma configuração específica para ser utilizada.

3.3.1.6 Simplicidade de Utilização

Um teste padronizado deve ser fácil de usar na prática, estando a sua potencial aceitação pela comunidade fortemente relacionada com a sua simplicidade, pois os utilizadores tendem a ignorar testes padronizados de difícil concretização e execução. Para além disso, a execução de um teste padronizado deve ser pouco dispendiosa, completamente automática e não deve consumir demasiado tempo.

Esta propriedade garante também que a especificação é clara e completa, assegurando que essa especificação cobre todas as questões relevantes que possam surgir durante a sua utilização.

Por último, esta propriedade garante que o tempo para executar o teste padronizado deve ser avaliado. O ideal é que a sua execução não deverá ocupar mais alguns minutos por ferramenta.

3.3.2 Classes de Vulnerabilidades

As vulnerabilidades estão na base dos ataques feitos a uma aplicação ou sistema, de modo a prevenir a existência de falhas é necessário definir e compreender as vulnerabilidades existentes. Assim, abaixo podem-se observar as diferentes categorias de vulnerabilidades que foram consideradas para inclusão no teste.

1. Inicialização;
2. Conversão explicita de tipo;
3. Validação de parâmetros;
4. Validação do valor de retorno;
5. Gestão de memória;
6. Funções com limitação de valores;
7. Acesso a ficheiros;
8. Redução de privilégios;
9. Criptografia;
10. Bibliotecas;
11. Tratamento de excepções;

12. Multi-tarefas.

A cada uma destas classes de vulnerabilidades foram associados vários exemplos práticos onde elas se materializam (ver Apêndice F para uma lista detalhada). A cada momento é possível adicionar e/ou remover os exemplos de vulnerabilidades considerados, dado que, a lista actual se encontra em evolução. A versão actual da lista de vulnerabilidades foi construída através de material recolhido de diferentes fontes. Nomeadamente, foram estudados os excertos de código de teste que eram fornecidos com as ferramentas, e foram investigados vários sítios na *web* que se dedicam a disponibilizar informação sobre a descoberta de novas vulnerabilidades, e também se adicionou vulnerabilidades criadas pelo autor da tese.

No Apêndice F, para além de se encontrarem as vulnerabilidades categorizadas, também são fornecidas algumas indicações de como podem ser evitadas através de uma programação mais cuidada (ou defensiva[1]). Esta categorização é muito útil para definir o tipo de ocorrências vulneráveis no teste padronizado, e assim avaliarmos todas as ferramentas com a mesma equidade.

3.3.3 Medidas de Avaliação

De seguida, vai-se estudar a eficácia e a precisão que são as medidas de avaliação das ferramentas, cuja definição se baseia no trabalho de Alessandri (2004) para detectores de intrusões:

- *Eficácia*: é a percentagem do número total de vulnerabilidades detectadas por uma ferramenta (m_d). O número total de vulnerabilidades detectadas corresponde verdadeiros positivos (m_d) reportados pela ferramenta. O número total de vulnerabilidades existentes é a soma entre as vulnerabilidades detectadas e as não detectadas (m_{nd}), mas que se sabia à partida que existiam no teste padronizado:

[1] Informações sobre programação defensiva em, `http://en.wikipedia.org/wiki/Defensive_programming` (visitado em Setembro de 2007), ou `http://www.embedded.com/1999/9912/9912feat1.htm` (visitado em Setembro de 2007).

$$e = \frac{m_d}{m_d + m_{nd}}. \tag{3.1}$$

- *Precisão*: é a percentagem do número total de vulnerabilidades detectadas pela ferramenta que corresponde aos verdadeiros positivos. O número total de alertas é o número de verdadeiros positivos (a_{vp}) mais o número de falsos positivos (a_{fp}):

$$p = \frac{a_{vp}}{a_{vp} + a_{fp}}. \tag{3.2}$$

As duas métricas dão-nos informação sobre a confiança dos alertas gerados pelas ferramentas de análise estática. Utilizou-se estas duas métricas porque englobam todos os dados relevantes para o estudo.

Existem outras métricas descritas em Alessandri (2004), o que não se aplicava ao comportamento das ferramentas utilizadas neste trabalho, alguns exemplos:

- *Rácio de ambiguidade*: não foram encontradas ambiguidades no relatório de cada ferramenta, ou seja, uma ferramenta informar que detectou a vulnerabilidade memória sobrelotada e devia ser acesso a ficheiros.

- *Rácio de identidade*: não houve problemas em que uma ferramenta informou que há uma vulnerabilidade na linha n sem descrição, impedindo assim a identificação do problema. Por isso não foi necessário efectuar uma abordagem quanto às vulnerabilidades detectadas sem ou com descrição.

3.3.4 Versão Preliminar de um Teste Padronizado

O teste padronizado desenvolvido garante praticamente todas as propriedades anunciadas anteriormente. No Apêndice G pode-se observar o programa que deve ser analisado pelas ferramentas. Na versão actual, o teste padronizado tem concretizado as classes de vulnerabilidades 1 a 9, O teste padronizado tem

concretizado as classes de vulnerabilidades 1 a 9 (ver Secção 3.3.2), no entanto, na propriedade de criptografia só foi possível trabalhar com números aleatórios.

Relativamente às propriedades, a versão actual do teste padronizado é representativo porque contém diversos tipos de problemas que são observados na realidade. Quanto à portabilidade, o teste só foi testado para o ambiente Linux, no entanto, não se vê nenhum equivoco quando for executado em outro ambiente, desde que, se mantenha os requisitos que inicialmente foram dispostos. A reprodutibilidade é cumprida no teste padronizado, este não sofre variações durante a sua execução reproduzindo invariavelmente os mesmo resultados. Na escalabilidade, o teste padronizado não garante esta propriedade porque o teste tem um tamanho fixo e não foi adicionado um número significativo de vulnerabilidades (mais linhas de código) para provar a propriedade. O teste padronizado criado não necessita de nenhuma alteração para ser executado pelas ferramentas, daí a não-intrusividade. Por fim, a simplicidade de utilização é plena porque o teste padronizado é fácil de trabalhar, de acrescentar novas vulnerabilidades e novos testes, e cada vulnerabilidade está identificada para o seu tipo. Porém pode-se detalhar melhor a razão da vulnerabilidade corrente. À medida que o teste for crescendo com funções vulneráveis, convém dividir cada categoria de vulnerabilidades por ficheiro, para uma melhor utilização.

3.4 Desenvolvimento do Mute

As ferramentas de análise estática têm como objectivo detectar vulnerabilidades existentes nos programas, e que possam pôr em causa a segurança. A grande vantagem de se utilizar ferramentas de análise estática é rever o código desenvolvido muito antes deste ser entregue para testes, evitando o dispêndio de tempo em testes e correcções.

Para auxiliar os programadores durante a fase de concretização da aplicação seria conveniente a utilização de um analisador estático para reclamar potenciais vulnerabilidades. O problema assenta no número de falsos positivos que estas ferramentas reportam, assim a ferramenta a criar deveria possuir uma estratégia para diminuição de falsos positivos.

Do nosso estudo sobre diversas ferramentas disponíveis na *Internet* (ver Capítulo 4) foi possível observar que existem ferramentas vocacionadas para encontrar diferentes tipos de vulnerabilidades, e que empregam distintas técnicas de análise. É de notar, no entanto, que não há uma ferramenta que analise todo o tipo de vulnerabilidades.

Para isso, desenvolveu-se uma ferramenta denominada por *Mute*, que trabalha com várias ferramentas em diferentes aspectos de análise e classes de vulnerabilidades. O Mute recolhe os resultados de análise de cada ferramenta, também designados por "relatórios", e formata os diversos relatórios num modelo pré-definido. A partir daqui, agrega os resultados de acordo com um determinado algoritmo, e depois decide-se que resultados devem ser retornados ao utilizador.

Na secção seguinte apresenta-se a arquitectura da ferramenta, detalhando os componentes da arquitectura.

3.4.1 Arquitectura

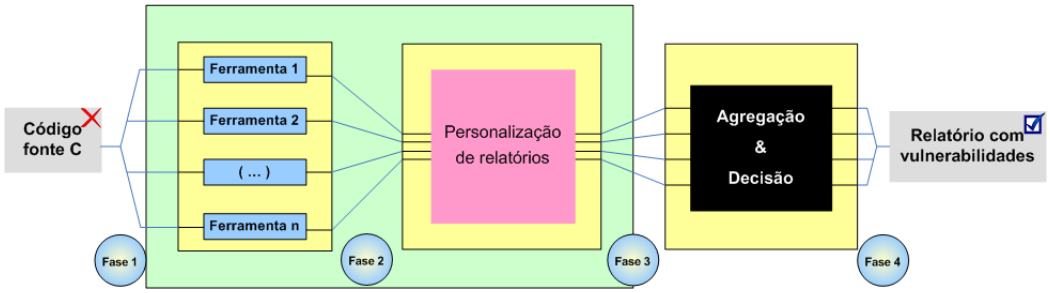


Figura 3.1: Arquitectura da ferramenta desenvolvida.

Na Figura 3.1, tem-se uma representação da arquitectura da ferramenta desenvolvida — Mute. Como se pode observar, a ferramenta encontra-se organizada em dois módulos independentes. O primeiro módulo divide-se em duas componentes, a primeira componente trata da recepção do código fonte C, possivelmente vulnerável, e entrega-o às ferramentas para análise (fase 1). Depois são extraídos os relatórios gerados por cada ferramenta, formatando-os para um relatório modelo (fase 2). O segundo módulo, que é totalmente independente do primeiro, tem por objectivo analisar os vários relatórios (fase 3), a partir do qual, se cria

um único relatório, que é elaborado através da procura de ocorrências em comum pelas outras ferramenta (fase 4).

As ferramentas utilizadas na arquitectura do Mute são: Crystal, Deputy, Flawfinder, ITS4, MOPS, PScan, RATS, Sparse e UNO.

Em seguida serão consideradas três questões importantes à concretização do Mute, que serão melhor analisadas, nas próximas secções:

Primeira questão - Como se coloca todas as ferramentas a analisar o mesmo código fonte?

Segunda questão - Como são formatados os relatórios gerados pelas ferramentas?

Terceira questão - Como se sabe que há vulnerabilidades em comum pelas diversas ferramentas?

O Mute foi construído de maneira a exibir as propriedades da modularidade e portabilidade. É modular uma vez que há independência entre cada componente, e é portável porque é independente da linguagem de programação analisada e pode processar um ou mais relatórios.

3.4.2 Ferramentas Usadas e Tratamento dos Relatórios

Nesta subsecção descreve-se o primeiro módulo da Figura 3.1, que por sua vez contém duas componentes. Este módulo foi concretizado por meio da linguagem *Shell Script*[1]. Escolheu-se o *Shell Script* por ser uma linguagem fácil de trabalhar, uma vez que é possível manipular de forma simples as aplicações por linha de comandos.

O primeiro componente recebe a localização do(s) ficheiro(s) de código a tratar pelas ferramentas e procede à sua análise. A análise é feita de forma sequencial, depois de uma acabar uma análise começa outra, e assim sucessivamente até não haver mais ferramentas. Depois de uma ferramenta analisar o código fonte C, deixa o seu relatório das possíveis vulnerabilidades numa directoria específica.

[1] Informação sobre o *Shell Script* em, `http://www.cyberciti.biz/nixcraft/linux/docs/uniqlinuxfeatures/lsst/` (visitado em Setembro de 2007).

A segunda componente pega nos relatórios e formata-os num modelo específico (exemplo, ver Figura 3.2 e 3.3). Após a formatação, cada relatório é copiado para uma outra directoria, isto para o segundo módulo analisar.

```
ficheiro.c:linha:ferramenta:vulnerabilidade:descrição
descrição
(...)
ficheiro.c:linha:ferramenta:vulnerabilidade:descrição
(...)
```

Figura 3.2: Modelo de formatação do relatório para o Mute.

Um ficheiro de relatório deverá ter o modelo apresentado na Figura 3.2, onde se tem o nome do ficheiro C analisado; a linha onde se encontrou a vulnerabilidade; o nome da ferramenta de análise; o tipo de vulnerabilidade que a ferramenta detectou, que corresponde a um tipo de classes de vulnerabilidades (ver Secções 3.3.2 e 4.2); por fim, pode-se ter uma descrição sobre o problema encontrado na mesma linha ou na(s) linha(s) seguinte(s).

```
vulncode.c:336:Deputy:BUFFER: gets(p)
in fct func_3_13, uninitialized variable 'p'
vulncode.c:359:Deputy:BOUND: aux_str[i]='\0'
in fct func_3_14, array index can exceed upper-bound (10>9), var 'i'

(...)

vulncode.c:213:PScan:FORMAT: syslog non-constant string for argument 1
vulncode.c:351:PScan:FORMAT: printf call should have "%s"as argument 0
```

Figura 3.3: Excerto de um relatório de duas ferramentas.

A segunda componente trata cada relatório com comandos UNIX, como o *sed*[1], para substituir caracteres específicos. Este é executado com comandos específicos para cada ferramenta, obtendo-se algo semelhante ao apresentado na Figura 3.3.

[1] Tudo sobre o comando *Sed* do Unix, `http://www.grymoire.com/Unix/Sed.html` (visitado em Setembro de 2007).

3.4.3 Agregação e Processamento de Resultados

Nesta subsecção descreve-se o "cérebro" da ferramenta, que é o segundo módulo apresentado na Figura 3.1. O módulo foi desenvolvido em Java[1], versão 1.5.

A ferramenta tem conhecimento da localização dos relatórios formatados e lê-os um a um, carregando o seu conteúdo para memória. Durante a leitura, o Mute verifica se a linha lida corresponde ao formato especificado (ver Figura 3.2), e se sim, então adiciona-a a uma *HashMap*[2]. Este *HashMap* usa como chave os seguintes dados: o ficheiro, a linha e tipo de vulnerabilidade que a ferramenta detectou. O valor da *HashMap* é um *Vector*[3] que irá conter um objecto do tipo *format*, correspondente ao tipo de formatação especificada na Figura 3.2, da linha lida correntemente, mais as linhas seguintes com descrição.

O Mute sempre que encontra uma linha que contenha uma chave já existente, adiciona um *format* ao conjunto de valores dessa chave, indicando que encontrou uma ferramenta com uma vulnerabilidade em comum. Depois de ler todos os relatórios, tem-se uma *HashMap* com os dados de todos os relatórios combinados, que é então usada para se produzir o relatório final. Para o construir basta ir a todas as ocorrências da tabela de dispersão e em cada ocorrência avalia-se a confiança associada. De acordo com essa avaliação, é então decidido se se deve ou não adicionar uma entrada no relatório final.

Para se fazer a avaliação é necessário definir um número de confiança ou certeza sobre cada ferramenta (ver exemplos práticos na Secção 4.3). Esse valor é calculado pelo o número de falsos positivos *versus* os verdadeiros positivos que são tipicamente observados nessa ferramenta, e daí surge um número entre 0.0 a 1.0. Este cálculo é feito antes do Mute executar, no entanto, a confiança pode variar no futuro consoante for o desempenho das ferramentas. Por isso, quando se afirma uma confiança de 80% numa ferramenta é porque esta consegue localizar

[1] Foi utilizado a linguagem Java porque possuí um conjunto de API's, que possibilitam uma fácil e rápida concretização do módulo.

[2] Esta classe é útil devido às suas propriedades como tabela de dispersão. Mais informação sobre a classe *HashMap* consulte, `http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html` (visitado em Setembro de 2007).

[3] Utilizou-se a classe *Vector*, para se poder adicionar mais valores, assim como é útil para percorrer os diversos valores. Mais informação sobre a classe consulte, `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Vector.html` (visitado em Setembro de 2007).

muitas vulnerabilidades com um grau de certeza muito grande, ou seja, com poucos falsos positivos.

O Mute actualmente avalia as ferramentas por base em dois tipos de confiança, uma genérica relacionada com a ferramenta, e outra relativa a cada tipo de vulnerabilidade que a ferramenta detecta.

No Mute a cada ocorrência do relatório global aplica-se um método de filtragem (soma ou média) ao valor resultante da avaliação da(s) vulnerabilidade(s). Se este for igual ou superior ao valor pré-definido de confiança em todas as ferramentas, por exemplo 0.5, então é adicionada informação sobre esta vulnerabilidade no relatório a devolver ao utilizador, caso contrário, a entrada é rejeitada.

O relatório fornecido ao utilizador é um ficheiro de texto com as vulnerabilidades descobertas pelas nove ferramentas, e que o Mute considera como potenciais verdadeiros positivos.

Capítulo 4

Avaliação

Neste capítulo procede-se a duas avaliações, uma sobre as ferramentas estudadas e outra sobre o desempenho do Mute. Um dos objectivos desta avaliação é a recolha de dados que ajudem a demonstrar que o Mute potencialmente consegue encontrar mais vulnerabilidades com um número de falsos positivos reduzido.

4.1 Ambiente de Execução

A avaliação foi conduzida sobre o sistema operativo Linux, com a distribuição Fedora[1] Core 3.0 e o kernel 2.6. O computador usado foi um Fujitsu Siemens Computers Scenic P300, com processador Intel Pentium IV 2.80GHz de 32 bits, com cache de 128KB e 512MB DDR SDRAM de memória.

Algumas ferramentas têm requisitos antes de instalar, por exemplo podem necessitar do CIL ou o OCAML, e todas requerem um compilador. O compilador utilizado no ambiente de testes foi o GCC 3.4.2, visto que é um requisito obrigatório para a ferramenta UNO[2]. O teste padronizado foi também compilado com o GCC 3.4.2.

Cada ferramenta tem determinadas opções, tendo sido necessário fazer uma selecção das que deveriam ser usadas. Em seguida, são mostrados os comandos

[1] Informação sobre o ambiente Fedora em, `http://fedoraproject.org/wiki` (visitado em Setembro de 2007).

[2] Caso contrário o relatório da ferramenta tem o número da linha da vulnerabilidade incorrecto.

utilizados para executar as ferramentas durante as experiências:

- *Crystal*: lc;
- *Deputy*: deputy;
- *Flawfinder*: flawfinder -m 1 -Q -F -D -S;
- *ITS4*: its4 -c 2 -s 2 -w;
- *MOPS*: mops -m <nome_propriedade.mfsa> -r trace -o output -t temp -v warn;
- *PScan*: pscan -w;
- *RATS*: rats –resultsonly -l c -w 3;
- *Sparse*: sparse;
- *UNO*: uno_local -picky -allerr -fullpaths.

4.2 Ferramentas

Nesta avaliação pretende-se determinar o quanto as ferramentas nos oferecem em termos de localização de vulnerabilidades (verdadeiros positivos), as que não detectam (falsos negativos) e as que são detectadas erradamente (falsos positivos). Todas as ferramentas partem de uma situação idêntica, ou seja, todas vão examinar o mesmo ficheiro de código fonte C em busca de vulnerabilidades. Só assim se pode comparar as ferramentas, até mesmo aquelas que utilizam a mesma técnica de análise e/ou procuram as mesmas vulnerabilidades. O ficheiro de código fonte C usado no teste pode ser consultado no Apêndice G.

O código padronizado tem vulnerabilidades e essas foram categorizadas de acordo com as classes de vulnerabilidades especificadas na Secção 3.3.2. Para simplificar as tabelas e gráficos ao longo deste capítulo, vão-se identificar as vulnerabilidades da seguinte maneira:

INIT - Não inicialização;

CAST - Incorrecta conversão entre tipos de dados;

PVAL - Falta de validação de parâmetros;

BUFFER - Escrever em zonas de memória ilegais;

FORMAT - Formatação de cadeias de caracteres em falta;

BOUND - Acesso fora dos limites de uma lista;

INTEGER - O espaço de endereçamento do inteiro excedido;

ZERO - Possível divisão por zero;

LOOP - Ciclo infinito;

RET - Valor de retorno esquecido ou não validado;

MEM - Falha na gestão de memória;

FILA - Incorrecto acesso a ficheiros;

LEAST - Necessário diminuir os privilégios do utilizador;

RANDOM - Geração de números aleatórios com limitações.

Uma vez definido os tipos de vulnerabilidades é possível executar as ferramentas com o código de teste. No entanto, antes de este ser analisado pelas ferramentas foi necessário compilar com a opção *-Wall* do GCC de modo a garantir a isenção de erros de sintaxe e semânticos. Optou-se por deixar passar apenas dois alertas, um que indicava a função *gets()* como insegura e outro que indicava a função *mktemp()* como insegura. Não foram utilizadas mais opções do compilador assumindo que os programadores no mínimo correm os seus códigos fonte com a opção *-Wall*.

Para analisar o teste padronizado procedeu-se à execução das ferramentas, e obteve-se os resultados apresentados na Tabela 4.1. Nesta tabela temos a primeira coluna com o nome das classes de vulnerabilidades e de seguida o número de vulnerabilidades existente no teste para cada tipo de vulnerabilidade. Depois é mostrado para cada ferramenta o número de verdadeiros positivos[1] (VP) e de falsos positivos[2] (FP). Esta tabela faz uma combinação entre o tipo de vulnerabilidades e o número de VP e FP para cada ferramenta.

[1] Vulnerabilidades correctamente detectadas.
[2] Falso alerta de identificação de uma vulnerabilidade.

	Vulnerabilidades no teste	Crystal		Deputy		Flawfinder		ITS4		MOPS		PScan		RATS		Sparse		UNO	
		VP	FP	VP	FP	VP	FP	VP	FP	VP	FP	VP	FP	VP	FP	VP	FP	VP	FP
INIT	21			3	2											0	2	13	1
CAST	5			1	1											1	0		
PVAL	9			9	4														
FORMAT	10			5	8	6	7	10	135	0	1	4	6	7	8				
BUFFER	40			16	10	17	26	11	13	1	0	1	1	25	35				
BOUND	35					0	7							0	7	5	0	20	1
INTEGER	46																		
ZERO	3																		
LOOP	11																		
RET	13			1	0													0	12
MEM	34	13	1											0	32				
FILA	16			2	6	15	12	9	13	9	1			7	12				
LEAST	11			2	0	10	3	10	3	0	1			10	6				
RANDOM	10					1	4	10	3					1	4				
Total	**264**	**13**	**1**	**39**	**31**	**49**	**59**	**50**	**167**	**10**	**3**	**5**	**7**	**50**	**104**	**6**	**2**	**33**	**14**

Tabela 4.1: Verdadeiros e falsos positivos (VP & FP) para cada ferramenta, depois da análise do teste.

Todas as ferramentas emitiram um relatório com vulnerabilidades, no entanto, este relatório não contém a informação relativa à veracidade da vulnerabilidade. Por isso, analisou-se cada relatório manualmente, linha a linha e comparando com o que era expectável em relação às vulnerabilidades existentes no teste padronizado. Deste modo, foi identificada a existência ou não de vulnerabilidades, uma vez que no teste padronizado já tenhamos identificado todas as vulnerabilidades possíveis de ocorrer, tendo bastado apenas comparar e contabilizar as vulnerabilidades correctamente descobertas ou em falta.

Observando a Tabela 4.1 percebe-se que não há nenhuma ferramenta a detectar todas as vulnerabilidades no teste padronizado, assim como, são pouquíssimas as ferramentas a detectarem a totalidade das vulnerabilidades. Isto demonstra que ainda existe muito trabalho a desenvolver na detecção estática de vulnerabilidades.

As ferramentas de análise de padrão reportaram muitos alertas, sendo que um alerta emitido por estas ferramentas pode ser verdadeiro ou falso. Para além disso, as ferramentas informam da potencial ocorrência de vulnerabilidades, caso o programador não tenha em conta alguma directivas para resolver a questão. Assim, é observado às vezes um maior número de falsos alertas.

A ferramenta Crystal é aquela com melhores resultados entre VP's e FP's, para a vulnerabilidade gestão de memória. A ferramenta ITS4 tem um número elevado de FP's na formatação de cadeias de caracteres, porque quando encontra uma

função *printf()* normalmente reporta-a como vulnerável. É de salientar também a qualidade do MOPS em detectar problemas de acesso a ficheiros, sendo uma das suas especialidades. Por último, o UNO é a ferramenta com melhores resultados em detectar a vulnerabilidade acesso fora do limite de uma lista.

As ferramentas que procuram vulnerabilidades de controlo de acesso e condições de disputa, requerem por omissão que o programa execute com privilégios de administrador (*root*). O teste padronizado tem código com a proposição de que o programa irá correr com privilégios *root*, caso contrário teria-se um maior número de falsos alertas, visto que as ferramentas analisam o código fonte considerando o pior caso.

Notou-se que algumas ferramentas reportam a mesma vulnerabilidade até duas vezes, porém são poucas as ferramentas que o fazem. Do ponto de vista de análise de resultados, este problema foi eliminado com o comando *unique*[1] (ao nível do Shell Script pelo segundo componente do primeiro módulo, ver Figura 3.1) que elimina linhas repetidas. Em alguns casos também foi observado relatórios com a mesma linha repetida mas com vulnerabilidades diferentes. Neste caso, foi considerado que a linha de código tinha mais do que um problema, e deste modo é contabilizada cada linha como uma vulnerabilidade diferente das outras.

Uma ferramenta pode reportar mais que uma vulnerabilidade no mesmo ficheiro e na mesma linha. Por exemplo, na Listagem 4.1, temos duas vulnerabilidades, *memória sobrelotada* e *acesso fora do limite da lista*, a primeira devido ao *dst* poder ser menor que o *src*, e no segundo pode faltar o caracter "\0". É aqui possível ver que não basta considerar uma vulnerabilidade como o nome do ficheiro que esta ocorreu e linha, mas é também necessário considerar o tipo de vulnerabilidade.

```
strncat(dst, src, sizeof(dst)-strlen(src));
```

Listagem 4.1: Duas vulnerabilidades numa só linha.

A Tabela 4.2 mostra a eficácia das ferramentas para detectar as vulnerabilidades no teste padronizado (onde se utilizou a Equação 3.1) e a precisão ilustra

[1] Sobre o comando em, `http://xgen.iit.edu/cgi-bin/man/man2html?8+unique` (visitado em Setembro de 2007).

a qualidade destas a encontrar vulnerabilidades de segurança (onde se utilizou a Equação 3.2).

Ferramentas	Vulnerabilidades Detectadas	Vulnerabilidade Não Detectadas	Eficácia	Vedadeiros Positivos	Falsos Positivos	Precisão
Crystal	13	251	4,9%	13	1	92,9%
Deputy	39	225	14,8%	39	31	55,7%
Flawfinder	49	215	18,6%	49	59	45,4%
ITS4	50	214	18,9%	50	167	23,0%
MOPS	10	254	3,8%	10	3	76,9%
PScan	5	259	1,9%	5	7	41,7%
RATS	50	214	18,9%	50	104	32,5%
Sparse	6	258	2,3%	6	2	75,0%
UNO	33	231	12,5%	33	14	70,2%

Tabela 4.2: Eficácia e precisão das ferramentas.

Analisando cuidadosamente as Tabelas 4.1 e 4.2, vê-se que não há ferramentas a detectar todos os tipos de vulnerabilidades e todas as vulnerabilidades. A única ferramenta que sobressai é o Crystal, responsável por detectar vulnerabilidades de gestão de memória, a partir de uma análise de fluxo de dados e controlo do fluxo por grafos. Estas duas técnicas levam a que a ferramenta seja muito precisa, embora tenha um baixo valor de eficácia, porque está muito especializada para encontrar problemas de gestão de memória, deixando as outras vulnerabilidades por detectar. A ferramenta com piores resultados é o ITS4, devido à utilização da análise de padrão e sua presunção. O ITS4 tem um problema que acabou por o influenciar negativamente, que é o elevado número de falsos positivos na formatação da cadeia de caracteres, quase sempre devido à função *printf()*, que era reportada como vulnerável.

Assim, vai-se analisar a eficácia e precisão das ferramentas escolhidas para análise do teste padronizado. Para tal, vai-se usar o Gráfico 4.1, que mostra a relação entre a eficácia e precisão para cada ferramenta.

Como se pode verificar no Gráfico 4.1, metade das ferramentas estão acima dos 50% de precisão o que é bom. Estas ferramentas são distinguidas pelas técnicas de análise adoptadas. As ferramentas abaixo dos 50% correspondem às ferramentas de análise de padrão, análise esta susceptível de reportar um número

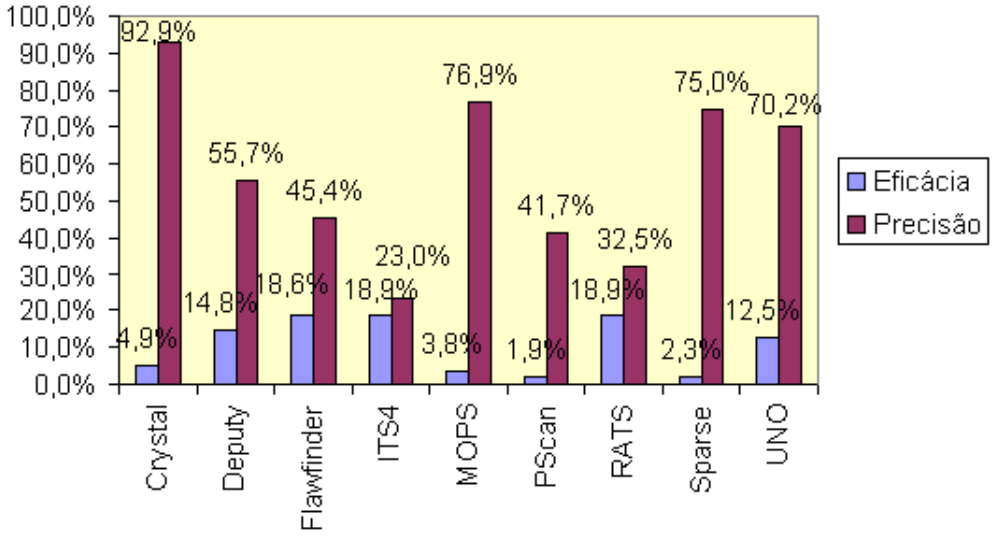


Figura 4.1: Comportamento da eficácia e precisão pelas ferramentas.

maior de falso alertas. Em contrapartida, as ferramentas de análise por padrão conseguem detectar um maior número de classes de vulnerabilidades , que as outras ferramentas — maior eficácia. As ferramentas que têm uma eficácia menor simplesmente detectam apenas uma categoria de vulnerabilidades.

4.3 Mute

Através da análise da secção anterior verificou-se que as ferramentas são capazes de detectar tipos diferentes de vulnerabilidades, assim como, é possível observar a qualidade de cada uma pela precisão. No total as ferramentas detectaram mais de metade das vulnerabilidades, o que é muito interessante porque o teste padronizado contém algumas "ratoeiras" para os analisadores.

Visto que as ferramentas separadamente não tiveram um desempenho muito satisfatório, a estratégia de combinar a detecção de ocorrências de vulnerabilidades tem o potencial de reportar mais vulnerabilidades que uma ferramenta só. No entanto, se agregação de resultados for efectuada de uma maneira simplista (exemplo, juntando todas as vulnerabilidades reportadas), pode-se cair na situação de se ter um número muito elevado de falsos positivos (logo, baixa precisão), o que reduziria em muito a utilidade dos resultados.

	Vulnerabilidades no teste	Falsos Negativos Combinados	Verdadeiros Positivos Combinados	Falsos Positivos Combinados
INIT	21	5	16	5
CAST	5	3	2	1
PVAL	9	0	9	4
FORMAT	10	0	10	138
BUFFER	40	15	25	46
BOUND	35	14	21	8
INTEGER	46	46	0	0
ZERO	3	3	0	0
LOOP	11	11	0	0
RET	13	12	1	12
MEM	34	21	13	33
FILA	16	0	16	19
LEAST	11	1	10	7
RANDOM	10	0	10	5
Total	264	131	133	278

Tabela 4.3: Verdadeiros e falsos positivos combinados das várias ferramentas.

É de salientar também a qualidade do teste porque caso fosse feito de forma simples todas as ferramentas detectavam as vulnerabilidades. Este no entanto foi criado para perceber até onde as ferramentas conseguem detectar, daí os resultados nas Tabelas 4.1 e 4.3. É possível observar que há algumas vulnerabilidades que nunca chegam a ser encontradas, o mesmo se diz para os falsos alertas que são produzidos.

Na Tabela 4.3 temos a informação sobre os verdadeiros e falsos positivos combinadas para cada tipo de vulnerabilidade, ou seja, retirando-se duplicados — se uma ou mais ferramentas detectaram a mesma vulnerabilidade então esta é contabilizada apenas uma vez.

É interessante comparar os verdadeiros positivos combinados com as vulnerabilidades no teste padronizado. É possível concluir que as ferramentas apenas não detectam as vulnerabilidades: INTEGER, ZERO e LOOP. Assim, pode-se assumir que o Mute irá detectar no máximo 133 vulnerabilidades em 264, faltando por encontrar 131, que correspondem aos falsos negativos. Porém, se se subtrair os três tipos de vulnerabilidades que não são detectadas, por não possuírem conhecimentos suficientes (INTEGER, ZERO e LOOP), então tem-se um total de 204 vulnerabilidades para 133 localizadas. Visto nesta perspectiva, poder-se-ia considerar um bom resultado.

Existe, no entanto, um outro problema — os falsos positivos. Se simplesmente se combinarem os resultados ir-se-á ter 278 falsos alertas para 133 vulnerabilidades detectadas. Apesar disto, uma ferramenta ao encontrar 133 vulnerabilidades de segurança é algo muito relevante para quem planeia desenvolver aplicações.

Uma vez que as ferramentas produzem um número considerável de falsos alertas com o teste padronizado, então deve-se pensar em técnicas para reduzir este número. A abordagem para combinar os relatórios das ferramentas é baseada em observar:

a) qual das ferramentas fez a detecção e;

b) quantas ferramentas localizaram a mesma vulnerabilidade.

Assim, a **primeira técnica** abordada consistiu em atribuir uma confiança a cada ferramenta. Quando se tem a mesma vulnerabilidade detectada por ferramentas diferentes (possivelmente com confianças diferentes), é necessário decidir qual o valor de confiança que deve ser associado à existência dessa vulnerabilidade. Foram considerados dois métodos diferentes, um baseado numa média em que se somam as confianças e divide-se pelo número total de confianças – *método 1*; e o outro baseado apenas na soma dos valores das confianças – *método 2*. Esta técnica também necessita da definição de um valor de limiar a partir do qual o Mute aceite a vulnerabilidade. Este valor é usado para diminuir os falsos alertas, mas não deve impedir a apresentação das vulnerabilidades que realmente existem.

A partir da Tabela 4.1 constata-se que há ferramentas que são melhores que outras para certas classes de vulnerabilidades. Como tal, propõe-se uma **segunda técnica**, que em vez de atribuir uma confiança global às ferramentas, atribui-se uma confiança que é especifica para cada tipo de vulnerabilidade e ferramenta. Os métodos de avaliação das confianças adoptados são semelhantes à técnica anterior, mas desta vez designou-se por *método 3* ao método 1, dado que trabalha com as confianças por tipo de vulnerabilidade; e foi também designado o *método 4* ao método 2, mas para confianças do tipo de vulnerabilidade.

As confianças atribuídas para a primeira técnica têm por base os resultados da Tabela 4.2 na coluna de precisão. A segunda técnica utiliza a confiança baseada na Tabela 4.1 a partir da qual se obteve a Tabela 4.4. Os valores apresentados na

	Crystal	Deputy	Flawfinder	ITS4	MOPS	PScan	RATS	Sparse	UNO
INIT		60%						0%	93%
CAST		50%						100%	
PVAL		69%							
FORMAT		38%	46%	7%	0%	40%	47%		
BUFFER		62%	40%	46%	100%	50%	42%		
BOUND			0%				0%	100%	95%
INTEGER									
ZERO									
LOOP									
RET		100%							0%
MEM	93%						0%		
FILA		25%	56%	41%	90%		37%		
LEAST		100%	77%	77%	0%		63%		
RANDOM			20%	77%			20%		

Tabela 4.4: Confiança para cada ferramenta e tipo de vulnerabilidade.

Tabela 4.4 foram calculados usando a definição de precisão no Capítulo anterior. A última técnica é possivelmente a mais favorável, uma vez que vai ao detalhe de cada ferramenta, ou seja, na vulnerabilidade que cada ferramenta é melhor a detectar.

O Mute foi então concretizado de maneira a que fornecesse resultados com algoritmos de agregação baseados nas duas técnicas. Os parâmetros de testes para avaliar o Mute foram os seguintes:

Primeiro teste Não é depositada qualquer confiança quer na ferramenta, quer no tipo de vulnerabilidade, deixando-se simplesmente que o Mute combine as vulnerabilidades que podem ser combinadas.

a) A confiança para cada ferramenta é 0.0, não sendo utilizado nenhum dos métodos de avaliação das confianças;

Segundo teste Todas as ferramentas partem com a mesma confiança, ou seja, 0.5, não sendo atribuída confiança para os tipos de vulnerabilidades. São somente usados o método 1 e 2, para avaliação das confianças.

b) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.3;

c) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.5;

d) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.7;

e) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.3;

f) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.5;

g) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.7;

Terceiro teste As ferramentas partem com confianças diferentes, baseada na coluna da precisão da Tabela 4.2. Assim, o Crystal fica com uma confiança de 0.9, o Deputy com 0.6, o Flawfinder com 0.5, o ITS4 com 0.2, o MOPS com 0.8, o PScan com 0.4, o RATS com 0.3, o Sparse com 0.8 e o UNO com 0.7. Não é atribuída confiança para os tipos de vulnerabilidades. São somente usados o método 1 e 2, para avaliação das confianças.

h) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.3;

i) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.5;

j) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.7;

k) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.3;

l) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.5;

m) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.7;

Quarto teste As ferramentas partem com uma confiança diferente, baseada na coluna da precisão da Tabela 4.2. Assim, o Crystal fica com uma confiança de 0.9, o Deputy com 0.6, o Flawfinder com 0.5, o MOPS com 0.8, o PScan com 0.4, o Sparse com 0.8 e o UNO com 0.7. Não é atribuída confiança para os tipos de vulnerabilidades. São somente usados o método 1 e 2, para avaliação das confianças. Este teste é semelhante ao terceiro, mas sem o ITS4 e o RATS.

n) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.5;

o) Utiliza-se o método 1, o limiar para se aceitar uma vulnerabilidade é de 0.7;

p) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.5;

q) Utiliza-se o método 2, o limiar para se aceitar uma vulnerabilidade é de 0.7;

Quinto teste Aqui não há uma confiança global para cada ferramenta, mas esta é atribuída por o tipo de vulnerabilidade e ferramenta. Os valores de confiança utilizados são obtidos na Tabela 4.4 (exemplo, 38% $\approx$ 0.4). São somente usados os métodos 3 e 4 para avaliação das confianças.

r) Utiliza-se o método 3, o limiar para se aceitar uma vulnerabilidade é de 0.4;

s) Utiliza-se o método 3, o limiar para se aceitar uma vulnerabilidade é de 0.6;

t) Utiliza-se o método 3, o limiar para se aceitar uma vulnerabilidade é de 0.8;

u) Utiliza-se o método 4, o limiar para se aceitar uma vulnerabilidade é de 0.4;

v) Utiliza-se o método 4, o limiar para se aceitar uma vulnerabilidade é de 0.6;

w) Utiliza-se o método 4, o limiar para se aceitar uma vulnerabilidade é de 0.8.

A Tabela 4.5 resume os resultados obtidos no Mute. Para cada linha temos o teste descrito em cima. A primeira coluna informa a eficácia do Mute a detectar as vulnerabilidades no teste padronizado; depois temos as vulnerabilidades detectadas e os falsos alertas; por fim, a precisão onde se compara a capacidade do Mute detectar vulnerabilidades contra os falsos alertas.

O quarto teste foi realizado sem duas ferramentas pela razão que estas duas têm o maior número de falsos positivos (ver Tabela 4.2). Logo, será do interesse perceber o comportamento do Mute sem o ITS4 e RATS. Neste teste, foi retirado o limiar de 0.3 para garantir que a ferramenta com pior precisão (0.4) não era capaz sozinha de impor a aceitação das suas vulnerabilidades. No entanto, não se observaram melhorias muito significativas com este teste, pela razão que existem outras ferramentas de análise de padrão (Flawfinder) a detectar os mesmo problemas e este tem uma confiança de 0.5. Porém, é de salientar que mesmo assim se conseguiu obter mais vulnerabilidades detectadas do que falsos alertas.

O quinto teste fornece um conjunto de dados mais apelativos [u), v) e w)]. Fazendo-se uma comparação com todos os outros, notam-se que estes testes levam a melhor porque conseguem um bom equilíbrio entre a eficácia e a precisão, ao

Teste	Eficácia	Verdadeiros Positivos	Falsos Positivos	Precisão
a)	50,38%	133	278	32,36%
b)	50,38%	133	278	32,36%
c)	50,38%	133	278	32,36%
d)	0,00%	0	0	0,00%
e)	50,38%	133	278	32,36%
f)	50,38%	133	278	32,36%
g)	15,15%	40	92	30,30%
h)	45,45%	120	143	45,63%
i)	26,14%	69	40	63,30%
j)	18,94%	50	19	72,46%
k)	45,83%	121	145	45,49%
l)	45,83%	121	96	55,76%
m)	43,18%	114	76	60,00%
n)	45,45%	120	94	56,07%
o)	19,70%	52	19	73,24%
p)	45,45%	120	94	56,07%
q)	30,68%	81	38	68,07%
r)	46,59%	123	83	59,71%
s)	32,95%	87	23	79,09%
t)	23,11%	61	4	93,85%
u)	49,24%	130	96	57,52%
v)	48,48%	128	66	65,98%
w)	46,97%	124	53	70,06%

Tabela 4.5: Resultados dos testes sobre o Mute.

invés dos outros onde se tem baixa eficácia e alta precisão ou vice versa. É difícil dizer qual é o melhor dos três, porém é possível afirmar que o teste v) é provavelmente o melhor. Estes três testes estiveram muito perto de detectar todas as vulnerabilidades (133).

No Gráfico 4.2 é possível observar um crescimento da precisão a partir do teste h), ou seja, do terceiro teste, o que é de esperar uma vez que as ferramentas deixaram de ter uma confiança estática de 0.5 e determinou-se uma confiança mais exacta para cada ferramenta.

Nos primeiros testes [a), b), c), e), f)], observa-se uma eficácia a 50% que corresponde ao máximo de vulnerabilidades que o Mute poderia detectar. O

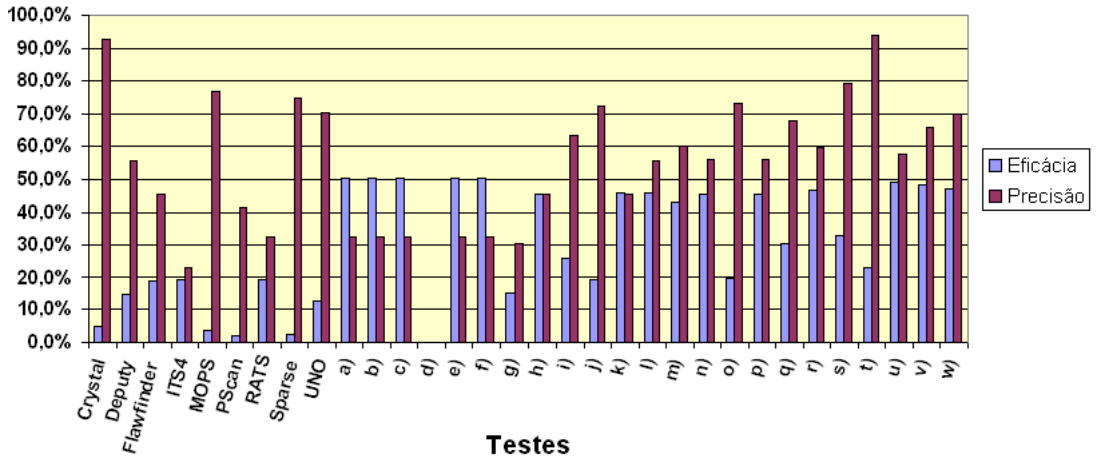


Figura 4.2: Eficácia e precisão do Mute sobre os testes realizados.

teste a) corresponde simplesmente à junção de vulnerabilidades em comum com outras ferramentas, onde temos uma alta eficácia, mas baixa precisão. No teste b) e c) também era de esperar o resultado porque todas as ferramentas têm a mesma confiança. No teste d) o limiar era de 0.7 e o máximo de confiança a obter era de 0.5, logo todas as vulnerabilidades foram rejeitadas. Os resultados do teste e) e f) também eram esperados por causa do valor de confiança depositado em cada ferramenta de 0.5. Os resultados do teste g) são explicados porque existem certas vulnerabilidades que não são detectadas por todas as ferramentas, daí ser impossível em alguns casos obter um valor superior ao limiar de 0.7. A partir do teste h) esperava-se que os resultados iriam melhorar, mas não se sabia em quanto. Como já foi dito, os melhores resultados foram no último teste, onde se detectou praticamente todas as vulnerabilidades e se teve uma precisão elevada.

Comparando com os resultados das ferramentas individualmente, o Mute é claramente melhor, quer em precisão como em eficácia. Não é possível comparar, cada métrica separadamente, uma vez que se torna a comparação desigual. A única ferramenta que ultrapassou claramente o Mute em precisão foi o Crystal, no entanto, isto é conseguido com baixa eficácia.

As duas técnicas que foram utilizadas para diminuir o número de falsos positivos são diferentes em vários pontos, sendo uma mais eficaz que outra. Ou seja, demonstrou-se ser mais interessante atribuir uma confiança para cada tipo

de vulnerabilidade em cada ferramenta, ao em vez de se ter uma confiança global por ferramenta. Esta solução tem como consequência que ferramentas com baixa confiança não conseguem por si só acentuar vulnerabilidades, necessitando que outras ferramentas também as encontrem. Além disso, há ferramentas que para um tipo de vulnerabilidades têm maior precisão que outras.

Em resumo, o Mute no máximo pode detectar 133 vulnerabilidades, uma vez que se encontra dependente do trabalho realizado pelas ferramentas de análise estática. Pode-se observar que com base nas duas métricas (eficácia e precisão) os testes u), v) e w) deram os melhores resultados, conseguindo obter no máximo uma eficácia de 50% (que equivale aproximadamente às 133 vulnerabilidades) e uma alta precisão, onde se tem menos de 75% de falsos positivos. Com estes valores é de considerar que o Mute é bastante eficaz no seu funcionamento.

Capítulo 5

Conclusões

Ano após ano, o número de vulnerabilidades que tem sido descobertas nas mais variadas aplicações tem vindo a aumentar. É possível apontar diversas causas entre elas a complexidade dos programas, falta de formação ou cuidado dos programadores, e interfaces mal especificadas.

Actualmente os compiladores têm diversas opções que ajudam a encontrar algumas falhas de segurança, mas as suas capacidades são insuficientes para resolver todos os tipos de erros.

Os analisadores estáticos são mais eficazes porque conseguem detectar um maior conjunto de classes de vulnerabilidades, dado que, o fazem com relativa rapidez e eficácia. As ferramentas de análise estática são úteis para os programadores porque possibilitam a correcção de um número substancial de vulnerabilidades antes do código entrar em testes ou ser disponibilizado para os utilizadores.

Uma ferramenta de análise estática detecta determinadas vulnerabilidades, daí pensou-se em criar uma ferramenta capaz de detectar diversos tipos de vulnerabilidades, e que ao mesmo tempo obtivesse um baixo número de falsos positivos — assim foi criado o Mute. O Mute é uma ferramenta capaz de detectar diferentes vulnerabilidades, usando os resultados de várias ferramentas, que são agregados para se produzir o seu relatório.

Antes de construir o Mute, analisou-se diversas ferramentas, sendo a sua avaliação feita por intermédio de uma versão preliminar de teste padronizado que contém código vulnerável. O teste padronizado tem um equilíbrio para cada tipo

de vulnerabilidade, ou seja, foram adicionadas vulnerabilidades e código não vulnerável para o mesmo tipo, e com algumas vulnerabilidades camufladas. Isto serve para perceber se o analisador estático consegue detectar as vulnerabilidades e compreender as suas dificuldades.

Depois de estudar e avaliar as capacidades das ferramentas, procedeu-se à construção do Mute. No Mute foram consideradas duas técnicas de análise/filtragem de vulnerabilidades, uma onde é depositada a confiança de cada ferramenta, e outra que consiste em atribuir uma confiança para cada tipo de vulnerabilidade de cada ferramenta. A avaliação do Mute demonstrou a sua utilidade tendo ele conseguido localizar mais vulnerabilidades que as ferramentas individualmente, mantendo os falsos positivos com valor reduzido.

Se se desenvolvesse uma ferramenta semelhante ao Mute para comercialização será que tinha sucesso? Na opinião do autor, sim, porque com um esforço relativamente reduzido um programador consegue encontrar vulnerabilidades de segurança, e com um baixo valor de falsos alertas, maximizando a qualidade e a produtividade.

5.1 Trabalho Futuro

A análise estática de código fonte ainda tem muito para progredir, e o Mute deu um passo na direcção de se desenvolver uma ferramenta que consegue encontrar vulnerabilidades. O trabalho apresentado pode servir para diversos fins e trazer novas ideias.

É possível criar novos métodos de avaliação (ou de filtragem) das vulnerabilidades encontradas por diversas ferramentas. Os métodos utilizados foram simples, baseados numa média e soma, mas agora é possível concretizar métodos de filtragem ainda mais sofisticados.

Um dos trabalhos a realizar seria melhorar o teste padronizado, ou seja, dividir o ficheiro de vulnerabilidades em múltiplos ficheiros, onde teríamos um ficheiro para cada classe de vulnerabilidades, para assim adicionar novos testes. Ainda sobre o teste, seria progredi-lo para a propriedade de escalabilidade e a da portabilidade para outros ambientes de execução. Também é de todo o interesse

expandir as classes de vulnerabilidades com testes para as classes que ainda não foram trabalhadas nesta versão preliminar.

Futuramente, para melhorar a precisão da ferramenta Mute, será possível adicionar novas ferramentas de análise estática, para novas vulnerabilidades e vulnerabilidades em comum.

Acrónimos

API **A**pplication **P**rogramming **I**nterface

ARPANET **A**dvanced **R**esearch **P**rojects **A**gency **Net**work

AST **A**bstract **S**yntax **T**ree

BOON **B**uffer **O**verrun Detecti**on**

CCE **C**ommon **C**onfiguration **E**numeration

Cil **C** **I**ntermediate **L**anguage

CVE **C**ommon **V**ulnerabilities and **E**xposures

Flawfinder **Find** Secure **Flaw**s

GCC **G**NU **C** **C**ompiler

GNU **G**NU is **N**ot **U**nix

ITS4 **I**t's **t**he **s**oftware, Stupid

LASIGE **La**boratório de **S**istemas **I**nformáticos de **G**rande **E**scala

MOPS **Mo**del Checking **P**rograms for **S**ecurity Properties

Mute **Mul**tiple Tools

OCaml **O**bjective **Caml**

PRNG **P**seudo-**R**andom **N**umber **G**enerator

ACRÓNIMOS

PScan **Scan** for **P**rintf style functions

RAD **R**eturn **A**ddress **R**epository

RAD(2) **R**apid **A**pplication **D**evelopment

RATS **R**ough **A**uditing **T**ool for **S**ecurity

RMMM **R**isk **M**itigation **M**onotoring and **M**anagement

RTL **R**egister **T**ransfer **L**anguage

SParse **S**emantic **Parse**r

SPLINT **S**ecure **P**rogramming **Lint**

SSA **S**tatic **S**ingle **A**ssignment

TOCTOU **T**ime **o**f **C**hech, **T**ime **o**f **U**se

UNO **U**se of uninitialized variable, **N**il-pointer references and **O**ut-of-bounds array indexing

V&V **V**alidação **e** **V**erificação

Glossário

Adversário – Sujeito com intuito indevido perante um outro utilizador de um sistema (ver também *Agente Malicioso, Atacante e Intruso*).

Agente Malicioso – Indivíduo com objectivo de criar perturbações num ambiente distribuido ou a um utilizador conectado a uma rede global (Veríssimo & Rodrigues, 2001) (ver também *Atacante*).

Análise Dinâmica – Método para detecção de vulnerabilidades de segurança enquanto a aplicação se encontra em execução (ver também *Tempo de Execução*).

Análise Estática – Método para detecção de vulnerabilidades de segurança com base no código fonte de uma aplicação (ver também *Tempo de Compilação*).

Atacante – Sujeito que pretende penetrar numa aplicação e consecutivamente explora-lá com diversos intuitos (do inglês, *hacker*) (ver também *Agente Malicioso*).

Chaves de Sessão – Conceito utilizado na criptografia, e corresponde a um número aleatório a atribuir numa comunicação entre dois ou mais interlocutores com vista a protege-la (Veríssimo & Rodrigues, 2001).

Classes de Vulnerabilidades – Constituido por um conjunto de tipos distintos de vulnerabilidades, cada categorização corresponde a uma classe de vulnerabilidades (ver também *Vulnerabilidade*).

Código Fonte – Conjunto de linhas formadas com base numa linguagem de programação, e ao qual se encontra associado a uma aplicação.

GLOSSÁRIO

Falso Alerta – ver *Falsos Positivos*.

Falsos Negativos – São trechos de código identificados como inofensivos pelas ferramentas, mas na realidade são vulnerabilidades presentes no código fonte (Younan *et al.*, 2004).

Falsos Positivos – São trechos de código apontados pelas ferramentas como sendo potencialmente vulneráveis e que na realidade não o são (Younan *et al.*, 2004).

Intruso Indivíduo que após um ataque a uma aplicação, explora um sistema informático possivelmente através de uma vulnerabilidade numa aplicação (Veríssimo & Rodrigues, 2001).

Programação Defensiva – Durante a concretização de uma aplicação, o programador não confia em nenhum parâmetro — daí validar todos os parâmetros.

Seed – No português semente; este conceito está associado aos algoritmos de geração de números aleatórios. Um *seed* é utilizado como um elemento de cálculo no produto resultante do algoritmo.

Tempo de Compilação – Identifica que uma ferramenta é concebida para analisar o código fonte, em conjunto com um compilador ou separadamente para prevenir e detectar vulnerabilidades (Younan *et al.*, 2004).

Tempo de Execução – Identifica que uma ferramenta detecta vulnerabilidades, enquanto a aplicação se encontra em execução (Younan *et al.*, 2004).

Teste Padronizado – Conjunto de informação sobre uma dada temática, com o objectivo de avaliar de forma equitativa um produto ou serviço (Vieira, 2005).

Verdadeiros Positivos – São trechos de código indicados pelas ferramentas como sendo potencialmente vulneráveis e que realmente o são (Younan *et al.*, 2004).

Vulnerabilidade – Ponto fraco num sistema computacional ou de comunicação que pode ser explorado com intenção maliciosa (Veríssimo & Rodrigues, 2001).

Bibliografia

Aho, A.V., Lam, M.S., Sethi, R. & Ullman, J.D. (1996). *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2nd edn.

Alessandri, D. (2004). *Attack Class Based Analysis of Intrusion Detection Systems*. Ph.D. thesis, University of Newcastle, School of Computing Science.

Alexander, S. (2005). Defeating compiler-level buffer overflow protection. *j-LOGIN*, **30**.

Antunes, J. (2006). *Vulnerability Assessement Through Attack Injection*. M.Sc. thesis, Faculdade de Ciências da Universidade de Lisboa.

ArKin, B., Stender, S. & McGraw, G. (2005). Software penetration testing. *IEEE Security & Privacy*, **3**, 84–87.

Ashcraft, K. & Engler, D. (2002). Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 143–159, IEEE Computer Society.

Austin, T.M., Breach, S.E. & Sohi, G.S. (1994). Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 290–301, ACM.

Azul, A.A. (1998). *Técnicas e Linguagens de Programação*. Porto Editora.

Basirico, J. (2004). Hacker report. `http://www.securityinnovation.com/pdf/si-report-static-analysis.pdf`.

BIBLIOGRAFIA

Bhatkar, S., DuVarney, D.C. & Sekar, R. (2003). Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 105–120, USENIX Association.

Binstock, A. & Streng, G. (2006). Extra-strength code cleaners. `http://infoworld.com/pdf/special_report/2006/05SRcode.pdf`.

Braude, E.J. (2001). *Software Engineering: An Object-Oriented Perspective*. John Wiley & Sons, Inc.

Chen, H., Dean, D. & Wagner, D. (2004). Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 171–185.

Chess, B. & McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, **2**, 76–78.

Chiueh, T.C. & Hsu, F.H. (2001). Rad: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, 409–420, IEEE Computer Society.

Condit, J., Harren, M., Anderson, Z., Gay, D. & Necula, G. (2006). Dependent types for low-level programming. Tech. Rep. UCB/EECS-2006-129, University of California at Berkeley.

Condit, J., Harren, M., Anderson, Z., Gray, D. & Necula, G. (2007). Dependent types for low-level programming. In *Proceeding of the 16th European Symposium on Programming*.

Correia, M.P. (2006). Segurança de software: Access control. `http://osc.di.fc.ul.pt/ses/slides/08%20Trust%20and%20input%20validation.pdf`.

Coverity, I. (2006). Static source code analysis for c and c++. `http://coverity.com/html/library.php`.

Cowan, C., Beattie, S., Day, R., Pu, C., Wagle, P. & Walthinsen, E. (1999). Protecting systems from stack smashing attacks with stackguard. In *Proceedings of The Linux Expo*, 18–22.

Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M. & Lokier, J. (2001). Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, 191–200, USENIX Association.

Cowan, C., Beattie, S., Johansen, J. & Wagle, P. (2003). Point-guard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, 91–104, USENIX Association.

DeKok, A. (2007). Pscan: A limited problem scanner for c source files. `http://www.striker.ottawa.on.ca/~aland/pscan/`.

Dix, A., Finlay, J., Abowd, G.D. & Beale, R. (2004). *Human Computer Interaction*. Person Prentice Hall, 3rd edn.

Foster, J., Fähndrich, M. & Aiken, A. (1999). A theory of type qualifiers. In *Proceedings of the Conference on Programming Language Design and Implementation*, 192–203, ACM Press.

Grégio, A.R.A., Duarte, L.O., Barbato, L.G.C. & Montes, A. (2005). Codificação segura: Abordagens práticas. `http://www.linorg.cirp.usp.br/SSI/SSI2005/Microcursos/MC01.pdf`.

Haffner, E.G., Engel, T. & Meinel, C. (1998). The flood-gate principle - a hybrid approach to a high security solution. In *Proceedings of the International Conference on Information Security and Cryptology*, 147–160.

Hastings, R. & Joyce, B. (1992). Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 125–138, USENIX Association.

Haugh, E. & Bishop, M. (2003). Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the 10th Network and Distributed System Security Symposium*, 39–48, Internet Society.

BIBLIOGRAFIA

Heffley, J. & Meunier, P. (2004). Can source code auditing software identify common vulnerabilities and be use to evaluate software security? In *Proceedings of the 37th Hawaii International Conference on System Sciences*, 196–207.

Holzmann, G. (2007). Uno software. `http://spinroot.com/uno/`.

Holzmann, G.J. (2002). Static source code checking for user-defined properties. In *Proceeding of the 6th World Conference on Integrated Design and Process Technology*, 26–30.

Horwitz, S. (1997). Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems*, **19**, 1–6.

Kernighan, B.W. & Ritchie, D.M. (1988). *The C Programming Language*. Prentice Hall, Inc, 2nd edn.

Kit, E. (1995). *Software Testing In The Real World: Improving The Process*. Addison-Wesley, 1st edn.

Koziol, J., Litchfield, D., Aitel, D., Anley, C., Eren, S., Mehta, N. & Hassell, R. (2004). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons.

Kruegel, C. (2004). Secure software programming and vulnerability analysis. `http://thor.auto.tuwien.ac.at/~chris/teaching/secprog.html`.

Langlois, T. (2003). Compiladores. `http://ctp.di.fc.ul.pt/teccomp`.

Larochelle, D. & Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 177–190, USENIX Association.

Larus, J.R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S.K. & Venkatapathy, R. (2004). Righting software. *IEEE Software*, **21**, 92–100.

Lavenhar, S. (2006). Code analysis. `http://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code/214.html`.

MERRILL, J. (2003). Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, 171–180.

NOVILLO, D. (2003). Tree ssa – a new optimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers Summit*, 181–195.

OLIVEIRA, J.P. (2006). Diagnóstico e correcção de problemas. `http://gsd.di.uminho.pt/members/jpo/2006-2007/SO_MIECom/`.

ONE, A. (1996). Smashing the stack for fun and profit. *Phrack*, **7**.

ORLOVICH, M. & RUGINA, R. (2006). Memory leak analysis by contradiction. In *Proceedings of the 13th International Static Analysis Symposium*, 405–424, Springer.

PATTERSON, J.R.C. (1995). Accurate static branch prediction by value range propagation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 67–78, ACM Press.

PRATIKAKIS, P., FOSTERY, J. & HICKS, M. (2006). Locksmith: Context-sensitive correlation analysis for race detection. Tech. Rep. CS-TR-4789, University of Maryland.

PUZO, J.E. (1986). Gnu's bulletin. `http://www.gnu.org/bulletins/bull1.txt`.

RAMALINGAM, G. (1994). The undecidability of aliasing. In *ACM Transactions on Programming Languages and Systems*, 1467–1471.

ROBERTS, L. (1986). The arpanet and computer networks. In *Proceedings of the ACM Conference on The History of Personal Workstations*, 51–58, ACM Press.

SCHWARZ, B., CHEN, H., WAGNER, D., MORRISON, G., WEST, J., LIN, J. & TU, W. (2005). Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 13–22.

BIBLIOGRAFIA

Secure Software, I. (2007). Rats – rough auditing tool for security. `http://www.fortifysoftware.com/security-resources/rats.jsp`.

Shankar, U., Talwar, K., Foster, J.S. & Wagner, D. (2001). Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 201–220, USENIX Association.

Sotirov, A.I. (2005). *Automatic Vulnerability Detection Using Static Source Code Analysis*. Ph.D. thesis, University of Alabama.

Stallman, R.M. & Community, G.D. (2005). Using the gnu compiler collection. `http://gcc.gnu.org/onlinedocs/gcc-4.2.1/gcc.pdf`.

Stallman, R.M., McGrath, R., Oram, A. & Drepper, U. (2006). The gnu c library. `http://www.gnu.org/software/libc/manual/pdf/libc.pdf`.

Veríssimo, P. & Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers.

Viega, J. & McGraw, G. (2006). *Building Secure Software: How to Avoid Security Problems the Righ Way*. Addison-Wesley.

Viega, J., Bloch, J.T., Kohno, T. & McGraw, G. (2000). Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference*, 257–269, IEEE Computer Society.

Viega, J., Bloch, J.T., Kohno, T. & McGraw, G. (2002). Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*, **5**, 238–261.

Vieira, M.P.A. (2005). *Testes Padronizados de Confiabilidade para Sistemas Transaccionais*. Ph.D. thesis, Universidade de Coimbra.

Wagner, D., Foster, J., Brewer, E. & Aiken, A. (2000). A first step towards automated detection of buffer overflow vulnerabilities. In *Proceedings of the 7th Networking and Distributed System Security Symposium*, 3–17.

Wheeler, D.A. (2007). Flawfinder. `http://www.dwheeler.com/flawfinder/`.

WHITTAKER, J.A. (2002). *How to Break Software Security: A Practical Guide to Testing*. Addison-Wesley, 1st edn.

WHITTAKER, J.A. & THOMPSON, H.H. (2003). *How to Break Software Security: Effective Techniques for Security Testing*. Addison-Wesley, 1st edn.

YOUNAN, Y. (2003). *An overview of common programming security vulnerabilities and possible solutions*. M.Sc. thesis, Vrije Universiteit Brussel.

YOUNAN, Y., JOOSEN, W. & PIESSENS, F. (2004). Code injection in c and c++: A survey of vulnerabilities and countermeasures. Tech. Rep. CW 386, Katholieke Universiteit Leuven from Department of Computer Science.

Apêndice A

Análise de Riscos

Muitos dos projectos correm o risco de não atingirem os objectivos delineados, uma vez que dependem dos recursos disponíveis para o trabalho e do tempo estabelecido. Aqui, o objectivo não é fugir dos riscos, mas antes reduzir a probabilidade de estes ocorrerem e saber controlá-los (Braude, 2001).

Uma prévia identificação e análise destes riscos, permite uma melhor preparação e possível resolução dos vários problemas, assim, quantos mais riscos forem identificados, mais fácil se previne e minimiza o impacto, dado que é só seguir o plano definido.

Para lidar com os riscos utilizam-se métodos de apoio à resolução de problemas. A estratégia é empregar um **Plano RMMM** — plano de contingência que descreve as acções a tomar se o risco for real. Este método de resolução, consiste em associar a cada risco um Plano de *Mitigação*, *Monitorização* e *Gestão*.

Mitigação Tentativa de evitar o aparecimento do risco, que está em concordância com o conceito de estratégia proactiva, através de acções de prevenção adequadas.

Monitorização Faz a verificação e registo de estados/valores do risco, identificando tendências.

Gestão Devido ao risco ser uma realidade bem plausível, que medidas deverão ser tomadas?

De seguida, são apresentados os riscos mais importantes a ter em conta e as formas de os prevenir. As formas de prevenir os riscos são recomendações a ter em conta durante o desenvolvimento do projecto, levando assim a uma concretização optimizada e segura.

I - Análise de requisitos incompleta ou ambígua

Mitigação

- Elaborar documentos de análise tão exaustivos e completos quanto possível, com identificação das interligações entre diferentes requisitos;
- Para cada requisito identificar e anotar as correspondentes referências;
- Se necessário criar diagramas de fluxo de dados, para melhor compreender os requisitos.

Monitorização

- Em cada fase rever os requisitos especificados, assim como as técnicas utilizadas;
- Nas reuniões de coordenação, descrever os requisitos para identificar possíveis incoerências.

Gestão

- Ler as especificações em comparação com as anotações escritas e documentos lidos, e corrigir problemas identificados;
- Rever todo o planeamento atribuído ao projecto e verificar se não foi afectado;
- Verificar se o processo de desenvolvimento escolhido é realmente o melhor.

II - Enquadramento sócio-cultural na organização

Mitigação

- Comunicar com as diversas pessoas existentes no posto de trabalho;
- Participar nas actividades colectivas com o grupo e instituição;
- Compreender a responsabilidade do quanto é representar a instituição.

Monitorização

- Acompanhar o trabalho de todos os membros do posto de trabalho e motivar;
- Verificar todo o tipo de informação (correio electrónico, placares) para um melhor enquadramento.

Gestão

- Criar iniciativas de grupo e contactar mais com a instituição;
- Colaborar na organização das diversas actividades criadas pela instituição.

III - Mudança do posto de trabalho

Mitigação

- Manter todo o trabalho realizado até ao momento organizado e arquivado;
- Efectuar uma rápida mudança no posto de trabalho;
- Obter as mesmas condições ou melhores com a ajuda dos responsáveis.

Monitorização

- Ocasionalmente perguntar quando se irá realizar a mudança;

- Manter todo o espaço de trabalho arrumado;

- Verificar se o espaço de trabalho se encontra adequado.

Gestão

- Pedir ajuda de colegas para facilitar o processo;

- Estabelecer regras de coordenação e orientação;

- Informar os responsáveis pela mudança e auxiliar na logística.

IV - Impossibilidade de acesso a documentação

Mitigação

- Procurar correctamente nas fontes mais fiáveis;

- Contactar responsáveis pelas ferramentas para que se possa obter referências ou mesmo documentos privados;

- Manter o coordenador do projecto informado da documentação acedida.

Monitorização

- Verificar se realmente as fontes utilizadas são boas, caso contrário procurar outras;

- Procurar em documentação relacionada, se existe alguma possibilidade de haver um apontador útil;

- Verificar se os responsáveis pela ferramenta receberam o pedido de ajuda;

- Procurar informação sobre os documentos a procurar.

Gestão

- Se algum momento não se conseguir uma referência mais extensível, retirar notas pela documentação que se vai lendo;
- Pedir ajuda a membros da instituição mais experientes na área;
- Pedir auxílio ao orientador do projecto.

V - Processo de desenvolvimento inadequado

Mitigação

- Verificar se os conceitos do processo de desenvolvimento aprendidos estão correctos;
- Analisar cada etapa em sintonia com o planeamento antes de avançar;
- Avaliar semana após semana o trabalho desenvolvido;
- Manter todas as tarefas bem coordenadas e terminadas a tempo certo.

Monitorização

- Verificar se todos os prazos e metodologias estão a ser cumpridas;
- Manter diálogo permanente com o coordenador do projecto, como meio de supervisão.

Gestão

- Rever as várias etapas do projecto e ajustar o planeamento;
- Equacionar possíveis processos de desenvolvimento, somente durante a fase de análise ou desenho.

VI - Dificuldade na instalação das ferramentas

Mitigação

- Obter manuais para auxiliar a instalação;
- Verificar se a instalação foi bem concretizada;
- Compreender os requisitos e limitações necessárias para a sua instalação;
- Recorrer aos autores para uma possível ajuda.

Monitorização

- Notar se a versão é a mais recente;
- Obter todos os passos necessários para que a instalação corra plenamente;
- Analisar os relatórios de instalação.

Gestão

- Rever com atenção os manuais pois pode estar a solução;
- Seleccionar outras ferramentas com a mesma técnica e tipos de vulnerabilidades;
- Verificar se alguém conseguiu trabalhar com a ferramenta e pedir ajuda;
- Procurar mais informação sobre problemas já ocorridos na ferramenta.

VII - Elaborar do relatório preliminar e final do trabalho

Mitigação

- Recorrer a metodologias de tratamento documental;

- Obter exemplos de outros projectos para reajustar ou aprender com novas necessidades;

- Sempre que possível escrever alguma coisa.

Monitorização

- Verificar os prazos de entrega semana a pós semana;

- Ler o documento escrito para rever possíveis erros;

- O coordenador do projecto será informado com regularidade da evolução.

Gestão

- Ajustar o planeamento às necessidades reais;

- Pedir auxílio do orientador do projecto;

- Contactar com ex-alunos para contornar o problema.

VIII - Impossibilidade de executar todos os testes planeados

Mitigação

- A nível do planeamento ponderar sobre a execução dos testes, não descurando da atribuição de tempo;

- Sempre que possível executar o máximo de testes durante a fase de concretização, de forma a analisar todas as estruturas criadas com o devido rigor e aplicar medidas de correcção caso se verifiquem erros;

- Realizar testes intermédios por tarefas antes de realizar os testes finais na ferramenta;

- Especificar o tipo de testes a realizar e com que objectivos.

Monitorização

- Nas reuniões mostrar que o planeado está a ser cumprido;
- Anotar todos os testes já realizados e rever os que faltam;
- Atribuir tempo a todos os testes, de modo que se realizem.

Gestão

- Selecção e execução apenas dos testes considerados imprescindíveis para o funcionamento geral da ferramenta e/ou os necessários para avaliação geral;
- Recrutamento de elementos que estejam disponíveis para ajudar nesta tarefa;
- Discutir com o orientador a possibilidade de angariar mais tempo para testes.

IX - Atraso na concretização do projecto

Mitigação

- Perceber que tipo de complexidade o projecto aborda;
- Concepção de métodos para compensar possíveis atrasos;
- Cumprir sempre os prazos estipulados no planeamento.

Monitorização

- Analisar as fases anteriores antes de passar à próxima;
- Planear as seguintes fases com maior coerência e precisão;
- Mostrar ao coordenador do projecto que tudo está a correr como planeado.

Gestão

- Analisar atrasos de projectos já existentes e obter conclusões para um melhor planeamento;
- Sobrecarregar tarefas caso necessário;
- Recorrer ao orientador do projecto.

X - Falta ou avaria dos recursos

Mitigação

- Planear que recursos são necessários para o desenrolar do projecto;
- Se possível duplicar o material que é realmente inseguro e prejudicial para o projecto.

Monitorização

- Periodicamente avaliar o material e quais os cuidados necessários com ele.

Gestão

- Rápida aquisição e/ou troca do material necessário;
- Contactar com o coordenador do projecto para angariar fundos, em possíveis avarias.

Os riscos são um elemento que pode influenciar bastante a qualidade final do trabalho, como tal estabeleceu-se um plano de riscos para o caso de aparecerem problemas inesperados.

Para além dos riscos apresentados, podem surgir outros durante o desenvolvimento do projecto, e uma vez que não foram criados planos para minimizar o seu impacto, há sempre a possibilidade de pedir aconselhamento técnico ao orientador e coordenador do projecto, que terá directivas para contornar o problema.

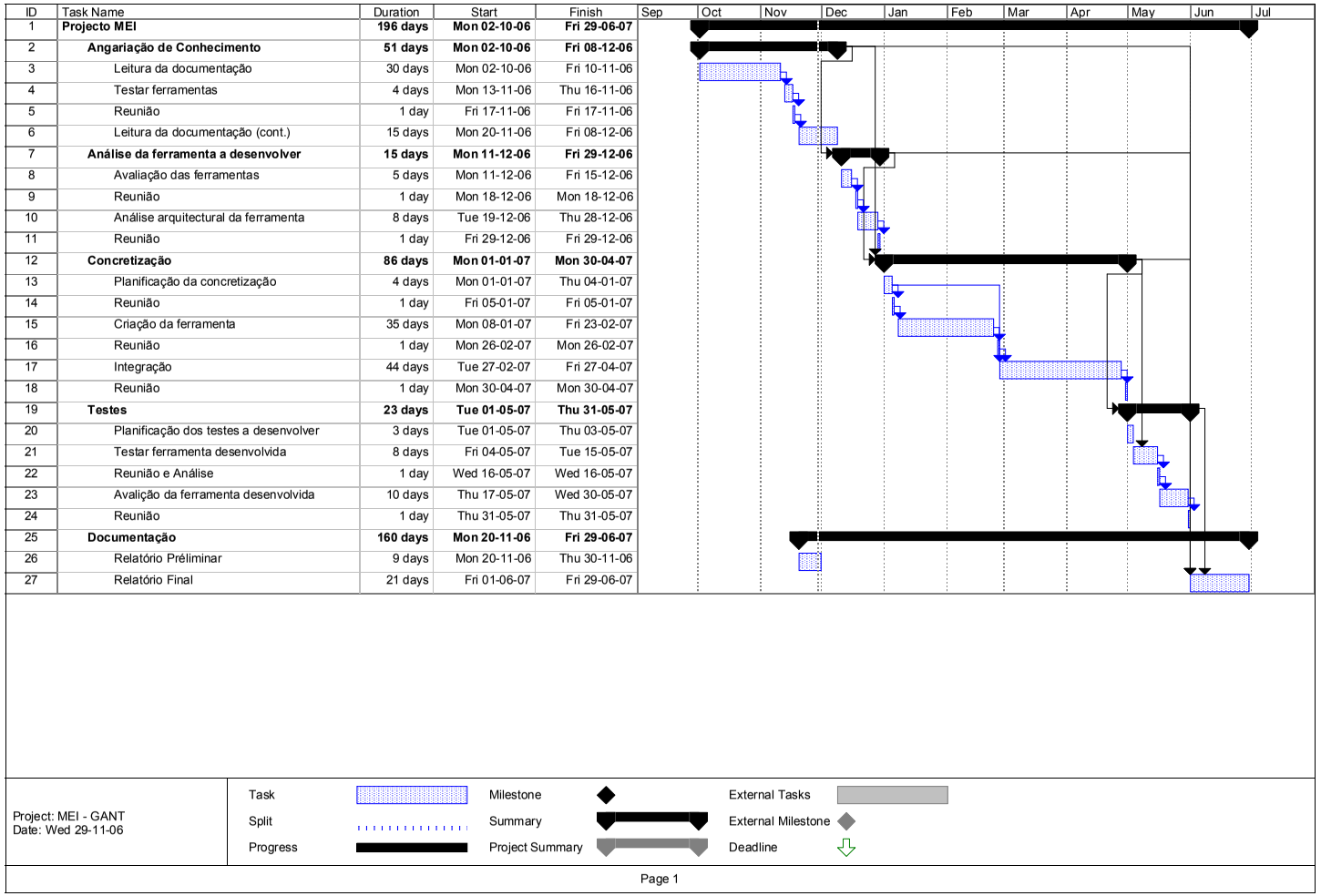
Apêndice B

Mapas de Gant

Todos os projectos têm uma calendarização — uma especificação temporal de cada fase para o projecto. Assim, este apêndice apresenta um mapa de Gant onde se espelha toda a sequência do trabalho realizado, assim como o modelo de desenvolvimento adoptado.

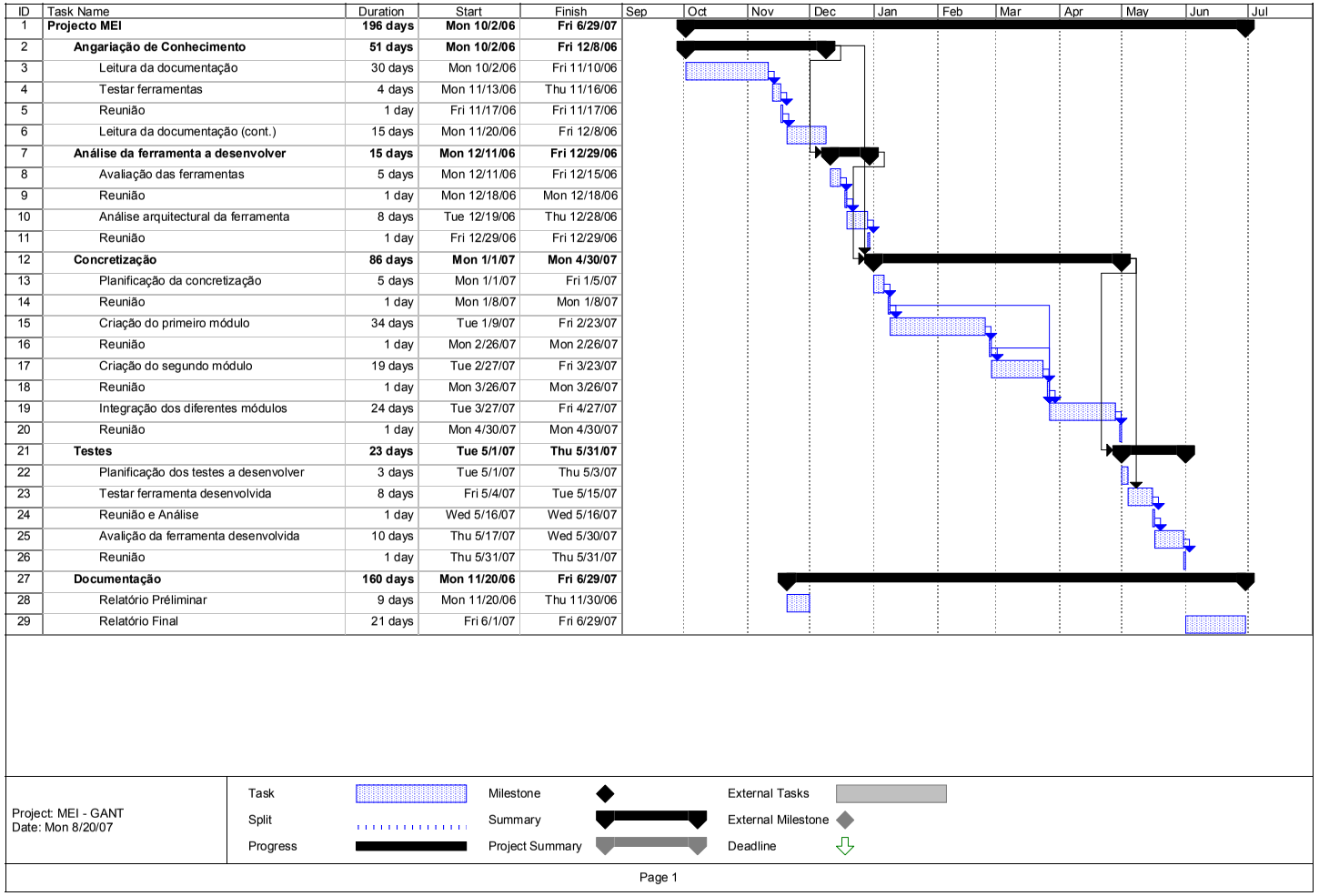
Um mapa de Gant é um guia para toda a equipa ou elemento quanto ao trabalho que se tem de realizar, e onde se encontram datas cruciais como reuniões e entregas. Desta forma, é apresentado um mapa que ilustra toda a actividade ao longo do desenvolvimento do projecto de investigação. O mapa encontra-se organizado por meses (colunas) e suas etapas (linhas). Todo o planeamento pode divergir um pouco com o previamente estabelecido, mas este somente foi alterado uma vez, para melhor clareza e precisão temporal.

Portanto, é possível observar dois mapas de Gant nas páginas seguintes, que mostra como foi planeado e calendarizado todo o projecto no seu estado inicial e final.



ID	Task Name	Duration	Start	Finish
1	**Projecto MEI**	**196 days**	**Mon 02-10-06**	**Fri 29-06-07**
2	**Angariação de Conhecimento**	**51 days**	**Mon 02-10-06**	**Fri 08-12-06**
3	Leitura da documentação	30 days	Mon 02-10-06	Fri 10-11-06
4	Testar ferramentas	4 days	Mon 13-11-06	Thu 16-11-06
5	Reunião	1 day	Fri 17-11-06	Fri 17-11-06
6	Leitura da documentação (cont.)	15 days	Mon 20-11-06	Fri 08-12-06
7	**Análise da ferramenta a desenvolver**	**15 days**	**Mon 11-12-06**	**Fri 29-12-06**
8	Avaliação das ferramentas	5 days	Mon 11-12-06	Fri 15-12-06
9	Reunião	1 day	Mon 18-12-06	Mon 18-12-06
10	Análise arquitectural da ferramenta	8 days	Tue 19-12-06	Thu 28-12-06
11	Reunião	1 day	Fri 29-12-06	Fri 29-12-06
12	**Concretização**	**86 days**	**Mon 01-01-07**	**Mon 30-04-07**
13	Planificação da concretização	4 days	Mon 01-01-07	Thu 04-01-07
14	Reunião	1 day	Fri 05-01-07	Fri 05-01-07
15	Criação da ferramenta	35 days	Mon 08-01-07	Fri 23-02-07
16	Reunião	1 day	Mon 26-02-07	Mon 26-02-07
17	Integração	44 days	Tue 27-02-07	Fri 27-04-07
18	Reunião	1 day	Mon 30-04-07	Mon 30-04-07
19	**Testes**	**23 days**	**Tue 01-05-07**	**Thu 31-05-07**
20	Planificação dos testes a desenvolver	3 days	Tue 01-05-07	Thu 03-05-07
21	Testar ferramenta desenvolvida	8 days	Fri 04-05-07	Tue 15-05-07
22	Reunião e Análise	1 day	Wed 16-05-07	Wed 16-05-07
23	Avalição da ferramenta desenvolvida	10 days	Thu 17-05-07	Wed 30-05-07
24	Reunião	1 day	Thu 31-05-07	Thu 31-05-07
25	**Documentação**	**160 days**	**Mon 20-11-06**	**Fri 29-06-07**
26	Relatório Préliminar	9 days	Mon 20-11-06	Thu 30-11-06
27	Relatório Final	21 days	Fri 01-06-07	Fri 29-06-07

Figura B.1: Mapa de Gant do início do projecto.



ID	Task Name	Duration	Start	Finish
1	**Projecto MEI**	**196 days**	**Mon 10/2/06**	**Fri 6/29/07**
2	**Angariação de Conhecimento**	**51 days**	**Mon 10/2/06**	**Fri 12/8/06**
3	Leitura da documentação	30 days	Mon 10/2/06	Fri 11/10/06
4	Testar ferramentas	4 days	Mon 11/13/06	Thu 11/16/06
5	Reunião	1 day	Fri 11/17/06	Fri 11/17/06
6	Leitura da documentação (cont.)	15 days	Mon 11/20/06	Fri 12/8/06
7	**Análise da ferramenta a desenvolver**	**15 days**	**Mon 12/11/06**	**Fri 12/29/06**
8	Avaliação das ferramentas	5 days	Mon 12/11/06	Fri 12/15/06
9	Reunião	1 day	Mon 12/18/06	Mon 12/18/06
10	Análise arquitectural da ferramenta	8 days	Tue 12/19/06	Thu 12/28/06
11	Reunião	1 day	Fri 12/29/06	Fri 12/29/06
12	**Concretização**	**86 days**	**Mon 1/1/07**	**Mon 4/30/07**
13	Planificação da concretização	5 days	Mon 1/1/07	Fri 1/5/07
14	Reunião	1 day	Mon 1/8/07	Mon 1/8/07
15	Criação do primeiro módulo	34 days	Tue 1/9/07	Fri 2/23/07
16	Reunião	1 day	Mon 2/26/07	Mon 2/26/07
17	Criação do segundo módulo	19 days	Tue 2/27/07	Fri 3/23/07
18	Reunião	1 day	Mon 3/26/07	Mon 3/26/07
19	Integração dos diferentes módulos	24 days	Tue 3/27/07	Fri 4/27/07
20	Reunião	1 day	Mon 4/30/07	Mon 4/30/07
21	**Testes**	**23 days**	**Tue 5/1/07**	**Thu 5/31/07**
22	Planificação dos testes a desenvolver	3 days	Tue 5/1/07	Thu 5/3/07
23	Testar ferramenta desenvolvida	8 days	Fri 5/4/07	Tue 5/15/07
24	Reunião e Análise	1 day	Wed 5/16/07	Wed 5/16/07
25	Avalição da ferramenta desenvolvida	10 days	Thu 5/17/07	Wed 5/30/07
26	Reunião	1 day	Thu 5/31/07	Thu 5/31/07
27	**Documentação**	**160 days**	**Mon 11/20/06**	**Fri 6/29/07**
28	Relatório Préliminar	9 days	Mon 11/20/06	Thu 11/30/06
29	Relatório Final	21 days	Fri 6/1/07	Fri 6/29/07

Figura B.2: Mapa de Gant com o projecto concluído.

Apêndice C

Funções Vulneráveis

Considera-se uma função vulnerável[1] quando o seu uso possibilita facilmente o aparecimento de erros de programação que originam ataques ao sistema, violando assim a sua especificação. Por exemplo, deixar o sistema inoperacional ou possibilidade de alterar informação confidencial e privada. Alguns ataques são explorados para causar erros, no entanto, alguns não têm consequências maiores.

Há que prevenir o quanto possível ataques às funções reportadas neste apêndice. Um ataque a uma dada função pode ser fácil de concretizar, mas há outros que requerem técnicas sofisticadas de intrusão. Todavia, os ataques encontram-se limitados à forma como a aplicação é codificada.

[1] As funções vulneráveis apresentadas neste apêndice podem ser encontradas em, `http://buildsecurityin.us-cert.gov` (visitado em Setembro de 2007).

Tabela C.1: Funções potencialmente vulneráveis.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
bcopy	Memória sobrelotada	Falha em garantir que o carácter de terminação não se encontra presente no destino. Uma solução que melhora substancialmente o problema é utilizar o *memcpy()*.
fscanf	Memória sobrelotada	É preciso garantir que só se lê o número de dados pretendidos, para não sobrescrever na memória. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).
getcwd	Memória sobrelotada	O apontador da cadeia de caracteres não pode ser NULL, caso contrário tem comportamentos diversos.
getopt	Memória sobrelotada	Problema quando o número e tamanho dos argumentos é maior do que esperado, para tal basta verificar o número e tamanho dos argumentos recebidos.
getpass	Memória sobrelotada	O problema é baseado na falta do carácter de terminação ("\0"). Por isso, a solução é garantir que a cadeia de caracteres tem o carácter de terminação.
gets	Memória sobrelotada	Utilizar a função *fgets()*, dado que tem o valor máximo a ler de dados. Uma má utilização provoca memória sobrelotada.
		Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
getwd	Memória sobrelotada	A cadeia de caracteres deve ser maior que o tamanho da directoria corrente mais um. A solução possível é utilizar *getcwd()*, obriga ao utilizador a especificar o tamanho.
realpath	Memória sobrelotada	O valor introduzido no primeiro argumento pode indicar um caminho muito grande causando memória sobrelotada no segundo argumento, devido a este ter um espaço de memória reservado muito pequeno. A solução é reservar o número máximo de bytes que um caminho pode ter, assim o segundo argumento terá que ser igual a PATH_MAX mais um.
scanf	Memória sobrelotada	É necessário controlar o número de dados lidos, para não escrever por cima de outros dados existentes na memória. Uma solução é utilizar o *fgets()*. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).
sscanf	Memória sobrelotada	É preciso garantir que só lê o número de dados pretendidos, para não sobrescrever na memória. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
strcat	Memória sobrelotada	Utilizar a função *strncat()* ou *strlcat()*, porque contém o número de dados a concatenar com o destino. Uma má utilização provoca memória sobrelotada.
strcpy	Memória sobrelotada	Utilizar a função *strncpy()* ou *strlcpy()*, porque tem o valor máximo a copiar para o destino. Uma má utilização provoca memória sobrelotada.
streadd	Memória sobrelotada	Deve-se garantir que a cadeia de caracteres de destino é o suficiente para guardar todo o conteúdo a tratar da cadeia de caracteres origem.
strecpy	Memória sobrelotada	Deve-se garantir que a cadeia de caracteres de destino é o suficiente para guardar todo o conteúdo a tratar da cadeia de caracteres origem.
strtrns	Memória sobrelotada	Garantir que a cadeia de caracteres irá guardar o resultado da troca de caracteres antigos por novos tenha espaço suficiente para a permutação. A solução passa por reservar espaço de memória suficiente para guardar o resultado da função.
vfscanf	Memória sobrelotada	É preciso garantir que só se lê o número de dados pretendidos, para não sobrescrever na memória. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
vscanf	Memória sobrelotada	É preciso garantir que só se lê o número de dados pretendidos, para não sobrescrever na memória. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).
vsprinf	Memória sobrelotada	É preciso garantir que só se lê o número de dados pretendidos, para não sobrescrever na memória.
vsscanf	Memória sobrelotada	É preciso garantir que só se lê o número de dados pretendidos, para não sobrescrever na memória. Se mesmo assim pretender usar a função, então use os seguintes "*truques*", (...,"%20s",...) – onde vinte é o número de bytes que a função vai ler ou (...,"%as",&string).
access	Condição de disputa	Utilizar a função *lstat()*, de modo a verificar se o ficheiro não é um atalho (*symbolic link*).
acct	Condição de disputa	Caso o argumento da função seja passado como NULL, então irá desactivar todos os registos de contas. A solução é não permitir que um intruso possa indicar o caminho do ficheiro, e não pode ter controlo sobre a especificação de caminhos, em qualquer outro lugar no programa.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
chgrp	Condição de disputa	O problema associado à função é designado por TOCTOU, onde se verifica antes de usar e depois usa-se a função. O problema é o tempo entre verificar e usar, que um atacante pode modificar o ficheiro para um atalho, ao qual o atalho pode esta associado um programa malicioso. Uma solução é eliminar a verificação e utilizar o *fchgrp()*.
chmod	Condição de disputa	No instante que o programa se encontra a executar o *chmod()*, um atacante muda o ficheiro alvo para um ficheiro de atalho, ao qual o atalho pode ser direccionado para um programa malicioso. Uma solução passa por eliminar a validação ao ficheiro alvo e utilizar *flags* como S_IRUSR |S_IRGRP |S_IROTH. Se possível utilizar *fchmod()*.
chown	Condição de disputa	O problema associado à função é designado por TOCTOU, onde se verifica antes de usar e depois usa-se a função. O problema é o tempo entre verificar e usar, que um atacante pode modificar o ficheiro para um atalho, ao qual o atalho pode esta associado um programa malicioso. Uma solução é eliminar a verificação e utilizar o *fchown()*.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
mkdir	Condição de disputa	Tem como problema a vulnerabilidade conhecida por *Time of check, Time of use*. Uma possível solução é deixar de verificar antes de usar e depois utilizar algumas *flags* para tentar eliminar a vulnerabilidade, que são S_IRWXU |S_IRWXG |S_IROTH |S_IXOTH.
mktemp	Condição de disputa	O problema associado à função é o TOCTOU. A possível solução seria utilizar o *mkstemp()*.
rename	Condição de disputa	Tem associado a vulnerabilidade TOCTOU, onde o programador verifica se o ficheiro é um atalho, se não então usa a função. Entre o verificar e usar, um atacante pode modificar o ficheiro novo para um programa malicioso. Uma solução assenta em diminuir os privilégios antes do uso da função e depois se usa a função, após isto restaura-se os privilégios.
stat	Condição de disputa	A esta função está associada o problema TOCTOU. Uma solução, dado que varia consoante o caso é utilizar o *lstat()* antes de abrir um ficheiro e guardar a estrutura de dados, depois abrir o ficheiro com as *flags* O_CREAT e O_EXCL, de seguida utilizar o *fstat()* e guardar a estrutura, por fim compara-se três elementos da estrutura o st_mode, st_info e st_dev com a estrutura do fstat e lstat, se tudo for válido pode-se editar o ficheiro.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
symlink	Condição de disputa	Devido à vulnerabilidade TOCTOU, um atacante pode indicar outro ficheiro em vez do pretendido na realidade. A solução passa por utilizar o *lstat()*, guardar a estrutura de dados, depois abre-se o ficheiro com o *open()* e usa-se o *fstat()* para comparar se o ficheiro não foi modificado.
tempnam	Condição de disputa	O problema associado à função é o TOCTOU, como tal deve-se usar de forma muito cautelosa, de modo a um intruso não modificar o nome. De certa maneira deve-se garantir que o ficheiro gerado é o pretendido e está somente autorizado para uso do autor.
tmpfile	Condições de disputa	Utilizar a função *mkstemp()*, dado ser mais segura que *tmpfile()*, sendo menos vulnerável a condições de disputa.
tmpnam	Condição de disputa	O problema associado à função é o TOCTOU, como tal deve-se usar de forma muito cautelosa, de modo a um intruso não modificar o nome. De certa maneira deve-se garantir que o ficheiro gerado é o pretendido e está somente autorizado para uso do autor.
truncate	Condição de disputa	Vulnerabilidade de TOCTOU, verifica-se primeiro depois usa-se. A solução passa por não utilizar a função, e utilizar mecanismos de controlo de acesso que algumas funções possuem. Se for para trabalhar num ficheiro temporário então pode-se utilizar o *umask()* com as permissões 0066, usar o *fopen()* com flags propícias, escrever e ler sempre pelo descritor e apagar o ficheiro, depois usa-se o *unlink()* e fecha-se o descritor. Se possível utilizar *funlink()* e/ou *ftruncate()*.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
unlink	Condição de disputa	Devido há vulnerabilidade TOCTOU, não se deve usar *unlink()* e utilizar *lstat()*, depois usa-se o *open()* e o *fstat()*, se a comparação dos dados das duas estruturas estiver correcta usa-se o ficheiro, após isto fecha-se *fclose()* e utiliza-se o *unlink()*.
utime	Condição de disputa	A vulnerabilidade TOCTOU provoca problema com a utilização do *utime()*. Para evitar utilize o *futimes()*.
utimes	Condição de disputa	A vulnerabilidade TOCTOU provoca problema com a utilização do *utimes()*. Para evitar utilize o *futimes()*.
chroot	Controlo de acesso	Depois de executar a função *chroot()*, deve-se executar a função *chdir()*.
execlp	Controlo de acesso	Esta função utiliza a variável de ambiente PATH. A solução pode passar pela diminuição dos privilégios e nunca deixar o utilizador inserir conteúdo para os argumentos da função.
execvp	Controlo de acesso	Esta função utiliza a variável de ambiente PATH. A solução pode passar pela diminuição dos privilégios e nunca deixar o utilizador inserir conteúdo para os argumentos da função.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
open	Controlo de acesso	Sse utiliza o *open()* para criar um ficheiro novo, então deve-se utilizar a função com o modo O_EXCL |O_CREAT.
popen	Controlo de acesso	Devido a possíveis alterações nas variáveis de ambiente.
system	Controlo de acesso	Utilizar o *exec()* para evitar a invocação de uma consola de comandos, a partir do *system()*.
umask	Controlo de acesso	Utilizar sempre o *umask()* para o "mundo" e grupos, estes não podem ler, escrever ou executar o ficheiro em causa.
fprintf	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. Se possível utilizar o *snprinf()*. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.
printf	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. Se possível utilizar o *snprintf()*. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
snprintf	Formatação de cadeias de caracteres	Alguns sistemas têm esta função a chamar directamente *sprintf()*.
sprintf	Formatação de cadeias de caracteres	Utilizar a função *snprintf()*, onde se pode limitar os dados impressos. Apesar de alguns sistemas quando é utilizado *snprintf()* chamam directamente *sprintf()*. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.
syslog	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.
vfnprintf	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.
vsnprintf	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.

Continua na página seguinte.

Continuação da página anterior.

Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
vsprintf	Formatação de cadeias de caracteres	Limitar os dados a imprimir ou utilizar expressões regulares para eliminar conteúdo impróprio. A forma de limitar os caracteres a imprimir é (...,"%20s",...) – onde 20 é o número de caracteres a imprimir.
strlen	Acesso fora dos limites	Assegurar que a cadeia de caracteres tem o carácter "\0".
rand	Números aleatórios	O algoritmo é conhecido e fácil de descobrir o próximo valor aleatório. Utilizar somente *srand()* ou *srandom()*, acompanhado com bons *seeds*.
random	Números aleatórios	O algoritmo é conhecido e fácil de descobrir o próximo valor aleatório. Utilizar somente *srand()* ou *srandom()*, acompanhado com bons *seeds*.
srand	Números aleatórios	Não utilizar *seeds* constantes, mas valores aleatórios ou algo semelhante como a hora corrente.
srandom	Números aleatórios	Não utilizar *seeds* constantes, mas valores aleatórios ou algo semelhante como a hora corrente.
select	Validação de parâmetros	A função em conjunto com as macros FD_SET, FD_CLR e FD_ISSET, não verifica o valor de *fd* e que deve ser verificado do seguinte modo, fd $\geq$ 0 ou fd $\leq$ FD_SETSIZE.

Continua na página seguinte.

Continuação da página anterior.		
Função vulnerável	**Tipo de vulnerabilidade**	**Descrição e Solução**
strcmp	Falta do terminador	Sempre que se compara duas cadeias de caracteres, deve-se ter presente nas duas o terminador de fim de uma cadeia de caracteres. A solução é garantir que o terminador "\0" se encontra no fim de cada cadeia de caracteres.

Apêndice D

Máximo e Mínimo para Tipos de Dados

Um computador está associado a uma arquitectura de processamento — a mais conhecida é a x86 da Intel. O objectivo do processador num computador consiste em processar bits, porém estes têm um máximo quanto ao número de bits que conseguem trabalhar, ou seja, realizar operações aritméticas. A maioria dos computadores já se encontra com processadores de 32 bits, isto é, 4 bytes, no entanto, já há processadores de 64 bits e brevemente de 128 bits.

Um processador quando é designado por 32 bits, significa que o processador acede à memória a 32 bits, ou seja, adquire da memória 32 bits de dados.

Por exemplo, um processador quando tem de somar dois inteiros e suporta um processamento de 32 bits (ver Tabela D.1), então soma os dois valores acedendo à memória a 32 bits e caso ultrapasse um dos limites (máximo ou mínimo) volta ao limite contrário e continua o cálculo, devolvendo sempre um valor de 32 bits.

Para compreender melhor os máximos e mínimos que um inteiro pode alcançar num processador de 32 bits, é apresentada na página seguinte numa tabela. Notar que para um processador de 64 bits ou 128 bits, os inteiros correspondem directamente ao máximo que o processador pode operar — 64 bits ou 128 bits. Um programador deve saber sempre qual a arquitectura do processador alvo, onde a aplicação irá executar, para evitar problemas de *Exceder o Espaço de Representação do Inteiro*.

Tipos de dados	Número de bits	Mínimo	Máximo
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	32	-2.147.483.648	2.147.483.647
unsigned int	32	0	4.294.967.295
signed int	32	-2.147.483.648	2.147.483.647
short	16	-32.768	32.767
unsigned short	16	0	65.535
signed short	16	-32.768	32.767
long	32	-2.147.483.648	2.147.483.647
unsigned long	32	0	4.294.967.295
signed long	32	-2.147.483.648	2.147.483.647
float	32	3,4E-38	3.4E+38
double	64	1,7E-308	1,7E+308
long double	80	3,4E-4932	3,4E+4932

Tabela D.1: Espaço de representação para tipos numéricos.

Apêndice E

Resumo das Ferramentas Utilizadas

As ferramentas estudadas e analisadas são resumidas nas páginas seguintes. Além do resumo encontra-se também informação extra não salientada na Secção 3.2.

A informação apresentada aqui é importante quando se pretende saber mais características das ferramentas e os seus comportamentos. Assim, informamos o tipo de código que pode analisar, a linguagem de programação que foi concretizada, as técnicas de análise utilizadas, as possíveis vulnerabilidades a encontrar e um resumo de observações relevantes. Notar que à frente do nome da ferramenta se encontra a versão utilizada neste trabalho, verificando-se até ao momento que algumas ferramentas foram actualizadas.

Tabela E.1: Informação adicional das ferramentas avaliadas.

Nome da Ferramenta	Código a analisar	Código fonte	Técnica de análise	Vulnerabilidades	Observações
Crystal (1.0)*	C	Java	Árvore abstracta de sintaxe, Análise do fluxo de dados e Controlo de fluxo por grafos	Gestão de memória	Muito recente Novembro de 2006. É uma plataforma de suporte ainda em desenvolvimento. Realiza uma análise *intraprocedural* e *interprocedural*. Só consegue analisar ficheiros, onde pelo menos um deles tem a função *main*. Se para dois ficheiros com conteúdos distintos a analisar, a ferramenta não analisa devido há existência de duas *main's*. Tem que se criar um ficheiro com extensão <*.i> para os ficheiros <*.c> a analisar, depois já analisa todos os ficheiros de uma só vez.
Deputy (1.0)*	C	Cil	Árvore abstracta de sintaxe e Controlo de fluxo por grafos	Acesso fora de limites de uma lista, Variáveis não iniciializadas, Memória sobrelotada, Controlo de acesso, Condições de disputa, Conversão explicita de tipo e Referências nulas de apontadores	Ferramenta recente, ano 2006. Possibilita ao programador introduzir anotações (21). Demora um minuto para analisar trinta mil linhas. Utiliza o Cil como linguagem e o compilador OCaml. Dá informação sobre o erro, assim como informa o utilizador, como ajudar a ferramenta a interpretar melhor o erro detectado. Declara erros e alertas. Verifica mais que um ficheiro consecutivamente.

Continua na página seguinte.

Continuação da página anterior.

Nome da Ferramenta	**Código a analisar**	**Código fonte**	**Técnica de análise**	**Vulnerabilidades**	**Observações**
Flawfinder (1.26)	C e C++	Phyton	Análise de padrão	Memória sobrelotada, Formatação de cadeias de caracteres, Condições de disputa, Ficheiros temporários, Números aleatórios e Controlo de acesso	Examina qualquer extensão de ficheiro C/C++ (e.g., <*.h>). Tem 158 funções perigosas, C/C++. Não cria uma AST. Comandos de preprocessamento podem resultar problemas de análise. Num pentium dois, um programa com 51.055 linhas demorou 40 segundos. Última actualização em 2004. Como qualquer analisador por padrão não compreende a semântica do código. Verifica vários ficheiros consecutivamente.
ITS4 (1.1.1)	C e C++	C	Análise de padrão	Memória sobrelotada, Formatação de cadeias de caracteres, Condições de disputa, Ficheiros temporários, Números aleatórios e Controlo de acesso	Não constrói uma árvore abstracta de sintaxe, simplesmente procura por funções já registadas como vulneráveis da biblioteca C e C++. Até 57.000 linhas conseguiu analisar em 6 segundos. Muitos falsos positivos e encontra-se dependente de uma base de dados com funções vulneráveis. Não é actualizada à muito tempo (+6anos). Tem 145 funções perigosas. Dá alerta de possível vulnerabilidade no seguinte caso, <printf(“”);>. Verifica mais que um ficheiro ao mesmo tempo.

Continua na página seguinte.

Continuação da página anterior.

Nome da Ferramenta	Código a analisar	Código fonte	Técnica de análise	Vulnerabilidades	Observações
MOPS (0.9.2)	C	Java	Árvore abstracta de sintaxe, Controlo de fluxo por grafos e Verificação por especificações	Formatação de cadeias de caracteres, Memória sobrelotada, Controlo de acesso e Condições de disputa	A sua base de dados é limitada. O seu objectivo é verificar se não existe violações das propriedades de segurança, por exemplo verifica propriedades de *setuid*. Última actualização em 2004. Para 279.000 linhas examina-as em 1 minuto, para uma só propriedade. Requer a indicação da propriedade a analisar, não examina o código fonte sobre todas as propriedades que o MOPS tem especificado. O programa não pode conter erros detectados pelo compilador (GCC), senão não é analisado. Só indica que há um erro, se a propriedade a validar for violada. Se a função onde foi detectado o erro é usada pelo programa, então cria um ficheiro senão indica que há um erro e não cria ficheiro.

Continua na página seguinte.

Continuação da página anterior.

Nome da Ferramenta	**Código a analisar**	**Código fonte**	**Técnica de análise**	**Vulnerabilidades**	**Observações**
PScan (1.2)	C	C	Análise lexical	Formatação de cadeias de caracteres e Memória sobrelotada	Procura por funções do tipo *printf()*, *scanf()*, *syslog()*, ... É muito limitada na sua pesquisa. Uma vez limitada torna a sua execução rápida e eficiente. Última actualização foi em 2000. Verifica vários ficheiros ao mesmo tempo. A versão 1.1 tem problemas quanto à localização das vulnerabilidades, respectivamente à linha onde a vulnerabilidade ocorreu.
RATS (2.1)	C, C++, Perl, Php e Phyton	C	Análise de padrão	Memória sobrelotada, Formatação de cadeias de caracteres, Condições de disputa, Números aleatórios e Controlo de acesso	Tem 334 "testes" para funções perigosas. Não cria uma árvore abstracta de sintaxe. Sempre que encontra uma variável com um nome de uma função, considera que há uma vulnerabilidade. No entanto, das várias ferramentas é a que descreve com melhor clareza o problema e solução. O autor do ITS4 também participou e fez parte do RATS. Verifica mais que um ficheiro ao mesmo tempo.

Continua na página seguinte.

Continuação da página anterior.

Nome da Ferramenta	**Código a analisar**	**Código fonte**	**Técnica de análise**	**Vulnerabilidades**	**Observações**
Sparse (0.2)*	C	C	Análise semântica	Validação de tipos de dados, Incoerências com a norma ANSI e Acesso fora dos limites de uma lista	Última actualização em Maio de 2007 e começou a ser desenvolvido em 2003. É possível inserir anotações sobre tipos e apontadores. Construído por Linus Torvalds. A ferramenta só dá algo interessante se não imprimir erros do programa – dado que, este tem de pertencer há norma ANSI. A ferramenta não reporta vulnerabilidades de segurança em forma de avisos, enquanto houver erros. Verifica mais de um ficheiro ao mesmo tempo. Consegue perceber que há uma vulnerabilidade a partir da substituição da variável onde é usada, e com a análise por intervalos de valor detecta vulnerabilidades de *cast* ou mesmo de acesso fora dos limites de uma lista.

Continua na página seguinte.

Continuação da página anterior.

Nome da Ferramenta	**Código a analisar**	**Código fonte**	**Técnica de análise**	**Vulnerabilidades**	**Observações**
UNO (2.11)*	C	C	Verificação por especificações, Análise de fluxo de dados, Controlo de fluxo por grafos e Árvore abstracta de sintaxe	Variáveis não inicializadas, Acesso fora dos limites de uma lista e Referências nulas de apontadores	Última actualização em Março de 2007. Efectua análise *intraprocedural* e *interprocedural* das várias propriedades. Desenvolvido nos Laboratórios Bell. É possível saber a complexidade ciclomática que uma função tem. Verifica mais de um ficheiro consecutivamente. Esta ferramenta tem um bom comportamento para detectar problemas de acesso fora do limite de uma lista, assim como variáveis que não são inicializadas ou não utilizadas.

No ITS4 e Flawfinder é possível saber que funções recebem dados do "exterior". Nestes, o ITS4 comporta-se mal com a formatação de cadeias de caracteres, ao passo que o Flawfinder já não. O RATS por vezes para uma dada função anuncia dois alertas com diferentes tipos de informação.

** Há uma nova versão da ferramenta, consulte a Secção 3.2 para encontrar os respectivos url's de cada ferramenta.*

Apêndice F

Classes de Vulnerabilidades

Neste apêndice pretende-se qualificar vulnerabilidades. Para tal decidiu-se usar uma hierarquia de classes. A cada classe de vulnerabilidade é associada um conjunto de soluções e problemas que causam a vulnerabilidade. Todo o conteúdo apresentado é possível reajustar e complementar com mais detalhes, soluções e causas. A última vulnerabilidade não foi possível categorizar, dado que é uma vulnerabilidade muito abstracta.

1. Inicialização:

1.1. Os apontadores convêm que sejam sempre inicializados;

1.2. Inicializar variáveis antes do seu uso.

2. Conversão explicita de tipo:

2.1. Verificar qual o tipo de dados antes da conversão, *signed* para *unsigned* ou *char** para *char*, etc.

3. Validação de parâmetros:

3.1. *Input* do exterior da aplicação:

3.1.1. Não confiar em ficheiros, *sockets*, variáveis de ambiente, *stdin* que podem ser controlados por intrusos;

3.1.2. Formatação de cadeias de caracteres:

3.1.2.1 Validar a cadeia de caracteres caso a caso.

3.1.3. Memória sobrelotada:

3.1.3.1 Dados de entrada maiores que o espaço reservado.

3.1.4. Acesso a fora do limite de uma lista:

3.1.4.1. Verificar os limites de acesso antes de aceder.

3.1.5. Exceder o espaço de representação do inteiro:

3.1.5.1. Validar os limites consoante a máquina e o que o programa deseja receber.

3.1.6. Nunca confiar nos argumentos do programa, incluindo o *argv*;

3.1.7. Ataques por injecção de código HTML;

3.1.8. Ataques por injecção de código SQL;

3.1.9. Ataques por injecção de código PL/SQL;

3.1.10. Ataques por injecção de código PHP;

3.1.11. *Cross-site scrip*:

3.1.11.1. Utilizar expressões regulares ou algo semelhante, para validar o caminho do ficheiro; omitir "/", nova linha, pontos, "../", *, ?, [,], , , <, >, %, &, ... — previne o *cross-site* e acesso mal intencionado.

3.2. Formatação de cadeias de caracteres:

3.2.1. Sempre que se pretende escrever ou ler deve-se adequar o parâmetro de formatação para o tipo em causa;

3.2.2. Quando se pretende ler um carácter (%c) e guardar o valor num inteiro, o inteiro deve ser inicializado a zero;

3.2.3. Sempre que se pretende ler um inteiro para *char*, só deve aceitar valores entre [0 .. 255] (caso, *signed char*), porque para números elevados, pode causar memória sobrelotada em variáveis do tipo *char*;

3.2.4. Uma cadeia de caracteres no seu limite deve conter o carácter de terminação "\0";

3.2.5. O número de formatadores deve corresponder ao número de parâmetros;

3.2.6. Verificar se os argumentos passados são válidos.

3.3. Memória sobrelotada:

3.3.1. Verificar os limites de escrita, quer em índices negativos como positivos.

3.4. Acesso fora dos limites de uma lista:

3.4.1. Verificar os limites de acesso, mesmo com aritmética de apontadores.

3.5. Exceder o espaço de representação do inteiro:

3.5.1. Validar os limites consoante o que se pretende desenvolver;

3.5.2. Cuidados com as conversões explicitas de tipo, *double* para *int*.

3.6. Utilização correcta dos operadores relacionais em instruções condicionais, (==, >, <,>=, <=, !=) e (*if*, *while*, *for*, *do .. while*).

3.7. Divisão por zero:

3.7.1. Validar sempre se o divisor é diferente de zero.

3.8. Ciclos infinitos.

4. Validação do valor de retorno:

4.1. Verificar se o valor de retorno é o esperado, caso contrário agir em conformidade.

5. Gestão de memória:

5.1. Sempre que se reserva espaço na memória (*malloc*), deve-se ter o cuidado de o libertar (*free*) após a sua utilização ou não;

5.2. Nunca libertar o espaço de memória mais que uma vez;

5.3. Nunca libertar o espaço de memória e depois utilizar a variável;

5.4. Nunca libertar um conteúdo que nunca foi reservado;

5.5. Falta em reservar memória;

5.6. Acesso concorrente à memória:

5.6.1. Verificar se o acesso a uma zona de memória não está a ser utilizado por outro processo;

5.6.2. Bloquear o acesso à memória quando se está a escrever.

6. Funções com limitação de valores:

6.1. Em vez de usar *strcpy* usar *strncpy* ... e outras mais, consultar Apêndice C.

7. Acesso a ficheiros:

7.1. Verificar a existência do ficheiro a usar;

7.2. Verificar as permissões de escrita ou leitura, consoante o caso;

7.3. Verificar se não estamos perante um ficheiro de atalho;

7.4. Utilizar sempre o descritor, em vez do nome do ficheiro;

7.5. Nunca usar a função *access()*;

7.6. Um programa não pode passar o mesmo caminho para duas ou mais chamadas ao sistema (*chdir()*, *chmod()*, *chroot()*, *creat()*, *execve()*, *lchown()*, *link()*, *lstat()*, *mkdir()*, *mknod()*, *mount()*, *open()*, *pivot_root()*, *quotactl()*, *readlink()*, *rename()*, *rmdir()*, *stat()*, *statfs()*, *symlink()*, *truncate()*, *umount()*, *unlink()*, *uselib()*, *utime()*, *utimes()*, ...);

7.7. Fazer chdir("/") depois de chroot();

7.8. Ficheiros temporários:

7.8.1. Criar ficheiros temporários seguros em directorias não partilhadas como, "/tmp", "/var/tmp";

7.8.2. Nunca usar *tempnam()*, *tmpfile()*, *tmpnam()* e *mktemp()*;

7.8.3. Utilizar *mkstemp()* seguro: *umask(077)* antes do *mkstemp(x)*;

7.8.4. Não reutilizar o parâmetro x em *mkstemp(x)*.

7.9. Não abrir um ficheiro em modo de escrita para o *stdout* ou *stderr*, mesmo que não interesse saber quais os descritores abertos, quando o programa começa. Solução: abrir o ficheiro "/dev/null" três vezes no inicio do programa, deste modo nenhum ficheiro pode ser aberto pelo *stderr*.

8. Redução de privilégios:

8.1. Sempre que se executar uma das funções *execl()*, *execv()*, *popen()* ou *system()*, reduzir privilégios;

8.2. Diminuir os privilégios quando é criado um processo filho (o processo filho herda os privilégios do pai);

8.3. Cuidados a ter com o uso do *umask()*;

8.4. Variáveis de ambiente.

9. Criptografia:

9.1. Números aleatórios:

9.1.1. Usar bons *seeds*;

9.1.2. Nunca utilizar *rand()* e *random()* com *seeds* constantes;

9.1.3. Sempre que possível usar "/dev/(u)random".

9.2. Tamanho das chaves deve ser tão grande quanto possível;

9.3. Criar *nonces* seguros.

10. Bibliotecas:

10.1. Quando funções de biblioteca que não estão na memória, o programa dirige ao *stub* e lê a função. O atacante pode adicionar uma biblioteca dinâmica vulnerável (aplicação tem de executar com privilégios *root*).

11. Tratamento de excepções:

11.1. Tratar todas as excepções o quanto mais específico que possível.

12. Multi-tarefa:

12.1. Em sistemas multi-tarefa cuidado com a possibilidade de existir *deadlocks*, estudar formas de prevenir, como a utilização de semáforos.

13. Outros:

13.1. Não usar '&' quando não se pretende o endereço de memória da respectiva variável, contém consequências graves, exemplo "free(&str)".

Apêndice G

Código do Teste Padronizado

Os testes padronizados assentam numa agregação de informação específica para avaliar serviços ou produtos. Neste caso, o teste usa código com diversos tipos de vulnerabilidades como, memória sobrelotada, formatação da cadeia de caracteres, etc (ver Secção 3.3.2). Este código segue as propriedades enunciadas na Secção 3.3.

Os testes padronizados são muito importantes pois servem de guia para a avaliação das ferramentas, garantindo que todas são tratadas da mesma maneira.

Assim, na página seguinte é apresentado o código com as vulnerabilidades. É possível identificar facilmente uma vulnerabilidade pois esta está seguida de um comentário "// vulnerability" e para cada linha tem o correspondente número.

G. CÓDIGO DO TESTE PADRONIZADO

```
1 #include <pwd.h>
2 #include <time.h>
3 #include <stdio.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7 #include <limits.h>
8 #include <syslog.h>
9 #include <string.h>
10 #include <sys/stat.h>
11 #include <sys/types.h>
12
13 #define SCHAR_MAX 127
14 #define SHRT_MAX 32767
15 #define INT_MAX  2147483647
16 #define LONG_MAX 2147483647L
17 #define OUT_DEF  1000000
18 #define WORDSIZE  __WORDSIZE/4
19
20 /*****************************************************************************
21  * INITIALIZATION
22  * - variables must be initialize before used
23  *
24  * CAST
25  * - cast from a diffent(less) type (this can cause buffer overflow or underflow)
26  *
27  * PARAMETER VALIDATION
28  * - input validation
29  * - format string
30  * - buffer overflow/underflow
31  * - array out of bound
32  * - relational operator in conditional statements
33  * - division by zero
34  * - loops
35  *
36  * RETRUN VALUES
```

```
37  * - check if is the right value
38  *
39  * MEMORY MANAGEMENT
40  * - double free
41  * - never release memory
42  * - never allocate memory
43  * - release memory and used
44  * - never release some which never been allocated
45  *
46  * MORE SAFETY FUNCTIONS WITH VALUE RANGE
47  * - strcpy -> strncpy (..)
48  *
49  * FILE ACCESS
50  * - check if exist
51  * - check permissions
52  * - check if the file is not a symbolic link
53  * - use the descripter, instead of const name
54  * - never use access() function
55  * - a program must not pass the Path in two or more functions as:
56  * .open, stat, symlink, umount, unlink
57  * - must do chdir() after chroot()
58  * - create temp files in directories that are not share (! \tmp , \var\tmp)
59  * - never use tempnam, tmpfile, tmpnam, mktemp
60  * - before use mkstemp use umask(077)
61  *
62  * LEAST PRIVILEGES
63  * - system calls
64  * . execl, execv, popen, system
65  * - when create a child (fork), the child will have the same privileges as father
66  *
67  * CRIPTOGRAFY
68  * - random numbers
69  * . get good seeds
70  * . never used rand and random with a constant seed
71  * . try to use /dev/(u)random
72  *
```

```
73  * LIBRABRIES
74  *
75  * TRY CASH EXCEPTIONS
76  *
77  *
78  ****************************************************************************/
79
80 /****************************************************************************
81  * INITIALIZATION
82  * func_1_**()
83  ****************************************************************************
84  * - variables must be initialize before used
85  ****************************************************************************/
86 void func_1_01 (void) {
87    int i;
88    int j;            // vulnerability
89    for (i = 0; i < 10; i++) {
90                      for ( ; j > 0; j--) {
91                              printf("func_1_01");
92                      }
93    }
94 }
95
96 void func_1_02 (void) {
97    char *str1;          // vulnerability
98    char *str2;
99    printf("String 1: %s", str1);
100   str2 = (char *) malloc (10); // vulnerability
101   str2 = "string 2";
102   printf("String 2: %s", str2);
103   //free(str2); miss free
104 }
105
106 void func_1_03 (void) {
107   char   *str = "WELL DONE";
108   int a, b, c, d, e, f, g, h, i;
```

```
109   unsigned short j;       // vulnerability
110   a = 0; b = 1;
111   c = b + 1; d = b + c;
112   e = j;              // vulnerability
113   printf("First%cword:␣%c%c%c%c", str[e], str[a], str[b], str[c], str[d]);
114   f = 5; g = 6; h = 7; i = 8;
115   printf("Second%cword:␣%c%c%c%c", str[e], str[f], str[g], str[h], str[i]);
116
117   while ( j <= 8 )        // vulnerability
118     printf("%c", str[j]);
119 }
120
121 void aux_func_1_04(char* str){
122           str = malloc(10);       // vulnerability
123           str = "olllla";
124           printf(str);
125 }
126
127 void func_1_04(void){
128           char *str;
129           aux_func_1_04(str);
130 }
131
132 void aux_func_1_05(char* str){  //vulnerability
133           printf(str);          //vulnerability
134 }
135
136 void func_1_05(void){
137           char str[10];         //vulnerability
138           aux_func_1_04(str);
139 }
140 /*****************************************************************************
141  * CAST
142  * func_2_**()
143  *****************************************************************************
144  * - cast from a diffent(less) type (this can cause buffer overflow or underflow)
```

```
145  *****************************************************************************/
146 void func_2_01 (void) {
147   int si_a = 28;
148   int si_b = 80;
149   float sf_f;
150
151   sf_f = si_a /
152          si_b;   // vulnerability - miss cast in variable 'a' and 'b'
153 }
154
155 void func_2_02 (void) {
156   int si_a = 28;
157   int si_b = 80;
158   float sf_f;
159
160   sf_f = (float)si_a /
161          (float)si_b;
162 }
163
164 void func_2_03 (void) {
165   int si_a = 28000;
166   int si_b = 80000;
167   unsigned short us_s;
168
169   us_s = (unsigned short)si_b /  // vulnerability - overflow variable b
170          (unsigned short)si_a;
171 }
172
173 void func_2_04 (char *str) {
174   char sc_c = (int) str;      // vulnerability
175
176   printf("%c",sc_c);
177 }
178
179 /*****************************************************************************
180  * PARAMETER VALIDATION
```

```
181  * func_3_**()
182  ************************************************************************
183  * - input validation (format string, buffer ... )
184  * - format string
185  * - buffer overflow/underflow
186  * - array out of bound
187  * - integer overflow/underflow
188  * - relational operator in conditional statements
189  * - division by zero
190  * - loops
191  ************************************************************************/
192 /**
193  * Input validation
194  * - Format String
195  */
196 void func_3_01 (void) {
197   char str_scanf [10];
198   char str_sprintf [10];
199   char fstr_untrusted[10];
200   char fstr___trusted[10] = "%9s";
201
202   // get input
203   fgets(fstr_untrusted, sizeof(fstr_untrusted), stdin); //vulnerability
204
205   // use format strings for reading from stdin
206   scanf("%s", str_scanf); // vulnerability
207   scanf("%9s", str_scanf);
208   scanf(fstr___trusted, str_scanf);
209   scanf(fstr_untrusted, str_scanf); // vulnerability
210
211   // use format strings for reading from str_scanf
212   sprintf(str_sprintf, "%s", str_scanf); // vulnerability
213   sprintf(str_sprintf, "9%s", str_scanf);
214   sprintf(str_sprintf, fstr___trusted, str_scanf);
215   sprintf(str_sprintf, fstr_untrusted, str_scanf); // vulnerability
216   snprintf(str_sprintf, sizeof(str_sprintf), fstr_untrusted, str_scanf); // vulnerability
```

```
217
218   // use format strings for writing to syslog
219   syslog(LOG_ERR, "%s\n", str_sca nf); // vulnerability
220   syslog(LOG_ERR, "%9s\n", str_scanf );
221   syslog(LOG_ERR, fstr___trusted, str_scanf );
222   syslog(LOG_ERR, fstr_untrusted, str_scanf ); // vulnerability
223 }
224
225 /**
226  * Input validation
227  * - Buffer Overflow
228  */
229 void func_3_02 (void) {
230   char str__gets_nok[10];
231   char str_fgets__ok[10];
232   char str_fgets_nok[10];
233   char str_scanf__ok[10];
234   char str_scanf_nok[10];
235
236   gets (str__gets_nok);                  // vulnerability
237   fgets(str_fgets__ok, sizeof(str_fgets__ok) - 1, stdin); // vulnerability
238   fgets(str_fgets__ok, sizeof(str_fgets__ok), stdin); // vulnerability
239   fgets(str_fgets_nok, sizeof(str_fgets__ok) + 10, stdin); // vulnerability
240
241   scanf("%9s", str_scanf__ok);
242   scanf("%10s", str_scanf_nok); // vulnerability - illegal
243   scanf("%s", str_scanf_nok); // vulnerability - possible overflow
244 }
245
246 void func_3_03 (void) {
247   char printf[10], execv[10];
248   fgets (printf, sizeof(printf) - 1, stdin);//vulnerability
249   strcpy(execv, printf);
250 }
251
252 /**
```

```
253  * Input validation
254  * - Array out of bound
255  */
256 void func_3_04 (void) {
257   char *str[] = {"sub", "exp", "div"};
258   char *p_str;       // vulnerability
259   int index;
260   scanf("%d", &index);    // vulnerability - must be between 0 .. 2
261   p_str = str[index];    // vulnerability
262 }
263
264 void func_3_05 (void) {
265   char *str[] = {"sub", "exp", "div"};
266   char *p_str;       // vulnerability
267   int index;
268   scanf("%d", &index);    // vulnerability - must be between 0 .. 2
269   if(index >= 0 && index <= 2)
270         p_str = str[index];
271 }
272
273 /**
274  * Input validation
275  * - Integer overflow/underflow
276  */
277 void func_3_06 (void) {
278   unsigned short number_1;
279   unsigned short number_2;
280   unsigned short result_1;
281   unsigned short result_2;
282   scanf("%hi", &number_1); // vulnerability - possible overflow or underflow
283   scanf("%hi", &number_2); // vulnerability
284   result_1 = number_1 * number_2; // vulnerability - integer overflow
285   result_2 = number_1 - number_2; // vulnerability - integer underlfow
286 }
287
288 void func_3_07 (void) {
```

```
289   unsigned short const_num = 65535;
290   unsigned short result_1 = 0;
291   unsigned short result_2 = 0;
292   unsigned short number_1;
293   unsigned short number_2;
294   char number[6];
295   fgets(number, 6, stdin);   // vulnerability
296   if ( number[0] != '-' ) {
297     number_1 = (unsigned short)atoi(number);
298
299     bzero(number, 6);
300     if( fgets(number, 6, stdin) != NULL ) {
301
302       if ( number[0] != '-' ) {
303         number_2 = (unsigned short)atoi(number);
304
305         if ( (const_num - number_2) > number_1 ) {
306           result_1 = number_1 + number_2; //never integer overflow
307
308           if ( number_1 >= number_2 )
309                result_2 = number_1 - number_2; //never integer underlfow
310                  }
311           }
312         }
313   }
314 }
315
316 /**
317  * Input validation
318  * - Division by zero
319  */
320 void func_3_08 (void) {
321   int number, result, diff_zero;
322   scanf("%d", &number);      // vulnerability
323   scanf("%d", &diff_zero);    // vulnerability
324   result = number / diff_zero;  // vulnerability
```

```
325 }
326
327 void func_3_09 (void) {
328   int number, result, diff_zero;
329   scanf("%d", &number);      // vulnerability
330   scanf("%d", &diff_zero);   // vulnerability
331   if ( diff_zero > 0 )
332     result = number / diff_zero;
333 }
334
335 /**
336  * Format strings
337  */
338 void func_3_10 (void) {
339   char str1[] = "First format string";
340   char str2[] = "Second format string";
341   char *p = str1;
342   printf(p);
343   printf(str2);
344 }
345
346 void func_3_11 (void) {
347   char buf[10];
348   if ( fgets (buf, sizeof(buf), stdin)!= NULL ){
349     printf(buf);  // vulnerability
350     printf("%s", buf);
351   } else {
352     printf("ERROR:: on stdin!");
353   }
354 }
355
356 void func_3_12 (void) {
357   char *msg = "test";
358   fprintf(stdout ,"fingerd: %s: \r\n", msg);
359   exit(1);
360 }
```

```
361
362 void func_3_13 (void) {
363   char *p, buf[10];
364   gets (p);       // vulnerability
365   system(p);      // vulnerability
366   strcpy(buf,"date");
367   system(buf);
368   printf(buf + 5);
369   printf("\n");
370   printf("%s", buf + 8); // vulnerability
371   printf("%s", buf + 11); // vulnerability
372   bzero (buf, 10);
373 }
374
375 void func_3_14 (void) {
376   const char *buf = "null terminate";
377   char aux_str[10];
378   char str [10];
379   int i;
380   for(i = 0; i < 4; i++) {
381     str[i] = buf[i];
382     aux_str[i] = buf[i];
383   }
384
385   printf("%s", str);  // vulnerability
386   i = 10;
387   aux_str[i] = '\0';  // vulnerability
388   printf("%s", aux_str);
389 }
390
391 /**
392  * Buffer overflow/underflow
393  *
394  */
395 void func_3_15 (void) {
396   char dst__ok[20];
```

```
397   char dst_nok[5];
398   char src[] = "1234567890";
399
400   // dangerous functions
401   strcpy(dst__ok, src);
402   strcpy(dst_nok, src); // vulnerability - dst_nok is too small
403   if (sizeof(dst_nok) >= sizeof(src))
404     strcpy(dst_nok, src);
405   strncpy(dst__ok, src, sizeof(dst__ok));
406   strncpy(dst_nok, src, sizeof(dst_nok));
407   strncpy(dst_nok, src, sizeof(src)); // vulnerability - dst_nok is too small
408   memcpy (dst__ok, src, sizeof(dst__ok));
409   memcpy (dst_nok, src, sizeof(dst_nok));
410   memcpy (dst_nok, src, sizeof(src)); // vulnerability - dst_nok is too small
411
412   // reset
413   strcpy(dst__ok, "1");
414   strcpy(dst_nok, "1");
415
416   strcat(dst__ok, src);
417   strcat(dst_nok, src);               // vulnerability
418   if (sizeof(dst_nok)-strlen(dst_nok)+1 >= sizeof(src))
419     strcat(dst_nok, src);
420
421   // reset
422   strcpy(dst__ok, "1");
423   strcpy(dst_nok, "1");
424
425   strncat(dst__ok, src, sizeof(dst__ok)-strlen(src)-1);
426   strncat(dst__ok, src, sizeof(dst__ok)-strlen(src)); // vulnerability
427   strncat(dst_nok, src, strlen(src));         // vulnerability
428   if (sizeof(dst_nok)-strlen(dst_nok)+1 >= sizeof(src))
429     strncat(dst_nok, src, strlen(src));
430 }
431
432 void func_3_16 (void) { //underflow
```

```
433   char str[10] = "OLA";
434   int i = 0;
435   while ( i <= 0 ) {
436     str[i] = 'a';   // vulnerability
437     i--;       // vulnerability
438   }
439 }
440
441 void func_3_17 (void) {
442   char buf[4];
443   char *p;
444   p = buf;
445   p += 50;       // vulnerability
446   *p = 'X';      // vulnerability
447 }
448
449 void func_3_18 (void) {
450   char buf[4] = "BLA";
451   buf[7] = 'X'; // vulnerability
452 }
453
454 void func_3_19 (void) {
455   char buf[1024];
456   int i = -1;
457   char c;
458   while ( (c = getchar()) != '\n') {
459     if ( c == -1 )
460       break;
461     buf[++i] = c;   // vulnerability
462   }
463 }
464
465 /**
466  * Array out of bound
467  * - read/write out of bounds in stack (access through array/pointer)
468  */
```

```
469 void func_3_20 (void) {
470   int array[8] = { 0, 1, 2, 3, 4, 5, 6, 7}; // A global array
471   int n = -1;
472
473   if(array[n] != 0)  // vulnerability
474     printf("%d", n);
475   if(array[-5] != 0)  // vulnerability
476     printf("-5%d", n);
477   if(array[2] != 7)
478     printf("2");
479 }
480
481 void func_3_21 (void) {
482   char *str = "booomm";
483   int i = 0;
484   while(str[i] != '*') {// vulnerability
485     i++;
486     str[i] = 'x';    // vulnerability
487   }
488 }
489
490 void func_3_22 (void) {
491   int value;
492   int array[10];
493   int OUT_VAR = OUT_DEF;
494
495   // array-form: read
496   value = array[0];
497   value = array[10]; // vulnerability - out of bounds
498   value = array[OUT_DEF]; // vulnerability - way out of bounds!
499   value = array[OUT_VAR]; // vulnerability - way out of bounds!
500   value = array[rand()]; // vulnerability - maybe out of bounds!
501
502   // array-form: write
503   value = 10;
504   array[0] = value;
```

```
505   array[10] = value; // vulnerability - out of bounds
506   array[OUT_DEF] = value; // vulnerability - way out of bounds!
507   array[OUT_VAR] = value; // vulnerability - way out of bounds!
508   array[rand()] = value; // vulnerability - maybe out of bounds!
509
510   // pointer-form: read
511   value = (int) array + (0*sizeof(int));
512   value = (int) array + (10*sizeof(int)); // vulnerability - out of bounds
513   value = (int) array + (OUT_DEF*sizeof(int)); // vulnerability - way out of bounds!
514   value = (int) array + (OUT_VAR*sizeof(int)); // vulnerability - way out of bounds!
515   value = (int) array + (rand()*sizeof(int)); // vulnerability - maybe out of bounds!
516
517   // pointer-form: write
518   value = 10;
519   *(array+0) = value;
520   *(array+10) = value; // vulnerability - out of bounds
521   *(array+OUT_DEF) = value; // vulnerability - way out of bounds!
522   *(array+OUT_VAR) = value; // vulnerability - way out of bounds!
523   *(array+rand()) = value; // vulnerability - maybe out of bounds!
524 }
525
526 /**
527  * Array out of bound
528  * - read/write out of bounds in heap (access through array/pointer)
529  */
530 void func_3_23 (void) {
531   int value;
532   int *array = (int *) malloc(10 * sizeof(int));
533   int OUT_VAR = OUT_DEF;
534
535   // array-form: read
536   value = array[0];
537   value = array[10]; // vulnerability - out of bounds
538   value = array[OUT_DEF]; // vulnerability - way out of bounds!
539   value = array[OUT_VAR]; // vulnerability - way out of bounds!
540   value = array[rand()]; // vulnerability - maybe out of bounds!
```

```
541
542   // array-form: write
543   value = 10;
544   array[0] = value;
545   array[10] = value; // vulnerability - out of bounds
546   array[OUT_DEF] = value; // vulnerability - way out of bounds!
547   array[OUT_VAR] = value; // vulnerability - way out of bounds!
548   array[rand()] = value; // vulnerability - maybe out of bounds!
549
550   // pointer-form: read
551   value = (int) array + (0*sizeof(int));
552   value = (int) array + (10*sizeof(int)); // vulnerability - out of bounds
553   value = (int) array + (OUT_DEF*sizeof(int)); // vulnerability - way out of bounds!
554   value = (int) array + (OUT_VAR*sizeof(int)); // vulnerability - way out of bounds!
555   value = (int) array + (rand()*sizeof(int)); // vulnerability - maybe out of bounds!
556
557   // pointer-form: write
558   value = 10;
559   *(array+0) = value;
560   *(array+10) = value; // vulnerability - out of bounds
561   *(array+OUT_DEF) = value; // vulnerability - way out of bounds!
562   *(array+OUT_VAR) = value; // vulnerability - way out of bounds!
563   *(array+rand()) = value; // vulnerability - maybe out of bounds!
564
565   free(array);
566 }
567
568
569 /**
570  * Integer overflow/underflow
571  * - underflow/overflow of different datatypes (access through variable/pointer)
572  */
573 void func_3_24 (void) {
574   // variables
575   signed char _char;
576   signed short _short;
```

```
577  signed int _int;
578  signed long _long;
579
580  unsigned char _uchar;
581  unsigned short _ushort;
582  unsigned int _uint;
583  unsigned long _ulong;
584
585  // pointers
586  signed char *p_char = &_char;
587  signed short *p_short = &_short;
588  signed int *p_int = &_int;
589  signed long *p_long = &_long;
590
591  unsigned char *p_uchar = &_uchar;
592  unsigned short *p_ushort = &_ushort;
593  unsigned int *p_uint = &_uint;
594  unsigned long *p_ulong = &_ulong;
595
596  /**
597   * Underflow
598   */
599  // variable reset
600  _char = CHAR_MIN;
601  _short = SHRT_MIN;
602  _int = INT_MIN;
603  _long = LONG_MIN;
604  _uchar = 0;
605  _ushort = 0;
606  _uint = 0;
607  _ulong = 0;
608
609  // variable decrement
610  _char -= 10; // vulnerability
611  _short -= 10; // vulnerability
612  _int -= 10; // vulnerability
```

```
613     _long -= 10; // vulnerability
614
615     _uchar -= 10; // vulnerability
616     _ushort -= 10; // vulnerability
617     _uint -= 10; // vulnerability
618     _ulong -= 10; // vulnerability
619
620     // variable reset
621     _char = CHAR_MIN;
622     _short = SHRT_MIN;
623     _int = INT_MIN;
624     _long = LONG_MIN;
625     _uchar = 0;
626     _ushort = 0;
627     _uint = 0;
628     _ulong = 0;
629
630     // pointer decrement
631     *p_char -= 10; // vulnerability
632     *p_short -= 10; // vulnerability
633     *p_int -= 10; // vulnerability
634     *p_long -= 10; // vulnerability
635
636     *p_uchar -= 10; // vulnerability
637     *p_ushort -= 10; // vulnerability
638     *p_uint -= 10; // vulnerability
639     *p_ulong -= 10; // vulnerability
640
641     /**
642      * Overflow
643      */
644     // variable reset
645     _char = CHAR_MAX;
646     _short = SHRT_MAX;
647     _int = INT_MAX;
648     _long = LONG_MAX;
```

```
649   _uchar = UCHAR_MAX;
650   _ushort = USHRT_MAX;
651   _uint = UINT_MAX;
652   _ulong = ULONG_MAX;
653
654   // variable increment
655   _char += 10; // vulnerability
656   _short += 10; // vulnerability
657   _int += 10; // vulnerability
658   _long += 10; // vulnerability
659
660   _uchar += 10; // vulnerability
661   _ushort += 10; // vulnerability
662   _uint += 10; // vulnerability
663   _ulong += 10; // vulnerability
664
665   // variable reset
666   _char = CHAR_MAX;
667   _short = SHRT_MAX;
668   _int = INT_MAX;
669   _long = LONG_MAX;
670   _uchar = UCHAR_MAX;
671   _ushort = USHRT_MAX;
672   _uint = UINT_MAX;
673   _ulong = ULONG_MAX;
674
675   // pointer increment
676   *p_char += 10; // vulnerability
677   *p_short += 10; // vulnerability
678   *p_int += 10; // vulnerability
679   *p_long += 10; // vulnerability
680
681   *p_uchar += 10; // vulnerability
682   *p_ushort += 10; // vulnerability
683   *p_uint += 10; // vulnerability
684   *p_ulong += 10; // vulnerability
```

```
685 }
686
687 /**
688  * Integer overflow/underflow
689  * - underflow/overflow of different datatypes (access through casted pointer)
690  */
691 void func_3_25 (void) {
692   // variables
693   unsigned char _uchar = 0;
694   unsigned short _ushort = 0;
695   unsigned int _uint = 0;
696   unsigned long _ulong = 0;
697
698   // pointers
699   signed char *p_char = (signed char *) &_uchar;
700   signed short *p_short = (signed short *) &_ushort;
701   signed int *p_int = (signed int *) &_uint;
702   signed long *p_long = (signed long *) &_ulong;
703
704   // casted pointer decrement
705   *p_char -= 10; // vulnerability
706   *p_short -= 10; // vulnerability
707   *p_int -= 10; // vulnerability
708   *p_long -= 10; // vulnerability
709
710   *p_char = 0;
711   *p_short = 0;
712   *p_int = 0;
713   *p_long = 0;
714
715   *p_char += 10; // vulnerability
716   *p_short += 10; // vulnerability
717   *p_int += 10; // vulnerability
718   *p_long += 10; // vulnerability
719 }
720
```

```
721 /**
722  * Relational operator in conditional statements
723  */
724 void func_3_26 (void) { //overflow
725   char str[10] = "Hello";
726   int i = 0;
727
728   while ( i <= 10 ) {          // vulnerability - the operator equal
729         if ( str[i] != 'e' && str[i] != 'o' ) // vulnerability
730           str[i] = 'a';            // vulnerability
731         i++;
732   }
733 }
734
735 void func_3_27 (void) { //overflow
736   char str[10] = "Hello";
737   int i = 0;
738
739   while ( i < 15 ) {           // vulnerability
740         if ( str[i] != 'e' && str[i] != 'o' ) // vulnerability
741           str[i] = 'a';            // vulnerability
742         i++;
743   }
744 }
745
746 void func_3_28 (void) {
747   int i;
748   int *p1 = malloc (10 * sizeof(int));
749   int *p2 = malloc (10 * sizeof(int));  // vulnerability
750   for ( i = 0; i < 10; i++ )
751     p2[i] = 2;
752   for ( i = 0; i < 15; i++ )       // vulnerability
753     p1[i] = 3;               // vulnerability
754
755   free(p1);
756   free(&p2);
```

```
757 }
758
759 void func_3_29 (void) {
760   int F_SYS = 0;
761   int root = 0;
762   //Compiler detect this problem
763   //if F_SYS or root use the operater '='
764   //but with -wall detect
765   if ( root == 0 && F_SYS == 1 )
766     root = 1;
767 }
768
769 void func_3_30 (void) {
770   int i;
771   char *name = "Code Security";
772
773   //The operater '&&' should be '||'
774   for(i = 0; *name == 'c' && i != (strlen(name) - 1); i++) // vulnerability
775     printf("No 'c' found yet!");
776 }
777
778 /**
779  * Division by zero
780  */
781 //detect by compiler -Wall
782 /*int func_3_31 (int num) {
783   return num / 0;
784 }*/
785
786 int func_3_32 (int num) {
787   int zero = 0;
788   return num / zero;          // vulnerability
789 }
790
791 int func_3_33 (int number, int d) {
792   if ( d == 0 )
```

```
793         return -1;
794   return (number * number) / (number * d);
795 }
796
797 int func_3_34 (int number, int d) {
798   return (number * number) / (number * d); // vulnerability
799 }
800
801 //compiler detect -Wall
802 /*int func_3_35 (int number) {
803         return (number * number) / (number * 0);
804 }*/
805
806 /**
807  * Loops
808  */
809 void func_3_36 (void) {
810   int i;        // vulnerability
811   for ( ; i <= 10; i++ ) // vulnerability
812     printf("HELLO");
813 }
814
815 void func_3_37 (void) {
816   int i = 6;
817   int j = 5;
818   i += j;
819   while ( i != 10 ) {   // vulnerability
820     j *= i;        // vulnerability
821     i++;         // vulnerability
822   }
823 }
824
825 int aux_func_3_38 (int i) {
826   if ( i == 0 )
827     return 1;
828   else
```

```
829       aux_func_3_38(--i);  // vulnerability - must be i--
830     return 0;
831 }
832
833 void func_3_38 (void) {
834     aux_func_3_38(10);
835 }
836
837 /*****************************************************************************
838  * RETURN VALUES
839  * func_4_**()
840  *****************************************************************************
841  * - check if is the right value
842  *****************************************************************************/
843 int *aux_func_4_01 (int i) {
844     int p[10];
845     p[i] = 1;     // vulnerability
846     return (int *)p[i]; // vulnerability
847 }
848
849 void func_4_01 (void) {
850     int *pi = aux_func_4_01(4);
851     int i = *pi;
852     printf("the number at 4 is %d", i);
853 }
854
855 char aux_func_4_02 (int i) {
856     char *array = "this is a array";
857     return array[i];
858 }
859
860 void func_4_02 (void){
861     int i;
862     for( i = 0; i < 10; i++ )
863       aux_func_4_02(i);
864 }
```

```
865
866 int aux_func_4_03 (int m, int n) {
867   int calc = (m * n) / n;
868   if ( calc >= 5 )
869     return calc;
870   return -1;
871 }
872
873 void func_4_03 (void) {
874   int index;
875   char *str = "some phrase to ...";
876   index = aux_func_4_03(5, 4);
877   printf("%c", str[index]);
878   index = aux_func_4_03(4, 4);
879   printf("%c", str[index]);  // vulnerability
880 }
881
882 /*****************************************************************************
883  * MEMORY MANAGEMENT
884  * func_5_**()
885  *****************************************************************************
886  * - double free
887  * - never release memory
888  * - never allocate memory
889  * - release memory and used
890  * - never release some which never been allocated
891  *****************************************************************************/
892
893 /**
894  * - double free
895  */
896 void aux_func_5_01 (char *buf) {
897   char *abc = "------- testing -------";
898   int j = 0;
899   while (abc++ != '\0') // vulnerability - loop :: *abc++
900     j++;        // vulnerability
```

```
901    printf("%d", j);
902    free(buf);
903  }
904
905  void func_5_01 (void) {
906    char *buf;
907    int i;
908    buf = (char *) malloc(256);
909    //(...)
910
911    for (i = 0; i<= 10; i++)
912      printf("%c", buf[i]);
913
914    aux_func_5_01(buf);
915
916    free(buf);        // vulnerability - double free
917  }
918
919  /**
920   * - never release memory
921   */
922  void func_5_02(void) {
923    int a = 1;
924    int d, b;
925    char *p = (char*) malloc (10); // vulnerability - no free (p)
926    char *ptr = NULL;
927    char str[50];
928
929    d = b * 2;           // vulnerability - initialization
930
931    for(a = 0; a < 100; a++)
932      ptr = (char*) malloc(200); // vulnerability - create and no free
933
934    printf("%c", str[51]);    // vulnerability
935    printf("%s", p);
936  }
```

```
937
938 /**
939  * - never allocate memory
940  */
941 void func_5_03 (void) {
942   int *p1 = NULL, *p2;
943   p1 = (int *) malloc (sizeof(int));
944   *p2 = 65;          // vulnerability
945   free(p1);
946 }
947
948 void func_5_04 (void) {
949   int i;
950   char *s;
951   for ( i = 0; i <= 10; i++ )
952     printf("%c", s[i]);      // vulnerability
953 }
954
955 /**
956  * - release memory and used
957  */
958 void func_5_05 (void) {
959   char *str = (char*) malloc (256);
960   fgets (str, 256, stdin);     // vulnerability
961   printf("%s", str);
962   free (str);
963   printf("One more time: %s", str); // vulnerability
964 }
965
966 /**
967  * - never release some, which never been allocated
968  */
969 void func_5_06 (void) {
970   char b = 'b';
971   int a = 33;
972   printf("%d %c", a, b);
```

```
973   free (&b);      // vulnerability
974   free (&a);      // vulnerability
975 }
976
977 void func_5_07 (void){
978   char *s = "Double Free";
979   char *m;
980   char *t;            // vulnerability
981   int count;           // vulnerability
982   m = malloc(1024);
983   fgets (m, 100, stdin);      // vulnerability
984   // if m assigmnent to something then in the 2ºnd printf, we'll have illegal access
985   printf("This is possible! %s", s);
986   free (m);
987   fgets (t, sizeof(t), stdin);   // vulnerability - miss malloc for 't'
988   while ( count != 10 ) {      // vulnerability - loop
989     //is possible access a region out of m
990     printf("%c", *(m+count)); // vulnerability
991     printf("%.10d \n", count--);
992   }
993   free(m);            // vulnerability
994 }
995
996 void func_5_free (int *pointer_fnc_freed) {
997   free(pointer_fnc_freed);
998 }
999
1000 void *func_5_malloc (size_t size) {
1001   return malloc(size);
1002 }
1003
1004 void func_5_08 (void) {
1005   unsigned int i = 0;
1006   int array[10];
1007   /**
1008    * Mallocs
```

```
1009    */
1010   int *pointer__malloc__free = (int *) malloc(10 * sizeof(int));
1011   int *pointer__malloc____ = (int *) malloc(10 * sizeof(int)); //vulnerability(1)
1012   int *pointer________free;
1013   int *pointer_fmalloc__free = (int *) func_5_malloc(10 * sizeof(int));
1014   int *pointer_fmalloc____ = (int *) func_5_malloc(10 * sizeof(int));//vulnerability(2)
1015   int *pointer__malloc_ffree = (int *) malloc(10 * sizeof(int)); //vulnerability(3)
1016   int *pointer_______ffree;
1017   int *pointer_fmalloc_ffree = (int *) func_5_malloc(10 * sizeof(int));
1018
1019   // simple write
1020   for (i = 0; i<10; i++) {
1021     array[i] = 10;
1022     pointer__malloc__free[i] = 10;
1023     pointer__malloc____[i] = 10;
1024     pointer________free[i] = 10; // vulnerability
1025     pointer_fmalloc__free[i] = 10;
1026     pointer_fmalloc____[i] = 10;
1027     pointer__malloc_ffree[i] = 10;
1028     pointer_______ffree[i] = 10; // vulnerability
1029     pointer_fmalloc_ffree[i] = 10;
1030   }
1031   /**
1032    * Frees
1033    */
1034   free(array); // vulnerability
1035   free(pointer__malloc__free);
1036   //free(pointer__malloc_____);
1037   free(pointer__________free); // vulnerability
1038   free(pointer_fmalloc__free);
1039   //free(pointer_fmalloc_____);
1040   func_5_free(pointer__malloc_ffree);
1041   func_5_free(pointer_________ffree);// vulnerability
1042   func_5_free(pointer_fmalloc_ffree);
1043
1044   // simple write
```

```
1045    for (i = 0; i<10; i++) {
1046      array[i] = 10;
1047      pointer__malloc__free[i] = 10;  // vulnerability
1048      pointer__malloc______[i] = 10;
1049      pointer__________free[i] = 10; // vulnerability
1050      pointer_fmalloc__free[i] = 10; // vulnerability
1051      pointer_fmalloc______[i] = 10;
1052      pointer__malloc_ffree[i] = 10; // vulnerability
1053      pointer_________ffree[i] = 10; // vulnerability
1054      pointer_fmalloc_ffree[i] = 10; // vulnerability
1055    }// (1)(2) vulnerability - MISS FREE
1056 }
1057
1058 void func_5_09 (void) {
1059    int *temp;
1060    /**
1061     * Mallocs
1062     */
1063    int *pointer__malloc__free__free = (int *) malloc(10 * sizeof(int));
1064    int *pointer_fmalloc__free__free = (int *) func_5_malloc(10 * sizeof(int));
1065    int *pointer__malloc_ffree__free = (int *) malloc(10 * sizeof(int));
1066    /**
1067     * Frees (1)
1068     */
1069    temp = pointer__malloc__free__free;
1070    free(pointer__malloc__free__free);
1071    pointer__malloc__free__free = temp;
1072
1073    temp = pointer_fmalloc__free__free;
1074    free(pointer_fmalloc__free__free);
1075    pointer_fmalloc__free__free = temp;
1076
1077    temp = pointer__malloc_ffree__free;
1078    func_5_free(pointer__malloc_ffree__free);
1079    pointer__malloc_ffree__free = temp;
1080    /**
```

```
1081     * Frees (2)
1082     */
1083    free(pointer__malloc__free__free); // vulnerability
1084    free(pointer_fmalloc__free__free); // vulnerability
1085    free(pointer__malloc_ffree__free); // vulnerability
1086 }
1087
1088 void func_5_10 (void) {
1089    /**
1090     * Mallocs (1)
1091     */
1092    int *pointer__malloc_ffree__malloc = (int *) malloc(10 * sizeof(int));
1093    int *pointer_fmalloc__free__malloc = (int *) func_5_malloc(10 * sizeof(int));
1094    int *pointer_fmalloc_ffree_fmalloc = (int *) func_5_malloc(10 * sizeof(int));
1095    /**
1096     * Frees
1097     */
1098    func_5_free(pointer__malloc_ffree__malloc);
1099    free( pointer_fmalloc__free__malloc);
1100    func_5_free(pointer_fmalloc_ffree_fmalloc);
1101    /**
1102     * Mallocs (2)
1103     */
1104    pointer__malloc_ffree__malloc = (int *) malloc(10 * sizeof(int));// vulnerability
1105    pointer_fmalloc__free__malloc = (int *) malloc(10 * sizeof(int));// vulnerability
1106    pointer_fmalloc_ffree_fmalloc = (int *) func_5_malloc(10 * sizeof(int));// vulnerability
1107 }
1108
1109 void func_5_11 (void) {
1110    /**
1111     * Mallocs (1)
1112     */
1113    int *pointer__malloc__malloc__free = (int *) malloc(10 * sizeof(int));// vulnerab
1114    int *pointer_fmalloc__malloc__free = (int *) func_5_malloc(10 * sizeof(int));// vulnerab
1115    /**
1116     * Mallocs (2)
```

```
1117     */
1118    pointer__malloc__malloc__free = (int *) malloc(10 * sizeof(int));
1119    pointer_fmalloc__malloc__free = (int *) malloc(10 * sizeof(int));
1120    /**
1121     * Frees
1122     */
1123    free(pointer__malloc__malloc__free);
1124    free(pointer_fmalloc__malloc__free);
1125 }
1126
1127 /*****************************************************************************
1128  * MORE SAFETY FUNCTIONS WITH VALUE RANGE
1129  * func_6_**()
1130  *****************************************************************************
1131  * - strcpy -> strncpy
1132  * - strcat -> strncat
1133  * - printf -> snprintf
1134  * - scanf -> fgets
1135  *****************************************************************************/
1136 void func_6_01 (void) {
1137    char dest[25];
1138    char src [75];
1139    fgets (src, 74, stdin); // vulnerability
1140    strcpy(dest, src);    // vulnerability
1141 }
1142
1143 void func_6_02 (void) {
1144    char dest[25];
1145    char src [75];
1146    fgets (src, 74, stdin);// vulnerability
1147    strncpy(dest, src, 25);// vulnerability
1148 }
1149
1150 void func_6_03 (void) {
1151    char str[10];
1152    scanf("%s", str);     // vulnerability
```

```
1153 }
1154
1155 void func_6_04 (void) {
1156   char str[10];
1157   fgets(str, 9, stdin);   // vulnerability
1158 }
1159
1160 /*****************************************************************************
1161  * FILE ACCESS
1162  * func_7_**()
1163  *****************************************************************************
1164  * - check if exist
1165  * - check permissions
1166  * - check if the file is not a symbolic link
1167  * - use the descripter, instead of const name
1168  * - never use access() function
1169  * - a program must not pass the Path in two or more functions as:
1170  * .open, stat, symlink, umount, unlink
1171  * - must do chdir() after chroot()
1172  * - create temp files in directories that are not share (! \tmp , \var\tmp)
1173  * - never use tempnam, tmpfile, tmpnam, mktemp
1174  * - before use mkstemp use umask(077)
1175  *****************************************************************************/
1176 /**
1177  * - check if exist
1178  */
1179 //read a file
1180 void func_7_01 (void) {
1181   FILE *ds;
1182   char *filename = "users.txt";
1183   char c;
1184   ds = fopen(filename, "r"); // vulnerability - file could not exist
1185   while ( (c = fgetc(ds)) != EOF )
1186     putchar(c);
1187
1188   fclose(ds);
```

```
1189 }
1190
1191 //check if exist and then read a file
1192 void func_7_02 (void) {
1193   FILE *ds;
1194   char *filename = "users.txt";
1195   char c;
1196   ds = fopen(filename, "r");
1197   if ( ds != NULL ) {
1198         while ( (c = fgetc(ds)) != EOF )
1199           putchar(c);
1200         fclose(ds);
1201   }
1202 }
1203
1204 void func_7_03 (char *first, char *second) { // vulnerability
1205   int a, b;
1206   a = open(first, O_RDONLY);
1207   b = open(second, O_RDONLY);
1208   if ( a == -1 || b == -1 ) {
1209     perror("open()");
1210     exit(1);
1211   }
1212   close(a);
1213   b = open(second, O_RDONLY);       // vulnerability
1214   //close(b);
1215 }
1216
1217 /**
1218  * - check permissions
1219  */
1220 //TOCTOU
1221 void func_7_04 (void) {
1222   int b;
1223   struct stat sbefore, safter;
1224   lstat("/usr/local/log.txt", &sbefore);
```

```
1225    if ( !access("/usr/local/log.txt", O_RDWR) ) { // Vulnerability
1226      b = open("/usr/local/log.txt", O_RDWR);   // vulnerability
1227      lstat("/usr/local/log.txt", &safter);
1228      if ( safter.st_ino == sbefore.st_ino ) {
1229        //do some
1230      }
1231    }
1232  }
1233
1234  //safe way
1235  void func_7_05 (void) {
1236    FILE *fs = NULL;
1237    char *filename = "/usr/local/log.txt";
1238    char c;
1239    int ds;
1240
1241    ds = open (filename, 600);
1242    fchmod(ds, 600);
1243    fs = fdopen(ds, "rw");
1244    if ( fs != NULL ) {
1245          while ( (c = fgetc(fs)) != EOF )
1246        putchar(c);
1247          fclose(fs);
1248    }
1249    fchmod(ds, 666);
1250  }
1251
1252  /**
1253   * - check if the file is not a symbolic link
1254   */
1255  //safe way
1256  void func_7_06 (void) {
1257    FILE *fs;
1258    char *filename = "/usr/local/log.txt";
1259    char buf[10];
1260    char c;
```

```
1261   int ds, link;
1262   ds = open(filename, O_RDWR);
1263
1264   link = readlink(filename, buf, 9);  // vulnerability
1265   if ( link == -1 )
1266     return;
1267   fs = fdopen(ds, "rw");
1268   if ( fs != NULL ) {
1269     while ( (c = fgetc(fs)) != EOF )
1270       putchar(c);
1271     fclose(fs);
1272   }
1273 }
1274
1275 /**
1276  * - use the descripter, instead of const name
1277  * &
1278  * - a program must not pass the Path in two or more functions as:
1279  * .open, stat, symlink, umount, unlink
1280  */
1281 //bad
1282 void func_7_07 (void) {
1283   FILE *fs;
1284   char *filename = "/usr/local/log.txt";
1285   char c;
1286   int ds;
1287
1288   ds = open (filename, O_RDWR);
1289   fs = fopen(filename, "rw");  // vulnerability
1290   if(fs != NULL) {
1291     while( (c = fgetc(fs)) != EOF )
1292       putchar(c);
1293     fclose(fs);
1294   }
1295 }
1296
```

```
1297 //good
1298 void func_7_08 (void) {
1299   FILE *fs;
1300   char *filename = "/usr/local/log.txt";
1301   char c;
1302   int ds;
1303   ds = open (filename, O_RDWR);
1304   fs = fdopen(ds, "rw");
1305   if ( fs != NULL ) {
1306     while ( (c = fgetc(fs)) != EOF )
1307       putchar(c);
1308    fclose(fs);
1309   }
1310 }
1311
1312 /**
1313  * - never use access() function
1314  */
1315 int func_7_09 (void) {
1316   FILE *fd;
1317   char *arq = "/bin/pwd.txt";
1318   if ( access(arq, W_OK) == 0 ) {      // vulnerability
1319     if ( (fd = fopen(arq, "w")) == NULL ) { // vulnerability
1320       perror(arq);
1321       return 0;
1322     }
1323   }
1324   // Save date
1325   return 1;
1326 }
1327
1328 int func_7_10 (void) {
1329   FILE *fd;
1330   char *arq = "/bin/pwd.txt";
1331   char *arq2 = arq;
1332   if ( access(arq, W_OK) == 0 ) {   // vulnerability
```

```
1333      if ( (fd = fopen(arq2, "w+")) == NULL ) {
1334        perror(arq);
1335        return 0;
1336      }
1337    }
1338    //Grava dados em arquivo
1339    return 1;
1340 }
1341
1342 /**
1343  * - must do chdir() after chroot()
1344  */
1345 //bad
1346 void func_7_11 (void) {
1347    FILE *f;
1348    char *file_name;      // vulnerability
1349    chroot("/home/web");    // vulnerability
1350    // Miss chdir("/")
1351    // Miss memory allocation
1352    scanf("%s", file_name);   // vulnerability - and if the file_name = ../passwd
1353    f = fopen(file_name, "w"); // vulnerability - miss chdir
1354 }
1355
1356 //good
1357 void func_7_12 (void) {
1358    FILE *f;
1359    char *file_name;      // vulnerability
1360    chroot("/home/web");//safe
1361    chdir ("/");
1362    // We got chdir("/")
1363    // Miss memory allocation
1364    scanf("%s", file_name);  // vulnerability - and if the file_name = ../passwd
1365    f = fopen(file_name, "w");
1366 }
1367
1368 /**
```

```
1369  * - create temp files in directories that are not share (! \tmp , \var\tmp)
1370  */
1371 void func_7_13 (void) {
1372    char *tempfile = "/tmp/passtmp.txt";
1373    char *tempfile2 = "/var/tmp/passtmp.txt";
1374    mktemp (tempfile);              // vulnerability
1375    mkstemp(tempfile2);              // vulnerability
1376    //(...)
1377 }
1378
1379 /**
1380  * - never use tempnam, tmpfile, tmpnam, mktemp
1381  */
1382 void func_7_14 (char *user, char *password) { //vulnerability
1383    FILE *fd;
1384    fd = tmpfile();                 // vulnerability
1385    fprintf(fd, "%s %s", user, password);
1386    // (...)
1387 }
1388
1389 /**
1390  * - before use mkstemp use umask(077)
1391  */
1392 void func_7_15 (void) {
1393    char *file = "temp.file";
1394    int err = 0;
1395    err = mkstemp(file);            // vulnerability
1396    if ( err != -1 ) {
1397            //(...)
1398    }
1399 }
1400
1401 void func_7_16 (void) {
1402    char *file = "temp.file";
1403    int err = 0;
1404    umask(077); //mask_file & 077 - group and others will be 0
```

```
1405    err = mkstemp(file);
1406    if ( err != -1 ) {
1407            // (...)
1408    }
1409 }
1410
1411 /*****************************************************************************
1412  * LEAST PRIVILEGES
1413  * func_8_**()
1414  *****************************************************************************
1415  * - system calls
1416  * . execl, execv, popen, system
1417  * - when create a child (fork), the child will have the same privileges as father
1418  *****************************************************************************/
1419 /**
1420  * -system calls
1421  */
1422 void aux_func_8_01 (void) {
1423    struct passwd *pwd;
1424    if ( (pwd = getpwuid(getuid())) == NULL )
1425      return;       // vulnerability - miss drop privilege
1426
1427    fprintf (stdout, "drop privilege fo %s", pwd->pw_name);
1428    seteuid (getuid());
1429 }
1430
1431 void func_8_01 (void) {
1432    aux_func_8_01();
1433    execl("/bin/sh", "/bin/sh", NULL);  // vulnerability
1434 }
1435
1436 void aux_func_8_02 (void) {
1437    struct passwd *passwd;
1438
1439    if ( (passwd = getpwuid(getuid())) != NULL ) {
1440      fprintf(stdout, "drop priv for %s", passwd->pw_name);
```

```
1441      setuid (getuid()); //drop privilege
1442    }
1443    execl("/bin/sh", "/bin/sh", NULL);  // vulnerability
1444 }
1445
1446 void func_8_02 (int argc, char *path, char **argv) { // vulnerability
1447    aux_func_8_02();
1448    printf("About to exec\n");
1449    execv (path, argv);          // vulnerability
1450 }
1451
1452 void func_8_03 (void) {
1453    char *syscall = "/bin/sh";
1454    //need to drop privileges
1455    system(syscall);          // vulnerability
1456 }
1457
1458 void func_8_04 (void) {
1459    char *syscall = "/bin/sh";
1460    setuid(0); //turn on as superuser
1461    system(syscall);          // vulnerability
1462 }
1463
1464 void func_8_05 (char **argv) {     // vulnerability
1465    int oldeuid = geteuid();
1466    // run possible setuid program
1467    system(argv[0]);          // vulnerability
1468
1469    // run setuid program
1470    setuid(0);
1471    system(argv[0]);          // vulnerability
1472
1473    // drop priviledges before run possible setuid program
1474    if ( seteuid(getuid()) )
1475      perror("ERROR");
1476
```

```
1477    system(argv[0]);
1478
1479    if ( seteuid(oldeuid) )
1480      perror("ERROR");
1481
1482 }
1483
1484 /**
1485  * - when create a child (fork), the child will have the same privileges as father
1486  */
1487 void func_8_06 (char **argv) {      // vulnerability
1488    // run possible setuid program
1489    switch ( fork() ) {
1490      case 0: // child
1491        execvp(argv[0], argv);        // vulnerability
1492        break;
1493          default: sleep(5); break;
1494    }
1495 }
1496
1497 void func_8_07 (char **argv) {      // vulnerability
1498    // run setuid program
1499    setuid(0);
1500    switch ( fork() ) {
1501      case 0: // child
1502        execvp(argv[0], argv);        // vulnerability
1503        break;
1504          default: sleep(5); break;
1505    }
1506 }
1507
1508 void func_8_08 (char **argv) {      // vulnerability
1509    int oldeuid = geteuid();
1510    if ( seteuid(getuid()) )
1511      perror("ERROR");
1512
```

```
1513   switch ( fork() ) {
1514     case 0: // child
1515       execvp(argv[0], argv);
1516       break;
1517         default: sleep(5); break;
1518   }
1519   if( seteuid(oldeuid) )
1520     perror("ERROR");
1521 }
1522
1523 /*************************************************************************
1524  * CRIPTOGRAFY
1525  * func_9_**()
1526  *************************************************************************
1527  * - random numbers
1528  * . get good seeds
1529  * . never used rand and random with a constant seed
1530  * . try to use /dev/(u)random
1531  *************************************************************************/
1532 /**
1533  * - Random numbers
1534  */
1535 void func_9_01 (void) {
1536   int x = rand();         // vulnerability
1537   printf("%d\n", x);
1538
1539   srand(time(0));
1540   x = rand();
1541   printf("%d\n", x);
1542
1543   srandom(time(0));
1544   x = random();
1545   printf("%d\n", x);
1546
1547   srandom(100);
1548   x = random();           // vulnerability
```

```
1549     printf("%d\n", x);
1550 }
1551
1552 /**
1553  * MAIN - Call all functions
1554  */
1555 int main(int argc, char **argv) { // vulnerability
1556
1557     // -- INITIALIZATION --
1558     func_1_01(); printf("func_1_01\n");
1559     func_1_02(); printf("func_1_02\n");
1560     func_1_03(); printf("func_1_03\n");
1561     func_1_04(); printf("func_1_04\n");
1562     func_1_05(); printf("func_1_05\n");
1563
1564     // -- CAST --
1565     func_2_01(); printf("func_2_01\n");
1566     func_2_02(); printf("func_2_02\n");
1567     func_2_03(); printf("func_2_03\n");
1568     func_2_04("func_2_04"); printf("func_2_04\n");
1569
1570     // -- PARAMETER VALIDATION --
1571     func_3_01(); printf("func_3_01\n");
1572     func_3_02(); printf("func_3_02\n");
1573     func_3_03(); printf("func_3_03\n");
1574     func_3_04(); printf("func_3_04\n");
1575     func_3_05(); printf("func_3_05\n");
1576     func_3_06(); printf("func_3_06\n");
1577     func_3_07(); printf("func_3_07\n");
1578     func_3_08(); printf("func_3_08\n");
1579     func_3_09(); printf("func_3_09\n");
1580     func_3_10(); printf("func_3_10\n");
1581     func_3_11(); printf("func_3_11\n");
1582     func_3_12(); printf("func_3_12\n");
1583     func_3_13(); printf("func_3_13\n");
1584     func_3_14(); printf("func_3_14\n");
```

```
1585    func_3_15(); printf("func_3_15\n");
1586    func_3_16(); printf("func_3_16\n");
1587    func_3_17(); printf("func_3_17\n");
1588    func_3_18(); printf("func_3_18\n");
1589    func_3_19(); printf("func_3_19\n");
1590    func_3_20(); printf("func_3_20\n");
1591    func_3_21(); printf("func_3_21\n");
1592    func_3_22(); printf("func_3_22\n");
1593    func_3_23(); printf("func_3_23\n");
1594    func_3_24(); printf("func_3_24\n");
1595    func_3_25(); printf("func_3_25\n");
1596    func_3_26(); printf("func_3_26\n");
1597    func_3_27(); printf("func_3_27\n");
1598    func_3_28(); printf("func_3_28\n");
1599    func_3_29(); printf("func_3_29\n");
1600    func_3_30(); printf("func_3_30\n");
1601  //func_3_31(0); printf("func_3_31\n");
1602    func_3_32(2); printf("func_3_32\n");
1603    func_3_33(2, 0); printf("func_3_33\n");
1604    func_3_34(2, 0); printf("func_3_34\n");
1605  //func_3_35(2); printf("func_3_35\n");
1606    func_3_36(); printf("func_3_36\n");
1607    func_3_37(); printf("func_3_37\n");
1608    func_3_38(); printf("func_3_38\n");
1609
1610    // -- RETRUN VALUES --
1611    func_4_01(); printf("func_4_01\n");
1612    func_4_02(); printf("func_4_02\n");
1613    func_4_03(); printf("func_4_03\n");
1614
1615    // -- MEMORY MANAGEMENT --
1616    func_5_01(); printf("func_5_01\n");
1617    func_5_02(); printf("func_5_02\n");
1618    func_5_03(); printf("func_5_03\n");
1619    func_5_04(); printf("func_5_04\n");
1620    func_5_05(); printf("func_5_05\n");
```

```
1621    func_5_06(); printf("func_5_06\n");
1622    func_5_07(); printf("func_5_07\n");
1623    func_5_08(); printf("func_5_08\n");
1624    func_5_09(); printf("func_5_09\n");
1625    func_5_10(); printf("func_5_10\n");
1626    func_5_11(); printf("func_5_11\n");
1627
1628    // -- MORE SAFETY FUNCTIONS WITH VALUE RANGE --
1629    func_6_01(); printf("func_6_01\n");
1630    func_6_02(); printf("func_6_02\n");
1631    func_6_03(); printf("func_6_03\n");
1632    func_6_04(); printf("func_6_04\n");
1633
1634    // -- FILE ACCESS --
1635    func_7_01(); printf("func_7_01\n");
1636    func_7_02(); printf("func_7_02\n");
1637    func_7_03("file1.txt", "file2.txt"); printf("func_7_03\n");
1638    func_7_04(); printf("func_7_04\n");
1639    func_7_05(); printf("func_7_05\n");
1640    func_7_06(); printf("func_7_06\n");
1641    func_7_07(); printf("func_7_07\n");
1642    func_7_08(); printf("func_7_08\n");
1643    func_7_09(); printf("func_7_09\n");
1644    func_7_10(); printf("func_7_10\n");
1645    func_7_11(); printf("func_7_11\n");
1646    func_7_12(); printf("func_7_12\n");
1647    func_7_13(); printf("func_7_13\n");
1648    func_7_14("3914", "At&T+EuA"); printf("func_7_14\n");
1649    func_7_15(); printf("func_7_15\n");
1650    func_7_16(); printf("func_7_16\n");
1651
1652    // -- LEAST PRIVILEGES --
1653    func_8_01(); printf("func_8_01\n");
1654    func_8_02(argc, argv[1], argv); printf("func_8_02\n");
1655    func_8_03(); printf("func_8_03\n");
1656    func_8_04(); printf("func_8_04\n");
```

```
1657    func_8_05(argv); printf("func_8_05\n");
1658    func_8_06(argv); printf("func_8_06\n");
1659    func_8_07(argv); printf("func_8_07\n");
1660    func_8_08(argv); printf("func_8_08\n");
1661
1662    // -- CRIPTOGRAFY --
1663    func_9_01(); printf("func_9_01\n");
1664
1665    // -- LIBRABRIES --
1666
1667    // -- TRY CASH EXCEPTIONS --
1668
1669    return 0;
1670 }
```

Listagem G.1: Código do teste padronizado.