Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Aplicação de Análise de Mutantes à Geração de Dados de Teste para Detecção de Vulnerabilidade do Tipo *Buffer Overflow*

Autora: Suseley Aparecida Guidetti
Orientador: Prof. Dr. Mario Jino
Co-orientador: Prof. Dr. Plínio Roberto Souza Vilela

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Mario Jino, Dr. .....DCA/FEEC/Unicamp
Edmundo Sérgio Spoto, Dr. .....UNIVEM/Marília
Marco Aurélio Amaral Henriques, Dr. .....DCA/FEEC/Unicamp
Ricardo Ribeiro Gudwin, Dr. .....DCA/FEEC/Unicamp

Campinas, SP
Fevereiro de 2005

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

G941a

Guidetti, Suseley Aparecida
Aplicação de análise de mutantes à geração de dados de teste para detecção de vulnerabilidade do tipo *Buffer Overflow* / Suseley Aparecida Guidetti. --Campinas, SP: [s.n.], 2005.

Orientadores: Mario Jino, Plínio Roberto Souza Vilela.
Dissertação (Mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Software - Testes. 2. Software - desenvolvimento. I. Jino, Mario. II. Vilela, Plínio Roberto Souza. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Título em Inglês: Application of mutation analysis to the generation of test data to detect buffer overflow vulnerability.
Palavras-chave em Inglês: Software tests e Software development.
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Edmundo Sérgio Spoto, Marco Aurélio Amaral Henriques e Ricardo Ribeiro Gudwin.
Data da defesa: 23/02/2005

Resumo

O crescimento acentuado da Internet e a conseqüente demanda por novos serviços levam ao desenvolvimento de aplicações cada vez mais complexas. Devido a esta complexidade e às pressões para o cumprimento de cronogramas cada vez mais restritivos, muitas vezes a segurança desses sistemas acaba sendo negligenciada, originando aplicações com mais vulnerabilidades e portanto sujeitas a um maior número de ataques. Um dos ataques mais freqüentes é o do tipo *buffer overflow*. Permite que um atacante insira códigos maliciosos em um programa, alterando seu fluxo de controle original, com o objetivo de conseguir acesso a um sistema ou aumentar seus privilégios. Neste trabalho foi utilizada a técnica denominada "Teste de Vulnerabilidade de Segurança" proposta para detectar vulnerabilidade do tipo *buffer overflow* no software por meio da aplicação do Teste de Mutação. Uma ferramenta chamada SEVMUT - ***Se**curity **V**ulnerabilities **Mu**tation **T**ool* foi desenvolvida e sua aplicabilidade, escalabilidade e eficácia foram validadas.

Palavras-Chave: vulnerabilidade de segurança, *buffer overflow*, análise de mutantes.

Abstract

The steep growth of the Internet and the associated demand for new products and services leads to the development of applications of increasing complexity. Due to this complexity and the pressure to meet squeezing schedules, security issues are left as an afterthought, increasing the number of vulnerabilities in the systems and making them susceptible to attacks. One of the most frequent attacks is the buffer overflow. It allows a hacker to insert malicious code in a program, changing its original flow of control with the goal of escalating privileges and gaining access into de system. In this work we present a technique called "Testing for Security Vulnerabilities" meant to detect buffer overflow vulnerabilities in software through the use of Mutation Testing. A tool called SEVMUT – **Se**curity **V**ulnerabilities **Mu**tation **T**ool was developed and validated for its applicability, scalability and effectiveness.

Keywords: security vulnerability, buffer overflow, mutant analysis.

À memória de Célia Simões Gouveia

Agradecimentos

Primeiramente agradeço àquele que está acima de tudo e de todos, **Deus**.

Agradeço a "D. Cida" por ter me ajudado durante todo este tempo, sempre pronta para conversar, confortando-me nas horas de desânimo e fazendo-me acreditar que eu conseguiria.

Ao meu orientador, Mario Jino e ao co-orientador, Plínio Vilela pela oportunidade de trabalho, ajudando-me e orientando-me na sua condução e principalmente pela disponibilidade constante.

Ao João Luís, pela paciência, compreensão, respeito e admiração pelo meu trabalho, sem nunca ter me cobrado nada. Obrigada pelo incentivo e por ter acreditado que eu conseguiria. Obrigada pelo suporte técnico e pela revisão desta dissertação.

Agradeço a minha filha Mariana, pelo seu respeito e compreensão e por ter tentado me ajudar de alguma forma para que eu pudesse realizar este trabalho, mesmo tendo os seus anseios negligenciados algumas vezes.

A minha mãe, D. Geni, por ter me substituído muitas vezes em minha casa, mesmo estando algumas vezes doente, liberando-me dos afazeres domésticos e desta forma possibilitando uma maior dedicação a este trabalho.

Agradeço ao Paulinho pela ajuda em relação à execução da Ferramenta Proteum e por ter me ajudado com a geração aleatória de dados de teste, agilizando o meu trabalho.

Agradeço ao pessoal de teste de software e a todos aqueles que contribuíram de forma direta ou indireta para realização deste trabalho.

Agradeço à CAPES pelo apoio financeiro.

Agradeço aos professores Edmundo Sérgio Spoto, Marco Aurélio Amaral Henriques e Ricardo Ribeiro Gudwin, pelas contribuições à dissertação.

Sumário

ANEXOS

Lista de Figuras

Lista de Tabelas

“Duas estradas separam-se numa floresta, e eu tomei a menos utilizada. E isto fez toda a diferença!”.

Robert Frost (1874-1963): ‘The Road Not Taken’ (1916).

Capítulo 1

Introdução

1.1 Contexto

A Internet, hoje, é acessível a qualquer indivíduo com um computador e uma conexão de rede, independentemente da sua posição geográfica. O crescimento explosivo da Internet deve-se à facilidade de conexão por meio de banda larga e ao oferecimento de serviços através de sistemas ou provedores, atraindo usuários de todo mundo. Vinton Cerf, presente no Congresso Internacional de Tecnologia de Informação e Comunicações - ITU World 2003 - em Genebra, Suíça, previu que em 2010 um terço da população mundial - mais de 2 bilhões de usuários - estarão conectados à Internet [Fau03].

Com o aumento da conectividade e com o advento da banda larga, aumenta a demanda por aplicações distribuídas e serviços *on-line*. No final do ano de 2002 havia 580 milhões de usuários de Internet; destes, 63 milhões utilizavam a banda larga. O instituto de pesquisa Ovum da Inglaterra prevê que em 2006 haverá mais de 300 milhões de usuários de banda larga em todo o mundo [Fau03].

A demanda cada vez maior por serviços leva ao desenvolvimento de aplicações cada vez mais complexas. Devido à complexidade, muitas vezes a segurança não é incluída na descrição inicial do software, quando este é especificado e projetado. As preocupações surgem quando o sistema entra em operação e vulnerabilidades são reportadas. Neste momento correções de última hora são realizadas e distribuídas, gerando custos adicionais, conseqüentemente abalando a credibilidade do software e da empresa que o desenvolveu.

Alguns profissionais desconsideram questões de segurança na fase de desenvolvimento do software devido a sua falta de entendimento sobre o assunto. Outros, devido às pressões existentes para o desenvolvimento em tempo reduzido, são forçados a ignorar algumas etapas que certamente reduziriam os problemas relacionados a vulnerabilidades de segurança do sistema. Essa displicência com os aspectos de segurança pode levar o software a se comportar de maneira inesperada, aumentando o número de vulnerabilidades que podem ser exploradas por indivíduos com intenções maliciosas.

Em conseqüência desse panorama, vulnerabilidades de segurança têm se tornado onipresentes nos softwares. De acordo com o CERT[1], em 2004 foram reportadas mais de 2800 vulnerabilidades de segurança. Erros de programação, de configuração e operacionais têm permitido a usuários não autorizados entrarem nos sistemas, ou a usuários autorizados atuarem

[1] CERT – *Computer Emergency Response Team* – http://www.cert.org

de forma não permitida. Esforços para eliminar tais problemas têm levado algumas vezes ao agravamento de tais situações.

Diante disto, técnicas e ferramentas que previnem a ocorrência de vulnerabilidades no software ou que evitam ataques aos sistemas têm se tornado cada vez mais imprescindíveis.

1.2 Motivação

Entre os diversos ataques ocorridos nos sistemas, encontra-se o do tipo *buffer overflow* descrito na seção 2.4. De acordo com Wilander [Wil03], cinqüenta por cento das vulnerabilidades reportadas pelo CERT em 2003 foram do tipo *buffer overflow*.

Embora muitas técnicas e ferramentas tenham sido propostas para detecção e prevenção deste tipo de ataque, o *buffer overflow* continua sendo um dos ataques que mais freqüentemente ocorre nos sistemas atualmente.

Muitos dos defeitos presentes nos softwares entregues ao cliente são decorrentes de uma atividade de teste aquém das necessidades. Acelerar o processo de desenvolvimento e subestimar a etapa de teste cria, em geral, uma sensação de uma margem maior de lucro com a comercialização do software. Essa postura implica na presença cada vez maior de defeitos nos sistemas e conseqüentemente em mais vulnerabilidades de segurança.

Como o ataque do tipo *buffer overflow* acontece através da exploração de defeitos existentes no código fonte, a eliminação de tais defeitos antes de o software entrar em operação, ameniza o problema de segurança destes sistemas.

Dessa forma, é interessante desenvolver uma ferramenta que ajude o testador a detectar os defeitos no software que podem ser explorados para a realização de ataques, durante a fase de teste no desenvolvimento do sistema.

A utilização de Análise de Mutantes, uma técnica de teste baseada em defeitos, pode auxiliar na identificação da presença de tais vulnerabilidades, possibilitando que sejam eliminadas e conseqüentemente aumentando a segurança dos softwares em relação aos ataques.

1.3 Objetivos

Os principais objetivos deste trabalho são:

- o estudo, a implementação e a validação de uma técnica que, ao ser aplicada, auxilie na detecção da presença de vulnerabilidades através da utilização do critério Análise de Mutantes;

- o estudo das vulnerabilidades de software que podem ser modeladas com operadores de mutação e a análise e extensão de um conjunto de operadores para aplicação específica em segurança;
- a implementação de uma ferramenta que valide a técnica aplicando os operadores de mutação ao programa sendo testado, possibilitando ao testador identificar uma possível presença de vulnerabilidade no software, desenvolvida no sistema operacional Linux distribuição Red Hat 9.0 utilizando a linguagem C e a linguagem do s*hell bash* do Linux na sua implementação;
- a condução de experimentos que validem a ferramenta desenvolvida com o objetivo de verificar a sua escalabilidade[2], avaliando a seu desempenho por meio da utilização do tempo de geração dos mutantes para programas de tamanhos variados. Sua aplicabilidade[3] será verificada, pois desde que exista algum tipo de vulnerabilidade que possa ser tratada com a técnica, a mesma pode ser considerada aplicável;
- a condução de experimentos que validem a eficácia da técnica em relação a sua habilidade em detectar a presença de vulnerabilidade através de um estudo de caso.

1.4 Considerações

Assume-se que, neste contexto, as expressões a seguir têm os seguintes significados:

1. DETECTAR A PRESENÇA DE VULNERABILIDADE DO TIPO *BUFFER OVERFLOW*.
   Considera-se que DETECTAR A PRESENÇA DE VULNERABILIDADE DO TIPO *BUFFER OVERFLOW* é dar a chance ao testador de descobrir que existe uma vulnerabilidade no software. Não significa indicar qual é a vulnerabilidade ou onde esta se encontra dentro da aplicação.

2. VULNERABILIDADE DO TIPO *BUFFER OVERFLOW*.
   Considera-se que se um espaço de memória pode sofrer transbordamento então existe uma VULNERABILIDADE DO TIPO *BUFFER OVERFLOW* que pode ou não ser explorada por um atacante. Não é objeto deste trabalho detectar a exploração em si, ou seja, o ataque.

1.5 Organização do Trabalho

Neste capítulo foram apresentados o contexto do trabalho, as motivações que justificaram o seu desenvolvimento e os seus principais objetivos. Os demais capítulos encontram-se estruturados da seguinte maneira:

O CAPÍTULO 2 apresenta os principais conceitos relacionados ao teste de software descrevendo a técnica de teste estrutural, a técnica funcional e a técnica baseada em defeitos,

[2] Uma ferramenta com boa escalabilidade mantém seu desempenho independentemente do tamanho do programa testado.
[3] Aplicabilidade é a capacidade do que pode ser aplicado [Fer04].

destacando o Critério Análise de Mutantes, cujo entendimento é importante para a compreensão da técnica descrita no Capítulo 3 dessa dissertação. Nesse capítulo são apresentados os principais problemas encontrados atualmente no desenvolvimento de software em relação à segurança, destacando a vulnerabilidade do tipo *buffer overflow,* que pode ser explorada por *hackers*[4] para realizarem ataques aos sistemas.

O CAPÍTULO 3 apresenta a técnica implementada neste trabalho denominada Teste de Vulnerabilidade de Segurança , destacando como a Análise de Mutantes pode ser aplicada à detecção da presença de vulnerabilidade no software através da aplicação de operadores de mutação. Destaca as características mínimas que as vulnerabilidades devem possuir para serem candidatas à aplicação da técnica e o motivo do *buffer overflow* ser um candidato em potencial.

O CAPÍTULO 4 apresenta a Ferramenta SEVMUT– ***Se**curity **V**ulnerabilities **Mu**tation **T**ool*, desenvolvida com o objetivo de validar a técnica apresentada no Capítulo 3, mostrando sua arquitetura e destacando as principais funções implementadas pela ferramenta com a finalidade de aplicar o critério Análise de Mutante à detecção de vulnerabilidade do tipo *buffer overflow*. Nesse capítulo apresenta-se também a condução de um experimento, realizado com o objetivo de validar a Ferramenta SEVMUT, através da aplicação da técnica descrita no CAPÍTULO 3 a vários programas, verificando se o comportamento da SEVMUT em relação ao tempo de geração dos mutantes ocorre de forma escalar, dependendo do tamanho dos programas e da quantidade de alocações de memória. Este experimento tem como principal objetivo mostrar a escalabilidade e a aplicabilidade da ferramenta.

No CAPÍTULO 5 apresenta-se a condução de um segundo experimento realizado com o objetivo de verificar a eficácia da técnica descrita no Capítulo 3, em relação a sua habilidade de detectar a presença de vulnerabilidade no software, apontando quais resultados na execução dos mutantes indicam ao testador uma possível presença de vulnerabilidade do tipo *buffer overflow* na aplicação.

No CAPÍTULO 6 apresentam-se as conclusões e contribuições desta dissertação e a indicação de possíveis extensões desse trabalho.

[4] *Hacker* é alguém que, sem ser convidado, tenta quebrar a segurança do sistema ganhando acesso.

Capítulo 2

Revisão Bibliográfica

Neste Capítulo são apresentados os principais conceitos relacionados ao Teste de Software, a descrição do teste funcional, do teste estrutural e o detalhamento do Critério Análise de Mutantes, um critério associado à técnica de teste baseado em defeitos.

Apresentam-se também as principais preocupações que envolvem o desenvolvimento do software em relação a sua segurança, descrevendo alguns dos ataques mais comuns que ocorrem atualmente nos sistemas e detalhando o ataque do tipo *buffer overflow*.

São também apresentados alguns dos trabalhos relacionados com ataques do tipo *buffer overflow* que, na maioria das vezes, ocorrem através da exploração de vulnerabilidades existentes em programas escritos na linguagem de programação C.

2.1 Teste de Software

O objetivo principal do teste é demonstrar que o programa tem defeitos. De acordo com Myers [Mye79], teste é o processo de executar um programa com a intenção de descobrir defeitos. Deve ser visto como um processo destrutivo, ao contrário dos demais passos da Engenharia de Software, nos quais os processos são direcionados para a construção do sistema.

Os CASOS DE TESTE, utilizados no teste de um programa, são gerados com o objetivo de atender um critério associado a alguma técnica de teste de software. Basicamente um caso de teste é formado [Mal91]:

- por um dado de entrada, que faz parte dos domínios de entrada válido e inválido do programa; e
- pela resposta esperada na execução do programa com o dado de entrada de acordo com a sua especificação.

Um bom caso de teste é aquele que revela um defeito ainda não descoberto. Se a saída produzida na execução do programa com o dado de entrada for diferente da resposta esperada significa que o caso de teste revelou a presença de um defeito no programa em teste.

A tarefa de analisar os resultados das execuções do programa identificando a ocorrência de uma falha fica por conta do oráculo[5], que na maioria das vezes é o próprio testador, podendo esta tarefa ser automatizada.

Segundo Pressman [Pre04] a medida em que os resultados de teste são reunidos e avaliados, uma indicação da qualidade e da confiabilidade do software começa a surgir. O Teste de Software é um procedimento crítico para a qualidade do software, representando a última atividade na qual é possível revelar defeitos na especificação, no projeto e na codificação do software.

Os CRITÉRIOS DE TESTE, associados às técnicas de teste, têm sido utilizados com o objetivo de escolher bons dados de teste que aumentem a probabilidade da revelação de defeitos.

De acordo com Bueno [Bue99], os CRITÉRIOS DE TESTE estabelecem requisitos a serem satisfeitos e que podem orientar:

- a seleção de dados de teste;
- a avaliação da qualidade do teste; e
- a definição de requisitos de suficiência a serem atingidos para o encerramento da atividade de teste.

Satisfazer um critério de teste significa satisfazer todos os requisitos de teste estabelecidos pelo critério.

De acordo com [IEE90], uma FALHA (*failure*) ocorre quando uma saída incorreta é produzida em relação à especificação do programa. Um DEFEITO *(fault)* é uma deficiência no programa que pode provocar uma saída incorreta para algum dado de entrada. Um ERRO (*error*) é uma diferença existente entre o valor correto e o valor esperado para alguma variável na execução do programa, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa. Um ENGANO (*mistake)* é uma ação humana que produz um resultado incorreto, por exemplo, uma entrada incorreta em um programa.

Basicamente, pode-se associar um critério de teste a três técnicas principais de teste de software: FUNCIONAL, ESTRUTURAL e BASEADA EM DEFEITO. Cada uma destas técnicas será apresentada nas subseções seguintes. O que diferencia essas três técnicas é o tipo de informação utilizada para derivar os requisitos de teste de cada critério.

2.1.1 Técnica Funcional

Os critérios funcionais associados a esta técnica também chamada de caixa preta, estabelecem casos de teste baseados na especificação do programa. Estes critérios não levam em consideração os detalhes de implementação do software, preocupando-se somente com os aspectos funcionais do sistema.

[5] A função do oráculo, através da saída do programa, é dizer apenas se o resultado está correto ou não. Ele não tem como função indicar o resultado correto.

Um critério de teste funcional é o PARTICIONAMENTO DE EQUIVALÊNCIA [Pre04]. Este critério divide o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste são derivados. Um caso de teste ideal revela por si só uma classe de defeitos que, de outro modo, poderia exigir que muitos casos de teste tivessem de ser executados antes que o defeito geral fosse detectado. Desta forma, seleciona-se apenas um subconjunto do domínio de entrada, reduzindo com isto o número total de casos de teste que devem ser desenvolvidos para testar o sistema.

Outro exemplo de critério funcional é a ANÁLISE DE VALOR LIMITE [Pre04]. Este critério complementa o critério PARTICIONAMENTO DE EQUIVALÊNCIA, pois deriva os casos de teste das fronteiras das classes de equivalência. Isto ocorre pelo fato de um número maior de defeitos ocorrer nas extremidades e não no meio do domínio de entrada. Este critério deriva os casos de teste também do domínio de saída do programa, dividindo-o em classes e exigindo casos de teste que gerem resultados dentro destas classes.

O GRAFO DE CAUSA-EFEITO é um critério funcional que obtém as possíveis condições de entrada (CAUSAS) e as possíveis ações (EFEITOS) do programa, utilizando estas informações para a construção de um grafo, que é então convertido em uma tabela de decisão utilizada na construção dos casos de teste [Pre04].

2.1.2 Técnica Estrutural

Os critérios associados à técnica estrutural levam em consideração os detalhes procedimentais do programa em teste.

Segundo Bueno [Bue99], "os critérios de teste estrutural estabelecem componentes estruturais do programa a serem exercitados. Satisfazer um critério de teste estrutural significa exercitar todos os componentes requeridos pelo critério".

Os critérios baseados no FLUXO DE CONTROLE usam as características de controle da execução do programa para determinar quais estruturas são requeridas. Estes critérios utilizam uma representação gráfica do programa chamada GRAFO DE FLUXO DE CONTROLE ou GRAFO DE PROGRAMA [Rap85] para determinar as estruturas que devem ser cobertas pelos casos de teste.

Resumidamente, um programa P é decomposto em um conjunto de BLOCOS disjuntos de comandos; a execução do primeiro comando do bloco acarreta a execução de todos os outros, na ordem dada, desse bloco. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor; e exatamente um único sucessor, exceto possivelmente o último comando. Neste grafo os NÓS correspondem aos blocos e os ARCOS representam os possíveis fluxos de controle entre os blocos. Um arco (n_i, n_j) é chamado de RAMO se o último comando de n_i é um comando de seleção ou repetição. A cada ramo pode ser associado um predicado denominado PREDICADO DE RAMO. Um SUB-CAMINHO é uma seqüência de nós de

ordem ≥ 2. Um CAMINHO ou CAMINHO COMPLETO é um sub-caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo de programa.

Os critérios de teste TODOS OS NÓS, TODOS OS RAMOS e TODOS OS CAMINHOS são alguns exemplos de critérios baseados no fluxo de controle de um programa. Estes critérios exigem que cada nó, cada ramo e cada caminho do grafo de programa sejam executados pelo menos uma vez.

Os critérios baseados no FLUXO DE DADOS usam informações do fluxo de dados do programa para derivar os requisitos de teste. Estes critérios analisam o uso das variáveis, determinando associações entre os pontos onde um valor é atribuído a uma variável, chamados de DEFINIÇÃO da variável e os pontos onde esse valor é utilizado, chamados de USO da variável. Estes critérios exigem a execução do caminho do ponto onde a variável foi definida até o ponto onde foi utilizada sem passar por uma redefinição.

Os critérios TODAS-DEFINIÇÕES, TODOS-USOS e TODOS-DU-CAMINHOS [Rap85] são alguns exemplos de critérios baseados no fluxo de dados de um programa.

O critério TODAS-DEFINIÇÕES requer que cada definição de variável seja exercitada pelo menos por um uso dessa variável.

O critério TODOS-USOS requer que todas as associações entre uma definição de variável e todos os usos da mesma sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida.

O critério TODOS-DU-CAMINHOS requer que toda associação entre uma definição e um uso de uma variável seja exercitada por todos os caminhos livres de laços e de redefinições que cubram essa associação.

2.1.3 Técnica Baseada em Defeitos

Os critérios associados à TÉCNICA BASEADA EM DEFEITOS, também denominada TÉCNICA BASEADA EM ERROS, utilizam informações sobre os defeitos freqüentemente introduzidos pelo programador durante a codificação do software na fase de desenvolvimento do sistema.

O CRITÉRIO ANÁLISE DE MUTANTES é um exemplo de critério baseado em defeitos do software e será descrito com detalhes na próxima seção, por ser de interesse para o contexto dessa dissertação.

2.2 Análise de Mutantes

A ANÁLISE DE MUTANTES ou TESTE DE MUTAÇÃO foi originalmente proposta por DeMillo [DeM78], sendo descrita a seguir.

Denomina-se P o programa em teste e D é o seu domínio de entrada. Um conjunto de dados de teste $T \subseteq D$ consiste de um ou mais dados de teste. Cada dado de teste denominado $t \in T$ é fornecido pelo testador para avaliar a qualidade de P. P é executado com T e caso algum dos resultados indique alguma falha, é corrigido o defeito revelado em P(t).

P pode conter defeitos que T não consegue revelar, então são criados programas mutantes de P. Seja m um programa sintaticamente correto obtido através de uma única mudança sintática em P. O programa m é considerado um mutante de P. Seja r uma regra que mudará P; r é conhecida como um operador de mutação. A cada aplicação do operador de mutação a P é gerado um novo programa mutante m. Sendo assim, a aplicação de um conjunto de operadores R a P resulta em um conjunto de mutantes $m_1, m_2,...,m_n$, onde $n \geq 1$. Cada mutante tem apenas uma modificação sintática em relação a P.

Através da variação de R pode-se obter uma variedade de critérios baseados em mutação. Entretanto, para manter o teste de mutação prático, o conjunto de operadores de mutação deve ser mantido pequeno e consistindo apenas de operadores simples. As modificações sintáticas são projetadas para modelar defeitos reais.

Considere, por exemplo, um operador de mutação r que gere mutantes de P substituindo o uso da variável x por x + 1 e x - 1. Na aplicação de r a P, para uma instrução igual a z = x + y; são gerados dois mutantes de P, um obtido através da substituição da instrução por z = x + 1 + y; e o outro pela substituição por z = x - 1 + y;. A Figura 2.1 ilustra este exemplo.

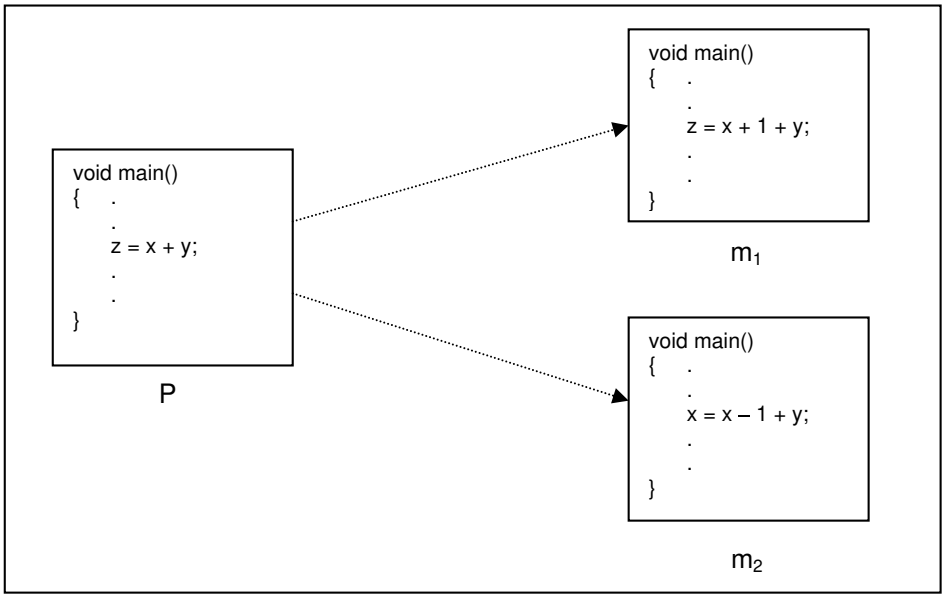

Figura 2.1 – Exemplo de Geração de Mutantes

Após a geração dos mutantes, estes são executados utilizando-se o conjunto de dados de teste T como entrada.

Quando executado com um dado de teste t, o mutante m_i é considerado distinto de P se $P(t) \neq m_i(t)$. Um dado de teste que é capaz de distinguir um mutante do programa original é aquele que produz na execução do mutante, um resultado diferente do resultado produzido, na execução do programa original. Quando isto ocorre o mutante é considerado MORTO, pois conseguiu identificar o defeito no mutante revelando a diferença entre P e m_i, sendo eliminado da lista dos mutantes que ainda precisam ser distinguidos.

Para ser capaz de matar o mutante m_i, um dado de teste $t \in T$ deve forçar o fluxo de controle de m_i a alcançar o ponto onde a mutação ocorreu. A execução da instrução modificada deve levá-lo a um estado diferente do produzido por P sob as mesmas circunstâncias.

Entretanto, se m_i apresentar respostas idênticas a P diz-se que m_i continua VIVO. Isto pode ocorrer por dois motivos: ou T não contém dados de teste capazes de distinguir m_i de P ou ambos executam as mesmas funções, ou seja, são EQUIVALENTES. O mutante m_i é considerado EQUIVALENTE a P se para todo $t \in T$, $P(t) = m_i(t)$. Sendo assim, nenhum dado de teste conseguirá distingui-lo de P e, portanto nunca será MORTO. Apesar dos avanços em técnicas e ferramentas automáticas para detectar mutantes equivalentes, esta atividade é, na maioria das vezes, feita manualmente, o que aumenta o custo da aplicação da técnica.

No caso do mutante m_i permanecer VIVO após a sua execução com o conjunto T, novos dados de teste devem ser adicionados a T para tentar eliminar o mutante m_i.

O objetivo do testador ao aplicar análise de mutantes é encontrar um conjunto de dados de teste T que consiga matar todos os mutantes m_i não equivalentes, sendo este conjunto T adequado para o teste do programa P.

O ESCORE DE MUTAÇÃO é a medida de cobertura do teste baseado em defeitos, indicando a qualidade do conjunto de dados de teste utilizado. Por meio do escore de mutação que relaciona o número de mutantes gerados com o número de mutantes mortos, pode-se avaliar a adequação dos dados usados, ou seja, a qualidade dos dados de teste.

O ESCORE DE MUTAÇÃO varia entre 0 e 1. Dado o programa P e o conjunto de dados de teste T, calcula-se o escore de mutação EscMut(P,T) da seguinte forma:

$$\text{EscMut}(P,T) = \frac{K}{M - E}$$

onde:

P : programa em teste,
T : conjunto de dados de teste,
K : quantidade de mutantes mortos,
M : total de mutantes gerados,
E : total de mutantes equivalentes.

A Figura 2.2 ilustra o processo de aplicação do Teste de Mutação tradicional. De acordo com Harman [Har00], existem três formas de teste de mutação: a FORTE, a FRACA e a FIRME.

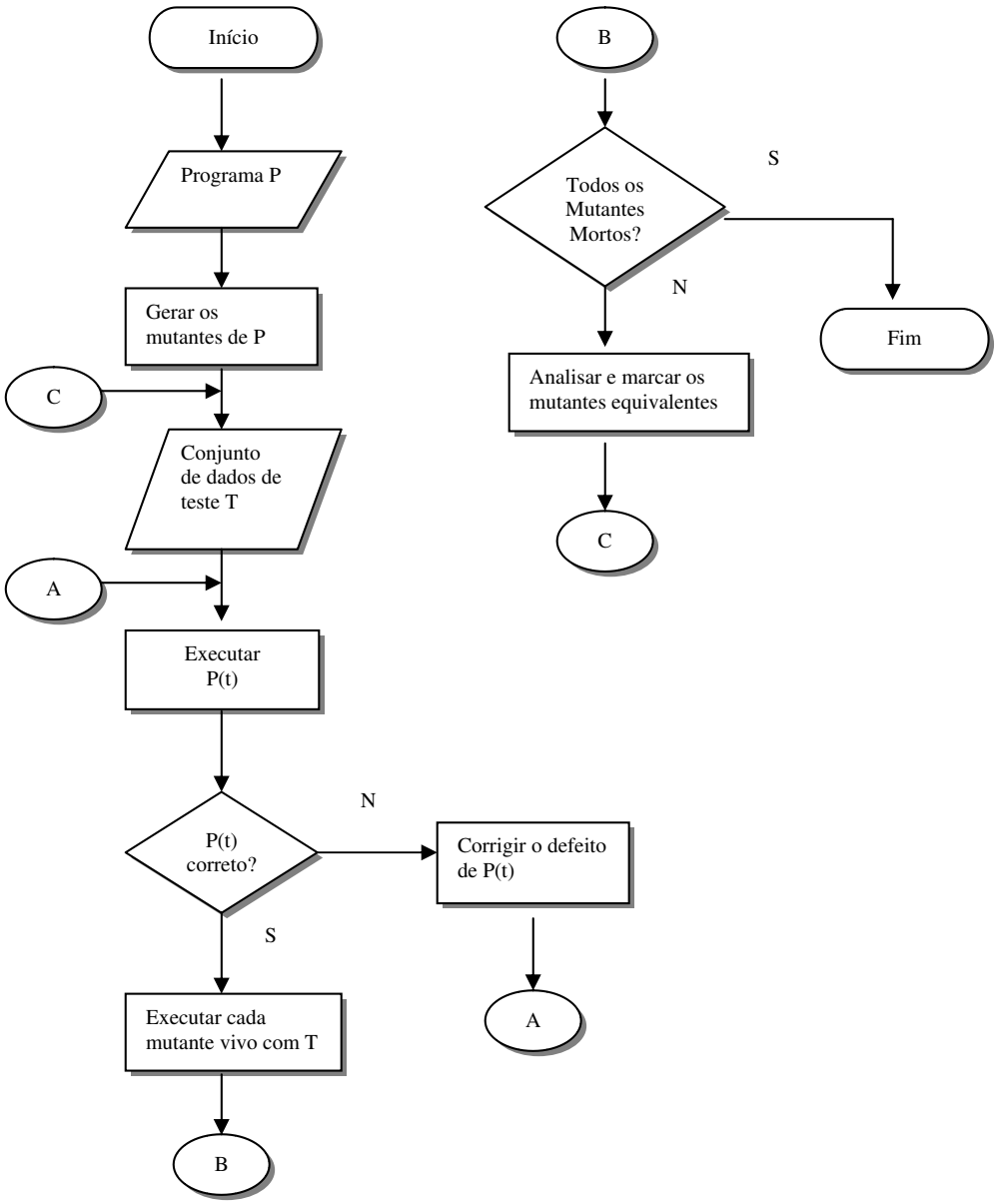

Figura 2.2 – Diagrama do Processo de Análise de Mutantes

O CRITÉRIO MUTAÇÃO FRACA considera o programa P formado de vários componentes, nos quais será aplicado o operador de mutação transformando P em m_i. Um componente pode ser uma expressão aritmética, uma referência a uma variável ou outros.

Este critério requer que um dado de teste $t \in T$ seja construído com a propriedade de que o componente modificado em m_i e o correspondente não modificado em P, sejam exercitados com t produzindo resultados diferentes. A comparação dos resultados deve ocorrer imediatamente após a execução do componente em P e em m_i.

A desvantagem da MUTAÇÃO FRACA é que mesmo que os resultados sejam diferentes entre P e m_i após a execução do componente modificado, as saídas de P e m_i podem ser as mesmas no final da execução dos programas. Desta forma, o conjunto de dados de teste T adequado à mutação fraca pode não ser adequado à Análise de Mutantes tradicional.

No CRITÉRIO MUTAÇÃO FORTE ou CRITÉRIO ANÁLISE DE MUTANTES, a avaliação dos resultados obtidos ocorre no final da execução dos programas P e m_i, como já explicado anteriormente.

A mutação fraca e a mutação forte são casos especiais do CRITÉRIO MUTAÇÃO FIRME. Neste critério são introduzidos no programa pontos onde as alterações são efetuadas e pontos onde os estados do programa original e do mutante são comparados, antes do final da execução, com isto diminuindo o custo da execução dos mutantes, pois apenas execuções parciais são necessárias para decidir se um mutante está vivo ou morto.

2.3 Segurança

Nesta seção, descrevem-se as principais preocupações que estão atualmente relacionadas com o desenvolvimento do software em relação a sua segurança e alguns dos ataques mais comuns que ocorrem nos sistemas.

O desenvolvimento de sistemas é um dos pontos críticos para a garantia de segurança do software. Um processo de desenvolvimento bem fundamentado e estruturado, com atenção específica aos aspectos de segurança, aliado ao uso de ferramentas adequadas e seguras, é a melhor arma que se tem contra os prejuízos decorrentes de falhas de segurança em software. Entretanto, nenhum processo garantirá a eliminação de todos os defeitos de desenvolvimento e nem a produção de um software totalmente seguro [Alb02].

Existem vários aspectos de segurança que devem ser considerados no desenvolvimento de sistemas, sendo os três mais importantes: a CONFIDENCIALIDADE, a INTEGRIDADE e a DISPONIBILIDADE da informação [Alb02].

O aspecto de segurança referente a CONFIDENCIALIDADE refere-se à capacidade de um sistema impedir que usuários não-autorizados vejam determinada informação. O aspecto INTEGRIDADE da informação é a capacidade do sistema detectar ou impedir que uma informação seja alterada sem autorização. O aspecto de segurança referente à DISPONIBILIDADE da informação é a capacidade do sistema impedir que uma informação seja apagada ou se torne inacessível a usuários autorizados.

Além destes três aspectos, existem outros tais como: AUTENTICAÇÃO, NÃO REPÚDIO e AUDITORIA. AUTENTICAÇÃO representa a capacidade de garantir que um usuário, sistema ou informação é mesmo quem alega ser; NÃO REPÚDIO é a capacidade do sistema de provar que um usuário executou determinada ação no sistema e AUDITORIA é a capacidade do sistema de analisar todas as tarefas realizadas pelos usuários, detectando fraudes ou tentativas de ataque.

Problemas de segurança são ocasionados pela perda de qualquer um destes e de outros aspectos não citados que são importantes para o sistema. Ataque ao sistema é um tipo de problema de segurança sério, pois o atacante objetiva obter algum retorno, podendo provocar grandes danos.

Segundo Albuquerque [Alb02], existem três preocupações básicas, que são complementares, em relação à segurança no desenvolvimento de software:

- **Segurança do ambiente de desenvolvimento**: manter os códigos-fonte seguros evitando que sejam roubados;
- **Segurança da aplicação desenvolvida**: desenvolver uma aplicação que seja segura, seguindo corretamente a especificação de segurança e que não contenha acessos ocultos (*backdoors*[6]), códigos maliciosos ou falhas que comprometam a segurança;
- **Garantia de segurança da aplicação desenvolvida**: garantir a segurança da aplicação em desenvolvimento através de testes de garantia de segurança.

Um sistema que consegue se proteger de um grande número de ataques, provavelmente dificultará uma série de outros, pois vários deles utilizam o mesmo mecanismo para invadir um sistema. Alguns dos principais ataques que ocorrem nos sistemas são: *buffer overflow*, *denial of service* e *cross-site scripting*.

O ataque *buffer overflow* explora a falta da verificação da informação recebida de fora do sistema, levando-o a executar tarefas não previstas na sua especificação. Este ataque é detalhado na seção 2.4.

O ataque *denial of service* tem como objetivo evitar que usuários legítimos do sistema tenham acesso aos serviços. Isto é obtido através do envio de grandes volumes de dados ao sistema, consumindo de forma deliberada todos os seus recursos [For01].

O ataque *cross-site scripting* tem como objetivo o *browser* do usuário. É a habilidade de inserir *scripts* maliciosos dentro de páginas *Web*. Os *scripts* são disfarçados como dados

[6] *Backdoor* é um ponto dentro de um programa que permite alguém ganhar acesso ao sistema sem ter que passar pelos procedimentos usuais.

legítimos e executados no *browser* do usuário resultando no comprometimento de informações confidenciais [For01].

Todos estes ataques ocorrem devido a vulnerabilidades existentes nas aplicações. De acordo com Vilela [Vil02], VULNERABILIDADE DE SEGURANÇA é definida como um problema técnico, manifestado em um defeito no software, que possibilita o uso ou acesso não autorizado a recursos do sistema. A vulnerabilidade só se constitui como tal se existir uma relação direta entre o problema técnico e o uso ou acesso não autorizado e se houver relato de pelo menos uma exploração em particular.

Existem disponíveis vários pacotes de software contendo ferramentas fáceis de usar e utilizadas pelos *hackers* para explorar as vulnerabilidades de um sistema, tornando o ataque mais efetivo. Estes pacotes contêm ferramentas do tipo *password cracking tools*[7]*, packet sniffers*[8], ferramentas que modificam os arquivos log do sistema e ferramentas que modificam os arquivos de configuração.

Na próxima seção descreve-se em detalhes como ocorre um ataque do tipo *buffer overflow*.

2.4 *Buffer Overflow*

Um *buffer overflow* ocorre quando o programador armazena uma informação em uma área de memória de tamanho insuficiente para recebê-la. Isto leva à sobrescrita das posições de memória adjacentes à área reservada, conseqüentemente modificando os seus conteúdos.

Um *buffer overflow* pode ser explorado por *hackers,* resultando em vários tipos de ataques de *bufer overflow,* que exploram esta vulnerabilidade do sistema.

Para uma melhor compreensão de como o *buffer overflow* pode ser explorado, primeiramente será apresentado como um processo no Linux é mapeado na memória principal.

2.4.1 *Layout* Básico da Memória do Linux

De acordo com Giasson [Gia01], cada processo no Linux tem o *layout* parcial de memória ilustrado na Figura 2.3.

Os argumentos de entrada do processo e as variáveis de ambiente ficam armazenadas no endereço mais alto da memória. Em seguida situa-se a seção do *stack frame* composta pelo *stack* (pilha) e pela *heap*. A pilha do processo é localizada no endereço de memória mais alto e a *heap,* no endereço mais baixo dentro do *stack frame*. Na pilha, são armazenadas as variáveis que são declaradas localmente e na memória *heap*, as variáveis declaradas dinamicamente. O

[7] *Password cracking tool* é uma ferramenta utilizada para quebrar senhas.
[8] *Packet sniffer* é uma ferramenta utilizada para capturar os pacotes que trafegam pela rede.

crescimento da pilha ocorre do endereço de memória mais alto para o endereço mais baixo e o crescimento da *heap* é o oposto, do endereço mais baixo para o mais alto.

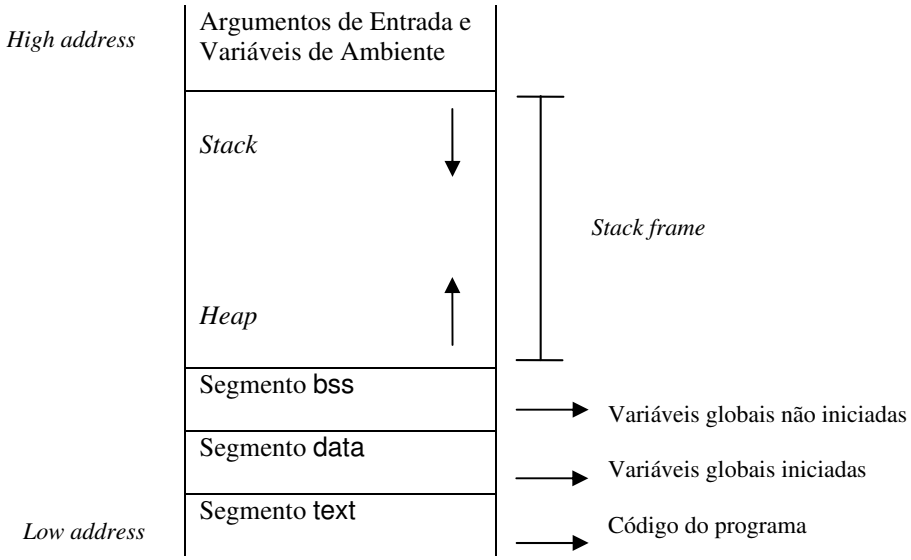

Figura 2.3 – *Layout* da Memória de um Processo Linux

No segmento bss[9] localizam-se as variáveis globais do processo que têm apenas um nome e um tamanho, mas nenhum valor. No segmento data estão as variáveis globais que inicialmente têm um valor e na seção text está o código de máquina do processo.

Toda chamada de função dentro de um processo leva à criação de um novo ambiente chamado de *stack frame,* criado no topo da pilha, como o da estrutura mostrada na Figura 2.4.

No novo *stack frame* são armazenados os parâmetros de entrada da função, o endereço da próxima instrução a ser executada após o retorno para a função que a chamou, o endereço base da função que a chamou e as variáveis declaradas localmente na função chamada. Apesar do crescimento da pilha ocorrer do *high address* para o *low address*, a atribuição dos valores às variáveis locais ocorre do *low address* para o *high address.*

O %eip[10] e o %ebp[11] são registradores de 32 bits utilizados para fins específicos. O %eip é o registrador que armazena o endereço da próxima instrução a ser executada e o registrador %ebp é o indicador do *frame* corrente que armazena o endereço do seu início.

[9] bss: *block started by symbol.*
[10] eip: *extended instruction pointer.*
[11] ebp: e*xtended base pointer.*

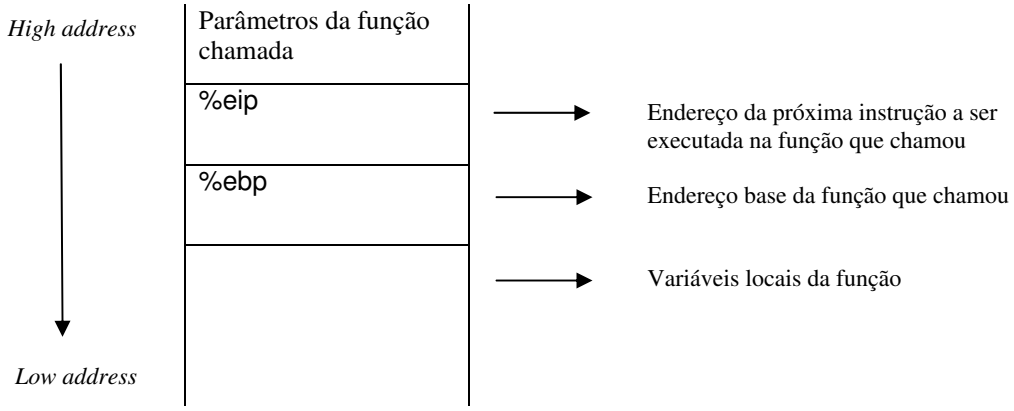

Figura 2.4 – Estrutura do *Stack Frame*

O %ebp armazenado no *stack frame* da função criada é o conteúdo do %ebp da função que a chamou e o %ebp da função chamada armazena o endereço desta célula do *frame*. Com isto gera-se uma lista ligada de *stack frame.*

Os conteúdos do %eip e do %ebp armazenados no *stack frame* da função chamada não podem ser alterados e caso isto aconteça, ocorrerá falha de segmentação quando o controle da execução do programa retornar para a função que a chamou.

A Figura 2.5 ilustra as células do *stack frame*. Cada célula é denominada *dword* e armazena até 32 bits de informação.

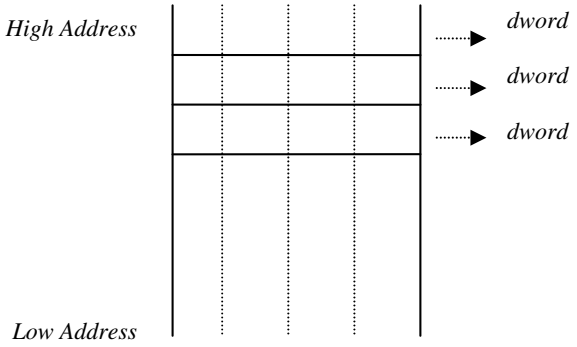

Figura 2.5 – *Dwords* do *Stack Frame*

O espaço reservado na pilha para armazenar as variáveis locais de uma função varia de máquina para máquina. Depende do compilador usado, do sistema operacional e da arquitetura da máquina [Gia01]. O compilador reserva o espaço para as variáveis em *dwords*, não em bytes.

Portanto, se houver uma declaração local a uma função do tipo:

```
char var;
```

O espaço necessário para armazenar esta variável é de um byte. O compilador reserva este espaço dentro de uma *dword*, que tem quatro bytes, não utilizando os outros três bytes. Dessa forma, é como se o compilador reservasse uma *dword* no *stack frame* da função para a variável acima.

2.4.2 Ataques do Tipo *Buffer Overflow*

Como já mencionado, um *buffer overflow* ocorre quando o programador armazena uma informação em uma área de memória de tamanho insuficiente para recebê-la. Isto acontece devido à falta de verificação do tamanho da informação sendo armazenada. Se a informação é maior que a área reservada para recebê-la, ocorrerá sobrescrita numa região adjacente à área reservada na memória, alterando o seu conteúdo.

Na Figura 2.6 mostra-se um exemplo da ocorrência de um *buffer overflow* em uma variável denominada buffer, local a uma função. A atribuição da informação x que tem doze bytes, à variável buffer, que tem quatro bytes de memória alocada, alterou o valor do %ebp e do %eip armazenados no *stack frame* da função.

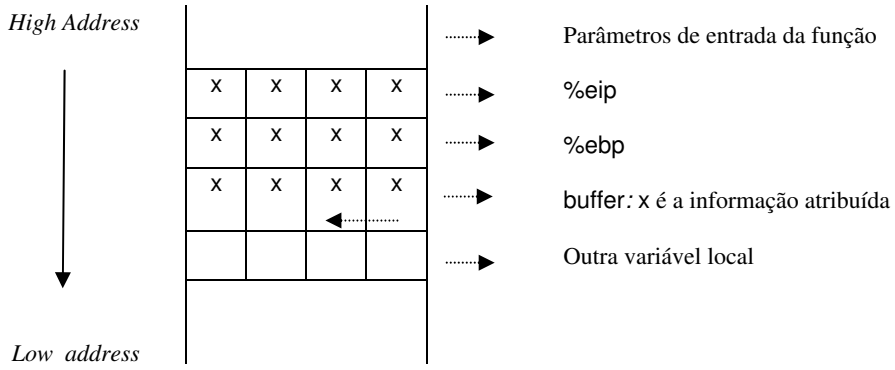

Figura 2.6 – Ocorrência de *buffer overflow* na variável buffer

Os ataques do tipo *buffer overflow* ocorrem através da exploração desta vulnerabilidade de software, geralmente em programas implementados utilizando a linguagem de programação C que usam funções como *strcpy*, *strcat* e *gets*, suscetíveis a este tipo de problema.

A função *strcpy* copia a informação da variável de origem para a variável de destino, sem verificar se a quantidade de bytes da informação de origem é maior que o tamanho em bytes suportado pela variável de destino. O mesmo ocorre com a função *strcat*, que concatena a informação da variável de origem à variável de destino, sem verificar se a variável de destino suporta a concatenação.

A exploração destas vulnerabilidades de software leva a ataques do tipo *buffer overflow* que têm como principais objetivos:

- inserção de códigos maliciosos; e
- mudança do fluxo do programa para o código binário inserido.

Enquanto a inserção de códigos maliciosos é opcional, a corrupção do fluxo de controle é essencial para que ataques desse tipo tenham sucesso.

Basicamente o ataque *buffer overflow* ocorre da seguinte maneira [Har99]:

- encontro de um código vulnerável a este tipo de ataque;
- introdução do código malicioso a ser executado no *buffer* alocado na pilha;
- modificação do endereço de retorno, o %eip, para apontar para o endereço do código inserido.

Os programas que são atacados através desta técnica são geralmente *daemons* privilegiados; programas que executam com privilégios de *root* para efetuar algum serviço. Geralmente os códigos inseridos iniciam uma nova *shell*, como por exemplo, "exec sh", levando o atacante a obter os mesmos direitos do processo atacado [Cow98]. A Figura 2.7 mostra um exemplo desse tipo de ataque.

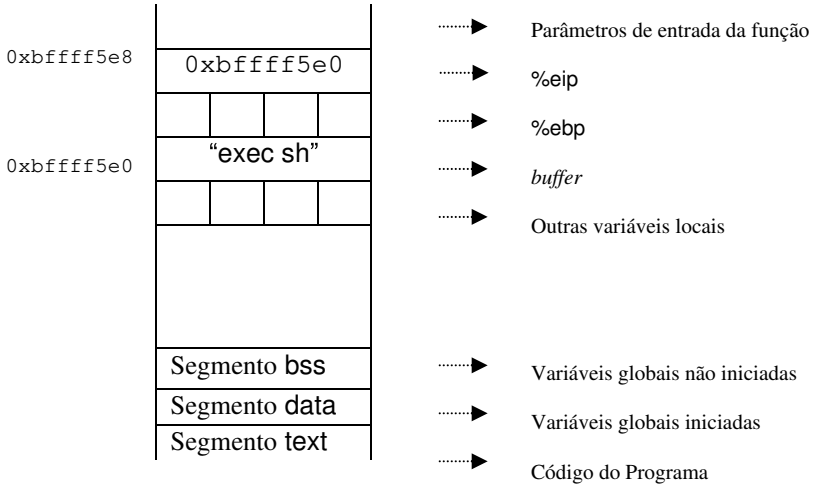

Figura 2.7 – Ataque do tipo *Buffer Overflow*

Algumas vezes o código que o atacante precisa para realizar o ataque já se encontra no espaço de endereçamento do programa, não havendo necessidade de introduzi-lo. Por exemplo, existem fragmentos de código na libc[12] que faz "exec (parâmetro)". O atacante usa este ataque para corromper o argumento e para modificar o %eip apontando-o para o endereço dentro do segmento de text onde se encontra o fragmento de código apropriado [Cow99].

[12] libc é a biblioteca de funções padrão da linguagem C.

A injeção de códigos maliciosos e o corrompimento do *buffer* não precisam acontecer em uma única ação. O atacante pode injetar códigos maliciosos dentro de um *buffer* sem transbordá-lo e transbordar um *buffer* diferente para corromper o %eip [Cow99].

Existem várias formas de corromper o fluxo de controle de um programa [Wil03]:

- pela modificação do registrador %eip;
- pela modificação do registrador %ebp;
- pelos ponteiros de funções;
- pelos *buffers longjmp*.

O ataque que tem como objetivo modificar o registrador %eip é denominado *stack smashing,* sendo o mais comum dos ataques do tipo *buffer overflow*. Neste tipo de ataque, como já descrito acima, os dados de entrada são projetados para redirecionar o %eip para o código malicioso do atacante através do transbordamento do *buffer* [Wil03].

O segundo tipo de ataque *buffer overflow,* não tão simples, tem como objetivo modificar o registrador %ebp que armazena o endereço base da função. Neste tipo de ataque, é criado um *stack frame* falso cujo %eip aponta para o código de ataque. O *buffer* é sobrescrito no *stack frame* real, de forma a redirecionar o %ebp armazenado para apontar para o *stack frame* falso. No término da função, o controle é passado para o *stack frame* falso que imediatamente redireciona, através do seu %eip, o fluxo de controle para o código de ataque [Wil03].

O terceiro tipo de ataque *buffer overflow* tem como objetivo modificar os ponteiros de funções. A função "void (* foo) ()" declara a variável foo que é do tipo "ponteiro de função que retorna void". Os ponteiros de funções podem estar alocados na heap, na pilha e no segmento bss e o atacante só precisa achar um *buffer* adjacente ao ponteiro, sobrescrevendo-o e conseqüentemente alterando o ponteiro de função. Quando o programa faz uma chamada através deste ponteiro, o programa irá desviar para a posição desejada do atacante [Cow99].

O quarto tipo de ataque *buffer overflow* tem como alvo os *buffers longjmp*. Na linguagem de programação C há as funções *setjmp* e *longjmp,* utilizadas para executar jump entre duas funções. A função *setjmp(buffer)* salva o conteúdo da pilha do sistema no *buffer* para mais tarde ser usada por *longjmp(buffer)*. Entre os dados salvos estão o %ebp e o %eip. O comando *longjmp(buffer)* leva à execução da próxima instrução após o *setjmp(buffer)*. Entretanto, se o atacante corromper o estado do *buffer*, então *longjmp(buffer)* desviará a execução para o código do atacante [Cow99].

Na próxima seção serão discutidos vários trabalhos desenvolvidos com o objetivo de detectar e prevenir, tanto estática como dinamicamente, os vários tipos de ataques envolvendo vulnerabilidade do tipo *buffer overflow*.

2.5 Trabalhos Relacionados

Existem várias maneiras estáticas e dinâmicas de prevenir intrusos de realizarem ataques do tipo *buffer overflow*. A estática previne ataques procurando por vulnerabilidades no código fonte, tal que o programador possa removê-las. A abordagem na prevenção dinâmica consiste em mudar o ambiente de *run-time* ou a funcionalidade do sistema tornando os programas menos vulneráveis, terminando a sua execução no caso de ataque [Wil03].

Nesta seção apresentam-se as principais ferramentas desenvolvidas para segurança dos sistemas contra ataques do tipo *buffer overflow*.

2.5.1 *StackGuard*

O *StackGuard* desenvolvido por Cowan [Cow98] é uma extensão do compilador gcc, que aumenta o código executável produzido por este e foi projetado para detectar e deter o *buffer overflow* do tipo *stack smashing*, cujo alvo é o endereço de retorno da função.

No ataque *buffer overflow* do tipo *stack smashing* ocorre sobrescrita de todas as células da pilha até atingir o alvo. Se o alvo for o endereço de retorno, o %eip, o atacante preencherá o *buffer* e todos os endereços da pilha acima do *buffer* até atingir o %eip.

A idéia é introduzir uma célula com um valor aleatório entre o %eip e o %ebp, denominada *canary value* e verificar se este valor foi modificado antes que o endereço de retorno seja utilizado, podendo detectar e prevenir este tipo de ataque. Se o valor for alterado, o processo é terminado.

2.5.2 *Stack Shield*

Segundo Wilander [Wil03] *Stack Shield* é um compilador desenvolvido que implementa dois tipos de proteção contra a sobrescrita do endereço de retorno da função.

A proteção *Global Ret Stack* do endereço de retorno é o *default* do *Stack Shield*. É uma pilha separada que armazena os endereços de retorno das funções chamadas durante a execução ao mesmo tempo em que o endereço é armazenado na pilha comum. Quando a função termina, o endereço de retorno da pilha comum é substituído pelo da *Global Ret Stack*, dessa forma prevenindo ataques do tipo *buffer overflow*.

Outra proteção do endereço de retorno é o *Ret Range Check*, que utiliza uma variável global para armazenar o endereço de retorno da função atual. Antes de retornar, o endereço da pilha é comparado com o da variável e caso sejam diferentes, ocorre o término da execução.

2.5.3 *ProPolice*

Segundo Wilander [Wil03], *ProPolice* é um compilador desenvolvido por Hiroaki e Kunikazu [Eto00] semelhante ao *StackGuard.* Utiliza-se também a *canary value* para detectar ataques na pilha, só que as variáveis alocadas na pilha são rearranjadas de forma que os vetores de char fiquem próximos ao %ebp. A célula da *canary value* é introduzida na pilha entre o %ebp e o vetor de char.

Dessa forma, o transbordamento do *buffer* não danifica as outras variáveis locais, somente a *canary value*, que nesta ferramenta é chamada de *guard value*. Quando o ataque é detectado o processo é terminado.

2.5.4 *Hand-coded Stack*

Segundo Cowan [Cow99], Snarskii [Sna97] desenvolveu uma implementação da libc que detecta *buffer overflow* na pilha. Esta implementação somente protege os *stack frames* das funções da biblioteca libc, protegendo os programas que as usam de vulnerabilidades dentro da biblioteca, mas não estende esta proteção para qualquer outro código.

2.5.5 *PointGuard*

De acordo com Cowan [Cow99], a Ferramenta *PointGuard* é uma generalização do *StackGuard* projetada para lidar com ataques a ponteiros de funções e a *longjmp buffer*. O *PointGuard* introduz *canaries* perto de todos as ponteiros de código (ponteiros de funções e *longjmp buffer*) e verifica a validade destas *canaries* toda vez que um ponteiro de código é dereferenciado. Se a *canary* está corrompida, a ferramenta emite um alerta e termina a execução.

2.5.6 STOBO

A STOBO - *Systematic Testing of Buffer Overflow* foi desenvolvida por Haugh e Bishop [Hau03], e tem como principal objetivo detectar vulnerabilidade do tipo *buffer overflow.* Dado um programa fonte em C, a ferramenta instrumenta-o, testa-o e usa os *warnings* gerados pela instrumentação como indicadores de prováveis *buffer overflows.*

Basicamente, são definidas para as funções da linguagem C suscetíveis ao problema de *buffer overflow*, tais como *strcat* e *strcpy*, algumas propriedades que devem ser testadas com a sua utilização, tal como “strcat(s,suffix); se alloc(s) < len(s) + alloc(suffix)?” onde len(s) significa o tamanho da *string* a ser armazenada no *buffer* s e alloc(s), a quantidade de espaço alocado para s.

A cada chamada às funções de C dentro do programa a ser testado, a ferramenta dispara uma função correspondente que verifica se alguma propriedade relacionada à função foi violada e caso isto ocorra, é gerado um relatório contendo os *warnings* de cada caso em particular.

2.5.7 *Libsafe*

A *Libsafe* apresentada por Baratloo [Bar99], estipula um limite seguro para *buffers* dentro da pilha em tempo de execução e verifica este limite antes que qualquer função vulnerável seja permitida escrever no *buffer*. Como valor de fronteira o *Libsafe* utiliza o %ebp armazenado na pilha após o %eip. Nenhuma variável na pilha é permitida se expandir além do %ebp, de forma a preservar o %eip.

Esta limitação é cumprida através de funções que verificam se o tamanho da informação de entrada está dentro do limite do *buffer*. Caso não esteja, a função envia um alerta para o arquivo log do sistema e termina o programa.

2.6 Considerações Finais

Neste capítulo, foram apresentados os principais conceitos relacionados ao teste de software, principalmente os referentes à Análise de Mutantes, necessários para o entendimento do trabalho apresentado nesta dissertação.

Apresentou-se também como o *buffer overflow* ocorre nos programas implementados utilizando a linguagem de programação C e como esta vulnerabilidade pode ser explorada através dos vários ataques do tipo *buffer overflow*.

Foram apresentadas várias ferramentas desenvolvidas com o objetivo de detectar ou prevenir ataques do tipo *buffer overflow* e suas principais características. A ferramenta proposta neste trabalho, da mesma forma que as apresentadas na seção anterior, tem como objetivo detectar *buffer overflow*, mas por meio de uma abordagem diferente – a aplicação de Análise de Mutantes.

No Capítulo 3 é apresentada a técnica "Teste de Vulnerabilidade de Segurança", que propõe a aplicação de Análise de Mutantes à detecção da presença de vulnerabilidade do tipo *buffer overflow*.

Capítulo 3

Teste de Vulnerabilidade de Segurança

Neste capítulo apresenta-se a definição da técnica estudada, implementada e validada neste trabalho denominada "TESTE DE VULNERABILIDADE DE SEGURANÇA", utilizada na detecção da presença de vulnerabilidades de software por meio da aplicação do critério Análise de Mutantes a programas fontes implementados utilizando a linguagem de programação C.

Apresenta-se também a descrição de cada um dos operadores de mutação utilizados pelo método na detecção da presença de vulnerabilidade do tipo *buffer overflow* e apresentam-se as características mínimas que as vulnerabilidades devem possuir para serem candidatas à aplicação da técnica.

3.1 Definição da Técnica

Nesta seção apresenta-se a técnica proposta por Vilela [Vil02] "TESTE DE VULNERABILIDADES DE SEGURANÇA" para detecção da presença de vulnerabilidades de software. Existem basicamente duas abordagens para inibir ataques e invasões aos sistemas.

A primeira é baseada na premissa de que nenhum software em operação está livre de vulnerabilidades de segurança. Portanto, técnicas e ferramentas de detecção de invasão devem ser desenvolvidas e aplicadas para proteger tais softwares após o sistema entrar em operação. Esta é uma abordagem importante e algumas empresas protegem-se contra ataques utilizando algumas destas ferramentas como principal defesa contra invasores.

As principais ferramentas desenvolvidas com o objetivo de proteger os sistemas após estes entrarem em operação são:

- software antivírus, que elimina os programas maliciosos que chegam ao sistema provenientes de entidades externas [For01];
- *firewall,* projetado para monitorar o tráfego da rede e os pedidos de serviços, filtrando e rejeitando os pacotes e os pedidos que falham ao atender aos critérios de segurança estabelecidos pela organização [Sta99]; e
- softwares de detecção de intrusos, que têm a tarefa de monitorar o uso dos sistemas detectando a aparição de estados inseguros [Deb99].

No entanto, nesta dissertação propõe-se trabalhar com uma abordagem complementar, tratando o problema na sua origem.

Segundo Gaur [Gau00], devido às pressões do mercado para entrega do produto final, desenvolvedores acabam negligenciando práticas para uma codificação segura, liberando uma versão da aplicação bastante vulnerável. Alguns *hackers* são capazes de perceber as práticas de codificação adotadas pelos programadores e tentam tirar vantagens delas.

De acordo com Bishop [Bis99] os princípios para uma programação segura são:

i. Princípio do Privilégio Mínimo:
   - projete seu programa para ser executado com os privilégios mínimos necessários;

ii. Princípio do *Default Failsafe*:
   - sempre negue privilégios por *default*;

iii. Princípio da Economia de Mecanismo:
   - código simples minimiza a chance de defeitos;

iv. Princípio da Mediação Completa:
   - verifique todo acesso a todo objeto:
     - verifique valores de retorno de todas as sub-rotinas que retornam valores, e
     - verifique valores de todos *system calls,* especialmente *opens*, *reads* e *writes*;
   - nunca confie nas tarefas executadas por outros códigos;

v. Princípio do Projeto Aberto:
   - evite a ocultação de detalhes ou medidas de segurança, pois resultam em falsa sensação de segurança,
   - use senhas (*passwords*) e criptografia para segurança, e
   - verifique abundantemente os erros para assegurar a funcionalidade do código;

vi. Princípio da Separação de Privilégio:
   - conceda privilégios somente após atender várias condições;

vii. Princípio do Mínimo Mecanismo Comum:
   - minimize ou elimine recursos compartilhados; e

viii. Princípio da Aceitabilidade Psicológica:
   - mantenha sempre o usuário final em mente durante o processo de projeto:
     - determine os requisitos de segurança e atenda os requisitos mínimos, e
     - precauções de segurança não devem se tornar inconvenientes no uso do sistema.

A falta de entendimento sobre questões de segurança e sobre como programas são quebrados, acaba levando o programador a cometer enganos que poderiam ser evitados se o mesmo adotasse o uso dos princípios citados acima, minimizando os problemas de quebra de segurança dos sistemas.

A escolha de uma linguagem de programação menos vulnerável também é um fator importante em relação à segurança de um sistema. Preferir utilizar a linguagem de programação Java à linguagem C na implementação de uma aplicação, acaba resultando num projeto mais seguro em relação a alguns aspectos de segurança, como os da manipulação de *string* pela linguagem Java.

A família das funções de manipulação de *string* existentes na linguagem C são bastante vulneráveis. A função *strcpy*, que atribui a informação da origem ao destino sem verificar se este suporta a informação recebida, torna este ponto do programa vulnerável a ataques do tipo *buffer overflow*.

A abordagem proposta tem como intenção detectar vulnerabilidades, como o exemplo da função *strcpy* citado acima, antes de o software entrar em operação, semelhante ao que acontece durante a fase de teste de um sistema.

O processo de desenvolvimento de um software contém três fases genéricas independente do paradigma de engenharia de software escolhido [Pre04]:

- a fase de definição do sistema;
- a fase de desenvolvimento do sistema; e
- a fase de manutenção do sistema.

Na fase de desenvolvimento do sistema, o teste é conduzido com o objetivo de achar o maior número possível de defeitos na aplicação e na fase após a sua implantação, as falhas restantes são tratadas através de técnicas de tolerância a falha ou outras técnicas dependendo do software ser uma aplicação com requisitos críticos em relação à qualidade.

A abordagem concentra-se na fase de desenvolvimento, conduzindo o teste de forma a remover o máximo possível de vulnerabilidades antes de o software entrar em operação. A técnica considera o uso do Critério Análise de Mutantes, descrito no Capítulo 2, para ajudar a identificar e eliminar tais vulnerabilidades. Esta técnica é chamada de "TESTE DE VULNERABILIDADE DE SEGURANÇA". A hipótese adotada supõe ser possível executar tais testes e conseqüentemente remover tais vulnerabilidades.

A técnica é similar às técnicas de teste tradicionais. No teste procura-se identificar a presença de um maior número possível de defeitos no software por meio da utilização de um conjunto de dados de teste de qualidade. A qualidade de um conjunto de dados de teste é, em geral, avaliada pela cobertura obtida pelo conjunto para um dado critério de teste. Por exemplo, considerando o critério de teste TODOS OS NÓS, um bom conjunto de dados de teste é aquele que exercita todos os nós de um programa. Assume-se que um bom conjunto de dados de teste tem grande capacidade de revelar a presença de defeitos no programa. Se a presença de defeitos não é revelada, a cobertura do critério de teste pode indicar a qualidade do teste aplicado e, conseqüentemente, a qualidade do próprio programa em teste.

Na análise de mutantes a cobertura apontada pelo ESCORE DE MUTAÇÃO, é dada pelo número de mutantes não equivalentes mortos. Na técnica "TESTE DE VULNERABILIDADE DE SEGURANÇA" os operadores de mutação simulam a ocorrência de vulnerabilidades no programa. Para satisfazer o CRITÉRIO ANÁLISE DE MUTANTES novos dados de teste devem ser utilizados. A hipótese é que esses novos dados de teste exercitem características que não foram previamente exercitadas pelo teste tradicional, dando a chance de o testador identificar a presença de uma vulnerabilidade no programa.

3.2 Características das Vulnerabilidades

Nesta seção são discutidas as características mínimas que as vulnerabilidades devem possuir para serem candidatas à aplicação da técnica e o motivo da vulnerabilidade do tipo *buffer overflow* ter sido selecionada como um estudo de caso inicial.

Existem vulnerabilidades que ocorrem devido a defeitos de software que se revelam no código fonte. Estes defeitos criam um ponto vulnerável na aplicação, que, se explorados de forma apropriada, forçam o programa a produzir resultados incorretos com um comportamento diferente do inicialmente esperado. Se estas vulnerabilidades puderem ser modeladas com operadores de mutação, então é possível testar o programa para vulnerabilidades de segurança.

O *buffer overflow* é um candidato a ser tratado pela abordagem, pois é uma falha que ocorre devido a um defeito do código fonte e que pode ser eliminado através de teste estrutural. Um *buffer overflow* é possível porque geralmente esta falha ocorre quando o programador armazena uma informação em uma área de memória de tamanho insuficiente para recebê-la.

Isto ocorre devido à falta de verificação no tamanho da área que recebe a informação. Se a informação é maior que a área reservada para recebê-la, ocorrerá sobrescrita numa região adjacente à área reservada na memória, ocorrendo o transbordamento, o que pode levar a aplicação a uma falha.

Entre as informações perdidas estão:

- o conteúdo das variáveis adjacentes à área reservada;
- o endereço base da função que chamou, o %ebp; e
- o endereço de retorno da função que chamou, o %eip.

Quando uma função é chamada, são armazenados no seu *stack frame,* como mostrado na Figura 3.1, os seus parâmetros de entrada, o endereço para o qual o controle do programa deve retornar quando a função terminar, o %eip, o endereço base da função que a chamou, o %ebp e as suas variáveis locais.

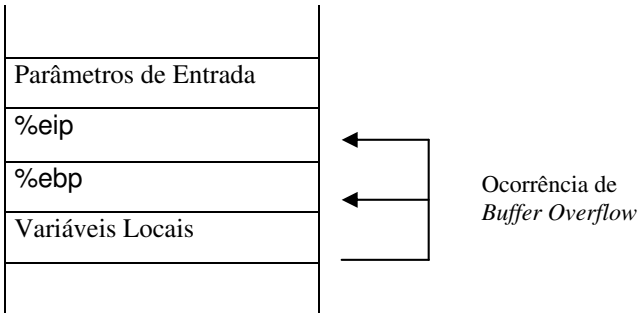

Figura 3.1 – *Stack Frame* de uma Função

A informação que é armazenada na área das variáveis locais localizada no *stack frame* da função pode sobrescrever o endereço base da função que a chamou, o %ebp e/ou o endereço de retorno, o %eip, como é mostrado na Figura 3.1, desviando o controle da execução do programa para um outro endereço quando a execução da função chamada terminar.

O ponto para o qual o programa retorna pode ser um endereço dentro da própria área do *stack frame* da função onde estão as suas variáveis locais, cujo conteúdo foi definido pelo atacante e cujo código pode passar o controle do sistema para o invasor, ou pode ser um outro endereço dentro da memória, que tem o código que o atacante necessita para realizar o seu ataque ao sistema [Hau03].

A abordagem proposta pode ser usada para prevenir o *buffer overflow* de estar presente no código fonte antes de o software entrar em operação.

3.3 Definição dos Operadores de Mutação

Nesta seção apresentam-se os operadores de mutação MSMA e MDMA, definidos com o objetivo de detectar a presença de vulnerabilidade do tipo *buffer overflow*.

Na utilização de Análise de Mutantes para detectar a presença de vulnerabilidade do tipo *buffer overflow*, os operadores devem inserir artificialmente um defeito que crie situações com a finalidade de possibilitar ao testador a detecção da presença de um possível *buffer overflow* e com isto eliminar o defeito do código fonte.

Como estaticamente não é possível mudar uma informação, para gerar o *buffer overflow* é necessário implementar operadores de mutação que modifiquem a quantidade de memória alocada para uma variável. A memória para uma variável dentro de um programa pode ser alocada tanto dinâmica como estaticamente.

Para isto Vilela [Vil02] propôs dois operadores de mutação, MSMA e MDMA, como mostrado na Tabela 3.1. Cada operador terá quatro variações que serão descritas abaixo.

Estes operadores devem ser aplicados a sistemas implementados utilizando o C como linguagem de programação.

Operador de Mutação	**Significado**
MSMA	*Modification of Static Memory Allocation*
MDMA	*Modification of Dynamic Memory Allocation*

Tabela 3.1 – Operadores de Mutação

3.3.1 Operador MSMA

O operador MSMA, descrito na Tabela 3.1, tem como objetivo modificar a quantidade de memória alocada para uma variável declarada estaticamente, visando com isto a revelação de defeitos que podem levar a um *buffer overflow*.

Aplica-se o operador de mutação MSMA a instruções C com as seguintes sintaxes:

```
TIPO IDENTIFICADOR_DA_VARIÁVEL[ÍNDICE];
```

```
TIPO IDENTIFICADOR_VARIÁVEL[ÍNDICE][ÍNDICE];
```

Onde o TIPO e o ÍNDICE correspondem a:

TIPO : o tipo básico char da linguagem C, e
ÍNDICE : um inteiro, uma constante ou uma expressão aritmética.

O operador MSMA será aplicado a vetores unidimensionais ou bidimensionais do tipo char, pois este trabalho restringe-se a detectar *buffer overflow* nos programas em linguagem C que utilizam funções suscetíveis ao *buffer overflow*, tais como *strcpy*, *strncpy*, *strcat*, *strncat* e *gets*, já que estas utilizam char como parâmetros.

A cada aplicação do operador MSMA são geradas quatro variações para cada alocação de memória identificada. As variações do operador MSMA propostas por Vilela [Vil02] são descritas na Tabela 3.2. Optou-se por implementar a aplicação do operador MSMA a vetores com até duas dimensões, pois vetores com mais dimensões são menos comuns nas aplicações.

A aplicação da primeira variação do operador MSMA a uma variável tem como objetivo alocar a quantidade de memória originalmente alocada para esta variável reduzindo-a em apenas um byte. A aplicação da segunda variação tem como objetivo alocar somente a metade da quantidade de memória originalmente alocada para a variável. Caso o ÍNDICE da variável seja ímpar, será alocada a quantidade de memória cujo tamanho equivale à parte inteira da divisão por dois. A aplicação da terceira variação do operador à variável tem como objetivo dobrar a quantidade de memória alocada originalmente para esta variável e a quarta variação tem como objetivo alocar somente um byte para a variável.

VARIAÇÕES DOS OPERADORES
• Quantidade de memória alocada menos um • Quantidade de memória alocada dividida por dois • Quantidade de memória alocada multiplicada por dois • Quantidade de memória alocada igual a um byte

Tabela 3.2 – Variações dos Operadores de Mutação

Sendo assim, a aplicação do operador MSMA à declaração abaixo, gera quatro mutantes distintos, cada um contendo uma alocação de memória de tamanho diferente.

O programa P com a declaração original da variável VAR:

```
CHAR VAR[TAM];
```

O mutante m_1 terá a declaração original substituída pela seguinte declaração:

```
CHAR VAR[TAM-1];
```

O mutante m_2 terá a declaração original substituída pela seguinte declaração:

```
CHAR VAR[TAM/2];
```

O mutante m_3 terá a declaração original substituída pela seguinte declaração:

```
CHAR VAR[TAM*2];
```

E o mutante m_4 terá a declaração original substituída pela seguinte declaração:

```
CHAR VAR[1];
```

3.3.2 Operador MDMA

O operador MDMA, descrito na Tabela 3.1, tem como objetivo modificar a quantidade de memória alocada dinamicamente para uma variável, visando a revelação de defeitos que podem levar a um *buffer overflow*.

Aplica-se o operador MDMA a instruções de alocação de memória dinâmica da linguagem C, que utilizam funções especificadas pelo padrão ANSI C que são MALLOC(), CALLOC() e REALLOC().

As sintaxes destas funções são:

```
VOID *MALLOC (TAM);
VOID *CALLOC (NUM, TAM);
VOID *REALLOC (PTR, TAM);
```

A função MALLOC aloca dinamicamente a memória para uma variável na *heap* do processo. Essa função tem como parâmetro de entrada a quantidade em bytes a ser alocada (TAM) e tem como valor de retorno, um ponteiro para o primeiro byte da região alocada na memória. Caso a quantidade de memória disponível na *heap* não seja suficiente para a alocação, a função MALLOC retorna um ponteiro nulo.

A função CALLOC aloca dinamicamente na *heap* do processo uma quantidade de memória para um vetor de NUM elementos de TAM bytes cada um, indicados através dos parâmetros de entrada desta função e tem como valor de retorno, um ponteiro para o primeiro byte da região alocada. Se não existir memória suficiente na *heap* para ser alocada é retornado pela função CALLOC um ponteiro nulo.

A função REALLOC altera a quantidade de memória já alocada anteriormente na *heap* pelas funções MALLOC ou CALLOC apontada pelo ponteiro PTR, alocando a nova quantidade fornecida em bytes por TAM. PTR e TAM são parâmetros de entrada da função REALLOC. Um ponteiro é retornado para o primeiro byte do bloco de memória alocado pela função. Se não existir memória suficiente na *heap* para ser alocada é retornado um ponteiro nulo.

A cada aplicação do operador MDMA, são geradas quatro variações para cada alocação de memória identificada. As variações do operador MDMA propostas por Vilela [Vil02] são descritas na Tabela 3.2.

Na identificação de uma alocação de memória dinâmica utilizando a função MALLOC, a aplicação do operador MDMA gera quatro mutantes distintos. Cada mutante terá no lugar da alocação original do programa P uma alocação com uma quantidade de memória alocada de

tamanho diferente. As alocações de memória modificadas de acordo com cada variação e que correspondem a cada mutante gerado são:

```
(P)     ...MALLOC(TAM);
(m1)    ...MALLOC(TAM-1);
(m2)    ...MALLOC(TAM/2);
(m3)    ...MALLOC(TAM*2);
(m4)    ...MALLOC(1);
```

No caso da função CALLOC, cada mutante terá no lugar da alocação original do programa P, uma alocação modificada correspondendo a uma variação do operador MDMA. As características de cada alocação modificada são:

```
(P)     ...CALLOC(NUM,TAM);
(m1)    ...CALLOC(NUM-1,TAM);
(m2)    ...CALLOC(NUM/2,TAM);
(m3)    ...CALLOC(NUM*2,TAM);
(m4)    ...CALLOC(1,TAM);
```

Como a função CALLOC tem como objetivo alocar vetores dinamicamente na *heap* do processo, modificou-se a quantidade de elementos do vetor, o parâmetro NUM, como no caso da alocação estática e não o parâmetro TAM, a quantidade em bytes a ser alocada para cada elemento do vetor. No caso do mutante resultante da aplicação da variação que aloca quantidade de memória dividida por dois, NUM/2, a quantidade alocada será a parte inteira desta divisão multiplicada por TAM.

Na identificação de uma alocação de memória dinâmica utilizando a função REALLOC, a aplicação do operador MDMA gera quatro mutantes distintos, cada um contendo no lugar da alocação original do programa P uma alocação com uma quantidade de memória alocada de tamanho diferente. As alocações de memória modificadas de acordo com cada variação e que correspondem a cada mutante gerado são:

```
(P)     ...REALLOC(PTR,TAM);
(m1)    ...REALLOC(PTR,TAM-1);
(m2)    ...REALLOC(PTR,TAM/2);
(m3)    ...REALLOC(PTR,TAM*2);
(m4)    ...REALLOC(PRT,1);
```

3.4 Indicação de *Buffer Overflow*

Nesta seção apresentam-se os prováveis resultados que podem indicar ao testador uma possível presença de *buffer overflow* no programa em teste.

Após a aplicação dos operadores de mutação MSMA e MDMA ao programa a ser testado, o programa original e os mutantes gerados devem ser executados com dados de teste. O conjunto de dados de teste utilizado deve exercitar as características de cada mutante gerado, de forma a levar ou o programa original ou os mutantes a um comportamento não esperado, desta forma indicando ao testador a possível presença de *buffer overflow* no programa sendo testado. A adequação do conjunto de dados de teste utilizado é apontada pelo escore de mutação que indica a qualidade do teste aplicado, aumentando a confiança de que o programa foi bem testado.

O principal objetivo do atacante, baseado no modo como ocorrem os ataques do tipo *buffer overflow* descritos no Capítulo 2, é modificar o endereço de retorno de uma função. Se o dado de teste ao ser armazenado no *buffer* alocado no *stack frame* da função alterar o endereço de retorno para um endereço de memória não alocado para o uso do programa, ocorrerá falha de segmentação no programa sendo executado. Portanto, a ocorrência de falha de segmentação no programa em teste é um dos resultados que podem indicar ao testador uma possível presença de vulnerabilidade do tipo *buffer overflow*.

A atribuição de uma informação ao *buffer* alocado no *stack frame* da função, pode sobrescrever os conteúdos das variáveis adjacentes ao *buffer*, que se alterados, podem levar o programa a fornecer um resultado diferente do esperado. Desta forma, um resultado diferente do esperado pelo programa original pode também ser uma indicação ao testador de uma possível presença de vulnerabilidade do tipo *buffer overflow*.

Estes resultados só são válidos na indicação de um possível *buffer overflow* se o programa em teste estiver utilizando funções da linguagem C que são suscetíveis a este tipo de problema tais como: *strcpy, strncpy, strcat, strncat* e *gets*.

No Capítulo 4 apresenta-se a Ferramenta SEVMUT – ***Se****curity* ***V****ulnerabilities* ***Mu****tation* ***T****ool,* desenvolvida com o objetivo de validar a técnica "Teste de Vulnerabilidade de Segurança". Dado um programa em teste a ferramenta deve ser capaz de percorrê-lo procurando por alocações de memória candidatas à aplicação do operador de mutação escolhido pelo testador e gerar os mutantes para cada alocação identificada.

O desempenho da Ferramenta SEVMUT em relação ao tempo de geração dos mutantes, deve ser compatível com o tamanho do programa em teste, ou seja, o número de linhas de código do programa e com o número de alocações de memória candidatas à aplicação dos operadores MSMA e MDMA. Um experimento será realizado com o objetivo de validar este desempenho.

3.5 Considerações Finais

Neste capítulo apresentou-se a técnica "TESTE DE VULNERABILIDADE DE SEGURANÇA" para detectar a presença de vulnerabilidade do tipo *buffer overflow,* através da aplicação do Critério Análise de Mutantes a programas fontes escritos em C.

Uma vulnerabilidade para ser candidata à aplicação da técnica, deve ser originada por uma imperfeição ou omissão no código do programa, criando um ponto vulnerável no programa que pode ser modelado com operadores de mutação.

Para aplicar análise de mutantes à detecção da presença de *buffer overflow*, foram definidos dois operadores de mutação, MSMA e MDMA, com quatro variações cada um:

- quantidade de memória alocada menos um;
- quantidade de memória alocada dividida por dois;
- quantidade de memória alocada vezes dois; e
- quantidade de memória alocada igual a um.

A indicação da presença de uma vulnerabilidade do tipo *buffer overflow* ao testador, ocorre através de uma falha de segmentação ou de um resultado diferente do esperado na execução do programa com dados de teste.

No Capítulo 4 apresenta-se a Ferramenta SEVMUT desenvolvida com o objetivo de validar a técnica descrita neste capítulo.

Capítulo 4

Descrição da Ferramenta SEVMUT

Neste capítulo apresenta-se a SEVMUT – ***Se****curity* ***V****ulnerabilities* ***Mu****tation* ***T****ool,* uma ferramenta desenvolvida para detecção da presença de vulnerabilidade do tipo *buffer overflow* através da aplicação do critério Análise de Mutantes. A detecção da presença de vulnerabilidade ocorre dinamicamente através da execução do programa original e dos mutantes com os dados de teste de entrada.

A SEVMUT foi desenvolvida para validar a técnica descrita no Capítulo 3 e tem como principal função a aplicação dos operadores MSMA e MDMA a programas escritos em C.

As funções implementadas pela SEVMUT são:

- geração do programa mutante-fonte;
- compilação do programa mutante-fonte;
- execução do programa mutante[13];
- análise dos mutantes vivos;
- cálculo do escore de mutação; e
- geração de relatórios.

Este conjunto de funções é necessário para aplicação do critério Análise de Mutantes à detecção da presença de vulnerabilidade do tipo *buffer overflow*.

Neste capítulo apresenta-se também a arquitetura da ferramenta com os seus principais aspectos de implementação e a condução de um experimento realizado com o objetivo de validar a SEVMUT.

4.1 Descrição Funcional da SEVMUT

Como já mencionado, a Ferramenta SEVMUT tem como principal função a aplicação dos operadores de mutação MSMA e MDMA a programas implementados utilizando o C como linguagem de programação, gerando o programa mutante-fonte.

O programa mutante-fonte é compilado e executado com um conjunto de dados de teste de entrada. A adequação do conjunto na detecção da presença de vulnerabilidade do tipo *buffer overflow* é indicado através de um escore de mutação fornecido ao usuário no final da execução dos mutantes.

[13] O termo programa mutante, neste trabalho, refere-se ao código executável do programa mutante-fonte.

A detecção da presença de vulnerabilidade do tipo *buffer overflow* ocorre através de uma falha de segmentação na execução do programa original ou na execução dos mutantes com os dados de teste de entrada, ou ocorre através de um resultado diferente do esperado pelo programa original, item melhor explicado no Capítulo 5.

Os requisitos necessários para a execução de cada uma das funções implementadas pela Ferramenta SEVMUT são descritos a seguir:

1. geração do programa mutante-fonte:
   1.1 identificar as alocações de memória candidatas à aplicação do operador escolhido pelo usuário; e
   1.2 gerar no programa mutante-fonte as alocações de memória modificadas.
2. compilação do programa mutante-fonte:
   2.1 definir uma variável de ambiente denominada MUTA que tem como objetivo controlar a compilação dos mutantes.
3. execução do programa mutante:
   3.1 criar um arquivo com os dados de teste de entrada utilizados na execução do programa mutante.
4. análise dos mutantes vivos:
   4.1 verificar se o resultado da execução do programa original diferencia do resultado do programa mutante na sua execução com o mesmo dado de teste de entrada utilizado pelo programa original.
5. cálculo do escore de mutação:
   5.1 aplicar o cálculo do escore de mutação descrito no Capítulo 2; e
   5.2 indicar a adequação do conjunto de dados de teste utilizado na execução dos mutantes através do escore de mutação.
6. geração dos relatórios:
   6.1 executar o programa mutante; e
   6.2 gerar relatório de acordo com os resultados obtidos.
7. detecção da presença de vulnerabilidade do tipo *buffer overflow* caso ela exista:
   7.1 verificar se o resultado da execução do programa mutante com o dado de teste de entrada resulta em uma falha de segmentação ou em um resultado diferente do esperado.

4.2 Arquitetura da Ferramenta SEVMUT

Nesta seção são discutidas a estrutura da SEVMUT e as principais funções implementadas pela ferramenta.

A Ferramenta SEVMUT é composta basicamente de quatro módulos principais. A Figura 4.1 ilustra sua estrutura e a ativação dos módulos. Neste diagrama, os retângulos representam os módulos, os quadrados com uma aresta inclinada representam as saídas geradas pelos módulos e também as entradas fornecidas aos módulos pelo usuário e as elipses representam as saídas fornecidas pela ferramenta; as linhas tracejadas representam o fluxo de controle e as linhas cheias o fluxo de informação.

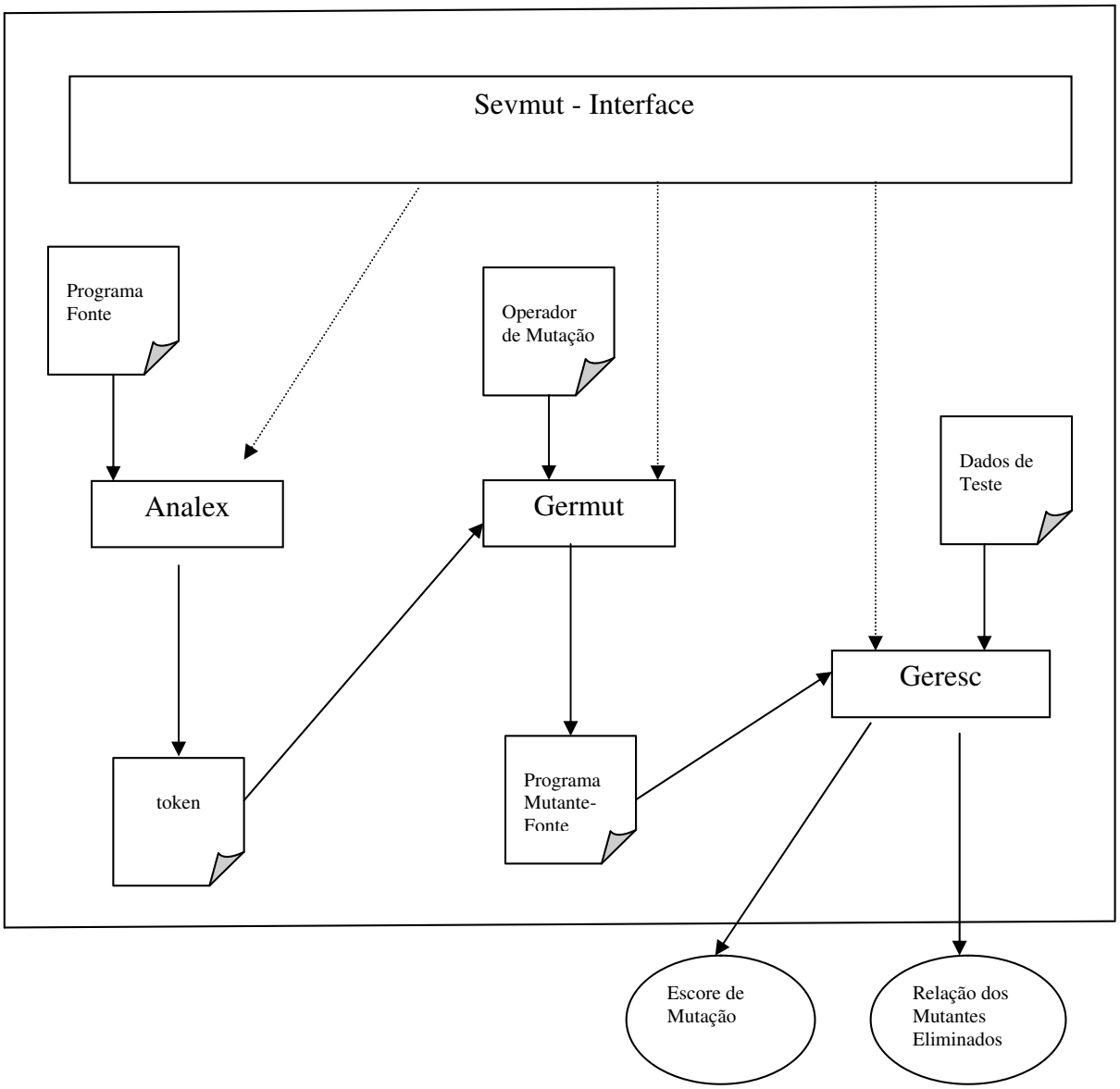

Figura 4.1 – Arquitetura da Ferramenta SEVMUT

O módulo Sevmut é o responsável pela interface da ferramenta com o usuário e tem como função a ativação de outros módulos da SEVMUT.

O módulo Analex tem como principal função fazer a análise léxica do programa a ser modificado gerando uma seqüência de *tokens* ao módulo Germut.

O módulo Germut tem como principais funções fazer a análise sintática do programa fonte, identificar as alocações estáticas e dinâmicas candidatas à aplicação dos operadores e gerar o programa mutante-fonte.

O módulo Geresc tem como funções compilar o programa original e os mutantes, executar estes programas com os dados de teste fornecidos pelo testador, gerar o escore de mutação para verificação da adequação do conjunto de dados de teste e gerar os relatórios com informações sobre os mutantes eliminados pelo conjunto de dados de teste.

A Figura 4.2 ilustra o DFD do sistema - Nível 1, indicando a transformação dos dados através das várias funções implementadas pelos módulos da Figura 4.1.

4.3 Aspectos de Implementação da SEVMUT

Nesta seção são discutidos os detalhes de implementação dos quatro módulos principais da Ferramenta SEVMUT.

A ferramenta foi desenvolvida no sistema operacional Linux distribuição Red Hat 9.0 utilizando a linguagem C na implementação dos módulos Analex e Germut e a linguagem do s*hell bash* do Linux na implementação dos *scripts* Geresc e Sevmut. Portanto, a SEVMUT deve ser executada sob este sistema operacional.

As subseções a seguir descrevem cada um dos módulos da Ferramenta SEVMUT de acordo com a Figura 4.1.

4.3.1 Módulo Sevmut

Este módulo é responsável pela interface da ferramenta com o usuário. Esta interação ocorre através de *menus*, recebendo as entradas via teclado sem a utilização de recursos gráficos. Esta característica facilita a utilização da ferramenta em programas do tipo *batch* que automatizam a aplicação da atividade de teste e a execução de experimentos.

A Figura 4.3 mostra a tela principal da SEVMUT. Através desta interface o usuário pode gerar e executar os mutantes, alterar o estado dos mutantes, obter um conjunto mínimo de dados de teste, visualizar e imprimir relatórios.

Inicialmente o usuário somente conseguirá escolher a opção “Geração de Mutantes”. Qualquer outra escolha leva à ferramenta a emitir uma mensagem alertando-o de que os mutantes ainda não foram gerados. Dentro desta opção, o módulo Sevmut pede ao testador o nome e o diretório do programa fonte e ativa o módulo Germut, que possibilita ao testador escolher qual o operador de mutação deseja aplicar às alocações de memória do programa sob teste.

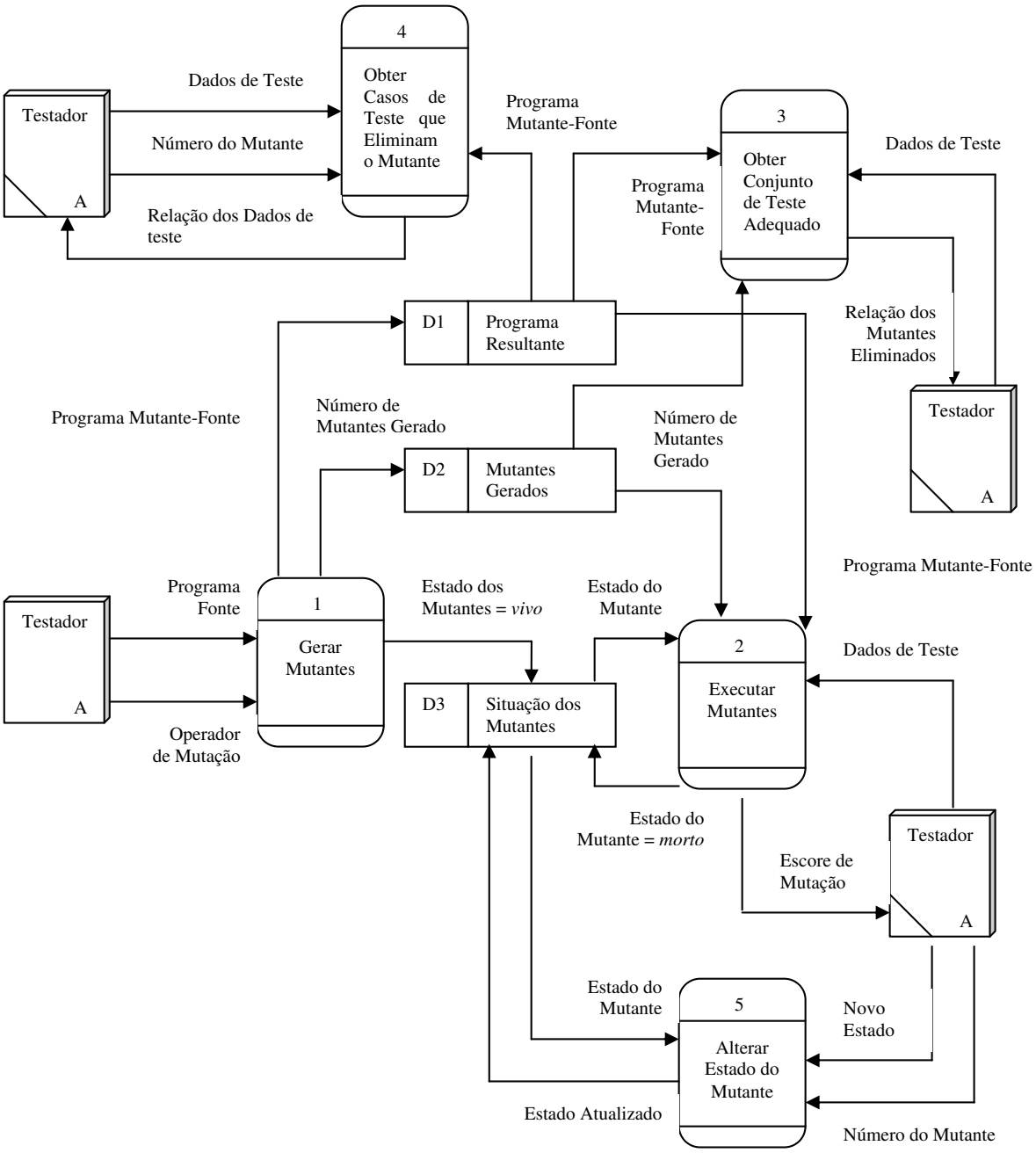

Figura 4.2 – Diagrama de Fluxo de Dados – Nível 1

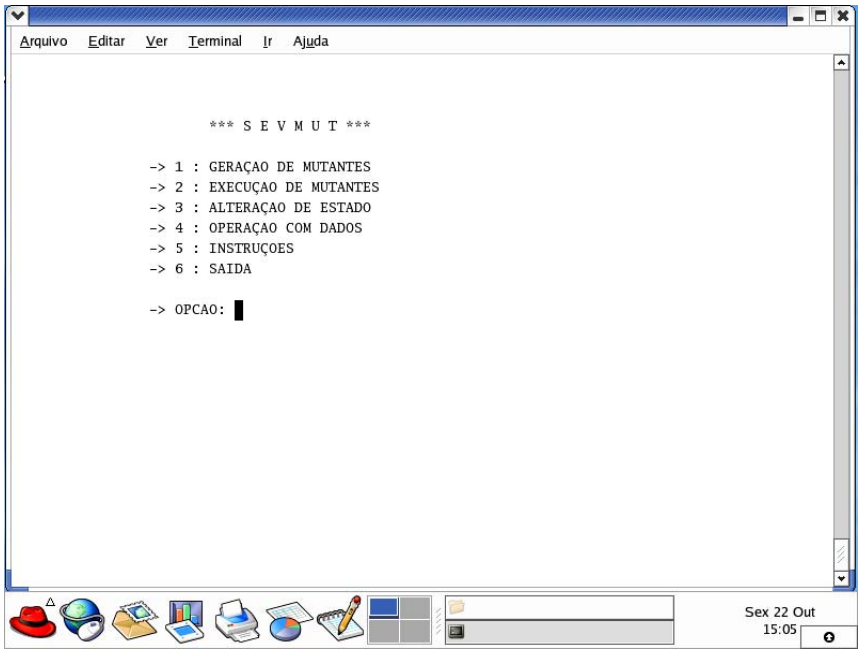

Figura 4.3 – Tela Principal da SEVMUT

Se a opção for o operador MSMA, o módulo Germut procura somente por alocações estáticas para fazer a mutação, ignorando as dinâmicas que são copiadas sem nenhuma alteração no programa mutante-fonte.

Se a opção for o MDMA ou os dois operadores simultaneamente, a ferramenta permite ao testador escolher a quais tipos de alocação dinâmica deseja aplicar o operador, podendo escolher aplicar aos tipos *calloc* ou *malloc* ou *realloc* ou a todos ao mesmo tempo.

Se for escolhida a opção “Execução de Mutantes”, o módulo Sevmut ativa o módulo Geresc, que pede ao testador o nome do programa fonte sob teste, o nome do arquivo com os dados de teste de entrada e os seus respectivos diretórios. O módulo Geresc executa o programa fonte e cada mutante utilizando estes dados de teste. Ao final da execução, o módulo indica a adequação do conjunto de dados de teste através de um escore de mutação. A Figura 4.4 mostra uma tela com o resultado de uma execução.

Na identificação de um mutante *equivalente* ao programa original, o testador utiliza a opção “Alteração de Estado” para modificar o estado do mutante.

A opção “Operação com Dados” permite ao testador obter, visualizar ou imprimir dados.

A opção “Levantamento de dados” dentro da opção “Operação com Dados” possibilita ao usuário obter quais mutantes são eliminados por um dado conjunto de dados de teste ou obter os dados de teste que eliminam um dado mutante. Estas funções são executadas pelo módulo Geresc descrito na seção 4.3.4.

Se a opção do usuário é “levantar quais mutantes são eliminados por um dado conjunto de casos de teste”, o módulo Sevmut aciona o módulo Geresc que pede ao testador o nome do arquivo com os dados de teste de entrada e o seu diretório. Executa o programa original e cada mutante obtendo quais destes casos eliminam os mutantes gerados.

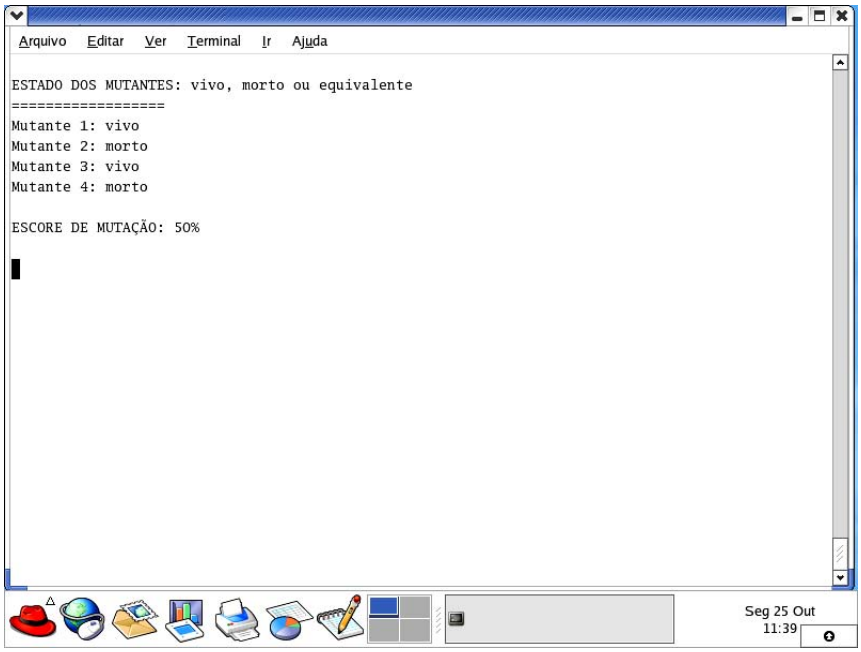

Figura 4.4 – Tela com Resultado da Execução

No caso da opção escolhida pelo usuário ser “levantar os casos de teste que eliminam um dado mutante”, o módulo Sevmut ativa o módulo Geresc que pede ao testador o número do mutante a ser executado, o nome do arquivo com os dados de teste de entrada e o seu diretório. Executa o programa original e o mutante obtendo quais destes dados de teste o eliminam.

A opção “Visualização de dados” permite ao testador visualizar o estado dos mutantes, visualizar o programa mutante-fonte gerado e visualizar os relatórios gerados com o uso da opção de levantamento de dados. A Figura 4.5 ilustra o relatório gerado para a opção “levantar quais mutantes são eliminados por um dado conjunto de casos de teste”.

A opção “Impressão de dados” permite ao usuário imprimir o programa mutante-fonte, a tabela de estado dos mutantes e os relatórios gerados com o uso da opção de levantamento de dados.

No Apêndice B são mostradas as principais telas utilizadas na interface com o usuário da ferramenta SEVMUT.

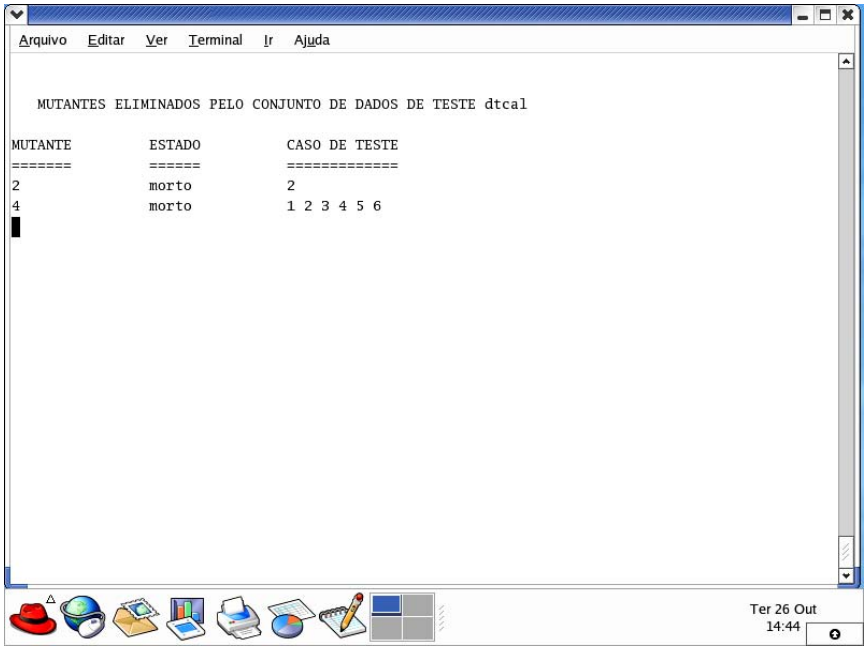

Figura 4.5 – Relatório gerado pela ferramenta

4.3.2 Módulo Analex

A partir da análise léxica do arquivo de entrada fornecido pelo testador, o módulo Analex produz como saída uma seqüência de *tokens* que será utilizada pelo módulo Germut para realizar a análise sintática. A análise léxica é realizada por este módulo com o auxílio da Ferramenta Lex [Mas90]. No Apêndice A.1 é mostrada a especificação léxica utilizada pelo módulo Analex de acordo com o padrão reconhecido pela Ferramenta Lex.

4.3.3 Módulo Germut

Este módulo realiza a análise sintática do programa fonte com o auxílio da Ferramenta Yacc [Mas90]. No Apêndice A.2 é mostrada a especificação sintática utilizada pelo módulo de acordo com o padrão reconhecido pela Ferramenta Yacc. Esta especificação foi montada baseada na gramática da linguagem C definida por Ritchie [Rit73].

A identificação das alocações de memória, candidatas à aplicação dos operadores MSMA e MDMA, ocorre no momento em que o módulo está fazendo a análise sintática. Se o operador de mutação escolhido pelo testador aplica-se à alocação de memória analisada, o módulo

chama uma função que gera no lugar da alocação identificada, códigos de diretivas de pré-processamento como os da Figura 4.6.

Estes operadores, tanto MSMA como MDMA, quando aplicados, geram para cada alocação de memória identificada, quatro mutantes distintos, cada um representando uma variação. Com isto, é possível gerar vários programas diferentes que devem ser compilados e executados.

O encapsulamento de todos os mutantes em um único mutante utilizando diretivas de pré-processamento foi a técnica utilizada para evitar esta explosão de programas. Desta forma todos os mutantes estão em um único programa mutante-fonte.

```
                .
                .
#if MUTA == 1
    quantidade alocada - 1;
#elif MUTA == 2
    quantidade alocada  / 2 ;
#elif MUTA == 3
    quantidade alocada * 2;
#elif MUTA == 4
    quantidade alocada = 1;
#else
    quantidade alocada original;
#endif
                .
                .
```

Figura 4.6 – Pseudocódigo das Diretivas de Pré-Processamento

Esta técnica baseou-se na forma como a ferramenta de teste Proteum/IM [Del97] gera os mutantes. Esta ferramenta constrói mutantes-fontes que correspondem a mais de um mutante, contendo código de diversos mutantes e não de apenas um deles.

A Ferramenta Proteum/IM [Del97] na geração dos mutantes cria várias funções, cada função contém o código correspondente a um mutante, que são chamadas durante a execução do programa dependendo de qual mutante deve ser executado. Desta maneira, todos os mutantes estão no mesmo programa, exigindo apenas uma compilação.

A Ferramenta SEVMUT gera os mutantes da seguinte maneira. Supondo o código fonte do programa *teste.c* ilustrado na Figura 4.7. Aplicando o operador MSMA, o programa mutante-fonte gerado tem no lugar da declaração da variável os códigos das diretivas de pré-

processamento como esboçado pela Figura 4.6 e todas as outras instruções do programa são copiadas no programa mutante-fonte sem nenhuma alteração. O programa resultante tem o mesmo nome do programa fonte, diferenciado com um *underline* prefixado. Sendo assim, o programa mutante-fonte resultante *_teste.c*, tem o código fonte C como o da Figura 4.8.

```
1.  int main()
2.  {
3.  char aux1[6];
4.  char aux2[6];

5.  strncpy (aux1,"teste",6);
6.  strncpy (aux2,"teste",6);

7.  printf ("\n%s",aux1);
8.  printf ("\n%s",aux2);
9.  }
```

Figura 4.7 - Código Fonte do Programa *teste.c*

Uma vantagem de se utilizar o encapsulamento é o tempo ganho com processamento. Se fosse gerado um programa diferente para cada mutante, gastar-se-ia muito tempo na geração de vários programas. Com o encapsulamento este tempo é reduzido.

Outra vantagem de encapsular todos os mutantes em um único programa mutante-fonte é o espaço que se ganha para armazenar o programa gerado. Se fossem gerados vários programas mutantes-fontes, o espaço ocupado em disco ou outro meio de armazenamento secundário seria maior que o espaço ocupado na geração de um único programa mutante-fonte.

A equação a seguir mostra a quantidade de espaço necessário para armazenar todos os programas mutantes-fontes quando gerados separadamente.

$$E_t = LoC * nro\_mut$$

onde:

E_t : espaço total utilizado;

LoC : total de linhas de código do programa fonte; e

nro_mut : total de mutantes gerados.

```
1.  int main()
2.  {
3.  #if MUTA == 1
4.     char aux1[6-1];
5.  #elif MUTA == 2
6.     char aux1[6/2];
7.  #elif MUTA == 3
8.     char aux1[6*2];
9.  #elif MUTA == 4
10.    char aux1[1];
11. #else
12.    char aux1[6];
13. #endif
14. #if MUTA == 5
15.    char aux2[6-1];
16. #elif MUTA == 6
17.    char aux2[6/2];
18. #elif MUTA == 7
19.    char aux2[6*2];
20. #elif MUTA == 8
21.    char aux2[1];
22. #else
23.    char aux2[6];
24. #endif
25. strncpy (aux1,"teste",6);
26. strncpy (aux2,"teste",6);
27. printf ("\n%s",aux1);
28. printf ("\n%s",aux2);
29. }
```

Figura 4.8 – Código Fonte do Programa Mutante *_teste.c*

No caso da SEVMUT, o espaço necessário para armazenar o programa mutante-fonte é calculado com a seguinte equação:

$$E_t = (qtde * 10) + LoC$$

onde:

E_t : espaço total utilizado;

qtde : quantidade de alocações de memória;

LoC : total de linhas de código do programa em teste; e

10 : número de linhas de código acrescentadas ao programa mutante-fonte para cada alocação modificada.

Desta forma, mostra-se que o espaço utilizado pela SEVMUT para armazenar o programa mutante-fonte é menor que o utilizado com a técnica de geração de vários programas mutantes-fontes, pois para cada alocação de memória candidata à aplicação dos operadores de mutação aumenta-se somente 10 linhas de código no programa mutante-fonte.

MUTA é uma variável de ambiente que é criada pelo módulo Geresc e que armazena o número do mutante que está sendo compilado e executado no momento. MUTA armazenará o valor zero quando estiver compilando e executando o programa com a alocação de memória do programa original.

O inconveniente desta abordagem é que o mutante dever ser recompilado com o MUTA apropriado a cada execução. Isto também ocorreria se fosse criado um programa para cada mutante. Todos teriam que ser compilados para sua execução. Portanto, o tempo gasto para compilar todos os mutantes, nas duas abordagens, teoricamente é o mesmo.

Uma desvantagem da SEVMUT em relação à Ferramenta Proteum/IM[Del97] é justamente o fato desta ferramenta realizar o processo de compilação do programa mutante-fonte apenas uma vez, o que já não ocorre na SEVMUT. Esta abordagem é utilizada pela Proteum pelo fato desta ferramenta não realizar mutações nas declarações de variáveis. Desta forma, é possível a geração de um único programa mutante-fonte, com várias funções contendo o código correspondente a cada mutante e conseqüentemente a realização de uma única compilação.

De acordo com Delamaro [Del93] o custo da aplicação do critério Análise de Mutantes está justamente na compilação e execução dos mutantes, sendo a criação do código executável do mutante um ponto crítico no processo da execução. Entretanto como a quantidade de alocações de memória candidatas à aplicação dos operadores MSMA ou MDMA é pequena, este problema acaba não se tornando crítico na utilização da SEVMUT.

4.3.4 Módulo Geresc

O módulo Geresc é o principal módulo da SEVMUT. Este módulo compila o programa original e os mutantes, executa os programas com os dados de teste, compara os resultados obtidos, gera o escore de mutação para verificação da adequação do conjunto de dados de teste e gera alguns relatórios.

O módulo Geresc tem como entrada o programa mutante-fonte gerado pelo módulo Germut. No final da geração dos mutantes, o módulo Germut cria um arquivo, no mesmo diretório do programa fonte a ser testado, que armazena o total de mutantes gerados. Este arquivo é utilizado pelo módulo Geresc no controle da compilação e execução dos mutantes.

Antes da execução dos mutantes, o testador deve criar um arquivo contendo os dados de teste de entrada. Cada linha deste arquivo deve conter seqüencialmente os dados de entrada necessários para a execução de um caminho do programa original e dos mutantes.

Estes dados devem ser escolhidos pelo testador de forma que o caminho onde foi introduzida a mutação seja exercitado [DeM91]. Devem ser sensíveis o bastante para que se consiga levar os mutantes a um resultado da saída diferente do resultado do programa original, matando o mutante e com isto detectando a presença de *buffer overflow* através de uma falha de segmentação ou de um resultado diferente do esperado pelo programa original.

No caso de programas interativos é necessária a instalação de uma ferramenta do tipo *Capture* e *Playback* para armazenar e reproduzir os eventos do programa sob teste em um arquivo, podendo ser utilizado pela SEVMUT para entrada dos dados.

A compilação do programa original e a sua execução com cada linha de dados de teste incluída no arquivo de dados de teste são feitas de maneira automática, sem intervenção do testador. O resultado de cada execução é armazenado seqüencialmente em um arquivo, no qual cada linha corresponde ao resultado da execução do programa com uma linha do arquivo de dados de teste.

Cada mutante é compilado e executado com os mesmos dados de teste. O resultado de cada execução é confrontado com o resultado do programa original armazenado no arquivo citado acima e caso seja diferente, a ferramenta possibilita ao testador escolher marcar ou não o mutante como *morto*.

Marcar como *morto* já no primeiro dado de teste significa diferenciar o seu resultado com a saída do programa original e não continuar testando-o com os outros dados de teste. Não marcá-lo como *morto* significa executá-lo com todos os dados de teste, avaliando quais deles são capazes de matá-lo, utilizando esta informação para obter um conjunto mínimo necessário de dados de teste que elimine todos os mutantes.

Caso o resultado da execução do mutante seja igual ao do programa original, a tarefa de decidir a equivalência entre eles fica a cargo do testador, já que a verificação da equivalência entre programas é em geral um problema indecidível [Off97].

O controle da execução dos mutantes ocorre através da utilização de um *timeout*. Se o tempo de execução de um mutante exceder o *timeout*, a ferramenta aborta a sua execução e o considera como *morto*. O *timeout* tem como *default* o valor cinco, ou seja, corresponde a cinco vezes o tempo de execução do programa original, podendo este valor ser configurado.

Após executar o programa original e os mutantes com o conjunto de dados de teste, a ferramenta calcula o escore de mutação que indica o quanto este conjunto é adequado na detecção da presença de vulnerabilidade do tipo *buffer overflow* e caso o conjunto não seja adequado, novos dados de teste devem ser gerados para uma nova execução dos mutantes e do programa original. O escore de mutação fornece uma quantificação da qualidade do teste aplicado. A definição do valor aceitável para considerar a atividade de teste encerrada depende do ambiente de desenvolvimento utilizado, do produto desenvolvido, do orçamento e dos prazos de término do projeto.

A tarefa da geração dos dados de teste fica por conta do testador. Se um dado de teste mostra-se ineficiente, ou seja, não consegue eliminar nenhum mutante, não é possível que ele seja retirado dinamicamente do conjunto de dados de teste, ficando a critério do usuário modificar estaticamente este conjunto.

4.4 Aspectos de Utilização da SEVMUT

A SEVMUT só pode ser utilizada para a detecção da presença de vulnerabilidade do tipo *buffer overflow* em programas escritos em C de acordo com o padrão ANSI C.

O processo de teste de um sistema de acordo com Pressman [Pre04] engloba as seguintes fases:

- teste de unidade;
- teste de integração;
- teste de validação; e
- teste de sistema.

Na fase de teste de unidade são utilizados vários critérios de teste estrutural ou funcional para eliminar os defeitos do software. Testar unidades geralmente é a primeira fase a ser realizada durante o processo de teste. A SEVMUT deve ser utilizada para detectar a presença de vulnerabilidade do tipo *buffer overflow* na última etapa do teste durante a fase de teste de unidade.

A ferramenta possibilita ao usuário gerar os mutantes, executar os mutantes, alterar o estado dos mutantes após a sua geração, obter um conjunto de dados de teste eficaz na eliminação dos mutantes, visualizar e imprimir relatórios.

A primeira atividade a ser executada pelo testador no início dos testes é a geração dos mutantes através da aplicação do operador MSMA, do operador MDMA ou dos dois simultaneamente ao programa em teste.

Pelo fato da ferramenta não utilizar uma base de dados para armazenar as informações necessárias para execução dos mutantes e pelo fato da aplicação de um novo operador no programa sendo testado levar à eliminação de todas as tabelas já geradas até o momento pela SEVMUT, recomenda-se que seja criado um subdiretório para cada operador de mutação, de forma a preservar os resultados obtidos. Este diretório deve conter inicialmente o programa fonte a ser testado e o arquivo com os dados de teste de entrada, cuja geração é descrita no módulo Geresc. Neste diretório são criados o programa mutante-fonte, o programa mutante e também todas as tabelas utilizadas durante o teste do programa.

Uma das tabelas criadas na geração dos mutantes é a que armazena o estado de cada mutante criado e que inicialmente tem o seu estado igual a *vivo.* Um mutante pode ter um estado igual a *vivo*¸ *morto* ou *equivalente*. A SEVMUT permite que esta tabela seja modificada pelo usuário alterando o estado dos mutantes. A ferramenta possibilita mudar o estado para *equivalente*, somente se o mutante estiver no estado *vivo* e mudar para *vivo*, somente se estiver no estado *equivalente*. A cada mudança de estado é gerado um novo escore de mutação.

A execução dos mutantes não ocorre automaticamente após a sua geração, devendo ser solicitada pelo testador. Isto ocorre pelo fato da ferramenta permitir que o usuário faça o

levantamento de quais mutantes são eliminados por um dado conjunto de dados de teste. Desta forma, o testador consegue obter um conjunto mínimo de dados de teste mais adequado na eliminação dos mutantes gerados antes destes serem executados.

Dado um determinado mutante a ferramenta permite ao testador obter quais os dados de teste de um conjunto de dados de teste eliminam o dado mutante. Os resultados destes levantamentos são armazenados em arquivos no diretório do programa mutante-fonte podendo ser consultados pelo testador.

Estes levantamentos são feitos sem que seja alterado o estado do mutante. Um mutante já *morto* também é executado com o conjunto de dados de teste fornecido. Isto porque este conjunto pode ser mais eficiente que os outros já utilizados, eliminando tanto os mutantes *vivos* quanto os mutantes já *mortos* por outros conjuntos. Já os mutantes *equivalentes* não são executados. Para que isto ocorra, o testador deve mudar o estado destes mutantes para *vivo* antes dos levantamentos citados acima.

Além destas atividades, a SEVMUT permite ao testador visualizar e imprimir os relatórios gerados com as atividades citadas acima, o programa mutante-fonte e também a tabela que contém o estado de cada mutante.

4.5 Validação da SEVMUT

Este experimento tem como objetivo verificar a aplicabilidade e a escalabilidade da ferramenta, avaliando a sua performance utilizando o tempo de geração dos mutantes para programas de tamanhos variados.

A hipótese é que a ferramenta apresentará boa escalabilidade, ou seja, o tempo de geração de mutantes é diretamente proporcional ao tamanho dos programas e ao número de alocações de memória de cada programa. A base para esta hipótese é que a SEVMUT processa da mesma maneira todos os programas fontes.

Para testar a hipótese planejou-se e conduziu-se um experimento. Para esta investigação empírica vários programas de tamanhos diferentes foram reunidos.

4.5.1 Descrição do Experimento

Questionou-se se o tempo de geração dos mutantes é linearmente proporcional ao tamanho do programa fonte e à quantidade de alocações de memória candidatas à aplicação dos operadores de mutação MSMA e MDMA.

O propósito deste experimento é validar a ferramenta através da aplicação da técnica descrita no Capítulo 3 a vários programas, verificando se o comportamento da SEVMUT em

relação ao tempo de geração dos mutantes ocorre de forma escalar, dependendo do tamanho dos programas e da quantidade de alocações de memória.

Para realizar este experimento foram selecionados 55 programas fontes reais para serem utilizados. Alguns destes programas foram escolhidos levando em consideração o fato de serem citados na literatura [Ker81] e já usados em outros experimentos [Cha91].

Foi utilizado também um sistema chamado SPACE, desenvolvido para a Agência Espacial Européia para configuração de antenas. O programa *sort* [Hae88] foi escolhido por ser muito utilizado na prática. O *cal,* um programa que mostra um calendário para um ano ou um mês específico, foi escolhido por ter sido utilizado em outro experimento conduzido por Wong [Won98]. Acrescentou-se a esta lista dois programas não usuais na literatura.

A relação das unidades e o sistema ao qual cada uma pertence é mostrada na Tabela C.2 do Apêndice C. Calculou-se para cada uma das unidades a quantidade de alocações estáticas, dinâmicas, a quantidade de alocações estáticas e dinâmicas e o LoC, número de linhas de código.

Obteve-se o tempo que a Ferramenta SEVMUT leva para gerar os mutantes para cada uma das unidades relacionadas na Tabela C.2 do Apêndice C.

Com estes dados pôde-se calcular o coeficiente de correlação entre o tempo de geração de mutantes e cada um dos dados calculados acima, avaliando na prática o custo da aplicação da técnica através da escalabilidade da ferramenta.

4.5.2 Métodos de Coleta de Dados

Os dados necessários para conduzir o experimento são:

- a quantidade de alocações estáticas;
- a quantidade de alocações dinâmicas;
- a quantidade de alocações estáticas e dinâmicas; e
- o LoC, número de linhas de código do programa.

Estes dados foram obtidos separadamente para cada unidade listada na Tabela C.2 do Apêndice C. Para calcular o LoC, todas as linhas do programa fonte foram contadas exceto aquelas em branco ou incluídas num comentário.

Estes dados foram obtidos, exceto o LoC, através da Ferramenta SEVMUT que fornece o número de mutantes gerados de acordo com o operador escolhido pelo testador. Através deste número consegue-se calcular a quantidade de alocações existentes em cada programa.

Outro dado necessário, que é o tempo utilizado pela Ferramenta SEVMUT para gerar os mutantes, foi obtido através da geração dos mutantes para cada um dos programas.

Como parâmetro de entrada da Ferramenta SEVMUT, utilizou-se a opção MSMA em programas que têm somente alocações estáticas na geração dos mutantes, como no caso de [Ker81] e nos casos de levantamento de tempo somente para este tipo de alocação, como no caso do SPACE e do *sort* que têm alocações estáticas e dinâmicas.

Todas as alocações dinâmicas das unidades do sistema SPACE são do tipo *malloc*, portanto utilizou-se sempre a opção MDMA e MALLOC como parâmetros de entrada da SEVMUT na geração de mutantes para estes programas.

Nas unidades que alocam memória tanto estática como dinamicamente, utilizou-se para a geração dos mutantes, os parâmetros de entrada BOTH e MALLOC da Ferramenta SEVMUT para levantamento da quantidade de alocações deste tipo. Programas com estas características pertencem ao sistema SPACE.

O programa *sort* também aloca memória estaticamente e dinamicamente. Entretanto, além do tipo de alocação *malloc*, há também o tipo *realloc*, portanto foi utilizado como parâmetro de entrada da ferramenta as opções MDMA e ALL para geração de mutantes nas alocações dinâmicas e as opções BOTH e ALL, nos casos de geração de mutantes para todos os tipos de alocações.

Para utilizar a SEVMUT não há necessidade de conhecer detalhes do código do programa a ser testado para a entrada dos parâmetros da ferramenta. Os parâmetros citados acima foram utilizados, pois já se conhecia previamente os tipos de alocações candidatas à aplicação dos operadores de mutação em cada programa utilizado no experimento. Mas isto não é um pré-requisito para a utilização da Ferramenta SEVMUT.

O processo de geração dos mutantes ocorreu durante 1000 vezes para cada programa, medindo-se o tempo utilizado pela ferramenta em cada execução. No final, calculou-se a média aritmética de todos os tempos para cada unidade. O tempo está em milissegundos e consta na Tabela C.2 do Apêndice C. Neste tempo está incluso o tempo que a ferramenta leva para fazer o *parsing* das unidades.

Para medir o tempo de geração dos mutantes, utilizou-se um PC Pentium 4, 2 GHz de *clock* e 256 Mbytes de memória RAM, executando Linux Red-Hat 9.0.

4.5.3 Condução do Experimento

Com os dados coletados calculou-se o coeficiente da correlação linear entre o tempo de geração dos mutantes e as quantidades de alocações de memória e o coeficiente da correlação entre o tempo de geração dos mutantes e os LoC's das unidades.

Com estes coeficientes consegue-se avaliar na prática a performance da Ferramenta SEVMUT, verificando a relação entre o tempo de geração dos mutantes e o tamanho dos dados.

O coeficiente da correlação linear ρ_{xy} é calculado através da equação [Lap00]:

$$\rho_{xy} = \frac{Cov(X,Y)}{\sigma_x \sigma_y} \quad onde \quad -1 \leq \rho_{xy} \leq 1$$

$$Cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y)$$

onde:

$\sigma_x \sigma_y$ é o produto do desvio padrão de *x* pelo desvio padrão de *y*;

μ_x é a média aritmética de todos os pontos da coordenada *x*;

μ_y é a média aritmética de todos os pontos da coordenada *y*; e

$Cov(X,Y)$ é a covariância das variáveis *X* e *Y*, sendo que:

$X = x_1, x_2, x_3, ..., x_n$ representa os dados, e

$Y = y_1, y_2, y_3, ..., y_n$ representa o tempo de geração dos mutantes para cada dado.

A Tabela 4.1 mostra os coeficientes de correlação linear calculados para cada um dos dados em relação ao tempo de geração dos mutantes.

Dados	**Coeficiente de Correlação**
LoC	0,994288
Alocações Estáticas	0,905019
Alocações Dinâmicas	0,150927
Alocações Estáticas/Dinâmicas	0,987081

Tabela 4.1 - Correlação Tempo de Geração de Mutantes versus Dados

4.5.4 Análise dos Resultados

Analisou-se graficamente a correlação entre o tempo de geração dos mutantes e cada dado separadamente, avaliando o comportamento da ferramenta de acordo com os resultados obtidos.

4.5.4.1 Tempo de Geração versus LoC

O primeiro estudo foi realizado utilizando o LoC das unidades para verificar como a Ferramenta SEVMUT se comportava na geração de mutantes em função desta variável. Foram utilizados os LoC's de todas as unidades citadas na Tabela C.2 do Apêndice C. A Figura 4.9 e a Figura 4.10 mostram o resultado do comportamento da SEVMUT.

Como o coeficiente linear da Figura 4.9 é próximo de um, pode-se concluir que o tempo de geração dos mutantes é linearmente proporcional ao LoC das unidades.

A Figura 4.10 expande os pontos na escala dos LoC's entre 0 e 200 e, os pontos na escala do Tempo em ms entre 0 e 6 da Figura 4.9, com o objetivo de visualizá-los melhor dentro desta faixa.

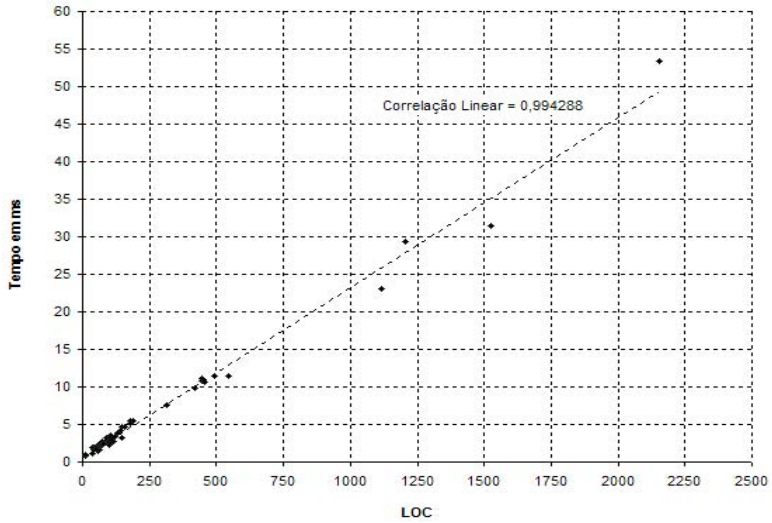

Figura 4.9 – Tempo de Geração de Mutantes versus LoC

4.5.4.2 Tempo de Geração versus Alocações Estáticas

O segundo estudo sobre a escalabilidade da Ferramenta SEVMUT foi realizado utilizando o tempo de geração dos mutantes em função do número de alocações estáticas das unidades. A Figura 4.11 retrata o desempenho da SEVMUT. Neste gráfico foram incluídas todas as unidades citadas na Tabela C.2 do Apêndice C que têm alocações estáticas.

Como se pode observar pela Figura 4.11, existe boa correlação linear entre as variáveis, tempo de geração e quantidade de alocações estáticas. Com isto, conclui-se que a geração dos mutantes tem uma progressão linear em relação ao tempo com o aumento das alocações estáticas das unidades.

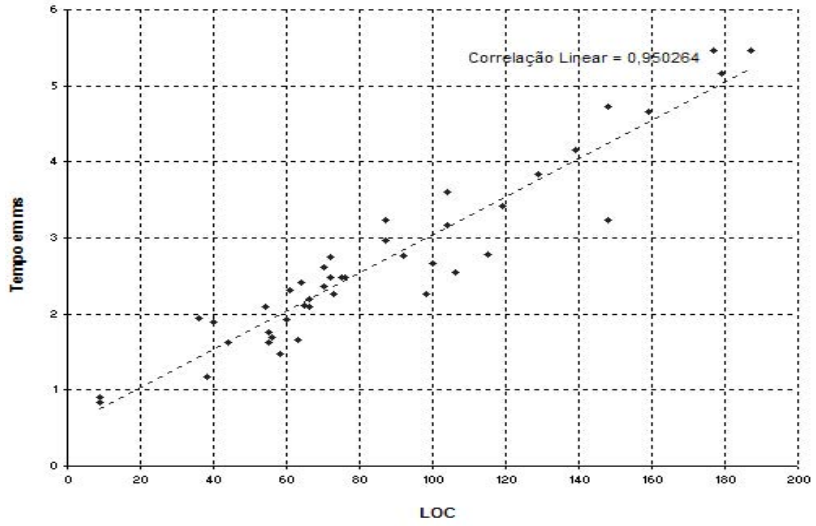

Figura 4.10 – Tempo de Geração de Mutantes versus LoC entre 0 e 200

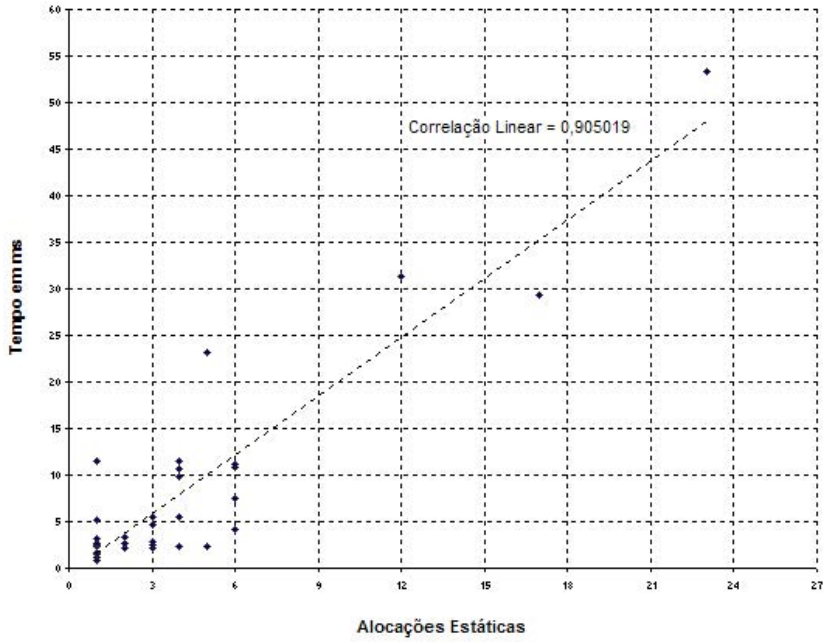

Figura 4.11 – Tempo de Geração de Mutantes versus Alocações Estáticas

4.5.4.3 Tempo de Geração versus Alocações Dinâmicas

O terceiro estudo sobre a performance da SEVMUT em relação à geração de mutantes foi realizado utilizando o tempo de geração em função do número de alocações dinâmicas das unidades. A Figura 4.12 sintetiza os resultados. Neste gráfico foram incluídas todas as unidades que têm alocações dinâmicas na Tabela C.2 do Apêndice C.

Como se pode observar, a correlação linear entre o tempo de geração dos mutantes e a quantidade de alocações dinâmicas é próximo de zero. Isto ocorre porque no tempo de geração dos mutantes também está incluso o tempo de *parsing* das unidades. Portanto, quando o LoC de uma unidade for grande, o seu tempo de geração dos mutantes será maior que de outras unidades com LoC's menores, mesmo que estas unidades tenham uma quantidade maior de alocações.

Sendo assim, a quantidade de alocações estáticas ou dinâmicas, não é o único fator que influencia no tempo de geração dos mutantes. O LoC das unidades também é uma variável que exerce influência.

Estas duas variáveis influenciam no aumento do tempo de geração dos mutantes por motivos diferentes. O LoC influencia no tempo de geração devido ao tempo utilizado com o *parsing* das unidades e a quantidade de alocações estáticas ou dinâmicas influencia devido ao tempo de geração das alocações mudadas.

No conjunto de unidades com alocações dinâmicas utilizadas para geração do gráfico da Figura 4.12, consta o programa *sort*, que tem 1523 linhas de código e somente duas alocações dinâmicas e constam outros programas, com LoC's menores e com quantidade maior de alocações. O tempo de geração dos mutantes destas unidades é menor que o do programa *sort.* Devido a isto, a correlação linear cai.

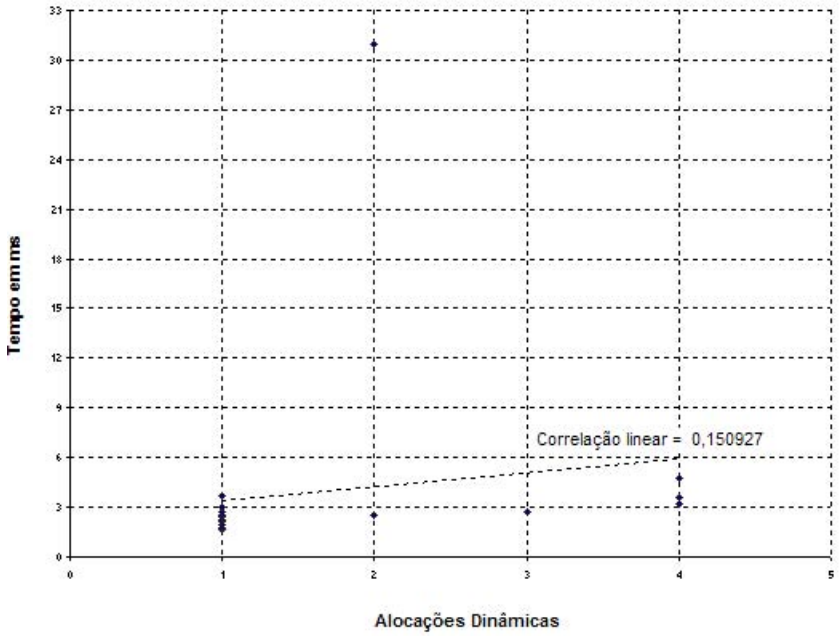

Figura 4.12 – Tempo de Geração de Mutantes versus Alocações Dinâmicas

Um outro estudo foi realizado utilizando as seguintes variáveis: as alocações dinâmicas, o tempo de geração dos mutantes e os LoC's das unidades. Neste estudo fixou-se o número de alocações dinâmicas igual a um e variou-se o tamanho dos programas para ver o quanto este influencia no tempo de geração dos mutantes.

A Figura 4.13 mostra os resultados. Como se pode observar, a correlação linear entre o tempo de geração e os LoC's das unidades está próximo de um. Com isto, prova-se que o LoC das unidades é um fator que influencia de forma significativa no tempo de geração dos mutantes.

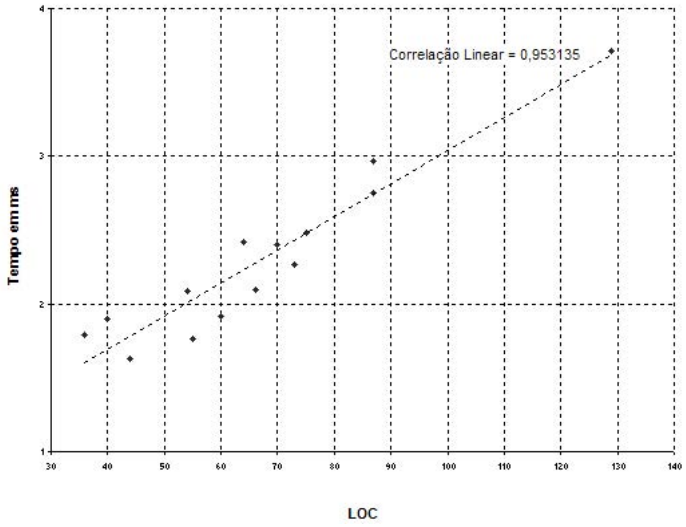

Figura 4.13 – Tempo de Geração de Mutantes versus LoC com uma Alocação Dinâmica

4.5.4.4 Tempo de Geração versus Alocações Estáticas e Dinâmicas

Um quarto estudo foi realizado utilizando as alocações estáticas e dinâmicas das unidades da Tabela C.2 do Apêndice C, para verificar como a Ferramenta SEVMUT se comportava na geração de mutantes em função destas duas variáveis.

A Figura 4.14 sintetiza os resultados. Como o coeficiente linear da Figura 4.14 é próximo de um, conclui-se que a geração dos mutantes tem uma progressão linear em relação ao tempo com o aumento das alocações estáticas e dinâmicas das unidades.

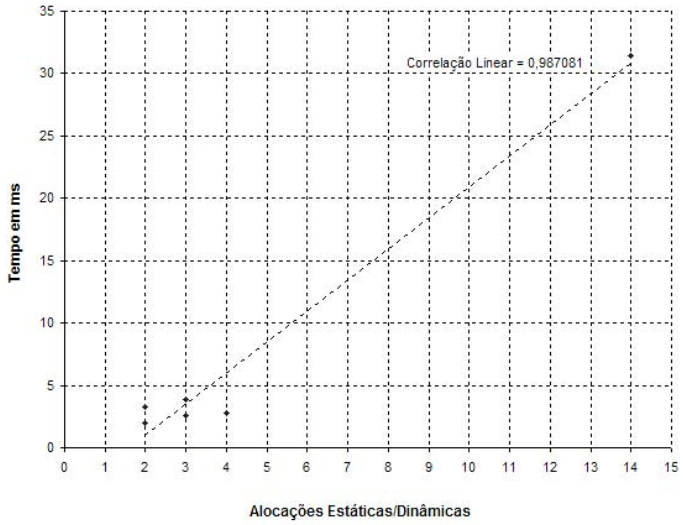

Figura 4.14 – Tempo de Geração de Mutantes versus Alocações Estáticas/Dinâmicas

4.5.5 Conclusão

Com este experimento, conclui-se que a Ferramenta SEVMUT apresenta boa escalabilidade, pois o tempo de geração dos mutantes é linearmente proporcional ao LoC das unidades e à quantidade de alocações tanto estática quanto dinâmica de cada programa.

4.6 Considerações Finais

Neste capítulo, foi apresentada a SEVMUT – ***Se****curity* ***V****ulnerabilities* ***Mu****tation* ***T****ool,* uma ferramenta desenvolvida para detecção da presença de vulnerabilidade do tipo *buffer overflow,* aplicando o critério Análise de Mutantes.

A SEVMUT foi desenvolvida para validar a técnica descrita no Capítulo 3 e tem como principal função a aplicação dos operadores MSMA e MDMA.

Esta ferramenta é composta basicamente de quatro módulos principais:

- o módulo Sevmut é o responsável pela interface da ferramenta com o usuário e tem como função a ativação de outros módulos da SEVMUT;
- o módulo Analex tem como principal função fazer a análise léxica do programa a ser modificado gerando uma seqüência de *tokens*;
- o módulo Germut tem como principais funções fazer a análise sintática do programa fonte, identificar as alocações estáticas e dinâmicas candidatas à aplicação dos operadores e gerar o programa mutante-fonte; e
- o módulo Geresc tem como funções compilar o programa original e os mutantes, executar estes programas com os dados de teste fornecidos pelo testador, gerar o escore de mutação para verificação da adequação do conjunto

de dados de teste e gerar os relatórios com informações sobre os mutantes eliminados pelo conjunto de dados de teste.

Realizou-se um experimento utilizando vários programas reais com o objetivo de avaliar a escalabilidade e aplicabilidade da Ferramenta SEVMUT.

Este experimento revelou que a Ferramenta SEVMUT tem boa escalabilidade, pois o tempo de geração dos mutantes é linearmente proporcional ao LoC das unidades e à quantidade de alocações tanto estática quanto dinâmica de cada programa.

No Capítulo 5 é apresentado um experimento com o objetivo de avaliar a eficácia da técnica na detecção da presença de vulnerabilidade do tipo *buffer overflow*.

Capítulo 5

Aspectos de Validação da Técnica

Neste capítulo apresenta-se um experimento conduzido com o objetivo de verificar a eficácia da técnica descrita no Capítulo 3, em relação a sua habilidade de detectar a presença de vulnerabilidade do tipo *buffer overflow*. Uma vez aplicados os operadores de mutação MSMA e MDMA ao programa a ser testado, os mutantes gerados e o programa original devem ser executados com dados de teste de entrada, de forma que os resultados das execuções indiquem ao testador a presença de vulnerabilidade do tipo *buffer overflow*.

Primeiramente, conduziu-se este experimento com o objetivo de verificar quais destes resultados indicariam ao testador a ocorrência de um *buffer overflow* no programa sendo testado. Partiu-se da hipótese de que a indicação de um provável *buffer overflow* no programa ocorre através de uma falha de segmentação ou através de um resultado da execução do mutante diferente do resultado da execução do programa original.

Esta hipótese é baseada no modo como ocorrem os ataques do tipo *buffer overflow* descritos no Capítulo 2. O atacante tem como objetivo principal, neste tipo de ataque, a mudança do endereço de retorno de uma função. De acordo com [CSD04], ocorre falha de segmentação quando o programa acessa um endereço de memória que não foi alocado para o seu uso.

Desta forma, a atribuição de uma informação ao *buffer* do programa sendo testado pode sobrescrever endereços de memória que, se alterados, levam o programa à falha, ou pode sobrescrever conteúdos de variáveis que levam o programa a fornecer resultados diferentes do esperado. Isto ocorre quando a informação tem um tamanho maior que o espaço reservado para armazená-la.

Um objetivo secundário na condução deste experimento é verificar a necessidade da aplicação de todas as variações dos operadores de mutação MSMA e MDMA às alocações de memória de um programa fonte. Neste experimento, partiu-se da premissa de que somente a variação que duplica a quantidade de memória alocada é suficiente para indicar ao testador a presença de uma vulnerabilidade do tipo *buffer overflow*.

Esta premissa é baseada no fato de que o testador, ao tentar eliminar mutantes com esta característica, utiliza informações de entrada de tamanho maior que o suportado pelo programa original, sobrescrevendo endereços de memória que se alterados levam o programa original à falha de segmentação, detectando portanto a presença de vulnerabilidade do tipo *buffer overflow*.

5.1 Projeto do Experimento

Planejou-se e conduziu-se esta investigação de acordo com a família de experimentos ilustrada na Figura 5.1.

Dividiram-se os experimentos em dois grupos:

- estático: cujo objetivo é realizar experiências aplicando o operador MSMA, e
- dinâmico: cujo objetivo é realizar experiências aplicando o operador MDMA.

Dividiu-se o grupo estático em duas categorias de experimentos:

- local: o objetivo dos experimentos nesta categoria é analisar os resultados das execuções do programa original e dos mutantes na aplicação do operador MSMA a variáveis declaradas localmente à função, e
- global: o objetivo dos experimentos nesta categoria é analisar os resultados das execuções do programa original e dos mutantes na aplicação do operador MSMA a variáveis declaradas globalmente no programa.

Dividiu-se a categoria local do experimento em três classes de experimentos:

- variáveis locais sem inversão: o objetivo dos experimentos desta classe é analisar os resultados das execuções do programa original e dos mutantes, quando a aplicação do operador MSMA ocorre em variáveis declaradas localmente à função com suas posições mantidas como as do programa em teste,
- variáveis locais invertidas: o objetivo dos experimentos desta classe é analisar os resultados das execuções do programa original e dos mutantes na aplicação do operador MSMA a variáveis locais, com suas posições alteradas nas declarações com referência ao programa em teste, e
- alteração de tamanho: o objetivo dos experimentos desta classe é analisar os resultados das execuções do programa original e dos mutantes na aplicação do operador MSMA a variáveis locais que tiveram os seus tamanhos alocados alterados com referência ao programa em teste.

Dividiu-se a categoria global em duas classes de experimentos:

- variável global: o objetivo dos experimentos desta classe é analisar os resultados das execuções do programa original e dos mutantes, quando a aplicação do operador MSMA ocorre em uma variável declarada globalmente em um programa fonte sem que o seu tamanho alocado tenha sido alterado, e
- alteração de tamanho: o objetivo dos experimentos desta classe é analisar os resultados das execuções do programa original e dos mutantes na aplicação do operador MSMA a variáveis globais que tiveram os seus tamanhos alocados alterados com referência ao programa em teste.

O grupo dinâmico tem como objetivo realizar experimentos aplicando o operador MDMA a variáveis declaradas dinamicamente dentro do programa, analisando os resultados obtidos na execução do programa original e dos mutantes.

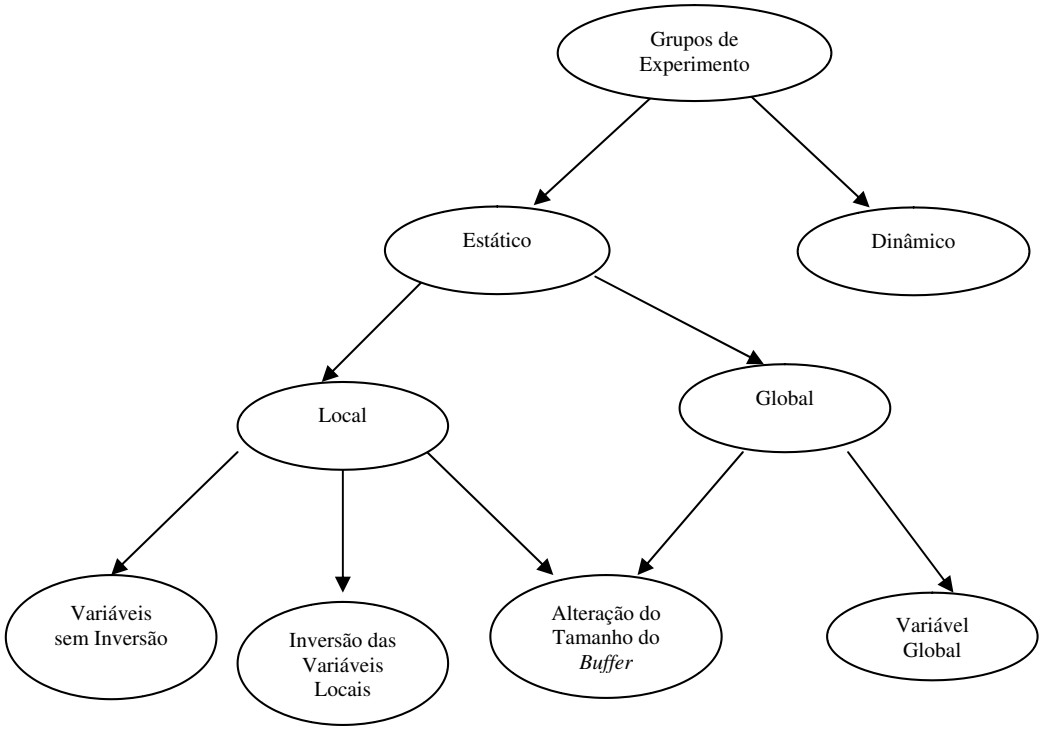

Figura 5.1 – Família de Experimentos

Para realizar a investigação empírica selecionou-se o programa fonte da Figura 5.2 escrito em C, obtido em [Col00]. A principal função deste programa é receber uma senha (*password*) como parâmetro, verificar a sua validade e dizer ao usuário se a senha está correta ou não.

Este programa apresenta uma vulnerabilidade do tipo *buffer overflow*. O *buffer* alocado localmente para armazenar a senha tem espaço para receber no máximo cinco caracteres. O armazenamento da senha no *buffer* ocorre por meio da função strcpy da linguagem C, que atribui a informação da variável de origem à variável de destino, sem verificar se a quantidade de bytes da informação de origem é maior que o tamanho em bytes suportado pela variável de destino.

Como a informação a ser armazenada no *buffer* tem origem externa ao sistema, este tipo de verificação é essencial para prevenir o *buffer overflow*. Se a senha fornecida pelo usuário ultrapassar cinco caracteres, ocorrerá transbordamento e os caracteres excedentes sobrescreverão as variáveis adjacentes ao *buffer* no *stack frame* da função.

Utilizou-se o programa da Figura 5.2 durante todas as etapas do experimento, alterando-o de forma a adequá-lo a cada fase da investigação. As modificações necessárias efetuadas neste programa são apresentadas em cada etapa da condução do experimento.

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;
     char Buffer[PWD_LEN + 1];

     strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
          iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
          return;
     if (IsPassValid(argv[1]))
          print ("Password Correto\n");
     else
          print ("Password Incorreto\n");
}
```

Figura 5.2 – Código do Programa Original

Para este experimento utilizou-se um PC Pentium 4, 2 GHz de *clock*, 256 Mbytes de memória RAM e sistema operacional Linux distribuição Red-Hat 9.0, para execução do programa original e dos mutantes gerados e utilizou-se o compilador gcc versão 3.2.2 para geração dos códigos executáveis destes programas.

5.2 Método de Geração de Dados

Os dados necessários para conduzir os experimentos são as senhas, que são utilizadas na execução do programa mutante resultante da aplicação dos operadores ao programa da Figura 5.2. Portanto, o domínio de entrada do programa são todos os caracteres que podem ser representados por números, letras ou caracteres especiais.

Para a geração destes dados de entrada utilizou-se o critério de geração aleatória de dados. Vários autores usam a geração aleatória para definir os dados de controle em experimentos na área de teste de software [Dur84]. Para isto, construiu-se um gerador aleatório implementado em C, utilizando o comando `rand()` para a geração de uma seqüência aleatória de caracteres entre 0x20 e 0x7e da tabela ASCII.

O intervalo entre 0x20 e 0x7e compreende os números, as letras maiúsculas e minúsculas e os caracteres especiais visíveis, representando desta forma todo o domínio de entrada do programa sendo testado. Gravou-se esta seqüência de caracteres em arquivos de dados de

teste, onde cada uma das linhas desses arquivos contém uma senha de entrada de tamanho e caracteres variados.

Foram criados seis conjuntos de dados de teste de entrada nomeados dtpass0, dtpass1, dtpass2, dtpass3, dtpass4, e dtpass5, que constam do Apêndice D, com o objetivo de realizar esta investigação empírica. A criação destes conjuntos ocorreu durante a condução de cada fase do experimento. Especificou-se o tamanho das senhas contidas nestes arquivos de acordo com a necessidade de cada etapa realizada, explicado durante a condução do experimento.

5.3 Condução do Experimento

Como já mencionado, dividiu-se o experimento em dois grupos: estático e dinâmico. No Apêndice E descrevem-se detalhadamente os pontos importantes da condução dos experimentos realizados em cada uma das classes de cada grupo. Nesta seção, são descritos os resultados obtidos com os experimentos em cada classe separadamente.

5.3.1 Grupo Estático

O objetivo é conduzir o experimento aplicando o operador MSMA ao programa da Figura 5.2, analisando os resultados das execuções dos mutantes. A hipótese é que a indicação de um provável *buffer overflow* no programa ocorre através de uma falha de segmentação ou através de um resultado da execução do mutante diferente do resultado da execução do programa original.

Como já mencionado, dividiu-se o grupo estático em duas categorias: local e global. A categoria local foi dividida em três classes de experimentos: variáveis locais sem inversão, variáveis locais invertidas e alteração de tamanho. A categoria global foi dividida em duas classes de experimentos: variáveis globais e alteração de tamanho.

Nesta seção, o termo:

- *mutante 1* representa o mutante resultante da aplicação da variação do operador MSMA que aloca quantidade de memória menos um ao programa em teste;
- *mutante 2* representa o mutante resultante da aplicação da variação do operador MSMA que aloca quantidade de memória dividida por dois ao programa em teste;
- *mutante 3* representa o mutante resultante da aplicação da variação do operador MSMA que aloca quantidade de memória multiplicada por dois ao programa em teste; e
- *mutante 4* representa o mutante resultante da aplicação da variação do operador MSMA que aloca quantidade de memória igual a um ao programa em teste.

Seguem-se os resultados obtidos em cada uma das classes de experimentos, dentro das categorias local e global do grupo estático da família de experimentos.

5.3.1.1 Categoria Local

O objetivo desta categoria de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador de mutação MSMA e suas variações ocorrem numa variável declarada localmente à função.

a. Variáveis Locais sem Inversão

Nesta classe de experimentos o operador MSMA é aplicado ao programa da Figura 5.2 sem que este seja modificado, analisando os resultados das execuções. Os detalhes dos experimentos realizados nesta classe constam do Apêndice E, seção E.1.1.1.

Com os resultados obtidos com este experimento, conclui-se que os mutantes 1, 2 e 3 são equivalentes ao programa original, pois a quantidade de memória alocada pelo compilador para as variáveis locais no *stack frame* é a mesma para o programa original e para cada um destes mutantes. Portanto, não se consegue detectar a presença de vulnerabilidade do tipo *buffer overflow*, pois a execução do programa mutante com qualquer dado de teste leva os mutantes 1, 2 e 3 ao mesmo resultado do programa original, pelo fato dos mutantes e do programa original terem a mesma quantidade de memória para ser invadida. Portanto, se o programa original falha, estes mutantes também falham, não podendo ser eliminados.

O mutante 4 é o único que detecta a presença de *buffer overflow*, pois a quantidade de memória alocada no *stack frame* da função deste mutante para as variáveis locais é diferente da quantidade alocada para o programa original. Desta forma, a execução do programa mutante com dados de teste dentro do tamanho esperado pelo programa original, leva o mutante 4 a detectar *buffer overflow* através de um resultado diferente do esperado pelo programa original ou através de falha de segmentação ocorrida neste mutante.

Portanto, o único operador de mutação MSMA eficaz na detecção da presença de vulnerabilidade do tipo *buffer overflow,* através de um resultado diferente do esperado pelo programa original ou através de falha de segmentação, é o que aloca quantidade de memória igual a um para uma variável local.

b. Variáveis Locais Invertidas

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada localmente à função, cuja posição foi alterada nas declarações do programa da Figura 5.2.

Trocaram-se as posições das declarações das variáveis locais dentro do programa ilustrado na Figura 5.2, para analisar o comportamento dos mutantes com o objetivo de verificar se a mudança altera a quantidade de memória alocada para as variáveis locais e se ocasiona alguma

modificação nos resultados obtidos com os experimentos já realizados na classe de experimentos "Variáveis Locais sem Inversão". Os detalhes dos experimentos realizados nesta classe constam do Apêndice E, seção E.1.1.2.

Conclui-se com esta classe de experimentos, que a inversão das declarações das variáveis locais não alterou os resultados obtidos com os experimentos realizados na classe "Variáveis Locais sem Inversão".

Como a quantidade de memória alocada para as variáveis locais no *stack frame* das funções continua a mesma para o programa original e para os mutantes 1, 2 e 3, permanece a equivalência entre estes mutantes e o programa original.

Portanto, a execução do programa mutante com qualquer dado de teste leva estes mutantes ao mesmo resultado do programa original, não detectando a presença de vulnerabilidade do tipo *buffer overflow* através de algum resultado diferente do programa original, caso exista.

Como as variáveis ficaram distribuídas em outra ordem na memória principal, alguns resultados não ocorrem como o "Password Correto", pois a variável iRet está abaixo da variável Buffer no *stack frame* da função IsPassValid, portanto nenhum strcpy altera o seu valor.

Desta forma, o mutante 4 é o único que detecta a presença de vulnerabilidade do tipo *buffer overflow,* através da ocorrência de falha de segmentação na sua execução com dados de teste de tamanho dentro do esperado pelo programa original, pois a quantidade de memória alocada no *stack frame* da função deste mutante é diferente da quantidade alocada para o programa original.

c. Alteração do Tamanho do *Buffer*

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada localmente à função, cujo tamanho em bytes é modificado em relação ao programa da Figura 5.2.

Questionou-se se independentemente do tamanho da variável Buffer, a aplicação do operador MSMA que aloca quantidade de memória menos um, quantidade dividida por dois e quantidade multiplicada por dois, leva o compilador a reservar sempre a mesma quantidade de memória para as variáveis locais do programa original e dos mutantes gerados com estas características.

Alterou-se o tamanho da variável Buffer de 6 caracteres para 31 caracteres do programa fonte esboçado na Figura 5.2 com o objetivo de comprovar a equivalência entre o programa original e os mutantes 1, 2 e 3, independentemente do tamanho do *buffer*. Os detalhes dos experimentos realizados nesta classe constam do Apêndice E, seção E.1.1.3.

Verificou-se a quantidade de memória reservada no *stack frame* para armazenar as variáveis locais do programa original e dos mutantes e percebeu-se que esta quantidade varia de mutante para mutante, diferentemente do que ocorre quando o Buffer tem tamanho igual a 6 caracteres, como no caso das classes de experimentos "Variáveis Locais sem Inversão" e "Variáveis Locais Invertidas", nas quais a quantidade de memória reservada pelo compilador para as variáveis locais é a mesma para o programa original e para os mutantes 1, 2 e 3.

O mutante 1 é o único que continua equivalente ao programa original, reservando a mesma quantidade de memória que a deste programa.

Os resultados obtidos nesta classe de experimento são diferentes dos obtidos nas classes anteriores, que revelam que somente o mutante 4 é eficaz na detecção da presença de vulnerabilidade do tipo *buffer overflow*.

A execução do programa mutante com informações de entrada dentro ou fora do esperado pelo programa original, mostra que os mutantes 2 e 3 também são eficazes em detectar a presença de vulnerabilidade do tipo *buffer overflow* através de um resultado diferente do programa original ou de falha de segmentação, revelando que nem sempre estes mutantes são equivalentes ao programa original.

d. Conclusão sobre os Experimentos da Categoria Local

Conclui-se que, com a aplicação do operador MSMA às variáveis locais, dependendo do seu tamanho, ocorre uma variação na quantidade de memória reservada no *stack frame* da função. Como o sistema aloca a memória na pilha em *dwords*, o fato de as variáveis locais terem um tamanho pequeno pode levá-lo a reservar a mesma quantidade de memória para o programa original e para os mutantes, tornando-os equivalentes. Somente o mutante cuja quantidade alocada é igual a um sempre difere do programa original com referência à quantidade de memória reservada no *stack frame*. Se as variáveis locais têm um tamanho maior, a quantidade de memória reservada na pilha varia de mutante para mutante, não levando à equivalência entre eles. Somente o mutante que aloca quantidade de memória menos um continua equivalente ao programa original.

Tamanho *pequeno*, no caso do programa da Figura 5.2 utilizado neste experimento, corresponde a um *buffer* de tamanho entre 3 e 7 caracteres, ou seja $3 \leq \text{PWD\_LEN} \leq 7$. Para estes casos, o compilador reserva a mesma quantidade de memória para as variáveis locais no *stack frame* da função tanto para os mutantes gerados quanto para o programa original. Tamanho *maior* significa um *buffer* de tamanho acima de 7 caracteres para que o compilador comece a reservar quantidades diferentes de memória para o programa original e para os mutantes nos *stack frames* das funções.

Desta forma, quando o *buffer* tem um tamanho pequeno, somente o operador que aloca a quantidade de memória igual a um é eficaz na detecção de *buffer overflow* por meio da execução do mutante com esta característica, com dados de teste de entrada dentro do tamanho

esperado pelo programa original, revelando a eficácia da técnica através de falha de segmentação ou de um resultado diferente do esperado.

Quando o *buffer* tem um tamanho maior todas as variações do operador MSMA, menos a que aloca quantidade de memória menos um, são eficazes na revelação de *buffer overflow*. O operador que aloca quantidade de memória dividida por dois é eficiente em detectar a presença de *buffer overflow* na execução do mutante com esta característica com dados de teste tanto dentro quanto fora do tamanho esperado pelo programa original. O operador que aloca quantidade de memória multiplicada por dois detecta a presença de *buffer overflow* apenas se o tamanho da informação de entrada estiver acima do esperado pelo programa original levando-o à falha, mas, no entanto não deve ser grande o suficiente para levar o mutante com esta característica à falha também, o que torna custosa a geração de dados de teste. O operador que aloca quantidade de memória igual a um é eficaz na detecção da presença de *buffer overflow* também no caso do *buffer* ter um tamanho maior, revelando a eficácia da técnica através da execução do mutante com esta característica com dados de teste dentro do esperado pelo programa original.

Sendo assim, conclui-se que somente a utilização do operador MSMA que aloca quantidade de memória igual a um nas variáveis locais é eficaz na detecção da presença de um possível *buffer overflow* independentemente do tamanho do *buffer*. Além disso, a geração de dados de teste é menos custosa, pois a execução do programa mutante com informação dentro do esperado pelo programa original leva o mutante com esta característica a detectar a presença de vulnerabilidade do tipo *buffer overflow* através de falha de segmentação ou de um resultado diferente do esperado pelo programa original.

Desta forma, somente a aplicação da variação do operador MSMA que aloca quantidade de memória igual a um às variáveis locais parece ser suficiente na detecção da presença de *buffer overflow* no programa em teste, não sendo necessária também a aplicação das demais variações deste operador, pois com os resultados dos experimentos realizados neste trabalho, as outras variações parecem ser redundantes e ineficazes.

Estes resultados são válidos para o programa da Figura 5.2 utilizado neste experimento e executado em um ambiente específico, composto por um PC Pentium 4, 2 GHz de *clock* e 256 Mbytes de memória RAM, sistema operacional Linux Red Hat 9.0 e utilizando o compilador gcc versão 3.2.2 para geração do código executável. Como já mencionado, a quantidade de memória alocada para as variáveis locais no *stack frame* da função depende do compilador, da arquitetura da máquina e do sistema operacional utilizados, não sendo possível garantir a generalização dos resultados obtidos.

Os resultados acima só são válidos se o programador estiver utilizando funções da linguagem C no programa em teste que são suscetíveis ao *buffer overflow* como, por exemplo, as funções *strcpy*, *strcncpy*, *strcat*, *strncat* e *gets*. Caso contrário, a detecção da presença de *buffer overflow* pode ocorrer na execução do programa mutante com os dados de teste de entrada sem que realmente exista esta vulnerabilidade na aplicação.

5.3.1.2 Categoria Global

O objetivo desta categoria de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA e suas variações ocorrem numa variável declarada globalmente em um programa.

a. Variáveis Globais

Como o objetivo desta classe de experimentos é analisar o comportamento dos mutantes quando o operador MSMA é aplicado às variáveis globais, alterou-se o programa fonte da Figura 5.2 mudando a declaração da variável Buffer de local para global. Os detalhes dos experimentos realizados nesta classe constam do Apêndice E, seção E.1.2.1.

Como já mencionado no Capítulo 2, o segmento de dados bss ilustrado na Figura 2.3, é o segmento dentro da memória de um processo do Linux reservado para armazenar as variáveis globais não iniciadas. A Figura E.12 do Apêndice E, repetida aqui como Figura 5.3, ilustra o *layout* deste segmento.

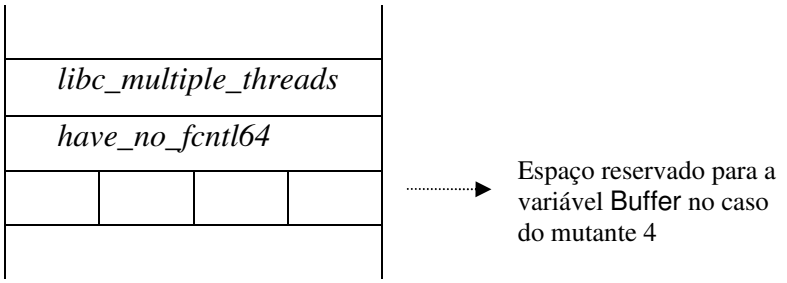

Figura 5.3 – *Layout* do Segmento de Dados bss

Conclui-se com os experimentos realizados que, quando o *buffer* tem tamanho pequeno, a quantidade reservada de *dwords* para os mutantes que alocam quantidade de memória menor que a do programa original é aproximadamente a mesma que a reservada para este programa. Isto ocorre devido a *dword have_no_fcntl64,* ilustrada na Figura 5.3, levar os mutantes 2 e 4 a resultados idênticos aos do programa original, pelo fato de poder ser sobrescrita com a informação armazenada no *buffer* sem que ocorra erro. Estes mutantes também mostraram-se equivalentes entre si, pois a quantidade de memória reservada no segmento de dados bss foi a mesma para ambos.

Desta forma, executá-los com dados de teste dentro do esperado pelo programa original não vai levá-los a indicar a presença de *buffer overflow* através de uma falha de segmentação, pois estes dados podem sobrescrever a *dword have_no_fcntl64* sem indicação de erro. A detecção da presença do *buffer overflow* através dos mutantes 2 e 4 só é possível se o tamanho do dado de entrada for maior que o suportado pelo programa original, não grande demais para levá-lo à falha de segmentação, mas grande o bastante para levar os mutantes 2 e 4 à falha devido a

sobrescrita da *dword libc_multiple_threads*, que não pode ter em nenhum dos seus bytes um valor diferente de zero.

A detecção da presença de *buffer overflow* por meio do mutante 3 ocorre na execução do programa original com uma informação de tamanho acima do esperado, levando este programa à falha de segmentação devido a sobrescrita da *dword libc_multiple_threads*.

O mutante 1 novamente mostrou-se equivalente ao programa original, pois a quantidade de memória reservada para o *buffer* no segmento de dados bss, é a mesma reservada para o programa original.

Portanto, o operador MSMA que dobra a quantidade de memória alocada para a variável global é o mais eficaz na detecção da presença de vulnerabilidade do tipo *buffer overflow* por meio da ocorrência de falha de segmentação e também o que necessita de um menor número de dados de teste gerados no caso do *buffer* ter um tamanho pequeno.

b. Alteração do Tamanho do *Buffer*

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada globalmente em um programa, cujo tamanho em bytes foi modificado de 6 para 31 caracteres com referência ao programa utilizado nos experimentos realizados na classe "Variáveis Globais".

Quando o *buffer* é declarado globalmente e tem um tamanho igual a 6 caracteres, o mutante 1 é equivalente ao programa original. São equivalentes entre si o mutante 2 e o mutante 4. Modificou-se o tamanho do *buffer* do programa utilizado nos experimentos da classe anterior, com o objetivo de verificar a influência desta mudança nos resultados já obtidos, como ocorreu com o mesmo tipo de experimento na categoria local. Os detalhes dos experimentos realizados nesta classe constam do Apêndice E, seção E.1.2.2.

Conclui-se com os experimentos realizados que o mutante 1 é equivalente ao programa original pois a quantidade de memória reservada para o *buffer* no segmento de dados bss é a mesma reservada para o programa original.

O mutante 2 e o mutante 4 indicam a presença de vulnerabilidade do tipo *buffer overflow* através da execução do programa mutante com informação de entrada dentro do tamanho esperado pelo programa original, levando estes mutantes a detectarem a presença de vulnerabilidade do tipo *buffer overflow* através de falha de segmentação, diferentemente do que ocorre quando o *buffer* tem um tamanho menor.

O mutante 3 detecta a presença de vulnerabilidade do tipo *buffer overflow* executando o programa mutante com informação de entrada acima do esperado pelo programa original, levando-o à falha de segmentação.

c. Conclusão sobre os Experimentos da Categoria Global

Conclui-se com os experimentos que a dificuldade de levar os mutantes a detectarem a presença de vulnerabilidade do tipo *buffer overflow* está relacionada com o tamanho da variável sendo testada. Se o *buffer* tem um tamanho pequeno, a quantidade reservada em *dwords* acaba sendo a mesma para os mutantes e o programa original. Isto pelo fato de existir uma *dword* adjacente a *dword* que armazena o *buffer* no segmento de dados bss, que pode ser sobrescrita sem que ocorra erro.

Desta forma, os mutantes que alocam quantidade de memória menor que a do programa original, acabam tendo a mesma quantidade de memória que a deste programa. A execução destes mutantes com dados de teste dentro do esperado pelo programa original, faz com que não revelem a presença de *buffer overflow*, caso exista. O mutante 1 revelou-se equivalente ao programa original. Os mutantes 2 e 4 mostraram-se equivalentes entre si.

A eliminação dos mutantes 2 e 4 quando a variável tem um tamanho pequeno acaba sendo custosa, pois a informação tem que ser maior que o esperado pelo programa original, desta forma levando estes mutantes à falha de segmentação, mas não grande o suficiente para levar o programa original também à falha. O mutante 3 é eliminado através de dados de teste que levam o programa original à falha de segmentação, detectando com isto a presença de *buffer overflow*.

Se o *buffer* tem um tamanho maior, o mutante 1 revela-se equivalente ao programa original. A eliminação do mutante 3 também ocorre através do uso de dados de teste que levam o programa original à falha de segmentação, detectando com isto a presença de *buffer overflow*. Os mutantes 2 e 4 são eliminados com dados de teste dentro do tamanho esperado pelo programa original. Neste caso, a eliminação destes mutantes é menos custosa do que no caso em que o *buffer* tem tamanho pequeno.

Sendo assim, conclui-se que somente a utilização do operador MSMA que dobra a quantidade de memória alocada para as variáveis globais é eficiente na detecção da presença de um possível *buffer overflow* independentemente do tamanho do *buffer.* Além disso, a geração de dados de teste é também menos custosa, pois a execução do programa original com a informação fora do esperado leva-o a detectar a presença de vulnerabilidade do tipo *buffer overflow* através de falha de segmentação.

Desta forma, somente a aplicação da variação do operador MSMA que dobra a quantidade de memória alocada às variáveis globais para geração dos mutantes parece ser suficiente na detecção da presença de *buffer overflow* no programa em teste, não sendo necessária também a aplicação das demais variações deste operador, pois com os resultados dos experimentos realizados neste trabalho, as outras variações parecem ser redundantes.

Mas questiona-se se um intruso utilizaria uma variável global, mais precisamente um vetor de caracteres, para realizar um ataque *buffer overflow* do tipo *stack smashing* no sistema. Uma

variável global está alocada no segmento de dados bss ou no segmento de dados data da memória principal em um endereço bem abaixo dos endereços onde estão alocados os *stack frames* das funções, como ilustrado na Figura 2.3 do Capítulo 2. Próximos a estes segmentos não há nenhuma *dword* %ebp ou %eip armazenadas, que são os alvos dos atacantes para concretizar um ataque *buffer overflow* deste tipo. Para atingir este objetivo, os atacantes teriam que entrar com uma informação de tamanho suficiente para sobrescrever estas *dwords* no *stack frame* de uma função, tornando este tipo de ataque quase impossível.

Assim, o operador MSMA, que dobra a quantidade de memória alocada das variáveis globais, poderia ser utilizado com o objetivo de verificar se o programa sendo testado tem defeitos de *buffer overflow* que podem ser explorados para modificar ponteiros de funções armazenados no segmento bss, como descrito no terceiro tipo de ataque *buffer overflow* apresentado no Capítulo 2. Um atacante poderia explorar uma vulnerabilidade do tipo *buffer overflow* em uma variável global também com o intuito de elaborar um ataque do tipo *denial of service* no sistema. O ataque do tipo *denial of service* tem como objetivo consumir todos os recursos da aplicação deixando-a inoperante [For01].

5.3.1.3 Conclusão sobre o Grupo Estático

Com os experimentos realizados dentro das categorias local e global do grupo estático da família de experimentos, conclui-se que a detecção da presença de vulnerabilidade do tipo *buffer overflow* ocorre através da execução do programa mutante com informações que o levam à falha de segmentação ou a um resultado diferente do esperado pelo programa original.

Como já mencionado, estes resultados indicam presença de vulnerabilidade do tipo *buffer overflow* apenas se o programador estiver usando funções da linguagem C que são suscetíveis a este tipo de problema, tais como *strcpy, strncpy, strcat, strncat* e *gets*. Caso contrário, a detecção da presença de *buffer overflow* pode ocorrer na execução do programa mutante com os dados de teste de entrada, sem que realmente exista esta vulnerabilidade na aplicação.

Conclui-se com os experimentos realizados neste trabalho, que não há necessidade de aplicar todas as variações de mutação do operador MSMA descritas no Capítulo 3 ao programa em teste. A variação que aloca quantidade de memória menos um mostra-se ineficiente na detecção da presença de *buffer overflow*, pelo fato da sua aplicação tornar o mutante gerado equivalente ao programa original.

No caso da aplicação do operador MSMA a variáveis declaradas localmente à função, somente a variação que aloca quantidade de memória igual a um é eficaz na detecção da presença de *buffer overflow,* através da execução do programa mutante com informação de tamanho dentro do esperado pelo programa original, levando o mutante com esta característica a sobrescrever o endereço de retorno da função, ocorrendo falha de segmentação ou mesmo um resultado diferente do esperado pelo programa original.

No caso da aplicação do operador MSMA a variáveis declaradas globalmente no programa, somente a variação que dobra a quantidade alocada de memória é eficaz na detecção da

presença de *buffer overflow,* através da execução do programa mutante com informação de entrada acima do esperado pelo programa original, levando este programa à falha de segmentação.

Como já mencionado, estes resultados são válidos para os experimentos realizados neste trabalho com o programa da Figura 5.2, executados dentro de um ambiente específico já descrito anteriormente. Desta forma, não é possível garantir a generalização dos resultados, no entanto, os dados obtidos já indicam a validação da técnica.

5.3.2 Grupo Dinâmico

O objetivo é conduzir o experimento aplicando o operador MDMA a variáveis declaradas dinamicamente dentro do programa, analisando os resultados obtidos na execução do programa original e dos mutantes. A hipótese é que a detecção da presença de um provável *buffer overflow* no programa ocorra através de uma falha de segmentação ou através de um resultado da execução do mutante diferente do resultado da execução do programa original.

Alterou-se o programa fonte da Figura 5.2 de modo que a alocação de memória para a variável Buffer aconteça de forma dinâmica. Os detalhes dos experimentos realizados neste grupo constam do Apêndice E, seção E.2.

A memória *heap* localiza-se perto da memória reservada para os *stack frames* das funções, como ilustrado na Figura 2.3 do Capítulo 2. Na *heap* aloca-se exatamente a memória que a variável precisa. Entretanto, a *heap* tem uma área grande de memória e totalmente livre. Sendo assim, não importa o tamanho da informação de entrada que o resultado da execução do programa mutante será sempre o mesmo dado pelo programa original, não detectando a presença de nenhuma ocorrência de *buffer overflow*.

Conclui-se que a aplicação da técnica às variáveis dinâmicas, utilizando o programa da Figura 5.2 modificado, não se mostrou eficaz na detecção da presença de um *buffer overflow*. Necessita-se realizar mais experimentos com a aplicação do operador MDMA a outros tipos de programas mais adequados para comprovar a eficácia da técnica com referência a este operador e suas variações.

5.4 Implementação Segura

Uma maneira de implementar o programa da Figura 5.2 de forma que não ocorra *buffer overflow* é ilustrada na Figura 5.4.

Somente a utilização da função *strncpy* da linguagem de programação C não é suficiente para evitar a ocorrência de *buffer overflow*. Esta função, quando atribui a informação da origem à variável de destino, não verifica se o tamanho desta variável é suficiente para

armazenar a informação de origem. Sendo assim, se o tamanho da informação for maior que o tamanho da variável destino ocorrerá *buffer overflow*.

No programa da Figura 5.4 verifica-se se a informação a ser armazenada tem um tamanho menor que o tamanho da variável Buffer, antes da atribuição. Desta forma, evita-se a ocorrência de um *buffer overflow*.

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
 int iRet = FALSE;
 char Buffer[PWD_LEN + 1];
 if (PWD_LEN >= strlen(szPWD))
    strncpy (Buffer,szPWD,strlen(szPWD)+1);
 else
 {
   printf ("Ocorreria Buffer Overflow em Buffer\n");
   exit (1);
 }
 if (strcmp (Buffer,PWD)==0)
  iRet = TRUE;
 return iRet;
}

int main (int argc, char *argv[])
{
 if (argc==1)
  return;
 if (IsPassValid(argv[1]))
  printf ("Password Correto\n");
 else
  printf ("Password Incorreto\n");
}
```

Figura 5.4 - Código Seguro do Programa Fonte

Capítulo 6

Conclusão

6.1 Síntese do Trabalho

Neste trabalho foi estudada, implementada e analisada uma técnica denominada "Teste de Vulnerabilidade de Segurança" com objetivo de detectar a presença de vulnerabilidades de software por meio da aplicação de Análise de Mutantes, a programas fontes implementados utilizando a linguagem de programação C.

Existem vulnerabilidades que ocorrem devido a defeitos de software que se revelam no código fonte. Estes defeitos criam pontos vulneráveis na aplicação, que, se explorados de forma apropriada, forçam o programa a produzir resultados incorretos com comportamentos diferentes do inicialmente esperado.

Se estas vulnerabilidades puderem ser modeladas com operadores de mutação, então é possível testar o programa para vulnerabilidades de segurança. Desta forma, uma vulnerabilidade de segurança para ser candidata à aplicação da técnica, deve ser originada por um defeito de software, que cria um ponto vulnerável no programa e que pode ser modelado com operadores de mutação.

A abordagem proposta concentra-se na fase de desenvolvimento do sistema, conduzindo o teste de forma a remover o máximo possível de vulnerabilidades antes de o software entrar em operação. A hipótese adotada supõe ser possível executar tais testes e conseqüentemente remover tais vulnerabilidades.

O *buffer overflow* é uma vulnerabilidade candidata a ser tratada pela abordagem proposta, pois ocorre devido a um defeito do código fonte e que pode ser eliminado através de teste estrutural.

Para eliminar o *buffer overflow* existente nos programas C, foram criados dois operadores de mutação, MSMA - ***M****odification of* ***S****tatic Memory* ***A****llocation* e MDMA - ***M****odification of* ***D****ynamic* ***M****emory* ***A****llocation*. Cada operador tem quatro variações:

- quantidade de memória alocada menos um;
- quantidade de memória alocada dividida por dois;
- quantidade de memória alocada multiplicada por dois;
- quantidade de memória alocada igual a um byte.

Após a aplicação dos operadores de mutação MSMA e MDMA ao programa em teste, o programa original e os mutantes gerados são executados com dados de teste de entrada. Os

resultados das execuções podem indicar ao testador a presença de vulnerabilidade do tipo *buffer overflow* na aplicação, caso exista.

Para validar a técnica, foi desenvolvida uma ferramenta denominada SEVMUT - ***Se****curity* ***V****ulnerabilities* ***Mu****tation* ***T****ool*. Esta ferramenta tem como principal função a aplicação dos operadores MSMA e MDMA aos programas escritos em C e a detecção da presença de possíveis *buffer overflows* nas aplicações.

A ocorrência de falha de segmentação ou um resultado diferente do esperado pelo programa original são os resultados que podem indicar ao testador a detecção da presença de vulnerabilidade do tipo *buffer overflow.*

Os experimentos foram divididos em duas fases: a primeira, com o objetivo de avaliar a escalabilidade e aplicabilidade da Ferramenta SEVMUT utilizando o tempo de geração dos mutantes e a segunda, com o objetivo de validar a técnica em relação a sua eficácia na detecção da presença de vulnerabilidade do tipo *buffer overflow*.

Na primeira fase, os experimentos foram realizados utilizando vários programas fontes com LoC's (*Lines of Code)* variados e com quantidades diferentes de alocações de memória. Neste experimento, utilizou-se a Ferramenta SEVMUT para aplicar a técnica a estes programas, calculando o tempo de geração dos mutantes para cada um. Os resultados revelaram que a Ferramenta SEVMUT tem boa escalabilidade, pois o tempo de geração dos mutantes é linearmente proporcional ao LoC das unidades e à quantidade de alocações tanto estática quanto dinâmica de cada programa.

A segunda fase do experimento foi conduzida através de um estudo de caso, com o propósito de validar a eficácia da técnica. Este estudo de caso aplicou a técnica, através da SEVMUT, a um programa que basicamente recebe uma senha como parâmetro de entrada, verificando a sua validade. Os resultados deste experimento comprovaram que a detecção da presença de vulnerabilidade do tipo *buffer overflow* no programa sendo testado ocorre através de falha de segmentação ou de um resultado diferente do esperado pelo programa original.

Com referência ao operador MDMA, os experimentos realizados não foram suficientes para confirmar a eficácia da técnica na detecção da presença de vulnerabilidade do tipo *buffer overflow* no uso de variáveis dinâmicas, pois o programa utilizado não era apropriado para a aplicação deste operador.

6.2 Contribuições

As principais contribuições desse trabalho são resumidas abaixo:

1. estudo, implementação e análise de uma técnica denominada "Teste de Vulnerabilidade de Segurança", utilizada na detecção da presença de vulnerabilidades de software, por meio da aplicação do critério Análise de Mutantes a programas

escritos em C, para ajudar a identificar a presença de tais vulnerabilidades antes do software entrar em operação;

2. estudo de um conjunto de operadores utilizados na detecção da presença de vulnerabilidade do tipo *buffer overflow*. Com a aplicação destes operadores ao programa, o usuário poderá identificar através dos resultados dos testes, se constam vulnerabilidades deste tipo no programa em teste, eliminando-as antes de o software entrar em operação e conseqüentemente, aumentando a confiança de que o software está livre de vulnerabilidades;

3. especificação e implementação de uma ferramenta que automatiza a aplicação da técnica "Teste de Vulnerabilidade de Segurança". A Ferramenta SEVMUT - ***Se****curity* ***V****ulnerabilities* ***Mu****tation* ***T****ool* foi desenvolvida para apoiar a aplicação de Análise de Mutantes à detecção da presença de vulnerabilidade do tipo *buffer overflow*. Dado um programa fonte escrito em C, a ferramenta aplica os operadores de mutação MSMA/MDMA ao programa em teste gerando um programa mutante-fonte, que é executado com dados de teste de entrada.

4. condução de experimentos com o objetivo de validar a Ferramenta SEVMUT e verificar a eficácia da técnica na sua habilidade em detectar a presença de vulnerabilidade do tipo *buffer overflow*. Os resultados indicam que a Ferramenta SEVMUT apresenta boa escalabilidade, pois o tempo de geração dos mutantes é linearmente proporcional ao LoC das unidades e à quantidade de alocações tanto estática quanto dinâmica de cada programa. Os resultados indicam também que para o operador MSMA, a variação que dobra a quantidade de memória alocada para as variáveis e a que aloca quantidade de memória igual a um, são as mais eficazes na detecção da presença de vulnerabilidade do tipo *buffer overflow*;

5. a ferramenta pode ser utilizada durante a fase de teste de uma aplicação, complementando os testes já efetuados com o objetivo de eliminar os seus defeitos através de técnicas estruturais e funcionais, com isto aumentando a confiança de que o software está livre de vulnerabilidades do tipo *buffer overflow*.

Os resultados obtidos são válidos para os experimentos realizados com o programa da Figura 5.2, executado em um ambiente específico composto por um PC Pentium 4, 2 GHz de *clock* e 256 Mbytes de memória RAM, sistema operacional Linux Red Hat 9.0 e utilizando o compilador gcc versão 3.2.2 para geração do código executável. Como já mencionado, a quantidade de memória alocada para as variáveis locais no *stack frame* da função depende do compilador sendo utilizado, da arquitetura da máquina e do sistema operacional, portanto não é possível garantir a generalização dos resultados, mas os dados obtidos indicam a validação da técnica.

6.3 Trabalhos Futuros

Destacam-se algumas atividades que podem ser realizadas como continuidade deste trabalho:

1. o estudo de caso realizado no 2º experimento, que teve como objetivo verificar a eficácia da técnica, não conseguiu indicar que o operador MDMA é eficaz na detecção da presença de *buffer overflow*. Isto pelo fato de não se estar utilizando um programa apropriado para aplicação deste operador.
   Novos experimentos devem ser realizados, aplicando a técnica através da SEVMUT a vários programas diferentes que alocam a memória dinamicamente, objetivando mostrar ao testador a eficácia da técnica em relação ao operador MDMA na detecção da presença de *buffer overflow*, desta forma ampliando a abrangência dos resultados obtidos nessa dissertação;

2. o conjunto de operadores de mutação estudado neste trabalho foi definido somente para detectar a presença de vulnerabilidade do tipo *buffer overflow*. Novas vulnerabilidades candidatas à aplicação da técnica devem ser descobertas, definindo novos operadores de mutação que as detectam, desta forma ampliando o conjunto de operadores de mutação;

3. gerar uma segunda versão da Ferramenta SEVMUT para aplicar os operadores de mutação MSMA e MDMA a programas C orientados a objetos. As mudanças ocorreriam nos módulos *Analex* e *Germut*, que fazem a análise léxica e sintática do programa, gerando os mutantes. Os outros módulos praticamente não sofreriam modificações;

4. evoluir a Ferramenta SEVMUT. Várias mudanças podem ser feitas na SEVMUT para melhorar o uso desta ferramenta:
   - a interação da SEVMUT com o testador ocorre através de menu. Esta interface poderia ser melhorada através do uso de ícones, de barras de ferramentas ou através de uma outra interface,
   - um banco de dados poderia ser adicionado a SEVMUT para armazenar as informações necessárias na condução das funções executadas pela ferramenta. Com isto, eliminar-se-ia a tarefa do usuário em criar diretórios para a SEVMUT armazenar informações, otimizando o uso da ferramenta,
   - implementação de sessões de teste. Com as sessões de teste, o teste do programa deve ocorrer através de etapas. Para isto é necessário que os estados intermediários da aplicação do teste sejam armazenados e depois recuperados, permitindo ao usuário iniciar o teste do programa, encerrar a atividade de teste e posteriormente retomá-la do ponto em que foi deixada, sem necessidade de reiniciá-la. Sessões de teste são usadas na Proteum [Del93], Proteum/IM [Del97] e na POKE-TOOL [Cha91].

Espera-se que com a realização das atividades propostas acima, a contribuição que este trabalho traz à área de Engenharia de Software, especificamente à área de teste de software, possa ser ampliada.

Referências Bibliográficas

[Alb02] Albuquerque, R. and Ribeiro, B, *Segurança no Desenvolvimento de Software*, Editora Campus, 1ª. Edição, 2002.

[Bar99] Baratloo, A., Singh, N., Tsai, T., *Libsafe: Protecting Critical Elements of Stacks,* http://www.research.avayalabs.com/project/libsafe, December 1999 – visitado em 05/01/2005.

[Bis99] Bishop, M., *How Attackers Break Programs and How to Write Programs Securely*, Tutorial presented in 8th USENIX Symposium Security, Washington - DC, August 1999.

[Bue99] Bueno, P.M.S., *Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software*, Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas – SP, 1999.

[Cow98] Cowan, C., and et al. *StackGuard:Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks*, 7th USENIX Security Symposium, San Antonio-Texas, January 1998.

[Cow99] Cowan, C., Wagle, P., Calton, P., Beattie, S., and Walpole, J., *Buffer Overflow: Attacks and Defenses for the Vulnerability of the Decade,* Proceedings of the DARPA Information Survivability Conference and Expo, 1999.

[Col00] Collyer, F., *Buffer Overflow : Entendimento e Prevenção,* Developer's Magazine - Brasil, Maio 2000.

[Cha91] Chaim, M., *Poke-tool – Uma ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados*. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas – SP, 1991.

[CSD04] CSD. *Computer Science Department. Boston University*, http://cs-www.bu.edu/help/unix/segmentation_fault.html - visitado em 11/11/2004.

[Deb99] Debar, H., Dacier, M., and Wespi, A., *Towards a Taxonomy of Intrusion Detection Systems.* Computer Networks 31, 8, 805-822. Special issue on Computer Network Security, 1999.

[Del93] Delamaro, Márcio E., *Proteum – Um Ambiente de Teste Baseado na Análise de Mutantes*. Dissertação de Mestrado, ICMSC – USP, São Carlos – SP, 1993.

[Del97] Delamaro, Márcio E., *Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração*. Tese de Doutorado, IFSC – USP, São Carlos – SP, 1997.

[DeM78] DeMillo, R.A., Lipton, R., Sayward, F., *Hints on Test Data Selection: Help for the Practicing Programmer*. Computer, (11)4, April 1978.

[DeM91] DeMillo, R.A., Offut, A.J., *Constraint-Based Automatic Test Data Generation*, IEEE Transaction on Software Engineering, vol. 17, No.9, 1991.

[Dur84] Duran, J.W., *An Evaluation of Random Testing*, IEEE Transaction on Software Engineering, vol.SE-10, No.4, 1984.

[Eto00] Etoh, H., *GCC Extension for Protecting Applications from Stack-Smashing Attacks,* http://www.trl.ibm.com/projects/security/ssp, June 2000 – visitado em 05/01/2005.

[Fau03] Faulhaber, H., *A inovação vem da Ásia*, Softex 2000, publicada em 16/10/2003 em http://www.softex.br/cgi/cgilua.exe/sys/start.htm?infoid=2948&sid=178 – visitado em 17/01/2005.

[Fer04] Ferreira, A. B. H., *Novo Dicionário Aurélio da Língua Portuguesa*, 3ª. Edição, Editora Positivo, 2004.

[For01] Forristal, J., *Hack Proofing Your Web Applications*, Syngress Publishing, 1a. Edition, 2001.

[Gia01] Giasson, F., *Memory Layout in Program Execution*, http://www.decatomb.com/articles/memorylayout.txt, 2001 – visitado em 11/11/2004.

[Gau00] Gaur, N., *Assessing the Security of your Web Applications*, Linux Journal, vol.20, Issue 72es, April 2000.

[Hae88] Haertel, M., sort.c, disponível em http://www.mit.edu/afs/sipb/project/darwin/src/modules/text_cmds/sort/sort.c, 1988 - visitado em 29/10/2004.

[Har99] Harari, E., *A Look at the Buffer Overflow Hack*, Linux Journal, vol.1999, Issue 61es, May 1999.

[Har00] Harman, M., Hierons, R. and Danicic, S., *The Relationship Between Program Dependence and Mutation Analysis,* Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose-CA, October 2000.

[Hau03] Haugh, E. and Bishop, M., *Testing C Programs for Buffer Overflow Vulnerabilities*, Network and Distributed System Security Symposium, San Diego-CA, 2003.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New York, 1990.

[Lap00] Lapponi, J.C., *Estatística usando Excel*, Lapponi Treinamento e Editora, 2000.

[Ker81] Kernighan, B.W., Plauger, P.J., *Software Tools in Pascal*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[Mal91] Maldonado, J.C., *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Tese de Doutorado, DCA/FEEC/UNICAMP, Campinas – SP, 1991.

[Mas90] Mason, T., Brown, D., *Lex & Yacc*, O`Reilly &Associates Inc., U.S.A., 1990.

[Mye79] Myers, G.J., *The Art of Software Testing*, J.Wiley, 1979.

[Off97] Offutt, A.J., Pan J., *Detecting Equivalent Mutants and the Feasible Path Problem,* The Journal of Software Testing, Verification, and Reliability, Vol.7, No.3, 1997.

[Pre04] Pressman, R.S., *Software Engineering: A Practitioner's Approach,* Sixth Edition, McGraw-Hill, 2004.

[Rap85] Rapps, S., Weyuker, E.J., *Selecting Software Test Data using Data Flow Information,* IEEE Transaction on Software Engineering, vol.SE-11, No.4, 1985.

[Rit73] Ritchie, D., Thompson, K., and Richards, M., *C Grammar*, http://www.devincook.com/goldparser/grammars/index.htm, 1973 – visitado em 11/11/2004.

[Sna97] Snarskii, A., *Stack Integrity Patch*, ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997 – visitado em 05/01/2005.

[Sta99] Stallings, W., *Cryptography and Network Security: Principles and Practice,* Prentice-Hall, 2a. Edition, 1999.

[Vil02] Vilela, P., Machado, M., and Wong, E., *Testing for Security Vulnerabilities in Software,* In IASTED International Conference on Software Engineering and Applications, Cambridge, USA, November 2002.

[Wil03] Wilander, J., and Kamkar, M., *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow,* Proceedings of the 10th Network on Distributed System Security Symposium, 2003.

[Won98] Wong, E., Horgan, J.R., London, S., and Mathur, A.P., *Effect of Test Set Minimization on Fault Detection Effectiveness,* Software-Practice and Experience, 28(4):347-369, April 1998.

Apêndice A

Gramática da Linguagem C

Neste apêndice são apresentadas as especificações léxica e sintática da linguagem C, introduzidas na Seção 4.3 do Capítulo 4, no formato reconhecido pelas Ferramentas Lex e Yacc [Mas90].

A.1. Especificação Léxica da Linguagem C

```
D               [0-9]
DL              [1-9]
L               [a-zA-Z_]
OCT_DIG         [0-7]
HEX_DIG         [0-9a-fA-F]

%%
"auto"          { return AUTO; }
"break"         { return BREAK; }
"calloc"        { return ALLOC; }
"case"          { return CASE; }
"char"          { return CHAR; }
"const"         { return CONST; }
"continue"      { return CONTINUE; }
"default"       { return DEFAULT; }
"do"            { return DO; }
"double"        { return DOUBLE; }
"else"          { return ELSE; }
"enum"          { return ENUM; }
"extern"        { return EXTERN; }
"float"         { return FLOAT; }
"for"           { return FOR; }
"goto"          { return GOTO; }
"if"            { return IF; }
"int"           { return INT; }
"long"          { return LONG; }
"malloc"        { return ALLOC; }
"realloc"       { return ALLOC; }
"register"      { return REGISTER; }
"return"        { return RETURN; }
"short"         { return SHORT; }
"sizeof"        { return SIZEOF; }
"signed"        { return SIGNED; }
"static"        { return STATIC; }
"struct"        { return STRUCT; }
"switch"        { return SWITCH; }
"typedef"       { return TYPEDEF; }
"union"         { return UNION; }
"unsigned"      { return UNSIGNED; }
"void"          { return VOID; }
"volatile"      { return VOLATILE; }
"while"         { return WHILE; }
"..."           { return ELIPSE; }
"+="            { return ADD_ASSIGN; }
"-="            { return SUB_ASSIGN; }
"*="            { return MUL_ASSIGN; }
"/="            { return DIV_ASSIGN; }
```

```
"&="            { return AND_ASSIGN; }
"^="            { return XOR_ASSIGN; }
"|="            { return OR_ASSIGN; }
"%="            { return MOD_ASSIGN; }
">>"            { return RIGHT_OP; }
"<<"            { return LEFT_OP; }
"++"            { return INC_OP; }
"--"            { return DEC_OP; }
"->"            { return PTR_OP; }
"&&"            { return AND_OP; }
"||"            { return OR_OP; }
"<="            { return LE_OP; }
">="            { return GE_OP; }
"=="            { return EQ_OP; }
"!="            { return NE_OP; }
";"             { return ';'; }
"{"             { return '{'; }
"}"             { return '}'; }
","             { return ','; }
":"             { return ':'; }
"="             { return '='; }
"("             { return '('; }
")"             { return ')'; }
"["             { return '['; }
"]"             { return ']'; }
"."             { return '.'; }
"&"             { return '&'; }
"!"             { return '!'; }
"~"             { return '~'; }
"-"             { return '-'; }
"+"             { return '+'; }
"*"             { return '*'; }
"/"             { return '/'; }
"%"             { return '%'; }
"<"             { return '<'; }
">"             { return '>'; }
"^"             { return '^'; }
"|"             { return '|'; }
"?"             { return '?'; }
{L}({L}|{D})*   { return ID; }
{DL}({D})*      { return DEC_LIT; }
0({OCT_DIG})*   { return OCT_LIT; }
0[xX]({HEX_DIG})+    { return HEX_LIT; }
{D}*\.{D}+           { return FLOAT_LIT; }
\"([^"]|\\[^"])*\"   { return STRING_LIT; }
\'([^']|\\[^'])\'    { return CHAR_LIT; }
[ \t\n]
.

%%
```

A.2. Especificação Sintática da Linguagem C

```
%token   ALLOC AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT
%token   DO DOUBLE ELSE ENUM EXTERN FLOAT FOR GOTO IF
%token   INT LONG REGISTER RETURN SHORT SIZEOF SIGNED
%token   STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED
%token   VOID VOLATILE WHILE ID ELIPSE ADD_ASSIGN SUB_ASSIGN
%token   MUL_ASSIGN DIV_ASSIGN AND_ASSIGN XOR_ASSIGN
%token   OR_ASSIGN MOD_ASSIGN RIGHT_OP LEFT_OP INC_OP DEC_OP
%token   PTR_OP AND_OP OR_OP LE_OP GE_OP EQ_OP NE_OP OCT_LIT
%token   HEX_LIT DEC_LIT STRING_LIT CHAR_LIT FLOAT_LIT


%start declarations

%%
declarations
      : declaration declarations
      | /* empty */
      ;

declaration
      : function_declare
      | function_prototype
      | struct_declare
      | union_declare
      | enum_declare
      | var_declare
      | TYPEDEF declaration
      ;

function_prototype
      : type var '(' param_list ')' ';'
      | type var '(' ')' ';'
      | type var '(' type ')' ';'
      ;

function_declare
      : type body_function_declare
      | type_other body_function_declare
      | type type_other body_function_declare
      | body_function_declare
      ;

body_function_declare
      : var '(' param_list ')' block
      | var '(' id_list ')' struct_def block
      | var '(' id_list ')' block
      | var '(' ')' block
      | var '(' type ')' block
      ;

param_list
      : type body_param_list
      | type_other body_param_list
      | ELIPSE
      ;

body_param_list
      : var ',' param_list
      | var
      ;

id_list
      : var ',' id_list
      | var
      ;
```

```
struct_declare
      : STRUCT ID '{' struct_def '}' ';'
      | type STRUCT ID '{' struct_def '}' id_list ';'
      ;

union_declare
      : UNION ID '{' struct_def '}'
      ;

struct_def
      : var_declare struct_def
      | var_declare
      ;

enum_declare
      : ENUM ID '{' enum_def '}' ';'
      | ENUM '{' enum_def '}' ID ';'
      ;

enum_def
      : enum_val ',' enum_def
      | enum_val
      ;

enum_val
      : ID
      | ID '=' OCT_LIT
      | ID '=' HEX_LIT
      | ID '=' DEC_LIT
      ;

var_declare
      : type body_var_declare
      | type_other body_var_declare
      ;


body_var_declare
      : var_list ';'
      | var_list '=' expression ';'
      ;

type
      : storage_spec
      | storage_spec type_quant
      | storage_spec type_mod
      | storage_spec type_base
      | storage_spec type_quant type_mod
      | storage_spec type_quant type_mod type_base
      | storage_spec type_mod type_base
      | type_quant
      | type_quant type_mod
      | type_quant type_mod type_base
      | type_mod
      | type_mod type_base
      | type_base
      | STRUCT ID
      | STRUCT ID '{' struct_def '}'
      | type STRUCT ID
      | type STRUCT ID'{' struct_def '}' type
      | STRUCT '{' struct_def '}'
      | UNION ID
      | UNION '{' struct_def '}'
      | ENUM ID
      ;

storage_spec
      : EXTERN
      | STATIC
```

```
      | REGISTER
      | AUTO
      ;

type_quant
      : VOLATILE
      | CONST
      ;

type_mod
      : SIGNED
      | UNSIGNED
      | LONG
      | SHORT
      | SIGNED SHORT
      | UNSIGNED SHORT
      | SIGNED LONG
      | UNSIGNED LONG
      ;

type_base
      : CHAR
      | INT
      | FLOAT
      | DOUBLE
      | VOID
      ;

type_other
      : ID
      ;

var
      : '*' var
      | ID
      | ID '[' body_var ']'
      | ID '[' body_var ']' '[' body_var ']'
      ;


body_var
      : /* empty */
      | DEC_LIT body_var
      | ID body_var
      | op body_var
      ;


op
      : '-'
      | '+'
      | '*'
      | '/'
      ;

var_list
      : var ',' var_list
      | var
      ;

statement
      : var_declare
      | label
      | IF '(' expression ')' statement
      | IF '(' expression ')' then_stm ELSE statement
      | WHILE '(' expression ')' statement
      | WHILE '(' expression ')' ';' statement
      | FOR '(' for_arg ';' for_arg ';' for_arg ')' ';' statement
      | FOR '(' for_arg ';' for_arg ';' for_arg ')' statement
      | normal_stm
```

```
      | normal_stm ';'
      ;

then_stm
      : IF '(' expression ')' then_stm ELSE then_stm
      | WHILE '(' expression ')' then_stm
      | WHILE '(' expression ')' ';' then_stm
      | FOR '(' for_arg ';' for_arg ';' for_arg ')' then_stm
      | FOR '(' for_arg ';' for_arg ';' for_arg ')' ';'
      | normal_stm
      ;

normal_stm
      : DO statement WHILE '(' expression ')' ';'
      | SWITCH '(' expression ')' '{' case_labels '}'
      | block
      | expression ';'
      | GOTO ID ';'
      | BREAK ';'
      | CONTINUE ';'
      | RETURN expression ';'
      | RETURN ';'
      ;

label
      : ID ':'
      | ID ':' ';'
      ;

for_arg
      : expression
      | /* empty */
      ;

case_labels
      : CASE r_value ':' ';'
      | CASE r_value ':' stm_list case_labels
      | DEFAULT ':' stm_list case_labels
      | /* empty */
      ;

block
      : '{' stm_list '}'
      ;

stm_list
      : statement stm_list
      | /* empty */
      ;

expression
      : expression ',' op_assign
      | expression ','
      | op_assign
      ;

op_assign
      : op_other '=' op_assign
      | op_other ADD_ASSIGN op_assign
      | op_other SUB_ASSIGN op_assign
      | op_other MUL_ASSIGN op_assign
      | op_other DIV_ASSIGN op_assign
      | op_other XOR_ASSIGN op_assign
      | op_other AND_ASSIGN op_assign
      | op_other OR_ASSIGN op_assign
      | op_other MOD_ASSIGN op_assign
      | op_if
      ;

op_if
```

```
      : op_or '?' op_if ':' op_if
      | op_or
      ;

op_or
      : op_or OR_OP op_and
      | op_and
      ;

op_and
      : op_and AND_OP op_binor
      | op_binor
      ;

op_binor
      : op_binor '|' op_binxor
      | op_binxor
      ;

op_binxor
      : op_binxor '^' op_binand
      | op_binand
      ;

op_binand
      : op_binand '&' op_equate
      | op_equate
      ;

op_equate
      : op_equate EQ_OP op_compare
      | op_equate NE_OP op_compare
      | op_compare
      ;

op_compare
      : op_compare '<' op_shift
      | op_compare '>' op_shift
      | op_compare LE_OP op_shift
      | op_compare GE_OP op_shift
      | op_shift
      ;

op_shift
      : op_shift LEFT_OP op_add
      | op_shift RIGHT_OP op_add
      | op_add
      ;

op_add
      : op_add '+' op_mult
      | op_add '-' op_mult
      | op_mult
      ;

op_mult
      : op_mult '*' op_other
      | op_mult '/' op_other
      | op_mult '%' op_other
      | op_other
      ;

op_other
      : '!' op_other
      | '~' op_other
      | INC_OP l_value
      | l_value INC_OP
      | DEC_OP l_value
      | l_value DEC_OP
      | '-' op_other
```

```
      | '+' op_other
      | '(' type ')'
      | '(' type ')' r_value
      | '(' type '*' ')'
      | '(' type '*' ')' r_value
      | '(' type '*' '*' ')' r_value
      | '(' ID '*' ')' r_value
      | '(' ID '*' expression ')'
      | '*' op_other
      | '&' op_other
      | SIZEOF op_other
      | r_value
      ;

r_value
      :'(' expression ')'
      |'{' expression '}'
      |'(' expression ')' l_value
      |'(' expression ')' '.' l_value
      | ALLOC '(' expression ')'
      | ID '(' expression ')'
      | ID '(' ')'
      | OCT_LIT
      | HEX_LIT
      | DEC_LIT
      | STRING_LIT
      | CHAR_LIT
      | FLOAT_LIT
      | l_value
      ;

l_value
      : ID
      | ID '.' l_value
      | ID '[' ']'
      | ID '[' expression ']'
      | ID '[' expression ']' '['expression ']'
      | ID '[' expression ']' '['expression ']' '['expression ']'
      | ID '[' expression ']' '.' l_value
      | PTR_OP l_value
      | ID PTR_OP l_value
      ;

%%
```

Apêndice B

Telas da SEVMUT

Neste apêndice apresentam-se algumas das principais telas da Ferramenta SEVMUT utilizadas na interface com o usuário e descritas detalhadamente no Capítulo 4.

A Figura B.1 mostra a tela que permite ao testador obter, visualizar e imprimir dados.

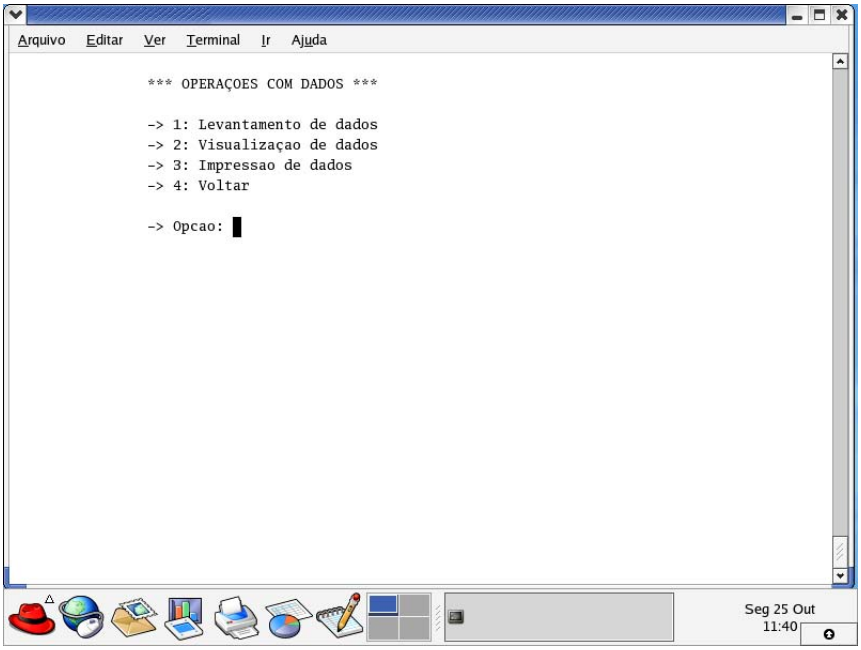

Figura B.1 – Tela de Operações com Dados

A Figura B.2 ilustra a tela da Ferramenta SEVMUT com as opções fornecidas ao testador para levantamento de dados. Por meio desta tela o testador consegue obter os mutantes eliminados por um dado conjunto de dados de teste e obter quais os dados de teste eliminam um determinado mutante.

A Figura B.3 ilustra a tela que possibilita ao testador visualizar alguns dados. Com esta interface o testador consegue visualizar os estados dos mutantes, visualizar os mutantes mortos dado um conjunto de dados de teste, visualizar os dados de teste que eliminam um determinado mutante e visualizar o programa mutante-fonte gerado.

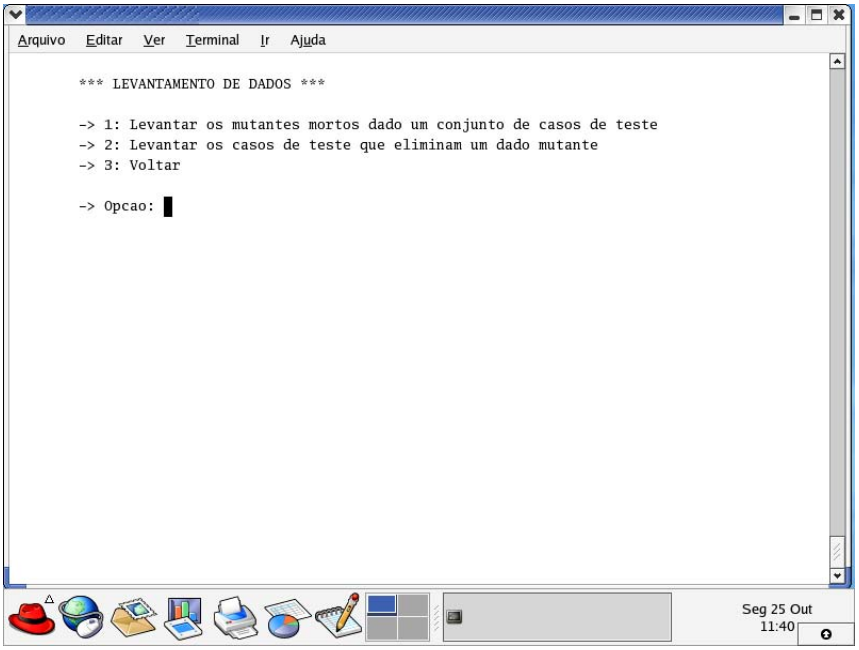

Figura B.2 – Tela de Levantamento de Dados

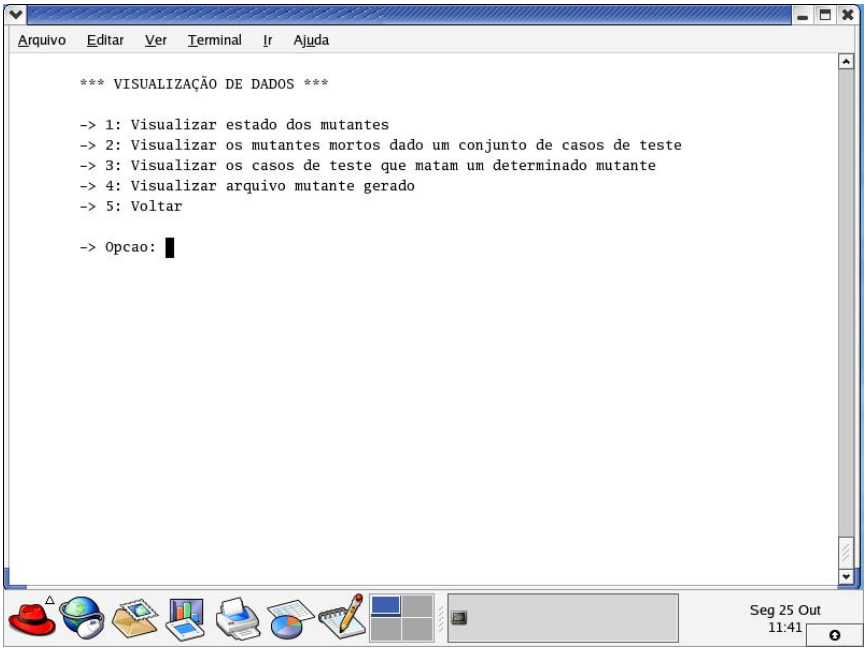

Figura B.3 – Tela de Visualização de Dados

A Figura B.4 mostra a tela de impressão de dados que permite ao usuário imprimir o programa mutante-fonte, a tabela de estado dos mutantes e os relatórios gerados com o uso da opção de levantamento de dados.

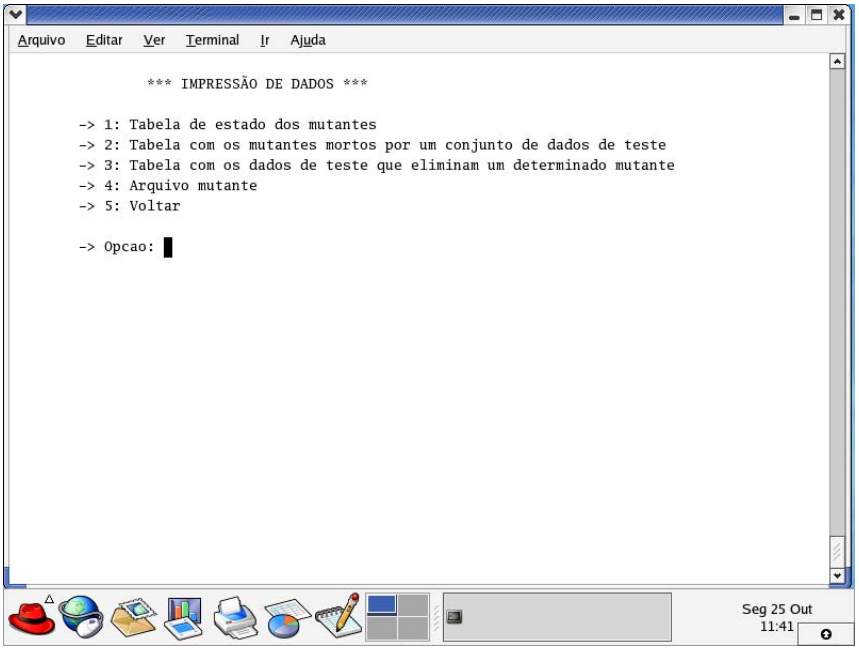

Figura B.4 – Tela de Impressão de Dados

A Figura B.5 ilustra a tela de alteração de estado do mutante. Com esta interface o usuário consegue alterar o estado do mutante de *vivo* para *equivalente* e vice-versa. A Figura B.6 mostra o resultado da aplicação dos operadores MSMA e MDMA em todos os tipos de alocações de memória de um dado programa resultando na geração de 92 mutantes.

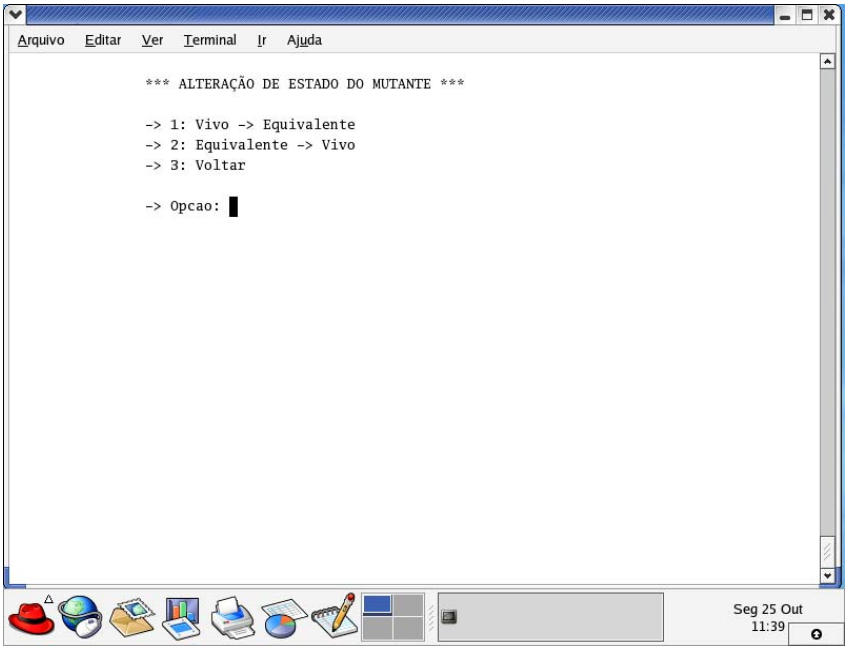

Figura B.5 – Tela de Alteração de Estado do Mutante

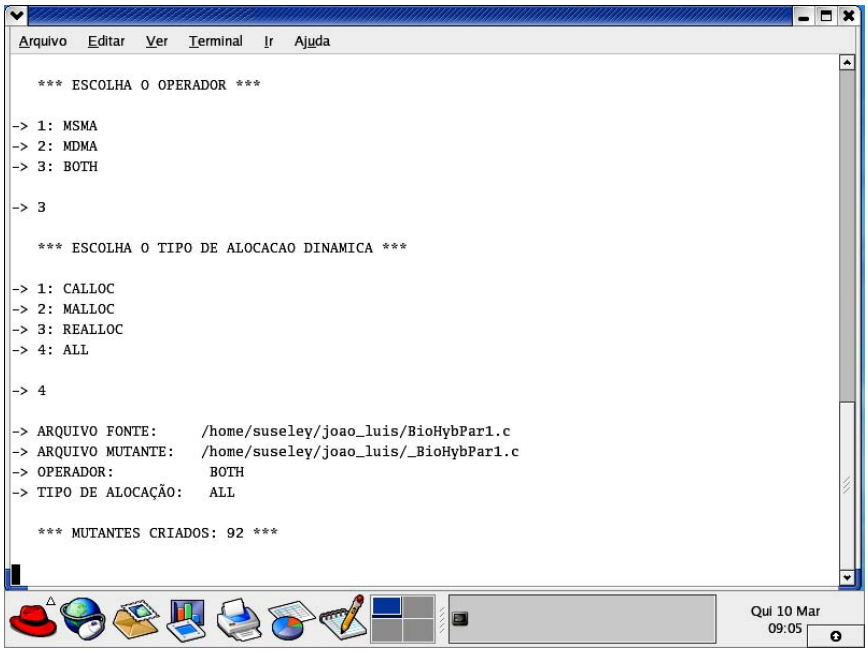

Figura B.6 – Tela de Resultado da Geração dos Mutantes

Apêndice C

Tabela de Dados

Neste apêndice são apresentados os dados utilizados no experimento descrito no Capítulo 4, que tem como objetivo avaliar a performance da Ferramenta SEVMUT utilizando para isto o tempo de geração dos mutantes.

A Tabela C.1 mostra o significado de cada coluna da Tabela C.2 que ilustra os dados coletados para a realização dessa investigação empírica.

Coluna	Significado
Unidade	As unidades utilizadas no experimento.
Sistema	O sistema ao qual pertencem as unidades.
LoC	O total de linhas de código de cada programa.
Linhas de Programa	O total de linhas de cada programa incluindo as linhas de comentário e as linhas em branco.
Alocações Estáticas	O total de alocações estáticas de cada unidade.
Alocações Dinâmicas	O total de alocações dinâmicas de cada unidade.
Total de Alocações	O total de alocações estáticas e dinâmicas.
Mutantes Gerados	A quantidade de mutantes gerados para cada unidade.
Tempo de Geração	O tempo de geração dos mutantes em milissegundos.

Tabela C.1 – Tabela de Significado da Tabela de Dados

Nro	Unidade	Sistema	LoC	Linhas Programa	Alocação Estática	Alocação Dinâmica	Total de Alocações	Mutantes Gerados	Tempo Geração (ms)
1	Amatch.c	*Benchmark* [Ker81]	159	277	3	0	3	12	4,660
2	Archive.c	*Benchmark* [Ker81]	92	159	3	0	3	12	2,770
3	Change.c	*Benchmark* [Ker81]	444	611	6	0	6	24	10,880
4	Ckglob.c	*Benchmark* [Ker81]	446	619	6	0	6	24	11,130
5	Compare.c	*Benchmark* [Ker81]	139	193	6	0	6	24	4,150
6	Edit.c	*Benchmark* [Ker81]	1204	1633	17	0	17	68	29,340
7	Entab.c	*Benchmark* [Ker81]	58	81	1	0	1	4	1,480
8	Getdef.c	*Benchmark* [Ker81]	115	133	3	0	3	12	2,790
9	Getfn.c	*Benchmark* [Ker81]	66	82	3	0	3	12	2,190
10	Getfns.c	*Benchmark* [Ker81]	106	212	3	0	3	12	2,540
11	Getlist.c	*Benchmark* [Ker81]	495	694	4	0	4	16	11,490
12	Getnum.c	*Benchmark* [Ker81]	418	593	4	0	4	16	9,910
13	Getone.c	*Benchmark* [Ker81]	456	639	4	0	4	16	10,650
14	Gtext.c	*Benchmark* [Ker81]	61	78	5	0	5	20	2,310
15	Makepat.c	*Benchmark* [Ker81]	177	231	3	0	3	12	5,460
16	Omatch.c	*Benchmark* [Ker81]	100	138	2	0	2	8	2,670
17	Optpat.c	*Benchmark* [Ker81]	179	244	1	0	1	4	5,160
18	Rquick.c	*Benchmark* [Ker81]	98	118	4	0	4	16	2,270
19	Spread.c	*Benchmark* [Ker81]	63	79	1	0	1	4	1,660
20	Subst.c	*Benchmark* [Ker81]	316	487	6	0	6	24	7,540
21	Translit.c	*Benchmark* [Ker81]	187	317	4	0	4	16	5,460
22	Unrotate.c	*Benchmark* [Ker81]	65	89	2	0	2	8	2,120
23	Adddef.c	SPACE	104	207	0	4	4	16	3,160
24	Answer.c	SPACE	9	20	1	0	1	4	0,840

Nro	Unidade	Sistema	LoC	Linhas Programa	Alocação Estática	Alocação Dinâmica	Total de Alocações	Mutantes Gerados	Tempo Geração (ms)
25	Blockdef.c	SPACE	76	154	0	2	2	8	2,480
26	Copia.c	SPACE	1117	1174	5	0	5	20	23,120
27	Elemdef.c	SPACE	36	91	1 0 1	1 1 0	2 1 1	8 4 4	1,950 1,790 1,660
28	Fixselem.c	SPACE	64	112	0	1	1	4	2,420
29	Getint.c	SPACE	55	95	1	0	1	4	1,630
30	Getkwd.c	SPACE	38	73	1	0	1	4	1,180
31	Getreal.c	SPACE	56	95	1	0	1	4	1,690
32	Grampexc.c	SPACE	72	146	0	2	2	8	2,480
33	Griddef1.c	SPACE	87	174	0	1	1	4	2,970
34	Groupdef.c	SPACE	87	175	1 0 1	1 1 0	2 1 1	8 4 4	3,230 2.750 2,690
35	Grphaexc.c	SPACE	148	254	0	4	4	16	4,730
36	Grwrite.c	SPACE	70	108	1	0	1	4	2,370
37	Hexdef.c	SPACE	44	104	0	1	1	4	1,630
38	Mksblock.c	SPACE	75	134	0	1	1	4	2,480
39	Mkshex.c	SPACE	70	131	2 2 0	1 0 1	3 2 1	12 8 4	2,620 2,170 2,400
40	Mksnodec	SPACE	55	99	0	1	1	4	1,760
41	Nodedef.c	SPACE	66	133	0	1	1	4	2,100
42	Oracolo2.c	SPACE	545	617	1	0	1	4	11,500
43	Polydef.c	SPACE	60	130	0	1	1	4	1,920
44	Portspec.c	SPACE	54	131	0	1	1	4	2,090
45	Prepro.c	SPACE	544	617	1	0	1	4	11,440

Nro	Unidade	Sistema	LoC	Linhas Programa	Alocação Estática	Alocação Dinâmica	Total de Alocações	Mutantes Gerados	Tempo Geração (ms)
46	Readfil3.c	SPACE	72	120	1 1 0	3 0 3	4 1 3	16 4 12	2,750 2,460 2,710
47	Remdef.c	SPACE	104	217	0	4	4	16	3,600
48	Risposta.c	SPACE	9	21	1	0	1	4	0,900
49	Sgramp2n.c	SPACE	129	216	2 2 0	1 0 1	3 2 1	12 8 4	3,840 3,280 3,710
50	Sgrampun.c	SPACE	40	98	0	1	1	4	1,900
51	Sgrpha2n.c	SPACE	119	197	2	0	2	8	3,420
52	Cal.c	-	148	202	1	0	1	4	3,240
53	BioHybPar1.c	-	2153	2980	23	0	23	92	53,320
54	Concat.c	-	73	90	0	1	1	4	2,270
55	Sort.c	-	1523	1873	12 12 0	2 0 2	14 12 2	56 48 8	31,410 31,310 30,950

Tabela C.2 – Tabela de Dados Coletados. Para o significado das colunas veja a Tabela C.1.

Apêndice D

Conjunto de Dados de Teste

Neste apêndice são apresentados os dados de teste de entrada, utilizados no experimento descrito no Capítulo 5, que tem como objetivo revelar a eficácia da técnica descrita no Capítulo 3, na sua habilidade de detectar a presença de vulnerabilidade do tipo *buffer overflow*.

D.1. Conjunto de Dados de Teste – DTPASS0

D

t

@

rRI

r@R

INv

n_WWJ

8M!Qr

U^-]y

nxI}C

D.2. Conjunto de Dados de Teste – DTPASS1

k^

$I

c2)J

{r"<

PbG;"

x_+oL

D.3. Conjunto de Dados de Teste – DTPASS2

Z8`YP.

SBt[]A

G:XfJj}Wy6

lKFmh"jK0{

:4(hKxVuQCGEOnS

KV@z.V_H{h)#hkZ

D.4. Conjunto de Dados de Teste – DTPASS3

y*u\-j+~A7+wevL%*uUv

XG%dV3A0kDVXyAaRe++Y

FEmMQ+6:)/mx2#BAds(Z^jp+L~-:*=

jWMoKN@9)Vrs,?13EAo2prQ1DCvfE$

TU1RgKyFwUxqSW#Q3L%?x%w'?;v11eN~_A=)dJw_

I1TcYh>bB"l<'e04Ok09nzO:]"GMfG"q7JY=9lUq

D.5. Conjunto de Dados de Teste – DTPASS4

kMq>^S<RT%pq!z,

B21ZBW,S@Ih_e#+2u,Cpqj,%3NG]sq

a*)=8g-.ZgV/!J]NS;>36D)J0j2{p58?4).`<&s[

B~z@pU\R[zDA=gK00p(Qi&bhm{L&>M%3v,nvHQ,^AifOj0WC4f|`:TL\X>?T

D.6. Conjunto de Dados de Teste – DTPASS5

dr"$mC?V4#|dg)MfN5tA`@i79Kfu<R

'=o(dG0@.)4?HswJ.T^HJ:)x@tbn{'nV&jep)y},

zJa$Cy\!`F'w7U6t#mmG5%lx/Ncjd4~-!?br=qjWUvD"6hQR1m

$RI<"LOKFX<TFm'\?)/L+~jD4@IO^Hz@x4'P'8qXdIZwO(O[15m't."3o|!H

Apêndice E

Experimentos

Neste apêndice descrevem-se os pontos importantes da condução dos experimentos realizados com o objetivo de verificar a eficácia da técnica descrita no Capítulo 3, em cada um dos grupos da família de experimentos ilustrada na Figura 5.1 do Capítulo 5. Os resultados obtidos em cada classe são descritos no Capítulo 5.

As Tabelas E.1, E.2, E.3, E.4, E.5 e E.6 mostram as propriedades de cada um dos conjuntos de dados de teste de entrada descritos no Apêndice D, criados com o objetivo de realizar esta investigação empírica. A primeira coluna indica a posição do dado de teste dentro do conjunto, a segunda coluna mostra o tamanho em caracteres de cada senha e a terceira coluna, a classe de dados de teste dentro do conjunto a qual o dado pertence.

O conjunto de dados de teste dtpass0 é constituído de dez dados de teste de entrada que constam do Apêndice D.1. A Tabela E.1 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º, 2º e 3º	1	DT1
4º, 5º e 6º	3	DT2
7º, 8º, 9º e 10º	5	DT3

Tabela E.1 – Propriedades do Conjunto de Dados de Teste dtpass0

O conjunto de dados de teste dtpass1 é constituído de seis dados de teste de entrada que constam do Apêndice D.2. A Tabela E.2 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º e 2º	2	DT1
3º e 4º	4	DT2
5º e 6º	5	DT3

Tabela E.2 – Propriedades do Conjunto de Dados de Teste dtpass1

O conjunto de dados de teste dtpass2 é constituído de seis dados de teste de entrada que constam do Apêndice D.3. A Tabela E.3 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º e 2º	6	DT1
3º e 4º	10	DT2
5º e 6º	15	DT3

Tabela E.3 – Propriedades do Conjunto de Dados de Teste dtpass2

O conjunto de dados de teste dtpass3 é constituído de seis dados de teste de entrada que constam do Apêndice D.4. A Tabela E.4 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º e 2º	20	DT1
3º e 4º	30	DT2
5º e 6º	40	DT3

Tabela E.4 – Propriedades do Conjunto de Dados de Teste dtpass3

O conjunto de dados de teste dtpass4 é constituído de quatro dados de teste de entrada que constam do Apêndice D.5. A Tabela E.5 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º	15	DT1
2º	30	DT2
3º	40	DT3
4º	60	DT4

Tabela E.5 – Propriedades do Conjunto de Dados de Teste dtpass4

O conjunto de dados de teste dtpass5 é constituído de quatro dados de teste de entrada que constam do Apêndice D.6. A Tabela E.6 ilustra as propriedades deste conjunto.

Dados de Teste (senhas)	Caracteres	Denominação
1º	30	DT1
2º	40	DT2
3º	50	DT3
4º	60	DT4

Tabela E.6 – Propriedades do Conjunto de Dados de Teste dtpass5

Os resultados obtidos durante a condução do experimento são apresentados através de legendas, cujos significados são descritos na Tabela E.7.

LEGENDA	SIGNIFICADO
PF – Programa Falha	Programa original resulta em falha de segmentação.
MF – Mutante Falha	Mutante resulta em falha de segmentação
RPI – Resultado *Password* Incorreto	Programa original ou mutante executou resultando em "Password Incorreto"
RPC – Resultado *Password* Correto	Programa original ou mutante executou resultando em "Password Correto"

Tabela E.7 – Tabela de Legendas

A seguir são descritos os experimentos conduzidos em cada grupo da família de experimentos separadamente.

E.1 Grupo Estático

O objetivo é conduzir o experimento aplicando o operador MSMA ao programa da Figura 5.2 do Capítulo 5, analisando os resultados das execuções dos mutantes.

Dividiu-se o grupo estático em duas categorias: local e global. A categoria local foi dividida em três classes de experimentos: variáveis locais sem inversão, inversão das variáveis e alteração de tamanho. A categoria global foi dividida em duas classes de experimentos: variáveis globais e alteração de tamanho.

Segue-se a descrição de cada uma das classes de experimentos, dentro das categorias local e global do grupo estático da família de experimentos.

E.1.1 Categoria Local

E.1.1.1 Variáveis Locais sem Inversão

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA e suas variações ocorrem numa variável declarada localmente à função.

A aplicação do operador MSMA ao programa da Figura 5.2 do Capítulo 5 resulta na geração de quatro mutantes com a quantidade de memória alocada para a variável Buffer de acordo com a Tabela E.8.

Mutante	Característica	Quantidade alocada de bytes para a variável **Buffer**
1	Quantidade alocada -1	5
2	Quantidade alocada / 2	3
3	Quantidade alocada * 2	12
4	Quantidade alocada = 1	1

Tabela E.8 – Mutantes Gerados

A Figura E.1 ilustra o código do programa mutante-fonte resultante. As linhas em negrito correspondem aos códigos introduzidos devido à geração dos mutantes.

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

#define PWD_LEN 5
#define PWD "myPas

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;
#if MUTA == 1
      char Buffer[(PWD_LEN + 1)-1];
#elif MUTA == 2
      char Buffer[(PWD_LEN + 1)/2];
#elif MUTA == 3
     char Buffer[(PWD_LEN + 1)*2];
#elif MUTA == 4
     char Buffer[1];
#else
     char Buffer[PWD_LEN + 1];
#endif

 strcpy (Buffer,szPWD);
 if (strcmp (Buffer,PWD)==0)
       iRet = TRUE;
 return iRet;
}

int main (int argc, char *argv[])
{
 if (argc==1)
       return;
 if (IsPassValid(argv[1]))
       printf ("Password Correto\n");
 else
      printf ("Password Incorreto\n");
}
```

Figura E.1 – Código do Programa Mutante-Fonte

a.1 Coleta de Dados utilizando dtpass0

Executou-se o programa mutante da Figura E.1 com o conjunto de dados de teste dtpass0 da Tabela E.1. Os dados de teste deste conjunto estão dentro do limite de tamanho esperado pelo programa original, de cinco caracteres. Os resultados das execuções constam na Tabela E.9. Através dos dados obtidos com as execuções, observou-se o comportamento dos mutantes.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	RPI
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	RPI	RPC	MF

Tabela E.9 – Resultado das execuções com o conjunto de dados de teste dtpass0. Para a compreensão das legendas veja a Tabela E.7.

a.2 Análise dos Resultados - dtpass0

Como se pode observar, o programa original comportou-se como esperado, executando todos os dados de testes normalmente dando como resultado “Password Incorreto”. Os mutantes 1, 2 e 3 forneceram o mesmo resultado do programa original. O único mutante cujo resultado diferiu do dado pelo programa original foi o mutante 4. Portanto, foi o único mutante morto por este conjunto de dados de teste.

Com o DT2, o mutante 4 resultou em “Password Correto”, resultado diferente do programa original e com o DT3, falha de segmentação. O atacante poderia ter um acesso não autorizado ao sistema sem necessitar injetar códigos maliciosos e sem precisar modificar o endereço de retorno da função para conseguir obter este privilégio. A vulnerabilidade apresentada por esta aplicação é um problema grave de segurança. Isto pelo fato da execução do mutante 4 com o DT2 levar à sobrescrita da variável iRet, mas não à sobrescrita do endereço base da função, portando validando a senha e resultando “Password Correto”, desta forma permitindo a qualquer usuário acessar o sistema.

A execução com o DT3 levou à sobrescrita do endereço base da função, pelo fato das senhas terem cinco caracteres, ocorrendo falha de segmentação. A não ocorrência de falha de segmentação ou um resultado diferente do programa original na execução do mutante 2 com o DT3 que tem senhas com cinco caracteres, leva à criação do conjunto de dados de teste dtpass1, descrito na Tabela E.2, com o objetivo de verificar se estes resultados mudam na execução do mutante 2 com o conjunto de dados de teste dtpass1.

b.1 Coleta de Dados utilizando dtpass1

Executou-se o programa mutante da Figura E.1 com o conjunto de dados de teste dtpass1 descrito na Tabela E.2. Os dados de teste deste conjunto estão dentro do limite de tamanho esperado pelo programa original, de cinco caracteres. Os resultados das execuções constam da Tabela E.10.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	RPI
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	RPC	RPC	MF

Tabela E.10 – Resultado das execuções com o conjunto de dados de teste dtpass1. Para a compreensão das legendas veja a Tabela E.7.

b.2 Análise dos Resultados – dtpass1

Como se pode observar, o programa original comportou-se como esperado, executou todos os dados de testes normalmente dando como resultado “Password Incorreto”. Os mutantes 1, 2 e 3 forneceram o mesmo resultado do programa original. O mutante 4 foi o único mutante eliminado também por este conjunto de dados de teste.

Com o DT1 e DT2 o mutante 4 resultou em “Password Correto”, resultado diferente do programa original e com DT3, falha de segmentação. A execução do mutante 4 com o DT1 e DT2 levou à sobrescrita da variável iRet, mas não à sobrescrita do endereço base da função, portando dando o resultado “Password Correto” e permitindo novamente o acesso do usuário ao sistema com uma senha incorreta. A execução com o DT3 levou à sobrescrita do endereço base da função, pelo fato da senha ter cinco caracteres, ocorrendo falha de segmentação.

Mas o mutante 2 que também reduz a quantidade de memória, não foi eliminado. O DT2 e DT3 deveriam forçá-lo a gerar um resultado diferente do programa original, o que não ocorreu. Gerou-se o conjunto de dados de teste dtpass2 descrito na Tabela E.3, que tem senhas de tamanho superior ao suportado pelo programa original, com o objetivo de verificar o comportamento dos mutantes com estes dados.

c.1 Coleta de Dados utilizando dtpass2

Executou-se o programa mutante da Figura E.1 com o conjunto de dados de teste dtpass2 descrito na Tabela E.3. Este conjunto contém seis dados de teste de entrada, todos com tamanho de caracteres acima do limite esperado pelo programa original, tentando com isto eliminar o mutante 3 e verificar o comportamento dos mutantes 1 e 2. Os resultados das execuções constam na Tabela E.11.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	RPI
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	MF	MF	MF

Tabela E.11 – Resultado das execuções com o conjunto de dados de teste dtpass2. Para a compreensão das legendas veja a Tabela E.7.

c.2 Análise dos Resultados – dtpass2

Como se pode observar, o programa original e os mutantes 1, 2 e 3 tiveram o mesmo comportamento. Todos forneceram como resultado “Password Incorreto”.

Os dados de entrada não alteraram o valor da variável iRet para resultar em “Password Correto”, nem o valor do endereço base da função e nem o endereço da próxima instrução a ser executada quando a função terminar, o que leva à falha de segmentação.

O único mutante que teve um comportamento diferente do programa original foi o mutante 4. Todos os dados de teste levaram-no à falha de segmentação, detectando com isto a

ocorrência de *buffer overflow*. Os mutantes 1, 2 e 3 mostraram-se equivalentes ao programa original.

d.1 Coleta de Dados utilizando dtpass3

A ocorrência de falha de segmentação no mutante 4 acontece na sua execução com dados de entrada de tamanho acima de cinco caracteres. O mutante 4 tem a quantidade de memória alocada para a variável Buffer, igual a um byte e somente depois de armazenar 5 bytes ou mais ocorre falha de segmentação. Senhas com tamanho entre 2 e 4 caracteres levam-no à "Password Correto".

Executou-se o programa mutante da Figura E.1 com o conjunto de dados de teste dtpass3 descrito na Tabela E.4, criado com o objetivo de verificar em qual circunstância o programa original e conseqüentemente os mutantes 1, 2 e 3 começam a fornecer um resultado diferente de "Password Incorreto" ou mesmo falha de segmentação.

Este conjunto contém seis dados de teste de entrada, todos com tamanho de caracteres acima do limite esperado pelo programa original. Os resultados das execuções constam na Tabela E.12.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPC	PF
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	MF	MF	MF

Tabela E.12 – Resultado das execuções com o conjunto de dados de teste dtpass3. Para a compreensão das legendas veja a Tabela E.7.

d.2 Análise dos Resultados – dtpass3

Como se pode observar, executando o programa original com o DT1 que tem dois dados de teste de vinte caracteres cada, o programa resulta em "Password Incorreto", com o DT2, "Password Correto" e com o DT3, falha de segmentação. Os mutantes 1, 2 e 3 têm os mesmos resultados do programa original confirmando a equivalência entre eles, e o mutante 4 resulta

numa saída diferente do programa original através de falha de segmentação na sua execução com o DT1 e o DT2.

Conclui-se que o único mutante que difere do programa original é o mutante que aloca a quantidade de memória igual a um. Portanto, executando o programa a ser testado com os dados de entrada dentro de esperado, o mutante 4 detecta a presença de uma vulnerabilidade do tipo *buffer overflow* através de um resultado diferente do esperado ou através de uma falha de segmentação. Os mutantes 1, 2 e 3 mostraram ser equivalentes ao programa original.

Somente com senhas de tamanho acima de 30 caracteres o programa original e os mutantes 1, 2 e 3 resultam em "Password Correto" e com senhas acima de 40 caracteres, falha de segmentação.

Monitorou-se a memória durante a execução do programa mutante, objetivando entender o motivo da não ocorrência de falha na execução do programa original e dos mutantes com conjuntos de dados de teste que, teoricamente, deveriam levar estas aplicações à falha de segmentação.

e. Monitoração da Memória

Executou-se o programa da Figura E.1 com os mesmos conjuntos de dados de teste dtpass0, dtpass1, dtpass2 e dtpass3 descritos nas Tabelas E.1, E.2, E.3 e E.4 respectivamente, monitorando as alocações de memória durante as execuções.

Na compilação do programa é reservada uma quantidade de memória para as variáveis locais de cada função. Dentro do depurador gdb versão 5.3, o comando *disassemble* seguido do nome da função, mostra o código *assembly* gerado para cada função. No início deste código há uma instrução que indica a quantidade de bytes reservada para estas variáveis [Gia01].

Verificou-se a quantidade de memória reservada para as variáveis locais das funções, concluindo que era a mesma para o programa original e os mutantes 1, 2 e 3, ou seja, 40 bytes ou 10 *dwords*. Para o mutante 4 foram reservados 8 bytes ou 2 *dwords*.

Para armazenar um int são necessários 4 bytes e um char, 1 byte. Como já mencionado no Capítulo 2, o espaço para as variáveis é reservado em *dwords* e não em bytes e, em uma *dword* armazenam-se quatro bytes. Sendo assim, no caso do mutante 4, apesar de serem necessários somente 5 bytes para as variáveis locais, são reservadas 2 *dwords*, ou 8 bytes.

As Figuras E.2 e E.3 ilustram parte do *stack frame* da função IsPassValid do programa original juntamente com os mutantes 1, 2 , 3 e do mutante 4 respectivamente. Pelo fato de o compilador reservar a mesma quantidade de memória para as variáveis locais do programa original e dos mutantes 1, 2 e 3, explica-se a equivalência entre eles.

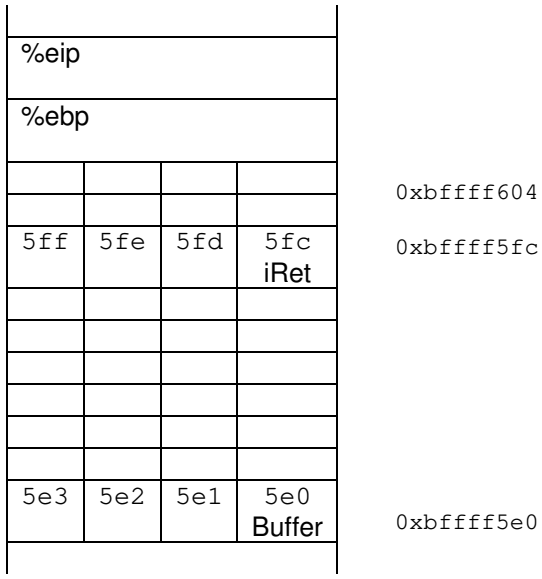

Figura E.2 – *Stack frame* da Função IsPassValid do Programa Original e dos Mutantes 1, 2, 3

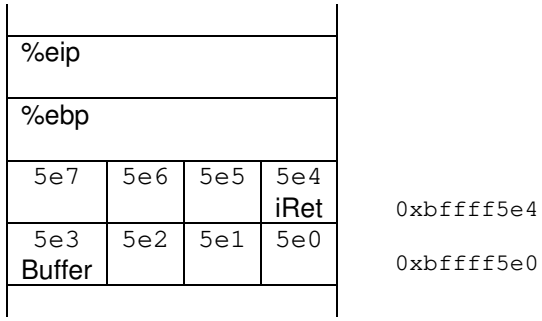

Figura E.3 – *Stack frame* da Função IsPassValid do Mutante 4

Como se pode observar pela Figura E.2, a variável Buffer está alocada no endereço de memória `0xbffff5e0` e a variável iRet, no endereço `0xbffff5fc`. Teoricamente é como se fossem reservados 28 bytes para a variável Buffer e 12 bytes para a variável iRet. Isto devido ao fato de não estar sendo utilizada a memória entre as duas variáveis e nem as duas *dwords* acima do endereço da variável iRet.

Apesar do crescimento da pilha ocorrer do *high address* para o *low address*, a atribuição dos valores às variáveis ocorre do *low address* para o *high address.*

Uma senha com até 28 caracteres não altera a variável iRet, pois o 29º caractere é o final de *string* '/0', o que não muda o seu valor inicial, dando como resultado "Password Incorreto".

Senhas com 29 até 39 caracteres fornecem como resultado “Password Correto” e senhas com 40 ou mais caracteres, leva o programa original e os mutantes 1, 2 e 3 à falha de segmentação, pois a partir do 40º caractere ocorre sobrescrita do %ebp da função que chamou.

Como a quantidade de memória reservada para o mutante 4 de acordo com a Figura E.3, é diferente da quantidade reservada para o programa original, este mutante é o único que consegue detectar a presença de vulnerabilidade do tipo *buffer overflow*, através de um resultado diferente do programa original ou através de falha de segmentação. Desta forma, consegue-se explicar os resultados obtidos na execução do programa mutante com os conjuntos de dados de teste dtpass0, dtpass1, dtpass2 e dtpass3 descritos nas Tabelas E.1, E.2, E.3 e E.4 respectivamente.

Executando o programa original e os mutantes 1, 2 e 3 com estes conjuntos de dados de teste:

- senhas com até 28 caracteres levam estes programas ao resultado “Password Incorreto” pelos motivos explicados acima;
- senhas entre 29 e 39 caracteres levam estes programas ao resultado “Password Correto”, pois sobrescrevem o conteúdo da variável iRet; e
- senhas com 40 ou mais caracteres levam estes programas à falha de segmentação, pois sobrescrevem o conteúdo do %ebp e %eip.

Executando o programa original e o mutante 4 com os conjuntos de dados de teste dtpass0, dtpass1, dtpass2 e dtpass3:

- senhas com 1 caractere levam o mutante 4 ao resultado “Password Incorreto”, pelos motivos explicados acima;
- senhas entre 2 e 5 caracteres levam o mutante a detectar a presença de *buffer overflow* através do resultado “Password Correto”, pois sobrescrevem o conteúdo da variável iRet; e
- senhas acima de 5 caracteres levam o mutante 4 à falha de segmentação, pois sobrescrevem o conteúdo do %ebp e do %eip detectando a presença de *buffer overflow*.

E.1.1.2 Variáveis Locais Invertidas

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada localmente à função, cuja posição foi alterada nas declarações em relação ao programa original em teste.

Trocaram-se as posições das declarações das variáveis locais dentro do programa ilustrado na Figura 5.2 do Capítulo 5, para analisar o comportamento dos mutantes com o objetivo de verificar se a mudança altera a quantidade de memória alocada para as variáveis locais e se ocasiona alguma modificação nos resultados obtidos com os experimentos já realizados na classe de experimento “Variáveis Locais sem Inversão”.

Com a modificação, o código do programa fonte assemelha-se ao da Figura 5.2, exceto as instruções das declarações das variáveis que ficam na ordem invertida. A Figura E.4 ilustra o programa fonte modificado.

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
     char Buffer[PWD_LEN + 1];
     int iRet = FALSE;

     strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
          iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
          return;
     if (IsPassValid(argv[1]))
          print ("Password Correto\n");
     else
          print ("Password Incorreto\n");
}
```

Figura E.4 – Código do Programa com Variáveis Locais Invertidas

A aplicação do operador MSMA a este programa resulta na geração de quatro mutantes com a quantidade de memória alocada para o Buffer de acordo com a Tabela E.8. A Figura E.5 mostra o programa mutante-fonte resultante.

a.1 Coleta de Dados utilizando dtpass0, dtpass1 e dtpass2

Executou-se o programa original e os mutantes com os mesmos conjuntos de dados de teste de entrada já utilizados no experimento da classe "Variáveis Locais sem Inversão", ou seja, dtpass0, dtpass1 e dtpass2 descritos nas Tabelas E.1, E.2 e E.3 respectivamente. A Tabela E.13 mostra os resultados das execuções do programa original e dos mutantes, que foram os mesmos para todos os conjuntos.

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

#define PWD_LEN 5
#define PWD "myPas

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
#if MUTA == 1
      char Buffer[(PWD_LEN + 1)-1];
#elif MUTA == 2
      char Buffer[(PWD_LEN + 1)/2];
#elif MUTA == 3
     char Buffer[(PWD_LEN + 1)*2];
#elif MUTA == 4
     char Buffer[1];
#else
     char Buffer[PWD_LEN + 1];
#endif
     int iRet = FALSE;

 strcpy (Buffer,szPWD);
 if (strcmp (Buffer,PWD)==0)
      iRet = TRUE;
 return iRet;
}

int main (int argc, char *argv[])
{
 if (argc==1)
      return;
 if (IsPassValid(argv[1]))
      printf ("Password Correto\n");
 else
     printf ("Password Incorreto\n");
}
```

Figura E.5 – Código do Programa Mutante-Fonte com Declarações Locais Invertidas

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	RPI
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	MF	MF	MF

Tabela E.13 – Resultado das execuções dos mutantes com as declarações invertidas. Para a compreensão das legendas veja a Tabela E.7.

a.2 Análise dos Resultados – dtpass0, dtpass1 e dtpass2

A quantidade de memória reservada para as variáveis locais no *stack frame* da função *IsPassValid* do programa original e dos mutantes 1, 2 e 3, é de 40 bytes ou 10 *dwords*, e a quantidade reservada para o mutante 4, é de 8 bytes ou 2 *dwords.* Isto leva à equivalência entre o programa original e os mutantes 1, 2 e 3, como no caso dos experimentos da classe de experimentos "Variáveis Locais sem Inversão".

As Figuras E.6 e E.7 ilustram parte do *stack frame* da função IsPassValid do programa original juntamente com os mutantes 1, 2 , 3 e do mutante 4 respectivamente. Pela Figura E.6, percebe-se que o sistema reservou 24 bytes para a variável local Buffer armazenar a sua informação. Isto para o programa original e para os mutantes 1, 2 e 3. Pela Figura E.7, o sistema reservou um byte para a variável Buffer armazenar a sua informação. Isto para o mutante 4, que aloca quantidade de memória igual a um.

Como se pode observar pela Figura E.6, uma senha com até 23 caracteres é suportada pelo programa original e pelos mutantes 1, 2 e 3 sem que ocorra falha. Somente senhas acima de 23 caracteres levam estes mutantes à falha de segmentação, pois sobrescrevem o endereço de memória que armazena o conteúdo do %ebp da função que chamou.

Já para o mutante 4, qualquer senha vai levá-lo à falha de segmentação. Como se pode observar pela Figura E.7, a posição de memória cujo endereço é `0xbffff5e7` que armazena o conteúdo da variável Buffer, está a uma posição abaixo do endereço que armazena o endereço base da função que chamou, o %ebp. Desta forma, qualquer senha armazenada na variável Buffer sobrescreve o %ebp, levando o mutante 4 à falha de segmentação.

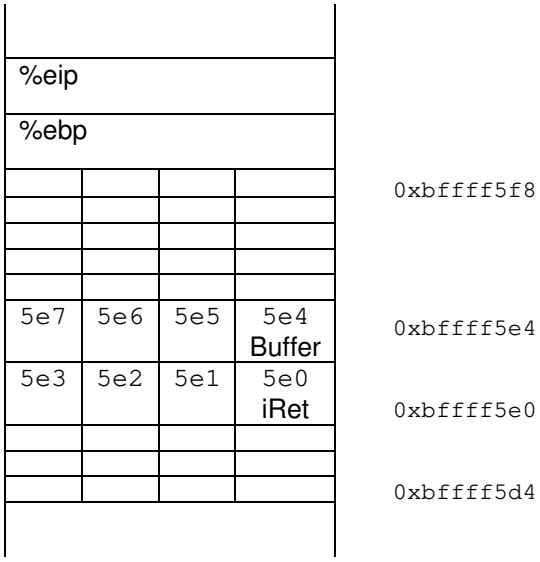

Figura E.6 – *Stack frame* da Função IsPassValid do Prog Original e dos Mutantes 1, 2, 3

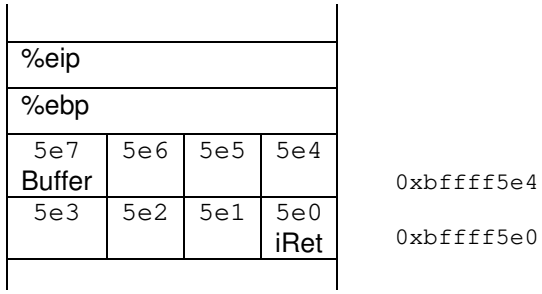

Figura E.7 – *Stack frame* da Função IsPassValid do Mutante 4

Pelos resultados da Tabela E.13, pode-se observar que o programa original e os mutantes 1, 2 e 3 são equivalentes e se comportaram como o esperado, dando como resultado "Password Incorreto". Isto pelo fato da senha com maior número de caracteres dos conjuntos de dados de teste dtpass0, dtpass1 e dtpass2 ter 15 caracteres. Como para estes programas pode-se entrar com uma senha de até 23 caracteres sem que ocorra falha, os resultados estão dentro do esperado, "Password Incorreto".

Já o mutante 4, resultou em falha de segmentação na execução com todos os dados de teste, como previsto.

b.1 Coleta de Dados utilizando dtpass3

Executou-se o programa original e os mutantes com o conjunto de dados de teste de entrada dtpass3 descrito na Tabela E.4. A Tabela E.14 mostra os resultados obtidos.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	PF	PF
Mutante 1 Quantidade alocada-1=5	RPI	MF	MF
Mutante 2 Quantidade alocada/2=3	RPI	MF	MF
Mutante 3 Quantidade alocada*2=12	RPI	MF	MF
Mutante 4 Quantidade alocada=1	MF	MF	MF

Tabela E.14 – Resultado das execuções com o dtpass3 dos mutantes com as declarações invertidas. Para a compreensão das legendas veja a Tabela E.7.

b.2 Análise dos Resultados – dtpass3

Como se pode observar pela Figura E.12, a execução do programa original e dos mutantes 1, 2 e 3 com DT1, que tem senhas de 20 caracteres, levam estes programas ao resultado ”Password Incorreto”. Isto pelo fato destes programas suportarem senhas de até 23 caracteres, sem que ocorra falha de segmentação.

A execução do programa original e dos mutantes 1, 2 e 3 com os dados de teste DT2 e DT3, levam-nos à falha de segmentação, pois o DT2 tem senhas de 30 caracteres e o DT3 tem senhas de 40 caracteres.

O mutante 4 que aloca quantidade de memória igual a um, teve como resultado falha de segmentação na execução com o DT1, o DT2 e o DT3. Isto pelo fato de o sistema ter reservado um byte para a variável Buffer armazenar a sua informação.

E.1.1.3 Alteração do Tamanho do *Buffer*

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada localmente à função.

Questionou-se se independentemente do tamanho da variável Buffer, a aplicação do operador MSMA que aloca quantidade de memória menos um, quantidade dividida por dois e quantidade multiplicada por dois, leva sempre o compilador a reservar a mesma quantidade de memória para as variáveis locais do programa original e dos mutantes gerados com estas características.

Alterou-se o tamanho da variável Buffer do programa fonte esboçado na Figura 5.2 do Capítulo 5, com o objetivo de comprovar a equivalência entre o programa original e os mutantes 1, 2 e 3, independente do tamanho do *buffer*. Modificou-se o tamanho declarado da variável Buffer de 6 para 31 caracteres. O programa fonte resultante é ilustrado na Figura E.8.

A aplicação do operador MSMA ao programa da Figura E.8, resulta na geração de quatro mutantes com a quantidade de memória alocada para a variável Buffer de acordo com a Tabela E.15. A Figura E.9 ilustra o código do programa mutante-fonte resultante.

a.1 Coleta de Dados utilizando dtpass4

Executou-se o programa mutante da Figura E.9 com o conjunto de dados de teste dtpass4 descrito na Tabela E.5. Os dados de teste deste conjunto estão dentro e fora do limite de tamanho esperado pelo programa original e constam do Apêndice D.5. Os resultados das execuções são mostrados na Tabela E.16.

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

#define PWD_LEN 30
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;
     char Buffer[PWD_LEN + 1];

     strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
          iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
          return;
     if (IsPassValid(argv[1]))
          print ("Password Correto\n");
     else
          print ("Password Incorreto\n");
}
```

Figura E.8 – Código do Programa com *Buffer* de Tamanho 31

Programa	**Característica**	**Quantidade Alocada em Bytes**	**Quantidade Reservada na Memória**
Original	-	31	56 bytes ou 14 *dwords*
Mutante 1	Quantidade alocada -1	30	56 bytes ou 14 *dwords*
Mutante 2	Quantidade alocada / 2	15	40 bytes ou 10 *dwords*
Mutante 3	Quantidade alocada * 2	62	88 bytes ou 22 *dwords*
Mutante 4	Quantidade alocada = 1	1	8 bytes ou 2 *dwords*

Tabela E.15 – Quantidade de Memória Alocada para os Programas

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

#define PWD_LEN 30
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
 int iRet = FALSE;

#if MUTA == 1
 char Buffer[(PWD_LEN + 1)-1];
#elif MUTA == 2
 char Buffer[(PWD_LEN + 1)/2];
#elif MUTA == 3
 char Buffer[(PWD_LEN + 1)*2];
#elif MUTA == 4
 char Buffer[1];
#else
 char Buffer[PWD_LEN + 1];
#endif

 strcpy (Buffer,szPWD);
 if (strcmp (Buffer,PWD)==0)
  iRet = TRUE;
 return iRet;
}

int main (int argc, char *argv[])
{
 if (argc==1)
  return;
 if (IsPassValid(argv[1]))
  printf ("Password Correto\n");
 else
  printf ("Password Incorreto\n");
}
```

Figura E.9 – Código do Programa Mutante-Fonte com *Buffer* de Tamanho 31

PROGRAMAS	DT 1	DT 2	DT 3	DT4
Programa Original Quantidade alocada=31	RPI	RPI	RPI	PF
Mutante 1 Quantidade alocada-1=30	RPI	RPI	RPI	MF
Mutante 2 Quantidade alocada/2=15	RPI	RPC	MF	MF
Mutante 3 Quantidade alocada*2=62	RPI	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	MF	MF	MF	MF

Tabela E.16 – Resultado das execuções com o conjunto de dados de teste dtpass4. Para a compreensão das legendas veja a Tabela E.7.

a.2 Análise dos Resultados – dtpass4

Verificou-se a quantidade de memória reservada para armazenar as variáveis locais do programa original e dos mutantes e percebeu-se que esta quantidade varia de mutante para mutante, diferente do que ocorre quando o Buffer tem tamanho igual a 6 caracteres, como no caso das classes de experimentos "Variáveis Locais sem Inversão" e "Inversão das Variáveis", nas quais a quantidade de memória reservada pelo compilador para as variáveis locais é a mesma para o programa original e para os mutantes 1, 2 e 3. A quantidade de memória reservada para o programa original e para cada mutante constam da Tabela E.15.

Como se pode observar, o mutante 1, que aloca quantidade de memória menos 1 byte, é o único que continua equivalente ao programa original, reservando a mesma quantidade de memória no *stack frame* da função *IsPassValid*.

Na monitoração da alocação da memória durante a execução do programa mutante, verificou-se que existem 11 *dwords* ou 44 bytes livres entre a variável Buffer e a variável iRet no *stack frame* da função *IsPassValid* do programa original e do mutante 1. Desta forma, senhas com até 44 caracteres não levam estes programas a um resultado diferente de "Password Incorreto". Senhas com 45 até 55 caracteres sobrescrevem a variável iRet, resultando em "Password Correto" e senhas com tamanho acima de 55 caracteres, leva o programa original e o mutante 1 à falha de segmentação.

No caso do mutante 2, que aloca para a variável Buffer quantidade de memória dividida por dois, o compilador reserva 40 bytes ou 10 *dwords* para as suas variáveis locais. Verificou-se que existem 28 bytes ou 7 dwords livres entre as variáveis Buffer e iRet no *stack frame* da função *IsPassValid*. Portanto, senhas com até 28 bytes não alteram o valor da variável iRet levando este mutante a resultar "Password Incorreto". Senhas com 29 até 39 caracteres sobrescrevem a variável iRet resultando em "Password Correto" e senhas com tamanho acima de 39 caracteres, levam o mutante 2 à falha de segmentação, pois sobrescrevem o conteúdo do %ebp.

Na análise dos resultados das execuções ilustrados na Tabela E.16, o programa original e os mutantes 1, 2 e 3 executaram normalmente com o DT1, que tem senhas de 15 caracteres, resultando "Password Incorreto", como esperado. Já a execução do mutante 4, que aloca quantidade de memória igual a um, ocasionou falha de segmentação na sua execução com o DT1, detectando a presença de *buffer overflow*.

Na execução do programa original e dos mutantes com o DT2, que tem senhas de 30 caracteres, os mutantes 2 e 4 detectaram a presença de vulnerabilidade do tipo *buffer overflow*. O mutante 2 indicou *buffer overflow* através de um resultado diferente do esperado pelo programa original, como "Password Correto" e o mutante 4 indicou através de falha de segmentação. Todos os resultados na execução do programa original e dos mutantes com o DT2 estão dentro do esperado.

Na execução dos programas com o DT3, que tem senhas de 40 caracteres, o programa original e os mutantes 1 e 3 apresentaram como resultado "Password Incorreto", como esperado. Já a execução do mutante 2 e do mutante 4 com o DT3 levam-os a detectar a presença de *buffer overflow* através de falha de segmentação.

Na execução dos programas com o DT4, que tem senhas de 60 caracteres, o programa original e os mutantes 1, 2 e 4 apresentaram falha de segmentação, dentro do esperado. A execução do mutante 3, que dobra a quantidade alocada para a variável Buffer, com o DT4, leva-o a detectar a presença de *buffer overflow* através de um resultado diferente do programa original, "Password Incorreto". A execução do programa original com este dado de teste leva-o à falha de segmentação, conseguindo eliminar, desta forma, o mutante 3. Assim sendo, para conseguir a detecção da presença de *buffer overflow* através do mutante 3, o dado de teste de entrada deve ter uma quantidade de caracteres que leve o programa original à falha de segmentação, mas não pode ser grande o suficiente para levar o mutante 3 também à falha.

Como se pode observar pela Tabela E.16, a execução do programa original e dos mutantes com o conjunto de dados de teste dtpass4 elimina todos os mutantes, exceto o mutante 1 que é equivalente ao programa original. Estes resultados são diferentes dos resultados obtidos nos experimentos das classes anteriores, quando o Buffer tem tamanho igual a 6 caracteres, revelando que somente o mutante 4, que aloca quantidade de memória igual a um, é eficaz na detecção da presença de *buffer overflow*.

Portanto, executar o programa original e os mutantes com valores de entrada dentro ou fora do limite esperado pelo programa original, mostram que os mutantes 2 e 3 também são eficazes, além do mutante 4, em detectar a presença de *buffer overflow* através de um resultado diferente do programa original ou através de falha de segmentação, revelando que estes mutantes nem sempre são equivalentes ao programa original.

E.1.2 Categoria Global

E.1.2.1 Variáveis Globais

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a aplicação do operador MSMA ocorre numa variável declarada globalmente em um programa.

O segmento de dados bss ilustrado na Figura 2.3 do Capítulo 2, é o segmento dentro da memória de um processo do Linux reservado para armazenar as variáveis globais não iniciadas.

Como o objetivo deste experimento é analisar o comportamento dos mutantes quando o operador MSMA é aplicado às variáveis globais, alterou-se o programa fonte da Figura 5.2 do Capítulo 5 mudando a declaração da variável Buffer de local para global. A Figura E.10 ilustra o código fonte do programa resultante.

A aplicação do operador MSMA ao programa da Figura E.10, resulta na geração de quatro mutantes com a quantidade de memória alocada para a variável Buffer de acordo com a Tabela E.17. A Figura E.11 ilustra o código do programa mutante-fonte resultante.

Programa	**Característica**	**Qtde Alocada em bytes**	**Qtde Reservada no** bss
Original	**-**	6	8 bytes ou 2 *dwords*
Mutante 1	Quantidade alocada -1	5	8 bytes ou 2 *dwords*
Mutante 2	Quantidade alocada / 2	3	4 bytes ou 1 *dword*
Mutante 3	Quantidade alocada * 2	12	12 bytes ou 3 *dwords*
Mutante 4	Quantidade alocada = 1	1	4 bytes ou 1 *dword*

Tabela E.17 – Quantidade Reservada no Segmento bss para cada Programa

A quantidade de memória reservada no segmento de dados bss, para a variável Buffer de cada mutante gerado, é ilustrada na Tabela E.17. Como se pode observar a quantidade alocada em *dwords* está bem próxima do necessário, diferentemente de quando a variável é local.

O mutante 1 é equivalente ao programa original e neste caso os mutantes 2 e 4 também são equivalentes entre si, pois foram reservadas duas *dwords* para cada um para armazenar o conteúdo do *buffer*. Desta forma, se for possível eliminar o mutante 2, também será possível eliminar o mutante 4.

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

#define PWD_LEN 5
#define PWD “myPas”

#define TRUE 1
#define FALSE 0

char Buffer[PWD_LEN + 1];

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;

     strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
          iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
          return;
     if (IsPassValid(argv[1]))
          print (“Password Correto\n”);
     else
          print (“Password Incorreto\n”);
}
```

Figura E.10 – Código do Programa com Declaração Global

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

#if MUTA == 1
        char Buffer[(PWD_LEN + 1)-1];
#elif MUTA == 2
        char Buffer[(PWD_LEN + 1)/2];
#elif MUTA == 3
        char Buffer[(PWD_LEN + 1)*2];
#elif MUTA == 4
        char Buffer[1];
#else
        char Buffer[PWD_LEN + 1];
#endif

int IsPassValid(char *szPWD)
{
      int iRet = FALSE;
      strcpy (Buffer,szPWD);
      if (strcmp (Buffer,PWD)==0)
       iRet = TRUE;
      return iRet;
}

int main (int argc, char *argv[])
{
      if (argc==1)
       return;
      if (IsPassValid(argv[1]))
       printf ("Password Correto\n");
      else
       printf ("Password Incorreto\n");
}
```

Figura E.11 – Código do Programa Mutante-Fonte com Declaração Global

a.1 Coleta de Dados utilizando dtpass0 e dtpass1

Executou-se o programa mutante da Figura E.11 com os conjuntos de dados de teste dtpass0 e dtpass1 descritos nas Tabelas E.1 e E.2. Os dados de teste destes conjuntos estão dentro do limite de tamanho esperado pelo programa original, de cinco caracteres e constam do Apêndice D.1 e D.2 respectivamente. Os resultados das execuções constam da Tabela E.18.

As execuções do programa mutante geraram os mesmos resultados para os dois conjuntos de dados de teste, dtpass0 e dtpass1, em relação ao DT1, DT2 e DT3.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	RPI
Mutante 1 Quantidade alocada-1=5	RPI	RPI	RPI
Mutante 2 Quantidade alocada/2=3	RPI	RPI	RPI
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	RPI	RPI	RPI

Tabela E.18 – Resultado das execuções com o conjunto de dados de teste dtpass0 e dtpass1 com variáveis globais. Para a compreensão das legendas veja a Tabela E.7.

a.2 Análise dos Resultados – dtpass0 e dtpass1

Como se pode observar, todos os programas executaram dando como resultado "Password Incorreto". Os dados de teste DT2 e DT3 do dtpass1, que têm senhas de 4 e 5 caracteres respectivamente e o DT3 do dtpass0, que tem senhas de 5 caracteres, teoricamente deveriam levar o mutante 2, que aloca quantidade de memória dividida por dois e o mutante 4, que aloca quantidade de memória igual a um, à falha de segmentação.

Pela Figura E.12 que ilustra o *layout* do segmento de dados bss da memória do mutante 4, a utilização dos dados de teste citados acima, o DT2 e o DT3, deveriam sobrescrever a *dword have_no_fcntl64* apresentando resultados diferentes.

Os mutantes foram executados com dados testes de tamanho acima do esperado pelo programa original, com o objetivo de verificar as mudanças ocorridas nos resultados.

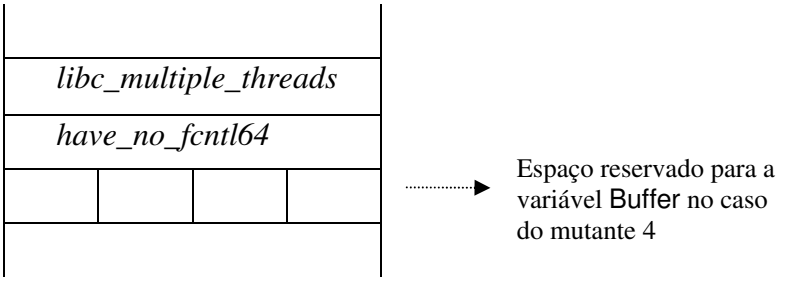

Figura E.12 – *Layout* do Segmento de Dados bss

b.1 Coleta de Dados utilizando dtpass2

Executou-se o programa mutante da Figura E.11 com o conjunto de dados de teste dtpass2 descrito na Tabela E.3. Todos os dados deste conjunto têm o tamanho de caracteres acima do esperado pelo programa original. Os resultados das execuções constam na Tabela E.19.

PROGRAMAS	DT 1	DT 2	DT 3
Programa Original Quantidade alocada=6	RPI	RPI	PF
Mutante 1 Quantidade alocada-1=5	RPI	RPI	MF
Mutante 2 Quantidade alocada/2=3	RPI	MF	MF
Mutante 3 Quantidade alocada*2=12	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	RPI	MF	MF

Tabela E.19 – Resultado das execuções com o conjunto de dados de teste dtpass2 com variáveis globais. Para a compreensão das legendas veja a Tabela E.7.

b.2 Análise dos Resultados – dtpass2

Como se pode observar, os mutantes 2 e 4 têm como resultados das suas execuções com DT1, que tem senhas de 6 caracteres, “Password Incorreto”.

Com DT2, que tem senhas de 10 caracteres e com DT3, que tem senhas de 15 caracteres, os resultados dos mutantes 2 e 4 revelaram falha de segmentação. O mutante 1 é equivalente ao programa original fornecendo os mesmos resultados.

Estes resultados ocorrem pelo motivo da *dword* nomeada *have_no_fcntl64* da Figura E.12 não ser utilizada por nenhuma função, podendo ser sobrescrita sem que ocorra erro.

Sendo assim, o programa original e os mutantes têm 4 bytes a mais, além dos já reservados para as suas variáveis globais, para serem utilizados sem que ocorra nenhum problema. Isto explica o motivo do DT1 não ter levado os mutantes 2 e 4 à falha de segmentação.

A *dword* nomeada *libc_multiple_threads* na Figura E.12, não pode ter em nenhum dos seus bytes um valor diferente de zero, caso isto ocorra, um acesso a esta *dword* por qualquer função do sistema leva à falha de segmentação. Desta forma, a ocorrência de falha de segmentação na execução dos mutantes 2 e 4 com os dados de teste DT2 e DT3, deve-se justamente pelo fato destes dados sobrescreverem esta *dword.* Portanto, a execução dos mutantes 2 e 4 com dados de teste de entrada de até 8 caracteres não os leva à falha de segmentação, pois apesar do

caractere de fim de string, o '/0', sobrescrever a *dword libc_multiple_threads,* este não altera o seu valor.

E.1.2.2 Alteração do Tamanho do *Buffer*

O objetivo desta classe de experimentos é analisar os resultados das execuções dos mutantes, quando a técnica é aplicada às variáveis declaradas globalmente em um programa.

Quando o *buffer* é declarado globalmente e tem um tamanho igual a seis caracteres, o mutante 1, que aloca quantidade de memória menos um, é equivalente ao programa original. São equivalentes entre si o mutante 2, que aloca quantidade de memória dividida por dois e o mutante 4, que aloca quantidade de memória igual a um. Modificou-se o tamanho do *buffer* com o objetivo de verificar a influência desta mudança nos resultados já obtidos, como ocorreu com este experimento na categoria local.

Alterou-se o tamanho da variável Buffer declarada globalmente no programa fonte ilustrado pela Figura E.10, modificando o seu tamanho de 6 para 31 caracteres. A Figura E.13 esboça o programa fonte resultante.

A aplicação do operador MSMA ao programa da Figura E.13 resulta na geração de quatro mutantes com a quantidade de memória alocada para o Buffer de acordo com a Tabela E.20. A quantidade de memória reservada no segmento de dados bss para a variável Buffer em cada mutante gerado também é mostrada na Tabela E.20. A Figura E.14 ilustra o código do programa mutante-fonte resultante.

Programa	**Característica**	**Qtde Alocada em bytes**	**Qtde Reservada no** bss
Original	**-**	31	32 bytes ou 8 *dwords*
Mutante 1	Quantidade alocada -1	30	32 bytes ou 8 *dwords*
Mutante 2	Quantidade alocada / 2	15	16 bytes ou 4 *dword*
Mutante 3	Quantidade alocada * 2	62	64 bytes ou 16 *dwords*
Mutante 4	Quantidade alocada = 1	1	4 bytes ou 1 *dword*

Tabela E.20 - Quantidade de Memória Alocada para os Programas

Como se pode observar, o mutante 1 é equivalente ao programa original independente do tamanho da variável Buffer. Entretanto, os mutantes 2 e 4 não são mais equivalentes entre si, pois são reservadas quantidades diferentes de memória para cada um.

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

#define PWD_LEN 30
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

char Buffer[PWD_LEN + 1];

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;

     strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
          iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
          return;
     if (IsPassValid(argv[1]))
          print ("Password Correto\n");
     else
          print ("Password Incorreto\n");
}
```

Figura E.13 - Código do Programa com *Buffer* de Tamanho 31

a.1 Coleta de Dados utilizando dtpass5

Executou-se o programa mutante da Figura E.14 com o conjunto de dados de teste dtpass5 descrito na Tabela E.6 e mostrado no Apêndice D.6. Os dados deste conjunto têm o tamanho de caracteres abaixo e acima do limite esperado pelo programa original. Os resultados das execuções constam da Tabela E.21

a.2 Análise dos Resultados – dtpass5

De acordo com o *layout* do segmento de dados bss da Figura E.12, existe uma *dword* nomeada *have_no_fcntl64* que pode ser sobrescrita sem que ocorra nenhum problema. Desta forma, o programa original e os mutantes têm uma *dword* a mais do que as descritas na Tabela E.20, para armazenar a informação de entrada.

Sendo assim, o programa original e o mutante 1 podem ser executados com senhas de até 36 caracteres sem que ocorra erro. O mutante 2, com senhas de até 20 caracteres e o mutante 4, com senhas de até 8 caracteres sem que ocorra falha de segmentação. No conjunto de dados dtpass5 o DT1 tem senhas de 30 caracteres, o DT2 de 40 caracteres, o DT3 de 50 caracteres e o DT4 tem senhas de 60 caracteres.

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

#define PWD_LEN 30
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

#if MUTA == 1
        char Buffer[(PWD_LEN + 1)-1];
#elif MUTA == 2
        char Buffer[(PWD_LEN + 1)/2];
#elif MUTA == 3
        char Buffer[(PWD_LEN + 1)*2];
#elif MUTA == 4
        char Buffer[1];
#else
        char Buffer[PWD_LEN + 1];
#endif

int IsPassValid(char *szPWD)
{
      int iRet = FALSE;
      strcpy (Buffer,szPWD);
      if (strcmp (Buffer,PWD)==0)
       iRet = TRUE;
      return iRet;
}

int main (int argc, char *argv[])
{
      if (argc==1)
       return;
      if (IsPassValid(argv[1]))
       printf ("Password Correto\n");
      else
       printf ("Password Incorreto\n");
}
```

Figura E.14 - Código do Programa Mutante-Fonte com *Buffer* de Tamanho 31

PROGRAMAS	DT 1	DT 2	DT 3	DT4
Programa Original Quantidade alocada=31	RPI	PF	PF	PF
Mutante 1 Quantidade alocada-1=30	RPI	MF	MF	MF
Mutante 2 Quantidade alocada/2=15	MF	MF	MF	MF
Mutante 3 Quantidade alocada*2=62	RPI	RPI	RPI	RPI
Mutante 4 Quantidade alocada=1	MF	MF	MF	MF

Tabela E.21 - Resultado das execuções com o conjunto de dados de teste dtpass5

Como se pode observar pela Tabela E.21, os mutantes 2 e 4 são eliminados através da execução do programa original com a senha dentro do tamanho esperado, o DT1, levando estes mutantes a detectarem a presença de *buffer overflow* através de falha de segmentação, diferentemente do que acontece quando a variável Buffer tem um tamanho pequeno, 6 bytes.

O mutante 3 detecta a presença de *buffer overflow*, executando o programa original com uma senha de tamanho acima do esperado, o DT2, o DT3 e o DT4, levando o programa original à falha de segmentação. Ou seja, a tentativa de eliminar o mutante 3 ocasiona falha no programa original.

E.2 Grupo Dinâmico

O objetivo é conduzir o experimento aplicando o operador MDMA a variáveis declaradas dinamicamente dentro do programa, analisando os resultados obtidos na execução do programa original e dos mutantes.

Alterou-se o programa fonte da Figura 5.2 do Capítulo 5 de modo que a alocação de memória para a variável Buffer aconteça de forma dinâmica. A Figura E.15 ilustra o código fonte resultante. A aplicação do operador MDMA ao programa da Figura E.15 e conseqüentemente a geração dos mutantes, resulta no programa mutante-fonte descrito na Figura E.16.

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
     int iRet = FALSE;
    char *Buffer;

     Buffer=(char *) malloc((PWD_LEN + 1) * sizeof(char));
    strcpy (Buffer,szPWD);
     if (strcmp(Buffer,PWD)==0)
      iRet = TRUE;
    return iRet;
  }

int main(int argc, char *argv[])
{
     if (argc==1)
      return;
    if (IsPassValid(argv[1]))
     print ("Password Correto\n");
     else
     print ("Password Incorreto\n");
}
```

Figura E.15 - Código do Programa com Alocação Dinâmica

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

#define PWD_LEN 5
#define PWD "myPas"

#define TRUE 1
#define FALSE 0

int IsPassValid(char *szPWD)
{
 int iRet = FALSE;
 char *Buffer;

#if MUTA == 1
     Buffer= (char*) malloc((PWD_LEN+1) * sizeof(char)-1);
#elif MUTA == 2
     Buffer= (char*) malloc((PWD_LEN+1) * sizeof(char)/2);
#elif MUTA == 3
     Buffer= (char*) malloc((PWD_LEN+1) * sizeof(char)*2);
#elif MUTA == 4
     Buffer= (char*) malloc(1);
#else
     Buffer= (char*) malloc((PWD_LEN+1) * sizeof(char));
#endif

 strcpy (Buffer,szPWD);
 if (strcmp (Buffer,PWD)==0)
  iRet = TRUE;
 return iRet;
}

int main (int argc, char *argv[])
{
 if (argc==1)
  return;
 if (IsPassValid(argv[1]))
  printf ("Password Correto\n");
 else
  printf ("Password Incorreto\n");
}
```

Figura E.16 - Código do Programa Mutante-Fonte com Alocação Dinâmica

A memória para o ponteiro do *buffer* é alocada no *stack frame* da função IsPassValid junto com a variável iRet, mas a alocação da memória para o Buffer ocorre na *heap*. A memória *heap* localiza-se perto da memória reservada para os *stack frames* das funções, como ilustrado na Figura 2.3 do Capítulo 2. Na *heap* aloca-se exatamente a memória que a variável precisa, diferentemente da alocação de memória para as variáveis locais. Entretanto, a *heap* tem uma área grande de memória e totalmente livre. Sendo assim, não importa o tamanho da informação de entrada que o resultado da execução do programa da Figura E.16 será sempre "Password Incorreto", não indicando nenhuma ocorrência de *buffer overflow*.