Ministério da
Ciência e Tecnologia

INPE-15297-TDI/1349

DESENVOLVIMENTO DE UM AMBIENTE PARA ANÁLISE DE CÓDIGOS-FONTE COM ÊNFASE EM SEGURANÇA

Luiz Otávio Duarte

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Antonio Montes Filho, aprovada em 27 de fevereiro de 2007

Registro do documento original:
<http://urlib.net/sid.inpe.br/mtc-m17@80/2007/04.23.11.26>

INPE
São José dos Campos
2008

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3945-6911/6923
Fax: (012) 3945-6919
E-mail: pubtc@sid.inpe.br

Ministério da
Ciência e Tecnologia

INPE-15297-TDI/1349

DESENVOLVIMENTO DE UM AMBIENTE PARA ANÁLISE DE CÓDIGOS-FONTE COM ÊNFASE EM SEGURANÇA

Luiz Otávio Duarte

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Antonio Montes Filho, aprovada em 27 de fevereiro de 2007

Registro do documento original:
<http://urlib.net/sid.inpe.br/mtc-m17@80/2007/04.23.11.26>

INPE
São José dos Campos
2008

Dados Internacionais de Catalogação na Publicação (CIP)

Duarte, Luiz Otávio.
D85d Desenvolvimento de um ambiente para análise de códigos-fonte com ênfase em segurança / Luiz Otávio Duarte. – São José dos Campos : INPE, 2008.
152 p. ; (INPE-15297-TDI/1349)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2007.
Orientador : Dr. Antonio Montes Filho.

1. Segurança. 2. Vulnerabilidade. 3.Linguagens de programação. 4. Verificação de programas. 5. Confiabilidade de software. I.Título.

CDU 004.056

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de Mestre em

Computação Aplicada

Dr. **José Carlos Becceneri**

Presidente / INPE / SJCampos - SP

Dr. **Antonio Montes Filho**

Orientador(a) / INPE / SJCampos - SP

Dr. **Nandamudi Lankalapalli Vijaykumar**

Membro da Banca / INPE / SJCampos - SP

Dr. **Edgar Toshiro Yano**

Convidado(a) / ITA / São José dos Campos - SP

Aluno (a): **Luiz Otávio Duarte**

São José dos Campos, 27 de Fevereiro de 2007

"Há imagens que o espelho mostra, e há coisas que não consegue refletir. Não há como esconder as rugas na face, mas o espelho não mostra as histórias que cada uma delas conta. Não há como esconder o corpo que não é de outrora, mas não ha reflexo das experiências vividas ao logo dos anos. Não há imagens que reflitam as lutas de um caminhar árduo.

Acostumados a ver o nosso exterior refletido, esquecemos que o que realmente somos está em nosso interior: nossos amores, nossas ilusões, nossas alegrias e tristezas podem até transparecer fisicamente, mas nunca serão refletidas totalmente."

JOSÉ AUGUSTO DUARTE

A meus pais,

LUIZ CARLOS DUARTE e

SOLANGE BARIONI DUARTE.

AGRADECIMENTOS

A Deus, pois sei que Ele vive.

A meus pais, por todo apoio dado em cada um dos passos deste trabalho e por muitas vezes deixarem de viver suas próprias vidas para caminharem com a minha.

A minha namorada, Lívia Pirillo Grecchi, pela compreensão e apoio durante minha infindável ausência.

Ao meu orientador, Antonio Montes, pelos brilhantes ensinamentos e incansáveis discussões.

A todos os colegas, familiares e amigos que de uma forma ou de outra colaboraram ou compreenderam minha ausência durante a realização deste trabalho.

RESUMO

Este trabalho apresenta uma proposta de ambiente para análise de códigos-fonte com ênfase em segurança, tendo como objetivo primordial auxiliar desenvolvedores a encontrar reais vulnerabilidades presentes em seus próprios softwares. Dentre elas, as que podem ser encontradas com maior ocorrência são as que permitem ataques de extravasamento de buffer ou *buffer overflow*. São muitas as formas de explorar estas vulnerabilidades dentre elas destacam-se *stack smash*, *arc injection* e *pointer subterfuge*. Para endereçar estas vulnerabilidades existem algumas abordagens que tentam eliminá-las ou minimizar o impacto gerado quando estas são exploradas. As abordagens são divididas em abordagens dependentes do compilador, dependentes do sistema e dependentes da aplicação. O ambiente proposto, através de uma abordagem preventiva e dependente da aplicação, analisa o código-fonte do programa a fim de encontrar instruções que possam ocasionar um extravasamento de buffer. Para isso, o ambiente procura suprir limitações encontradas em ferramentas similares. Serão mostrados através dos testes realizados as vantagens na utilização de análise sintática para melhor identificação de problemas e como um protótipo viável do ambiente pôde ser implementado.

DEVELOPMENT OF A SOURCE CODE ANALYSIS AID TOOL FOCUSING ON SECURITY

ABSTRACT

This work presents a proposal of a source code analysis aid tool focused on security. Its main goal is to help developers to find real vulnerabilities in their own software. Among the vulnerabilities that can be found in a software, *buffer overflows* have greater occurrence. There are several ways to exploit this vulnerability and some of them are *stack smash*, *arc injection*, *pointer subterfuge*. To address these vulnerabilities there are some approaches that try to mitigate these vulnerabilities or to minimize the impact generated when they are exploited. The approaches are divided into compiler-dependent, system-dependent and software-dependent. The proposed tool analyzes the source code of a software to find *buffer overflows* vulnerabilities through a preventive and software-dependent approach. To achieve it, the tool tries to supply limitations found in similar tools. The advantages of sintatic analysis to improve the buffer overflow identification and the development of a viable prototype of this tool will be shown in this work.

SUMÁRIO

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

CAPÍTULO 1

INTRODUÇÃO

Nas últimas décadas, observou-se um aumento expressivo no desenvolvimento de tecnologias computacionais, que permitem maiores taxas de processamento, capacidade de armazenamento, bem como de transmissão de dados. Tanto usuários domésticos como institucionais usufruem destas novas tecnologias.

O relativo barateamento dos sistemas computacionais fez com que houvesse uma grande disseminação destas tecnologias entre usuários domésticos. Estes procuram, sobretudo, facilidades, comodidade e lazer. Para suprir tais necessidades, empresas passaram a prover serviços de conectividade, de troca de mensagens, de venda de produtos, de acesso a bancos entre outros. Não somente serviços são disponibilizados aos usuários domésticos, mas também muitos programas são especificamente desenvolvidos para este segmento.

Usuários institucionais utilizam novas tecnologias seja por motivos estratégicos, de competitividade ou para prover os serviços e funcionalidades que seus clientes necessitam e exigem. O aumento da inclusão digital de empresas pode ser observado através do crescente número de registro de domínios realizados por estas entidades. O "Registro .br"[1], órgão responsável por registrar domínios com o sufixo `.BR`, possui mais de 937.000 domínios `.COM.BR`, que é destinado ao comércio em geral do Brasil, indexados em seu banco de dados. Atualmente são processadas mais de 40.000 requisições de registro de domínios por mês.

Os programas também vêm evoluindo na mesma proporção. Atualmente, muitos destes programas necessitam de conectividade com a Internet, alguns para funcionar de maneira adequada, outros para atualizações, registro, suporte ou ajuda. Além disso, existe um aumento na complexidade, no número de funcionalidades e no número de linhas de código destes programas.

Por outro lado, toda esta evolução é acompanhada por um número crescente de sistemas sendo sondados, invadidos, infectados, comprometidos e utilizados para gerar ataques a outros sistemas. Muitas vezes isto ocorre sem que o usuário do sistema alvo se quer perceba. Todos estes problemas de segurança tornaram-se o cotidiano de instituições tanto grandes como pequenas e de usuários domésticos.

Os ataques bem sucedidos nos sistemas computacionais são devidos a algum tipo

[1] Maiores informações sobre o registro .br estão disponíveis em: <http://registro.br>. Acesso em: 19 de Dez. de 2006

de vulnerabilidade. São justamente vulnerabilidades presentes em programas as mais exploradas em ataques(Viega e MacGraw, 2001). Estas vulnerabilidades podem possuir diferentes graus de criticidade e os programas afetados variam de aplicativos de usuários até sistemas operacionais.

1.1 Motivação

As vulnerabilidades em um sistema computacional podem ser classificadas como remotamente exploráveis ou localmente exploráveis. As vulnerabilidades remotamente exploráveis são as que podem ser exploradas a partir de máquinas externas. Geralmente, estas vulnerabilidades estão presentes em serviços disponibilizados na máquina afetada.

A exploração de uma vulnerabilidade remotamente explorável pode garantir, entre outras, acesso não autorizado ao sistema afetado (DUARTE et al., 2005b). Este tipo de vulnerabilidade quando explorada pode, por exemplo, invalidar o mecanimso de autenticação do programa ou o mecanismo de controle de acesso.

As vulnerabilidades localmente exploráveis são aquelas que só podem ser exploradas a partir do próprio sistema afetado. Durante o desenrolar de um ataque, estas vulnerabilidades costumam ser utilizadas para a realização do que é chamado de escalação de privilégios. Escalar privilégios é o processo realizado pelo atacante para permitir que um simples usuário consiga executar operações somente autorizadas aos super-usuários. Na grande maioria dos ataques envolvendo escalação de privilégios um simples usuário torna-se super-usuário no sistema.

Alguns institutos, como o *National Institute of Standards and Technologies* NIST[2], indexam em base de dados as vulnerabilidades descobertas e amplamente difundidas. O NIST com apoio do US-CERT[3] mantém uma base de dados, que é uma das mais conhecidas na atualidade, denominada *National Vulnerability Database* NVD[4]) que é uma das base de dados de vulnerabilidades mais conhecidas na atualidade.

A taxa de publicação de novas vulnerabilidades em Dezembro de 2006 no NVD girava em torno de vinte e três novas vulnerabilidades publicadas por dia. Sendo que no ano de 2006 mais de seis mil vulnerabilidades foram indexadas. A Figura 1.1 ilustra o crescente

[2]O NIST(*National Institute of Standards and Technologies*) é o Instituto Nacional de Padrões e Tecnologias dos Estados Unidos. Maiores informações podem ser obtidas em: <http://www.nist.gov>. Acesso em: 20 de Dez. de 2006.

[3]O *United States - Computer Emergency Readiness Team* (US-CERT) é um grupo estabelecido para coordenar as defesas e respostas à cyber-crimes no território norte-americano. Maiores informações podem ser obtidas em: <http://www.us-cert.gov/>. Acesso em: 20 de Dez. de 2006.

[4]Maiores informações sobre a base de dados NVD podem ser obtidas em <http://nvd.nist.gov/>. Acessado em 20 de Dez. de 2006

número de vulnerabilidades que vêm sendo descobertas e inseridas na NVD.[5]

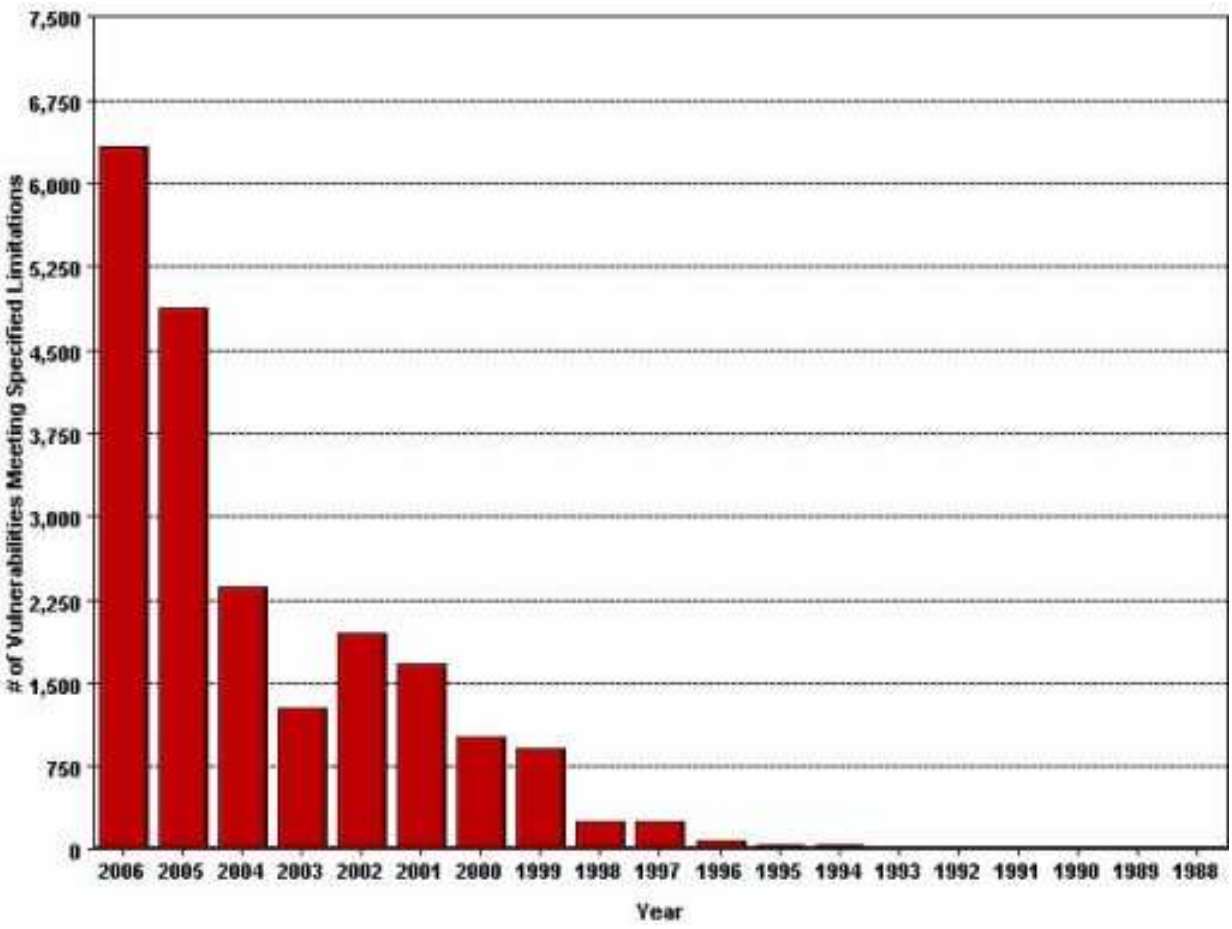

Figura 1.1 - Total de vulnerabilidades indexadas através dos anos no NVD.

Através da observação da Figura 1.1 é possível identificar que nos últimos três anos o número de vulnerabilidades reportadas ao ano está crescendo. Este crescimento pode estar atrelado a vários fatores. Entre eles certamente está o surgimento de novas tecnologias problemáticas, a reincidencia de erros conhecidos por parte dos desenvolvedores e uma política mais agressiva na detecção e correção de falhas adotada pelos próprios mantenedores dos programas.

A NVD também classifica as vulnerabilidades em remotamente exploráveis e localmente exploráveis. Como pode ser observado na Figura 1.2, a porcentagem de vulnerabilidades remotamente exploráveis ainda se mantem superior.[6] Estas vulnerabilidades são as que permitem que um atacante através de um equipamento externo possa ganhar acesso a um

[5] A Figura 1.1 foi gerada a partir do site oficial do NVD em 20 de Dez. de 2006.
[6] A Figura 1.2 foi gerada a partir do site oficial do NVD em 20 de Dez. de 2006.

dado sistema de maneira maliciosa.

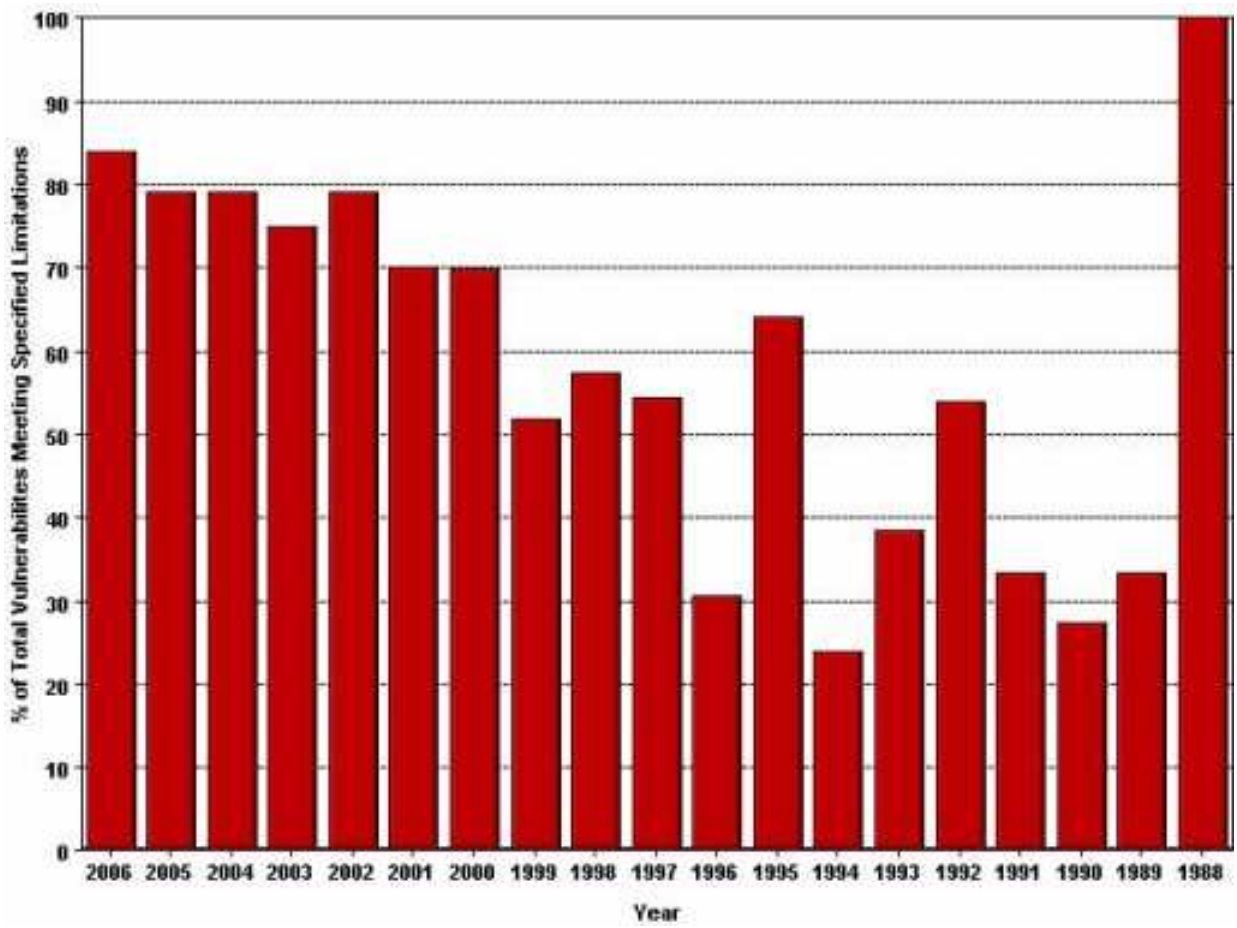

Figura 1.2 - Porcentagem de vulnerabilidades remotamente exploráveis indexadas no NVD.

Pode-se notar na Figura 1.2 que no período de 2000 a 2006 a porcentagem de vulnerabilidades remotamente exploráveis descobertas e indexadas gira em torno de 75%. Este dado pode indicar que o aumento no número total de vulnerabilidades vem mantendo um proporção estável de três vulnerabilidades remotamente exploráveis para cada vulnerabilidade localmente explorável descoberta.

Na Figura 1.3 é ilustrada aporcentagem de vulnerabilidades que são diretamente relacionadas à problemas no processo de validação de entradas. O processo de validação e entradas em um programa é a seqüência de passos que cada um dos módulos do programa deve realizar para garantir que o processamento de um conjunto é seguro. Mais especificamente, são os conjuntos de passos que uma função deve realizar sobre dados não confiáveis para assegurar que a função irá executar sem problemas. Dados não confiáveis

são aqueles dados obtidos de fontes externas ao programa como por exemplo a entrada de dados realizada por um usuário, um arquivo de configuração entre outras.

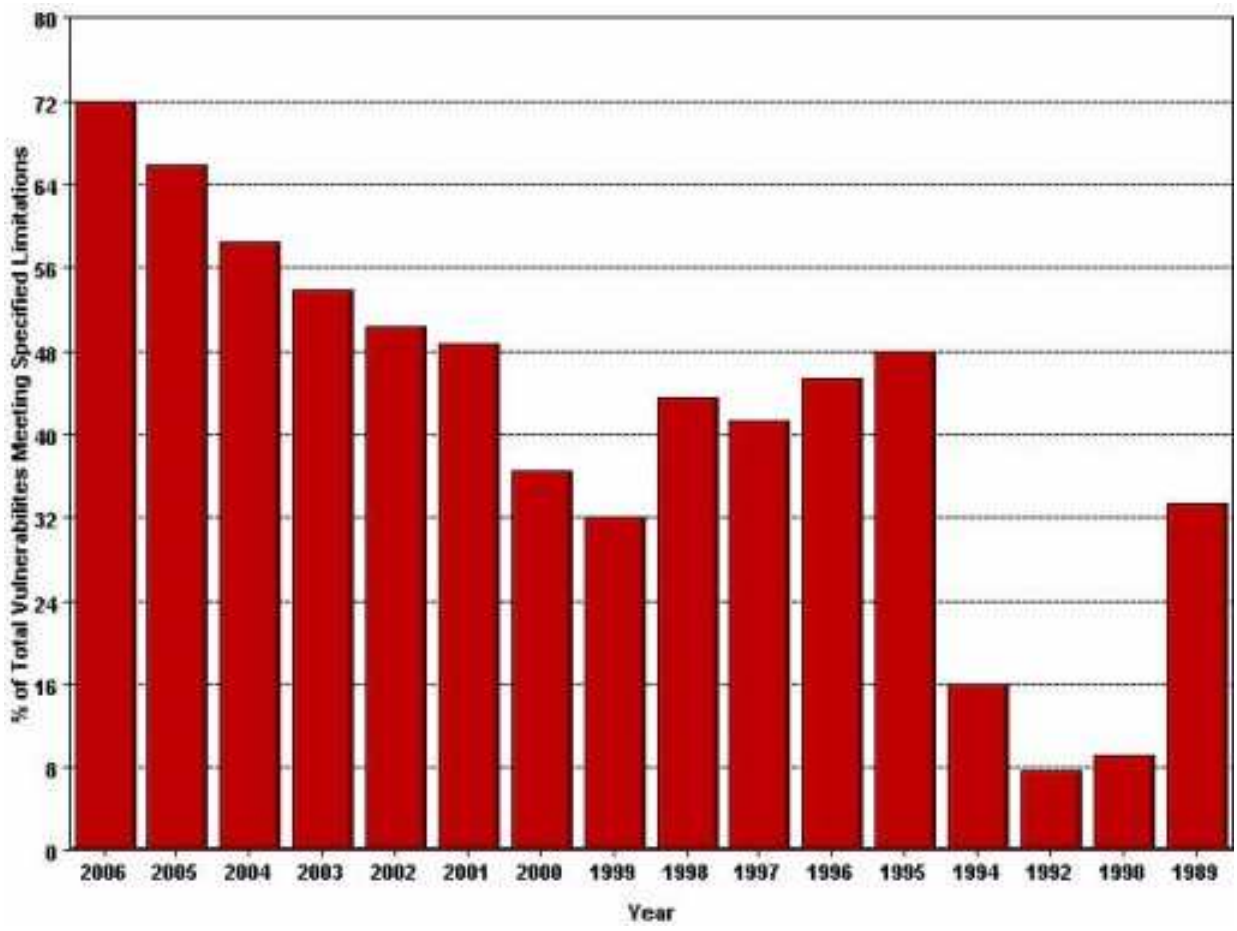

Figura 1.3 - Porcentagem de vulnerabilidades indexadas no NVD relacionadas à problemas de validação de entrada.

Atualmente, existe uma grande gama de ataques que exploram falhas no processos de validação de entradas. Dentre eles destacam-se: ataques de *buffer overflow*, ataques de *SQL injection*, ataques de *code injection*. Cada um destes ataques exploram peculiaridades diferentes presentes em um programa. Posteriormente, no segundo capítulo deste trabalho, estes ataques serão melhor abordados.

Um dado alarmante pode ser observado na Figura 1.3 que mostra um crescimento na porcentagem de vulnerabilidades encontradas em programas que são causadas por falhas no processo de validação de entradas. Até Dezembro de 2006, 72% das vulnerabilidades descobertas estavam relacionadas à problemas de validação de entradas.

Estas vulnerabilidades são diretamente decorrentes de problemas inseridos durante o processo de codificação de um programa.

Este trabalho motivou-se essencialmente pelo conjunto de fatores apresentados anteriormente. Em suma, o aumento de sistemas sendo diariamente conectados à Internet, o aumento no número de programas e funcionalidades neles inseridas. Além disso, o crescente número de vulnerabilidades que são descobertas através dos anos que são remotamente exploráveis e ocasionadas por problemas no processo de validação de entrada de um programa.

1.2 Objetivo e Organização deste Trabalho

O objetivo deste trabalho é desenvolver um ambiente que seja capaz de auxiliar o processo de análise e identificação de possíveis problemas em programas. Em especial vulnerabilidades, que possam permitir que alguma norma da política de segurança da instituição seja violada. Para isso, o trabalho envolve a pesquisa tanto das vulnerabilidades mais comuns, quanto das formas através das quais estas são habitualmente exploradas. Além disso, serão estudados os mecanismos atualmente disponíveis nesta área que procuram minimizar os problemas.

No segundo capítulo deste documento serão descritas as formas de codificação funcional, de erro e segura. As vulnerabilidades freqüentemente encontradas em um código-fonte, como *format string*, *code injection*, serão abordadas, enfatizando a vulnerabilidade *buffer overflow*. Por fim, será mostrado como um *bufffer overlflow* pode ocorrer além de algumas das técnicas mais utilizadas para explorar esta vulnerabilidade.

O terceiro capítulo é reservado para a revisão dos estudos que estão sendo realizados na tentativa de eliminar as vulnerabilidades de *buffer overflow* ou ao menos minimizar o impacto no sistema quando estas são exploradas. Será mostrado que alguns estudos usam técnicas que tentam eliminar os potenciais problemas antes da distribuição do programa em larga escala enquanto outros usam técnicas para que medidas paliativas sejam tomadas quando uma dada vulnerabilidade aparenta estar sendo explorada. Estes estudos serão divididos segundo as abordagens que eles utilizam, que pode ser dependente do compilador, do sistema ou da aplicação. Como o ambiente proposto utilizará uma abordagem dependente da aplicação, as metodologias utilizadas para auditoria de um programa, as ferramentas atualmente disponíveis e comparativos sobre a eficiência das mesmas serão apresentados também neste capítulo.

No quarto capítulo será apresentado o ambiente proposto para realização de análises de códigos-fonte com ênfase em segurança. Serão apresentados os módulos que devem

fazem parte deste ambiente, bem como a interação entre eles. Neste capítulo também será mostrada a implementação de um protótipo para o ambiente proposto chamado de *Source Code Analysis Prototype* (`SCAP`). Os principais módulos deste protótipo terão sua implementação descrita de forma também neste capítulo.

Um conjunto de testes realizados com o `SCAP` será apresentado no quinto capítulo. Informações a cerca dos casos de teste utilizados para a realização dos testes, os resultados obtidos e um comparativo com os resultados de outras ferramentas também serão descritos.

As conclusões sobre a pesquisa desenvolvida, a viabilidade de implementação do ambiente proposto, limitações e vantagens identificadas serão apresentadas no sexto e último capítulo. Neste também serão discutidas prossibilidades de trabalhos futuros.

CAPÍTULO 2

DESCRIÇÃO DO PROBLEMA

A maior parte das vulnerabilidades encontradas em programas está relacionada a maneira como este programa foi codificado. Ou seja, estas vulnerabilidades são inseridas durante o processo de escrita do código-fonte do programa. Logicamente podem existir vulnerabilidades no projeto de um programa ou no ambiente onde o programa irá executar.

Por exemplo, uma falha de projeto em um programa pode expor uma vulnerabilidade com alta criticidade, como no caso da vulnerabilidade `CVE-2006-6722`[1], indexada no NVD. Nesta vulnerabilidade, um atacante pode criar contas administrativas remotamente no sistema `Bandsite` através de requisições ao *script* `admin.php` com o parâmetro `login` definido como 1. A falha no projeto ocorre quando o *script* verifica se o acesso administrativo ao *script* é permitido através do valor do parâmetro `login` que pode ser manipulado por um atacante.

Apesar de existirem vulnerabilidades críticas decorrentes de falhas de projeto, estas não perfazem 1% das vulnerabilidades indexadas no NVD no ano de 2006. Por este motivo os esforços foram focados em vulnerabilidades expostas por falhas durante o processo de codificação. Entretanto, para melhor entender os potenciais problemas durante este processo é importante entender os níveis de codificação existentes.

2.1 Processo de Codificação

Todos os programas computacionais possuem um código-fonte. A linguagem utilizada para gerar este código-fonte pode ser de baixo nível, alto nível ou uma abstração para outra linguagem. Mas, independentemente da linguagem, a etapa de codificação é realizada por programadores. Infelizmente programadores cometem pequenos erros o tempo todo.(Chess e McGraw, 2004)

Na maior parte das vezes, estes erros podem ser encontrados facilmente, pois o compilador os nota. Entretanto, existem erros que não são identificados pelos compiladores e que podem trazer sérias conseqüências.

Os erros estão diretamente ligados à forma com que o sistema foi codificado. Stytz e Whittaker (2003) fazem uma distinção entre as formas de codificação de um programa. Esta distinção pode ser observada a seguir:

[1] Maiores informações sobre a vulnerabilidade com o identificador `CVE-2006-6722` podem ser obtidas em: <`http://nvd.nist.gov/nvf.cfm?cvename=CVE-2006-6722`>. Acessado em 29 de Dez. de 2006.

2.1.1 Tipos de Codificação

- **Código funcional**: Neste tipo de codificação, o programador se preocupa em atingir os requisitos do cliente. Ou seja, faz com que o programa execute da maneira correta se todo o resto do sistema estiver executando da mesma forma. Geralmente este é o primeiro código realizado pelo programador e muitas vezes é utilizado para assegurar que sua lógica está correta.

- **Código de erro**: Neste tipo de codificação, existe a preocupação por parte do programador de que as entradas esperadas, obtidas dos requisitos do cliente, sejam realmente inseridas e que o programa não retorne respostas erradas ou falhe caso uma entrada não esperada seja inserida. Geralmente, neste tipo de codificação são desenvolvidas funções para tratamento de erros e a checagem do valor de retorno das funções das bibliotecas utilizadas.

- **Código seguro**: Neste tipo de codificação, o programador procura garantir que o sistema fará apenas o que ele deve fazer e nada mais. Novos códigos para tratamento de erros e requisitos mais fortes quanto à entrada de dados, troca de informações, abertura de arquivos, manipulações de variáveis entre outros devem ser implementados. Nesta forma de codificação, também é importante que o código consiga gerar informações sobre quais usuários, em qual momento, tentaram quais tipos de acessos, partindo de quais endereços na rede.(DUARTE et al., 2005c)

Um programador possui muita informação disponível para ajudá-lo a escrever bons códigos funcionais. Além disso, a gerência de projeto considera o código funcional o mais importante dos três tipos de codificação, pois vai ao encontro das necessidades do cliente. Estes são alguns dos motivos pelos quais a maior parte dos códigos-fonte desenvolvidos por programadores está na categoria de códigos funcionais.

Além da dificuldade em se escrever um código seguro, identificar se um determinado programa possui uma codificação segura não é uma tarefa simples, ainda mais em sistemas não triviais. Existem muitos tipos de vulnerabilidades que podem ser inseridas inadvertidamente no código-fonte de um prorama. Os tipos de vulnerabilidades comumente encontrados precisaram ser estudados para que um tipo específico pudesse ser abordado.

2.2 Tipos de Vulnerabilidades Comumente Encontradas

Mesmo com toda a preocupação em segurança durante o processo de desenvolvimento de um progrma, alguns problemas podem permanecer ocultos. Segundo Heffley e Meunier (2004) alguns dos problemas típicos na codificação de programas que possuem implicações em segurança são:

- **Format string vulnerability**: *Format string vulnerabilities* ou vulnerabilidades nas cadeias de caracteres de formatação são perigosas, pois essencialmente permitem que usuários modifiquem o programa e como este é executado. *Format strings* são especificações utilizadas pelo programa indicando como as informações devem ser representadas. Funções como as da família `printf` e `syslog` costumam utilizar *format strings*. As *format strings* são cadeias de caracteres que contém caracteres especiais de formatação como '`%s`'. A vulnerabilidade em *format string* ocorre se este estiver sobre o controle do atacante, podendo fazer com que o programa processe informações arbitrárias. Maiores informações de como esta vulnerabilidade ocorre e como explorá-la podem ser encontradas em Scut (2001).

- **Code injection vulnerability**: A vulnerabilidade de *code injection* ou injeção de código ocorre quando entradas não validadas são utilizadas para gerar comandos no sistema que o programa executa, permitindo a execução de código arbitrário. Um exemplo é a criação de chamadas a programas externos utilizando entradas geradas por usuários como parâmetros.

- **SQL injection vulnerability**: A vulnerabilidade de *SQL injection* ou injeção de comandos SQL é muito similar à *code injection*. Entretanto, dados externos são utilizados na geração de comandos SQL para banco de dados. Se um atacante puder forjar uma requisição SQL específica o banco de dados pode ser modificado, novos usuários podem ser criados, o sistema de autenticação pode ser burlado, entre outros.

- **Symbolic link vulnerability**: Um *symbolic link*, *link* simbólico, é um *link* que aponta para outro arquivo e atua como se fosse o arquivo apontado. Uma vulnerabilidade ocorre quando o arquivo apontado pelo *link* ou o próprio *link* simbólico passa para o controle do atacante. Com isso, é possível que dados escritos pelo programa sejam escritos em arquivos indevidos e informações obtidas sejam tomadas, também, de arquivos indevidos. O problema mais freqüente com *links* simbólicos ocorre quando o atacante modifica ou cria um

link para um arquivo que ele não tem permissão de acesso, mas o programa em execução possui esta permissão.

- **Race condition vulnerability**: As *race conditions* ou condições de disputa ocorrem em ambientes que suportam multiprogramação. Este problema acontece quando dois ou mais processos utilizam a mesma variável, mesmo arquivo ou outros recursos simultaneamente. Um recurso pode ser modificado, intencionalmente ou não, por um processo e ser requisitado por um segundo, fazendo que este se comporte de maneira não esperada. Um exemplo deste tipo de problema está relacionado com a utilização de arquivos temporários criados pelos programas. Se um atacante conseguir predizer qual arquivo será criado e utilizado pelo programa, ele poderá tentar forjar as informações contidas neste arquivo, modificando a execução do programa.

- **Buffer overflow vulnerability**: *buffer overflow* ou extravasamento de *buffer* é a vulnerabilidade mais comum presente nos programas. Ela, geralmente, ocorre quando um número de dados maior do que o suportado é escrito em um *buffer*. Esta vulnerabilidade pode ocorrer por falta de cuidados com os índices dos *buffers* em *loops*. Um *buffer overflow* também pode ocorrer quando o caractere `'\0'`[2] de final de *string* não estiver presente no *buffer* ou quando ocorrer algum erro de aritmética de ponteiros ou índices dos *buffers*.

Segundo estudos como o realizado por Hoglund e MacGraw (2004) *buffer overflow* continua sendo a principal vulnerabilidade explorada em um programa. Por este motivo os esforços deste trabalho foram direcionados ao estudo desta vulnerabilidade, como esta pode ser explorada, bem como nas ferramentas atualmente disponíveis que visam eliminar estas vulnerabilidades dos programas.

2.3 Buffer Overflow

Um *buffer* é uma área de memória contiguamente alocada que suporta inúmeras instâncias de um mesmo tipo de dado, como as cadeias de caracteres. Os *buffers* podem ser estáticos ou dinâmicos. Os estáticos são alocados quando o programa é carregado, *loadtime*, já os dinâmicos são alocados em tempo de execução, *runtime*.

O extravasamento de um *buffer* ocorre quando dados são lidos e/ou armazenados fora dos limites alocados para o *buffer*. Além disso, em algumas linguagens como `C` e `C++` não existem checagens automáticas de limites, o que significa que existe a possibilidade de um *buffer* ser extravasado.

[2]Na linguagem C, '\0' é o caractere que identifica o final de uma cadeia de caracteres.

O principal objetivo de um ataque que se aproveite de um *buffer overflow* é a obtenção de acesso ao sistema afetado, fazendo com que ações arbitrárias sejam executadas. Existem várias técnicas de ataque de *buffer overflow* que serão discutidas mais adiante neste trabalho. Segundo Pincus e B. (2004) *buffer overflows* são muito utilizados por vetores de infecção de *worms* ou para ataques direcionados a determinados serviços em determinadas máquinas.

Para que as tentativas de ataque de *buffer overflow* possam ser compreendidas é preciso identificar como um buffer pode ser extravasado. A seguir são mostradas três formas básicas de extravasamento de um *buffer*.

2.3.1 Como *Buffer Overflows* Ocorrem

Existem três formas básicas de um *buffer overflow* ocorrer. A mais comum é ocasionada quando um programa ou processo tenta inserir mais dados em um *buffer* do que este pode armazenar. Esta inserção faz com que informações sejam armazenadas fora dos limites do *buffer*. Com isso, outras variáveis podem ser sobrescritas e podem fazer com que o fluxo de execução do programa seja alterado. Um exemplo de *buffer overflow* que ocorre devida à passagem de um número maior de dados do que o suportado por um *buffer* pode ser observado na Figura 2.1.

```
1  /* ********************************************************************* *
2   * Programa que ilustra a variavel buff sendo extravasada quando mais *
3   * dados do que os suportados lhe sao passados para serem armazenados *
4   * ********************************************************************* */
5  #include <stdio.h>
6  int main(void){
7     char buff[4];
8
9     memset(buff,0,4);
10    strcpy(buff,"0123456789");
11    return(0);
12 }
```

Figura 2.1 - Exemplo de um *buffer overflow* na variável buff.

O programa de exemplo da Figura 2.1 possui em sua função principal, `main`, uma variável estaticamente alocada `buff`, definida na sétima linha do exemplo. Esta variável permite que 4 *bytes* sejam armazenados montando assim uma cadeia de caracteres. A primeira ação da função principal é a chamada da função `memset`. Esta sobrescreve o conteúdo

da variável `buff` por um conjunto de quatro *bytes* "0". A ação seguinte é a chamada da função `strcpy`, na linha 10, que realiza a cópia da cadeia de caracteres "0123456789" para `buff`. Por fim, a função principal retorna na linha 11.

O extravasamento do *buffer* ocorre na décima linha do exemplo da Figura 2.1, pois a função `strcpy` tenta copiar dez *bytes* para um *buffer* que só suporta quatro. Quando esta função é executada dados são escritos fora dos limites da variável `buff` e o *buffer overflow* ocorre.

Outra forma de um *buffer overflow* ocorrer é se um ponteiro para um dado *buffer* apontar para um endereço de memória fora dos limites do *buffer*. Neste caso, o extravasamento ocorre se este endereço apontado for lido ou escrito. Este tipo de *buffer overflow* é ilustrado na Figura 2.2.

```
1  /* ************************************************ *
2   * Programa que ilustra um buffer overflow pela     *
3   * ma manipulacao do endereco apontado por um ponteiro. *
4   * ************************************************ */
5  int main(void){
6     char buff[10];
7     char *p;
8
9     p=buff;
10    p=p+50;
11
12    *p = 'L';
13    return(0);
14 }
```

Figura 2.2 - Exemplo de um *buffer overflow* usando ponteiro.

O código-fonte ilustrado na Figura 2.2 possui uma função principal, `main`. Nesta função existem duas variáveis declaradas: `buff` na linha 6, que é uma cadeia de caracteres que suporta 10 *bytes*; e `p` na linha 7, que é um ponteiro para uma variável do tipo `char`.

Na nona linha do exemplo da Figura 2.2, o ponteiro `p` é apontado para o endereço do primeiro caracter de `buff`. Na décima linha o ponteiro é deslocado 50 *bytes* à frente do primeiro *byte* de `buff`. Na linha 12, a região apontada por `buff` recebe o caracter `'L'`, mudando assim seu conteúdo.

Pode-se notar que, no exemplo da figura 2.2, o espaço alocado para a variável `buff` é de

10 bytes. Quando o ponteiro `p` aponta para `buff` o espaço alocado para região de memória apontada por este é o mesmo. Entretanto, o ponteiro sofre um deslocamento de 50 *bytes* e recebe o caractere `'L'`, fazendo com que o buffer `buff` sofra um *buffer overflow* na linha doze. A capacidade de se modificar o local apontado por ponteiros utilizando aritmética é conhecida como aritmética de ponteiros.

Uma outra situação em que um *buffer overflow* ocorre é através da má manipulação do índice de um *buffer*. Com isso, este índice pode acabar referenciando um local fora dos limites do *buffer*. A posição indicada pelo índice quando lida ou quando dados lhe forem inseridos causa o *buffer overflow*. Um exemplo de *buffer overflow* devido à má manipulação do índice de uma cadeia de caracteres pode ser observado na linha sete do código ilustrado na Figura 2.3.

```
1  /* ************************************************ *
2   * Programa que ilustra um buffer overflow devida a ma *
3   * manipulacao de indices de cadeias de caracteres.    *
4   * ************************************************ */
5  int main(void){
6     char buffer[10];
7     buffer[20] = 'L';
8     return(0);
9  }
```

Figura 2.3 - Exemplo de um *buffer overflow* devido à má manipulação de índices.

Na Figura 2.3 o tamanho alocado para a variável `buffer` é de 10 bytes. Quando a vigésima posição é escrita na sétima linha ocorre o extravasamento de *buffer*. Dependendo de onde a escrita for realizada o programa acaba sendo finalizado. O erro *segmentation fault* geralmente está relacionado com a escrita de posições de memória não percentencentes ao processo. Em outras palavras, caso um dado processo tente escrever em um espaço de memória não alocado para ele, o sistema operacional se encarrega de finalizar sua execução.

Um problema com *buffer overflow* podem ser chamados de *internal buffer overflow*, *buffer overflow* interno, quando este problema ocorre em funções desenvolvidas no próprio projeto do programa. Ou seja, são problemas inseridos pelos desenvolvedores do programa em questão.

2.3.2 Funções Inseguras da `libc`

Um *buffer overflow* também pode ocorrer em algumas funções da biblioteca padrão da linguagem `C`, `libc`. Estas funções problemáticas não fazem as devidas checagens dos argumentos que lhe são passados e são ditas funções inseguras. Um exemplo deste tipo de função é a função `strcpy` que copia o conteúdo de uma cadeia de caracteres em uma outra cadeia. Entretanto, não checa se a cadeia de destino suporta o número de caracteres que a cadeia de origem possui. Um exemplo de quando este problema pode ocorrer pode ser observado na Figura 2.4.

```
1  /* ************************************************** *
2   * Programa que ilustra um buffer overflow devido a um  *
3   * problema em uma implementacao de uma funcao da       *
4   * biblioteca padrao da linguagem C, glibc.             *
5   * ************************************************** */
6  int main(void){
7          char buff_orig[] = "1234567890123456789";
8          char buff_dest[10];
9
10         strcpy(buff_dest,buff_orig);
11         return(0);
12 }
```

Figura 2.4 - Função `strcpy` sem checagem dos argumentos antes da chamada.

A função principal `main` do código-fonte da Figura 2.4 possui duas variáveis `buff_orig` e `buff_dest` respectivamente com 20 e 10 *bytes*. A função `strcpy` recebe como parâmetro um *buffer* de destino, no qual os dados serão gravados, e um *buffer* de origem, do qual os dados serão obtidos. A variável `buff_dest` acaba sendo extravasada ao passar pela função `strcpy`, que não checa se a variável `buff_orig` pode ser copiada para `buff_dest`.

Alguns autores, como em Viega e MacGraw (2001) sugerem que as funções inseguras não sejam utilizadas ou utilizadas com a devida cautela. Para que estas funções sejam utilizadas da maneira correta é necessário analisar os parâmetros de entrada para permitir ou não que a função seja executada. O exemplo da Figura 2.5 mostra como uma checagem pode ser realizada, permitindo a utilização segura da função `strcpy`.

Na Figura 2.5, uma checagem é necessária para verificar se o *buffer* `buff_dest` de destino suporta a quantidade de dados presentes no *buffer* de origem `buff_orig`. Para realizar esta checagem, a função `strnlen` pode ser utilizada. No exemplo a função `strlen` retorna

```
/* ****************************************************** *
 * Programa que ilustra uma forma de se utilizar a funcao *
 * strcpy(); de maneira segura, atraves da checagem do    *
 * tamanho do buffer de destino antes da chamada.         *
 * ****************************************************** */
#define BUFFER_DEST_SIZE 10

int main(void){
   char buff_orig[] = "12345678901234567890123456789O";
   char buff_dest[BUFFER_DEST_SIZE];


   if (strnlen(buff_orig,BUFFER_DEST_SIZE) < BUFFER_DEST_SIZE) {
      strcpy(buff_dest,buff_orig);
   }
   else {
      /* Um extravasamento do buffer buff_dest teria ocorrido */
      printf("!- ERRO !");
   }
   return(0);
}
```

Figura 2.5 - Função `strcpy` com checagem de argumentos antes da chamada.

o número de caracteres atualmente armazenados na variável `buff_orig`.

A função `strnlen` retorna o número de *bytes* utilizados sem contar o caractere '`\0`'. Para que a função `strcpy` possa ser executada o número de *bytes* alocados para a cadeia de destino deve ser maior ou igual ao valor retornado pela função `strnlen` mais um. Ou seja, o `strnlen` da cadeia de origem deve ser menor do que o número de *bytes* alocados para a cadeia de destino.

Viega e MacGraw (2001) listam, em uma tabela não exaustiva, funções que devem ser utilizadas com cautela ou simplesmente evitadas. Estes dados estão sumarizados na Tabela 2.1.

Algumas funções presentes na Tabela 2.1 não possuem formas de utilização segura, como é o caso da função `gets`. Esta função obtém dados da entrada padrão `stdin` e os armazena em um `buffer` passado como parâmetro. Infelizmente, não é possível definir *a priori* qual a quantidade de *bytes* que serão lidos, impossibilitando definir um tamanho mínimo aceitavel para o `buffer` de destino.

Algumas funções como a `strncpy` podem facilmente serem utilizadas de forma segura,

Tabela 2.1 - Sumário de funções inseguras.

Função	Função	Função
[f]gets();	strcpy();	strcat();
strecpy();	[v][vs][vf][f][s]scanf();	bcopy();
streadd();	[vsn][sn][vs][s]printf();	strtrns();
realpath();	syslog();	getopt();
getopt_long();	getpass();	getchar();
fgetc();	getc();	read();
strccpy();	strcadd();	strncpy();
memcpy();		

bastando a utilização correta dos parâmetros que lhes são passados. Em alguns casos como o da função `scanf` é importante saber como o parâmetro que representa a cadeia de formação, *format string*, foi definido para saber se a utilização é ou não segura.

A possibilidade de extravasamento de um *buffer* expõe uma fragilidade da arquitetura dos sistemas, que é a capacidade de modificar valores contidos em regiões de memória não disponíveis para uma determinada variável. Para entender como esta fragilidade pode ser utilizada por atacantes para obtenção de vantagens, foi realizado um estudo de como um *buffer overflow* pode ser explorado.

2.4 Como *Buffer Overflows* são Explorados

Um *buffer overflow* pode ser caracterizado ou como um *stack buffer overflow* ou como um *heap buffer overflow* dependendo da região de memória afetada. Os compiladores das linguagens `C` e `C++` tipicamente utilizam a *stack* para variáveis locais, parâmetros de funções, *frame pointers* assim como para armazenar endereços de retorno.

Heaps são utilizadas para armazenar implementações dinâmicas de memória como no caso das funções `malloc()` e `free()` da biblioteca padrão da linguagem `C` e das funções `new()` e `delete()` da linguagem `C++`. Tanto implementações dinâmicas de memória como variáveis locais são armazenadas dentro do espaço de endereçamento do processo, *process address space*.

2.4.1 Espaço de Endereçamento de um Processo

O espaço de endereçamento de um processo no sistema `Linux` pode ser observado na Figura 2.6.

Quando um novo processo é colocado em execução, uma região de memória lhe é alocada.

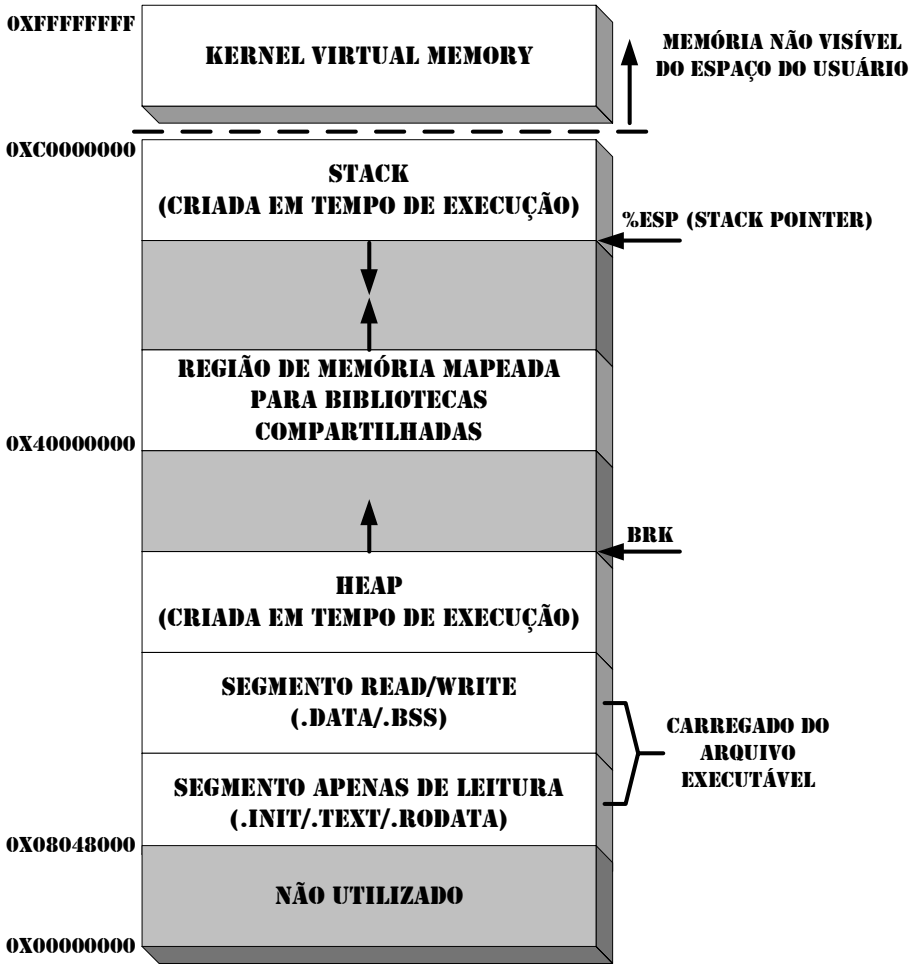

Figura 2.6 - Espaço de endereçamento de um processo no Linux.

Uma estrutura bem definia é inserida nesta região, que passa a ser chamada de espaço de endereçamento do processo. As duas estruturas mais estritamente relacionadas com os problemas de *buffer overflow* são a `stack` e a `heap`. Estas são implementações de duas pilhas com características e objetivos diferentes.

Além da `stack` e da `heap` existem dois ponteiros `brk` e `stack pointer`. O ponteiro `brk` é utilizado para informações sobre a quantidade de memória dinamicamente alocada na `heap`. Já o ponteiro `stack pointer` é utilizado para apontar para o topo da `stack` ou para a próxima posição livre na `stack` dependendo da implementação.

2.4.2 Organização de uma *Stack*

A `stack` é utilizada para armazenar variáveis locais estaticamente declaradas e que são utilizadas nas inúmeras funções de um programa. Os parâmetros que são passados para cada uma das funções também são armazenados bem como o *return address* ou endereço de retorno de cada uma delas.

O objetivo de uma *stack* é armazenar inúmeros *stack frames*. Em um *stack frame* são guardadas todas as informações necessárias para que uma função possa executar de forma correta. Ou seja, são armazenadas as variáveis locais estaticamente decladaras, os parâmetros passados, o próximo endereço que deve ser executado após o retorno da função, entre outras. A estrutura básica de um *stack frame* em um sistema `Linux` pode ser observada na Figura 2.7.

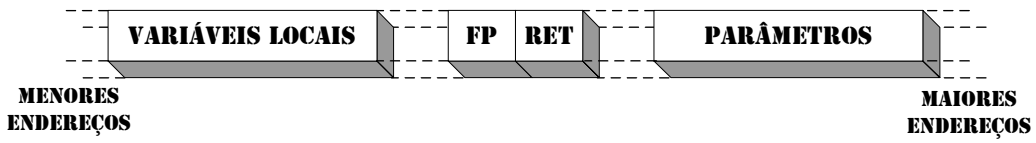

Figura 2.7 - Estrutura de um *stack frame*.

Um *stack frame* é inserido quando uma função é chamada e é retirado quando a função retorna. Como pode ser observado na Figura 2.7, para cada *stack frame* existe um *frame pointer* (`FP`), também conhecido como *local base pointer* (`LBP`).

O *frame pointer* é utilizado para referência de variáveis locais e parâmetros na *stack*, pois a distância entre o *frame pointer* e as variáveis e parâmetros não se altera durante a execução.

Outro dado armazenado em um *stack frame* é o endereço de retorno, (`RET`), ou *return address* que é o endereço que deve ser executado depois do epílogo da função. O valor definido no endereço de retorno de uma função é estabelecido no prólogo da função. Ou seja, é um valor atribuído antes da função ser propriamente executada. Além disso, o valor presente nesta variável não deve ser alterado durante a execução da função.

A forma como os dados são preenchidos em um *stack frame* é uma informação importante sobre o comportamento dos elementos de um *stack frame*. Quando um *stack frame* é criado, os parâmetros da função são os primeiros a serem inseridos, sendo que o último parâmetro é alocado no maior endereço. A inserção dos parâmetros é realizada da esquerda

para a direita no exemplo da Figura 2.7. O endereço de retorno (`RET`) e o *frame pointer* (`FP`) são inseridos entre os parâmetros e as variáveis locais estaticamente alocadas. As variáveis locais são inseridas na ordem em que são declaradas dos maiores endereços para os menores sendo que a primeira variável declarada em uma dada função é a que fica mais próxima do *retun address*.

O preenchimento de uma variável local é realizado dos menores endereços para os maiores. Como pode ser observado na Figura 2.7, caso uma variável local seja extravasada, dados podem sobrescrever os demais valores definidos para o *frame pointer*, *return address* ou parâmetros.

Na Figura 2.8 um exemplo de programa escrito na linguagem `C` é utilizado para ilustrar a estrutura de *stack frame* de uma função da linguagem `C`. Este programa possui uma função chamada `function`. Esta requer como parâmetros de entrada três inteiros `a`, `b` e `c`. Duas variáveis locais estaticamente alocadas, *buffer1* e *buffer2* também são utilizadas na função `function`.

```
1  /* ************************************ *
2   * Programa para exemplo do preenchimento *
3   * de um stack frame.                     *
4   * ************************************ */
5  void function (int a, int b, int c ) {
6     char buffer1[5];
7     char buffer2[10];
8  }
9
10 int main(int argc, char *argv[]) {
11    function(1,2,3);
12    return 0;
13 }
```

Figura 2.8 - Exemplo de um programa em linguagem C.

Durante a execução deste programa, um *stack frame* deve ser criado e inserido na *stack* para a função `function`. A estrutura deste *stack frame* pode ser observada na Figura 2.9. Apesar da *stack* crescer do maior endereço para o menor, o preenchimento, como visto anteriormente, de um *buffer* é realizado do menor endereço para o maior. O *buffer* `buffer2`, por exemplo, se extravasado pode fazer com que o valor de `buffer1` seja alterado. Entretanto, algo pior ocorre se `RET` for sobrescrito, pois o fluxo de execução do programa pode ser modificado, visto que o endereço contido no `RET` é o próximo a ser executado

depois que a função termina.

Figura 2.9 - Estrutura do *stack frame* criado para a função `function`.

2.4.3 Stack Smash

A primeira forma amplamente divulgada de se explorar um *buffer overflow* para conseguir gerar um ataque contra um sistema é conhecida como *stack smash* e foi publicada por Aleph One. (One, 1996). A idéia por trás deste tipo de exploração é modificar o endereço de retorno armazenado na *stack* e fazer com que este aponte para um código injetado pelo atacante. (Pincus e B., 2004)

Para explorar um *buffer* utilizando a técnica de *stack smash* o atacante deve alimentar o sistema com um código executável que deve ser armazenado no espaço de endereçamento do processo além de sobrescrever o endereço de retorno na *stack* para que este aponte para a posição onde o código inserido está localizado.

Originalmente, o código executável utilizado na exploração de vulnerabilidades de *buffer overflow* permitia ao atacante um acesso à uma `shell` no sistema comprometido. Por este motivo este código é chamado de *shellcode*. Atualmente as *shellcodes* permitem ao atacante obter um `shell` direto, `shell` reverso, envio e execução de arquivos, envio de comando para baixar e executar arquivos entre outros.

O envio do código executável e a sobrescrita do endereço de retorno podem ocorrer no mesmo *buffer* ou em *buffers* separados. Uma situação muito útil para a separação do código executável da sobrescrição do endereço de retorno é quando o *buffer* é muito pequeno para suportar o código executável. Portanto, o atacante precisa utilizar algum outro *buffer*, mesmo que este não possa ser extravasado, para armazenar seu código executável e explorar um *buffer overflow* para sobrescrever o endereço de retorno. Um exemplo de como um *stack smash* pode ocorrer pode ser observado através das modificações no *stack frame* da função afetada. Na Figura 2.10 é mostrado como um ataque de *stack smash* pode modificar o *stack frame* da função `function` gerado do código-fonte da Figura 2.8

Caso a variável *buffer1* pudesse ser extravasada, seria possível inserir dados executáveis

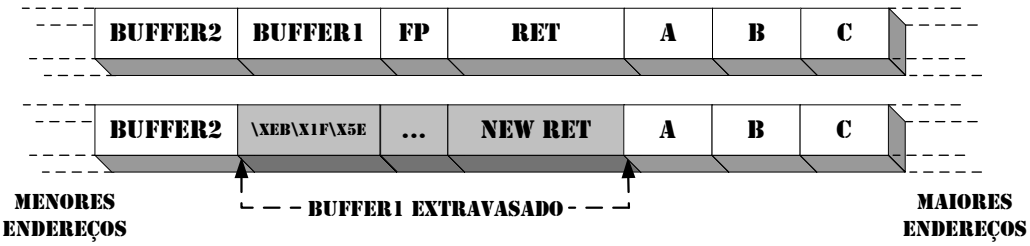

Figura 2.10 - Estrutura do *stack frame* criado para a função `function` e as modificações decorrentes de um ataque de *stack smash*.

nesta variável, sobrescrever o *frame pointer* e o endereço de retorno, fazendo com que este último aponte para o endereço de memória da variável *buffer1*. Com isso, quando a função retornar, o endereço de *buffer1* será executado, fazendo com que a *shellcode* seja executada.

Stack smash não é a única forma de explorar uma vulnerabilidade de *buffer overflow*. Existem outras formas de explorar esta vulnerabilidade, tais como *Arc Injection* e *Pointer Subterfuge*.(Pincus e B., 2004)

2.4.4 Arc Injection

Um atacante pode explorar um *buffer overflow* inserindo dados não executáveis em um *buffer* de tal forma que alguma função já existente no sistema irá utilizar estes dados como parâmetro de entrada.

Neste tipo de ataque, por vezes referenciado como *return-into-libc*, um *buffer overflow* pode ser utilizado para modificar o endereço de retorno para um lugar dentro do espaço de endereçamento do processo. Este endereço pode conter, por exemplo, a função `system()` da biblioteca padrão da linguagem `C`.

Neste tipo de exploração, o atacante insere simplesmente uma nova aresta no gráfico do fluxo de execução do programa, ao contrário do ataque de *stack smash* que deve inserir um novo nó executável. (Pincus e B., 2004)

Este tipo de exploração é muito utilizado quando o programa sendo atacado possui alguma forma de proteção da memória. Por exemplo, algumas implementações de *stack* não executável[3]. Como o atacante nunca insere dados executáveis, estas soluções não previnem ataques do tipo *arc injection*.

Caso existisse uma vulnerabilidade no programa ilustrado na Figura 2.8 que permitisse que o `buffer1` fosse extravasado, seria possível um ataque de *arc injection*. Um exemplo

[3] *Stack* não executável é um mecanismo de segurança que não permite que regiões de memória que possam ser escritas pelo processo possam ser executadas.

de como o *stack frame* é organizado antes do ataque e como este deve permanecer depois do ataque pode ser observado na Figura 2.11.

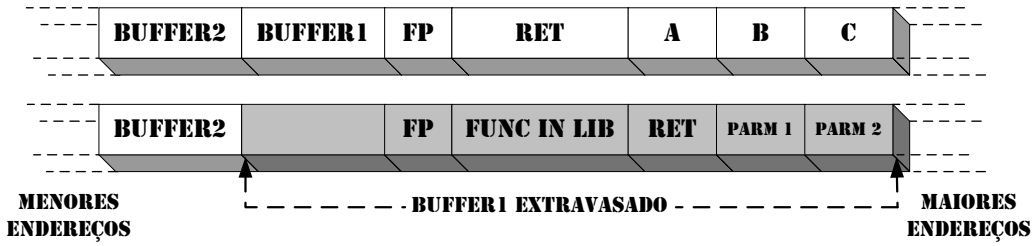

Figura 2.11 - Estrutura do *stack frame* criado para a função `function` e as modificações criadas por um ataque de *arc injection*.

No exemplo da Figura 2.11, se um atacante extravasasse a variável `buffer1`, o valor do `FP`, o endereço de retorno e os dados armazenados nos argumentos de entrada da função `function` poderiam ser modificados. Em um ataque de *arc injection*, o endereço de retorno é apontado para um endereço ocupado por uma função do sistema. Quando esta função é executada, os dados armazenados na variável `a` são interpretados como sendo o local para onde a função deverá retornar e as variáveis `b` e `c` são tratadas como parâmetros para a função. Portanto, se o endereço de retorno apontar para a função `system` e o parâmetro `b` contiver o valor `/bin/sh` um ataque de *arc injection* pode ter sucesso.

2.4.5 Pointer Subterfuge

Ataques de *pointer subterfuge* envolvem modificar o valor de um ponteiro de tal forma que este aponte para um lugar arbitrário na memória. Existem pelo menos quatro tipos de *pointer subterfuge*: *function-pointer clobbering*, *data-pointer modification*, *exception-handler hijacking* e *virtual pointer smashing*. Pincus e B. (2004) discute cada um destes tipos de exploração.

Dentre estes tipos de ataque, o *data-pointer modification* permite modificar o endereço que um ponteiro aponta dentro de espaço de endereçamento do processo. A informação relacionada ao endereço apontado por um ponteiro está presente na `stack` e pode ser modificado caso ocorra um *buffer overflow* de uma variável declarada após o ponteiro. Na Figura 2.12 é apresentado um código-fonte que é vulnerável a um ataque de *data-pointer modification*.

O código-fonte da Figura 2.12 apresenta duas funções. Uma é a função principal `main` e a outra é uma função vulnerável `pointerfnc`. Estas funções são declaradas respectivamente nas linhas 24 e 6 do código-fonte.

```
/* ********************************** *
 *  Programa que ilustra uma funcao   *
 *   vulneravel a pointer subterfuge  *
 * ********************************** */

void pointerfnc(char *str, int size) {
  int *pointer;
  char buff[8];
  int a = 10;
  int b = 20;

  pointer=&a;

  printf("Valores: a(%x)=%d b(%x)=%d pointer(%x)=%d\n",\
                              &a,a,&b,b,pointer,*pointer);
  memcpy(buff,str,size);

  *pointer=30;

  printf("Valores: a(%x)=%d b(%x)=%d pointer(%x)=%d\n",\
                               &a,a,&b,b,pointer,*pointer);
}

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

Figura 2.12 - Código-fonte de um programa vulnerável a *data-pointer modification* na função `pointerfnc`.

O objetivo da função principal é simplesmente obter um argumento da chamada do programa, definir seu tamanho em *bytes*, através da função `strlen` e enviar estes dados para a função `pointerfnc`. A função `pointerfnc` por sua vez aguarda um parâmetro `str` que é uma cadeia de caracteres, e um parâmetro `size` que é um inteiro correspondente ao tamanho em *bytes* de `str`.

A função `pointerfnc` possui quatro variáveis declaradas: um ponteiro para inteiros `pointer`, uma cadeia de caracteres `buff` de oito posições, um inteiro `a` e um inteiro `b`. O inteiro `a` é iniciado com o valor 10 e o inteiro `b` é iniciado com o valor 20. Quando chamada, esta função faz com que o ponteiro `pointer` aponte para o endereço de memória de `a`, ou seja, modificações no conteúdo do endereço apontado por `pointer` também modificam o valor de `a`.

O passo seguinte na execução da função `pointerfnc` é a cópia da cadeia de caracteres `str`

para a cadeia de caracteres `buff`, que ocorre na linha 16. Por fim, o endereço de memória apontado pelo ponteiro `pointer` tem seu conteúdo modificado para o valor 30.

Para verificar as modificações realizadas nos valores das variáveis `a` e `b`, bem como obter o conteúdo da região de memória apontada por `pointer` a função `printf` é chamada duas vezes, uma antes da modificação do valor apontado por `pointer` e outra depois. Um exemplo de como uma utilização não intrusiva do programa `pointersub` gerado do código-fonte da Figura 2.12 poderia ser realizada pode ser observada na Figura 2.13. Nesta figura, o programa é chamado com parâmetros de entrada que podem ser armazenados dentro da variável `buff` e possuem as saídas esperadas. Ou seja, são mostrados os endereços e os valores para `a`,`b` e `pointer`. Note que `pointer` está relacionado com a variável `a`, possuindo o mesmo valor e endereço.

```
1 theuser2@onion:~$ ./pointersub abcd
2 Valores: a(bfe72c54)=10 b(bfe72c50)=20 pointer(bfe72c54)=10
3 Valores: a(bfe72c54)=30 b(bfe72c50)=20 pointer(bfe72c54)=30
4 theuser2@onion:~$ ./pointersub abcd
5 Valores: a(bfdd9b84)=10 b(bfdd9b80)=20 pointer(bfdd9b84)=10
6 Valores: a(bfdd9b84)=30 b(bfdd9b80)=20 pointer(bfdd9b84)=30
7 theuser2@onion:~$ ./pointersub abcd
8 Valores: a(bfacd074)=10 b(bfacd070)=20 pointer(bfacd074)=10
9 Valores: a(bfacd074)=30 b(bfacd070)=20 pointer(bfacd074)=30
```

Figura 2.13 - Exemplo de execução do programa pointersub com parâmetros não intrusivos em um sistema GNU/Linux de Kernel 2.6.

Pode-se notar que a execução do programa `pointersub` em um sistema GNU/Linux com Kernel 2.6 faz uma randomização do endereço em que o *stack frame* deve ser inserido, como pode ser observado na Figura 2.13. Entretanto, mesmo com esta medida é possível que um ataque de *data-pointer modification* ocorra.

A Figura 2.14 ilustra o *stack frame* criado para a função *pointerfnc* e as alterações que são realizadas neste *stack frame* durante a execução da função. O *stack frame* original, observado na Figura 2.14(a) mostra que se a variável `buff` puder ser extravasada ela pode sobrescrever, por exemplo, o endereço apontado por `pointer`, o conteúdo do *frame pointer* e o conteúdo do endereço de retorno.

Entretanto, o extravasamento do *buffer* `buff` não pode sobrescrever o valor das variáveis `a` e `b`, mas pode fazer com que o ponteiro `pointer` aponte para um local arbitrário dentro do *stack frame*. É justamente o que ocorre na Figura 2.14(b). Por fim, quando o valor 30

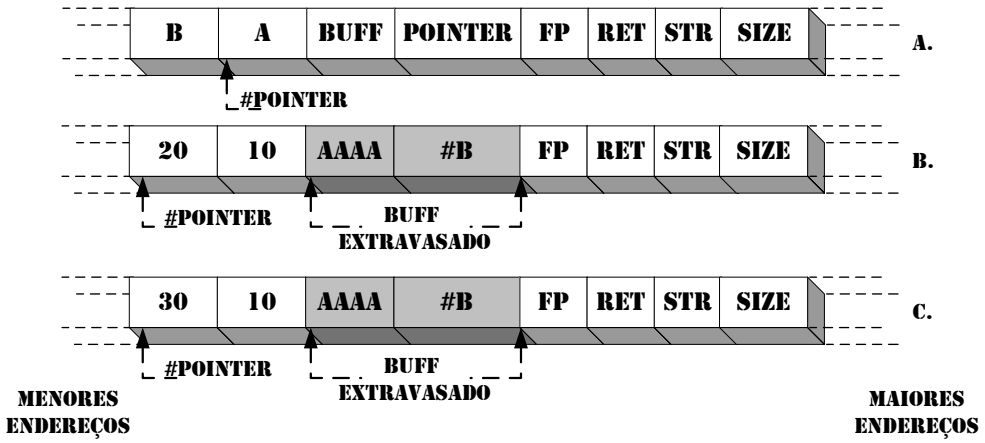

Figura 2.14 - Estrutura do *stack frame* criado para a função `pointerfnc` e as modificações criadas por um ataque de *pointer subterfuge*.

é inserido no conteúdo da posição de memória apontada por `pointer`, o valor da variável `b` é modificado, ao invés do valor da variável `a`. A composição do *stack frame* após a atribuição do valor 30 pode ser observado na Figura 2.14(c).

Para comprovar que a um ataque de *data-pointer modification* pode ocorrer em um sistema GNU/Linux com Kernel 2.6, observe a Figura 2.15. Analisando a randomização dos endereços de memória, é possível notar que a diferença entre os valores de memória da variável `a` e da variável `b` são mínimas, sendo necessário apenas modificar o último hexadecimal do endereço de memória.

```
1  theuser2@onion:~$ ./pointersub aaaaaaaaaaaa
2  Valores: a(bf87fe14)=10 b(bf87fe10)=20 pointer(bf87fe14)=10
3  Valores: a(bf87fe14)=30 b(bf87fe10)=20 pointer(bf87fe14)=30
4  theuser2@onion:~$ ./pointersub aaaaaaaaaaaa
5  Valores: a(bf9a2f44)=10 b(bf9a2f40)=20 pointer(bf9a2f44)=10
6  Valores: a(bf9a2f44)=10 b(bf9a2f40)=20 pointer(bf9a2f61)=30
7  theuser2@onion:~$ ./pointersub aaaaaaaaaaaa$(perl -e '{ print("\x40"); }')
8  Valores: a(bfd96b34)=10 b(bfd96b30)=20 pointer(bfd96b34)=10
9  Valores: a(bfd96b34)=10 b(bfd96b30)=20 pointer(bfd96b40)=30
10 theuser2@onion:~$ ./pointersub aaaaaaaaaaaa$(perl -e '{ print("\x40"); }')
11 Valores: a(bff5c4f4)=10 b(bff5c4f0)=20 pointer(bff5c4f4)=10
12 Valores: a(bff5c4f4)=10 b(bff5c4f0)=20 pointer(bff5c440)=30
13 theuser2@onion:~$ ./pointersub aaaaaaaaaaaa$(perl -e '{ print("\x40"); }')
14 Valores: a(bf8a5e44)=10 b(bf8a5e40)=20 pointer(bf8a5e44)=10
15 Valores: a(bf8a5e44)=10 b(bf8a5e40)=30 pointer(bf8a5e40)=30
16
```

Figura 2.15 - Exemplo de exploração do programa pointersub com parâmetros intrusivos em um sistema GNU/Linux de Kernel 2.6.

Como primeiro passo, é identificado a partir de qual tamanho de entrada o endereço apontado por `pointer` começa a ser modificado. Isto é observado na linha 6 da Figura 2.15, quando o último o octeto recebe o valor 61, a representação hexadecimal para a letra ‘a‘. O passo seguinte é modificar este último octeto para que este seja como o último octeto da variável `b`. Como a alocação dos endereços é dinâmica este octeto precisa ser inferido, sendo que o último quarteto deve ser o hexadecimal ‘0‘, que se mantém constante durante as execuções. No exemplo, foi mantido o octeto da última execução ‘0x40‘. A chamada do programa com este parâmetro pode ser observada na linha 7 do exemplo.

Durante as execuções sucessivas, existe a possibilidade de que o último octeto da posição de memória de `b` seja novamente ‘0x40‘, o que ocorre na execução da linha 13. Neste caso, é possível modificar o valor contido na variável `b` como mostrado na linha 15 da Figura 2.15

2.5 Considerações Finais

Os problemas com vulnerabilidades de *buffer overflow* não são novos e também não são poucas as formas de explorar estas vulnerabilidades. Como visto, existem problemas que podem afetar o funcionamento de um programa em diferentes aspectos, permitindo que um atacante faça com que o programa execute de uma forma maliciosa.

Muitos trabalhos vêm sendo realizados com o objetivo de eliminar estas vulnerabilidades dos sistemas computacionais. Algumas abordagens são mais eficientes do ponto de vista do sistema operacional, outras do ponto de vista do programa e outras levam em consideração a forma como o programa foi compilado. No capítulo seguinte, serão discutidas as técnicas atualmente disponíveis que abordam problemas de *buffer overflow*.

CAPÍTULO 3

REVISÃO DO ESTADO DA ARTE

Segundo Cowan (2003), o programa perfeito, sem falhas, é infactível para sistemas não triviais. Portanto, é preciso encontrar outros meios para se assegurar que programas grandes e complexos façam o que devem fazer e nada mais.

São três as principais abordagens que tentam garantir que um programa faça somente o necessário (DUARTE et al., 2005a). São elas:

- Abordagens dependentes do compilador;
- Abordagens dependentes do sistema;
- Abordagens dependentes da aplicação.

3.1 Abordagens para o Tratamento de *Buffer Overflows*

3.1.1 Abordagens Dependentes do Compilador

Esta abordagem insere algumas informações adicionais, que podem ser pequenas rotinas de checagem, ao programa em tempo de compilação. Isto, para que vulnerabilidades possam ser encontradas em tempo de execução.

Geralmente as ferramentas que utilizam esta técnica são integradas ao compilador do sistema, por isso são chamadas de dependentes do compilador. Com isso, os programas podem ser compilados normalmente e ao final do processo estarem protegidos.

Esta abordagem não necessita de uma grande interação com os desenvolvedores do programa. Portanto, se mostra extremamente prática quando utilizada para sanar vulnerabilidades em programas muito extensos, com milhões de linhas de código.

A tabela 3.1 sumariza quatro das ferramentas mais conhecidas que utilizam entradas em tempo de compilação para sanar vulnerabilidades em tempo de execução. A ferramenta `FormatGuard` trata somente problemas de *format string*, portanto não será abordada em detalhes neste trabalho. Maiores informações sobre o funcionamento da ferramenta `FormatGuard` podem ser encontradas em Cowan et al. (2001a). A seguir serão discutidas as ferramentas que procuram sanar problemas decorrentes de *buffer overflows*.

Tabela 3.1 - Ferramentas dependentes do compilador.

Ferramenta	Efeito	Licença
StackGuard	O programa é finalizado ao ocorrer uma tentativa de ataque de *buffer overflow*	GPL
ProPolice	O programa é finalizado ao ocorrer uma tentativa de ataque de *buffer overflow*	GPL
StackShield	O programa é finalizado ao ocorrer uma tentativa de ataque de *buffer overflow*	GPL
FormatGuard	O programa é finalizado ao ocorrer uma tentativa de ataque de *format string* na função `printf`	GPL

3.1.1.1 StackGuard

O `StackGuard`(Cowan et al., 1998) é uma extensão do compilador `GCC`(GCC, 1987), *GNU Compiler Collection*, que modifica o código executável produzido por este, detectando e prevenindo ataques que exploram *buffer overflows* contra a *stack*.

Ataques como *stack smash* precisam modificar o endereço de retorno de funções. A característica principal do `StackGuard` é justamente não permitir que modificações nos endereços de retorno de funções em programas escritos em linguagem `C` ocorram. Inibindo desta forma os ataques de *stack smash*. (Cowan et al., 1998)

Quando uma função é chamada, um *canary* randômico associado a esta função é inserido entre as variáveis da *stack* e o endereço de retorno. Um *canary* aparenta-se a uma assinatura que não deve ser modificada durante a execução de uma função. Caso isso ocorra, um problema de *buffer overflow* pode ter ocorrido. Este *canary* é inserido na etapa de precompilação, sendo a primeira variável declarada de cada uma das funções. Desta forma, um ataque de *stack smash* em qualquer outra variável da função precisaria sobrescrever o valor do *canary* para conseguir alcançar o endereço de retorno.

Caso uma ataque de *stack smash* ocorra, o `StackGuard` identifica o corrompimento do *canary* e o endereço de retorno passa a não ser mais confiável. Originalmente o *canary* era um valor randomicamente definido no prólogo da função e checado no epílogo da mesma. Um exemplo de um *stack frame* com o uso de um *canary* `CRY` pode ser observado na Figura 3.1.

Canaries randômicos que possuam o caracter "`\0`" também são utilizados para previnir que os *canaries* sejam forjados. Um atacante não será capaz de inserir este símbolo, pois a função que obtém os dados irá terminar ao encontrar este caracter. Em outras

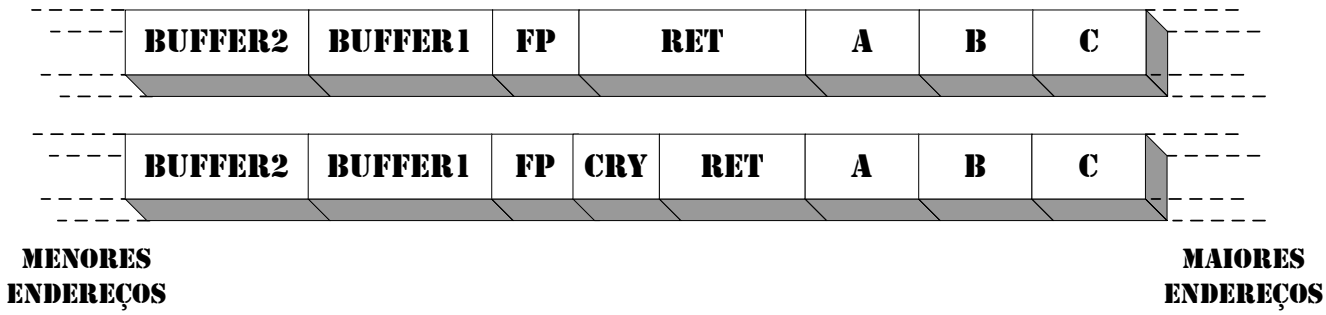

Figura 3.1 - Exemplo de *stack frames* sem e com o uso de *canary*.

palavras, o atacante não conseguirá sobrescrever outras regiões de memória além do *canary* se o caracter "\0" for inserido. Portanto, não conseguirá modificar o endereço de retorno.(Cowan et al., 2000)

Novos ataques podem tentar acessar diretamente o endereço de retorno, sem que seja necessária a modificação do *canary*. Por isso, atualmente o *canary* é XOR randômico. Ou seja, é o resultado da operação XOR do endereço de retorno original com o *canary* randômico inicialmente gerado. Portanto, mesmo que os *canaries* não sejam modificados, as alterações nos endereços de retorno podem ser identificadas.

3.1.1.2 ProPolice

O `ProPolice`(Etoh, 2001) é uma outra implementação similar ao `StackGuard`. Esta ferramenta também utiliza *canaries*, chamados no projeto de guardiões, para fazer a proteção do endereço de retorno de funções em programas escritos em linguagem `C`.

Atualmente, o `ProPolice` reordena as variáveis locais da *stack* com o objetivo de manter a declaração de todos os ponteiros da funcão depois da declaração dos *buffers* para evitar problemas de corrupção de locais arbitrários na memória e ataques de *pointer subterfuge*. Entretanto, nem sempre isso é possível. Por exemplo, em casos onde cadeias de caracteres e ponteiros fazem parte de alguma estrutura.

3.1.1.3 StackShield

A abordagem utilizada no `StackShield`(Vendicator, 2000) é diferente das abordagens tomadas pelo `StackGuard` e `ProPolice`. Ao invés de utilizar *canaries* para garantir integridade do endereço de retorno, ele simplesmente armazena o valor do endereço de retorno antes de fazer a chamada da função em um lugar "seguro" e posteriormente no epílogo da função faz a checagem deste valor.

As soluções apresentadas pelo `StackGuard`, `ProPolice` e `StackShield` dependem fortemente dos compiladores utilizados pelos desenvolvedores dos programas. Além

disso, mesmo que utilizando os compiladores corretos, as proteções geradas por estas ferramentas, em alguns casos, podem ser ineficazes.

3.1.1.4 Problemas com Abordagens Dependentes do Compilador

Segundo (Bulba e Kil3r, 2000), é possível superar todo o sistema de segurança gerado por estas ferramentas, através de ataques de *pointer subterfuge* ou *arc injection*. Basicamente a técnica utilizada modifica o endereço referenciado por um ponteiro para um endereço arbitrário dentro da *stack*. Este endereço pode ser, por exemplo, o código de retorno da função ou o local onde o `StackShield` armazena os endereços de retorno antes da chamada da função para comparação.

Com isso, é possível modificar o valor do endereço de retorno sem modificar o *canaries* no caso do `StackGuard` e `ProPolice` ou modificar a tabela contendo os endereços de retorno originais do `StackShield`. Apesar do `StackGuard` endereçar este problema utilizando *canaries* XOR randômicos do endereço de retorno, a exploração pode ocorrer de outras formas.(Bulba e Kil3r, 2000)

Portanto, estas ferramentas podem ser muito úteis para minimizar os efeitos de problemas de ataques de *stack smash*, mas em alguns casos podem não prevenir ataques de *arc injection* ou *pointer subterfuge*.

Infelizmente, as abordagens tomadas pelo `StackGuard`, `ProPolice` e `StackShield` podem não ser realmente efetivas no combate a problemas de *Buffer Overflow*. O problema continua existindo no código-fonte do programa, simplesmente ações paliativas são tomadas quando o problema vem à tona por determinados tipos de ataques. Por continuar existindo no código-fonte, o distribuidor do programa deve garantir que somente os códigos-binarios, previamente compilados com um compilador específico, sejam disponibilizados.

3.1.2 Abordagens Dependentes do Sistema

As abordagens dependentes do sistema descrevem a proteção inteiramente em tempo de execução, usualmente fazendo com que bibliotecas e o `Kernel` do sistema operacional sejam fortificados. Dentre as ferramentas disponíveis que utilizam abordagens dependentes do sistema estão algumas ditas bloqueadoras de comportamento (*behavior blockers*).

Os bloqueadores de comportamento impedem que certos comportamentos ocorram no sistema, pois estes comportamentos são sabidamente associados a algum tipo de ataque. Algumas ferramentas que garantem em tempo de execução as proteção contra problemas

de *buffer overflow* são: `Openwall`, `Libsafe`, `RaceGuard` e `Systrace`.

3.1.2.1 Openwall

O Openwall[1] é um projeto de *patch* para o `Kernel` do `Linux` que tem como objetivo aumentar o nível de segurança através de bloqueadores de comportamento. Ele implementa três formas de bloqueio básicas:

- *Stack* não executável. A princípio, sistemas *Intel x86* não permitem separar os atributos de leitura e escrita das páginas de memória virtual. O *Openwall* permite que esta distinção seja feita. Com isso é possível fazer com que a *stack* não seja executável, pois uma página de memória com atributo de escrita não pode possuir capacidade de execução. Portanto, ataques de *stack smash* são impedidos.
- Usuários convencionais, não superusuário, não podem criar *link* para arquivos que não sejam dono. Isto previne uma forma de ataque contra arquivos que um programa que rode com maiores privilégios deve abrir. Um atacante poderia ter modificado o arquivo original por um *link* para um arquivo de senhas, por exemplo.
- Processos que rodam com privilégios de superusuário não devem seguir *links*. Isto previne outros tipos de ataques contra arquivos temporários.

3.1.2.2 Libsafe

A `Libsafe`[2] é um invólucro para a biblioteca padrão do `C`, `glibc`. A idéia é checar os argumentos que são passados para as funções da `glibc` para verificar se são razoáveis e não irão causar problemas de segurança. O modo de operação da `Libsafe` pode ser observado na Figura 3.2

A Figura 3.3 ilustra um exemplo de como a função `strcpy` é reimplementada pela `Libsafe`. Na linha 16 do exemplo uma checagem é executada com a função `strnlen` antes que a função `strcpy` original possa ser executada. Note que a função `strcpy` original é obtida da biblioteca padrão através da função `getLibraryFunction`.

Esta abordagem é capaz de prevenir a exploração de *buffer overflows* e exploração de

[1]Maiores informações sobre o *patch* Openwall podem ser obtidas em:<http://www.openwall.com/linux>. Acesso em: 19 Jan. 2007

[2]Maiores informações sobre a `Libsafe` podem ser encontradas em:<http://directory.fsf.org/libsafe.html>. Acesso em 19 de Jan. de 2007.

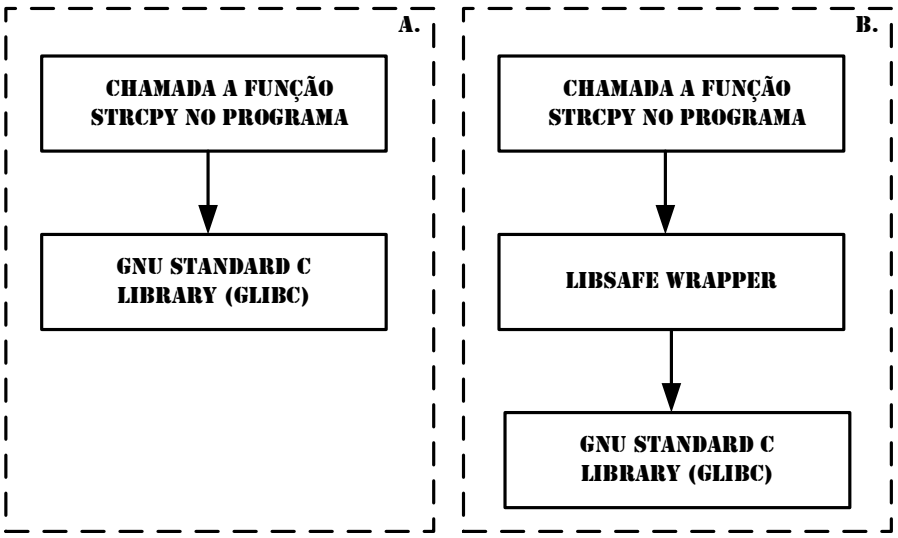

Figura 3.2 - (a) Chamada para a função `strcpy` em um sistema sem `libsafe`. (b) Chamada para a função `strcpy` em um sistema com `libsafe`.

```
1  char *strcpy(char *dest, const char *src)
2  {
3          static strcpy_t real_strcpy = NULL;
4          size_t max_size, len;
5          if (!real_memcpy)
6                  real_memcpy = (memcpy_t)
7                          getLibraryFunction("memcpy");
8          if (!real_strcpy)
9                  real_strcpy = (strcpy_t)
10                         getLibraryFunction("strcpy");
11         if ((max_size = _libsafe_stackVariableP(dest)) == 0) {
12                 LOG(5, "strcpy(<heap var> , <src>)\n");
13                 return real_strcpy(dest, src);
14         }
15         -- ... --
16                 if ((len = strnlen(src, max_size)) == max_size)
17                         _libsafe_die("Overflow caused by strcpy()");
18         real_memcpy(dest, src, len + 1);
19         return dest;
20 }
```

Figura 3.3 - Função `strcpy` reimplementada pela `Libsafe`. Todas as checagens necessárias são executadas antes da chamada da função `strcpy` original.

vulnerabilidades de *format string* sobre funções da biblioteca padrão do `C`, `glibc`. A maior limitação deste sistema é que ele é desabilitado se o programa for compilado com a opção `-formit-frame-pointer`.

3.1.2.3 RaceGuard

O `RaceGuard`(Cowan et al., 2001b) foi desenvolvido para não permitir que problemas de *race condition* ocorram na criação de arquivos temporários. Geralmente existe um intervalo de tempo entre a checar se um arquivo existe e a escrita de fato neste arquivo. Um ataque pode acontecer neste intervalo. A abordagem tomada pelo `RaceGuard` é justamente fazer com que a ação de checagem e o efetivo acesso ao arquivo seja atômica.(Cowan et al., 2001b), (Bishop e Dilger, 1996)

Esta abordagem poderia endereçar problemas de *buffer overflow* que necessitem de uma *race condition* para que a exploração ocorra. Entretanto, não tem mecanismos efetivos para identificar, eliminar ou limitar o impacto de qualquer exploração de *buffer overflow*.

3.1.2.4 Systrace

O `Systrace`[3] é uma implementação híbrida de controle e acesso e bloqueador de comportamento utilizado pelos sistemas `OpenBSD` e `NetBSD`. Este permite que o administrador especifique quais recursos um dado programa pode obter acesso.

Além disso, o `Systrace` possibilita que o administrador defina quais chamadas de sistema um programa pode executar, permitindo, desta forma, que o administrador implemente uma forma de bloquear certos comportamentos.

Esta abordagem não elimina, nem toma medidas quando uma tentativa de exploração de um *buffer overflow* ocorre. Ela simplesmente garante que menores privilégios de acesso sejam dados a um atacante quando este efetivamente invadir o sistema.(Cowan, 2003)

3.1.3 Abordagens Dependentes da Aplicação

As abordagens dependentes da aplicação prevêem as vulnerabilidades antes que a aplicação seja amplamente distribuída. Estas abordagens realizam o que é chamado de auditoria do programa. Esta auditoria pode ser realizada tanto de maneira estática como dinâmica. A análise estática visa determinar as propriedades da aplicação inspecionando ou o código-fonte ou o binário da mesma sem que esta seja executada. A suposta vantagem deste tipo de abordagem é que esta pode detectar erros que podem ser muito difíceis de serem identificados por outros métodos. Analisadores estáticos de código-fonte variam de reconhecedores de funções, *functions spotters*, que dificilmente são mais sofisticados do

[3]Maiores informações sobre o `Systrace` podem ser obtidas em: <`http://www.citi.umich.edu/u/provos/systrace/`>. Acesso em: 19 de Jan. de 2007.

que a ferramenta `grep`[4] e precompiladores. Novas abordagens envolvem a detecção de vulnerabilidades utilizando otimização de restrições, *constraint optimization*. (Heffley e Meunier, 2004)

Análise dinâmica da aplicação busca encontrar problemas de segurança durante a execução de um programa. Isto pode exigir que todos os caminhos de execução do programa sejam percorridos e exaustivamente testados com todos os tipos de entradas possíveis. Uma forma rudimentar é utilizar uma entrada aleatória, como as geradas pela ferramenta `fuzz`(Miller et al., 1990) ou `ISIC`[5]. Entretanto, em grandes e complexas aplicações a causa de um dado mau funcionamento pode ser difícil de localizar. (Heffley e Meunier, 2004)

Abordagens dependentes da aplicações necessitam de maior atenção do desenvolvedor. Abordagens dependentes do compilador requerem maiores cuidados do desenvolvedor no caso de uma distribuição de binários pré-compilados ou do usuário no caso de distribuição de códigos-fonte. Por fim, as abordagens dependentes do sistema precisam mais de ações tomadas pelo usuário.

Para que o programa se comporte de uma maneira mais uniforme nas diferentes plataformas, desenvolvedores devem se esforçar para que a segurança do programa dependa mais deles do que dos usuários. Desta forma, conseguindo uma maior segurança do programa independente do sistema e do usuário. Além de maior portabilidade sem riscos à segurança.

Das abordagens mostradas anteriormente, somente as que dependem da aplicação são capazes de efetivamente eliminar as vulnerabilidades presentes em um dado sistema. As outras formas não eliminam as vulnerabilidades do código-fonte. Portanto, não fazem com que o programa seja seguro por si só.

3.2 Metodologias Utilizadas por Analisadores Estáticos de Códigos-fonte

Em Chess e McGraw (2004) são discutidas algumas metodologias utilizadas por analisadores estáticos de códigos-fonte e quais cuidados devem ser tomados com estas ferramentas. Principalmente por que estas ferramentas são ditas de auxílio e não conseguem por si só resolver todos os problemas de segurança.

A forma mais rudimentar de se realizar a análise estática de códigos é através do uso da

[4] A ferramenta grep procura por expressões regulares em arquivos, ou em dados passados via entrada padrão. Maiores informações podem ser obtidas em: <http://unixhelp.ed.ac.uk/CGI/man-cgi?grep>. Acesso em: 12 Dez. 2006.

[5] Maiores informações sobre a ferramenta ISIC, *Ip Stack Integrity Checker*, podem ser encontradas em: <http://www.nestonline.com/TrinuxPB/isic.txt>. Acesso em: 12 Dez. 2006.

ferramenta `grep`. Possuindo uma boa lista de expressões regulares é possível revelar partes interessantes do código. Por outro lado, a estrutura gramatical da linguagem fonte não consegue ser analisada através da procura por expressões regulares. Comentários, cadeias de caracteres, declarações, instruções para compilação e chamadas de funções são tratadas da mesma maneira. (Chess e McGraw, 2004)

Para uma maior exatidão, a análise estática de códigos requer que as regras léxicas que definem a linguagem de programação do código sob análise sejam levadas em consideração. Através de um bom analisador léxico, é possível a distinção entre chamadas de funções vulneráveis e comentários. Como no exemplo da Figura 3.4

```
1 gets(&buf);
2 /* never ever call gets */
```

Figura 3.4 - Diferenciação entre uma chamada de função e um comentário.

O sistema de varredura, análise léxica, é responsável por ler os caracteres do código-fonte e organizá-los em unidades lógicas. Estas unidades são denominadas marcas, *tokens*. Na análise léxica é possível identificar um atributo de valor "3256" como pertencente à marca `NUM` de número. (Louden, 2004)

Na Figura 3.4, um analisador léxico poderia identificar o `gets` da primeira linha como sendo um atributo da marca `BADFUNC` de má função. Já o `gets` da segunda linha seria identificado como pertencente a um comentário.

Muitas ferramentas realizam análise estática do código-fonte em um nível léxico. Dentre elas temos a `ITS4`, `FlawFinder` e `RATS`[6]. Todas estas ferramentas fazem a varredura do código-fonte, gerando as marcas correspondentes. Estas marcas são então casadas contra um banco de funções inseguras. (Chess e McGraw, 2004)

Enquanto a varredura é um passo adiante da análise realizada com auxílio de ferramentas como `grep`, estas possuem um grande número de falsos positivos, o que pode ser comprovado por experimentos realizados por alguns autores[7]. Este grande número de falsos positivos ocorre, pois não existe nenhum esforço por parte destas ferramentas em realizar a análise sintática do código-fonte. Uma seqüência de marcas é muito melhor do que um conjunto de caracteres. Mas, ainda está muito distante de dizer como o programa vai se comportar quando este estiver em execução.

[6]Referências e maiores informações sobre estas ferramentas poderão ser encontradas adiante.
[7]Estes experimentos serão abordados na sessão 3.4.1 e 3.4.2

Segundo Chess e McGraw (2004) para que ocorra um acréscimo na precisão da análise estática é necessário que seja construída uma árvore sintática abstrata (AST - *abstract syntax tree*) (Louden, 2004). Tendo-se esta árvore, o próximo passo seria identificar o escopo da análise.

Uma abordagem possível com o uso da AST seria a análise local, *local analysis* onde o analisador realiza a análise de uma função do código-fonte por vez não considerando a relação entre as funções. Uma outra abordagem seria a análise em módulos, onde grupos de funções poderiam ser analisados quanto às interações entre si.

Outro escopo no qual a análise poderia ser realizada é o escopo global. Neste, todo o código-fonte do programa é analisado, levando em consideração todas as relações entre as funções.

3.3 Ferramentas de Auxílio à Análise Estática

Algumas ferramentas estão disponíveis hoje para análise estática de códigos. As mais difundidas são: `Flawfinder`, `ITS4`, `PScan`, `RATS`, `Splint` e `BOON`.

3.3.1 Flawfinder

A ferramenta `Flawfinder`[8] é uma ferramenta escrita em *Python* que examina códigos-fonte da linguagem `C` e reporta possíveis fraquezas de segurança. É uma ferramenta de código aberto e licença `GPL`[9].

Esta ferramenta trabalha examinando o código-fonte do programa em um nível léxico. Possui um banco de dados de funções das linguagens `C` e `C++` com problemas bem conhecidos. Como: `strcpy`, `strcat`, `gets`, `sptrintf` e funções da família `scanf`. Além disso, identifica problemas de *format string* nas funções: `[v][f]printf [v]snprinf` e `syslog` e identifica, também, condições de disputa encontradas em funções como: `access`, `chown`, `chgrp`, `chmod`, `tmpfile`, `tmpnam` e `mktemp`.

Os riscos são ordenados do maior risco "5" ("*maximum risk*") para o menor risco "0" ("*no risk*"). O usuário pode setar a partir de qual risco deseja ser alertado. Além disso, o relatório gerado pelo programa pode ser tanto em "texto puro" quanto em `HTML`.

[8] A ferramenta Flawfinder pode ser encontrada em: <http://www.dwheeler.com/flawfinder/>. Acesso em: 12 mai. 2005.

[9] A licença GPL, *"GNU Public License"*, pode ser encontrada em: <http://www.gnu.org/copyleft/gpl.html>. Acesso em: 12 mai. 2005.

3.3.2 ITS4

ITS4[10] é o acrônimo de "*It's The Software, Stupid! [Security Scanner]*". Como o *Flawfinder*, é uma ferramenta que identifica possíveis problemas em trechos de códigos C ou C++. Esta ferramenta faz uma análise léxica do programa gerando um conjunto de marcas. Estas marcas são então casadas contra funções vulneráveis listadas em um banco de dados. A razão pela qual a ferramenta ITS4 não realiza uma análise mais apurada com a ajuda de uma análise sintática, *parsing*, está no fato de que esta análise não pode ser feita juntamente com a programação. A ITS4 é desenvolvida para dar suporte a programadores durante o processo de codificação, mostrando quais os potenciais problemas de segurança estão presentes no código.(Wilander e Kamkar, 2002)

Esta ferramenta permite que o usuário defina qual o método de ordenação do relatório de saída. Ou seja, o relatório pode ser ordenado de maneira crescente quanto ao risco, pelos arquivos que foram analisados, pela vulnerabilidade ou pela ordem em que cada possível problema foi identificado.

Outra característica que distingue o ITS4 das outras ferramentas é o fato de permitir sua integração com o editor GNU Emacs[11], possibilitando ao programador escolher quais partes do código devem ser ignoradas, como no exemplo da Figura 3.5

```
strcpy(dst, src);  /* ITS4: ignore */
```

Figura 3.5 - Exemplo de uma anotação no estilo ITS4.

3.3.3 PScan

O PScan[12] é uma ferramenta de licença GPL que procura por problemas nas cadeias de formatação das funções da família printf. Como esta ferramenta não checa por quaisquer outras funções ou tipo de problema o seu uso é bastante restrito. A menos que definições adicionais de funções sejam passadas para o programa.

Entretanto, mesmo que novas funções fossem inseridas, a checagem de formatação

[10] A ferramenta ITS4 pode ser encontrada em: <http://www.cigital.com/its4/>. Acesso em: 12 mai. 2005.

[11] Maiores informações sobre o editor GNU Emacs podem ser obtidas em: <http://www.gnu.org/software/emacs/emacs.html>. Acesso em: 12 mai. 2005.

[12] Maiores informações sobre a ferramenta PScan podem ser obtidas através do *Internet Archive Wayback Machine* em:<http://web.archive.org/web/20060423012759/http://www.striker.ottawa.on.ca/~aland/pscan/>. Acesso em 20 Jan. 2007.

realizada nestas funções é a mesma realizada para as funções da família `printf`.

3.3.4 RATS

RATS[13], acrônimo de *"Rough Auditing Tool for Security"*, é uma ferramenta de código aberto distribuída sob a `GPL` e desenvolvida pela *Secure Software Inc.* O objetivo desta ferramenta é encontrar possíveis problemas como *buffer overflow* e `TOCTOU`[14] em códigos-fonte escritos em `C/C++`, `Perl`, `PHP` e `Python`.

Para cada uma destas linguagens o RATS possui um banco de dados `XML` correspondente. Isto facilita a inserção de novos possíveis problemas, mas faz com que seja necessário que a biblioteca `Expat` esteja instalada no sistema.

Esta ferramenta, como a `ITS4` e a `Flawfinder`, faz a análise léxica do código para encontrar identificadores que casem com os encontrados no banco de dados de funções inseguras.

3.3.5 Splint

O nome original desta ferramenta, `LCLint`, bem como suas funcionalidades foram herdadas de uma popular ferramenta de análise estática para linguagem `C` chamada `Lint` (Johnson, 1978). A ferramenta `LCLint` foi então aprimorada para encontrar problemas específicos de segurança e passou a se chamar *Secure Programing Lint* ou `Splint`.(Evans et al., 1994)

A abordagem da ferramenta `Splint` é utilizar comentários inseridos pelos programadores para realizar a análise estática do código em nível sintático. Isto significa que esta ferramenta pode diferenciar entre a utilização correta ou incorreta de uma função melhor do que ferramentas que trabalham em um nível léxico. Estas anotações são inseridas tanto nos códigos dos programas quanto nas bibliotecas padrões.

Novas anotações não antes suportadas pelo `LCLint` permitem que o programador especifique pré-condições e/ou pós-condições. A cláusula *requires* requer que uma dada pré-condição seja satisfeita para que problemas não ocorram. Já a cláusula *ensures* diz qual é a pós-condição das variáveis após a execução de dada função. Um exemplo de utilização de anotações com estas cláusulas pode ser observado na Figura 3.6.

[13] A ferramenta RATS pode ser encontrada em: <http://www.securesw.com/rats/>. Acesso em: 12 mai. 2005.

[14] TOCTOU: Time of Check, Time of Use. É uma forma de race condition ou condição de disputa. Geralmente ocorre quando um processo checa alguma propriedade em um arquivo e nas instruções subseqüentes tenta acessar este arquivo. O problema é que, mesmo que a instrução de uso venha logo após a instrução de chegacem da propriedade, existe a possibilidade de que um segundo processo possa de uma forma maliciosa invalidar a primeira checagem.

```
char *strcpy (char *s1, char *s2)
/*@requires maxSet(s1) >= maxRead(s2)@*/
/*@ensures maxRead(s1) == maxRead (s2)
  /\ result == s1@*/;
```

Figura 3.6 - Exemplo de um programa com anotação no estilo `Splint`.

A anotação anterior tem como pré-condição a necessidade de que o *buffer* `s1` seja grande o bastante para armazenar o *buffer* `s2`. Já a pós-condição mostra que ao finalizar a execução a quantidade de *bytes* utilizados pelo *buffer* `s1` é a mesma que a utilizada pelo *buffer* `s2`. Note que a cláusula *result* informa qual deve ser o valor de retorno da função. No caso, a função `strcpy` tem como retorno um ponteiro para o *buffer* de destino, no caso, `s1`.

Para que as possíveis vulnerabilidades possam ser encontradas as restrições geradas a partir das anotações no código do programa devem ser resolvidas. Através da resolução destas restrições, os devidos alertas são gerados caso algum problema seja identificado.

3.3.6 BOON

A ferramenta `BOON`[15], acrônimo de *Buffer Overrun DetectiON*, foi desenvolvida para detectar vulnerabilidades de *buffer overflow* em códigos escritos na linguagem `C`. Como grande parte dos *buffer overflows* ocorrem em cadeias de caracteres, a idéia da ferramenta é modelar cada uma das cadeias de caracteres com duas propriedades. A primeira é a quantidade alocada para determinada variável, a segunda é a quantidade de *bytes* atualmente em uso.

Todas as funções que manipulam estas cadeias são então modeladas pelos efeitos causados nas duas propriedades analisadas. Estes efeitos são utilizados na realidade para gerar o que é chamado de restrições. Por fim estas restrições são resolvidas. As restrições resolvidas são então confrontadas com os valores inicialmente declarados para as cadeias de caracteres. Caso possa ocorrer um *buffer overflow*, os devidos alertas são gerados.

3.4 Comparação entre as Ferramentas de Auxílio à Análise Estática de Código-fonte

Alguns autores como Wilander e Kamkar (2002) e Heffley e Meunier (2004) fizeram algumas comparações dos resultados obtidos através da utilização de ferramentas de auxílio à análise estática que são publicamente disponíveis e que possuem grande

[15]Maiores informações sobra a ferramenta `BOON` podem ser obtidas em: <`http://www.cs.berkley.edu/~daw/boon/`>. Acesso em 20 de Jan. de 2007.

notoriedade. Wilander e Kamkar (2002) geraram testes próprios para identificar a eficiência das ferramentas. Heffley e Meunier (2004) optaram por testar as ferramentas contra programas de código aberto que já continham um histórico de vulnerabilidades bastante rico, como o caso do `WU-FTPD`[16].

3.4.1 Testes de Wilander e Kamkar

O código-fonte de teste gerado por Wilander e Kamkar (2002) contém vinte funções vulneráveis retiradas do banco de dados do `ITS4`, algumas vulnerabilidades listadas em Wheeler (2003) e todas as funções da família `[fvsn]printf`. Segundo o próprio autor, este teste não prima por ser completo ou justo, mas traz resultados bastante interessantes. As ferramentas utilizadas na análise foram: `Flawfinder`, `ITS4`, `RATS`, `Splint` e `BOON`. Como a ferramenta `BOON` trabalha somente com vulnerabilidades de *buffer overflow*, os testes que envolviam outros tipos de vulnerabilidades não foram realizados com esta ferramenta. O resultado resumido do experimento realizado pode ser observado na Tabela 3.2.

Tabela 3.2 - Resumo dos resultados obtidos por John Wilander.

	Flawfinder	**ITS4**	**RATS**	**Splint**	**BOON**
Verdadeiro Positivo	22 (96%)	21 (91%)	19 (83%)	7 (30%)	4 (27%)
Falso Positivo	15 (71%)	11 (52%)	14 (67%)	4 (19%)	4 (31%)
Verdadeiro Negativo	6 (29%)	10 (48%)	7 (33%)	17 (81%)	9 (69%)
Falso Negativo	1 (4%)	2 (9%)	4 (17%)	16 (70%)	11 (73%)

Na Tabela 3.2, os verdadeiros positivos são locais indicados pelas ferramentas como sendo potencialmente vulneráveis e que realmente os são. Já os falsos positivos são locais apontados pelas ferramentas como sendo potencialmente vulneráveis e que na realidade não os são.

Os verdadeiros negativos são trechos de código identificados como inofensivos pelas ferramentas e que são realmente trechos inofensivos presentes no arquivo de teste. Já os falsos negativos são trechos identificados como inofensivos pelas ferramentas, mas que na realidade são vulnerabilidades presentes de fato no arquivo de teste.

Wilander e Kamkar (2002) concluem que as três ferramentas que fazem análise léxica, `ITS4`, `Flawfinder` e `RATS`, são muito parecidas quanto ao descobrimento de verdadeiros positivos. Apesar de que grande parte das vulnerabilidades utilizadas como teste poderem

[16]Maiores informações sobre o programa WU-FTPD podem ser encontradas em: <`http://www.wu-ftpd.org/`>. Acesso em: 12 mai. 2005.

ser encontradas no banco de dados destas ferramentas. Entretanto, estas ferramentas se diferenciam consideravelmente no número de falsos positivos onde a `ITS4` é melhor.

Para programadores que utilizam técnicas de programação segura as saídas das ferramentas devem ser sucintas, visto que grande parte dos problemas já estão sendo solucionados em tempo de programação, antes que as ferramentas sejam necessárias. Para programadores menos experientes a saída pode ser muito extensa e como as ferramentas não dão instruções de como solucionar os problemas, estes programadores devem procurar ajuda em outras fontes.

Ferramentas como `Splint` e `BOON` encontraram pouquíssimas vulnerabilidades. O autor do `Splint`, David Larochelle diz que as vulnerabilidades não encontradas por sua ferramenta não são uma grande ameaça já que poderiam ser encontradas com o uso do comando `grep`. Já a ferramenta `BOON` é um protótipo e deve ser julgada desta forma.

Nenhuma das ferramentas possui um grande número de verdadeiro positivos combinado com um baixo número de falsos positivos. A conclusão de Wilander e Kamkar (2002) é que nenhuma ferramenta é boa o bastante para trazer tranqüilidade ao programador e combinar a saída das ferramentas pode ser muito trabalhoso.

3.4.2 Testes de Heffley e Meunier

Em Heffley e Meunier (2004), os autores escolheram um programa aberto baseados no número e variedade de vulnerabilidades presentes no mesmo. Optaram pelo programa `WU-FTPD` em uma versão que possuía dezoito vulnerabilidades bem documentadas, que proviam exemplos reais de *race conditions*, *code injection*, *memory leak*, vulnerabilidades em *format strings* e alguns *buffer overflows*. Portanto, as ferramentas analisadas foram executadas sobre o código da versão 2.6.0 do programa.

A idéia dos testes realizados era a seguinte, sem saber onde especificamente as vulnerabilidades ocorreram, tentar utilizar as saídas geradas pelos programas para redescobrir as vulnerabilidades documentadas. Durante esse processo também foi medida a facilidade de uso da ferramenta, o tempo de inicialização e execução dos programas, além do tempo requerido para se fazer uso das saídas obtidas.

O estudo realizado também contabilizou o número de falsos positivos e falsos negativos que ocorreram durante o experimento. As ferramentas utilizadas neste experimento foram: `Cqual`, `Splint`, `FlexeLint`, `RATS`, `ITS4`, `Flawfinder` e `Pscan`.

Segundo os autores, três ferramentas de auditoria não passaram pelo primeiro estágio

dos testes, que era identificar problemas no programa `WU-FTPD`, são elas: `Cqual`, `Splint` e `FlexeLint`. A ferramenta `Cqual`, além de requerer que anotações sejam realizadas no código-fonte do programa, é totalmente dependente do editor `Emacs` o que impossibilitou sua utilização.

O `Splint` oferece a possibilidade de que o programador faça anotações no código-fonte do programa. No caso, como a estrutura do funcionamento do programa testado não é conhecida, estas anotações não puderam ser inseridas. Além disso, o funcionamento do analisador é muito complicado, visto que ele necessita percorrer todos os arquivos de cabeçalho incluindo os arquivos da biblioteca padrão da linguagem `C`.

O programa `FlexeLint` possuía quase os mesmos problema que o `Splint`. Além disso, os resultados obtidos deste programa não são focados em segurança.

Segundo Heffley e Meunier (2004) as ferramentas `Flawfinder`, `RATS` e `ITS4` são pouco mais do que analisadores léxicos. Estas ferramentas podem indicar em qual linha a vulnerabilidade ocorreu, a severidade da vulnerabilidade encontrada, se a identificação da vulnerabilidade foi precisa e o que pode significar o problema apontado.

Entretanto, as ferramentas que foram analisadas geram um grande número de falsos positivos o que faz com que o trabalho de análise manual de todo o código-fonte ainda gire em torno de 67% até 75%. Estes falsos positivos podem ser facilmente gerenciados em pequenos e médios projetos, em grandes projetos podem gerar muito retrabalho.

Tentar melhorar um programa através de uma rápida análise de segurança utilizando uma destas ferramentas de auxilio como `Flawfinder`, `RATS` e `ITS4` pode não ser uma boa idéia. Visto que, muitas das vulnerabilidades podem não ser tratadas de maneira adequada por estarem "camufladas" em um grande número de falsos positivos.

Ferramentas que fazem a análise léxica do código-fonte podem ser úteis para programadores que não estejam familiarizados com conceitos de segurança. Através da análise de seu próprio código, o programador terá uma lição prática sobre segurança. Estas ferramentas também podem ser úteis para programadores intermediários que sabem que algumas chamadas podem causar problemas mas, não sabem quais são estes problemas. Isto faz com que eles entendam melhor como os problemas de segurança ocorrem ao invés de apenas saberem o que não devem fazer.

Os autores concluem que ferramentas que retornam poucos falsos positivos, mesmo que existam falsos negativos, são melhores do que ferramentas que geram alertas para todas as entradas que casam um determinado padrão, sendo estas entradas vulneráveis ou não.

Um conceito de segurança que não é endereçado por nenhuma ferramenta de análise léxica é o fato de existirem ou não validações de entrada antes da chamada de funções inseguras. Com isso, sempre que uma função insegura aparecer, estas ferramentas gerarão alertas, mesmo que as devidas validações tenham sido realizadas anteriormente.

Por fim, o desenvolvimento de ferramentas de auditoria de códigos é extremamente benéfica para aumentar a qualidade do programa e sua segurança. Traz também economia de tempo e dinheiro para os desenvolvedores e consumidores.

3.5 Considerações Finais

Neste capítulo mostrou-se que existem várias ferramentas disponíveis que podem ser utilizadas nos esforços de eliminação de vulnerabilidades de *buffer overflow*. A abordagem mais completa seria a utilização de ferramentas nas três abordagens apresentadas: Dependente do compilador, dependente do sistema e dependente da aplicação.

Certamente, o nível de segurança do sistema como um todo aumenta conforme estas ferramentas vão sendo utilizadas. Entretanto, este trabalho foi focado em soluções desenvolvidas para uma abordagem dependente da aplicação, mais especificamente nas relacionadas à análise estática de códigos-fonte.

CAPÍTULO 4

AMBIENTE DE ANÁLISE

Este trabalho foca-se em uma abordagem dependente do programa. O objetivo é propor um ambiente que permita a realização de análises estáticas de código-fonte com ênfase em segurança. Para isso são levados em consideração dois aspectos levantados nos estudos de Wilander e Heffley. Segundo Heffley e Meunier (2004) ferramentas com poucos falsos positivos são mais interessantes que as atuais mesmo que existam falsos negativos. Segundo Wilander e Kamkar (2002) nenhuma ferramenta possui altas taxas de verdadeiro positivo com baixas taxas de falso positivos. Além disso, segundo Heffley e Meunier (2004) nenhuma ferramenta verifica se checagens anteriores foram realizadas.

4.1 Objetivos do Ambiente

Para que o ambiente proposto possa atingir seu principal objetivo, que é permitir análises estáticas eficazes de código-fonte, algumas características principais foram levantadas. Dentre elas, destacam-se:

- O ambiente deve ser capaz de entender o fluxo de execução do programa;
- Especificações de novas funções inseguras devem poder ser inseridas no ambiente sem que exista a necessidade de recompilá-lo;
- O ambiente deve encontrar o maior número de problemas reais de *buffer overflow*, mesmo que isso signifique um grande consumo de tempo por parte da ferramenta;
- A identificação de funções inseguras, antes, realizada na análise léxica por outras ferramentas como `ITS4`, `FlawFinder` e `RATS` deve ser realizada em um nível sintático;
- A análise de restrições, apesar de não ser implementada nos protótipos desenvolvidos, deve ser suportada pelo ambiente;
- Esforços para reduzir o número de falsos positivos devem ser empregados.
- A análise pode ser realizada localmente, em módulos ou em todo o código-fonte;

4.2 O Ambiente

A estrutura proposta para o ambiente é como a observada na Figura 4.1.

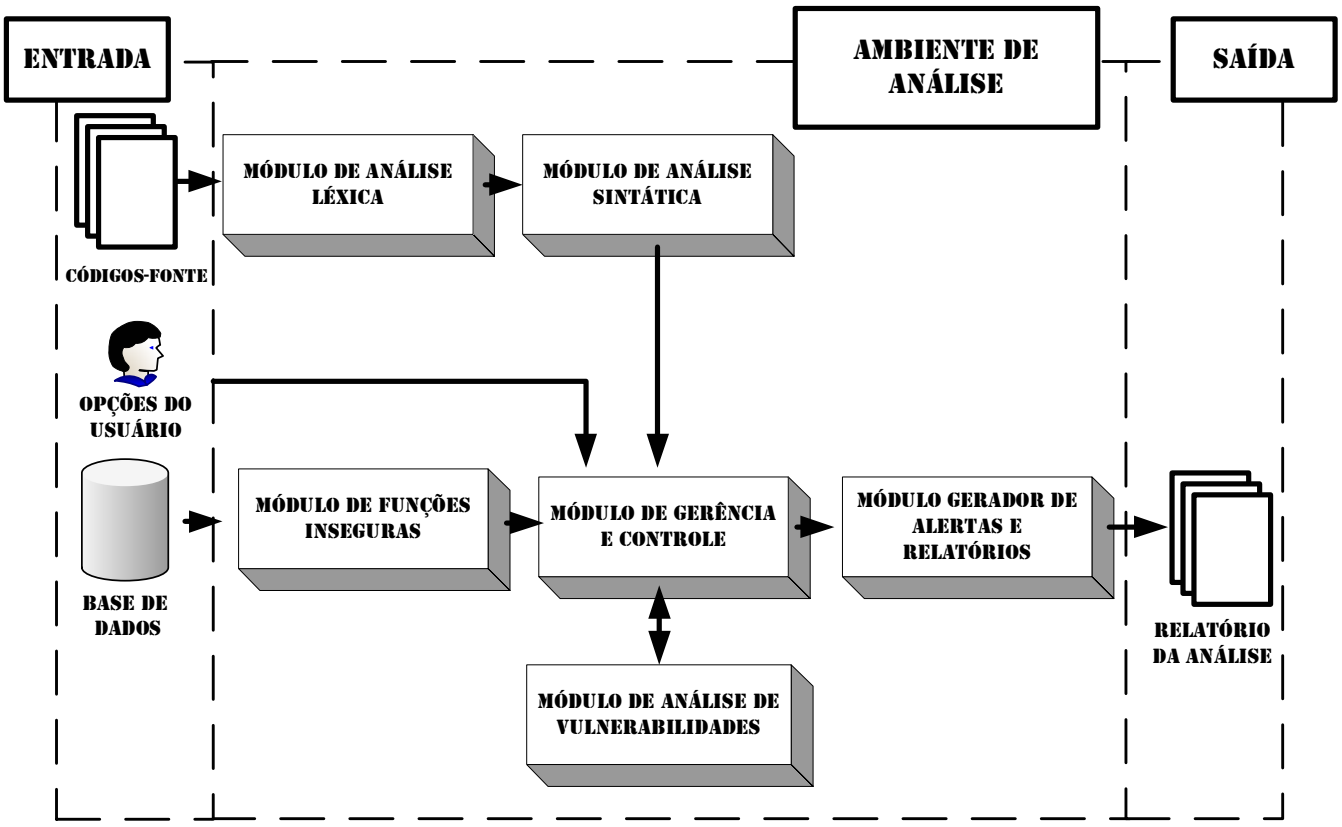

Figura 4.1 - Estrutura do ambiente de análises proposto.

O ambiente de análises é subdividido em módulos. Cada um destes módulos possui características e funcionalidades distintas. Além disso, o programa recebe como entrada dados vindos de códigos-fonte, usuários e um banco de dados. A saída do ambiente é um conjunto de relatórios de análise, que apontam possíveis problemas encontrados nos códigos-fonte analisados.

O ambiente é iniciado através de uma interação com o usuário que estabelece o que deverá ser realizado com os códigos-fonte fornecidos. Além de passar as opções desejadas, o usuário deverá informar quais são e onde estão localizados os códigos-fonte.

A partir do momento em que o ambiente recebe os códigos-fonte, este inicia a etapa de análise léxica. Esta análise gera as marcas, *tokens*, que são utilizadas pelo módulo de análise sintática.

A análise sintática preencherá parte das informações da árvore de análise sintática. Esta árvore será então utilizada pelo módulo de gerência e controle para que o fluxo de execução

do programa possa ser construído. Também é parte do módulo de análise sintática, inserir informações sobre o contexto, para que se possa saber se uma dada variável é local ou não.

O módulo de gerência e controle recebe uma estrutura preenchida pelo módulo de funções inseguras. Esta estrutura armazena todas as funções sabidamente inseguras e quais são as restrições na utilização de cada uma delas.

O módulo de análises de vulnerabilidade recebe partes ou toda a árvore de análise sintática. Com isso, poderá realizar análises como: busca por funções inseguras, análise de extravasamento de cadeias através de análise de restrições, análise de ponteiros, entre outras.

O módulo gerador de alertas e relatórios deve receber informações do módulo de gerência e controle, formatar a saída para um padrão estabelecido e informar o usuário sobre os problemas encontrados. É possível que este módulo também acesse o módulo de funções inseguras para que formas de sanar o problema encontrado possam ser sugeridas. Entretanto, esta funcionalidade não está explícita na Figura 4.1.

4.2.1 Módulo de Análise Léxica

O módulo de análise léxica fará exatamente a análise léxica ou varredura do código-fonte. A varredura tem por objetivo transformar seqüências de caracteres em uma representação interna chamada "marca", *tokens*. As marcas podem ser divididas em categorias. Dentre estas categorias destacam-se as palavras reservadas, identificadores, números e símbolos especiais.

As marcas **WHILE**, **IF** e **THEN** são exemplos de marcas relacionadas a palavras reservadas. **ID** é uma marca relacionada a identificadores, **NUM** está relacionada a números. Por fim, **PLUS** e **MINUS** são marcas relacionadas a símbolos especiais. (Louden, 2004)

A marca diferencia-se da seqüência de caracteres que esta representa. Esta seqüência por vezes é chamada de lexema ou valor. Algumas marcas possuem somente um lexema como a marca **IF** que possui somente o lexema "if". Entretanto, algumas marcas podem representar infinitos lexemas como no caso da marca **ID**.

Para que seja possível identificar a que marca um dado lexema pertence, expressões regulares costumam ser utilizadas. A Figura 4.2 ilustra a definição de algumas marcas, utilizando-se expressões regulares.

```
letra = [a-zA-Z]
digito = [0-9]
numero = digito+
identificador = letra(letra|digito)*
```

Figura 4.2 - Exemplos de expressões regulares

No exemplo da Figura 4.2 a expressão regular `letra`, casa qualquer letra presente no alfabeto. A expressão dígito casa qualquer dígito de "0" até "9". A expressão regular identificador pode ser composta de uma única letra ou de uma letra seguida de uma ou mais letras ou dígitos. Exemplos de lexemas que casam com a expressão regular identificador são: `a`, `ab`, `bac`, `a23`, `a1`, `a43c`.

As marcas, compostas por expressões regulares, são definidas pela gramática da linguagem fonte. No ambiente proposto, o módulo de análise léxica irá atuar conforme o módulo de análise sintática requiser. Ou seja, ao invés de determinar todas as marcas presentes em um dado código-fonte e posteriormente enviá-las para o analisador sintático, o analisador sintático é que irá requisitar que a análise léxica seja realizada conforme o necessário.

Diferentemente de outras abordagens, tomadas em algumas ferramentas de análise estática, os lexemas que possam corresponder a funções inseguras não serão marcados como tal neste nível de análise. Note que, a marca relacionada a identificadores não distingue chamadas de funções de variáveis ou declarações de funções em uma análise léxica tradicional.

4.2.2 Módulo de Análise Sintática

O módulo de análise sintática é responsável por requisitar as marcas do módulo de análise léxica, determinar a estrutura sintática de um dado programa segundo o conjunto de marcas observadas na análise léxica e construir uma árvore de análise sintática, ou árvore sintática, que represente a estrutura do programa.

Dentro do ambiente proposto, o objetivo da análise sintática é preencher um conjunto de estruturas previamente definido. Estas estruturas são semelhantes às árvores de análise sintática convencionais. Entretanto, diferenciam-se no fato de que requisitos de segurança devem ser implementados nestas estruturas e nem toda regra gramatical da linguagem fonte irá, de fato, estar contemplada nas estruturas.

As estruturas básicas que poderão ser utilizadas no módulo de análise sintática são: `stt_variavel`, `stt_funcao`, `stt_argumento`, `stt_restricao`, `stt_acao`, `stt_parametro`

e `stt_se`.

4.2.2.1 stt_funcao

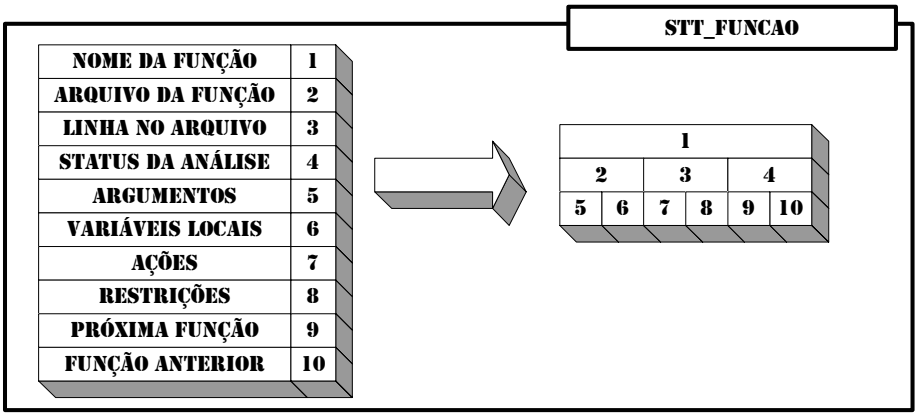

Figura 4.3 - Organização da estrutura `stt_funcao`.

A estrutura de dados `stt_funcao` pode ser observada na Figura 4.3. Esta estrutura possui um total de dez campos onde os quatro primeiros são estáticos e os seis últimos são ponteiros para outras estruturas. Os campos estáticos indicam o nome da função, qual o arquivo e a linha do código-fonte que esta função está sendo implementada. O quarto campo estático indica se a função já foi analisada e qual o *status* da análise.

Os outros campos são ponteiros para outras estruturas. No caso, o campo "argumentos" aponta para uma estrutura do tipo `stt_argumento` que é o primeiro argumento da função. Já o campo "variáveis locais" aponta para uma estrutura do tipo `stt_variavel` que representa a primeira variável local da função.

O campo "ações" indica qual o conjunto de funções que são executadas a partir da função representada na estrutura atual. Portanto, este campo aponta para uma estrutura do tipo `stt_acao` que representa a primeira ação tomada a partir da função sob análise.

Para que seja possível uma análise com o uso de restrições, um campo de restrições também é inserido em cada uma das funções. Este campo aponta para uma estrutura do tipo `stt_restricao` que representa a primeira restrição que é aplicada à função.

Os últimos dois campos são utilizados para que seja criada uma lista duplamente encadeada que permita que qualquer função seja alcançada a partir de qualquer outra

função.

4.2.2.2 stt_argumento

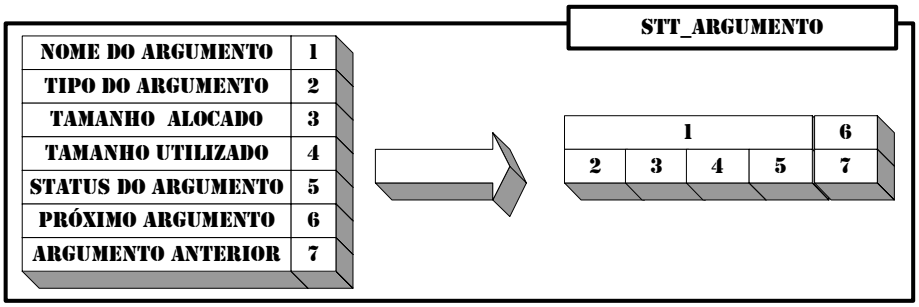

Figura 4.4 - Organização da estrutura `stt_argumento`.

A estrutura proposta para armazenar os argumentos é como a observada na Figura 4.4. Esta estrutura armazena informação sobre o nome do argumento, o tipo deste argumento, tamanho alocado, tamanho utilizado e status do argumento.

Por fim, estão presentes dois ponteiros para outras estruturas do tipo `stt_argumento`, que faz com que seja possível alcançar qualquer argumento partindo-se de qualquer outro argumento. Isto é possível através da implementação de uma lista duplamente encadeada.

Note que uma lista duplamente encadeada de estruturas do tipo `stt_argumento` está associada a uma única função. Portanto, o número de listas de argumentos é proporcional ao número de funções do código-fonte que possua argumentos.

4.2.2.3 stt_variavel

A estrutura `stt_variavel` pode ser observada na Figura 4.5. Esta estrutura possui as mesmas informações presentes na estrutura do tipo `stt_argumento`.

Da mesma forma que a estrutura `stt_argumento`, uma lista duplamente encadeada de estruturas `stt_variavel` é utilizada para cada função. Sendo o número de listas duplamente encadeadas proporcional ao número de funções do programa.

4.2.2.4 stt_acao

A Figura 4.6 ilustra a estrutura `stt_acao` que é utilizada para sinalizar as ações que são tomadas em cada uma das funções do programa. Por ação, estabelece-se chamadas a

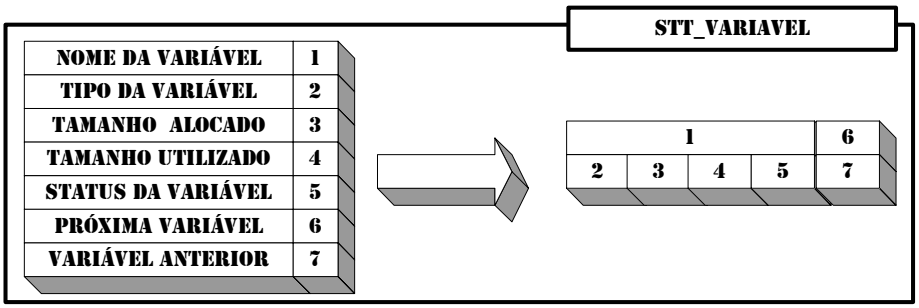

Figura 4.5 - Organização da estrutura `stt_variavel`.

outras funções do programa.

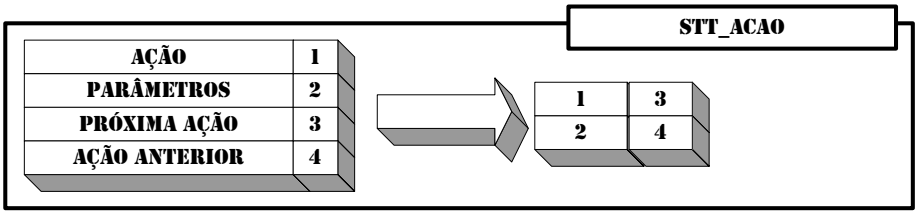

Figura 4.6 - Organização da estrutura `stt_acao`.

A idéia é que para cada função exista uma lista, cronológica, de ações. Com isso, é possível rastrear em quais instantes determinadas chamadas a funções específicas foram realizadas. É justamente o correto preenchimento desta estrutura que permite que o fluxo de execução de um programa possa ser esboçado.

Os campos presentes nesta estrutura são um ponteiro para a função que está sendo requisitada, um ponteiro para uma lista de parâmetros que serão passados para a função quando esta é requisitada. Por fim, existem dois ponteiros, um para a próxima função e um para a função anterior.

4.2.2.5 stt_parametro

A Figura 4.7 ilustra a estrutura `stt_parametro`. Esta estrutura tem o objetivo de simplesmente montar uma lista de parâmetros para cada ação realizada por uma função.

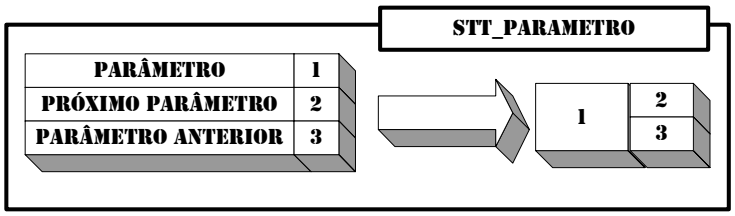

Figura 4.7 - Organização da estrutura `stt_parametro`.

4.2.2.6 stt_restricao

A estrutura `stt_restricao` pode ser observada na Figura 4.8. Esta estrutura simples possui um campo para uma cadeia de caracteres que representa uma restrição. Esta restrição pode ser utilizada em uma análise de restrições.

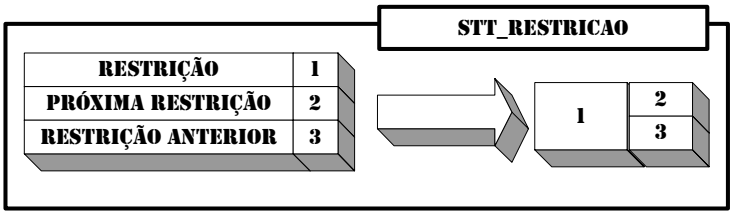

Figura 4.8 - Organização da estrutura `stt_restricao`.

Os outros dois campos apontam para a restrição imediatamente posterior e para a restrição imediatamente anterior, isto para que exista uma lista duplamente encadeada de restrições.

4.2.2.7 stt_se

A estrutura apresentada na Figura 4.9 é a `stt_se` e é utilizada para indicar uma modificação condicional no fluxo de execução do programa. Inicialmente, esta estrutura será utilizada somente quando a instrução `IF` ocorrer.

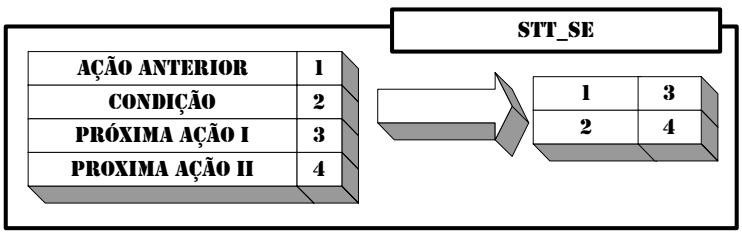

Figura 4.9 - Organização da estrutura `stt_se`.

Neste caso, a estrutura possui três ponteiros e a expressão que foi utilizada para definir o fluxo de execução. O primeiro ponteiro é utilizado para indicar qual a ação imediatamente anterior. Os outros dois ponteiros são utilizados para indicar os fluxos criados pela instrução `IF`.

4.2.3 Módulo de Funções Inseguras

O módulo de funções inseguras é responsável por obter dados sobre funções sabidamente inseguras de um arquivo ou banco de dados e preencher uma estrutura que poderá ser utilizada posteriormente para que análises de vulnerabilidades possam ser realizadas. Para todos os efeitos as funções inseguras são agrupadas em um assim chamado banco de funções inseguras.

Neste módulo é importante que novas funções ou novas restrições sabidamente inseguras possam ser inseridas sem que seja necessário modificar drasticamente o ambiente de análise. Além disso, um analista de segurança não precisa possuir um profundo conhecimento do ambiente para inserir funções particularmente interessantes.

Inúmeras informações relacionadas a uma dada função devem estar presentes no banco de funções inseguras. Informações de como a vulnerabilidade pode ser eliminada, de como a função insegura pode ser utilizada sem expor uma vulnerabilidade, restrições que devem ser satisfeitas para que a função possa ser executada sem maiores problemas, entre outras.

Na Figura 4.10 é ilustrada a estrutura que deve ser preenchida neste módulo de funções inseguras. Os campos presentes na figura são:

- **ID:** É o identificador da função na base de dados. Este é um inteiro de quatro *bytes* que distingüe univocamente uma função.
- **FNC_NAME:** É o nome da função. Este campo é dinamicamente definido e é a cadeia de caracteres utilizada pelos programas para chamar uma dada função.
- **SEVERITY:** É o grau de impacto gerado no sistema afetado para uma dada vulnerabilidade, definido através de um inteiro de quatro *bytes*.
- **REQUIRES:** São os pressupostos necessários para que a função utilizada possa ser considerada a mesma presente no banco de dados. Por exemplo, verificando-se os arquivos de cabeçalhos ou o sistema operacional. Este campo é dinamicamente alocado.
- **RESTRICTION:** Mostra como a função em questão pode ser utilizada de

forma a não expor uma vulnerabilidade. A exemplo do campo `FNC_NAME` e `REQUIRES` este campo é dinamicamente alocado.

- **SRCARG:** Muitas funções inseguras obtêm os dados que podem ser extravasados de fontes de dados comuns. Este campo é definido por um inteiro de quatro *bytes* e é reservado a identificar qual a origem dos dados utilizada.

- **DSTARG:** Similarmente ao campo `SRCARG` muitas funções inseguras têm o destino dos dados bem definido. Muitas das vezes o destino dos dados é justamente o *buffer* que pode ser extravasado.

- **FORMATARG:** É um inteiro de quatro *bytes* que define se a função insegura precisa ou não possuir um *format string* e define se este *format string* deve possuir ou não um modificador de tamanho, *length modifier*.

- **FORMATARGNRO:** É um inteiro de quatro *bytes* que define qual o argumento da função que deve ser o *format string*. Se este número for positivo o *format string* é procurado dos primeiros argumentos para os últimos. Se este for negativo este argumento é procurado dos últimos para os primeiros. Caso este valor seja definido como '0' qualquer argumento pode ser o *format string*.

- **PREVFUNC:** Descreve a existência de uma chamada anterior que permite que a chamada da função insegura possa ser realizada de maneira segura. Geralmente, estas funções são funções de checagem do tamanho do *buffer* de origem. Este campo é dinamicamente alocado.

- **PREVARG:** É um inteiro de quatro *bytes* que identifica o argumento que deve ser checado na chamada anterior. Geralmente, este campo é definido como o *buffer* de origem da função insegura. Esta entrada é utilizada em conjunto com a entrada `PREVFUNC`.

- **COMMENT:** É um campo dinamicamente alocado que descreve resumidamente o funcionamento de uma função insegura.

- **MITIGATION:** É uma descrição resumida de como o programador pode modificar seu código-fonte para eliminar a vulnerabilidade identificada. Este campo também é dinamicamente alocado.

- **MITI_NAME:** Em alguns casos, uma função que realiza as mesmas operações que a função insegura de uma forma mais segura pode ser utilizada. Este campo, dinamicamente alocado, possui o nome da função que pode ser utilizada ao invés da função atual.

- **NEXT:** É um campo que aponta para a próxima estrutura na lista de funções inseguras.

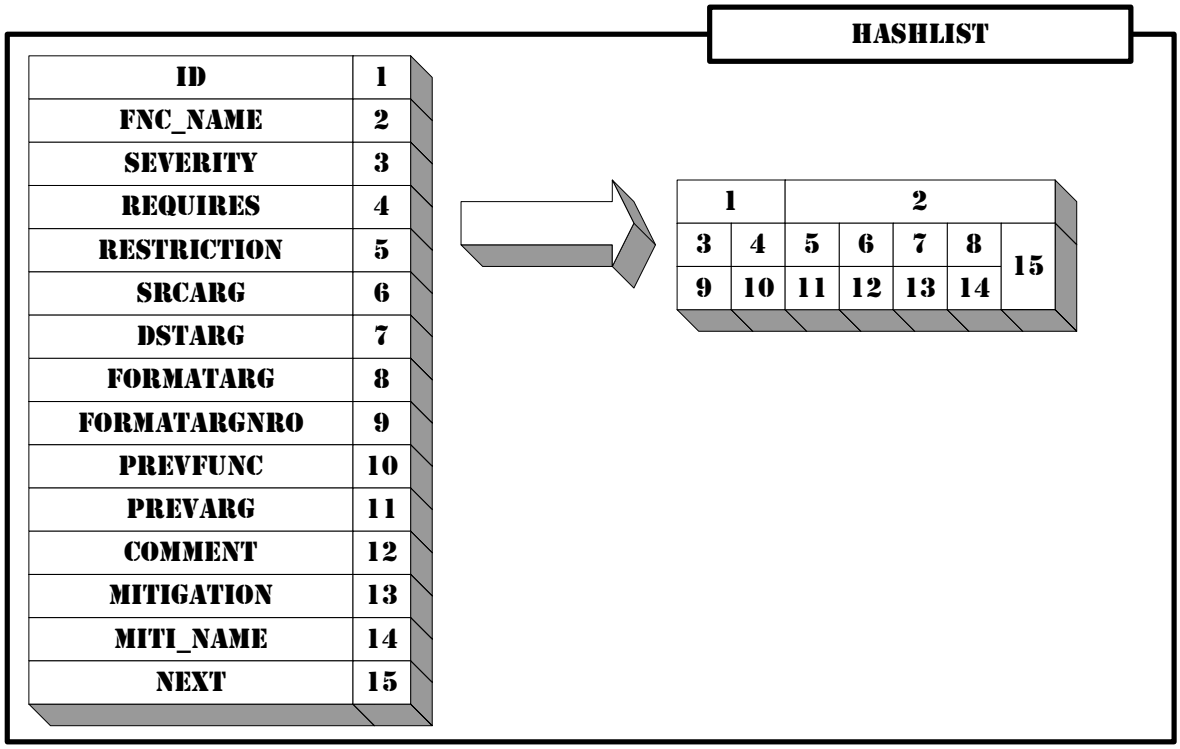

Figura 4.10 - Representação da estrutura `hashlist` que deverá ser preenchida no módulo de funções inseguras.

4.2.4 Módulo de Gerência e Controle

O módulo de gerência e controle percorre as estruturas criadas pelo módulo de análise sintática e, tomando por base os opções entradas pelo usuário, repassa ponteiros para funções ao módulo de análises de vulnerabilidades.

Este módulo também pode corrigir incoerências ocorridas no módulo de análise sintática. Para cada análise realizada, o módulo de controle pode receber um ponteiro para uma estrutura que armazena os possíveis problemas caso tenham sido encontrados. Caso existam problemas, estes são enviados para o módulo gerador de alertas e relatórios.

4.2.5 Módulo de Análises de Vulnerabilidades

O módulo de análises de vulnerabilidades recebe um ponteiro para a primeira função que deverá ser analisada. Então, as demais funções chamadas a partir da primeira são analisadas.

A princípio o objetivo é implementar a análise das funções para se identificar

chamadas a funções inseguras. Entretanto, outros tipos de análise também poderiam ser implementados, dadas as informações presentes nas estruturas apresentadas no módulo de análise sintática.

4.2.6 Módulo Gerador de Alertas e Relatórios

O módulo gerador de alertas e relatórios é responsável por enviar ao usuário quais foram as potenciais vulnerabilidades encontradas. A princípio, a saída deste módulo deverá ser em texto plano. Outras formas de saída poderiam ser implementadas, como saídas em `HTML` ou `TeX`.

4.3 Implementação de um Protótipo, `SCAP`

Para atestar a factibilidade do ambiente, um protótipo foi desenvolvido. Denominado *Source Code Analysis Prototype* este procura implementar as principais idéias presentes no ambiente. Dentre estas características encontram-se a implementação do módulo de análise léxica, do módulo de análise sintática, do módulo de funções inseguras, parte do módulo de gerência e controle, o módulo de análises de vulnerabilidades e o módulo gerador de alertas e relatórios.

Nesta seção serão abordadas as características de cada um dos módulos implementados do `SCAP`, mostrando detalhes do funcionamento interno de alguns dos módulos quando oportuno. No Apêndice A são mostradas as ferramentas utilizadas durante o desenvolvimento do protótipo.

4.3.1 Linguagem Fonte

Inicialmente, imaginou-se utilizar uma pseudo-gramática proposta por Louden em (Louden, 2004) chamada C-. O objetivo de utilizar esta linguagem era simplificar a implementação do módulo de análise léxica e sintática. Entretanto, poderia limitar os testes, não permitindo uma comparação mais real com outras ferramentas disponíveis na atualidade.

Optou-se, portanto, em se desenvolver um módulo de análise léxica para a linguagem ANSI C, que é uma linguagem largamente difundida e que permite que vulnerabilidades de *buffer overflow* ocorram.

4.3.2 Módulo de Análise Léxica

O desenvolvimento deste módulo foi baseado em um analisador léxico desenvolvido por Jeff Lee em Abril de 1985 para um rascunho da linguagem ANSI C. Este analisador léxico

foi difundido através de listas de discussões e foi reenviado para o *newsgroup* `net.sources` em 1987 por Tom Stockfisch. A menos dos códigos-fonte, a mensagem enviada para `net.sources` pode ser observada no Anexo A.

Por ser baseado no analisador léxico de Jeff Lee, o analisador desenvolvido foi escrito para a ferramenta `lex`. Apesar de parecer completo algumas novas marcas precisaram ser definidas, pois não estavam previstas instruções de pré-compilação como `#define`, `#ifdef` e etc.

Além de novas marcas, foi inserida uma rotina para verificar à qual linha a marca pertence. Para que, se necessário, uma mensagem de erro fosse gerada, mostrando a linha onde o erro ocorreu no código-fonte. Este erro gerado está relacionado a erros durante o processo de escrita do código-fonte, como a falta de um ';' no final de uma instrução.

Por vezes, é importante saber o lexema relacionado a uma dada marca, para que se possa, na análise sintática, preencher algumas estruturas. Nestes casos algumas modificações no analisador léxico foram implementadas para armazenar o lexema em variáveis que o analisador sintático consegue acessar.

O conjunto de todas as marcas utilizadas para a análise léxica podem ser observadas na Tabela 4.1.

Tabela 4.1 - Lista das marcas utilizadas no módulo de análise léxica.

GCC_DEFINE	GCC_INCLUDE	GCC_IFDEF	GCC_IFNDEF	'}'
GCC_ENDIF	AUTO	BREAK	CASE	','
CHAR	CONST	CONTINUE	DEFAULT	':'
DO	DOUBLE	ELSE	ENUM	'='
EXTERN	FLOAT	FOR	GOTO	'('
IF	INT	LONG	REGISTER	')'
RETURN	SHORT	SIGNED	SIZEOF	'['
STATIC	STRUCT	SWITCH	TYPEDEF	']'
UNION	UNSIGNED	VOID	VOLATILE	'.'
WHILE	CONSTANT	STRING_LITERAL	RIGHT_ASSIGN	'&'
LEFT_ASSIGN	ADD_ASSIGN	SUB_ASSIGN	MUL_ASSIGN	'!'
DIV_ASSIGN	MOD_ASSIGN	XOR_ASIGN	OR_ASSIGN	'~'
AND_ASSIGN	IDENTIFIER	TYPE_NAME	RIGHT_OP	'-'
LEFT_OP	INC_OP	DEC_OP	PTR_OP	'+'
AND_OP	OR_OP	LE_OP	GE_OP	'*'
NE_OP	'|'	'?'	';'	'{'
'/'	'%'	'<'	'>'	'^'

4.3.3 Módulo de Análise Sintática

A exemplo do módulo de análise léxica, pensou-se em desenvolver, no protótipo do ambiente, um módulo utilizando-se como linguagem alvo a linguagem C- dada a sua simplicidade. Entretanto, pelos mesmos motivos apontados anteriormente esta idéia foi deixada de lado.

A linguagem alvo utilizada foi então a linguagem ANSI C. Neste caso, o desenvolvimento do módulo de análise sintática também foi baseado no trabalho de Jeff Lee. Entretanto, grandes modificações precisaram ser realizadas.

No trabalho realizado por Jeff Lee, o objetivo do analisador sintático era validar um arquivo escrito em linguagem C. Nenhuma estrutura específica era criada para armazenar os dados obtidos. Este preenchimento é necessário no módulo de análise sintática proposto neste trabalho.

Da teoria de compiladores, parte do código binário pode ser diretamente obtida após a análise sintática. Ou seja, o código-executável já pode ser gerado durante esta etapa. Este fato fez com que não fosse interessante utilizar um analisador sintático presente em um compilador como o *GNU Compiler Compiler* (GCC)(GCC, 1987).

Como mostrado anteriormente, algumas estruturas precisam ser preenchidas para que o fluxo de execução e a análise estática possam ser realizadas de forma mais efetiva. Durante o desenvolvimento do módulo de análise sintática as principais estruturas puderam ser preenchidas de forma total ou parcial.

As estururas foram implementadas em linguagem C. Um exemplo da implementação de uma estrutura pode ser observada na Figura 4.11. Nesta Figura, é mostrada a implementação da estrutura `stt_function`. Implementações de outras estruturas podem ser observadas no Apêndice B.

A implementação da estrutura `stt_function` é muito importante, pois é justamente a partir do preenchimento desta estrutura que o fluxo de execução de um dado programa pode começar a ser montado. Por definição, os dados declarados globalmente são inseridos em uma estrutura do tipo `stt_function`, que possui o campo `name` definido como sendo "GLOBAL".

A partir da entrada "GLOBAL", são inseridas todas as demais funções do programa na ordem em que aparecem no arquivo contendo o código-fonte. É possivel notar na Figura 4.11 que uma lista duplamente encadeada é montada permitindo que qualquer função possa ser encontrada a partir da função atual.

```
#define NAMESIZE 60

typedef struct stt_function {
  char name[NAMESIZE];
  char file[NAMESIZE];
  unsigned int line;
  unsigned int status;
  struct stt_arg *argument;
  struct stt_var *variable;
  struct stt_act *action;
  struct stt_rest *restriction;
  struct stt_function *nextfunc;
  struct stt_function *prevfunc;
} stt_func;
```

Figura 4.11 - Implementação da estrutura `stt_function`.

A menos dos ponteiro `nextfunc` e `prevfunc`, existem quatro ponteiros definidos na estrutura apresentada na Figura 4.11. Para cada um deles existem listas duplamente encadeadas de estruturas. Por exemplo, uma dada função pode possuir vários argumentos que são montados em uma lista em separado. Estas estruturas são definidas de forma dinâmica a medida que as informações vão sendo obtidas pela análise sintática.

A Figura 4.12 ilustra um programa exemplo escrito no linguagem C. Neste existem três funções definidas: `sum` na linha 9, `copystr` na linha 14 e `copystr2` na linha 19. A função principal se inicia e tem como primeira ação a chamada a função `printf`, seguida das chamadas as funções `sum`, `copystr`, `printf`, `copystr2` e `printf`. As chamadas vulneráveis também estão marcadas no código-fonte.

No exemplo existem duas chamadas vulneráveis, que estão identificadas e definidas nas linhas 15 e 29. Durante a realização da análise sintática um conjunto as estruturas utilizadas neste módulo são preenchidas. O Apêndice C ilustra a saída do módulo de análise sintática para o código-fonte da Figura 4.12.

Para melhor visualizar como os dados são estruturadas na memória do sistema, uma representação gráfica dos dados retornados pelo módulo de análise sintática são sumarizados na Figura 4.13.

Na Figura 4.13 é possível identificar todas as seis ações tomadas na função principal `main`, bem como a ordem em que ocorrem. Se uma análise tiver início nesta função, as funções serão análisadas na ordem em que aparecem: `printf`, `sum`, `copystr`, `printf`, `copystr2`, `printf`.

```
/* ************************************ *
 * Programa para exemplo do preenchimento *
 * de um stack frame.                     *
 * ************************************ */
#define BUFFSIZE 10

char buff[BUFFSIZE];

int sum(int a, int b) {
  int c=a+b;
  return c;
}

void copystr(char *in, char *out) {
  strcpy(out,in); /* Vulneravel */
  return;
}

void copystr2(char *in, char *out, int size) {
  if ( strlen(in) < size) strcpy(out,in);
  return;
}

int main(int argc, char *argv[]) {
   int a;
   printf(":- Main Function Start\n");
   a=sum(1,2);
   copystr(argv[1],buff);
   printf(buff); /* Vulneravel */
   copystr2(argv[1],buff,BUFFSIZE);
   printf("\n%s\n",buff);

   return 0;
}
```

Figura 4.12 - Código-fonte, `ex1.c`, de exemplo com algumas vulnerabilidades.

Alguns campos não são preenchidos durante o processo de análise sintática realizado pelo protótipo, como os argumentos ou as restrições de uma dada função. Estes campos seriam importantes para análises de restrições, por exemplo. Entretanto, como este protótipo não realiza este tipo de análise o campo restrições não precisou ser preenchido.

4.3.4 Módulo de Funções Inseguras

O módulo de funções inseguras é responsável por obter funções sabidamente inseguras de um banco de funções e preencher estruturas previamente definidas. O correto preenchimento destas estruturas possibilita a análise de vulnerabilidades através da análise de funções inseguras. Esta análise busca por funções inseguras presentes nas estruturas sempre que uma chamada de função é realizada no código-fonte sob análise.

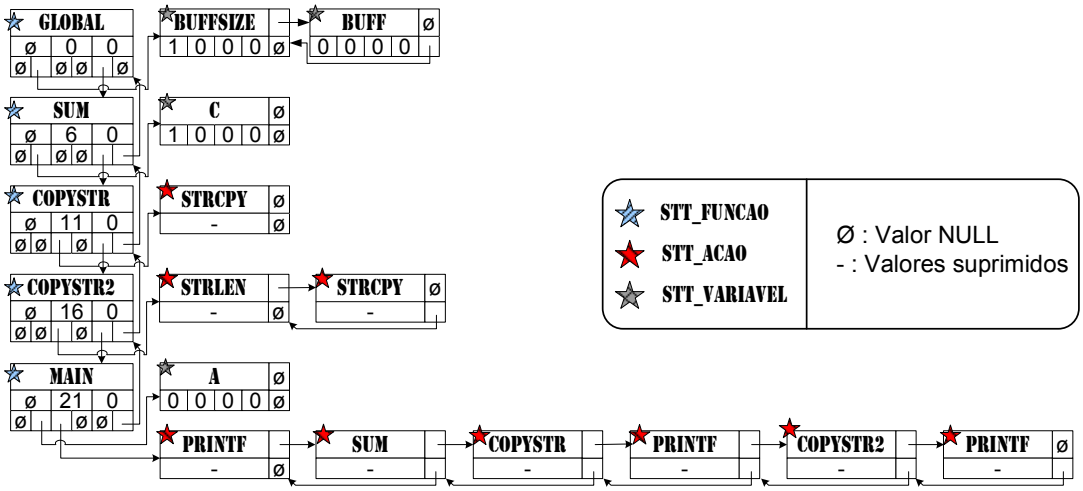

Figura 4.13 - Representação gráfica das estruturas geradas na análise sintática.

Esse banco de funções pode ser alterado à medida que novas funções inseguras são descobertas ou quando uma função já inserida precisa ser modificada. Portanto, o banco foi implementado de maneira a permitir estas modificações sem que fosse necessário, por exemplo, recompilar o módulo de funções inseguras.

Outro ponto importante é a facilidade na inserção de novas funções no banco. Um analista não diretamente relacionado ao desenvolvimento do ambiente deve poder inserir uma nova função sem que seja necessário entender o funcionamento interno do sistema. Para que isso fosse possível era importante que um padrão fosse definido para a base de dados de funções inseguras.

Para que o banco de funções possa ser alterado com relativa facilidade e sem a necessidade de recompilar o módulo de funções inseguras, pensou-se em algumas possiblidades. Uma seria utilizar um arquivo de texto convencional onde as funções vulneráveis fossem inseridas uma a uma em linhas demarcadas como ocorre no caso do `Snort`(Beale et al., 2004), que permite que cada uma das assinaturas do sistema detector de intrusão seja colocada em uma linha separada em um arquivo de texto. Um exemplo de entrada presente em um arquivo de regras do `Snort` pode ser observado na Figura 4.14.

Outra alternativa seria a utilização de um sistema gerenciador de banco de dados (SGBD) como `PostgreSQL` ou `MySQL`, cujo principal objetivo é gerenciar, manipular e organizar os dados presentes em um banco de dados. Além disso, utilização de um SGBD permitiria buscas mais eficientes no conteúdo armazenado. Entretanto, faria com que a máquina na qual a análise esta sendo realizada possuisse um SGBD instalado e configurado. Além de exigir o conhecimento, por parte do analista, da linguagem apropriada pra inserir ou alterar uma entrada presente no banco de dados.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 749 (msg:"EXPLOIT kadmind \
buffer overflow attempt"; flow:established,to_server; content:"|00 \
C0 05 08 00 C0 05 08 00 C0 05 08 00 C0 05 08|"; reference:bugtraq, \
5731; reference:bugtraq,6024; reference: cve,2002-1226; reference: \
cve,2002-1235; reference:url,www.kb.cert.org/vuls/id/875073; \
classtype:shellcode-detect; sid:1894; rev:8;)
```

Figura 4.14 - Exemplo de uma regra de detecção de intrusão presente no Snort.

Como mostrado anteriormente, o objetivo do banco de funções inseguras é preencher um conjunto de estruturas. Todas as buscas seriam então realizadas nestas estruturas. Portanto, um SGBD seria apenas utilizado para percorrer todas as entradas do banco retornando para cada uma das tuplas os dados de cada um dos campos.

Levando em consideração as duas alternativas, funções armazenadas em um arquivo texto ou funções armazenadas com o auxílio de um SGBD, optou-se por utilizar um arquivo texto. Os dados deste arquivo texto são estruturados como um arquivo *Extensible Markup Language* (XML). O XML é uma linguagem de marcação derivada do *Standard Generalized Markup Language* (SGML), definida pelo *World Wide Web Consortium* (W3C)[1] e permitiu a implementação de todas as características desejáveis.

A utilização de linguagens como o XML permitem que sejam realizadas descrições de diversos tipos de dados diferentes. Esta capacidade permite que as características das funções vulneráveis sejam definidas. A ferramenta `RATS`, apresentada em 3.3.4, também se utiliza da linguagem XML para agrupar as funções vulneráveis.

A forma como este módulo foi desenvolvido faz com que todas as funções armazenadas no banco sejam mantidas em memória, objetivando um melhor desempenho na busca por funções durante o processo de análise por vulnerabilidades. Apesar de não ser medido neste projeto, assume-se que o desempenho em se buscar uma informação em estruturas mantidas em memória seja melhor do que nos casos onde buscas são realizadas em arquivos armazenados em discos rígidos. É importante salientar que esta solução só é interessante para um conjunto limitado de funções, pois se o número de funções for excessivamente grande é inviável mantê-las em memória.

Um exemplo da representação em XML de uma função insegura pode ser observada na Figura 4.15. Nesta figura a entrada correspondente a função `gets` é mostrada.

[1]Maiores informações sobre o XML e o W3C podem ser encontradas em: <http://www.w3.org/>. Acesso em: 12 de Dez. de 2006

```
 <Vulnerability ID="01001">
   <Function>gets</Function>
   <Description>
     <Severity>0</Severity>
     <Requires>stdio.h</Requires>
     <Restriction>1st argument must be larger than the number of
 characters gets(); will read from stdin.</Restriction>
     <SrcArg>0</SrcArg>
     <DstArg>1</DstArg>
     <Comment>Reads a line from stdin into the buffer on the 1st
 argument until either a new line terminator or EOF is found,
wich it replaces with '\0'.</Comment>
   </Description>
   <Mitigation>
     <Info> The function gets(); reads a line from stdin that can
 be larger than the destination buffer. Use fgets(); instead.</Info>
     <MitiFunction>fgets();</MitiFunction>
   </Mitigation>
   <Dbinfo>
     <IncludedBy>L.O.D.</IncludedBy>
     <Date>08/08/2005</Date>
   </Dbinfo>
 </Vulnerability>
```

Figura 4.15 - Entrada da função `gets` no banco de funções inseguras.

4.3.4.1 Estruturas Utilizadas no Banco de Funções Inseguras

As funções inseguras mantidas em memória são armazenadas em uma lista simplesmente encadeada. Esta mantém relacionadas todas as informações obtidas a cerca de cada uma das função insegura definidas no banco de funções. A estrutura `hashlist` é definida no protótipo como na Figura 4.16.

Para cada função insegura presente no banco de funções XML uma estrutura do tipo `hashlist` é criada e preenchida. Esta estrutura possui um grupo de informações que são importantes do ponto de vista de segurança. Informações sobre como cada um dos campos pode ser preenchido podem ser observadas em 4.2.3.

Para cada uma das funções inseridas no banco de dados um inteiro é utilizado, partindo-se do número `01001` para que se possa distingüir uma função das demais presentes no banco de dados. O nome da função insegura é armazenado em `fnc_name`. É justamente através do nome da função que uma função é chamada em um código-fonte. Este nome também é utilizado durante o processo de análise para identificação de uma chamada a funções

```
typedef struct hashlist {
        int id;
        char *fnc_name;
        int severity;
        char *requires;
        char *restriction;
        int srcarg;
        int dstarg;
        int formatarg;
        int formatargnro;
        char *prevfunc;
        int prevarg;
        char *comment;
        char *mitigation;
        char *miti_name;
        struct hashlist *next;
} hashlist;
```

Figura 4.16 - Estrutura `hashlist`.

inseguras.

A severidade de uma função está relacionado com os problemas oriundos da exploração da vulnerabilidade no sistema afetado, levando em consideração os efeitos gerais causados no sistema. Para definir a severidade de uma função é utilizado um inteiro de quatro bytes que pode variar de '0' para as funções com menor severidade até '9' para as funções com maior severidade. Neste protótipo, porém, não foram definidas as severidades de cada uma das funções.

Os requisitos para utilização da função, como por exemplo as bibliotecas necessárias para a utilização da função ou o sistema operacional, são armazenados em uma variável `requires`. Estes requisitos são importantes durante o processo de análise pois podem permitir que o analista identifique se a função mostrada como insegura realmente é a função que o programador utilizou durante o processo de codificação.

Muitas funções inseguras não são vulneráveis incondicionalmente. Ou seja, independentemente da forma como foi utilizada ela é vulnerável. Portanto, também é mapeada na estrutura a forma como a função pode ser utilizada de forma segura. Este campo é definido como `restriction` também possui um tamanho variável e dinamicamente alocado. Um exemplo pode ser observado na Figura 4.17, que ilustra o campo `restriction` para a função `strcat`[2]

[2]A função `strcat` concatena o segundo parâmetro passado para função no primeiro. Ou seja, no primeiro parâmetro são armazenados todos os caracteres do primeiro mais todos os caracteres do segundo

```
--
1st argument must be larger than the current size of the 1st argument
plus size of the 2nd argument.
--
```

Figura 4.17 - Exemplo do campo `restriction` para a função `strcat`.

As funções inseguras implementadas em bibliotecas do sistema geralmente obtêm os dados de origem de uma fonte bem definida e os armazena em um destino também definido. A fonte dos dados geralmente é um dos argumentos passados para a função, entretanto, pode ser a entrada padrão, `STDIN`, uma variável global e etc. Estas informações de fonte e destino dos dados são mapeados em duas variáveis `srcarg` e `dstarg`.

O inteiro `srcarg` define a fonte dos dados que são utilizados para extravasar a cadeia de caracteres de destino `dstarg`. Como a fonte dos dados podem não ser argumentos, valores superiores ou iguais a um são reservados aos argumentos. Ou seja, o número um representa o primeiro argumento. O número dois o segundo argumento e assim sucessivamente. Para casos onde a fonte dos dados não está marcada como um argumento este valor é mantido como '`0`'.

De forma similar ao que ocorre com a variável `srcarg`, a variável `dstarg` define qual o argumento de destino. Valores superiores ou iguais a um definem argumentos passados e o valor zero define outros casos como retorno de uma função ou `STDOUT` por exemplo.

O campo `formatarg` pode ser utilizado para identificar que uma função pode ser utilizada de forma segura se possuir um *format string*. Neste caso esta entrada é definida como '`1`'. Se o *format string* precisar possuir um *length modifier* para ser segura, este campo é definido como '`2`'. Quando o argumento em específico precisa ser analisado, o campo `formatargnro` é utilizado.

Por vezes, verificações sobre o tamanho dos argumentos são realizadas antes da chamada da função insegura. Nestes casos a chamada pode não ser vulnerável. Para endereçar estes casos, duas variáveis são definidas `prevfunc` e `prevarg`. No primeiro caso, o nome da função que deve ser chamada para verificar as entradas é especificado. No segundo caso, é definido qual o argumento que deve ser verificado antes da chamada da função.

Uma descrição sucinta do funcionamento da função é armazenada na variável `comment`. Esta descrição pode auxiliar o analista na comprovação dos dados retornados pelo

parâmetro.

ambiente de análises. Um exemplo de como este campo seria preenchido pode ser observado na Figura 4.18, que mostra o campo `comment` da função `strcat`.

```
--
This function appends the 2nd argument (including the terminating
\0 character) to the array pointed by the 1st argument.
--
```

Figura 4.18 - Exemplo do campo `comment` para a função `strcat`.

Para cada função insegura identificada, as formas de utilização segura são armazenadas em `restriction`. Entretanto, podem existir outras funções que eliminam ou ao menos minimizam potenciais problemas. Estas informações são armazenadas no campo *mitigation* da estrutura `hashlist`. Para o caso da função `strcat` o campo `mitigation` seria preenchido como mostrado na Figura 4.19.

```
--
Be sure that the size of the first function argument is large enough to
support the size of the first argument plus second one. Use strncat();
with proper arguments instead.
--
```

Figura 4.19 - Exemplo do campo `mitigation` para a função `strcat`.

A sugestão de uma função menos insegura que pode ser utilizada é armazenada no campo `miti_name`. No caso da função `strcat`, este campo simplesmente conteria o nome da função `strncat`. Este campo também tem tamanho variável e é dinamicamente alocado.

Por fim, a estrutura `hashlist` possui um ponteiro para outra estrutura `hashlist` que seria a próxima função indexada. Desta forma, uma lista de estruturas é montada permitindo-se percorrer esta lista pra buscar dada função.

Este módulo foi implementado utilizando-se a linguagem `C` com auxílio da biblioteca `expat` para interpretação do banco de dados escrito em XML.

4.3.4.2 Utilização de Tabelas Hash

Segundo Cormen et al. (2001), para a implementação de dicionários a tabela *hash* se mostra como uma estrutura de dados eficiente. Dicionários são conjuntos dinâmicos que

permitam apenas a execução de funções de inserção, remoção e busca (Cormen et al., 2001). Estas operações são as únicas necessárias no banco de funções desenvolvido.

A implementação de uma tabela *hash* pode melhorar o desempenho da busca. Sendo que o pior caso seria quando todas as chaves, que neste módulo são os nomes das funções inseguras, possuissem o *hash* para a mesma posição. Neste caso, todas as entradas podem precisar ser pecorridas para que uma em particular seja encontrada.

Neste protótipo foi implementada uma tabela *hash* com resolução de colisões por encadeamento. Ou seja, chaves com o mesmo valor *hash* são inseridas no início de uma lista simples. A função *hash* é definida pelo método da divisão utilizando-se as instruções ilustradas no código da Figura 4.20.

```
1 for(; *str != '\0'; str++) hashvalue = *str + 2 * hashvalue;
2 return hashvalue % hasht->size;
```

Figura 4.20 - Cálculo do *hash* para o nome de uma função.

Na primeira linha do código exposto na Figura 4.20 é definida a chave utilizada para cada uma das funções inseguras inseridas. Na segunda linha é definido o *hash* propriamente dito que é o resto da divisão da chave pelo tamanho da tabela *hash*.

Neste protótipo foi utilizada uma tabela *hash* de 83 posições. Este número foi escolhido entre os números primos maiores do que 70 e que fosse não próximo à uma potência de 2. O número 70, é resultado da divisão de 140 por 2, que foi definido como fator de carga para a tabela hash. O fator de carga é o número médio esperado para as colisões em cada uma das posições da tabela hash.

4.3.4.3 Funções Presentes

As funções colecionadas no banco de funções inseguras somam mais de 140 entradas. Apesar de não ser exaustivo, este banco de funções agrega as funções inseguras mais utilizadas. O Apêndice E mostra uma representação das funções do banco organizadas em uma tabela hash.

4.3.5 Módulo de Gerência e Controle

No prótotipo desenvolvido, o módulo de gerência e controle é responsável por iniciar a tabela hash com as 83 posições e realizar as chamadas necessárias para que o módulo de funções inseguras obtenha os dados do banco de funções inseguras. Também é parte

das atividades realizadas pelo módulo de gerência e controle iniciar o módulo de análise sintática, que se encarrega de realizar as chamadas necessárias para o módulo de análise léxica.

O módulo de gerência e controle então obtém do módulo de funções inseguras um ponteiro para a tabela hash devidamente preenchida e recebe do módulo de análise sintática um ponteiro para a estrutura que descreve as informações que puderam ser obtidas a partir da análise do código-font-fontee. Estes dois ponteiros então são passados para o módulo de análises de vulnerabilidades que deve definir quais entradas da estrutura montada pelo módulo de análise sintática é ou não vulnerabilidade.

Neste protótipo não são levadas em consideração opções passadas por usuários. Ou seja, a verificação é realizada em todo o código-fonte passado para o protótipo. Entretanto, as análises poderiam ser realizadas em funções específicas, módulos ou em todo o código-fonte bastando que um subconjunto das estruturas enviadas pelo módulo de análise sintática fossem enviados para o módulo de análise de vulnerabilidades.

4.3.6 Módulo de Análises de Vulnerabilidades

O módulo de análises de vulnerabilidades recebe do módulo de gerência e controle duas estruturas. Uma é a estrutura representando as informações obtidas pelo módulo de análise sintática do código-fonte do programa. A outra é a estrutura preenchida com os dados das funções inseguras.

Para realizar a análise de vulnerabilidade através da busca por funções inseguras, o módulo percorre a estrutura obtida do código-fonte, função após função, identificando quando ações foram realizadas. Estas ações, como visto anteriormente, são chamadas a outras funções.

A análise é realizada a partir da primeira função encontrada na estrutura do código-fonte. Busca-se nesta função todas as ações que foram realizadas e para cada uma delas é identificado se a função foi implementada pelo desenvolvedor do programa ou se a função é realizada para uma biblioteca externa.

Caso a chamada de função seja uma requisição à uma função externa, esta é buscada no banco de funções inseguras. Ou seja, dado que o desenvolvedor não implementou a função da ação realizada é importante identificar se esta função está ou não presente no banco de funções.

A princípio quando a ação é identificada no banco de funções inseguras a ação é marcada

como vulnerável. Entretanto, esta abordagem gera um grande número de falsos positivos, pois não leva em consideração se a função insegura foi utilizada de forma segura. Portanto, um conjunto de checagens deve ser realizado para que um alerta não seja emitido desnecessariamente. O conjunto de testes realizados quando uma ação é encontrada no banco de dados de funções inseguras pode ser observado na Figura 4.21

A Figura 4.21 ilustra as dez verificações básicas realizadas pelo módulo de análises de vulnerabilidades. A primeira verificação está ligada à existência ou não de ações na função que esta sendo analisada. Caso estas ações não existam uma nova função é tomada para análise.

A segunda verificação ocorre sobre cada uma das ações realizadas na função sob análise. Como cada ação corresponde a uma chamada de função[3], é verificado se esta função da ação foi ou não implementada pelo desenvolvedor do programa. Caso tenha sido desenvolvida pelo desenvolvedor, uma nova ação é tomada para análise.

Caso a função da ação tenha sido implementada por terceiros, existe a possibilidade desta estar presente no banco de funções inseguras. Portanto, a terceira verificação está relacionada a identificar se a função da ação está ou não no banco de funções inseguras. Caso não esteja presente no banco de funções, uma nova ação é tomada para análise.

Se a função da ação estiver presente no banco de funções inseguras, uma quarta verificação é realizada. Nesta, é identificado se a função da ação pode ou não ser executada de forma segura dependendo dos parâmetros utilizados para a sua chamada. Mais especificamente, é verificado se um *format string* pode tornar a chamada segura. Caso um *format string* puder tornar a chamada segura, uma quinta verificação é realizada.

Na quinta verificação é identificado se a simples presença de um *format string* em um determinado parâmetro torna a chamada segura. Em caso afirmativo uma sexta verificação é realizada para buscar por este *format string* na ação que esta sendo analisada. Caso um simples parâmetro não torne a chamada segura ou se o *format string* não for encontrado na sexta verificação, uma sétima verificação é realizada.

Nesta sétima verificação o módulo identifica se a presença de uma *format string* com *length modifier* pode tornar a execução da chamada segura. Se puder, uma oitava verificação que busca por este tipo de *format string* na chamada da ação é realizada. Se a *format string* não for encontrada ou se anteriormente, na verificação, uma *format string* não tornar a função da ação segura, uma nona verificação é realizada, buscando identificar se uma

[3]Por cada ação ser uma chamada a uma função, a função relacionada à uma ação será chamada de função da ação.

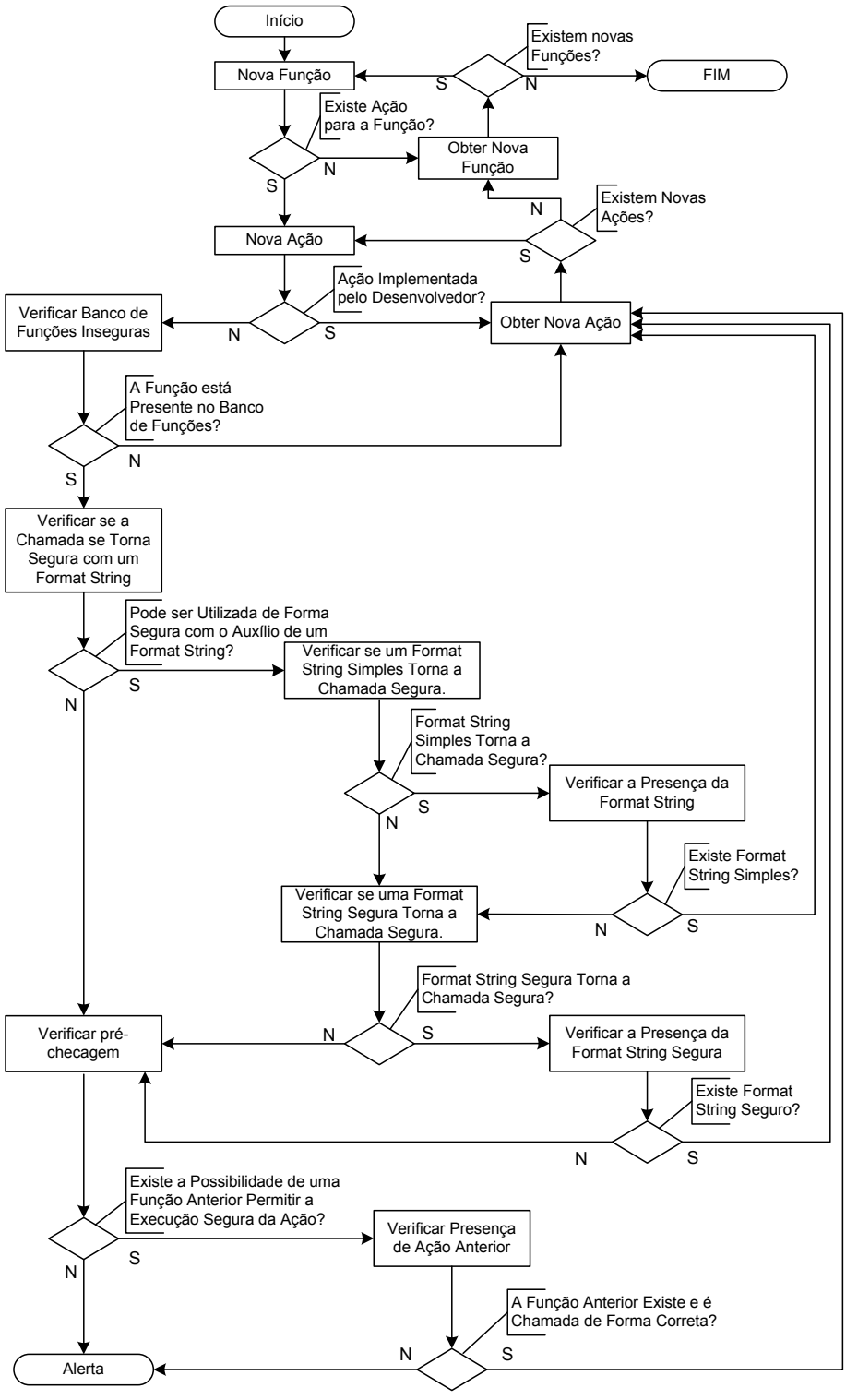

Figura 4.21 - Fluxograma do módulo de análises de vulnerabilidades.

checagem anterior, realizada em ações anteriores, pode tornar a chamada segura.

Se existir a possibilidade de chamadas anteriores uma décima verificação é realizada, buscando justamente estas chamadas dentro das ações já executadas pela função da ação analisada. Caso não sejam encontradas as ações necessárias uma marcação é realizada na ação mostrando que esta foi identificada como sendo potencialmente vulnerável.

Tomando por base o exemplo descrito em 4.12, algumas modificações são realizadas pelo módulo de análises de vulnerabilidades para que sejam marcadas quais vulnerabilidades foram de fato identificadas. Estas modificações podem ser observadas no Apêndice F. Neste apêndice, as ações identificadas como vulneráveis são marcadas com '!'.

4.3.7 Módulo Gerador de Alertas e Relatórios

Uma vez as possíveis vulnerabilidades tendo sido identificadas no módulo de análises de vulnerabilidades, o módulo de gerencia e controle repassa as estruturas antes enviadas para o módulo de análises de vulnerabilidades para o módulo gerador de alertas e relatórios.

Neste protótipo, o objetivo do módulo de alertas e relatórios é obter do banco de funções inseguras as informações pertinentes à uma dada função quando esta é identificada no código-fonte como sendo vulnerável.

Para o exemplo da Figura 4.12 o relatório gerado por este módulo pode ser observado na Figura 4.22, que deve ser a saída final do protótipo para a análise realizada.

```
:- Function: copystr
:-   Action: (strcpy*) (15)
:-   x Comment: This function copies the string of the 2nd argument (including
 the terminating '\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is
larger enough to support the size of the second argument. Use strncpy(); with
proper arguments instead.
:-   x Mitigation Function: strncpy();

:- Function: main
:-   Action: (printf*) (29)
:-   x Comment: This function writes informations to STDOUT according to the
1st argument format string.
:-   x Mitigation: This function can allow a Format String exploitation.
Please, define the format string.
```

Figura 4.22 - Relatório gerado a partir da análise do programa.

4.4 Considerações Finais

Neste capítulo o ambiente para análise de códigos-fonte com ênfase em segurança foi abordado, mostrando a viabilidade deste ambiente através da implementação de um protótipo.

O ambiente é composto de inúmeros módulos com objetivos e funcionalidades distintas, sendo que as principais características de cada um dos módulos do ambiente foram descritas. Sucintamente, existem módulos que trabalham diretamente com o código-fonte da aplicação que deve ser analisada, um módulo que trabalha diretamente com um banco `XML` de funções inseguras e módulos que trabalham sobre estruturas preenchidas pelos demais.

Para implementar um protótipo do ambiente proposto um banco de funções inseguras precisou ser montado com as características descritas nas seções anteriores. As funções presentes foram extraídas de bancos de outras ferramentas, livros e outras referências.

Alguns códigos-fonte simples, criados expecificamente para testes, foram utilizados para que o módulo de análise léxica e sintática pudessem ser desenvolvidos. Apesar de todos os esforços no sentido de se gerar um analisador sintático genérico, muitos códigos fontes não conseguem passar pela etapa de *parsing* dado o analisador sintático implementado. Entretanto, para os testes necessários o ambiente conseguiu atuar da forma esperada.

Algumas checagens adicionais à simples busca por funções na estrutura montada pelo módulo de funções inseguras foram realizadas na implementação do módulo de análises de vulnerabilidades. Estas checagens visam diminuir o número de falsos positivos retornados pela ferramenta. Como será abordado posteriormente estas checagens conseguiram reduzir sobremaneira esse tipo de alerta.

O módulo gerador de alertas e relatórios simplesmente percorre as estruturas montadas a partir do código-fonte mostrando, quando necessário, informações a cerca de funções inseguras.

CAPÍTULO 5

TESTES E RESULTADOS

A partir do ambiente proposto um protótipo foi desenvolvido com o objetivo de comprovar a possibilidade de implementação do ambiente. Uma vez obtendo um protótipo funcional, um conjunto de testes foi realizado para verificar a eficácia deste frente a outras ferramentas.

Foram realizados três testes distintos. O primeiro deles sobre um código-fonte proposto por José Nazario na revista digital `Linux Journal`. O segundo teste foi realizado com um código-fonte desenvolvido pelos mantenedores da ferramenta `Flawfinder`. Por fim, o teste sugerido em Wilander e Kamkar (2002) foi efetuado.

5.1 Testes de José Nazario

José Nazario utiliza um código-fonte retirado da ferramenta OpenLDAP para a realização de seus testes. Um trecho do arquivo `openldap-2.0.11/libraries/libldap/print.c` da versão 2.0.11, que foi escolhido por Nazario, pode ser observado na Figura 5.1

```
35 int ldap_log_printf( LDAP *ld, int loglvl, const char *fmt, ... )
36 {
37         char buf[ 1024 ];
38         va_list ap;
39
40         if ( !ldap_log_check( ld, loglvl )) {
41                 return 0;
42         }
43
44         va_start( ap, fmt );
45
46 #ifdef HAVE_VSNPRINTF
47         buf[sizeof(buf) - 1] = '\0';
48         vsnprintf( buf, sizeof(buf)-1, fmt, ap );
49 #elif HAVE_VSPRINTF
50         vsprintf( buf, fmt, ap ); /* hope it's not too long */
51 #else
52         /* use doprnt() */
53         chokeme = "choke me! I don't have a doprnt manual handy!";
54 #endif
55
56         va_end(ap);
57
58         (*ber_pvt_log_print)( buf );
59         return 1;
60 }
```

Figura 5.1 - Trecho do código-fonte `print.c` do `OpenLDAP` versão 2.0.11.

Nos testes de Nazario, o arquivo `print.c` completo foi analisado pelas ferramentas `Flawfinder`, `ITS4` e `RATS`. [1] O objetivo do teste é identificar se a utilização de ferramentas de auxílio é útil durante os esforços de escrita de melhores códigos.

De forma similar a realizada por Nazario o código-fonte `print.c` foi analisado pelo `SCAP`. Internamente, os módulos de análise léxica, sintática, funções inseguras e análises de vulnerabilidades são executados. O resultado da análise realizada pelo módulo de análises de vulnerabilidades pode ser observado na Figura 5.2.

```
:- FUNCAO: GLOBAL
:-  File:
:-  Line: 0
:-  Status: 0

:- FUNCAO: ldap_log_check
:-  File:
:-  Line: 22
:-  Status: 0
:-  Variables:
:-   VarName: (errlvl) T: 0 A: 0 U: 0 S: 0

:- FUNCAO: ldap_log_printf
:-  File:
:-  Line: 35
:-  Status: 0
:-  Variables:
:-   VarName: (buf) T: 0 A: 0 U: 0 S: 0
:-   VarName: (ap) T: 0 A: 0 U: 0 S: 0
:-  Actions:
:-   Action: (ldap_log_check)
:-    Parameters: (ld) (loglvl)
:-   Action: (va_start)
:-    Parameters: (ap) (fmt)
:-   Action: (sizeof)
:-    Parameters: (buf)
:-   Action: (sizeof)
:-    Parameters: (buf)
:-   Action: (vsnprintf)(*F1!)(48)
:-    Parameters: (buf) ((PREV)) ((CONSTANT)) (fmt) (ap)
:-   Action: (vsprintf)(*F2!P1!)(50)
:-    Parameters: (buf) (fmt) (ap)
:-   Action: (va_end)
:-    Parameters: (ap)
```

Figura 5.2 - Resultados do módulo de análises de vulnerabilidades para o teste de Nazario.

O módulo de análises de vulnerabilidades pode executar um conjunto de checagens para

[1] O artigo completo discorrendo sobre o teste de Nazario pode ser encontrado em: <http://www.linuxjournal.com/article/5673>. Acesso em: 12 de Dez. de 2006.

cada uma das ações presentes em um código-fonte. Estas checagens premitem definir se uma dada chamada é vulnerável ou não. A princípio se uma dada função da ação está presente no banco de dados de funções inseguras e não requer nenhuma checagem ela é marcada como vulnerável.

Caso possam existir checagens, estas são realizadas. Caso uma checagem não consiga definir se a chamada é segura, uma marca '`!`' é emitida. Por outro lado, se a checagem verificar as condições necessárias para a função poder executar, nenhuma marca é emitida.

Na Figura 5.2 observa-se que dois problemas foram identificados pelo `SCAP`. O primeiro localizado na linha 48 e o segundo localizado na linha 50.

No caso do problema da linha 48 do código-fonte, a resposta das checagens foi `(*F1!)`. A presença do '`*`' significa que a função da ação foi encontrada no banco de funções inseguras. O '`F`' simboliza que para esta chamada ser segura é preciso um *format string* simples, '`1`'. Como este *format string* não foi encontrado, uma possível vulnerabilidade, '`!`', foi identificada. É importante frisar que esta vulnerabilidade não está relacionada a problemas de *buffer overflow*, pois o segundo argumento da ação permite a cópia. Entretanto, uma vulnerabilidade de *format string* pode ocorrer.

Para o problema reconhecido na linha 50, duas checagens, `(*F2!P1!)`, foram realizadas uma para verificar se um *format string* com *length modifier* foi definido '`F2`' e outra para verificar se existem funções anteriores que comprovem o tamanho da variável de origem '*P1*'. Neste caso, nenhum *format string* com *length modifier* foi identificado e nenhuma função anterior para checagem do tamanho do *buffer* de origem. Portanto, esta função permite que uma vulnerabilidade de *buffer overflow* ocorra.

Um analista utilizando o `SCAP` deve obter, para o código-fonte da Figura 5.1, o relatório exposto na Figura 5.3.

José Nazario, checa o código-fonte `print.c` utilizando três ferramentas: `RATS`, `Flawfinder` e `ITS4`. Os resultados obtidos pelas três ferramentas, mais o resultado obtido pelo `SCAP` são sumarizados na Tabela 5.1

Tabela 5.1 - Resultados obtidos para o teste de José Nazario.

Função Insegura	Flawfinder		ITS4		RATS		SCAP	
	True	False	True	False	True	False	True	False
vsnprintf()	1	-	1	-	1	-	1	-
vsprintf()	1	-	1	-	1	-	1	-

```
:- Function: ldap_log_printf
:-   Action: (vsnprintf*) (48)
:-   x Comment: This function writes the output under the control of a format string
to a string pointed in the 3rd argument.
:-   x Mitigation: Be sure to use a secure format string as the 3rd argument. In
case of strings, use length modifiers.

:- Function: ldap_log_printf
:-   Action: (vsprintf*) (50)
:-   x Comment: This function writes the output under the control of a format string
to a string pointed in the 1st argument.
:-   x Mitigation: Be sure to use a secure format string as the 2nd argument. In
case of strings, use length modifiers.
:-   x Mitigation Function: vsnprintf
```

Figura 5.3 - Saída do protótipo SCAP para `print.c`.

Na Tabela 5.1 as colunas '`True`' representam os verdadeiros positivos encontrados pela ferramenta. Para cada função, é preciso encontrar um verdadeiro positivo. As colunas '`False`' representam os falsos positivos que no arquivo `print.c` não existem. Como pode ser observado, o `SCAP` teve desempenho equivalente ao das outras ferramentas no que diz respeito a identificação de chamadas inseguras presentes no código em questão.

5.2 Testes do Flawfinder

Na página *web* oficial da ferramenta `Flawfinder`, mantida por David Wheeler, existe um código-fonte com um grande número de vulnerabilidades. Este é utilizado para testar o `Flawfinder` e mostrar como algumas vulnerabilidades podem ser encontradas.

Da mesma forma que a utilizada no caso do `Flawfinder` este código-fonte pode ser utilizado pelo `SCAP`. Os testes realizados objetivaram identificar o quão efetivo o `SCAP` é com relação a identificação dos reais problemas presentes no exemplo descrito. O código-fonte deste teste pode ser observado no Anexo B.

No Apêndice G são mostrados respectivamente a organização das estruturas obtidas do código-fonte depois que estas passam pelo módulo de análises de vulnerabilidades e a saída gerada para o analista. Na Tabela 5.2 são sintetizados os resultados do `SCAP` ao analisar o código citado.

Na Tabela 5.2 a coluna '`True`' representa os verdadeiros positivos. Neste teste, o número de verdadeiros positivos dentro do código-fonte é mostrado após a '`/`'. Por exemplo, a função `strcpy` é chamada de forma insegura duas vezes no código. Os resultados obtidos pela ferramenta `SCAP` são inseridos antes do simbolo '`/`'. A coluna '`False`' identifica os falsos

Tabela 5.2 - Resultados obtidos para o teste do `Flawfinder`.

Função Insegura	SCAP	
	True	False
printf()	1/1	0/12
strcpy()	2/2	0/1
sprintf()	3/3	0/2
scanf()	2/2	0/2
gets()	3/3	-/-
syslog()	1/1	0/2
_mbscpy()	1/1	0/0
memcpy()	1/1	0/0
CopyMemory()	1/1	0/0
lstrcat()	1/1	0/0
strncpy()	1/1	0/0
_tcsncpy()	1/1	0/0
strncat()	2/2	0/0
_tcsncat()	1/1	0/0
strlen()	0/1	0/0
MultiByteToWideChar()	0/2	0/2
getopt_long	1/1	0/0

positivos presentes no código, que são chamadas legítmas e que não devem ser reportadas pela ferramenta.

Para este teste, a ferramenta `SCAP` retornou 22 chamadas inseguras, das 25 presentes no código e que podem causar um problema de *buffer overflow*. As três vulnerabilidades não encontradas dependem de um único fator, as funções `strlen` e `MultiByteToWideChar` não estão presentes no banco de funções inseguras do `SCAP`.

Para os casos de falsos positivos, o `SCAP` não retornou nenhum para as 21 possibilidades apresentadas. Este resultado é interessante pois a ferramenta `Flawfinder`, por utilizar uma abordagem diferente, acaba reportando grande parte dos falsos positivos, por vezes definindo um risco mais baixo.

5.3 Testes de Wilander e Kamkar

Wilander e Kamkar (2002) realizaram um estudo aprofundado para comparar, de forma empírica, ferramentas publicamente disponíveis que auxiliam o processo de análise estática de códigos.

Cinco ferramentas foram testadas com um código-fonte desenvolvido pelos próprios autores. Sendo que chamadas seguras e vulneráveis foram realizadas para a grande maioria

das funções presentes no código. O objetivo era aferir o número de verdadeiros e falsos positivos retornados por cada uma das ferramentas.

O código proposto que pode ser observado no Anexo C foi utilizado na ferramenta `SCAP` aos moldes do que foi realizado com as ferramentas: `Flawfinder`, `ITS4`, `RATS`, `Splint` e `BOON`.

A organização das estruturas obtidas do código-fonte depois que estas passam pelo módulo de análises de vulnerabilidades e a saída gerada para o analista são mostrados no Apêndice H. Os resultados sintetizados das cinco ferramentas em conjunto com os resultados do `SCAP` podem ser observados na Tabela 5.3.

Tabela 5.3 - Resultados obtidos para o teste de Wilander.

Função Insegura	Flawfinder		ITS4		RATS		Splint		BOON		SCAP	
	True	False	True	False	True	False	True	False	True	False	True	False
gets()	1	-	1	-	1	-	1	-	1	-	1	-
scanf()	1	0	1	0	1	1	0	0	0	0	1	0
fscanf()	1	0	1	0	1	1	0	0	0	0	1	0
sscanf()	1	0	1	0	1	1	0	0	0	0	1	0
vscanf()	1	0	1	0	1	1	0	0	0	0	1	0
vsscanf()	1	0	1	0	1	1	0	0	0	0	1	0
vfscanf()	1	0	1	0	1	1	0	0	0	0	1	0
cuserid()	0	-	1	-	1	-	0	-	0	-	1	-
sprintf()	1	1	1	0	1	1	0	0	1	1	1	0
strcat()	1	1	1	1	1	1	1	0	1	1	1	0
strcpy()	1	1	1	1	1	1	1	0	1	1	1	0
streadd()	1	1	1	1	1	0	0	0	0	0	1	0
strecpy()	1	1	1	1	1	0	0	0	0	0	1	0
vsprintf()	1	1	1	0	1	1	1	1	0	0	1	0
strtrns()	1	1	1	1	1	0	0	0	0	0	1	0
printf()	1	1	1	1	1	1	1	1	-	-	1	1
fprintf()	1	1	1	1	1	1	1	1	-	-	1	1
sprintf()	1	1	1	1	1	1	1	1	-	-	1	1
snprintf()	1	1	1	1	0	0	0	0	-	-	1	1
vprintf()	1	1	0	0	0	0	0	0	-	-	1	1
vfprintf()	1	1	0	0	0	0	0	0	-	-	1	1
vsprintf()	1	1	1	1	1	0	0	0	-	-	1	1
vsnprintf()	1	1	1	1	0	0	0	0	-	-	1	1

Na Tabela 5.3, as funções inseguras até `strtrns` devem ser analisadas quanto a possiblidade de *buffer overflow* já as demais funções devem sofrer análise de *format string*. Neste teste a ferramenta `SCAP` obteve bons resultados, pois todas as vulnerabilidades de

buffer overflow puderam ser identificadas, como pode ser observado na coluna '`True`'. Entretanto, as ferramentas `ITS4` e `RATS` obtiveram os mesmos resultados.

Quanto ao número de falsos positivos, a ferramenta `SCAP` não retornou nenhum falso positivo, enquanto todas as outras ferramentas identificaram pelo menos uma chamada vulnerável que era inofensiva. Neste teste, portanto, a ferramenta `SCAP` obteve as melhores taxas de verdadeiros positivos e falso positivos.

5.4 Considerações Finais

Os três testes realizados foram muito úteis para comprovar a possibilidade do emprego do ambiente proposto durante a realização dos trabalhos de análise estática de códigos-fonte. Cada um dos testes abordados possui peculiaridades diferentes.

Nos testes de José Nazário, um código-fonte real com vulnerabilidades foi utilizado para verificar a eficácia das ferramentas através da análise de códigos-fonte de aplicações existentes e atualmente disponíveis. Para este teste a ferramenta `SCAP` conseguiu identificar dois possíveis pontos vulneráveis, os únicos existentes.

Nos testes do `Flawfinder` o objetivo era mostrar exemplos de saídas do `Flawfinder`. Existiam conjuntos de chamadas vulneráveis e de chamadas não vulneráveis. O código não foi escrito para ser executado, portanto, algumas modificações tiveram que ser realizadas para que este fosse sintaticamente compatível com a linguagem ANSI C. A ferramenta `SCAP` obteve bons resultados neste teste, mesmo que os resultados não tenham sido comparados com os de outras ferramentas. Entretanto, expôs um problema presente em todas as ferramentas que auxiliam a análise estática de códigos-fonte. Estas podem analisar somente as chamadas cuja função está presente no banco de funções inseguras. Justamente por não possuir as funções `strlen` e `MultiByteToWideChar` no banco de funções inseguras, a ferramenta `SCAP` não gerou alertas para as chamadas.

Os testes realizados em Wilander e Kamkar (2002) são os mais interessantes do ponto de vista da comparação das ferramentas atualmente disponíveis. Em um código-fonte com chamadas seguras e inseguras para a mesma função, as checagens realizadas por cada uma das ferramentas puderam ser melhor estudadas. No reconhecimento das chamadas inseguras que expunham um *buffer overflow* a ferramenta `SCAP` obteve os melhores resultados. Entretanto, também ficou evidente que a ferramenta não foi desenvolvida para minimizar o número de falsos positivos na identificação de *format strings*.

CAPÍTULO 6

CONCLUSÃO

6.1 Considerações

No segundo capítulo foram discutidos alguns dos problemas mais comuns encontrados em programas e que podem ter alguma implicação em segurança. Foram vistas vulnerabilidades como *format string*, *race condition*, *code injection*, *symbolic link* e *buffer overflow*, que são as vulnerabilidades em maior evidência na atualidade. Dentre todas estas vulnerabilidades, porém, o *buffer overflow* é a de maior destaque.

Pôde-se observar, também no segundo capítulo, que existem mais do que uma maneira de se explorar a vulnerabilidade exposta por um *buffer overflow*. Dentre estas maneiras destacaram-se a exploração por *stack smash*, por *arc injection* e *pointer subterfuge*. Entretanto, estas podem não ser as únicas formas de exploração existentes, podendo existir outras não amplamente divulgadas.

Também não são poucas as formas de se tentar eliminar as vulnerabilidades de *buffer overflow*, como mostrado no terceiro capítulo. Algumas abordagens tentam prevenir os impactos oriundos da exploração da vulnerabilidade, outras tentam de uma forma preventiva eliminar os problemas no próprio código-fonte.

As alternativas chamadas dependentes do compilador, geralmente modificam o prólogo ou o epílogo das chamadas de funções no próprio compilador. Algumas ferramentas como `stackguard`, `stackshield` e `ProPolice` fazem justamente modificação no epílogo e no prólogo da função. Entretanto, mostram-se ineficazes no combate a todas as formas de exploração possíveis contra um *buffer overflow*. (Bulba e Kil3r, 2000)

As abordagens dependentes do sistema são importantes para aumentar o nível de segurança de um sistema como um todo. Entretanto, por vezes não inibe todas as formas de ataques contra *buffers overflow*, mesmo que o sistema implemente *stacks* não executáveis.

As abordagens dependentes da aplicação ao contrário das soluções anteriores, visam identificar os problemas de segurança antes que o programa seja amplamente distribuído. Duas abordagens são possíveis: a análise estática, que pode ser realizada no código-fonte ou binário, e a análise dinâmica do programa. Para que a análise seja a mais precisa possível seria interessante que as duas análises fossem complementares. Ou seja, fazer com que possíveis problemas de segurança encontrados por um analisador estático fossem validados com uma análise dinâmica e vice-versa.

Entretanto, este trabalho focou-se na análise estática de códigos-fontes. Para tanto, as ferramentas mais difundidas tiveram seu funcionamento estudado para que fosse possível identificar pontos que pudessem ser melhorados. Através de trabalhos como os de Wilander e Kamkar (2002) e Heffley e Meunier (2004) observou-se alguns destes pontos. Sucintamente estes trabalhos mostram a necessidade de uma ferramenta com baixas taxas de falso positivo mesmo que falsos negativos existam, e que consiga entender o comportamento do programa, ou seja, quais funções são chamadas antes de uma dada ação.

Levando em consideração as possibilidades apresentadas através dos estudos realizados um ambiente foi proposto. Este ambiente não tem por objetivo ser completo nem mesmo identificar todo tipo de problemas de segurança presentes em um código-fonte. Entretanto este possui características peculiares que o destingue de todas as demais ferramentas.

O ambiente foi desenvolvido para suportar a realização da análise sintática do código-fonte. Das ferramentas estudadas a única que possibilita uma análise parecida é a `Splint`, entretanto esta análise não é realizada sobre a gramática do código-fonte e sim sobre os comentários que devem ser inseridos em cada uma das funções utilizadas pelo desenvolvedor.

Além da análise sintática, o fluxo de execução de um programa também pode ser obtido através do ambiente proposto. Ou seja, é possível identificar o conjunto de ações realizadas antes que uma dada função seja chamada. Esta característica permite que análises que usam técnicas de *constraint optimization* sejam realizadas. O escopo das variáveis também é uma informação importante em técnicas de *constraint optimization* e este pode ser obtido durante a análise sintática do código-fonte.

Outra questão abordada pelo ambiente é a existência de chamadas a funções de checagem que tornem a ação sob análise segura. Através do fluxo de execução é possível verificar se a seqüência de chamadas é segura ao invés de analisar somente as funções isoladamente.

Baseando-se no ambiente proposto, um protótipo batizado de `SCAP` foi implementado. Para que este pudesse se tornar funcional, uma forma de análise de vulnerabilidades foi implementada. Portanto, os esforços foram dispendidos na análise baseada na identificação de funções inseguras.

Todos os principais módulos foram desenvolvidos para a realização desta forma de análise e o protótipo pôde ter seu funcionamento testado. Alguns experimentos foram realizados para comprovar a eficiência em se utilizar um protótipo baseado no ambiente. Destes testes foi possível identificar vantagens, limitações e possibilidades de melhoria do protótipo.

A maior vantagem identificada nos testes realizados é que o fluxo de execução de um programa, mapeado nas estruturas do módulo de análise sintática, permite que o número de falsos positivos seja reduzido através do estudo das chamadas que foram executadas anteriormente. Desta forma, durante a análise de uma chamada é possível percorrer toda estrutura do programa em busca de informações

Outra vantagem observada é a análise de funções buscando-se por *length modifiers*. Ou seja, *format strings* que possuam uma definição clara de quantos *bytes* serão retornados na saída formatada. Com esta abordagem também foi possível reduzir o número de falsos positivos gerados.

Atualmente, o protótipo apresenta uma série de limitações. Dentre elas destaca-se a não realização de outros tipos de análise de vulnerabilidades. Por exemplo, não são levados em consideração os tipos declarados para as variáveis ou argumentos. Sabendo como a variável foi declarada por vezes é possível eliminar alguns casos de falsos positivos.

Outra grande limitação do protótipo está no módulo de análise sintática. A princípio não são implementadas técnicas de pré-compilação ou extensões da linguagem ANSI C, sendo que existem casos que o analisador retorna erros de sintaxe em códigos-fonte que não possuem esses erros.

Apesar das limitações, o `SCAP` teve um desempenho comparável às outras ferramentas atualmente disponíveis. Além de comprovar que o ambiente proposto pode efetivamente ser implementado e utilizado.

6.2 Melhorias e Trabalhos Futuros

Existem algumas melhorias que podem ser implementadas no protótipo desenvolvido sem que seja necessárias modificações no ambiente proposto:

- O módulo de funções inseguras pode ser modificado para suportar entradas da mesma função mais de uma vez, pois é possível que uma mesma função possa ser insegura de formas diferentes. Desta forma, requerendo checagens diferentes;

- O módulo de análise sintática precisa de um pré-processamento do código-fonte. Pois, como notado anteriormente existem diretivas de pré-compilação e algumas extensões que não são atualmente suportadas pelo protótipo;

- A saída gerada para o usuário do protótipo poderia ser melhor apresentada. Utilizando-se por exemplo uma saída em formato HTML.

Algums trabalhos futuros que podem ser sugeridos a partir deste trabalho são:

- Implementação de análises adicionais, como por exemplo através de otimização de restrições;
- Implementação de análise de vulnerabilidades de *format string*;
- Desenvolvimento de protótipos para outras linguagens de programação;
- Estudo de formas para redução de laços condicionais como `for`, `do`, `while`;
- Implementação de identificadores de comportamentos ilícitos, podendo verificar problemas de *double free* ou *chroot jails*, por exemplo.

REFERÊNCIAS BIBLIOGRÁFICAS

Beale, J.; Baker, A. R.; Caswell, B.; Poor, M. **Snort 2.1 intrusion detection, 2. ed.** Massachutsetts: Syngress Publishing, 2004. 85

Bishop, M.; Dilger, M. Checking for race conditions in file accesses. **Computing Systems**, v. 9, n. 2, p. 153–154, 1996. 57

Bulba; Kil3r. Bypassing stackguard and stackshield. **Phrack Magazine**, v. 56, n. 5, maio 2000. Disponível em: <http://www.phrack.org/archives/56/p56-0x05>. Acesso em: 12 Dez. 2006. 54, 105

Chess, B.; McGraw, G. Static analysis for security. **IEEE Security & Privacy**, v. 2, n. 6, p. 76–78, nov. 2004. Disponível em: <http://www.computer.org/security/>. Acesso em: 12 Dez. 2006. 31, 58, 59, 60

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; C., S. **Introduction to algorithms, 2. ed.** Massachusetts: MIT Press and McGraw-Hill, 2001. 90, 91

Cowan, C. Software security for open-source systems. **IEEE Security & Privacy**, v. 1, n. 1, p. 38–45, jan. 2003. Disponível em: <http://www.computer.org/security/>. Acesso em: 12 Dez. 2006. 51, 57

Cowan, C.; Barringer, M.; Beattie, S.; Kroah-Hartman, G.; Frantzen, M.; Lokier, J. FormatGuard: Automatic protection from printf format string vulnerabilities. In: Usenix Security Symposium, 10., 2001, Washington, United States of America. **Proceedings...** Washington: USENIX Association, 2001a. Disponível em: <http://www.usenix.org/events/sec01/full_papers/cowanbarringer/cowanbarringer.pdf>. Acesso em: 12 Dez. 2006. 51

Cowan, C.; Beattie, S.; Wright, C.; Kroah-Hartman, G. RaceGuard: Kernel protection from temporary file race vulnerabilities. In: USENIX Security Symposium, 10., 2001, Washington DC, United States of America. **Proceedings...** Washington: USENIX Association, 2001b. p. 165 – 172. Disponível em: <http://www.usenix.org/events/sec01/>. Acesso em: 12 Dez. 2006. 57

Cowan, C.; Pu, C.; Maier, D.; Hinton, H.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P.; Zhang, Q. StackGuard: adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium, 7th., 1998, Texas, United States of America. **Proceedings...** San Antonio: USENIX Association, 1998. p. 63–78. 52

Cowan, C.; Wagle, P.; C., P.; Beattie, S.; Walpole, J. Buffer overflows: attacks and defenses for the vulnerability of the decade. In: DARPA Information Survivability Conference & Exposition, 2000, Hilton Head, South Carolina, United States of America. **Proceedings...** New York: IEEE, 2000. p. 119–129. Disponível em: <http://csdl.computer.org/comp/proceedings/discex/2000/0490/02/0490toc.htm>. Acesso em: 12 Dez. 2006. 53

DUARTE, L. O.; BARBATO, L. G. C.; MONTES, A. Vulnerabilidades de software e formas de minimizar suas explorações. In: GTS - Reunião do Grupo de Trabalho em Segurança, 5., 2005, São Paulo. **Proceedings...** São Paulo: GTS, 2005a. 51

DUARTE, L. O.; GREGIO; A., A. R.; BARBATO, L. G. C.; MONTES, A.; HOEPERS, C.; JESSEN, K. Taxonomias de vulnerabilidades: situação atual. In: Simpósio Brasileiro em Segurança da Informação, 5., 2005, Florianópolis. **Proceedings...** Florianópolis: SBC, 2005b. 24

DUARTE, L. O.; GREGIO, A. R. A.; BARBATO, L. G. C.; MONTES, A. Codificação Segura: Abordagens Práticas. In: SSI - Simpósio de Segurança em Informática, 2005, São José dos Campos. **Proceedings...** 2005c. 32

Etoh, H. **GCC extension for protecting applications from stack-smashing attacks**, 2001. Disponível em: <http://www.trl.ibm.com/projects/security/ssp/>. Acesso em: 12 Dez. 2006. 53

Evans, D.; Guttag, J.; Horning, J.; Tan, Y. M. LCLint: a tool for using specifications to check code. In: ACM SIGSOFT symposium on Foundations of software engineering, 2nd., 1994, San Diego, California, United States of America. **Proceedings...** California: ACM Press, 1994. p. 87 – 96. Disponível em: <http://portal.acm.org>. Acesso em: 12 Dez. 2006. 62

GCC the GNU Compiler Collection, 1987. Disponível em: <http://gcc.gnu.org/>. Acesso em: 12 Dez. 2006. 52, 82

Heffley, J.; Meunier, P. Can source code auditing identify common vulnerabilities and be used to evaluate software security? In: Hawaii International Conference on System Sciences, 37., 2004, Waikoloa, Hawaii. **Proceedings...** Hawaii: IEEE, 2004. Disponível em: <http://csdl.computer.org/comp/proceedings/hicss/2004/2056/09/205690277abs.htm>. Acesso em: 12 Dez. 2006. 33, 58, 63, 64, 65, 66, 69, 106

Hoglund, G.; MacGraw, G. **Exploiting software:** howto break code. Addison Wesley, 2004. 471 p. 34

Johnson, S. **Lint, a C program checker**. 12 p. ©Copyright by AT&T Bell Laboratories, 1978. Disponível em: <http://citeseer.ist.psu.edu/>. Acesso em: 12 Dez. 2006. 62

Louden, K. C. **Compiladores: princípios e práticas**. São Paulo: Pioneira Thomson Learning, 2004. 569 p. 59, 60, 71, 80

Miller, B. P.; Fredriksen, L.; So, B. An empirical study of the reliability of UNIX utilities. **Communications of the Association for Computing Machinery**, v. 33, n. 12, p. 32–44, 1990. 58

One, A. Smashing The Stack For Fun And Profit. **Phrack Magazine**, v. 49, n. 14, novembro 1996. Disponível em: <http://www.phrack.org/archives/49/P49-14>. Acesso em: 12 Dez. 2006. 44

Pincus, J.; B., B. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. **IEEE Security & Privacy**, v. 2, n. 4, p. 20–27, jul. 2004. Disponível em: <http://www.computer.org/security/>. Acesso em: 12 Dez. 2006. 35, 44, 45, 46

Scut. Exploiting Format String Vulnerabilities, 2001. Disponível em: <http://althing.cs.dartmounth.edu/local/formats-teso.html>. Acesso em: 12 Dez. 2006. 33

Stytz, M. R.; Whittaker, J. A. Caution: This Product Contains Security Code. **IEEE Security & Privacy**, v. 1, n. 5, p. 86–88, sep. 2003. Disponível em: <http://www.computer.org/security/>. Acesso em: 12 Dez. 2006. 31

Vendicator. `Stack shield`: a "stack smashing" technique protection tool for Linux, 2000. Disponível em: <http://www.angelfire.com/sk/stackshield/>. Acesso em: 12 Dez. 2006. 53

Viega, J.; MacGraw, G. **Building secure software:** how to avoid security problems the right way. New York: Addison Wesley, 2001. 493 p. 24, 38, 39

Wheeler, D. A. **Secure programming for Linux and Unix HOWTO**. 2003. 168 p. Disponível em: http://www.dwheeler.com/secure-programs/. Acesso em: 12 Dez. 2006. 64

Wilander, J.; Kamkar, M. A Comparison of publicly available tools for static intrusion prevent. In: Nordic Workshop on Secure IT Systems, 7th., 2002, Karlstad, Sweden. **Proceedings...** Karlstad, Sweden: ISOC, 2002. Disponível em: <http://www.cs.kua.se/nordsec2002>. Acesso em: 12 Dez. 2006. 61, 63, 64, 65, 69, 97, 101, 103, 106, 151

APÊNDICE A

FERRAMENTAS UTILIZADAS PARA O DESENVOLVIMENTO DO SCAP

Durante a implementação do protótipo `SCAP` foram utilizadas ferramentas livres e de domínio público. Estas ferramentas além de livres são de código-aberto e distribuidas sob a licença *Gnu Public License* (GPL).

A edição dos códigos-fonte foi realizada utilizando-se a ferramenta `Vim` que é a ferramenta mais popular baseada no popular editor de arquivos `Vi`. Através do *website* `http://www.vim.org/`[1] é possível obter maiores informações sobre esta ferramenta.

Além do editor `Vim`, a ferramenta `Lex` foi utilizada para gerar o analisador léxico e a ferramenta `Yacc` para o analisador sintático. Estas ferramentas podem ser encontradas em `http://dinosaur.compilertools.net/`[2], onde também estão disponíveis informações sobre estas ferramentas.

Os demais códigos necessários para a implementação do `SCAP` foram desenvolvidos em `Ansi C` e compilados com a ferramenta *GNU Compiler Collection* (GCC). Este compilador esta disponível em `http://gcc.gnu.org/`[3] onde maiores informações podem ser obtidas.

[1] Acessado em 25 de Mar. de 2007
[2] Acessado em 25 de Mar. de 2007.
[3] Acessado em 25 de Mar. de 2007.

APÊNDICE B

ESTRUTURAS DESENVOLVIDAS PARA O MÓDULO DE ANÁLISE SINTÁTICA

Na Figura B.1 e B.2 estão inseridas as estruturas utilizadas no módulo de análise sintática. As funções desenvolvidas para este módulo têm o objetivo de preecher corretamente estas estruturas.

```
typedef struct stt_func {
  char name[NAMESIZE];
  char file[NAMESIZE];
  unsigned int line;
  unsigned int status;
  struct stt_arg *argument;
  struct stt_var *variable;
  struct stt_act *action;
  struct stt_rest *restriction;
  struct stt_func *nextfunc;
  struct stt_func *prevfunc;
} stt_func;

typedef struct stt_arg {
  char name[NAMESIZE];
  unsigned int type;
  unsigned int sizealoc;
  unsigned int sizeuse;
  unsigned int status;
  struct stt_arg *nextarg;
  struct stt_arg *prevarg;
} stt_arg;
```

Figura B.1 - Primeira parte das estruturas utilizadas no módulo de análise sintática.

```
typedef struct stt_var {
  char name[NAMESIZE];
  unsigned int type;
  unsigned int sizealoc;
  unsigned int sizeuse;
unsigned int status;
struct stt_var *nextvar;
  struct stt_var *prevvar;
} stt_var;

typedef struct stt_act {
  char name[NAMESIZE];
  int status;
  struct stt_parm *parameter;
  struct stt_act *nextact;
  struct stt_act *prevact;
} stt_act;

typedef struct stt_parm {
  char name[NAMESIZE];
  struct stt_parm *nextparm;
  struct stt_parm *prevparm;
} stt_parm;

typedef struct stt_rest {
  char name[NAMESIZE];
  struct stt_rest *nextrest;
  struct stt_rest *prevrest;
} stt_rest;

typedef struct stt_se {
  struct stt_act *prevact;
  char condition[NAMESIZE];
  struct stt_act *firstnextact;
  struct stt_act *secondnextact;
}
```

Figura B.2 - Segunda parte das estruturas utilizadas no módulo de análise sintática.

APÊNDICE C

EXEMPLO DE ESTRUTURAS PREENCHIDAS DURANTE A ANÁLISE SINTÁTICA

Na Figura C.1 são ilustras estruturas preenchidas para o programa de exemplo 4.12.

```
:- FUNCAO: GLOBAL
:-  Line: 0
:-  Variables:
:-   VarName: (BUFFSIZE) T: 1 A: 0 U: 0 S: 0
:-   VarName: (buff) T: 0 A: 0 U: 0 S: 0

:- FUNCAO: sum
:-  Line: 9
:-  Variables:
:-   VarName: (c) T: 1 A: 0 U: 0 S: 0

:- FUNCAO: copystr
:-  Line: 14
:-  Actions:
:-   Action: (strcpy)
:-    Parameters: (out) (in)

:- FUNCAO: copystr2
:-  Line: 19
:-  Actions:
:-   Action: (strlen)
:-    Parameters: (in)
:-   Action: (strcpy)
:-    Parameters: (out) (in)

:- FUNCAO: main
:-  Line: 24
:-  Variables:
:-   VarName: (a) T: 0 A: 0 U: 0 S: 0
:-  Actions:
:-   Action: (printf)
:-    Parameters: (":- Main Function Start\n")
:-   Action: (sum)
:-    Parameters: (CONSTANTE) (CONSTANTE)
:-   Action: (copystr)
:-    Parameters: (argv) (CONSTANTE) (buff)
:-   Action: (printf)
:-    Parameters: (buff)
:-   Action: (copystr2)
:-    Parameters: (argv) (CONSTANTE) (buff) (BUFFSIZE)
:-   Action: (printf)
:-    Parameters: ("\n%s\n") (buff)
```

Figura C.1 - Representação de algumas estruturas geradas na análise sintática.

APÊNDICE D

FUNÇÕES IMPLEMENTADAS NO MÓDULO DE FUNÇÕES INSEGURAS

A seguir estão sumarizadas as funções que foram desenvolvidas para o protótipo do módulo de funções inseguras, que foi implementado na linguagem `C`.

- **ValidStr()**: Esta função tem por objetivo receber duas cadeias de caracteres. A primeira cadeia é um expressão regular `expr` que deve casar ou não a segunda cadeia de caracteres `cmd`. Caso a cadeia `cmd` case com a expressão regular `expr` a função retorna `OK`, senão retorna `FAIL`.

- **NewHash()**: Esta função aloca espaço para uma nova tabela hash. Esta tabela pode ter tamanho variável. Este é passado como um inteiro `size` como parâmetro. Esta função é chamada para criar inicialmente uma tabela hash vazia.

- **Hash();**: Esta função calcula e retorna o hash para uma dada cadeia de caracteres `str` que representa uma função vulnerável. O retorno desta função é um inteiro que representa a posição onde uma dada função vulnerável se encontra ou deve ser inserida.

- **FindHash()**: Esta função procura na tabela hash por uma dada cadeia de caracteres `str` que representa uma função vulnerável. Caso esta entrada seja encontrada, a função retorna um inteiro positivo indicando qual o valor do hash que a cadeia `str` possui.

- **FindHashL()**: Esta função atua de forma similar a função `FindHash()`, entretanto não retorna o valor de hash, caso a entrada exista. Ao invés disso, retorna um ponteiro para a entrada propriamente dita. Com isso, não é necessário procurar pela entrada novamente em funções posteriores.

- **InsertHash()**: Esta função recebe como entrada uma cadeia de caracteres `str` que representa uma nova função insegura. Com esta informação, a função `InsertHash` verifica se esta entrada é nova através da função `FindHash()`. Se a função insegura não for encontrada, uma nova entrada na tabela hash é criada e a função insegura é inserida.

- **RemoveHash()**: Esta função utiliza a função `FindHash()` para procurar uma determinada entrada na tabela hash. Se esta entrada for encontrada, então ela é removida do hash.

- **FreeHash()**: Esta função tem por objetivo remover um dado hash da memória. Os passos tomados por esta função são: zerar os dados presentes no hash e liberar o espaço de memória utilizado anteriormente pelo hash.

- **ShowHash()**: Esta função exibe de forma sucinta o estado atual da tabela hash. Pode ser utilizada para depuração e para verificar se a distribuição da função de hash é adequada para utilização. Ou seja, se não existem muitas entradas com conflito no número do hash.

- **ShowHashEntry()**: Esta função exibe informações mais detalhadas sobre uma dada entrada da tabela de hash. A função vulnerável, se encontrada através da função `FindHashL()`, tem suas características exibidas.

- **ParseStart()**: Esta função é chamada sempre que uma nova *tag* XML é encontrada. Como por exemplo "`<Vulnerability ID="01001">`".

- **ParseData()**: Esta função realiza o tratamento dos dados contidos entre as *tags* XML. Um exemplo é o tratamento dos dados contidos entre as *tags*: `<Function>gets</Function>`.

- **ParseEnd()**: Função executada quando uma *tag* finalizadora é encontrada. Como por exemplo a *tag*: `</Function>`.

APÊNDICE E

FUNÇÕES VULNERÁVEIS ATUALMENTE NO BANCO DE DADOS

O conjunto de funções atualmente agrupadas no arquivo XML de funções vulneráveis pode ser observado na Figura E.1. Nesta figura as funções estão organizadas na tabela *hash* que as suporta em memória.

```
0
1 -> PathCombineA  -> PathAppend  -> wcscpy
2 -> vfwprintf  -> StrFormatByteSize  -> _mbscpy
3 -> OemToAnsiBuffA
4 -> cuserid  -> getc
5 -> PathAppendW
6 -> _ftscanf  -> StrFormatByteSizeW  -> vsnprintf
7 -> wnsprintf  -> StrFormatByteSize64A  -> StrCatN  -> StrNCat
8 -> OemToCharBuff  -> wcsncat
9 -> syslog  -> vsscanf
10 -> swscanf  -> _tcscpy
11 -> wnsprintfW  -> StrCatNW
12 -> OemToCharBuffW  -> StrCat  -> getpass
13 -> system  -> strcpy
14 -> fprintf  -> MultiByte  -> StrCpyA
15 -> vswprintf
16 -> StrCatW
17 -> _tcsncat
18 -> OemToCharA  -> lstrcat
19 -> _stscanf  -> PathCombine  -> _snwprintf  -> realpath
20 -> strncat  -> lstrcpyA  -> gets
21 -> OemToAnsiBuff
22
23 -> _cprintf  -> PathCombineW  -> fgetc
24 -> PathAddExtensionA  -> wvnsprintfA  -> _mbsnbcat  -> getopt_long
25 -> OemToAnsiBuffW  -> StrFormatByteSize64  -> scanf
26 -> StrFormTimeIntervalA
27 -> StrCpyN  -> sprintf
28 -> wcsncpy
29 -> StrFormatByteSize64W  -> _mbsncpy  -> strccpy
30 -> vprintf
31 -> wprintf  -> OemToAnsiA  -> StrCpyNW  -> wcsxfrm  -> strecpy
32 -> StrCpy
33
34 -> PathCanonicalizeA
35
```

```
36 -> OemToChar  -> StrCpyW
37 -> _tcsncpy
38 -> lstrcpy
39 -> fgets
40 -> _tprintf  -> OemToCharW  -> _tcsxfrm  -> strncpy
41 -> getwd
42 -> PathAddExtension  -> wvnsprintf  -> lstrcpyW
43 -> strxfrm  -> bcopy
44 -> _mbsnbcpy  -> StrFormTimeInterval  -> fscanf
45 -> wsprintfA
46 -> PathAddExtensionW  -> wvnsprintfW
47
48 -> CopyMemory  -> StrFormTimeIntervalW  -> popen
49 -> vfprintf  -> OemToAnsi
50 -> fwprintf
51
52 -> PathCanonicalize
53 -> _cscanf  -> StrFormatKBSizeA  -> strtrns
54 -> _vsnwprintf  -> snprintf
55
56 -> PathCanonicalizeW
57 -> ToWideChar  -> sscanf
58 -> PathAddBackslashA
59 -> _ftprintf
60 -> vscanf
61 -> wscanf
62 -> vsprintf
63 -> swprintf  -> wsprintf
64 -> wcscat
65 -> lstrcpyn  -> _mbscat
66 -> vwprintf  -> PathAppendA  -> _snprintf
67 -> StrFormatByteSizeA  -> StrCatBuff  -> wsprintfW
68
69
70 -> _tscanf  -> memcpy  -> getchar
71 -> StrCatBuffW  -> WideCharToMultiByte
72 -> _stprintf  -> wnsprintfA
73 -> OemToCharBuffA  -> _tcscat
74
75 -> StrFormatKBSizeW
76 -> PathAddBackslash  -> strcat
77 -> StrCatA  -> strcadd
78 -> printf  -> getopt
79 -> streadd  -> vfscanf
80 -> fwscanf  -> PathAddBackslashW  -> read
81
```

82

Figura E.1 - Funções presentes no banco de dados de funções inseguras, ordenadas na tabela *hash* implementada.

APÊNDICE F

MODIFICAÇÕES REALIZADAS PELO MÓDULO DE ANÁLISES DE VULNERABILIDADES

Na Figura F.1 são ilustradas as modificações realizadas no exemplo 4.12.

```
:- FUNCAO: GLOBAL
:-  Line: 0
:-  Variables:
:-   VarName: (BUFFSIZE) T: 1 A: 0 U: 0 S: 0
:-   VarName: (buff) T: 0 A: 0 U: 0 S: 0

:- FUNCAO: sum
:-  Line: 9
:-  Variables:
:-   VarName: (c) T: 1 A: 0 U: 0 S: 0

:- FUNCAO: copystr
:-  Line: 14
:-  Actions:
:-   Action: (strcpy)(*P0!)(15)
:-    Parameters: (out) (in)

:- FUNCAO: copystr2
:-  Line: 19
:-  Actions:
:-   Action: (strlen)
:-    Parameters: (in)
:-   Action: (strcpy)(*P0)(20)
:-    Parameters: (out) (in)

:- FUNCAO: main
:-  Line: 24
:-  Variables:
:-   VarName: (a) T: 0 A: 0 U: 0 S: 0
:-  Actions:
:-   Action: (printf)(*F1)(26)
:-    Parameters: (":- Main Function Start\n")
:-   Action: (sum)
:-    Parameters: (CONSTANTE) (CONSTANTE)
:-   Action: (copystr)
:-    Parameters: (argv) (CONSTANTE) (buff)
:-   Action: (printf)(*F1!)(29)
:-    Parameters: (buff)
:-   Action: (copystr2)
:-    Parameters: (argv) (CONSTANTE) (buff) (BUFFSIZE)
:-   Action: (printf)(*F1)(31)
:-    Parameters: ("\n%s\n") (buff)
```

Figura F.1 - Modificações realizadas nas estruturas do exemplo 4.12

APÊNDICE G

SAÍDA DO SCAP PARA O TESTE DO FLAWFINDER

```
:- FUNCAO: GLOBAL
:-  File:
:-  Line: 0
:-  Status: 0

:- FUNCAO: main
:-  File:
:-  Line: 8
:-  Status: 0
:-  Actions:
:-   Action: (printf)(*F1)(9)
:-    Parameters: ("hello\n")

:- FUNCAO: demo
:-  File:
:-  Line: 14
:-  Status: 0
:-  Actions:
:-   Action: (strcpy)(*F1)(15)
:-    Parameters: (a) ("\n")
:-   Action: (gettext)
:-    Parameters: ("Hello there")
:-   Action: (strcpy)(*F1!P1!)(16)
:-    Parameters: (a) ((PREV))
:-   Action: (strcpy)(*F1!P1!)(17)
:-    Parameters: (b) (a)
:-   Action: (sprintf)(*F2)(18)
:-    Parameters: (s) ("\n")
:-   Action: (sprintf)(*F2)(19)
:-    Parameters: (s) ("hello")
:-   Action: (sprintf)(*F2!)(20)
:-    Parameters: (s) ("hello %s") (bug)
:-   Action: (gettext)
:-    Parameters: ("hello %s")
:-   Action: (sprintf)(*F2!)(21)
:-    Parameters: (s) ((PREV)) (bug)
:-   Action: (sprintf)(*F2!)(22)
:-    Parameters: (s) (unknown) (bug)
:-   Action: (printf)(*F1!)(23)
:-    Parameters: (bf) (x)
:-   Action: (scanf)(*F2)(24)
:-    Parameters: ("%d") (x)
```

```
:-   Action: (scanf)(*F2!)(25)
:-    Parameters: ("%s") (s)
:-   Action: (scanf)(*F2)(26)
:-    Parameters: ("%10s") (s)
:-   Action: (scanf)(*F2!)(27)
:-    Parameters: ("%s") (s)
:-   Action: (gets)(*!)(28)
:-    Parameters: (f)
:-   Action: (printf)(*F1)(29)
:-    Parameters: ("\\")
:-   Action: (gets)(*!)(31)
:-    Parameters: (f)
:-   Action: (gets)(*!)(32)
:-    Parameters: (f)
:-   Action: (strerror)
:-    Parameters: (errno)
:-   Action: (syslog)(*F1)(35)
:-    Parameters: (LOG_ERR) ("cannot open config file (%s): %s") (filename) ((PREV))
:-   Action: (syslog)(*F1)(36)
:-    Parameters: (LOG_CRIT) ("malloc() failed")
:-   Action: (syslog)(*F1!)(38)
:-    Parameters: (LOG_ERR) (attacker_string)

:- FUNCAO: demo2
:-  File:
:-  Line: 44
:-  Status: 0
:-  Variables:
:-   VarName: (d) T: 0 A: 0 U: 0 S: 0
:-   VarName: (s) T: 0 A: 0 U: 0 S: 0
:-   VarName: (n) T: 0 A: 0 U: 0 S: 0
:-  Actions:
:-   Action: (_mbscpy)(*!)(49)
:-    Parameters: (d) (s)
:-   Action: (memcpy)(*!)(50)
:-    Parameters: (d) (s)
:-   Action: (CopyMemory)(*!)(51)
:-    Parameters: (d) (s)
:-   Action: (lstrcat)(*!)(52)
:-    Parameters: (d) (s)
:-   Action: (strncpy)(*!)(53)
:-    Parameters: (d) (s)
:-   Action: (_tcsncpy)(*!)(54)
:-    Parameters: (d) (s)
:-   Action: (strncat)(*!)(55)
:-    Parameters: (d) (s) ((CONSTANT))
```

```
:-   Action: (sizeof)
:-    Parameters: (d)
:-   Action: (strncat)(*!)(56)
:-    Parameters: (d) (s) ((PREV))
:-   Action: (sizeof)
:-    Parameters: (d)
:-   Action: (_tcsncat)(*!)(57)
:-    Parameters: (d) (s) ((PREV))
:-   Action: (strlen)
:-    Parameters: (d)
:-   Action: (sizeof)
:-    Parameters: (wszUserName)
:-   Action: (MultiByteToWideChar)
:-    Parameters: (CP_ACP) ((CONSTANT)) (szName) ((CONSTANT)) (wszUserName) ((PREV))
:-   Action: (sizeof)
:-    Parameters: (wszUserName)
:-   Action: (MultiByteToWideChar)
:-    Parameters: (CP_ACP) ((CONSTANT)) (szName) ((CONSTANT)) (wszUserName) ((PREV))
:-   Action: (sizeof)
:-    Parameters: (wszUserName)
:-   Action: (sizeof)
:-    Parameters: ((CONSTANT))
:-   Action: (MultiByteToWideChar)
:-    Parameters: (CP_ACP) ((CONSTANT)) (szName) ((CONSTANT)) (wszUserName) ((PREV))
(wszUserName) ((PREV))
:-   Action: (sizeof)
:-    Parameters: (wszUserName)
:-   Action: (sizeof)
:-    Parameters: ((CONSTANT))
:-   Action: (MultiByteToWideChar)
:-    Parameters: (CP_ACP) ((CONSTANT)) (szName) ((CONSTANT)) (wszUserName) ((PREV))
(wszUserName) ((PREV))
:-   Action: (SetSecurityDescriptorDacl)
:-    Parameters: (sd) (TRUE) (NULL) (FALSE)
:-   Action: (CreateProcess)
:-    Parameters: (NULL) ("C:\\Program Files\\GoodGuy\\GoodGuy.exe -x") ("")
:-   Action: (printf)(*F1)(77)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(78)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(79)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(80)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(81)
:-    Parameters: ("%c\n") ((CONSTANT))
```

```
:-   Action: (printf)(*F1)(82)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(83)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(84)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(85)
:-    Parameters: ("%c\n") ((CONSTANT))
:-   Action: (printf)(*F1)(86)
:-    Parameters: ("%c\n") ("'")

:- FUNCAO: getopt_example
:-  File:
:-  Line: 90
:-  Status: 0
:-  Actions:
:-   Action: (getopt_long)(*!)(91)
:-    Parameters: (argc) (argv) ("a") (longopts) (NULL)

:- FUNCAO: testfile
:-  File:
:-  Line: 95
:-  Status: 0
:-  Actions:
:-   Action: (fopen)
:-    Parameters: ("/etc/passwd") ("r")
:-   Action: (fclose)
:-    Parameters: (f)

:- FUNCAO: accesstest
:-  File:
:-  Line: 113
:-  Status: 0
:-  Variables:
:-   VarName: (access) T: 1 A: 0 U: 0 S: 0
```

Figura G.1 - Resultados do módulo de análises de vulnerabilidades para o teste do `Flawfinder`.

```
:- Function: demo
:-   Action: (strcpy*) (16)
:-   x Comment: This function copies the string of the 2nd argument (including the
terminating '\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the second argument. Use strlcpy(); with proper
arguments instead.
```

```
:-   x Mitigation Function: strlcpy();

:- Function: demo
:-   Action: (strcpy*) (17)
:-   x Comment: This function copies the string of the 2nd argument (including the
terminating '\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the second argument. Use strlcpy(); with proper
arguments instead.
:-   x Mitigation Function: strlcpy();

:- Function: demo
:-   Action: (sprintf*) (20)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
proper arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: demo
:-   Action: (sprintf*) (21)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
proper arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: demo
:-   Action: (sprintf*) (22)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
proper arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: demo
:-   Action: (printf*) (23)
:-   x Comment: This function writes informations to STDOUT according to the 1st
argument format string.
:-   x Mitigation: This function can allow a Format String exploitation. Please,
```

```
define the format string.

:- Function: demo
:-   Action: (scanf*) (25)
:-   x Comment: This function scans input, stdin, according to a format string
present on the 1st argument. The result is written to a variable number of
arguments. These arguments can be character arrays that must support the formatted
string provided by the 1st argument.
:-   x Mitigation:  Be sure that the size of all character arrays is larger enough
to support the output formatted by the first argument. Use precision specifiers
to ensure that.

:- Function: demo
:-   Action: (scanf*) (27)
:-   x Comment: This function scans input, stdin, according to a format string
present on the 1st argument. The result is written to a variable number of
arguments. These arguments can be character arrays that must support the formatted
string provided by the 1st argument.
:-   x Mitigation:  Be sure that the size of all character arrays is larger enough
to support the output formatted by the first argument. Use precision specifiers to
 ensure that.

:- Function: demo
:-   Action: (gets*) (28)
:-   x Comment: Reads a line from stdin into the buffer on the 1st argument until
either a new line terminator or EOF is found, wich it replaces with '\0'.
:-   x Mitigation:  The function gets(); reads a line from stdin that can be larger
than the destination buffer. Use fgets(); instead.
:-   x Mitigation Function: fgets();

:- Function: demo
:-   Action: (gets*) (31)
:-   x Comment: Reads a line from stdin into the buffer on the 1st argument until
either a new line terminator or EOF is found, wich it replaces with '\0'.
:-   x Mitigation:  The function gets(); reads a line from stdin that can be larger
than the destination buffer. Use fgets(); instead.
:-   x Mitigation Function: fgets();

:- Function: demo
:-   Action: (gets*) (32)
:-   x Comment: Reads a line from stdin into the buffer on the 1st argument until
either a new line terminator or EOF is found, wich it replaces with '\0'.
:-   x Mitigation:  The function gets(); reads a line from stdin that can be larger
than the destination buffer. Use fgets(); instead.
:-   x Mitigation Function: fgets();
```

```
:- Function: demo
:-   Action: (syslog*) (38)

:- Function: demo2
:-   Action: (_mbscpy*) (49)

:- Function: demo2
:-   Action: (memcpy*) (50)

:- Function: demo2
:-   Action: (CopyMemory*) (51)

:- Function: demo2
:-   Action: (lstrcat*) (52)

:- Function: demo2
:-   Action: (strncpy*) (53)

:- Function: demo2
:-   Action: (_tcsncpy*) (54)

:- Function: demo2
:-   Action: (strncat*) (55)
:-   x Mitigation Function: Consider using strlcat() instead.

:- Function: demo2
:-   Action: (strncat*) (56)
:-   x Mitigation Function: Consider using strlcat() instead.

:- Function: demo2
:-   Action: (_tcsncat*) (57)

:- Function: getopt_example
:-   Action: (getopt_long*) (91)
```

Figura G.2 - Saída do protótipo SCAP para o teste do Flawfinder.

APÊNDICE H

SAÍDA DO SCAP PARA O TESTE DE WILANDER

```
:- FUNCAO: GLOBAL
:-  File:
:-  Line: 0
:-  Status: 0
:-  Variables:
:-   VarName: (BUFSIZE) T: 1 A: 0 U: 0 S: 0
:-   VarName: (global_buffer) T: 0 A: 0 U: 0 S: 0
:-   VarName: (local_buffer) T: 0 A: 0 U: 0 S: 0

:- FUNCAO: main
:-  File:
:-  Line: 6
:-  Status: 0
:-  Actions:
:-   Action: (gets)(*!)(7)
:-    Parameters: (buffer)
:-   Action: (scanf)(*F2)(8)
:-    Parameters: ("%8s") (buffer_safe)
:-   Action: (scanf)(*F2!)(9)
:-    Parameters: ("%s") (buffer_unsafe)
:-   Action: (fopen)
:-    Parameters: (file_name) ("w")
:-   Action: (fscanf)(*F2)(10)
:-    Parameters: ((PREV)) ((PREV)) ("%8s") (buffer_safe)
:-   Action: (fopen)
:-    Parameters: (file_name) ("w")
:-   Action: (fscanf)(*F2!)(11)
:-    Parameters: ((PREV)) ((PREV)) ("%s") (buffer_unsafe)
:-   Action: (sscanf)(*F2)(12)
:-    Parameters: (input_string) ("%8s") (buffer_safe)
:-   Action: (sscanf)(*F2!)(13)
:-    Parameters: (input_string) ("%s") (buffer_unsafe)
:-   Action: (vscanf)(*F2)(14)
:-    Parameters: ("%8s") (arglist)
:-   Action: (vscanf)(*F2!)(15)
:-    Parameters: ("%s") (arglist)
:-   Action: (vsscanf)(*F2)(16)
:-    Parameters: (input_string) ("%8s") (arglist)
:-   Action: (vsscanf)(*F2!)(17)
:-    Parameters: (input_string) ("%s") (arglist)
```

```
:-   Action: (fopen)
:-    Parameters: (file_name) ("w")
:-   Action: (vfscanf)(*F2)(19)
:-    Parameters: ((PREV)) ((PREV)) ("%8s") (arglist)
:-   Action: (fopen)
:-    Parameters: (file_name) ("w")
:-   Action: (vfscanf)(*F2!)(21)
:-    Parameters: ((PREV)) ((PREV)) ("%s") (arglist)
:-   Action: (sprintf)(*F2)(22)
:-    Parameters: (buffer_safe) ("%8s") (input_string)
:-   Action: (sprintf)(*F2!)(23)
:-    Parameters: (buffer_unsafe) ("%s") (input_string)
:-   Action: (strlen)
:-    Parameters: (input_string)
:-   Action: (strcat)(*P1)(25)
:-    Parameters: (buffer_safe) (input_string)
:-   Action: (strcat)(*P1!)(26)
:-    Parameters: (buffer_unsafe) (input_string)
:-   Action: (strlen)
:-    Parameters: (input_string)
:-   Action: (strcpy)(*F1!P1)(28)
:-    Parameters: (buffer_safe) (input_string)
:-   Action: (strcpy)(*F1!P1!)(29)
:-    Parameters: (buffer_unsafe) (input_string)
:-   Action: (cuserid)(*!)(30)
:-    Parameters: (buffer_unsafe)
:-   Action: (vsprintf)(*F2)(31)
:-    Parameters: (buffer_safe) ("%8s") (arglist)
:-   Action: (vsprintf)(*F2!P1!)(32)
:-    Parameters: (buffer_unsafe) ("%s") (arglist)
:-   Action: (streadd)(*F1)(33)
:-    Parameters: (buffer_safe) ("a") ("")
:-   Action: (streadd)(*F1!)(34)
:-    Parameters: (buffer_unsafe) (input_string) ("")
:-   Action: (strecpy)(*F1)(35)
:-    Parameters: (buffer_safe) ("a") ("")
:-   Action: (strecpy)(*F1!)(36)
:-    Parameters: (buffer_unsafe) (input_string) ("")
:-   Action: (strtrns)(*F1)(37)
:-    Parameters: ("a") ("a") ("A") (buffer_safe)
:-   Action: (strtrns)(*F1!)(38)
:-    Parameters: (input_string) ("a") ("A") (buffer_unsafe)
:-   Action: (printf)(*F1!)(40)
:-    Parameters: (static_global_buffer)
:-   Action: (printf)(*F1!)(41)
:-    Parameters: (global_buffer)
```

```
:-   Action: (fprintf)(*F2!)(42)
:-    Parameters: (stdout) (static_global_buffer)
:-   Action: (fprintf)(*F2!)(43)
:-    Parameters: (stdout) (global_buffer)
:-   Action: (sprintf)(*F2!)(45)
:-    Parameters: (local_buffer) (static_global_buffer) (input_string)
:-   Action: (sprintf)(*F2!)(47)
:-    Parameters: (local_buffer) (global_buffer) (input_string)
:-   Action: (snprintf)(*F1!)(50)
:-    Parameters: (local_buffer) (BUFSIZE) (static_global_buffer) (input_string)
:-   Action: (snprintf)(*F1!)(52)
:-    Parameters: (local_buffer) (BUFSIZE) (global_buffer) (input_string)
:-   Action: (vprintf)(*F1!)(53)
:-    Parameters: (static_global_buffer) (arglist)
:-   Action: (vprintf)(*F1!)(54)
:-    Parameters: (global_buffer) (arglist)
:-   Action: (vfprintf)(*F1!)(56)
:-    Parameters: (stdout) (static_global_buffer) (arglist)
:-   Action: (vfprintf)(*F1!)(58)
:-    Parameters: (stdout) (global_buffer) (arglist)
:-   Action: (vsprintf)(*F2!P1!)(61)
:-    Parameters: (local_buffer) (static_global_buffer) (arglist)
:-   Action: (vsprintf)(*F2!P1!)(63)
:-    Parameters: (local_buffer) (global_buffer) (arglist)
:-   Action: (vsnprintf)(*F1!)(66)
:-    Parameters: (local_buffer) (BUFSIZE) (static_global_buffer) (arglist)
:-   Action: (vsnprintf)(*F1!)(68)
:-    Parameters: (local_buffer) (BUFSIZE) (global_buffer) (arglist)
```

Figura H.1 - Resultados do módulo de análises de vulnerabilidades para o teste de Wilander.

```
:- Function: main
:-   Action: (gets*) (7)
:-   x Comment: Reads a line from stdin into the buffer on the 1st argument until
either a new line terminator or EOF is found, wich it replaces with '\0'.
:-   x Mitigation:  The function gets(); reads a line from stdin that can be larger
than the destination buffer. Use fgets(); instead.
:-   x Mitigation Function: fgets();

:- Function: main
:-   Action: (scanf*) (9)
:-   x Comment: This function scans input, stdin, according to a format string
```

present on the 1st argument. The result is written to a variable number of arguments. These arguments can be character arrays that must support the formatted string provided by the 1st argument.
:- x Mitigation: Be sure that the size of all character arrays is larger enough to support the output formatted by the first argument. Use precision specifiers to ensure that.

:- Function: main
:- Action: (fscanf*) (11)
:- x Comment: This function scans the 1st argument, that is a FILE, according to a format string present on the 2nd argument. The result is written to a variable number of arguments. These arguments can be character arrays that must support the formatted string provided by the 2nd argument.
:- x Mitigation: Be sure that the size of all character arrays are larger enough to support the output formatted by the second argument. Use precision specifiers to ensure that.

:- Function: main
:- Action: (sscanf*) (13)
:- x Comment: This function scans the 1st argument, that is a constant character stream, according to a format string present on the 2nd argument. The result is written to a variable number of arguments. These arguments can be character arrays that must support the formatted string provided by the 2nd argument.
:- x Mitigation: Be sure that the size of all character arrays are larger enough to support the output formatted by the second argument. Use precision specifiers to ensure that.

:- Function: main
:- Action: (vscanf*) (15)
:- x Comment: This function scans the STDIN according to a format string present on the 1st argument. The result is written to a va_list. These arguments can be character arrays that must support the formatted string provided by the 1st argument.
:- x Mitigation: Be sure that the size of all character arrays are larger enough to support the output formatted by the first argument. Use precision specifiers to ensure that.

:- Function: main
:- Action: (vsscanf*) (17)
:- x Comment: This function scans the 1st argument, that is a const char stream, according to a format string present on the 2nd argument. The result is written to a va_list. These arguments can be character arrays that must support the formatted string provided by the 2nd argument.
:- x Mitigation: Be sure that the size of all character arrays are larger enough to support the output formatted by the second argument. Use precision specifiers to ensure that.

```
:- Function: main
:-   Action: (vfscanf*) (21)
:-   x Comment: This function scans the 1st argument, that is a FILE, according to
a format string present on the 2nd argument. The result is written to a va_list.
These arguments can be character arrays that must support the formatted string
provided by the 2nd argument.
:-   x Mitigation:  Be sure that the size of all character arrays are larger enough
to support the output formatted by the second argument. Use precision specifiers to
ensure that.

:- Function: main
:-   Action: (sprintf*) (23)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
propers arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: main
:-   Action: (strcat*) (25)
:-   x Comment: This function appends the 2nd argument (including the terminating
'\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the first argument plus second one. Use strlcat();
with proper arguments instead.
:-   x Mitigation Function: strlcat();

:- Function: main
:-   Action: (strcat*) (26)
:-   x Comment: This function appends the 2nd argument (including the terminating
'\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the first argument plus second one. Use strlcat();
with proper arguments instead.
:-   x Mitigation Function: strlcat();

:- Function: main
:-   Action: (strcpy*) (28)
:-   x Comment: This function copies the string of the 2nd argument (including the
terminating '\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the second argument. Use strlcpy(); with propers
arguments instead.
:-   x Mitigation Function: strlcpy();
```

```
:- Function: main
:-   Action: (strcpy*) (29)
:-   x Comment: This function copies the string of the 2nd argument (including the
terminating '\0' character) to the array pointed by the 1st argument.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the size of the second argument. Use strlcpy(); with propers
arguments instead.
:-   x Mitigation Function: strlcpy();

:- Function: main
:-   Action: (cuserid*) (30)
:-   x Comment: The cuserid() function generates a character representation of the
name associated with the effective user ID of the process.
:-   x Mitigation: This interface is obsolescent; getpwuid() should be used instead.
:-   x Mitigation Function: getpwuid

:- Function: main
:-   Action: (vsprintf*) (32)
:-   x Comment: This function writes the output under the control of a format string
to a string pointed in the 1st argument.
:-   x Mitigation: Be sure to use a secure format string as the 2nd argument. In case
of strings, use length modifiers.
:-   x Mitigation Function: vsnprintf

:- Function: main
:-   Action: (streadd*) (34)
:-   x Comment: This function copies the 2nd argument, up to a null byte, to the 1st
argument, expanding non-graphic characters to their equivalent C-language escape
sequences (for example, \n, \001). The return is a pointer to the null byte that
terminates the output.
:-   x Mitigation:  Be sure that the size of the 1st argument is four times larger
than the 2nd argument.

:- Function: main
:-   Action: (strecpy*) (36)
:-   x Comment: This function copies the 2nd argument, up to a null byte, to the 1st
argument, expanding non-graphic characters to their equivalent C-language escape
sequences (for example, \n, \001). The return is a pointer to the begin of output
array.
:-   x Mitigation:  Be sure that the size of the 1st argument is four times larger
than the 2nd argument.

:- Function: main
:-   Action: (strtrns*) (38)
:-   x Comment: strtrns(const char *str, const char *
```

```
:-   x Mitigation: Be sure that the size of the last argument is large enough to
support the size of the first one.

:- Function: main
:-   Action: (printf*) (40)
:-   x Comment: This function writes informations to STDOUT according to the 1st
argument format string.
:-   x Mitigation: This function can allow a Format String exploitation. Please,
define the format string.

:- Function: main
:-   Action: (printf*) (41)
:-   x Comment: This function writes informations to STDOUT according to the 1st
argument format string.
:-   x Mitigation: This function can allow a Format String exploitation. Please,
define the format string.

:- Function: main
:-   Action: (fprintf*) (42)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (fprintf*) (43)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (sprintf*) (45)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
propers arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: main
:-   Action: (sprintf*) (47)
:-   x Comment: This function produce output according to a format string present
on the 2nd argument. The result is written to the character string of the 1st
argument. Others arguments are used by the 2nd one to produce the output.
:-   x Mitigation:  Be sure that the size of the first function argument is larger
enough to support the output formatted by the second one. Use snprintf(); with
propers arguments instead or precisions specifiers.
:-   x Mitigation Function: snprintf();

:- Function: main
```

```
:-   Action: (snprintf*) (50)

:- Function: main
:-   Action: (snprintf*) (52)

:- Function: main
:-   Action: (vprintf*) (53)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (vprintf*) (54)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (vfprintf*) (56)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (vfprintf*) (58)
:-   x Comment: This function can allow a Format String exploitation.

:- Function: main
:-   Action: (vsprintf*) (61)
:-   x Comment: This function writes the output under the control of a format
string to a string pointed in the 1st argument.
:-   x Mitigation: Be sure to use a secure format string as the 2nd argument. In
case of strings, use length modifiers.
:-   x Mitigation Function: vsnprintf

:- Function: main
:-   Action: (vsprintf*) (63)
:-   x Comment: This function writes the output under the control of a format
string to a string pointed in the 1st argument.
:-   x Mitigation: Be sure to use a secure format string as the 2nd argument. In
case of strings, use length modifiers.
:-   x Mitigation Function: vsnprintf

:- Function: main
:-   Action: (vsnprintf*) (66)
:-   x Comment: This function writes the output under the control of a format
string to a string pointed in the 3rd argument.
:-   x Mitigation: Be sure to use a secure format string as the 3rd argument. In
case of strings, use length modifiers.

:- Function: main
:-   Action: (vsnprintf*) (68)
```

```
:-   x Comment: This function writes the output under the control of a format
string to a string pointed in the 3rd argument.
:-   x Mitigation: Be sure to use a secure format string as the 3rd argument. In
case of strings, use length modifiers.
```

Figura H.2 - Saída do protótipo SCAP para o teste de Wilander.

ANEXO A

E-MAIL DE TOM STOCKFISCH PARA NET.SOURCES

Na Figura A.1 está transcrito parte do e-mail de Tom Stockficsh para a lista `net.sources` com uma implementação de um analisador léxico e um analisador sintático para um rascunho da linguagem `C`.

A parte que está relacionada ao código-fonte dos analisadores foi suprimida pois é extremamente extensa.

```
From tps@sdchemf.UUCP Tue Mar  3 16:31:17 1987
Path: beno!seismo!lll-lcc!ames!ucbcad!ucbvax!sdcsvax!sdchem!tps
From: tps@sdchem.UUCP (Tom Stockfisch)
Newsgroups: net.sources
Subject: ANSI C draft yacc grammar
Message-ID: <645@sdchema.sdchem.UUCP>
Date: 3 Mar 87 21:31:17 GMT
References: <403@ubc-vision.UUCP>
Sender: news@sdchem.UUCP
Reply-To: tps@sdchemf.UUCP (Tom Stockfisch)
Organization: UC San Diego
Lines: 775

People keep asking me for a copy, so I am reposting a yacc grammar (actually,
a complete program) for the April 1985 draft of the ANSI C standard.  It
finds syntax errors in its input.

Oh yeah, this was originally posted by Jeff Lee.

|| Tom Stockfisch, UCSD Chemistry        tps%chem@sdcsvax.UCSD

#!/bin/sh
# to extract, remove the header and type "sh filename"
...
```

Figura A.1 - E-mail enviado por Tom Stockficsh para o *newsgroup* `net.sources`.

ANEXO B

CÓDIGOS-FONTE DOS TESTES DO FLAWFINDER

Neste apêndice é transcrito na Figura B.1 o código-fonte de teste utilizado pelo `Flawfinder`, que pode ser obtido em: <http://www.dwheeler.com/flawfinder/test.c>. Acesso em: 12 de Dez. de 2006.

```
1  /* Test flawfinder.  This program won't compile or run; that's not necessary
2     for this to be a useful test. */
3
4  #include <stdio.h>
5  #define hello(x) goodbye(x)
6  #define WOKKA "stuff"
7
8  main() {
9   printf("hello\n");
10 }
11
12 /* This is a strcpy test. */
13
14 int demo(char *a, char *b) {
15  strcpy(a, "\n"); // Did this work?
16  strcpy(a, gettext("Hello there")); // Did this work?
17  strcpy(b, a);
18  sprintf(s, "\n");
19  sprintf(s, "hello");
20  sprintf(s, "hello %s", bug);
21  sprintf(s, gettext("hello %s"), bug);
22  sprintf(s, unknown, bug);
23  printf(bf, x);
24  scanf("%d", &x);
25  scanf("%s", s);
26  scanf("%10s", s);
27  scanf("%s", s);
28  gets(f); // Flawfinder: ignore
29  printf("\\");
30  /* Flawfinder: ignore */
31  gets(f);
32  gets(f);
33  /* These are okay, but flawfinder version < 0.20 incorrectly used
34     the first parameter as the parameter for the format string */
35  syslog(LOG_ERR,"cannot open config file (%s): %s",filename,strerror(errno))
36  syslog(LOG_CRIT,"malloc() failed");
37  /* But this one SHOULD trigger a warning. */
```

```
 syslog(LOG_ERR, attacker_string);

}



demo2() {
  char d[20];
  char s[20];
  int n;

  _mbscpy(d,s); /* like strcpy, this doesn't check for buffer overflow */
  memcpy(d,s);
  CopyMemory(d,s);
  lstrcat(d,s);
  strncpy(d,s);
  _tcsncpy(d,s);
  strncat(d,s,10);
  strncat(d,s,sizeof(d)); /* Misuse - this should be flagged as riskier. */
  _tcsncat(d,s,sizeof(d)); /* Misuse - flag as riskier */
  n = strlen(d);
  /* This is wrong, and should be flagged as risky: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof(wszUserName));
  /* This is also wrong, and should be flagged as risky: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof wszUserName);
  /* This is much better: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof(wszUserName)/...);
  /* This is much better: */
  MultiByteToWideChar(CP_ACP,0,szName,-1,wszUserName,sizeof wszUserName /...);
  /* This is an example of bad code - the third paramer is NULL, so it creates
     a NULL ACL.  Note that Flawfinder can't detect when a
     SECURITY_DESCRIPTOR structure is manually created with a NULL value
     as the ACL; doing so would require a tool that handles C/C++
     and knows about types more that flawfinder currently does.
     Anyway, this needs to be detected: */
  SetSecurityDescriptorDacl(&sd,TRUE,NULL,FALSE);
  /* This one is a bad idea - first param shouldn't be NULL */
  CreateProcess(NULL, "C:\\Program Files\\GoodGuy\\GoodGuy.exe -x", "");
  /* Test interaction of quote characters */
  printf("%c\n", 'x');
  printf("%c\n", '"');
  printf("%c\n", '\"');
  printf("%c\n", '\'');
  printf("%c\n", '\177');
  printf("%c\n", '\xfe');
  printf("%c\n", '\xd');
```

```
84    printf("%c\n", '\n');
85    printf("%c\n", '\\');
86    printf("%c\n", "'");
87  }
88
89
90  int getopt_example(int argc,char *argv[]) {
91      while ((optc = getopt_long (argc, argv, "a",longopts, NULL )) != EOF) {
92      }
93  }
94
95  int testfile() {
96    FILE *f;
97    f = fopen("/etc/passwd", "r");
98    fclose(f);
99  }
100
101 /* Regression test: handle \\\n after end of string */
102
103 #define assert(x) {\
104  if (!(x)) {\
105  fprintf(stderr,"Assertion failed.\n"\
106  "File: %s\nLine: %d\n"\
107  "Assertion: %s\n\n"\
108  ,__FILE__,__LINE__,#x);\
109  exit(1);\
110  };\
111  }
112
113 int accesstest() {
114   int access = 0; /* Not a function call.  Should be caught by the
115                      false positive test, and NOT labelled as a problem. */
116 }
117
```

Figura B.1 - Código-fonte de teste utilizado pelo `Flawfinder`.

ANEXO C

CÓDIGO-FONTE DOS TESTES DE WILANDER E KAMKAR MODIFICADO

Na Figura C.1 está transcrito o código-fonte de teste utilizado em Wilander e Kamkar (2002).

```
#define BUFSIZE 9
static char static_global_buffer = 'A';
static char global_buffer[BUFSIZE];
char local_buffer[BUFSIZE];
/***** Buffer Overflow Vulnerabilities *****/
void main(void) {
  pointer = gets(buffer); /* Unsafe */
  scanf("%8s", buffer_safe); /* Safe */
  scanf("%s", buffer_unsafe); /* Unsafe */
  fscanf(fopen(file_name, "w"), "%8s", buffer_safe); /* Safe */
  fscanf(fopen(file_name, "w"), "%s", buffer_unsafe); /* Unsafe */
  sscanf(input_string, "%8s", buffer_safe); /* Safe */
  sscanf(input_string, "%s", buffer_unsafe); /* Unsafe */
  if(choice==0) vscanf("%8s", arglist); /* Safe */
  else vscanf("%s", arglist); /* Unsafe */
  if(choice==0) vsscanf(input_string, "%8s", arglist); /* Safe */
  else vsscanf(input_string, "%s", arglist); /* Unsafe */
  if(choice==0)
  vfscanf(fopen(file_name, "w"), "%8s", arglist); /* Safe */
  else
  vfscanf(fopen(file_name, "w"), "%s", arglist); /* Unsafe */
  sprintf(buffer_safe, "%8s", input_string); /* Safe */
  sprintf(buffer_unsafe, "%s", input_string); /* Unsafe */
  if(strlen(input_string)<BUFSIZE)
  strcat(buffer_safe, input_string); /* Safe */
  strcat(buffer_unsafe, input_string); /* Unsafe */
  if(strlen(input_string)<BUFSIZE)
  strcpy(buffer_safe, input_string); /* Safe */
  strcpy(buffer_unsafe, input_string); /* Unsafe */
  cuserid(buffer_unsafe); /* Unsafe */
  if(choice==0) vsprintf (buffer_safe, "%8s", arglist); /* Safe */
```

```
32    else vsprintf (buffer_unsafe, "%s", arglist); /* Unsafe */
33    res = streadd(buffer_safe, "a", ""); /* Safe */
34    res = streadd(buffer_unsafe, input_string, ""); /* Unsafe */
35    res = strecpy(buffer_safe, "a", ""); /* Safe */
36    res = strecpy(buffer_unsafe, input_string, ""); /* Unsafe */
37    res = strtrns("a", "a", "A", buffer_safe); /* Safe */
38    res = strtrns(input_string, "a", "A", buffer_unsafe); /* Unsafe */
39    /***** Format String Vulnerabilities *****/
40    printf(&static_global_buffer); /* Safe */
41    printf(global_buffer); /* Unsafe */
42    fprintf(stdout, &static_global_buffer); /* Safe */
43    fprintf(stdout, global_buffer); /* Unsafe */
44    /* Safe */
45    sprintf(local_buffer, &static_global_buffer, input_string);
46    /* Unsafe */
47    sprintf(local_buffer, global_buffer, input_string);
48
49    /* Safe */
50    snprintf(local_buffer, BUFSIZE, &static_global_buffer, input_string);
51    /* Unsafe */
52    snprintf(local_buffer, BUFSIZE, global_buffer, input_string);
53    if(choice==0) vprintf(&static_global_buffer, arglist); /* Safe */
54    else vprintf(global_buffer, arglist); /* Unsafe */
55    if(choice==0) /* Safe */
56    vfprintf(stdout, &static_global_buffer, arglist);
57    else /* Unsafe */
58    vfprintf(stdout, global_buffer, arglist);
59
60    if(choice==0) /* Safe */
61    vsprintf(local_buffer, &static_global_buffer, arglist);
62    else /* Unsafe */
63    vsprintf(local_buffer, global_buffer, arglist);
64
65    if(choice==0) /* Safe */
66    vsnprintf(local_buffer, BUFSIZE, &static_global_buffer, arglist);
67    else /* Unsafe */
68    vsnprintf(local_buffer, BUFSIZE, global_buffer, arglist);
69  }
```

Figura C.1 - Código-fonte de Wilander e Kamkar modificado para ser sintaticamente correto.

PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programa de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. São aceitos tanto programas fonte quanto executáveis.