Detecção de Vulnerabilidades de Inteiros na Adaptação de Software de 32 para 64 bits [1]

Ibéria Medeiros, Miguel Correia

LASIGE, Faculdade de Ciências da Universidade de Lisboa
Campo Grande, Edifício C6, 1749-016 Lisboa, Portugal
ibemed@gmail.com, mpc@di.fc.ul.pt

Resumo

Fabricantes como a Intel e a AMD começaram a comercializar processadores com arquitecturas de 64 bits mas muito do software executado nesses processadores foi desenvolvido inicialmente para arquitecturas de 32 bits. As aplicações concebidas na linguagem de programação C para a arquitectura de 32 bits ao serem portadas para 64 bits podem ficar com vulnerabilidades relacionadas com a manipulação de inteiros. Este artigo estuda as vulnerabilidades que podem surgir quando se adapta ("porta") sem os necessários cuidados software de 32 para 64 bits, considerando o modelo de dados LP64, muito utilizado em software de código aberto. O artigo propõe também a ferramenta DEEEP que faz a detecção dessas vulnerabilidades através de análise estática de código fonte. A ferramenta é baseada em duas ferramentas de análise estática de código abertas, que são usadas para encontrar bugs na manipulação de inteiros através de verificação de tipos, e para fazer análise de fluxo de dados para verificar se funções perigosas (p.ex., memcpy, strcpy) estão acessíveis de fora do programa. Após estas duas formas de análise, a ferramenta DEEEP permite correlacionar essas duas informações identificando se os bugs encontrados são realmente vulnerabilidades, ou seja, se são atacáveis. O artigo apresenta resultados da utilização da ferramenta com código vulnerável sintético, criado especificamente para avaliar a ferramenta, e com o Sendmail.

1 Introdução

Durante muitos anos, a segurança no processo de desenvolvimento de *software* foi vista como uma questão secundária, sendo a ênfase posta nas funcionalidades fornecidas. Com o passar do tempo, com um mundo empresarial e uma sociedade mais exigentes e onde as novas tecnologias da informação e comunicação desempenham um papel fundamental, o *software* começou a ser desenvolvido em grande escala, com um aumento substancial da sua complexidade e da exigência em termos de fiabilidade e segurança. Esta evolução foi também acompanhada por um aumento do número de atacantes que procuram constantemente vulnerabilidades em programas para que possam atacá-los.

Uma vulnerabilidade por si só não impede o normal funcionamento do *software*, podendo permanecer durante muito tempo no sistema sem ser descoberta. No entanto, quando descoberta e explorada por um ataque bem sucedido acontece uma intrusão que pode levar à falha do sistema [1].

Um tipo de vulnerabilidades bem conhecido é os das vulnerabilidades de inteiros, de que são exemplo os *overflows* e *underflows* [2][3]. Estas vulnerabilidades estão muito relacionadas com as de *buffer overflow*, pois muitas vulnerabilidades deste último tipo não são causadas pela ausência de verificação de limites de tampões de memória (*buffers*) mas pela verificação errónea causada por um *overflow/underflow*.

Recentemente começou a surgir alguma preocupação com a adaptação de código C de 32 para 64 bits por causa da possibilidade de serem introduzidas vulnerabilidades de inteiros. O problema é que no modelo de dados ILP32, definido pela ANSI para processadores de 32 bits [6], os tipos de dados *int*, *long int* e ponteiro ocupam 32 bits, o que permite aos programadores fazerem conversões arbitrárias de variáveis entre esses tipos – especialmente entre *int* e *long* – sem perda de dados [19]. Apesar dessa prática ser indesejável, acontece frequentemente. Quando um programa é portado para 64 bits para o modelo LP64, adoptado pelas variantes do

[1] Este trabalho foi parcialmente financiado pela FCT através da unidade de investigação LASIGE e do projecto POSC/EIA/61643/2004 (AJECT).

Unix e por isso usado em aplicações de código aberto (p.ex., pelo *gcc*), esses três tipos de dados deixam de ter o mesmo número de bits. Se num determinado programa existirem essas conversões entre esses tipos e o programa for portado para 64 bits sem os necessários cuidados, podem surgir vulnerabilidades de inteiros.

Este artigo propõe a detecção deste tipo de vulnerabilidade com recurso a *análise estática de código fonte*. As ferramentas deste tipo percorrem o código fonte de um programa procurando vulnerabilidades de diversos tipos e fazendo diversos tipos de análise, como análise de fluxo, verificação de tipos, análise de fluxo de dados, verificação de modelos, ou análise de léxico [14][15][16][17][18].

Este artigo apresenta a ferramenta *DEEEP (Detector of integEr vulnerabilitiEs in softwarE Portability)* que permite detectar vulnerabilidades de inteiros causadas por *má adaptação* de código de 32 para 64 bits, ou seja, por uma adaptação na qual não seja tomada em conta a referida diferença entre o número de bits dos tipos de dados. A *DEEEP* assenta em duas ferramentas de análise estática de código previamente existentes, o *Lint* [10] e o *Splint* (*Secure Programming LINT*) [11]. Ambas fazem verificação de tipos (*type checking*) e o *Splint* faz também análise de fluxo de dados (*data flow analysis*) de código fonte. A detecção das referidas vulnerabilidades é realizada por um algoritmo com as seguintes fases de processamento: (1) verificação de tipos de dados para detecção de possíveis *bugs* de 64 bits, por não obedecerem à relação entre os tamanhos dos tipos de dados inteiros do modelo LP64; (2) análise de fluxo de dados, utilizando para o efeito atributos associados a objectos do programa (tipos de dados) e anotações para sinalizar as variáveis de entrada e a passagem das mesmas em funções que podem despoletar a exploração de vulnerabilidades de inteiros; (3) correlação dos resultados das duas primeiras fases, apurando-se assim se há *bugs* que são realmente vulnerabilidades. Para que um *bug* de *software* seja uma *vulnerabilidade*, ou seja, para que seja atacável, *input* que entre no programa através de alguma das suas interfaces tem de chegar até esse *bug*. Exemplos dessas interfaces são o teclado, a rede/*sockets*, ficheiros e linha de comando. A ferramenta ilustra também como se podem combinar ferramentas generalistas previamente existentes de análise de código para detectar vulnerabilidades específicas.

Para avaliar a ferramenta *DEEEP* foi criado código com as vulnerabilidades que se pretendem descobrir. O artigo reporta a utilização da ferramenta com esse código, demonstrando a sua capacidade para fazer essa detecção. A ferramenta foi também usada para fazer a verificação de um pacote de *software* de código aberto emblemático, o *Sendmail*. Foi descoberta uma vulnerabilidade do tipo tratado no artigo nesse pacote.

O artigo tem três contribuições: (1) é o primeiro estudo sistemático das vulnerabilidades de inteiros causadas por uma má adaptação do código de 32 para 64 bits; (2) apresenta a ferramenta *DEEEP* que faz a detecção dessas vulnerabilidades; e (3) ilustra uma abordagem prática para a construção de ferramentas de análise estática de código fonte para detecção de vulnerabilidades específicas com base em ferramentas generalistas pré-existentes.

O artigo encontra-se organizado da seguinte forma. A secção 2 apresenta as categorias de vulnerabilidades de inteiros e os tipos de ataques que delas podem decorrer. A secção 3 dá uma visão das vulnerabilidades de inteiros que podem ser exploradas pela má adaptação de *software* de 32 bits para 64 bits. A secção 4 descreve a arquitectura e o funcionamento da ferramenta *DEEEP*. A secção 5 apresenta alguns resultados experimentais que demonstram a utilidade da ferramenta para a detecção de vulnerabilidades. A secção 7 conclui o artigo.

2 Vulnerabilidades de Inteiros

Nesta secção, em primeiro lugar são apresentados os tipos de vulnerabilidades de inteiros e seguidamente as categorias de ataques que exploram essas vulnerabilidades.

2.1 Tipos de Vulnerabilidades

Na linguagem de programação C os valores máximo e mínimo representáveis pelos tipos de dados inteiros dependem da representação do tipo na máquina (p.ex., complemento para 1 ou complemento para 2), se o tipo é com sinal (*signed*) ou sem sinal (*unsigned*), e da sua largura (16 bits, 32 bits,…) [2]. O artigo considera a representação de inteiros sempre em complemento para 2, já que é a representação usada nos processadores

AMD e Intel considerados.

Uma vulnerabilidade de inteiros aparece porque o programador não tem em conta os valores máximo e mínimo que uma variável inteira pode tomar, podendo levar a que um ou ambos os limites sejam ultrapassados. Existem quatro tipos de vulnerabilidades de inteiros: *overflow*, *underflow*, *signedness* e truncamento.

Overflow. Uma situação de *overflow* de inteiro ocorre quando o resultado de uma operação algébrica inteira (adição, subtracção, multiplicação ou divisão) ultrapassa o limite máximo permitido pelo tipo [2][3]. Por exemplo, o produto de dois inteiros sem sinal de 8 bits em geral requer 16 bits para ser representado, podendo existir por isso um *overflow* se a variável onde se tentar colocar o resultado tiver 8 bits. O valor máximo representável por uma variável *signed* de *n* bits em complemento para 2 é 2^{n-1}-1 e *unsigned* é 2^n-1.

Quando ocorre um *overflow,* o valor resultante é sempre menor do que o resultado correcto, independentemente de ter ou não sinal (*signed* ou *unsigned*) e da operação realizada. Caso o tipo seja *signed* e a operação uma soma de valores positivos, o valor resultante será negativo, tomando o valor da adição do valor mínimo do tipo de dados com a quantidade de unidades excedidas menos uma. Por outro lado, se o tipo for *unsigned* e a operação uma soma de dois valores positivos, o valor resultante será um inteiro positivo, obtido pelo resto da divisão inteira do número em causa pelo maior número inteiro possível do referido tipo inteiro, acrescido de uma unidade [4][5].

A figura 1(a) mostra uma vulnerabilidade *overflow* de inteiro na leitura e escrita do nome de um servidor. Um atacante pode explorar esta vulnerabilidade se fornecer um nome de um servidor (`server`) de comprimento igual a 254 ou 255 caracteres. A adição do comprimento do nome do servidor com duas unidades causa um *overflow*, ficando a variável `namelen` com o valor de 0 ou 1. Esse resultado é passado à função `malloc`, resultando numa reserva de memória demasiado pequena, gerando um *buffer overflow* na função `sprintf`.

Underflow. Um *underflow* de inteiro ocorre quando uma subtracção ou soma de inteiros ultrapassa o limite mínimo permitido para o tipo da variável onde é guardado o resultado [2] [3] (subtracção ou soma dependendo dos operandos terem o mesmo sinal ou sinais opostos). Por exemplo, a atribuição da subtracção 0-1 a um inteiro sem sinal de 16 bits não irá resultar no valor -1, mas sim em 2^{16}-1. O valor mínimo representável por uma variável de *n* bits *signed* em complemento para 2 é -2^{n-1} (*unsigned é* 0, obviamente).

O valor inteiro resultante é sempre maior do que o esperado, independentemente de ter ou não sinal (*signed* ou *unsigned*). Tanto no caso de variáveis com e sem sinal, o resultado da subtracção de dois valores positivos que gera um *underflow* é o valor máximo positivo permitido para o tipo inteiro menos o número de unidades excedidas, acrescido de uma unidade

A figura 1(b) apresenta uma vulnerabilidade deste tipo, a qual pode ser explorada quando o valor da variável `len` for 0 ou 1. Ocorrerá *underflow* de inteiro ao subtrair-se duas unidades ao valor de `len`, ficando `size` com um valor próximo de 4 GB. A chamada à função `malloc` irá tentar reservar 4 GB de memória, resultando numa possível negação de serviço (*DoS*), ou 0 *bytes* de memória reservados permitindo o um eventual atacante fazer um *buffer overflow* na função `memcpy`.

Signedness. Este tipo de vulnerabilidade está relacionado com a conversão implícita ou explícita entre tipos de dados inteiros *signed* e *unsigned*. O problema ocorre quando um tipo inteiro *signed* é interpretado como um *unsigned*, podendo, por exemplo, um número inteiro negativo ser convertido num número inteiro positivo grande. Por seu turno, o inverso, ou seja, a interpretação de um inteiro *unsigned* como um inteiro *signed* também pode gerar uma vulnerabilidade deste tipo, por exemplo, por um número inteiro positivo grande ser convertido num número negativo [3].

Na figura 1(c) a variável `len` (inteiro *signed*) representa o comprimento da *string* `buf`. Suponhamos que um atacante põe um valor negativo na variável `len`. Da operação de comparação resulta falso, pelo facto de `len` ser negativo, mas na chamada a `memcpy len` é implicitamente convertida de *signed* para *unsigned*, por o terceiro argumento da função `memcpy` ser do tipo *size_t* (*unsigned int*). Assim, o valor `len` é convertido num número inteiro positivo grande e dá-se um *buffer overflow* de `kbuf`.

Truncamento. A atribuição de um valor armazenado numa variável inteira com *n* bits a outra com menos bits (p.ex., respectivamente 64 e 32), resulta num truncamento, onde o valor da variável de maior largura é alterado se for maior do que o limite superior da variável de menor largura [3]. O truncamento é efectuado pela perda

dos bits mais significativos, resultando num valor final menor do que o esperado. Por exemplo, a conversão explícita (*cast*) de um *int* num *short*, levará à perda dos bits mais significativos do *int*, resultando em perda de informação se o valor do *int* for superior ao limite máximo do *short*.

Na figura 1(d) o valor da variável `cbBuf` (*int*) é atribuído à variável `cbCalculatedBufSize` (*unsigned short*). Essas variáveis têm respectivamente 32 e 16 bits no modelo IPL32. Se um atacante conseguir colocar em `cbBuf` um valor superior ao limite máximo do tipo *unsigned short* (65535), por exemplo 65568, `cbCalculatedBufSize` será truncada e ficará com o valor de 32. Este valor será utilizado na função `malloc` que reservará erradamente uma pequena quantidade de memória, que será extravasada quando escrita pela função `memcpy`.

```
int DoSomething(const char* server) {
    unsigned char namelen = strlen(server) + 2;
    if(namelen < 255) {
           char* UncName = malloc(namelen);
           if(UncName != 0)
                  sprintf(UncName, "\\\\%s", server);

           //do more things here
    }
    return 0;
}
```

(a)

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

(b)

```
int copy_something(char *buf, int len) {
    char kbuf[800];
    if (len > (int)sizeof(kbuf))
           return -1;

    return memcpy(kbuf, buf, len);
}
```

(c)

```
int func(char *name, int cbBuf) {
    unsigned short cbCalculatedBufSize = cbBuf;
    char *buf = (char*)malloc(cbCalculatedBufSize);
    if (buf) {
           memcpy(buf, name, cbBuf);
           //do more things here
           return 0;
    }
    return -1;
 }
```

(d)

Figura 1: Exemplos dos quatro tipos de vulnerabilidades de inteiros: (a) *overflow*; (b) *underflow* (*Netscape JPEG comment length integer under?ow vulnerability*); (c) *signedness*; (d) truncamento. Para cada uma está assinalada a linha onde está o erro de programação que a causa, e a função afectada.

2.2 Categorias de Ataques

A secção anterior apresentou um conjunto de vulnerabilidades de inteiros. Esta secção apresenta os ataques que podem ser desencadeados com essas vulnerabilidades. O *input* de um programa é armazenado em variáveis, que se directa ou indirectamente forem usadas para determinados fins – por exemplo em funções de manuseamento de memória – permitirão a realização de ataques bem sucedidos. As vulnerabilidades de inteiros podem ser exploradas por cinco categorias de ataques [2]:

Buffer Overflow. Um ataque deste tipo acontece quando o resultado de uma operação com inteiros com um dos quatro tipos de vulnerabilidades acima é utilizado em funções de reserva de memória (p.ex., *malloc*), reservando espaço insuficiente, sendo depois extravasado o tamanha disponível nessa zona e sobrescritas zonas de memória usadas para outros fins (v. figura 1(a), (b), (c), (d)).

***Negação de Serviço* (*DoS*).** Este tipo de ataque ocorre quando o valor resultante de uma vulnerabilidade de inteiro é maior do que o previsto e é utilizado em funções de manuseamento de memória, tais como *malloc* e *memcpy*, reservando ou copiando excessivo espaço de memória, levando ao esgotamento da mesma. O mesmo ataque também é bem sucedido se o resultado da vulnerabilidade de inteiro for utilizado em variáveis de controlo de ciclos causando ciclos infinitos (p.ex., figura 1(b)).

***Índice de Array*.** Um ataque deste tipo acontece quando o valor resultante de uma vulnerabilidade de inteiro (por exemplo, um valor negativo causado por um *overflow*) é utilizado como índice de um *array*, resultando em escrita de memória em endereços incorrectos.

***Contornar Verificação de Limites*.** A não verificação dos limites superior e inferior de determinada variável faz com que um atacante conhecedor de tais não verificações possa efectuar um ataque (v. p.ex. a figura 1(c)).

***Erros Lógicos*.** Acontecem quando uma vulnerabilidade de inteiro permite a um atacante manipular um ou mais objectos do programa, resultando em erros do mesmo.

3 Vulnerabilidades na Adaptação para LP64

O modelo de dados das aplicações de 32 bits definido pela ANSI, o ILP32, indica que os tipos de dados *int*, *long int* (ou simplesmente *long*) e ponteiro têm uma largura de 32 bits [4][6]. Assim, podem ser feitas atribuições entre estes três tipos de dados sem qualquer perda de dados e/ou valores inesperados. Mais, é sabido que os programadores tendem a ser descuidados e a fazer de facto essas atribuições, sobretudo entre os tipos *int* e *long* [19].

No caso das arquitecturas de 64 bits existem três modelos de dados: LP64, LLP64 e ILP64. O primeiro é o adoptado nas diversas variantes do Unix, incluindo o Linux e o *software de código aberto* em geral. O modelo LLP64 é o adoptado pela Microsoft e o ILP64 é adoptado por alguns *mainframes* e supercomputadores, como os Cray.

O modelo de dados LP64 define que os tipos de dados *long int* e ponteiro têm um comprimento de 64 bits, enquanto que o tipo *int* mantém os 32 bits de largura [6][2]. Uma constatação imediata é que neste modelo, ao invés do que acontece no ILP32, a atribuição de um ponteiro ou um *long int* a um *int* gera perda de dados.

Na portabilidade de aplicações de 32 bits para 64 bits considerada neste artigo, ou seja, entre os modelos de dados ILP32 e LP64, é geralmente necessário que sejam efectuados uma série de ajustes ao código para que não ocorra, por exemplo, truncamento de dados em instruções onde um *long int* é atribuído a um *int*. Este tipo de instrução *int = long int* não causa um truncamento no modelo de dados ILP32, mas no modelo de dados LP64 o mesmo não se verifica. Assim, tem-se uma das vulnerabilidades listadas acima e se as variáveis nela envolvidas nessa atribuição forem acessíveis a um atacante, está-se perante uma vulnerabilidade que pode

[2] No modelo LLP64 (Microsoft), os tipos *int* e *long* mantêm os 32 bits, o *long long* tem 64 bits e os apontadores 64 bits. No modelo ILP64 os tipos *int*, *long* e apontador têm todos 64 bits. Os três modelos definem ainda os tipos *char* e *short* mas esses tipos têm respectivamente 8 e 16 bits em todos os modelos.

ser explorada por um dos tipos de ataque acima apresentados.

É então intuito deste trabalho apurar a existência de vulnerabilidades de inteiros em aplicações que são portadas de ILP32 para LP64, sem os cuidados necessários para evitar estes problemas. Assim sendo, esta secção encontra-se dividida pelos tipos de vulnerabilidades de inteiros.

3.1 *Overflow* e *Underflow* de Inteiro

A ocorrência de *overflow* e *underflow* de inteiros está relacionada com operações aritméticas nas quais os resultados ultrapassam os limites dos tipos inteiros. Quando se adapta mal código de ILP32 para LP64, as vulnerabilidades de *overflow* e *underflow* de inteiros mantêm-se, podendo no entanto o problema ocorrer em LP64 com valores superiores aos de ILP32.

Assim, na operação adição (acontecendo o mesmo na multiplicação) ocorre uma vulnerabilidade de inteiro nas instruções:

- *int = int + int*, em ambos os modelos para os mesmos valores, pelo facto de *int* ter comprimento de 32 bits em ambos os modelos de dados;
- *long = long + int* ou *long = long + long*, ocorre em ambos os modelos, mas em LP64 os valores de *long* serão diferentes dos de ILP32, pelo facto de *long* ter largura diferente em ambos os modelos de dados.

Na divisão uma vulnerabilidade de *overflow* de inteiro mantém-se quando se adapta o código para LP64, uma vez que esta só ocorre quando um dos operandos da divisão for um dos limites do tipo inteiro e o outro for o valor -1 e o tipo da variável receptora for o do operando de valor igual a um dos limites. Assim, independentemente do tipo inteiro, constata-se sempre vulnerabilidade quando:

- *MIN signed / -1*, em ambos os modelos de dados, pelo facto do valor resultante exceder o limite superior do tipo da variável receptora;
- *unsigned / -1*, onde *unsigned* é maior do que o máximo valor positivo do tipo *signed* do mesmo tipo. Em ambos os modelos de dados existe vulnerabilidade, porque o valor *unsigned* será convertido para *signed*, obtendo-se um valor superior ao máximo permitido do valor *signed* e consequentemente resultando um valor negativo;
- *-1 / MAX unsigned*, em ambos os modelos de dados, pelo facto de ser realizada, em primeiro lugar, uma promoção do tipo *signed* para *unsigned* para o valor -1, convertendo-se o referido valor para um inteiro positivo grande.

Por fim, relativamente à vulnerabilidade *underflow* associada à subtracção, esta existe nos casos:

- *int--*, em ambos os modelos de dados;
- *long--*, em ambos os modelos de dados, quando o valor de *long* for igual ao menor valor possível para o tipo no modelo de dados;
- *long = long - int*, *ou long = long - long*, ocorre em ambos os modelos, mas em LP64 os valores de *long* serão diferentes dos de ILP32, pelo facto de *long* ter largura diferente nos dois modelos.

Para além das operações aritméticas que estão na base destes tipos de vulnerabilidades de inteiros, as operações de *bit shift* e de ponteiros também podem originar estes tipos de vulnerabilidades:

- *bit shift*, quando deslocamos (*shift*) o valor 1 do tipo *int* de 32 bits, ocorre um *overflow* e o resultado é zero. O problema ocorre nos dois modelos de dados, já que o tipo *int* tem o mesmo comprimento. Contudo, se o tipo inteiro for *long* e o número de *bits* a deslocar forem os mesmos 32, continuaríamos a verificar a vulnerabilidade em ILP32, mas não em LP64 [8];
- *ponteiros*, uma vez que estes são interpretados como um *unsigned int*, do tipo *size_t*, cujo tamanho depende da plataforma, as vulnerabilidades de inteiros decorrentes das operações aritméticas com inteiros são as mesmas na aritmética de ponteiros.

3.2 Signedness

Os problemas ocorridos em promoção de tipos de dados e/ou conversão entre tipos *signed* e *unsigned*, que resultam nas vulnerabilidades de *signedness*, mantêm-se na adaptação de ILP32 para LP64, nomeadamente nas seguintes situações:

- conversão de um *signed* para *unsigned* e vice-versa;
- variáveis representativas de tamanho de *buffers*, índices de *arrays* e contadores de ciclos do tipo *signed*, ao invés de *unsigned*;
- comparação entre variáveis *signed* e *unsigned*.

3.3 Truncamento

Um truncamento de dados acontece quando se atribui uma variável de maior comprimento a uma variável de menor comprimento. Quando se adapta código de ILP32 para LP64, como os tipos de dados *long* e ponteiro têm tamanhos diferentes nos dois modelos dá-se truncamento de dados nos seguintes casos [8] [9]:

- atribuição de um *long* a um *int*;
- atribuição de um ponteiro a um *int*;
- atribuição do valor retornado por uma função, podendo ser um *long* ou um ponteiro, a um *int*;
- comparação de um *int* com um ponteiro;
- atribuição de um **long* a um **int*;
- atribuição do resultado de um deslocamento de bits (*bit shift*), efectuado sobre um *long*, a um *int*;
- atribuição pela função *scanf()* de um *long* a uma variável usando o formato "%d";
- escrita num *buffer*, pela função *sprintf()*, de um *long* usando o formato "%d".

4 Ferramenta *DEEEP*

Com o objectivo de detectar vulnerabilidades causadas pela má adaptação de aplicações de ILP32 para LP64 foi concebida a ferramenta de análise estática de código *DEEEP*, tendo por base as ferramentas *Lint* e *Splint* [10] [11], sobre o sistema operativo *Open Solaris*.

A ferramenta efectua análise semântica de código fonte. Mais concretamente, faz verificação de tipos de dados (*type checking*), análise de fluxo de dados (*data flow analysis*), e permite correlacionar o resultado dessas duas formas de análise. A primeira forma de análise tem o intuito de detectar *bugs* na manipulação de inteiros, enquanto que a segunda analisa o fluxo de dados de forma a detectar se dados vindos de fora do programa atingem funções de biblioteca perigosas, como *memcpy* ou *strcpy.* Após estas duas análises, a ferramenta faz o correlacionamento dos resultados das fases anteriores de forma a perceber se dados que vêm dos *inputs* do programa são afectados pelas vulnerabilidades da adaptação para LP64, e depois chegam a funções de biblioteca perigosas (*memcpy, strcpy,…*). Se isso acontecer, deixa-se de ter simples *bugs* e passa-se a ter vulnerabilidades, atacáveis em certas circunstâncias.

4.1 Arquitectura da Ferramenta

A figura 2 apresenta a arquitectura da ferramenta *DEEEP*, onde se destacam as suas quatro principais componentes: *Pré-Processador*, *Detector de Bugs*, *Analisador de Fluxo de Dados* e *Visualizador/ Correlacionador*. O funcionamento é resumidamente o seguinte. O Pré-Processador faz uma primeira compilação do programa que se pretende analisar usando a correspondente *Makefile.* O objectivo consiste em obter as opções que é necessário passar ao compilador, já que é também necessário passá-las ao *Lint* e ao *Splint.* Os resultados deste componente são passados ao Detector de Bugs que executa o *Lint* e o *Splint* para fazer verificação de tipos, descobrindo os *bugs* listados na secção 3. Como essas ferramentas detectam muitos outros tipos de *bugs*, os resultados têm de ser filtrados pelo filtro representado na figura. Em paralelo, no Analisador de Fluxo de Dados o *Splint* faz análise de fluxo de dados de forma a descobrir as funções perigosas nas quais são inseridos dados vindos dos *inputs* do programa, mesmo que passados entre variáveis,

combinados, etc. (uma forma de análise também denominada de *taint analysis*). Por *funções perigosas* pretende-se designar as funções das bibliotecas C que são passíveis de ser atacadas com os ataques listados na secção 2.2 como resultado das vulnerabilidades listadas na secção 3. O resultado desta análise é filtrado para descartar resultados que não estejam relacionados com *bugs* de inteiros. Por fim, o Visualizador/Correlacionador combina os resultados das duas análises (detalhes abaixo).

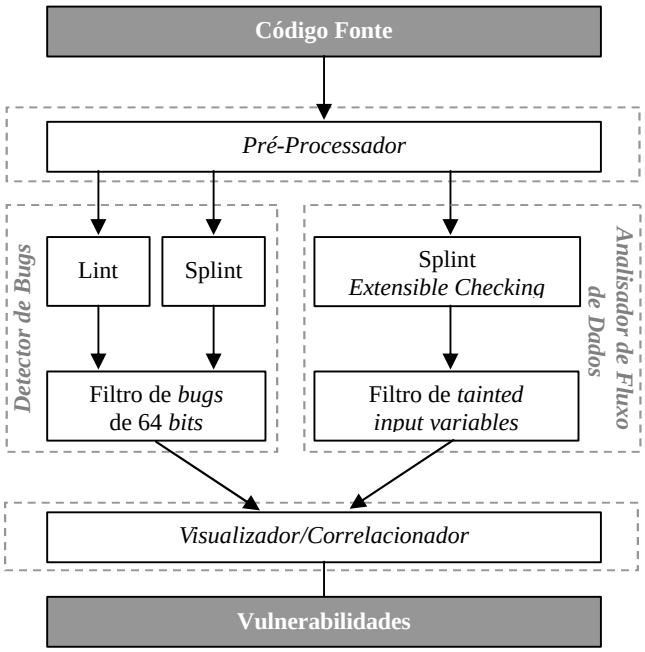

Figura 2: Arquitectura da ferramenta *DEEEP*.

4.2 Pré-Processador

Para que as ferramentas *Lint* e *Splint* possam funcionar e analisar correctamente código fonte é necessário fornecer-lhes as mesmas opções de linha de comando passadas ao compilador. Estas opções são utilizadas para definir ou anular constantes que serão passadas ao pré-processador C, das quais as necessárias para as referidas ferramentas são: D (*Define*), para definir constantes; U (*Undefine*), para anular constantes; e I (*Include*), para incluir caminhos (*paths*) de directórios que conterão as bibliotecas necessárias à compilação e não existentes na *path standard* [11].

Para obtenção dessas opções, que muitas vezes diferem de ficheiro para ficheiro dentro de uma aplicação, é utilizado o ficheiro *Makefile* da aplicação, o qual contém, explicitamente ou implicitamente, as linhas de comando para o compilador *gcc* ou *cc*, que contêm as directivas D, U, e/ou I necessárias às ferramentas *Lint* e *Splint*. Para extracção dessas opções é necessário executar o *Makefile*, uma vez que as linhas de comando de compilação podem não estar nele explícitas. Assim, para o efeito, foi construído um pequeno programa em linguagem *Perl* que é passado como parâmetro ao comando *make*, por forma a envolver a execução do *Makefile*, capturando as linhas de comando para o compilador, extraindo as directivas necessárias e criando as linhas de comando para as ferramentas de análise estática de código referidas.

4.3 Detector de Bugs

O *Detector de Bugs* tem por objectivos detectar e devolver as linhas de código da aplicação que necessitam de reajustes para que sejam correctamente portadas de 32 para 64 bits, bem como as mensagens de aviso (*warnings*) correspondentes.

A detecção desses problemas é feita pelas ferramentas *Lint* e *Splint* parametrizadas com os resultados do pré-processador. A escolha da ferramenta *Lint* é justificada por esta conter um parâmetro que permite detectar os *bugs* de 64 bits, excepto os de *signedness* [10]. Por sua vez a ferramenta *Splint* permite colmatar essa lacuna, mas não detecção da atribuição de um ponteiro a um *int* (por exemplo, *int = *void*), a qual é feito pelo *Lint* [11].

As duas ferramentas devolvem muitas outras mensagens de aviso, para além das relacionadas com os *bugs* que interessam a este estudo (v. secção 3). Por essa razão, os resultados têm de ser filtrados, de modo a obterem-se somente as linhas de código e os avisos relativos aos *bugs* relevantes. Assim, e para que sejam filtradas as mensagens correctamente, primeiramente foi construída uma base de dados de texto com todas as mensagens que interessam para o estudo, tendo sido necessário para o apuramento das mesmas executar as duas ferramentas em código contendo todos esses *bugs*.

A execução desta fase de processamento é iniciada com a execução da ferramenta *Lint* e filtragem dos seus resultados, seguindo-se a execução da ferramenta *Splint* com filtragem de resultados. Apesar das ferramentas serem executadas em sequência, as execuções são independentes logo conceptualmente são feitas em paralelo como representado na figura 2.

As duas ferramentas fazem verificação de tipos (*type checking*), detectando as linhas do código que não respeitam a relação entre os tamanhos dos tipos de dados inteiros no modelo de dados LP64 (*sizeof(int)* < *sizeof(long)* <= *sizeof(pointer)*), e emitindo mensagens de aviso para cada uma.

4.4 Analisador de Fluxo de Dados

Nesta componente é utilizada a ferramenta *Splint* para efectuar a análise de fluxo de dados (*data flow analysis*), ou melhor, uma forma de análise deste tipo denominada *taint analysis*. A ferramenta *Splint* para além de fazer um conjunto de verificações por omissão (as usadas pelo Detector de Bugs), fornece mecanismos que permitem ao utilizador definir novos tipos de verificações e de anotações para a detecção de vulnerabilidades ou violação de propriedades específicas das aplicações, sendo estes mecanismos denominados por *extensible checking* [12]. Estas verificações podem ser descritas por condições sobre atributos associados a objectos do programa ou ao estado global de execução. No entanto, ao contrário dos tipos, os valores destes atributos podem mudar ao longo da execução do programa. O *Splint* dá ao utilizador a possibilidade de definir atributos associados com tipos diferentes de objectos do programa, bem como as regras de transferência que fazem que os valores dos atributos mudem.

Neste sentido e para o presente estudo no qual é preciso estipular que todos os dados provenientes do exterior podem ser maliciosos, foram criados dois atributos, *inputness* e *inputness1*. O primeiro está associado aos objectos retornados pelas funções de *input*, onde as fontes de *input* consideradas são: teclado (*stdin*), ficheiros, *sockets* e linha de comando. O segundo atributo está associado à passagem de variáveis de *input* em funções *perigosas*. Para ambos foram definidas as anotações *inputtainted* e *inputuntainted* para indicar hipóteses sobre o *inputness* de uma referência.

A definição completa dos atributos é apresentada na figura 3. As primeiras linhas de cada um indicam que o atributo está associado aos objectos dos tipos referenciados na cláusula *context* (*int*, etc) e que pode ter um de três estados: *tainted*, *untainted* e *nostate* (para *inputness*) e *tainted* e *untainted* (para *inputness1*), onde *tainted*, *untainted* e *nostate* significam respectivamente comprometido, não comprometido e desconhecido. A cláusula *transfers* (respectivamente linhas 7-9 e 7-8 de *inputness* e *inputness1*) especifica as regras de transferência de objectos entre referências. Por exemplo, em *inputness*, a regra *tainted as untainted ==> error* indica que a mensagem "*Tainted input variable or integer passed as untainted*" será apresentada quando um objecto de estado *tainted* for transferido para uma referência declarada como *untainted*. Esta situação ocorre se um objecto no estado *tainted* é passado como um parâmetro *untainted* ou retornado como um resultado *untainted*. A cláusula *merge* indica o estado do objecto resultante da combinação de dois objectos. Na figura, esta cláusula indica que qualquer objecto, independente do seu estado, combinado com um objecto de estado *tainted* produz um objecto de estado *tainted*. A cláusula *annotations* define as anotações que podem ser utilizadas em declarações para documentar hipóteses para os atributos *inputness* e *inputness1*. Na execução da análise de código, cada anotação utilizada como declaração será substituída pelo seu respectivo estado. Por exemplo, a anotação *inputtainted reference ==> tainted* indica que uma referência declarada com a anotação *inputainted* terá estado *tainted*. Por fim, a cláusula *defaults* especifica os estados que serão utilizados por omissão para declarações que não estejam anotadas.

```
1   attribute inputness
2     context reference /* type int and unsigned int and long int and
            unsigned long and size_t and char * and void * and
            wchar_t * and wint_t and struct msghdr * */
3     oneof tainted, untainted, nostate
4     annotations
5           inputtainted reference ==> tainted
6           inputuntainted reference ==> untainted
7     transfers
8           tainted as nostate ==> error "Tainted input variable
                                    passed to function/procedure"
9           tainted as untainted ==> error "Tainted input variable
                                    or integer passed as untainted"

10    merge
11          tainted + * ==> tainted
12    defaults
13          reference ==> nostate
14  end
```

(a)

```
1   attribute inputness1
2     context reference /* type int and unsigned int and long int and
            unsigned long and size_t */
3     oneof tainted, untainted
4     annotations
5           inputtainted reference ==> tainted
6           inputuntainted reference ==> untainted
7     transfers
8           tainted as untainted ==> error "Possible tainted integer
                                    passed as untainted"
9     merge
10          tainted + * ==> tainted
11    defaults
12          reference ==> tainted
13  end
```

(b)

Figura 3: Definição dos atributos. (a) *inputness*. (b) *inputness1*.

Após definição dos atributos, foi necessário documentar com as anotações *inputtainted* e *inputuntainted* as funções que interessam para o estudo. Assim, os valores retornados por todas as funções de *input* foram anotados com estado *inputtainted*, indicando que o valor contido na variável retornada poderá ser malicioso (estar comprometido). Por outro lado, todos os argumentos inteiros de entrada das funções de reserva de memória, de escrita em memória, ficheiros e *sockets* foram anotados com o estado *inputuntainted*, obrigando a que esses argumentos sejam *untainted*, ou seja, não estejam comprometidos. Esses dois conjuntos de anotações, de acordo com as regras de transferência (*transfers*) estipuladas nos atributos, garantem que é capturada a passagem de objectos *tainted* (variáveis de *input*) em argumentos *untainted* ou mesmo em argumentos *nostate*.

A figura 4(a) apresenta a título de exemplo as anotações das funções *fgets* e *getchar*. Na primeira é garantido que o estado do argumento de retorno *s* é *tainted*, mas o estado do argumento de entrada *n* (número de caracteres a serem lidos da *stream*) é esperado que seja *untainted*, de modo a assegurar que o valor de *n* não seja uma vulnerabilidade de inteiro. Na função *getchar* o estado do tipo *int* retornado é *tainted*.

Nas funções de manuseamento de memória (figura 4(b)), o estado requerido para os seus argumentos de entrada é *untainted*, uma vez que ambas são funções passíveis de exploração de vulnerabilidades de inteiros.

Assim, é possível detectar se o valor de uma variável de *input* é parâmetro de entrada de uma função de manuseamento de memória.

```
char *fgets (/*@returned@*/ char *s,
             /*@inputuntainted@*/ int n,
             FILE *stream)
/*@ensures inputtainted s@*/;

/*@inputtainted@*/ int getchar(void)
/*@ensures inputtainted@*/;
```

(a)

```
void *malloc (/*@inputuntainted@*/ size_t size);

void memcpy (/*@returned@*/ void *s1,
             void *s2,
             /*@inputuntainted@*/ size_t n);
```

(b)

Figura 4: Exemplos de anotação de funções. (a) Funções de *input*. (b) Funções de manuseamento de memória.

A ideia do funcionamento dos atributos *inputness* e *inputness1* é a seguinte: no início da análise todos os objectos não anotados têm estado *nostate* (cláusula *defaults* do atributo *inputness* – fig. 3(a)); todos os objectos de valores provenientes do exterior do sistema e lidos por alguma função de *input* anotada terão estado *tainted*, bem como todos os objectos cujo seu valor é calculado com base em objectos *tainted*; por cada regra de transferência de estados de objectos aplicada é emitida a mensagem correspondente; finda a sinalização das variáveis de *input* e a sua passagem em procedimentos não anotados e em *funções perigosas*, a análise é reiniciada, tendo todos os objectos não anotados o estado *tainted* (cláusula *defaults* do atributo *inputness1* – fig. 3(b)); por cada transferência de objectos *tainted* em parâmetros *untainted* é emitida a mensagem correspondente.

A análise realizada com o atributo *inputness* assegura que todos valores de entrada são marcados como *tainted*, que os valores de variáveis calculados a partir de variáveis de entrada são marcados também *tainted*, e que sempre que os mesmos sejam passados por referência em funções cujos argumentos de entrada requerem valores *untainted* será emitido um aviso. Também assegura que se estes valores forem passados por referência em procedimentos cujos argumentos sejam *nostate* será emitido um aviso. Desta forma, pode-se analisar o fluxo de dados das variáveis de entrada e capturar as linhas de código onde se poderá dar a exploração de vulnerabilidades e a chamada de procedimentos criados pelo programador. Por seu turno, a análise realizada com o atributo *inputness1* assegura a análise dentro dos procedimentos criados pelo programador, capturando os lugares onde há passagem de valores possivelmente *tainted* em locais *untainted*, emitindo uma mensagem, e permitindo detectar vulnerabilidades nestes procedimentos.

4.5 Visualizador/Correlacionador

O objectivo da quarta componente da ferramenta *DEEEP* é correlacionar os resultados das duas componentes explicadas anteriormente (v. figura 2). Para o efeito, foi criado um programa em linguagem *Perl* (*programa correlacionador*) que, com base nos ficheiros resultantes das componentes "Detector de Bugs" e "Analisador de Fluxo de Dados", faz o cruzamento dos resultados e apresenta, sob a forma de um ficheiro, as possíveis vulnerabilidades de inteiros detectadas pela ferramenta.

Os resultados da detecção feita na terceira fase mostram as passagens de variáveis *tainted* a parâmetros de funções que têm de ser *untainted/nostate* (atributo *inputness*), ou passagens do estado *tainted* para *untainted* (atributo *inputness1*). Tais detecções são identificadas pelo *programa correlacionador* pela linha de código onde é inicializada a variável *Li* e pela linha onde ocorre a passagem da variável na chamada de funções/procedimentos *Lv*. Tendo como ponto de partida os resultados do atributo *inputness*, representativos da passagem de variáveis de *input* (variáveis *tainted*) ou outras, cujo valor é calculado com base em variáveis de *input*, em funções/procedimentos, temos que: numa detecção de passagem de variável *tainted* em lugares

untainted, será verificado, nos resultados da segunda fase de processamento, se no bloco de linhas de código delimitado pelos dois números de linha *Li* e *Lv* há algum aviso de *bug* envolvendo a variável que causou a detecção na fase três. Se tal se verificar está-se com grande probabilidade na presença de uma vulnerabilidade de inteiro; numa detecção de passagem de variável *tainted* em lugares *nostate*, será verificado, nos resultados da segunda fase de processamento, se a variável detectada na passagem de estados está envolvida em algum aviso de *bug*. Como neste tipo de detecção (*tainted as nostate*) estamos na presença de chamada de procedimentos criados pelo programador, poderá não se verificar o envolvimento da variável de *input* em avisos de *bugs* de 64 bits, no bloco de linhas de código delimitado por *Li* e *Lv*, mas este poderá ainda ocorrer dentro do procedimento. Identificado então o procedimento da detecção de passagem de estados, será verificado, nos resultados do atributo *inputness1*, se o mesmo lá consta. Se assim acontecer, para cada detecção de passagem de variável *tainted* em parâmetros *untainted* (de *funções perigosas*), será verificado, nos resultados da segunda fase de processamento, se a variável em causa está envolvida em algum aviso de bug, entre as linhas de código delimitadas por *Li* e *Lv*. Se tal se verificar está-se possivelmente na presença de uma vulnerabilidade de inteiro, contida dentro de um procedimento criado pelo programador.

Com este cruzamento dos resultados pode-se visualizar o percurso de uma variável *tainted* até à entrada numa função que requer argumentos *untainted* e/ou *nostate* e se, ao longo do seu percurso, ocorre algum dos *bugs* de má adaptação de 32 para 64 bits. Se isso acontecer está-se na presença de uma vulnerabilidade.

5 Experimentação da Ferramenta

Esta secção mostra resultados obtidos da experimentação com a ferramenta *DEEEP*. O código verificado numa primeira fase foi código sintético, escrito para o efeito, contendo as diversas vulnerabilidades que se pretendem descobrir. Posteriormente a *DEEEP* foi experimentada em *software de código aberto*.

5.1 Exemplos de Detecção

A figura 5 apresenta código que quando mal portado para 64 bits fica com vulnerabilidades de inteiros. A figura 6 é o resultado da segunda fase de processamento sobre o código fonte da figura 5, ou seja, a identificação das linhas de código que contêm *bugs* de manipulação de inteiros e as respectivas mensagens de aviso. A figura 7 contém o resultado da terceira fase de processamento sobre o mesmo código fonte. Por fim, a figura 8 é o resultado da correlação, efectuada pelo *programa correlacionador*, contendo as vulnerabilidades de inteiros e apresentado ao utilizador.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()  {
5          char *src= "8 teste";
6          char *buf;
7          unsigned long len;
8          unsigned int size;
9          len = getchar();
10         printf("%d", len);
11
12         size = len;
13         buf = (char*)malloc(size);
14         if (buf)
15                memcpy(buf, src, len);
16         return 0;
17  }
```

Figura 5: Código fonte vulnerável.

```
9   Sign extension from 32-bit to 64-bit integer
9   Assignment of int to unsigned long int: len = getchar()
10  Function argument type inconsistent with format:
    printf(arg 2) unsigned long and (format) int
12  Assignment of 64-bit integer to 32-bit integer
13  Function malloc expects arg 1 to be size_t gets unsigned int: size
15  Function memcpy expects arg 3 to be size_t gets unsigned long int: len
```

Figura 6: Resultados do Detector de Bugs.

```
(in function main)

10: Invalid transfer from implicitly tainted len to implicitly
    nostate
    (Tainted input variable passed to function/procedure):
           printf(..., len, ...)
     9: len becomes implicitly tainted

13: Invalid transfer from implicitly tainted size to untainted
    (Tainted input variable or integer passed as untainted):
           malloc(..., size, ...)
    12: size becomes implicitly tainted

15: Invalid transfer from implicitly tainted len to untainted
    (Tainted input variable or integer passed as untainted):
           memcpy(..., len, ...)
     9: len becomes implicitly tainted
```

Figura 7: Resultados do Analisador de Fluxo de Dados.

Uma análise dos resultados obtidos pela correlação (fig. 8), ou seja, das vulnerabilidades de inteiros detectadas pelo cruzamento das figuras 6 e 7, permite observar o seguinte:

- Na primeira vulnerabilidade detectada pode-se observar uma passagem inválida do estado *tainted* para *untainted* na chamada à função *malloc*, onde a variável *size* (*tainted* pois vem da variável *len* que foi lida do teclado) é passada a um argumento que tem de ser *untainted* (v. figura 4(b)). Pode-se também observar que a variável *size* existe em avisos de *bugs* de 64 bits, entre as linhas 12 e 13 (*Li* e *Lv*, respectivamente). Observa-se de facto a existência de um aviso de truncamento de dados (atribuição de um inteiro de 64 bits a um inteiro de 32 bits) na variável *size*, logo o valor esperado para *size* é menor do que o esperado. Assim, quando *size* é passado à função *malloc* (linha 13) esta pode reservar menos espaço do que o necessário e esperado. Trata-se de facto de uma vulnerabilidade originada por má adaptação de 32 para 64 bits, que foi correctamente detectada.
- Na segunda vulnerabilidade detectada verifica-se a passagem da variável *len* que é *tainted* num parâmetro *untainted* da função *memcpy*. Pode-se observar, nos avisos de bugs de 64 bits, que a variável *len* aparece na linha 12, mas apesar disso não é afectada por um *bug* na manipulação de inteiros. Existe de facto uma vulnerabilidade na linha 15 mas é um *buffer overflow* convencional, no qual uma variável vinda do *input* (teclado neste caso) é passada como comprimento da zona de memória a copiar. De salientar, contudo, que o *buffer overflow* é consequência da reserva de espaço insuficiente causada pela primeira vulnerabilidade detectada.

De referir também que, ao observar os resultados das figuras 7 e 8, a primeira violação (linhas 9 e 10) da figura 7 não está patente na figura 8, por a função *printf* não ser relevante para o tipo de vulnerabilidades e ataques aqui estudados (secções 2 e 3), independente de se verificar uma passagem inválida do estado *tainted* para *nostate* na chamada à função *printf*, ao passar a variável *len* num argumento *nostate*.

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ +             Possible Vulnerability N. 1               + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

Source Code: art_trunc_1.c
============
12  size = len;
13  buf = (char*)malloc(size);

Bugs of 64 bits:
================
12  Assignment of 64-bit integer to 32-bit integer
13  Function malloc expects arg 1 to be size_t gets unsigned int: size

Taint analysis:
===============
(in function main)
13: Invalid transfer from implicitly tainted size to untainted
    (Tainted input variable or integer passed as untainted):
            malloc(..., size, ...)
    12: size becomes implicitly tainted


+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ +             Possible Vulnerability N. 2               + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

Source Code: art_trunc_1.c
============
 9  len = getchar();
10  printf("%d", len);
12  size = len;
15        memcpy(buf, src, len);

Bugs of 64 bits:
================
9   Sign extension from 32-bit to 64-bit integer
9   Assignment of int to unsigned long int: len = getchar()
10  Function argument type inconsistent with format:
    printf(arg 2) unsigned long  and (format) int  in art_trunc_1.c(10)
12  Assignment of 64-bit integer to 32-bit integer
15  Function memcpy expects arg 3 to be size_t gets unsigned long int: len

Taint analysis:
===============
(in function main)
15: Invalid transfer from implicitly tainted len to untainted
    (Tainted input variable or integer passed as untainted):
            memcpy(..., len, ...)
    9: len becomes implicitly tainted
```

Figura 8: Resultados do Visualizador/Correlacionador.

Para dar uma visão da análise realizada na chamada e dentro de funções, a figura 9 apresenta o código da figura 5 modificado de forma a conter uma função. As figuras 10 e 11 apresentam os resultados da segunda e terceira fases de detecção, enquanto que a figura 12 apresenta as vulnerabilidades detectadas pelo *programa correlacionador*.

Uma análise dos resultados da figura 12, ou seja, a vulnerabilidade detectada mostra o seguinte:

- Função *main*, violação entre as linhas 17 e 19: pode-se observar na chamada à função *string_copy* a

passagem de uma variável de *input tainted*, ficando a variável do segundo argumento da função com estado *tainted*. Como a função *string_copy* foi definida pelo programador os seus argumentos não têm qualquer anotação, e o seu estado é o por omissão (*nostate*). Para garantir e capturar a passagem de variáveis de estado *tainted* em funções deste tipo é definida a regra de transferência *tainted as nostate ==> error*, indicando a passagem de variáveis de *input* em funções criadas pelo programador.

- Função *string_copy*, violação nas linhas 5 e 6: análise efectuada dentro da função *string_copy*, onde os estados dos seus argumentos tomam o valor do estado por omissão *tainted* (estado por defeito do atributo *inputness1*), significando que pelo menos um dos valores dos estados dos seus argumentos são os da sua chamada, o detectado pelo atributo *inputness*. Isto é visível porque o valor da variável *size* é calculado com base na variável *len*, ou seja, o estado de *size* é igual ao de de *len* (*tainted*), uma vez que o estado de *len* o é na chamada da função. Verifica-se então que a variável *size* (linha 5) é obtida por truncamento de dados (bug de 64 bits) e utilizada como argumento da função *malloc* (linha 6). Ou seja, *size* é resultado da vulnerabilidade de inteiro truncamento que é passada para uma função de manuseamento de memória, reservando menos espaço do que o esperado.
- Função *string_copy*, violação entre as linhas 4 e 8: passagem de uma variável de estado *tainted* num argumento da função *memcpy* que requer *untainted*. Neste ponto é visível que a variável de entrada *len* explora a vulnerabilidade de inteiro truncamento. Também é visível que a mesma variável mantém o estado *tainted*, desde a chamada da função.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int string_copy(char *src, unsigned long len) {
5          unsigned int size = len;
6          char *buf = (char*)malloc(size);
7          if (buf) {
8                 memcpy(buf, src, len);
9                 return 0;
10         }
11         return -1;
12  }
13
14  int main() {
15         char *src= "8 teste";
16         unsigned long len;
17         len = getchar();
18         printf("%d", len);
19         string_copy(src, len);
20         return 0;
21  }
```

Figura 9: Código fonte com função vulnerável.

```
5   Assignment of 64-bit integer to 32-bit integer
6   Function malloc expects arg 1 to be size_t gets unsigned int: size
8   Function memcpy expects arg 3 to be size_t gets unsigned long int: len
17  Sign extension from 32-bit to 64-bit integer
17  Assignment of int to unsigned long int: len = getchar()
18  Function argument type inconsistent with format:
    printf(arg 2) unsigned long  and (format) int
```

Figura 10: Resultados do Detector de Bugs.

```
(in function string_copy)
6: Invalid transfer from implicitly nostate size to untainted
   (Possible tainted input variable or integer passed as untainted):
          malloc(..., size, ...)
    5: size becomes implicitly nostate

8: Invalid transfer from implicitly nostate len to untainted
   (Possible tainted input variable or integer passed as untainted):
          memcpy(..., len, ...)
    4: len becomes implicitly nostate

(in function main)
18: Invalid transfer from implicitly tainted len to implicitly
    nostate
    (Tainted input variable passed to function/procedure):
          printf(..., len, ...)
    17: len becomes implicitly tainted

19: Invalid transfer from implicitly tainted len to implicitly
    nostate
    (Tainted input variable passed to function/procedure):
          string_copy(..., len, ...)
    17: len becomes implicitly tainted
```

Figura 11: Resultados do Analisador de Fluxo de Dados.

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ +             Possible Vulnerability N. 1                 + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

Source Code: art_trunc.c
============
17  len = getchar();
18  printf("%d", len);
19  str_copy(src, len);

Bugs of 64 bits:
================
17  Sign extension from 32-bit to 64-bit integer
17  Assignment of int to unsigned long int: len = getchar()
18  Function argument type inconsistent with format:
    printf(arg 2) unsigned long  and (format) int  in art_trunc.c(18)

Taint analysis:
===============
(in function main)
19: Invalid transfer from implicitly tainted len to implicitly nostate
    (Tainted input variable passed to function/procedure):
           string_copy(..., len, ...)
    17: len becomes implicitly tainted


>>>> Function Source Code: art_trunc.c --> string_copy()
     ====================
     4     int string_copy(char *src, unsigned long len) {
     5           unsigned int size = len;
     6           char *buf = (char*)malloc(size);
     8                 memcpy(buf, src, len);
```

Figura 12: Resultados do Visualizador/Correlacionador.

```
>>>> Bugs of 64 bits:
     ================
     5 Assignment of 64-bit integer to 32-bit integer
     6 Function malloc expects arg 1 to be size_t gets unsigned int: size
     8 Function memcpy expects arg 3 to be size_t gets unsigned long int: len

>>>> Taint analysis:
     =================
     (in function str_copy)
     6: Invalid transfer from implicitly tainted size to untainted
        (Possible tainted integer passed as untainted):
                  malloc(..., size, ...)
        5: size becomes implicitly tainted

     8: Invalid transfer from implicitly tainted len to untainted
        (Possible tainted integer passed as untainted):
                  memcpy(..., len, ...)
        4: len becomes implicitly tainted
```

Figura 12: Resultados do Visualizador/Correlacionador (continuação).

À semelhança do exemplo anterior, observa-se também que apesar de na figura 11 constar a detecção de passagem de estados na função printf (linhas 17 e 18), a mesma não é apresentada no output final, pela razão mencionada no exemplo anterior.

5.2 Detecção no *Sendmail*

Para a experimentação da ferramenta em programas *de código aberto* optou-se pelo clássico *Sendmail* 8.14.1, uma implementação do protocolo SMTP (*Simple Mail Transfer Protocol*) para transmissão de mensagens de *e-mail* [13].

Analisando o código do *Sendmail* observa-se que os autores desta aplicação construíram a biblioteca *libsm* com funções de *input* similares às contidas na biblioteca *stdio* da linguagem de programação C, com o intuito de introduzir o argumento *timeout* nas referidas funções, para estipular o máximo de tempo permitido para a conclusão da leitura de dados do exterior. Por conseguinte, a aplicação utiliza ambas as bibliotecas de *input - stdio* e *libsm -*, recorrendo à segunda somente quando necessita funcionar com limite de tempo [20].

A descoberta de vulnerabilidades no *Sendmail* utilizando a ferramenta *DEEEP* e perante a biblioteca *libsm*, coloca uma dificuldade à detecção das mesmas, pelo facto da ferramenta estar assente sobre as funções de *input* da *libc*, não marcando, assim, como *tainted* as variáveis de *input* retornadas pelas funções de *input* de *libsm*. Desta forma, para a detecção de possíveis vulnerabilidades provenientes de *inputs* de funções de *libsm* é necessário analisar manualmente os resultados da análise do fluxo de dados do atributo *inputness1* e intersectá-los com os avisos de *bugs* de 64 bits, ou anotar as declarações das funções da *libsm*, à luz do que foi efectuado para as funções da *libc*, e configurar a ferramenta DEEEP por forma a aceitar tais funções, para além das por defeito.

A ferramenta *DEEEP* efectuou toda análise estática, bem como a filtragem dos resultados, a 127 ficheiros de código C, com aproximadamente 112.700 linhas de código, num tempo de 2 minutos e 10 segundos, numa máquina com um processador *Intel* a 2.4GHz, sobre o sistema operativo *Open Solaris*.

O resultado de *DEEEP* apresentou duas possíveis vulnerabilidades em procedimentos criados pelo programador, mas, no entanto, as mesmas não se verificam. Contudo, após a observação cuidada dos resultados da segunda e terceira fases de processamento e da sua intersecção, foi detectada uma vulnerabilidade de inteiro, originada por má portabilidade de código, que pode ser usada para causar uma negação de serviço ou um *buffer overflow*.

Na figura 13(a) está patente o código vulnerável pertencente à função *sm_io_fgets* do ficheiro *fget.c*. Esta função è semelhante à função *fgets* da *libc*, que tem por finalidade a leitura de uma quantidade de *bytes* de um ficheiro, do tamanho do *buffer* que os irá armazenar, numa espaço de tempo estipulado. Os seus argumentos, constantes na declaração *sm_io_fgets(fp, timeout, buf, n)* são: *fp*, o ficheiro que contem os dados a

serem lidos; *timeout*, tempo, em milissegundos, permitido para completar a leitura da *string*; *buf*, *buffer* que armazenará os dados lidos; e *n*, tamanho de *buf* (*sizeof(buf)*).

Na identificação da má adaptação constante na figura 13(b) é visível a ocorrência da vulnerabilidade de *signedness* nas linhas de código 92 e 104, pela conversão de um *signed int* num *unsigned int* (*size_t*), que poderá converter um número negativo num inteiro positivo grande. Esta vulnerabilidade pode ser atacada pondo no ficheiro que é lido um valor negativo para que seja colocado na variável *len*. A figura 13(c) apresenta o resultado da análise de fluxo de dados onde se pode observar nos dois casos apresentados que a variável *len* é passada a argumentos que requerem estado *untainted*. Se o valor de *len* for negativo, da vulnerabilidade de inteiro *signedness* poderá resultar um grande número positivo, onde, na linha 92, poderá ocorrer *DoS* por a quantidade de memória a procurar poder ser superior à memória da máquina. Por seu turno, na linha 104 poderá ocorrer *buffer oveflow* ou *DoS*, por tentar escrever uma quantidade de memória muito superior à comportada pela variável *s* ou ler uma quantidade de memória superior à existente na máquina.

```
     sm_io_fgets(fp, timeout, buf, n)
          register SM_FILE_T *fp;
          int timeout;
          char *buf;
          register int n;
     {
     register int len;

64   if ((len = fp->f_r) <= 0){ (...)
92   t = (unsigned char *) memchr((void *) p, '\n', len);
104  (void) memcpy((void *) s, (void *) p, len);
```

(a)

```
92   Function memchr expects arg 3 to be size_t gets int: len
104  Function memcpy expects arg 3 to be size_t gets int: len
```

(b)

```
92: Invalid transfer from implicitly nostate len to untainted
    (Possible tainted input variable or integer passed as
     untainted):
          memchr(..., len, ...)
     64: len becomes implicitly nostate

104: Invalid transfer from implicitly nostate len to untainted
     (Possible tainted input variable or integer passed as
      untainted):
          memcpy(..., len, ...)
      64: len becomes implicitly nostate
```

(c)

Figura 13: (a) Código fonte do *Sendmail* com vulnerabilidade. (b) Mensagens de 64 *bits*. (c) *Analisador de Fluxo de Dados.*

6 Conclusão

Nos dias de hoje, a competitividade e a pressão exercida sobre os fabricantes de *software* leva a que a quantidade de *software* fabricado seja crescente, abundando as interfaces apelativas e todo o tipo de funcionalidades. Contudo, a qualidade e a segurança desse mesmo *software* continua a deixar a desejar. Uma forma visível de competição nesse mercado é na evolução para a arquitectura de 64 bits, onde os fabricantes de *software*, para não perderem tempo em escrever aplicações de raiz para 64 bits, aproveitam o código das aplicações de 32 bits e o portam para 64 bits.

A detecção de vulnerabilidades, durante a construção das aplicações por análise estática de código, apesar de limitada às regras e padrões para que as ferramentas são programadas e dos falsos negativos gerados é uma mais valia no desenvolvimento de *software* seguro.

A solução aqui apresentada, a ferramenta *DEEEP*, tem por objectivos detectar onde é necessário efectuar alterações ao código, para uma correcta adaptação de ILP32 para LP64, bem como onde ocorrerá a exploração de vulnerabilidades de inteiro. A ideia de combinar e estender ferramentas pré-existentes é prática e mostrou ter resultados interessantes.

Referências

[1] N. Neves, J. Antunes, M. Correia, P. Veríssimo, R. Neves. **Using Attack Injection to Discover New Vulnerabilities**. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, volume 00, pp. 457-466. *Philadelphia, USA*. Junho de 2006.

[2] D. Brumley, T. Chiueh, R. Johnson, H. Lin, D. Song. **RICH: Automatically Protecting Against Integer-Based Vulnerabilities**. *In: Proceedings of the Network and Distributed System Security (NDSS)*. Janeiro de 2007.

[3] R. Seacord. **Secure Coding in C and C++**. Addison Wesley Professional. ISBN 0321335724. 2005.

[4] ISO/IEC. **9899 – Programming Languages C**. Maio de 2005. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf

[5] Blexim. **Basic Integer Overflows**. Phrack #60. http://www.phrack.org/archives/60/p60-0x0a.txt

[6] The Open Group. **Go Solo 2 - The Authorized Guide to Version 2 of the Single UNIX Specification**. Source Books from The Open Group. Andrew Josey. ISBN 0135756898. 1997. http://www.unix.org/version2/whatsnew/login_64bit.html

[7] The Open Group. **64-Bit Programming Models: Why LP64?** Aspen Data Model Committee. 1997-1998. http://www.unix.org/version2/whatsnew/lp64_wp.html

[8] Hewlett Packard. **HP-UX 64-bit Porting and Transition Guide HP 9000 Computers**. Junho de 1998. http://docs.hp.com/en/5966-9887/5966-9887.pdf

[9] Sun Microsystems. **Solaris 64-bit Developer's Guide**. Janeiro de 2005. http://docs.sun.com/app/docs/doc/816-5138?a=load

[10] Sun Microsystems. **C User's Guide**. http://docs.sun.com/source/806-3567/C_Users_GuideTOC.html

[11] Splint Manual: http://www.splint.org/manual/ (2003).

[12] D. Evans, D. Larochelle. **Improving Security Extensible Lightweight Static Analysis**. *IEEE Software*, vol. 19, no. 1, pp. 42-51. Janeiro/Fevereiro de 2002.

[13] Sendmail. http://www.sendmail.org/ (2007).

[14] B. Chess, G. McGraw. **Static Analysis for Security**. *IEEE Security and Privacy*, vol. 02, no. 6, pp. 76-79. 2004

[15] C. Michael, S. R. Lavenhar. **Source Code Analysis Tools – Overview**. Janeiro de 2006. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/code/263.html

[16] W. Bush, J. Pincus, D. Sielaff. **A Static Analyzer for Finding Dynamic Programming Errors**. *Software Practice & Experience* 30:775–802, 2000.

[17] U. Shankar, K. Talwar, J. Foster, D. Wagner. **Detecting Format-String Vulnerabilities with Type Qualifiers**. *In Proceedings of the 10th USENIX Security Symp*osium, Agosto de 2001.

[18] H. Chen, D. Dean, D. Wagner, **Model Checking One Million Lines of C Code**. *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*. Fevereiro de 2004.

[19] A. Jaeger. **Porting to 64-bit GNU/Linux Systems**. *In Proceedings of the GCC Developers Summit*, pp. 107-121. Maio de 2003.

[20] Sendmail libsm library. http://postal.uv.es/libsm/ (2001).