UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

DETECÇÃO DE VULNERABILIDADES DE INTEIROS NA ADAPTAÇÃO DE SOFTWARE DE 32 PARA 64 BITS

Ibéria Vitória de Sousa Medeiros

MESTRADO EM INFORMÁTICA

Dezembro de 2007

DETECÇÃO DE VULNERABILIDADES DE INTEIROS NA ADAPTAÇÃO DE SOFTWARE DE 32 PARA 64 BITS

Ibéria Vitória de Sousa Medeiros

Dissertação submetida para obtenção do grau de
MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Miguel Nuno Dias Alves Pupo Correia

Dezembro de 2007

Resumo

Os processadores de 64 bits, comercializados por fabricantes como a *Intel* e a *AMD*, começaram a equipar os mais recentes computadores, sendo muito do *software* neles executado 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. Esta tese 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. Também propõe 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, de código aberto, que são utilizadas 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* correlaciona a informação delas resultante, identificando se os *bugs* encontrados são realmente vulnerabilidades, ou seja, se são atacáveis. São apresentados resultados experimentais da utilização da ferramenta com código vulnerável sintético, criado especificamente para avaliar a ferramenta, e com diversos pacotes de código aberto.

PALAVRAS-CHAVE: Vulnerabilidades de inteiros, portabilidade, análise estática de código, detecção de vulnerabilidades, processadores de 64 bits.

Abstract

64-bit processors, available from manufacturers such as Intel and AMD, started to equip many recent computers, but much of the software running in them has initially been developed for 32-bit architectures. Applications designed in the C programming language for 32-bit architecture when adapted to 64 bits can show vulnerabilities related to integer handling. This thesis studies the vulnerabilities that can arise when porting software from 32 to 64 bits without the necessary care, considering the LP64 data model, widely used in open source software. This thesis also proposes the *DEEEP*, a tool that detects these vulnerabilities through static analysis of source code. The tool is based on two open source static analysis tools. Type checking is used for finding bugs on the way integers are handled, and data-flow analysis is used to see if hazardous functions (eg. *memcpy*, *strcpy*) are accessible from outside the program. After these two forms of analysis, the *DEEEP* tool correlates their outputs, identifying if the found bugs are really vulnerabilities, i. e., if they are attackable. The tool was evaluated using synthetic code and several open source packages, like *Sendmail*.

KEYWORDS: Integer vulnerabilities, portability, static analysis, vulnerability detection, 64-bit processors.

Agradecimentos

Quase finda mais uma etapa académica e profissional, seria egoísmo meu não mostrar gratidão para com as pessoas que me ajudaram a concretizá-la. Não há primeiros, nem segundos lugares, porque, independentemente da sua contribuição, sem elas possivelmente poderia não completar este projecto de vida.

Ao meu orientador, o Professor Miguel Correia, que desde o início sempre se prontificou, com entusiasmo, empenho, interesse e disponibilidade neste projecto, acompanhando-me e ajudando-me de forma motivadora. Não demais a ele, agradeço por ter acreditado que seria possível a construção e concretização deste projecto, nos moldes em que se desenvolveu. A minha sincera gratidão. Fica a esperança de com ele poder iniciar e finalizar outros projectos.

Ao meu amigo, companheiro e marido, Pedro Marques, que sempre me apoiou e ajudou compreensivamente e pacientemente, principalmente nos momentos de maior trabalho, nos quais teve de abdicar do meu companheirismo.

A todos os demais, amigos e colegas de trabalho, que sempre verbalizaram palavras amigas, de apoio e encorojamento.

Não obstante a todos, não posso deixar de agradecer ao Professor Vasco Vasconcelos, por ter aceite a minha candidatura no mestrado, consciente de como, por mim, seria realizado – "à distância" –, por motivos geográficos.

Ponta Delgada, Dezembro de 2007

Ibéria Vitória de Sousa Medeiros

Índice Geral

Índice de Figuras

Índice de Tabelas

Capítulo 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 desempenho, fiabilidade e segurança.

Esta evolução passou, primeiramente, pelo desenvolvimento de arquitecturas de processadores e, posteriormente, pelo de *software*, que acaba por ser o efeito do aproveitamento dos recursos fornecidos pela arquitectura.

Inicialmente, quando a *Intel* desenvolveu a sua arquitectura de microprocessadores de 32 bits [1], pensava ser impossível existirem ficheiros de dimensão superior a 4 GB. Com o decorrer dos anos, surgiu a necessidade de algumas bases de dados excederem os 4 GB. Tais ficheiros eram acedidos através da memória virtual e podiam atingir os 64 GB. Mas, o tempo de acesso aos dispositivos de memória secundária para aceder a esses ficheiros e o seu carregamento para a memória principal tinha um significativo impacto negativo no desempenho dos sistemas equipados com processadores de 32 bits.

O tamanho da palavra do processador indica qual o máximo valor inteiro que pode ser manipulado pelo mesmo. Este factor põe em causa o desempenho de um computador, se o tamanho da palavra for inferior ao necessário para representar os inteiros que se pretende manipular. Por exemplo, um processador com comprimento de palavra de 16 bits pode manipular um valor inteiro máximo de 65.535 (2^{16} - 1). Se o valor do número inteiro a manipular for 100.000, por exemplo, o processador terá de efectuar a operação em dois ciclos de relógio. No caso dos processadores de 32 bits, o processador pode manipular, numa só operação, números inteiros até 4.294.967.295 (2^{32} - 1) [2]. O problema do desempenho dos processadores de 32 bits coloca-se quando aplicações necessitam que estes efectuem operações com números inteiros de valor superior ao limite desses processadores e/ou necessitam de aceder a mais de 4 GB de memória principal.

Para ultrapassar esses problemas, foi desenvolvida a arquitectura de 64 bits, que possui um barramento de endereços de 64 bits, perfazendo um espaço de endereçamento linear na ordem dos 16 TB (2^{64}) e permitindo carregar na memória principal ficheiros desta dimensão. Também permite manipular números inteiros de até $1{,}84467441 \times 10^{19}$ (2^{64} - 1), fazendo com que o processador as execute num só ciclo [1][3].

Contudo, uma pergunta se coloca: será que as aplicações de 64 bits irão mesmo usufruir dos recursos da arquitectura do processador? Ou melhor, será que qualquer aplicação que seja construída sob 64 bits ou recompilada de 32 para 64 bits necessitará de aceder a mais de 4 GB de memória física e/ou efectuar cálculos matemáticos que envolvam operandos inteiros de 64 bits?

A resposta é claramente negativa. Somente algumas aplicações de 32 bits necessitam de ser convertidas para 64 bits. As aplicações de 32 bits que não necessitam dos recursos oferecidos pela arquitectura de 64 bits podem manter-se como de 32 bits, sendo executadas em modo de compatibilidade na arquitectura de 64 bits. Uma aplicação deverá ser convertida de 32 para 64 bits, quando necessitar de utilizar os recursos apresentados pela arquitectura, ou seja, necessitar de: aceder a mais do que 4 GB de memória linear; trabalhar com ficheiros de grandes dimensões; fazer cálculos matemáticos complexos que envolvam operações aritméticas com operandos inteiros de 64 bits, que necessitem de ser efectuadas por um só ciclo do processador; e aumentar o desempenho da aplicação, utilizando o conjunto de instruções do processador de 64 bits. As aplicações que necessitam de tais características são, por exemplo: bases de dados de grandes dimensões, que necessitam de residir em memória física para poderem manusear os dados e dar respostas mais rapidamente; programas de simulação e modelação que têm de residir em memória física, para o seu bom funcionamento; aplicações que efectuam computação científica que necessitam de residir em memória física e dos registos de 64 bits do processador; e *web caches* que, estando em memória física, reduzem a sua latência [10][11].

A evolução do *software* atrás referida foi, igualmente, acompanhada por um aumento do número de atacantes que procuram constantemente vulnerabilidades em programas para que possam atacá-los. A complexidade das aplicações leva a que sejam mais susceptíveis de conterem erros, podendo muitas delas esconder vulnerabilidades que, quando exploradas por um ataque, podem comprometer as suas propriedades de segurança. As vulnerabilidades não são mais do que um produto de erros que levam a aplicação a estados não previstos pelos seus programadores. A aplicação pode, então, transitar entre estados válidos e possivelmente inválidos. Perante tal, podemos dizer que as garantias de segurança dos sistemas informáticos estão dependentes da não existência de vulnerabilidades. 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 [12].

Um tipo de vulnerabilidades bem conhecido é o das vulnerabilidades de inteiros, de que são exemplo os *overflows* e *underflows* [13][14]. Estas vulnerabilidades estão muito relacionadas com as de *buffer overflow*, pois muitas das 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*.

Para melhorar a segurança do *software*, aquando do seu desenvolvimento e antes de ser colocado no mercado, são realizados testes de segurança, baseados nas técnicas de injecção de ataques e análise estática de código, que têm o propósito de detectar vulnerabilidades para que os programadores e analistas as possam colmatar.

1.1 Contribuição

Recentemente, começou a surgir alguma preocupação com a adaptação de código C de 32 para 64 bits devido à possibilidade de serem introduzidas vulnerabilidades de inteiros. O problema é que no modelo de dados ILP32, definido pela ANSI para processadores de 32 bits [15], os tipos de dados *int*, *long int* e ponteiro ocupam 32 bits, o que permite aos programadores realizarem conversões arbitrárias de variáveis entre esses tipos – especialmente entre *int* e *long* – sem perda de dados [21]. Apesar desta prática ser indesejável, acontece frequentemente. Quando um programa é portado para 64 bits para o modelo LP64, adoptado pelas variantes do Unix e, por isso, utilizado 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.

Esta tese é o primeiro estudo sistemático do problema da detecção deste tipo de vulnerabilidades de inteiros causadas por uma má adaptação do código de 32 para 64 bits 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 de controlo, verificação de tipos, análise de fluxo de dados, verificação de modelos, ou análise de léxico [16][17][18][19][20].

A tese 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* [22] e o *Splint* (*Secure Programming LINT*) [23]. 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 automática dos resultados das duas primeiras fases, apurando 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.

Para avaliar a ferramenta *DEEEP* foi criado código com as vulnerabilidades que se pretendem descobrir. A tese reporta a utilização da ferramenta com esse código, demonstrando a sua capacidade para fazer essa detecção. A ferramenta foi, analogamente, utilizada para fazer a verificação de vários pacotes de *software* de código aberto, num total de mais de um milhão de linhas de código, entre os quais o *Sendmail* [25].

A tese 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 produto final desta tese, ou seja, a ferramenta *DEEEP* está alojada em *http://deeep.homeunix.org* e uma síntese do trabalho da tese apareceu no artigo "Detecção de Vulnerabilidades de Inteiros na Adaptação de Software de 32 para 64 bits", apresentado na 3ª Conferência Nacional sobre Segurança Informática nas Organizações [24].

1.2 Estrutura da Tese

A tese encontra-se organizada da seguinte forma: o capítulo 2 apresenta as áreas de investigação que influenciaram e contribuiram para o trabalho aqui apresentado, nomeadamente números inteiros, modelos de dados, portabilidade de código para LP64, análise estática de código e detecção de vulnetabilidades de inteiros; o capítulo 3 descreve os tipos de vulnerabilidades de inteiros e as categorias de ataques que delas podem decorrer; o capítulo 4 descreve a arquitectura e o funcionamento da ferramenta *DEEEP*, bem como as opções tomadas para o seu desenvolvimento. O trabalho experimental e seus resultados que demonstram a utilidade da ferramenta para a detecção de vulnerabilidades são apresentados no capítulo 5. A tese termina com algumas conclusões e ideias para trabalho futuro.

Capítulo 2

Trabalho Relacionado

A tese descreve uma metodologia para detecção de vulnerabilidades de inteiros na adaptação de código de 32 para 64 bits e uma ferramenta que a implementa. O trabalho desenvolvido é assente e influenciado por diversas áreas de investigação, nomeadamente números inteiros, modelos de dados, portabilidade de código de 32 para 64 bits, análise estática de código e detecção de vulnetabilidades de inteiros.

2.1 Números Inteiros

Esta secção apresenta, de uma forma geral, a representação e tipos de inteiros, bem como a conversão, implícita ou explícita, entre tipos de dados inteiros.

2.1.1 Tipos de Dados Inteiros

Na caracterização das arquitecturas dos microprocessadores, o comprimento da palavra do processador (medido em *bits*), bem como o valor numérico máximo que o processador pode processar numa mesma instrução *versus* ciclo, reporta-se sempre a números do tipo de dados inteiro.

Em termos computacionais, os tipos de dados inteiros podem ser *unsigned* (sem sinal), que representam somente números inteiros positivos, e *signed* (com sinal), que, para além de números inteiros positivos, também representam negativos.

A representação de um número inteiro é efectuada em bits. Desta forma, se para determinado tipo de dados inteiro for definida uma largura de oito bits, significa que este tipo de dados comporta 2^8 números inteiros (256). Por exemplo, um tipo de dados *unsigned* de 8 bits, representa os números inteiros positivos de 0 a 2^8-1, ou seja, de 0 a 255.

Os microprocessadores suportam diferentes tipos de dados inteiros, tanto com como sem sinal (*signed* e *unsigned*), cada qual com a sua largura de bits (Tabela 2.1), estipulando, desta forma, o subconjunto de números inteiros possíveis para cada tipo de dados inteiro [14].

Bits	Nome	Inteiros	Intervalo
8	*byte, octet*	256	*Signed:* -128 a + 127 *Unsigned:* 0 a +255
16	*halfword, word*	65.536	*Signed:* -32.768 a +32.767 *Unsigned:* 0 a +65.535
32	*word, doubleword, longword*	4.294.967.296	*Signed:* -2.147.483.648 a + 2.147.483.647 *Unsigned:* 0 a +4.294.967.295
64	*doubleword, longword, quadword*	18×10^{18}	*Signed:* −9.223.372.036.854.775.808 a +9,223,372,036,854,775,807 *Unsigned:* 0 a +18.446.744.073.709.551.615
n	*n-bit integer*	2^n	*Signed:* -2^{n-1} a $2^{n-1}-1$ *Unsigned:* 0 a 2^n-1

Tabela 2.1: Tipos de dados inteiros

Nas linguagens de programação, cada tipo de dados inteiro pode ter um nome diferente, fazendo distinção entre *signed* e *unsigned* como se pode observar na Tabela 2.2 para a linguagem de programação C [14].

Bits	*Signed*	Nomes utilizados
8	Sim	char
16	Sim	short, int
32	Sim	int, long
64	Sim	long long
8	Não	unsigned char
16	Não	unsigned short, unsigned int
32	Não	unsigned int, unsigned long, pointer
64	Não	unsigned long long

Tabela 2.2: Tipos de dados inteiros na linguagem de programação C

As variáveis de um programa têm o propósito de armazenar valores inseridos pelo utilizador ou resultantes de operações. As mesmas encontram-se armazenadas na memória principal do computador. Por vezes, desejamos armazenar numa variável o local da memória onde determinada variável se situa na memória principal, isto é, o seu endereço de memória. Para tal, utilizamos uma variável do tipo de dados ponteiro, onde o seu valor será o endereço de localização da memória. É comum dizermos que o "ponteiro *p* aponta para a variável *x*" ou que "o ponteiro *p* tem a referência de *x*", por ele referenciar uma variável. O tipo de dados *pointer* (ponteiro) é frequentemente representado por um número inteiro sem sinal de determinada largura, que geralmente é igual à do número inteiro máximo passível de ser manipulado pelo processador. Por exemplo, se o processador for de 32 bits, então a largura do maior tipo de dados inteiro

também o é, significando que o tipo de dados ponteiro é representado por um número inteiro sem sinal de 32 bits.

2.1.2 Conversão de Inteiros em ILP32

A conversão entre tipos de dados inteiros na linguagem de programação C (segundo a norma C99 para 32 bits [28]) pode ser explícita, quando indicada pelo programador (pela palavra reservada *cast*), ou implícita, quando efectuada pela própria linguagem de programação, na presença de operações que envolvem tipos de dados inteiros diferentes.

Por vezes, as conversões entre tipos de dados inteiros, que são efectuadas para um correcto funcionamento do programa, podem originar perda de dados ou valores incorrectos. Esta secção tem o intuito de apresentar o como e o quando estas conversões são efectuadas e identificar as que resultam em falha.

2.1.2.1 Promoção de Inteiros

Por convenção, padronizou-se que as operações de tipos de dados inteiros são efectuadas segundo o tipo de dados *int*, originando, assim, conversão implícita entre tipos de dados inteiros. Neste sentido, uma operação que envolva valores de tipos de dados inteiros de tamanho menor do que o tipo de dados *int* e onde todos eles podem ser representados por um *int*, então estes valores são promovidos para *int* e a operação efectuada. Caso não seja possível, os valores em questão são convertidos para *unsigned int*.
A promoção de tipos de dados é observada em operandos envolvidos em operações de adição (+), subtracção (-), complemento (~) e deslocamento de bits (*shift*).

O motivo principal de ocorrência de conversão implícita de tipos de dados inteiros é para evitar erros aritméticos, resultantes de *overflow* de resultados/valores intermédios. No exemplo de código abaixo (Figura 2.1) [14], na linha 5, o valor de *c1* é adicionado ao valor de *c2* e a soma destes adicionada ao valor de *c3*. A adição de *c1* com *c2* (*c1 + c2 = 190*) excederia (*overflow*) o valor máximo permitido do tipo de dados *signed char* (127), mas tal não acontece devido à promoção de tipos de dados de todas as variáveis do tipo *char* para *int*. Assim, a expressão matemática é efectuada sobre o tipo de dados *int* e o seu

valor resultante truncado e armazenado na variável *result*. No entanto, como o resultado da expressão matemática (70) pertence ao intervalo do tipo de dados *signed char*, o truncamento de dados ocorrido não resulta em perda de dados.

```
1  char result, c1, c2, c3;
2  c1 = 100;
3  c2 = 90;
4  c3 = -120;
5  result = c1 + c2 + c3;
```

Figura 2.1: Prevenção de erros aritméticos por conversão implícita

2.1.2.2 Conversão de Inteiros sem Sinal (*unsigned*)

As conversões entre números inteiros *signed* e *unsigned* de tipos de dados inteiros de tamanhos diferentes podem resultar em perda de dados e valores incorrectos, quando um valor inteiro não pode ser representado no tipo de dados inteiro pretendido.
Assim, a conversão de um número inteiro *unsigned* *x* [14]:

- de um tipo de dados de menor comprimento para um tipo de dados inteiro *unsigned* de maior comprimento é segura e os bits mais à esquerda excedentes tomam o valor de zero;
- de um tipo de dados de maior comprimento para um tipo de dados inteiro *unsigned* de menor comprimento, o valor de *x* é truncado e são preservados os bits menos significativos (os mais à direita de *x*), resultando em perda de dados e alteração de valor, se o valor de *x* sofreu alteração;
- de um tipo de dados de menor comprimento para um tipo de dados inteiro *signed* de maior comprimento é segura e os bits mais à esquerda excedentes tomam o valor de zero;
- de um tipo de dados de maior comprimento para um tipo de dados inteiro *signed* de menor comprimento, o valor de *x* é truncado e são preservados os *bits* menos significativos (os mais à direita de *x*), sendo o novo *bit* mais significativo o representativo do sinal. Poderá, também, resultar em perda de dados se o valor de *x* sofreu alteração;
- para o seu correspondente número inteiro *signed*, a máscara de *bits* (*bit pattern*) é preservada e o *bit* mais significativo torna-se o representativo de sinal, onde se este é 1 o valor de *x* sofre alterações, havendo assim perda de dados.

Na tabela seguinte (Tabela 2.3) [14], é possível visualizar as conversões de um tipo de dados inteiro *unsigned* para outro tipo de dados inteiro, onde as conversões que podem resultar em perda de dados por truncamento estão identificadas por um losango preto, enquanto que as conversões que podem originar valores incorrectos estão identificadas por um losango branco.

De		Para	Conversão
◇	unsigned char	Char	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
	unsigned char	Short	Conversão segura. Preenchimento com zeros à esquerda
	unsigned char	Int	Conversão segura. Preenchimento com zeros à esquerda
	unsigned char	Long	Conversão segura. Preenchimento com zeros à esquerda
	unsigned char	unsigned short	Conversão segura. Preenchimento com zeros à esquerda
	unsigned char	unsigned int	Conversão segura. Preenchimento com zeros à esquerda
	unsigned char	unsigned long	Conversão segura. Preenchimento com zeros à esquerda
◆	unsigned short	Char	Preserva os bits mais à direita
◇	unsigned short	Short	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
	unsigned short	Int	Conversão segura. Preenchimento com zeros à esquerda
	unsigned short	Long	Conversão segura. Preenchimento com zeros à esquerda
◆	unsigned short	unsigned char	Preserva os bits mais à direita
	unsigned short	unsigned int	Conversão segura. Preenchimento com zeros à esquerda
	unsigned short	unsigned long	Conversão segura. Preenchimento com zeros à esquerda
◆	unsigned int	Char	Preserva os bits mais à direita
◆	unsigned int	Short	Preserva os bits mais à direita
◇	unsigned int	Int	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
◇	unsigned int	Long	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
◆	unsigned int	unsigned char	Preserva os bits mais à direita
◆	unsigned int	unsigned short	Preserva os bits mais à direita
	unsigned int	unsigned long	Conversão segura. Preenchimento com zeros à esquerda
◆	unsigned long	Char	Preserva os bits mais à direita
◆	unsigned long	Short	Preserva os bits mais à direita
◇	unsigned long	Int	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
◇	unsigned long	Long	Preserva a *bit pattern*. Sinal dado pelo bit mais significativo
◆	unsigned long	unsigned char	Preserva os bits mais à direita
◆	unsigned long	unsigned short	Preserva os bits mais à direita
	unsigned long	unsigned int	Conversão segura. Preenchimento com zeros à esquerda

◆ Perda de dados, por truncamento ◇ Valor incorrecto

Tabela 2.3: Conversões de tipos de dados inteiros *unsigned* de 32 bits

2.1.2.3. Conversão de Inteiros com Sinal (*signed*)

Também as conversões de números inteiros *signed* para tipos de dados inteiros *unsigned* ou *signed*, de tamanhos iguais ou diferentes, podem resultar em perda de dados e valores incorrectos, quando um valor inteiro não pode ser representado no tipo de dados inteiro

pretendido, agravando-se o problema quando convertemos um número inteiro negativo para um número inteiro sem sinal.

Assim, a conversão de um número inteiro *signed x* [14]:

- de um tipo de dados de menor comprimento para um tipo de dados inteiro *signed* de maior comprimento é segura e os bits mais à esquerda excedentes tomam o valor do bit do sinal de *x* (*sign-extend*);
- de um tipo de dados de maior comprimento para um tipo de dados inteiro *signed* de menor comprimento, o valor de *x* é truncado, preservando os seus bits mais à direita (resultando em perda de dados), e o bit mais significativo, do novo valor, representa o sinal do novo número inteiro;
- de um tipo de dados de menor comprimento para um tipo de dados inteiro *unsigned* de maior comprimento, os bits mais à esquerda excedentes tomam o valor do bit do sinal de *x* (*sign-extend*). Se o valor de *x* for positivo, não haverá qualquer alteração de valor, caso contrário originará um grande inteiro positivo;
- de um tipo de dados de maior comprimento para um tipo de dados inteiro *unsigned* de menor comprimento, o valor de *x* é truncado e são preservados os bits menos significativos. Poderá, de igual modo, resultar em perda de dados, se o valor de *x* sofreu alteração;
- para o seu correspondente número inteiro *unsigned*, a máscara de bits (*bit pattern*) é preservada, não havendo perda de dados, e o bit mais significativo perde a sua função de representação de sinal, podendo originar um grande inteiro positivo, caso o valor de *x* seja negativo.

De		Para	Conversão
	char	short	Conversão segura. Preenchimento dos bits à esquerda com o bit de sinal
	char	int	Conversão segura. Preenchimento dos bits à esquerda com o bit de sinal
	char	long	Conversão segura. Preenchimento dos bits à esquerda com o bit de sinal
◇	char	unsigned char	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal
◆	char	unsigned short	*Sign-extend* para *short*. Conversão de *short* para *unsigned short*
◆	char	unsigned int	*Sign-extend* para *int*. Conversão de *int* para *unsigned int*
◆	char	unsigned long	*Sign-extend* para *long*. Conversão de *long* para *unsigned long*
◆	short	char	Preserva os bits mais à direita
	short	int	Conversão segura. Preenchimento dos bits à esquerda com o bit de sinal
	short	long	Conversão segura. Preenchimento dos bits à esquerda com o bit de sinal
◆	short	unsigned char	Preserva os bits mais à direita
◇	short	unsigned short	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal
◆	short	unsigned int	*Sign-extend* para *int*. Conversão de *int* para *unsigned int*
◆	short	unsigned long	*Sign-extend* para *long*. Conversão de *long* para *unsigned long*

◆ Perda de dados, por truncamento ◇ Valor incorrecto

Tabela 2.4: Conversões de tipos de dados inteiros *signed* de 32 bits

	De	Para	Conversão
◆	int	Char	Preserva os bits mais à direita
◆	int	short	Preserva os bits mais à direita
	int	long	Conversão segura
◆	int	unsigned char	Preserva os bits mais à direita
◆	int	unsigned short	Preserva os bits mais à direita
◇	int	unsigned int	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal
◇	int	unsigned long	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal
◆	long	Char	Preserva os bits mais à direita
◆	long	short	Preserva os bits mais à direita
	long	int	Conversão segura
◆	long	unsigned char	Preserva os bits mais à direita
◆	long	unsigned short	Preserva os bits mais à direita
◇	long	unsigned int	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal
◇	long	unsigned long	Preserva a *bit pattern*. O bit mais significativo perde a função de sinal

◆ Perda de dados, por truncamento ◇ Valor incorrecto

Tabela 2.4: Conversões de tipos de dados inteiros *signed* de 32 bits (continuação)

A tabela anterior (Tabela 2.4) [14] é representativa das conversões de um tipo de dados inteiro *unsigned* para outro tipo de dados inteiro, onde as conversões que podem resultar em perda de dados por truncamento estão identificadas por um losango preto, enquanto que as conversões que podem originar valores incorrectos estão identificadas por um losango branco.

Por fim, há que salientar que o tipo de dados *char* pode ser, dependendo da linguagem de programação e plataforma, somente do tipo *unsigned* ou dos dois tipos. Se estivermos na presença dos dois tipos (*signed* e *unsigned*), aumentará em número os "problemas" advindos das conversões entre tipos de dados inteiros [14].

Quando armazenamos um *signed char*, com o valor do seu bit mais significativo igual a 1, num tipo de dados inteiro, o resultado é um número inteiro negativo. Assim, para não originar este tipo de problema, onde o caracter (o *char*) perde o seu significado, devemos utilizar *unsigned char*, ao invés do *char* ou *signed char*. Esta medida é conveniente quando utilizamos *buffers* e ponteiros. Por outro lado, devemos recorrer à conversão explícita (*cast*) quando o valor do *unsigned char* toma valores superiores a 127.

2.2 Modelos de Dados

No desenvolvimento dos microprocessadores, um dos objectivos sempre presente foi o aumento da velocidade de processamento e consequente melhoramento de desempenho, conseguidos, entre muitos outros factores, pelo aumento do espaço de endereçamento linear e do número de bits que o processador pode processar em simultâneo num só ciclo, isto é, o comprimento da palavra do processador. Esta última representa, também, o máximo número inteiro que pode ser manipulado num ciclo do processador. Contudo, para que possamos atingir os níveis de processamento e desempenho projectados pelo processador, temos de ter em consideração as aplicações a serem instaladas na máquina equipada com o processador, nomeadamente o sistema operativo.

O funcionamento do computador e o seu desempenho está directamente ligado à relação entre o sistema operativo e o *hardware*. O sistema operativo é desenvolvido de forma a aproveitar o máximo dos recursos da arquitectura do processador. Por exemplo, antes do aparecimento dos processadores de arquitecturas de 64 bits, os sistemas operativos, tais como o *Microsoft Windows XP*, *Linux* e *Solaris*, foram concebidos para trabalhar com processadores de arquitectura de 32 bits e não com a de 64 bits. O mesmo se passou com os sistemas operativos concebidos para processadores de arquitectura de 16 bits, que não estavam preparados para processadores de 32 bits.

Assim sendo, se instalarmos num computador equipado com um processador de 64 bits um sistema operativo de 32 bits, este irá comportar-se como se estivesse instalado numa máquina equipada com um processador de 32 bits, e todas as aplicações nele instaladas terão, também, de ser de 32 bits. O *software* instalado neste computador não irá beneficiar das instruções projectadas para a arquitectura do processador de 64 bits, ficando limitado às características de um processador de 32 bits.

Devido à evolução dos microprocessadores, foi necessário que o desenvolvimento de *software* a acompanhasse, para que pudesse funcionar e explorar os recursos que a arquitectura dos processadores oferece, aumentando, desta forma, significativamente o desempenho dos computadores. No entanto, os fabricantes de *software* têm de construir as suas aplicações à luz de um modelo de dados.

O modelo de dados define os tamanhos dos tipos de dados na arquitectura do processador. Isto leva a que as linguagens de programação o tenham de respeitar, no desenvolvimento de aplicações da arquitectura.

Na linguagem de programação C existe somente um modelo de dados para a arquitectura de 16 bits, o LP32, e um só modelo de dados para a arquitectura de 32 bits, o ILP32. No entanto, para o desenvolvimento de aplicações de 64 bits surgiram três modelos de dados, ILP64, LLP64 e LP64 [10][11][15][21][30][32]. Desta forma, existem compiladores para cada um dos modelos de dados da arquitectura de 64 bits.

De seguida, serão apresentadas as características de cada um dos modelos de dados supracitados, dando especial relevância ao número de bits que cada tipo de dados representa e manipula.

2.2.1 16 *bits*

As aplicações de 16 bits foram as desenvolvidas para os processadores de arquitectura de 16 bits (IA-16, por exemplo), nomeadamente os Intel 8086, 8088 e 80286. O *software* mais conhecido dessa época foram os sistemas operativos *MS-DOS*, *OS/2* e as primeiras versões do *Microsoft Windows* [1].

Qualquer aplicação era construída de acordo com as características da arquitectura dos microprocessadores, como por exemplo para os 20 e 24 bits do barramento de endereço dos microprocessadores 8088 e 80286, respectivamente. Do mesmo modo, quase todas as operações de inteiros manipulavam um máximo de 16 bits, uma vez que o comprimento da palavra do microprocessador era de 16 bits (dois octectos ou uma *word*).

Na altura das aplicações de 16 bits, a linguagem de programação C suportava tipos de dados inteiro (*integer*) e ponteiro (*pointer*) de 16 bits e, também, o tipo de dados inteiro de 32 bits, o qual podia ser emulado em *hardware* que não suportava operações aritméticas com operadores de 32 bits [15][32].

Tipo de Dado	Bits
char	8
short	16
int	16
long	32
pointer	32

Tabela 2.5: Modelo de dados LP32

O tipo de dados inteiro de 32 bits passou-se a denominar de *long* (inteiro longo), o qual podia armazenar um ponteiro de 32 bits. Por esse motivo, temos o modelo de dados

LP32, que significa que os tipos de dados *long* e *pointer* são de 32 bits, enquanto que o tipo de dados *int* (inteiro) é de 16 bits, como se pode observar na Tabela 2.5

2.2.2 32 *bits*

Quando surgiram os microprocessadores de arquitectura de 32 bits (como a IA-32), que suportavam operações aritméticas com operadores inteiros de 32 bits, bem como o tipo de dados ponteiro de igual comprimento, começaram-se a desenvolver aplicações que usufruíssem de tais recursos [1].

O modelo de dados da linguagem de programação C denominou-se ILP32 (Tabela 2.6) por suportar os tipos de dados *int*, *long* e *pointer* de 32 bits. No entanto, continuou a suportar o tipo de dados inteiro de 16 bits, mas denominado por *short* (inteiro pequeno), de forma a permitir a compatibilidade entre aplicações de 16 com 32 bits [10][11][15][21][32].

Tipo de Dado	Bits
char	8
short	16
int	32
long	32
pointer	32

Tabela 2.6: Modelo de dados ILP32

A transição de *software* de 16 bis para 32 bits em sistemas de arquitecturas compatíveis tornou-se possível com a construção do processador 80386 e o DOS/4GW, em que o referido processador e os seus sucessores suportam um espaço de endereçamento de 16 e 32 bits, por razões de compatibilidade de aplicações.

2.2.3 64 *bits*

Na arquitectura de processadores de 64 bits, surgiram três novos modelos de dados e os respectivos compiladores de 64 bits, para o desenvolvimento de aplicações. Assiste-se ao desenvolvimento de aplicações que usufruem dos recursos oferecidos pela arquitectura, bem como ao reescrever e recompilar aplicações de 32 bits para 64 bits.

A norma ISO/IEC 9899:1990 (ANSI C99) [28], que padroniza a nomenclatura e estrutura da linguagem de programação C, deixou a definição dos tipos de dados *short int*, *int*, *long int*, e *pointer* deliberadamente vaga para evitar constrangimentos artificiais nas arquitecturas de *hardware*, que podem beneficiar destas definições de tipos de dados, independentes uns dos outros. O único constrangimento imposto é que o tamanho do tipo de dados *int* tem de ser maior ou igual ao do *short*, o tamanho do tipo de dados *long* tem de ser maior ou igual ao do *int*, e o tamanho do tipo de dados *size_t* representa o maior *unsigned* tipo de dados inteiro suportado pela implementação. Assim sendo, é possível, por exemplo, definir o *short* como 16 bits, o *int* como 32 bits, o *long* como 64 bits e o *pointer* como 128 bits.

A relação entre os vários tipos de dados fundamentais pode ser expressa como:

```
sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)=sizeof(size_t)
```

Dada esta relação, foram definidos três modelos de dados de 64 bits que a satisfazem - ILP64, LLP64 e LP64, tendo todos eles o tipo de dados *pointer* de comprimento de 64 bits, devido à largura do maior tipo de dados inteiro ser 64 bits.

2.2.3.1 ILP64

O modelo de dados ILP64, apresentado na Tabela 2.7, é, também, conhecido por 8/8/8, por os três tipos de dados *int*, *long* e *pointer* terem o mesmo tamanho (64 bits cada). Este modelo tenta manter a relação entre os tipos de dados *int*, *long* e *pointer* do modelo de dados ILP32 [15][30].

Tipo de Dado	Bits
char	8
short	16
int32	32
int	64
long	64
pointer	64

Tabela 2.7: Modelo de dados ILP64

A atribuição de um ponteiro a um inteiro ou a um inteiro longo não irá resultar em perda de dados. Contudo, definir o tamanho de todos os tipos de dados (*int*, *long* e

pointer) com 64 bits resultará em desperdício de espaço, uma vez que muitas aplicações não necessitarão desse aumento de tamanho.

A característica menos positiva deste modelo de dados reside no facto de ser necessário adicionar um tipo de dados de 32 bits (*int32*), para representar o tipo inteiro de 32 bits e assegurar a conversão de aplicações de 32 bits para 64 bits. No entanto, a adição deste tipo de dados poderá originar conflito com tipos de dados definidos (*typedefs*) pelo programador e existentes na aplicação por ele desenvolvida.

Uma aplicação que seja desenvolvida no modelo de dados ILP32 e, posteriormente, recompilada para o modelo de dados ILP64 pode ser forçada a utilizar frequentemente o tipo de dados *int32* para poder preservar o tamanho e o alinhamento dos dados, devido aos requisitos da inter-operacionalidade ou da compatibilidade binária com os arquivos de dados existentes.

O modelo ILP64 é utilizado, por exemplo, em computadores *Cray* e nos processadores *Alpha*.

2.2.3.2 LLP64

O modelo de dados LLP64 (Tabela 2.8), também conhecido por 4/4/8, mantém o tamanho dos tipos de dados *int* e *long* igual ao modelo ILP32. Também é denominado somente por P64, por o "LL" referir-se ao tipo de dados *long long (int64)*, que não é um tipo de dados primitivo da linguagem.

Tipo de Dado	Bits
char	8
short	16
int	32
long	32
long long (int64)	64
pointer	64

Tabela 2.8: Modelo de dados LLP64

Por manter a relação com o ILP32, os objectos de dados, como as estruturas, que não contêm ponteiros terão o mesmo tamanho em ILP32, o que por tal o faz ser também designado por um modelo de 32 bits com endereçamento de 64 bits (ponteiro) [15][30].

Os problemas inerentes a este modelo, aquando da conversão de aplicações de 32 para 64 bits, estão associados à atribuição de um ponteiro a um inteiro. No entanto, a

resolução deste tipo de problema passa por alterar as variáveis dos tipos de dados *int* ou *long*, que deverão ter comprimento de 64 bits, para o tipo de dados *long long*.

Tal como foi observado no modelo anterior de dados - ILP64 -, o facto de o tipo de dados *int64* (*long long*) não ser um tipo de dados primitivo da linguagem de programação poderá originar conflitos com os tipos de dados definidos (*typedefs*) na aplicação desenvolvida, segundo este modelo de dados (LLP64).

2.2.3.2.1 O Modelo de Dados da *Microsoft*

O único sistema que é construído sob este modelo de dados é o *Windows*, da *Microsoft*, bem como aplicações para ele desenvolvidas pela *Microsoft*. No sistema operativo *Windows* de 64 bits somente o tipo de dados *pointer* é expandido para 64 bits, enquanto que os tipos de dados *int* e *long* mantêm o tamanho de 32 bits, como no modelo ILP32 [15][29][30][31][32].

Inicialmente, a maioria das aplicações de 64 bits da *Microsoft* provinham das de 32 bits, as quais eram reescritas e recompiladas por forma a poderem ser executadas em ambos os sistemas operativos, *Windows* de 32 e de 64 bits. Para tal, foi necessário assegurar que o modelo de dados somente afectava o tipo de dados *pointer*, uma vez que este passou a um comprimento de 64 bits, e definir novos tipos de dados que permitissem relacionar os tamanhos dos ponteiros. Isto veio permitir que, ao alterar o tamanho do tipo de dados ponteiro de 32 para 64 bits, o tamanho dos dados associados ao ponteiro também se alterasse. No entanto, os tipos de dados primitivos mantêm o seu tamanho em 32 bits, facilitando a recompilação das aplicações de 32 bits, por não haver, por exemplo, quaisquer alterações do tamanho dos dados em suportes de memória secundária (disco rígido), partilha de dados em rede de computadores, ou partilha de dados através de mapeamento de memória de ficheiros [29][30][31].

Se tivermos em conta a relação entre os tipos de dados da linguagem de programação C++ (linguagem de programação proveniente da linguagem C e utilizada pela *Microsoft*) com o tipo de dados ponteiro, a mesma em C++ não é garantida, uma vez que se assume que o tamanho de um ponteiro é igual ao do tipo de dados inteiro de maior largura. Onde este está associado à largura do barramento de endereços do processador (tipo de dados ponteiro) com o tamanho da palavra do processador (tipo de dados inteiro de maior largura). Contudo, tal suposição não é correcta, fazendo com que em

programação tenhamos de forçar uma dada variável a um tipo de dados (*cast*), originando perda de dados. Por exemplo, se considerarmos um processador de espaço de endereçamento de 48 bits e comprimento de palavra de 16 bits, iremos perder informação quando atribuirmos aos 16 bits os 48 bits [29][30][31].

Tipo de Dado C++ [windows]	win16 (bits)	win32 (bits)
(unsigned) short [WORD]	16	16
(unsigned) int [UINT]	16	32
(unsigned) long [DWORD]	32	32
void *	???	32
void near *	16	N/A
void far *	32	N/A

??? – desconhecido N/A – Não Aplicável

Tabela 2.9: Tipos dados e tamanhos em *Windows*

Definição do Tipo de Dado	Tipo de Dado Windows
typedef unsigned long	DWORD
typedef long	LONG
typedef int	BOOL
typedef unsigned char	BYTE
typedef unsigned short	WORD
typedef short	SHORT
typedef float	FLOAT
typedef FLOAT	*PFLOAT
typedef BOOL near	*PBOOL
typedef BOOL far	*LPBOOL
typedef BYTE near	*PBYTE
typedef BYTE far	*LPBYTE
typedef int near	*PINT
typedef int far	*LPINT
typedef WORD near	*PWORD
typedef WORD far	*LPWORD
typedef long far	*LPLONG
typedef DWORD near	*PDWORD
typedef DWORD far	*LPDWORD
typedef void far	*LPVOID
typedef int	INT
typedef unsigned int	UINT
typedef unsigned int	*PUINT

Tabela 2.10: Extracto da biblioteca *windef.h,* da *Microsoft*

Em *Windows* existem diversos tipos de tamanhos de ponteiros, representados por 16 bits (*near pointer*) e por 32 bits (*far pointer*), os quais são visíveis já nas aplicações de 16 bits (Tabela 2.9). Comparando os modelos aplicacionais de 16 e 32 bits em *Windows*, verifica-se que somente houve alteração no tamanho do tipo de dados inteiro (*int*).

A *Microsoft* não utiliza directamente os tipos de dados primitivos da linguagem. Define tipos de dados (*typedefs*) para representar os mesmos, como se pode observar na Tabela 2.10, e outros para representar tipos de dados não existentes na linguagem.

Na biblioteca *windef.h*, representada pela Tabela 2.10, estão definidas as equivalências de tipos de dados, bem como a definição de novos tipos de dados.

Os tipos de dados DWORD e LONG são os mais utilizados no *Windows*, que comparativamente com a linguagem C++ seriam, respectivamente, os tipos de dados *int* e *long*. No entanto, o tipo de dado DWORD é utilizado quando se pretende representar um número inteiro positivo demasiado grande, uma vez que ele é equivalente ao *unsigned long*. Os tipos de dados BYTE e SHORT são utilizados para representar o tipo de dados inteiro de 1 byte (8 bits = *char*) e o inteiro de 2 bytes (16 bits = *short*), respectivamente. Por seu turno, os tipos de dados ponteiro, de prefixo P ou LP, referenciam ponteiros dos tipos de dados inteiros de 16 bits (P = *near pointer*) e de 32 bits (LP = *far pointer* = *long pointer*), respectivamente.

Os tipos de dados das aplicações *win32* (Tabela 2.10) são utilizados na sua íntegra no modelo de dados LLP64. O problema coloca-se, na recompilação de uma aplicação *win32* para *win64*, quando um ponteiro de 32 bits é atribuído a um tipo de dados inteiro ou longo inteiro, ou quando, no desenvolvimento de aplicações *win64*, é necessário utilizar inteiros de 64 bits, em vez de utilizar DWORD ou LONG (ambos de 32 bits). Para ambas as situações, a *Microsoft* definiu novos tipos de dados para manipulação de inteiros, como o *int64* (*long long*), e ponteiros de 64 bits.

Em suma, podemos dizer que o modelo de dados utilizado pela *Microsoft* é único e feito à medida, porque se todas as aplicações desenvolvidas até então, independentemente da plataforma aplicacional (*win16* ou *win32*), assentavam em tipos de dados definidos pela própria *Microsoft*, ignorando os oferecidos pela linguagem de programação, e se as aplicações *win64* são oriundas das *win32*, então é mais fácil construir um modelo de dados de 64 bits que aproveite os tipos de dados existentes em *win32* do que reescrever por completo as aplicações de *win32* e/ou construir as aplicações *win64*, tendo por base um dos outros modelos de dados de 64 bits, os quais assentam nos tipos de dados inteiros primitivos da linguagem de programação que se expandem de 32 para 64 bits.

Para um melhor entendimento da não adopção dos outros modelos de dados (ILP64 e LP64), se recompilarmos, de *win32* para *win64*, a estrutura de dados do

cabeçalho de um ficheiro *bitmap* (de extensão *bmp*) (Figura 2.2) [31] para um desses modelos, o tipo de dado LONG expande-se de 32 para 64 bits, levando a que o programa de 64 bits não consiga identificar os diversos componentes que compõem o cabeçalho do ficheiro, e consequentemente não poderá utilizar a estrutura de dados, pelo facto de alguns dos tipos de dados terem aumentado de tamanho.

```
typedef struct tagBITMAPINFOHEADER {
        DWORD      biSize;
        LONG       biWidth;
        LONG       biHeight;
        WORD       biPlanes;
        WORD       biBitCount;
        DWORD      biCompression;
        DWORD      biSizeImage;
        LONG       biXPelsPerMeter;
        LONG       biYPelsPerMeter;
        DWORD      biClrUsed;
        DWORD      biClrImportant;
} BITMAPINFOHEADER, FAR *LPBITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Figura 2.2: Estrutura de dados do cabeçalho de um ficheiro de *bitmap,* em *Windows*

2.2.3.3 LP64

O modelo de dados LP64, apresentado na Tabela 2.11, também conhecido por 4/8/8, é o mais utilizado pelos programadores de aplicações, em sistemas, tais como, *Mac OS X*, *Linux*, *Sun Solaris*, *SGI Irix* e em muitos outros sistemas *Unix* de 64 bits [10][11] [15][32].

Pelo facto dos sistemas *Unix* utilizarem, em grande parte, variáveis do tipo de dado inteiro (*int*), ao invés do tipo de dados inteiro longo (*long*), e por questões de conversão de aplicações de 32 para 64 bits, neste modelo de programação o tamanho do tipo de dados inteiro mantém-se em 32 bits, enquanto que o tamanho dos tipos de dados inteiro longo e ponteiro (*pointer*) expande-se para 64 bits.

Tipo de Dado	Bits
char	8
short	16
int	32
long	64
pointer	64

Tabela 2.11: Modelo de dados LP64

Neste modelo, a existência do tipo de dados *long* de 64 bits permitirá operações aritméticas com operandos de 64 bits e a utilização conjunta com o tipo de dados *pointer* de 64 bits, por forma a poder-se atribuir endereços de memória (*pointer*) a um *long*, em vez de a um *int*.

Como se pode observar na tabela anterior, este modelo utiliza somente os tipos de dados primitivos da linguagem de programação, não são adicionados novos tipos de dados e mantém a relação entre os seus tamanhos, ou seja, o tamanho de um dado tipo de dados é maior ou igual ao do seu antecessor. O facto de os tipos de dados *int* e *long* terem diferentes tamanhos (32 e 64, respectivamente) não se desvia muito do já sucedido no modelo de dados padrão de 16 bits (LP32), em que estes tinham um tamanho de 16 e 32 bits, respectivamente.

Segundo este modelo de dados, uma aplicação de 32 bits não necessita de grandes alterações ao seu código para poder ser executada numa plataforma LP64. Tais modificações ao código podem ser efectuadas por forma a serem compiladas tanto para plataformas de 32 como para 64bits. Se bem analisadas, as referidas modificações ao código são no sentido de assegurar ponteiros de 64 bits, tendo especial atenção à suposição errónea sobre os tamanhos dos tipos de dados *int* e *pointer*, em que eles são iguais. No entanto, a relação dos tamanhos dos tipos de dados *char*, *short*, *int* e *float* mantém-se por não causarem problemas na compilação de 32 para 64 bits, uma vez que no modelo LP64 esses tipos de dados não sofrem alteração de tamanho, comparativamente com o modelo ILP32.

Numa visão geral e analisando a Tabela 2.12, representativa dos três modelos de dados de 64 bits, conseguimos observar e depreender que o LP64 é o que apresenta menor ocupação de espaço em memória quando uma aplicação é executada. Tal observação reside nos factos de este modelo não criar qualquer novo tipo de dados, nem desperdiçar espaço em memória com variáveis que não necessitam de tamanhos grandes e só necessitar de assegurar o tipo de dados ponteiro de 64 bits [10][11][15][32].

Tipo de Dado	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int32			32		
int	16	32	64	32	32
long	32	32	64	32	64
long long (int64)				64	
pointer	32	32	64	64	64

Tabela 2.12: Comparação dos modelos de dados

2.3 Portabilidade de Código para LP64

Apresentados os modelos de dados de 64 bits, nesta secção iremos debruçar-nos sobre a adaptação de código de 32 bits (ILP32) para 64 bits, nomeadamente LP64, por ser o modelo de dados que trabalha somente com os tipos dados primitivos da linguagem de programação C e o que, segundo o qual, se desenvolve *software* de código aberto. Será dada a visão dos problemas encontrados na adaptação (portabilidade) de código de ILP32 para LP64, no que concerne aos tipos de dados inteiro, inteiro longo e ponteiro. Também se dará uma visão na portabilidade de aplicações de 32 bits (ILP32) para os outros modelos de dados de 64 bits (ILP64 e LLP64).

Quando falámos em portabilidade de código de ILP32 para LP64 temos de considerar dois cenários possíveis: a conversão de aplicações desenvolvidas em ILP32, para que sejam somente executadas em sistemas operativos de 64 bits; ou a alteração e recompilação de aplicações ILP32 para LP64, para que sejam executadas em sistemas operativos de 32 e de 64 bits.

Em qualquer um dos cenários é necessário efectuar alterações ao código que respeitem as regras do modelo de dados da arquitectura a que se destina a aplicação. Por forma a que as aplicações resultantes da transição do ILP32 para o LP64 não ocorram em erros e/ou em *crash* de sistema, é necessário fazer alterações ao código a portar que passam por reajustes ao mesmo, seguindo as regras do modelo de dados LP64, e que são apresentadas, seguidamente, como "problemas na portabilidade" [10][11][33].

2.3.1 Problemas na Portabilidade

As aplicações resultantes da adaptação de ILP32 para LP64, sem se efectuar qualquer alteração ao código, serão afectadas funcionalmente devido ao comprimento de um ou mais tipos de dados inteiros ser diferente em ambos os modelos de dados. Assim [10][11][33]:

- os tipos de dados *long* e *int* deixam de ter o mesmo comprimento;
- os tipos de dados *int* e ponteiro deixam de ter o mesmo comprimento;
- os tipos de dados *long* e ponteiro têm um comprimento e alinhamento de 64 bits;
- os tipos de dados pré-definidos *size_t* e *ptrdiff_t* designam um inteiro de 64 bits;
- os objectos, tais como as estruturas de dados definidas pelo programador, definidos com tipos de dados de 64 bits terão um tamanho diferente se declarados em modelos de dados de 16 ou 32 bits;
- a relação *sizeof(int) = sizeof(long) = sizeof(pointer)* válida em ILP32 deixa de ser válida em LP64.

Os efeitos acima indicados podem ter impacto nos itens abaixo, os quais passam a ser os problemas que teremos na portabilidade de código [10][11][33]:

- truncamento de dados;
- ponteiros;
- promoção de tipos de dados;
- alinhamento de dados;
- constantes;
- *bit shifts* e *bit masks*;
- formatação de *strings*.

Para cada um serão apresentados os problemas que ocorrem na portabilidade de ILP32 para LP64 e exemplos de código em linguagem de programação C (compilador *gcc*) que denotam tais problemas, bem como a(s) solução(ões) para colmatar os mesmos, que passam sempre por alterações ao código fonte.

2.3.1.1 Truncamento de Dados

Os programadores erroneamente assumem que o tipo de dados ponteiro tem o mesmo comprimento do que o do tipo de dados inteiro. Por tal, assumem que podem transferir quantidades de informação entre tipos de dados sem perder informação. Tal suposição não é verdadeira na passagem de ILP32 para LP64, porque no modelo LP64 os tipos de dados *int*, *long* e *pointer* têm comprimentos diferentes (*int* 32 bits e *long* e *pointer* 64 bits), levando à perda de informação (truncamento de dados). Assim, a afectação de um *pointer* a um *int* ou de um *long* a um *int* resulta em truncamento, por atribuição de um tipo de dados de maior comprimento a um tipo de dados de menor comprimento [10][11][33].

Na conversão para o modelo de dados LP64, o truncamento de dados pode ocorrer durante a inicialização, a atribuição, a passagem de parâmetros, o retorno do valor de uma função e a conversão de tipos de dados (*cast*).

Exemplo 1: *Atribuição de um long a um int*

No modelo LP64, a instrução *a = b* (Figura 2.3) efectua a atribuição de um inteiro longo (*b*) a um inteiro (*a*), isto é, atribui 64 bits a 32 bits, levando à perda dos 32 bits mais significativos de *b*. Tal truncamento de dados produzirá valores errados se o valor da variável *b* for maior do que o máximo valor inteiro permitido para o tipo de dados *int*.

```
int a;
long b;
...
a = b;
```

Figura 2.3: Truncamento de dados: atribuição de um *long* a um *int*

Solução:

Na operação de atribuição, utilizar variáveis que tenham o mesmo tipo. Verificar se a variável *b* necessita de ser do tipo *long*. Se assim se verificar, declarar ambas as variáveis como *long*, caso contrário declará-las como *int*.

Exemplo 2: *Atribuição de um pointer a um int*

No modelo LP64, os ponteiros serão truncados se atribuídos a variáveis de tipo inteiro, pelo facto de se atribuir 64 bits a 32 bits. Na Figura 2.4, a linha 2 efectua a atribuição do ponteiro da variável *i* à variável *j*, que resultará em perda de dados e, consequentemente, em valores errados.

```
1  int i = 10, j;
2  j = (int) &i;
```

Figura 2.4: Truncamento de dados: atribuição de um *pointer* a um *int*

Solução:

Existem diversas soluções, dependendo do cenário de portabilidade:

- se a aplicação em causa for portada somente para 64 bits, é suficiente declarar as variáveis *i* e *j* como *long*, uma vez que os tipos de dados *long* e *pointer* têm o mesmo comprimento (Figura 2.5 a));
- se a aplicação em causa for executada em sistemas de 32 e 64 bits, então o valor do endereço de memória da variável *i* é armazenado numa variável declarada como ponteiro (Figura 2.5 b)) ou declarar *j* como *intptr_t* (definido na biblioteca <*inttypes.h*>), isto é, recorrer a um tipo de dados pré-definido de 64 bits portável, que armazena um ponteiro num inteiro (Figura 2.5 c)).

```
1  long i = 10, j;
2  j = (long) &i;
```

a) declarar *i* e *j* como *long*

```
1  int i = 10;
2  int *j;
3  j = &i;
```

b) declarar *j* como ponteiro para um inteiro

```
1  #include <inttypes.h>
2  int i = 10;
3  intptr_t j = &i;
```

c) declarar *j* como *intptr_t*

Figura 2.5: Solução para a atribuição de um *pointer* a um *int*

Exemplo 3: *Retorno do valor de uma função*

No código seguinte, na Figura 2.6, a função *calcula_offset()* retorna a diferença entre dois ponteiros. No modelo ILP32, este resultado pode ser atribuído a uma variável do tipo *int* ou *long*, mas, no modelo LP64, ele só pode ser atribuído a uma variável do

tipo *long*, pelo facto de o ponteiro ter comprimento de 64 bits e a diferença entre ponteiros também. Assim sendo, no modelo LP64, o resultado devolvido pela função *calcula_offset()* será truncado, uma vez que serão atribuídos 64 bits a 32 bits (variável *offset*).

```
 1  int calcula_offset(int *ptr_1, int *ptr_2);
 2  int main()
 3  {
 4     int *ptr_1, *ptr_2;
 5     int offset;
 6     offset = calcula_offset(ptr_1, ptr_2);
 7     printf("O valor da diferenca dos ponteiros e' %d\n", offset);
 8     return 0;
 9  }
10
11  int calcula_offset(int *ptr_1, int *ptr_2)
12  {
13     return (ptr_2 - ptr_1);
14  }
```

Figura 2.6: Truncamento de dados: retorno do valor de uma função

<u>*Solução*</u>:

Uma vez que a diferença entre dois ponteiros, no modelo LP64, é um inteiro longo, bastará declarar a função *calcula_offset()* e a variável *offset* do tipo *long* (Figura 2.7 a)).

Outra solução passa por recorrer ao tipo de dados pré-definido *ptrdiff_t* (da biblioteca <*stddef.h*>) para declarar a função *calcula_offset()*. O comprimento do valor de retorno desta função está dependente do comprimento do tipo de dados ponteiro da plataforma (32 ou 64 bits) onde a aplicação está a ser executada. Podemos então declarar a variável *offset* como *long*, pois em qualquer uma das plataformas ela terá o comprimento do tipo de dados ponteiro (Figura 2.7 b)).

```
 1  long calcula_offset(int *ptr_1, int *ptr_2);
    ...
 5  long offset;
```

a) declarar *calcula_offset()* e *offset* como *long*

```
    #include <stddef.h>
 1  ptrdiff_t calcula_offset(int *ptr_1, int *ptr_2);
    ...
 5  long offset;
```

b) declarar *calcula_offset()* como *ptrdiff_t* e *offset* como *long*

Figura 2.7: Solução para o valor retornado por uma função

2.3.1.2 Ponteiros

Os seguintes itens apresentam os problemas no manuseamento de ponteiros, corrompendo-os, aquando da conversão para LP64 [10][11][33]:

- atribuição de uma constante de 32 bits (representada em hexadecimal) ou um inteiro a um ponteiro, irá resultar num endereço de memória inválido, que poderá causar erros quando o ponteiro for dereferenciado (conteúdo da variável para a qual o ponteiro está a apontar);
- converção (*cast*) de um ponteiro para um inteiro, ocorrerá truncamento de dados;
- converção (*cast*) de um inteiro para um ponteiro, poderá resultar em erros quando o ponteiro for deferenciado, por resultar num endereço de memória inválido;
- devolução de ponteiros por funções e seu armazenamento em inteiros, poderá retornar valores truncados;
- comparação de um inteiro com um ponteiro, poderá causar resultados inesperados;
- conversão (*cast*) de um long* para um int*, ocorrerá truncamento de dados, porque o ponteiro do conteúdo de um inteiro longo tem um comprimento de 64 bits, enquanto que o ponteiro do conteúdo de inteiro tem um comprimento de 32 bits.

Um outro cuidado a ter são as operações aritméticas de ponteiros, uma vez que a linguagem de programação C incrementa um ponteiro com o tamanho do tipo de dados que ele referencia. Por exemplo, se a variável p é um ponteiro para um *long*, a operação ($p = + 1$) incrementa o valor de p em 4 bytes no modelo de dados ILP32, onde *long* tem um comprimento de 32 bits, e em 8 bytes no modelo de dados LP64, onde *long* tem um comprimento de 64 bits.

Exemplo 1: *Aritmética de ponteiros entre longs e ints*

Quando utilizamos tipos de dados incorrectos no deferenciamento de ponteiros, podemos obter resultados incorrectos. No modelo ILP32, um ponteiro de uma variável do tipo *long* pode ser deferenciado e armazenado numa variável do tipo *int* e vice-versa, porque ambos os tipos de dados (*long* e *int*) têm o mesmo comprimento e alinhamento. No modelo LP64, o mesmo não se verifica. Se um valor do tipo *long* é deferenciado por um ponteiro de *int*, somente os primeiros 32 bits do *long* irão ser deferenciados.

No código da Figura 2.8 observa-se que temos valores do tipo *long* na variável vector (linha 3) e dois ponteiros (*j* e *k*) a apontar, cada um, para um *int* e um *long*, nas linhas 5 e 6, respectivamente.

```
 1  int main()
 2  {
 3     long vector[5];
 4     int indice;
 5     int *j;
 6     long *k;
 7     for (indice = 0; indice < 4; indice++)
 8        vector[indice] = indice + 1;
 9     j = vector + 2;
10     printf ("O valor deferenciado por j e' %d\n", *j );
11     k = vector + 2;
12     printf ("O valor deferenciado por k e' %d\n", *k );
13     return 0;
14  }
```

Figura 2.8: Ponteiros: aritmética de ponteiros entre *longs* e *ints*

Estando os dois ponteiros a apontar para o mesmo endereço de memória, quando deferenciamos o valor de cada um obtemos resultados diferentes, nomeadamente 0 e 3, para *j* e *k*, respectivamente. O ponteiro *j* deferência erradamente os primeiros 32 bits dos 64 bits do valor tipo *long* (64 bits), pelo facto de ser um ponteiro de *int* (32 bits). Por sua vez, o ponteiro *k* deferencia todos os 64 bits, por ser um ponteiro de *long*.

Solução:
Para colmatar este resultado inesperado, basta alterar o tipo de dados do ponteiro *j* de *int* para *long*, porque o tipo de dados do objecto (*vector*) que irá referenciar e deferenciar é *long*.

Exemplo 2: *Conversão (casting) de pointers em ints ou ints em pointers*

No modelo LP64, a conversão (*casting*) de ponteiros em inteiros (Figura 2.9 - linha 7) irá causar resultados inesperados, porque os tipos de dados *pointer* e *int* não têm o mesmo comprimento, como acontece no modelo ILP32, resultando em truncamento de dados. Verifica-se que o contrário também causará resultados inconsistentes, ou seja, armazenar um *int* num *pointer* (linha 8), quando o ponteiro for deferenciado, por conter um endereço de memória inválido.

```
 1  int main()
 2  {
 3    int i = 7;
 4    int j;
 5    int *p;
 6    p = &i;
 7    j = (int)p;
 8    p = j;
 9    j = *p;
10    return 0;
11  }
```

Figura 2.9: Ponteiros: conversão (casting) de *pointers* em *ints* ou *ints* em *pointers*

Solução:

Para contornar os problemas indicados é suficiente declarar as variáveis *j* e **p* do tipo *long*, que em ambos os modelos terão o comprimento do tipo de dados ponteiro.

2.3.1.3 Promoção de Tipos de Dados

Quando operações aritméticas e de comparação são efectuadas entre variáveis e constantes com tipos de dados diferentes, a linguagem de programação C converte, antes de efectuar as operações, os referidos tipos de dados para tipos de dados compatíveis, dando-se assim lugar à promoção de tipos de dados (ver secção 2.1.2). Por exemplo, quando um *short* é comparado com um *long*, ele primeiramente é convertido num *long*, ou seja, o *short* é promovido para o tipo *long*.

No entanto, a promoção de tipo de dados no modelo LP64 é efectuada de forma diferente do que no modelo ILP32, quando inteiros sem sinal (*unsigned int*) são comparados com inteiros longos (*long*) e quando inteiros (*int*) são comparados com inteiros longos sem sinal (*unsigned long*). Para evitar resultados inesperados, o código dos programas deve contemplar operações aritméticas e de comparação, somente quando os operandos forem todos com sinal (*signed*) ou sem sinal (*unsigned*) [10][11][33].

Exemplo 1: *Operação aritmética*

Certas promoções de tipos de dados resultam em números com sinal (*signed*), os quais são tratados como números sem sinal (*unsigned*). Quando esta situação ocorre, poderemos obter resultados inesperados. Por exemplo, na Figura 2.10, o resultado da

expressão (*i* + *j*) resulta num valor *long* com sinal (*result*), mas os operandos *i* e *j* são tratados como números inteiros sem sinal, por um deles ser sem sinal (*j*).

```
long result;
int i = -2;
unsigned int j = 1;
result = i + j;
```

Figura 2.10: Promoção de tipos de dados: operação aritmética

Se o excerto de código da Figura 2.10 for executado sob o modelo de dados ILP32, observamos que o resultado intermédio da expressão (*i* + *j*), antes de ser atribuído à variável *result*, é um número inteiro sem sinal (*unsigned int*), pelo facto de um dos operandos ser do tipo inteiro sem sinal (*j*). O tipo de dados do resultado intermédio quando atribuído à variável *result* é promovido de *int* para *long* com sinal (*signed long*), pelo facto da variável *result* assim o ser. O resultado final (-1) está correcto porque os tipos de dados *int* e *long* têm o mesmo comprimento (32 bits) no modelo de dados ILP32.

Valores originais:

0xFFFFFFFE	+	0x00000001
i = -2		j = 1
(*signed*)		(*unsigned*)

Valores intermédios:

0xFFFFFFFE	+	0x00000001	=	0xFFFFFFFF
4.294.967.294		1		4.294.967.295
(*unsigned*)		(*unsigned*)		(*unsigned*)

Resultado final:

	=	0xFFFFFFFF
		result = -1
		(*signed*)

Figura 2.11: Promoção de tipo de dados em ILP32

Por seu turno, se o mesmo excerto de código for executado sob o modelo de dados LP64 (Figura 2.12), observamos também que o resultado intermédio da expressão (*i* + *j*), antes de ser atribuído à variável *result*, é um número inteiro sem sinal (*unsigned int*) pela mesma razão verificada anteriormente. O tipo de dados do resultado intermédio, quando atribuído à variável *result*, é promovido de *int* para *long* com sinal (*signed long*), pelo facto da variável *result* assim o ser. No entanto, o resultado final é bem diferente do que o previsto, porque os tipos de dados *int* e *long* não têm o mesmo

comprimento (32 e 64 bits, respectivamente), sendo os 32 bits mais à esquerda do *long* de 64 bits preenchidos a zero, resultando desta forma um inteiro longo positivo.

Valores originais:

0xFFFFFFFE	+	0x00000001
i = -2		j = 1
(*signed*)		(*unsigned*)

Valores intermédios:

0xFFFFFFFE	+	0x00000001	=	0xFFFFFFFF
4.294.967.294		1		4.294.967.295
(*unsigned*)		(*unsigned*)		(*unsigned*)

Resultado final:

= 0x00000000FFFFFFFF
result = 4.294.967.295
(*signed*)

Figura 2.12: Promoção de tipo de dados em LP64

Solução:

Por forma a evitar o valor inesperado resultante no modelo LP64, poderíamos declarar a variável *result* como *int*, em vez de *long*.

Exemplo 2: *Operação de comparação*

Quando o código da Figura 2.13 é executado no modelo ILP32, o valor da variável *L* (*long*) é promovido a um inteiro sem sinal, pelo facto de ser comparado com um *unsigned int*, passando a ser um inteiro grande positivo. O resultado da comparação é erradamente verdadeiro, imprimindo no ecrã a mensagem

```
L e' maior do que i
```

Este mesmo código executado no modelo LP64, ambas as variáveis (*L* e *i*) são promovidas para inteiros longos com sinal (*signed long*), pelo facto do tipo *long* ter maior comprimento do que o tipo *int* e de ser com sinal (*signed*). No entanto, ambos os valores das variáveis *L* e *i* são inalterados porque *L* já é *signed* e o valor de *i* pertence ao intervalo positivo do tipo de dados *signed long*, respectivamente. O resultado da comparação é correctamente falso, imprimindo no ecrã a mensagem

```
L e' menor do que i
```

```
 1  int main()
 2  {
 3    long L = -1;
 4    unsigned int i= 1;
 5    if (L > i)
 6        printf ("L e' maior do que i\n");
 7    else
 8        printf ("L nao e' maior do que i\n");
 9    return 0;
10  }
```

Figura 2.13: Promoção de tipos de dados: operação de comparação

Solução:

Efectuar a operação de comparação com tipos de dados iguais, passando por declarar a variável *i* (linha 4) como um inteiro longo (Figura 2.14 a)) ou, na linha 5, converter (*cast*) a variável *i* para *long* (Figura 2.14 b))

```
4  long i = 1;
```

a) declarar *i* como *long*

```
5  if (L > (long) i)
```

b) converter *i* para *long*

Figura 2.14: Solução para operação de comparação

2.3.1.4 Alinhamento de Dados

Quando, no código de um programa, criamos estruturas de dados (*struct*), contendo tipos de dados de comprimentos diferentes, o alinhamento dos dados é efectuado pelo tipo de dados de maior comprimento, onde o espaço de memória ocupado pela estrutura é igual ao número de variáveis nela definida multiplicado pelo tamanho do tipo de dados de maior comprimento. Desta forma, podemos facilmente concluir que se as estruturas de dados contiverem tipos de dados do tipo *long* e/ou *pointer* as mesmas terão alinhamentos diferentes quando declaradas em ILP32 ou em LP64. Tal diferença de alinhamento, ou melhor, ocupação de espaço da estrutura, poderá produzir efeitos inesperados quando trabalhamos com ficheiros, ou mesmo com chamadas de procedimento remoto (RPC – *Remote Procedure Calls*), ou com protocolos de rede, uma vez que estes terão alinhamentos diferentes dos esperados, levando à leitura incorrecta e/ou bloqueio do sistema [10][11][33].

Exemplo 1: *Alinhamento de dados de uma estrutura de dados*

A seguinte estrutura de dados ilustrada na Figura 2.15 representa um aluno. Esta quando alinhada segundo o modelo de dados ILP32 ocupa 20 bytes (Figura 2.16), apurados pela multiplicação do número de variáveis que compõe a estrutura por 4 bytes, onde estes representam o tipo de dados de maior comprimento da estrutura (*long* e *pointer*).

```
struct aluno {
   long BI;
   char sexo;
   short idade;
   int matricula;
   struct aluno *next;
}
```

Figura 2.15: Estrutura de dados, representativa de um aluno

		Bits			
		8	16	24	32
Bytes	4	BI			
	8	sexo	padding	padding	padding
	12	idade		padding	padding
	16	matricula			
	20	*next			

Figura 2.16: Alinhamento da estrutura de dados aluno em ILP32

Por sua vez, e aplicando o mesmo raciocínio, a estrutura quando alinhada pelo modelo de dados LP64 (Figura 2.17) ocupará 40 bytes, porque o tipo de dados de maior comprimento é o *long* ou o *pointer*, com 64 bits (8 bytes).

		Bits							
		8	16	24	32	40	48	56	64
Bytes	8	BI							
	16	sexo	padding	padding	padding	padding	padding	padding	padding
	24	idade		padding	padding	padding	padding	padding	padding
	32	matricula				padding	padding	padding	padding
	40	*next							

Figura 2.17: Alinhamento da estrutura de dados aluno em LP64

Solução:

De acordo com a Figura 2.18, definir dentro da estrutura primeiramente os campos (membros) de tipo de dados de maior comprimento e seguidamente os de menor

comprimento, por forma a conter menos espaços vazios (*padding*), e recorrer a macros (ver secção 2.3.2.2), da linguagem de programação C.

```
struct aluno {
   long BI;
   struct aluno *next;
   int matricula;
   short idade;
   char sexo;
}
```

Figura 2.18: Solução do alinhamento de dados de uma estrutura de dados

É de referir que uma aplicação portável entre 32 e 64 bits, com inter-operacionalidade (troca de dados entre sistemas de plataformas diferentes), tem de assegurar que o tamanho e alinhamento dos dados trocados/partilhados entre as aplicações de 32 e 64 bits são iguais dentro delas.

Nesta troca de dados estão implícitos os ficheiros de dados, mensagens de protocolos de rede e de RPC, onde todos eles são definidos por estruturas de dados (*struct*) dentro das aplicações que os criam e/ou manuseiam. Por tal, um dos maiores problemas e cuidados a ter na portabilidade de código de ILP32 para LP64 é o alinhamento de dados, que, como vimos, é conseguido por estruturas de dados que têm diferentes tamanho nos dois modelos, caso não façamos alterações ao seu código.

Uma estrutura de dados definida incorrectamente originará truncamento de dados, caso um dos seus membros seja do tipo *long* ou *pointer* e a inter-operacionalidade seja efectuada de uma aplicação de 64 bits para uma de 32 bits. A inter-operacionalidade em sentido contrário, de uma aplicação de 32 para 64 bits, poderá ocorrer em aumento de espaço ocupado pela estrutura de dados e, consequentemente, mapeamento de memória incorrecto.

Em suma, para evitar problemas de inter-operacionalidade é recomendado, o quanto possível, a utilização de tipos de dados, nas estruturas definidas nos programas, com o mesmo comprimento e alinhamento em ambos os modelos de dados. Caso não seja possível, recorrer, então, a macros e a tipos de dados pré-definidos.

2.3.1.5 Constantes

Um programa com constantes expressas em hexadecimal quando portado de ILP32 para LP64, as variáveis dos tipos de dados afectos às constantes podem sofrer alterações.

No modelo LP64, as constantes hexadecimal de 32 bits podem deixar de representar os valores correctos de ponteiros ou máscaras (Tabela 2.13). Na conversão, o ponteiro será aumentado de 32 bits à esquerda, todos de valor zero, ficando com um valor diferente do que em ILP32 [10][11][33].

Constante	ILP32	LP64
0x7fffffff	*int*	*int*
0x7fffffffL	*long*	*long*
0x80000000	*unsigned int*	*unsigned int*
0x80000000L	*unsigned long*	*long*

Tabela 2.13: Constantes hexadecimais em ILP32 e LP64

Os programadores que assumem que uma constante de um longo inteiro é representada por 32 bits, independentemente do modelo de dados, terão resultados errados, pelo facto de no modelo de dados LP64 o tipo de dados inteiro longo ter 64 bits. Por forma a evitar a ocorrência de valores hexadecimal e máscaras de bits incorrectas, recorre-se a macros (ver secção 2.3.2.2) que permitem especificar o código correcto, consoante a plataforma.

2.3.1.6 *Bit Shifts* e *Bit Masks*

Uma das suposições erradas que os programadores habitualmente fazem, quando codificam operações de deslocamento de bits (*bit shifts*) e máscaras de bits (*bit masks*), é a de que o tipo de dados das variáveis envolvidas nas operações de bits é igual ao tipo de dados da variável que irá receber o resultado da operação.

Conhecendo o conceito de "promoção de tipo de dado", apresentado anteriormente, em que perante a instrução

```
a = b operador c
```

sabe-se que o tipo de dados do resultado intermédio (*b operador c*) depende dos tipos de dados das variáveis *b* e *c*, e que esse tipo de dados, caso haja necessidade, é promovido para o tipo de dados da variável que receberá o resultado intermédio (*a*). Se o comprimento do tipo de dados da variável *a* for 64 bits, mas o comprimento das variáveis *b* e *c* for 32 bits, então o resultado intermédio ocorrerá em *overflow*, ou será truncado antes de ser atribuído à variável *a* [10][11][33].

Exemplo 1: *Bit Shift overflow*

Analisando o código da Figura 2.19, o operando 1 é uma constante inteira, a qual é tratada como um valor de 32 bits em qualquer um dos modelos ILP32 e LP64. Verifica-se que ocorrerá *overflow* em ambos os modelos de dados pelo facto do tipo de dados do resultado intermédio ter 32 bits e o deslocamento efectuado ser de 32 bits, ultrapassando o limite do comprimento do tipo de dados do resultado intermédio.

```
unsigned long y;
y = 1 << 32;
```

Figura 2.19: *Bit shift overflow* à esquerda

No entanto, no modelo de dados LP64 o resultado intermédio atribuído à variável *y* será indefinido (Figura 2.20), uma vez que esta tem 64 bits de comprimento e receberá um valor intermédio que foi extravasado (ocorrência de *overflow*).

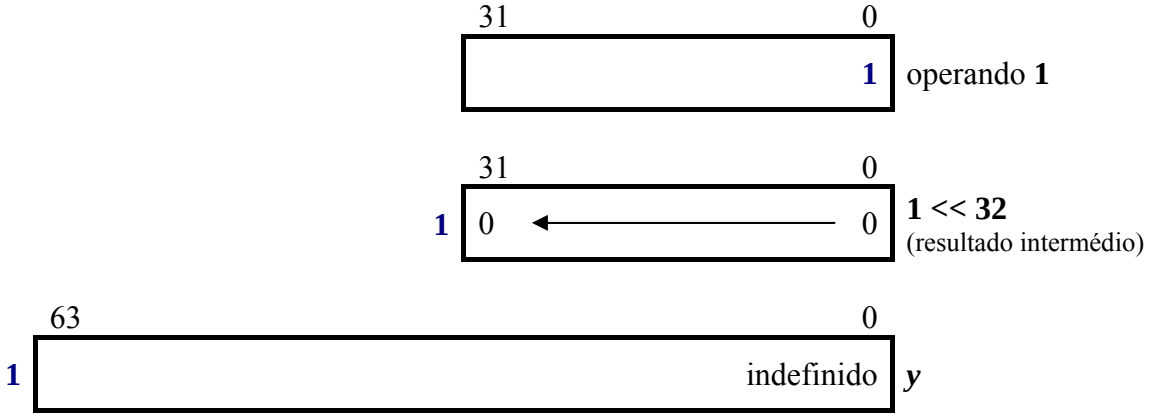


Figura 2.20: *Bit shift overflow* em LP64

Solução:

Para esta situação, a solução passa por colocar o sufixo L ou UL no operando 1 de forma a defini-lo como *long* ou *unsigned long*, respectivamente, como se pode observar na Figura 2.21.

```
unsigned long y;
y = 1L << 32;
```

Figura 2.21: Solução de *bit shift* à esquerda em LP64

Desta forma, asseguramos que pelo menos uma das variáveis do resultado intermédio é do tipo de dados da variável final e que o deslocamento de bits não resultará em *overflow*, quando executado em LP64, uma vez que o tipo *long* tem o comprimento de 64 bits, o qual é superior ao número de bits do deslocamento. Contudo, o

extravasamento de bits continuará perante o modelo ILP32, pelo facto do tipo *long* ter comprimento igual ao do tipo *int* (32 bits), ou seja, o número de bits a ser deslocados é igual ao comprimento do tipo de dados da variável.

Exemplo 2: *Bit Shift à esquerda*

Observando o código da Figura 2.22, verifica-se a atribuição de um *long* a um *int* (linha 6). No modelo ILP32 funciona correctamente, por ambos os tipos de dados terem o mesmo comprimento, mas resultará em truncamento de dados, quando executado no modelo LP64, por ser atribuído 64 bits a 32 bits, onde os primeiros 32 bits serão truncados.

```
 1  #include <limits.h>
 2  int main()
 3  {
 4    long base = LONG_MAX; /* maximo longo inteiro positivo possivel
*/
 5    int resultado;
 6    resultado = base << (LONG_BIT-1) /* LONG_BIT-1=63 bits, em LP64
*/
 7    printf("\n%016x", resultado);
 8    return 0;
 9  }
```

Figura 2.22: *Bit shift* à esquerda

Solução:

Ao trabalharmos com deslocamento de bits (*bits shifts*) devemos declarar, tanto a variável que sofrerá o deslocamento de bits, como a que receberá o resultado, do mesmo tipo de dados. Neste exemplo, as variáveis *base* e *resultado* deverão ser declaradas como *long*.

2.3.1.7 Formatação de *Strings*

As funções que envolvem formatação de *strings*, tais como *printf()*, *sprintf()*, *scanf()* e *sscanf()*, na portabilidade de ILP32 para LP64 podem gerar resultados inesperados, quando, nos seus parâmetros, especificamos o tipo *long* como um *int* e/ou o tipo de dados *pointer* (endereço de memória) como um valor hexadecimal [10][11][33].

Exemplo 1: *scanf*

Na linha 4 do código seguinte (Figura 2.23) observa-se a leitura de um número inteiro, especificado pelo parâmetro *%d*, para a variável *varlong*, do tipo *long*.

```
1 int main()
2 {
3    long varlong;
4    scanf("%d", &varlong);
5    return 0;
6 }
```

Figura 2.23: Formatação de *strings: scanf*

Neste excerto de código, quando executado em ILP32, não é observado nenhum resultado incongruente, pelo facto dos tipos de dados *int* e *long* terem o mesmo comprimento. Contudo, no modelo LP64 o mesmo não acontece, verificando-se o truncamento de dados, pela perda de 32 bits, quando se efectua a suposta leitura de um *long*.

Solução:

Substituir (Figura 2.24) o parâmetro *%d* por *%ld*, especificando assim o formato de um inteiro longo, o qual é reconhecido por ambos os modelos de dados.

```
4    scanf("%ld", &varlong);
```

Figura 2.24: Solução do *scanf*

Exemplo 2: *printf*

Na linha 6 do código da Figura 2.25 observa-se a escrita para o ecrã do conteúdo da variável *varlong*, especificada como um número inteiro, pelo parâmetro *%d*, e do endereço de memória, em hexadecimal, da variável *ptr*, especificado pelo parâmetro *%x*.

```
1 int main()
2 {
3    long varlong;
4    long *ptr;
5    scanf("%ld", &varlong);
6    printf("varlong: %d ptr: %x", varlong, ptr);
7    return 0;
8 }
```

Figura 2.25: Formatação de *strings: printf*

Se executarmos este código no modelo ILP32 não ocorre qualquer truncamento de dados por os tipos de dados *int*, *long* e *pointer* terem todos o mesmo comprimento. Contudo, no modelo LP64, o mesmo não acontece, verificando-se o truncamento de dados aquando da escrita de ambas as variáveis.

Solução:
Quando desejamos escrever um ponteiro, fazê-lo sempre pelo parâmetro *%p* (Figura 2.26). Para escrever um inteiro longo, combinar o parâmetro *l* com *d*, *u*, *o* e *x* formatando a *string* como longo inteiro (*%ld*), longo inteiro sem sinal (*%lu*), longo octal (*%lo*) e longo hexadecimal (*%lx*), respectivamente.

```
6    printf("varlong: %ld ptr: %p", varlong, ptr);
```

Figura 2.26: Solução do *printf*

2.3.2 Escrever Código Portável

Quando pretendemos portar código, para que o mesmo seja executável em sistemas de 32 e 64 bits, temos de efectuar alterações ao mesmo, tendo em conta os problemas da portabilidade acima apresentados. No entanto, existe por vezes a necessidade de recorrer a tipos de dados pré-definidos, contidos nas bibliotecas *inttypes.h* e *stddef.h*, onde a primeira define os tipos de dados inteiros, enquanto que a segunda define variáveis padrões, tais como *size_t* e *ptrdiff_t*. Para além das referidas bibliotecas, podemos também ter de construir macros da linguagem C, que definem as acções a serem efectuadas, consoante o sistema onde seja executada a aplicação.

2.3.2.1 Definição de Tipos de Dados Inteiros

A biblioteca *inttypes.h* contém a definição dos tipos de dados inteiros que podem ser utilizados na construção ou recompilação de programas, por forma a que estes sejam executados em ambas as plataformas de 32 e 64 bits. De uma forma geral, o conteúdo da referida biblioteca contém a representação de cada tipo de dados primitivo da linguagem

de programação C, de acordo com o comprimento em bits (Tabela 2.14) [15], a representação de ponteiros para tipos de dados inteiros com e sem sinal, de comprimento de 32 e 64 bits (Tabela 2.15) [15], e macros para constantes e para funções de formatação de *strings* (*printf()* e *scanf()*).

Tipo de dado	Descrição
int8_t	Inteiro de 8 bits com sinal (*signed*)
uint8_t	Inteiro de 8 bits sem sinal (*unsigned*)
int16_t	Inteiro de 16 bits com sinal (*signed*)
uint16_t	Inteiro de 16 bits sem sinal (*unsigned*)
int32_t	Inteiro de 32 bits com sinal (*signed*)
uint32_t	Inteiro de 32 bits sem sinal (*unsigned*)
int64_t	Inteiro de 64 bits com sinal (*signed*)
uint64_t	Inteiro de 64 bits sem sinal (*unsigned*)

Tabela 2.14: Tipos de dados inteiros definidos em *inttypes.h*

No que concerne à representação dos ponteiros, a mesma é desta maneira para que um ponteiro possa ser movido para ou de um tipo de dados inteiro sem ser corrompido.

Tipo de dado	ILP32	LP64
intptr_t	Inteiro de 32 bits com sinal (*signed*)	Inteiro de 64 bits com sinal (*signed*)
uintptr_t	Inteiro de 32 bits sem sinal (*unsigned*)	Inteiro de 64 bits sem sinal (*unsigned*)

Tabela 2.15: Tipos de dados de ponteiros definidos em *inttypes.h*

A utilização destes tipos de dados pré-definidos é efectuada pela substituição dos tipos de dados primitivos da linguagem de programação, de acordo com o seu comprimento em bits.

2.3.2.2 Macros

As macros são instruções que servem para especificar secções de código para determinada plataforma. O recurso às mesmas é efectuado quando não é possível resolver a portabilidade de código pelos tipos de dados primitivos da linguagem e/ou pelos tipos de dados pré-definidos [10][11][33][34].

A definição das macros pode ser realizada em qualquer parte do código do programa, ficando nele encapsulada, e tem a estrutura ilustrada na Figura 2.27.

```
1  #ifdef _LP64
2     ... /* codigo especificamente para LP64 */
3  #else
4     ... /* codigo especificamente para ILP32 */
5  #endif
6     ... /* codigo comum para as duas plataformas */
```

Figura 2.27: Estrutura de uma macro

No código acima, pode visualizar-se as secções delimitadas para cada uma das plataformas (LP64 e ILP32) e também a palavra-chave para identificar e testar a presença da plataforma de 64 bits (_LP64). No entanto, esta palavra-chave varia consoante o compilador de 64 bits (Figura 2.28), onde no teste lógico da macro (linha 1) pode-se colocar várias palavras-chaves para os diversos compiladores.

```
1  #if defined (_LP64) || defined (__LP64__) || defined (__64BIT__) ||
     (__WORDSIZE == 64)
2     printf("na presenca de LP64\n");
3  #else
4     printf("na presenca de ILP32\n");
5  #endif
```

Figura 2.28: Macro para identificar LP64 em diferentes compiladores

Nos exemplos de "Alinhamento de Dados" e "Constantes", acima apresentados, a solução para a portabilidade de ILP32 para LP64 passa pela construção de macros, encapsuladas no código.

Exemplo 1: *Alinhamento de dados*

Nas linhas 2 a 6, do código da Figura 2.29, é definida a macro para a variável BI do tipo *long*, onde no modelo LP64 ela ocupa 8 bytes e no modelo ILP32 também se definida com o tipo de dados *long long*.

```
 1  struct aluno {
 2     #ifdef _LP64
 3        long BI;
 4     #else
 5        long long BI;
 6     #endif
 7     struct aluno *next;
 8     int matricula;
 9     short idade;
10     char sexo;
11  }
```

Figura 2.29: Macros: alinhamento de dados de uma estrutura de dados

Estando os membros da estrutura de dados aluno declarados por ordem decrescente do seu comprimento e com a definição da macro, em ambos os modelos de dados a referida estrutura encontra-se alinhada e ocupa 24 bytes.

Exemplo 2: *Constantes*

O código seguinte (Figura 2.30) define a constante *const_id* do tipo de dados inteiro longo, com *bit mask* em hexadecimal, recorrendo a macros e aos tipos de dados pré-definidos na biblioteca *inttypes.h*.

```
1  #include <inttypes.h>
2  #ifdef _LP64
3     int64_t const_id = 0xfffffffffffffffc;
4  #else
5     int32_t const_id = 0xfffffffc;
6  #endif
```

Figura 2.30: Macros: constantes

Com o recurso a macros a constante fica com o valor correcto na plataforma. Caso contrário, a mesma seria definida como inteiro longo (Figura 2.31), resultando em código não portável.

```
long const_id = 0xfffffffc;
```

Figura 2.31: Constante não portável

2.3.3 Portabilidade de Código para ILP64 e LLP64

No que concerne aos problemas de portabilidade ocorridos na portabilidade de código do modelo ILP32 para os modelos de dados ILP64 e LLP64, à luz dos problemas detectados na portabilidade de código para LP64, temos que:

- **Modelo ILP64:**

  O único cuidado a ter na portabilidade é o alinhamento de dados, onde os tipos de dados *int*, *long* e ponteiro expandem-se para 64 bis, tornando o tamanho de cada registo da estrutura maior do que em ILP32.

  Tal como no modelo LP64, é necessário recorrer a macros para resolver este problema de alinhamento, de modo a garantir a inter-operacionalidade entre 32 e 64 bits. No

entanto, se for desejável manter os 32 bits de tamanho para o tipo de dados inteiro, é sempre possível utilizar o tipo *int32* ao contrário do tipo *int*.

- **Modelo LLP64:**

  Neste modelo de dados ocorrem problemas de portabilidade semelhantes aos do modelo LP64, pelo facto do ponteiro ser expandido para 64 bits, enquanto que os tipos de dados *int* e *long* mantêm os seus tamanhos de 32 bits. Assim, poderão ocorrer problemas de portabilidade em:

  - Truncamento de dados:
    - atribuir um ponteiro a um *int* ou a um *long*;
    - afectar um ponteiro, retornado por uma função, a um *int* ou a um *long*.
  - Ponteiros:
    - atribuir uma constante hexadecimal de 32 bits ou um inteiro ou um inteiro longo a um ponteiro irá resultar num endereço de memória inválido, que poderá causar erros quando o ponteiro for dereferenciado;
    - converter (*cast*) um ponteiro para um inteiro ou para um inteiro longo, ocorrerá truncamento de dados;
    - retornar ponteiros por funções e armazená-los em inteiros ou inteiros longos, poderá retornar valores truncados;
    - comparar um inteiro ou um inteiro longo a um ponteiro, poderá causar resultados inesperados.
  - Alinhamento de dados:

    O problema de portabilidade em alinhamentos de dados é visível somente quando a estrutura de dados contiver campos do tipo ponteiro, pois este é o único tipo que sofre alteração de tamanho entre os modelos de dados em questão.

2.4 Análise Estática de Código

A técnica de análise estática de código tem por objectivo detectar vulnerabilidades no código fonte ou binário (código objecto) da aplicação, sem que seja necessário executá-lo, ou seja, permite analisar e detectar problemas no código, através de análise de léxico,

sintáctica e semântica (verificação de tipos e modelos, análise de fluxo e de controlo de dados) [16][17][18][19][20].

A análise estática de código é efectuada por ferramentas que podem ser utilizadas durante a concepção/desenvolvimento da aplicação, não sendo necessário que a mesma esteja completa. Todavia, as ferramentas existentes procuram somente as vulnerabilidades para que foram programadas, com base nas regras e padrões definidos para o tipo de análise, não descobrindo novas vulnerabilidades. Desta forma, as ferramentas de análise produzem falsos negativos, significando que as aplicações contêm vulnerabilidades que não são reportadas pelas ferramentas, e falsos positivos, significando que as ferramentas reportam vulnerabilidades não existentes nas aplicações. Os falsos positivos causam grande preocupação aos analistas, porque exigem um exame minucioso do código. Por outro lado, os falsos negativos são muito mais perigosos porque conduzem a um falso sentido de segurança, pois uma ferramenta pode não encontrar qualquer vulnerabilidade, não significando isso que a aplicação não contenha vulnerabilidades e que é segura. Significa simplesmente que a aplicação não contem as vulnerabilidades para que foi testada [16].

Nas sub-secções seguintes apresentamos os diversos tipos de análise estática de código, com referência a algumas ferramentas de código aberto. No entanto, existem ferramentas comerciais que efectuam um ou mais destes tipos de análise estática, tornando-as complexas.

2.4.1 Análise de Léxico

Este tipo de análise estática de código é a mais simples, onde as ferramentas que a implementa procuram funções de bibliotecas ou chamadas de sistemas não fiáveis, tais como as funções *gets* e *strcpy*. Estes tipos de funções são designadas de não fiáveis por a sua utilização poder despoletar/explorar alguma vulnerabilidade, por regra geral serem funções que não verificam os limites possíveis para as variáveis. Por exemplo, a função *strcpy* é marcada como não fiável, porque a quantidade de *bytes* que são copiados da *string* origem para o tampão de memória (*buffer*) destino não é comparada com o tamanho do *buffer* destino. Assim, poderá originar a vulnerabilidade *buffer overflow* se a referida quantidade de *bytes* a copiar for superior ao tamanho do *buffer* receptor.

As ferramentas deste tipo realizam *parsing* do código fonte, de forma a dividi-lo em fragmentos (*tokens*) que possam ser comparados com informação armazenada em base de dados. Esta informação não é mais do que as funções não confiáveis e consideradas *perigosas* [17].

A forma como este tipo de análise funciona (por *parsing* e *tokens*) leva a que, por exemplo, variáveis cujo nome contém nome de funções pertencentes à base de dados sejam sinalizadas como funções, originando, assim, falsos positivos [17].

O *ITS4*, *Flawfinder* e *RATS* são ferramentas de análise de léxico para programas desenvolvidos em linguagem de programação C e C++.

O *ITS4* é o acrónimo de "*It's The Software, Stupid! [Security Scanner]*". Esta ferramenta faz uma análise de léxico do programa, gerando um conjunto de *tokens*, os quais são verificados contra funções vulneráveis pertencentes a uma base de dados [16][17][36][37][38].

O *Flawfinder* é uma ferramenta de código aberto escrita em *Python*. Possui uma base de dados com as funções não fiáveis das linguagens de programação C e C++. Identifica também vulnerabilidades de formatação de *strings*, nas funções *printf*, *fprintf*, *vprintf*, *snprintf*, *vsnprintf* e *syslog*. Também identifica vulnerabilidades *race condition*, encontradas em funções como *access*, *chown*, *chgrp*, *chmod*, *tmpfile*, *tmpnam* e *mktemp* [16][36][39].

O *RATS* (*Rough Auditing Tool for Security*) também é uma ferramenta de código aberto, desenvolvida pela *Secure Software Inc*. O objectivo desta ferramenta é encontrar possíveis problemas como *TOCTOU* e *buffer overflow* em códigos fonte escritos em C/C++, Perl, PHP e Python. Para cada uma destas linguagens o *RATS* possui uma base de dados XML com as funções não fiáveis [16][36][40].

2.4.2 Análise Semântica

Por análise semântica designam-se um conjunto de técnicas que permitem verificar aspectos da semântica do código fonte, desde a não declaração de variáveis e seus limites, análise de variáveis de controlo de ciclos e o fluxo de dados [17][44].

Este tipo de análise comporta as análises de verificação de tipos (*type checking*), análise de fluxo de controlo (*control-flow analysis*), análise de fluxo de dados (*data-flow analysis*) e verificação de modelos (*model checking*) que são apresentadas seguidamente.

2.4.2.1 Verificação de Tipos

A verificação de tipos de dados (*type cheking*) está associada à verificação dos limites máximo e mínimo que uma variável pode tomar, dependendo do seu tipo.

Algumas linguagens de programação, como o Java, têm implementada a verificação de tipos, garantindo que os valores das variáveis não ultrapassam os limites das mesmas. No entanto, outras, como o C e o C++, não fazem essa verificação.

Como consequência da não verificação de tipos temos a ocorrência de vulnerabilidades de inteiros, nomeadamente, *overflow* de inteiro, *underflow* de inteiro, *signedness* e truncamento, que podem estar associadas aos tamanhos dos tampões de memória (*buffers*). Estas, se exploradas, originam, por exemplo, *buffer overflow* e negação de serviço (*DoS*) [17].

Uma das formas de se realizar verificação de tipos passa por colocar linhas de código, nos lugares correctos, que verifiquem os limites possíveis para as variáveis e por recorrer a qualificadores de tipo (*type qualifiers*) para anotar o código fonte.

Ferramentas de análise estática de código que realizam este tipo de análise têm de rastrear as variáveis inteiras, tendo continuamente presente os limites máximo e mínimo para estas mesmas variáveis e assegurar que na atribuição de um valor a uma variável o mesmo primeiramente é verificado para os limites da variável.

A *CQual* [50] é uma destas ferramentas que recorre a qualificadores de tipos para rastrear as variáveis do tipo de dado do qualificador [47]. O seu modo de funcionamento passa por anotar o código fonte com os qualificadores de tipo estipulados. Estas anotações nada transmitem a um compilador, mas a *CQual* identifica as variáveis dos tipos a rastrear. De salientar, no entanto, que estes qualificadores de tipos somente tratam do tipo de dados por eles definidos, não garantindo a detecção ou ocorrência de vulnerabilidades em outros tipos de dados.

Em *CQual* [19], os qualificadores de tipos, definidos pelo programador, acabam por ser uma extensão da linguagem de programação, criando, assim, inferências de qualificadores de tipos de dados e seus relacionamentos. Assim, uma função, que espera um dado tipo de dados num dos seus parâmetros, poderá também receber qualquer um dos sub-tipos de dados do tipo de dados do qualificador.

Uma outra ferramenta de verificação de tipos é a *BOON* (*Buffer Overrun Detection*) [16][36][41][45][46], a qual detecta somente *buffers overflow*. Como grande parte dos *buffers overflows* ocorrem em *strings*, a metodologia desta ferramenta é modelar

cada uma das *strings* com duas propriedades. A primeira é a quantidade de *bytes* alocada para determinada variável, a segunda é a quantidade de *bytes* actualmente em utilização. Todas as funções que manipulam estas *strings* são então modeladas pelos efeitos causados nas duas propriedades analisadas. Estes efeitos são utilizados na realidade para gerar o que é apelidado de restrições. Por fim, estas restrições são resolvidas, as quais são então confrontadas com os valores inicialmente declarados para as *strings*. Caso possa ocorrer um *buffer overflow*, as devidas mensagens de aviso são geradas.

Para além destas duas ferramentas, também o *Lint* e o *Splint* (*Secure Programming Lint*) efectuam verificação de tipos.

A ferramenta *Lint* [22] está na origem de muitas outras ferramentas de análise estática de código, incluindo as comerciais. Esta ferramenta de sistemas *Unix* originalmente foi desenvolvida para ajudar a garantir a consistência das chamadas de funções, pela verificação dos limites possíveis para as variáveis passadas nos parâmetros das funções. Esta verificação de limites é o que apelidamos de verificação de tipos. Nas chamadas das funçõoes são verificados se os tipos de dados das variáveis que são passadas são iguais aos tipos de dados requeridos pelos argumentos das funções. Também é verificado, na adaptação de código de ILP32 para LP64, a existência de truncamento na atribuição de uma variável inteira (*int*, *long* ou *pointer*) de tipo de dados de maior largura a outra de menor largura.

A outra ferramenta mencionada, o *Splint* [16][23][35][36][41], é uma versão melhorada da ferramenta *LCLint*, cujas funcionalidades foram herdadas de ferramenta *Lint*. O *Splint*, comparativamente com o *Lint*, realiza as mesmas verificações de tipos, com excepção da atribuição de ponteiros a variáveis inteiras de menor largura do que os ponteiros. Mas, a funcionalidade de *signedness* é mais efeciente do que no *Lint*. A metodologia do *Splint* é recorrer a anotações inseridas pelo programador, tanto no programa como nas funções das bibliotecas padrões. Esta metodologia permite ao programador especificar pré-condições e/ou pós-condições para as funções. Para que as vulnerabilidades possam ser detectadas, as restrições geradas a partir das anotações no código fonte do programa devem ser resolvidas. Caso algum problema seja identificado na resolução desta restrições, será emitida uma mensagem de aviso ao utilizador. Por exemplo, se para a função *strcpy* for especificada uma pré-condição de que o comprimento da *string* destino (*buffer*) tem de ser maior do que o tamanho da *string* origem, caso esta restrição não seja resolvida será emitida uma mensagem de aviso. Às

especificações criadas e resolução de restrição é o que denominamos por verificação de tipos, onde está patente a verificação dos limites possíveis para as variáveis. No entanto, é bom referir que por defeito (sem anotação ao código) esta ferramenta monitoriza a criação e o acesso a *buffers*, detectando, assim, a violação de limites. Querendo com isso dizer que o programador recorre a anotações ao código somente quando o *Splint* não faz verificação de tipos em certas situações.

2.4.2.2 Análise de Fluxo de Controlo

Muitos dos algoritmos de análise estática de código exploram os diferentes caminhos de execução que um programa pode seguir, quando uma instrução é executada. Esta forma de análise é designada por análise de fluxo de controlo. Muitas das ferramentas deste tipo de análise são baseadas na construção de um grafo de fluxo de controlo. Os nodos do grafo são blocos de instruções sequenciais que serão sempre executadas, começando na primeira instrução e continuando até à última, sem a possibilidade de qualquer uma dessas instruções não ser executada [44].

O grafo de fluxo de controlo é baseado na construção de árvores sintácticas abstractas (AST – *Abstract Syntax Trees*), as quais representam relacões entre os diferentes módulos e funções da aplicação. A referida análise é realizada a três níveis: nível local, onde cada função é analisada separadamente; nível modular, no qual é analisada a interacção entre as funções de um módulo do programa; e nível global, onde é analisado o programa na sua totalidade, incluindo as interacções entre os diferentes módulos.

Este tipo de análise é utilizado para detectar problemas, tais como referências de ponteiros inválidas, utilização de memória não inicializada ou operações impróprias sobre os recursos do sistema (por exemplo, tentar fechar um ficheiro já fechado).

As ferramentas que efectuam este tipo de análise, numa primeira instância filtram (*parse*) o código fonte e constroem as AST. De seguida, a ferramenta efectua travessias nas AST, determinando os caminhos de fluxo de controlo. Por fim, os caminhos são simulados e as inconsistências, ou seja, as vulnerabilidades são identificadas.

Como exemplo de uma ferramenta deste tipo de análise para código C/C++ temos o *PREfix*, o qual simula a execução das funções, a acção de cada operador, as chamadas de funções, e traça diferentes caminhos de execução. As inconsistências (potenciais

vulnerabilidades) são detectadas quando a execução dos caminhos violam algum constrangimento [18].

2.4.2.3 Análise de Fluxo de Dados

Uma outra forma de analisar programas consiste em observar os caminhos possíveis que os dados do programa podem percorrer, ou seja, consiste em efectuar análise de fluxo de dados. Para tal, é necessário adicionar mais informação aos tipos de dados das variáveis que queremos rastrear e definir regras de inferência que permitirão detectar vulnerabilidades.

Uma das formas de fazer este tipo de análise é por *taint analysis*. A realização de *taint analysis* requer saber onde a informação entra no programa e como ela se move no mesmo. A isso designa-se de propagação *tainted*, a qual serve para identificar muitas das validações de *input* e representar defeitos. Por exemplo, um programa que contém uma vulnerabilidade de *buffer overflow*, contém sempre um caminho de fluxo de dados desde uma função de *input* até à instrução vulnerável. Assim, é possível rastrear variáveis de *input* e localizar as suas passagens em funções não fiáveis [44].

A *Taint analysis* permite definir a propriedade abstracta *taint* para as variáveis a tratar. O uso mais frequente desta propriedade abstracta *taint* é sinalizar variáveis como *tainted*, se o seu valor puder ser influenciado por um atacante (tipicamente, as variáveis que recebem *input*). Se, por acaso, uma variável *tainted* é utilizada para computar o valor de uma segunda variável, esta última também se tornará *tainted*, e assim sucessivamente. O objectivo consiste em detectar se dados *tainted* chegam a parâmetros de funções que tenham de ser *untainted*.

A ferramenta *LCLint* [48] realiza análise de fluxo de dados por anotações ao código, de forma a detectar vulnerabilidades de *buffers overflow*. *LCLint* permite ao programador definir estados para as funções, através de pré e pós condições. As anotações indicam hipóteses sobre os *buffers* que são passados para as funções. Também especificam o estado desses *buffers* quando retornados pelas funções.

Uma outra ferramenta que já foi anteriormente referida, em verificação de tipos, é o *CQual*, que tem a capacidade de detectar vulnerabilidades de formatação de *strings* por *taint analysis*. Os tipos são qualificados como *tainted* ou *untainted*, que combinados com os da linguagem de programação C, resultam em tipos de dados como *int, tainted int,*

*untainted int, tained char**. *CQual* considera todos os tipos de dados provenientes do exterior como dados *tainted*. Desta forma, *CQual* consegue traçar caminhos de fluxo de dados para as variáveis dos tipos de dados *tainted*, através da propagação das mesmas. A aplicação destes caminhos na detecção de vulnerabilidades de formatação de *strings* segue o seguinte princípio: se na execução de um caminho de fluxo de dados os dados *tainted* atingem funções de formação de *strings* e são interpretados como tal, então *CQual* emite uma mensagem de erro [16][41][50].

A ferramenta *Splint* [35] (sucessora de *LCLint*) também faz *taint analysis*. Esta ferramenta 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* [35]. 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 (dos qualificadores de tipos de *CQual*), 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 com que os valores dos atributos mudem. Por sua vez, as anotações definidas nos atributos servem para indicar hipóteses sobre os objectos em estudo.

Por fim, uma outra ferramenta que efectua este tipo de análise é a *Eau Claire* [16] [42][43]. Esta ferramenta converte o código fonte C em comandos de guarda, reforçados com excepções, afirmações, atribuição de instruções e estados erróneos. As vulnerabilidades são modeladas utilizando a linguagem de especificação ESC/Modula2, onde o programador define para as funções os requisitos de execução, as modificações após execução e o que garantem após execução. Os comandos de guarda são comparados com as especificações das funções.

É de salientar que as análises de fluxo de controlo e de dados são mais eficientes, por reduzirem o número de falsos positivos, comparativamente com as análises de léxico e sintática. Estas técnicas tentam determinar se as aparentes vulnerabilidades podem ser realmente exploradas [17].

2.4.2.4 Verificação de Modelos

A verificação de modelos (*model checking*) [51][52] tem como intuito verificar se um dado programa satisfaz um conjunto de propriedades. Por exemplo, a verificação de modelos pode verificar se um programa satisfaz uma série de propriedades temporais de segurança, ou seja, se um programa desempenha certas operações de segurança numa certa ordem. As propriedades de segurança podem ser descritas por um autómato finito (FSA – *Finite State Automaton*).

Este tipo de análise enumera de forma sistemática possíveis estados de um sistema, explorando eventos não determinísticos deste. Começando por um estado inicial, a verificação de modelos gera recursivamente sucessivos estados do sistema. Um modelo é uma especificação abstracta do sistema, que acaba por ser uma descrição do código, abstraindo muitos detalhes da implementação.

Para um dado estado do sistema, a verificação de modelos procura sistematicamente erros no comportamento do sistema. Esta técnica é usualmente utilizada para provar que um sistema satisfaz uma ou mais propriedades. Esta exploração de estados de sistema é um bom método para detectar erros não muito comuns.

Uma ferramenta que implementa este tipo de verificação é a *MOPS* [20][41][53], que verifica se um dado sistema viola propriedades de segurança. Por exemplo, pode verificar se um programa com *setuid-root* não elimina os privilégios de *root* antes de executar um programa não confiável. Estas propriedades de segurança são modeladas por autómatos finitos (FSA) e fornecidos ao *MOPS*.

Uma outra ferramenta de verificação de modelos é a *UNO* [54]. O seu nome provém de três defeitos de *software* para a detecção dos quais *UNO* foi concebida, nomeadamente a utilização de variáveis não inicializadas, deferenciamento de *nil-pointers* e índices de *arrays* fora dos limites dos *arrays*. Esta ferramenta cria árvores de *parsing* que constituirão grafos de fluxo de controlo, os quais serão analisados por verificação de modelos para descobrir erros.

2.5 Detecção de Vulnerabilidades de Inteiros

Antes de terminar este capítulo de trabalho relacionado, é de salientar que, recentemente, foram efectuados avanços no estudo das vulnerabilidades de inteiros, pela construção da ferramenta *RICH* (*Run-time Integer CHecking*) [13], para detecção deste tipo de vulnerabilidades.

Esta ferramenta é uma extensão do compilador *gcc* que compila o código fonte, cria o código objecto e monitoriza a execução deste último, de forma a detectar vulnerabilidades de inteiros. Desta forma, um ataque a tipos de dados inteiros é detectado por esta ferramenta.

A metodologia da *RICH* está assente na teoria de sub-tipos de dados inteiros, a qual expressa relacções entre sub-tipos de dados, como por exemplo *int8_t* é um sub-tipo de *int16_t*, sendo o relacionamento representado por *int8_t <: int16_t* e significando que os valores do sub-tipo *int8_t* $\subseteq$ *int16_t*. Esta metodologia é uma aproximação às linguagens de programação seguras (como Ada), que permitem o utilizador criar sub-tipos que qualificam tipos de dados primitivos.

Com base nesta teoria, esta ferramenta adiciona novos relacionamentos de sub-tipos de dados inteiros, bem como transforma os existentes, por foma a obter uma linguagem de programação segura para sub-tipos de dados inteiros. Estas adições/modificações serão a extensão do compilador *gcc*. Assim, quando o compilador está na fase de criação do código objecto, *RICH* analisa a representação intermédia do código e instrumenta o código objecto com verificações para todas as operações de inteiros inseguras, para que na execução do programa as mesmas sejam executadas. Quando uma destas verificações detecta um bug de inteiro é gerado uma mensagem de aviso.

Ao invés deste tipo de análise, em *run-time*, a ferramenta apresentada nesta tese baseia-se na análise estática de código, onde uma das ferramentas de análise que a sustenta - *Lint* - permite a detecção de vulnerabilidades de inteiros, nomeadamente *integer overflow* e *underflow* [22].

Capítulo 3

Vulnerabilidades na Portabilidade

Os números inteiros, desde de há alguns anos para cá, têm vindo a preocupar os técnicos e analistas de segurança de *software*, pelo facto de terem sido crescentes e subestimadas as origens das vulnerabilidades existentes em programas desenvolvidos nas linguagens de programação C e C++.

A raiz de tais vulnerabilidades reside na ausência de condições de verificação dos limites máximo e mínimo que os números inteiros podem representar. Estas ausências tornam o *software* vulnerável, pois, quando exploradas, originam números inteiros inesperados que serão utilizados como índices de um vector (*array*), tamanho de um *buffer*, ou contador de um ciclo.

Com este capítulo pretende-se apurar e apresentar as vulnerabilidades de inteiros que ocorrem na portabilidade de código de 32 para 64 bits. Contudo e para uma melhor entendimento das mesmas, serão apresentados, numa primeira instância, os diferentes tipos de *bugs* de inteiros, com exemplos de código ilustrativos. Neste ponto, dar-se-á grande relevância ao *bug* de truncamento, o qual resulta, regra geral, numa vulnerabilidade de *buffer overflow*. Numa segunda parte, apresentar-se-ão as categorias de ataques que os *bugs* de inteiros podem despoletar quando explorados.

3.1 Tipos de *Bugs* de Inteiros

Para que exista uma vulnerabilidade os valores que a exploram têm de provir do exterior do programa, nomeadamente do teclado, *sockets*/rede, ficheiros ou linha de comandos [62]. Um *bug* (valor inesperado/incorrecto resultante de uma operação) de um programa estará na origem de vulnerabilidade, se resultar de valores provenientes do exterior do programa e se for utilizado como parâmetro de funções susceptíveis de exploração. No entanto, poderá acontecer que uma vulnerabilidade exista num programa, mas não seja explorada, permanecendo no programa sem originar qualquer anormalia do seu funcionamento.

As operações sobre números inteiros podem originar valores inesperados, regra geral, devido a conversões entre tipos de dados inteiros e/ou atribuição de um número maior ou menor a um tipo de dados que não o comporta. Assim, um programa que contenha código susceptível de originar valores inesperados (pela não verificação dos

limites máximo e mínimo possíveis para as variáveis), e que durante a sua execução receba valores do exterior, por qualquer *interface* ou dispositivo de *input*, está vulnerável a ataques que podem explorar as vulnerabilidades originadas pelos valores inesperados e incorrectos (*bugs* de inteiros). São eles: *integer overflow*, *integer underflow*, *signedness* ou *sign error* e truncamento de dados.

Cada um destes bugs, por si só, não causa vulnerabilidade, mas se o código do programa apresentar uma ou mais destas vulnerabilidades e manipular a memória física do computador, então a exploração destas vulnerabilidades poderá resultar em *buffer overflow* ou na paragem da aplicação, ou seja, em problemas de segurança de *software*.

Por conseguinte, antes de caracterizarmos, nas sub-secções seguintes, as vulnerabilidades de inteiros supracitadas, quando executadas numa plataforma de 32 bits, será pertinente definir sucintamente *buffer overflow*, uma vez que será o ataque mais referido na exploração dessas vulnerabilidades e patente nos excertos de código exemplificadores das mesmas. Assim sendo, entende-se por *buffer overflow* o acto de extravasão do espaço definido para um *buffer* pela escrita de uma quantidade de *bytes* superior ao seu tamanho. Este extravasar da memória pode modificar valores de variáveis, modificar o fluxo de execução, ou parar o programa.

3.1.1 *Integer Overflow*

Um *integer overflow* ocorre quando o resultado de uma operação algébrica inteira ultrapassa o limite máximo permitido pelo tipo. Assim sendo, as operações que podem resultar em *integer overflow* são a adição, a subtracção, a multiplicação e a divisão [13][14][27][55][56].

Como qualquer tipo de dados inteiro pode ser *signed* ou *unsigned*, o *integer overflow* será do tipo *signed* ou *unsigned*. Estamos na presença do primeiro, quando o valor do resultado da expressão sobrepõe o *bit* representativo do sinal e, estamos na presença do segundo, quando a representação de um tipo de dados inteiro não suporta o valor do resultado da expressão.

Por outras palavras, quando o valor máximo permitido é excedido [14]:

- *signed* (Figura 3.1), o valor resultante é a adição do valor mínimo permitido do tipo do dados com a quantidade de unidades excedidas, excepto uma.

```
1  char c;
2  c = SCHAR_MAX;  /* SCHAR_MAX = 127 */
3  c = c + 3;
4  printf("c = %d", c);  /* c = -126 */
```

Figura 3.1: ***Integer overflow: signed overflow***

Ao observarmos o código constante na figura anterior, verifica-se, na linha 3, que será adicionado 3 unidades a 127, perfazendo 130. Mas como o valor 130 excede em 3 unidades o valor máximo permitido para o tipo de dados *char*, então o número inteiro resultante será o valor mínimo permitido do tipo de dados *char* (-128) mais a quantidade de unidades excedidas (3), diminuída em uma unidade, o que perfaz -126 (-128 + 3 – 1 = -126).

- *unsigned* (Figura 3.2), segundo a norma ANSI C99 [28], que contém as regras de manipulação de inteiros da linguagem de programação C:

> *"A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo to the number that is one greater than the largest value that can be represented by the resulting type."*

Ou seja, um valor inteiro *unsigned*, quando excede o valor máximo permitido pelo tipo da variável que o receberá, é convertido num outro número inteiro *unsigned*, que não é mais do que o resto da divisão inteira do número em causa pelo maior número inteiro possível do referido tipo de dados inteiro, acrescido de uma unidade.

```
1  unsigned char c;
2  c = UCHAR_MAX;  /* UCHAR_MAX = 255 */
3  c = c + 3;
4  printf("c = %u", c);  /* c = 2 */
```

Figura 3.2: ***Integer overflow: unsigned overflow***

Na linha 3 do excerto de código acima, podemos observar que o valor de *c* irá ser excedido em 3 unidades. No entanto, como estamos na presença de um número

inteiro *unsigned*, o número inteiro resultante será 2, o qual é apurado pelo 258 MOD 256, onde 258 é o número inteiro excedido e 256 é o maior valor possível do tipo de dados *unsigned char*, acrescido de uma unidade.

3.1.1.1 Adição

A operação aritmética adição é, geralmente, utilizada para adicionar dois operandos aritméticos ou um número inteiro e um ponteiro, uma vez que o valor deste último é interpretado como um *unsigned int*. Contudo, o resultado da adição de um inteiro com um ponteiro é um ponteiro [14].

A adição de inteiros resulta num *integer overflow*, se o valor resultante não pode ser representado pelo tipo de dados inteiro da variável que o irá receber.

Os exemplos seguintes ilustram este tipo de vulnerabilidade, mas com ocorrência de *buffer overflow*.

Exemplo 1: *Adição com input signed e buffer overflow*

Como já foi descrito, na adição ocorre *integer overflow* quando o valor máximo positivo do tipo de dados da variável receptora da operação é ultrapassado.

```
1  int DoSomething(const char* server)
2  {
3     unsigned char namelen = strlen(server) + 2;
4     if(namelen < 255)
5     {
6        char* UncName = malloc(namelen);
7        if(UncName != 0)
8           sprintf(UncName, "\\\\%s", server);
9
10       //do more things here
11    }
12    return 0;
13 }
```

Figura 3.3: *Integer overflow:* adição com *input signed* e *buffer overflow* [58]

Na Figura 3.3 quando o comprimento da variável *server* (variável de *input*) for igual a 254 ou a 255 *bytes* e, na linha 3, adicionarmos duas unidades, ocorrerá *integer overflow unsigned* (Figura 3.2) ao armazenarmos na variável *namelen* o valor 256 ou 257, respectivamente.

O valor final de *namelen* será de 0 ou 1, consoante o comprimento da variável *server* seja 254 ou 255, respectivamente.

O teste lógico (linha 4) será verdadeiro, a função *malloc()* retornará um ponteiro de memória válido para a variável *UncName*, reservando-lhe 0 ou 1 *byte*, e a função *sprintf()* escreverá os 254 ou 255 bytes contidos na variável *server*, concatenados com os caracteres "\\" (mais dois bytes), na variável *UncName*, originando, assim, um *buffer overflow*.

Exemplo 2: *Adição com input unsigned e buffer overflow*

Uma das causas que estão na origem de *integer overflows* é a utilização de variáveis do tipo *signed* para representar tamanhos de *strings*, *buffers* ou *array*, as quais deveriam ser do tipo *unsigned*.

No fragmento de código da Figura 3.4, podemos observar que as variáveis de *input len1* e *len2* representam os tamanhos das *strings s1* e *s2* e são do tipo *size_t* (*unsigned int*), indo, assim, ao encontro à declaração correcta de variáveis representativas de tamanhos.

Suponhamos então que os valores para *len1* e *len2* são 64 bytes e 4 GB (valor máximo permitido para o tipo *unsigned size_t*), respectivamente.

```
1  int func(char *s1, size_t len1, char *s2, size_t len2)
2  {
3     if (1 + len1 + len2 > 64)
4        return -1;
5
6     char *buf = (char*)malloc(len1+len2+1);
7     if (buf)
8     {
9        strncpy(buf,s1,len1);
10       strncat(buf,s2,len2);
11    }
12    //do more things here
13    return 0;
14 }
```

Figura 3.4: ***Integer overflow:*** **adição com** ***input unsigned*** **e** ***buffer overflow*** (adaptado de [59])

Na linha 3, como a expressão matemática do teste lógico contém o operando *signed 1*, é necessário que os valores das variáveis *len1* e *len2* sejam primeiramente convertidos para o tipo *signed*, onde assumirão os valores de 64 e -1, respectivamente, verificando-se um *integer overflow signed* (Figura 3.1) na conversão de tipo de *len2*. O resultado da expressão matemática será 64, onde o teste lógico será falso e será reservado para a variável *buf* espaço em memória de 64 bytes (linha 6).

Na linha 9, será copiado o conteúdo da variável *s1* (64 bytes) para *buf*, ocupando-o na totalidade. Quando o conteúdo da variável *s2* for concatenado ao da variável *buf* (linha 10), ocorrerá um *buffer overflow* e consequentemente *crash* do sistema, uma vez que o espaço em memória para o *buf* será extravasado em 4 GB (tamanho de *s2*).

3.1.1.2 Multiplicação

A multiplicação de números inteiros pode gerar um *integer oveflow*, quando multiplicamos operandos de um tipo de um dado, sendo este resultado superior ao valor máximo permitido pelo tipo de dados receptor [14].

Exemplo 3: *Multiplicação com buffer overflow*

No código da Figura 3.5, suponhamos que os valores das variáveis de *input StructSize* e *Count* são 250 e 265, respectivamente.

```
1  int AllocateStructs(void** ppMem, unsigned short StructSize,
                       unsigned short Count)
2  {
3     unsigned short bytes_req;
4     bytes_req = StructSize * Count;
5     *ppMem = malloc(bytes_req);
6
7     //do more things here, like write to *ppMem
8
9     if(*ppMem == NULL)
10       return -1;
11    else
12       return 0;
13 }
```

Figura 3.5: *Integer overflow:* operação de multiplicação e *buffer overflow* [57]

Na linha 4, o resultado da multiplicação é 66250, significando que é necessário aquele valor de *bytes* para armazenar 265 registos, cada qual de 250 *bytes* de tamanho.

No entanto, quando este valor é armazenado na variável *unsigned bytes_req*, dar-se-á um *integer overflow unsigned* (Figura 3.2), resultando então o valor 714 (66250 MOD 65536). Será assim reservado em memória física 714 *bytes* (linha 5), para a variável **ppMem*.

Por fim, originará um *buffer overflow* quando ao tentar-se escrever os 265 registos pretendidos em **ppMem*.

3.1.1.3 Divisão

Na divisão de inteiros não era muito óbvia a ocorrência de *integer overflow*, ou melhor, não se esperava que o resultado da divisão de dois números inteiros pudesse ultrapassar o valor máximo permitido de um dado tipo de dados inteiro. Por outras palavras, que o quociente da divisão inteira fosse maior do que o dividendo[14].

Exemplo 4: *Divisão de dois signed*

Observando o código da Figura 3.6, quando um número inteiro negativo (*signed*), de valor igual ao mínimo permitido pelo tipo de dados inteiro, é dividido pelo valor -1 (linha 3), por forma a convertê-lo num número inteiro positivo, estamos a originar um *integer overflow*, uma vez que o número resultante excede o maior número positivo do tipo de dados da variável receptora. Assim sendo, o resultado é obtido pela regra de excesso para números inteiros *signed* ilustrada na Figura 3.1.

```
1  char c;
2  c = SCHAR_MIN;  /* SCHAR_MIN = -128 */
3  c = c/-1;
4  printf("c = %d", c);  /* c = -128 */
```

Figura 3.6: *Integer overflow:* operação de divisão de dois *signed*

Exemplo 5: *Divisão de um unsigned por um signed*

No fragmento de código da Figura 3.7, podemos observar que, quando um número inteiro *unsigned*, de valor superior ao máximo permitido para o tipo *signed* do mesmo tipo de dados inteiro, é dividido pelo valor -1 (linha 4), estamos a originar um *integer overflow*. Isto acontece porque para a execução da expressão matemática de tipos *signed* e *unsigned*, primeiramente o valor *unsigned* (130) é convertido para *signed*, originando então o *integer overflow*, sendo o resultado da conversão (-126) obtido pela regra de excesso para números inteiros *signed* ilustrada na Figura 3.1. O valor final da expressão será 126 (-126/-1), o qual será atribuído a *c* sem quaisquer alterações, pelo facto deste pertencer ao intervalo de valores positivos do tipo *char*.

Um facto curioso e que dará a ilusão ao utilizador de que o valor final da variável *c* está correcto é quando assumimos para a variável *uc* 128, resultando para *c* -128. Ao analisarmos cuidadosamente a execução do código, verificámos que *uc* primeiro é

convertido para *signed*, originando um *integer overflow* e assumindo o valor de -128. Quando a expressão matemática é executada, teremos como resultado 128 (-128/-1).

No entanto, quando atribuímos o 128 à variável *signed c*, ocorre novamente um *integer overflow*, pelo facto de 128 ser superior ao valor máximo positivo permitido para *c*, resultando, assim, um valor final de -128. Podemos então dizer que ocorreram dois *integer overflows* e que por coincidência o resultado final está correcto.

```
1  unsigned char uc;
2  char c;
3  uc = 130;
4  c = uc/-1;
5  printf("c = %d", c);  /* c = 126 */
```

Figura 3.7: *Integer overflow:* operação de divisão de um *unsigned* por um *signed*

Exemplo 6: *Divisão de um signed por um unsigned*

Quando dividimos um número negativo por um número positivo grande, o resultado esperado é um valor muito próximo de zero.

Observando o exemplo de código da Figura 3.8, um número inteiro negativo (*signed*), de valor igual a -1, é dividido pelo valor máximo do tipo de dados *unsigned int* (4 GB). Como a expressão envolve os tipos *signed* e *unsigned*, então o valor *signed* (-1) é convertido para *unsigned int*, ou seja, o valor -1 é convertido para 4 GB. O resultado da divisão será então 1 (4 GB / 4 GB).

```
1  unsigned int uc = UINT_MAX;
2  int c;
3  c = -1/uc;
4  printf("c = %d", c);  /* c = 1 */
```

Figura 3.8: *Integer overflow:* operação de divisão de um *signed* por um *unsigned*

Em suma, podemos afirmar que a vulnerabilidade *integer overflow*, resultante das operações aritméticas adição (ou subtracção), multiplicação e divisão, gera resultados menores do que os correctos esperados, que se associados ao manuseamento de memória física do computador origina *buffer overflow*. Neste sentido, um atacante conhecedor da exploração destas vulnerabilidades (*integer* e *buffer overflows*) poderá beneficiar das mesmas e conseguir, por exemplo, executar código seu (injectado na ocorrência do *buffer overflow*) na máquina alvo, para conseguir "roubar" informações da mesma.

3.1.2 *Integer Underflow*

Ao invés do *integer overflow*, o *integer underflow* ocorre quando o resultado de uma expressão matemática ultrapassa o menor valor permitido pelo tipo inteiro da variável que irá receber o resultado.

Esta categoria de vulnerabilidade está associada à subtracção, onde ambos os operandos devem ser do mesmo tipo aritmético ou serem ponteiros compatíveis. Na subtracção, é possível subtrair um número inteiro de um ponteiro [13][14][27][55][56].

A vulnerabilidade *integer underflow* é semelhante quer para *unsigned* ou *signed*. Assim, independentemente do sinal do tipo de dados inteiro da variável que receberá o resultado da subtracção, o valor resultante do *integer underflow* será igual à subtracção do valor máximo positivo permitido pelo tipo de dados inteiro menos o número de unidades excedidas, acrescido de uma unidade, como o visualizado no código seguinte, nas Figura 3.9 e Figura 3.10:

- *signed*

```
1  char c;
2  c = SCHAR_MIN;  /* SCHAR_MAX = -128 */
3  c = c - 3;
4  printf("c = %d", c);  /* c = 125 */
```

Figura 3.9: ***Integer underflow: signed underflow***

Podemos verificar, no exemplo de código acima, que ao valor mínimo permitido do tipo de dados *char*, armazenado na variável *c* (-128), é subtraído 3 unidades, resultando -131. O mesmo será atribuído a *c*, que ultrapassa o valor mínimo permitido para o seu tipo de dados. Logo, o valor resultante da expressão da linha 3 será o valor máximo permitido do tipo de dados *char* (127) menos a quantidade de unidades excedidas (3), acrescida em uma unidade, o que perfaz 125 (127 - 3 + 1 = 125).

- *unsigned*

No código abaixo, a variável *c* assume o valor mínimo possível do tipo de dados *unsigned char*. Quando é-lhe subtraído 3 unidades (linha 3), o valor resultante (-3) quando atribuído à variável *c* origina um *integer underflow*, pelo facto de -3 ser menor do que o valor mínimo permitido pelo tipo de dados *unsigned char*. Assim, o valor que será atribuído a *c* será o valor máximo permitido do tipo de dados

char (255) menos a quantidade de unidades excedidas (3), acrescida em uma unidade, o que perfaz 253 (255 - 3 + 1 = 253).

```
1  unsigned char c;
2  c = 0;
3  c = c - 3;
4  printf("c = %d", c);  /* c = 253 */
```

Figura 3.10: ***Integer underflow: unsigned underflow***

Seguidamente, apresentamos alguns exemplos de código com manuseamento de memória principal, onde se poderá visualizar que a ocorrência de *integer underflow*, na execução de subtracções, originará valores inteiros muito superiores aos correctos esperados. Os mesmos se utilizados como parâmetros de funções de manuseamento de memória resultará em *buffer overflow*, ou melhor, em "estouro" da memória RAM, pela quantidade de memória a reservar ser superior à existente no computador.

Exemplo 1: *Alocação de memória com buffer overflow*

No código da Figura 3.11, suponhamos que o valor da variável de *input unsigned cbAllocSize* é 0.

```
1  void AllocMemory(size_t cbAllocSize)
2  {
3     cbAllocSize--;
4     char *szData = malloc(cbAllocSize);
5
6     //do more things here
7  }
```

Figura 3.11: ***Integer underflow:*** **alocação de memória e** ***denial of service*** [57]

Na linha 3, o resultado da subtracção é -1, que quando armazenado na variável *unsigned cbAllocSize*, dar-se-á um *integer underflow*, resultando então o valor 4 GB (4 BG -1 + 1). Será assim reservado em memória física 4 GB (linha 4), para a variável **szData*. Por fim, originará negação de serviço *(DoS)*, ou melhor, o "estouro" da memória, caso o computador não possua 4 GB de memória RAM, originando o *crash* do sistema.

Exemplo 2: *Alocação de memória com validação de input e buffer overflow*

No exemplo de código da Figura 3.12, observamos que o valor da variável de *input unsigned cbSize* é validado superiormente (linha 3), mas não inferiormente.

```
1  int func(size_t cbSize)
2  {
3     if (cbSize < 1024)
4     {
5        char *buf = malloc(cbSize-1);
6
7        //do more things here
8        return 0;
9     {
10    else
11       return -1;
12 }
```

Figura 3.12: ***Integer underflow:*** **alocação de memória com validação de** ***input e buffer overflow*** (adaptado de [57])

Suponhamos, então, que o valor de *unsigned cbSize* é igual a 0.

Na linha 3, a condição de validação é verdadeira, sendo então, na linha 5, reservado espaço em memória para a variável *buf*.

A quantidade a reservar para a variável *buf* é calculada pela subtracção de *cbSize – 1*, onde resulta -1, que originará um *integer underflow*, resultando então o valor 4 GB (4 GB -1 + 1). Será assim reservado em memória física 4 GB pela função *malloc()*, para a variável **buf*. Assim sendo, ocorrerá um *buffer overflow*, com as consequências idênticas às apontadas no exemplo anterior.

Exemplo 3: *Browser Netscape com buffer overflow*

O excerto de código da Figura 3.13 faz parte do *browser Netscape*, versões 3.0-4.73, onde está patente uma vulnerabilidade *integer underflow* e ocorrência de *buffer overflow*.

Um atacante que deseje explorar esta vulnerabilidade poderá realizá-lo com dois valores possíveis para a variável de *input unsigned len*, nomeadamente:

- *variável de input unsigned len = 0*

  Na linha 4, o resultado da subtracção é -2, que quando armazenado na variável *unsigned size*, dar-se-á um *integer underflow*, resultando então o valor de 4.294.967.294. Será assim reservado em memória física *size + 1 bytes*, ou seja, 4 GB (linha 5), para a variável **comm*.

  A quantidade de memória a alocar para a variável **comm* originará uma negação de serviço (*DoS*), caso a máquina não possua tal quantidade de memória principal.

- *variável de input unsigned len = 1*

Na linha 4, o resultado da subtracção é -1, que quando armazenado na variável *unsigned size*, dar-se-á um *integer underflow*, resultando então o valor de 4 GB.
Na linha 5, quando for executado a expressão *size + 1* verificar-se-á um *integer overflow unsigned*, pelo facto de adicionarmos uma unidade ao valor máximo permitido para o tipo de dados inteiro *unsigned int*. Assim sendo, a quantidade de memória a ser reservada para a variável **comm* será de 0 bytes.
Na linha 6, quando a função *memcpy()* tentar escrever o conteúdo da variável *src*, de 4 GB (*size*), se irá verificar um *buffer overflow* por duas razões, nomeadamente tentar escrever num espaço de 0 bytes e tentar escrever 4 GB em memória física.

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

Figura 3.13: ***Integer underflow: browser Netscape* e *buffer overflow*** [13]

Neste exemplo, é visível que um atacante conseguirá o mesmo resultado por dois métodos diferentes, ou seja, conseguirá ultrapassar o tamanho máximo da memória principal do computador pela tentativa de armazenamento de uma grande quantidade de memória e pela tentativa de escrita em memória de uma grande quantidade de informação, onde será somente reservado 0 bytes de memória.

3.1.3 *Signedness*

A vulnerabilidade *signedness*, também conhecida por *sign error*, ou, ainda, por *signed versus unsigned integers*, está relacionada com a conversão entre valores de tipos de dados inteiros *signed* e *unsigned*. Neste sentido, a explicação de como são apurados os valores resultantes da conversão está assente na conversão implícita ou explícita entre tipos, apresentada nas secções 2.1.2.2 e 2.1.2.3 [13][14][27][55][56].

A ocorrência desta vulnerabilidade acontece quando um:

- *signed* inteiro é interpretado como um *unsigned* inteiro (Figura 3.14), ou seja, quando um número inteiro *signed* é convertido num número inteiro *unsigned*.

```
1  int i = -3; /* i = 0xFFFFFFFD */
2  unsigned short u;
3  u = i;
4  printf("u = %hu\n", u); /* u = 0xFFFD, u = 65533 */
```

Figura 3.14: *Signedness:* conversão de *signed* para *unsigned* [14]

Ao observarmos o código acima, identificamos uma conversão do tipo *signed* para *unsigned* (linha 3). Ao atribuirmos o valor da variável *i* (-3) à variável *u*, e, de acordo com as regras de conversão de tipos, os dezasseis bits mais à direita da variável *i* são atribuídos à variável *u*, resultando um número positivo grande.

- *unsigned* inteiro é interpretado como um *signed* (Figura 3.15), ou seja, quando um número inteiro *unsigned* é convertido num número inteiro *signed*.

```
1  unsigned char u = 253;  /* u = 0xFD */
2  char c;
3  c = u;
4  printf("c = %d\n", c); /* c = -3 */
```

Figura 3.15: *Signedness:* conversão de *unsigned* para *signed*

No código acima, na linha 3, é realizada uma conversão do tipo *unsigned* para *signed*, atribuirmos o valor da variável *u* (253) à variável *c*. De acordo com as regras de conversão de tipos entre variáveis do mesmo tamanho, mantém-se a máscara de bits (*bit pattern*), ficando o bit mais significativo com a função de representação do sinal. O número resultante será, então, um valor negativo (-3), uma vez que o bit mais significativo da variável *u* tem o valor de 1.

Em seguida, apresentamos alguns exemplos de código com manuseamento de memória principal, onde se poderá visualizar que a ocorrência de *signedness*, em operações de comparação e/ou conversões implícitas da linguagem de programação, originará valores incorrectos, que se utilizados como parâmetros de funções de manuseamento de memória poderá resultar em *buffer overflow*.

Exemplo 1: *Operação de comparação*

No código da Figura 3.16, suponhamos que o valor da variável de *input signed len* é negativo, por exemplo -1.

Na linha 4, é efectuada a comparação entre um número *signed* (*len*) com um *unsigned* (*sizeof(kbuf)*, do tipo *size_t*), ou seja, a comparação entre tipos diferentes.

```
1  int copy_something(char *buf, int len)
2  {
3     char kbuf[800];
4     if (len > (int)sizeof(kbuf))
5        return -1;
6
7     return memcpy(kbuf, buf, len);
8  }
```

Figura 3.16: ***Signedness:*** **operação de comparação** [55]

No entanto, se considerarmos que na comparação o número inteiro *unsigned* será convertido explicitamente para *signed*, então a condição será falsa.

Como o limite inferior aceitável para a variável *len* não é verificado, logo ela poderá ter valor negativo e passará no teste de validação da linha 4, o qual só testa o limite superior para a referida variável.

Na linha 7, como a função *memcpy()* impõe que o seu terceiro parâmetro seja do tipo *unsigned*, então o valor da variável *signed len* é convertido para *unsigned*, originando *signedness* e resultando um grande valor positivo (4 GB), por *len* ter valor negativo. Quando a função *memcpy()* for executada dar-se-á um *buffer overflow*, pelo facto de extravasar os 800 bytes afectos à variável *kbuf*.

Exemplo 2: *operação aritmética*

No código da Figura 3.17, verifica-se, novamente, que o tipo *signed* é opção errada para representar o tamanho das *strings s1* e *s2*. Suponhamos então que o valor das variáveis de *input signed len1* e *len* é 127 e -2, respectivamente.

Na linha 4, a operação do teste de validação tem resultado 126, onde todos os operandos envolvidos são do tipo *signed*. O teste de validação é então verdadeiro.

Na linha 9, são copiados os 127 bytes contidos na variável *s1* para o *array buf*.

Na linha 10, a função *strncat()* concatena o conteúdo da variável s2 ao do *array buf*, onde o número de bytes a concatenar é especificado por *len2*. Como *len2* é do tipo *signed* e negativo e na função de concatenação o parâmetro que especifica o número

de bytes a concatenar é do tipo *unsigned*, então *len2* é convertido para o tipo *unsigned*, originando *signedness* e resultando um valor aproximado de 4GB. Assim, quando a função é executada ocorrerá um *buffer overflow*, uma vez que o índice máximo do *array buf* será ultrapassado.

```
1  int func(char *s1, int len1, char *s2, int len2)
2  {
3     char buf[128];
4     if (1 + len1 + len2 > 128)
5        return -1;
6
7     if (buf)
8     {
9        strncpy(buf,s1,len1);
10       strncat(buf,s2,len2);
11    }
12    return 0;
13 }
```

Figura 3.17: *Signedness:* operação aritmética [59]

Por fim, podemos salientar, novamente, que, para variáveis que representarão tamanhos de *buffers*, *strings* e índices de *arrays*, o tipo das mesmas deverá ser *unsigned*, por forma evitar *buffers overflows*, aquando da escrita em memória, uma vez que nas funções de manuseamento de memória os parâmetros especificadores de quantidades são do tipo *unsigned*.

3.1.4 Truncamento

O *bug* de truncamento de dados ocorre quando um inteiro é convertido para um inteiro de menor tamanho e o valor do inteiro original está fora do intervalo permitido para o tipo de dados de menor tamanho. Por outras palavras, o truncamento de dados acontece quando atribuímos um inteiro de maior largura a um inteiro de menor largura, sendo o seu valor superior ao máximo permitido pelo o inteiro de menor largura.

O truncamento é realizado pela preservação dos bits mais à direita, perdendo assim os bits mais significativos, ou seja, perda de dados (Figura 3.18), e consequentemente num valor final menor do que o esperado [13][14][27][55][56].

```
1 unsigned short u = 30000;  /* u = 0x7530 */
2 short s = 300;  /* s = 0x12C */
3 char c;
4
5 c = u;
6 printf("c = %d\n", c); /* c = 0x30 = 48 */
7
8 c = s;
9 printf("c = %d\n", c); /* c = 2C = 44 */
```

Figura 3.18: Vulnerabilidade truncamento de dados [14]

No excerto de código anterior, podemos observar que as variáveis *u* e *s*, de comprimento maior (16 bits), são atribuídas à variável *c*, de comprimento menor (8 bits). Em qualquer uma das atribuições (linhas 5 e 8) ocorre truncamento, onde o valor armazenado em *c* são os oito bits mais à direita dos números inteiros *u* e *s*.

Como exemplo de truncamento com *buffer overflow* segue-se o código da Figura 3.19. Se observarmos cuidadosamente o referido código, identificamos na linha 3 a origem do truncamento de dados quando se atribui a um *short* um *int*, ou seja, a atribuição de 32 bits a 16 bits. Suponhamos então que o valor da variável de *input cbBuf* é 65568 (0x00010020), que quando atribuído à variável *cbCalculatedBufSize* é truncado para 32 (0x0020).

Na linha 4, são alocados 32 bytes de memória para a variável *buf*. Por fim, na linha 7, ocorrerá *buffer overflow* quando a quantidade de bytes, especificada por *cbBuf*, é copiada da variável *name* para a variável *buf*, ou seja, a cópia de 65568 bytes para um espaço de 32 bytes.

```
1  int func(char *name, int cbBuf)
2  {
3     unsigned short cbCalculatedBufSize = cbBuf;
4     char *buf = (char*)malloc(cbCalculatedBufSize);
5     if (buf)
6     {
7        memcpy(buf, name, cbBuf);
8        //do more things here
9        return 0;
10    }
11    return -1;
12 }
```

Figura 3.19: Vulnerabilidade truncamento de dados com *buffer overflow* (adaptado de [59])

Podemos dizer que, relativamente à vulnerabilidade truncamento de dados, o valor resultante é, regra geral, menor do que o original e que se utilizado em funções de alocação de memória física irá reservar menor espaço do que o previsto, o que incorrerá

em *buffer overflow*, quando a quantidade de bytes esperados for escrita no espaço de memória reservado.

Após a caracterização e apresentação detalhada dos tipos de *bugs* de inteiros, podemos afirmar que todos eles assentam num só problema: "exceder o limite superior ou inferior do tipo de dados inteiro". Por tal, julgamos ser correcto apelidar todas as vulnerabilidades por *integer overflow*.

A consequência principal do *integer overflow* é a de não ser detectado após a sua ocorrência, fazendo com que não haja forma do programa emitir uma mensagem avisando se um resultado calculado previamente está correcto. Esta consequência pode ser perigosa se os referidos cálculos estão ligados a tamanhos de *buffers*, a quantidade de *bytes* a escrever em *buffers*, a índices de *arrays*, a contadores de ciclos (*counter loop*), a verificação de limites superior e/ou inferior, ou a referência de objectos (ponteiros) que resultarão em sucesso de ataques, tais como o *buffer overflow*.

3.2 Categorias de Ataques

Após a caracterização das vulnerabilidades, esta secção tem por objectivo apresentar os ataques que poderão explorar uma ou mais vulnerabilidades de inteiros.

Alguns destes ataques fazem parte integrante dos excertos de código ilustrativos das vulnerabilidades de inteiros, nomeadamente o *buffer overflow* e a *negação de serviço*. Por isso, as sub-secções destinadas a estes tipos de ataques não serão tão detalhadas como a dos restantes, uma vez que os seus detalhes estão ilustrados na secção anterior.

3.2.1 *Buffer Overflow*

Um *buffer overflow* ocorre quando o tamanho definido para um tampão de memória (*buffer*), declarado na *stack* (pilha) ou na *heap* da memória física do computador, é sobrescrito pela cópia de uma quantidade de dados maior do que o tamanho do *buffer* [13].

Este tipo de vulnerabilidade pode surgir em linguagens de programação tais como C e C++, que não verificam os limites dos *buffers* de escrita *versus* a quantidade de bytes a escrever neles.

Quando os *buffers* são declarados na *stack*, ou seja, quando é reservado espaço em memória para variáveis locais, tais como os *arrays*, a sua exploração origina o *buffer overflow*, também conhecido por *stack overflow/smashing*. Por seu turno, quando o *buffer* pertence à *heap*, ou seja, quando trabalhámos com memória dinâmica, a sua exploração é também denominada por *heap overflow/smashing* [57][60][61].

Para uma melhor compreensão de como o *buffer overflow* é explorado, de forma a se obter, por exemplo, acessos indevidos ao sistema, é necessário, em primeiro lugar, compreender como a memória física está organizada.

Na figura Figura 3.20 observamos que a área de *Texto* armazena o código executável do programa (instruções do programa). A área de *Dados* está afecta a todas as variáveis globais e estáticas. Enquanto que a área de *Heap* é reservada para alocação local e dinâmica de memória, a área de *Stack Frame* é utilizada para guardar valores de registos, o endereço de retorno de funções/sub rotinas, criar variáveis locais, bem como para passar parâmetros na chamada de funções.

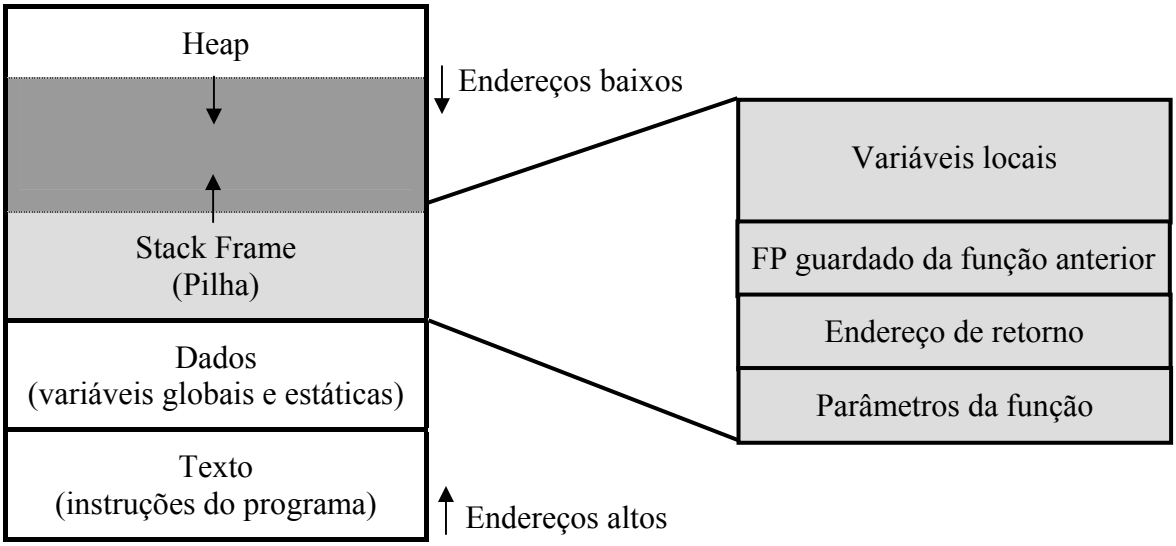
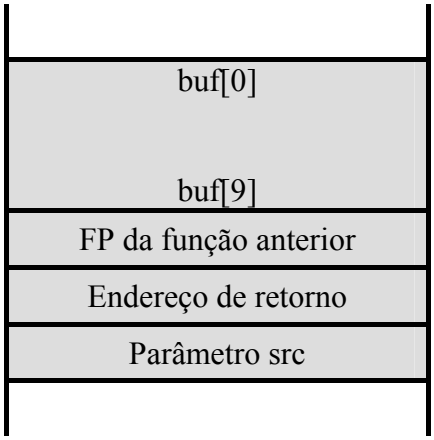


Figura 3.20: Organização da memória física do computador [60]

Também pode ser observado na figura anterior que os ponteiros da pilha e da *heap* crescem em sentidos opostos, convergindo para o centro da área livre, que é comum às duas estruturas de memória. Esse artifício é utilizado para optimizar o uso da memória livre na área de dados do processo.

Na execução normal de um programa, onde se reserva e se escreve para a pilha, podemos verificar, pela Figura 3.21, que o tamanho da pilha cresce realmente no sentido da *heap*, quanto maior for o tamanho do *buffer* a reservar em memória. Podemos também verificar que o endereço de retorno irá conter o endereço da sub rotina *cp()*, após a execução da mesma.




```
void cp (char *src)
{
        char buf[10];
        strcpy(buf, src);
}
```

Figura 3.21: Criação e escrita no *buffer*

Um atacante que deseje fazer um *buffer overflow* terá de alterar o valor do endereço de retorno e redireccioná-lo para o seu código malicioso, que está sobrescrito no *buffer*. A partir desse momento, o ponteiro de instruções do processo passa a ser inteiramente controlado pelo atacante, que poderá fazer qualquer chamada a funções disponíveis no sistema.

A alteração do endereço de retorno pode ser efectuada quer pelo "estouro" de uma variável local (*buffer*) armazenada na pilha, como se pode observar na Figura 3.22, quer pelo "estouro" da área de *heap*. De qualquer forma, o código malicioso, para onde o programa será desviado, pode ser colocado tanto no *heap* como na pilha.

Ao simularmos uma execução do código da figura anterior, sabendo que a função *strcpy()* não verifica o tamanho do *input* na função com os limites do *buffer* destino, verificamos que, quando o conteúdo da variável *src* a ser copiado para a variável *buf[]* contiver mais do que 10 caracteres, estaremos na presença de um ataque *buffer ovrflow*, e

consequentemente o estouro da pilha. Deste modo, o endereço de retorno para a função *cp()* vai ser modificado, o qual irá apontar para outro local, uma vez que o referido endereço irá ser sobrescrito.

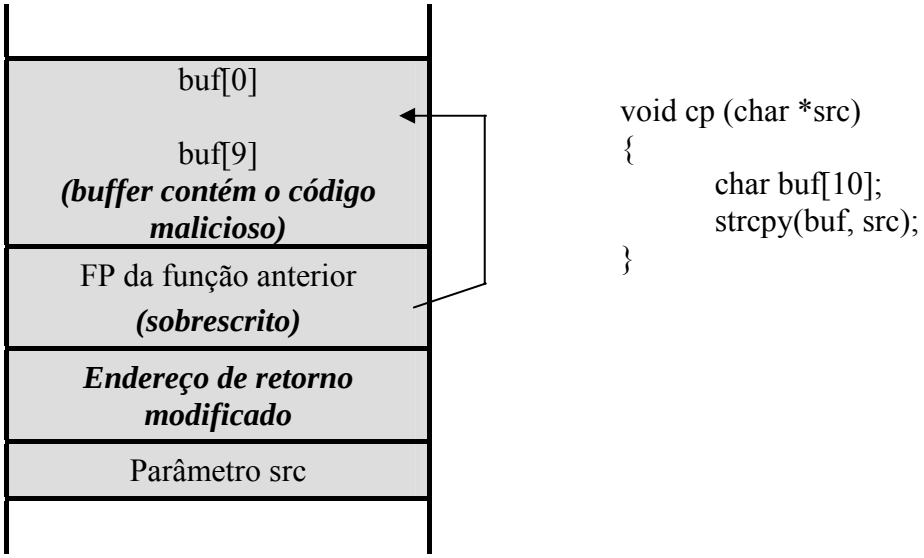


Figura 3.22: Ataque de *buffer overflow* à *stack*

Num ataque mais elaborado, o código injectado na variável *buf[]* irá despoletar o *buffer overflow*, reescrevendo o endereço de retorno e fazendo-o apontar para o código injectado (malicioso), por forma a que o código malicioso seja executado após a execução da função *cp*().

Será correcto afirmar que um atacante, salvo excepções, explora as vulnerabilidades de inteiros para conseguir o *overflow* de um *buffer*. Para tal, o atacante as despoleta pelo envio de um valor que, regra geral, é igual a um dos limites superior ou inferior do tipo de dados inteiro. O valor resultante do *bug* de inteiro e/ou o valor enviado pelo atacante são utilizados em funções de manuseamento de memória que originarão o *buffer overflow*.

Em termos de conclusão e tendo por base as vulnerabilidades de inteiros estudadas na secção anterior, um ataque de *buffer overflow* ocorre quando é explorada a vulnerabilidade:

- *integer overflow unsigned* e truncamento de dados, onde o valor resultante será menor do que o esperado e parâmetro da função *malloc()*;
- *integer overflow signed*, *integer underflow* e *signedness* de variáveis com valores negativos, onde o valor resultante será maior do que o esperado e parâmetro de funções de escrita em *buffers*, tais como *memcpy()*, *strncpy()*. No entanto, o valor

resultante da ocorrência da vulnerabilidade terá de ser menor do que a quantidade de memória RAM do computador.

3.2.2 Negação de Serviço *(DoS)*

O ataque de negação de serviço (*DoS – Denial of Service*), no presente estudo, passa por injectar no programa uma quantidade de dados que o deixe inoperacional, não respondendo aos processos e pedidos de acesso a recursos e/ou sistema [13].

Ao longo da caracterização das vulnerabilidades de inteiros foram apresentados alguns exemplos de código que resultam em *DoS*, onde constava manipulação de funções de memória, que, aplicados os valores resultantes das vulnerabilidades, resultavam no esgotamento da memória física do computador, ou seja, sistema inoperacional – *DoS* (Figura 3.11, Figura 3.13, Figura 3.16)

Assim, podemos resumir que o resultado da manipulação de funções de memória origina *negação de serviço*, quando os parâmetros destas funções são os valores negativos resultantes dos *bugs integer overflow signed*, *integer underflow* e *signedness*. Estes valores negativos resultantes são maiores do que os esperados e "estourarão" a memória do computador. As situações onde poderá ocorrer o preenchimento total da memória RAM são:

- pela função *malloc()*, reservar um espaço de memória demasiado grande, como por exemplo 4 GB;
- pelas funções de escrita em memória, tais como *memcpy()*, *strncpy()*, *strncat()* *sprintf()*, *strcpy()*, onde a quantidade de bytes a escrever no *buffer* é extremamente grande, originando *buffer overflow* e simultaneamente negação de serviço.

Para além das funções de memória, o *DoS* também ocorre com sucesso se forem conseguidos ciclos infinitos (Figura 3.23), pela manipulação dos seus contadores, uma vez que o sistema fica ocupado com o ciclo e não responde a pedidos.

```
1  int i = 1;
2  for (i != 0)
3  {
4     //do more things here
5     i = i - 2;
6  }
```

Figura 3.23: *Denial of Service:* ciclo infinito

Na figura anterior, após diversas iterações e quando o valor da variável *i* estiver muito próxima do menor valor possível do tipo *int*, ocorrerá *integer underflow*, onde o valor resultante será um número positivo grande. Verifica-se um ciclo infinito porque a variável *i* é inicializada com um valor impar, que decrementado de duas unidades nunca resultará em zero. Outro aspecto que se pode concluir, é o facto de que a variável *i* poderia ser do tipo *unsigned*, que continuaríamos na presença da vulnerabilidade *integer underflow* e ciclo infinito, se o valor de *i* inicial for impar.

3.2.3 Índice de *Array*

Um outro tipo de ataque, explorado indirectamente por vulnerabilidade de inteiros, é a atribuição a variáveis com índices de *arrays* valores fora do intervalo de índices possíveis para os mesmos [13]. Por outras palavras, consiste em atribuir a variáveis de índices de *arrays* valores negativos ou valores positivos superiores ao índice máximo permitido para o *array*, como ilustra a Figura 3.24.

```
1  int *table = NULL;
2  int insert_in_table(unsigned int posic, int value)
3  {
4     if (!table)
5        table = (int *)malloc(sizeof(int) * 100);
6     int pos = posic;
7     if (pos > 99)
8        return -1;
9     table[pos] = value;
10    return 0;
11 }
```

Figura 3.24: Índice negativo de *array* [14]

O excerto de código acima pretende escrever um determinado valor (*value*) na posição (*pos*) do *array table*. Suponhamos então que o valor da variável de *input posic* tem o bit mais significativo igual a 1. Para o *array table* é reservado espaço em memória (linha 5) e à variável *signed pos* é atribuído o valor da variável *unsigned int posic* (linha 6), ocorrendo a vulnerabilidade *signedness* e ficando a variável *pos* com um valor negativo. O valor do índice (*pos*) é menor do que 99, logo o teste de validação de índice do *array* é verdadeiro (linha 7) e o conteúdo da variável *value* é escrito no *array table*, pela linha de código (linha 9)

```
table[pos] = value;
```

que é equivalente a

```
*(table + (pos * sizeof(int))) = value;
```

ou seja, à posição de memória inicial da variável *table* será adicionada o valor da posição do *array* onde desejamos escrever, multiplicada pelo tamanho do tipo de dados inteiro (4 bytes). Assim, o valor da variável *value* será escrito na posição de memória resultante da expressão anterior. No entanto, como o valor da variável *pos* é negativo, logo a posição de memória resultante da expressão estará antes do inicio do *buffer* afecto à variável *table*, fazendo com que o valor da variável *value* seja escrito fora do *buffer*, ou seja, originando um *array indexing error* (ocorre quando o valor do índice do *array* não pertence ao intervalo de valores definidos para o *array*).

Para além do exemplo acima (ocorrência de *array indexing error*), podemos ter a mesma situação, quando o valor da variável do índice do *array* é o resultado de um:

- *integer overflow signed*, que resultará num valor negativo;
- *integer underflow unsigned*, que resultará num valor positivo superior ao máximo índice do *array*;
- *signedness*, que poderá resultar num valor negativo, consoante o valor a ser convertido e os tipos de dados envolvidos na conversão;
- truncamento de dados, que poderá resultar num valor negativo.

3.2.4 Contornar Verificação de Limites

Como já foi possível observar, na secção 3.1, em diversos exemplos de código consta a não verificação dos limites possíveis para determinada variável, antes de ser efectuada qualquer operação ou execução de instrução. Verifica-se também que um atacante conhecedor de tais não validações, pode explorar tais falhas que originarão outros tipos de ataques.

Uma outra situação que pode acontecer é a verificação somente de um dos limites da variável, ficando o valor da mesma vulnerável à não verificação do outro limite. Um exemplo patente desta situação é o da Figura 3.24, onde é verificado e validado o limite superior da variável *pos*, mas a verificação do limite inferior é descurada, aceitando, assim, valores negativos para índice do *array* [13].

Como exemplos da não verificação dos limites superior e/ou inferior, temos passagem de valor:

- negativo, onde o limite inferior de determinada variável não é validado. Podemos então utilizar o valor negativo como índice de *array*. Também pode ser utilizado na quantidade de espaço a reservar em memória ou quantidade de bytes a escrever em memória, resultando em ambos os casos *signedness* e consequentemente *buffer overflow* ou *DoS*;
- positivo, onde o limite superior não é validado. Podemos utilizar esse valor em todos casos indicados no ponto acima.

3.2.5 Erros Lógicos

Os erros lógicos acontecem, quando uma vulnerabilidade de inteiro permite a um atacante manipular um ou mais objectos do programa, resultando em erros do mesmo, ou seja, quando não está previsto um valor inesperado, mesmo após terem sido validados os valores pretendidos [13][14].

O objectivo da Figura 3.25 é escrever a quantidade de bytes *len*, da variável *str*, no *array buf[]*, onde os dois primeiros bytes de *str* indicam o tamanho da *string* a escrever. No código a variável *len* é do tipo *signed*, a qual pode tomar valores positivos ou negativos. Na linha 4, é extraído da variável *str* o tamanho da mesma, o qual é armazenado na variável *len*, e, na linha 5, o apontador de *str* avança dois bytes no seu interior, para que passe a apontar para o início da *string* a ser escrita no *buf[]*. Para que não ocorra *buffer oveflow*, ou seja, para evitar a escrita para além do número de bytes reservados para o *array buf[]*, a linha 6 verifica se o número de bytes pretendidos na escrita não ultrapassa o tamanho de *buf[]*.

No entanto, como a variável *len* é do tipo *signed*, o valor extraído na linha 4 pode ser negativo e o teste de verificação do limite superior de *buf[]* é verdadeiro. Quando, na linha 9, é executada a função *memcpy()*, o valor de *len* é convertido para o tipo *unsigned*, ocorrendo desta forma *signedness* e resultando um valor maior do que o tamanho de *buf[]*. Ocorrerá então *buffer overflow*. Como o valor em questão (variável *len*) consegue passar num teste de validação, não é esperável que o mesmo valor origine uma vulnerabilidade, o que a tal se denomina de Erro Lógico.

```
 1  char processCopy(char *str)
 2  {
 3     char buf[512];
 4     short len = *(short *)str;
 5     str += sizeof(len);
 6     if (len > 512)
 7        return -1;
 8     else
 9        memcpy(buf, str, len);
10     return 0;
11  }
```

Figura 3.25: Erro Lógico

Para finalizar as secções de vulnerabilidades de inteiros e ataques, um aspecto a ter em conta e que ainda não foi abordado é que as categorias de vulnerabilidades de inteiros podem estar associadas ao manuseamento de ponteiros, pelo facto do ponteiro ser interpretado como um *unsigned int* que contem um endereço de memória. Assim, na aritmética de ponteiros ocorre os mesmos problemas que na aritmética de números inteiros, que quando explorados ocorrerão em *integer overflow*, por exemplo.

Por forma, então, a contrariar o sucesso do atacante, em qualquer uma das categorias de ataques, ao declarar e manusear as variáveis do tipo inteiro devemos:

- verificar com testes de validação os limites superior e inferior das variáveis;
- utilizar o tipo de dados *unsigned int* para tamanhos de *buffers*, índices de *arrays* e contadores de ciclos (*loop counters*), os quais nunca terão valores negativos, e verificar os seus limites superior e inferior;
- evitar a mistura de variáveis de tipos *signed* e *unsigned* em expressões matemáticas e comparações;
- ter em atenção a utilização do tipo de dados *size_t*, uma vez que o seu comprimento depende da plataforma, sendo igual ao comprimento do endereço de memória. Assim, na plataforma de 32 bits, *size_t* tem um comprimento de 32 bits, mas numa plataforma de 64 bits terá um comprimento de 64 bits;
- verificar se os valores de *input* estão dentro do intervalo de valores predefinido;
- verificar se o código do programa contém operações com números inteiros (adição, multiplicação, entre outras), onde o resultado será utilizado como índice de um *array* ou tamanho de um *buffer* e garantir que os mesmos estão dentro dos limites pretendidos;

- ter cuidado em utilizar variáveis do tipo *signed* em funções de manipulação de memória física (*malloc()*, *memcpy()*, entre outras), onde serão tratadas como *unsigned*.

3.3 Vulnerabilidades na Adaptação para LP64

O objectivo principal desta tese é o apuramento da existência de vulnerabilidades de inteiros em aplicações que são portadas da arquitectura de 32 bits (ILP32) para a arquitectura de 64 bits, modelo de dados LP64.

Na secção 2.3, foram apresentados e/ou detectados os problemas ocorridos na portabilidade de código entre os modelos de dados supracitados, que se não resolvidos por alterações de código incorrem em problemas de funcionamento correcto da aplicação e/ou sistema.

Nesta secção, pretende-se, então, detectar a ocorrência das referidas vulnerabilidades em aplicações que serão portadas para 64 bits, sem quaisquer alterações ao seu código, mas que quando executadas em plataforma de 32 bits podem ou não ocorrer. Por tal, esta secção encontra-se dividida pelos tipos de vulnerabilidades, cada qual com um exemplo ilustrativo. A primeira, *integer overflow* e *underflow*, diz respeito às possíveis situações de portabilidade nas operações aritméticas que ocorrem em vulnerabilidade.

3.3.1 *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.

Para além das operações aritméticas implicadas nas vulnerabilidades *integer overflow* e *underflow*, serão apresentadas as operações de *bit shifts* e de ponteiros, as quais decorrem, também, em vulnerabilidades quando portadas para 64 bits.

3.3.1.1 Operações Aritméticas

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.

No que concerne à operação aritmética da divisão, a 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 *integer underflow*, associada à subtracção, esta ocorre:

- **`int--`**, em ambos 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;

3.3.1.2 *Bit Shifts*

Em *bit shifts* há ocorrência da vulnerabilidade *integer overflow* quando [10]:

- deslocamos, ao valor 1, do tipo *int*, o seu comprimento em número de bits. O resultado é zero, porque houve um excesso de um bit, ocorrendo *integer overflow*. Em ambos os modelos de dados constata-se esta vulnerabilidade, pelo tipo de dados *int* ter o mesmo comprimento;
- deslocamos, ao valor 1, do tipo *long*, 32 bits (Figura 3.26), sendo o resultado atribuído a uma variável do tipo *long*. Como em ILP32 o comprimento do tipo de dados *long* é igual ao do *int*, então estamos perante a situação anterior, e consequentemente na ocorrência de *integer overflow*. Mas se for analisado segundo LP64, não há ocorrência da referida vulnerabilidade, porque o deslocamento dos 32 bits é efectuado numa variável de comprimento de 64 bits.

```
1  unsignd long y;
2  y = 1L << 32;
```

Figura 3.26: Portabilidade: vulnerabilidade *integer overflow* em *bit shifts*

3.3.1.3 Ponteiros

Como é sabido, o conteúdo de um ponteiro (endereço de memória) é interpretado como um *unsigned*, do tipo *size_t*, cujo tamanho depende da plataforma. Também já foi referido que os problemas decorridos nas operações aritméticas de números inteiros são os mesmos na aritmética de ponteiros. Por isso, para além do truncamento de dados entre um

ponteiro e um *int*, na adaptação de código de ILP32 para LP64, temos os mesmos de aritmética de ponteiros já existentes, como se pode observar na Figura 3.27.

```
1  unsigned long ui = 0xfffffffbUL;  /* ULONG_MAX - 4, em 32 bits */
2  long *p;
3  p = ui;
4  p = p + 1;
```

Figura 3.27: Portabilidade: vulnerabilidade *integer overflow* em ponteiros

Na aritmética de ponteiros, o incremento de uma unidade a um ponteiro equivale à adição do valor do ponteiro (*unsigned int*) com o tamanho (*sizeof()*) do tipo de dados que o ponteiro representa.

O excerto de código acima, o ponteiro *p* representa um *long* e é inicializado com o valor máximo permitido do tipo de dados *unsigned long*, decrementado de quatro unidades. Se executarmos em ILP32 estas linhas de código, o valor resultante do ponteiro *p*, após a execução da linha 4, é 0xFFFFFFFF, ou seja, a adição do valor *p* com 4 unidades, que representam o *sizeof(long)*. Não há ocorrência de *integer overflow*, porque o valor final de *p* é igual ao limite superior do tipo de dados *unsigned long*.

Ao executarmos o mesmo código sob o modelo LP64, o valor final de *p* será 3, pelo facto do *sizeof(long)* neste modelo de dados ser igual a 8, que adicionado a (*ULONG_MAX – 4*) ultrapassa o limite superior permitido do tipo de dados *unsigned long*, resultando, assim, um *integer overflow unsigned* (Figura 3.2) e valor final de 3 para *p*.

3.3.2 *Signedness*

Os problemas ocorridos na 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*.

No entanto, alguns casos que em ILP32 não ocorrem em *signedness*, como a comparação de *long* com *unsigned int* ou a comparação de *int* com *unsigned long*, concretizam-se no modelo LP64, por estarmos a comparar variáveis de tamanhos diferentes. Tais factos levam a uma promoção de tipo de *unsigned int* ou *int* para *long* ou *unsigned long*, respectivamente, ou seja, promoção do tipo de dados de menor tamanho para o de maior tamanho.

Quando analisado o código da Figura 3.28, segundo o modelo ILP32, verificámos que, ainda que uma das variáveis (*len1*) representativas de tamanho de *string* seja do tipo *signed*, está garantido a não ocorrência de *integer overflow unsigned* na execução da linha 6. Perante duas variáveis de tipos *unsigned* e *signed* e sabendo que a variável *len1* terá de ser convertida em *unsigned*, para que seja realizada a adição entre as duas variáveis (*len1* e *len2*), as linhas 4 e 5 garantem, após a subtracção entre o limite superior do tipo *unsigned int* e o valor de *len1* (convertido para *unsigned int*), o espaço em memória para *len2*. Como o resultado da linha 6 pode ser negativo, pelo mesmo ser armazenado numa variável do tipo *signed*, a linha 9 garante que o total de espaço a alocar para o *buffer* seja positivo.

```
1  int func(char *s1, int len1, char *s2, unsigned int len2)
2  {
3     long totalSize;
4     unsigned int aux = UINT_MAX - (unsigned int)len1;
5     if (aux >= len2)
6        totalSize = len1 + len2;
7     else
8        return -1;
9     if (totalSize < 0)
10       return -1;
11
12    char *buf = (char *)malloc(totalSize);
13    strncpy(buf, s1, len1);
14    strncat(buf, s2, len2);
15    return 0;
16 }
```

Figura 3.28: Portabilidade: vulnerabilidade *signedness* com *buffer overflow*

Por exemplo, para valores -2 e 1 para as variáveis *len1* e *len2*, respectivamente, teremos:

```
aux = 4.294.967.295 - 4.294.967.294 = 1
teste lógico da linha 5 é verdadeiro (1 >= 1)
totalSize = -1 (4.294.967.294 + 1 = 4.294.967.295)
teste lógico da linha 9 é verdadeiro (-1 < 0)
```

Tendo como exemplo os valores de *len1* e *len2* apresentados para ILP32, quando adaptamos o código da figura acima para LP64, a garantia de que não ocorre *integer overflow unsigned* mantém-se, no entanto, quando atribuímos o resultado *unsigned int* da adição à variável *signed long totalSize* o mesmo é convertido de 32 para 64 bits, onde o valor da variável *totalSize*, que deveria ser negativo, é um inteiro positivo. Desta forma o teste lógico da linha 9 é falso, logo será reservado 4 GB para a variável **buf*, onde poder-se-á dar o "estouro" da memória (*DoS*).

Mesmo considerando que a máquina possui mais do que 4 GB de memória RAM, podendo-se alocar os 4GB de espaço para a variável **buf*, quando a linha 13 for executada o valor de *len1* (-2) será convertido para *size_t*, resultando um valor de quase 2^{64} GB, dando-se então *buffer overflow*.

Também é de salientar que a portabilidade para LP64 se agrava, quando uma variável do tipo *signed* de valor negativo (por exemplo -1) é utilizada como parâmetro de tamanho de *buffers* em funções de manuseamento de memória física. Enquanto que no modelo ILP32 o preenchimento total da memória era com 4 GB (valor *unsigned int* (*size_t*) equivalente a -1), no modelo LP64 o mesmo é efectuado com (2^{64} - 1) GB, pelo facto da variável *signed* ser convertida para o tipo de dado *size_t*, o qual em LP64 tem tamanho de 64 bits, equivalente a um *unsigned long*.

3.3.3 Truncamento

O truncamento de dados acontece quando se atribui uma variável de maior comprimento a uma variável de menor comprimento [10][11]. Quando tal acontece, estamos perante a vulnerabilidade de inteiros Truncamento, onde o seu valor resultante quase sempre está associado a funções de manuseamento de memória, fazendo com que ocorra, à *posteriori*, *buffer overflow*.

Na adaptação de 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:

- 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 shifts*), efectuado sobre um *long*, a um *int*;
- atribuição, pela função *scanf()*, de um *long* a uma variável utilizando o formato "%d";
- escrita num *buffer*, pela função *sprintf()*, de um *long* utilizando o formato "%d".

No código da Figura 3.29, podemos verificar que, se o mesmo for executado numa plataforma de 32 bits, não há qualquer vulnerabilidade. A variável representativa do tamanho de **buf* é do tipo *unsigned* e não há qualquer alteração de valor na atribuição de um *unsigned long* a um *unsigned int*, por ambos terem o mesmo comprimento. No entanto, o mesmo código se executado segundo o modelo LP64, está vulnerável à exploração da vulnerabilidade de truncamento de dados e consequentemente à ocorrência de *buffer overflow*.

```
1  int copia(char *src, unsigned long len)
2  {
3     unsigned int size = len;
4     char *buf = (char*)malloc(size);
5     if (buf)
6     {
7        memcpy(buf, src, len);
8        return 0;
9     }
10    return -1;
11 }
```

Figura 3.29: Portabilidade: vulnerabilidade truncamento de dados com buffer *overflow*

Tal vulnerabilidade é explorada quando o valor da variável de *input len* for superior ao valor máximo permitido para o tipo de dados *unsigned int*. Assim sendo, o valor da variável *len* será truncado (linha 3), o resultado do truncamento será utilizado na função de manuseamento de memória *malloc()* (linha 4), e, na linha 7, ocorrerá *buffer overflow*, quando for escrito o número de bytes *len* num espaço de *size* bytes.

Com esta análise de verificação de vulnerabilidade *integer overflow* na adaptação de código de 32 para 64 bits, podemos dizer que, se as alterações ao código de 32 bits não forem efectuadas, com os necessários cuidados, aumenta o número de vulnerabilidades deste tipo. Também é correcto afirmar que a categoria truncamento de dados é a mais visível na portabilidade, pelos factores já conhecidos e mencionados ao longo do estudo, bem como a comparação entre tipos de dados *long* e *int*, que originam graves problemas de segurança de *software*, resultando em "estouro" de memória.

3.4 Vulnerabilidades na Adaptação para ILP64 e LLP64

No que respeita às vulnerabilidades ocorridas na adaptação de código do modelo ILP32 para os modelos de dados ILP64 e LLP64, à luz das vulnerabilidades identificadas na adaptação para LP64, temos que:

- **Modelo ILP64:**

  A adaptação de código para este modelo de dados, sem quaisquer alterações ao mesmo, não origina novas vulnerabilidades, ou seja, um programa ausente de vulnerabilidades em ILP32 quando adaptado para ILP64 não origina nenhuma vulnerabilidade.

  Por outro lado, um programa vulnerável em ILP32 quando portado para ILP64 mantém as mesmas vulnerabilidades, mas com limites máximos de tipos de dados inteiros expandidos para 64 bits, significando que os valores inteiros que explorarão as vulnerabilidades são de 64 bits.

- **Modelo LLP64:**

  Neste modelo de dados ocorrem vulnerabilidades de adaptação de código semelhantes às do modelo LP64, pelo facto do ponteiro expandir-se para 64 bits, enquanto que os tipos de dados *int* e *long* manterem os seus tamanhos de 32 bits. Assim aplicações mal adaptadas poderão originar vulnerabilidades:

- *Integer overflow e underflow*:
  - as mesmas vulnerabilidades de ILP32 que envolvem tipos de dados *int* e *long*, uma vez que neste modelo estes tipos de dados não alteram o seu tamanho;
  - as mesmas vulnerabilidades oriundas de ponteiros, mas tendo em conta que o tamanho do mesmo é de 64 bits.

- *Signedness*:
  - mantem-se as mesmas vulnerabilidades ocorridas em ILP32 que envolvam tipos de dados *int* e *long*;
  - atribuição de um *int*, *long* ou constante de 32 bits a um ponteiro;
  - comparação de um *int* ou *long* com um ponteiro.

- Truncamento de dados:
  - atribuição de um ponteiro a um *int* ou a um *long*;
  - atribuição do valor retornado por uma função, sob ponteiro, a um *int* ou a um *long*.

Capítulo 4

Concepção da Ferramenta *DEEEP*

O principal ojectivo desta tese é contribuir para o desenvolvimento e implementação de segurança de *software*. Para sua concretização, concebeu-se a ferramenta *DEEEP* para que possa ser utilizada como auxílio no desenvolvimento de *software*.

A referida ferramenta efectua análise estática de código, com recurso a duas outras ferramentas de análise estática. Assim, para que *DEEEP* pudesse ser desenvolvida, foi necessário avaliar ferramentas de análise estática de código que realizassem análise semântica ao código fonte. Também foi necessário seleccionar linguagens de *script* adequadas para filtragem e unificação de resultados, bem como para desenvolver a ferramenta propriamente dita.

Neste sentido, este capítulo, numa primeira instância, apresenta a selecção das ferramentas de análise estática de código e das linguagens de *script* utilizadas, para posteriormente apresentar a arquitectura e as fases de processamento da ferramenta *DEEEP*. Por fim, dá-se lugar à apresentação e explicação da *interface* da ferramenta.

4.1 Ferramentas de Análise Estática de Código

Partindo de todo o estudo efectuado até então, foi necessário seleccionar ferramentas de análise estática de código, de análise semântica, nomeadamente verificação de tipos e análise de fluxo de dados.

Sendo o objectivo a detecção de vulnerabilidades de inteiros, aquando da má adaptação de código de 32 para 64 bits, a ferramenta *DEEEP* teria de localizar *bugs* de 64 bits, ou melhor, localizar as linhas de código que necessitariam de reajustes de código e rastrear as variáveis que armazenariam dados provenientes do exterior (teclado, *sockets*/rede, ficheiros ou linhas de comando).

Para tal, foi necessário avaliar ferramentas de análise estática de código, de código aberto, que desempenhassem a verificação de tipos, para localização dos bugs de 64 bits, e a análise de fluxo de dados, para rastrear as variáveis de entrada de dados.

De entre algumas ferramentas de análise estática de código avaliadas para o efeito, nomeadamente *BOON* [46], *Uno* [54], *Eau Claire* [43], *CQual* [50], *Splint* [23] e *Lint* [22], as seleccionadas para a análise de:

- *verificação de tipos*: foram *Lint* e *Splint*, por ambas detectarem a maioria dos problemas de portabilidade para LP64.

A escolha da ferramenta *Lint* é justificada por esta conter um parâmetro que permite detectar os *bugs* de 64 bits, excepto os de *signedness* [22]. 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* [23].

De salientar que a ferramenta *Lint* seleccionada é a da *Sun Microsystems*, por esta conter o parâmetro dirigido para a adaptação de código para LP64. Tal facto levou a que a plataforma de trabalho da ferramenta *DEEEP* seja o *Open Solaris* (plataforma de código aberto, proveniente do *Solaris*).

De referir também que a ferramenta *Splint* faz análise em código fonte construído segundo a norma ANSI C89 (ISO 9899:1989). Isso significa que esta ferramenta resulta em erro, abortando a sua execução, se a declaração de variáveis não for efectuada no início do programa/funções.

- *de fluxo de dados*: de entre as ferramentas *CQual* e *Splint*, apostou-se na segunda por já ser a da análise anterior e por se querer aprofundar a funcionalidade de *extensible cheking* por ela oferecida. A aplicabilidade da ferramenta *CQual*, até então conhecida, é para detecção de vulnerabilidades de formatação de strings, enquanto que a aplicabilidade do *Splint* nos pareceu mais abrangente.

4.2 Linguagens de *Script*

Para uma maior facilidade de desenvolvimento da ferramenta *DEEEP*, optou-se por linguagens de *script* que pudessem filtrar resultados, manusear mais do que um ficheiro e interagir com as variáveis de ambiente do sistema operativo. Assim, foram seleccionadas e estudadas as linguagens

- *bash* [63]: linguagem do *shell* para desenvolver o programa principal da aplicação *DEEEP*. Por ser de execução rápida e de facilmente se poder interagir com as variáveis de ambiente do sistema, redefinindo os seus valores ou definindo novas.

  Desta forma, toda a sintaxe, menu, *outputs* e caminhos de execução da ferramenta são definidos nesta linguagem de programação.

- *awk (ou gawk)* [64]: para efectuar processamento de dados baseados em texto. Linguagem de programação escolhida para, com base num ficheiro de texto, extrair linhas do mesmo que contivessem expressões regulares (*regular patterns*).

  Com base numa pequena base de dados de texto, de registos as mensagens de aviso dos bugs de 64 bits, filtra os resultados obtidos na análise de verificação de tipos, efectuada pelo *Lint* e *Splint*. Também faz a filtragem à análise de fluxo de dados, efectuada pelo *Splint*. Desta forma, consegue-se obter somente as linhas de texto necessárias ao estudo.

  Também foi utilizada em pequenos filtros, necessários para o correcto funcionamento do programa principal da ferramenta *DEEEP*.

- *Perl* [65]: para efectuar processamento de dados de vários ficheiros e intervir com as variáveis de ambientes passadas pelo programa principal da ferramenta.

  Também permite a pesquisa de texto por expressões regulares, herdadas do *awk*.

  Linguagem de programação escolhida para manipular vários ficheiros, previamente resultantes da linguagem *awk*, por forma a uni-los. Também utilizada para envolver as linhas de execução do ficheiro *Makefile*, bem como para fazer a correlação dos resultados dos dois tipos de análises estática de código e apresentar as possíveis vulnerabilidades encontradas.

4.3 Arquitectura da Ferramenta

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* (de código aberto), tendo por base as ferramentas *Lint* e *Splint*, 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 correlaciona automaticamente 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 automático 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.

Na arquitectura da ferramenta *DEEEP*, ilustrada na Figura 4.1, 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* de 64 bits. Como essas ferramentas detectam muitos outros tipos de *bugs,* os resultados têm de ser filtrados pelo filtro representado na figura.

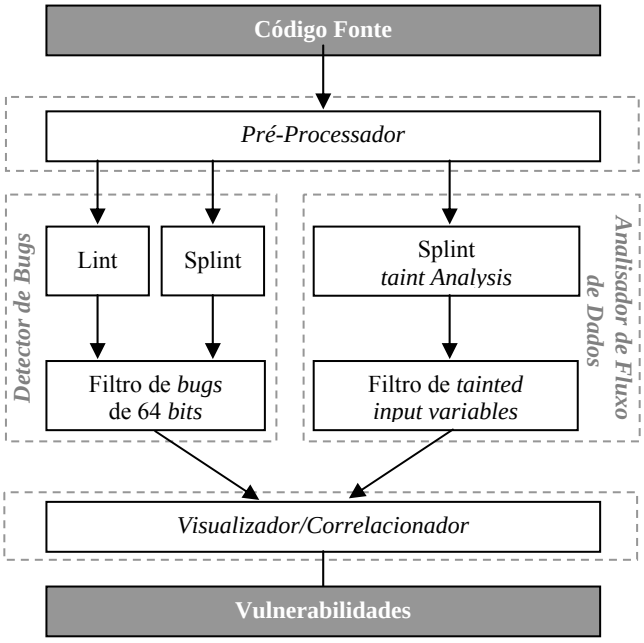


Figura 4.1: Arquitectura da ferramenta *DEEEP*

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 estudados na secção 3.2 como resultado das vulnerabilidades listadas na secção 3.1. 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.

4.4 Fases de Processamento

Esta secção destina-se a apresentar as diversas fases de processamento efectuadas pela ferramenta *DEEEP*, nomeadamente Pré-Processador, Detector de Bugs, Analisador de Fluxo de Dados e Visualizador/ Correlacionador.

4.4.1 Pré-Processador

Para que as ferramentas *Lint* e *Splint* possam funcionar e analisar correctamente o 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* (*/usr/include*) [22][23].

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*, pela linha de comando

```
make CC="lintsplintgcc gcc"
```

fazendo com que o ficheiro *Makefile* interprete como compilador de C o programa de *Perl* linsplintgcc e o *gcc* (compilador efectivo). Assim, para cada linha de comando dirigida ao compilador será executado em primeiro lugar o programa *lintsplintgcc* com os parâmetros inclusos na linha de comando e posteriormente a sua execução pelo *gcc*.

Desta forma com o programa *lintsplintgcc* a envolver a execução do *Makefile* é possível capturar as linhas de comando para o compilador, extrair as directivas necessárias e criar as linhas de comando para as referidas ferramentas de análise estática de código.

4.4.2 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 referida detecção é efectuada pelas ferramentas *Lint* e *Splint* parametrizadas com os resultados do pré-processador e com as suas próprias *flags* de detecção de bugs de 64 bits.

As duas ferramentas devolvem muitas outras mensagens de aviso, para além das relacionadas com os *bugs* que interessam a este estudo (ver secções 2.3 e 3.1). Por essa razão, os resultados têm de ser filtrados, de modo a obter-se somente as linhas de código e os avisos relativos aos *bugs* do estudo. 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*. Os filtros aqui mencionados são *scripts* construídos em linguagem de programação *awk*, que verificam, para cada mensagem emitida pelas ferramentas *Lint* e *Splint*, se coincide com algum dos registos da base de dados.

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 arquitectura da ferramenta (ver Figura 4.1).

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.

4.4.3 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, fornece mecanismos para realizar *taint analysis* [44], pela funcionalidade de *extensible checking* [23], conseguindo assim realizar análise de fluxo de dados.

As verificações que o utilizador pretende definir podem ser descritas por condições sobre atributos associados a objectos do programa ou ao estado global de execução. No entanto, ao contrário dos tipos (dos qualificadores de tipos utilizados pela ferramenta *CQual*), 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*, ilustrados na Figura 4.2. 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.

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*.

```
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) *inputness*

```
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*) inputness1*

Figura 4.2: Definição dos atributos

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.

Após a definição dos atributos, foi necessário construir um ficheiro com todas as funções da biblioteca *glibc* relevantes para o estudo, o qual fica associado aos atributos e lhes indica quais as funções que são objecto de análise. Este ficheiro contém todas as declarações das funções, cujos parâmetros foram documentados com as anotações *inputtainted* e *inputuntainted*.

Assim, os valores retornados por todas as funções de *input* foram anotados com a anotação *inputtainted*, indicando que o valor contido na variável retornada poderá ser malicioso (estar comprometido). Contudo, há a salientar que o *Splint* apresenta limitações quanto à anotação de certos argumentos de funções. Deste modo, nem todos os argumentos de retorno, de funções de *input*, puderam ser anotados, nomeadamente os *varargs* das funções *fscanf*, *fwscanf*, *scanf*, *wscanf*, *sscanf* e *swscanf*, levando a que as variáveis por elas retornadas não possam ser rastreadas.

Também foram anotados com a anotação *inputuntainted* todos os argumentos inteiros de entrada das funções de reserva de memória e de escrita em memória, ficheiros e *sockets*, obrigando a que esses argumentos sejam *untainted,* ou seja, que 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 título de exemplo, a Figura 4.3 a) ilustra 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 comprometido por uma vulnerabilidade de inteiro. Na função *getchar*, o estado do tipo *int* retornado é *tainted*.

Nas funções de manuseamento de memória, apresentadas na Figura 4.3 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) Funções de *input*

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

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

b) Funções de manuseamento de memória

Figura 4.3: Exemplos de anotação de funções

A ideia do funcionamento dos atributos *inputness* e *inputness1* é a seguinte: numa primeira análise ao código (primeira passagem), todos os objectos não anotados têm estado *nostate* (cláusula *defaults* do atributo *inputness* – Figura 4.2 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 (passagem de variáveis *tainted* em chamada de *funções perigosas* (*tainted as untainted*) e de procedimentos criados pelo programador (*tainted as nostate*)); finda a sinalização das variáveis de *input* e a sua passagem em *funções perigosas* e em procedimentos não anotados, a análise recomeça (segunda passagem), tendo todos os objectos não anotados o estado *tainted* (cláusula *defaults* do atributo *inputness1* – Figura 4.2 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 (em *funções perigosas*). Também assegura que se estes valores forem passados por referência em procedimentos cujos argumentos sejam *nostate*, ou seja, em procedimentos criados pelo programador 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* (locais de *funções perigosas*) emitindo uma mensagem, e permitindo detectar vulnerabilidades nestes procedimentos.

Nesta fase, à semelhança da anterior, o *Splint* dá muitas outras mensagens, para além das necessárias para o estudo. Assim, novamente foi necessário criar *scripts* em linguagem de programação *awk* para filtrar as mensagens necessárias, ou seja, as da *taint analysis*.

4.4.4 Visualizador/Correlacionador

O objectivo da quarta componente da ferramenta *DEEEP* é correlacionar os resultados das duas componentes explicadas anteriormente (ver Figura 4.1). 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* (no atributo *inputness*), ou passagens do estado *tainted* para *untainted* (no 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*.

Uma vulnerabilidade para ser explorada, os valores das variáveis que os recebem têm de provir do exterior do programa. Se estes valores estiverem envolvidos em algum *bug* de 64 bits, entre a sua atribuição à variável que o representa (*Li*) e a

passagem na chamada de *funções perigosas*/procedimentos (*Lv*), então estamos perante a exploração de uma vulnerabilidade de inteiro.

Uma vez que se pretende rastrear as variáveis de entrada (*tainted*), sendo estas sinalizadas pelo atributo *inpuness*, então a correlacção dos resultados inicia-se a partir dos resultados do referido atributo. Assim sendo, 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;

- *nostate*, nos resultados da segunda fase de processamento será verificado 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 poderá 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, uma vez que este atributo é o responsável pela análise ao código dentro dos procedimentos criados pelo programador. 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.

4.5 *Interface* da Ferramenta

A ferramenta *DEEEP*, de código aberto, permite analisar o código de um projecto ou alguns ficheiros de código, pertencentes a pequenos programas.

```
*****************************************************************
*                     DEEEP Static Analysis Tool                *
*   Detector of integEr vulnerabilitiEs in softwarE Portability   *
*                           version 0.3                         *
*****************************************************************
Usage: deeep [-fl files [-hf<header file>]] |
             [-pj project [[-cf configure flags] [-mk make flags] [-tmp] [-cm]]]
             [-w64 -help]

Options:
   -fl files: verify one or more .c files, that aren't included in a project.

   -hf: specifies extra header files/directory.
        One flag for each file/directory (use only with -fl)
        Example: -hf/pathto/headerfile.h -hf/pathto/dir

   -pj project: Analyze a project, which contains source code, configure,
                Makefile and header files. In this case the user specify
                the path of the project.

   -cf: pass optional flags to configure.
        Example: -cf --prefixe=/usr

   -mk: pass optional flags to make.
        By default '-wki CC=gcc' are passed
        Example: -mk -n

   -tmp: preserve temporary files like the result from configure and make.

   -cm: do not execute configure.

   -w64: print only 64 bits warnings.

   -help: print this text.
```

Figura 4.4: Sintaxe da ferramenta *DEEEP*

A Figura 4.4 apresenta a sintaxe da ferramenta, onde podemos observar que a ferramenta oferece um conjunto de opções que permitem, para além do já referido no parágrafo anterior:

- incluir ficheiros ou directórios de bibliotecas (*header files*) construídos pelo programador;
- passar *flags* de configuração e compilação, respectivamente, ao *configure* e ao *Makefile*;
- analisar somente o código para localização dos *bugs* de 64 bits;
- perservar os ficheiros temporários, criados ao longo da execução da ferramenta;
- anular a execução da configuração do projecto.

Ao longo da execução da ferramenta, após especificação de linha de comando com opções correctas, é mostrado ao utilizador um *output* da mesma, para que o utilizador possa acompanhar os diversos passos na execução da análise. No final da mesma, também o utilizador poderá visualizar uma estatística de execução da ferramenta, com os ficheiros resultantes e sua localização, os tempos de execução da ferramenta (total, análise estática com compilação, configuração e limpeza do projecto).

Por exemplo, para a análise do projecto *Sendmail*, a linha de comando de *DEEEP*

```
#deeep -pj /export/home/iberiam/src/sendmail/sendmail-8.14.1
```

gera o output apresentado na Figura 4.5.

```
*****************************************************************
*                    DEEEP Static Analysis Tool                 *
*   Detector of integEr vulnerabilitiEs in softwarE Portability   *
*                           version 0.3                         *
*****************************************************************
---> Compiling and analysing the project:
       - Makefile file
       - Lint and Splint
     This operation can take a few minutes..................... Done.
---> Parsing 64 bits warnings from Lint results.............. Done.
---> Parsing 64 bits warnings from splint results............ Done.
---> Tainted/untainted input and integer variables........... Done.
---> Cleaning the project (distclean/clean).................. Done.

     Resulting Files:
     ================
          - Directory: /export/home/iberiam/src/sendmail/sendmail-8.14.1/_DEEEP
          - Filenames file: filenames
          - 64 bits warnings file: 64bits
          - Tainted/untainted file(s): inputness and inputness1 (if it exist)
          - Possible Integer Vulnerabilities: YES (vulnerabilities file)

     Execution time and analized files:
     ==================================
          - Total: 0:2:13
          - Makefile and Static Analysis: 0:2:10
          - Configure and clean project: 0:0:3
          - Number of analyzed .c files: 127
          - Number of analyzed lines of code: 112700
```

Figura 4.5: *Output* da ferramenta DEEEP

Capítulo 5

Avaliação Experimental

Este capítulo apresenta os resultados obtidos da experimentação da 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

O código da Figura 5.1, quando mal portado para 64 bits, fica com vulnerabilidades de inteiros. A Figura 5.2 é o resultado da segunda fase de processamento sobre o código da Figura 5.1, ou seja, a identificação das linhas de código que contêm *bugs* de manipulação de inteiros e as respectivas mensagens de aviso. A Figura 5.3 contém o resultado da terceira fase de processamento sobre o mesmo código fonte. Por fim, a Figura 5.4 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 5.1: 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 5.2: 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 5.3: Resultados do Analisador de Fluxo de Dados

Uma análise dos resultados obtidos pela correlação (apresentados na Figura 5.4), ou seja, das vulnerabilidades de inteiros detectadas pelo cruzamento das figuras 5.2 e 5.3, 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* (ver Figura 4.3 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 5.3 e 5.4, a primeira violação (linhas 9 e 10) da Figura 5.3 não está patente na Figura 5.4, por a função *printf* não ser relevante para o tipo de vulnerabilidades e ataques aqui estudados (secções 3.1 e 3.2), 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 5.4: Resultados do Visualizador/Correlacionador

Para dar uma visão da análise realizada na chamada e dentro de procedimentos criados pelo programador, o código da Figura 5.5 apresenta o código da Figura 5.1 modificado de forma a conter um procedimento. As Figura 5.6 e Figura 5.7 apresentam os resultados da segunda e terceira fases de processamento, enquanto que a Figura 5.8 apresenta as vulnerabilidades detectadas pelo *programa correlacionador*.

```
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 5.5: 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 5.6: Resultados do Detector de Bugs (com função)

Uma análise dos resultados da Figura 5.8, ou seja, a vulnerabilidade detectada mostra o seguinte:

- Função *main*, violação entre as linhas 17 e 19: pode-se observar na chamada do procedimento *string_copy* a passagem de uma variável de *input tainted*, ficando a variável do segundo argumento do procedimento com estado *tainted*. Como o procedimento *string_copy* foi definido 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.

```
(in function string_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

(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 5.7: Resultados do Analisador de Fluxo de Dados (com função)

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ +              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);

>>>> 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 string_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 5.8: Resultados do Visualizador/Correlacionador (com função)

À semelhança do exemplo anterior, observa-se, também, que apesar de na Figura 5.7 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* [25].

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 [26].

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 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 mais uma possível 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 5.9 a) está patente o possível código vulnerável pertencente à função *sm_io_fgets* do ficheiro *fget.c*. Esta função é semelhante à função de *input 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)*).

```
     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){
72        if (sm_refill(fp, timeout) != 0) {(...)}
79        len = fp->f_r; }
92   t = (unsigned char *) memchr((void *) p, '\n', len);
104  (void) memcpy((void *) s, (void *) p, len);
```

a) Código fonte vulnerável

```
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) Mensagens de 64 *bits*

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

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

c) *Analisador de Fluxo de Dados*

Figura 5.9: Vulnerabilidade no Sendmail

Na identificação da má adaptação constante na Figura 5.9 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 5.9 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* e/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.

No entanto, apesar de *DEEEP* detectar a possibilidade de vulnerabilidade, a mesma não se verifica, sendo, então, mais um falso positivo. Tal é comprovado pelas linhas 72 e 79, da Figura 5.9 a). Após verificação que o valor de *len* é negativo (pela linha 64), na linha 72, na chamada da função *sm_refill* o valor negativo *fp->f_r* (valor do ficheiro que é atribuído à variável de entrada *len*) é transformado em positivo e, na linha 79, o mesmo é atribuído à variável *len*, ficando esta com um valor positivo. Como o estado de *len* é *tainted*, por *len* ser variável de entrada, este mantem-se, mesmo após a modificação do valor de *len*. Nas linhas 92 e 104 as mensagens de passagem de estado são apresentadas, quando a variável *len* é passada a argumentos que requerem estado *untainted*, dando a ilusão de que uma variável de entrada *tainted* atingiu funções "perigosas".

5.3 Detecção em outros Pacotes de *Software*

Para além do *Sendmail*, a ferramenta *DEEEP* foi utilizada para analisar os pacotes de *software* de código aberto da Tabela 5.1. O número de avisos de *bugs* relacionados com a adaptação para 64 bits foi elevado na maior parte dos pacotes, variando entre cerca de 90 e mais de 21.500. Esses *bugs* deviam ser corrigidos para uma correcta adaptação de ILP32 para LP64. No entanto, só foram encontrados falsos positivos dos tipos de vulnerabilidades de inteiros, os já referidos no *Sendmail*, e nenhuma vulnerabilidade de inteiro. A razão para esta disparidade é que apesar de existirem os bugs, estes não recebem dados de *inputs* e/ou não afectos a funções "perigosas".

Aplicação	Versão	Warnings 64 bits	Vulnerabs	Falsos Positivos	Ficheiros	Linhas de Código	Tempo*
wu-ftpd	2.6.2	217	0	-	50	22.629	25 seg
vsftpd	2.0.5	91	0	-	34	12.376	21 seg
sendmail	8.14.1	1132	0	3	160	112.700	1:52 min
samba	3.0.26a	21566	0	-	651	494.688	23:11 min
proftpd	1.3.0a	409	0	-	95	87.868	1:30 min
lighttpd	1.4.18	886	0	-	93	52.134	2:00 min
inetutils	1.5	980	0	-	175	79.793	1:16 min
dovecot	1.0.5	1984	0	-	359	111.026	5:58 min
bind	9.4.1	1298	0	-	604	323.860	4:24 min

* Tempo de execução do Pré-Processador mais a Análise Estática de Código

Tabela 5.1: Resultados da análise efectuada por *DEEEP*

Capítulo 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.

É interessante verificar que existe um relacionamento efectivo entre a portabilidade de código de ILP32 para LP64 com vulnerabilidades de inteiros, onde a má adaptação do código origina vulnerabilidades de inteiros, onde predomina a de truncamento de dados. Isto leva a pensar que também é necessário nos preocuparmos com este tipo de vulnerabilidade e não só somente com o tipo de vulnerabilidade de inteiros mais conhecida, o *integer overflow*.

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.

De entre as ferramentas avaliadas para a construção de *DEEEP*, verfica-se que a ferramenta *Lint*, apesar de ter uma complexidade baixa e de ter caído em desuso (considerado por alguns), continua a ser eficiente. É uma das poucas ferramentas, das avaliadas, que identifica correctamente os problemas de adaptação de código. Verifica-se que a ferramenta *Splint* é versátil, no sentido de efectuar diversos tipos de análise estática de código (verificação de tipos e análise de fluxo de dados). É uma ferramenta de complexidade muito maior do que a primeira, pela quantidade de *flags* que oferece, levando a que o utilizador tenha uma curva de aprendizagem maior, comparativamente com o *Lint*. A conjugação das duas ferramentas funciona muito bem, por se complementarem.

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.

6.1 Trabalho Futuro

Como trabalho futuro há alguns aspectos que podem ser desenvolvidos e/ou melhorados na *DEEEP*, por insuficiência da ferramenta de análise estática *Splint*, nomeadamente a implementação em *DEEEP* da funcionalidade que permite rastrear os dados provenientes de funções de *input* pertencentes à família *scanf*.

Uma vez que ainda nada se encontra sintetizado sobre a portabilidade de código de ILP32 para os restantes modelos de dados de 64 bits (ILP64 e LLP64), fazer um estudo mais detalhado dos problemas decorrentes desta e avaliar ferramentas de análise estática de código que possam detectar estes problemas.

À luz das arquitecturas actuais e pelo decurso natural do desenvolvimento das arquitecturas de processadores, a arquitetura de 128 bits pressupõem que os processadores podem manipular números inteiros de comprimento de 128 bits, espaço de memória e de dados de largura de 128 bits. Actualmente não existe nenhum processador de uso geral desenvolvido para operar com números inteiros e endereços de 128 bits, embora existam actualmente processadores que operam sobre dados de 128 bits. Por tal, como trabalho futuro poder-se-á dar uma antevisão do que poderá suceder na portabilidade de código de 32 bits ou 64 bits para 128 bits.

Divergir da análise estática de código fonte, passando para a análise estática de código objecto ou ainda construir extensões de compilador, à semelhança da ferramenta RICH [13] e outros trabalhos, para integrar no compilador a detecção de outras classes de vulnerabilidades, podem ser chaves de trabalho futuro interessantes.

Bibliografia

[1] Intel. **Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture**. Setembro de 2006. http://developer.intel.com/design/processor/manuals/253665.pdf

[2] Emerson Alecrim. **Processadores de 64 bits x Processadores de 32 bits**. Maio de 2006. http://www.infowester.com/64bitsx32bits.php

[3] AMD. **AMD64 Architecture Programmer's Manual Volume 1: Application Programming**. Setembro de 2006. http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html

[4] Gabriel Torres. **Inside AMD Architecture**. Maio de 2006. http://www.hardwaresecrets.com/article/324/

[5] Ricardo Zelenovsky, Alexandre Mendonça. **AMD 64-bit architecture (x86-64)**. Outubro de 2004. http://www.hardwaresecrets.com/article/56/

[6] Gabriel Torres. **Intel EM64T Technology Explained**. Dezembro de 2005. http://www.hardwaresecrets.com/article/262/

[7] José António Carriço. **Hard&Software - Curso de Computadores**. ISBN 9728401272. 1997.

[8] António Sampaio. **Hardware para Profissionais**. FCA. ISBN 9727222811. 1998.

[9] Guilherme Arroz, José Monteiro, Arlindo Oliveira. **Introdução aos Sistemas Digitais e Microprocessadores**. (ainda não editado). 2007. http://mega.ist.utl.pt/~ic-ac/livro.pdf

[10] 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

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

[12] 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.

[13] 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.

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

[15] 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

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

[17] 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

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

[19] 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.

[20] 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.

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

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

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

[24] Ibéria Medeiros, Miguel Correia, **Detecção de Vulnerabilidades de Inteiros na Adaptação de Software de 32 para 64 bits**. *In 3ª Conferência Nacional sobre Segurança Informática nas Organizações.* Lisboa, Portugal, pp. 113-132, Novembro 2007.

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

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

[27] Mark Dowd, John Mcdonald, Justin Schuh. **Art of Software Security Assessment, The: Identifying and Preventing Software Vulnerabilities.** Addison Wesley Professional, 2006.

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

[29] Matt Pietrek. **Under The Hood**. Microsoft Systems Journal. Julho de 1998. http://www.microsoft.com/msj/0798/hood0798.aspx

[30] Microsoft Msdn. **64-bit Windows Programming**. Fevereiro de 2006. http://msdn2.microsoft.com/en-us/library/ms775157.aspx

[31] Microsoft Msdn. **The Old New Thing Why did the Win64 team choose the LLP64 model?**. Janeiro de 2005. http://blogs.msdn.com/oldnewthing/archive/2005/01/31/363790.aspx

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

[33] Rodney Mach. **Moving to 64-Bits**. Maio de 2005. http://www.ddj.com/dept/64bit/184401972?pgno=1

[34] Hewlett Packard. **Application Interoperability White Paper, Version 1.0**. Setembro de 1997. http://docs.hp.com/en/938/iop.pdf

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

[36] André Grégio, Luíz Duarte, Luís Barbato, António Montes. **Codificação Segura: Abordagens Práticas**. *In 7º Simpósio Segurança em Informática.* São Paulo, Brasil, Novembro 2005.

[37] J. Viega, J. T. Bloch, Y. Kohno, G. Mcgraw. ITS4: **A Static Vulnerability Scanner for C and C++ Code**. *In Proceedings of the Computer Security Applications Conference*, page 257. 2000.

[38] ITS4. http://www.cigital.com/its4/

[39] Flawfinder. http://www.dwheeler.com/flawfinder/

[40] RATS. http://www.fortifysoftware.com/security-resources/rats.jsp

[41] John Wilander. **Modeling and Visualizing Security Properties of Code Using Dependence Graphs**. *In Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, pp. 65-74. 2005.

[42] Brian V. Chess. **Improving Computer Security using Extended Static Checking**. *In Proceedings of the 2002 IEEE Symposium*, pp. 160-173. 2002.

[43] Eau Claire. http://www.vantuyl.com/chess/EauClaire/

[44] Brian Chess, Jacob West. **Secure Programming with Static Analysis**. Addison Wesley Professional. ISBN 9780321424778. 2007.

[45] D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken. **A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities**. *In Proceedings of the Network and Distributed System Security Symposium*, pages 3–17. 2000.

[46] BOON. http://www.cs.berkeley.edu/~daw/boon/

[47] J. S. Foster, M. Fähndrich, A. Aiken. **A Theory of Type Qualifiers**. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203. 1999.

[48] D. Evans, J. Guttag, J. Horning, Y. M. Tan. **LCLint: A Tool for Using Specifications to Check Code**. *In Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96. 1994.

[49] Jeffrey S. Foster. **CQual User's Guide**. EECS Department University of California. 2002.

[50] CQual. http://www.cs.umd.edu/~jfoster/cqual/

[51] E. M. Clarke, O. Grumberg, D. E. Long. **Model Checking and Abstraction**. *In ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542. 1994.

[52] E. M. Clarke, O. Grumberg, D. A. Peled. **Model Checking**. The MIT Press. 2000.

[53] MOPS. http://www.cs.berkeley.edu/~daw/mops/

[54] UNO. http://spinroot.com/uno/

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

[56] Oded Horovitz. **Big Loop Integer Protection**. Phrack #60. http://www.phrack.org/archives/60/p60-0x09.txt

[57] Michael Howard, David Leblanc. **Writing Secure Code**. Second Edition. Microsoft Corporation. ISBN 9780735617223. 2002.

[58] David Leblanc. **Integer Handling with C++ SafeInt Class**. Janeiro de 2004. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

[59] Michael Howard. **Reviewing Code for Integer Manipulation Vulnerabilities**. Abril de 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp

[60] Firewalls Security Corporation. **Buffer Overflow**. http://www.firewalls.com.br/files/buffer.pdf

[61] Aléxis Rodrigues de Almeida. **Como Funcionam os Exploits**. http://www.firewalls.com.br/files/alexisExploit.pdf

[62] James Whittaker, Herbert Thompson. **How to Break Software Security - Effective Techniques for Security Testing**. Addison Wesley. ISBN13: 9780321194336. 2004.

[63] GNU bash. http://www.gnu.org/software/bash/

[64] GNU awk. http://www.gnu.org/software/gawk/manual/gawk.html

[65] Perl. http://www.perl.org/

Glossário

AMD64

A *Advanced Micro Device*, mais conhecida por AMD, desenvolveu a arquitectura AMD's x86-64, que mais tarde foi denominada por AMD64, a qual é uma extensão do conjunto de instruções (*instruction set*) da arquitectura IA-32 da Intel, e teve como propósito ser uma alternativa à arquitectura IA-64, que era radicalmente diferente da arquitectura IA-32.

Algumas das características desta arquitectura imortantes são: suporte para números inteiros de 64 bits; aumento do espaço de endereçamento linear; aumento do espaço de endereçamento físico; e suporte para aplicações de 32 bits [2][3][4][5].

Barramento de Dados (*data bus*)

Conjunto de linhas que têm por finalidade transportar a informação entre o processador e os dispositivos externos (memória e periféricos de *input*/*output*), ou seja, é um bus bidireccional. A comunicação do processador com os dispositivos externos pode ser de dois tipos: operação de escrita, comunicação efectuada do processador para um dispositivo externo; e operação de leitura, comunicação efectuada de um dispositivo externo para o processador. Contudo, a leitura/escrita da informação nos dispositivos externos só é efectuada após o endereço de memória colocado no barramento de endereços estiver correcto.

A largura do barramento de dados, medida em *bits*, é caracterizada por uma potência de base dois (8, 16, 32, 64, …), define a gama de valores inteiros que é possível guardar numa só célula de memória e, regra geral, é igual à largura da palavra do processador (quantidade de *bits* que o processador processa num só ciclo) [7][8][9].

Barramento de Endereços (*address bus*)

Utilizado para identificar o dispositivo externo (memória RAM ou periféricos de *input/output*) com o qual o processador pretende comunicar, bem como a localização exacta (endereço de memória) a que o processador pretende aceder dentro desse dispositivo.

Cada dispositivo dispõe de um descodificador de endereços (*address decoder*), que quando o processador coloca um endereço de memória no bus de endereços, o descodificador de cada dispositivo identifica as "mensagens" que são dirigidas a esse dispositivo.

Cada localização de memória principal (RAM) e cada periférico de *input/output* são reconhecidos através de endereços. A cada localização de memória principal corresponde um endereço. A cada periférico *de input/output* podem corresponder vários endereços [7][8][9].

Bit

Palavra que resulta da contracção de *binary digit*. Representa a mais pequena parcela de informação que pode ser representada num computador. Fisicamente, pode ser materializado em qualquer dispositivo capaz de assumir dois estados diferentes. Logicamente, utilizam-se os símbolos 0 e 1 para representar cada um dos dois estados [7].

Ciclos-máquina

Períodos fixos de tempo regulados pelo sinal de relógio do processador e que correspondem a uma sucessão de acções elementares reportadas à designação que possuem (leitura, escrita, acesso à memória, entre outros) [7][8][9].

DOS/4GW

Livraria utilizada para permitir que, numa máquina de arquitectura de 32 bits, seja ultrapassado o limite de memória de 640 KB (memória convencional) quando executados programas de DOS (16 bits), podendo endereçar memória estendida [7].

EFI

Extensible Firmaware Interface (EFI), é o nome de um sistema desenvolvido pela Intel, projectado em substituir o sistema de BIOS dos computadores pessoais mais

antigos. É o responsável pelo processo do POST (*power-on-self-test*), no arranque do sistema operativo, e pelo fornecimento do *interface* entre o sistema operativo e o hardware[1].

EM64T

A Intel face à crescente conquista de mercado pela sua concorrente AMD, com a sua arquitectura AMD64, contra o seu processador *Itanium* (arquitectura IA-64), que apresentava grandes problemas de desempenho na coabitação de aplicações de 32 e 64 bits, num mesmo sistema, anuncia uma nova arquitectura de processadores de 64 bits. Assim, surge a arquitectura EM64T (*Extended Memory 64 bit Technology*), que ironicamente adopta a arquitectura AMD64 para o conjunto de instruções, a qual, por esse facto, também é denominada por iAMD64 (o "i" faz referência à Intel). O conjunto de instruções desta arquitectura, advindo da arquitectura AMD64, acaba por ser uma extensão do conjunto de instruções da arquitectura IA-32, o que a faz também ser conhecida por IA-32e (o "e" faz referência a *extended*).

A arquitectura EM64T permite os sistemas endereçar mais do que 4 GB de memória, em ambos os espaços de endereçamento, e fornece suporte para: barramento de endereços de largura de 64 bits, ponteiros de endereçamento de memória de 64 bits, registos de comprimento de palavra de 64 bits, números inteiros de 64 bits e espaço de endereçamento físico de 1 TB (2^{40} bytes).

Tal como a arquitectura AMD64, os processadores da arquitectura EM64T têm operacionalidade em modo de compatibilidade (legacy mode) com aplicações de 32 bits [1][6].

IA-16

Processadores da arquitectura de 16 bis da Intel, também são designados por x86-16 ou simplesmente por IA16. A principal característica destes processadores é o seu comprimento de palavra, de 16 bits, significando que o processador por ciclo processa simultaneamente 16 bits [1].

IA-32

Arquitectura de microprocessadores de 32 bits da Intel, também designada por x86-32 ou IA32, é caracterizada por ter um comprimento de palavra de 32 bits,

significando que o processador processa 32 bits em simultâneo, e por ter os barramentos de dados e de endereços também de 32 bits [1].

IA-64

Em 1999 as empresas Hewlett-Packard e Intel, cooperativamente, fabricaram o microprocessador de 64 bits *Itanium*, que implementava as soluções para espaço de endereçamento linear superior ao existente em IA-32, manipulação de números inteiros demasiadamente grandes e outras mais, lançando no mercado um processador de maior velocidade e desempenho.

A Intel denomina a arquitectura do processador *Itanium* por IA-64, mas ela não é directamente compatível com o conjunto de instruções da IA-32, porque na construção deste processador a Intel não se baseou no conjunto de instruções dos processadores da arquitectura antecessora, descartando o conjunto de instruções da IA-32 e começando um novo conjunto de instruções completamente inovador, utilizando a tecnologia VLIW (*Very Long Instruction Word*) em vez da execução de instruções fora-da-ordem correcta (*out-of-order execution*), como nos processadores das arquitecturas anteriores. Por esse motivo, o processador *Itanium* é apelidado por alguns de "puro sangue", já que é totalmente concebido para execução de aplicações de 64 bits.

Os processadores da linha *Itanium* têm suporte de *hardware* para a arquitectura IA-32, mas são extremamente lentos quando executados, uma vez que as arquitecturas são bastante diferentes. A execução em modo IA-32 é conseguida pelo programa EFI (*Extensible Firmaware Interface*), carregado quando a máquina é ligada, de forma a ler e criar o *interface* para o *hardware* de IA-32, na IA-64[1].

Instruction Set

O *Instruction Set* do processador é o conjunto de instruções básicas que o processador pode interpretar e executar. Cada processador possui o seu próprio conjunto de instruções, o qual também é designado por linguagem máquina do processador [7][8][9].

Out-of-order execution

A execução fora-da-ordem de instruções é um paradigma utilizado na maioria dos processadores de grandes velocidades para fazer uso dos ciclos do processador que

seriam desperdiçados de outra maneira, provocando atraso. Assim, a execução das instruções fora-da-ordem correcta é efectuada, porque as instruções anteriores ainda não terminaram e as que o processador vai executar não dependem do resultado das que ainda não terminaram [1][7].

Sistema operativo

É o conjunto de programas que tornam o computador uma máquina operacional. As principais funções de um sistema operativo são: proporcionar um *interface* de trabalho para o utilizador; permitir o funcionamento de programas de aplicação; criar e manter um sistema de ficheiros em disco; gerir a memória utilizada pelos programas de aplicação; e controlar os periféricos de *input/output* [7].

RAM

Random Access Memory (RAM) é uma memória onde se pode ler, escrever e apagar. Quando se desliga o computador, todo o seu conteúdo desaparece, que, por tal, se designa de memória volátil [7][8].

VLIW

Very Long Instruction Word (VLIW), refere-se à aproximação da arquitectura do processador ao nível do paralelismo de instruções.

As arquitecturas escalares executam uma instrução de cada vez, usualmente na ordem em que as mesmas sucedem no programa. As arquitecturas superescalares tentam aumentar a velocidade dos programas por reordenação e/ou execução de instruções em paralelo, utilizando especializado e frequentemente *hardware* complexo para descobrir tais oportunidades de paralelismo, enquanto o código do programa é executado. Por seu turno, as arquitecturas VLIW executam instruções em paralelo, tendo por base um agendador de instruções fixo, que é determinado quando o código é compilado, não sendo necessário hardware especializado para o efeito, e confiando nos compiladores que analisam e agendam as instruções em paralelo. Como resultado, os processadores com VLIW oferecem poder computacional significativo com menos complexidade de *hardware*, mas, em contra-partida, maior complexidade em compiladores [1].