FACULDADE · DE · CIÊNCIAS UNIVERSIDADE · DE · LISBOA
DEPARTAMENTO DE INFORMÁTICA
Bloco C6 - Piso 3 - Campo Grande, 1749-016 Lisboa
Tel & Fax: +351 217500084

RELATÓRIO DE PROJECTO

sobre

Projecto e Concretização de uma Ferramenta de Injecção de Ataques

realizado na

Faculdade de Ciências da Universidade de Lisboa

por

João Antunes

4 de Julho de 2005

Universidade de Lisboa

Faculdade de Ciências

DEPARTAMENTO DE INFORMÁTICA
Faculdade de Ciências – Universidade de Lisboa
Bloco C6 - Piso 3 - Campo Grande, 1749-016 Lisboa
Tel & Fax: +351 217500084

RELATÓRIO DE PROJECTO

sobre

Projecto e Concretização de uma Ferramenta de Injecção de Ataques

realizado na

Faculdade de Ciências da Universidade de Lisboa

por

João Antunes

Responsável pela FCUL: ***Nuno Fuentecilla Maia Ferreira Neves***

Lisboa, 4 de Julho de 2005

DEPARTAMENTO DE INFORMÁTICA
Bloco C6 - Piso 3 - Campo Grande, 1749-016 Lisboa
Tel & Fax: +351 217500084

Declaração

João Alexandre Simões Antunes, aluno nº 26541 da Faculdade de Ciências da Universidade de Lisboa, declara ceder os seus direitos de cópia sobre o seu Relatório de Projecto em Engenharia Informática, intitulado **"Projecto e Concretização de uma Ferramenta de Injecção de Ataques"**, realizado no ano lectivo 2004/2005 à Faculdade de Ciências da Universidade de Lisboa, para o efeito de arquivo e consulta nas suas bibliotecas e publicação do mesmo em formato electrónico na Internet.

FCUL, 4 de Julho de 2005

Nuno Fuentecilla Maia Ferreira Neves, supervisor do projecto de ***João Alexandre Simões Antunes***, aluno da Faculdade de Ciências da Universidade de Lisboa, declara concordar com a divulgação do Relatório do Projecto em Engenharia Informática, intitulado **"Projecto e Concretização de uma Ferramenta de Injecção de Ataques"**.

Lisboa, 4 de Julho de 2005

Resumo

Nos primeiros anos da Internet (na altura chamada Arpanet), a segurança não desempenhava um papel importante nas comunicações informáticas. Mas à medida que a tecnologia evoluiu e se tornou mais acessível, o número de utilizadores explodiu, bem como as diferentes utilizações que se poderiam dar a esta gigantesca rede. Esta abertura originou a situação actual onde a Internet é utilizada por centenas de milhões de utilizadores como meio de negócios, lazer e comunicação, trazendo consigo um aumento da complexidade e do tamanho dos sistemas distribuídos e das aplicações aí usadas. Os projectos requerem meios cada vez mais complexos durante o seu desenvolvimento, deixando de ser construídos por uma única pessoa ou equipa, para estarem agora afectos a várias equipas. Estas equipas criam e utilizam bibliotecas e componentes que serão depois usados por outras equipas, num esforço controlado e coordenado.

No entanto, a complexidade dos erros também tem aumentado, sendo cada vez mais difíceis de detectar, como é o caso dos erros de segurança. Os métodos convencionais (e automáticos) de detecção não conseguem encontrar muitas vezes este tipo de erros, passando despercebidos pelos arquitectos, programadores, compiladores, *debuggers* ou qualquer tipo de detecção manual e/ou automática de erros. Muitas das novas vulnerabilidades encontradas são detectadas por equipas de segurança especializadas, ou quando já é tarde demais, por piratas informáticos (*hackers*).

Assim, existe a necessidade de um sistema automatizado de detecção de erros, que assista a equipa de desenvolvimento na detecção de vulnerabilidades. A ferramenta *AJECT*, que nasceu do trabalho aqui realizado, tenta satisfazer essa necessidade. Foi assim construído um sistema de injecção de ataques que permitisse obter uma grande flexibilidade e independência em toda a sua arquitectura, possibilitando o diagnóstico de um tipo de erros de difícil detecção — os erros de segurança, que podem levar à criação de vulnerabilidades.

Este trabalho focou-se na criação de um modelo de injecção de ataques — arquitectura — e da respectiva concretização — ferramenta *AJECT* — , tendo em vista a descoberta de novas de vulnerabilidades. O resultado será a detecção e remoção de vulnerabilidades das aplicações, antes da respectiva colocação no mercado. Diminuindo o número de vulnerabilidades, as aplicações tornam-se menos susceptíveis de serem atacadas e consequentemente, mais seguras.

O estágio aqui realizado (de 36 unidades de crédito ECTS) foi efectuado no âmbito do Curso de Especialização Profissional em Engenharia Informática (CEPEI), em conjunto com a realização e aprovação de quatro disciplinas do tipo B.

Conteúdo

1 Introdução **1**

1.1 Organização do Documento 2

2 Objectivos do Projecto e Contexto do Trabalho **3**

2.1 Origem das Vulnerabilidades 4

2.2 Funcionalidades Não Previstas 6

2.3 Modelo de Faltas e Confiabilidade 6

2.4 Tipo de Aplicações-alvo 9

2.5 Detecção de Vulnerabilidades 9

3 Metodologia e Calendarização do Trabalho **11**

3.1 Processo de Desenvolvimento de *Software* 11

3.2 Análise de Requisitos 13

3.3 Análise de Riscos 14

3.3.1 Categorização de Riscos 14

3.3.2 Identificação de Riscos 14

3.3.3 Gestão de Riscos 17

3.4 Calendarização 17

4 O Trabalho Realizado **21**

4.1 Desenho 21

4.1.1 Arquitectura de Injecção de Ataques 21

4.1.2 Modularidade e Independência 27

4.2 Concretização 27

4.2.1 Ferramentas Utilizadas 28

4.2.2 Injector, Monitor e Sistema-Alvo 29

4.2.3 Injecção de Ataques 34

4.2.4 Sincronização de Ataques 37

4.2.5 Monitorização de Ataques 37

4.2.6 Comunicação 40

4.2.7 Pacotes e Tipos de Dados 40

4.2.8 Análise de Detecção de Vulnerabilidades 42

4.3 Resultados Obtidos 43
4.3.1 Tipos de Testes 43
4.3.2 *AJECT* vs *YPOPs!* 45

5 Sumário e Conclusões **51**
5.1 Conclusão 51
5.2 Trabalho Futuro 52

A Análise de Requisitos por Componentes **55**

B Plano RMMM (Risk Mitigation, Monitoring and Management) **59**

C Diagramas de Gantt **65**

Lista de Figuras

2.1 Funcionalidade pretendida *versus* funcionalidade obtida. 6
2.2 Modelo de faltas (Falta, Erro e Falha). 7
2.3 Modelo composto de faltas AVI (Ataque, Vulnerabilidade e Intrusão). . 8

3.1 Modelo incremental. 12

4.1 Entidades envolvidas na arquitectura do *AJECT*. 22
4.2 Desenho da arquitectura do *AJECT*. 23
4.3 Gerador de Ataques. 26
4.4 Diagrama de classes do *Injector*. 30
4.5 Diagrama de classes do `Monitor`. 31
4.6 Diagrama de fluxo do `Monitor`. 33
4.7 Especificação e tipos de mensagem do protocolo de sincronização. . . . 37
4.8 Protocolo de sincronização entre o *Injector* e o *Monitor*. 38
4.9 Disparidade de especificação de diferentes protocolos. 41
4.10 Diagrama de classes da representação de pacotes e campos. 42
4.11 Diagrama de classes da representação do tipo de dados de um campo. . 43
4.12 Interface gráfica da ferramenta *AJECT*. 46
4.13 Início de execução da ferramenta *AJECT*. 46
4.14 Injecção de ataques da ferramenta *AJECT*. 47
4.15 Ataque com sucesso. 48
4.16 Resultado dos ataques (pacotes enviados e sinais recebidos pela aplicação-alvo). 49

C.1 Diagrama de Gantt original 66
C.2 Diagrama de Gantt actualizado 67

Lista de Tabelas

3.1 Avaliação dos diferentes tipos de impacto de riscos 15
3.2 Identificação e análise dos riscos 17

Listagens

4.1 Geração de ataques. 35
4.2 *Thread* de rastreio. 39
4.3 Código fonte do *YPOPs!* com vulnerabilidade. 50

Capítulo 1

Introdução

Este documento descreve o trabalho realizado no âmbito do Curso de Especialização Profissional em Engenharia Informática (CEPEI[1]) para a disciplina **Projecto de Engenharia Informática**. Esta disciplina obriga à realização de um trabalho de fôlego, âmbito e complexidade adequada a uma pós-graduação profissionalizante.

Optei pela Faculdade de Ciências da Universidade de Lisboa como instituição de acolhimento de modo a poder prosseguir, mais facilmente na minha carreira académica.

O estágio foi realizado na unidade de investigação *LaSIGE*[2], do Departamento de Informática (DI) da Faculdade de Ciências da Universidade de Lisboa (FCUL). Financiado pela FCUL, Fundação do Ministério de Ciência e Tecnologia (FCT) e por projectos europeus, esta unidade de investigação tem como objectivos promover a investigação, aprendizagem e transferência de tecnologia em áreas da informática tão vastas como: redes de computadores, algoritmos distribuídos, segurança, tempo-real, bases de dados, recuperação de informação de multimédia, publicação digital, comércio electrónico e bioinformática.

Dentro desta unidade existem vários grupos de investigação, nomeadamente *Navigators*, *HCIM*, *XLDB* e *DIALNP*, todos eles com grandes contributos na investigação científica. No entanto, o meu interesse pelo tema da segurança, inseriu-me no grupo *Navigators*, sob a orientação do Professor Nuno Ferreira Neves. Liderado pelo Professor Paulo Veríssimo, este grupo de investigação de trabalho e renome internacional, está envolvido em actividades de investigação nos domínios dos sistemas distribuídos, da segurança, da tolerância a faltas e do tempo-real.

Um dos motivos pelos quais optei por um projecto na área de segurança em sistemas de *software*, foi a sua crescente importância e reconhecimento no universo da ciência informática. Num mundo de comunicação global e distribuída, é fundamental que os sistemas informáticos hoje utilizados ofereçam cada vez mais garantias de segurança.

No entanto, a complexidade crescente das aplicações torna-as mais susceptíveis de conterem erros, sendo que muitos destes erros escondem vulnerabilidades, que quando

[1] `http://cepei.di.fc.ul.pt/`
[2] `http://lasige.di.fc.ul.pt/`

activadas por um ataque, podem comprometer as propriedades de segurança da aplicação ou de todo o sistema. As garantias de segurança dos sistemas informáticos estão então dependentes da não existência dessas vulnerabilidades.

Porém, o tipo de erros que introduzem vulnerabilidades são bastante difíceis de detectar. As ferramentas que existem não estão preparadas para este tipo de erros (e.g., os compiladores detectam pouco mais do que erros de sintaxe de programação), a maior parte necessitando do código fonte e muitas vezes alterando-o. As que não necessitam do acesso ao código fonte, apenas cruzam a informação presente num banco de dados de vulnerabilidades (previamente detectadas), com a versão da aplicação em teste, o que é inútil em fase de desenvolvimento ou na descoberta de novas vulnerabilidades (e.g., ferramenta SAINT).

Alguns destes erros de segurança são introduzidos desde o início do desenvolvimento de *software*, por vezes até no modelo do sistema, passando despercebidos por todas as fases de desenvolvimento da aplicação. São detectados mais tarde, por equipas de segurança ou por pessoas mal intencionadas, algumas vezes com prejuízo para os clientes que confiaram na segurança das aplicações.

Foi com a segurança das aplicações em mente que a ferramenta *AJECT* foi criada — um novo sistema automatizado de detecção de vulnerabilidades. Para tal, foi construído um sistema de injecção de ataques, que permitisse obter uma grande flexibilidade e independência em toda a sua arquitectura, possibilitando o diagnóstico de um tipo de erros — erros de segurança, que podem levar à criação de vulnerabilidades. Foi este o objectivo do estágio: criar mecanismos e meios (modelo, arquitectura e respectiva concretização) para a detecção de vulnerabilidades em componentes de *software*, através da injecção de ataques.

1.1 Organização do Documento

O próximo capítulo sensibiliza para a segurança informática e erros aplicacionais, que podem levar à violação das garantias de segurança de uma aplicação ou sistema. Aborda em maior profundidade, os objectivos do projecto e contexto onde este se insere, expondo assim a sua importância.

O Capítulo 3 descreve o processo de desenvolvimento levado a cabo na concretização do projecto, nomeadamente: metodologia, calendarização e planeamento.

A realização da ferramenta *AJECT*, passando pelos processos de desenho e respectiva concretização, são explicados no Capítulo 4. Este capítulo começa por descrever a arquitectura e modelo criados, passando pela sua concretização e terminando com os resultados obtidos com a ferramenta desenvolvida.

O relatório conclui com o Capítulo 5, que descreve, de forma crítica e sucinta, o resultado final do projecto e trabalho futuro.

Capítulo 2

Objectivos do Projecto e Contexto do Trabalho

Neste projecto foi desenvolvida uma nova arquitectura de injecção de ataques e construída uma ferramenta que detecte vulnerabilidades através da injecção de faltas maliciosas — a ferramenta *AJECT*. Mas, para uma melhor compreensão de como é feita essa detecção, impõe-se que primeiro se perceba como surgem e como se manifestam essas vulnerabilidades.

Durante a infância da Arpanet, a segurança ainda não desempenhava um papel importante nas comunicações. Mas, à medida que a tecnologia evoluiu e se disponibilizou, o número de utilizadores e de diferentes utilizações da maior rede de redes (Internet) explodiu. De apenas algumas centenas de máquinas ligadas em 1982, para 2,5 milhões em Janeiro de 1994, e em Julho de 1999 onde já existiam cerca de 60 milhões de endereços de computadores registados no serviço de nomes de domínio (DNS) [7].

Esta abertura originou o que se assiste hoje em dia, com a utilização da Internet por milhões de utilizadores, como meio de negócios, lazer, comunicação, etc. Uma das últimas estimativas, aponta para mais de 605 milhões de utilizadores em 2002 [10]. Como em qualquer outro domínio, também aqui é necessário garantir uma utilização segura, oferecendo-se propriedades tais como: confidencialidade, autenticidade, disponibilidade e integridade. A segurança informática é portanto uma área fundamental e bastante activa, onde todos os dias somos deparados com novos desafios.

Juntamente com a evolução da Internet, também o tamanho e complexidade das aplicações criadas nos dias de hoje está em constante crescimento. Os projectos, cada vez de maior dimensão, requerem meios igualmente mais complexos para os resolver: deixaram de ser construídos por uma única pessoa ou equipa, para estarem agora afectos a várias equipas. Estas equipas criam e utilizam bibliotecas e componentes que serão depois usados por outras equipas, num esforço controlado e coordenado.

O resultado são aplicações complexas e de grandes dimensões, produto do trabalho directo e indirecto de várias pessoas, com cada vez mais linhas de código, muitas das

quais provenientes de fontes externa (e.g., bibliotecas de terceiros), e que inevitavelmente induz a introdução de erros.

Existem também, cada vez mais ferramentas de auxílio na detecção e correcção desses erros: compiladores, *debuggers*, bibliotecas e aplicações especiais, etc. No entanto, a complexidade dos erros tem também aumentado, sendo cada vez mais difíceis de detectar. Muitos destes erros escondem vulnerabilidades, que quando activadas por um ataque, podem comprometer as propriedades de segurança de uma aplicação ou de todo o sistema. Em 2004 foram reportados ao CERT 3780 vulnerabilidades, o que corresponde a um crescimento de mais de 346% em relação a 2000, ou de cerca de 1095% face a 1996 [2].

Os métodos convencionais (e automáticos) de detecção de erros não conseguem encontrar este tipo de problemas, passando despercebidos pelos arquitectos, programadores, compiladores, *debuggers* ou qualquer tipo de detecção de erros (manual ou automática). Muitos das novas vulnerabilidades encontradas são detectadas por equipas de segurança especializadas, ou quando já é tarde demais, por piratas informáticos (*hackers*).

Aliada à difícil detecção deste tipo de erros, acresce a dependência e confiança que existe nos sistemas informáticos actuais e o risco das consequências da exploração das vulnerabilidades existentes. Existem muitas aplicações na Internet que confiamos como serem seguras e as utilizamos no dia-a-dia, sem nos apercebermos da quantidade de vulnerabilidades que lhe são encontradas. E todos os dias são publicadas soluções temporárias (*patches*), ou mesmo novas versões, que removem essas vulnerabilidades (e muito possivelmente introduzindo mais erros e novas vulnerabilidades). Exemplos destas aplicações são os *browsers*, clientes de *e-mail*, serviços de banca ou de comércio electrónico, entre outros. A (frágil) segurança deste tipo de aplicações depende de uma constante corrida contra o tempo e contra os *hackers*. Basta um ataque dirigido a uma única vulnerabilidade, para um atacante poder comprometer a segurança do sistema: prejudicando outros e/ou tirando proveito próprio dessa exploração.

Quem pode garantir que o *software* não tem vulnerabilidades? Quem pode garantir que todas as linhas de código estão isentas deste tipo de erros? E quando se utilizam bibliotecas e código de outras fontes? Encontrar manualmente estas vulnerabilidades é, nos dias que correm, uma incomportável e interminável tarefa. Torna-se portanto necessária a criação de ferramentas automatizadas que contribuem para a detecção destes erros.

2.1 Origem das Vulnerabilidades

No contexto da área de segurança informática, as aplicações podem conter numerosos erros que podem ser explorados (vulnerabilidades). Estas vulnerabilidades surgem de defeitos da aplicação que podem resultar de um mau desenho da mesma, da sua

concretização ou mesmo de uma má configuração. Quanto maiores e mais complexas são as aplicações, mais propícias a este tipo de erros serão.

Podem-se classificar três tipos de erros, consoante a fase na qual são introduzidos:

Projecto: Fazem parte do modelo e arquitectura da aplicação. Não sendo erros acidentais, estes serão levados para a implementação de forma propositada, não podendo ser *corrigidos* por qualquer ferramenta de detecção de erros, pois fazem parte do próprio desenho.

Implementação: São inseridos na fase da concretização da aplicação, talvez por descuido de um programador ou má comunicação entre a equipa. São erros de natureza acidental, que poderão ser detectados nas fases de testes, ou *à posteriori*, depois da *release* final.

Configuração: Aparecem devido a más configurações das aplicações ou sistemas, possibilitando a existência estados não previstos no sistema.

Os erros de projecto, inseridos nas primeiras fases de desenvolvimento das aplicações, são os mais graves, pois afectam a arquitectura da aplicação, comprometendo todas as aplicações que concretizem esse desenho. Por exemplo, o protocolo Hypertext Transfer Protocol (HTTP) permite a transferência de conteúdo de *web* como páginas ou outro tipo de ficheiros. No entanto, toda esta informação passa em claro[1] na rede, podendo isto ser uma vulnerabilidade, caso se trate de informação sensível e privada. Trata-se portanto de uma característica inerente ao protocolo — nenhuma implementação do protocolo cifrará os dados. Este erro só será resolvido com alterações ao desenho do protocolo, como por exemplo adicionar a capacidade de ser usado em conjunto com o protocolo Secure Socket Layer[2] (SSL).

Os erros de implementação são os chamados *bugs*, que levam a que muitas vezes criem vulnerabilidades que poderão ser mais tarde maliciosamente exploradas. Apesar de ser teoricamente possível uma aplicação estar livre de erros, esta hipótese é altamente improvável, dada a extensão e complexidade do *software* que é hoje criado. É fácil, por exemplo, que algures na implementação não exista uma verificação no limite dos dados de entrada do utilizador, podendo surgir assim uma vulnerabilidade do tipo *buffer overflow*.

No entanto, ainda existe uma terceira fonte de vulnerabilidades, que é a gestão ou configuração da aplicação. Uma má configuração pode, por exemplo, deixar activas contas de utilizador criadas por omissão (com palavras-chave bem conhecidas), ou não restringir o acesso aos recursos do sistema.

[1] Dados não cifrados, passíveis de serem lidos.

[2] Protocolo que oferece mecanismos de segurança (integridade, confidencialidade e autenticidade) sobre o protocolo TCP.

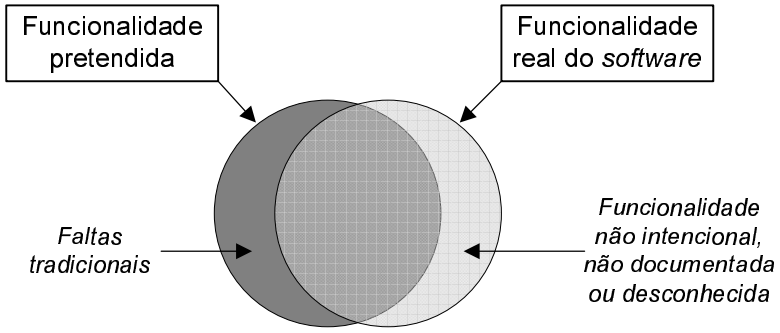


Figura 2.1: Funcionalidade pretendida *versus* funcionalidade obtida.

2.2 Funcionalidades Não Previstas

As aplicações podem estar *correctas*, sem no entanto serem *seguras*. Na verdade, o *software* pode obedecer a todos os seus requisitos e realizar correctamente todas as suas acções, podendo ainda ser explorado por um atacante [18].

Isto deve-se ao facto dos erros de segurança serem diferentes dos erros tradicionais. Os erros tradicionais *impedem* o bom funcionamento do programa, enquanto que os erros de segurança *oferecem* um mau funcionamento do programa. Um erro de segurança pode criar vulnerabilidades que poderão ser vistas como funcionalidades não previstas.

A Figura 2.1 mostra a funcionalidade *pretendida* de uma aplicação (círculo escuro), face à funcionalidade *realmente obtida* (círculo claro). Da intersecção destes dois círculos resulta a funcionalidade *correctamente obtida*. A figura mostra também que existem funcionalidades por implementar, e outras que, por consequência de erros ou por falta de correcta provisão, foram acrescidas. Estas funcionalidades acrescidas, que por não terem sido intencionais e/ou serem totalmente desconhecidas, podem levar a comprometer a segurança da aplicação ou do sistema.

As aplicações podem conter assim numerosos defeitos, inseridos nas fases de desenho, de implementação, ou mesmo devido a uma má configuração, criando vulnerabilidades, que quando maliciosamente exploradas, podem levar à intrusão no sistema e consequente violação das propriedades de segurança. Devido à potencialidade das vulnerabilidades, estas podem ser vistas como funcionalidade acrescida, não intencional, não documentada ou desconhecida.

2.3 Modelo de Faltas e Confiabilidade

Um sistema deve ser projectado e implementado de acordo com uma especificação, que descreve o seu correcto funcionamento. Se a especificação não for violada, diz-se que o sistema está correcto.

Mas até que ponto se pode confiar em que o sistema não viole a sua especificação?

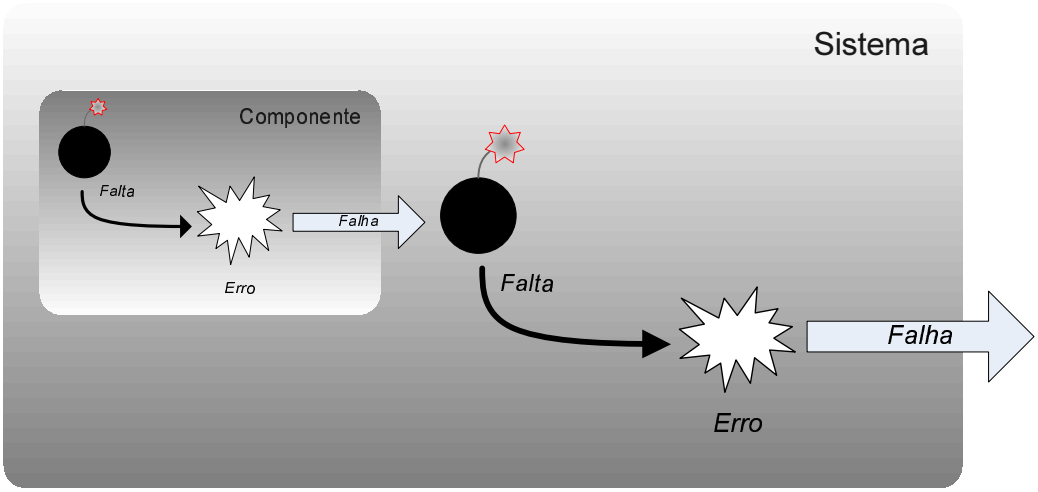


Figura 2.2: Modelo de faltas (Falta, Erro e Falha).

Um sistema deve dar garantias de que está correcto. Chega-se assim à noção de que a **confiabilidade** de um sistema é dada pela medida justificada, da fiabilidade em que um sistema fornece correctamente o seu serviço [15].

A construção de sistemas confiáveis envolve assim a criação de sistemas que oferecem garantias de que nunca violam a sua especificação. Para isso é importante perceber como é que o sistema pode não estar correcto, ou seja, *como* e *porquê* os sistemas falham. O modelo de faltas representado na Figura 2.2, tenta explicar como podem os sistemas falhar, seguindo a sequência falta–erro–falha.

Imagine-se um sistema de ficheiros, cujo propósito passa por disponibilizar informação organizada em registos (ficheiros), que estão guardados fisicamente num disco. A especificação de um sistema de ficheiros diz, por exemplo, que as leituras reflectem a escritas, e portanto, uma leitura de um registo deve devolver o valor da última escrita. Tem-se aqui uma especificação básica. Se o comportamento do sistema violar esta especificação, diz-se que ocorreu uma **falha**.

A construção de sistemas confiáveis envolve assim a prevenção das falhas ocorrerem. Para isso, é necessário compreender-se o processo que leva ao surgimento dessas falhas, que começam devido a uma causa interna ou externa, a **falta**.

Por exemplo, uma descarga eléctrica (falta) pode alterar os *bits* num determinado sector de um disco. A falha pode suceder, caso a falta se manifeste num **erro** (estado erróneo do sistema) que, se não for tratado, pode levar à violação da especificação. Neste caso o erro será um registo corrompido. Se não houver mecanismos de detecção e de correcção deste tipo de erros (como por exemplo *checksums* e replicação), a próxima leitura ao ficheiro afectado, não irá reflectir a última escrita, o que corresponde a um violação da especificação, a falha.

No entanto, o sistema de ficheiros pode ser apenas um componente de um sistema maior, como o sistema operativo. Assim, do ponto de vista do sistema operativo, a

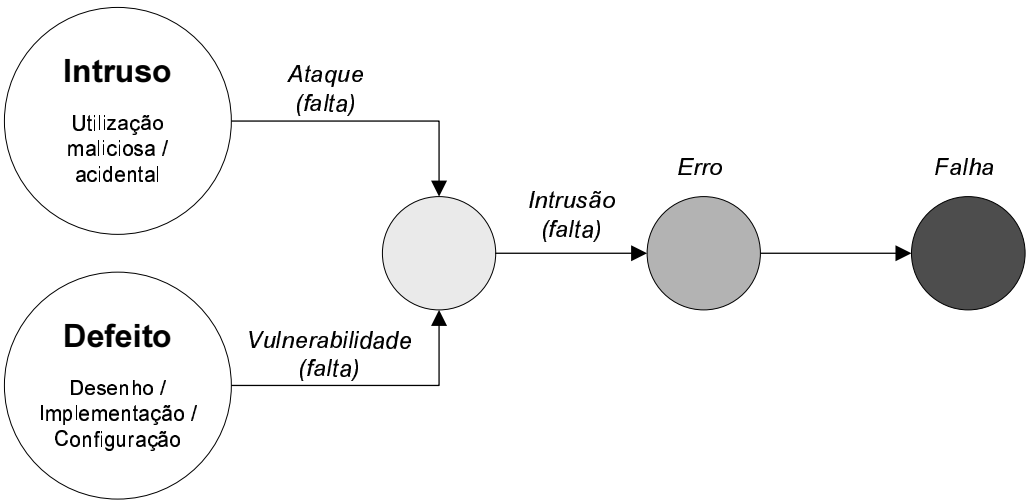


Figura 2.3: Modelo composto de faltas AVI (Ataque, Vulnerabilidade e Intrusão).

falha do sistema de ficheiros é vista como a falta de um componente.

Imagine-se que a zona do disco afectada contém ficheiros de paginação (páginas de memória que, para libertar memória e por não estarem a ser utilizadas, são copiadas para o disco). Se o sistema operativo utilizar o ficheiro corrompido como sendo correcto, o seu conteúdo será copiado para a memória, resultando num erro de paginação. Esta sequência encadeada de faltas, erros e falhas pode ainda continuar, como se mostra na Figura 2.2.

Se existirem mecanismos de detecção e tolerância de faltas, pode-se manter a correcção do sistema, garantindo assim uma maior confiabilidade. No entanto, os tipos de faltas que podem suceder num sistema não se resumem a faltas físicas ou arbitrárias, como uma descarga electromagnética ou um defeito num disco. As faltas podem ser mais complexas e surgir com maior probabilidade. Um exemplo disso são as faltas intencionais, como os "ataques" de um potencial intruso. Este tipo de **faltas maliciosas** traz um risco muito maior para a correcta execução do sistema visto serem propositadas e muito específicas. São faltas que não obedecem a nenhuma distribuição probabilística, nem a nenhum padrão de comportamento.

No entanto, um ataque só é bem sucedido se for dirigido a uma determinada vulnerabilidade. Igualmente, uma vulnerabilidade só apresenta perigo quando explorada pelo ataque. A conjugação dessas duas faltas poderá resultar numa intrusão (falta), que se não for convenientemente tratada, resultará num estado erróneo (erro) do sistema e comprometer a sua segurança (falha). Este comportamento pode ser observado na Figura 2.3, onde se mostra o modelo composto de faltas AVI [14], que estende o anterior modelo de faltas.

Este projecto propõem-se a detectar vulnerabilidades de *software* por intermédio da injecção de ataques no sistema, num ambiente controlado. Os ataques bem suce-

didos irão resultar em erros ou falhas do sistema, revelando assim as vulnerabilidades existentes.

2.4 Tipo de Aplicações-alvo

Do ponto de vista de interacção da aplicação, é na interface com o utilizador, onde muitas vezes se investe mais tempo e dinheiro. É aqui que é feita a interacção entre a aplicação e o utilizador, e em certos casos é a interface o factor mais importante na luta de concorrência entre diferentes vendedores.

Uma aplicação pode receber dados de entrada (*input*) de vários modos: parâmetros na linha de comandos, caixas de diálogo, ficheiros em disco e comunicação na rede [18]. Cada um destes *input* altera o fluxo do programa, podendo expor certas funcionalidades inesperadas.

Apesar de todos os tipos de *input* poderem ser explorados por um atacante, as aplicações mais "interessantes" são aquelas que interagem na rede. A própria função e natureza destas aplicações obriga-as a estarem mais expostas, não necessitando um atacante de estar fisicamente próximo da aplicação para a poder atacar.

Uma aplicação que recebe *input* exclusivamente local, não está *directamente* vulnerável a ataques do exterior, a não ser que os dados tenham origem indirecta no exterior.

Em vários casos, as acções do utilizador são apenas restringidas na interface gráfica, sendo possível que a aplicação interaja pela rede, com poucas ou nenhumas restrições.

Por isso, as aplicações mais "interessantes", do ponto de vista do atacante ou de um injector de ataques, começam por ser aquelas que oferecem acessibilidade remota. As aplicações expostas na rede são as que se enquadram neste perfil, nomeadamente: servidores *web*, de correio electrónico, de autenticação ou de partilha de ficheiros, etc. Será este tipo de aplicações, as aplicações-alvo da ferramenta desenvolvida.

2.5 Detecção de Vulnerabilidades

Do ponto de vista do atacante, os ataques são feitos a partir do envio de pacotes especialmente criados, que exploram determinadas vulnerabilidades, conhecidas *à priori*. Estas vulnerabilidades não são mais do que o produto de erros que levam a aplicação a estados não previstos pelos seus criadores.

A aplicação vai então transitar entre os diferentes estados (válidos e possivelmente inválidos), consoante o protocolo e os dados que recebe pela rede. É sobre estes diferentes estados que a aplicação vai ser testada face a um ambiente malicioso.

Existe portanto a necessidade de ferramentas que permitam a detecção destes erros/vulnerabilidades. Uma boa ferramenta de detecção de erros permite que seja testada a maior cobertura possível de todos os estados do sistema.

Este projecto tem como objectivo a criação de uma ferramenta (*AJECT*) que permita testar a fiabilidade de uma qualquer aplicação que interaja com a rede. Está portanto inserida numa classe de ferramentas de detecção de erros que visam a descoberta de erros semânticos, i.e., erros que resultem num comportamento *incorrecto* da aplicação;

A ferramenta irá então enviar pacotes maliciosos à aplicação-alvo (injecção de ataques), simulando as acções de um atacante num ambiente malicioso. Os ataques poderão revelar as vulnerabilidades existentes, com a observação dos erros e falhas do sistema, através de mecanismos de monitorização próprios.

Já existem várias ferramentas de diagnóstico de vulnerabilidades, mas de um modo geral, todos eles seguem um modelo de diagnóstico padrão, que passa em reconhecer o sistema-alvo (sistema operativo, serviços que estão a correr, etc.), testá-lo face a vulnerabilidades bem conhecidas e mostrar os resultados. Alguns exemplos são:

- QualysGuard [12];
- Internet Scanner [8];
- FoundStone Enterprise [3];
- STAT Scanner Professional [4] ou
- SAINT [13].

No entanto, este tipo de ferramentas só procura vulnerabilidades conhecidas: as novas vulnerabilidades só serão encontradas se o detector de vulnerabilidades as tiver previamente adicionado na sua base de dados. O diagnóstico é realizado através de uma correlação entre a versão da aplicação e as vulnerabilidades conhecidas naquelas versões, que pode passar também pela própria experimentação/exploração da vulnerabilidade.

O modelo de injecção de ataques aqui proposto assenta na criação dinâmica de ataques, não estando dependente de nenhum conhecimento prévio sobre as vulnerabilidades conhecidas da aplicação que se quer diagnosticar. Pretende-se com isto a criação de uma ferramenta que consiga não só encontrar vulnerabilidades conhecidas, como também vulnerabilidades novas e não documentadas.

Capítulo 3

Metodologia e Calendarização do Trabalho

Este capítulo explica como foi projectado, organizado e escalonado o trabalho. Vários factores foram tidos em conta na definição do projecto e de como este seria concretizado. Estes aspectos são fundamentais no desenvolvimento de um projecto e assentam nas fundações da engenharia de *software*.

3.1 Processo de Desenvolvimento de *Software*

Vários processos de desenvolvimento de *software* foram tidos em conta, na concretização da ferramenta *AJECT*, nomeadamente:

- Modelo em cascata
- Modelo em espiral
- Modelo incremental
- Processo unificado de desenvolvimento de *software* (USDP)
- Modelo de programação extrema
- Modelo de prototipagem

Todos os modelos partilham, de grosso modo, as mesmas fases (desenho, implementação, integração e testes), diferindo noutros aspectos.

Podem-se debater as vantagens e as desvantagens de cada um dos modelos, mas será sempre difícil chegar-se à conclusão em relação a qual o melhor processo de desenvolvimento de *software*. É certo que será melhor seguir um processo de desenvolvimento de *software* (mesmo que não seja o mais adequado) do que nenhum. Todos os modelos são viáveis e úteis, mas a sua utilidade está ligada aos recursos disponíveis, natureza, complexidade, tipo e duração do projecto. É então imperativo, analisar a adequação dos vários modelos a este projecto em particular.

Figura 3.1: Modelo incremental.

Este é um projecto que, não sendo de grande dimensão, dispõe de recursos e duração bastante reduzidos, destinado a uma equipa pequena (uma pessoa); pelo que modelos que lidem com grandes projectos e afectos a muitos recursos, não se adequam.

No entanto, devido ao facto de ser um projecto novo, criado de raíz, não existe uma base sólida e amadurecida por detrás deste. Será então necessário uma contínua e constante definição da natureza, âmbito e objectivos junto dos "clientes" (orientador do projecto). Este regular aperfeiçoamento implica várias revisões e iterações ao longo do tempo de vida do projecto, sendo necessário um modelo dinâmico, flexível e com alguma recursividade.

Com base nos critérios acima descritos e na natureza/definição dos vários modelos possíveis, dois desses modelos se evidenciaram: o modelo espiral e o modelo incremental. Ambos são bastante semelhantes — destinados a projectos cujos objectivos estão algo vagos ou imprevistos, envolvendo uma série de frequentes iterações cíclicas —, diferindo no seguinte aspecto: em cada ciclo, o **modelo espiral** refina continuamente as mesmas funcionalidades, enquanto que o **modelo incremental** permite obter uma maior flexibilidade, podendo ser criadas novas funcionalidades em cada ciclo.

O modelo incremental está mais adequado em situações onde a direcção do projecto é mais difusa. Os rápidos ciclos de desenvolvimento permitem obter rápidos resultados, permitindo um rápido *feedback* no sucesso das últimas iterações, e na direcção das próximas. **Foi então escolhido o modelo incremental como processo de desenvolvimento do *AJECT*.**

A Figura 3.1 apresenta um diagrama simplificado do modelo incremental, podendo-se ver as várias fases (simplificadas), organizadas em várias iterações. Foram planeadas duas iterações. No final de ambas as iterações deverá resultar uma versão do *AJECT*, melhorada a partir da anterior. Os diagramas de Gantt (versão original e versão actualizada) apresentados no Anexo C, ilustram as duas iterações, bem como as fases de cada uma destas.

3.2 Análise de Requisitos

Durante as primeiras reuniões com o orientador do projecto, foi discutido o trabalho a realizar, os seus objectivos, requisitos, expectativas e restrições da ferramenta a concretizar. Isto permitiu uma clarificação do problema, bem como uma melhor especificação da ferramenta que o iria resolver.

A informação daqui resultante permitiu elaborar uma lista inicial de requisitos, que serviu de ponto de partida para o desenvolvimento do projecto. No entanto, os requisitos também evoluíram ao longo do trabalho até chegarem à sua versão final. Segue-se os requisitos identificados junto do cliente:

- executar um determinado conjunto de ataques (teste);
- criar pacotes maliciosos, de forma automatizada e em conformidade com um determinado teste;
- enviar pacotes (maliciosos) para um determinado endereço e porto, de forma transparente e independente do protocolo de transporte (TCP ou UDP);
- lançar a aplicação para receber os ataques (aplicação-alvo);
- terminar a aplicação-alvo;
- possuir mecanismos para suportar vários protocolos (e.g., POP, SMTP, HTTP) e ser independente do sistema operativo;
- registar a comunicação efectuada durante os ataques (pacotes);
- observar e registar o estado da aplicação-alvo durante os ataques (monitorização);
- a informação resultante dos ataques (pacotes e monitorização) é legível ao utilizador;
- reportar os resultados de cada ataque, tal como possíveis erros e vulnerabilidades encontradas;

Mas esta análise não se deve limitar a uma simples enumeração de requisitos. Uma vez definido o modelo conceptual e arquitectura, devem-se especificar e analisar não só os requisitos do sistema, como também os requisitos dos componentes que o compõem. Remete-se o leitor para o Anexo A onde encontrará a análise de requisitos por componentes. Além de uma melhor compreensão do problema e do que se espera da solução, esta análise aprofundada de requisitos desempenhou um outro papel fundamental no desenvolvimento da ferramenta, clarificando a definição de cada um dos componentes a concretizar.

3.3 Análise de Riscos

A primeira Lei de Murphy [19] diz que:

> "Se há duas ou mais formas de fazer alguma coisa e uma das formas resultar em catástrofe, então alguém a fará."

Qualquer projecto tem associado riscos que podem comprometer o seu sucesso. *Imprevistos*, cuja definição implica a ocorrência não prevista de algo, nascem da realização de riscos (i.e., um risco tornado realidade).

No entanto, uma prévia identificação e análise destes riscos, permite uma melhor preparação e possível resolução para este tipo de problemas. Assim, a identificação do maior número de riscos e a criação de planos de contingência para cada um deles, permite que estes sejam atempadamente *previstos* e que o seu impacto seja minimizado.

3.3.1 Categorização de Riscos

Para uma melhor identificação e compreensão dos riscos associados a um projecto, é importante ter em linha de conta os mais graves e quais aqueles que têm uma maior probabilidade de ocorrerem. A seguinte classificação pretende agrupar os riscos consoante a sua natureza:

Riscos de projecto: Ameaças relacionadas com o planeamento e calendarização do trabalho, estrutura do projecto e recursos (humanos ou materiais), que podem causar atrasos nos prazos.

Riscos técnicos: Problemas que possam surgir e que põem em causa a qualidade do *software* a desenvolver, bem como o prazo de entrega do mesmo. Estão associados à criação do produto em si, ou seja problemas de desenho, codificação, verificação e manutenção do *software*;

Uma avaliação das consequências dos riscos, consoante o seu impacto, está descrita na Tabela 3.1. Cada linha da tabela representa um tipo de impacto, identificado pelo grau de impacto, na primeira coluna. Os resultados mais graves são do tipo catastrófico (grau 5), que poderão pôr em causa a conclusão do projecto.

Além do impacto, deve-se também considerar a probabilidade com que o risco pode ocorrer. Nem todos os riscos devem ser considerados. Não existe grande proveito em considerar riscos com uma probabilidade bastante baixa (próxima de zero); pelo contrário, quanto mais provável a ocorrência do risco, maior o cuidado que se deverá ter na respectiva análise e gestão de risco.

3.3.2 Identificação de Riscos

Seguindo a categorização atrás criada, são apresentados os riscos de projecto e técnicos encontrados.

		Impacto no produto		Impacto no projecto	
ID	**Descrição**	**Desempenho**	**Suporte**	**Custo**	**Escalonamento**
1	quase insignificante	sem redução	suporte trivial	quase sem custo	pouco ou sem atraso
2	desprezável	redução mínima	suporte trivial	baixo custo	atraso facilmente recuperável
3	marginal	pouca redução	suporte rápido	alguns custos	ligeiro atraso
4	crítico	redução significativa	suporte demorado	custos significativos — possível derrapagem	atraso significativo — prazo em risco
5	catastrófico	degradação significativa	suporte demasiado demorado	derrapagem financeira	atraso irrecuperável — entrega adiada

Tabela 3.1: Avaliação dos diferentes tipos de impacto de riscos

Riscos de Projecto

a) *Incorrecta interpretação ou especificação das exigências do cliente.*
Este risco resulta de uma análise deficiente de requisitos ou devido a alguma complexidade e incompreensão relativamente às exigências do cliente. Podem existir outros factores, como a dificuldade ou meio de comunicação entre o autor e o cliente (e.g., *e-mail*, reuniões curtas e sem preparação), que dificultem uma boa análise de requisitos.

b) *Impossibilidade da realização de reuniões com o cliente, sempre que necessário.*
O cliente, que acumula o papel de orientador do projecto, é quem melhor sabe do que se espera do trabalho a desenvolver. É por isso essencial que exista facilidade de comunicação entre o orientador e o aluno, principalmente quando necessário.

c) *Indisponibilidade de um membro da equipa.*
Pode, a qualquer momento, haver indisponibilidade quer do aluno, quer do orientador (que também pode ser visto como um membro da equipa). Existem diversos tipos de indisponibilidade a ter em conta, podendo variar com a frequência (esporádica, frequente) ou com a intensidade (pequenos ou longos períodos), dependendo da causa (indisposição, doença, etc.).

d) *Falta ou avaria de material informático.*
Um projecto pode, em qualquer altura, necessitar de mais recursos, quer por escassez, quer por avaria ou mau funcionamento dos mesmos.

e) *Perda de dados informáticos.*
O trabalho realizado é guardado sob a forma de dados informáticos. Ao longo do desenvolvimento do projecto, os dados são constantemente alterados e manipulados,

havendo um risco inerente de que algo corra mal, quer por falha humana (e.g., remoção acidental de ficheiros), ou por falha do suporte físico (e.g., disco rígido com sectores defeituosos).

Riscos técnicos

f) *Desenho ou arquitectura inadequadas à concretização do sistema.*
As primeiras fases do projecto podem especificar incorrectamente determinados requisitos e não contemplar certas funcionalidades, resultando num modelo errado ou inadequado do sistema. Isto pode apenas ser detectado em fases de desenvolvimento posteriores, durante a implementação do modelo.

g) *Falta de conhecimentos sobre segurança e injecção de faltas.*
O trabalho aqui desenvolvido requer algum conhecimento específico e avançado de várias áreas como a segurança, a injecção de faltas, sistemas distribuídos, detecção e exploração de vulnerabilidades, entre outros.

h) *Problemas ou inadequação da(s) linguagem(s) de programação utilizada(s).*
A linguagem de programação, como meio de concretização, é um factor de grande importância para o sucesso do projecto. Por isso é imperativo que a linguagem utilizada esteja à altura do desafio e que o próprio programador possua os conhecimentos e capacidades técnicas suficientes para a utilizar.

i) *Inadequação das ferramentas escolhidas para realização do projecto.*
As ferramentas e meios disponíveis podem não se revelar os mais adequados para o desenvolvimento do trabalho. Existem diversas causas para que este risco se concretize, tal como inexperiência ou dificuldade de utilização, inutilidade da ferramenta para a tarefa proposta ou mesmo devido a defeitos da mesma.

j) *Número de erros superior ao esperado.*
É impossível garantir um trabalho livre de erros, portanto estes são esperados ao longo do desenvolvimento, existindo até métricas que estimam o número de erros por cada linha de código. No entanto, o número e tipo de erros pode comprometer a qualidade do projecto.

No entanto, a simples identificação de riscos não é suficiente. Devem-se considerar outros factores para além da categoria a que pertence, tais como: o impacto do risco no projecto, a probabilidade deste ocorrer e mesmo as acções a tomar de forma a minimizar as suas consequências. Na Tabela 3.2 estão identificados os riscos deste projecto, ordenados de acordo com a respectiva probabilidade de se realizarem (muito provável, provável e pouco provável) e pelo grau de impacto.

	Risco	Categoria	Probabilidade	Impacto
g)	Falta de conhecimentos sobre segurança e injecção de faltas	técnicos	muito provável	4
f)	Desenho ou arquitectura inadequadas à concretização do sistema	técnicos	provável	5
h)	Problemas ou inadequação da(s) linguagem(s) de programação utilizada(s)	técnicos	provável	4
j)	Número de erros superior ao esperado	técnicos	provável	4
a)	Incorrecta interpretação ou especificação das exigências do cliente	de projecto	provável	3
i)	Inadequação das ferramentas escolhidas para realização do projecto	técnicos	provável	3
c)	Indisponibilidade de um membro da equipa	de projecto	pouco provável	5
e)	Perda de dados informáticos	de projecto	pouco provável	5
d)	Falta ou avaria de material e recursos informáticos	de projecto	pouco provável	4
b)	Impossibilidade da realização de reuniões com o cliente, sempre que necessário	de projecto	pouco provável	2

Tabela 3.2: Identificação e análise dos riscos

3.3.3 Gestão de Riscos

Sendo um elemento que pode influenciar bastante a qualidade final do trabalho, há que criar medidas de gestão e planos de acção alternativos, no caso do aparecimento de problemas (inesperados).

Uma gestão *proactiva* dos riscos tenta evitar que estes surjam, tomando acções preventivas, que começam logo desde início a observar e registar os factores e causas que possam aumentar ou diminuir a ameaça. Devem-se também criar planos de contingência que descrevem as acções a tomar no caso em que o risco é real. Foi criado um plano de prevenção, monitorização e gestão de risco (*Risk Mitigation, Monitoring and Management Plan*) apresentado no Anexo B.

A título de exemplo, e voltando à Tabela 3.2, pode-se observar que um dos riscos mais importantes é o risco *g) Falta de conhecimentos sobre segurança e injecção de faltas*. De forma a minimizar o grau de ameaça deste risco, foi planeada uma contínua contextualização e enquadramento na área de segurança. No entanto, caso surjam problemas durante o desenvolvimento do projecto, há sempre a possibilidade de marcar uma reunião com o orientador em busca de aconselhamento técnico, que poderá indicar um conjunto de material bibliográfico adicional que permita resolver o problema.

3.4 Calendarização

Devido à pequena dimensão do projecto (não se tratando de um projecto que envolva muitos recursos ou uma grande equipa) e ao facto do estágio ser efectuado

no âmbito de uma especialização que engloba a realização de quatro disciplinas, a concretização do projecto seguiu um plano algo flexível.

Assim, houve uma contextualização contínua na área de segurança, que envolveu o estudo da descoberta e exploração de erros de segurança e a experimentação de ferramentas de diagnóstico de vulnerabilidades. Esta preparação permitiu depois passar à criação de um modelo de injecção de ataques e para a construção da ferramenta *AJECT* e respectiva experimentação.

O projecto teve as seguintes fases:

- Contextualização e enquadramento nos conceitos de segurança;
- Análise conceptual e análise de requisitos;
- Modelação da arquitectura (construção de diagramas);
- Implementação, testes dos componentes e integração;
- Teste do sistema (global);
- Execução da ferramenta face a um aplicação-alvo (experimentação);
- Resultados obtidos e conclusões.

De lembrar que a maior parte das fases acima descritas, são visitadas a cada iteração do modelo incremental escolhido como processo de desenvolvimento de *software* (exceptuando a primeira e última fases).

Foi criada uma calendarização detalhada, de acordo com o tipo de projecto, tamanho da equipa de trabalho (uma pessoa) e processo de desenvolvimento de *software*, onde foi especificado o tempo que cada fase e respectivas tarefas deveriam tomar. No entanto, não foi possível seguir o planeamento e calendarização originais, devido ao aparecimento de alguns problemas na fase de concretização. Este contratempo teve origem na realização do risco *f) Desenho ou arquitectura inadequadas à concretização do sistema*, em particular:

- na especificação dos pacotes e do protocolo-alvo[1] e
- na representação genérica dos tipos de dados.

Além disso, e como em qualquer projecto, foram surgindo os problemas de implementação, não previstos durante o planeamento e que inevitavelmente atrasaram o projecto, como foi o caso da exploração e utilização de mecanismos específicos do sistema operativo (*Linux*) que permitissem a monitorização de processos.

[1]Este é o protocolo concretizado pela aplicação-alvo, como o protocolo HTTP, concretizado por um servidor *web*.

Com o objectivo de resolver os problemas acima descritos é criada uma nova fase de desenvolvimento, não contemplada no planeamento original. Daqui resultou um *novo diagrama de Gantt*, actualizado com a tarefa adicional *Alterações ao Modelo*, composta pelas duas sub-tarefas de *Desenho* e de *Implementação*. O antigo e novo diagrama de Gantt podem ser observados nas Figuras C.1 e C.2, no Anexo C.

Apesar dos obstáculos e atrasos que surgiram, e o facto da calendarização ter sofrido alterações, é importante referir que sem o referido planeamento, as consequências poderiam ser mais adversas, podendo mesmo colocar em risco a conclusão do projecto dentro do prazo estipulado.

Capítulo 4

O Trabalho Realizado

Este capítulo descreve o trabalho desenvolvido ao longo do projecto de engenharia. Com o propósito de melhor levar a cabo esta tarefa, começou-se por definir o modelo do injector de ataques, apresentado na primeira secção. Esta secção pretende também demonstrar *o que* foi feito. As secções seguintes revelam *como* foi o trabalho realizado (Secção 4.2), bem como *quais* os resultados que se obtiveram (Secção 4.3).

4.1 Desenho

4.1.1 Arquitectura de Injecção de Ataques

Foi anteriormente apresentado o modelo de faltas AVI, onde se mostrou um tipo específico de faltas — o *ataque* —, e de como poderiam despoletar uma falha no sistema. Num ambiente controlado, podem-se injectar faltas (e.g., atrasar um relógio, simular um ficheiro corrompido, etc.) e observar os resultados, i.e., verificar se o sistema se comporta correctamente e de acordo com a sua especificação. A injecção de ataques é um tipo específico de injecção de faltas, que procura detectar se um sistema é vulnerável a faltas maliciosas.

Durante o período inicial deste trabalho, foram estudados vários sistemas de injecção de faltas (alguns por *hardware*, outros por *software*)[6][16][17], enquadrando-se este projecto nos modelos de injecção de faltas por *software*, nomeadamente de faltas maliciosas.

Existem três entidades básicas envolvidas na arquitectura do $AJECT$: o *Injector*, o *Monitor* e o *Sistema-Alvo*. As duas primeiras entidades compõem o $AJECT$, enquanto que a terceira corresponde ao sistema que se deseja testar. Na Figura 4.1 estão representadas estas três entidades, cada uma delas como uma "caixa preta", ainda desprovida de detalhes.

De um modo geral, o *Injector* tem o papel de injectar os ataques no *Sistema-Alvo* e receber as respectivas respostas. Existe ainda o *Monitor*, cujo objectivo é o de observar e registar cada um dos ataques e respectivos resultados, para posterior análise — o que exige uma cuidada sincronização com o *Injector*. Por fim, o *Sistema-Alvo*, que é

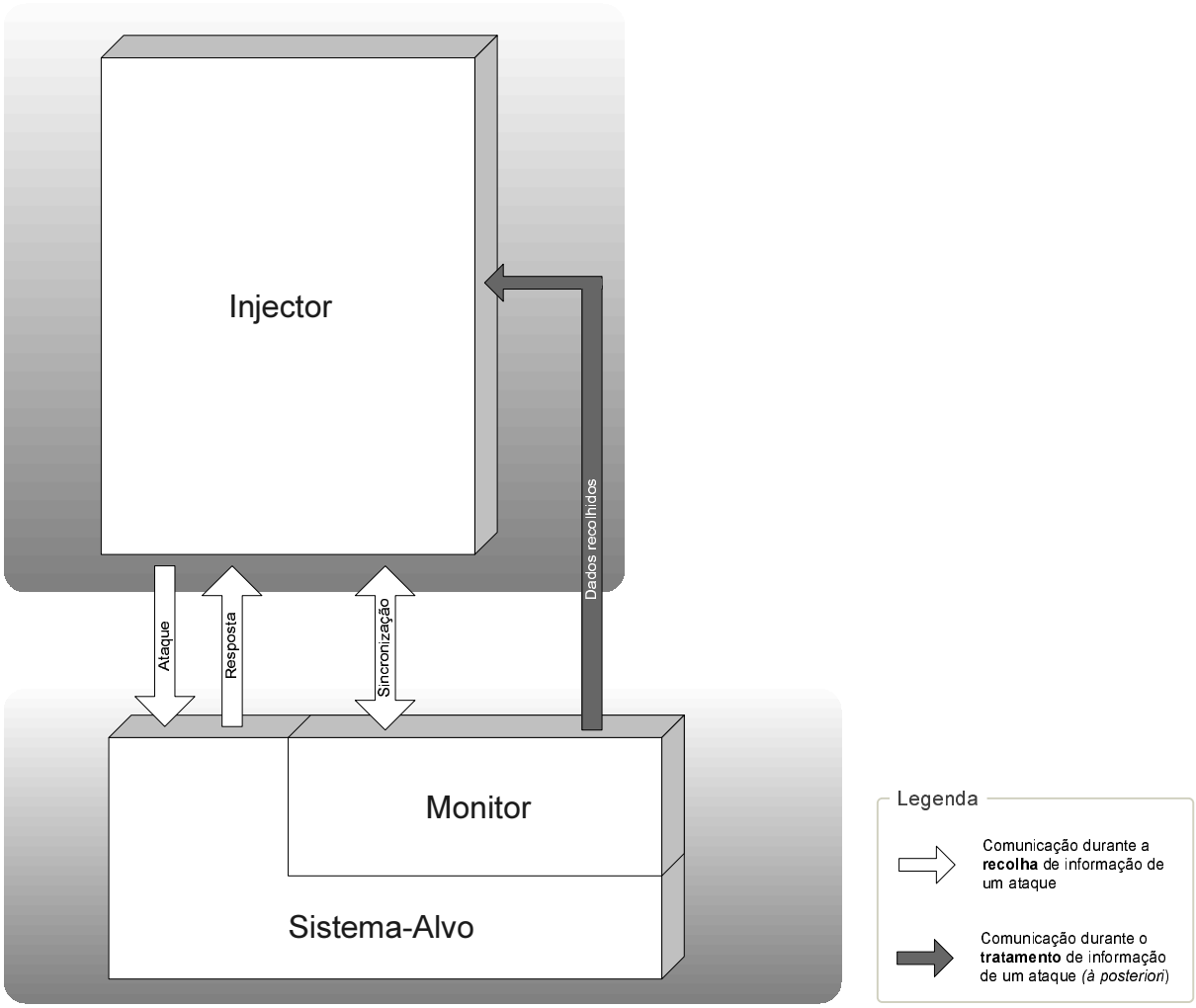


Figura 4.1: Entidades envolvidas na arquitectura do *AJECT*.

composto pela aplicação-alvo (que se deseja testar), plataforma e sistema operativo, representando todo o ambiente no qual a aplicação corre.

Como se pode verificar, esta arquitectura pressupõe a injecção de ataques e a respectiva análise. No entanto, existe uma *clara separação* entre estes aspectos: *Injector* e *Monitor*. Por um lado, a monitorização de uma aplicação requer uma grande proximidade, talvez até a partilha de recursos com o sistema operativo ou intercepção de sinais recebidos. Por outro lado, a injecção de ataques pela rede não necessita dessa proximidade; é até conveniente que o sistema que ataque (*Injector*) esteja o mais independente possível do atacado (*Sistema-Alvo*), para não prejudicar os testes e a respectiva análise.

Olhando para o interior de cada uma destas entidades, podem-se observar os componentes que as constituem e as respectivas relações, como se pode verificar na Figura 4.2. O modelo modular apresentado, mostra os componentes e suas relações. Os componentes são orientados à tarefa e específicos em relação ao seu propósito, corres-

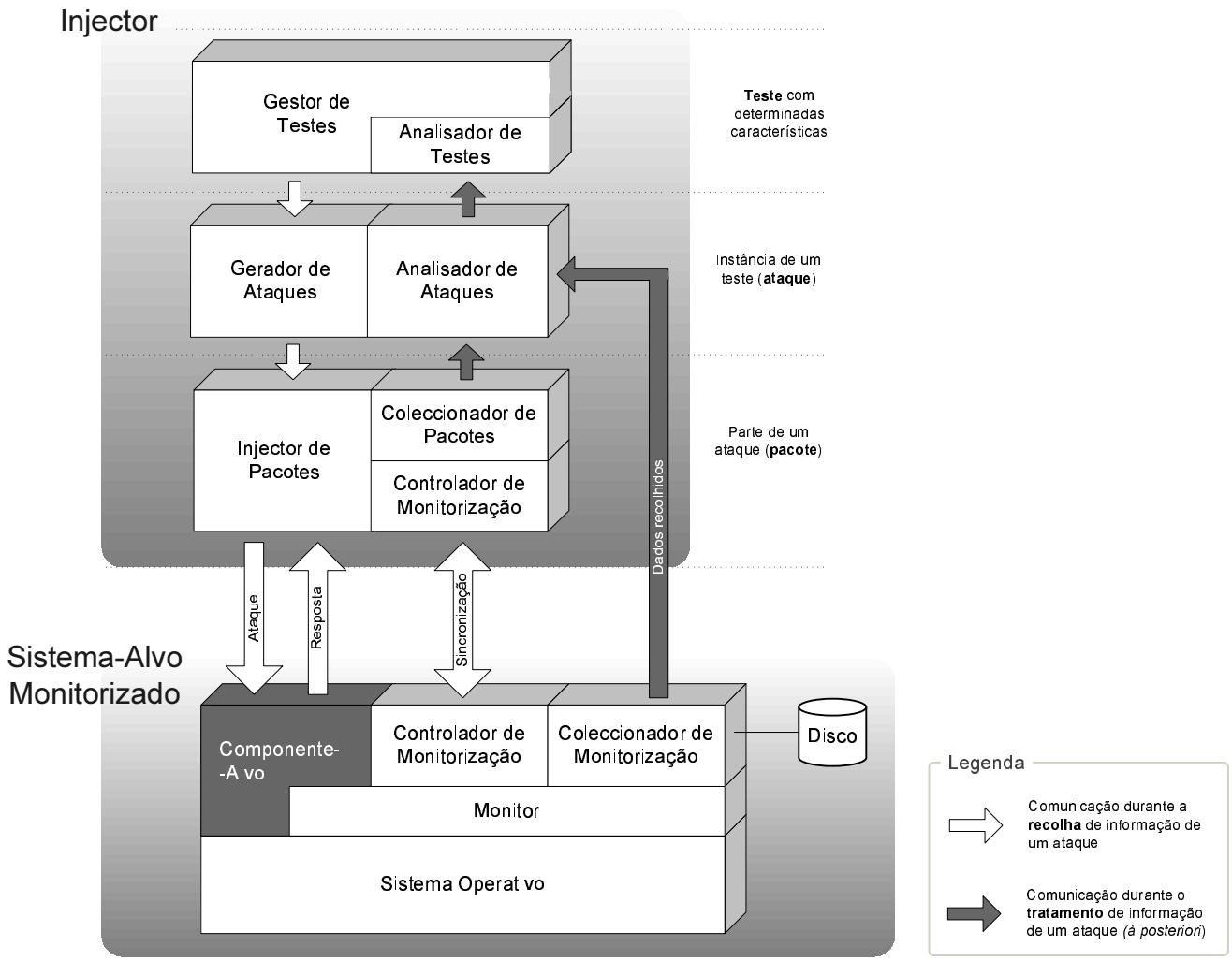


Figura 4.2: Desenho da arquitectura do *AJECT*.

pondendo aos blocos de construção das entidades do *AJECT* atrás mencionadas — o *Injector* e o *Monitor*.

Teste, Ataque e Pacote

A injecção de ataques está ligada aos testes pretendidos e personificada pelos pacotes a enviar. O conceito de ataque é assim bastante vago, podendo ser bastante abrangente. Dependendo do ponto de vista, um ataque pode significar algo tão genérico como modificar a sintaxe de um protocolo, ou algo mais simples e específico como o envio de um pacote com um conteúdo específico.

Definiu-se assim que o processo de injecção de ataques passa por definir primeiro um conjunto de **testes** (visão mais genérica). Cada um destes testes pode-se decompor em várias instâncias — **ataques**. Por sua vez cada ataque é concretizado na rede através do envio dos **pacotes** que o constituem. É esta a relação de decomposição do conceito de injecção de ataques.

Esta decomposição de teste, ataque e pacote, está presente na arquitectura (ver Figura 4.2), onde os componentes que formam o *Injector* estão organizados nestes três níveis:

- Os componentes do primeiro nível (**teste**): `Gestor de Testes` e `Analisador de Testes`.
- De seguida apresentam-se os componentes relativos às instâncias de um teste (**ataque**): `Gerador de Ataques` e `Analisador de Ataques`.
- Por fim, a última decomposição de um teste corresponde ao **pacote** a ser injectado: `Injector de Pacotes`, `Coleccionador de Pacotes` e `Controlador de Monitorização`.

A título de exemplo, imagine-se que o `Gestor de Testes` suporta um leque alargado de testes, escolhendo este começar pelo **teste** de sintaxe. Por agora, basta saber que este teste consiste na validação da especificação formal dos pacotes que pertencem ao protocolo, como por exemplo o número e ordem dos campos de um pacote. Explica-se em maior detalhe este, e outros tipos de testes, mais à frente na Secção 4.2.3.

Daqui resultará a criação de um número considerável de ataques diferentes, pelo `Gerador de Ataques`. Cada **ataque** pode ser considerado uma instância de um teste. No caso do teste de sintaxe, uma sequência de ataques possíveis serão criados, de modo a verificar as várias combinações de campos e pacotes com especificações alteradas.

Da mesma forma, um **pacote** é uma parte de um ataque. Neste tipo de teste (de sintaxe), um ataque é constituído por um único pacote, mas poderão existir testes mais complexos que necessitem de mais pacotes, como os testes de estado [1]. O pacote é então enviado pelo `Injector de Pacotes`, concretizando assim a injecção de um ataque.

Injector

Observou-se na Figura 4.1 as entidades básicas envolvidas na injecção de ataques, nomeadamente o *Injector*. Os componentes fundamentais desta entidade estão representados na Figura 4.2. O principal componente desta entidade, o `Gestor de Testes`, que trabalha ao nível dos testes, irá iniciar e controlar todo o processo de injecção de ataques através dos restantes componentes. Acumula também o papel de analisar os resultados dos respectivos testes, através do subcomponente `Analisador de Testes`. Os tipos de teste, concretizados sob a forma de injecção de ataques, são dirigidos a vários aspectos do protocolo concretizado pelo *Componente-Alvo*.

O `Gerador de Ataques` é responsável por criar ataques para um dado tipo de teste. Este delega a respectiva injecção e posterior tratamento ao `Injector de Pacotes` e `Analisador de Ataques`.

É ao nível da injecção de pacotes que os ataques são sincronizados entre o *Injector* e o *Monitor*, através do componente `Controlador de Monitorização`.

Esta sincronização permite ao *Monitor* *reinicializar* as condições de teste antes de qualquer ataque. Isto é importante, de modo a que todos os testes sejam realizados sob idênticas condições e para que o resultado de um teste não seja influenciados pelos testes anteriores.

A injecção do ataque é concretizada no `Injector de Pacotes`, através do efectivo envio dos pacotes que constituem o ataque. Este componente permite que a transmissão dos pacotes seja realizada de forma transparente e totalmente independente de qual o protocolo de transporte utilizado: TCP ou UDP.

Os ataques são também registados pelo `Coleccionador de Pacotes`, nomeadamente o tipo de ataque e pacote enviado, para uma futura análise do ataque.

Monitor

Durante o ataque, o `Monitor`[1] observa o estado do `Componente-Alvo`, guardando informação sobre o seu funcionamento de forma permanente (em disco), através do `Coleccionador de Monitorização`.

Após a realização do ataque, os dados resultantes (de comunicação e de monitorização) são recolhidos e correlacionados pelo `Analisador de Ataques`, que assim poderá encontrar o par $<ataque,\ vulnerabilidade>$ que poderiam proporcionar ao atacante a intrusão, como foi exemplificado no modelo AVI (ver Figura 2.3).

Apesar da simples aparência do *Monitor*, esta é uma entidade fundamental, escondendo aspectos mais complexos do que aparenta. Por um lado, é esta entidade que terá de preparar o ambiente para os ataques no *Sistema-Alvo*: iniciar o `Componente-Alvo` (lançar a aplicação), começar a monitorização e no final, libertar os recursos utilizados. Por outro lado, é aqui que é feita a observação dos resultados de cada ataque, estando esta análise muito dependente dos mecanismos oferecidos pelo `Sistema Operativo` (e.g., em *Linux*, um processo pode ser monitorizado, interceptando os sinais que recebe). As vulnerabilidades de segurança que se poderão detectar, vão depender da informação que se obtém dos resultados dos ataques. Alguma desta informação é dada pelo próprio `Componente-Alvo`, como resposta aos ataques. Mas é o *Monitor* o responsável por obter e fornecer os restantes dados necessários no diagnóstico de vulnerabilidades — informação não divulgada pelo `Componente-Alvo`, proveniente da sua observação directa.

Geração de ataques

Apesar do principal resultado deste projecto ter sido a criação de um modelo de injecção de ataques e respectiva concretização, não se pode fugir ao facto de que talvez

[1] Repare-se que este `Monitor` diz respeito a um *componente* de monitorização que faz parte de uma entidade, mais abstracta e abrangente, também chamada aqui de *Monitor*.

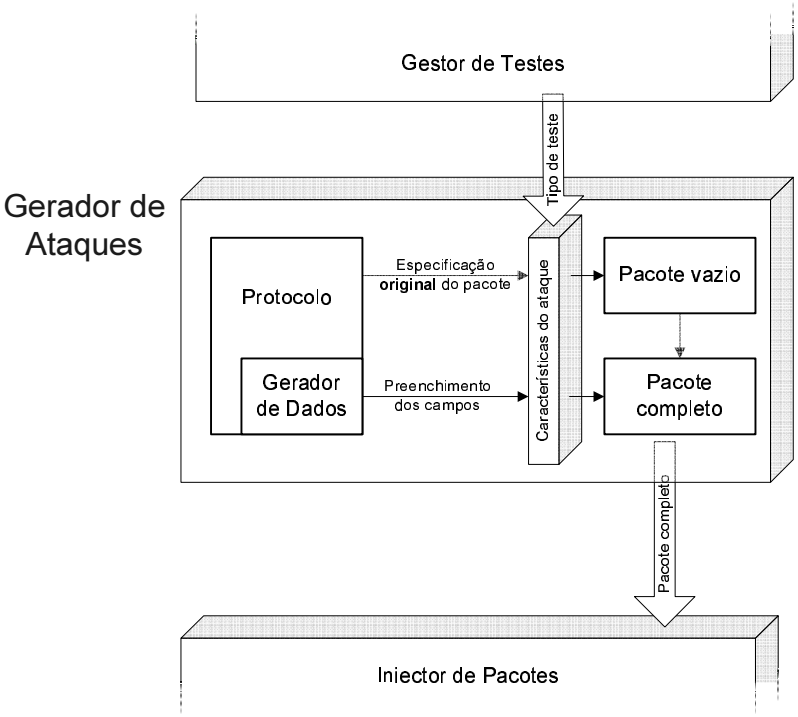


Figura 4.3: Gerador de Ataques.

o aspecto mais importante para a injecção de ataques, seja a própria especificação e criação dos ataques. Só assim é que se podem criar ataques efectivos que poderão descobrir vulnerabilidades que os métodos convencionais normalmente não encontram.

Na Figura 4.3 apresenta-se em maior detalhe o componente `Gerador de Ataques`. Este componente é responsável por criar os pacotes que serão injectados no `Componente-Alvo`.

O `Componente-Alvo` concretiza um determinado protocolo (por exemplo o POP, SMTP, HTTP, IRC, etc.), e será a correcta implementação deste protocolo que será diagnosticada contra vulnerabilidades. Existe assim, a necessidade de um componente que possua mecanismos próprios, que lhe permitam "conhecer" a especificação e tipos de dados dos pacotes deste protocolo. Este componente, chamado `Protocolo`, pode assim criar especificações de pacotes (`Pacote vazio`) e preencher os mesmos (`Pacote completo`), criando assim os pacotes que fazem parte de um determinado ataque.

Mas como o ataque está dependente do tipo de teste, tanto a especificação como o preenchimento dos campos serão controlados e filtrados de acordo com as características do ataque. A título de exemplo, se desejarmos criar um ataque que detecte vulnerabilidades de *buffer overflow*, basta que a especificação nos crie um pacote com um campo de tamanho superior à memória reservada pela aplicação-alvo, e preencher esse campo normalmente.

Estão criadas as condições para a construção de um ambiente de injecção de faltas maliciosas, composto pelo injector de ataques e por um sistema-alvo monitorizado.

4.1.2 Modularidade e Independência

O modelo atrás descrito reflecte a grande modularidade e independência presente no produto final. Repare-se que o *Injector* é totalmente independente do *Sistema-Alvo*, podendo estar a correr em plataformas e sistemas operativos diferentes, tendo apenas em comum o protocolo de transporte (TCP ou UDP) utilizado na comunicação entre ambos.

No entanto, não se poderá dizer o mesmo em relação a todo o *AJECT* (constituído pelo *Injector* e *Monitor* na Figura 4.1), pois a monitorização está intimamente dependente do *Sistema-Alvo*, ou mais precisamente do `Sistema Operativo` do mesmo.

Apesar de tudo, a modularidade alcançada permite que o *Monitor* seja apenas dependente do `Sistema Operativo`, não dependendo da aplicação-alvo (`Componente-Alvo`) ou do protocolo neste concretizado.

4.2 Concretização

A implementação reflecte também a modularidade e independência presentes nas fases de desenho. O que era inicialmente visto como sendo um projecto concretizado numa única linguagem, revelou-se vantajoso, após uma reflexão mais profunda, a escolha de duas linguagens de programação:

- Java para um *Injector* multi-plataforma e independente do sistema operativo;
- C++ para o *Monitor*, devido às funcionalidades de baixo nível oferecidas pela linguagem, útil à monitorização da aplicação-alvo.

A arquitectura do *Injector*, apresentada anteriormente, torna-o independente de qualquer plataforma, sistema operativo ou componente de monitorização. Assim, optou-se pela utilização da linguagem de programação Java na implementação do *Injector*, oferecendo assim uma independência de plataforma tanto na arquitectura, como na concretização.

Relativamente ao *Monitor*, a escolha da linguagem está directamente dependente do *Sistema-Alvo*, que por sua vez depende de qual o `Componente-Alvo` que se deseja testar. Assim, como o projecto foi desenvolvido em *Linux*, escolheu-se como `Componente-Alvo`, uma aplicação que também "corresse" neste ambiente. Na concretização dos componentes de monitorização, nomeadamente o `Monitor` e o `Colecci-onador de Monitorização`, resolveu-se utilizar a linguagem de programação C++, pois permite tirar proveito das funcionalidades próprias do sistema operativo, particularmente na criação e rastreio de processos.

As próximas secções mostram com maior detalhe, a concretização das principais entidades e componentes da ferramenta *AJECT*, bem como alguns aspectos relevantes como a injecção, monitorização e sincronização de ataques. Nestas secções serão

apresentados diagramas de classes e de fluxo UML e listagens de pseudo-código, que descrevem com maior pormenor *como* foi concretizada a ferramenta *AJECT*. Tenta-se também demonstrar a correlação quase perfeita entre o modelo e arquitectura criados e a respectiva concretização, onde as classes (retiradas directamente do código fonte) tentam traduzir os componentes das secções anteriores.

4.2.1 Ferramentas Utilizadas

A escolha e tipo de ferramentas utilizadas no processo de desenvolvimento de *software* são bastante importantes na engenharia de *software*. Estas podem determinar a criação de um produto com maior ou menor qualidade.

As ferramentas mais importantes, utilizadas durante o estágio podem ser divididas nas seguintes categorias:

- programação;
- documentação;
- experimentação.

Programação

Este tipo de ferramentas permite auxiliar o processo de codificação de *software*. Foi usado um ambiente de desenvolvimento integrado (IDE), que possibilita a edição de código fonte, compilação, execução e teste (*debug*). Foram estudadas várias soluções, sendo no final escolhido o IDE *Eclipse*[2], tanto para a linguagem Java (*plugin* JDT) como para C++ (*plugin* CDT). De entre as não escolhidas encontravam-se o *Anjuta*[3], o *GNU Emacs*[4] com o módulo *Xrefactory* e o *NetBeans*[5].

Os testes foram efectuados com o auxílio do *debugger gdb*, quer pelas interfaces gráficas do *xgdb*, quer pelo IDE *Eclipse*.

Documentação

Foi utilizada a aplicação de gestão de projectos *Imendio Planner*[6] na calendarização do estágio e para a realização dos diagramas de Gantt no Anexo C.

A documentação escrita, nomeadamente na redacção de relatórios, foi realizada em *LaTeX*[7], pelo editor *GNU emacs* com o módulo *AUCTeX*. As figuras e diagramas foram criados pelo *Microsoft Visio 2003*[8].

[2] `http://www.eclipse.org/`
[3] `http://anjuta.sourceforge.net/`
[4] `http://www.gnu.org/software/emacs/`
[5] `http://www.netbeans.org/`
[6] `http://developer.imendio.com/wiki/Planner/`
[7] `http://www.latex-project.org/`
[8] `http://office.microsoft.com/`

Foi utilizado o *JavaDoc* na a linguagem Java e o *Doxygen*[9] na a linguagem C++, para a geração automática de documentação de código fonte. Ambas as ferramentas permitem a criação de páginas HTML com a API das classes e métodos criados.

Experimentação

Foram utilizadas ferramentas normalmente existentes num ambiente *Linux*, nomeadamente o *Multi-Gnome-Terminal*[10]. Este terminal permite a divisão (horizontal ou vertical) em mais terminais virtuais, o que possibilita o acesso simultâneo a vários terminais, numa única instância.

4.2.2 Injector, Monitor e Sistema-Alvo

Mostra-se agora a concretização das entidades básicas do modelo de injecção de ataques, nomeadamente do *Injector* e do *Monitor*. Descrevem-se também outros aspectos da implementação como a comunicação, tipos de dados, monitorização e injecção de ataques.

Injector

Esta é a entidade principal do *AJECT*, sendo responsável por todos os aspectos da injecção de ataques à excepção da monitorização e rastreio do `Componente-Alvo`. Na Figura 4.4 está representado o diagrama de classes UML desta entidade. Pormenores não relevantes foram omitidos para simplificação e uma melhor compreensão.

Repare-se que a classe central do *Injector* é a classe `Test`, que concretiza o componente `Gestor de Testes`, cujos principais métodos são: `run()` e `generate()`. O primeiro é responsável por executar o teste, enquanto o segundo fica a cargo de gerar cada um dos ataques a serem injectados durante a execução do teste.

Cada tipo de teste não é mais que uma especialização da classe `Test`, nomeadamente as classes `TestSyntax` e `TestValue`. A primeira classe avalia a sintaxe do protocolo (especificação dos pacotes e respectivos campos), enquanto que a segunda verifica os valores possíveis dos campos que compõem um dado pacote.

Um teste necessita de três elementos fundamentais: uma aplicação (1) que implementa um determinado protocolo (2) que queremos testar, e um monitor (3) para observar o resultado do teste. Assim, directamente ligadas à classe `Test` existem três relações, uma para cada um dos elementos mencionados.

Como o `Injector` tem de "conhecer" o protocolo a avaliar, este é representado aqui pela classe `Protocol`. Tal como a classe `Test`, também esta é uma classe genérica, sendo necessário uma classe "especializada" no protocolo a testar (classe `POP`). A especialização está dependente do protocolo que o `Componente-Alvo` implementa

[9] `http://www.doxygen.org/`
[10] `http://multignometerm.sourceforge.net/`

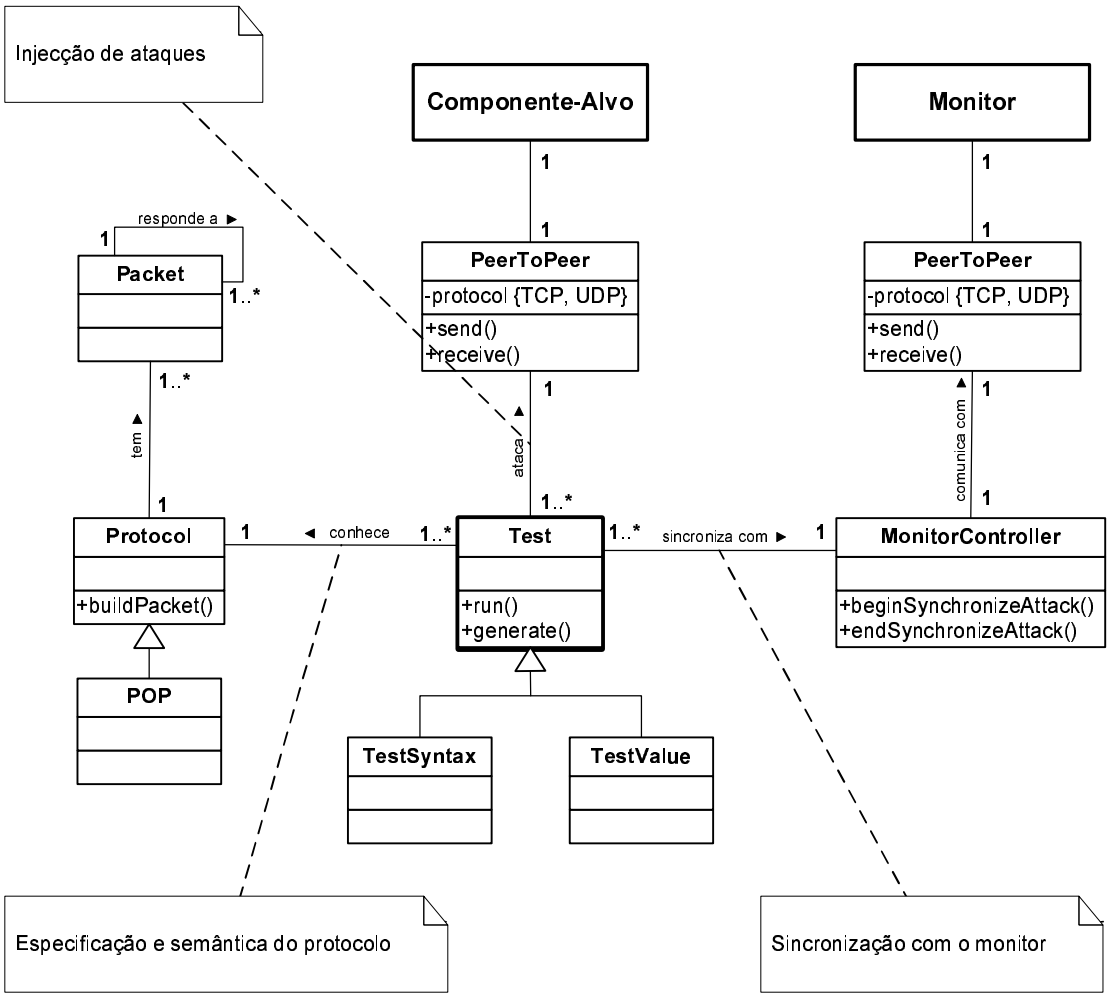


Figura 4.4: Diagrama de classes do *Injector*.

e que se deseja testar, sendo que a classe `Protocol` se mantém genérica o suficiente para qualquer protocolo. Esta classe tem acesso a um conjunto de informação (classe `Packet`) que define a especificação dos pacotes e respectivos campos, bem como as precedências dos pacotes do protocolo (pedido/resposta).

Toda a comunicação é feita através da classe `PeerToPeer`, que modela um canal de comunicação entre dois pontos. Esta classe fornece primitivas de comunicação, como o `send()` e `receive()`, que permite o envio e a recepção de pacotes.

O *Injector* utiliza duas instâncias da classe `PeerToPeer`: uma para a comunicação com o `Componente-Alvo` (injecção de ataque) e outra para a sincronização com o *Monitor*.

A sincronização com o *Monitor* é controlada pela classe `MonitorController`, que concretiza o componente `Controlador de Monitorização`. Os método `beginSynchronizationAttack()` e `endSynchronizationAttack()` são invocados para notificar o *Monitor* do início e fim de cada ataque.

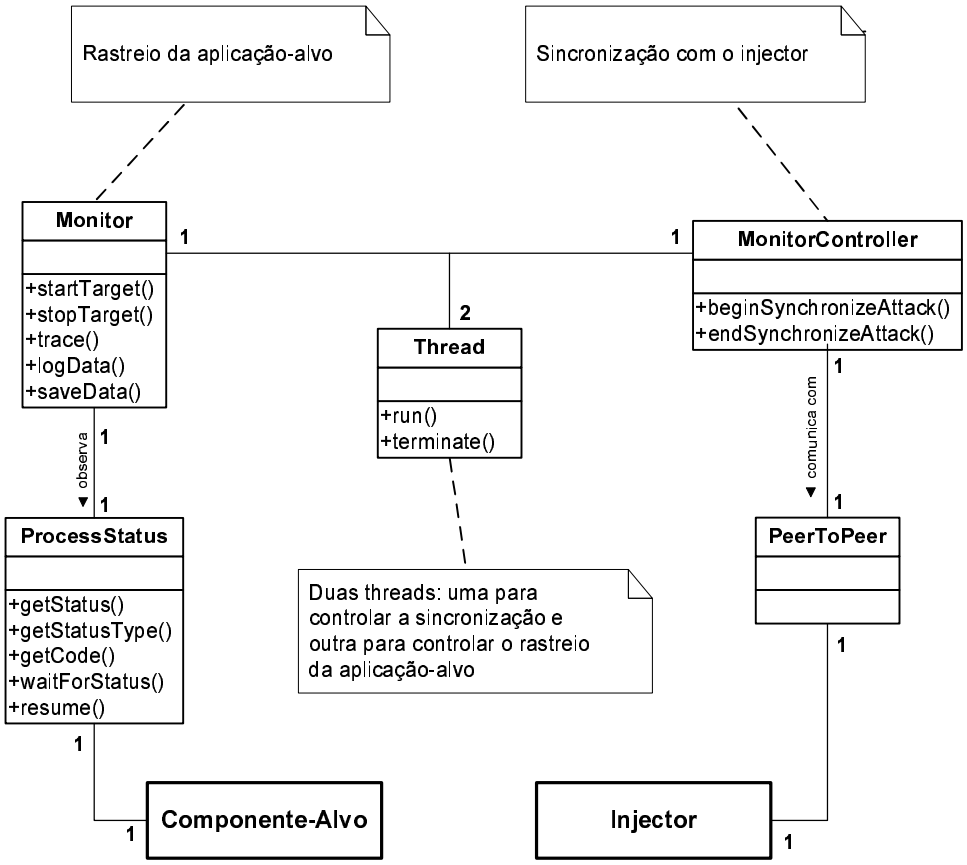


Figura 4.5: Diagrama de classes do `Monitor`.

Monitor

O *Monitor*, cujo papel é o de observar e registar os resultados da injecção de ataques, tem uma outra responsabilidade igualmente importante: a re-inicialização das condições de teste antes de cada ataque. Como se irão efectuar vários testes, torna-se imperativo que as condições do ambiente em que se realizam os testes sejam sempre idênticas.

A Figura 4.5 mostra o diagrama de classes que compõe o *Monitor*. Daqui pode-se observar que existem três classes centrais directamente relacionadas: `Monitor`, `MonitorController` e `Thread`. Veja-se a Figura 4.2 onde mostram os seus homólogos na fase de modelação: os componentes `Monitor` e `Controlador de Monitorização`.

Existe portanto, uma classe que trata da observação e registo dos ataques dirigidos ao `Componente-Alvo` — classe `Monitor` —, e uma outra que irá rastrear o `Componente-Alvo` — classe `ProcessStatus`.

Como a monitorização terá de ser coordenada de alguma forma com o `Injector`, existe também um classe responsável por essa sincronização — classe `MonitorController`. Tal como no *Injector*, esta classe implementa o componente `Controlador de Monitorização`, concretizando o protocolo de sincronização criado para o efeito

(como se verá mais à frente na Secção 4.2.4).

Como se pode observar, o *Monitor* terá então dois papéis concorrentes, o de **monitorização** e o de **sincronização**. A existência da classe `Thread` permite que ambos os papéis sejam desempenhados em paralelo, havendo uma instância para cada um destes. A Figura 4.6 mostra o diagrama de fluxo do *Monitor*. Nesta imagem pode-se ver os estados internos do *Monitor* e os diferentes fios de execução concorrentes: o processo-pai (linhas a negrito) e duas *threads* (para sincronização e monitorização, por esta ordem).

Como se pode verificar, no início existe o processo-pai, que lança uma *thread* que irá tratar da comunicação com o `Injector`, a ***thread* de sincronização**. Depois entra no ciclo principal de **espera de sinalização**. Daqui só pode sair quando uma das duas *threads* sinalizar um evento (que pode ser início de ataque, fim de ataque ou fim de rastreio).

Neste momento, o processo-pai só pode sair da espera quando a *thread* de sincronização receber uma mensagem do *Injector* de início de ataque. O processo-pai sai então do ciclo para criar uma ***thread* de rastreio**, que irá por sua vez criar um processo-filho.

Existem neste momento quatro fios de execução dentro do `Monitor`: o processo-pai, a *thread* de sincronização, a *thread* de rastreio e o processo-filho desta última.

O processo-filho criado pela *thread* de rastreio tem como propósito executar a aplicação-alvo (`Componente-Alvo`). Isto é necessário pois o método de monitorização utilizado implica a interceptação de sinais, só possível através da relação processo-pai e processo-filho. A Secção 4.2.5 mostra os detalhes da monitorização levada a cabo pela *thread* de rastreio.

O processo-pai volta então para a espera de sinalização. A monitorização pode terminar devido a uma notificação do `Injector` através da *thread* de sincronização (fim de ataque) ou devido ao facto da aplicação-alvo não poder continuar a sua execução (e.g., teve uma paragem). Em qualquer dos casos, o processo-pai recebe a notificação, terminando de seguida a aplicação-alvo e a *thread* de rastreio.

Sistema-Alvo

Como foi dito na Secção 4.1.1, o *Sistema-Alvo* é composto por todo o ambiente computacional que envolve a plataforma de *hardware*, sistema operativo e a própria aplicação que se deseja testar (aplicação-alvo).

O objecto do teste não será todo o *Sistema-Alvo*, mas apenas o `Componente-Alvo`, concretizado pela aplicação-alvo. Apesar disso, tanto a plataforma como o sistema operativo são importantes, pois a aplicação-alvo realiza o seu propósito através das funcionalidades oferecidas pelo *Sistema-Alvo*, estando portanto intimamente ligada à forma como este está construído.

Esta relação de dependência está também associada ao facto da monitorização do

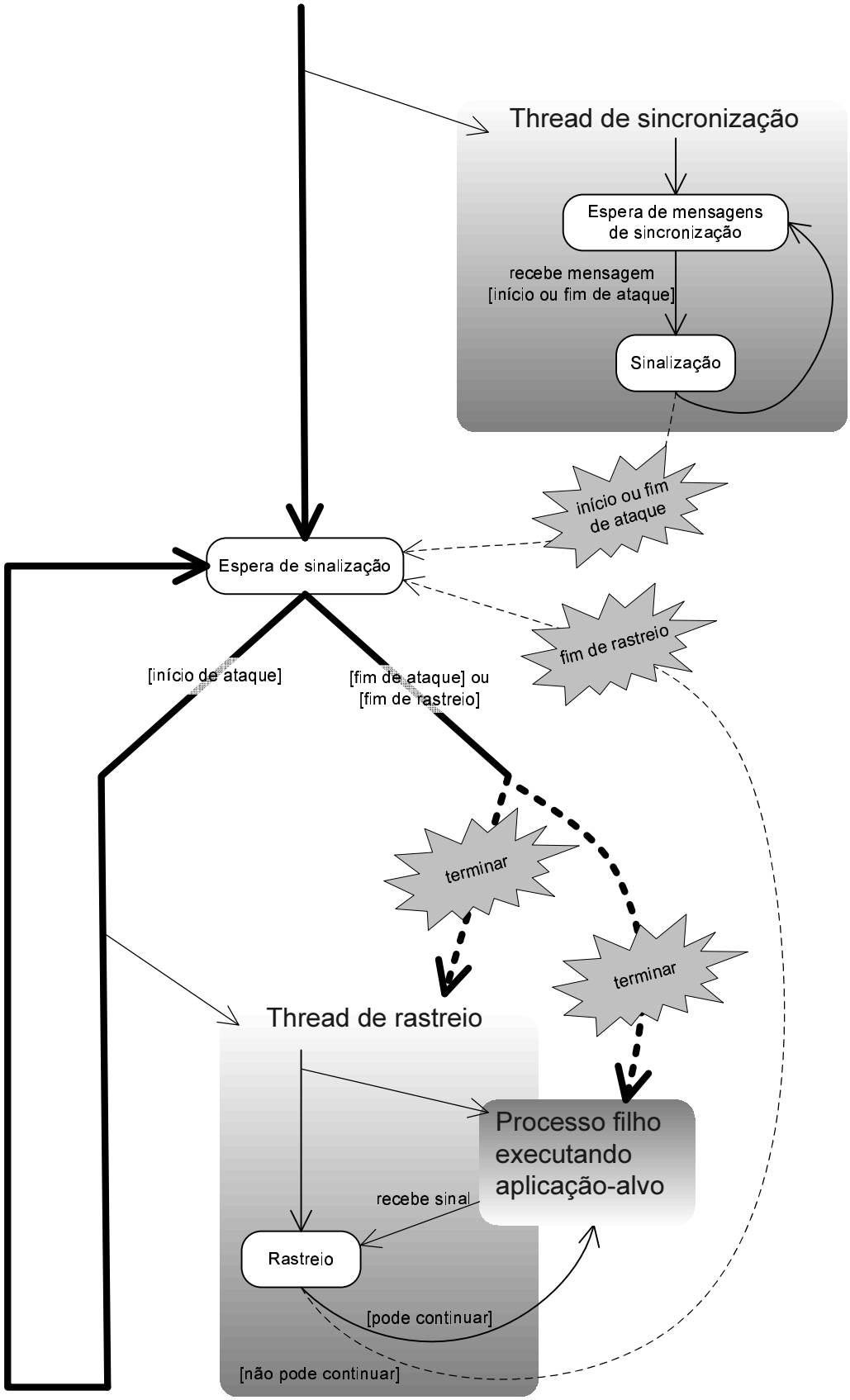


Figura 4.6: Diagrama de fluxo do `Monitor`.

AJECT estar o mais próxima possível da aplicação-alvo, estando igualmente dependente de todo o *Sistema-Alvo*. O *Sistema-Alvo* é então um aspecto muito importante, pois sustém a monitorização e, mais importante, a própria aplicação-alvo.

Como foi visto na Secção 2.4, existe um tipo específico de aplicações propício à injecção de ataques. Uma aplicação que não tenha qualquer comunicação com a rede, está de certa forma, algo protegida contra ataques do exterior.[11] Por isso, as aplicações de rede, como os servidores *web* ou de correio electrónico, são as aplicações candidatas mais "interessantes" ao diagnóstico de vulnerabilidades.

4.2.3 Injecção de Ataques

Foram anteriormente descritas as entidades do *AJECT*, os componentes que o constituem, os meios de comunicação e representação de pacotes e de protocolos, e os métodos de monitorização e sincronização de ataques. Falta agora explicar o processo de criação de um ataque e a injecção do mesmo.

Na arquitectura construída (ver Figura 4.2), observa-se que o componente `Injector de Pacotes` depende do componente imediatamente acima, o componente `Gerador de Ataques`. Esta relação é também visível na implementação, como se pode verificar pela Listagem 4.1, que descreve a execução de um teste — mais concretamente, a concretização do método `run` na classe `Test` da Figura 4.4.

Tanto na Figura 4.4 como na listagem, se podem observar as referências aos seguintes componentes/classes:

- `Controlador de Monitorização` (classe `MonitorController`, variável `controlador`);
- aplicação-alvo (classe `PeerToPeer`, variável `alvo`);
- protocolo-alvo (classe `Protocol`, variável `Protocolo`).

Um ataque poderia, em teoria, gerar vários pacotes. Mas como não foram criados testes que envolvessem o envio de mais de um pacote (testes mais complexos e que, por exemplo, levassem a aplicação-alvo a diferentes estados), observa-se que na prática, cada ataque gera apenas um pacote (`p`).

O protocolo-alvo é constituído por um conjunto de mensagens, que serão utilizadas na geração de ataques. Estas mensagens correspondem a pacotes que especificam a forma de comunicar do protocolo. A partir de cada uma dessas mensagens, é (internamente) criada uma lista de ataques/pacotes (`inicializarAtaques()`). Essa lista de ataques é acedida internamente pelo método `geraNovoAtaque()` e de forma iterativa, i.e., cada acesso vai devolver um ataque/pacote.

[11] Não está totalmente segura, mas está bastante menos exposta e vulnerável a ataques que aplicações de rede.

```
controlador ← controlador de monitorização
alvo ← ligação ponto-a-ponto com a aplicação-alvo
Protocolo ← conjunto de pacotes do protocolo da aplicação-alvo


/*
 * A partir de cada pacote do protocolo,
 * constróiem-se vários pacotes de teste a enviar.
 */
para cada pacote ∈ Protocolo
.
.    // inicializa lista de ataques a partir do pacote
.    inicializarAtaques (pacote)
.
.
.    // para cada ataque
.    enquanto (p ← geraNovoAtaque()) ≠ vazio
.    .
.    .    /*
.    .     * Início do ataque: sincronização com o Monitor e abertura da
.    .     * ligação com a aplicação-alvo.
.    .     */
.    .    controlador.inicioSincronização()
.    .    alvo.abrirLigação()
.    .    alvo.ping()
.    .
.    .    /*
.    .    * Teste à aplicação-alvo até esta estar pronta (ping) e
.    .    * efectiva injecção do ataque.
.    .     */
.    .    alvo.envia(p)
.    .    sleep(1500)
.    .
.    .    /*
.    .     * Fim do ataque: sincronização com o Monitor, fecho da ligação
.    .     * com a aplicação-alvo e espera algum tempo para que os recursos
.    .     * sejam libertados do outro lado antes de iniciar um novo ataque
.    .     */
.    .    controlador.fimSincronização()
.    .    alvo.fecharLigação()
.    .    sleep(1500)
.    .
.    .
.
```

Listagem 4.1: Geração de ataques.

Agora que se tem um ataque gerado, é necessário proceder com a respectiva injecção. A injecção de um ataque é efectuada em três fases:

- início de ataque;
- injecção de ataque;
- fim de ataque.

O **início de ataque** vai preparar, tanto o *Injector* como o *Monitor*, para o ataque. Começa-se pela sincronização com o *Monitor*, que se verá mais adiante na Secção 4.2.4, passando depois pela abertura de ligação com o alvo (que vai estabelecer o canal de comunicação TCP ou UDP).

No entanto, a aplicação-alvo também necessita de estar preparada para o ataque — a aplicação tem de estar num determinado estado, que lhe permita ficar à espera de pedidos/ataques do utilizador. Qualquer aplicação deste tipo, após ser iniciada, efectua um série de cálculos e computações, passando por vários estados, até que por fim, chega ao estado "normal" de espera. Sendo lançada pelo *Monitor*, este não tem meio de saber quando é que a aplicação-alvo chegou a esse estado. Será o *Injector* que terá de ir contactando a aplicação-alvo, até conseguir obter resposta. Isto é feito através do envio de um pacote (do qual já se sabe a resposta que a aplicação-alvo irá enviar), periodicamente até se obter a resposta (`alvo.ping()`).

Só nesse momento é que o injector, o monitor e a aplicação, estão preparados a **injecção de ataque**, que é realizada através do envio do respectivo pacote gerado (`alvo.envia(p)`). Cada ataque irá obrigar a aplicação-alvo a transitar por uma série de estados, levando-a possivelmente a estados "incorrectos"[12]. Contudo, os efeitos do ataque podem só se fazer sentir após algum tempo, daí a espera (`sleep(1500)` no injector.

Por fim, chega-se ao **fim de ataque** onde, num esforço coordenado (sincronização), *Injector* e *Monitor*, removem os efeitos do ataque do ambiente. O *Injector* vai fechar a ligação com a aplicação-alvo e esperar algum tempo antes de prosseguir para o próximo teste. Esta espera é necessária devido à gestão das *sockets* pelo sistema operativo, principalmente quando utiliza o protocolo de transporte TCP. Também o *Monitor* terá de reinicializar as condições de teste, como será descrito mais à frente nas Secções 4.2.4 e 4.2.5 sobre a sincronização e monitorização, respectivamente.

Todo este processo é repetido para cada ataque gerado, até que por fim, se esgotem os ataques daquele teste em particular.

[12] O conceito de estado "incorrecto" depende do teste e da sua análise, mas pode ser um simples falha por paragem (*crash.*)

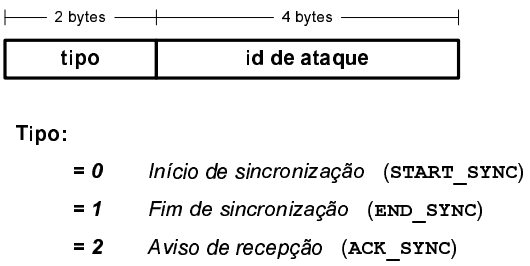


Figura 4.7: Especificação e tipos de mensagem do protocolo de sincronização.

4.2.4 Sincronização de Ataques

Como foi dito anteriormente, um ataque é realizado pelo *Injector* e os seus resultados observados pelo *Monitor*. Esta coordenação de esforços terá de ser sincronizada de alguma forma. Para este efeito foi criado um protocolo de sincronização.

O protocolo de sincronização envolve três tipos de mensagens diferentes (como representado na Figura 4.7):

- a mensagem que inicia a sincronização de um ataque (`START_SYNC`),
- a mensagem que termina a sincronização de um ataque (`END_SYNC`) e
- uma mensagem de aviso de recepção de mensagens (`ACK_SYNC`).

Todos estas mensagens têm em comum, a identificação do ataque a sincronizar.

A Figura 4.8 representa a interacção entre o *Injector* e o *Monitor*, utilizando este protocolo. De notar que o componente do *Monitor* responsável por esta sincronização é o `Controlador de Monitorização`, concretizado na classe `MonitorController`.

Como se pode observar pela figura, o *Injector* inicia o processo de sincronização enviando uma mensagem `START_SYNC` para o *Monitor*. A *thread* de sincronização do *Monitor* irá receber estas mensagens, e depois o processo-pai irá criar uma *thread* de rastreio para lançar e monitorizar a aplicação-alvo.

Quando o *Monitor*, ou mais concretamente a *thread* de rastreio, estiver preparada para a monitorização, irá responder ao *Injector* com uma mensagem `ACK_SYNC`. Neste ponto, o *Injector* irá proceder com o ataque, que quando finalizado, enviará uma mensagem `END_SYNC` ao *Monitor*. Isto permite ao *Monitor*, saber que o ataque terminou, podendo então parar a monitorização. O *Monitor* envia um `ACK_SYNC` ao *Injector*, indicando que terminou a monitorização e que está pronto para que se inicie o próximo ataque.

4.2.5 Monitorização de Ataques

Na secção anterior descreveu-se como está concretizado o *Monitor*, sem no entanto se entrar em detalhes, relativamente a como é feita exactamente essa monitorização.

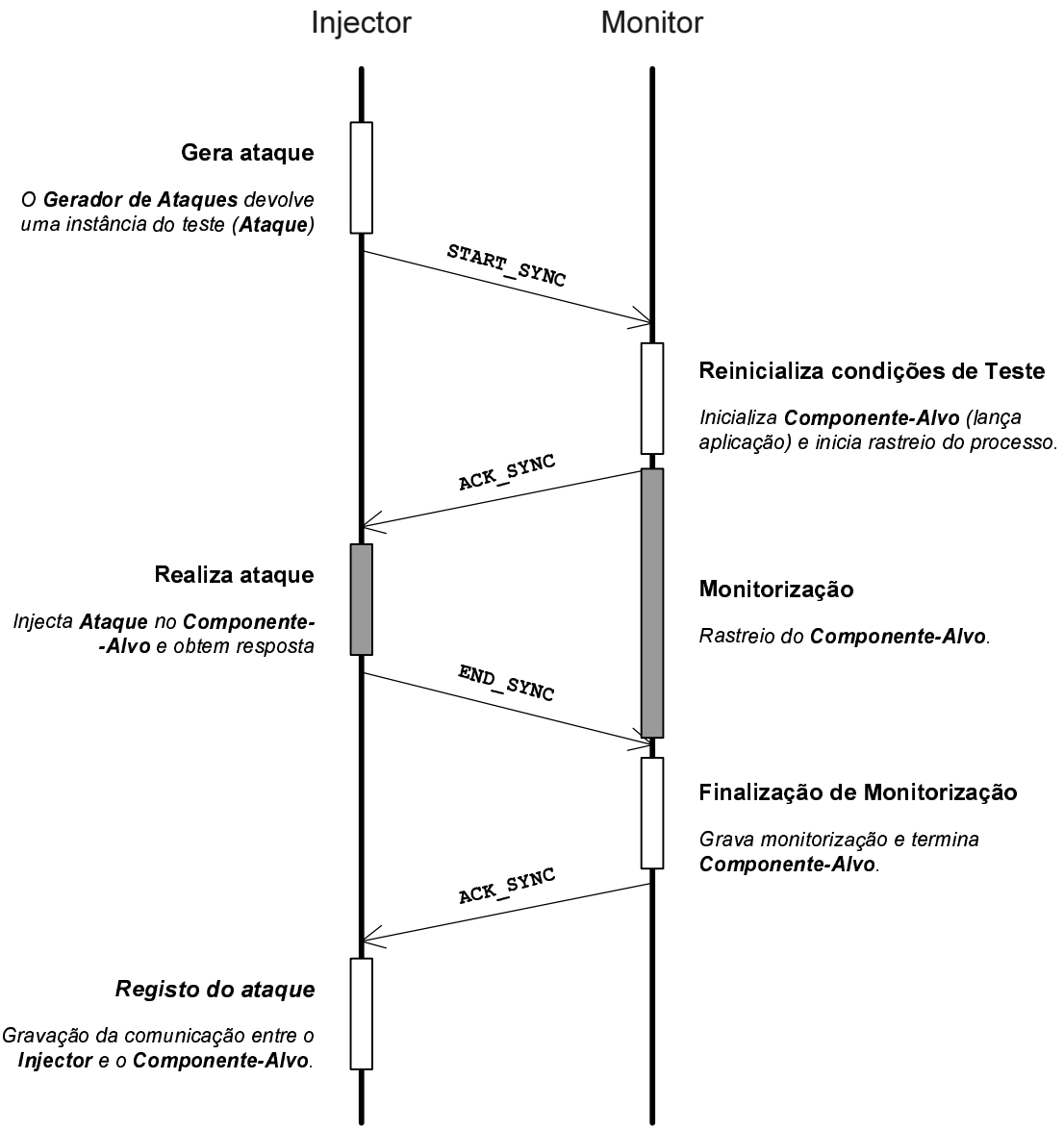


Figura 4.8: Protocolo de sincronização entre o *Injector* e o *Monitor*.

Relembrando a Figura 4.6, que mostra o fluxo de execução do *Monitor*, chama-se a atenção para a ***thread* de rastreio**, pois é aqui que é efectuado o rastreio ao `Componente-Alvo`. O método aqui utilizado é semelhante ao usado pelos *debuggers*, como se verá mais à frente.

A *thread* de rastreio começa por criar um processo-filho, cujo propósito é executar a aplicação-alvo (`Componente-Alvo`). Este passo é necessário porque o método de monitorização utilizado implica a interceptação de sinais (métodos da família `ptrace` e `waitpid`), como se pode verificar pelo pseudo-código da Listagem 4.2. Este método implica que o fluxo de execução de um processo só pode ser monitorizado e/ou controlado pelo respectivo processo-pai — daí a necessidade da *thread* de rastreio criar um processo-filho para executar a aplicação-alvo.

```
ficheiro ← nome do executável da aplicação-alvo
argumentos ← argumentos do executável da aplicação-alvo


/*
 * Criação do processo-filho.
 * Código executado pelos dois processos (pai e filho).
 */
pid ← fork() // pid fica com identificação do processo-filho


se processo-filho
.
.    // dá permissão ao pai para o controlar
.    ptrace(PTRACE_TRACEME)
.
.    // o código do processo-filho é subsituído pelo da aplicação-alvo
.    execv(ficheiro, argumentos)
.


se processo-pai
.
.    // Monitoriza o processo-filho,
.    // enquanto este não recebe um sinal terminal
.    fazer
.    .
.    .    // esperar que processo-filho receba um sinal e regista-o
.    .    sinal ← waitpid(pid)
.    .    registarSinal (sinal)
.    .
.    .    // deixar a aplicação-alvo prosseguir o sinal o permita
.    .    se sinal não é terminal
.    .    .    ptrace(PTRACE_CONT, pid)
.    .    senao
.    .    .    ptrace(PTRACE_KILL, pid)
.    .
.    enquanto sinal não é terminal
.
```

Listagem 4.2: *Thread* de rastreio.

O processo-filho (aplicação-alvo) é interrompido a cada sinal que recebe, esperando que o seu processo-pai, responsável pelo seu rastreio (`PTRACE_TRACEME`), o instrua a prosseguir. Os *debuggers* utilizam este método para interromper a execução do programa a cada instrução/*breakpoint*/sinal.

O método `waitpid` permite à *thread* de rastreio esperar que o seu processo-filho (`pid`[13] da aplicação-alvo) receba algum sinal. O rastreio é efectuado através da interpretação destes sinais. Enquanto a aplicação-alvo não receber nenhum sinal terminal (por exemplo *segmentation fault*) esta pode prosseguir (`PTRACE_CONT`); senão o rastreio e o processo-filho são terminados (`PTRACE_KILL`).

4.2.6 Comunicação

A aplicação-alvo concretiza a implementação de um protocolo (que se deseja testar) que, por sua vez usará qualquer um dos protocolos de transporte: TCP ou UDP. Foi por isso necessário, concretizar a abstracção de uma ligação ponto-a-ponto ao nível da camada de transporte, i.e. um meio de comunicação idêntico para os protocolos TCP e UDP.

Esta abstracção de comunicação está implementada pela classe `PeerToPeer`. Esta classe é um *wrapper* de primitivas de comunicação que esconde certos detalhes, como a forma de endereçamento (32 *bits* ou nome da máquina) ou mesmo o protocolo de transporte subjacente (TCP ou UDP).

São fornecidas várias funcionalidades como: abertura de ligações activas (remotas, utilizado por clientes) ou passivas (locais, utilizadas por servidores) e fecho das mesmas; envio simples ou com temporizador e repetições; e recepção bloqueante ou não bloqueante.

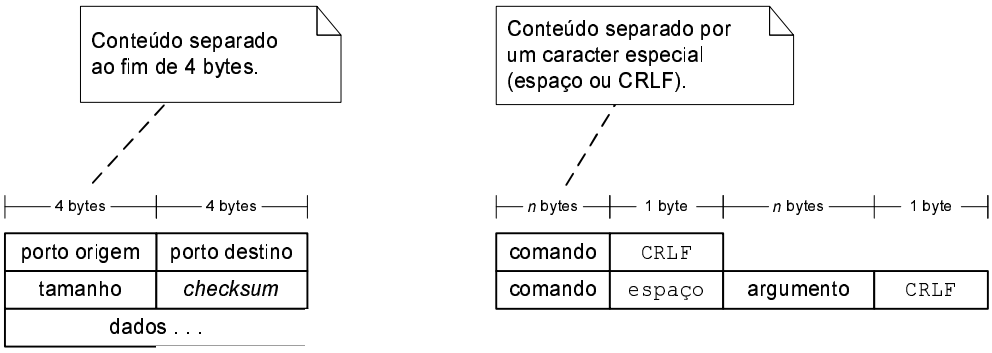
4.2.7 Pacotes e Tipos de Dados

Um aspecto que se revelou um grande desafio, foi o de manter a generalidade dos tipos de dados suportados, na representação de um pacote. Um pacote é um elemento abstracto que representa um conjunto de *bits* com determinado significado, que depende de uma determinada estrutura conhecida *à priori*. Essa estrutura divide esse conjunto de *bits* em grupos lógicos (campos).

A dificuldade na representação de um pacote não vem pela sua constituição abstracta, mas sim pelo seu efectivo conteúdo e respectiva interpretação. Repare-se que um campo tanto pode representar um identificador de 32 *bits*, como pode ser uma *string* codificada em UTF-8 separada por um espaço. A designação de um campo pode adquirir então, uma elevada complexidade.

A título de exemplo, veja-se a Figura 4.9 que mostra a especificação de mensagens de dois tipos de protocolos diferentes. A especificação de um datagrama UDP (à

[13] Variável que guarda a identificação do processo (*process identification*).



UDP – User Datagram Protocol *POP – Post Office Protocol*

Figura 4.9: Disparidade de especificação de diferentes protocolos.

esquerda), com campos de tamanho fixo de 4 bytes cada, é bastante diferente da especificação do protocolo POP (à direita), onde o tamanho dos campos depende do seu conteúdo.

Existe portanto, um universo de protocolos que se gostariam de testar, mas que adicionam uma grande disparidade de especificações. Assim, como é necessário criar meios para a "compreensão" dos protocolos, de modo a ser possível testá-los, ao alargar o leque de protocolos possíveis (por vezes com diferentes naturezas de especificação), está-se consequentemente a aumentar a complexidade da representação desses protocolos.

Como a constituição desses protocolos passa necessariamente pela especificação das mensagens que os compõem, torna-se imperativa uma representação suficientemente genérica, de modo a poder corresponder a qualquer tipo de mensagem de um qualquer protocolo. A forma encontrada para fazer face a este problema foi representar cada campo como um elemento do pacote. Elemento este, que representa um conjunto finito de *bits* (dados e respectivo tamanho). A informação sobre o tipo de dados do campo está delegada noutras classes. Assim, ao não se especificar aqui a natureza do tipo de dados, mantém-se a generalidade pretendida.

Na Figura 4.10 pode-se observar a constituição de um pacote. A classe `Element` descreve a abstracção representada por um conjunto de *bits*, que tanto pode ser um pacote (`Packet`) como um campo (`Field`). Um pacote é assim constituído por campos e a representação destes está delegada para a classe `DataValues` (responsável por definir o tipo de dados, bem como conjunto de dados válidos e inválidos). Pode-se observar pela Figura 4.11, que a representação dos tipos de dados é realizada por intermédio das classes `DataValues` e `Datatype`.

Temos assim uma classe que normaliza a especificação do tipo de dados — qualquer classe que extenda a classe `DataTypes` representa um tipo específico de dados. Estas classes especificam também regras sobre o seu conteúdo, como por exemplo: um campo

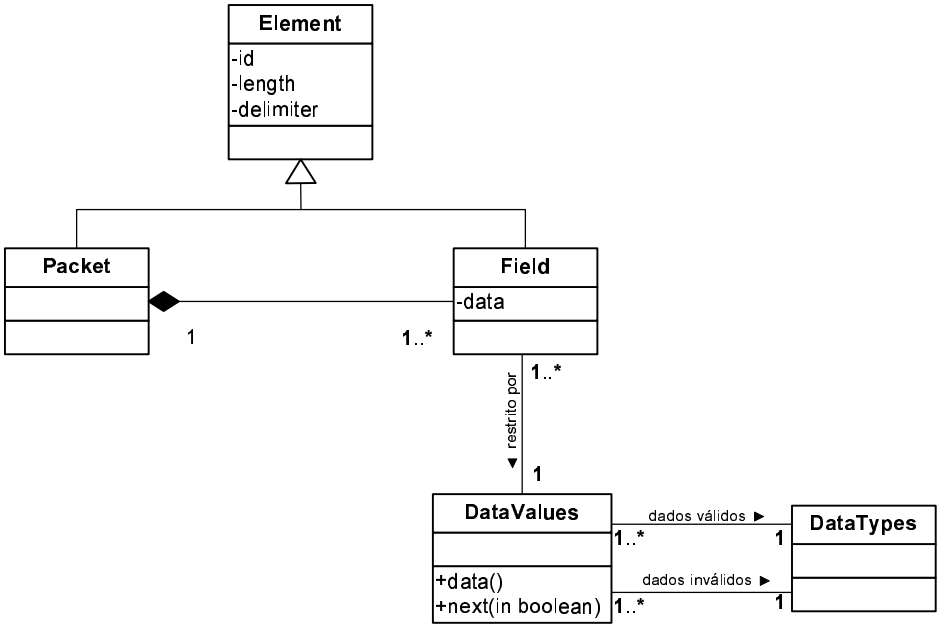


Figura 4.10: Diagrama de classes da representação de pacotes e campos.

do tipo `NumberDataType` pode só conter números entre 0 e 1000; ou um campo do tipo `ASCIISelectiveType` apenas pode ter os valores de "USER" e "PASS".

Estas classes implementam também um gerador interno de dados que obedece às regras de valor impostas (dados válidos e não válidos). Isto é útil, por exemplo, para testes de valor onde se pretende injectar pacotes com campos de valores válido e/ou não válido.

Resumindo: existem classes que representam pacotes e campos, e classes que definem tipos de dados. Quem define o tipo de dados deve também providenciar meios de manipulação de dados, úteis na geração de ataques.

4.2.8 Análise de Detecção de Vulnerabilidades

A detecção de erros é realizada através da exposição da aplicação-alvo num ambiente controlado, onde lhe serão enviados vários pacotes maliciosos (criados com o propósito de testar vários aspectos — sintaxe do protocolo, valor dos campos, etc.). O estado da aplicação é observado e analisado, durante e após o processamento desses pacotes, através das seguintes entidades:

- o *Injector*, pelos pacotes que recebe da aplicação-alvo, que correspondem a respostas aos pacotes enviados no ataque;
- o *Monitor*, através dos dados fornecidos pelo rastreio do processo correspondente à aplicação-alvo.

O *Injector* terá posteriormente que correlacionar esta informação. No entanto, apesar de terem sido criados mecanismos de detecção, a actual concretização deste

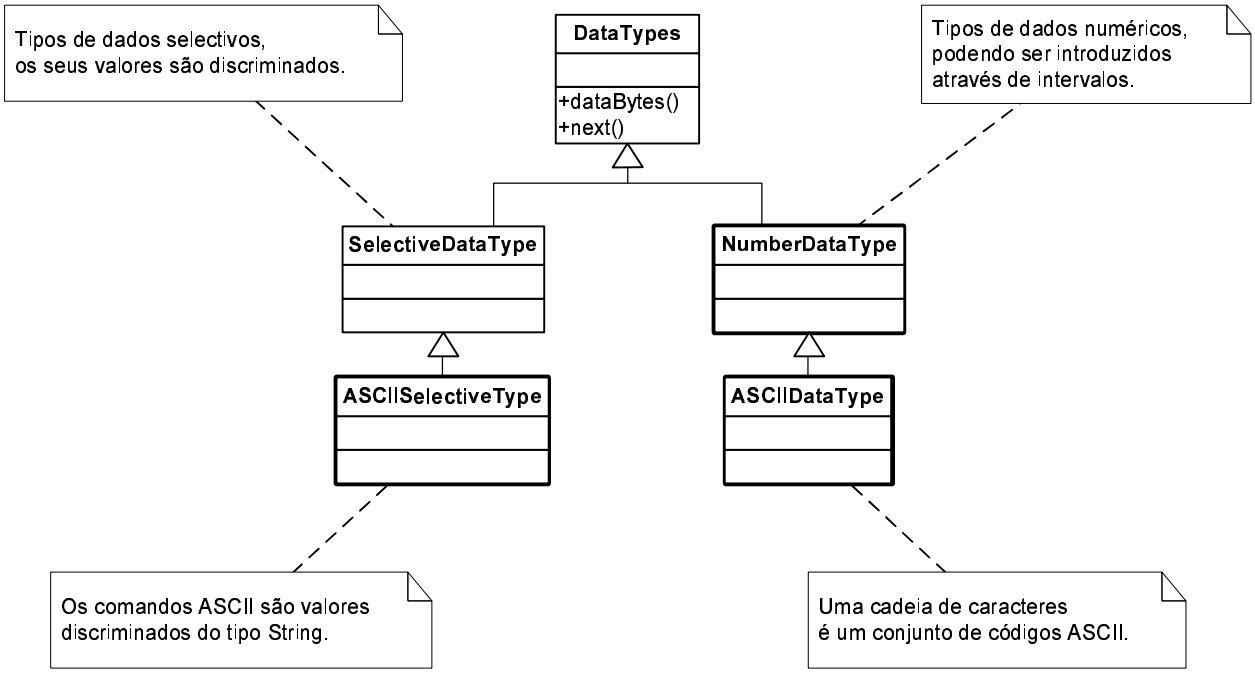


Figura 4.11: Diagrama de classes da representação do tipo de dados de um campo.

aspecto ainda não permite análise de vulnerabilidades automatizada. A informação de diagnóstico é registada, mas não existe nenhum processamento posterior destes dados. Será o utilizador da ferramenta *AJECT* que terá que, *manualmente*, verificar os *dados de monitorização*, observando por exemplo, se a aplicação-alvo recebeu algum sinal "estranho" (como um *segmentation fault*); ou mesmo as *respostas da aplicação-alvo*, por exemplo detectando se esta, permitiu indevidamente o acesso a algum recurso (como o acesso a *e-mails* de outro utilizador).

Repare-se que esta análise está também ligada ao tipo de testes realizados. Testes complexos deverão necessitar de análises igualmente complexas. Um teste que valide mecanismos de autorização de uma aplicação, envolverá a transição entre diferentes estados através do envio de vários pacotes e recepção das respectivas respostas.

No entanto, para os testes actualmente criados a análise pode ser tão simples como a observação dos sinais que a aplicação recebeu durante os ataques, ficando remetido como trabalho futuro, a criação de novos testes, bem como a automatização da análise de vulnerabilidades.

4.3 Resultados Obtidos

4.3.1 Tipos de Testes

Neste projecto foram criados dois tipos de teste: teste de sintaxe e teste de valor. A ferramenta desenvolvida é independente dos testes criados, havendo assim a pos-

sibilidade de se construirem mais testes diferentes, que geram ataques com maior ou menor complexidade.

Teste de sintaxe

Este tipo de testes cria ataques que violem a especificação do protocolo, relativamente à adição e remoção de elementos (campos) que o constituem.

Como foi dito na Secção 4.2.7, onde se fala sobre os pacotes e tipos de dados, um pacote corresponde a uma sequência de campos. Um pacote constituído por três campos diferentes seria representado por:

```
[A] [B] [C]
```

Considerando um protocolo com um único tipo de mensagens, sendo a mensagem constituída pelos três campos atrás descritos, observa-se que existem portanto três tipos de elementos: `A`, `B` e `C`. A especificação de cada um dos elementos é irrelevante: tanto podem corresponder a inteiros de 32 *bits*, como a sequências de caracteres delimitados por espaços.

Os ataques gerados por este teste poderão corresponder aos seguintes pacotes:

```
[B] [C]
[A] [C]
[A] [B]
[A] [A] [B] [C]
[A] [B] [A] [C]
[A] [B] [C] [A]
[B] [A] [B] [C]
[A] [B] [B] [C]
[A] [B] [C] [B]
[C] [A] [B] [C]
[A] [C] [B] [C]
[A] [B] [C] [C]
```

Como se pode verificar, todos os pacotes foram criados a partir do original (com três elementos). Os ataques gerados correspondem à remoção de um dos campos (resultando nos três primeiros pacotes) ou à adição de um campo (restantes pacotes).

A injecção destes pacotes inválidos, permite testar a implementação do protocolo por parte da aplicação-alvo, relativamente à sintaxe do protocolo.

Teste de valor

Este tipo de testes vai avaliar o conteúdo dos campos das mensagens do protocolo. Detalhes como o tipo de dados, valores válidos e inválidos são aqui bastante importantes.

Assim, se um determinado campo comportar dados numéricos entre 0 e 1000, o teste criará ataques que injectarão pacotes com valores válidos e não válidos. Além dos válidos, os pacotes terão campos com valores *quase válidos* (e.g., os valores de fronteira -1, 0, 1000 ou 1001), "bastante inválidos" (e.g., negativos ou bastante superiores como -1000 ou 100000).

Cada ataque gerado por este teste, corresponderá a um pacote (que testa um determinado campo com dados válidos ou inválidos). Contudo, existem várias combinações possíveis de dados que esse campo pode tomar, bem como a existência de vários campos diferentes numa mensagem. Daqui resulta um elevado número de ataques/pacotes.

Fica como exemplo, o teste de valor ao protocolo de sincronização utilizado entre o *Injector* e o *Monitor* (apresentado na Secção 4.2.4). Utilizando a notação de há pouco, mas onde as letras que identificavam os campos são substituídas pelo respectivo conteúdo. Os seguintes pacotes poderão corresponder a ataques ao segundo campo (identificação do ataque):

```
[1] [0]
[1] [-1]
[1] [1000]
[1] [1000000]
[1] [10000000000]
[1] [1000000000000000]
[1] [-1000]
[1] [-1000000000]
```

Como se pode verificar, o primeiro campo tem um valor válido (1 = mensagem de fim de sincronização), sendo que o segundo campo toma vários valores (válidos e inválidos). Uma falha por paragem (*crash*) da aplicação ao receber uma destas mensagens, revela um erro de implementação, que pode significar uma vulnerabilidade do tipo *buffer overflow*[11].

4.3.2 *AJECT* vs *YPOPs!*

Para experimentação da ferramenta *AJECT* foi escolhida a aplicação *YPOPs!*[14], um programa *open-source* que oferece acesso, via POP3[9], à conta de correio electrónico do *Yahoo!*.

Desde Abril de 2002, que o *Yahoo!* invalidou o acesso grátis ao seu serviço POP3. Como resposta, foi criado o *YPOPs!*, disponível para *Microsoft Windows*, *Linux* e *Apple Macintosh*, que simula um servidor POP3 (local), permitindo a qualquer cliente de *e-mail* (Outlook, Netscape, Eudora, Mozilla, etc.) descarregar mensagens de contas *Yahoo! Mail*.

[14] `http://yahoopops.sourceforge.net/`

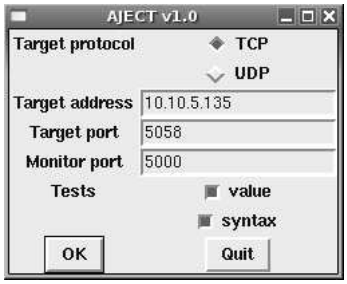


Figura 4.12: Interface gráfica da ferramenta *AJECT*.

Figura 4.13: Início de execução da ferramenta *AJECT*.

O *YPOPs!* é lançado como um serviço local (um servidor de POP3), estando configurado para aceder a uma determinada conta de correio electrónico *Yahoo!*. Os clientes de *e-mail* ligam-se localmente ao servidor *YPOPs!*(`127.0.0.1:5058`) e, através do protocolo POP3, descarregam as mensagens de correio.

O *AJECT* vai então simular um cliente de correio electrónico malicioso, interagindo com o *YPOPs!* através do protocolo POP3. As Figuras 4.12–4.16 ilustram o funcionamento da ferramenta (injector e monitor) face à aplicação-alvo.

Execução da Ferramenta

A ferramenta é lançada através da linha de comandos, como exemplificado pela Figura 4.13, mas também foi criada uma interface gráfica, para a execução do injector (ver Figura 4.12). Os parâmetros utilizados foram:

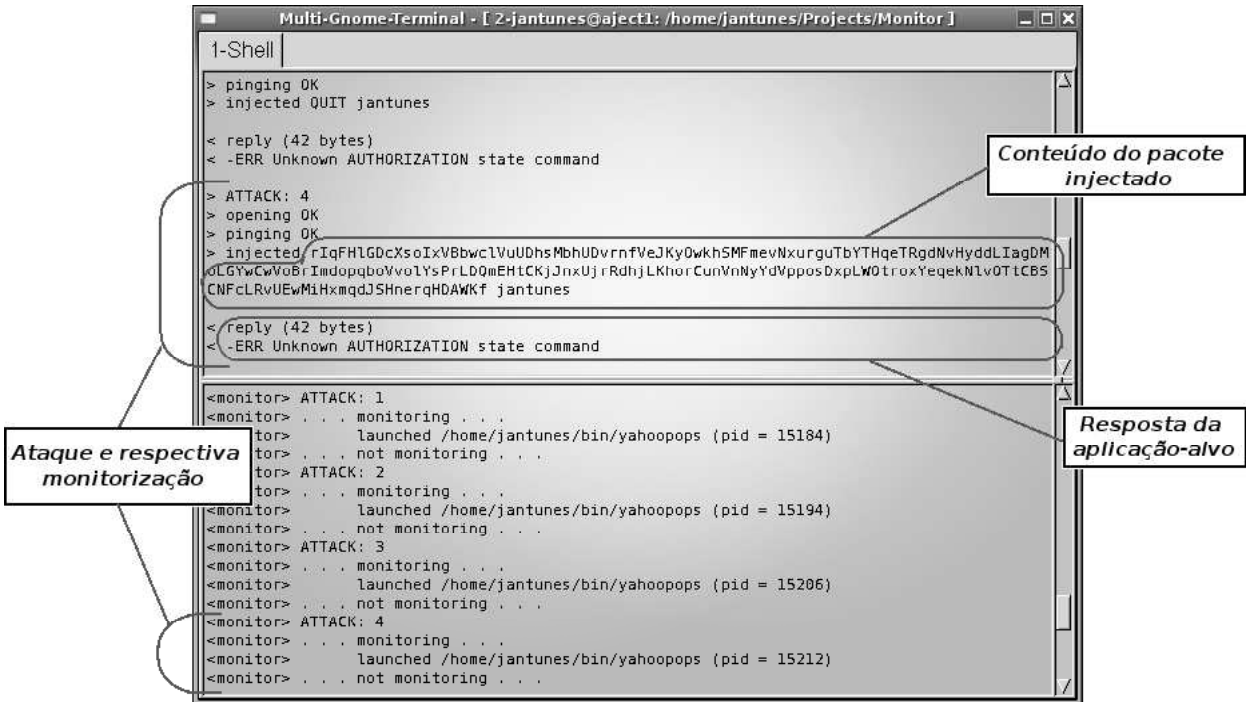


Figura 4.14: Injecção de ataques da ferramenta *AJECT*.

Injector:

`5000` – porto onde o monitor está à escuta (para sincronização com o monitor);

`TCP` – protocolo de transporte utilizado pela aplicação-alvo;

`10.10.5.135:5058` – Endereço e porto da aplicação-alvo;

`-t1` – teste de valor;

`-t2` – teste de sintaxe;

Monitor:

`5000` – porto local de escuta (para sincronização com o injector);

`/home/jantunes/bin/yahoopops` – caminho local para o executável da aplicação-alvo;

Assim, a ferramenta *AJECT* irá diagnosticar a aplicação *YPOPs!*, recorrendo aos testes de sintaxe e de valor. Tanto a aplicação (`/home/jantunes/bin/yahoopops`) como o monitor estão na mesma máquina (com o endereço IP 10.10.5.135), estando a aplicação à escuta no porto 5058 (por onde o injector envia os pacotes criados pelos testes), enquanto o monitor utiliza o porto 5000 na comunicação com o injector.

A Figura 4.14 ilustra a execução da ferramenta *AJECT*, onde se pode observar a injecção dos ataques (em cima) a par com a respectiva monitorização (em baixo). Por vezes os ataques podem produzir um comportamento "estranho" na aplicação-alvo, visível na Figura 4.15, o que normalmente indica que foi encontrado um erro.

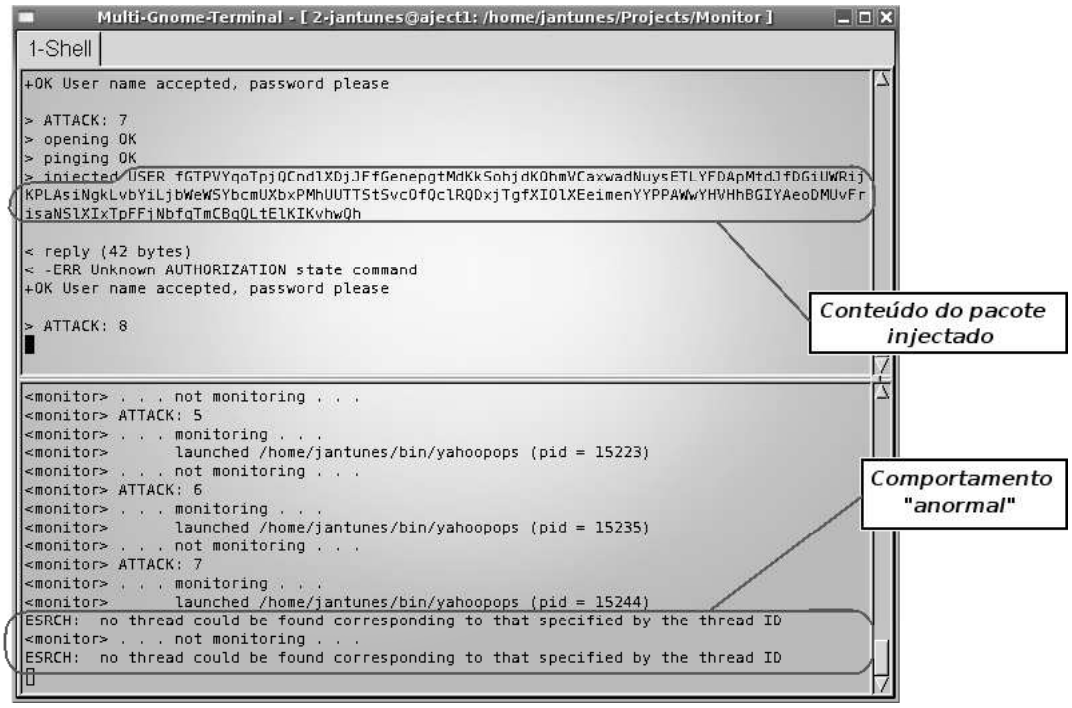


Figura 4.15: Ataque com sucesso.

Detecção de Vulnerabilidades

Por fim, é feita a análise dos resultados da injecção de ataques. São examinados os ficheiros de resultados do injector e do monitor em busca de dados que permitam concluir em que circunstâncias os ataques foram bem sucedidos (se algum).

A Figura 4.16 apresenta parte do conteúdo desses ficheiros. Pode-se constatar que o ataque nº 7 teve como resultado a terminação anormal da aplicação (através do sinal `9` — `SIGSEGV`), o indicando uma *falha* na aplicação. Em cima pode-se visualizar o conteúdo do pacote — a *falta* — que levou ao erro. **O *teste de valor* permitiu detectar uma vulnerabilidade no *YPOPs!*.**

O conteúdo do pacote permite verificar que os primeiros 4 *bytes* (`55.53.45.52 em hexadecimal`) correspondem à cadeia de caracteres (*string*) "USER", seguido de 199 *bytes* (que corresponde ao nome do utilizador). O erro de violação de memória, indicado pelo sinal `SIGSEGV`, indica que a aplicação não verificou o tamanho máximo que o campo da mensagem pode tomar, recebendo o respectivo sinal quando acede aos 199 *bytes* copiados (que excederam o tamanho máximo alocado). Estes erros escondem um tipo de vulnerabilidades muito conhecido, as vulnerabilidades de *buffer overflow*[11].

Esta vulnerabilidade[15] foi recentemente reportada em Setembro de 2004[5], estando documentada em vários sítios na Internet. Observando o código fonte da aplicação, pode-se identificar a linha de código com o problema. Na linha 375 da Listagem 4.3

[15]BugTraq nº 11256 (`http://www.securityfocus.com/bid/11256`).

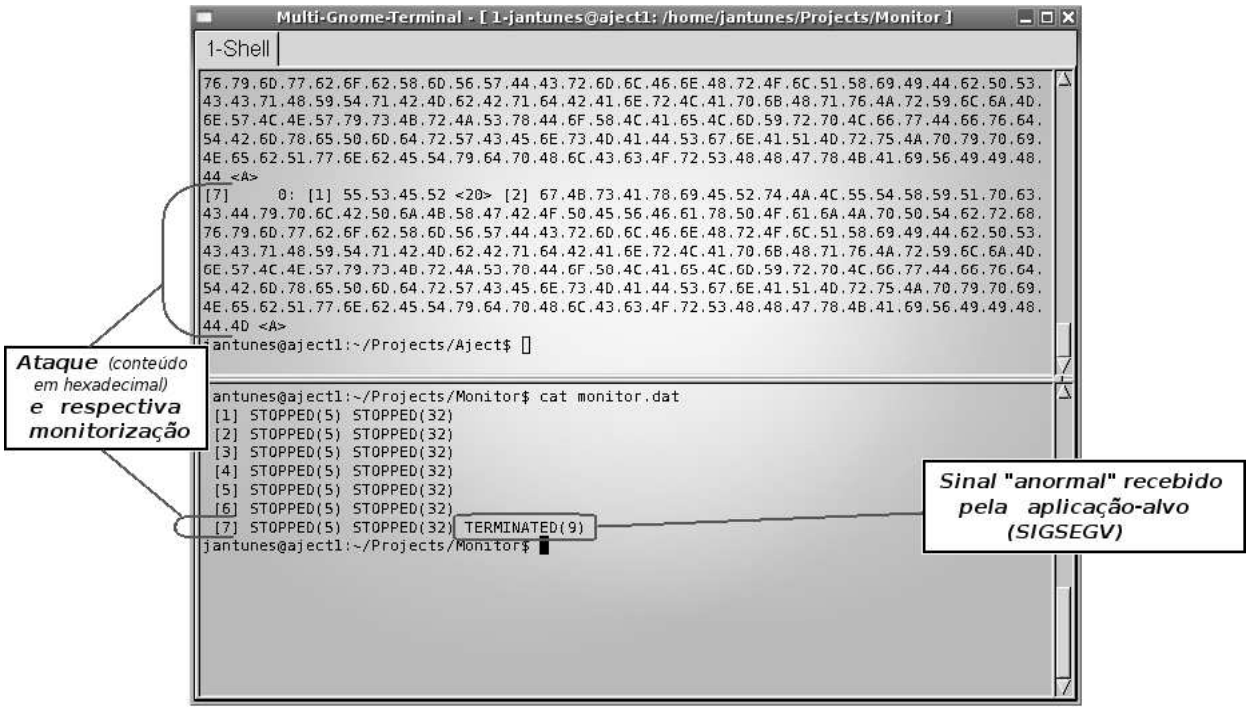


Figura 4.16: Resultado dos ataques (pacotes enviados e sinais recebidos pela aplicação-alvo).

pode-se ler:

```
sscanf(buf, "%*4s %s\r\n", username);
```

Esta linha de código C, vai copiar uma cadeia de caracteres, ignorando a primeira *string* de 4 caracteres ("USER"), para um endereço de memória (`username`), sem no entanto efectuar qualquer verificação dos limites da cadeia de caracteres ou da memória previamente reservada.

Desempenho

A eficiência de uma ferramenta de diagnóstico de vulnerabilidades está directamente relacionada com os erros de segurança detectados, existindo dois factores que caracterizam o desempenho da ferramenta:

- o desempenho dos testes criados (e.g., sintaxe, valor, etc.) e
- o número de ataques injectados durante um curto período de tempo.

O primeiro aspecto, é talvez o mais importante, pois é o que caracteriza a utilidade da ferramenta. Como o *AJECT* pode ser enriquecido com mais e melhores testes, tem-se uma ferramenta com um grande potencial no diagnóstico de vulnerabilidades.

Contudo, a rapidez na detecção de vulnerabilidades é um outro aspecto importante no desempenho de um detector de vulnerabilidades. Quantos ataques consegue a aplicação realizar por minuto? Quantas vulnerabilidades detectou na primeira hora?

```
372         if( (ptr = strchr(buf, ' ')) != NULL && strnicmp(buf,
               "USER", 4) == 0)
            {
374                 /* Fix provided by glibdud@users.sourceforge.
                       net */
                    sscanf(buf, "%*4s %s\r\n", username);
376 #ifdef WIN32
                    str.LoadString(IDS_POP3_USER_OK);
378         send(sock, str, str.GetLength(), 0);
    #else
380                 char *temp_str = "+OK User name accepted, 
                       password please\r\n";
            send(sock,temp_str,strlen(temp_str), 0);
382 #endif
            }
```

Listagem 4.3: Código fonte do *YPOPs!* com vulnerabilidade.

Foi realizado um teste de desempenho da ferramenta *AJECT* face à aplicação *YPOPs!*. O teste consistiu na execução da ferramenta, configurada do seguinte modo:

- injector, monitor e aplicação-alvo na mesma máquina de modo a não ser contabilizado a latência e largura de banda da rede;
- teste de valor (resultando em 594 pacotes com variadas combinações de dados);
- teste de sintaxe (criando 17 novos pacotes, com a adição de 2 campos e remoção de 1 campo);

O resultado foi um total de 611 ataques em cerca de 35 minutos e 43 segundos (uma média de 17 ataques/minuto), com a correcta detecção da vulnerabilidade atrás descrita — comando "USER" do protocolo POP3.

Capítulo 5

Sumário e Conclusões

O projecto realizado no âmbito do Curso de Especialização Profissional em Engenharia Informática (CEPEI), permitiu-me conhecer o ambiente da investigação científica e oferecer-me uma perspectiva mais real de um outro tipo de carreira profissional, a carreira académica e de investigação científica.

O projecto realizado foi pessoalmente muito enriquecedor e entusiasmante, permitindo que trabalhasse num área da engenharia informática pela qual me interesso bastante, a segurança. Foi sob a orientação de um grande profissional e investigador nesta área, o Professor Nuno Ferreira Neves, que tive a oportunidade de fazer investigação junto do grupo *Navigators* e de dar o meu modesto contributo à comunidade científica.

Em seguida são apresentadas as conclusões obtidas após quase dez meses de estágio, e o que ainda poderá ser feito na área da injecção de ataques e melhorado na ferramenta *AJECT*.

5.1 Conclusão

Os benefícios do diagnóstico de vulnerabilidades através da injecção de ataques são enormes. Permite a detecção atempada de alguns erros de segurança, sem a necessidade de alterar ou mesmo ter acesso ao código fonte das aplicações.

Foi criado um modelo de injecção de ataques, posteriormente concretizado pela ferramenta *AJECT*, que permite o diagnóstico de vulnerabilidades através do envio de mensagens propositadamente criadas. As potencialidades deste tipo de ferramenta são imensas. Produtos como os servidores *web*, servidores de correio electrónico, aplicações de comércio electrónico ou banca *online*, poderão ser auxiliados pela detecção (e remoção) de vulnerabilidades, antes da colocação no mercado.

Contudo, a qualidade dos resultados é tão boa quanto melhor forem as anomalias simuladas pelos ataques. Mesmo que apenas uma pequena fracção de todas as anomalias possíveis seja explorada na injecção de ataques, os resultados providenciarão informação útil sobre quão *correcto* os sistemas testados se comportarão.

Esta primeira versão da ferramenta aqui apresentada, foi posta à prova face a uma aplicação de servidor de correio electrónico — o *YPOPs!* —, conseguindo com sucesso, a detecção de uma vulnerabilidade do tipo *buffer overflow*.

Além da arquitectura e ferramenta criadas, há um outro aspecto que considero de maior importância para a engenharia informática, que foi a própria gestão do projecto e organização do estágio. Cuidadosamente analisado, planeado e posteriormente executado, sinto que ganhei sensibilidade e experiência na construção de sistemas informáticos. Face a todo o trabalho realizado, considero que atingi plenamente os objectivos do estágio que me foram propostos e que concretizei com sucesso o projecto aqui apresentado.

5.2 Trabalho Futuro

Ainda não existe solução para o problema da detecção de vulnerabilidades. A solução aqui apresentada não tem pretensões de ser a resposta definitiva para este problema, mas parece indicar um caminho com grandes potencialidades.

A injecção de ataques é uma tecnologia promissora no diagnóstico de vulnerabilidades e na avaliação da qualidade e confiabilidade do *software*.

Existem alguns aspectos que poderiam ser melhorados na ferramenta *AJECT*, nomeadamente:

- a criação de mais tipos de testes, com maior complexidade;
- a implementação de uma análise automatizada da detecção de vulnerabilidades;
- melhorar o mecanismo de especificação do protocolo-alvo, talvez até sem envolver a codificação numa linguagem de programação;

O primeiro ponto diz respeito ao número e complexidade dos ataques simulados. Se a qualidade das faltas injectadas for melhorada, será também melhorada a qualidade global do diagnóstico de vulnerabilidades.

No entanto a ferramenta *AJECT* carece ainda de uma análise automatizada de resultados, por isso este será outro ponto a melhorar numa versão posterior.

Por fim, a transição de aplicação-alvo requer conhecimentos de programação, pois é necessário codificar a especificação do protocolo-alvo. Um melhoramento prático seria a criação de um mecanismo mais intuitivo na especificação do protocolo, talvez através de ficheiros de configuração ou mesmo de uma aplicação própria.

Bibliografia

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[2] CERT/CC Statistics. `http://www.cert.org/stats/`, June 2005.

[3] FoundStone Enterprise. `http://www.foundstone.com/`.

[4] Harris Corp. STAT Scanner. `http://www.stat.harris.com/`.

[5] Hat-Squad Security Group. `http://www.hat-squad.com/en/000075.html`, September 2004.

[6] M.-C. Hsueh and T. K. Tsai. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[7] C. Huitema. *Routing in the Internet (2nd Edition).* Prentice Hall PTR, 2000.

[8] Internet Security Systems. `http://www.iss.net/`.

[9] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Standard), May 1996. Updated by RFCs 1957, 2449.

[10] Nua.com Internet Surveys. `http://www.nua.ie/surveys/how_many_online/`, June 2005.

[11] A. One. Smashing the stack for fun and profit. In *Phrack Magazine*, number 49 in 7, 1996.

[12] Qualys. `http://www.qualys.com/`.

[13] Saint Corp. `http://www.saintcorporation.com/`.

[14] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of maftia: A blueprint. In *Proceedings of the IEEE Third Survivability Workshop*, pages 157–161, October 2000.

[15] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, 2001.

[16] J. Voas, A. Ghosh, G. McGraw, and K. Miller. Glueing together software components: How good is your glue? In *Proceedings of Pacific Northwest Software Quality Conference*, October 1996.

[17] J. Voas, G. McGraw, A. K. Ghosh, F. Charron, and K. Miller. Defining an adaptive software security metric from a dynamic software failure-tolerance measure. In *Proceedings of the 11th Annual Conference on Computer Assurance*, 1996.

[18] J. A. Whittaker and H. H. Thompson. *How To Break Software Security.* Addison Wesley, 2004.

[19] Wikipedia — A enciclopédia livre. `http://pt.wikipedia.org/`, June 2005.

Apêndice A

Análise de Requisitos por Componentes

Apresenta-se um glossário com alguns termos importantes, seguido da análise de requisitos por componentes. Os requisitos estão agrupados por componentes, juntamente com uma breve descrição do papel do componente e de um exemplo prático de uma funcionalidade que este concretiza.

Especificação: Estrutura de dados que representa um pacote, mas sem conteúdo (campos, tamanho).

Pacote: Conjunto de bits (dados) organizados de acordo com uma especificação, prontos a serem enviados pela rede.

Ataque: Sequência de pacotes, com determinadas características, a enviar. Corresponde a uma instância de um teste.

Teste: Conjunto de ataques com determinadas características e respectivos resultados.

A. Gestor de Testes *Dado um conjunto de testes, corre cada um e guarda os resultados.*	
A1	Correr vários testes.
A2	Ver os resultados de cada teste.
ex:	Seleccionar os testes a realizar (sintaxe, valor) e executá-los .

B. Gerador de Ataques *Criar os ataques a injectar, e respectivos pacotes, para um determinado teste.*	
B1	Dado um teste, gerar vários ataques (diferentes e não repetidos).
B2	Criar pacotes através de uma especificação e gerador de dados.
B3	Enviar cada pacote de um ataque, ao injector de pacotes.
ex:	Dadas as características do teste, cria uma ataque (que gera o(s) pacote(s) com dados).

C. Injector de Pacotes	
	Enviar e receber pacotes (especificação + dados).
C1	Enviar um pacote para a componente-alvo e utilizar o controlador de monitorização (para sincronizar a monitorização).
C2	Passar o pacote a enviar para o coleccionador de pacotes.
C3	Receber resposta ao pacote e passá-la para o coleccionador de pacotes.
ex:	Envia o pacote ao 10.10.5.226:399 usando o protocolo UDP.

D. Coleccionador de Pacotes	
	Guardar os pacotes enviados e recebidos de um ataque.
D1	Guarda os pacotes injectados e respectivas respostas do componente-alvo.
ex:	Guarda a resposta da aplicação-alvo relativamente ao 2º pacote do 102º ataque.

E. Controlador de Monitorização	
	Sincronizar um ataque entre o Injector e o Monitor.
E1	Sincroniza o ataque entre o injector de pacotes e o monitor.
ex:	Informa o monitor que ataque nº 102 vai ter início.

F. Monitor	
	Supervisiona o componente de teste ao nível do sistema operativo (SO).
F1	Lançar o componente-alvo (criar processo).
F2	Terminar o componente-alvo (matar processo).
F3	Saber os sinais do SO que o componente recebe (*crash*, etc.).
ex:	A aplicação recebeu o sinal SIGSEGV (erro de memória) durante o 102º ataque.

G. Coleccionador de Monitorização	
	Guardar a informação de monitorização do Monitor.
G1	Guarda os dados de monitorização do monitor em disco (sinais recebidos desde o início do ataque).
ex:	Ficheiro com informação sobre o estado da aplicação-alvo, durante o 102º ataque de um teste em particular.

H. Analisador de Ataques	
	Analisar todos os dados de um ataque (à posteriori).
H1	Recolhe e trata os dados do coleccionador de pacotes e do coleccionador de monitorização, aferindo o resultado do ataque.
ex:	O 102º ataque, com o pacote com estes dados, foi bem sucedido, sendo este o estado final da aplicação (*crash*, processo fica *zombie*, etc.).

I. Analisador de Testes	
	Analisar todos os dados de um teste (à posteriori).
I1	Recolhe e trata informação do analisador de ataques, sobre o resultado de cada um dos ataques daquele teste de forma a saber se o ataque foi bem sucedido.
I2	Analisa os ataques bem sucedidos de um teste de forma a realçar os aspectos comuns e descobrir a vulnerabilidade.
ex:	O teste teve 50 em 5000 ataques bem sucedidos. Os ataques foram bem sucedidos devido ao facto do campo 3 ter valores fora da gama permitida.

Apêndice B

Plano RMMM (Risk Mitigation, Monitoring and Management)

Risco: a) Incorrecta interpretação ou especificação das exigências do cliente
Mitigação: – Ter um cuidado especial na análise de requisitos, que deverá ser convenientemente planeada. – Reuniões regulares com o cliente onde se esclarecem quaisquer dúvidas e se discute o que foi e será realizado.
Monitorização: – Durante as reuniões com o cliente, deve ser apresentado e discutida a evolução do projecto. – Observar e registar todas as opiniões do cliente sobre o projecto e trabalho realizado.
Gestão: – Corrigir o mais rapidamente possível, a especificação do sistema e análise de requisitos, o mais rápido possível, junto do cliente.

Risco: b) Impossibilidade da realização de reuniões com o cliente, sempre que necessário
Mitigação: – Planeamento do trabalho tendo em conta a disponibilidade do cliente.
Monitorização: – Manter um contacto regular com o cliente, mantendo-o a par da evolução do projecto, das dificuldades sentidas e dos futuros desafios por enfrentar, o que facilita a marcação de reuniões de carácter excepcional.
Gestão: – Minimizar a interrupção causada (e.g. planear outras tarefas que não dependam da reunião com o cliente).

Risco: c) Indisponibilidade de um membro da equipa
Mitigação: – Planeamento do trabalho tendo em conta a disponibilidade dos membros da equipa (estagiário e orientador). – Estar prevenido para o facto da possível existência de súbitas indisponibilidades (e.g. doença, acidente, morte, etc.);
Monitorização: – Manter a equipa em contacto regular.
Gestão: – Em caso de prolongada indisponibilidade (que possa afectar o prazo do projecto) deve-se redefinir os objectivos do trabalho (e.g. diminuição ou simplificação das funcionalidades).

Risco: d) Falta ou avaria de material informático
Mitigação: – Planear cuidadosamente os recursos necessários para o desenvolvimento do projecto. – Utilizar material de qualidade. – Redundância no material informático (e.g. ter um outro portátil disponível no caso do PC se avariar).
Monitorização: – Ter em mente o material afecto ao projecto para uma mais rápida verificação da falta de meios. – Qualidade do serviço fornecido pelo material informático começa a deteriorar-se: aumento de erros, diminuição de desempenho, etc.
Gestão: – Rápida aquisição ou troca do material necessário (em falta ou danificado).

Risco: e) Perda de dados informáticos
Mitigação: – Disponibilizar mecanismos que ofereçam redundância de informação: servidores para alojamento de cópias de segurança (*backup*), serviços de controlo de versões (CVS ou subversion), suportes físicos alternativos (unidade ZIP, memórias *flash*, *e-mail*, etc.).
Monitorização: – Registo regular (semanal) dos dados em suportes de *backup*. – Utilização de CVS remoto (*LaSIGE*) com cópia local. – Manter as cópias de segurança em lugares geograficamente afastados (trabalho, casa, servidor de correio electrónico de fora do país).
Gestão: – Reposição dos dados pela última cópia de segurança.

Risco: f) Desenho ou arquitectura inadequadas à concretização do sistema
Mitigação: – Realizar cuidadosamente as primeiras fases do projecto, tendo sempre em mente a concretização do sistema.
Monitorização: – Discutir os detalhes do desenho e arquitectura do sistema com o orientador, bem como a respectiva concretização. – Obter opiniões de terceiros sobre os modelos e arquitectura criados.
Gestão: – Identificar a causa e voltar à fase de desenho com o propósito de melhorar a arquitectura do sistema.

Risco: g) Falta de conhecimentos sobre segurança e injecção de faltas
Mitigação: – Realizar previamente um levantamento do tipo de conhecimentos e experiência necessários e preparar o estagiário de acordo com essa análise. – Pedir ao orientador livros e artigos científicos da área em que o projecto se encontra, nomeadamente sobre segurança informática em geral, e em particular sobre injecção de faltas, testes de *software*, ataques informáticos e vulnerabilidades.
Monitorização: – Atraso ou dificuldade na realização de um tarefa. – Tarefa incorrectamente realizada ou com demasiados erros.
Gestão: – Consoante o tipo de deficiências técnicas, devem-se encontrar meios que permitam ao estagiário rapidamente suplantar essas deficiências (e.g. ajuda do orientador, livros técnicos, informação na Internet ou em artigos científicos, etc.).

Risco: h) Problemas ou inadequação da(s) linguagem(s) de programação utilizada(s)
Mitigação: – Após a análise de requisitos, identificar junto do orientador, as linguagens de programação que melhor se adequam ao projecto. – Identificar as tarefas mais problemáticas (e mais exigentes do ponto de vista de programação) para definir o que se espera da linguagem a utilizar.
Monitorização: – Estar atento a alguma demora na concretização de alguns problemas de programação. – Elevado número de erros encontrados nas fases de testes. – Linguagem utilizada não oferece solução para o problema encontrado.
Gestão: – Caso seja necessário mudar de linguagem de programação, deve-se escolher uma linguagem alternativa e elaborar um plano de substituição de código (se possível, auxiliado por métodos automatizados de conversão entre as linguagens). – Se apenas existirem dificuldades na linguagem, não havendo a necessidade de mudança, deve-se optar por investir na formação e aprendizagem do estagiário: aquisição de livros práticos sobre a linguagem, tutoriais, artigos e portais de programação na Internet.

Risco: i) Inadequação das ferramentas escolhidas para realização do projecto
Mitigação: – Escolher cuidadosamente as ferramentas a utilizar tendo em conta a sua qualidade (e.g. escolher o IDE (ambiente de desenvolvimento integrado) com base nas críticas que recebeu). – A eleição das melhores ferramentas só deve ter início após a devida determinação das necessidades que pretende satisfazer (e.g. só escolher o IDE após identificada a linguagem de programação).
Monitorização: – Determinar se a causa da presença de eventuais atrasos ou dificuldades na realização das tarefas, se deve ao uso das ferramentas.
Gestão: – Tal como na utilização de uma linguagem de programação (que também pode ser entendido como uma ferramenta), também aqui se deve determinar qual a melhor acção a tomar: escolher uma ferramenta alternativa com possíveis custos e atrasos nessa transição; ou investir numa utilização melhorada da mesma ferramenta, com mais treino ou alguma formação extra.

Risco: j) Número de erros superior ao esperado
Mitigação: – Elaborar (e seguir) um plano de desenvolvimento de *software*.
Monitorização: – Preparar bem a equipa de desenvolvimento (estagiário e possivelmente o orientador) antes de cada fase e só após uma correcta conclusão das fases anteriores.
Gestão: – Reunir a equipa de trabalho onde se irá determinar a fonte de erros e eventuais medidas para uma futura prevenção. – Planear uma fase adicional, com início imediato e cujo objectivo é o de remover o máximo número de erros e testar a funcionalidade e robustez do trabalho.

Apêndice C

Diagramas de Gantt

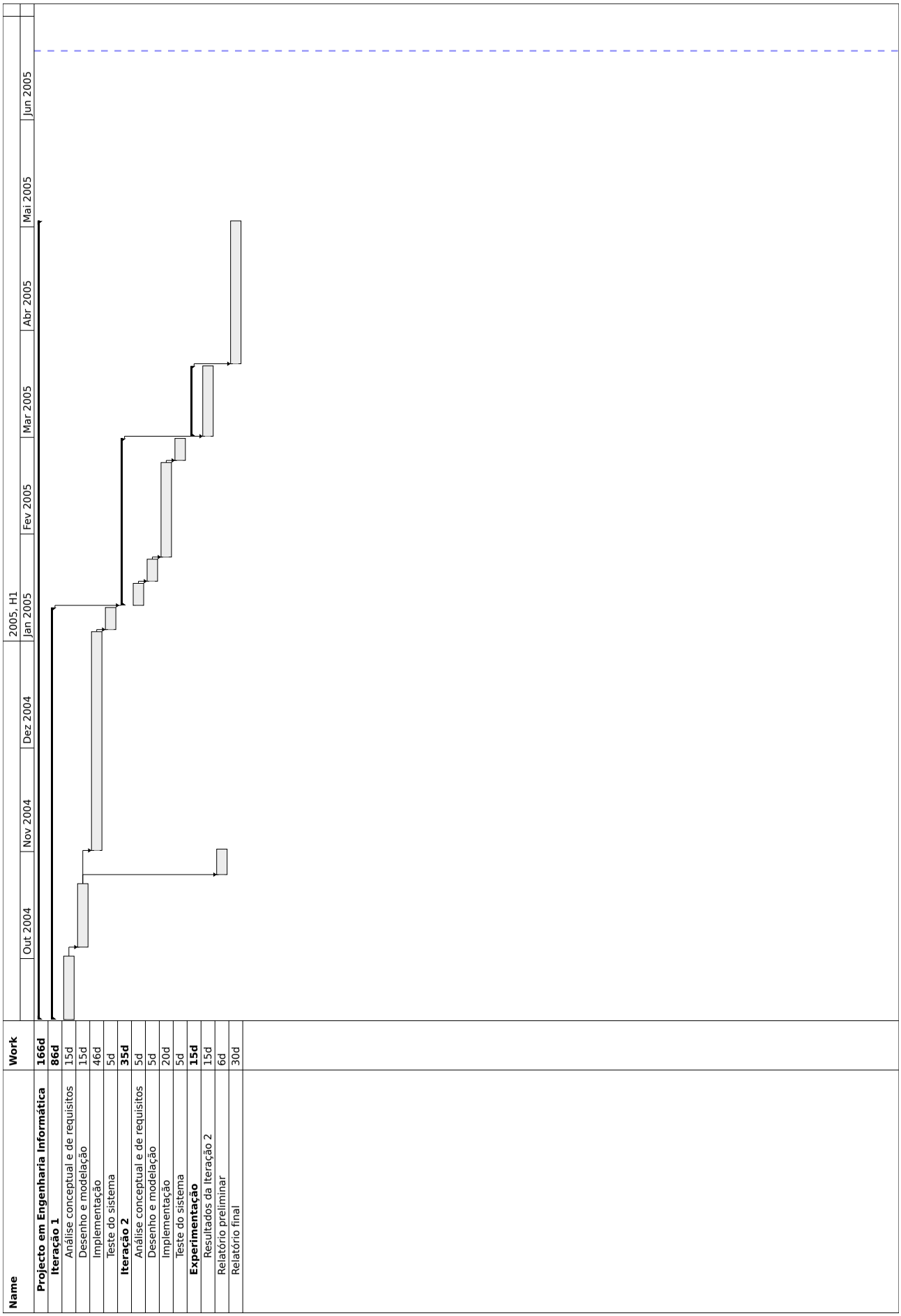


Figura C.1: Diagrama de Gantt original

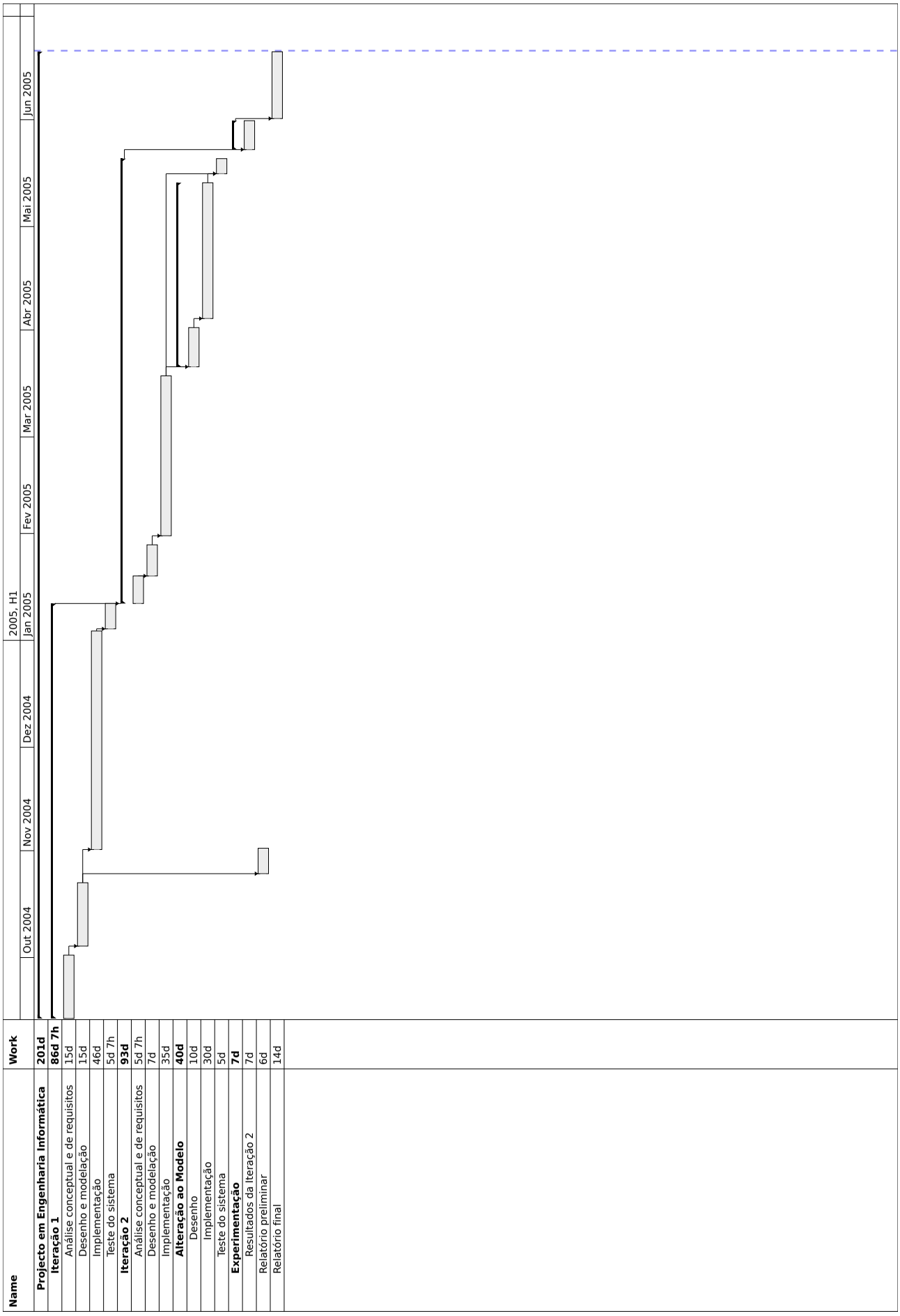


Figura C.2: Diagrama de Gantt actualizado