Diagnóstico de Vulnerabilidades através da Injecção de Ataques*

João Antunes[1], Nuno Neves[1],
Miguel Correia[1], Paulo Veríssimo[1],
Rui Neves[2]

[1] Faculdade de Ciências da Universidade de Lisboa,
Campo Grande, Edifício C6,
1749 – 016 Lisboa, Portugal
{jantunes,nuno,mpc,pjv}@di.fc.ul.pt
[2] Instituto Superior Técnico,
Av. Rovisco Pais,
1049 – 001 Lisboa, Portugal
rui.neves@tagus.ist.utl.pt

Resumo

Os sistemas informáticos requerem meios cada vez mais complexos durante o seu desenvolvimento, tendo deixado de ser construídos por uma única pessoa ou equipa, para estarem agora afectos a várias equipas e recursos. A complexidade dos erros que aparecem nestes sistemas tem também aumentado, tornando-se cada vez mais difícil a sua detecção, como é o caso dos erros de segurança. Este artigo apresenta um método automatizado para o diagnóstico de novas vulnerabilidades através dum mecanismo de injecção de faltas maliciosas (ataques). O sistema de injecção de ataques *AJECT*, oferece uma grande flexibilidade e independência em toda a sua arquitectura, possibilitando o teste de diversos tipos de aplicações cliente-servidor através de um variado número de testes.

1 Introdução

No mundo actual de comunicação global e distribuída, é fundamental que os sistemas informáticos 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 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

*Este trabalho foi suportado parcialmente pela FCT, através do projecto *AJECT* (POSI/EIA/61643/2004), e do Laboratório de Sistemas Informáticos de Grande-Escala (LaSIGE).

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 [1]).

O sistema de injecção de ataques aqui apresentado, vem possibilitar a detecção de vulnerabilidades desconhecidas através do envio de pacotes cuidadosamente criados. É o comportamento das aplicações-alvo, face a este ambiente (controlado) potencialmente malicioso, que vai permitir a descoberta das vulnerabilidades.

O artigo encontra-se organizado da seguinte maneira. A Secção 2 sensibiliza para a existência de erros aplicacionais e de como pode a injecção de ataques contribuir para a sua detecção. A Secção 3 descreve a arquitectura e modelo desenvolvidos na criação do sistema de injecção de ataques, passando depois pela Secção 4 onde são apresentados alguns resultados experimentais que demonstram a utilidade do método de diagnóstico de vulnerabilidades aqui proposto. Por fim, o artigo conclui com a Secção 5, que descreve de forma crítica e sucinta, o resultado final e trabalho futuro.

2 Detecção de Vulnerabilidades Desconhecidas

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.

2.1 Funcionalidades Não Previstas

Um sistema informático deverá obedecer a todos os seus requisitos e realizar correctamente todas as suas acções. Contudo, estes sistemas podem ainda ser explorados por piratas informáticos [2]. Assim, apesar de uma aplicação obedecer à sua especificação e não ter erros funcionais — apresentando o comportamento normal e esperado —, pode conter erros de segurança que só serão expostos face a um comportamento malicioso.

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 mesmo. Um erro de segurança pode criar vulnerabilidades que poderão ser vistas como funcionalidades não previstas.

A Figura 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 concretizar, e outras que, por con-

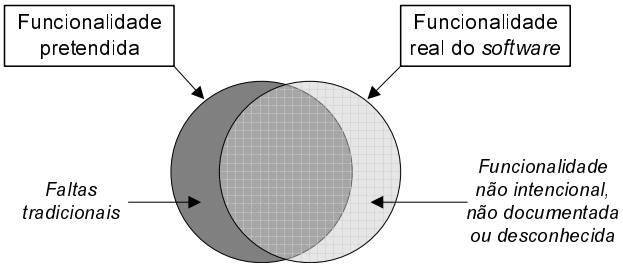

Figura 1: Funcionalidade pretendida versus funcionalidade obtida (adaptado de [2]).

sequência de erros ou por falta de correcta provisão, foram acrescidas. Estas funcionalidades acrescidas, por não terem sido intencionais e/ou serem totalmente desconhecidas, podem levar a comprometer a segurança da aplicação ou do sistema. Por exemplo, a recepção de uma mensagem não especificada no protocolo pode levar a aplicação a um estado que possibilite o acesso não previsto a algum recurso.

As aplicações podem conter assim numerosos defeitos, inseridos nas fases de desenho, na concretização, 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 a esta potencial utilidade, as vulnerabilidades podem ser vistas como uma funcionalidade acrescida, não intencional, não documentada ou desconhecida.

2.2 Modelo de Faltas e Confiabilidade

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

Mas até que ponto se pode confiar em que o sistema não viole a sua especificação? 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 em que se confia na capacidade deste fornecer o seu serviço [3, 4].

A construção de sistemas confiáveis envolve assim a criação de sistemas que oferecem garantias de que nunca violam a sua especificação. Para se atingir este objectivo 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, tenta explicar como podem os sistemas falhar, seguindo a sequência falta–erro–falha [5].

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

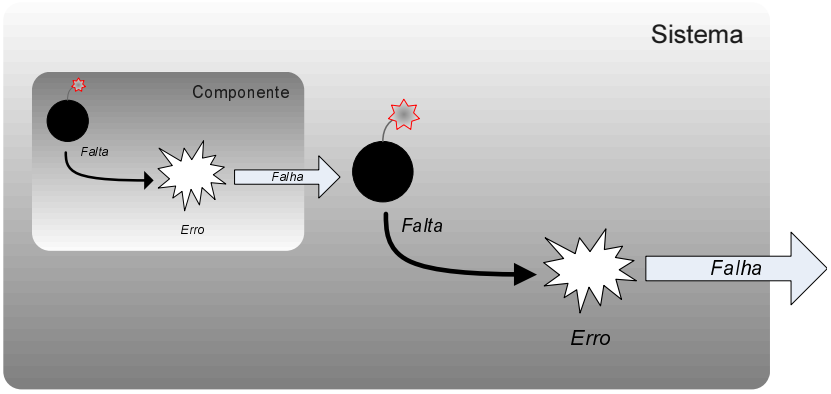

Figura 2: Modelo de faltas (Falta, Erro e Falha).

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**.

Como a construção de sistemas confiáveis envolve a prevenção da ocorrência de falhas, torna-se necessário compreender o processo que desencadeia o aparecimento destas, 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 (e.g., *checksums*, replicação), a próxima leitura ao ficheiro afectado, não irá reflectir a última escrita, o que corresponde a um violação da especificação (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 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 mais tarde num erro de paginação. Esta sequência encadeada de faltas, erros e falhas pode ainda continuar, como se mostra na Figura 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

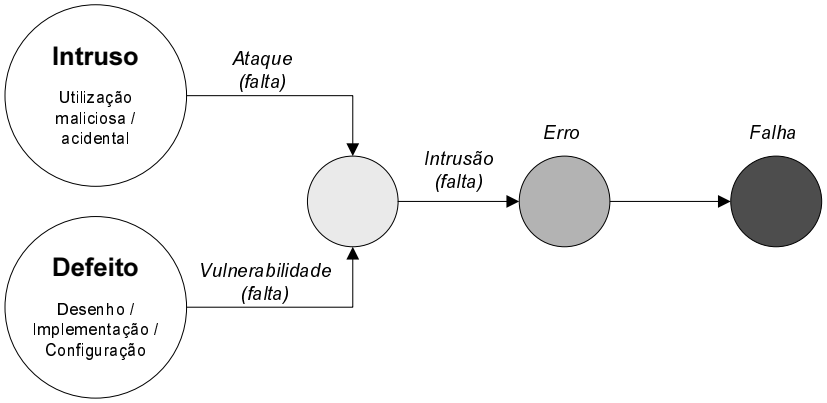

Figura 3: Modelo composto de faltas AVI (Ataque, Vulnerabilidade e Intrusão).

faltas que não obedecem a nenhuma distribuição probabilista, 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 respectivo 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 3, onde se mostra o modelo composto de faltas AVI [6], 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 sucedidos irão resultar em erros ou falhas do sistema, revelando assim as vulnerabilidades existentes.

2.3 Detecção de Vulnerabilidades

Os métodos tradicionais para o diagnóstico de vulnerabilidades são eficazes para com as vulnerabilidades conhecidas, existentes nas aplicações mais utilizadas. A maioria das ferramentas de diagnóstico de vulnerabilidades segue um modelo de diagnóstico tradicional, que passa em reconhecer o sistema-alvo (sistema operativo, aplicações e serviços que estão a correr, etc.), testá-lo face a vulnerabilidades bem conhecidas e mostrar os resultados. Alguns exemplos são: QualysGuard [7], Internet Scanner [8], FoundStone Enterprise [9], STAT Scanner Professional [10] ou SAINT [1].

A grande desvantagem deste método de detecção é que só as vulnerabilidades previamente conhecidas serão procuradas — tipicamente, é criada uma entrada numa base de dados da ferramenta contemplando a nova vulnerabilidade e o respectivo teste, necessário à sua detecção. O diagnóstico é realizado

através de uma correlação entre a versão da aplicação e as vulnerabilidades conhecidas naquela versão em particular, podendo passar também pela própria experimentação/exploração da vulnerabilidade.

Uma abordagem que não esteja dependente deste conhecimento prévio de tuplos *¡aplicação, versão, vulnerabilidade¿*, pode permitir que novas vulnerabilidades sejam detectadas. Pretende-se que a solução aqui apresentada consiga não só encontrar vulnerabilidades conhecidas, como também vulnerabilidades novas e não documentadas.

Do ponto de vista do adversário, 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.

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. Este método vai-se basear então, na observação da ocorrência de falhas no sistema num ambiente controlado de injecção de faltas maliciosas. Existindo uma vulnerabilidade, esta irá-se manifestar com a introdução do respectivo ataque. Assim, a injecção de ataques irá activar certos erros/vulnerabilidades que serão observados e consequentemente detectados.

A ferramenta *AJECT* 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.

3 Sistema de Injecção de Ataques

Qualquer sistema informático pode conter vulnerabilidades e ser passível de ser maliciosamente explorado. Há no entanto, um conjunto de aplicações mais "interessantes" do ponto de vista do atacante, e que acabam 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.

Além do fácil acesso, este tipo de aplicações apresenta um risco de segurança elevado, pois costumam oferecer serviços essenciais e delicados, na medida em que se forem comprometidos afectam a segurança de todo o sistema. É devido à importância que estas aplicações têm nos sistemas informáticos e ao elevado risco de exploração que apresentam, que serão estas em que o diagnóstico de vulnerabilidades se irá concentrar.

O sistema de injecção de ataques aqui apresentado foi concretizado na ferramenta *AJECT*. Esta ferramenta permite o diagnóstico de vulnerabilidade através

da execução de um conjunto de testes (injecção de ataques) na aplicação/sistema alvo. As próximas subsecções apresentam o modelo proposto de injecção de ataques e a respectiva arquitectura do *AJECT*.

3.1 Arquitectura do Sistema

Na Figura 4 apresenta-se a arquitectura do *AJECT* onde se destacam as três entidades básicas envolvidas: o *Injector*, o *Monitor* e o *Sistema-Alvo*. Os dois últimos formam o chamado sistema-alvo monitorizado. O *Sistema-Alvo* é exterior ao *AJECT*, correspondendo ao sistema que se deseja testar.

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 composto pela aplicação-alvo (que se deseja testar), plataforma e sistema operativo, representa 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. Este modelo modular mostra os componentes e suas relações. Os componentes são orientados à tarefa e específicos em relação ao seu propósito, correspondendo aos blocos de construção das entidades do *AJECT* atrás mencionadas — o *Injector* e o *Monitor*.

3.2 Teste, Ataque e Pacote

A injecção de ataques está relacionada com os testes e ataques pretendidos e finalmente 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 em particular.

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 então decompor em várias instâncias — **ataques**. Por sua vez, cada ataque é concretizado na rede através do envio dos **pacotes** que o constituem. Esta decomposição do conceito de injecção de ataques em teste, ataque e pacote,

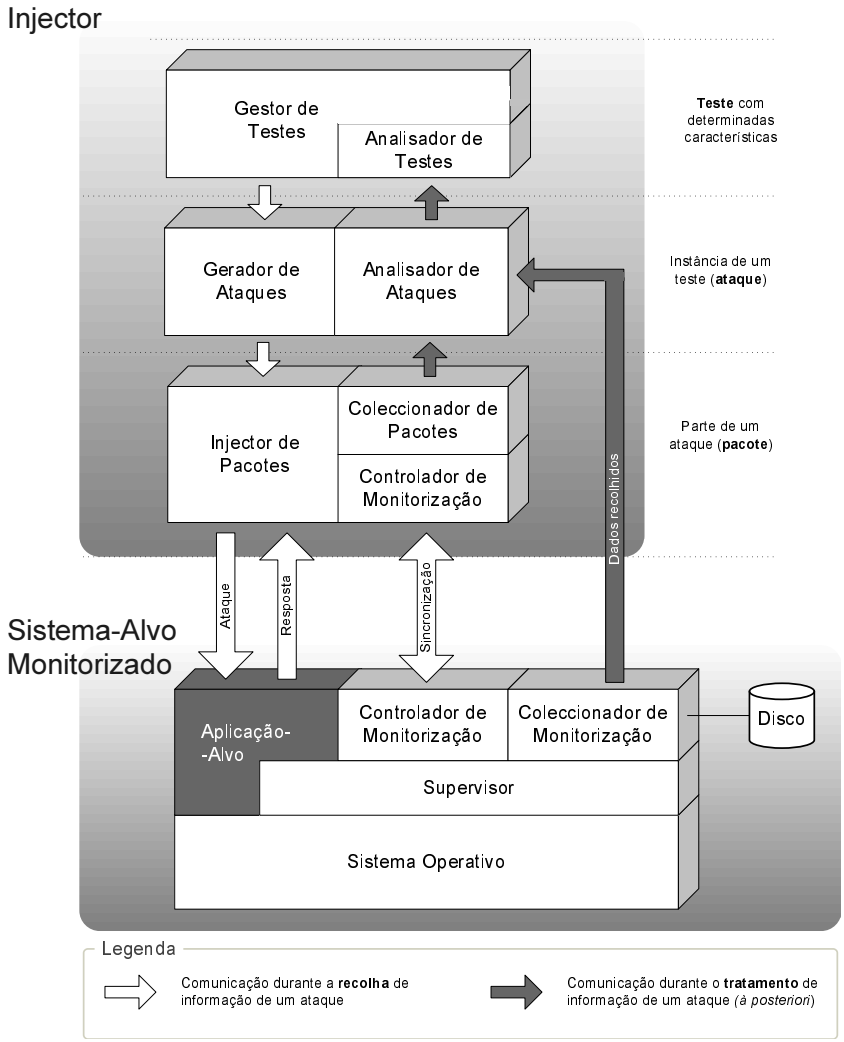

Figura 4: Arquitectura do AJECT.

está presente na arquitectura (ver Figura 4), 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`.
- 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.

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 (e.g., um pacote sem o primeiro campo ou com o mesmo em duplicado).

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 [11] (e.g., um primeiro pacote de autenticação leva a aplicação a um estado completamente diferente, donde novas vulnerabilidades podem ser descobertas). O pacote é então enviado pelo `Injector de Pacotes`, concretizando assim a injecção de um ataque.

3.3 Injector

Os componentes fundamentais desta entidade estão representados na Figura 4. 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 usado pela *Aplicação-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 cada ataque. Isto é importante para que se consiga garantir que todos os testes sejam realizados sob idênticas condições e para que o resultado de um teste não seja influenciado pelos testes anteriores.

A injecção do ataque é concretizada no `Injector de Pacotes`, através do efectivo envio dos pacotes que o constituem. 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 teste e pacotes enviados, para uma futura análise do ataque.

3.4 Monitor

Durante o ataque, o componente `Supervisor` observa o estado da `Aplicação-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 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 a `Aplicação-Alvo`, 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 através da interceptação dos 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 pela próprio `Aplicação-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 pela `Aplicação-Alvo`, mas proveniente da sua observação directa, através do componente `Supervisor`.

3.5 Geração de ataques

Um sistema de injecção de ataques necessita de mecanismos apropriados para a 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 5 apresenta-se em maior detalhe o componente `Gerador de Ataques`. Este componente é responsável por criar os pacotes que serão injectados na `Aplicação-Alvo`.

A `Aplicação-Alvo` concretiza um determinado protocolo (por exemplo o POP, SMTP, HTTP, IRC, etc.), e será a correcta concretização 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`, permite assim criar especificações de pacotes (`Pacote vazio`) e preencher os mesmos (`Pacote completo`), gerando, em tempo de execução, os pacotes que farão parte de um determinado ataque.

Mas como o ataque depende 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 crie um

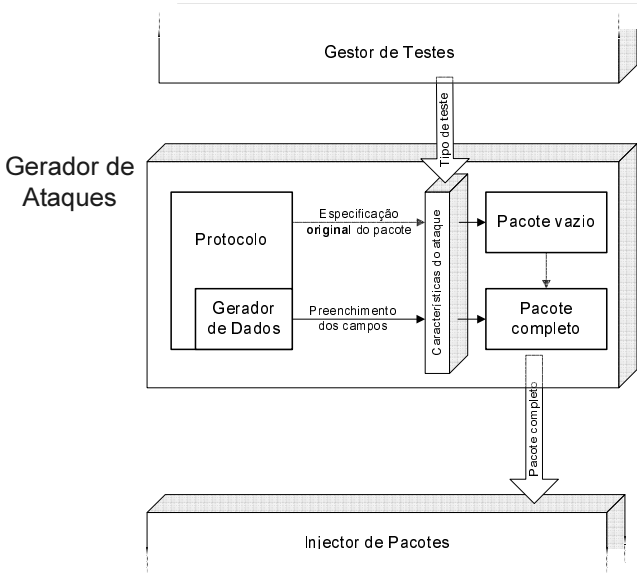

Figura 5: Gerador de Ataques.

pacote com um campo de tamanho superior à memória reservada pela aplicação-alvo, e preencher esse campo normalmente (i.e., com dados válidos).

4 Experimentação da Ferramenta

Esta secção mostra os resultados obtidos pelo *AJECT* face a uma aplicação de correio electrónico. Para o diagnóstico de vulnerabilidades foram criados dois tipos de teste: um teste de sintaxe e um teste de valor. No entanto, a ferramenta desenvolvida suporta facilmente a inserção de outros testes, possibilitando a geração de ataques com maior ou menor complexidade.

4.1 Tipos de Testes

4.1.1 Teste de sintaxe

Este tipo de testes cria ataques que violem a especificação do protocolo, relativamente à adição e remoção dos campos que o constituem. Por exemplo, consideremos um pacote constituído por três campos diferentes, representado por:

```
[A] [B] [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 concretização do protocolo por parte da aplicação-alvo, relativamente à sintaxe do protocolo.

4.1.2 Teste de valor

Este tipo de testes vai avaliar o conteúdo dos campos das mensagens do protocolo. Detalhes relativos ao tipo de dados, como os valores válidos e inválidos, são aqui bastante importantes. Assim, se um determinado campo comportar dados numéricos entre 0 e 1000, o teste irá construir 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) ou *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*. 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] [100000000000000]
[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 concretização, que pode significar uma vulnerabilidade do tipo *buffer overflow* [12].

4.2 *AJECT* vs *YPOPs!*

Para experimentação da ferramenta *AJECT* foi escolhida a aplicação *YPOPs!*[1], um programa *open source* que oferece acesso via POP3 [13] à conta de correio electrónico do *Yahoo!*.

Desde Abril de 2002, que o *Yahoo!* deixou de suportar 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*.

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.

4.2.1 Detecção de Vulnerabilidades

Este método de diagnóstico de vulnerabilidades permitiu detectar um erro de segurança conhecido na aplicação *YPOPs!*. A vulnerabilidade é activada quando a aplicação recebe um pacote com a cadeia de caracteres "USER", seguido de pelo menos 199 *bytes* (que corresponde ao nome do utilizador) arbitrários.

O erro de violação de memória, sinal `SIGSEGV` detectado pelo *Monitor*, 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*.

Esta vulnerabilidade[2] foi recentemente reportada em Setembro de 2004 [14], 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

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

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

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

Listagem 1: Código fonte do YPOPs! com vulnerabilidade.

375 da Listagem 1 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.

4.2.2 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:

- a eficácia ou cobertura dos testes criados (e.g., sintaxe, valor, etc.) e
- o número de ataques injectados por unidade 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?

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.

5 Conclusão

O diagnóstico de vulnerabilidades através da injecção de ataques pode permitir a detecção atempada de alguns erros de segurança, sem a necessidade de alterar ou mesmo ter acesso ao código fonte das aplicações. 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.

A solução aqui apresentada não tem pretensões de ser a solução definitiva, mas parece indicar um caminho promissor no diagnóstico de vulnerabilidades e na avaliação da qualidade e confiabilidade do *software*. Existem contudo, alguns aspectos a melhorar no futuro, nomeadamente: aumento do número e complexidade dos testes, automatização da análise final dos resultados e simplificação da especificação do protocolo-alvo, talvez até sem envolver a codificação numa linguagem de programação.

Referências

[1] Saint Corp.: `http://www.saintcorporation.com/` (2005)

[2] Whittaker, J.A., Thompson, H.H.: How To Break Software Security. Addison Wesley, ISBN 0321194330 (2004)

[3] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing **1** (2004) 11–33

[4] Adelsbach, A., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J.C., Powell, D., Randell, B., Riodan, J., Ryan, P., Simmionds, W., Stroud, R.J., Veríssimo, P., Waidner, M., Wespi, A.: Conceptual Model and Architecture of MAFTIA. DI/FCUL TR 03–01, Departamento de Informática, Universidade de Lisboa (2003) Projecto MAFTIA IST–1999–11583 deliverable D21.

[5] Veríssimo, P., Rodrigues, L.: Distributed Systems for System Architects. Kluwer Academic Publishers, ISBN 0792372662 (2001)

[6] Veríssimo, P., Neves, N.F., Correia, M.: The Middleware Architecture of MAFTIA: A Blueprint. In: Proceedings of the IEEE Third Survivability Workshop. (2000) 157–161

[7] Qualys: `http://www.qualys.com/` (2005)

[8] Internet Security Systems: `http://www.iss.net/` (2005)

[9] FoundStone Enterprise: `http://www.foundstone.com/` (2005)

[10] Harris Corp. STAT Scanner: `http://www.stat.harris.com/` (2005)

[11] (2nd Edition, B.B.: Software Testing Techniques. Van Nostrand Reinhold, ISBN 1850328803 (1990)

[12] One, A.: Smashing the Stack for Fun and Profit. In: Phrack Magazine. Number 49 in 7 (1996)

[13] Myers, J., Rose, M.: Post Office Protocol – Version 3. RFC 1939 (Standard) (1996) Updated by RFCs 1957, 2449.

[14] Hat-Squad Security Group: `http://www.hat-squad.com/en/000075.html` (2004)