Injeção de Ataques Baseado em Modelo para Teste de Protocolos de Segurança

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Anderson Nunes Paiva Morais e aprovada pela Banca Examinadora.

Campinas, 15 de Maio de 2009.

Eliane Martins

Prof[a]. Dr[a]. Eliane Martins (Orientadora)

Prof. Dr. Ricardo de Oliveira Anido (Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Morais, Anderson Nunes Paiva
M792i Injeção de ataques baseados em modelo para teste de protocolos de segurança/Anderson Nunes Paiva Morais -- Campinas, [S.P. : s.n.], 2009.

Orientador : Eliane Martins ; Ricardo de Oliveira Anido.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Tolerância a falha (Computação). 2.Redes de computação - Medidas de segurança. 3.Engenharia de software - Medidas de segurança. 4.Redes de computação - Protocolos. I. Martins, Eliane. II.Anido, Ricardo de Oliveira.. III. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. IV. Título.

(mfbm/imecc)

Título em inglês: Model-based attack injection for security protocols testing

Palavras-chave em inglês (Keywords): 1. Fault-tolerant computing. 2. Computer networks – Security measures. 3. Software engineering – Security measures. 4. Computer network protocols.

Área de concentração: Tolerância a Falhas

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Eliane Martins
Profa. Dra. Taisy Silva Weber (II-UFRGS)
Prof. Dr. Ricardo Dahab (IC-Unicamp)

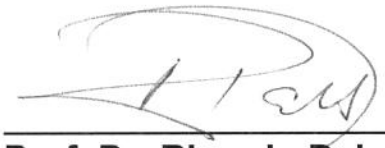
Data da defesa: 15/05/2009

Programa de Pós-Graduação: Mestrado em Ciência da Computação

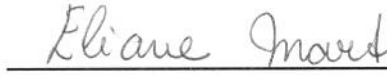
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 15 de maio de 2009, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Taisy Silva Weber
Instituto de Informática / UFRGS



Prof. Dr. Ricardo Dahab
IC / UNICAMP.



Profª. Drª. Eliane Martins
IC / UNICAMP.

Instituto de Computação

Universidade Estadual de Campinas

Injeção de Ataques Baseados em Modelo para Teste de Protocolos de Segurança

Anderson Nunes Paiva Morais

Maio de 2009

Banca Examinadora:

- Prof[a]. Dr[a]. Eliane Martins
  Instituto de Computação – UNICAMP (Orientadora)
- Prof[a]. Dr[a]. Taisy Silva Weber
  Instituto de Informática – UFRGS
- Prof. Dr. Ricardo Dahab
  Instituto de Computação – UNICAMP
- Dr[a]. Ana Maria Ambrósio
  Desenvolvimento de Sistemas de Solo – INPE (Suplente)
- Prof[a]. Dr[a]. Cecília Mary Fischer Rubira
  Instituto de Computação – UNICAMP (Suplente)

Resumo

Neste trabalho apresentamos uma proposta de geração de ataques para testes de protocolos de segurança. O objetivo é detectar vulnerabilidades de um protocolo, que um atacante pode explorar para causar falhas de segurança. Nossa proposta usa um injetor de falhas para emular um atacante que possui total controle do sistema de comunicação. Como o sucesso dos testes depende principalmente dos ataques injetados, nós propomos uma abordagem baseada em modelos para a geração de ataques. O modelo representa ataques conhecidos e reportados do protocolo sob teste. A partir deste modelo, cenários de ataque são gerados. Os cenários estão em um formato que é independente do injetor de falhas usado. Usando refinamentos e transformações, pode-se converter a descrição do cenário de ataque em scripts específicos do injetor de falhas. A proposta pode ser completamente apoiada por ferramentas de software. Nós ilustramos o uso da proposta com um estudo de caso, um protocolo de segurança para dispositivos móveis.

Abstract

We present an attack injection approach for security protocols testing. The goal is to uncover protocol vulnerabilities that an attacker can exploit to cause security failures. Our approach uses a fault injector to emulate an attacker that has control over the communication system. Since the success of the tests depends greatly on the attacks injected, we propose a model-based approach for attack generation. The model represents reported known attacks to the protocol under test. From this model, attack scenarios are generated. The scenarios are in a format that is independent of the fault injector used. Using refinements and transformations, the abstract scenario specification can be converted to the specific fault injector scripts. The approach can be completely supported by tools. We illustrate the use of the approach in a case study, a security protocol for mobile devices.

Agradecimentos

À Deus por toda força espiritual, conforto nas horas mais difíceis e pela sua presença constante.

À UNICAMP e ao Instituto de Computação, por contribuírem para o enriquecimento da minha formação acadêmica.

A Prof[a]. Dr[a]. Eliane Martins, por me receber, me orientar, pela disponibilidade, pelas valiosas idéias e críticas, pelos conselhos, pelo aprendizado e por todo apoio dados nos momentos mais desafiadores.

A minha esposa Klicia, pelo amor, amizade, companheirismo e pela ajuda nos momentos mais desgastantes e difíceis.

Glossário

BNF: *Backus-Naur Form*. É um dos mais completos conjuntos de regras para descrever mutações de PDUs mais complicadas.

CBC: *Cipher-Block Chaining*. É um modo de operação de cifra de bloco (*block cipher* em inglês). Em cada bloco de texto em claro (*plaintext* em inglês) é aplicada uma operação XOR usando o bloco de texto cifrado (*ciphertext* em inglês) anterior antes do bloco ser cifrado.

DES: *Data Encryption Standard*. É uma cifra simétrica de blocos que foi selecionada como um FIPS (*Federal Information Processing Standard*) oficial em 1976 pelos EUA.

DoS: *Denial-of-Service*. É um tipo de ataque que objetiva fazer um recurso de computação indisponível para seus usuários.

FSM: *Finite State Machine*. É um modelo de comportamento composto de um número finito de estados, transições entre estes estados e ações.

HTTP: *Hypertext Transfer Protocol*. É um protocolo de comunicação (na camada de aplicação) utilizado para transferir dados por intranets e pela Web.

IDEA: *International Data Encryption Algorithm*. É uma cifra de blocos projetada para ser o substituto do DES.

IDS: *Intrusion Detection System*. Ou Sistema de Detecção de Intrusos, é um software ou hardware projetado para detectar tentativas indesejáveis de acessar, manipular ou desabilitar sistemas de computação principalmente através da rede, como a Internet.

IF: Injeção de Falhas. Técnica que introduz falhas ou erros em um sistema alvo com o objetivo de observar o seu comportamento e testar o tratamento de erros do sistema.

IV: *Initialization Vector*. É um bloco de bits requerido na criptografia de cifra de bloco que é usado no primeiro bloco para produzir um bloco único independente dos outros blocos produzidos pela mesma chave de ciframento.

MAC: *Message Authentication Code*. É um bloco de informação cifrado usado para autenticar uma mensagem.

MD5: *Message-Digest algorithm 5*. É um algoritmo de espalhamento (*hash* em inglês) criptográfico desenvolvido pela RSA *Data Security*.

MITM: *Man-in-the-Middle*. É uma forma de Escuta Ativa no qual o atacante realiza conexões independentes com as vítimas e retransmite mensagens entre elas.

P2P: *Peer-to-Peer*. É uma topologia de redes caracterizada pela descentralização das funções na rede, onde cada terminal realiza tanto funções de servidor quanto de cliente.

PDU: *Protocol Data Unit*. Informação que é entregue como uma unidade entre duas entidades de uma rede e que pode conter informações de controle ou dados.

PKCS: *Public-Key Cryptography Standards*. É um grupo de especificações de criptografia de chave pública que são produzidas e publicadas pela empresa RSA

RC5: É uma cifra de bloco notável por sua simplicidade. Foi desenvolvida por Ronald Rivest em 1994, RC permaneceu como "**R**ivest **C**ipher".

RSA: É um algoritmo para criptografia de chave pública criado por criado por **R**ivest, **S**hamir e **A**dleman. Pode ser usado para ciframento e assinaturas digitais.

SAE: *Script de Ataque Executável*. Script de falhas descrito na linguagem de um injetor de falhas pronto para ser executado.

SAG: *Script de Ataque Genérico*. Script de falhas descrito em uma linguagem abstrata, o qual deve ser convertido para um SAE antes de ser executado.

SHA: *Secure Hash Algorithm*. É uma função *hash* criptográfica projetada pela NSA (*National Security Agency*)

SIP: *Session Initiation Protocol*. É um protocolo de aplicação, que utiliza o modelo "requisição-resposta", similar ao HTTP, para iniciar sessões de comunicação interativas.

SNMP: *Simple Network Management Protocol*. É um protocolo de gerência típica de redes TCP/IP, da camada de aplicação, que facilita o intercâmbio de informação entre os dispositivos da rede, como placas e comutadores (*switches* em inglês).

TLS: *Transaction Layer Security*. Protocolo criptográfico sucessor do *Secure Sockets Layer* – SSL, que prove comunicação segura na Internet para serviços como email (SMTP), navegação por páginas (HTTP) e outros tipos de transferência de dados.

UML: *Unified Modeling Language*. É uma linguagem de especificação padronizada de uso geral no campo da Engenharia de Software.

WAP: *Wireless Protocol Application*. É um padrão internacional para aplicações que utilizam comunicações de dados sem fio (Internet móvel), como por exemplo, o acesso à Internet a partir de um telefone móvel.

WTLS: *Wireless Transport Layer Security*. É um protocolo de segurança, parte da camada WAP, derivado do TLS.

Sumário

LISTA DE TABELAS

Lista de Figuras

Capítulo 1

Introdução

Distribuição é um aspecto dominante em sistemas computacionais hoje em dia, tanto para grandes aplicações orientadas a serviços quanto para sistemas embarcados, como os usados em carros e naves espaciais. Um sistema de comunicação é um fator chave para que distribuição aconteça. Segurança e confiabilidade (descritas na seção 2.1) têm se tornado uma preocupação maior para esses sistemas e aplicações, e se tornam ainda mais importantes já que nós dependemos desses sistemas para nossas atividades diárias. Por exemplo, um estudo feito em 2004 por *Pew Internet & American Life Project* estimou que aproximadamente 53 milhões de pessoas, ou 1 em cada 4 adultos nos EUA, usam operações bancárias pela Internet[1]. Estes indivíduos podem utilizar sua conta bancária através de redes com fio tradicionais, mas também via dispositivos móveis, por exemplo, através do protocolo WAP.

Contudo, o que nós não podemos ignorar, são as vulnerabilidades de segurança desses sistemas de comunicação, que são resultado não apenas de *bugs* introduzidos durante a fase de desenvolvimento, mas também de características intrínsecas de tais sistemas. Por exemplo, redes sem-fio são mais vulneráveis do que redes com fio equivalentes devido aos recursos limitados de terminais móveis, ou a ausência de limites físicos para a conexão. Dessa forma, qualquer dispositivo pode se conectar à rede sem fio, dado que está no limite do sinal transmitido. Por outro lado uma nova geração de hackers de rede, equipados com sólidos conhecimentos de protocolos de redes com fio (ex.: TCP/IP) e sem fio (ex.: padrão IEEE 802.11[2]), e habilidades de intrusão avançadas vêm mostrando mais interesse em infra-estruturas de comunicação.

Uma classe de vulnerabilidades que vem crescendo assustadoramente nesses últimos anos são as vulnerabilidades em protocolos de segurança. O NVD (*National*

[1] http://www.researchandmarkets.com/reports/c40160

[2] http://www.ieee802.org/11/

Vulnerability Database)[3] reportou um aumento de 138% de vulnerabilidades de autenticação e um aumento de 48% de vulnerabilidades de criptografia em 2008 em relação ao ano de 2007.

A existência de uma vulnerabilidade por si não é uma ameaça à segurança, e de fato muitas vezes ela pode permanecer adormecida por muitos anos. Uma intrusão só é materializada quanto o ataque certo é aplicado para explorar determinada vulnerabilidade. Depois de uma intrusão, o sistema pode ou não falhar, dependendo do tipo de capacidade que ele possui para lidar com os erros introduzidos pelo adversário. Algumas vezes a intrusão pode ser tolerada, mas na maioria dos sistemas atuais, ela conduz à violação de uma ou mais propriedades de segurança (ex.: *confidencialidade* ou *disponibilidade*). Portanto remoção de vulnerabilidade é muito importante para reduzir o número de ataques que tenham sucesso. Diferentes técnicas de Verificação[4] e Validação[5] (V&V – *Verification and Validation*) podem ser usadas para este propósito, tal como Análise [1], Métodos Formais [2], Verificação de Modelo [3] ou Testes [4].

Técnicas de Verificação de protocolo têm sido extensivamente estudadas para ajudar a descobrir problemas no *design* do protocolo, mas falhas também podem ser introduzidas no desenvolvimento durante as fases de implementação do código, integração e montagem [7]. O tamanho e a complexidade dos atuais sistemas usados na vida real requerem um método "composto" de V&V [32], onde a sinergia e a complementaridade entre vários métodos possam render resultados proveitosos. Por exemplo, já é demonstrado e amplamente reconhecido que modelagem e experimentação completam um ao outro, no mínimo no nível conceitual. No entanto as duas abordagens não são freqüentemente combinadas na literatura e usadas para avaliar sistemas na vida real.

Neste trabalho nós propomos o uso de Modelagem de Ataques e Injeção de Falhas por Software (SWIFI, do inglês *Software Implemented Fault Injection*) para ajudar a "descobrir" vulnerabilidades de segurança. Modelagem de Ataques tem sido usada para: **i)**

---

[3] National Vulnerability Database – NIST (http://nvd.nist.gov/)

[4] Verificação é o processo de avaliar se o sistema adere às propriedades impostas no começo da fase de desenvolvimento, denominadas condições de verificação [12].

[5] Validação consiste em assegurar que o sistema satisfaça aos requisitos especificados e as especificações de projeto em relação aos serviços providos ao usuário [12].

analisar a segurança de sistemas críticos, ajudando a identificar vulnerabilidades conhecidas e pontos fracos que poderiam ser explorados no sistema (ex.: usar árvore de ataques para modelar e classificar as ameaças a um sistema online bancário [1]); **ii)** modelar componentes de software ou sistemas sob a perspectiva de segurança (ex.: *UMLsec* [42] procura vulnerabilidades em um diagrama UML e verifica se os requisitos de segurança do sistema são satisfeitos); **iii)** especificar cenários de intrusão, fornecendo um meio de identificar ataques e ameaças (ex.: *UMLintr* [16] gera assinaturas[6] de intrusão para IDSs (*Intrusion Detection Systems*) usando UML). A técnica de SWIFI permite a introdução deliberada de falhas e erros em um sistema de software durante sua execução. SWIFI é uma poderosa técnica para validar aspectos de dependabilidade (definida na seção 2.1) e tem sido bastante usada em vários domínios de aplicação, incluindo protocolos de comunicação [5, 6]. Injetores de falhas de comunicação tradicionais geralmente emulam falhas de hardware em baixo nível, como falhas de hospedeiro (ex.: processador) ou conexão de rede. Neste trabalho nós apresentamos como esses injetores podem ser usados para emular cenários de falhas mais sofisticados obtidos a partir da modelagem de ataques reais.

1.1 Proposta e Contribuições da Dissertação

Nossa proposta, ao contrário da maioria dos trabalhos de Testes de Segurança de protocolos (ver seção 3.3), não usa a técnica de mutação da especificação das mensagens do protocolo para gerar ataques. Nossa abordagem para remoção de vulnerabilidade é baseada em Modelagem de Ataques, a qual o modelo representa tentativas bem-sucedidas e vulnerabilidades conhecidas. Essas informações estão disponíveis em diferentes fontes: base de dados de vulnerabilidades, livros, Internet e outros. O modelo dos ataques pode ser atualizado tão logo que novos ataques bem-sucedidos sejam reportados. Além disso, o modelo dos ataques pode ser reusado para testes de protocolos similares, como por exemplo, protocolos de segurança semelhantes ou derivados do TLS podem ser testados com base no mesmo modelo de ataques do TLS.

Cenários de ataque são também automaticamente gerados a partir do modelo. Estes

[6] Assinatura é um padrão que é verificado, por exemplo, no tráfego de rede com o objetivo de detectar certo tipo de ataque [47].

cenários estão especificados em um formato independente de plataforma, i.e., um formato que não é específico da ferramenta usada para injetar os ataques. Contudo, até onde sabemos, injeção de ataques obtidos de cenários de ataque especificados em uma linguagem abstrata é uma nova contribuição. Usando transformações e refinamentos nesta especificação de ataque, nós podemos obter cenários de ataque executáveis (scripts de ataque executáveis) em um formato específico da ferramenta escolhida – o injetor de falhas. Dessa forma a proposta não é dependente do código fonte do protocolo, nem da ferramenta específica usada para injetar as falhas. Para ilustrar que a proposta é adequada e possui grande utilidade, ela foi aplicada à camada de segurança do WAP: o protocolo de segurança WTLS. Experimentos foram executados com cenários de ataque, derivados de ataques de diferentes classes, com o objetivo de violar propriedades de segurança do protocolo. O objetivo principal desses experimentos foi mostrar que a proposta oferece uma forma sistemática de descobrir vulnerabilidades que foram reportadas em trabalhos anteriores. Os testes confirmaram que a proposta detectou eficientemente vulnerabilidades no WTLS. Além disso, a abordagem pode ser usada para descobrir novas vulnerabilidades, criando-se variações dos atuais cenários de ataque usados. Variações de ataque são bastante usadas para testar a qualidade das assinaturas de intrusão de IDSs [35].

Embora nós ilustremos o uso da proposta para realização de testes de segurança, ela também pode ser aplicada a outros domínios. As principais contribuições que este trabalho traz são:

- Fornecer um método para automaticamente gerar cenários de ataque a partir de um modelo genérico de ataques reais de diferentes classes, favorecendo a reusabilidade [7] e a portabilidade [8] dos cenários de ataque;
- Fornecer um método automatizado para refinamento de cenários de ataque para um formato executável por um injetor de falhas;
- Ilustrar a aplicação da abordagem proposta usando um estudo de caso real e

[7] "Reusabilidade" (*reusability* em inglês) é o grau de facilidade ou potencialidade que um componente (ex.: parte de um código fonte) possui para ser reusado [40].

[8] Portabilidade é a característica das aplicações serem executáveis em outras plataformas além daquela de origem [40].

mostrando sua eficiência em encontrar vulnerabilidades;

- Fornecer uma abordagem para realização de Testes de Segurança preciso e efetivo usando as técnicas de Modelagem de Ataques e Injeção de Falhas em protocolos.

1.2 Organização da Dissertação

Esta dissertação está organizada da seguinte forma:

- O capítulo 2 apresenta os principais conceitos necessários para entender a descrição da proposta.
- O capítulo 3 relata os trabalhos relacionados e estabelece uma comparação entre os trabalhos com o objetivo de apresentar problemas em aberto na área de Testes de Segurança.
- O capítulo 4 contém uma descrição da proposta mostrando em detalhes como obter os cenários de ataque a partir de ataques reais, como mapear os cenários em falhas de comunicação e como gerar scripts de ataque.
- O capítulo 5 aplica a proposta a um estudo de caso: um protocolo de segurança usado em dispositivos móveis.
- O capítulo 6 analisa os resultados dos testes experimentais usando o estudo de caso (protocolo segurança) e o injetor de falhas, dessa forma mostrando a aplicabilidade da proposta.
- O capítulo 7 fecha o trabalho relatando os resultados e contribuições que foram alcançadas e apresentando algumas direções para trabalhos futuros.

Capítulo 2

Aspectos de Segurança

Este capítulo apresenta os principais conceitos relacionados à dependabilidade, segurança, vulnerabilidades e modelagem de ataques que serão usados ao longo desse trabalho, principalmente na metodologia proposta no capítulo 4 e na aplicação da metodologia nos capítulos 5 e 6.

Os conceitos de falha, erro e defeito, bem como a classificação de falhas e defeitos, e a definição segurança e suas propriedades são apresentados na seção 2.1. As definições de ataque, vulnerabilidade e intrusão são exibidas na seção 2.2. Os principais ataques contra redes sem-fio e as propriedades de segurança violadas por eles são descritos na seção 2.3. Na seção 2.4 é apresentado o conceito de Modelagem de Ataques, bem como a técnica de Modelagem de Ataques usada nesse trabalho para representar os ataques contra o protocolo alvo – árvore de ataques.

2.1 Terminologia relacionada à Segurança

Dependabilidade (*dependability* em inglês) [12] é a habilidade de um sistema de computação em fornecer um serviço que pode ser considerado confiável. Um **defeito** (*failure* em inglês) de sistema é um evento que ocorre quando o serviço oferecido diverge do *serviço correto*, que é o serviço que implementa uma função do sistema descrita por uma especificação funcional. Um defeito é então uma transição do *serviço correto* para o *serviço incorreto*, i.e., a não implementação de uma função do sistema. Um sistema pode falhar porque ele não obedece à especificação ou porque a especificação não descreve de maneira satisfatória a função do sistema que implementa o serviço. Um **erro** (*error* em inglês) é a parte do estado do sistema que pode causar um subseqüente defeito: um defeito ocorre quando o erro alcança a interface do serviço e altera o serviço. Uma **falha** (*fault* em inglês) é a causa confirmada ou teórica de um erro. Uma falha é ativada quando ela produz um erro, se não ela continua dormente. Um sistema nem sempre falha da mesma maneira. As maneiras nas quais o sistema pode falhar são seus **modos de defeito** (*failure mode* em

inglês), que podem ser classificados de acordo com a severidade dos defeitos. Os modos de defeito de comunicação usados nesse trabalho são descritos na seção 3.1.

Falhas podem ser classificadas de acordo com seis principais critérios: 1) fase de criação ou ocorrência: **falhas de desenvolvimento** e **falhas operacionais**; 2) limites do sistema: **falhas internas** e **falhas externas**; 3) domínio: **falhas de hardware** e **falhas de software**; 4) causas fenomenológicas: **falhas naturais** e **falhas humanas**; 5) intenção: **falhas acidentais** ou **falhas não-maliciosas intencionais** e **falhas intencionalmente maliciosas**; 6) persistência: **falhas permanentes** e **falhas transientes**. Combinando essas seis classes de falhas essenciais obtemos três classes de falhas que são levadas em consideração para definir os mecanismos de defesa de um sistema: 1) **falhas de projeto** que incluem: vulnerabilidades de software, lógicas maliciosas e erros de hardware; 2) **falhas físicas** que incluem: lógicas maliciosas, erros de hardware, defeitos de produção, deterioração física, interferência física e ataques; 3) **falhas de iteração** que incluem: interferência física, ataques (lógicas maliciosas e tentativas de intrusão) e erros de entrada.

Dependabilidade é um conceito integrado que compreende os seguintes atributos básicos: **i)** disponibilidade: prontidão para o *serviço correto*; **ii)** confiabilidade: continuidade do *serviço correto*; **iii)** segurança (*safety* em inglês): ausência de conseqüências catastróficas no ambiente e para os usuários; **iv)** confidencialidade: ausência de divulgação não-autorizada da informação; **v)** integridade: ausência de alterações impróprias do estado do sistema; **vi)** "manutenabilidade" (*maintainability* em inglês): habilidade de estar sujeito a reparos e modificações.

Segurança (*security* em inglês) é a combinação dos seguintes atributos de dependabilidade, os quais foram especializados: disponibilidade (somente para usuários autorizados), confidencialidade e integridade (ausência de alterações "não-autorizadas" do estado do sistema). Apresentamos a seguir uma descrição mais detalhada das propriedades de segurança [12]:

- ***Confidencialidade*** é a propriedade que garante que usuários não-autorizados não obtenham conhecimento de informação sensível, i.e., ausência de revelação não-autorizada de informação. É normalmente expressa em termos de controle apropriado nos fluxos de informação.

- ***Integridade*** é a ausência de alteração imprópria de estado do sistema, i.e., a prevenção de modificação ou supressão não-autorizada de informação. A interpretação de *integridade* pode ser vista como uma dualidade de *confidencialidade*. Enquanto *confidencialidade* requer que informações sensíveis não sejam levadas a usuários não-autorizados, *integridade* requer que usuários não-autorizados não influenciem em informações sensíveis.
- ***Disponibilidade*** é a prevenção de retenção desnecessária da informação, em outros termos, é prontidão para o uso imediato. *Confidencialidade* e *integridade* estão relacionadas em prevenir a ocorrência de eventos indesejáveis, por outro lado *disponibilidade* garante que eventos desejáveis certamente ocorram. Quando um usuário requisita um serviço o qual ele está autorizado, o sistema deveria garantir que o serviço é fornecido de uma maneira correta e em tempo adequado.

Muitas propriedades de segurança podem ser definidas em termos de *confidencialidade*, *integridade* e *disponibilidade* da informação, ou do serviço em si, ou de algumas metas-informações (ex.: tempo de entrega do serviço ou tempo de sua criação, identidade da pessoa que invocou uma operação) relacionadas à informação ou serviço. No que se refere aos protocolos de segurança, outras propriedades são normalmente usadas. ***Privacidade*** é *confidencialidade* com respeito a dados pessoais, os quais podem ser: uma informação, como o conteúdo de um registro de base de dados, uma meta-informação, como a identidade de um usuário que executou uma operação particular, o envio de uma mensagem particular, o recebimento de uma mensagem, etc. ***Autenticidade*** é propriedade de ser "genuíno". Para uma mensagem, *autenticidade* é equivalente a *integridade* do conteúdo da mensagem (*integridade* de informação), da origem da mensagem, e possivelmente de outras informações da mensagem, como horário de emissão, nível de classificação, etc. (*integridade* de meta-informação). **Autenticação** é o processo que garante confiança a *autenticidade*. ***Não-repúdio*** corresponde à *disponibilidade* e *autenticidade* de algumas metas-informação da mensagem, como a identidade do criador (e possivelmente horário de criação) para não-repúdio de origem ou identidade do receptor para não-repúdio de recepção.

Os requisitos ou objetivos de segurança de um sistema são declarações em alto nível

de quais propriedades de segurança o sistema deve garantir. A violação de um requisito de segurança corresponde a uma falha de segurança do sistema. Típicos requisitos de segurança incluem: "a *confidencialidade* de dados sensíveis deve ser mantida" ou "a *integridade* ou a *disponibilidade* de dados do sistema para usuários autorizados deve ser mantida".

2.2 Entraves a Segurança

Entraves a segurança podem ser definidos em termos do modelo de falhas composto: *ataque – vulnerabilidade – intrusão* [12]. A Figura 1.1 apresenta as causas de defeitos de um sistema devido a várias classes de falhas (explicadas na seção anterior). Na Figura 1.1 podemos observar que defeitos podem ser causados por dois tipos de categorias de falhas [12]: falhas maliciosas (ataques) e falhas acidentais. O modelo *ataque – vulnerabilidade – intrusão* é uma especialização da seqüência *falha* → *erro* → *defeito* aplicada a falhas maliciosas. Esse modelo limita o espaço de falhas que interessa à composição: (ataque + vulnerabilidade) → intrusão.

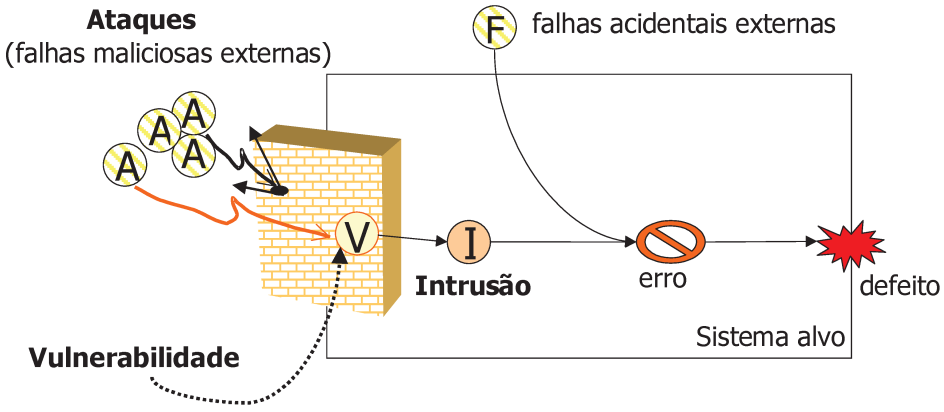


Figura 2.1: Ameaças a Segurança.

Ataque é uma falha de iteração maliciosa, através da qual um atacante intencionalmente viola uma ou mais propriedades de segurança do sistema (equivalente a uma tentativa de intrusão), por exemplo, um usuário interno ou externo da rede (ex.: hacker ou administrador) tentando utilizar informações sensíveis armazenadas em um servidor.

Ataques podem ser passivos ou ativos. Ataques também podem ser definidos tecnicamente como falhas de iteração maliciosas que tentam ativar uma ou mais vulnerabilidades (ex.: *applets* Java maliciosos, email com vírus). Um ataque que ativa com sucesso uma vulnerabilidade causa uma intrusão. Um atacante passivo tenta aprender ou fazer uso de informações do sistema, mas não afeta os recursos do sistema. Um atacante ativo, por outro lado, tenta alterar os recursos do sistema ao afetar sua operação.

Vulnerabilidade é uma falha maliciosa ou não-maliciosa [12] introduzida acidentalmente ou intencionalmente durante as fases de desenvolvimento (requisitos, análise, projeto ou configuração) do sistema (ex.: erros de código permitindo um overflow da pilha do programa), ou durante sua operação (ex.: arquivos com permissão de administrador no Unix, portas TCP/IP que não são protegidas), que poderia ser explorada para criar uma intrusão.

Intrusão é uma falha operacional [12] intencionalmente maliciosa no domínio do software, a qual foi originada externamente aos limites do sistema e que foi resultante de um ataque bem-sucedido em explorar uma vulnerabilidade. Até uma falha de iteração maliciosa cometida por um usuário interno pode ser classificada como uma intrusão, pois a intenção é executar uma operação em algum recurso que não é desejável pelo dono desse recurso. Uma intrusão pode levar o sistema a um estado errôneo (ex.: uma *shell* ou uma conta não-autorizada com privilégios de administrador). No caso em que erros não são detectados e isolados um defeito irá ocorrer, i.e., a especificação do sistema e, em particular, as propriedades de segurança do sistema são violadas.

Alguns ataques podem não gerar intrusões, como mostrado na Figura 1.1, pois podem ser prevenidos pelos mecanismos de defesa existentes do sistema. Um ataque é uma tentativa de intrusão, e a intrusão resulta de um ataque que foi bem-sucedido. A Figura 1.1 também mostra que defeitos do sistema podem ocorrer como conseqüência de falhas não-maliciosas (acidentais) externas, i.e., falhas que são conseqüência de defeitos em componentes externos aos limites do sistema que interagem com ele (ex.: falhas de hardware ou sistema operacional [12]).

Podem-se usar quatro métodos de prevenção de falhas maliciosas [12] para garantir a segurança (definida usando os atributos de dependabilidade), considerando o modelo *ataque – vulnerabilidade – intrusão* (que é uma especialização do modelo de falhas):

prevenção, **tolerância**, **remoção** e **previsão**.

- **Prevenção** de falhas maliciosas objetiva: prevenir a ocorrência de ataques, introduzindo mecanismos como autenticação, autorização e firewalls; prevenir a ocorrência ou introdução de vulnerabilidades, incluindo medidas nas áreas de especificação formal, projeto rigoroso e gerenciamento de sistema. Prevenção de intrusão pode ser vista como a aplicação combinada de prevenção de ataque e prevenção de vulnerabilidade, tanto como remoção de ataque e vulnerabilidade.
- **Tolerância** a falhas maliciosas objetiva tolerar intrusões, assim como tolerar vulnerabilidades e ataques, fornecendo serviços corretos na presença de intrusões, i.e., assegurando que o sistema alvo forneça garantias de segurança, apesar dos ataques parcialmente bem-sucedidos. Note que a presença de vulnerabilidades pode ser tolerada se nenhum ataque ocorrer, ou seja, prevenção e remoção de ataques são também formas de tolerância a vulnerabilidades.
- **Remoção** de falhas maliciosas objetiva a remoção de ataques, i.e., reduzir o número ou a severidade de ataques, o qual inclui ações de manutenção que removem lógicas maliciosas que agem, ou são capazes de agir como agentes de ataque. Também objetiva a remoção de vulnerabilidades, i.e., reduzir a presença de vulnerabilidades no sistema, logo restringindo o poder dos atacantes. Esta pode ser executada durante a fase de desenvolvimento do sistema, usando procedimentos de Verificação, como Métodos Formais, Verificação de Modelos e Testes, os quais identificam falhas que poderiam ser exploradas por um atacante. Também pode ser executada durante a fase de operação do sistema, que corresponde a ações de manutenção preventivas e corretivas, como aplicação de *patches* de segurança, isolamento de determinados serviços, alteração de passwords periodicamente.
- **Previsão** de falhas maliciosas objetiva a previsão de ataques, i.e., como estimar o número atual de ataques, a freqüência de ataques no futuro e as possíveis conseqüências dos ataques; e objetiva a previsão de vulnerabilidades que inclui reunir estatísticas sobre o estado atual de conhecimento das vulnerabilidades do sistema, e as dificuldades que um atacante teria que superar para tirar vantagem delas.

Neste trabalho o foco da Validação de segurança é a **remoção** de falhas maliciosas. A intenção é detectar vulnerabilidades do sistema na presença de falhas maliciosas intencionais (ataques). Para isso foram injetadas falhas para emular ataques. Dado que nosso objetivo é injetar ataques bem-sucedidos e conhecidos do protocolo de segurança do WAP, é apresentada na sessão 2.3 uma classificação de ataques que são realizados contra redes sem-fio.

2.3 Ataques a Redes Sem-fio

Apesar de seu futuro promissor, segurança tem se tornado a principal preocupação em redes sem-fio, devido ao risco de comprometer ou limitar a migração de serviços críticos (e.x.: *e-commerce*, *e-banking*) para plataformas sem-fio. Embora a maioria, senão todas as ameaças de segurança contra o protocolo TCP/IP em uma rede com fios sejam igualmente aplicáveis a uma rede sem-fios baseada em IP, essa última possui um número de características adicionais que a torna ainda mais desafiante em relação à segurança. Redes sem-fio apresentam mais ameaças à segurança do que redes equivalentes com fio, por causa de sua natureza. Redes sem-fio não estão fisicamente limitadas, no sentido de que as ondas de rádio usadas como canais de comunicação se propagam muito bem no meio, o que torna as ameaças de injeção e captura de mensagens ainda mais preocupantes. Outras limitações podem ser apontadas, como capacidade de canal e largura de banda de rede menores, memória reduzida e poder de processamento limitado dos dispositivos, os quais tornam as redes sem-fio mais vulneráveis a ataques de negação de serviço (DoS). Também o aumento da complexidade do sistema, devido a necessidades especiais de utilização mais eficiente dos canais e uso de mobilidade, introduz potenciais vulnerabilidades de segurança. Além disso, redes sem-fio são mais sujeitas a ruídos e outras interferências no canal. Essas são algumas de suas vulnerabilidades inerentes que podem ser exploradas por atacantes.

Foi usada a taxonomia apresentada por Welch e Lathrop [13], na qual eles classificam ataques de acordo com as propriedades de segurança (descritas na seção 2.1) que eles tentam violar. Ataques contra *confidencialidade* incluem **Análise de Tráfego**,

Escuta Passiva (*Passive Eavesdropping* em inglês) e ***Wardriving***[9]. Estes são exemplos de ataques passivos que são difíceis de detectar, pois um atacante pode monitorar o tráfego de rede para capturar informações para análise apenas usando um adaptador de redes sem-fio.

Análise de Tráfego é uma técnica simples, na qual o atacante determina a carga de tráfego do meio de comunicação através do número e tamanho dos pacotes sendo transmitidos, da origem e do destino dos pacotes, e dos tipos de pacotes. Supõe-se que os dados do pacote (*payload* em inglês) estão cifrados, e o atacante não pode decifrá-los, deixando apenas o cabeçalho visível para o atacante. O atacante apenas precisa de um cartão sem-fio operando em modo promíscuo e um software para contar o número e o tamanho dos pacotes sendo transmitidos. Análise de tráfego permite ao atacante obter informações como a existência de atividade na rede, identificação de servidores, localização física de pontos de acesso (AP – *Access Point* em inglês) no caso de redes sem-fio padrão IEEE 802.11 e os tipos de protocolo sendo usados na transmissão.

Em **Escuta Passiva** o atacante passivamente monitora a sessão sem-fio e o conteúdo das mensagens. Se os dados dos pacotes estão cifrados, o ciframento é quebrado para ler o texto não-cifrado (texto em claro), comprometendo a *privacidade* das informações, especialmente sua origem, destino, identificador e horário de transmissão. A única condição é que o atacante tenha acesso à transmissão. O atacante pode ter acesso a dois tipos de informações usando este tipo de ataque: pode ler os dados transmitidos em uma sessão e pode reunir informações indiretamente examinando os pacotes da sessão.

Embora ataques passivos não interfiram no tráfego de mensagens, eles podem ser usados por um atacante para obter informações suficientes para executar ataques ativos. Dentre os ataques ativos destacam-se **Escuta Ativa** (*Active Eavesdropping* em inglês), **Acesso não-autorizado**, **MIMT**, **Seqüestro de Sessão**, **Ataque por Repetição** e **DoS**.

Escuta Ativa é um ataque contra *confidencialidade* e *integridade*. Esta técnica envolve a injeção de dados na comunicação para ajudar a decifrar os dados da mensagem. O atacante monitora a sessão sem-fio, como descrito no ataque de Escuta Passiva, mas também injeta ativamente mensagens no meio de comunicação para auxiliar a determinar o conteúdo das mensagens. As pré-condições são que o atacante tenha acesso à transmissão,

[9] *Wardriving* é o ato de procurar por redes sem-fio dirigindo um veículo (ex.: um carro). Para isto, usa-se um computador equipado com um cartão sem-fio, como um laptop ou um PDA, para detectar a rede.

bem como a parte do texto em claro da mensagem, como endereço de IP de destino, ainda que não tenha acesso ao conteúdo inteiro da mensagem. Esse tipo de ataque pode ter duas formas: o atacante pode modificar um pacote ou pode injetar pacotes completos no fluxo de dados. Um exemplo de Escuta Ativa com texto em claro parcialmente conhecido é **Falsificação de IP** (*IP Spoofing* em inglês). Nesse ataque, o atacante altera o endereço de IP de destino do pacote para o endereço de IP do host que o atacante controla.

Acesso não-autorizado é diferente de qualquer um dos tipos de ataques anteriores discutidos, no sentido que não está direcionado para um usuário final ou grupo de usuários, mas sim contra a rede como um todo. Uma vez que o atacante tem acesso à rede, ele/ela pode lançar ataques adicionais ou apenas usar os serviços da rede livremente. Embora uso da rede de graça possa não ser uma ameaça significativa para muitas redes, ter acesso à rede é o passo chave para vários ataques, como ataques MITM.

Seqüestro de Sessão é um ataque que viola a *integridade* de uma sessão. O atacante toma uma sessão autenticada e autorizada de um usuário autêntico. O usuário alvo sabe que não tem mais acesso à sessão, mas não está ciente que a sessão foi tomada por um atacante. Uma vez que o atacante possui uma sessão válida, ele/ela pode usar a sessão para o propósito que quiser e manter a sessão por um longo tempo.

Ataques por Repetição violam a *integridade* da informação na rede ou a *integridade* de uma sessão específica. Esses ataques são usados para ganhar acesso à rede com a autorização do alvo, sendo que a sessão ou as sessões que são atacadas não sofrem alteração ou interferência. Durante um ataque por repetição o atacante captura a autenticação de uma ou mais sessões. Então repete a sessão autenticada futuramente ou usa múltiplas sessões para sintetizar a parte de autenticação de uma sessão.

Ataques Homem no Meio (**MITM** – *Man-in-the-Middle* em inglês) também violam *confidencialidade* e *integridade* de dados. Um ataque MITM pode ser usado para ler dados privados de uma sessão ou modificar os pacotes violando a *integridade* da sessão. Este ataque é realizado em tempo real, significando que o ataque ocorre durante uma sessão ativa da máquina alvo. Existem múltiplas maneiras de implementar esse ataque. Um exemplo é quando o alvo possui uma sessão de autenticação ativa, então os seguintes passos são realizados: **1)** o atacante interrompe a sessão ativa e não permite que o alvo se re-conecte com o servidor (ex: AP); **2)** a máquina alvo tenta se re-conectar com a rede sem-

fio através do servidor e só é capaz de se conectar com a máquina do atacante que se faz passar pelo servidor. Também nesse passo o atacante se conecta e se autentica com o servidor em nome da máquina alvo. Logo, se um túnel cifrado, o qual foi estabelecido por um protocolo de tunelamento[10], é usado, o atacante estabelece dois túneis cifrados: entre ele e o alvo, e entre ele e o servidor. Variações desta técnica de ataque são baseadas nos mecanismos de segurança empregados. Por exemplo, quando ciframento e autenticação não são usados pelo servidor e cliente, o atacante imita um servidor real na rede sem-fio e a máquina alvo inadvertidamente se conecta com o servidor falso que passa a agir como um *proxy* na rede sem-fio.

Ataques de Negação de Serviço (**DoS**[11] – *Denial of Service* em inglês) violam a *disponibilidade* e são caracterizados por tentativas explícitas de ataques para evitar que usuários legítimos usem determinado serviço. Alguns exemplos de ataques incluem: tentativas de "inundar" a rede com pacotes e impedir o tráfego legítimo da rede; tentativas de encerrar uma conexão entre duas máquinas e impedir o acesso a um determinado serviço por uma das máquinas; tentativas de evitar que determinado indivíduo use um serviço; tentativas de interromper serviços para um sistema ou pessoas específicas. Ataques de DoS possuem ser realizados de inúmeras formas e têm como alvo vários tipos de serviços. O tipo de ataque mais comum objetiva o consumo de recursos escassos como largura de banda da rede, memória, espaço em disco e tempo de CPU.

Uma arquitetura de segurança deve possuir quatro componentes essenciais para atenuar as ameaças das técnicas de ataque listadas até agora. O primeiro deles é a **autenticação mútua**, onde o cliente e o servidor autenticam um ao outro; dessa forma o servidor sabe que está abrindo uma sessão com um cliente autorizado e o cliente sabe que está abrindo uma sessão com um servidor legítimo. A autenticação do servidor impede ataques MITM, e a autenticação do cliente torna ataques de Seqüestro de Sessão e por

[10] O protocolo de tunelamento encapsula o pacote com um cabeçalho adicional que contém informações de roteamento que permitem a travessia dos pacotes ao longo da rede intermediária.

[11] http://www.cert.org/tech_tips/denial_of_service.html

Repetição mais difíceis. O segundo componente é o **ciframento de cifra de bloco**[12] (*block cipher encryption* em inglês) dos dados da mensagem, o qual impede ataques de Escuta Passiva. Outro aspecto importante do ciframento é a camada na qual os pacotes são cifrados. O ciframento na camada 2 (enlace de lados) limita a Análise de Tráfego e torna ataques de Escuta Ativa muito mais difíceis. O terceiro componente é a **proteção de integridade** usando criptografia forte, i.e, algoritmos que utilizam chaves maiores, que é essencial para deter ataques de Escuta Ativa. Também possui um papel importante junto com ciframento dos dados da mensagem para impedir ataques de Seqüestro de Sessão e por Repetição. Por último, o uso de ***Firewall*** entre a rede sem-fio e os elementos da rede com fio para impedir o tráfego de rede entre clientes não-autenticados e a rede com fio, sendo um ponto crítico para uma arquitetura segura.

A intenção deste trabalho é validar os mecanismos de segurança usados injetando ataques no sistema de comunicação. A proposta usada para gerar tais ataques é apresentada capítulo 4.

2.4 Modelando Ataques

A modelagem de ataques é usada para descrever as etapas de um ataque bem-sucedido, podendo variar de modelos simples de árvores [17] a métodos baseados em redes de Petri [15]. Nos últimos anos surgiram também os modelos baseados em UML [16], como por exemplo, diagramas de seqüência, casos de uso e diagramas de estado.

O que distingue as abordagens de Modelagem de Ataques de outros métodos de modelagem de aspectos funcionais é que na Modelagem de Ataques o ponto de vista de um atacante é enfatizado. Como os objetivos e os métodos de um atacante possuem um papel central em praticamente todas as considerações de segurança, modelos de ataques acabam se aproximando mais dos problemas de segurança e, portanto são úteis em [14]: **i)** descobrir vulnerabilidades em sistemas novos; **ii)** evitar vulnerabilidades durante o desenvolvimento do software; **iii)** avaliar implementações existentes para vulnerabilidades conhecidas. Neste

[12] Método usado para cifrar blocos de bits (tipicamente 64 ou 128 bits), onde o texto em claro é dividido em blocos de tamanho fixo que são processados, um por vez, com a mesma chave de ciframento, gerando assim os respectivos blocos de texto cifrado [29].

trabalho iremos usar Modelagem de Ataques principalmente para avaliar implementações de protocolos usando vulnerabilidades conhecidas.

Modelos de ataque podem representar: os métodos de ataque gerais e específicos contra um sistema, as propriedades de um sistema e as pré-condições para que um ataque tenha sucesso. Também devido a sua representação gráfica compacta, modelos de ataques não precisam de explicações detalhadas e não exigem conhecimento prévio em métodos de Modelagem de Ataques.

2.4.1 Árvore de Ataques

Árvore de ataques [17] é uma estrutura de dados que pode descrever todos os possíveis ataques a um sistema de uma forma bem organizada para facilitar a análise de segurança. A árvore de ataques representa os passos de um ataque e suas interdependências, e também pode ser usada para representar e calcular probabilidades, riscos, custos ou outras ponderações.

Neste trabalho nós decidimos usar árvores de ataques, pois elas possuem algumas vantagens que as tornam uma boa escolha para a representação de uma base de conhecimento de ataques, e assim sendo apropriadas para nossos propósitos. Essas vantagens foram observadas a partir de trabalhos de análise de segurança que usaram diferentes tipos de modelos (ex.: redes de Petri, modelos UML) para representação de ataques.

- Árvore de ataques foca em objetivos que podem ser transformados em ataques contra a implementação do protocolo;
- Árvore de ataques permite descrever, de uma maneira mais estruturada do que linguagem natural, as ações executadas por um ataque bem-sucedido;
- O modelo é fácil de compreender para pessoas com pouca prática em modelos formais; também é conciso e possibilita uma ampla gama de pessoas contribuírem para sua manutenção;
- A estrutura hierárquica, na qual objetivos em níveis mais altos são divididos em sub-objetivos até que o nível de refinamento (detalhes) desejado seja alcançado,

simplifica a navegação e possibilita várias pessoas trabalharem em diferentes ramos em paralelo.

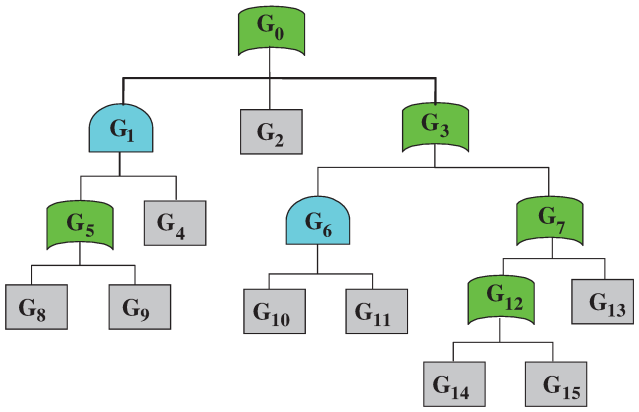
- A estrutura modular permite o reuso da árvore de ataques para representar outros ataques maiores, i.e., uma nova árvore de ataques maior pode usar módulos menores (outras árvores de ataques).
- O projeto modular da árvore permite analistas trabalharem simultaneamente em componentes separados de uma árvore de ataques maior, e depois integrar os múltiplos componentes e completar a árvore de ataques maior.
- O projeto modular torna o processo de criação da árvore de ataques automático e mais fácil, permitindo a atualização dos componentes (ataques) em face de novas tecnologias, e assim possibilitando o uso de bibliotecas de árvores de ataques [18].
- Com base em ataques mais comuns é possível definir padrões de ataque [18] para determinado sistema. Estes padrões podem ser reusados na Validação de outras implementações, mostrando boa praticidade.

Em árvore de ataques, o nó raiz representa a realização do objetivo final do ataque. Cada nó filho representa sub-objetivos que precisam ser realizados para que o objetivo do nó pai tenha sucesso. Nós pais podem estar relacionados com seus filhos por uma relação de tipo **OR** ou de tipo **AND**. Em uma relação do tipo OR, se qualquer um dos sub-objetivos dos nós filhos é realizado então o nó pai é bem-sucedido. Com uma relação do tipo AND, todos os sub-objetivos dos nós filhos devem ser realizados para que o nó pai seja bem-sucedido. As folhas da árvore, i.e., nós que não são mais decompostos, representam ações do atacante. Um atacante só pode influenciar no sistema onde ele pode interagir com ele. Estes pontos de influência são representados exclusivamente pelos nós folhas.

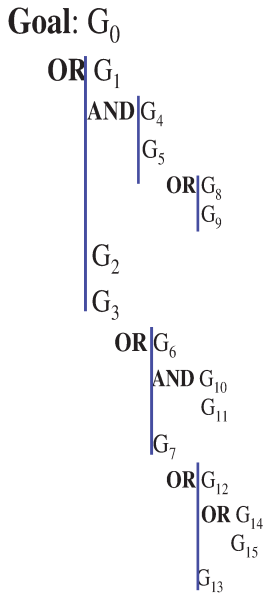
Cenários de ataque individuais podem ser gerados percorrendo a árvore em profundidade, i.e., um cenário de ataque é um caminho da raiz até a folha da árvore, assim gerando uma combinação mínima de eventos das folhas, no sentido que se qualquer nó for omitido do cenário de ataque então o objetivo da raiz não é alcançado. Apenas folhas são mantidas no cenário de ataque, pois elas representam ações a serem executadas para que um ataque tenha sucesso. O conjunto completo de cenários de ataque de uma árvore mostra todos os ataques que estão disponíveis para um atacante que possui recursos “infinitos”.

Fazendo analogia com geração de casos de teste, o objetivo é cobrir todas as ações representadas nas folhas.

Árvores de ataques podem ser representadas gráfica ou textualmente. A Figura 1.2 (a) mostra um exemplo de uma notação gráfica e a Figura 1.2 (b) mostra a notação textual correspondente. A descrição de cada nó é feita em linguagem natural. Na parte de baixo da figura, os cenários de ataque que podem ser gerados a partir da árvore de ataques também são mostrados. A notação <a, b, c> representa um cenário, com as folhas sendo consideradas nesta ordem: a → b → c.



(a) Árvore de ataques gráfica.



(b) Árvore de ataques textual.

Cenários de ataque: <G8, G4>, <G9, G4>, <G2>, <G10, G11>, <G13>, <G14>, <G15>.

Figura 2.2: Árvores de ataques e cenários de ataque.

A idéia básica por trás da abordagem de árvore de ataques é a Análise de Ataques baseada em Capacidade [28], que possui uma hipótese sobre atacantes: “SE eles querem E eles podem ENTÃO eles farão”. Cada nó folha da árvore é associado com um nível aproximado de recursos requeridos para realizar um ataque específico, como custo financeiro, habilidade técnica, tempo. Os recursos necessários em qualquer nó da árvore de ataques são um reflexo direto da complexidade da vulnerabilidade naquele ponto. Essas

métricas de recursos incorporadas no modelo da árvore de ataques determinam a probabilidade dos ataques, pois influenciam no comportamento dos atacantes, já que até um atacante muito motivado só pode executar um ataque se seus recursos satisfazem ou excedem os recursos requeridos para executar um ataque. Usando os atributos associados aos nós como custo, probabilidades ou valores lógicos, é possível selecionar cenários de ataque baseado em um ***critério***, por exemplo, os cenários que são mais prováveis ou os menos custosos. A Figura 1.2 mostra atributos sendo usados nas folhas da árvore representando o custo e um valor lógico (P: possível ou I: impossível) para cada folha.

Para ilustrar o uso de árvore de ataques podemos usar uma versão simplificada do exemplo dado no trabalho de Schneier [17]. O exemplo considera um ataque a um cofre, em que o objetivo é abrir o cofre. A árvore de ataques representando as possibilidades de ataque é mostrada na Figura 1.3.

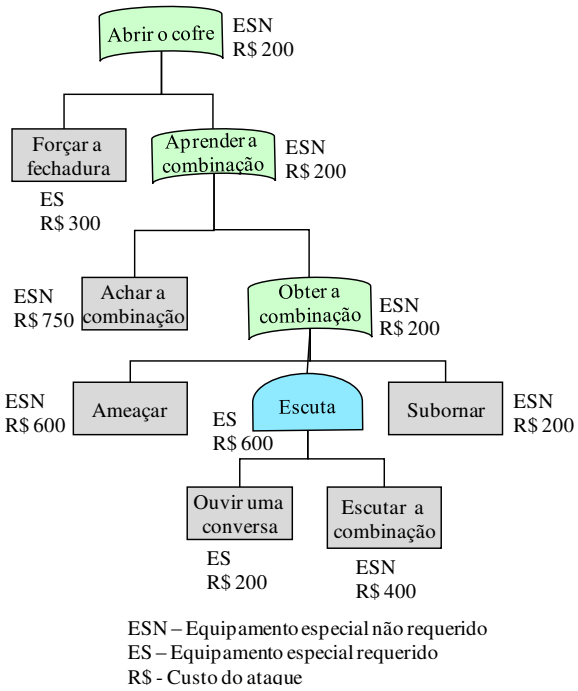


Figura 2.3: Árvore de ataques para abrir o cofre.

Neste exemplo, para alcançar o objetivo final de "Abrir o cofre", o atacante deveria "Forçar a fechadura" **OR** "Aprender a combinação do cofre". Para "Aprender a combinação do cofre" ele deveria "Achar a combinação escrita" **OR** "Obter a combinação do dono do cofre". Uma maneira de "Obter a combinação do dono do cofre" é por

"Escuta", o qual requer um equipamento especial. A "Escuta" deve ser feita de modo que o atacante possa "Ouvir uma conversa" **AND** "Escutar o dono do cofre dizer a combinação". Atacantes não podem alcançar o objetivo a menos que os sub-objetivos sejam satisfeitos.

Na árvore de ataques da Figura 1.3 o cenário de ataque < "Subornar" > é o ataque mais barato (R$ 200) que não requer nenhum equipamento especial. Outro possível cenário é o que já foi explicado < "Ouvir a conversa", "Escutar o dono do cofre dizer a combinação" >, que possui um custo médio (R$ 600) e requer equipamento especial.

Esse modelo de ataques é bastante útil para gerar cenários de ataque para Testes de Segurança, pois os cenários que correspondem a ataques reais podem ser gerados de acordo com os custos ou a freqüência dos ataques na vida real. Também o modelo, que é fácil de estender, possibilita a atualização em caso de descoberta de novos ataques.

2.4.2 Padrão de Ataque

Padrão de ataque (*attack pattern* em inglês) [18] é uma representação genérica de um ataque malicioso e intencional que geralmente ocorre em contextos específicos. Cada padrão de ataque contém: **i)** o *objetivo* do ataque especificado pelo padrão; **ii)** uma lista de *precondições* para que aconteça; **iii)** os *passos* para executar o ataque; **iv)** uma lista de *pós-condições* se o ataque foi bem-sucedido. As *precondições* incluem suposições feitas sobre o atacante ou o estado do sistema que são necessárias para que o ataque tenha sucesso, como por exemplo, habilidades, recursos, acesso, ou conhecimento que um atacante deve possuir, e o nível de risco que ele deve estar disposto a tolerar. As *pós-condições* incluem o conhecimento adquirido pelo atacante e alterações do estado do sistema que resultaram dos *passos* do ataque bem-sucedido.

O tipo mais comum de vulnerabilidade de segurança é o tratamento incorreto de *buffer overflow* [18], que é uma condição anômala onde a aplicação tenta armazenar dados além dos limites de um buffer de tamanho fixo. O padrão de ataque para o *buffer overflow* é representado na Figura 1.4.

Uma árvore de ataques pode ser refinada desde o nó raiz usando uma combinação de extensões manuais e aplicação de padrões de ataque. Extensões manuais dependem enormemente do conhecimento de segurança da pessoa construindo a árvore de ataques. A

aplicação de padrões de ataque também depende de tal conhecimento, mas em menor proporção, pois muito desse conhecimento está implementado na biblioteca de padrões de ataque.

Padrão de Ataque de *Buffer Overflow*:

Objetivo: Explorar a vulnerabilidade de *buffer overflow* para executar uma função maliciosa no sistema alvo.

Precondição: Atacante pode executar programas no sistema alvo.

Ataque:

AND 1. Identificar um programa executável no sistema alvo suscetível a vulnerabilidade de *buffer overflow*.

2. Identificar o código que irá executar a função maliciosa quando ele for executado com privilégio de programa.

3. Construir o valor de entrada que forçará com que o código esteja no espaço de endereço do programa.

4. Executar o programa de uma maneira que ele pule para o endereço onde o código reside.

Pós-condição: O sistema alvo executa a função maliciosa.

Figura 2.1: Padrão de ataque de *buffer overflow*.

Capítulo 3

Abordagens de Validação

Este capítulo aborda os principais métodos de Validação que usam a técnica de Injeção de Falhas. O conceito de Injeção de Falhas é descrito na seção 3.1. O conceito de Testes de Robustez é descrito na seção 3.2.1. O uso de Injeção de Falhas para realizar Testes de Robustez é descrito na seção 3.2.2. Na seção 3.3 são descritas as principais abordagens e ferramentas propostas para realização de Testes de Segurança, onde o objetivo é descobrir vulnerabilidades em aplicações usando a técnica de Injeção de Falhas.

Avaliação de dependabilidade (descrita na seção 2.1) de um sistema envolve o estudo de defeitos e erros, sendo particularmente difícil recriar o cenário defeituoso para um sistema grande e complexo para identificar as causas do defeito no ambiente operacional. Assim para identificar e entender potenciais defeitos do sistema usa-se uma abordagem experimental que objetiva compreender melhor a dependabilidade do sistema. Para usar essa abordagem deve-se primeiro entender a arquitetura do sistema, sua estrutura e seu comportamento, depois é preciso conhecer sua tolerância a falhas e defeitos, incluindo seus mecanismos integrados de detecção e recuperação de erros. Portanto é necessário especificar instrumentos e ferramentas para **injetar falhas**, criar defeitos ou erros no sistema, e monitorar seus efeitos.

3.1 Injeção de Falhas

A Injeção de Falhas (IF) consiste na introdução de falhas ou erros em um sistema alvo com o objetivo de observar o seu comportamento. IF pode ser usada para vários propósitos, como estudar o comportamento do sistema em cenários de falhas, determinar a cobertura dos mecanismos de detecção, isolamento e recuperação de falhas do sistema, avaliar a eficiência e o desempenho do sistema, dentre outros.

IF pode ser divida em três categorias [30]: **i) IF por hardware**: Essa técnica usa um hardware adicional para introduzir falhas no hardware do sistema alvo com o objetivo de verificar a eficácia de mecanismos tolerantes a falhas implementados em hardware. **ii) IF**

por simulação: Esta técnica é usada na fase de projeto e conceitual de um sistema, sendo útil para avaliar a eficácia de mecanismos de tolerância a falhas e a dependabilidade do sistema, onde as falhas são introduzidas num modelo do sistema alvo. **iii) IF por software (SWIFI)**: É uma técnica atrativa, pois não requer um hardware custoso, podendo ser usada para injetar falhas em sistemas operacionais e aplicações alvo. A injeção de falhas pode ser facilmente controlada através da ferramenta de injeção, i.e., possui alta "controlabilidade" (habilidade de controlar a IF no tempo e espaço). Se o alvo é uma aplicação, o injetor de falhas é inserido na própria aplicação ou em uma camada entre a aplicação e o sistema operacional. Se o alvo é o sistema operacional, o injetor de falhas deve ser embutido no sistema operacional, pois é muito difícil adicionar uma camada entre a máquina e o sistema operacional. O foco desta técnica é alterar o estado do sistema alvo, que está sob o controle de um software de injeção, emulando falhas de hardware e software [12] (como classificado na seção 2.1). O software de injeção interrompe ou altera de modo controlado a execução do sistema alvo através de algum mecanismo, e executa o código de injeção, que emula falhas através da inserção de erros em diferentes componentes de acordo com o modelo de falhas usado (ex.: conteúdo de registradores, posição de memória, bits em instruções, ponteiros de endereço, etc.).

Outro ponto importante na IF é o método para acionar (ativar) as falhas. Esses métodos podem ser categorizados com base no momento em que as falhas são injetadas: em **tempo de compilação** ou em **tempo de execução**. Para injetar falhas em **tempo de compilação**, as instruções do programa devem ser modificadas antes que a imagem binária do programa seja lida e executada na memória. Este método injeta erros no código-fonte, ou no código objeto, ou no código de máquina do programa alvo para emular o efeito de falhas de hardware, de software e falhas transientes [12]. Em injetores em **tempo de execução** é preciso um mecanismo para disparar (ativar) a IF – um ativador – e as falhas são injetadas durante a execução do sistema alvo. Ativadores de falhas comuns incluem: **ativadores baseados em tempo**, **ativadores baseados em exceção ou "trap"** e **ativadores baseados em código**. O ativador baseado em tempo usa um temporizador de hardware (que deve ser associado ao vetor de tratamento de interrupção do sistema) ou software, que expira em um tempo pré-determinado e gera uma interrupção para disparar a injeção de falha. No ativador baseado em exceção ou "trap", uma exceção de hardware

(ex.: acesso a um endereço de memória específico) ou uma instrução de desvio de software ("trap") transfere o controle para o injetor de falhas. Ambos os mecanismos (exceção ou "trap") devem ser associados ao vetor de tratamento de interrupção. No ativador baseado em código, instruções são inseridas no programa alvo, as quais permitem que a injeção das falhas ocorra antes de determinadas instruções do programa alvo.

A IF é uma ferramenta experimental importante para a análise de dependabilidade e a Validação da arquitetura e da implementação de protocolos e sistemas distribuídos. Muitas ferramentas de IF por software têm sido desenvolvidas, utilizando diferentes métodos para injetar falhas e testar o tratamento de falhas de protocolos em sistemas distribuídos. Os testes geralmente são executados manipulando o conteúdo e a troca de mensagens entre os nós do sistema alvo. Pode-se chamar isso de **IF baseado em mensagem** [32], o qual é usado em Testes de Robustez de protocolo, discutido na sessão 3.3.

Os modos de defeito assumidos fornecem um modelo de como falhas em diferentes subsistemas (nós de computação, interfaces de comunicação e redes) afetam o sistema distribuído. Um modo de defeito descreve o impacto de defeitos de subsistemas no sistema distribuído. Modos de defeito comumente assumidos incluem [20]: falhas **Bizantinas**, falhas **temporais**, falhas de **omissão** e falhas de **crash**. Falhas de crash podem ser emuladas interceptando todas as mensagens enviadas ou recebidas por um nó específico. Falhas de omissão podem ser emuladas interceptando algumas mensagens enviadas por um nó (falhas de omissão de envio), ou algumas mensagens recebidas por um nó (falhas de omissão de recebimento). Falhas Bizantinas (ou Arbitrárias) podem ser emuladas corrompendo o conteúdo das mensagens, ou enviando mensagens contraditórias a diferentes nós (ex.: duplicar mensagens). Falhas temporais, mais especificamente, falhas de desempenho podem ser emuladas atrasando a entrega de mensagens por um período maior do que o especificado ou adiantando a entrega de mensagens. A nível de sistema, estas falhas de subsistemas correspondem a falhas de implementação ou projeto. Logo, ferramentas de IF baseado em mensagem têm a intenção de injetar falhas que correspondem a diferentes modos de defeito de subsistemas.

O ambiente experimental para algoritmos tolerantes a falhas (EFA – *experimental environment for fault tolerance algorithms*) [31] foi uma das primeiras ferramentas de IF

baseado em mensagem. A ferramenta insere injetores de falhas em cada nó do sistema alvo e pode implementar diferentes tipos de falhas, incluindo omissão de mensagens, envio de uma mensagem várias vezes, geração de mensagens espontâneas, alteração de tempo de envio das mensagens e corrupção de conteúdo das mensagens.

O framework para testar aplicações distribuídas e protocolos de comunicação chamado ORCHESTRA [5] insere uma camada de IF (PFI – *protocol fault injection layer*) entre a camada do protocolo testado e as camadas inferiores da pilha do protocolo, permitindo filtrar e manipular as mensagens trocadas entre os participantes. Mensagens podem ser atrasadas, perdidas, reordenadas, duplicadas, modificadas, e novas mensagens podem ser introduzidas espontaneamente no sistema testado para conduzi-lo a um determinado estado global desejado. ORCHESTRA foi usada em um estudo comparativo de seis implementações comerciais do protocolo TCP.

COMFIRM [6] é uma ferramenta de IF de rede com a pretensão de validar mecanismos de tolerância a falhas de protocolos de redes e sistemas distribuídos. A ferramenta explora a possibilidade de inserir código no núcleo do sistema operacional Linux através de módulos adicionados em baixo nível na pilha de protocolo, usando o framework Netfilter[13] para o processamento das mensagens trocadas entre os participantes. As falhas são injetadas diretamente no subsistema de troca de mensagens, permitindo a definição de cenários de testes a partir de um amplo modelo de falhas, que pode afetar mensagens sendo enviadas ou recebidas. Semelhantemente a ferramenta ORCHESTRA, os tipos de falhas usados são: atraso, duplicação, remoção de mensagens; e alteração de seu conteúdo.

O uso de IF por software para obtenção de medidas tais como cobertura de falhas e avaliação do mecanismo de tolerância a falhas em sistemas, tem sido proposta e pesquisada por décadas. A maior preocupação é garantir que as falhas injetadas representem falhas reais, pois é a condição necessária para obter resultados significativos. Como explicado até agora, típicas propostas incluem a inserção de erros em nível de código fonte e a emulação de falhas externas [33] (como classificado na seção 2.1). Contudo IF também pode ser usada para avaliar a robustez do software, inserindo artificialmente condições e entradas

[13] http://www.netfilter.org/documentation/

excepcionais, e observando a resposta e o comportamento do sistema que está sob essas condições. Propostas tradicionais de IF por software de protocolos e sistemas de comunicação são provadamente efetivas em capturar defeitos de software usando mensagens trivialmente alteradas, perdidas, duplicadas ou atrasadas, que simulam problemas de transmissão. Contudo ataques ativos de segurança em protocolos de comunicação de rede são mais difíceis de simular usando esses métodos, devido a sua constante relevância com o modelo de comportamento do protocolo e suas condições específicas de execução.

3.2 Testes de Robustez

Robustez é um atributo especializado de dependabilidade (explicada na seção 2.1) que mede o comportamento de um sistema sob condições errôneas. Na seção 3.2.1 são apresentados alguns conceitos relacionados à robustez. Na seção 3.2.2 são descritas as principais abordagens para realização de Testes de Robustez.

3.2.1 Conceitos

Robustez é definida no padrão IEEE 610.12.1990 como o grau no qual um sistema opera corretamente na presença de entradas excepcionais ou condições ambientais estressantes [34], como por exemplo, indisponibilidade de recursos do sistema, falhas de comunicação, entradas inválidas.

Defeitos de robustez [34] são tipicamente falhas de projeto ou de programação que são ativadas por essas entradas excepcionais e condições estressantes, as quais resultam em uma operação incorreta (ex., aborto, travamento) do sistema. Defeitos de robustez são classificadas de acordo com o critério *C.R.A.S.H.* [36]: **catastrófica** (***C****atastrophic*), onde o sistema inteiro sofre crash ou reinicia; **reinicio** (***R****estart*), onde a aplicação precisa ser reiniciada; **aborto** (***A****bort*), onde a aplicação finaliza anormalmente; **silencioso** (***S****ilent*), onde uma operação inválida é executada no sistema e não ocorre exibição de erro; e **obstrutivo** (***H****indering*), onde um código de erro incorreto é retornado – note que o retorno de um código de erro apropriado é considerado como uma operação robusta.

O domínio de entrada dos testes de robustez pode ser caracterizado por dois conjuntos; a carga de trabalho (*workload* em inglês) que são as entradas para ativar a funcionamento normal do sistema, e a carga de falhas (*faultload* em inglês) que contém as entradas excepcionais e as condições estressantes aplicadas ao sistema. Dependendo de como estas duas cargas são balanceadas, os testes de robustez podem ser usados para propósitos de Verificação ou para Avaliação da robustez.

3.2.2 Abordagens

Muitas abordagens têm sido propostas para automatizar Testes de Robustez de software, sendo que a maioria delas é baseada em IF, i.e., elas consistem em alimentar o sistema sob teste com seqüências de entradas inválidas dentro de um modelo de falhas para revelar a presença de defeitos de robustez. Nesse tipo de abordagem o foco é obter uma **carga de falhas** emulada representativa, i.e, o objetivo fundamental é induzir erros e defeitos que são similares àqueles provocados por falhas de ambientes reais. As abordagens descritas a seguir estão categorizadas conforme a maneira como essas entradas são escolhidas [32]:

- **Uso de entradas aleatórias**: É uma técnica simples, denominada Teste **Fuzz** [37], que consiste na geração de entradas aleatórias para o sistema. A proposta de Teste Fuzz possui três características: 1) toda entrada é aleatória; 2) ela não usa nenhum modelo de comportamento de programa, de tipo de aplicação ou de descrição de sistema; 3) o critério de confiabilidade é simples: se a aplicação aborta (termina anormalmente) ou trava (para de responder depois de um período de tempo razoável), considera-se que o sistema gerou um defeito. Contudo, Teste Fuzz só pode fornecer uma amostra reduzida do comportamento do sistema e em muitos casos o fato do sistema passar em um Teste Fuzz apenas demonstra que parte do sistema trata exceções sem abortar inesperadamente, ao invés de retornar as saídas esperadas pelo usuário ou sistema.

- **Uso de entradas inválidas**: Essa técnica de Testes de Robustez consiste em selecionar valores que são usados nas chamadas de interface, os quais incluem:

valores limites e valores fora do domínio de valores de entrada permitidos. Por exemplo, se um parâmetro da interface aceita um inteiro positivo (*unsigned integer* em inglês), então valores fora do domínio que podem ser selecionados para o testes de robustez são: zero, um número negativo e o valor máximo de inteiro (MAXINT – *maximum integer* em inglês).

- **Testes de tipo específico**: Nessa abordagem entradas válidas e inválidas são definidas para os tipos de dados usados nas funções públicas do sistema, e os testes de robustez são gerados combinando esses valores definidos para os diferentes parâmetros. O tamanho do domínio de entradas inválidas pode ser reduzido usando herança entre os tipos de teste. A ferramenta **Ballista** [36] usou essa abordagem para comparar a robustez de 15 sistemas operacionais POSIX usando um conjunto de testes para 233 chamadas de funções. O objetivo da pesquisa foi implementar métodos para medir a robustez do mecanismo de tratamento de exceções dos sistemas.

Como explicado nas abordagens, a técnica de **IF de interface** tem como principal objetivo "bombardear" a "interface pública" da aplicação, sistema ou rotinas com entradas excepcionais (limites e inválidas) e válidas. O critério de sucesso é na maioria dos casos: se a aplicação não aborta ou trava, então é considerada robusta. Essas abordagens e práticas servem para avaliar o comportamento dos mecanismos de tratamento de entradas excepcionais desses sistemas e fornecem meios quantitativos de medir a robustez do sistema. Cumpre notar que o tratamento inválido dessas entradas tem se mostrado ser a maior causa de defeitos na execução de sistemas [36]. Além disso, essas falhas podem constituir vulnerabilidades de segurança para o sistema afetado, pois em alguns casos podem ser consideradas como falhas de iterações maliciosas fornecidas ao sistema, podendo ativar em algum momento uma vulnerabilidade, como explicado na seção 2.2.

3.3 Testes de Segurança

Validação de segurança de protocolos de rede, que é realizada durante as fases de

desenvolvimento, inclui [12]: i) **Verificação estática**; ii) **Verificação dinâmica**. A primeira tenta localizar falhas inseridas durante o projeto, como um estado não-alcançável, ou uma condição de loop na FSM do protocolo, ou possíveis erros humanos introduzidos no código. Nesse caso são utilizados métodos de análise estática (ex.: inspeção de código, analisadores de vulnerabilidade estáticos), Verificação de Modelo (*Model Checking* em inglês) [3], ou prova de teorema, os quais não necessitam executar o sistema. A segunda foca na Verificação da implementação durante sua execução, i.e, verificar o sistema exercitando seu código, onde entradas reais são fornecidas para verificar os mecanismos de segurança; os Testes de Segurança usando IF se enquadram nessa categoria.

Neste trabalho iremos focar no método de Validação de segurança da implementação do protocolo usando **Verificação dinâmica** (avaliação durante a execução), mostrando como as técnicas de Modelagem de Ataques e IF são usadas para avaliar e verificar a segurança da implementação de protocolos de rede de forma eficiente.

Alguns trabalhos usaram IF para testar a segurança de sistemas sem usar uma metodologia sistemática para gerar cenários de ataque. Thompson et al [4] testaram a segurança de aplicações em ambientes hostis. Falhas foram injetadas durante a execução da aplicação, onde chamadas de sistema feitas pela aplicação ao sistema operacional foram monitoradas e alteradas. Nesse sentido eles emularam o comportamento de ambientes hostis. Contudo eles não mencionam como associar este comportamento a ataques. Outro trabalho apresenta uma arquitetura e o desenvolvimento de um injetor de falhas para testes de firewalls e sistemas de detecção de intrusão (IDSs) [8]. O injetor de falhas fornece funcionalidades para o usuário, como Falsificação de IP (descrito na seção 2.3), para simular ataques ao protocolo TCP.

Teste Fuzz (explicado na seção 3.2.2) e IF têm sido bastante usados para descobrir falhas de segurança e validar propriedades de segurança em aplicações e protocolos durante sua execução. A maioria das vulnerabilidades de segurança, desde *buffer overflows* a ataques de *cross-site scripting*, são o resultado da validação insuficiente dos dados de entrada fornecidos pelo usuário ou sistema. Falhas de segurança encontradas usando Teste Fuzz são freqüentemente severas e poderiam ser exploradas por um atacante real, pois são inseridas através de uma interface pública. Isto se torna mais preocupante já que Teste Fuzz é uma técnica amplamente conhecida e as mesmas ferramentas vêm sendo usadas por

atacantes para explorar sistemas em pleno funcionamento.

A seguir mostraremos e comentaremos algumas ferramentas e abordagens usando como base as informações contidas nos trabalhos publicados sobre elas. Essas ferramentas e abordagens geram ataques usando as técnicas de Fuzzing e Teste de Sintaxe[14], onde, por exemplo, corrompem mensagens do protocolo usando padrões de ataque maliciosos que contêm longas cadeias de caracteres e caracteres inválidos.

Apresentamos primeiramente os Testes de Segurança realizados pelo projeto **PROTOS** [9], desde 1999, que utilizou as técnicas citadas para aumentar a eficiência dos testes em termos do número de vulnerabilidades descobertas. Seu método sistemático possui as seguintes características: **i)** usa a combinação das técnicas de Teste de Sintaxe e IF; **ii)** é realizado de uma maneira **caixa preta** (o testador não possui acesso ao código fonte da aplicação); **iii)** é baseado apenas na especificação do formato das mensagens do protocolo. Sua abordagem de Testes de Segurança utiliza o espaço de entrada definido por várias interfaces de comunicação e formatos de dados, para gerar as entradas dos casos de teste. A técnica consiste na mutação da descrição abstrata em alto nível do comportamento do sistema (o conjunto de iterações corretas), baseada em uma gramática livre de contexto (BNF), para produzir entradas anormais. Os casos de teste são gerados a partir dessa descrição abstrata. Surpreendentemente ele reportou várias vulnerabilidades de segurança em algumas implementações de protocolo, incluindo SNMP, HTTP e SIP, muitas das quais têm estado em uso por anos. Contudo sua eficiência é questionável, pois é gerada uma grande carga de falhas para se ter algum sucesso. Além disso, as vulnerabilidades de segurança são exploradas de forma limitada, pois não há uma relação precisa entre a carga de falhas e as possíveis vulnerabilidades existentes no sistema. Também as vulnerabilidades encontradas são limitadas a falhas de segurança introduzidas durante a implementação do software (como *buffer overflow*), pois a ferramenta considera apenas entradas individuais, ignorando as vulnerabilidades que falhas de projeto, de especificação e de configuração podem causar, as quais deixam o sistema vulnerável a outros tipos de ataques. A ferramenta também não usa modelagem para as entradas geradas para os casos de teste, i.e., não permite a reusabilidade dos testes para outras especificações de protocolos similares e os

[14] Teste de Sintaxe cria entradas para aplicação, usadas nos casos de teste, as quais são baseadas em especificações de linguagens que são entendidas pela interface da aplicação [9].

casos de testes não podem ser portados para outros tipos injetores de falhas.

O projeto **SPIKE** [38] produziu um framework que usa testes automatizados caixa preta de protocolos de rede. SPIKE usa um método de modelagem de protocolo baseado em bloco. Conjuntos de blocos dentro de uma mensagem caracterizam os valores dos tipos de dados dos campos da mensagem, e rotulam cada um desses blocos com o tipo de dado que ele contém. Quando os dados de testes são gerados para a mensagem, os dados de cada bloco da mensagem podem ser restritos a um conjunto de valores que são válidos para este tipo de dado, dessa forma tirando vantagem dos fatores conhecidos do protocolo de rede usado (os tipos de dados dos campos do protocolo), e delimitando o efeito dos fatores desconhecidos, como por exemplo, gerar tipos de dados de um campo da mensagem que seriam rejeitados pelo protocolo. Assim o tamanho do espaço de entradas e os casos de teste são reduzidos "inteligentemente" por um testador. O framework facilita o trabalho de engenheiros em criar PDUs com encapsulamento em multicamada e permite a exploração manual ou sistemática do espaço de entradas do protocolo, fornecendo ao testador uma maneira de encontrar falhas como *buffer overflow*, *overflow* de inteiros e erros de alocação de memória. Esse método de modelagem de protocolo baseado em bloco e sua estratégia de Fuzzing mostram sua utilidade em encontrar vulnerabilidades, as quais foram exploradas dentro de um período de tempo razoável. No entanto o grande número de casos de teste executados reduziram a eficiência. Como pode ser notada, a classe de ataques usada pela ferramenta é restrita, não simulando ataques reais já reportados. Também o framework não usa modelagem para os ataques gerados, e não existe portabilidade dos ataques para outros injetores de falhas.

Xiao et al [39] descrevem um sistema para injetar dados inválidos em protocolos de redes, a fim de revelar vulnerabilidades usando uma abordagem caixa preta. Esse sistema consiste de uma máquina de teste, um gerador de PDU e algumas ferramentas auxiliares. O gerador de PDU objetiva fornecer uma solução sistemática para uma criação rápida de casos de teste, o qual é baseado na linguagem descritiva *Strengthened* BNF (SBNF), criada a partir da extensão da notação BNF – *Augmented* BNF (ABNF)[15] – com novas capacidades para mutação da especificação do protocolo (sintaxe, conteúdo e seqüência) e é

[15] ABNF á capaz de descrever um protocolo de forma a combinar seu fluxo de dados em elementos funcionais. Fonte: www.ietf.org/rfc/rfc2234.txt.

baseado na técnica IF. A máquina de testes, que executa os testes, é capaz de injetar falhas em camadas que usam os protocolos IP/TCP/UDP. Os protocolos *multicast* IGMP e PIM-DM foram usados como estudo de caso e alguns defeitos relacionados à segurança foram reportados. Contudo não existe relação dos casos de teste com ataques reais já reportados e a classe de ataques injetada é restrita. As saídas do gerador de PDU, que são usadas para criar os casos de teste, não são modeladas e os casos de teste criados não podem ser portados para outros injetores de falhas. Nos experimentos, um grande número de casos de teste foi gerado, com ou sem mutação de PDU, os quais demandaram um grande tempo de experimentação, e o número de vulnerabilidades encontradas nos estudos de caso foi baixa, i.e., baixa eficiência.

Allen et al [11] descreveram uma abordagem sistemática para testes automatizados de protocolos de redes (ex.: protocolos usados para comunicação entre cliente-servidor). A técnica é baseada em métodos de teste caixa preta, como testes baseados em modelos e IF, melhorados pela geração de casos de teste baseados em uma semântica que fornecem uma cobertura mais completa do espaço de código. O protocolo em teste é modelado usando uma FSM que descreve comunicações válidas entre o cliente e o servidor. "Padrões de teste" criados a partir da FSM são associados com dados de entrada "inteligentes", i.e, dados que foram baseados no modelo da sintaxe das mensagens trocadas (os valores das mensagens são gerados usando a mesma técnica baseada em bloco de SPIKE [38] e Fuzzing), dessa forma gerando um conjunto grande de casos de testes executáveis (carga de falhas). A ferramenta foi demonstrada usando diferentes implementações de servidores de FTP para detectar vulnerabilidades de *buffer overflow*. Os resultados dos testes, no entanto mostraram baixa eficiência na detecção de vulnerabilidades. Os dados de entrada dos casos de testes não são baseados em ataques reais, as classes de ataque geradas são restritas (apenas *buffer overflow*), e também não existe modelagem para os cenários de ataques e portabilidade para diferentes injetores de falhas.

Hsu et al [7] propuseram uma abordagem baseada em modelo para detecção de falhas de segurança de implementações de protocolo. A abordagem possui os seguintes passos: primeiro, a abordagem sintetiza um modelo de comportamento abstrato da implementação do protocolo; em seguida usa o modelo para guiar os passos dos testes para detecção de falhas de segurança. O método foi implementado e aplicado em protocolos de

rede reais, e revelou novos problemas de segurança ao quebrar implementações do protocolo *Microsoft MSN instant messaging* (MSNIM)[16], mostrando ser precisa para descobrir vulnerabilidades. Contudo a injeção de entradas não é baseada em ataques reais, e sim no método Fuzz, o qual seleciona os dados de entrada, dessa forma restringindo a abrangência das classes de ataque. Esse método é baseado no modelo de comportamento formal do protocolo, o qual usa as mensagens trocadas entre as duas partes, e não usa modelagem para os ataques selecionados. Não existe, portanto, reusabilidade e portabilidade. Pelo fato de usar um número menor de funções de Fuzzing de mensagens do que as outras ferramentas citadas até agora, a carga de falhas gerada não é considerada excessivamente alta, melhorando a eficiência da injeção e a duração dos experimentos.

A maioria dos trabalhos citados até agora usam a sintaxe da interface do protocolo e métodos baseados em Fuzzing para gerar ataques, ou uma evolução desses métodos, melhorando a eficiência dos testes, como a técnica de geração de entradas baseado em bloco introduzida por SPIKE [38] e depois melhorada pela abordagem de Allen et al [11], bem como o trabalho mais recente de Hsu et al [7] que usa um modelo de comportamento abstrato da implementação junto com funções de Fuzz otimizadas. Ainda assim a maioria dos casos de teste é gerada de forma aleatória, o qual demanda um grande número de experimentos usando IF (carga de falhas) para detectar vulnerabilidades. Também existem várias outras ferramentas de Fuzzing caixa preta (*Fuzzers*[17]) implementadas para teste de protocolos de rede, mas que geralmente são projetadas para um tipo de protocolo específico (ex.: TCP/IP, HTTP, P2P, padrão IEEE 802.11, dentre outras) e de uma maneira ad hoc, sendo que a seleção das mensagens de entrada para fazer a mutação pode ser feita aleatoriamente ou manualmente.

É sabido que, para abordagens baseadas em carga de falhas, i.e., abordagens onde o planejamento dos experimentos é focado apenas na carga de falhas, é bastante comum reportar uma grande proporção (cerca de 30% ou até mais em alguns casos) de experimentos de IF que não resultam em nenhum efeito perceptível (não geram nenhum defeito) [40]. Do ponto de vista de testes, estes experimentos podem ser considerados como

---

[16] O protocolo MSNIM é um protocolo proprietário baseado em texto desenvolvido para os serviços de mensagem do Microsoft MSN [7].

[17] http://www.mixro.com/?Fuzz_Testing_Tools_and_Techniques

"testes não-significativos". Para melhorar a eficiência em Testes de Segurança é necessário que algum conhecimento do sistema alvo esteja disponível, e também informações de tipos de ataques reais mais freqüentes contra sistema estejam disponíveis.

A ferramenta **AJECT** [10] usa a especificação da interface de um protocolo de comunicação de um servidor para gerar automaticamente um grande número de ataques. Enquanto a ferramenta executa os ataques na rede, ela também monitora o comportamento da aplicação alvo (ex.: servidor) da perspectiva do usuário (ex.: cliente) e dentro do sistema alvo. A observação de um comportamento incorreto indica que um ataque foi bem-sucedido e existe uma vulnerabilidade. AJECT realizou um considerável número de experimentos com vários servidores IMAP usando três tipos de teste: Teste de Sintaxe, teste de valor (como "Testes de tipo específico" descritos na seção 3.2.2) e teste de exposição de informação (*Information Disclosure* em inglês, que tenta obter informações privadas no disco ou na memória do sistema alvo) a partir de ataques reais reportados (ex.: *buffer overflow*). Mas a abrangência dos ataques injetados (classes de ataques) ainda se mostrou restrita, se limitando aos ataques já citados. Os resultados mostraram que AJECT pode descobrir vários tipos de vulnerabilidades já reportadas, bem como algumas vulnerabilidades desconhecidas. Contudo o número de ataques injetados pela ferramenta é bastante alto, demandando muito tempo de experimentação. O número de vulnerabilidades encontradas não é expressivo considerando essa enorme carga de falhas, i.e, a eficiência da ferramenta é baixa. Apesar de AJECT se basear em ataques reais ela não usa modelagem para os ataques selecionados, então os cenários de ataque testados não podem ser reusados em protocolos similares e com outros injetores de falhas.

Nos trabalhos descritos até agora não é mencionado como modelar os ataques que são usados durante os testes. De fato a geração automática de casos de teste para Testes de Segurança baseado em modelos com a finalidade de encontrar vulnerabilidades de segurança continua ainda sendo um desafio. A seguir é apresentado um trabalho mais recente que caminha nesse sentido.

Wang et al [43] propuseram uma abordagem de Testes de Segurança dirigida por modelo, onde as ameaças (ataques) são modeladas usando diagramas de seqüência de UML, o qual facilita a reusabilidade. A partir do modelo de ataque é extraído um conjunto de eventos, o qual é uma seqüência de mensagens que representa o comportamento do

ataque e não deveria ocorrer durante a execução normal do sistema. O mesmo modelo de ataque é também usado para decidir qual tipo de informação deveria ser coletada em tempo de execução, e guiar a instrumentação do código fonte para gerar uma seqüência de eventos que representa o comportamento do sistema durante sua execução. Portanto existe uma dependência com o código fonte. O código fonte instrumentado é compilado de novo, e executado usando casos de teste gerados aleatoriamente (Fuzzing) como entrada, i.e, um grande número de entradas é necessário para ativar uma vulnerabilidade, o qual compromete a eficiência da abordagem e a duração dos experimentos. Embora seja usado, como estudo de caso, um exemplo simples não-real e apenas um ataque simples de violação de autorização, a abordagem poderia ser aplicada a outras classes de ataques reais já que depende exclusivamente do modelo. Mas o atual uso da abordagem ainda se mostra bastante restrito.

De fato a maioria dos trabalhos apresentados até agora são baseados na técnica de Fuzzing e Teste de Sintaxe. Apenas o trabalho de Wang et al [43] usa a técnica de Modelagem de Ataques, que poderia ser usada para modelar diferentes classes de ataques para a geração de casos de teste, os quais poderiam ser portados para outras ferramentas. Mas o trabalho de Wang et al também usa Fuzzing para gerar os dados de entrada, produzindo uma grande carga de falhas para o sistema alvo. A ferramenta AJECT [10], que usa ataques reais para produzir os casos de teste, gera as entradas dos casos de teste usando principalmente a técnica Fuzz, o qual também produz uma grande carga de falhas para o sistema alvo. Os outros métodos citados não possibilitam a geração de casos de teste a partir de diferentes classes de ataques reais, pois se baseiam exclusivamente na mutação da especificação do protocolo para revelar erros de implementação de software. Também os casos de teste gerados não podem ser reusados em implementações similares usando outros injetores de falhas.

Nossa proposta tem como objetivo gerar casos de teste baseados em diferentes classes de ataques reais (descritos na seção 2.3), os quais são previamente modelados, variando o mínimo possível de parâmetros, e assim diminuindo drasticamente a carga de falhas e aumentando a eficiência da metodologia. O objetivo é aumentar a "controlabilidade" e reduzir o tamanho da campanha de testes sem reduzir a eficiência dos resultados, i.e, detectar o máximo possível de vulnerabilidades a partir dos cenários de

ataque injetados. Para alcançar esses objetivos é necessário que algum conhecimento do sistema alvo esteja disponível, i.e, informações de ataques contra o sistema estejam disponíveis. Dessa forma, a carga de falhas é gerada a partir de cenários de ataques reais, os quais foram previamente selecionados de uma base de dados de ataques catalogados. É também desejável que essa carga de falhas gerada possa ser reusada para implementações similares podendo ser implementado por diversos injetores de falhas. Assim o cenário de ataque é modelado antes de ser convertido na carga de falhas, possibilitando que a carga de falhas gerada a partir do modelo seja usada com o mínimo de modificações em outros sistemas e seja implementada por diferentes injetores de falhas. A carga de falhas também pode ser usada variando seus parâmetros de ataque, dessa forma, gerando novas variações (instâncias) do ataque modelado, as quais podem ser usadas para encontrar novas vulnerabilidades na implementação. Técnicas de variações de ataque são usadas para testar e avaliar modelos de detecção de intrusão (assinaturas) de IDSs, i.e., testar se instâncias de um ataque podem ser identificadas pelo IDS da mesma forma que o ataque original [35].

Nossa abordagem se baseia em ataques reais e modelos desses ataques reais, o que permite o reuso da carga de falhas, permitindo também usar outros injetores. A Tabela 3.1 mostra as características das abordagens e ferramentas descritas nessa seção que realizam Testes de Segurança por IF.

Abordagem / Ferramenta	Ataques reais	Classe de ataques	Acesso ao código fonte	Reusabilidade (Modelagem dos ataques)	Fuzzing / Sintaxe	Eficiência (vulnerabilidades / carga de falhas)	Carga de falhas (ataques)	Duração dos experimentos	Portabilidade (outros Injetores de falhas)
PROTOS	Não	Restrita	Não	Não	Sim	Baixa	Grande	Alta	Não
SPIKE	Não	Restrita	Não	Não	Sim	Baixa	Grande	Média	Não
Xiao et al	Não	Restrita	Não	Não	Sim	Baixa	Grande	Alta	Não
Allen et al	Não	Restrita	Não	Não	Sim	Baixa	Grande	Alta	Não
Hsu et al	Não	Restrita	Não	Não	Sim	Média	Médio	Média	Não
AJECT	Sim	Moderada	Não	Não	Sim	Baixa	Grande	Alta	Não
Wang et al	Não	Moderada	Sim	Sim	Sim	Baixa	Grande	Alta	Sim
Abordagem do trabalho	Sim	Ampla	Não	Sim	Não	Alta	Baixa	Baixa	Sim

Tabela 3.1: Características das abordagens e ferramentas.

Capítulo 4

Geração e Injeção de Ataques

Uma das dificuldades em realizar testes para encontrar vulnerabilidades no sistema alvo é a coleta de cenários de ataque apropriados para os testes. Cenários de ataque podem ser obtidos usando diferentes fontes (Internet, livros, etc.). Contudo é difícil encontrar os ataques relevantes e leva tempo para automatizá-los de acordo com o ambiente de teste. Nosso propósito é fornecer a um testador de segurança: **i)** meios para gerar cenários de ataques contra o sistema alvo, no caso protocolos de comunicação; **ii)** meios para injetar esses ataques usando injetores de falhas disponíveis, portanto uma abordagem caixa preta. Dessa forma evita-se que o testador forneça à implementação do protocolo entradas totalmente aleatórias, como as técnicas de Fuzz e Teste de Sintaxe o fazem, dessa forma melhorando a eficiência dos testes e diminuindo o tempo de execução da campanha de testes. A Figura 4.1 ilustra as etapas da abordagem apresentadas no restante desse capítulo.

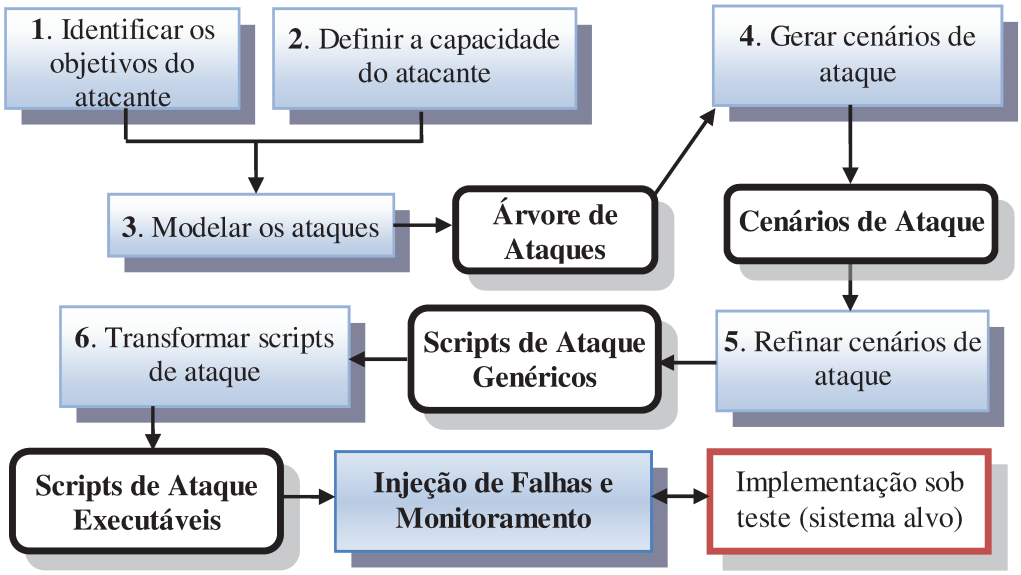


Figura 4.1: Metodologia proposta para Testes de Segurança.

As seções a seguir descrevem cada etapa da metodologia mostrada na Figura 4.1.

4.1 Identificação dos Objetivos do Atacante

O conhecimento sobre vulnerabilidades tanto como ataques reais bem-sucedidos é necessário para definir os ataques a serem injetados. Estas informações estão disponíveis em diferentes fontes sobre segurança, como por exemplo, NIST – *National Vulnerability Database* (NVD)[18], SANS *Institute*[19], CERT *Coordination Center*[20], artigos, etc., e servirão de base para modelar os ataques na etapa **3** (seção 4.3).

Por exemplo, o banco de vulnerabilidades do NIST (NVD), que atualmente é considerado um dos repositórios mais abrangentes e completos de informação pública sobre vulnerabilidades em sistemas computacionais e redes, fornece: **i)** informações gerais de uma vulnerabilidade: data de publicação, data da última revisão, **descrição**, versões de software vulneráveis, *patches* e recomendações fornecidas por empresas; **ii)** métricas de "explorabilidade" e de impacto que descrevem características inerentes de uma vulnerabilidade: alcance do ataque (local ou na rede), complexidade do ataque (baixa, média, ou alta), nível de autenticação necessário (nenhuma, única, ou múltiplas), **propriedades de segurança** violadas (*confidencialidade*, *integridade*, ou *disponibilidade*); **iii)** métricas que descrevem o efeito da vulnerabilidade dentro do ambiente de uma organização: potencial das perdas (baixa, média ou alta), porcentagem de sistemas vulneráveis (baixa, média, ou alta), nível dos requisitos de segurança impactados (baixa, média, ou alta); **iv)** métricas que descrevem elementos da vulnerabilidade dependentes do tempo: disponibilidade da vulnerabilidade (teórica, ou prática), tipo de correção disponível (oficial, temporária ou indisponível), nível de certeza que ela existe (confirmado, ou não-confirmado).

Nesse trabalho iremos usar as seguintes informações do ataque: 1) "**descrição** do ataque"; 2) "**propriedades de segurança** violadas pelo ataque". Sendo assim, o que temos que fazer é: **i)** coletar essas informações das diferentes classes de ataques relacionadas ao protocolo alvo; **ii)** categorizar esses ataques de acordo com cada **propriedade de segurança** violada. As técnicas de ataque mais comuns e as propriedades de segurança

[18] http://nvd.nist.gov.

[19] http://www.sans.org/top20/.

[20] http://www.cert.org/.

violadas por elas foram descritos na seção 2.3. Este passo é útil porque torna mais fácil determinar as propriedades de segurança violadas durante a análise dos resultados experimentais que é feita capítulo 6.

4.2 Definição da Capacidade do Atacante

Um modelo de intruso comum, que é implementado na maioria das abordagens existentes de Verificação Estática da Segurança, é o modelo de intruso de **Dolev-Yao** [41], onde se supõe que o intruso possui todos os meios para interferir na rede e pode capturar o tráfego de rede desejado para análise. Supõe-se também que o intruso possui tempo ilimitado para atacar a rede, e que suas capacidades, em termos de memória disponível e tempo de processamento, também são ilimitadas para quebrar o sistema. Além disso, o intruso pode interceptar ou emitir mensagens, dividir ou constituir mensagens, e pode regenerar mensagens, mas ele não pode evitar que participantes legítimos recebam mensagens. Também assume se que as funcionalidades criptográficas usadas no protocolo são perfeitas, i.e., ataques de criptografia não são possíveis de serem executados.

Portanto, baseado no modelo de Dolev-Yao, consideramos que o atacante possui a seguinte **capacidade**:

- Controle total da rede, podendo capturar o tráfego de mensagens para simples análise.
- Capacidade de **interceptar**, **corromper** (modificar), **remover**, **atrasar** ou **duplicar** (replicar) mensagens do tráfego. Também possui a capacidade de enviar mensagens para qualquer participante (**personificar** um participante);
- Conhecimento do estado de todos participantes, i.e., ele/ela sabe quais mensagens um participante espera receber e qual mensagem ele enviará depois de receber determinada mensagem.

A capacidade do atacante atual pode ser atualizada ou redefinida de acordo com as necessidades do ambiente de teste usado ou devido a outras necessidades específicas.

4.3 Modelagem dos Ataques

Uma vez que as ameaças ao sistema de comunicação foram identificadas, um método é necessário para especificá-las, a fim de que os ataques possam ser obtidos de preferência automaticamente. Nesta etapa nós usamos árvore de ataques para essa modelagem, pois ela é apropriada para nossos propósitos, devido às razões listadas na seção 2.4.1. A partir das informações de ataques disponíveis em linguagem natural, obtidas na etapa **1** (seção 4.1), são definidos os requisitos de ataque para construir a Árvore. A abordagem que usamos para analisar e sistematizar as informações de ataques é a mesma usada no trabalho de Edge et al [1], pois o objetivo dessa etapa é produzir a árvore de ataques.

Não existem orientações específicas para construir árvores de ataques. Em nossa abordagem nós propomos que a **Árvore de Ataques** seja construída de uma maneira topo-base, i.e, começa-se pela raiz até chegar às folhas, para facilitar a categorização dos ataques de acordo com as propriedades de segurança identificadas na etapa **1** (seção 4.1) e dividir os ataques de acordo com os mecanismos e elementos do sistema alvo explorados. Os passos para a construção da árvore são:

1. O nó raiz representa o objetivo genérico final de todos os ataques, que é obter sucesso contra a implementação do protocolo alvo. Portanto o nó raiz é do tipo **OR**.
2. O segundo nível representa as "**propriedades de segurança** violadas pelos ataques", de acordo com a categorização feita na etapa **1** (seção 4.1). Cada nó desse nível representa uma propriedade.
3. O terceiro nível representa os **mecanismos** explorados pelo atacante, como por ex.: troca-de-chaves (*key exchange* em inglês), truncamento de mensagens, entre outros, para violar as propriedades de segurança do nível 2. Essa informação é obtida da "**descrição** do ataque". Se a "**descrição** do ataque" não disponibiliza essa informação esse nível é omitido.
4. Níveis subseqüentes representam as "**descrições** dos ataques" ou passos da descrição de um ataque, que foram identificados na etapa **1** (seção 4.1), para realizar os sub-objetivos do nível 3.
5. Por último são associados atributos a cada nó folha, i.e., valores lógicos de acordo

com a capacidade do atacante definida na seção 4.2.

Esse modelo pode ser atualizado na medida em que novos ataques são descobertos. Outros atributos relacionados às informações coletadas das vulnerabilidades na etapa **1** (seção 4.1) também podem ser considerados, como complexidade do ataque ou freqüência do ataque.

4.4 Geração de Cenários de Ataque

Nesta etapa os cenários de ataque são produzidos de forma automática segundo um ***critério***. Conforme, explicado na seção 2.4.1, o *critério* indica quais atributos associados aos nós devem ser considerados para realizar a busca na árvore, de forma a cobrir todos os ataques que satisfaçam a esses atributos. O *critério* usado foi "cobrir todos os cenários possíveis de acordo com a capacidade do atacante", onde foram considerados os atributos da capacidade do atacante atribuídos na etapa **3** (seção 4.3) para a seleção dos cenários.

Esta etapa é completamente automatizada, pois a ferramenta que auxilia na construção da Árvore de Ataques também seleciona os cenários de ataque. A saída dessa etapa são **Cenários de Ataque** descritos no mesmo formato das folhas da árvore, que possuem a "**descrição** do ataque". Os cenários obtidos podem ser usados para criar uma biblioteca de ataques, o qual é útil para testar outros protocolos da mesma família, facilitando a reusabilidade.

4.5 Refinamento dos Cenários de Ataque

Os Cenários de Ataque gerados na etapa **4** (seção 4.4) estão descritos em linguagem textual, i.e., no mesmo nível de abstração da Árvore de Ataques da qual eles foram gerados. Esse tipo de descrição é útil para o manuseio por analistas de teste e especialistas em segurança, mas não para serem processados por uma ferramenta.

Nesta etapa, analistas de teste devem realizar um conjunto de passos de refinamentos, i.e., eles devem usar ferramentas e métodos para transformar os cenários em um script adequado para o processamento por ferramentas, até se obter um script de ataque

específico pronto para execução. Essa etapa foi definida para permitir que esses scripts executáveis sejam obtidos de forma automática.

4.5.1 Primeiro Passo – Padrão de Ataque

Para realizar o refinamento, o primeiro passo é descrever os cenários usando uma representação estruturada. Para isso utilizamos o conceito de **padrão de ataques**, descrito na seção 2.4.2, pois é uma estrutura que permite caracterizar o ataque em elementos bem definidos, como passos específicos para realizar o ataque e os requisitos de ataque. Utilizaremos então os seguintes elementos do padrão de ataque: **i)** o *"objetivo do ataque"* que é o Cenário de Ataque em si; **ii)** a *"lista de precondições"* para que o ataque aconteça, que são eventos de rede ou de ambiente para que o ataque seja ativado (ex.: um pacote é enviado do servidor para o cliente), extraídos da "**descrição** do ataque"; **iii)** os *"passos para executar o ataque"* que são tarefas que o atacante deve executar para que ataque tenha sucesso, também extraídos da "**descrição** do ataque". Não é necessário utilizar o último elemento do padrão de ataque – uma *"lista de pós-condições"* se o ataque foi bem-sucedido – para esse passo da etapa de refinamento do Cenário de Ataque.

Para exemplificar utilizaremos o ataque de truncamento do SSL v.3 / TLS v.1 [45], que é realizado durante o fechamento de uma conexão SSL com o objetivo de remover dados. A "**descrição** do ataque" é: o atacante remove o último registro (PDU) de dados de aplicação e o registro (PDU) de Alerta de fechamento de conexão *close_notify*[21] do TLS, ambos enviados do servidor para o cliente; dessa forma parece que a mensagem inteira enviada ao cliente é menor. A "**propriedade de segurança** violada pelo ataque" é a *integridade*. O Cenário para esse ataque seria < *Ataque de truncamento no último registro e no registro close_notify* > que contém a "**descrição** do ataque" e a "**propriedade de segurança** violada pelo ataque". Este Cenário ("**descrição** do ataque") é então detalhado usando os seguintes elementos: **i)** o *"objetivo do ataque"*; **ii)** a *"lista de precondições"*; **iii)** os *"passos para executar o ataque"*. A saída desse passo é o Cenário de Ataque descrito na forma de padrão de ataque na Figura 4.2.

[21] Este registro de Alerta é usado para notificar à outra parte que não serão enviadas mais mensagens na conexão SSL / TLS atual.

Objetivo: Ataque de truncamento no último registro e no registro *close_notify*.

Precondição: Pacotes TLS são enviados do servidor para o cliente.

Ataque:

AND 1. ***Se*** o registro (PDU) é do tipo dados de aplicação.

2. ***Se*** o registro (PDU) é o último da mensagem.

3. Remover o último registro (PDU).

4. ***Se*** o registro (PDU) é do tipo Alerta.

5. ***Se*** o registro (PDU) de Alerta é do tipo *close_notify*.

6. Remover o registro (PDU) *close_notify*.

Figura 4.2: Padrão de ataque para o Cenário de Ataque.

4.5.2 Segundo Passo – Regra ECA

O **padrão de ataques** permite descrever o Cenário de Ataque de uma maneira mais estruturada, mas ainda assim, em linguagem natural. Nesse ponto utilizamos a notação ***e**vento-**c**ondição-**a**ção* (**ECA**) [23] para a representação do cenário descrito na forma de padrão de ataques. A razão de se usar regra ECA é que sua sintaxe é bastante simples, sendo ainda um paradigma poderoso para representar comportamentos reativos usando apenas uma regra, i.e., ações executadas como efeito ou conseqüência de eventos e condições.

A regra ECA possui uma sintaxe genérica que funciona da seguinte forma: **ON** *evento* **IF** *condição* **DO** *ação*. Onde:

- *evento* é um evento externo, como a recepção ou transmissão de uma mensagem específica, e indica quando a regra é ativada. Por exemplo, corresponde ao elemento "**Precondição**" (*"lista de pós-condições"*) do padrão de ataque da Figura 4.2.
- *condição* pode ser estabelecida em termos de: 1) conteúdo de campos da mensagem ou 2) valor de alguma variável de estado [6], que determina em quais casos a regra é ativada. A descrição do ataque no padrão (*"passos para executar o ataque"*) corresponde às restrições para a execução das ações. Na Figura 4.2, são identificadas pelo "***Se***" que precede à sua descrição.

- *ação* é estabelecida levando em consideração a capacidade do atacante definida na etapa **2** (seção 4.2), e descreve as ações a serem executadas se a regra foi ativada. Corresponde às ações realizadas pelo atacante.

Para a descrição das partes ***e**vento-**c**ondição-**a**ção* da regra ECA utilizamos um vocabulário de **palavras-chave**. Esse método é usado na abordagem de Teste Dirigido por Palavras-chave[22] (ou Teste Dirigido por Tabela), onde palavras-chave são determinadas por um testador de acordo com o domínio do software usado para os testes durante a fase de planejamento. Os casos de testes são armazenados em tabelas de dados usando um vocabulário de palavras-chave que descrevem as ações que o testador executa, sendo independentes da ferramenta de testes usada para executá-los.

A Figura 4.3 exibe a tabela de dados da regra ECA, que contém as **palavras-chave** que descrevem cada parte da regra em cada coluna. O domínio de software considerado é um sistema de comunicação, onde participantes trocam mensagens, que é o meio de comunicação.

A parte *ação* pode conter vários passos. A sua sintaxe é estabelecida em relação ao conjunto de **palavras-chave** de ações na tabela da Figura 4.3. As **palavras-chave** de ações foram baseadas na descrição da capacidade do atacante da seção 4.2, onde o domínio é o protocolo de segurança sob teste. As palavras-chave **`personificar.criar(m)`** e **`personificar.env(m)`** significa que o atacante pode enviar uma mensagem para um participante legítimo como se ele também o fosse. Outras palavras-chave não relacionadas à capacidade do atacante também foram definidas: **`armazenar(m.<campo>)`**, para guardar um campo de uma mensagem temporariamente; **`restaurar(m.<campo>)`**, para recuperar um campo de uma mensagem armazenado; **`sinalizar.set(var)`**, para atribuir um valor booleano para uma variável de estado; e **`sinalizar.get(var)`**, para recuperar um valor booleano atribuído a uma variável de estado. No caso de alguma alteração na **capacidade do atacante** da etapa 2 (seção 4.2), deve-se atualizar essas **palavras-chave** de acordo com a capacidade do atacante.

[22] http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

Parte	Palavras-chave
EVENTO	env(A,B,m,<PT>,<IP_A>,<IP_B>,<Po_A>,<Po_B>) rec(B,A,m,<PT>,<IP_B>,<IP_A>,<Po_B>,<Po_A>)
CONDIÇÃO	m.tipo_PDU m.<campo> var protocolo.<prop>
AÇÃO	interceptar(m) corromper(m.<campo>) remover(m) duplicar(m) atrasar(m) personificar.criar(m) personificar.env(m) armazenar(m.<campo>) restaurar(m.<campo>) sinalizar.set(var) sinalizar.get(var)

Legenda:

env – função de envio de mensagem.
rec – função de recepção de mensagem.
A – remetente.
B – destinatário.
m – mensagem enviada ou recebida.
<PT> – protocolo de transporte (ex.: TCP, UDP).
<IP_A> – IP da máquina do remetente.
<IP_B> – IP da máquina do destinatário.
<Po_A> – porta de envio da máquina do remetente.
<Po_B> – porta de destino da máquina do destinatário.
m.tipo_PDU – representação do tipo de PDU da mensagem (ex.:PDU de dados).
m.<campo> – representação de um campo qualquer da mensagem.
var – variável de estado.
protocolo.<prop> – propriedades da especificação do protocolo.

Figura 4.3: Tabela de dados da regra ECA.

Naturalmente, a semântica vai ser seguida: **ON** *evento* **IF** *condição* **DO** *ação*, i.e., a partir do momento que a parte *evento* e a parte da regra são satisfeitas, a regra é ativada, e o a parte *ação* executa as ações do atacante da regra. A partir dessa semântica definiremos

cada regra ECA usando as partes *condição* e *ação*, sendo que parte *evento* é usada para todas as regras. Também várias partes *condição* podem ser agrupadas em uma parte *condição* semanticamente usando os operadores lógicos **AND**, significando que duas ou mais condições devem ser satisfeitas para a execução da regra.

Como exemplo, podemos usar o padrão de ataque da Figura 4.2, o qual é convertido para formato ECA. O elemento "**Precondição**" é mapeado para a parte *evento*, os elementos 3 e 6 (*"passo para executar o ataque"*) são mapeados para a parte *ação*, pois "Remover" – **`remover(m)`** – é um das palavras-chave estabelecidas, e os elementos 1, 2, 4 e 5 são mapeados para a parte *condição*. Também iremos agrupar os dois elementos 1 e 2, e os dois elementos 4 e 5 semanticamente usando o operador AND, pois possuem a parte *condição* mapeada. Dessa forma, são criadas 2 regras ECA que possuem a mesma parte *evento*. As partes dessas regras ECA são então descritas usando as **palavras-chave** da tabela na Figura 4.3, como é mostrado na Figura 4.4. A saída desse passo é o padrão de ataque do Cenário de Ataque descrito no formato ECA usando **palavras-chave**.

Regra 1:

ON *evento*: `env(A,B,m,<PT=TCP>,<IP_A>,<IP_B>,<Po_A>,<Po_B>)`

IF *condição*: (**1.** `m.tipo_PDU == protocolo.<prop=application_data>`)

AND (**2.** `m.<campo=record_sequence> == LAST`)

DO *ação*: **3.** `remover(m)`

Regra 2:

ON *evento*: `env(A,B,m,<PT=TCP>,<IP_A>,<IP_B>,<Po_A>,<Po_B>)`

IF *condição*: (**4.** `m.tipo_PDU == protocolo.<prop=alert>`) **AND**

(**5.** `m.<campo=alert_description> ==`

`protocolo.<prop=close_notify>`)

DO *ação*: **6.** `remover(m)`

Figura 4.4: Formato ECA para o padrão de ataque.

4.5.3 Terceiro Passo – Componentes e Interfaces

O próximo passo é identificar os participantes que implementam a regra ECA e as

palavras-chave descritas na seção 4.5.1, de forma a estruturar o Script de Ataque Genérico (SAG) que é o resultado final dessa etapa. Para este propósito, nossa proposta foi inspirada na abordagem UMLintr [16] (*UML for intrusion specifications*). UMLintr usa estereótipos e *tagged values*[23] para especializar vários diagramas UML, como casos de uso, diagrama de classes, pacotes e diagrama de estados, para o domínio de cenários de intrusão. As classes estereotipadas representam as entidades do ***Atacante***, *Intrusão* e *Vítima*.

O nosso método usa o conceito de componentes [46] ao invés de usar classes, o qual permite o uso de um método caixa preta e facilita a reusabilidade. Isto é mais adequado porque nosso objetivo não é desenvolver um perfil UML para especificar cenários de intrusão e gerar assinaturas para um IDS, como em UMLintr. Nosso objetivo é definir os "elementos" necessários para realizar o ataque. Então ao invés de definir três classes estereotipadas, nós definimos três interfaces: ***Atacante***, *Ataque* e *Vítima*, que representam três componentes que fornecem os serviços (operações) necessários para a estruturação e execução do ataque. Essas interfaces fornecem operações que representam respectivamente: **i)** as ações que o atacante é capaz de executar para realizar o ataque, dessa forma representando a capacidade do atacante definida na etapa **2** (seção 4.2), bem como as **palavras-chave** da Figura 4.3; **ii)** o Cenário de Ataque derivado da Árvore de Ataque; **iii)** o protocolo alvo (sistema alvo). Essas interfaces melhoram a flexibilidade para a construção do SAG, já que elas são independentes da plataforma de teste.

A interface ***Atacante*** é responsável por fornecer todas as operações que representam as **palavras-chaves** da regra ECA que foram identificados no passo anterior na seção 4.5.2. No caso de alterações nessa capacidade ou no componente que o implementa, as operações dessa interface relacionadas às **palavras-chave** da seção 4.5.2 também devem ser atualizadas, para manter a conformidade na etapa de refinamento. A interface *Vítima* representa o protocolo alvo e suas propriedades descritas na especificação. A interface *Ataque* representa o Cenário de Ataque selecionado na etapa **4** (seção 4.4) e irá coordenar os passos do ataque usando as operações das interfaces ***Atacante*** e *Vítima*. A Tabela 4.1 mostra as interfaces criadas e suas operações identificadas.

---

[23] *tagged value* é uma combinação de uma *tag* e um valor para fornecer informação suplementar que é anexada a um elemento do modelo. Um *tagged value* pode ser usado para adicionar propriedades a qualquer elemento do modelo. Fonte: http://resource.visual-paradigm.com.

Interface	Parte	Operações

Atacante	*evento*	`getProtocolType: Protocol_Type`
		`getSourceIP: IP_Address`
		`getDestinationIP: IP_Address`
		`getSourcePort: Port`
		`getDestinationPort: Port`

	condição	`getPDUType: PDU_Type` `getPDUField: PDU_Field`

	ação	`InterceptPacket: Packet`
		`CorruptPDUField: Action`
		`DropPacket: Action`
		`DuplicatePacket: Action`
		`DelayPacket: Action`
		`CreatePacket: Packet`
		`SendPacket: Packet`
		`StorePDUField: PDUField`
		`RestorePDUField: PDUField`
		`SetFlag: Boolean`
		`GetFlag: Boolean`

Vítima		`getProperties: Protocol_properties`
Ataque		`mainAttack`

Tabela 4.1: Interfaces e métodos.

As partes da regra ECA definidas na seção 4.5.2 estão diretamente relacionadas com as operações das interfaces ***Atacante*** e *Vítima* na Tabela 4.1, da seguinte forma:

- *evento* (parte **ON**) é mapeado para as operações de análise de pacotes da interface ***Atacante***, as quais identificam mensagens específicas no nível da camada de transporte.
- *condição* (parte **IF**) é mapeada para as operações `getPDUType` e `getPDUField` de extração de dados nas mensagens sendo transportadas (conteúdo de algum

campo do protocolo) que são comparados com as propriedades do protocolo extraídas pela operação `getProperties` da interface *Vítima*.

- *ação* (parte **DO**) é mapeada para as operações da interface ***Atacante*** relacionadas a capacidade do atacante definida na etapa **2** (seção 4.2).

Cada operação das interfaces ***Atacante*** e *Vítima* na Tabela 4.1 são então mapeadas para as **palavras-chave** definidas na seção 4.5.2, da seguinte forma:

Palavras-chave	Operações
env(...,<PT>,...),rec(...,<PT>,...)	getProtocolType: Protocol_Type
env(...,<IP_A>,...),rec(...,<IP_A>,...)	getSourceIP: IP_Address
env(...,<IP_B>,...),rec(...,<IP_B>,...)	getDestinationIP: IP_Address
env(...,<Po_A>,...),rec(...,<Po_A>,...)	getSourcePort: Port
env(...,<Po_B>,...),rec(...,<Po_B>,...)	getDestinationPort: Port
m.tipo_PDU	getPDUType: PDU_Type
m.<campo>	getPDUField: PDU_Field
protocolo.<prop>	getProperties: Prot_properties
interceptar(m)	InterceptPacket: Packet
corromper(m.<campo>)	CorruptPDUField: Action
remover(m)	DropPacket: Action
duplicar(m)	DuplicatePacket: Action
atrasar(m)	DelayPacket: Action
personificar.criar(m)	CreatePacket: Packet
personificar.env(m)	SendPacket: Packet
armazenar(m.<campo>)	StorePDUField: PDUField
restaurar(m.<campo>)	RestorePDUField: PDUField
sinalizar.set(var)	SetFlag: Boolean
sinalizar.get(var)	GetFlag: Boolean

Tabela 4.2: Relação das **palavras-chave** com as operações da interface ***Atacante***.

Nesse ponto as regras ECA foram criadas e descritas usando **palavras-chave** no passo anterior. Os participantes (interfaces) também foram identificados. Agora devemos usar as operações da interface ***Atacante*** mapeadas para as **palavras-chave** da seguinte

forma:

- Para cada **palavra-chave** da parte **DO** (*ação*) está associada uma operação da interface ***Atacante*** correspondente, como pode ser visto na Tabela 4.2.
- Para a **palavra-chave** `m.tipo_PDU` da parte **IF** (*condição*) iremos utilizar a operação `getPDUType`, pois a condição da regra está relacionada a um tipo específico de PDU da especificação do protocolo alvo; ou a operação `getPDUField` para a **palavra-chave** `m.<campo>`, pois a condição está comparando campos do protocolo alvo com valores fixos da especificação ou valores pré-fixados.
- Para as **palavras-chave** `env(...)` e `rec(...)` da parte **ON** (*evento*) iremos usar a operação `getSourceIP` ou a operação `getDestinationIP` caso o acionamento do ataque dependa de uma mensagem originada de um host com endereço específico (parâmetro `<IP_A>`) ou endereçada para um host com endereço específico (parâmetro `<IP_B>`) respectivamente; a operação `getSourcePort` ou a operação `getDestinationPort` caso o acionamento do ataque dependa de uma mensagem sendo originada de uma aplicação específica (parâmetro `<Po_A>`) ou enviada para uma aplicação específica (parâmetro `<Po_B>`), (ex.: servidor Web); e a operação `getProtocolType` para selecionar mensagens sendo transportadas por determinado protocolo (ex.: apenas analisar pacotes TCP) ou quando o acionamento depender apenas de um tipo específico de protocolo de transporte (parâmetro `<PT>`). Também sempre que a parte **ON** depender de algum evento do tráfego da rede (ex.: mensagens trocadas), como é o nosso caso, a **palavra-chave** `interceptar(m)` (operação `InterceptPacket`) é executada implicitamente, pois não podemos satisfazer a parte **ON** sem antes capturar as mensagens do tráfego de rede.

Por fim estruturaremos a implementação do SAG mostrado na Figura 4.5 da seguinte forma:

i. Primeiro usaremos a operação `mainAttack` da interface *Ataque*, o qual coordenará

os passos do ataque através das operações das outras duas interfaces.

ii. Depois iremos chamar a operação `getProperties` da interface *Vítima* para obter os valores fixos da especificação do protocolo que serão necessários para as **palavras-chave** `protocolo.<prop>` usadas nas operações identificadas para as partes **ON** (ex.: o valor da porta padrão que um servidor Web recebe as requisições) e *condição* (ex.: o valor que identifica uma PDU de controle) das regras do ataque. Esses valores fixos da especificação serão os atributos fixos (constantes) do SAG ao longo da execução do ataque.

iii. Em seguida iremos estabelecer a lógica do ataque usando as operações da interface ***Atacante*** identificas para as **palavras-chave** de cada regra do ataque, seguindo a semântica: **ON** *evento* **IF** *condição* **DO** *ação*.

1. Primeiro as operações relacionadas às **palavras-chave** da parte **ON** serão chamadas (não se esquecendo de chamar a operação `InterceptPacket`), que é o tratador inicial de todos as regras estabelecidas;

2. Depois as operações relacionadas às **palavras-chave** da parte **IF** serão chamadas para verificar as condições;

3. Em seguida as operações relacionadas às **palavras-chave** da parte **DO** serão chamadas para executar a ação do atacante.

A Figura 4.5 mostra a implementação do SAG estruturado de acordo com os passos explicados. Esse SAG é uma classe que implementa a interface *Ataque*. A implementação das outras interfaces é explicada na etapa **6** (seção 4.6).

A saída desta etapa é o **Script de Ataque Genérico (SAG)**, que é o Cenário de Ataque descrito com base nas interfaces dos componentes da Arquitetura de Teste. O Script de Ataque é preciso o suficiente para ser processado automaticamente e ser convertido em um script específico para uma ferramenta. Esse script pode ser reusado para testes de outras implementações baseadas em especificações similares a do protocolo alvo. No caso de reuso, apenas os parâmetros específicos à especificação do protocolo serão alterados no SAG.

```
1. class Script_Ataque implements Ataque {
2.
3.  mainAttack() {
4.
5.   Packet pacote;
6.   propriedades = <>.getProperties(protocolo_alvo);
7.   while (pacote = <>.InterceptPacket(interface)) {
8.    /* operações de evento */
9.    Regra_1:
10.    /* operações de condição */
11.    /* operações de ação */
12.   Regra_2:
13.    /* operações de condição */
14.    /* operações de ação */
15.   …
16.   Regra_n:
17.    /* operações de condição */
18.    /* operações de ação */
19.    return;  /* Fim do ataque */
20.}}}
```

Figura 4.5: Implementação do Script de Ataque Genérico (SAG).

4.6 Transformação dos Scripts de Ataque Genéricos

Esta etapa refere-se à transformação das operações do SAG, que estão em um nível mais abstrato, para uma linguagem específica dos componentes usados para os testes.

Para implementar a **capacidade do atacante** (intruso) definida na etapa **2** (seção 4.2) e emular as tentativas deste de explorar uma falha de segurança no sistema alvo, são usados **injetores de falhas**, como por exemplo, injetores de falhas de comunicação, os quais emulam os modos de defeito de comunicação tradicionais [20], discutidos na seção 3.1. Outros tipos de injetores de falhas, que emulem diferentes modos de defeito, também podem ser usados para implementar a capacidade do atacante e simular tipos específicos de ataques contra o sistema.

A transformação pode ser feita por um ***conversor***. O ***conversor*** transforma as

operações de alto nível das interfaces *Ataque*, *Vítima* e ***Atacante***, em uma linguagem de script do componente atual (injetor de falhas) antes da execução do ataque. O uso do ***conversor*** é apropriado, pois permite uma conversão automatizada e transparente do SAG em um script da linguagem do injetor de falhas, permitindo a reusabilidade do SAG para outros injetores de falhas.

O ***conversor*** foi modelado usando o padrão de projeto **adaptador** [47]. Esse padrão "converte" a interface de alguma classe B em outra interface compatível A para que determinada classe cliente C possa usar suas operações. Um **adaptador** permite que classes funcionem juntas sendo que normalmente não poderiam por causa de incompatibilidade de interfaces. O **adaptador** traduz chamadas feitas para sua interface em chamadas para uma interface original, a qual precisa ser usada por outras interfaces.

A classe `Script_Ataque()` da Figura 4.6, que implementa a interface *Ataque*, representa a classe cliente que deseja usar as operações da classe do injetor de falhas. O **adaptador** é a classe do ***conversor*** – `Converter()` – que faz a tradução das operações da classe do injetor de falhas – `Fault_Injector()` – para as suas operações, de forma que a classe `Script_Ataque()` possa usá-las. Então as operações da classe `Converter()` serão chamadas diretamente pela classe `Script_Ataque()`. A classe `Converter()` implementa as operações das interfaces ***Atacante*** e *Vitima*, e suas operações usam as operações da classe `Fault_Injector()`, que também implementa as operações da interface ***Atacante***, mas usando código especifico do injetor de falhas. A Figura 4.6 mostra essas classes e suas relações, que foram criadas usando a ferramenta MagicDraw UML[24] v16.0 de demonstração.

A implementação das operações da classe `Fault_Injector()` é feita usando a linguagem do injetor escolhido, e o retorno dessas operações é convertido pelas operações da classe `Converter()` para um formato que as operações da classe `Script_Ataque()` possam processar.

[24] http://www.magicdraw.com/

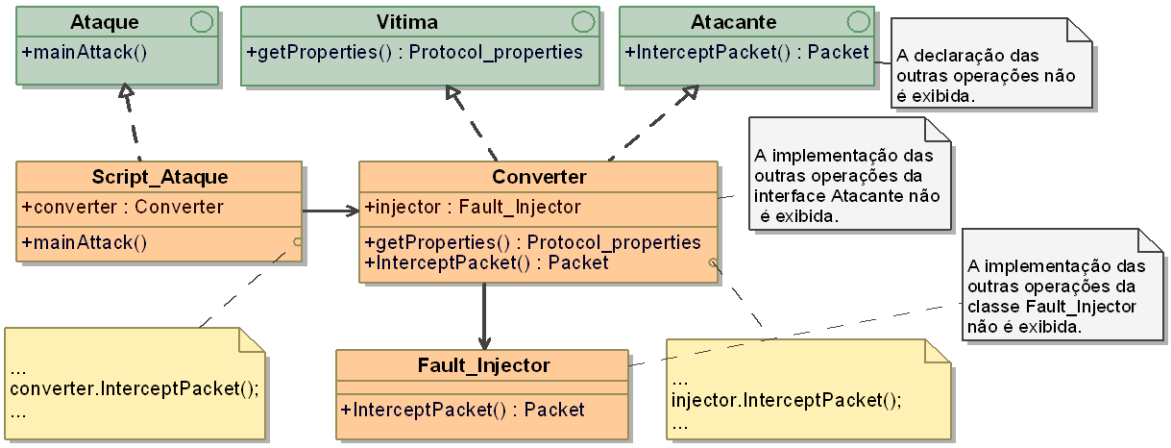


Figura 4.6: Diagrama de classes dos componentes do ataque.

Logo a saída dessa etapa é o **Script de Ataque Executável** (SAE) que é o Cenário de Ataque descrito na linguagem do injetor e pronto para ser executado pelo injetor de falhas. No caso de alteração do componente de testes (ferramenta de injeção de ataques) responsável pela execução do SAE, esta etapa é a única a ser impactada.

4.7 Outros Métodos para Refinar os Cenários de Ataque

Essa seção apresenta alternativas para o refinamento dos Cenários de Ataque obtidos na etapa **4** (seção 4.4) e a geração de SAEs, como na etapa **6** (seção 4.6). As ferramentas e métodos citados aqui poderiam ser usados para transformar os Cenários de Ataque em SAEs específicos para o injetor de falha usado.

Uma alternativa às palavras-chave seria o uso de linguagens de especificação de eventos, como JESL (*Java Event Specification Language*) usada no trabalho [49]. A JESL permite definir eventos simples e compostos de sistemas distribuídos (sendo este último similar ao conceito de regra ECA explicada na seção 4.5.2), onde eventos compostos são especificados em termos de certas condições de *timing*, ou de restrições de valores de atributos de outros eventos, e são capazes de executar ações, i.e., um conjunto de operações que atribuem valores aos atributos de outros eventos compostos uma vez que as condições ou restrições são satisfeitas.

O trabalho de Gerchman e Weber [50] apresenta uma metodologia para geração de

carga de falhas a partir de modelos de artefatos de teste, os quais são descritos usando uma extensão do Perfil UML 2.0 de Testes (U2TP) – U2TP-FI, UML 2.0 *Testing Profile Fault Injection*. Essa metodologia seria equivalente as etapas **5** (seção 4.5) e **6** (seção 4.6) da abordagem proposta, onde os passos do Cenário de Ataque (convertidos em elementos de padrão de ataque) são modelados e o modelo é transformando em uma linguagem específica do injetor de falhas. As etapas da metodologia do trabalho [50] são: **1)** criação do modelo de falhas, **2)** modelagem do caso de teste, **3)** criação do cartucho[25] AndroMDA[26], **4)** geração dos artefatos de teste e execução. Na etapa **1)**, um modelo UML é criado com classes representando os tipos de falhas de interesse e os tipos de ativação dessas falhas. Na etapa **2)** os casos de teste são criados como classes com comportamentos associados, os quais são especificados por diagramas de seqüência ou de comunicação mostrando as interações entre os componentes do teste. Nos diagramas de seqüência, que definem o comportamento e as ações dos casos de teste, são incluídos elementos que representam as falhas a serem injetadas. Na etapa **3)**, um cartucho é criado para cada injetor de teste a ser usado durante a atividade de teste, i.e., os diagramas UML são processados e a carga de falhas é extraída para os testes. A etapa **4)** consiste na execução de AndroMDA para geração dos artefatos, fornecendo como entrada os arquivos de modelos UML e os cartuchos AndroMDA desenvolvidos, portanto aderindo à proposta MDA (*Model Driven Architecture*). Os arquivos gerados são usados na etapa seguinte, a execução do teste.

A saída da etapa 1) é similar a interface ***Atacante***, definida na seção 4.5.3, o qual possui operações que representam as ações do injetor de falhas e os modos de defeito de comunicação usados [20]. No entanto, a criação das classes de falhas e das classes auxiliares que representam as condições de ativação das falhas é feita manualmente, através da análise do modelo de falhas, dos objetivos do teste e da capacidade das ferramentas de injeção, necessitando de muito esforço manual. A saída da etapa 2) seria similar a saída da etapa **4** (seção 4.5) – SAG – na forma de um diagrama de seqüência, o qual define o comportamento e as ações do Cenário de Ataque (falhas a serem injetadas durante o

[25] Um cartucho é um pacote contendo regras para a seleção de elementos de interesse e gabaritos para a geração de arquivos de saída para a plataforma alvo, como J2EE, Spring, .NET [50].

[26] AndroMDA é um framework para a geração de artefatos a partir de modelos UML que adere a abordagem *Model Driven Architecture* (MDA) [22]. Fonte: http://www.andromda.org/.

ataque). Nessa etapa 2) supõe-se que os elementos do sistema-alvo, como classes e interfaces, já estariam prontos para serem importados para o modelo dos casos de teste, i.e., a modelagem do sistema alvo é pré-requisito dessa etapa. Também é necessário criar outro diagrama de classes à parte, com as instâncias de descrição e de ativação das falhas. Esta etapa 3) é similar à implementação das operações da classe `Fault_Injector()` na seção 4.6, que são usadas no SAG, e usam código específico do injetor de falhas para implementar as falhas usadas. A saída dessa etapa é similar a saída da etapa **6** (seção 4.6), o SAE pronto para ser executado pelo injetor de falhas nos casos de teste do ataque.

As etapas 3) e 4) dessa metodologia [50] apresentam um bom grau de automação com o uso da ferramenta AndroMDA, uma vez que os diagramas e modelos UML das falhas e do sistema alvo usados estejam prontos. Também o modelo UML, criado na etapa 1) com as classes que representam as falhas permite a reusabilidade desse modelo com diferentes injetores. Mas, como descrito na etapa 2), é necessário modelar cada caso de teste a ser executado através de diagramas de seqüência, e também um diagrama de classes auxiliar para especificar os parâmetros das falhas e das ativações usadas em cada caso de teste, os quais requerem grande esforço manual. Geralmente a execução de injeção de ataques requer que muitos casos de testes sejam criados usando diferentes SAEs, como é descrito nos experimentos do capítulo 6, o qual dificulta o uso desses modelos. O uso da metodologia [50] requereria bastante tempo de modelagem dos casos de testes e das falhas usadas em cada caso de teste, aumentando a duração dos experimentos.

Também podem ser usadas algumas linguagens de ataque existentes, que servem para especificar cenários de ataque usados por IDSs para detectar atividades maliciosas e quebras do sistema. Dentre essas linguagens podemos citar: **i)** a abordagem UMLintr [16], citada na seção 4.5.3, que foca na especificação de cenários de ataques através de digramas UML; **ii)** a linguagem AsmL (*Abstract State Machine Language*)[27] baseada no conceito de *Abstract State Machine* (ASM), o qual foi desenvolvida para propósitos de especificação e verificação de software usando diagramas de estado, sendo que pode ser usada para especificar e gerar SAEs; **iii)** a linguagem de ataque STATL[28], que é capaz de descrever diferentes cenários de ataque, onde um ataque é modelado como uma seqüência de passos

[27] http://research.microsoft.com/en-us/projects/asml/

[28] http://www.cs.ucsb.edu/~seclab/projects/stat/index.html

que levam o sistema de um estado "seguro" para um estado "comprometido". STATL usa o mesmo conceito de transição de estado usado na linguagem AsmL, porém com um nível de detalhamento maior; **iv)** a ferramenta de prevenção e detecção de intrusão Snort[29], o qual usa uma linguagem dirigida por regras semelhante ao conceito de regra ECA explicado na seção 4.5.2, que analisa o tráfego de rede e executa ações baseadas na regras definidas.

O objetivo do uso de uma linguagem de ataque na nossa abordagem seria transformar os Cenários de Ataque em SAGs, de forma equivalente a etapas **5** (seção 4.5) da abordagem proposta. A linguagem AsmLSec [48], uma extensão de AsmL que incorpora funcionalidades requeridas para a especificação de cenários de intrusão, seria apropriada para representar os passos do Cenários de Ataques e gerar os SAGs e SAEs, pois pode ser usada para escrever cenários de ataque complexos de uma maneira fácil, concisa e prática. AsmLSec modela cenários de ataque como um conjunto de estados e transições. Os estados representam fotografias do sistema durante o curso do ataque, e as transições são rotuladas com eventos do sistema que causam a mudança de um estado para outro. Também uma transição de estado só pode acontecer se certas condições associadas com a transição são satisfeitas. Os eventos do sistema precisam acontecer respeitando certa ordem para que o ataque tenha sucesso.

As **vantagens** do uso dessas linguagens são: i) a sintaxe e a semântica são bem definidas e se encontram em um estágio avançado e maduro, facilitando a escrita do Cenário de Ataque; ii) possuem ferramentas e pacotes mais estáveis para o uso diário, sendo que algumas em escala comercial, como a ferramenta Snort, que também facilita à escrita e manutenção do Cenário de Ataque; iii) assinaturas do mesmo ataque podem ser importadas dessas ferramentas ou linguagens e reutilizadas na nossa abordagem. As **desvantagens** são: i) a modelagem do ataque é feita manualmente, não possuindo etapas e passos específicos, como descrito na etapa **5** (seção 4.5), para a conversão dos passos do Cenário de Ataque na linguagem; ii) o uso de uma linguagem de ataque específica, como Snort que possui sintaxe e semântica específicas para suas regras, que tornaria o método de refinamento do Cenário de Ataque dependente e preso a uma representação específica, a qual pode se tornar obsoleta e desatualizada, i.e, não podendo ser estendida e atualizada de

[29] http://www.snort.org/

acordo com atualizações da capacidade do atacante (intruso) definida na seção 4.2.

4.8 Injeção de Ataques e Monitoramento

Apesar dessa etapa não fazer parte da metodologia proposta, que propõe a geração de SAEs a partir de um modelo de ataques, vale a pena fazer algumas considerações, dado que é importante em validação experimental. Enquanto os ataques são injetados, é útil monitorar e coletar as mensagens sendo trocadas, para determinar se os ataques planejados foram efetivamente realizados, ou ainda, para monitorar o uso de recursos.

O monitoramento da injeção de ataques dos experimentos pode ser feito usando analisadores de protocolo que capturam o tráfego da rede e os pacotes de dados trocados entre as partes, já que são usados **injetores de falhas de comunicação** para os experimentos, como citado na seção 4.2. Também as funcionalidades nativas de log das ferramentas de injeção podem ser usadas para o monitoramento dos ataques injetados, pois podem fornecer dados valiosos para as análises pós-experimento. Podem-se também utilizar os recursos e mecanismos de monitoramento oferecidos pelos sistemas operacionais (Windows/Linux), onde são executados os injetores de falhas e o sistema alvo.

Capítulo 5

Estudo de Caso

Este capítulo apresenta um exemplo do uso da abordagem para mostrar sua aplicabilidade no teste de protocolos reais. Como estudo de caso foi usada a camada de segurança da pilha do protocolo WAP: o protocolo WTLS, o qual é explicado na seção 5.1. As razões para a escolha desse protocolo foram duas: **i)** as vulnerabilidades do WTLS são bem conhecidas e documentadas; **ii)** a familiaridade com a pilha do WAP, pois foram executados testes de robustez com a camada de transação (WTP – *Wireless Transaction Protocol*) do gateway WAP na presença de falhas em um trabalho anterior [21] . A mesma arquitetura WAP do trabalho anterior [21] foi usada neste trabalho para avaliar a abordagem proposta e realizar Testes de Segurança na presença de falhas, como será mostrado no capítulo 6. Assim, a aplicação da abordagem e a instrumentalização dos experimentos (no capítulo 6) foram realizadas de forma a dar continuidade ao trabalho anterior demonstrando o uso da metodologia.

As outras seções ilustram o uso das etapas da metodologia proposta no capítulo 4 ao estudo de caso escolhido: o WTLS. A identificação dos objetivos do atacante extraída de fontes de ataques do WTLS é mostrada na seção 5.2 (etapa 1). Os injetores de falhas escolhidos para implementar a capacidade do atacante da seção 4.2 (etapa 2) também são explicados na seção 5.2. Na seção 5.3 é mostrado como as informações de ataques do WTLS são modeladas na forma de Árvore de Ataque (etapa 3). Na seção 5.4 são exibidas as seguintes etapas da metodologia: geração de Cenários de Ataque a partir da Árvore de Ataques construída (etapa 4) e criação de Scripts de Ataque Genéricos (SAGs) a partir do refinamento dos Cenários de Ataque gerados (etapa 5).

5.1 Protocolo WTLS

Nos últimos anos, devido à existência do WAP (Wireless Application Protocol), a internet não se limitou apenas a computadores *desktops*, mas muitas pessoas podem usar seu PDA (*Personal Digital Assistants*) ou telefone móvel para navegar na internet. O WAP é uma

pilha de protocolos para redes de comunicações móveis, o qual é especificado pelo WAP *Forum* [24]. O WAP é equivalente a pilha de protocolos usada na internet (TCP/IP), mas para redes sem-fio. O WAP usa o protocolo WTLS, uma variação do protocolo TLS (o qual foi desenvolvido como uma versão padrão da internet do SSL) para redes sem-fio, para fornecer funcionalidades de segurança na comunicação entre o telefone móvel e o servidor WAP (gateway) [25], permitindo o uso do WAP em aplicações seguras como transações bancárias e e-commerce, onde requisitos básicos de segurança são exigidos.

A razão para projetar um novo protocolo em concordância com as linhas do TLS e não usar o TLS em si se deve aos seguintes motivos: **i)** o TLS foi projetado para ser usado sobre uma camada de transporte segura (como TCP) ao passo que o WTLS precisa operar sobre uma camada de transporte não-confiável (WDP - *Wireless Datagram Protocol* ou UDP), onde datagramas podem ser perdidos, duplicados ou reordenados, precisando de um mecanismo de confirmação e retransmissão de datagramas; **ii)** o protocolo WTLS foi modificado para se ajustar aos longos tempos de atraso da rede (*roundtrip*), largura de banda limitada, e longa latência, típicos de ambientes sem-fio; **iii)** o protocolo deve operar em dispositivos móveis com poder de processamento e memória limitados, o qual requer algoritmos mais eficientes e mais rápidos; **iv)** a comunicação via sinais de rádio é particularmente vulnerável a ataques de Escuta (como explicado na seção 2.3).

O WTLS é dividido em camadas, tendo o Protocolo de Registro (*Record Protocol* – RP) como base [25]. O RP é responsável pela *privacidade*, *integridade* e *autenticação*, sendo dividido em quatro protocolos clientes. **1)** O Protocolo de *Handshake* abre uma conexão e negocia os parâmetros de segurança, como o *cipher suite* (algoritmo de cifras de bloco e algoritmo de MAC). Também é responsável pela troca-de-chaves (*key exchange*) e autenticação de ambos os lados (servidor ou/e cliente) durante o *handshake*, que é explicado no apêndice B.1. **2)** O Protocolo de *Change Cipher Spec* confirma o uso do *cipher suite* negociado e da chave de sessão (*key session*) calculada durante o *handshake*. **3)** O Protocolo de Alerta envia mensagens de advertência, mensagens críticas e mensagens fatais para o tratamento de erros do WTLS. **4)** O Protocolo de Aplicação é meramente uma interface para transmissão de dados.

5.2 Objetivos do Atacante e Injetores

As informações sobre as vulnerabilidades de segurança e ataques do WTLS ("**descrição** dos ataques" e "**propriedades de segurança** violadas pelos ataques" definidas na seção 4.1) foram coletadas do trabalho de Saarinen [26], do trabalho de Análise técnica do SSL [27], do trabalho de Testes de Robustez usando o SSL/TLS [45], e do trabalho sobre diferentes tipos de ataque DoS contra o WTLS [44]. Essas informações identificam os objetivos do atacante. Os ataques coletados de diferentes classes revelam vulnerabilidades reais na implementação do WTLS, que poderiam ser exploradas por um atacante que possui os recursos apropriados. Os ataques foram categorizados de acordo com as **propriedades de segurança** violadas. Os detalhes e a descrição dos ataques usados nos experimentos estão descritos nas respectivas seções. Dessa forma foi executada a etapa **1** (seção 4.1) da metodologia da Figura 4.1.

O objetivo da etapa **2** (seção 4.2) é descrever a **capacidade do atacante**, o qual é implementada por injetores de falhas escolhidos. O injetor de falhas *Firmament* v.0.26[30] foi usado no trabalho de testes de robustez [21], onde o protocolo *Wireless Transaction Protocol* (WTP) da pilha WAP foi usado como estudo de caso. Nesse trabalho, PDUs da camada WTP foram alteradas, atrasadas ou removidas, com o objetivo de verificar a robustez da implementação no gateway WAP, onde foi monitorado seu comportamento.

Firmament é uma evolução da ferramenta COMFIRM [6], apresentada na seção 3.1. *Firmament* emprega o conceito de *faultlet* que é usado para especificação de cenários de falhas. Um *faultlet* é executado em cada pacote que cruza o fluxo de comunicação sendo capaz de executar as ações de **interceptar** o pacote e **corromper** o conteúdo do pacote, além de **remover**, **duplicar** ou **atrasar** o pacote. Além de pacotes, um *faultlet* pode agir em um conjunto de variáveis de estado do fluxo para realizar as ações **armazenar**, **restaurar** e **sinalizar**. Portanto, um *faultlet* é capaz de mover dados entre o pacote e variáveis de estado, e executar operações lógicas, aritméticas e de controle nesses dados. *Faultlets* são configurados independentemente para os fluxos de entrada e saída do protocolo IP. Os motivos que levaram a escolha da ferramenta *Firmament* foram: **i)** a ferramenta possui total

[30] http://firmament.sourceforge.net/

acesso ao fluxo de mensagens que entram e saem do sistema operacional de uma maneira clara e não-intrusiva, pois não altera o código dos sistemas envolvidos; **ii)** está localizada dentro do SO, dessa forma minimizando a intrusão no protocolo testado, pois não são necessárias trocas de contexto entre a execução do protocolo e a execução do injetor; **iii)** a ferramenta é totalmente operacional e pode ser facilmente adicionada ou removida do sistema operacional Linux para realizar os testes a qualquer momento.

Para executar a ação **personificar** foi escolhida a ferramenta *Network Packet Generator*[31] v.1.3.0 (NPG), pois o injetor de falhas *Firmament* não possui a capacidade de injetar pacotes pré-definidos na interface de rede. A ferramenta NPG é de fácil uso e configuração para essa atividade. O NPG é um injetor de pacotes que utiliza a biblioteca WinPcap[32] do Windows para enviar pacotes específicos em uma ou mais interfaces de rede. As configurações desses pacotes (o cabeçalho e os dados) e outras opções (ex.: número de vezes que o pacote dever ser injetado, interface de rede a ser usada) são definidos em um arquivo texto que usa uma formatação específica definida pela ferramenta. Dessa forma os pacotes são injetados pela ferramenta através de linha de comando na Shell do Windows.

Esses foram os injetores escolhidos para implementar a **capacidade do atacante** (intruso) da etapa **2** (seção 4.2), e que serão usados nos experimentos da capítulo 6.

5.3 Modelagem dos Ataques

Foram investigadas ferramentas disponíveis para auxiliar na construção da Árvore de Ataques e seleção dos Cenários de Ataque, pois não é viável construir e analisar manualmente uma árvore de ataques para sistemas complexos. A ferramenta *SecureITree* [28] foi usada para construir a Árvore de Ataques gráfica do WTLS, pois demonstrou ser eficiente e prática para essa tarefa.

A Árvore de Ataques foi construída e estruturada de acordo com os passos propostos na etapa **3** (seção 4.3), e usando a maioria dos ataques identificados na seção 5.2 (etapa **1** da metodologia). A Figura 5.1 (a) mostra a representação textual da **Árvore de Ataques** do WTLS e a Figura 5.1 (b) mostra sua representação gráfica correspondente.

[31] http://www.wikistc.org/wiki/Network_packet_generator

[32] http://www.winpcap.org/

Podemos observar que várias classes de ataques são mostradas na Árvore de Ataques da Figura 5.1 (a), as quais violam diferentes tipos de **propriedades de segurança**, desde ataques de truncamento que violam a propriedade *integridade* até ataques de força bruta no protocolo de ciframento que viola a propriedade *privacidade*.

OR 1 – **Objetivo**: Atacar o WTLS

OR 1.1 – Violar ***integridade***

OR 1.1.1 – Violar *integridade* da mensagem

OR 1.1.1.1 – Ataque de truncamento usando mensagens de Alerta **<P, I>**

1.1.1.2 – Ataque de truncamento no último registro[33] **<P, P>**

1.1.2 – Violar *integridade* do registro através do algoritmo de MAC SHA_XOR_40 **<P, P>**

1.1.3 – Ataques MITM

OR 1.1.3.1 – Ataque de rollback no *cipher suit*e **<P, P>**

1.1.3.2 – Remover o *Change Cipher Spec* **<P, P>**

1.1.3.3 – Ataque de rollback no algoritmo de troca-de-chaves **<P, I>**

1.2 – Violar ***privacidade***

OR 1.2.1 – Explorar a chave (força bruta)

OR 1.2.1.1 – Quebrar o ciframento do algoritmo DES 40bit **<I, I>**

1.2.1.2 – Prováveis ataques de texto em claro **<I, I>**

1.2.2 – Vulnerabilidades de texto em claro (Escuta Passiva)

OR 1.2.2.1 – Determinar o IV inicial **<P, P>**

1.2.2.2 – Determinar a troca-de-chaves **<P, P>**

1.2.2.3 – Determinar mensagens de Erro **<P, P>**

1.3 – Violar ***autenticação***

AND 1.3.1 – Atacar a troca-de-chaves no *handshake* através do algoritmo RSA que usa PKCS #1 v.1.5 **<P, I>**

1.4 – Violar ***disponibilidade*** (ataques de DoS)

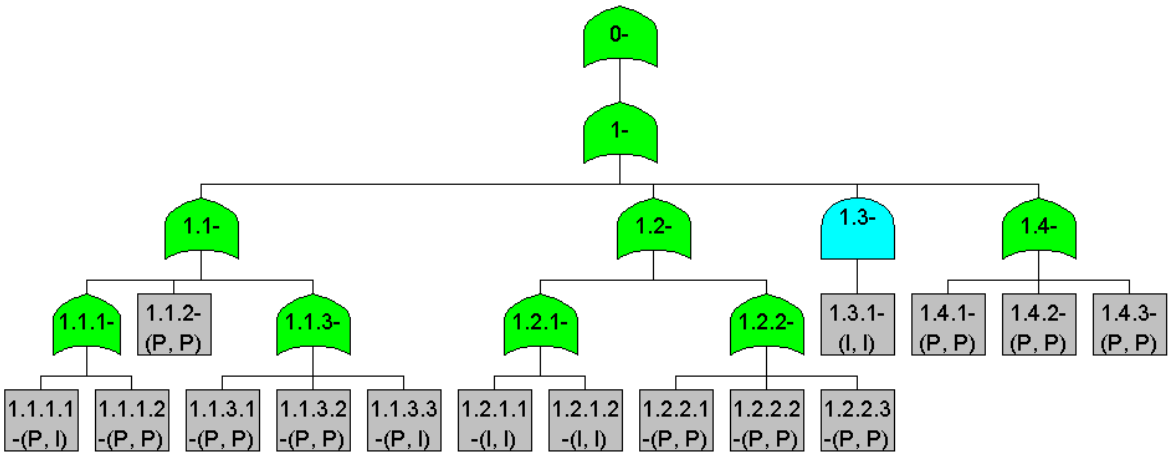
OR 1.4.1 – Ataque de DoS para exaustão de CPU (início do *handshake*) **<P, P>**

1.4.2 – Ataque de DoS para exaustão de memória (fim do *handshake*) **<P, P>**

1.4.3 – Ataque de DoS para exaustão de CPU (meio do *handshake*) **<P, P>**

[33] Registro é uma PDU na camada do Protocolo de Registro [25].

(a) Árvore de ataques textual do WTLS.



(b) Árvore de ataques gráfica do WTLS.

Figura 5.1: Árvore de Ataques do WTLS.

As folhas da Árvore de Ataques da Figura 5.1 (a) possuem dois atributos associados, que são valores lógicos representando a possibilidade de um ataque **<*capacidade do atacante, Arquitetura de Teste*>: i)** a capacidade do atacante definida na etapa **2** (seção 4.2); **ii)** a Arquitetura de Teste usada, que é explicada na seção 6.1. Embora alguns ataques estejam dentro do escopo da capacidade do atacante proposta, eles não podem ser injetados devido às limitações técnicas e funcionais dos injetores (escolhidos na seção 5.2) da Arquitetura de Teste. O uso desses dois atributos é útil no sentido que nós separamos as limitações da metodologia proposta das limitações da Arquitetura de Teste usada. Por exemplo, os valores <**P**ossível, **I**mpossível> para os dois atributos que estão colocados do lado direito da folha <1.1.1.1> da Árvore de Ataques da Figura 5.1 (a) indicam que: **i)** a capacidade do atacante é capaz de executar a ação; **ii)** o injetor de falhas não é capaz de emular o ataque, o que indica que o ataque não pode ser executado usando a Arquitetura de Teste atual. Nesse caso, o testador possui a opção de escolher outro injetor de falhas apropriado para a execução desse ataque ou não selecionar esse tipo de ataque ao realizar os testes.

5.4 Geração e Refinamento dos Cenários de Ataque

Os Cenários de Ataque foram selecionados de acordo os valores dos atributos das folhas. Para produzir os Cenários de Ataque, o ***critério*** usado foi "cobrir todos os ataques que satisfazem os valores **<P, P>** para os dois atributos das folhas", pois indica que estão dentro da capacidade do atacante e podem ser emulados pelos injetores descritos na seção 5.2. A ferramenta *SecureITree* [28] foi usada para fazer a busca e selecionar os Cenários de Ataque de uma maneira automatizada. Os Cenários que satisfizeram ao ***critério*** foram: <1.1.1.2>, <1.1.2.1>, <1.1.3.1>, <1.1.3.2>, <1.2.2.1>, <1.2.2.2>, <1.2.2.3>, <1.4.1>, <1.4.2> e <1.4.3>. Estes são os **Cenários de Ataque** selecionados de acordo com a etapa **4** (seção 4.4). Alguns desses Cenários de Ataque podem ser reusados com outro protocolo móvel similar, pois provavelmente o mesmo tipo de ataque também se aplicaria a esse protocolo. A sub-árvore da Árvore de Ataques do WTLS, o qual derivou esse Cenário de Ataque também poderia ser reusada na outra árvore de ataques da implementação similar, como foi mencionado na seção 2.4.1.

Para ilustrar os passos de refinamento da etapa **5** (seção 4.5) e os métodos propostos para obter o SAG de forma automatizada, foi escolhido o Cenário de Ataque *<1.1.1.2 – Ataque de truncamento no último registro>* que contém a "**descrição** do ataque" e a "**propriedade de segurança** violada pelo ataque". O ataque desse Cenário é baseado no ataque de truncamento do SSL v.3 / TLS v.1 [45], descrito na seção 4.5.1. Na versão desse ataque para o contexto do WTLS apenas o último registro (PDU) de dados de aplicação é removido durante uma sessão de transferência de dados. Conforme citado na seção 5.1, o WAP usa o UDP como protocolo de transporte, ao contrário do TLS que usa o TCP como protocolo de transporte, então não é necessário enviar a mensagem de Alerta *close_notify* do WTLS para sinalizar o fechamento de uma conexão na camada de segurança.

A Figura 5.2 mostra o primeiro passo do refinamento do Cenário <1.1.1.2> para o contexto do WTLS. Nesse passo foi utilizado o **padrão de ataque**, explicado na seção 2.4.2 e citado na etapa **5** (seção 4.5.1), onde o Cenário de Ataque ("**descrição** do ataque") é detalhado usando os seguintes elementos: **i)** o *"objetivo do ataque"*; **ii)** a *"lista de precondições"*; **iii)** os *"passos para executar o ataque"*.

Objetivo: Ataque de truncamento no último registro

Precondição: Pacotes são enviados do gateway WAP para o cliente

Ataque:

AND 1.1.1.2.*1* – ***Se*** o registro é do tipo dados de aplicação

1.1.1.2.*2* – ***Se*** o registro é o último da mensagem

1.1.1.2.*3* – Remover o último registro

Figura 5.2: Padrão de ataque para o Cenário de Ataque <1.1.1.2>.

O padrão de ataque da Figura 5.2 foi baseado no padrão de ataque do Cenário de Ataque *<Ataque de truncamento no último registro e no registro close_notify>* da seção 4.5.1, que é mostrado na Figura 4.2. Foram feitas as seguintes alterações nos elementos desse padrão de ataque: **i)** o "**Objetivo**" e a "**Precondição**" foram atualizados; **ii)** os elementos (*"passos para executar o ataque"*) 4, 5 e 6 foram removidos, pois como explicado anteriormente, o WTLS não usa a mensagem de Alerta (*close_notify*) para fechar uma conexão na camada de segurança.

Em seguida o padrão de ataque da Figura 5.2 é convertido para formato ECA, como explicado na seção 4.5.2, onde o elemento "**Precondição**" é mapeado para a parte *evento*, os dois primeiros elementos 1.1.1.2.*1* e 1.1.1.2.*2* são mapeados para a parte *condição*, e o terceiro elemento 1.1.1.2.*3* é mapeado para a parte *ação*, pois "Remover" – **`remover(m)`** – é uma das **palavras-chave** estabelecidas na Figura 4.3. A regra criada para esse ataque é mostrada na Figura 5.3. As partes dessa regra ECA são então descritas usando as **palavras-chave** da tabela na Figura 4.3, como é mostrado na Figura 5.3.

Regra 1:

ON *evento*: `env(A,B,m,<PT=UDP>,<IP_A>,<IP_B>,<Po_A=gateway_port>,<Po_B>)`

IF *condição*: (**1.1.1.2.*1*** – `m.tipo_PDU == protocolo.<prop=application_data>`)

AND (**1.1.1.2.*2*** – `m.<campo=record_sequence> == LAST`)

DO *ação*: **1.1.1.2.*3*** – `remover(m)`

Figura 5.3: Regra ECA do Cenário de Ataque <1.1.1.2>.

Agora iremos identificar as operações da interface ***Atacante*** (Tabela 4.1) para as **palavras-chave** das partes da regra ECA da Figura 5.3 de acordo com a Tabela 4.2.

- Para a parte **ON** (*evento*) primeiro é usada a operação `InterceptPacket` relacionada à **palavra-chave** `interceptar(m)`, como explicado na seção 4.5.3. A ativação dessa regra está relacionada à **palavra-chave** `env(...)`, que depende de uma mensagem originada do gateway WAP (parâmetro `<Po_A>=gateway_port`), então usaremos a operação `getSourcePort` para verificar se um pacote é enviado da porta padrão do gateway WAP. Também utilizaremos a operação `getProtocolType` para selecionar o protocolo de transporte (parâmetro `<PT>=UDP`) usado pelo protocolo alvo no fluxo de pacotes interceptados.
- Para a parte **IF** (*condição*): 1) para a **palavra-chave** `m.tipo_PDU` da primeira condição é usada a operação `getPDUType`, pois ela depende de um tipo específico de PDU do protocolo alvo – `application_data`; 2) para a **palavra-chave** `m.<campo=record_sequence>` da segunda condição é usada a operação `getPDUField`, pois precisamos obter o valor de um campo do protocolo alvo e compará-lo com um valor pré-fixado – `LAST`.
- Para a **palavra-chave** `remover(m)` da parte **DO** (*ação*) é usada a operação `DropPacket` para remover o pacote contento o registro WTLS.

O SAG mostrado na Figura 5.4 é estruturado de acordo com as instruções da etapa **5** (seção 4.5.3), onde são usadas as operações identificados para as **palavras-chave** da regra ECA do Cenário de Ataque <1.1.1.2>:

i. Primeiro é usada a operação `mainAttack` da interface *Ataque*, o qual coordenará os passos do ataque através das operações das outras duas interfaces, sendo implementada pela classe `Script_Ataque()`.

ii. A operação `mainAttack` chama a operação `getProperties` (**palavra-chave** `protocolo.<prop>`) da interface *Vítima*, que é implementa pela classe

`Converter()` como explicado na seção 4.6, para obter os seguintes valores da especificação do protocolo: a porta do gateway WAP (`gateway_port`) para comparar com a porta retornada pela operação `getSourcePort`, o valor do protocolo `UDP` para comparar com o valor de protocolo de transporte retornado pela operação `getProtocolType`, e o valor da PDU do tipo `application_data` para comparar com o valor de PDU retornado pela operação `getPDUType`;

iii. Depois é estabelecida a lógica do ataque, a qual chama as operações da interface ***Atacante*** também implementadas pela classe `Converter()`, que foram identificadas para as **palavras-chave** da regra ECA, obedecendo à semântica: **ON** *evento* **IF** *condição* **DO** *ação*.

```
1. class Script_Ataque implements Ataque {
2.
3.  mainAttack() {
4.
5.   Protocol_Type UDP; Port gateway_port;
6.   PDU_Type application_data; PDU_Field record_sequence;
7.   PDU_Field lastRecord = LAST;
8.   Boolean truncate;
9.   Packet pacote;
10.  converter Converter;
11.
12.  UDP, gateway_port, application_data \
13.  record_sequence = converter.getProperties(WTLS);
14.  while (pacote = converter.InterceptPacket(interface)) {
15.   if (converter.getProtocolType(pacote) == UDP) and
16.      (converter.getSourcePort(pacote) == gateway_port) {
17.    Regra_1:
18.    if (converter.getPDUType(pacote) == application_data) and
19.    (converter.getPDUField(pacote, record_sequence)==lastRecord){
20.      converter.DropPacket(pacote);
21.      return;  /* Fim do ataque */
22.}}}}}
```

Figura 5.4: Script de Ataque Genérico (SAG) do Cenário de Ataque <1.1.1.2>.

A Figura 5.4 mostra o SAG que implementa a operação `mainAttack` da interface *Ataque* e os passos de execução do ataque.

A saída dessa etapa **5** – o **Script de Ataque Genérico (SAG)** da Figura 5.4 – poderia ser reusada para outro protocolo similar, necessitando de pequenas alterações. Por exemplo, o Script de Ataque poderia ser reusado para o mesmo ataque do protocolo TLS, explicado na seção 4.5.1, pois os passos de ataque são bastante similares, como explicado anteriormente, necessitando apenas alterar a chamada da operação `getProperties()` na linha 13 do script que é feita especificamente para o protocolo WTLS e adicionar mais operações para as outras **palavras-chave** das regras ECA do ataque do TLS.

A abordagem usada nessa seção favorece a reusabilidade do modelo de ataque, tanto através dos Cenários de Ataque obtidos da Árvore de Ataques quanto para o SAG obtido a partir do refinamento do Cenário de Ataque, ao contrário da maioria das abordagens discutidas na seção 3.3.

O SAG representa o cenário de falhas, sendo independente da linguagem do injetor de falhas. Uma implementação específica é necessária para obter o SAE, i.e., as operações das interfaces precisam ser convertidas para a linguagem do injetor. Esse passo é explicado na seção 6.2.1 do capítulo 6.

Capítulo 6

Experimentos e Resultados

Este capítulo apresenta a parte experimental desse trabalho, bem como a análise dos resultados obtidos nos experimentos de Testes de Segurança com o estudo de caso: o protocolo de segurança WTLS. Para realização dos experimentos foram usados os Cenários de Ataque <1.1.1.2>, <1.1.3.1>, <1.4.1> selecionados na seção 5.4. Esses ataques violam as propriedades de *integridade* e *disponibilidade*. A simulação desses ataques é capaz de demonstrar o real potencial e a utilidade da abordagem proposta, pois são usados diferentes tipos de falhas implementados pelas ferramentas de injeção para simular diferentes classes de ataques.

O Cenário de Ataque <1.1.1.2> usado na seção 5.4 para ilustrar os passos de refinamento da etapa **5** (seção 4.5), é usado na seção 6.2.1 para explicar a etapa **6** (seção 4.6) da metodologia a fim de se obter o SAE (SAE) para realização dos experimentos. Os experimentos e os resultados obtidos para esse Cenário de Ataque são mostrados ao longo da seção 6.2.

Para os outros dois Cenários (<1.4.1> e <1.1.3.1>) não foram mostrados os detalhes do passo da etapa **5** da metodologia já que o processo de obtenção do SAG é similar ao que foi usado para o cenário <1.1.1.2>. Mas para esses Cenários são descritos os experimentos e os resultados obtidos nas seções 6.3 e 6.4. A seção 6.5 fecha o capítulo apresentando os comentários finais em relação aos resultados experimentais.

6.1 Arquitetura de Teste

A Figura 6.1 apresenta a Arquitetura de Teste usada nos experimentos de injeção de ataques. O protocolo WTLS é implementado pelo *Nokia Mobile Browser Simulator*[34] (NMB v.4.0) que é o cliente com capacidade WAP, e pelo *Columbitech WAP Connector*

[34] http://www.forum.nokia.com/

v.1.3 [35] que é o simulador de gateway WAP. O atacante é implementado pelo injetor de falhas de comunicação *Firmament* v.0.26, e pelo injetor de pacotes *Network Packet Generator* (NPG v.1.3.0), ambos descritos na seção 5.2.

Para executar os experimentos foi usado um PC com CPU Intel Core2 6600 a 2.40 GHz e 2,0 GB de memória principal RAM com SO Windows XP SP3 instalado. O simulador de browser NMB, o simulador de gateway WAP *Columbitech* e a ferramenta de injeção de pacotes NPG foram executados na máquina com SO Windows XP SP3, pois são aplicações nativas do Windows. A ferramenta de injeção *Firmament*, nativa do Linux, foi executada no SO Linux Ubuntu kernel v.2.6.22-14 como uma máquina virtual.

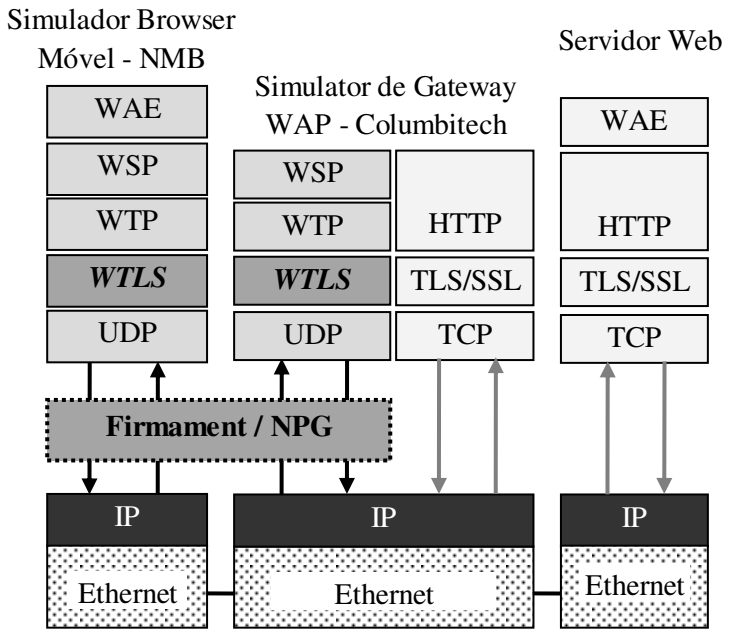


Figura 6.1: Arquitetura de Teste.

6.2 Ataque de Truncamento

O Cenário de Ataque *<1.1.1.2 – Ataque de truncamento no último registro>* obtido na seção 5.4 representa um tipo de ataque de truncamento contra o cliente WAP (browser NMB), o qual foi explicado na seção 5.4. Para simular esse Cenário de Ataque foi usado o

[35] http://www.columbitech.com/

injetor de falhas *Firmament*, o simulador de browser (NMB) e o simulador de gateway WAP mostrados na Arquitetura de Testes da Figura 6.1.

6.2.1 Transformação do Script de Ataque Genérico

Nessa seção iremos transformar o SAG da Figura 5.4, obtido na seção 5.4 como produto final do refinamento do Cenário de Ataque <1.1.1.2>, em uma linguagem de script específica do injetor *Firmament* para a execução de acordo com a etapa **6** (seção 4.6).

A transformação é feita como explicado na seção 4.6. Basicamente as operações das interfaces *Vítima* e ***Atacante*** implementadas pela classe `Converter()`, que são chamadas pela classe `Script_Ataque()`da Figura 5.4, convertem as instruções da ferramenta *Firmament*. Esse conjunto de instruções de *Firmament* usadas seriam as operações da classe `Fault_Injector()`, o qual são chamadas pelas operações da classe `Script_Ataque()` indiretamente através da classe `Converter()`. A Figura 6.2 mostra o diagrama de seqüência para essas classes.

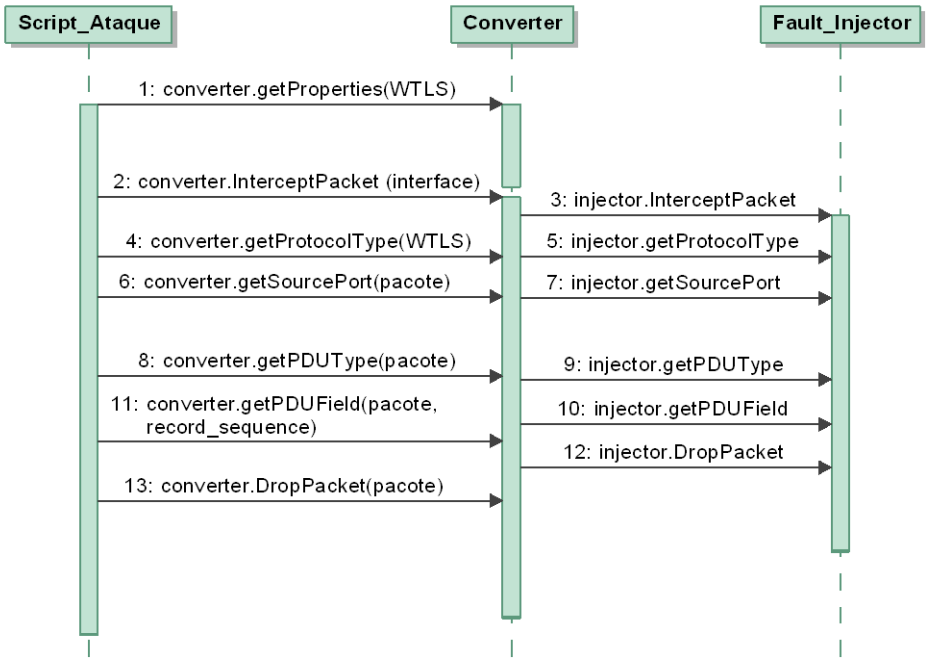


Figura 6.2: Digrama de seqüência da transformação.

A Tabela 6.1 mostra o mapeamento indireto entre as operações usadas da classe

`Script_Ataque()` e as instruções de *Firmament*. As instruções da ferramenta *Firmament* são detalhadas no apêndice A.1.

Operação	Instruções
`converter.getProperties`	Os valores retornados por essa operação são obtidos a partir da especificação do WTLS [25].
`converter.InterceptPacket(interface)`	Essa operação é diretamente executada pelo modulo de *Firmament* que se anexa ao Netfilter.

`converter.getProtocolType(pacote) == UDP`	`SET 0x9 R0  #offset do campo do UDP`
	`READB R0 R1`
	`SUB R1 R6`
	`JMPZ R6 IS_UDP`
	`JMP ERROR_OK`

`converter.getSourcePort(pacote) == gateway_port`	`SET 0x14 R0 #offset do campo da porta`
	`READS R0 R1`
	`SUB R1 R7`
	`JMPZ R7 TO_BROWSER`
	`JMP ERROR_OK`

`converter.getPDUType(pacote) == application_data`	`SET 0x1C R0  #offset do campo da PDU`
	`READB R0 R1`
	`SET 0x0F R0  #máscara do byte da PDU`
	`AND R0 R1`
	`SUB R8 R1`
	`JMPZ R1 PDU_AD`
	`JMP ERROR_OK`

`converter.getPDUField(pacote, record_sequence) == lastRecord`	`READS R9 R1`
	`SUB R5 R1`
	`JMPZ R1 DO_TRUNC`
	`JMP ERROR_OK`

`converter.DropPacket(pacote)`	**`DRP`** `JMP INIT`

Tabela 6.1: Mapeamento das operações e instruções de falhas.

Usando o mapeamento da Tabela 6.1 e a lógica de ataque da operação `mainAttack` da Figura 5.4, obtemos o SAE para o *Firmament*, que é mostrado na Figura 6.3.

```
1.  INIT:            # converter.getProperties
2.    SET 0xXX R5    # lastRecord = LAST (0xXX)
3.    SET 0x11 R6    # UDP           = 17
4.    SET 0x23F3 R7  # gateway_port = 9203
5.    SET 0x04 R8    # application_data = 4
6.    SET 0x1D R9    # record_sequence = 29
7.    SET 0x9 R0     # converter.getProtocolType(pacote) == UDP
8.    READB R0 R1
9.    SUB R1 R6
10.   JMPZ R6 IS_UDP
11.   JMP ERROR_OK
12. IS_UDP:
13.   SET 0x14 R0 # converter.getSourcePort(pacote)== gateway_port
14.   READS R0 R1
15.   SUB R1 R7
16.   JMPZ R7 TO_BROWSER
17.   JMP ERROR_OK
18. TO_BROWSER:     #### Regra 1
19.   SET 0x1C R0  # converter.getPDUType(pacote) == application_data
20.   READB R0 R1
21.   SET 0x0F R0
22.   AND R0 R1
23.   SUB R8 R1
24.   JMPZ R1 PDU_AD
25.   JMP ERROR_OK
26. PDU_AD:
27.   READS R9 R1   # converter.getPDUField(pacote,…) == lastRecord
28.   SUB R5 R1
29.   JMPZ R1 DO_TRUNC
30.   JMP ERROR_OK
39. DO_TRUNC:
40.   DRP           # converter.DropPacket(pacote)
41.   JMP INIT      # Fim do ataque
42. ERROR_OK:
43.    ACP
44.    JMP INIT
```

Figura 6.3: Script da *Firmament*.

O valor do "último registro" (`0xXX`) da linha 2 da Figura 6.3 atribuído à variável do script será alterado de acordo com a instância de script usada no caso de teste para executar o ataque, como será explicado na seção 6.2.2. Esse é o único parâmetro configurável do SAE.

A saída dessa etapa é o **SAE** (SAE) pronto para realizar os testes de injeção de ataque de truncamento. Podemos observar que esse script possui uma carga de falhas mínima, i.e., possui apenas a falha (instrução "**`DRP`**" na linha 40, que corresponde à capacidade do atacante **remover**) que é necessária para a realização do ataque, ao invés de gerar uma enorme carga de falhas para detectar vulnerabilidades, como é feito nas abordagens baseadas em Fuzz e Teste de Sintaxe, discutidas na seção 3.3.

Também com a utilização do ***conversor***, o SAG desse Cenário, resultado da etapa **5**, torna-se mais flexível a outros ambientes de testes, i.e., independente de plataforma. O ***conversor*** permite que o SAG seja portado para diferentes injetores de falhas, pois o ***conversor*** implementa operações para transformar diferentes linguagens de injetores em operações e retornos que são usados pelo SAG, funcionando como intermediador. Essa característica de portabilidade, no qual diferentes tipos de injetores de ataques com linguagens próprias podem ser usados, não é encontrada nas abordagens e ferramentas de Testes de Segurança descritos na seção 3.3.

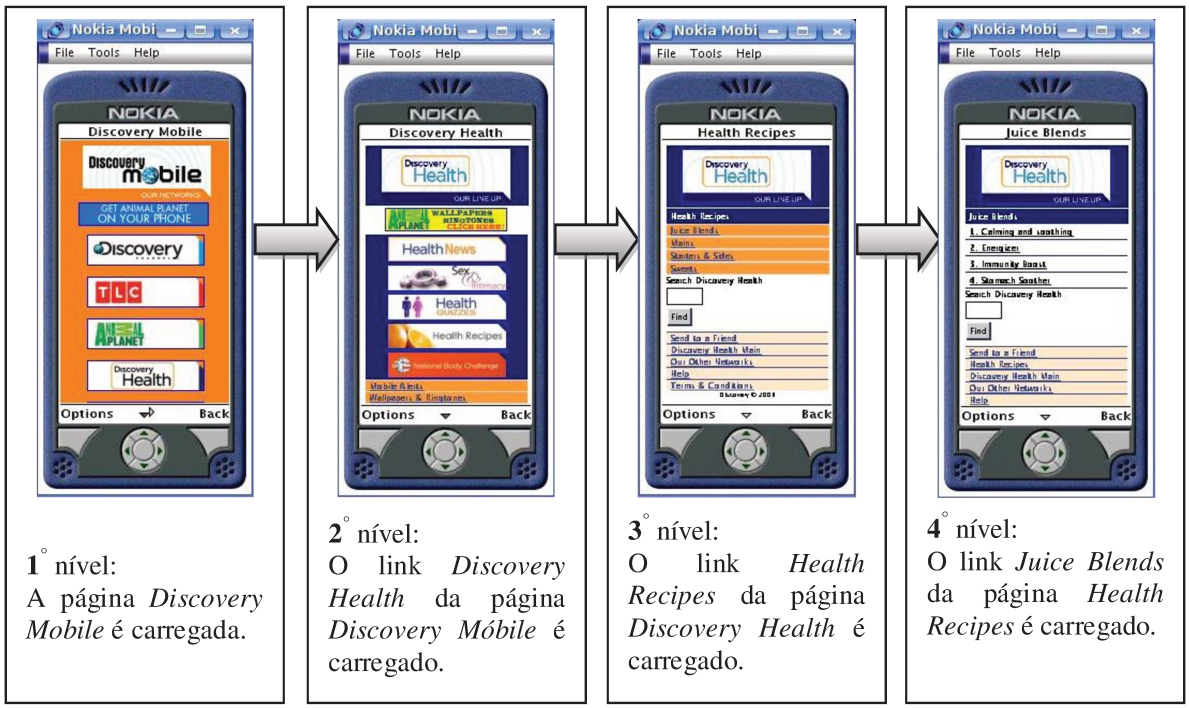
6.2.2 Injeção do Ataque de Truncamento

Um experimento de IF é caracterizado pelo modelo **FAR** [19], onde: **F – Falhas** (*Faults* em inglês) representa a carga de falhas selecionada, **A – Carga de trabalho** (***A****ctivations* em inglês) representa a carga de trabalho usada, e **R – Resultados coletados** (**R***eadouts* em inglês) representa os resultados e dados coletados. A seguir descreveremos os conjuntos usados nestes experimentos.

Carga de trabalho

Um aspecto importante no Teste de Protocolos é a geração do tráfego de rede – a carga de trabalho (**A**). A carga de trabalho são as entradas para ativar a execução do WTLS. Para ser o mais realista possível durante os testes, dever-se-ia gerar tráfego bem próximo do fluxo

real de pacotes de uma rede real. Para este estudo de caso foram usados simuladores para o cliente WAP e o gateway WAP, como mostrado na Figura 6.1. O tráfego gerado consiste em acessos do cliente a páginas WAP reais com propósito de emular um usuário real navegando na internet através do dispositivo móvel. Neste trabalho não julgamos necessário usar uma carga de trabalho representativa como as usadas, por exemplo, para benchmark de robustez, no qual o sistema é submetido a altas cargas de testes com o intuito de realizar medições de desempenho ou verificar se o sistema satisfaz seus requisitos funcionais. Dado que o objetivo do trabalho foi verificar o comportamento em presença de ataques, o enfoque foi a carga de falhas gerada.



1° nível:
A página *Discovery Mobile* é carregada.

2° nível:
O link *Discovery Health* da página *Discovery Móbile* é carregado.

3° nível:
O link *Health Recipes* da página *Discovery Health* é carregado.

4° nível:
O link *Juice Blends* da página *Health Recipes* é carregado.

Figura 6.4: Casos de teste e níveis de navegação.

Para a campanha de injeção foram selecionados **4 sites WAP** (ex.: *wap.yahoo.com*) e foram considerados **4 níveis de navegação** de cada site para serem carregados no browser simulado, i.e., **4 níveis de navegação / site WAP**. O 1° nível corresponde a página principal do site carregada, o 2° nível corresponde a um link da página principal carregado, o 3° nível corresponde a um link da página do 2° nível carregado e o 4° nível corresponde a um link da página do 3° nível carregado. Foi criado um **caso de teste** para carregar cada

nível de navegação de cada site WAP, que resulta em **4 casos de teste / site WAP**. Então para cobrir todos os níveis de navegação dos sites WAP temos **16 casos de teste**. O ataque é sempre injetado quando o último nível de navegação do site WAP é alcançado no browser durante a execução do caso de teste correspondente. A Figura 6.4 mostra os 4 níveis de navegação dos 4 casos de teste do site WAP *www.discoverymobile.com*.

Falhas

O SAE – a carga de falhas (**F**) – obtido na seção 6.2.1 (Figura 6.3) foi usado para emular os ataques contra o cliente. Foi criada uma **instância** desse SAE para cada um dos 16 casos de teste, alterando o parâmetro configurável "último registro" – linha 2 da Figura 6.3 – em cada uma das **16 instâncias** criadas. Quando o browser simulado (NMB) requisita uma página e o gateway WAP responde, o injetor de falhas intercepta os pacotes trocados entre as duas partes e executa as ações do SAE, sendo que a ação **remover** corresponde a ação efetivamente realizada pelo injetor.

Cada instância do SAE criada para o caso de teste correspondente foi executada 90 vezes, i.e., **90 execuções / instância**, o que equivale a **90 execuções / caso de teste**, ou **90 execuções / nível de navegação** de site WAP. Como foram considerados 4 níveis de navegação / site WAP, então temos **360 execuções / site WAP**. Foram selecionados 4 sites WAP, o que resulta em **1440 execuções** do SAE no total.

Foi usado esse número de 90 execuções / nível de navegação porque durante a execução das instâncias do Script foi notado que em média a cada 10 execuções / nível de navegação do caso de teste, o comportamento da aplicação alvo se mostrou diferente. Isto pode ser encarado como uma falha (humana ou de ambiente) de execução do caso teste, o qual pode comprometer 10% das execuções. Assim sendo, para aumentarmos o grau de confiança da execução dos testes e minimizar as falhas introduzidas durante a execução dos testes, o número de execuções / nível de navegação foi aumentado para 90, o qual poderia comprometer no mínimo 1,1% das execuções do caso de teste e no máximo 10% das execuções do caso de teste, dessa forma não impactando no resultado final.

Resultados coletados

A duração de execução de um caso de teste foi de aproximadamente 75 segundos,

totalizando uma duração total de 30 horas de experimentos de injeção. Durante a execução de um caso de teste foram realizados os seguintes passos: **i)** o browser simulado (NMB) carrega o nível de navegação desejado do site WAP; **ii)** a instância do SAE é executada pelo injetor de falhas *Firmament*; **iii)** os pacotes trocados pelas duas partes são coletados pelo Analisador de Protocolo *Wireshark*[36] e os eventos de execução são coletados através do mecanismo de log de *Firmament* – resultados coletados (**R**).

O cenário de ataque <1.1.1.2>, implementado de acordo com o script da Figura 6.3, objetiva violar a propriedade *integridade*, como indicado na Árvore de Ataques da Figura 5.1 (a). Portanto, um defeito para o usuário ocorre quando uma página não é exibida no dispositivo móvel como esperado, como conseqüência da alteração da informação trafegada. Os resultados observados foram então classificados em **AS** (**A**taque bem-**S**ucedido) e **AF** (**A**taque **F**alho). Os diferentes tipos de saída observados são:

- **AS-1**: truncamento no topo da página – o browser falha em ler o conteúdo da parte superior da página.
- **AS-2**: truncamento na base da página – o browser falha em ler o conteúdo da parte inferior da página.
- **AS-3**: outro tipo de truncamento – o browser falha em ler outro tipo de conteúdo da página (ex.: itens de formatação da página, imagens da página).
- **AF**: não houve truncamento – nenhuma alteração do layout da página foi observada, i.e, o browser carregou a página com sucesso.

Os diferentes tipos de **AS** observados demonstram como um ataque bem-sucedido pode causar diferentes tipos de defeitos na aplicação alvo, manifestando diferentes comportamentos externos. De fato quando o último pacote é removido do fluxo de dados do WAP, os dados transportados nesse pacote podem diferir entre as várias sessões WAP estabelecidas, fazendo com que o mecanismo da camada de aplicação da pilha WAP processe os dados recebidos de diferente forma para exibir a página WAP, dessa forma apresentando diferentes efeitos na página exibida no browser WAP.

[36] http://www.wireshark.org

Um **AS** significa que ocorreu uma falha de segurança, i.e., o WTLS não detectou o truncamento da mensagem e o browser exibiu uma página corrompida para o cliente. De fato, o NMB continua esperando e solicitando pelo "último registro" (PDU) por aproximadamente 65 segundos. Depois de passado esse tempo, ele mostra a mensagem de erro: "*Operation aborted by server*". Contudo, ao invés de abortar a exibição do conteúdo da pagina, o NMB exibe a página carregada parcialmente com o conteúdo truncado. A Figura 6.5 mostra um exemplo de exibição do NMB quando o resultado **AS-1** foi observado.

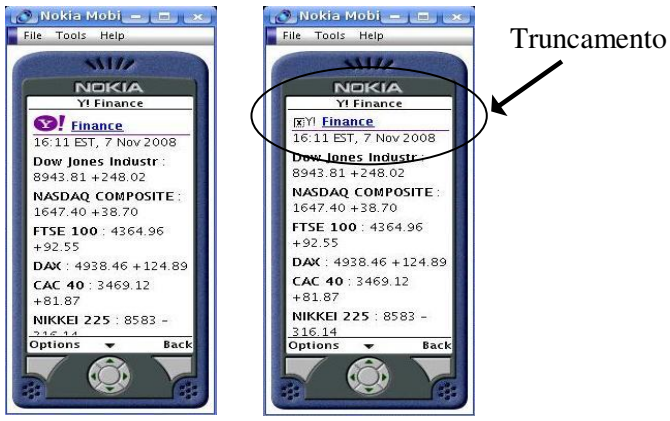


Figura 6.5: Truncamento da imagem no topo da página.

Um **AF** significa que o ataque não obteve sucesso, logo não ocorrendo alteração no comportamento de exibição da página solicitada, pois o SAE não executou a ação de ataque esperada (**remover**). Analisando os pacotes trocados entre o cliente e o gateway WAP, que foram coletados usando a ferramenta *Wireshark*, e também os arquivos de log de execução da *Firmament*, – resultados coletados (**R**) – nós pudemos concluir que esse comportamento do SAE é resultado do modo de operação da interface de rede da Arquitetura de Teste usada, como explicado adiante.

Resultados e Discussão

Os resultados dos ataques para os quatro sites WAP estão sumarizados na Tabela 6.2 e na Figura 6.6.

Site WAP	Execuções / site WAP	AS-1	AS-2	AS-3	AS	AF
1	360	164 (45,6%)	158 (43,9%)	6 (1,7%)	338 (91,1%)	32 (8,9%)
2	360	200 (55,6%)	120 (33,3%)	37 (10,3%)	357 (99,2%)	3 (0,8%)
3	360	47 (13,1%)	92 (25,6%)	89 (24,7%)	228 (63,3%)	132 (36,7%)
4	360	311 (86,4%)	0 (0%)	24 (6,7%)	335 (93,1%)	25 (6,9%)
Total	**1440**	**722 (50,1%)**	**370 (25,7%)**	**156 (10,8%)**	**1248 (86,7%)**	**192 (13,3%)**

Tabela 6.2: Sumário dos resultados do ataque de truncamento.

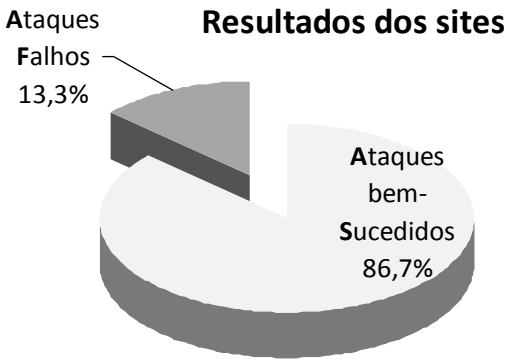


Figura 6.6: Resultados do ataque de truncamento.

A partir da Tabela 6.2 podemos perceber que as porcentagens dos tipos de **AS** (**AS-1**, **AS-2** e **AS-3**) de cada site diferem entre si para todos os sites. Essa diferença ocorre porque foi observado um resultado significativo de um tipo de **AS** para cada nível de navegação do site WAP no qual o ataque foi injetado. Por exemplo, para o site 1 os resultados de **AS-1** e **AS-2** tiveram as maiores porcentagens e esses valores foram semelhantes, pois no primeiro e no segundo níveis de navegação desse site o resultado de **AS-1** prevaleceu e no terceiro e quarto níveis de navegação o resultado de **AS-2** prevaleceu. Também podemos observar que a porcentagem de um tipo de **AS** (ex.: **AS-1**) varia bastante entre os sites, devido ao mesmo fator, pois de acordo com o nível de navegação no qual o ataque foi injetado, um tipo de resultado **AS** prevaleceu.

Analisando os resultados de **AF**, nota-se que o site 3 apresentou maior porcentagem

de resultado **AF** em relação aos outros sites (que possuem um valor mínimo). Também a porcentagem de **AF** do site 3 é maior que cada tipo de **AS** desse mesmo site, diferentemente dos outros sites que tiveram a maior parte de porcentagens de tipo **AS** maior que **AF**. Os casos de teste do site 3 usaram como carga de trabalho (**A**) páginas WAP com mais conteúdo (mais itens de formatação da página) do que os casos de testes dos outros sites. Esse tipo de página requer que mais pacotes de dados sejam transmitidos, fazendo com que a interface de rede do ambiente de teste ajuste o número total de pacotes de dados a serem transmitidos dinamicamente, causando variações no tráfego. Portando o último pacote do fluxo de mensagens não foi identificado pelo SAE, e a ação **remover** não foi executada no "último registro" de dados conduzindo ao resultado **AF**. Isto significa que na vida real em um ambiente similar o atacante precisaria de muitas tentativas para conseguir realizar o ataque de truncamento bem-sucedido para páginas WAP complexas (muitos itens de formatação) ou mensagens muito grandes sendo transmitidas, i.e., quando um grande número de pacote de dados é transmitido entre o servidor e o cliente.

Os resultados apresentados na Figura 6.6 mostram que a porcentagem total de **AS** foi superior a porcentagem total de **AF**, i.e., a ampla maioria de ataques injetados nos sites WAP resultou em uma falha de segurança, violando a propriedade *integridade*. Portanto vulnerabilidades de segurança foram exploradas de forma ampla e precisa.

Podemos calcular a eficiência, que é a relação de vulnerabilidades descobertas (falhas de segurança) pela carga de falhas injetada: vulnerabilidades / carga de falhas = **AS** / ataques injetados = 1248 / 1440 = 86,7%. Esse valor de eficiência se mostra superior aos valores das abordagens tradicionais de Teste Segurança baseadas na técnica de Fuzz e Teste de Sintaxe, onde grande proporção dos experimentos de injeção de ataques (30% no mínimo [40]) resulta em nenhum efeito perceptível, i.e., são irrelevantes e não ativam nenhuma vulnerabilidade de segurança, como é mostrado na Tabela 3.1 e discutido na seção 3.3. Portanto, esses resultados obtidos são promissores.

O valor da duração total dos experimentos foi reflexo do número de execuções do mesmo caso de teste usando a mesma instância de SAE (90 execuções / caso de teste). Os motivos para o uso desse valor alto de "execuções / caso de teste" foram explicados anteriormente.

Também poderíamos usar o SAE da Figura 6.3 para criar novas **variações** desse

Cenário de Ataque (novos ataques), i.e., alterar o parâmetro configurável da linha 2 do script para, por exemplo, um valor de pacote que corresponde a um registro intermediário ou um registro inicial do fluxo de pacotes WAP, e não apenas o "último registro" da requisição WAP. Dessa forma seriam criados **novos ataques** baseados no SAE definido, onde apenas seria alterado o parâmetro configurável do SAE. A vantagem de se criar novos ataques a partir desse nível de refinamento (a saída da etapa **6** da metodologia) é que os SAEs já estariam disponíveis para serem executados pelo injetor *Firmament*, não necessitando executar nenhuma outra etapa de refinamento.

O objetivo dos experimentos dessa seção não foi detectar novas vulnerabilidades na implementação do WTLS, mas validar a abordagem proposta usando vulnerabilidades reportadas do protocolo alvo. Além disso, variações do ataque citadas anteriormente, as quais configuram novos ataques, poderiam ser usadas para detectar novas vulnerabilidades. Portanto, a abordagem pode servir tanto para detectar vulnerabilidades conhecidas quanto para detectar novas vulnerabilidades usando variações de ataques bem-sucedidos.

6.3 Ataque de Negação de Serviço (DoS)

O cenário de ataque *<1.4.1 – Ataque de DoS para exaustão de CPU (início do handshake)>*, obtido na seção 5.4 de acordo com a etapa **4** (seção 4.4) da metodologia, representa um tipo de ataque de DoS contra o gateway WAP que é relativamente fácil de ser executado e pode causar sérios danos.

Embora baseado em conexões não-confiáveis (funciona sobre o WDP/UDP), o protocolo WTLS especifica apenas temporizadores de conexão e *timeouts* de retransmissão para o cliente WAP, não especificando nenhum temporizador de conexão para o servidor WAP, o que torna o protocolo suscetível a ataques de DoS que podem exaurir a CPU e/ou memória do servidor WAP. Para executar tal ataque, um atacante precisa apenas usar um endereço de IP forjado e enviar várias mensagens. Usando esse procedimento simples, o atacante pode fazer com que o servidor WAP pare de fornecer serviços requisitados por clientes legítimos [44].

O ataque de DoS do Cenário <1.4.1 > é um tipo de ataque para exaustão de CPU. O ataque ocorre quando o atacante, agindo como um cliente legítimo abandona a fase de

handshake (explicada no apêndice B.1) anormalmente depois de enviar uma mensagem de *Client Hello* com o endereço de IP forjado. O servidor WAP processa a mensagem de *Client Hello* do atacante, constrói e envia as mensagens de *Server Hello*, *Server Key Exchange* e *Server Hello Done* para o endereço forjado. Logo, se o atacante enviar inúmeras mensagens hostis de *Client Hello* ele irá causar a exaustão da CPU do servidor WAP, pois o servidor estará ocupado processando e respondendo às inúmeras requisições porque não sabe que o endereço da requisição é forjado.

Para simular esse ataque foi utilizada a Arquitetura de Teste descrita na seção 6.1, onde o simulador de browser (NMB) e o injetor de pacotes NPG foram executados na máquina com SO Windows. O gateway WAP foi executado em outro ambiente Windows como uma máquina virtual, para que houvesse maior "controlabilidade" dos experimentos de injeção, i.e., todos os recursos do ambiente, especialmente a CPU, estivessem disponíveis para o gateway WAP e pudessem ser monitorados.

6.3.1 Transformação do Script de Ataque Genérico

O refinamento do Cenário de Ataque <1.4.1> (etapa **5** da metodologia) segue os mesmos passos mostrados para o Cenário <1.1.1.2> na seção 5.4 e não serão repetidos aqui. Nessa seção iremos apresentar apenas o SAE obtido (etapa **6** da metodologia), exibido na Figura 6.7 e explicado a seguir.

Usando a transformação indicada na etapa **6** (seção 4.6) com a saída da etapa **5** e as instruções do injetor NPG obtivemos o SAE para o NPG, que é mostrado na Figura 6.7.

O script da Figura 6.7 consiste de duas partes. A primeira parte (PARTE 1) possui três parâmetros configuráveis que são definidos na instância de script antes de sua execução no caso de teste: **1)** `<repeat_count>`, na linha 2 do script, é o valor que especifica o número de vezes que o pacote é injetado; **2)** `<time_interval>`, também na linha 2 do script, especifica o intervalo de tempo em milissegundos entre a injeção de um pacote e outro; **3)** `<interface_device>`, na linha 4 do script, especifica o dispositivo de interface de rede no qual serão injetados os pacotes.

```
1.  #### WTLS Client Hello request ####  PARTE 1
2.  [XXXXXX,xxx]        # [<repeat_count>, <time_interval>]
3.  <WTLS Client Hello> # <ID do Pacote>
4.  rpcap://\Device\NPF_XXXX)  # <interface_device> = NPF_XXXX
5.  ## ETHERNET2 HEADER ----  PARTE 2 – Estrutura do pacote
6.  08 00 27 98 2c ce    # Destination MAC: 10.1.1.4 ← Configurado
7.  00 1b fc 26 d7 7a    # Source MAC: 10.1.1.3  ← Forjado
8.  08 00                # Protocol: IP
9.  ## IP HEADER -----------
10. 45                   # Version / Header Length: 4 / 20 bytes
11. 00                   # Type of service: 00
12. 00 55                # Total length: 85
13. ff c7                # Identification: 65479
14. 00 00                # Flags / Fragment offset: 0
15. 80                   # Time to live: 128
16. 11                   # Protocol: UDP
17. 24 c7                # Checksum  ← Calculado
18. 0a 01 01 03          # Source address: 10.1.1.3 ← Forjado
19. 0a 01 01 05     # Destination address - 10.1.1.4 ← Configurado
20. ## UDP HEADER ----------
21. 0f b9                # Source port: 4025
22. 23 f3                # Destination port : 9203 (Gateway WAP)
23. 71 da                # Checksum  ← Calculado
24. ## WTLS HEADER ----------
25. c3                   # Record Type: handshake (3)
26. 00 00                # Record Sequence: 0
27. 00 34                # Record Length: 52
28. 01                   # Type: Client Hello (1)
29. 00 31                # Lenght: 49
30. 01                   # Version: 1
31. 49 4d 66 2b          # Time GMT
32. d5 7c 50 f4 f9 d4 37 e5 9e 79 d4 08  # Random
33. 00                   # Session ID: Null
34. 00 12 08 00 00 0a 00 00 09 00 00 05 00 00 07 \
    00 00 06 00 00       # Client Keys
35. 00 00                # Trusted Keys
36. 04 03 03 02 02       # Cipher Suites
37. 01 00                # Compression Methods
38. 02                   # Sequence Mode: Explict
39. 04                   # Refresh: 4
```

Figura 6.7: Script da NPG.

A segunda parte do script (PARTE 2), a partir da linha 5, consiste da estrutura do pacote a ser injetado, que foi construída usando a sintaxe do NPG. Alguns parâmetros precisam ser pré-definidos na instância do script: **1)** os endereços MAC e IP da máquina de destino (gateway WAP) nas linhas 6 e 19 do script, respectivamente; **2)** os endereços forjados de MAC e IP da máquina de origem nas linha 7 e 18 do script, respectivamente; **3)** os *checksums* dos cabeçalhos IP e UDP nas linha 17 e 23 do script, respectivamente, que foram calculados usando a ferramenta *Wireshark*. Uma vez definidos esses valores de parâmetros para o Ambiente de Teste, não existe necessidade de atualizá-los durante a execução dos experimentos, pois são valores fixos.

O **SAE** da Figura 6.7 corresponde a um tipo de falha específica (capacidade do atacante **personificar**, descrita na seção 4.2), sendo que o parâmetro `<repeat_count>` da PARTE 1 do script é usado para designar o número de falhas a serem injetadas para cada instância do script. Portanto a carga de falhas gerada é específica para a realização do ataque de DoS. Também podemos notar que foi usado um tipo de falha diferente (ação **personificar**) da falha usada no Cenário de Ataque da seção 6.2.1 (ação **remover**), para simular esse Cenário de ataque.

Vale notar que, a partir do mesmo Cenário <1.4.1>, pode-se criar SAEs para diferentes injetores de pacotes[37], o que não se encontra em outras abordagens de Testes de Segurança baseadas em injeção de ataques, como pode ser visto na Tabela 3.1.

6.3.2 Injeção do Ataque de Negação de Serviço

Os experimentos realizados para esse tipo de ataque são descritos a seguir.

Carga de trabalho

Durante a injeção dos ataques nos casos de testes foram realizadas tentativas de carregar páginas WAP (ex.: http://wap.yahoo.com) periodicamente (a cada 5 minutos) usando o simulador de browser. Essas tentativas representam a carga de trabalho (**A**), i.e., entradas legítimas para ativar a execução do WTLS, e verificam a *disponibilidade* dos serviços do

[37] Outros injetores de pacotes incluem: Nemesis (http://nemesis.sourceforge.net/), Hping (http://www.hping.org/), Yersinia (http://www.yersinia.net/), Winject (http://home19.inet.tele.dk/moofz/).

gateway WAP, i.e., se um usuário legítimo é capaz de utilizar os serviços oferecidos pelo gateway WAP.

Falhas

O SAE obtido na seção anterior (Figura 6.7) representa a carga de falhas (**F**). A PARTE 2 do script possui a estrutura do pacote a ser injetado, que corresponde à ação **personificar** do injetor. Uma instância desse SAE pode injetar um número de falhas (pacotes) configurado através do parâmetro `<repeat_count>` da PARTE 1 do script.

Foram criados **4 casos de testes** para realização da campanha de injeção, onde cada caso de teste usa um número diferente de instâncias do SAE, como é explicado nos próximos parágrafos.

No **caso de teste 1** foi usada **1 instância** do SAE da Figura 6.7, onde foram alterados os parâmetros configuráveis da PARTE 1 do script da seguinte forma: `<repeat_count>` = 540000 pacotes, `<time_interval>` = 10 milissegundos (ms) e `<interface_device>` = "Intel Network Interface Controller". O uso desses valores para os parâmetros `<repeat_count>` e `<time_interval>` equivale à duração média de execução de um caso de teste que foi estipulada em 90 minutos, i.e., 1 pacote é injetado a cada 10 ms, que equivale a 5400000 ms = 5400 segundos = 90 minutos de duração de execução.

No **caso de teste 2** foram criadas **11 instâncias** do SAE da Figura 6.7 para serem executadas simultaneamente, já que uma única instância usada no caso de teste 1 não foi suficiente para exaurir o recurso de CPU, como explicado na análise dos Resultados coletados. Para os parâmetros da PARTE 1 de cada instância foram usados os mesmos valores dos parâmetros da instância usada no caso de teste 1. Na PARTE 2 de cada instância foram alterados os seguintes parâmetros: **i)** os endereços forjados de MAC e IP da máquina de origem, os quais foram modificados de forma seqüencial, por exemplo, na linha 7 do SAE o valor do MAC corresponde ao IP 10.1.1.3, então a próxima instância criada possui um valor do MAC corresponde ao IP 10.1.1.4 e assim por diante; **ii)** os *checksums* dos cabeçalhos IP e UDP, que foram modificados com o auxilio da ferramenta *Wireshark*.

No **caso de teste 3** foram criadas **22 instâncias** do SAE da Figura 6.7, onde os parâmetros da PARTE 1 e da PARTE 2 dessas instâncias foram alterados da mesma

maneira que no caso de teste 2. As instância usadas no caso de teste 2 não foram suficientes para exaurir o recurso de CPU, como explicado na análise dos Resultados coletados.

No **caso de teste 4** foram criadas **33 instâncias** do SAE, também alterando os parâmetros da PARTE 1 e da PARTE 2 das instâncias da mesma maneira que no caso de teste 2. Nesse caso de teste foi usado um número de instâncias maior que no caso de teste 3 para assegurar que o comportamento do gateway WAP e do browser seriam os mesmos em relação ao caso de teste 3.

A duração de execução de cada caso de teste foi em média de 90 minutos, totalizando 360 minutos de injeção de ataques para todos os casos de teste, i.e., 360 minutos de campanha de injeção.

As alterações, nos casos de teste 2, 3 e 4, dos parâmetros da PARTE 2 das instâncias do script podem ser caracterizadas como **variações** do SAE da Figura 6.7, já que a alteração desses valores pré-definidos, como o endereço forjado da máquina de origem, não foi descrita no Cenário de Ataque original <1.4.1> na seção 6.3.1, i.e, era esperado que o Cenário de Ataque <1.4.1> fosse injetado alterando apenas os parâmetros da PARTE 1.

Resultados coletados

A partir dos resultados coletados (**R**) pelo Analisador de Protocolo *Wireshark* e pelo mecanismo de log de *Firmament* em todos os casos de teste: os pacotes injetados pelo NPG (mensagens de *Client Hello*) e as respectivas repostas do gateway WAP (mensagens de *Server Hello*, *Server Key Exchange* e *Server Hello Done*); foi observado que o gateway WAP processou as mensagens de *Client Hello* enviadas pelo NPG e respondeu a todas as requisições de acordo com o esperado.

Durante a execução da instância do **caso de teste 1**, a porcentagem média de uso da CPU pelo gateway WAP foi baixa, em torno de 10% (não consumiu toda CPU disponível) e atingiu picos de 15%, como pode ser visto na Tabela 6.3; também se mantendo constante ao longo da execução do caso de teste, como é mostrado no gráfico da Figura 6.8 (a). Mesmo quando o valor do parâmetro `<time_interval>` da instância do script foi aumentado até o nível máximo (1 ms), i.e., um pacote era injetado a cada 1 milissegundo, a porcentagem média de uso da CPU não aumentou. Este fato explica a escolha do valor de 10 ms para esse parâmetro, para garantir que a maioria dos pacotes injetados fossem

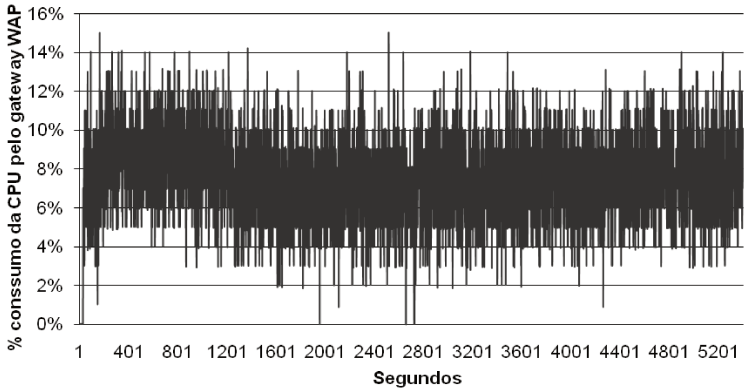
processados. Esse comportamento mostra que o gateway WAP processa até um valor limite de requisições feitas por um mesmo endereço de IP, ignorando as outras requisições. Esse mecanismo de proteção do gateway WAP contra ataques de DoS que usam várias mensagens de *Client Hello* com o mesmo endereço é descrito na especificação do WTLS [25]. O ataque desse caso de tese não foi bem-sucedido, como é mostrado na Figura 6.8 (a).

No **caso de teste 2** observou-se que houve aumento significativo de uso médio da CPU em relação ao caso de teste 1, de 10% para 35%, como é mostrado na Tabela 6.3. Mas a porcentagem média de uso da CPU pelo gateway WAP também se manteve constante ao longo da execução do caso de teste (não consumindo toda a CPU disponível), onde o máximo de CPU consumida foi de 47%, como é mostrado no gráfico da Figura 6.8 (b) e na Tabela 6.3. Esse comportamento é explicado pelo mesmo motivo do caso de teste 1 – o mecanismo de proteção do gateway WAP contra ataques de DoS, que ignora requisições do mesmo endereço se recebidas com muita freqüência. Logo, o ataque desse caso de tese também não foi bem-sucedido.

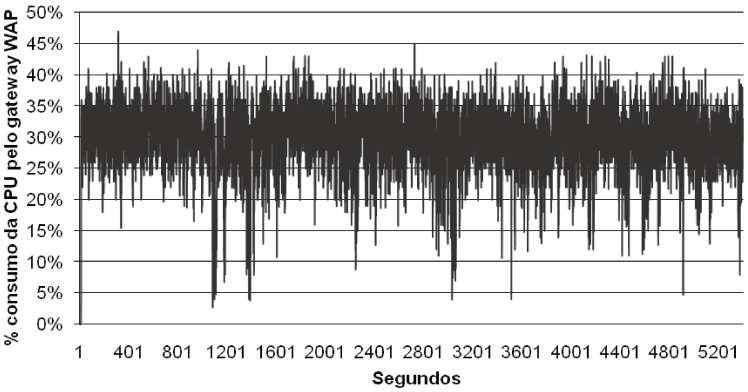
No **caso de teste 3**, onde foi usado um valor de instâncias maior que no caso de teste 2, foi observado que o uso médio de CPU aumentou significativamente em relação ao caso de teste 2, de 35% para 78%, como mostrado na Tabela 6.3. Essa porcentagem média de uso de CPU pelo gateway WAP chegou ao seu valor máximo ao longo da execução do caso de teste (consumindo toda a CPU disponível), com picos de 84%, como pode ser visto no gráfico da Figura 6.8 (c). Essa média é considerada máxima porque durante a execução do caso de teste, em torno de 21% de CPU foi usada pelo serviço de rede. Então podemos considerar que praticamente toda a CPU foi consumida (em torno de 99%), exaurindo totalmente esse recurso do sistema. Isso mostra que o mecanismo de defesa do gateway WAP contra ataques de DoS não foi suficiente para evitar que o recurso de CPU fosse totalmente consumido tornando indisponíveis os serviços para requisições legítimas, como é explicado adiante. O ataque desse caso de tese foi bem-sucedido.

No **caso de teste 4** a porcentagem média de uso de CPU pelo gateway WAP foi de 65%, atingindo picos de 77%, como pode ser visto na Tabela 6.3 e no gráfico da Figura 6.8 (d). Nesse caso de teste também foi observado que a CPU foi totalmente consumida como no caso de teste 3, pois em média 34% de CPU foi usada para o serviço de rede. Logo, o ataque desse caso de tese também foi bem-sucedido.

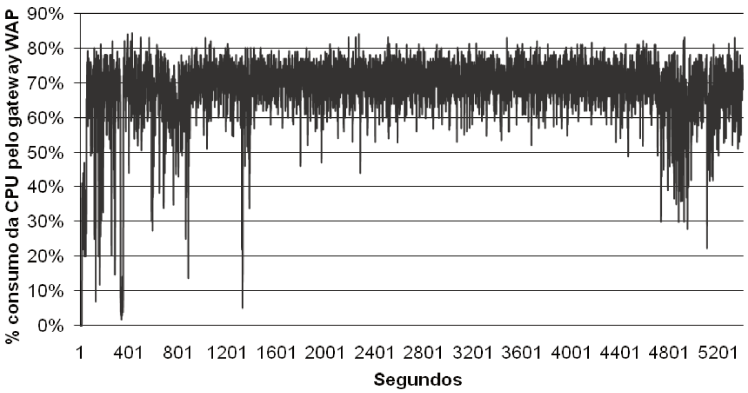
A Figura 6.8 mostra os gráficos da porcentagem de uso de CPU pelo processo do gateway WAP ao longo de cada execução de caso de teste. A porcentagem de uso da CPU foi medida a cada segundo durante a execução do caso de teste, sendo realizadas 5400 medições para cada caso de teste. A Tabela 6.3 exibe as respectivas médias de uso de CPU calculadas e os picos de uso, juntamente com o consumo médio total de CPU.



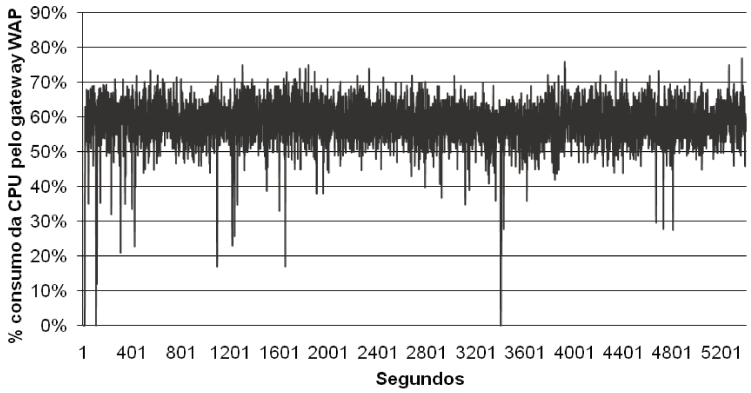
(a) Gráfico de uso da CPU do caso de teste 1



(b) Gráfico de uso da CPU do caso de teste 2



(c) Gráfico de uso da CPU do caso de teste 3



(d) Gráfico de uso da CPU do caso de teste 4

Figura 6.8: Resultados dos casos de teste.

Caso de teste	Gateway WAP		Média de uso da CPU pelo serviço de rede	Média de uso total da CPU
	Média de uso da CPU	Máximo de uso da CPU		
1	10%	15%	1%	**11%**
2	35%	47%	9%	**44%**
3	78%	84%	21%	**99%**
4	65%	77%	34%	**99%**

Tabela 6.3: Sumário dos resultados do ataque de DoS.

Resultados e Discussão

O objetivo desse Cenário de Ataque <1.4.1> foi violar a propriedade *disponibilidade* consumindo o recurso de CPU disponível, dessa forma impossibilitando usuários legítimos de usar os serviços do gateway WAP. Portanto, um defeito para o usuário ocorre quando ele requisita um serviço o qual tem direito e esse é negado.

No caso de teste 1, apesar do gateway WAP estar sob ataque de DoS, ele respondeu a todas as requisições legítimas feitas pelo NMB, carregando todas as páginas WAP requisitadas sem apresentar nenhum defeito para o usuário, pois nesse caso de teste a CPU

não foi totalmente consumida. No caso de teste 2, o gateway WAP também respondeu a todas as requisições feitas pelo NMB, mas as páginas WAP com mais conteúdo (mais itens de formatação da página) requisitas foram carregadas lentamente, pois o gateway WAP levou mais tempo para responder às requisições, i.e., houve uma queda de desempenho do gateway WAP para esse tipo de página, o que não representa um defeito, pois não houve divergência do serviço oferecido. Portanto, o ataque de DoS injetado através dos casos de teste 1 e 2 não obteve sucesso.

Nos casos de teste 3 e 4, o gateway WAP não respondeu às requisições legítimas feitas pelo NMB e as páginas WAP não foram carregadas pelo browser. Nesses casos de teste a CPU foi totalmente consumida, impossibilitando o gateway WAP de processar essas requisições legítimas. Ao invés disso, o NMB esperou aproximadamente 65 segundos pela reposta do gateway WAP, e depois desse tempo (*timeout*) ele exibiu a mensagem de erro: "*Unable to establish secured connection*". O ataque de DoS injetado nos casos de teste 3 e 4 foi bem-sucedido, i.e., houve um defeito, pois o serviço requisitado foi negado para o usuário legítimo. Isso mostra que o mecanismo de defesa do gateway WAP contra ataques de DoS não funcionou para evitar que o recurso de CPU fosse totalmente consumido.

O alto consumo de CPU pelo gateway WAP, como mostrado na Figura 6.8 (c) e na Figura 6.8 (d), e a negação de requisições legítimas, ambos observados nos casos de teste 3 e 4, mostra que a ampla maioria dos ataques injetados contra o gateway WAP nesses casos de teste resultou em falhas de segurança, violando a propriedade *disponibilidade*. Esses casos de teste mostraram boa eficiência para explorar vulnerabilidades de segurança, i.e, todos os ataques injetados nos casos de teste 3 e 4 geraram uma falha de segurança. As técnicas de Fuzzing e Teste de Sintaxe não permitem a realização de ataques dessa classe, que tentam violar a *disponibilidade* do sistema alvo.

A duração total de execução dos casos de teste poderia ser reduzida para um valor menor que 90 minutos, pois como foi observado nos gráficos dos casos de teste 3 e 4 na Figura 6.8 (c) e na Figura 6.8 (d), o consumo de CPU pelo gateway WAP se mostrou alto (a CPU foi totalmente consumida) e constante durante a injeção dos pacotes, i.e., o gateway WAP se mostrou vulnerável durante a maior parte da execução desses casos de teste. Então poderíamos reduzir a duração de execução dos casos de teste para um valor mais baixo.

As novas **variações** do SAE da Figura 6.7 usadas nos casos de teste 2, 3 e 4, como

explicado anteriormente, corresponderiam a um novo tipo de ataque de DoS contra o WTLS, onde múltiplas instâncias enviam inúmeras requisições hostis com diferentes endereços forjados para tornar os serviços do gateway WAP indisponíveis. Este ataque é similar ao ataque de DDoS (*Distributed Denial-of-Service*)[38], onde múltiplos sistemas (máquinas) comprometidos são controlados pelo atacante para lançar ataques coordenados contra o sistema alvo. O atacante instala previamente um software de DDoS nessas máquinas para permitir tal controle. Os ataques lançados consomem a largura de banda, ou a capacidade de processamento de um roteador, ou os recursos da rede, interrompendo a conectividade de rede e impossibilitando o acesso para as vítimas.

Como foi declarado no final da seção 6.2.2, o objetivo do experimento não foi detectar novos tipos de vulnerabilidades na implementação do WTLS, mas sim validar a abordagem proposta através da realização de experimentos, que usam ataques reportados do WTLS. Conseguimos mostrar também que a partir do ataque de DoS reportado foi possível criar novas variações desse ataque com a finalidade de atingir o objetivo final, violar a propriedade *disponibilidade* do protocolo.

6.4 Ataque de Cipher Rollback

O Cenário de Ataque *<1.1.3.1– Ataque de rollback no cipher suite>*, obtido na seção 5.4 de acordo com a etapa **4** (seção 4.4) da metodologia, é um tipo de ataque MITM contra o cliente WAP que objetiva violar a propriedade *integridade*.

Essa vulnerabilidade de segurança foi reportada para o protocolo de *Handshake* do SSL v.2.0 [27]. Nesse caso o atacante modifica silenciosamente a lista de preferências de *cipher suites* (algoritmo de cifras de bloco e algoritmo de MAC) da mensagem de *Client Hello*, que é enviada de forma não-protegida (não-cifrada e não-autenticada) do cliente para o servidor WAP durante o *handshake* (explicado no apêndice B.1), para forçar ambos os lados a usar uma forma de ciframento mais fraca, mesmo se ambos os lados poderiam usar algoritmos mais fortes [27].

Esse ataque pode ser usado como um pré-requisito para ataques de força bruta, os

[38] http://www.linuxsecurity.com/resource_files/intrusion_detection/ddos-whitepaper.html

quais objetivam violar a *privacidade* do protocolo. Ataques de força bruta podem quebrar algoritmos de ciframento mais fracos e explorar a chave de sessão (*key session*) usada para o ciframento das mensagens. Um exemplo é o Cenário de Ataque *<1.2.1.1 – Quebrar o ciframento do algoritmo DES 40bit>* da Figura 5.1 (a), que objetiva quebrar a chave de 40 bits do algoritmo de cifras de bloco DES-40 [25], que é considerado fraco e dificilmente é selecionado no *handshake*, mesmo se disponibilizado por ambas as partes. A especificação do WTLS [25] declara que cifras de 40 bits são altamente suscetíveis a ataques de busca exaustiva. Portanto é fortemente recomendado que elas não sejam usadas, e se possível descontinuadas. Servidores WAP podem precisar usar cifras de 40 bits por razões de compatibilidade com versões anteriores, mas clientes não devem usar cifras de 40 bits. O atacante então altera os algoritmos de ciframento da lista de *cipher suites* para esse algoritmo (DES_CBC_40), forçando ambas as partes a escolher o *cipher suite* com esse algoritmo fraco e assim possibilitando a execução do Cenário de Ataque <1.2.1.1>.

Para simular esse ataque foi utilizada a Arquitetura de Teste descrita na seção 6.1, onde o injetor de falhas *Firmament* foi executado em ambiente Linux como uma máquina virtual; e o simulador de browser (NMB) e o gateway WAP foram executados na máquina Windows.

6.4.1 Transformação do Script de Ataque Genérico

O refinamento do Cenário de Ataque <1.1.3.1> (etapa **5** da metodologia) segue os mesmos passos mostrados para o Cenário <1.1.1.2> na seção 5.4 e não serão repetidos aqui. Nessa seção iremos apresentar apenas o **SAE** obtido (etapa **6** da metodologia), exibido na Figura 6.9 e explicado a seguir.

Os valores de *cipher suites* (`0xXXXXXXXX`) da linha 2 da Figura 6.9, atribuídos à variável do script, serão alterados de acordo com a instância de script usada no caso de teste para executar o ataque, como será explicado na seção 6.4.2. Esse é o único parâmetro configurável do SAE.

O SAE possui uma carga de falhas mínima, i.e., possui apenas a falha (instrução "**`WRTEW`**" na linha 34, que corresponde à capacidade do atacante **corromper**) necessária para a realização do ataque.

```
1.  INIT:            # converter.getProperties
2.    SET 0xXXXXXXXX R5   # cipher_Suites = CIPHER_SUITES(0xXXXXXXXX)
3.    SET 0x11 R6    # UDP            = 17
4.    SET 0x23F3 R7  # gateway_port = 9203
5.    SET 0x03 R8    # handshake = 3
6.    SET 0x01 R9    # client_Hello = 01
7.    SET 0x9 R0     # converter.getProtocolType(pacote) == UDP
8.    READB R0 R1
9.    SUB R1 R6
10.   JMPZ R6 IS_UDP
11.   JMP ERROR_OK
12. IS_UDP:
13.   SET 0x16 R0 #converter.getDestinationPort(p.) == gateway_port
14.   READS R0 R1
15.   SUB R1 R7
16.   JMPZ R7 TO_GATEWAY
17.   JMP ERROR_OK
18. TO_GATEWAY:     #### Regra 1
19.   SET 0x1C R0   # converter.getPDUType(pacote) == handshake
20.   READB R0 R1
21.   SET 0x0F R0
22.   AND R0 R1
23.   SUB R8 R1
24.   JMPZ R1 PDU_HANDS
25.   JMP ERROR_OK
26. PDU_HANDS:
27.   SET 0x21 R0   # converter.getPDUField(pacote,…) == client_Hello
28.   READB R0 R1
29.   SUB R9 R1
30.   JMPZ R1 DO_CORRUP
31.   JMP ERROR_OK
32. DO_CORRUP:
33.   SET 0x4D R0
34.   WRTEW R0 R5   # converter.CorruptPDUField(pacote,…)
45.   ACP
36.   JMP INIT      # Fim do ataque
37. ERROR_OK:
38.    ACP
39.    JMP INIT
```

Figura 6.9: Script da *Firmament*.

6.4.2 Injeção do Ataque de Cipher Rollback

Os experimentos realizados para esse tipo de ataque são descritos a seguir.

Carga de trabalho e Resultados coletados

A execução de um caso de teste consiste: **i)** acesso a uma página WAP (ex.: http://wap.yahoo.com), o qual gera uma requisição ao cliente (mensagem de *Client Hello* com a lista de *cipher suites*), que é processada pelo gateway WAP. Esta é a entrada que ativa a execução do WTLS – a carga de trabalho (**A**). **ii)** execução de uma instância do SAE pelo injetor de falhas *Firmament*, o qual injeta a ação **corromper** para alterar os *cipher suites*. **iii)** coleta dos pacotes trocados entre as duas partes usando a ferramenta *Wireshark*; coleta dos eventos de execução usando o mecanismo de log de *Firmament*; e coleta dos resultados observados no simulador de browser – estes são os resultados coletados (**R**).

Falhas

Durante o estabelecimento de uma sessão segura, como explicado no apêndice B.1, o cliente WAP (simulador de browser) propõe ao gateway WAP uma lista padrão com dois valores de *cipher suites* (algoritmo de cifras de bloco e algoritmo de MAC): **1)** RC5_CBC, SHA (0x0303); **2)** RC5_CBC_56, SHA_80 (0x0202). No primeiro *cipher suite* da lista o algoritmo de cifras de bloco RC5_CBC [25] possui 128 bits de chave e algoritmo de MAC SHA possui um MAC de 20 bytes; no segundo *cipher suite* da lista o algoritmo de cifras de bloco RC5_CBC_56 [25] possui 56 bits de chave e algoritmo de MAC SHA_80 possui um MAC de 10 bytes. Para realização dos experimentos foram usados *cipher suites* mais fracos do que esses valores padrões propostos pelo cliente.

Esses algoritmos foram escolhidos de acordo com as tabelas de algoritmos disponíveis da especificação do WTLS [25]. Os algoritmos de ciframento da família IDEA_CBC [25] não foram selecionados, pois não são usados pelo gateway WAP da Arquitetura de Teste (*Columbitech WAP Connector*). Os algoritmos selecionados são mostrados na Tabela 6.4.

Algoritmo de ciframento	Bits de chave (bits)	Algoritmo de MAC	Tamanho de MAC (bytes)
NULL	0	SHA_0	0
DES_CBC_40	40	MD5_40	5
RC5_CBC_40	40	SHA_40	5
DES_CBC	56		

Tabela 6.4: Algoritmos de ciframento e de MAC.

Em seguida, os algoritmos selecionados foram combinados de forma a gerar *cipher suites* mais fracos do que os listados no começo da seção. As listas com os valores dos *cipher suites* criados é mostrada na Tabela 6.5.

O SAE da Figura 6.9 representa a carga de falhas (**F**) - a ação **corromper** do injetor. Cada caso de teste corresponde a uma instância desse script. Ao todo, foram criados **12 casos de testes** para execução da campanha de injeção. Cada caso de teste usa uma entrada da Tabela 6.5 como o valor do parâmetro configurável do SAE da Figura 6.9. Por exemplo, a entrada 2 da Tabela 6.5 corresponderia ao valor 0x04000400 atribuído à variável do script (o parâmetro configurável do script).

Cada caso de teste foi executado três vezes usando como entrada diferentes páginas WAP para assegurar que a alteração da entrada não influenciaria na análise dos resultados observados. Ao todo foram realizadas **36 execuções**, i.e., **3 execuções / caso de teste**. A duração da execução de um caso de teste foi de aproximadamente 45 segundos, totalizando uma duração 27 minutos de campanha de injeção.

Lista	*Cipher suites*
1	NULL, SHA_0 (0x0000) NULL, SHA_0 (0x0000)
2	DES_CBC_40, SHA_0 (0x0400) DES_CBC_40, SHA_0 (0x0400)
3	DES_CBC_40, MD5_40 (0x0405) DES_CBC_40, MD5_40 (0x0405)
4	DES_CBC_40, SHA_40 (0x0401) DES_CBC_40, SHA_40 (0x0401)
5	RC5_CBC_40, SHA_0 (0x0100) RC5_CBC_40, SHA_0 (0x0100)
6	RC5_CBC_40, MD5_40 (0x0105) RC5_CBC_40, MD5_40 (0x0105)
7	RC5_CBC_40, SHA_40 (0x0101) RC5_CBC_40, SHA_40 (0x0101)
8	DES_CBC, SHA_0 (0x0500) DES_CBC, SHA_0 (0x0500)
9	DES_CBC, MD5_40 (0x0505) DES_CBC, MD5_40 (0x0505)
10	DES_CBC, SHA_40 (0x0501) DES_CBC, SHA_40 (0x0501)
11	RC5_CBC_56, SHA_80 (0x0202) RC5_CBC_56, SHA_80 (0x0202)
12	RC5_CBC, SHA (0x0303) RC5_CBC, SHA (0x0303)

Tabela 6.5: Listas de *cipher suites*.

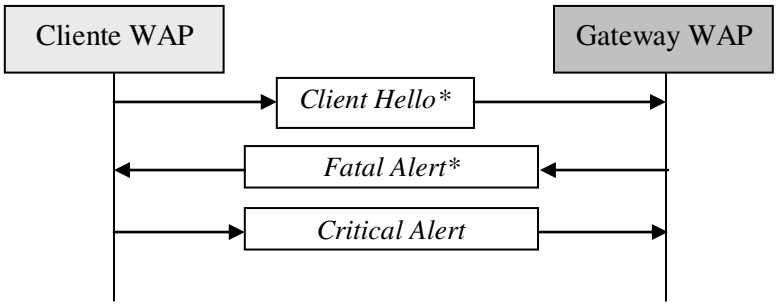
Resultados e Discussão

O objetivo desse Cenário de Ataque <1.1.3.1> é violar a propriedade *integridade* alterando a lista de *cipher suites* enviada pelo cliente para algoritmos mais fracos. Portanto, um defeito ocorre quando o servidor WAP concorda em usar um *cipher suite* mais fraco na

lista de *cipher suites* alterada sendo que habitualmente ele escolheria um *cipher suite* forte, e o cliente também aceita usar o *cipher suite* mais fraco que o servidor concordou em usar. O *handshake* é finalizado e as mensagens de dados trocadas são cifradas e autenticadas usando os algoritmos mais fracos selecionados.

A análise das informações coletadas (**R**) permitiu determinar três tipos de comportamento para os casos de teste executados, como descrito nos próximos parágrafos.

No **caso de teste 1**, onde a troca de mensagens é mostrada na Figura 6.10, os *cipher suites* foram alterados na mensagem de *Client Hello* pela instância do script (atacante), o gateway WAP respondeu com a mensagem de Alerta fatal *handshake_failure* de tratamento de erros do WTLS, o qual indica que o remetente (gateway) foi incapaz de negociar um conjunto de parâmetros de segurança aceitáveis dada as opções disponíveis, i.e., o gateway não aceitou o valor de *cipher suite* enviado pelo browser (NULL, SHA_0). O browser então enviou a mensagem de *Client Hello*, a qual foi modificada novamente pelo atacante, mas o gateway respondeu novamente com a mensagem de Alerta fatal *handshake_failure*. No total o browser executou seis tentativas de envio da mensagem de *Client Hello*, que foram modificadas pelo atacante, mas todas as tentativas foram rejeitadas pelo gateway WAP com a mensagem de Alerta fatal *handshake_failure*. Apesar da especificação do WTLS [25] declarar que, ao receber uma mensagem de Alerta fatal, ambas as partes devam fechar imediatamente a conexão segura, o browser persistiu em estabelecer a conexão segura com o gateway WAP. Somente após seis tentativas (máximo de tentativas permitido), o browser respondeu com a mensagem de Alerta crítica *connection_close_notify* para fechamento de conexão, o qual notifica o receptor (gateway) que o remetente (browser) não enviará mais mensagens usando o estado de conexão atual. Depois dessa troca de mensagens que duraram aproximadamente 30 segundos, o browser exibiu a mensagem de erro: "*Unable to establish secured connection*". O ataque injetado não foi bem-sucedido, i.e., não houve defeito. A implementação de gateway WAP *Columbitech WAP Connector* não considera o algoritmo de ciframento NULL passível de ser selecionado, o que resultou na rejeição dos *cipher suites* propostos pelo browser na mensagem de *Client Hello*.



* A mensagem foi enviadas seis vezes.

Figura 6.10: Diagrama de mensagens do caso de teste 1.

Nos **casos de teste 2, 3, 4, 5, 7, 8, 9 e 10**, onde a troca de mensagens é mostrada na Figura 6.11, os *cipher suites* foram alterados na mensagem de *Client Hello* pela instância do script, o gateway WAP respondeu com a mensagem de *Server Hello* concordando com os respectivos *cipher suites* alterados e determinando o uso do *cipher suite* alterado durante a sessão WAP, e respondeu também com as mensagens de *Server Key Exchange* e *Server Hello Done*. O browser respondeu com a mensagem de Alerta crítica *illegal_parameter* de tratamento de erros do WTLS, o qual indica que um campo no *handshake* estava fora do limite permitido ou inconsistente com outros campos. Em outros termos, o browser constatou que o *cipher suite* alterado, o qual foi enviado de volta na mensagem de *Server Hello*, estava inconsistente com os valores de *cipher suites* padrões enviados por ele na mensagem de *Client Hello*. O gateway WAP respondeu a mensagem de Alerta crítica *connection_close_notify* para fechamento de conexão, o qual notifica o receptor (browser) que o remetente (gateway) não enviará mais mensagens usando o estado de conexão atual. Depois dessa troca de mensagens que durou aproximadamente 2 segundos, o browser exibiu a mensagem de erro: "*Unable to establish secured connection*". O ataque injetado nesse caso de tese também não foi bem-sucedido. Apesar do gateway WAP ter selecionado e confirmado o *cipher suite* alterado, o browser rejeitou esse *cipher suite*, i.e., o browser possui restrições de segurança, aceitando apenas os *cipher suites* padrões propostos por ele para uso na sessão WAP.

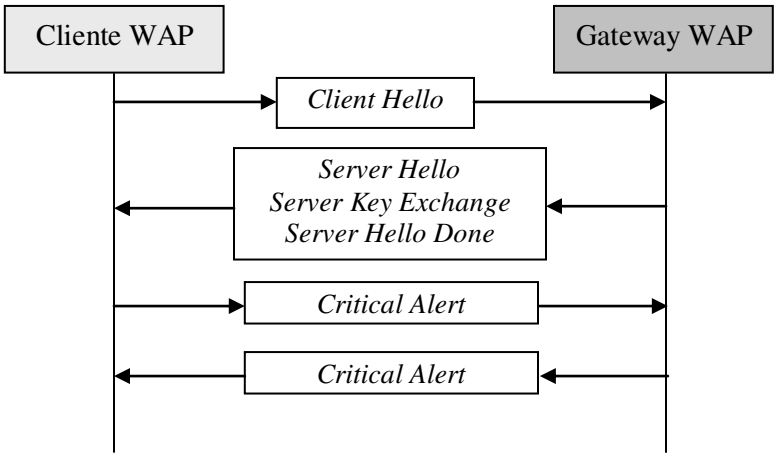


Figura 6.11: Diagrama de mensagens dos casos de teste 2, 3, 4, 5, 7, 8, 9 e 10.

Nos **casos de teste 11 e 12** foram usados os valores de *cipher suites* da lista padrão proposta pelo browser, onde para caso de teste foi usada uma entrada da lista. Esses casos de teste foram executados com o objetivo de assegurar que a restrição de segurança do browser NMB descrita nos casos de testes anteriores iria se repetir. A troca de mensagens desses casos de teste, explicada abaixo, é mostrada na Figura 6.12. Os *cipher suites* foram alterados na mensagem de *Client Hello* pela instância do script, o gateway WAP respondeu com a mensagem de *Server Hello* concordando com os respectivos valores de *cipher suites* alterados; respondeu também com as mensagens de *Server Key Exchange* e *Server Hello Done*. O browser enviou a mensagem de *Client Key Exchange*, a mensagem de *Change Cipher Spec* confirmando o uso do *cipher suite* negociado e da chave de sessão (*key session*) que foi calculada, e também enviou a mensagem de *Finished*. O gateway WAP respondeu com a mensagem de Alerta fatal *decrypt_error* de tratamento de erros do WTLS, o qual indica uma operação criptográfica do *handshake* falhou, sendo incapaz de validar a mensagem de *Finished*. O browser respondeu com uma mensagem de Alerta cifrada (não foi possível identificar a severidade e a descrição do alerta), pois seu estado de conexão atual já havia sido alterado para estado de ciframento. O gateway WAP respondeu com a mensagem de Alerta crítica *no_connection* de tratamento de erros do WTLS, o qual indica que uma mensagem foi recebida pelo remetente (gateway) enquanto não havia conexão segura estabelecida. Depois dessa troca de mensagens que durou aproximadamente 2 segundos, o browser exibiu a mensagem de erro: "*Unable to establish secured connection*".

O browser NMB não rejeitou o *cipher suite* alterado, o qual foi enviado na mensagem de *Server Hello*, como nos casos de teste anteriores, pois é um *cipher suite* padrão que foi proposto pelo browser na mensagem de *Client Hello*. Porém o ataque injetado nesse caso de teste também não foi bem-sucedido, pois o gateway WAP rejeitou a mensagem de *Finished* enviada pelo browser, que corresponde a um registro cifrado com os algoritmos mais fracos selecionados.

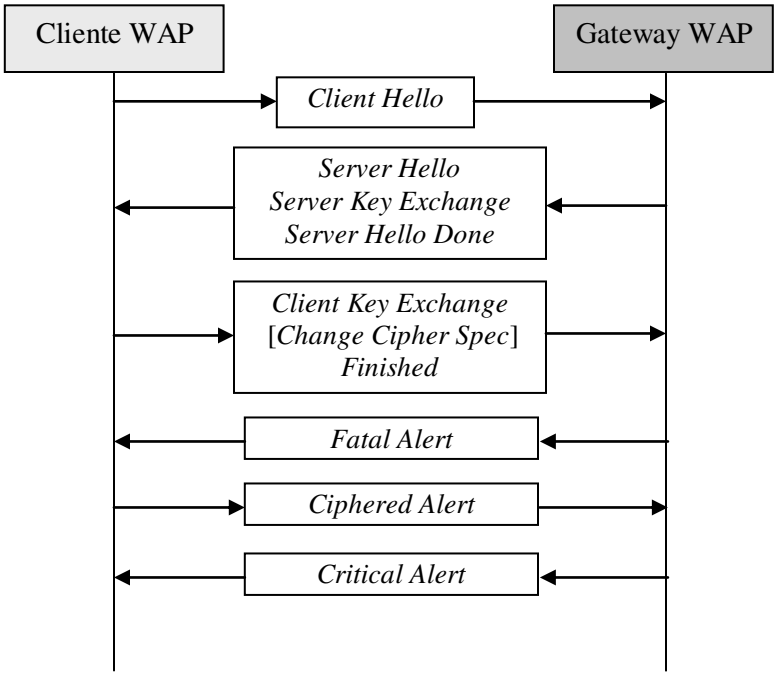


Figura 6.12: Diagrama de mensagens dos casos de teste 11 e 12.

Esse comportamento dos casos de teste 11 e 12 ocorreram porque foi adicionada uma correção para essa vulnerabilidade na versão 3.0 do SSL. O SSL v.3.0 corrige essa vulnerabilidade autenticando todas as mensagens do protocolo de *Handshake* com o *master_secret* calculado, dessa forma identificando qualquer adulteração realizada pelo atacante em alguma mensagem durante o *handshake* e terminando a sessão segura se necessário. Podemos concluir que a versão do protocolo WTLS implementada no gateway WAP, o qual é baseada nos protocolos SSL/TLS, também possui essa correção na implementação testada, o que fez com que o ataque falhasse para esses casos de teste.

Todos os ataques injetados contra o gateway WAP nos casos de teste executados

não resultaram em violação de segurança, i.e, a propriedade *integridade* não foi violada. Caso algum dos casos de teste tivesse obtido sucesso, teríamos uma eficiência de 100% para explorar a vulnerabilidade de segurança, i.e, apenas o sucesso de um caso de teste é suficiente para comprometer o sistema. Diferentemente das técnicas baseadas em Fuzz e Teste de Sintaxe que precisam de uma grande proporção de experimentos de injeção de ataques para alcançar algum efeito perceptível, i.e., ativar uma vulnerabilidade de segurança, como discutido na seção 3.3.

Também o SAE exibido na Figura 6.9 possui uma carga de falhas mínima ao invés de gerar uma enorme carga de falhas para detectar vulnerabilidades, como é feito nas abordagens baseadas em Fuzz e Teste de Sintaxe.

O objetivo do experimento, que era validar a abordagem proposta através da realização da injeção de ataques reportados do WTLS, foi alcançado, pois através da injeção dos ataques e das análises dos resultados foi possível verificar que a correção para esse tipo de vulnerabilidade reportada está presente na implementação do WTLS usada como alvo.

6.5 Considerações finais

Neste capítulo foram apresentados vários experimentos demonstrando o uso prático da abordagem detalhada ao longo deste trabalho. Além de mostrar os resultados dos experimentos, procurou-se detalhar a metodologia adotada assim como as ferramentas e critérios utilizados. Outra preocupação foi manter um paralelo entre os resultados alcançados usando a abordagem proposta e os resultados dos vários outros trabalhos e abordagens atuais que possuem o mesmo objetivo: realizar Testes de Segurança, onde a maior preocupação é encontrar o maior número possível de vulnerabilidades no sistema alvo de forma a obter maior eficiência nesse tipo de teste. É notado que a maioria das abordagens atuais se baseia na técnica de Fuzz e/ou Teste de Sintaxe, razão pela qual, se decidiu comparar os resultados dessas abordagens com os resultados do trabalho proposto.

Relatórios técnicos recentes sobre Testes de Segurança / Robustez [40] apontam uma tendência em se combinar diferentes técnicas, como usar abordagens baseadas na carga de trabalho junto com abordagens baseadas na carga de falhas para aumentar a

controlabilidade dos casos de testes e reduzir o tamanho da campanha de testes. Por exemplo, já é demonstrado e amplamente reconhecido que técnicas de modelagem e técnicas de experimentação se completam conceitualmente [32]. Nesse trabalho foi combinada a técnica de Modelagem de Ataques juntamente com a técnica de IF para melhorar a eficiência desse tipo de teste (reduzir o tamanho da carga de falhas gerada sem reduzir o número de falhas de segurança encontradas). Foi encontrado apenas um trabalho que emprega esse tipo de combinação de técnicas, o qual usa diagramas UML para modelar as vulnerabilidades, e a partir desse modelo realizar Testes de Segurança [43], o que representa ainda um esforço tímido nessa direção, já que ferramentas de modelagem, principalmente baseadas em UML, são bastante populares na área da Engenharia de Software, sendo amplamente usadas.

Logo o foco desse trabalho foi obter "melhores resultados práticos" em relação às técnicas atuais. Para isso foi empregada a técnica de Modelagem de Ataques para facilitar a construção e a manutenção dos cenários de ataques usados por engenheiros e testadores, de forma que os ataques selecionados pudessem ser modelados de forma simples e fácil, e os scripts de ataque pudessem ser gerados de maneira automatizada para serem executados usando ferramentas de injeção de falhas disponíveis.

Capítulo 7

Conclusões

7.1 Resultados e Contribuições

Neste trabalho nós propusemos uma abordagem caixa preta para a geração e injeção de cenários de ataque para realização de Testes de Segurança efetivos. Os cenários de ataque são gerados, de forma automatizada, a partir de uma árvore de ataques, que modela diferentes vulnerabilidades reais do sistema alvo (ex.: protocolo). Essas vulnerabilidades são obtidas de bancos de vulnerabilidades e outras fontes de segurança. O método mostra como selecionar Cenários de Ataque a partir da Árvore de Ataques, que são viáveis para uma dada arquitetura de testes, a qual determina a capacidade do atacante. Os Cenários de Ataque são então refinados em Scripts de Ataque Genéricos (SAGs), os quais podem ser reusados para outras implementações similares, assim como o Cenários de Ataque. Os SAGs podem ser convertidos automaticamente, usando um ***conversor***, em Scripts específicos para um injetor de falhas, que foram escolhidos neste estudo para a execução dos ataques. A abordagem permite o uso de diferentes injetores de falhas, favorecendo a portabilidade do SAG para esses injetores, graças ao uso de um ***conversor***.

Os Scripts de Ataque Executáveis (SAEs) foram injetados na implementação alvo, usando uma carga de falhas mínima para a emulação desses ataques, durante um período de experimentos considerado baixo. A abordagem proposta foi ilustrada usando, como sistema alvo, o protocolo de segurança WTLS que é utilizado em dispositivos móveis com capacidade WAP. Vários experimentos foram realizados, onde foram injetadas com sucesso, no protocolo alvo, diferentes classes de ataques. Os resultados dos experimentos mostraram que a abordagem possui boa eficiência. Também foi possível a criação de variações de alguns desses ataques já reportados; apesar de terem sido poucos experimentos, de toda forma conseguimos mostrar que a abordagem permite facilmente criar novos ataques ao WTLS. A criação de novos ataques contribui para a elaboração de medidas de prevenção de ataques e de intrusão.

Através da análise dos resultados foi verificado que a eficiência e a duração dos experimentos se mostraram satisfatórias quando comparadas com outras técnicas de Testes de Segurança baseadas em Fuzz e Teste de Sintaxe.

Dentre as contribuições dessa dissertação, podemos citar:

- Até onde nós sabemos, esta é a primeira vez que uma árvore de ataques é usada para derivar cenários de ataque de uma maneira sistemática para propósitos de injeção de ataque.
- A abordagem injeta ataques conhecidos e bem-sucedidos, e também podem ser criadas variações desses ataques (novos ataques), de forma localizada, variando os parâmetros do ataque original, que podem ser executados usando o atual injetor de falhas.
- A abordagem é capaz de executar diferentes classes de ataques de acordo com a capacidade do atacante definida, como executar as ações **remover** e **personificar**, o que não é possível usando abordagens baseadas em Fuzz e Teste de Sintaxe, que se limitam à ação **corromper**.
- Foi também proposta uma notação baseada em UML para representar os Cenários obtidos da Árvore de Ataques e gerar Scripts independentes de plataforma. Com isso melhora-se a reusabilidade desses scripts para implementações semelhantes, que são depois implementados usando linguagens específicas de injetores.
- Foi proposto o uso de um ***conversor*** que traduz as operações de SAGs para instruções específicas de injetores de falhas, facilitando sua portabilidade para diferentes injetores de falhas e tornando a abordagem completamente independente de um injetor de falhas específico.

7.2 Trabalhos Futuros

A seguir listamos algumas sugestões para o desenvolvimento de trabalhos futuros.

- Usar uma linguagem de intrusão / ataque existente, como citado na seção 4.7, para representar e refinar os Cenários de Ataque obtidos da Árvore de Ataques em

SAGs, e assim melhorar o grau de flexibilidade e portabilidade da abordagem.

- Usar outros protocolos de redes sem-fio (ex.: padrão IEEE 802.11) como estudo de caso para a realização de Testes de Segurança efetivos.
- Usar as informações coletadas durante o monitoramento do tráfego de rede durante a injeção dos ataques para uma análise mais refinada do comportamento do protocolo alvo.
- Usar atributos específicos, obtidos de bases de vulnerabilidade (como explicado na seção 4.7), para os Cenários da Árvore de Ataques, de forma a planejar e priorizar o número de injeções de ataque nos experimentos de acordo com essas métricas de ataque, que podem ser: a freqüência de um ataque ao sistema de alvo durante um determinado período de tempo ou se uma correção já foi fornecida para esse ataque.
- Usar um processo alternativo de modelagem, refinamento e transformação dos Cenários de Ataque para SAEs do injetor de falhas, o qual adere à abordagem MDA (*Model Driven Architecture*) [22] e usa ferramentas MDA de transformação de modelos UML, como citado na seção 4.7.
- Sistematizar a criação de novos ataques a partir de ataques conhecidos já reportados, i.e, definir uma abordagem para obter de forma sistemática variações de ataques bem-sucedidos que caracterizem novos ataques.

Apêndice A

Firmament

A.1 Instruções de Firmament

Um *Faultlet*, explicado na seção 5.2, pode ser especificado usando um conjunto de instruções que são divididas em classes. As instruções usadas nesse trabalho são divididas nas seguintes classes:

1. Instruções de entrada e saída:
   - **READB Ry Rx**: Lê no registrador **Rx** o conteúdo do byte (8 bits) apontado no pacote pelo valor (offset) de **Ry**.
   - **READS Ry Rx**: Lê no registrador **Rx** o conteúdo de 2 bytes (16 bits) apontados no pacote pelo valor (offset) de **Ry**.
   - **READW Ry Rx**: Lê no registrador **Rx** o conteúdo de 4 bytes (32 bits) apontados no pacote pelo valor (offset) de **Ry**.
   - **WRTEB Ry Rx**: Escreve no byte do pacote apontado pelo valor (offset) de **Ry** o conteúdo do byte (8 bits) menos significativo do registrador **Rx**.
   - **WRTES Ry Rx**: Escreve em 2 bytes do pacote apontados pelo valor (offset) de **Ry** o conteúdo dos 2 bytes (16 bits) menos significativos do registrador **Rx**.
   - **WRTEW Ry Rx**: Escreve em 4 bytes do pacote apontados pelo valor (offset) de **Ry** o conteúdo dos 4 bytes (32 bits) do registrador **Rx**.
   - **SET y Rx**: Escreve no registrador **Rx** o valor absoluto de **y** (32 bits).

2. Instruções aritméticas e lógicas:
   - **SUB Ry Rx**: O registrador **Rx** recebe o valor da subtração aritmética dos registradores (**Rx** – **Ry**).
   - **AND Ry Rx**: O registrador **Rx** recebe o valor da operação lógica bit a bit AND

dos registradores (**Rx** & **Ry**).

3. Instruções de ação no pacote:
   - **ACP**: O pacote é aceito no fluxo e o processamento do *faultlet* é concluído.
   - **DRP**: O pacote é removido do fluxo e o processamento do *faultlet* é concluído.

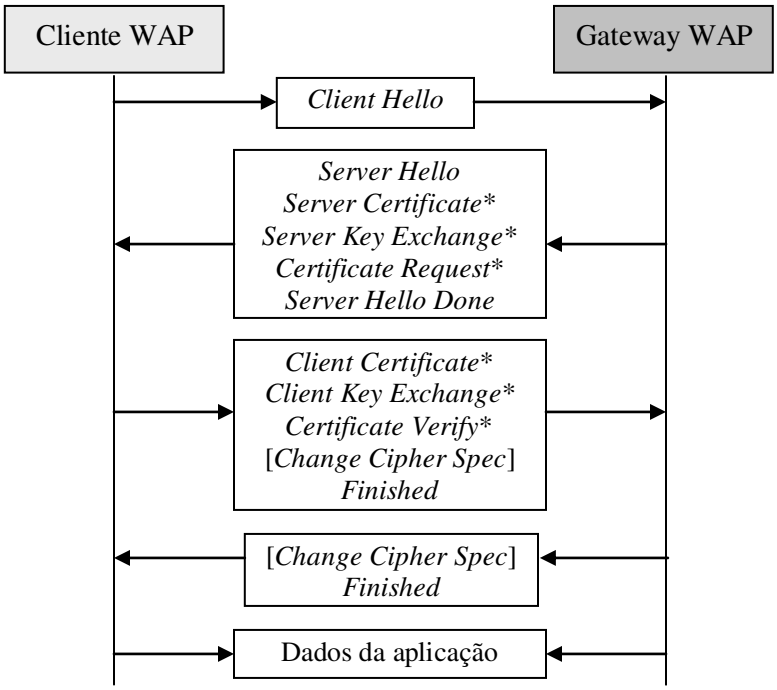
4. Instruções de salto:
   - **JMP x**: A execução no fluxo do *faultlet* é desviada para a instrução localizada na posição **x** do *faultlet*.
   - **JMPZ Ry x**: A execução no fluxo do *faultlet* é desviada para a instrução localizada na posição **x** do *faultlet* se o conteúdo do registrador **Ry** é zero.

Apêndice B

WTLS

B.1 Handshake do WTLS

O Protocolo de *Handshake*, citado na seção 5.1, é responsável por estabelecer uma conexão segura e negociar os parâmetros de segurança durante o *handshake*. O diagrama de mensagens da fase de *handshake* é mostrado na Figura B.1.



* Indica mensagens opcionais ou dependentes de situação, as quais nem sempre são enviadas.

[] Indica mensagens do Protocolo de *Change Cipher Spec* (citado na seção 5.1).

Figura B.1: Diagrama de mensagens do *handshake*.

O *handshake* começa com uma mensagem *Hello*. O cliente WAP envia uma mensagem *Client Hello* para o gateway WAP, anunciando que ele deseja estabelecer uma

conexão segura. O gateway WAP responde com uma mensagem *Server Hello*. Durante esses "cumprimentos", as duas partes estão negociando os parâmetros de segurança da sessão, como por exemplo, o *cipher suite* (algoritmo de cifras de bloco e algoritmo de MAC), e trocando valores aleatórios para o cálculo do segredo compartilhado (*master secret*).

Depois que o cliente envia a mensagem *Client Hello*, ele recebe mensagens do gateway WAP até receber a mensagem *Server Hello Done*, que indica o fim da fase de *Hello*. O gateway WAP envia a mensagem *Server Certificate* se autenticação é requerida no lado do gateway WAP. Além disso, o gateway WAP pode requerer que o cliente se autentique enviando a mensagem *Certificate Request*. A mensagem *Server Key Exchange* pode ser usada para fornecer ao cliente a chave pública do gateway WAP, assim realizando a troca-de-chaves (*key exchange*). A mensagem *Server Key Exchange* é enviada se o gateway WAP não possuir um certificado ou se o certificado for apenas para assinatura.

Depois de receber a mensagem *Server Hello Done*, o cliente continua sua parte do *handshake*. Se a mensagem *Certificate Request* foi recebida, o cliente envia a mensagem *Client Certificate* onde ele autentica-se. O cliente envia a mensagem *Client Key Exchange* se a mensagem *Server Key Exchange* foi recebida, a qual contém seu *pre-master secret*[39] cifrado com a chave pública do gateway WAP; ou envia a mensagem *Client Key Exchange* se a mensagem *Client Certificate* enviada não contém informação suficiente para a troca-de-chaves, o qual irá conter a informação que ambas as partes poderem completar a troca-de-chaves. Adicionalmente a mensagem *Change Cipher Spec* deve ser enviada, que é um meio para ambas as partes concordarem com os parâmetros da sessão negociados e começarem a usar esses parâmetros de segurança. Finalmente o cliente envia a mensagem *Finished* que contém a verificação de todos os dados trocados no *handshake* e das informações de segurança calculados, i.e., o primeiro registro cifrado com a chave de sessão (*session key*) calculada. Neste ponto o gateway WAP também deve enviar uma mensagem *Finished* onde ele também verifica as informações trocadas e calculadas.

[39] O valor do *pre-master secret* é usado para calcular o *master secret*, e este, por sua vez, é usado para calcular a chave de sessão (*session key*).

Referências Bibliográficas

[1] K. S. Edge, R. A. Raines, R. O. Baldwin, M. A. Grimaila e R. Bennington, "The Use of Attack and Protection Trees to Analyze Security for an Online Banking System", the Hawaii International Conference on System Sciences-HICSS-40, Kauai, Hawaii, January 2007, 8 pages (CD)

[2] James W. Gray III, John McLean: "Using temporal logic to specify and verify cryptographic protocols". Proc. of 8th. IEEE Computer Security Foundations Workshop (CSFW '95), March 13-15, 1995, Kenmare, County Kerry, Ireland, pp 108-117.

[3] G. Lowe. "Towards a Completeness Result for Model Checking of Security Protocols", Proceedings of the 11th Computer Security Foundations Workshop (PCSFW), IEEE Computer Society Press, 1998.

[4] Herbert H. Thompson, James A. Whittaker, Florence E. Mottay. "Software security vulnerability testing in hostile environments". ACM Symposium on Applied Computing (SAC) 2002: 260-264. Madrid, Spain.

[5] S. Dawson, F. Jahanian e T. Mitton. "Orchestra: A fault injection environment for distributed systems". Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS), pp 404-414, Sendai, Japan, June 1996.

[6] R.J. Drebes, G. Jacques-Silva, J. F. da Trindade, T. S. Weber. "A Kernel-based Communication Fault Injector for Dependability Testing of Distributed Systems". Parallel and Distributed Systems: Testing and Debugging (PADTAD-3), 2005, Haifa, Israel.

[7] Y. Hsu, G. Shu e D. Lee. "A Model-based Approach to Security Flaw Detection of

Network Protocol Implementations". In: Proceedings of the Sixteenth IEEE International Conference on Network Protocols (ICNP 2008), Orlando, Florida, USA, 2008

[8] P.C.H.Wanner, R.F.Weber. "Fault Injection Tool for Network Security Evaluation". LNCS Volume 2847/2003. Dependable Computing. Pp 127-136, Sept/2003.

[9] PROTOS - Security Testing of Protocol Implementations. Obtained in Mai/2008 at: http://www.ee.oulu.fi/research/ouspg/protos/.

[10] N. Neves, J. Antunes, M. Correia, P. Veríssimo, R.Neves. "Using Attack Injection to Discover New Vulnerabilities". In: Proc. of the International Conference on Dependable Systems and Networks (DSN), 2006, pp 457-466.

[11] W. H. Allen, C. Dou e G. A. Marin. "A Model-based Approach to the Security Testing of Network Protocol Implementations". In: Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN 2006), Tampa, FL, USA, 2006.

[12] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.C. Laprie, J.C. Lebraud, D. Long, T. McCutcheon, J. Muller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Verissimo, G. Trouessin, R.J. Stroud, M. Waidner e I. Welch, "Malicious- and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases", LAAS report no. 00280, MAFTIA, Project IST-1999-11583, p. 113, Aug. 2000.

[13] D.Welch, S. Lathrop. "Wireless security threat taxonomy". In Proc. Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society, Jun 18-20, 2003, pp76-83.

[14] J. Steffan and M. Schumacher. "Collaborative Attack Modeling". In SAC '02: Proceedings of the 2002 ACM Symposium on Applied Computing, pp. 253–259,

Madrid, Spain, Mar. 2002. ACM Press.

[15] McDermott, J. "Attack Net Penetration Testing". In The 2000 New Security Paradigms Workshop (Ballycotton, County Cork, Ireland, Sept. 2000), ACM SIGSAC, ACM Press, pp. 15-22.

[16] Hussein, M., Zulkernine, M.: "UMLintr: a UML profile for specifying intrusions". In Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Tucson, AZ, USA, pp. 279–288 (2006).

[17] B. Schneier, B., "Attack Trees: Modeling Security Threats", Dr. Dobb's Journal, December 1999.

[18] A.P.Moore, R.J.Ellison, R.C.Linger. "Attack Modeling for Information Security and Survivability". Technical Note CMU/SEI-2001-TN-001. March 2001.

[19] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 166-182, February 1990.

[20] F. Cristian, H. Aghili, H. R. Strong e D. Dolev. "Atomic broadcast: From simple message diffusion to Byzantine agreement", in Proc. 15th Fault-Tolerant Comput. Symp., Ann Arbor, MI, 1985, pp. 200-205.

[21] A. Cavalli, E. Martins, A. Morais. "Use of invariant properties to evaluate the results of fault-injection-based robustness testing of protocol implementations". Proc. of 4th Workshop on Advances in Model Based Testing (AMOST'08), Lillehammer, Norway, 9-11/Abr/2008.

[22] OMG, MDA Guide Version 1.0.1. Obtained in November/2008 at:

http://www.omg.org/mda/specs.htm#MDAspecSupport.

[23] N. Paton, O. Díaz, M.H. Williams, J. Campin, A. Dinn e A. Jaime. “Dimensions of Active Behaviour”. In N. Paton et M. Williams, editor, Proceedings of 1st Int. Workshop on Rules in Database Systems (RIDS’93), Edinburgh - Scotland, September 1993. Springer Verlag.

[24] WAP Forum, WAP Architecture, Version 12-July-2001. Obtained in May/2008 at: http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html.

[25] WAP Forum, Wireless Transport Layer Security, Version 06-Apr-2001. Obtained in May/2008 at: http://www.openmobilealliance.org/tech/affiliates/wap/wapindex. html.

[26] M.J. Saarinen, M.J., “Attacks Against the WAP WTLS Protocol”, May 2000. Obtained in May/2008 at: http://www.cc.jyu.fi/~mjos/.

[27] D. Wagner and B. Schneier. “Analysis of the SSL 3.0 protocol”. In Proceedings of the 2nd USENIX Workshop on Electronic Commerce, pp. 29–40, Berkeley, Nov. 18–21 1996.

[28] Amenaza Technologies Limited, SecureITree. Obtained in May/2008 at: http://www.amenaza.com/.

[29] R. K. Nicholls e P. C. Lekkas. “Wireless Security Models, Threats, and Solutions”, McGraw-Hill, 2002.

[30] M.C.Hsueh, T.T.Tsai, R.S.Iyer. “Fault Injection Techniques and Tools”, IEEE Computer, April/1997, pp75-82

[31] K. Echtle and M. Leu, "The EFA fault injector for fault-tolerant distributed system testing," in IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems,

Amherst, MA, USA, 1992, pp. 28-35.

[32] A. v. Moorsel, A. Bondavalli, G. Pinter, H. Madeira, I. Majzik, J. Durães, J. Karlsson, L. Falai, L. Strigini, M. Vieira, M. Vadursi, P. Lollini e R. Esposito, "State of the Art", AMBER - Assessing, Measuring, and Benchmarking Resilience - report. Obtained in Jan/2009 at: http://www.amber-project.eu/.

[33] H. Madeira, D. Costa e M. Vieira, "On the Emulation of Software Faults by Software Fault Injection". In Proceedings of the International Conference on Dependable Systems and Networks, New York, NY, USA, June 2000. IEEE, pp 417—426

[34] IEEE Standard Glossary of Software Engineering Terminology, URL: http://standards.ieee.org/

[35] G. Vigna, W. Robertson , D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits", In Proceedings of the 11th ACM conference on Computer and communications security, October 25-29, 2004, Washington DC, USA

[36] Philip Koopman, John DeVale: "The Exception Handling Effectiveness of POSIX Operating Systems", IEEE Transactions on Software Engineering, Vol. 26, No. 9, Sept. 2000.

[37] B. Miller et al.: "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", Computer Sciences Technical Report #1268, University of Wisconsin-Madison, April 1995.

[38] D. Aitel, "The Advantages of Block-Based Protocol Analysis for Security Testing". Immunity Inc., February 2002. Disponível em: http://www.immunitysec.com/ downloads/advantages_of_block_based_analysis.html. Último acesso em 13/01/2008.

[39] S. Xiao, L. Deng, S. Li e X. Wang, "Integrated TCP/IP Protocol Software Testing for

Vulnerability Detection", 2003. International Conference on Computer Networks and Mobile Computing, Shanghai, October 2003.

[40] ReSIST NoE. Resilience-Building Technologies: State of Knowledge, Deliverable D12, URL: http://www.resist-noe.org/outcomes/outcomes.html

[41] D. Dolev and A. Yao. "On the security of public-key protocols". IEEE Transactions on Information Theory, 2(29): 198-208, 1983.

[42] Jurjens, J., "Towards Development of Secure Systems using UML", In H. Hubmann, editor, Fundamental Approaches to Software Engineering (FASE/ETAPS, International conference), Lecture Notes in Computer Scinece, Springer, Genova, Italy, 2001.

[43] L. Wang, E. Wong e D. Xu. "A Threat Model Driven Approach for Security Testing". In: Proceedings of the Third International Workshop on Software Engineering for Secure Systems, Minneapolis, MN, USA, May 20 – 26, 2007.

[44] R. Zhang, and K. Chen, "Improvements on the WTLS protocol to avoid denial of service attacks", Computers and Security, Elsevier Science, Vol. 24, No 1, pp. 76-82, 2005.

[45] Diana Berbecaru, Antonio Lioy. "On the Robustness of Applications Based on the SSL and TLS Security Protocols". EuroPKI 2007, 248-264.

[46] F. Bachmann *et al*. "Volume II:. Technical Concepts of Component-Based Software Engineering - 2nd Edition". Obtained in Fev/2009 at: www.sei.cmu.edu/pub/ documents/00.reports/pdf/00tr008.pdf.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Addison Wesley, 1995.

[48] M. Raihan, M. Zulkernine, "AsmLSec: An Extension of Abstract State Machine Language for Attack Scenario Specification". In The Second International Conference on Availability, Reliability and Security (ARES'07), 2007, 775-782.

[49] G. Liu and A. Mok. “Implementation of JEM - A Java Composite Event Package”. In Proc. of Real-Time Technology and Applications Symposium, 1999.

[50] J. Gerchman, C. Menegotto e T. Weber. “Geração de cargas de falha para campanhas de teste com injeção de falhas a partir de modelos UML de teste”. In: XXII Simpósio Brasileiro de Engenharia de Software (SBES), 2008, Campinas.