FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
TRABALHO DE CONCLUSÃO DE CURSO

RODRIGO FRAXINO DE ARAUJO

IMPLEMENTAÇÃO DE MECANISMOS DE SEGURANÇA PARA CÓDIGO MÓVEL

MARÍLIA
2005

RODRIGO FRAXINO DE ARAUJO

IMPLEMENTAÇÃO DE MECANISMOS DE SEGURANÇA PARA CÓDIGO MÓVEL

Monografia apresentada como Trabalho de Conclusão de Curso do Centro Universitário Eurípides de Marília, mantido pela Fundação Eurípides Soares da Rocha, para a obtenção do título de Bacharel em Ciência da Computação

Orientador:
Prof. Dr. Márcio Eduardo Delamaro

MARÍLIA
2005

ARAUJO, Rodrigo Fraxino
Implementação de mecanismos de segurança para código móvel / Rodrigo Fraxino de Araújo; orientador: Prof. Dr. Marcio Eduardo Delamaro. Marília, SP: [s.n.], 2005.

56 f.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. Engenharia de Software

CDD: 005.1

ARAUJO, Rodrigo Fraxino. **Implementação de mecanismos de segurança para código móvel**. 2005. 56 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

As tecnologias de código móvel são uma abordagem para a implementação de sistemas distribuídos. Nesse contexto é possível citar o ambiente μCode, que proporciona facilidade para o trabalho com mobilidade de código por meio da linguagem Java. Entretanto, este ambiente não possui mecanismos que proporcionem segurança aos servidores contra a execução de aplicações móveis mal intencionadas, como também não proporciona segurança aos agentes móveis contra servidores maliciosos. A partir destas limitações, um esquema de proteção foi desenvolvido por meio do uso de tecnologias de criptografia e assinatura digital.

Palavras-chave: mobilidade de código, segurança, criptografia.

ARAUJO, Rodrigo Fraxino. **Implementação de mecanismos de segurança para código móvel**. 2005. 56 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

Mobile code technologies are an approach to the implementation of distributed systems. The μCode environment appears in this context providing facility to manipulate mobile code in Java language. Nevertheless, this environment does not have mechanisms responsible for providing security to servers against the execution of malicious mobile applications, as well as does not provide security to mobile agents against malicious servers. Focusing improving these limitations, a protection scheme was developed through the use of cryptography and digital signatures technologies.

Key-words: code mobility, security, criptography

LISTA DE FIGURAS

Figura 1 – Geração e verificação de assinatura 21

Figura 2 – Cifragem do objeto 25

Figura 3 - cryptAgent 26

Figura 4 – deCryptAgent 27

Figura 5 – crypt 28

Figura 6 - decrypt 29

Figura 7 – rsa.public.keys 29

Figura 8 – signReadOnly 32

Figura 9 – verifyReadOnly 33

Figura 10 – appendObject 34

Figura 11 – getObject 34

Figura 12 – Interface de *MobileChat* 38

Figura 13 – Servidor MobileChat 40

Figura 14 – Cliente MobileChat 40

Figura 15 – Servidor MobileChat com chave privada inválida 41

Figura 16 – Servidor MobileChat com chave privada não encontrada 42

Figura 17 – Adicionando arquivos somente leitura 43

Figura 18 – Verificando arquivos somente-leitura 43

Figura 19 – Utilização de appendObject 44

Figura 20 – Utilizando getObject 44

SUMÁRIO

INTRODUÇÃO 9

CAPÍTULO 1 - CONTEXTO 10

1.1 Código Móvel 10

1.2 Segurança 11

1.2.1 Agentes maliciosos 11

1.2.2 Servidores Maliciosos 12

1.3 Ataques contra agentes móveis 13

1.4 Esquemas de Proteção 15

1.5 Considerações Finais 18

CAPÍTULO 2 - SEGURANÇA NO AMBIENTE μCODE 19

2.1 Ambiente μCode 19

2.2 Proteção do servidor 20

2.3 Proteção do agente 21

CAPÍTULO 3 - IMPLEMENTAÇÃO DO ESQUEMA DE SEGURANÇA 24

3.1 Mecanismo "*TargetedState*" 24

3.2 A abstração TargetedMuAgent 29

3.3 Mecanismo ReadOnly 31

3.4 AppendOnly 33

3.5 Esquema de proteção *versus* formas de ataques 35

CAPÍTULO 4 - UTILIZANDO OS MECANISMOS DE SEGURANÇA 37

4.1 TargetedState em Mobile Chat 37

4.1.1 Execução do MobileChat 39

4.1.2 MobileChat com erro na decifragem 41

4.2 ReadOnly em FileSearcher 42

4.3 AppendOnly em CarAgent 43
4.4 Considerações Finais 44
CAPÍTULO 5 - CONCLUSÕES 46
REFERÊNCIAS BIBLIOGRÁFICAS 48
APÊNDICE A – SecureAgent 50

INTRODUÇÃO

Os agentes móveis se tornaram uma alternativa para a implementação de sistemas distribuídos. Nesse contexto, o aspecto de segurança deve ser levado em consideração para que aplicações baseada em agentes móveis possam ser utilizados em aplicações reais. O ambiente μCode fornece uma plataforma simples e flexível para a implementação de agentes móveis Java mas, originalmente, não oferece qualquer suporte a operações seguras.

Em (ARAUJO, 2005) foi implementado um mecanismo baseado em assinatura digital e permissões que permitem o reconhecimento da autoria de um agente, com o objetivo de proteger um determinado *host* da ação de agentes maliciosos. Posteriormente abordou-se o problema de proteger a informação contida nos agentes contra a ação de servidores maliciosos. Foram estudadas técnicas nessa categoria e identificadas três delas (TargetedState, AppendOnly e ReadOnly) que pudessem ser implementadas no ambiente μCode.

O trabalho está dividido nos seguintes capítulos:

- o Capítulo 1 apresenta um levantamento sobre código móvel e a segurança existente;
- o Capítulo 2 apresenta a descrição do ambiente μCode juntamente com os mecanismos de segurança implementados;
- o Capítulo 3 apresenta os detalhes referentes aos mecanismos de segurança desenvolvidos, destacando os aspectos de implementação;
- o Capítulo 4 mostra como os mecanismos foram utilizados em aplicações móveis que utilizam o ambiente μCode; e
- o Capítulo 5 apresenta as contribuições deste projeto e as considerações finais.

CAPÍTULO 1 - CONTEXTO

Nesta seção são apresentados alguns conceitos básicos sobre o trabalho realizado, procurando mostrar o contexto em que ele se insere. Em particular é discutido o que são agentes móveis, e como podem representar ameaças à segurança de uma aplicação através do uso de agentes maliciosos ou de servidores maliciosos.

1.1 Código Móvel

Sistemas distribuídos de larga escala estão se tornando de suma importância em função da evolução da tecnologia e do interesse proveniente do mercado. É possível citar como parte desta evolução o surgimento de sistemas de código móvel, uma abordagem bem flexível para ambientes de rede (CARZANIGA et al, 1997; FUGETTA et al, 1998).

Mobilidade de código pode ser informalmente definida como a capacidade de mudar dinamicamente as ligações entre fragmentos de código e a localidade de sua execução. A habilidade de relocar código é um conceito muito poderoso que originou um vasto leque de desenvolvimentos. Todavia, uma metodologia terminológica para sua estrutura, uma padronização, ainda está faltando, e não há um consenso em relação a esta área. É importante ressaltar também que código móvel não é um conceito novo. Em um passado recente, por exemplo, vários mecanismos foram implementados para mover código através de nós de uma rede, como por exemplo, a submissão remota de arquivos em lote (FUGETTA et al, 1998).

Existem diversas formas de um código móvel ter sua execução realizada. Um dos paradigmas mais utilizados é o de agente móvel. Um agente móvel pode ser caracterizado como um programa que age de forma autônoma, em busca de um objetivo programado anteriormente. Para isso ele pode utilizar interações com outros agentes ou ambientes móveis, se necessário (JANSEN, 2000).

1.2 Segurança

Com a crescente popularização, demanda e sofisticação dos sistemas móveis, uma preocupação maior relacionada à vulnerabilidade de segurança tornou-se necessária. Não só para que os usuários se sintam seguros com a execução de aplicações móveis em suas máquinas, mas também para que tenham certeza de que suas informações estarão trafegando pela rede de forma segura.

Com isso, é possível perceber que podem existir dois tipos de vulnerabilidades em um sistema móvel. A primeira delas é a falta de proteção que o servidor possui ao executar uma aplicação móvel remota. A outra seria a falta de proteção que um agente móvel possui ao trafegar pela rede. Estes problemas serão discutidos a seguir.

1.2.1 Agentes maliciosos

Quando se tem um agente móvel que migra entre diversos servidores, corre-se o risco de que ele execute uma operação indesejável em um determinado servidor. Uma aplicação móvel pode possuir permissões para apagar um arquivo ou abrir uma conexão de rede, por exemplo. Se for esse o caso, ou seja, se ela não possuir restrições de execução, um agente mal intencionado pode tentar prejudicar ou levar algum tipo de vantagem da plataforma de execução.

Para que um ambiente móvel seja utilizado sem um mecanismo de segurança explícito, é necessário que haja um alto grau de confiança entre as partes que irão interagir no sistema. Como não há uma forma de se controlar a execução, o servidor de aplicações móveis pode ficar subordinado à aplicação.

Entretanto, o desejável para um sistema móvel seria que cada parte fosse autenticada e que cada usuário possuísse seus próprios níveis de permissão. Desta forma, garantiria-se que

uma aplicação móvel mal intencionada não pudesse causar danos ao servidor. Uma plataforma que não proporciona a segurança adequada tem sua aceitação dificultada.

1.2.2 Servidores Maliciosos

O outro lado desse problema é garantir a integridade das informações contidas nos agentes móveis, evitando que sejam, por exemplo, roubadas ou adulteradas por algum servidor por onde o agente passe.

Em linhas gerais, uma aplicação de código móvel segue uma série de decisões lógicas para qual é programada. Assim um *host* malicioso poderia, por exemplo, tentar espiar a estratégia deste código e com isso manipular o resultado das operações a serem realizadas em benefício próprio. Outra possível situação seria a manipulação da forma com que um determinado código móvel se comunica com os nós de uma rede de computadores.

De maneira simplificada, pode-se dizer que o código móvel corre o risco de se tornar totalmente subordinado ao *host* que realiza o ataque e desta forma pode se tornar vulnerável a modificação, deleção, execução parcial ou inserção de código.

Como exemplo das ações desempenhadas por um servidor malicioso, suponha que um agente móvel visite um determinado número de lojas em busca do menor preço de um buquê de rosas. Caso um determinado ambiente hostil consiga se passar por um dos *hosts* para qual o agente possuía a intenção de migrar, é possível que, através desta simulação, outros ataques sejam encadeados. Um *host* malicioso poderia abaixar o preço original das flores para ganhar a concorrência com outros *hosts*, ou, então, redirecionar uma possível operação de compra para um *host* qualquer (HOHL, 1998; JANSEN, 2000).

Proporcionar segurança a agentes móveis é considerado uma tarefa um pouco mais complicada se comparada à segurança de servidores móveis. Isto acontece pelo fato da aplicação depender do ambiente de execução, além de ser necessário a proteção das informações durante o trajeto do agente.

1.3 Ataques contra agentes móveis

É considerado muito difícil proteger um programa em execução a partir do ambiente responsável por esta execução. Isto porque em sistemas de código móvel, eles podem não pertencer ao mesmo proprietário. Com isso, um *host* malicioso pode atacar o programa em execução com o objetivo de conseguir algum benefício para si (JANSEN e KARYGIANNIS, 1999; SAU-KOON, 2000).

A seguir são discutidas algumas formas de ataques que podem ser realizados por um *host* malicioso contra agentes móveis (SAU-KOON, 2000; JANSEN, 2000; HOHL, 1998). Para isto, algumas exemplificações são realizadas para a obtenção de uma melhor compreensão.

a) Espionagem e manipulação de código

Para que o *host* execute sua próxima instrução, ele precisa ler o código do agente. Entretanto, com o conhecimento do código, o *host* também passa a possuir conhecimento sobre a estratégia do agente, a estrutura física do código e dos dados alocados na memória do *host*, além de ser possível a obtenção de partes dos dados do agente.

Se um *host* possui conhecimento do código e acesso a sua memória, ele possui capacidade para modificar o programa do agente. Um *host* pode modificar um código permanentemente ou simplesmente modificar o seu comportamento em um *host* específico. Nesse último caso, a vantagem seria que o próximo *host* ao qual o agente migraria não teria como perceber tal modificação.

b) Espionagem e manipulação de dados

A ameaça que um *host* pode causar ao ler um dado privado do agente é bem preocupante pelo fato de ele não deixar nenhum rastro que pode ser detectado. Isto seria um problema para aplicações que lidam com senhas ou aplicações financeiras, em cujo simples conhecimento de dados poderia causar perda de privacidade ou de dinheiro, por exemplo.

Se o *host* possui a localização física exata dos dados na memória, também se torna possível a alteração dos mesmos.

c) Espionagem e manipulação do fluxo de controle

Com o conhecimento do fluxo de controle juntamente com o conhecimento do código, um *host* malicioso pode deduzir mais informações sobre o estado do agente.

Além disso, o *host* pode tentar manipular o comportamento do agente. Por exemplo, ele pode desviar o fluxo de controle após uma sentença de *if*, modificando assim o comportamento que o agente iria ter.

d) Execução incorreta de código

Um *host* pode alterar a forma como irá executar o código do agente, podendo causar os mesmos problemas citados anteriormente. Outra forma de se beneficiar seria através do retorno de um resultado falso quando uma chamada é feita ao sistema.

e) Aplicação de máscara

Um *host* malicioso pode aplicar uma máscara para se passar por outro *host*. A intenção é a de o *host* conseguir interceptar ou copiar o agente destinado a ele.

f) Negação de execução

Um *host* pode se negar a executar um agente ou a não prover os serviços que ele necessite. Isto pode ser usado como um ataque no caso de um agente que necessite ser executado em um determinado período de tempo, por exemplo.

g) Espionagem e manipulação de interação entre agentes

Se a interação entre dois agentes não é protegida, um *host* pode espionar as operações realizadas entre os mesmos. Ele também pode manipular a interação usando a identidade de um outro agente, através da aplicação de uma máscara, por exemplo.

Como é possível se notar, sem o uso de um mecanismo de proteção, as tecnologias de código móvel podem se tornarem muito vulneráveis a ataques que um *host* possa executar.

1.4 Esquemas de Proteção

Nesta seção é realizada uma descrição das formas de proteção das tecnologias de código móvel contra ambientes computacionais maliciosos, além de um detalhamento sobre as necessidades de um esquema de proteção.

Uma defesa eficiente precisa atender a alguns requisitos de segurança. Contudo, é muito difícil um mesmo esquema conseguir suprir todas estas necessidades. A seguir são descritas algumas características desejáveis de um esquema de proteção (SAU-KOON, 2000).

a) Integridade

O esquema de proteção precisa proteger o código móvel contra modificações sem autorização, mantendo-o intacto. Caso alguma modificação na comunicação ou nas informações aconteça, é necessário que esta seja detectada.

b) Contabilidade (Accountability)

O esquema de proteção precisa possuir mecanismos que gravem as ações do *host*. Desta maneira, posteriormente, no caso de uma ação maliciosa ter sido realizada, é possível identificar e rastrear a origem da mesma.

c) Confidencialidade

O esquema de proteção precisa garantir privacidade ao código móvel. É necessário o uso de mecanismos específicos para que o código executável, os dados e os estados não sejam espionados ou modificados.

Um *host* malicioso pode tentar analisar as decisões lógicas, o fluxo de informações ou tentar acessar alguma informação confidencial. Este tipo de espionagem pode ser realizado na comunicação entre agentes ou entre um agente e uma plataforma.

d) Anonimidade

É interessante que o esquema de proteção também possa proteger a identidade do proprietário do código móvel, como também os *hosts* que foram visitados.

e) Disponibilidade

O esquema de proteção deve garantir que o código móvel seja alocado de forma a acessar os recursos providos pelo *host*. O código móvel não pode ficar preso para sempre no caso de um *host* não fornecer os recursos que ele necessita.

Procurando evitar que ataques possam ser realizados, a seguir são discutidos alguns esquemas de proteção existentes que dão suporte às características descritas acima. As três primeiras abordagens têm como objetivo detectar uma ação maliciosa, após ela ter sido realizada (JANSEN e KARYGIANNIS, 1999; JANSEN, 2000). Já as três últimas possuem como objetivo prevenir o código móvel contra uma eventual ação (HOHL, 1998; SANDER e TSCHUDIN, 1998; JANSEN, 2000).

a) Encapsulação parcial de resultados

Uma abordagem que pode ser utilizada na detecção de *hosts* maliciosos é através do encapsulamento do resultado das ações da aplicação móvel em todas as plataformas que ela visitar, para que possa ser realizada uma análise posterior. Esta verificação pode ser feita em pontos intermediários (que sejam confiáveis) do percurso, ou somente no ponto de origem. A informação que é armazenada pode incluir respostas a algum tipo de ação ou alguma transação realizada na plataforma.

b) Geração mútua de itinerário

Esta abordagem propõe que o itinerário da aplicação móvel seja gravado por uma aplicação colaboradora e vice-versa.

Conforme um agente móvel vai se movendo entre plataformas, seu caminho vai sendo armazenado por um outro agente através de um canal seguro de comunicação. Quando eles retornam à origem, é realizada uma verificação do caminho percorrido pelos agentes com o suposto percurso que ele deveria ter percorrido. Se alguma discrepância é notada, o servidor verifica que medida deve ser tomada.

Este esquema assume que apenas algumas plataformas são maliciosas. Desta forma, dividindo as operações entre dois agentes, não se correria o risco de ambos não conseguirem

retornar ao servidor de origem. Assim, torna-se possível perceber qual foi a plataforma responsável pelo ataque.

c) Traço de execução

Esta técnica é utilizada para detectar modificações sem autorização realizadas no código através da gravação do comportamento do mesmo durante sua execução em cada plataforma.

O seu funcionamento requer que um *log* seja criado com as operações realizadas pela aplicação móvel em cada uma das plataformas envolvidas. Cada um destes *logs* deve ser enviado à plataforma de origem assim que a aplicação terminar suas operações na plataforma. Este envio deve ser feito através do uso de chaves privadas e públicas, usadas para cifragem, juntamente com uma assinatura digital.

d) Computação com funções criptografadas

O objetivo desta técnica é o cálculo de primitivas criptográficas ao código, que permitam que este possa ser executado em ambientes computacionais sem segurança. Esta abordagem permite que a plataforma execute a função criptografada, mas não que ela possa decifrar a função original. Desta forma, não é possível que seja realizada uma modificação do comportamento da aplicação móvel. Para uma melhor compreensão, será utilizado o exemplo a seguir.

Alice possui um algoritmo para computar a função `f`. Bob possui um dado de entrada `x` e deseja computar `f(x)` para Alice. Entretanto, Alice não quer que Bob saiba como funciona a função `f`.

Se f pode ser cifrado em uma outra função `E(f)`, então Alice pode criar um programa `P(E(f))`, que implementa `E(f)` e mandá-lo a Bob, juntamente com seu agente. Bob executa o agente, que faz a computação `P(E(f))` em `x` e retorna o resultado à Alice, que o decifra e obtém `f(x)`.

Entretanto, esta abordagem pode ser utilizada somente com funções racionais e

polinomiais.

e) Geração de chaves ambientais

Esta abordagem descreve um esquema que permite que a aplicação móvel tome medidas pré-definidas quando uma condição ambiental é verdadeira. O objetivo é o de construir uma aplicação que ao se deparar com certa condição, gere uma chave, que seria utilizada para destravar criptograficamente o código executável. Esta técnica garante que a plataforma não possua acesso à leitura do código da aplicação móvel.

f) Caixa preta com tempo limitado

A estratégia desta técnica é a de bagunçar o código de uma forma que ninguém consiga entender completamente o seu funcionamento ou modificar seu resultado sem que seja detectado.

Entretanto, não se assume que esta proteção durará para sempre, mas somente por um período de tempo. A aplicação original é convertida através de algoritmos de ofuscação de código. Estes algoritmos convertem o programa e as partes de dados em uma aplicação "caixa preta" que faz basicamente o mesmo que a aplicação original. Contudo, esta nova aplicação móvel proporciona dificuldades para que possam ser feitas análises de programas que verificam o fluxo de dados ou que tentariam tomar vantagem de suas ações.

1.5 Considerações Finais

Este capítulo apresentou uma introdução à tecnologia de código móvel, juntamente com formas de ataques que podem ser realizadas, além de possíveis maneiras de proteção para código móvel. No próximo capítulo será introduzido o ambiente de mobilidade μCode, seu modo de funcionamento, juntamente com a definição de um esquema de proteção para o mesmo.

CAPÍTULO 2 - SEGURANÇA NO AMBIENTE μCODE

Nesta seção será apresentada uma breve introdução ao ambiente μCode, destacando-se aspectos de segurança, implementados neste projeto.

2.1 Ambiente μCode

O μCode pode ser definido como um ambiente de trabalho desenvolvido em Java cujo princípio motivador foi o de oferecer suporte à construção de aplicações que utilizem código móvel, através de uma ferramenta que oferecesse suporte aos diversos mecanismos de mobilidade de código, não se restringindo apenas a um paradigma em específico. Por essa razão, um dos principais diferenciais do ambiente em relação a outras plataformas está justamente no fato de permitir aos desenvolvedores que utilizem as abstrações providas pelo ambiente, ou que possam construir suas aplicações de acordo com suas necessidades e conveniências, criando suas próprias abstrações (PICCO, 2005).

Um dos fatores de grande influência e importância na adoção de soluções baseadas em código móvel são as questões relacionadas à segurança dessas aplicações, visto que apesar dos benefícios advindos com o uso da mobilidade de código, grande parte dos ambientes não oferecem mecanismos explícitos de segurança.

No μCode isto não é diferente, pois em função dos objetivos estabelecidos em sua concepção, não foi incluído nenhum mecanismo para verificação e controle das aplicações executadas. Caso haja necessidade de segurança no transporte de informações, seria preciso realizar a implementação desta na própria aplicação do usuário.

O núcleo do *μCode* é composto pelas classes *Group*, *ClassSpace, MuServer* e a interface *GroupHandler*. A funcionalidade do núcleo, mesmo estando disponível ao programador, é de compor operações que lidam com a realocação de classes e threads em um

nível de abstração que seja conveniente. Estas operações são disponibilizadas pela classe *MuServer*, que pode ser definida como a classe que provém suporte ao ambiente de execução.

A unidade de mobilidade do *µCode* é denominada de grupo. O grupo proporciona ao programador um recipiente que pode ser preenchido com classes e objetos, e enviados para seu destino. Este destino e todos os outros servidores µ*Code* podem ser definidos como *µServers*. O *µServer* é uma abstração do suporte em tempo de execução do *µCode*, provê os mecanismos básicos para a realocação de código e estado.

Quando uma aplicação móvel chega em um *µServer*, as classes e objetos precisam ser extraídos do grupo e usados de forma coerente, possibilitando a geração de uma nova unidade de execução. Esta tarefa é realizada pelo *GroupHandler*, classe responsável pelo manuseamento do grupo em seu destino. As classes extraídas precisam ser armazenadas em algum espaço nomeado, para evitar conflitos de nomes com outras classes. Esta capacidade é provida pela *ClassSpace*, que especifica um espaço de classe privado, associado com o respectivo grupo. No entanto, mais tarde elas podem ser colocadas em um espaço compartilhado associado ao *µServer*, onde se tornam disponíveis às outras threads (PICCO, 2005).

2.2 Proteção do servidor

Em uma primeira etapa deste trabalho foi desenvolvido um mecanismo que visou à proteção do servidor µCode contra aplicações móveis que tentassem realizar alguma atividade maliciosa. Para isso, a primeira alteração realizada no ambiente foi a inserção da capacidade de geração de uma assinatura digital, juntamente com sua respectiva validação. Desta forma foi possível garantir que as classes que estão migrando (não ubíquas) não fossem alteradas. A assinatura também é necessária para que se possa fazer a verificação da origem do código em um servidor remoto. Caso o agente assinado possua uma origem confiável, obterá permissão

para continuar a sua execução. A figura 1 esquematiza como é realizada a geração e verificação de uma assinatura digital.

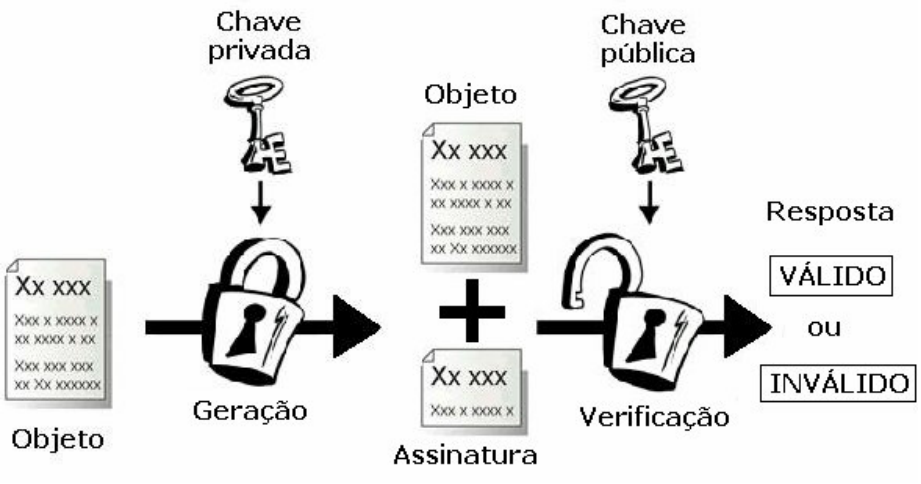

Figura 1 – Geração e verificação de assinatura

É preciso ressaltar que apenas a validação da origem do código móvel não é suficiente para garantir toda a segurança necessária a uma aplicação. É necessário que se definam níveis de segurança que sejam aplicados a cada um dos diferentes usuários. Para isto foi necessário realizar, no servidor que recebe um agente, a associação entre determinado usuário e seu conjunto de permissões (MCGRAW e FELTEN, 1999; GONG, 1999; OAKS, 2003). As permissões de cada usuário são definidas em um arquivo de políticas Java e são atribuídas no momento do carregamento das classes de uma aplicação móvel.

2.3 Proteção do agente

Pelo fato do ambiente contemplar somente a segurança de servidores contra eventuais agentes maliciosos, a segunda etapa deste trabalho visou à extensão do mecanismo de segurança implementado no ambiente μCode. Através do desenvolvimento e incorporação de novas funcionalidades, objetivou-se proteger códigos móveis contra *hosts* maliciosos, que porventura tentem obter informações de forma não autorizada em benefício próprio.

Após o estudo, a partir da literatura disponível, de diversas abordagens foram selecionados três mecanismos, denominados de *TargetedState, ReadOnly* e *AppendOnly* (KARNIK, 1998). Embora limitados em alguns aspectos, e direcionados a situações específicas, a escolha destas técnicas permitiu que mecanismos de proteção do agente fossem implementadas sem necessidade de re-estruturações radicais no ambiente μCode, possibilitando sua execução no escopo deste trabalho.

O primeiro mecanismo, *TargetedState,* tem como princípio armazenar as informações de um agente móvel de forma cifrada, permitindo que elas somente sejam acessadas por um servidor específico. Desta forma, informações confidenciais não poderão ser visualizadas por terceiros.

Este mecanismo é baseado em cifragem de chave pública, ou seja, a cifragem das informações é realizada a partir da chave pública do destinatário das mesmas. Desta forma, como o servidor de destino será o único a possuir a chave privada necessária para a decifragem dos dados, somente este poderá acessar tais informações.

Os outros dois mecanismos existentes complementam o esquema de segurança proposto. O *ReadOnly* permite que o programador armazene objetos no agente com o atributo de somente leitura, permitindo que qualquer modificação possa ser descoberta.

ReadOnly é baseado em um mecanismo de assinatura de código. Desta forma, as informações armazenadas possuem uma assinatura digital gerada a partir da chave privada do proprietário das mesmas. Com isso, qualquer servidor que possuir a chave pública correspondente poderá verificar se as informações não foram alteradas. Um exemplo de uso seria o armazenamento da lista de servidores pelos quais o agente irá passar, que não serão alterados.

O mecanismo de *AppendOnly* funciona através do armazenamento de informações que forem adquiridas na medida em que o código móvel for completando seu percurso. O agente móvel poderá obter alguma informação nos servidores pelos quais passar, e armazená-

la utilizando este mecanismo. Neste caso, também será utilizada assinatura digital para garantir a integridade das informações.

Além disto, as informações poderão ou não ficar visíveis, através do uso de cifragem através de chave pública. Com isso, garante-se que as informações não serão visualizadas por terceiros, similarmente ao que acontece no mecanismo *TargetedState*. Um exemplo de uso seria a obtenção de preços de carros por diversos servidores em busca do mais barato.

A união destes três mecanismos em um esquema de proteção disponibiliza ao usuário grandes possibilidades de proteção de sua informação. No capítulo seguinte será detalhada a classe *SecureAgent*, responsável por prover métodos que proporcionam a utilização dos mecanismos de segurança citados. Será mostrado como a implementação foi realizada, além de como fazer uso do esquema de proteção criado.

CAPÍTULO 3 - IMPLEMENTAÇÃO DO ESQUEMA DE SEGURANÇA

Para que se estabelecesse uma relação de proteção mútua entre o servidor e o agente móvel, foi necessário que a segurança do ambiente μCode fosse complementada. Para isto, a implementação de um mecanismo com a função de proteger os agentes móveis contra servidores maliciosos foi proposta abrangendo as três abordagens destacadas na seção anterior (ARAUJO, 2005).

Para esta implementação ser realizada, a principal alteração foi a criação da classe *SecureAgent*. Esta classe proporciona métodos para que o usuário possa fazer o uso dos três mecanismos de segurança implementados - *TargetedState, ReadOnly* e *AppendOnly*.

Além desta classe, com a intenção de facilitar o uso do mecanismo *TargetedState,* ainda foi criada a classe *TargetedMuAgent.* Esta classe é uma abstração de um agente móvel, que automaticamente realiza a cifragem e decifragem dos dados baseados nos métodos da classe *SecureAgent*.

Outra modificação necessária foi a implementação de um método na classe *MuServer.* Este método, denominado *decrypt*, é responsável por realizar a obtenção da chave privada do servidor e posteriormente realizar a decifragem do objeto. O motivo de estar separado dos demais métodos é o fato da chave privada ser exclusiva de cada servidor, não podendo ser obtida por nenhum agente móvel.

A descrição, juntamente com o detalhamento da implementação será visto a seguir. O código completo da classe *SecureAgent* pode ser encontrado no APÊNDICE A.

3.1 Mecanismo "*TargetedState*"

O objetivo deste mecanismo é o de permitir ao programador o poder de armazenar o objeto a ser migrado, de forma cifrada, para um destinatário específico. Este destinatário é o

servidor para o qual estas informações são endereçadas. Este mecanismo é muito útil quando se deseja transportar informações confidenciais entre dois servidores na rede. Mesmo que se consiga interceptar o objeto em migração, não será possível visualizar as informações que estão sendo trafegadas (KARNIK, 1998).

Para garantir que as informações sejam acessadas somente pelo servidor escolhido, elas serão cifradas a partir da chave pública do servidor de destino. Com isso, somente quem possuir a respectiva chave privada (no caso o servidor de destino) conseguirá decifrar tais informações. A Figura 2 mostra como isto é realizado.

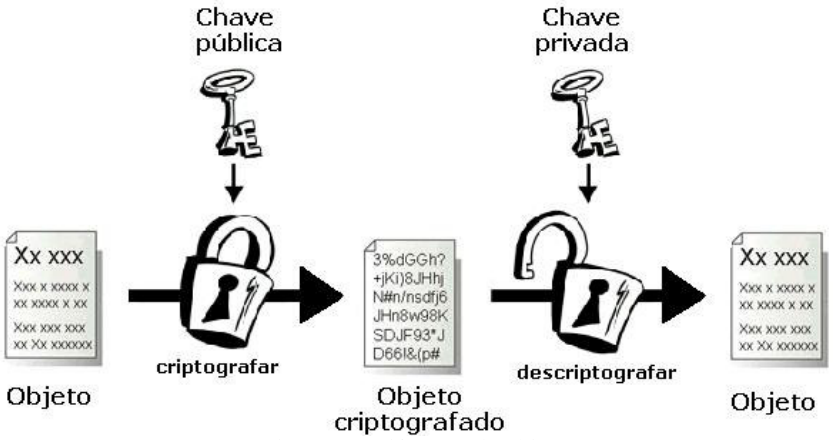

Figura 2 – Cifragem do objeto

Por ser um algoritmo confiável, reconhecido como padrão ISO 9796, e sendo provido pela Sun (SUN MICROSYSTEMS, 2005) a partir da versão 1.5 de Java, o algoritmo RSA (RSA SECURITY, 2005) foi escolhido para realizar a cifragem das informações. O tamanho da chave escolhido foi o de 1024 bits. Com uma chave grande, um ataque por força bruta é considerado extremamente custoso e demorado. Desta forma, pode-se garantir a segurança das informações.

Para que o programador tenha acesso à funcionalidade de cifrar o agente, criou-se na classe *SecureAgent* o seguinte método:

public byte[] cryptAgent(String destination, Object obj)

O método *cryptAgent* é o responsável por permitir ao programador a possibilidade de cifragem do *Object* a ser migrado em um vetor de *bytes*.

Primeiramente, a partir do servidor de destino fornecido, a respectiva chave pública é obtida através do método *getRSAPublickey*. A partir dela, os bytes podem ser cifrados com o método *crypt.* Este vetor de *bytes* cifrados é o retorno da função, que, então, poderá ser acessado somente no servidor μCode de destino, a partir da chave privada do mesmo. A Figura 3 facilita a compreensão do funcionamento do método.

```
public byte[] cryptAgent(String destination, Object obj) throws
        FileNotFoundException {
        byte[] result = null;
        byte[] bytes = null;
        String publicKey = getRSAPublicKey(destination);
        PublicKey pubKey;
        /* É realizado a obtenção da chave publica
           para um objeto PublicKey. */
        /* O objeto passado como parametro é transformado
           em um vetor de bytes. */
        result = crypt(bytes,pubKey);
        return result;
}
```

Figura 3 - cryptAgent

Em contra partida, o método *decryptAgent,* apresentado a seguir*,* deve ser utilizado pelo programador para decifrar o vetor de *bytes*, transformando-o em seu tipo original, *Object*.

public Object decryptAgent(byte[] obj, MuServer server)

Este método tem a função de passar o vetor bytes para o método *decrypt,* existente na classe MuServer do servidor, que de fato irá realizar a decifragem dos dados. Após esta operação ter sido realizada, posteriormente as informações são transformadas em um O*bject,* que é associado ao *MuServer* passado como parâmetro.

Este método é utilizado para decifragem dos dados no momento em que um determinado servidor recebe o agente móvel. Cabe ao programador verificar se os dados

necessitam ser decifrados ou utilizar a abstração *TargetedMuAgent.* A abstração realiza todo o processo de criptografia e decifragem do agente automaticamente, necessitando apenas que os arquivos contendo as chaves necessárias estejam presentes. A Figura 4 facilita a compreensão do funcionamento deste método.

```
public Object deCryptAgent(byte[] obj, MuServer server) throws
        GeneralSecurityException, FileNotFoundException {
        Object object = null;
        byte[] result = MuServer.getServer().decrypt(obj);
        ByteArrayInputStream bais = new ByteArrayInputStream(result);
        MuObjectInputStream ois = new
                MuObjectInputStream(bais,server.incomingLoader);
        object = ois.readObject();
        ois.close();
        return object;
}
```

Figura 4 – deCryptAgent

Apesar deste método normalmente não ser utilizado diretamente pelo programador, o método a seguir é que de fato realiza o ciframento dos dados.

public byte[] crypt(byte[] object, Key key)

Ele é chamado pelo método *cryptAgent*. Em seus parâmetros são necessários o objeto na forma de um vetor de *bytes* e a chave pública.

Basicamente, o método faz uso da classe *Cipher*, provida pela *API* padrão da Sun, para cifrar as informações. Assim, resta ao programador a especificação do algoritmo a ser utilizado (no caso, RSA), e a passagem dos dados.

Além disto, é necessário que o vetor de bytes seja dividido em blocos no momento da cifragem. O algoritmo RSA, quando utiliza uma chave de 1024 bits, requer bloco de tamanho 117. Após cifrado, este bloco passa a ter tamanho 128. Somente o último bloco possivelmente terá um tamanho menor. A Figura 5 mostra o código deste método.

```
public byte[] crypt(byte[] object, Key key) throws
        GeneralSecurityException {
        byte[] result, data;
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(mode, key);
        ByteArrayOutputStream dest = new ByteArrayOutputStream();
        int rest = object.length%117;
        int length = object.length-rest;
        int i;
        for (i=0; i<length;i+=117)
                dest.write(cipher.doFinal(object, i, 117));
        dest.write(cipher.doFinal(object, i, rest));
        dest.flush();
        dest.close();
        result = dest.toByteArray();
        return result;
}
```

Figura 5 – crypt

De forma contrária, o método

public byte[] decrypt(byte[] object)

realiza a obtenção da chave privada e a decifragem dos dados. Um dos pontos principais referentes a este método é o fato dele estar na classe *MuServer*, e não em *SecureAgent*. Isto acontece porque a obtenção da chave privada precisar ser realizada pelo servidor, e não pelo agente móvel. A chave privada não deve ser disponibilizada a terceiros, pois assim seria possível que outros acessassem informações destinadas ao proprietário da chave.

Diferentemente de *crypt*, *decrypt* realiza a obtenção da chave antes da decifragem das informações. A figura 6 mostra aspectos de sua implementação.

```
public byte[] decrypt(byte[] object) throws
        GeneralSecurityException, FileNotFoundException {
        byte[] result, data;
        String privkey = null;
        PrivateKey privKey;
        /* Nome do arquivo contendo a chave privada é obtido. */
        /* Objeto PrivateKey obtém a chave privada do arquivo. */
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, privKey);
        ByteArrayOutputStream dest = new ByteArrayOutputStream();
        for (int i=0; i<object.length;i+=128)
                dest.write(cipher.doFinal(object, i, 128));
        dest.flush();
        dest.close();
        result = dest.toByteArray();
        return result;
}
```

Figura 6 - decrypt

O método

public String getRSAPublicKey(String destination)

possui a função de obter a chave pública do servidor para o qual o agente será cifrado. Para isto, o arquivo "rsa.public.keys" é lido em busca da respectiva chave pública do servidor *MuServer* de destino. A Figura 7 é um exemplo de conteúdo de um arquivo "rsa.public.keys".

```
# Arquivo contendo chaves públicas de servidores
127.0.0.1:2000 rodrigo.rsa.pubkey
10.0.0.2:2002 joao.rsa.publickey
200.183.155.180:1968 lost.rsa.pubkey
```

Figura 7 – rsa.public.keys

Linhas começadas pelo caracter '#' serão descartadas na análise do arquivo.

3.2 A abstração TargetedMuAgent

O ambiente µCode possui em sua *API* uma abstração de agente móvel, denominada de *MuAgent*. Esta abstração tem como objetivo facilitar o trabalho do programador, uma vez que com a utilização da mesma não se torna necessário nenhum tipo de preocupação com a forma em como a mobilidade do agente será realizada, nem em como este agente será reconstruído no servidor *MuServer* de destino.

Na seção 3.1 foram descritos os métodos que o programador pode utilizar para fazer uso do mecanismo *TargetedState* desenvolvido. Entretanto, com a intenção de facilitar a utilização deste mecanismo pelos programadores, foi criada a abstração *TargetedMuAgent*.

Para o programador, o uso de *TargetedMuAgent* é idêntico ao de um *MuAgent*. Para isso, deverá existir, no ambiente de execução do agente, um arquivo "rsa.private.key" contendo o nome da chave privada do servidor local e um arquivo "rsa.public.keys" contendo o nome das chaves públicas dos eventuais servidores para onde será feito o envio do agente.

Considere-se um exemplo com 3 servidores: A, B e C. Suponha-se que o agente comece em A e necessite migrar para B, C e depois retornar à origem. Para migrar de A para B, o agente realiza uma busca pela chave pública de B no arquivo "rsa.public.keys". Quando chegar em B, no momento da decifragem dos dados, o servidor irá buscar a chave privada de B no arquivo "rsa.private.key". Da mesma forma, quando for migrar para C, uma nova busca pela chave pública é realizada no arquivo "rsa.public.keys" contido em B. E assim sucessivamente até retornar a origem A.

É importante ressaltar que o agente não pode acessar a chave privada do servidor de destino, pois esta chave é exclusiva de cada um dos servidores. Desta forma, no momento da decifragem dos dados, deve-se utilizar o método *decrypt*, provido pela classe *MuServer* do servidor.

As principais alterações realizadas foram no método *go* e no método *unpack* originalmente existentes na classe *MuAgent*, que respectivamente possuem a função de empacotar e reconstruir o grupo que será migrado.

No método *go*, antes do agente móvel ser adicionado ao grupo que será migrado, este tem seus *bytes* cifrados através do método *cryptAgent*. É de valia ressaltar que neste método é feita uma busca pela chave pública do servidor de destino.

Desta forma, no método *unpack*, antes do agente ter sua reconstrução realizada, este tem seus *bytes* decifrados pelo método *decryptAgent*. Em *decryptAgent* os dados cifrados são

passados ao método *decrypt* pertencente ao servidor, para que este acesse a chave privada e faça a decifragem das informações.

3.3 Mecanismo ReadOnly

Este mecanismo é responsável pela proteção de informações estáticas a partir do uso de cifragem. Funciona como um *container* onde pode-se armazenar objetos que não poderão ser modificados durante o percurso do agente. Após serem adicionados, uma assinatura digital é gerada a partir da chave privada de seu proprietário. Desta forma, quando o código retorna ao servidor de origem, utiliza-se a chave pública deste proprietário, juntamente com a assinatura para verificar se o objeto não foi modificado. Esta verificação também pode ocorrer em servidores intermediários, bastando apenas que eles utilizem a chave pública do servidor de origem.

O método

public void addReadOnly(String label, Object obj)

proporciona ao programador a capacidade de adicionar um objeto ao *container* das informações estáticas. Este *container* nada mais é que um objeto *Hashtable* presente na classe *SecureAgent.*

O funcionamento é bem simples, bastando que se especifique um rótulo(*label*), passado como parâmetro, para identificar o objeto que está sendo adicionado. De forma simétrica, o método

public Object getReadOnly(String label)

retorna o objeto que foi previamente adicionado ao *container* de informações estáticas, a partir do rótulo fornecido. Para que o servidor possa tomar conhecimento dos objetos transportados por um agente, implementou-se o método

public String[] getReadOnlyLabels()

Este método tem a intenção de facilitar o trabalho do programador retornando um vetor de

Strings com os rótulos dos objetos que foram adicionados ao *container*.

Após os objetos terem sido adicionados ao *container* de informações estáticas, o método

public void signReadOnly()

deve ser chamado para que uma assinatura seja gerada para os mesmos a partir do algoritmo *DSA* (*Digital Signature Algorithm*), provido pela própria *Sun* em sua *API* (SUN MICROSYSTEMS, 2005). Desta forma, será possível verificar se alguma informação foi modificada.

Primeiramente, para que uma assinatura seja gerada, é necessário que se obtenha a chave privada do proprietário a partir do método *getDSAPrivateKey*. Posteriormente, o objeto *Hashtable* contendo as informações estáticas é transformado em um vetor de *bytes*, e passado para *Signature*. Assim, a assinatura pode ser gerada a partir da chave privada obtida anteriormente e armazenada no *Hashtable* signatures. A Figura 8 mostra o funcionamento do método.

```
public void signReadOnly() {
        String privKey = getDSAPrivateKey();
        PrivateKey priv;
        byte[] data;
        /* A chave privada é lida a partir de um arquivo
             para um objeto PrivateKey. */
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        sig.initSign(priv);
        /* Hashtable readOnly é transformado em um vetor de
           bytes(data) para ser passado a Signature. */
        sig.update(data);
        byte[] realSig = sig.sign();
        signatures.put("readOnlySign", realSig);
}
```

Figura 8 – signReadOnly

Para verificar se alguma informação foi modificada durante o percurso do agente, implementou-se o método

public boolean verifyReadOnly(String destination)

A partir do fornecimento do servidor que gerou a assinatura uma busca é feita pela chave pública, que então é utilizada para verificar a assinatura das informações armazenadas. O agente móvel pode utilizar este método quando estiver em um dos servidores, ou quando

retornar à sua origem. A Figura 9 mostra esse processo.

```
public boolean verifyReadOnly(String destination)
        throws SignatureException {
        String pubkey = getDSAPublicKey(destination);
        PublicKey pubKey;
        byte[] data;
        /* A chave publica é lida a partir de um arquivo
             para um objeto PublicKey. */
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        sig.initVerify(pubKey);
        /* Hashtable readOnly é transformado em um vetor de
           bytes(data) para ser passado a Signature. */
        sig.update(data);
        boolean verifies = false;
        verifies = sig.verify((byte[]) signatures.get("readOnlySign"));
        return verifies;
}
```

Figura 9 – verifyReadOnly

3.4 AppendOnly

Este mecanismo é responsável por coletar dados dos servidores que são visitados, garantindo que será possível verificar a integridade dos mesmos durante o percurso do agente.

AppendOnly funciona como um *container* que contém um vetor de objetos, um vetor das respectivas assinaturas destes objetos e um vetor de identidade dos usuários que os inseriram. Para um servidor inserir um dado no container, ele deve assinar o objeto com sua chave privada. A partir daí, armazena-se o objeto, a assinatura digital gerada e o nome do usuário. Desta forma, pode-se sempre verificar se estes objetos foram modificados ou não.

Além disto, é possível que se cifre o objeto, utilizando o método *cryptAgent* detalhado na seção 3.1.1, para que assim não se torne possível a visualização do conteúdo da informação por uma parte não confiável. Posteriormente, somente o servidor para o qual este objeto foi endereçado conseguirá decifra-lo através do método *decryptAgent*, descrito na seção 3.1.2.

O método

public void appendObject(String server, Object obj)

tem a função de adicionar um objeto ao *container* de informações. Para isto, é necessário que seja fornecido o servidor que está adicionando o objeto.

Primeiramente, é necessário que a chave privada do servidor em uso seja obtida.

Posteriormente, o objeto é transformado em um vetor de bytes para ter sua assinatura gerada. Finalmente, o objeto e a assinatura são armazenados em um objeto *Hashtable*, referenciados pelo servidor proprietário dos mesmos passado como parâmetro. A Figura 10 facilita a compreensão do método.

```
public void appendObject(String server, Object obj) {
        String privKey = getDSAPrivateKey();
        PrivateKey priv;
        byte[] realSig, data;
        /* A chave privada é lida a partir de um arquivo
             para um objeto PrivateKey. */
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        sig.initSign(priv);
        /* Objeto passado como parâmetro é transformado em um
         vetor de bytes(data) para ser passado a Signature. */
        sig.update(data);
        realSig = sig.sign();
        signatures.put(server, realSig);
        appendOnly.put(server, obj);
}
```

Figura 10 – appendObject

O método

public Object getObject(String server)

tem a função de resgatar um objeto inserido anteriormente por algum servidor. Para isso, basta que seja passado como parâmetro o servidor responsável por ter inserido tal objeto. Além disso, no momento em que o objeto está sendo obtido, sua assinatura é verificada. Desta forma, se a verificação ocorrer com sucesso, pode-se garantir que o objeto não foi modificado. Caso contrário, uma exceção será gerada. A Figura 11 mostra a implementação deste método.

```
public Object getObject(String server) throws SignatureException {
        String pubkey = getDSAPublicKey(server);
        byte[] data;
        /* A chave pública é lida a partir de um arquivo
             para um objeto PublicKey. */
        PublicKey pubKey;
        /* Objeto requisitado é transformado em um vetor de
           bytes(data) para ser passado a Signature. */
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        sig.initVerify(pubKey);
        sig.update(data);
        boolean verifies = sig.verify((byte[]) signatures.get(server));
        if (!verifies) throw new SignatureException("Signature do not"
        + " match specified public key");
        return appendOnly.get(label);
}
```

Figura 11 – getObject

O método

public String[] getObjectLabels()

retorna para o programador a lista de servidores que adicionaram algum objeto no *container AppendOnly*. A lista é retornada na forma de um vetor de *String*. Os servidores são necessários para que se possa obter os seus respectivos objetos, pois os rótulos dos objetos são os servidores que os armazenaram.

3.5 Esquema de proteção *versus* formas de ataques

Por meio do esquema de proteção desenvolvido, é possível que se garanta ao programador grandes possibilidades de proteger seu agente móvel contra as formas de ataques existentes.

Grande parte dos ataques acontecem quando o agente móvel trafega entre os nós de uma rede. Neste caso, pode-se garantir que o código deste agente, juntamente com suas informações, estratégia e fluxo de controle sejam protegidos pelo uso da abstração *TargetedMuAgent*. Com o uso do mecanismo *TargetedState*, a abstração garante que as informações sejam trafegadas de forma cifrada, proporcionando segurança às mesmas.

É de valia mencionar que essas informações são cifradas para um destinatário específico. Assim, quando o agente chega em um servidor, suas informações somente serão reveladas se este for o destinatário determinado. Entretanto, para que isto de fato aconteça, o servidor deve possuir a chave privada necessária para o deciframento. Garante-se então, com base em uma relação de confiança entre o remetente e o destinatário do agente, a segurança das informações transmitidas.

Contudo, se a intenção for proteger a manipulação de apenas alguns dados trafegados, pode-se utilizar o mecanismo *AppendOnly* ou *ReadOnly* para armazená-los. Por meio destes é possível verificar a autenticidade das informações através de suas assinaturas digitais. Por

outro lado, se for necessário protegê-las contra espionagem, estes mecanismos devem ser utilizados em conjunto com *TargetedState,* para que as informações sejam transportadas de forma cifrada.

Outro artifício que um *host* malicioso pode-se valer é o de uma máscara falsa para seu endereço de rede. Entretanto, este não obterá grandes vantagens se a aplicação estiver utilizando o mecanismo *TargetedState*. Isto porquê, mesmo que ele consiga obter as informações do agente móvel, não conseguirá acessá-las, pois não irá possuir a chave privada necessária.

No próximo capítulo serão mostrados alguns exemplos onde os mecanismos de segurança implementados são aplicados.

CAPÍTULO 4 - UTILIZANDO OS MECANISMOS DE SEGURANÇA

Para fazer uma primeira avaliação dos mecanismos de segurança implementados, algumas aplicações foram selecionadas. São exemplos simples que têm como objetivo realizar uma avaliação da implementação, principalmente no seu aspecto funcional. O uso mais intensivo da *API* desenvolvida deve ser realizado para que uma avaliação mais profunda possa ser obtida. Com esses exemplos, foi possível corrigir pequenos erros existentes durante a implementação, e posteriormente garantir que o mecanismo foi implementado com sucesso.

O primeiro mecanismo implementado, *TargetedState,* foi utilizado em um chat baseado em agentes móveis, denominado de *MobileChat*. O segundo, *ReadOnly*, foi utilizado por um agente que percorre diversos servidores em busca de um arquivo. Finalmente, *AppendOnly*, foi utilizado por um agente que busca o menor preço de um determinado carro em diversos *MuServers.* Estas aplicações precisaram sofrer modificações bem pequenas para utilizarem os mecanismos implementados, e serão detalhadas nas sub-seções seguintes.

4.1 TargetedState em Mobile Chat

O *MobileChat* é um sistema de comunicação em tempo real que permite a interação entre diversos usuários conectados em rede através de mensagens de texto. Foi desenvolvido como parte de um trabalho acadêmico dentro do próprio Grupo de Teste de Software do UNIVEM (CITRO, 2005).

O grande diferencial desse sistema de *chat* é o emprego de agentes móveis na função de efetuar a troca de mensagens e a transferência de informações entre os clientes e o servidor. Além disso, o próprio servidor tem a potencialidade de um agente móvel que migra através dos nós da rede de acordo com as variações ocorridas no ambiente, como alterações na latência (custo) entre determinados nós ou na localização física de determinados clientes.

Ao migrar o servidor carrega consigo informações sobre os clientes conectados e o custo envolvido na comunicação entre os nós da rede (RONDINA, 2005).

O *MobileChat* é composto de quatro pacotes: *server*, *client*, *chat* e *agents*. O pacote server contém a abstração de um servidor móvel (*ChatServer*) e de um servidor estático (*MainServer*). Este permanece fixo no nó da rede em que foi iniciado. O pacote client abrange as interfaces gráficas e a implementação do cliente, além de um servidor utilizado pelo cliente (*ClientServer*) para receber e enviar os agentes necessários. Os treze agentes que fazem parte do *MobileChat* estão no pacote *agents*. No pacote chat estão alguns componentes mais gerais do sistema, como lista de usuários, abstração de um usuário e tabelas de custo. A interface do ambiente é mostrada na Figura 12. Mais detalhes sobre a implementação e funcionamento do *MobileChat* podem ser encontrados em (RONDINA, 2005).

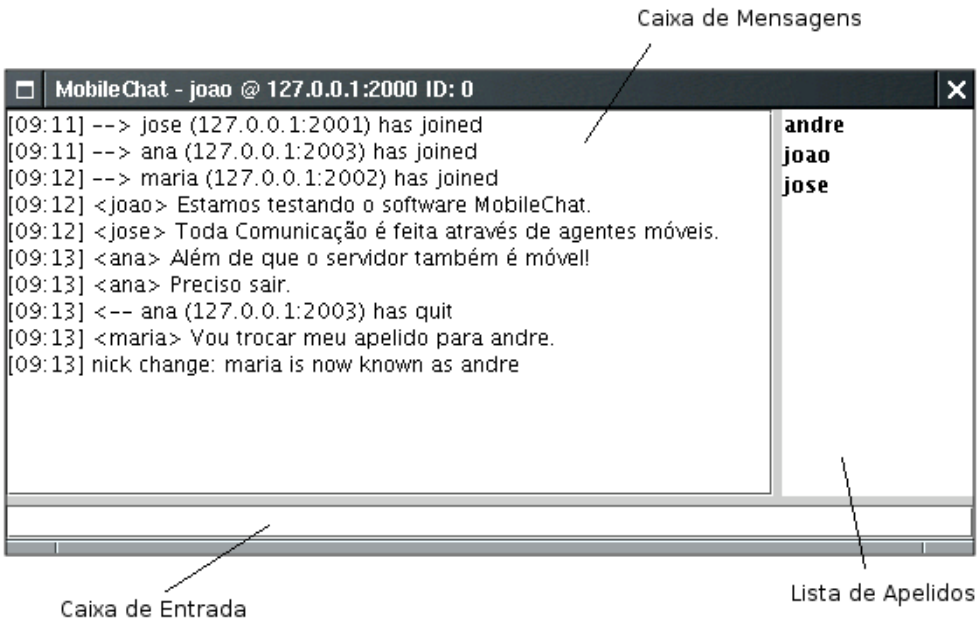

Figura 12 – Interface de *MobileChat*

O processo de adequação do *MobileChat* para utilizar o mecanismo *TargetedState* foi relativamente simples, o que de certa maneira é um aspecto positivo pois mostra que aplicações podem ser facilmente adaptadas para utilizar os mecanismos implementados. Bastou-se modificar a especificação da classe herdada pelos agentes móveis. Em vez de

estenderem *MuAgent*, passaram a estender a classe *TargetedMuAgent*. Dessa forma, qualquer agente, no momento de sua migração, será cifrado na origem e decifrado no servidor de destino.

Como a mensagem que um usuário envia ao chat também é entregue através de um agente móvel, *TargetedState* se torna útil no caso de se necessitar de privacidade das informações. Como as informações são trafegadas de forma cifrada, não é possível que sejam visualizadas durante o percurso do agente.

Vale ressaltar que cada *MuServer* necessita de um arquivo "rsa.public.keys" e "rsa.private.key". Isto porque o agente móvel necessita saber qual a chave pública de seu destinatário para a cifragem dos dados, além de, quando chegar em seu destino, o servidor possa acessar a sua chave privada para realizar a decifragem das informações.

4.1.1 Execução do MobileChat

Neste exemplo foi aberto um servidor, no qual um cliente irá se conectar. No servidor não é necessário que se passe nenhum parâmetro especial para o seu funcionamento. A Figura 13 mostra o servidor em execução, com o primeiro cliente acabando de se conectar. A interface gráfica não será exibida, pois é similar a da Figura 12.

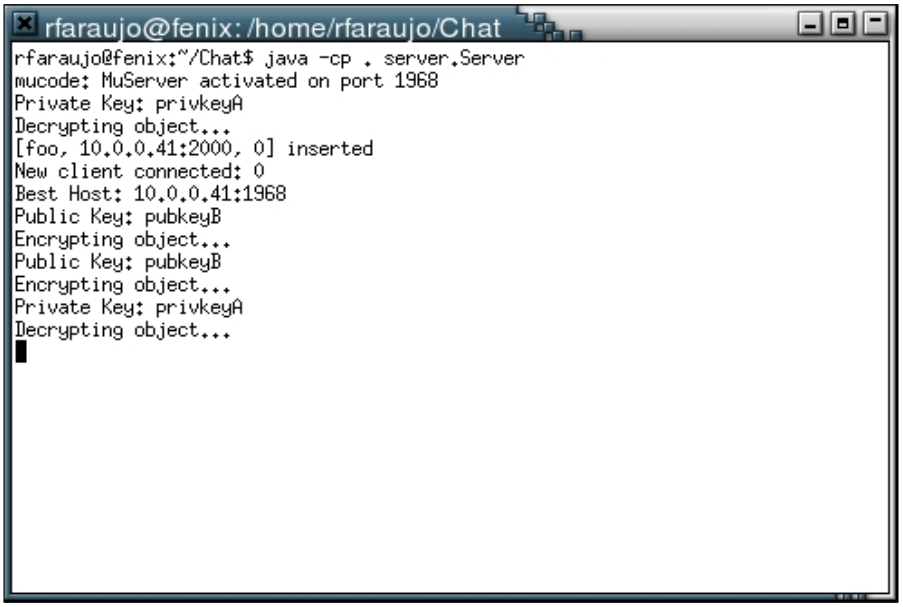

Figura 13 – Servidor MobileChat

Para se conectar ao chat, é necessário que se forneça um *nick*, um endereço de rede, o endereço do servidor e a tabela de custo. Esta tabela armazena o custo existente entre os servidores (cada cliente também é um servidor μCode). A Figura 14 mostra o cliente se conectando. Deve-se notar que a cada migração de agente, a chave é obtida, este é cifrado e posteriormente decifrado no destino.

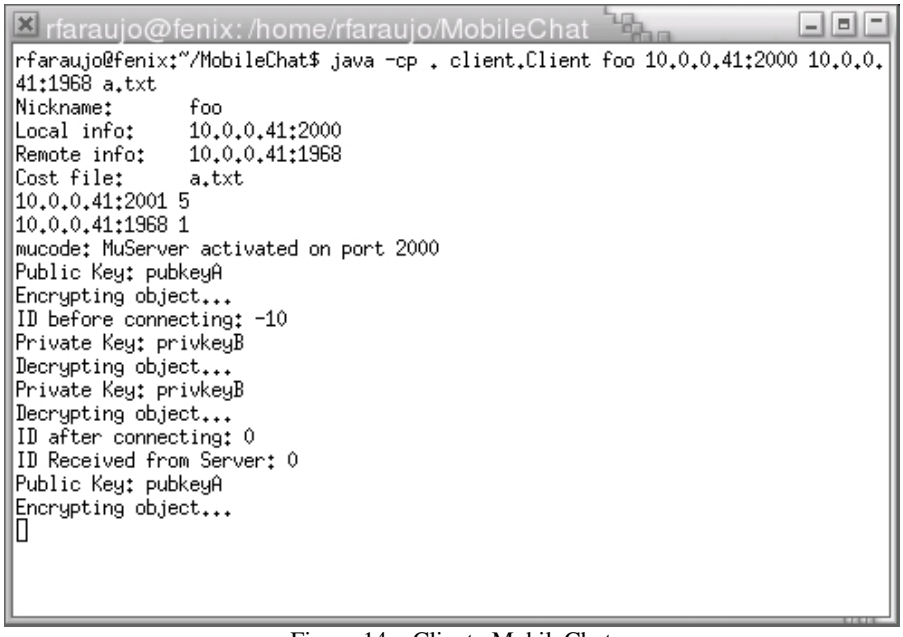

Figura 14 – Cliente MobileChat

4.1.2 MobileChat com erro na decifragem

Na Figura 15 é mostrado o que ocorre quando a chave privada não corresponde à chave pública utilizada para a cifragem do objeto. O servidor que estiver sendo executado não é finalizado, ele apenas não continua o processo de reconstrução do agente.

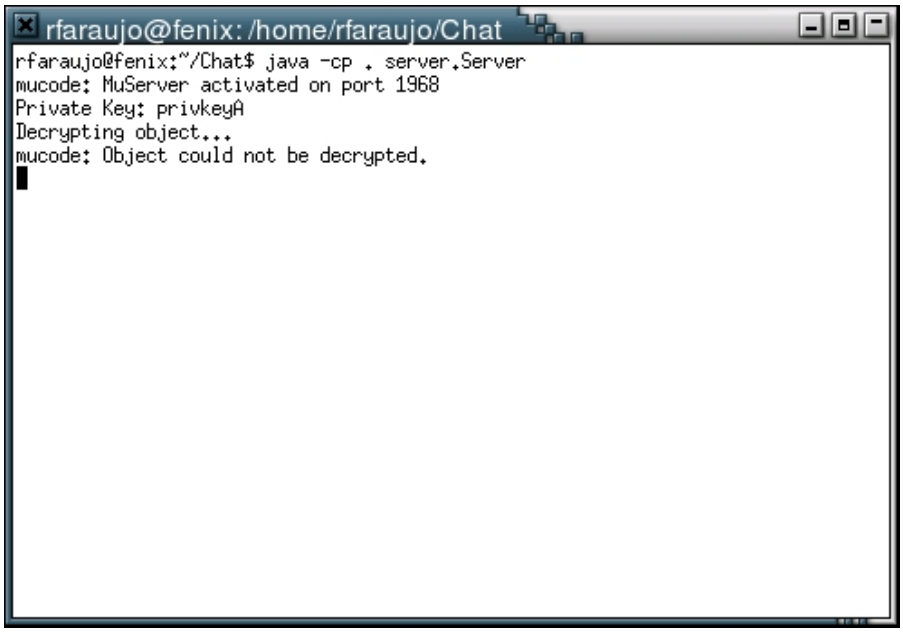

Figura 15 – Servidor MobileChat com chave privada inválida

A Figura 16 mostra a mensagem de erro quando a chave requisitada não é encontrada. Como no caso anterior, aqui o servidor também continua a executar normalmente, ele apenas não consegue reconstruir o agente recebido.

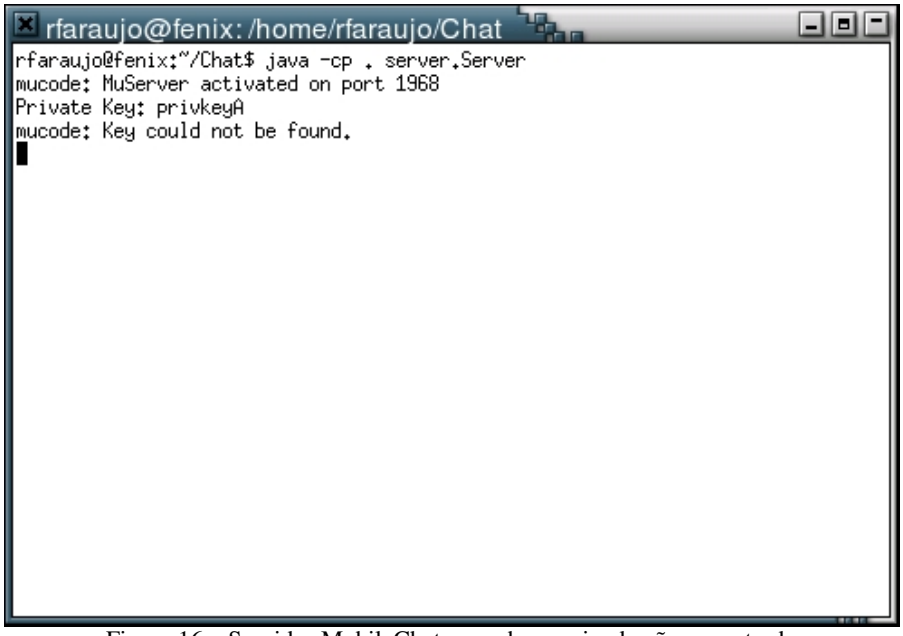

Figura 16 – Servidor MobileChat com chave privada não encontrada

4.2 ReadOnly em FileSearcher

FileSearcher é uma aplicação simples, que tem apenas a função de passar pelos servidores especificados em busca de um arquivo. Como o nome do arquivo a ser pesquisado não precisará ser modificado durante o percurso do agente, este foi adicionado ao *container ReadOnly*. Além disso, a lista de servidores pelos quais o agente irá trafegar também foi adicionada a este mesmo *container*. Dessa forma, um servidor não será capaz de modificar o nome do arquivo a ser procurado nem a lista de servidores. A habilidade de alterar tais informações permitiria que um determinado servidor obtivesse dados (o arquivo) de maneira ilícita.

Após os objetos serem adicionados ao *container*, é necessário que se chame o método *signReadOnly* para que a assinatura seja gerada. A Figura 17 mostra como isto é realizado.

```
SecureAgent sa = new SecureAgent();
String hosts[] = { "localhost:1968",
"localhost:2001", "localhost:2002" };
String file = "arquivo.txt";
sa.addReadOnly("hosts", hosts);
sa.addReadOnly("arquivo", file);
sa.signReadOnly();
```

Figura 17 – Adicionando arquivos somente leitura

Antes dos objetos serem obtidos, é aconselhável que se utilize o método *verifyReadOnly* para verificar se os mesmos não foram alterados. *A* Figura 18 mostra como isto é feito.

```
boolean verify = sa.verifyReadOnly();
if (verify) {
        String[] hosts = (String[]) sa.getReadOnly("hosts");
        String fname = (String) sa.getReadOnly("arquivo");
}
```

Figura 18 – Verificando arquivos somente-leitura

Como o retorno da função *getReadOnly* é um *Object*, é necessário que se faça um *cast* para o tipo de arquivo adicionado anteriormente.

4.3 AppendOnly em CarAgent

A aplicação *CarAgent* se trata de um agente móvel que percorre uma lista especificada de servidores em busca do menor preço de um carro. Ela é dividida em dois pacotes: *server* e *agent.* O pacote *server* provê classes e métodos para que o agente possa buscar o preço do carro do servidor em questão. O pacote *agent* provê o agente que irá percorrer os servidores móveis especificados, armazenando os preços de carro por onde passa.

O mecanismo *AppendOnly* está sendo utilizado para que nenhum servidor tente obter vantagem pela visualização dos preços adquiridos anteriormente pelo agente. Através deste mecanismo, é possível que cada servidor cifre sua informação (no caso, o preço), e desta forma ela somente poderá ser acessada no servidor local do agente, onde ele verificará qual é o mais baixo.

Para a cifragem das informações, está sendo utilizado o método *cryptAgent* e

decryptAgent, além dos métodos *appendObject* e *getObject*. A Figura 19 mostra como é feita a cifragem e o armazenamento das informações em cada um dos servidores pelos quais o agente passa. Primeiramente um objeto *SecureAgent* é instanciado. A partir daí, o preço do carro é obtido e cifrado através do método *cryptAgent*. Posteriormente o objeto é adicionado ao *container* de informações do mecanismo *AppendOnly* pelo método *appendObject*.

```
SecureAgent sa = new SecureAgent();
String model = "GOL";
PriceFinder pf = new PriceFinder();
double price = pf.getPrice(model);
Double d = new Double(price);
byte[] b = sa.cryptAgent(destination[0], d);
sa.appendObject(destination[nextDest], b);
```

Figura 19 – Utilização de appendObject

A Figura 20 mostra como é feita a decifragem e obtenção das informações armazenadas no servidor local do agente. Como no caso anterior, primeiramente é necessário que um objeto *SecureAgent* seja instanciado. A partir disto, as informações cifradas em forma de *bytes* são obtidas através do método *getObject* e passadas para o método *decryptAgent*, responsável por decifra-las.

```
SecureAgent sa = new SecureAgent();
double prices[] = new double[destination.length];
for (int i = 0; i < prices.length; i++) {
        byte[] b;
        Double d;
        b = (byte[]) sa.getObject(destination[i]);
        d = (Double) sa.decryptAgent(b, MuServer.getServer());
        prices[i] = d.doubleValue();
}
```

Figura 20 – Utilizando getObject

4.4 Considerações Finais

Um ponto positivo a ser ressaltado com base nos exemplos utilizados é que o processo de adequação de aplicações existentes para o uso dos mecanismos implementados foi relativamente simples, principalmente por não acarretar em mudanças bruscas no código dos

mesmos. Por exemplo, no caso da aplicação que fazia uso da abstração MuAgent e que necessitava do mecanismo TargetedState, foi necessário apenas que a herança da classe MuAgent fosse substituída pela classe TargetedMuAgent.

A partir dos exemplos utilizados acredita-se que a implementação realizada possa suprir as necessidades encontradas no início de desenvolvimento deste trabalho. Entretanto, mais exemplos ainda devem ser desenvolvidos com o intuito de garantir uma confiabilidade plena dos mecanismos de segurança implementados.

CAPÍTULO 5 - CONCLUSÕES

Através da implementação de um esquema de proteção para o ambiente μCode, espera-se garantir uma aceitação ainda maior para a tecnologia móvel, facilitando-se o uso desse ambiente. Acredita-se que a partir dos mecanismos de segurança desenvolvidos seja possível garantir tanto a segurança dos agentes móveis contra eventuais servidores maliciosos, como também a segurança dos servidores móveis contra possíveis ataques.

Primeiramente, estudos realizados com a intenção da elaboração de um esquema de proteção ao servidor μCode foram bem sucedidos, resultando na implementação de um mecanismo baseado em assinatura de código e atribuição de níveis de permissões. Desta forma cada usuário passou a possuir permissões customizadas, determinadas previamente pelo servidor em um arquivo de políticas Java.

Entretanto, foi possível perceber que era necessário garantir não somente a proteção do servidor contra aplicações maliciosas, mas também a proteção dos agentes móveis contra servidores mal intencionados. Assim, três mecanismos (*TargtetedState, ReadOnly* e *AppendOnly*) foram selecionados para que se estabelecesse um esquema de proteção adequado ao ambiente μCode.

TargetedState faz uso de criptografia de chave pública e é direcionado para situações em que o conteúdo do agente móvel é confidencial. *ReadOnly* utiliza assinatura digital e é direcionado para situações em que determinadas informações não serão modificadas durante o percurso de um agente móvel. O último deles, *AppendOnly,* utiliza assinatura digital e, se necessário, criptografia. Este é direcionado para situações em que informações serão adquiridas ao longo do percurso do agente.

Ressalta-se o fato de que após a fase de definição das técnicas a serem utilizadas para a implementação de um mecanismo de segurança efetivo, os estudos com base na bibliografia utilizada continuaram a ser realizados, uma vez que sempre que uma dificuldade foi

encontrada, uma orientação para a superação da mesma foi obtida através da bibliografia estudada.

Além do processo de implementação citado, este projeto contribuiu ainda para um aprendizado ainda mais profundo da linguagem Java, juntamente com conceitos inicialmente novos para o aluno, como os paradigmas de mobilidade de código e também mecanismos com a intenção de prover segurança às aplicações.

Acredita-se que este período tenha sido muito importante para uma formação acadêmica, não somente pela experiência de trabalhar em um projeto de pesquisa, mas também pela grande quantidade de conhecimento adquirido durante este período.

Um possível trabalho futuro relacionado ao assunto está na avaliação da performance da transferência de informações de forma cifrada, através dos mecanismos de segurança desenvolvidos, como também a realização de uma comparação de performance do algoritmo *RSA* com os demais algoritmos existentes.

Além disto, mais exemplos ainda devem ser desenvolvidos com o intuito de garantir uma confiabilidade plena dos mecanismos de segurança implementados, além de ser possível encontrar alguma necessidade que não seja suprida pelos mecanismos.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAUJO, Rodrigo. Implementação de Mecanismos de Segurança para Código Móvel. Terceiro Relatório Semestral FAPESP, 2005.

CARZANIGA, A.; PICCO, G.; VIGNA, G. Designing Distributed Applications With Mobile Code Paradigms. Proceedings of the 19th International Conference on Software Engineering (ICSE´97), Boston, MA, p. 22-32, Maio 1997.

CITRO, Elisangela. Utilização de Teste Estrutural em Código Móvel Java. Dissertação de Mestrado, 2005.

FUGETTA, A.; PICCO, G.; VIGNA, G. Understanding code mobility. IEEE Transactions on Software Engineering, vol. 24, p. 343-353, Maio 1998.

GONG, L. Inside Java 2 Platform Security: Architecture, API Design and Implementation. Editora Addison Wesley Longman, p. 34-37, p. 199-227, 1999.

HOHL, Fritz. A Model of Attacks of Malicious Hosts against Mobile Agents. In Fourth Workshop on Mobile Object Systems: Secure Internet Mobile Computations, 1998.

JANSEN, Wayne; KARYGIANNIS, Tom. Mobile Agent Security. NIST Special Publication 800-19, Agosto 1999.

JANSEN, Wayne. Countermeasures for Mobile Agent Security. Computer Communications, vol. 23, p. 1667-1676, Novembro 2000.

KARNIK, Neeran. Security in Mobile Agent Systems. Dissertação de Ph.D., Departamento de Ciência da Computação, Universidade de Minnesota, p. 81-92, Outubro 1998.

MCGRAW, G; FELTEN, E. Securing Java: Getting Down to Business with Mobile Code. Editora Wiley Computer Publishing, p. 37-114, 1999.

OAKS, Scott. Java Security. 2ª Edição. Editora O'Reilly, p. 225-247, 2003.

PICCO, Gian. A Mobile Code Toolkit. Disponível em: <http://mucode.sourceforge.net>. Acesso em Outubro de 2005.

RONDINA, Gustavo. Teste Estrutural em Código Móvel. Segundo Relatório Semestral FAPESP, 2005.

RSA Security. Disponível em <http://www.rsasecurity.com>. Acesso em Outubro de 2005.

SAU-KOON. Protecting Mobile Agents Against Malicious Hosts. Dissertação de Mestrado, Universidade Chinesa de Hong Kong, p. 15-41, Junho 2000.

SANDER, Tomas; TSCHUDIN, Christian. Protecting Mobile Agents Against Malicious Hosts. Lecture Notes in Computer Science, vol. 1419, p. 44-60, 1998.

SUN MICROSYSTEMS. Java 2 Platform SE 5.0. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/api/javax/crypto/package-summary.html>. Acesso em Outubro de 2005.

APÊNDICE A – SecureAgent

```
public class SecureAgent implements Serializable {

	private Hashtable signatures = new Hashtable();
	private Hashtable readOnly = new Hashtable();
	private Hashtable appendOnly = new Hashtable();

	public byte[] cryptAgent(String destination, Object obj) throws
FileNotFoundException {
		byte[] result = null;
		byte[] bytes = null;
		try {
			String publicKey = getRSAPublicKey(destination);
			FileInputStream keyfis = new FileInputStream(publicKey);
			byte[] encKey = new byte[keyfis.available()];
			keyfis.read(encKey);
			keyfis.close();
			X509EncodedKeySpec pubKeySpec = new
X509EncodedKeySpec(encKey);
			KeyFactory keyFactory = KeyFactory.getInstance("RSA");
			PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

			ByteArrayOutputStream baos = new ByteArrayOutputStream();
			ObjectOutputStream out = new ObjectOutputStream(baos);
			out.writeObject(obj);
			out.flush();
			out.close();
			bytes = baos.toByteArray();
			System.out.println("Encrypting object...");
			result = crypt(bytes, pubKey);
		} catch (FileNotFoundException f) {
			throw new java.io.FileNotFoundException("Key could not be found");
		} catch (Exception e) {
			e.printStackTrace();
		}
		return result;
	}

	public Object deCryptAgent(byte[] obj, MuServer server) throws
	GeneralSecurityException, FileNotFoundException {
				Object object = null;
		try {
			System.out.println("Decrypting object...");
			byte[] result = MuServer.getServer().decrypt(obj);
			ByteArrayInputStream bais = new ByteArrayInputStream(result);
			MuObjectInputStream ois = new
MuObjectInputStream(bais,server.incomingLoader);
			object = ois.readObject();
```

```
                ois.close();
            } catch (FileNotFoundException f) {
                throw new java.io.FileNotFoundException("Key could not be found");
            } catch (java.security.GeneralSecurityException g) {
                throw new java.security.GeneralSecurityException("Object could not
be decrypted");
            } catch (Exception e) {
                e.printStackTrace();
            }
            return object;
        }


        public byte[] crypt(byte[] object, Key key) throws
        GeneralSecurityException {
            byte[] result = null;
            byte[] data = null;
            try {
                Cipher cipher = Cipher.getInstance("RSA");
                cipher.init(Cipher.ENCRYPT_MODE, key);
                ByteArrayOutputStream dest = new ByteArrayOutputStream();
                int rest = object.length%117;
                int length = object.length-rest;
                int i;
                for (i=0; i<length;i+=117)
                    dest.write(cipher.doFinal(object, i, 117));
                dest.write(cipher.doFinal(object, i, rest));
                dest.flush();
                dest.close();
                result = dest.toByteArray();
            } catch (java.security.GeneralSecurityException b) {
                throw new java.security.GeneralSecurityException("Object could not
be encrypted");
            } catch(Exception e) {
                e.printStackTrace();
            }
            return result;
        }

        public void addReadOnly(String label, Object obj) {
            readOnly.put(label, obj);
        }

        public Object getReadOnly(String label) {
            return readOnly.get(label);
        }

        public String[] getReadOnlyLabels() {
            Enumeration e = readOnly.keys();
            int size = readOnly.size();
            String st[] = new String[size];
            for (int i=0; i<size;i++)
```

```
                    st[i]= (String) e.nextElement();
                return st;
        }

        public void signReadOnly() {
                String privKey = getDSAPrivateKey();
                try {
                        FileInputStream keyfis = new FileInputStream(privKey);
                        byte[] encKey = new byte[keyfis.available()];
                        keyfis.read(encKey);
                        keyfis.close();
                        PKCS8EncodedKeySpec privKeySpec = new
PKCS8EncodedKeySpec(encKey);
                        KeyFactory keyFactory = KeyFactory.getInstance("DSA");
                        PrivateKey priv = keyFactory.generatePrivate(privKeySpec);

                        // Inicia um objeto Signature com a chave primária
                        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
                        sig.initSign(priv);

                        ByteArrayOutputStream baosSignedData = new
ByteArrayOutputStream();
                        ObjectOutputStream out = new ObjectOutputStream(baosSignedData);
                        out.writeObject(readOnly);
                        out.flush();
                        out.close();
                        byte[] signedData = baosSignedData.toByteArray();
                        byte[] realSig = null;
                        if (sig!=null) {
                                sig.update(signedData);
                                realSig = sig.sign();
                        }
                        signatures.put("readOnlySign", realSig);

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
        public boolean verifyReadOnly(String destination) {
                String pubkey = getDSAPublicKey(destination);
                boolean verifies = false;
                try {
                        ByteArrayOutputStream baosSignedData = new
ByteArrayOutputStream();
                        ObjectOutputStream out = new ObjectOutputStream(baosSignedData);
                        out.writeObject(readOnly);
                        out.flush();
                        out.close();
                        byte[] signedData = baosSignedData.toByteArray();
                        FileInputStream keyfis = new FileInputStream(pubkey);
                        byte[] encKey = new byte[keyfis.available()];
                        keyfis.read(encKey);
```

```
                    keyfis.close();
                    X509EncodedKeySpec pubKeySpec = new
X509EncodedKeySpec(encKey);
                    KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
                    PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

                    // Inicia o objeto do tipo Signature com a chave pública
                    Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
                    sig.initVerify(pubKey);
                    sig.update(signedData);

                    // Verifica se a chave pública confere com a assinatura
                    verifies = sig.verify((byte[]) signatures.get("readOnlySign"));

            } catch (Exception e) { e.printStackTrace(); }
            return verifies;
        }

        public void appendObject(String label, Object obj) {
            String privKey = getDSAPrivateKey();
            byte[] realSig = null;
            try {
                    FileInputStream keyfis = new FileInputStream(privKey);
                    byte[] encKey = new byte[keyfis.available()];
                    keyfis.read(encKey);
                    keyfis.close();
                    PKCS8EncodedKeySpec privKeySpec = new
PKCS8EncodedKeySpec(encKey);
                    KeyFactory keyFactory = KeyFactory.getInstance("DSA");
                    PrivateKey priv = keyFactory.generatePrivate(privKeySpec);

                    // Inicia um objeto Signature com a chave primária
                    Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
                    sig.initSign(priv);

                    ByteArrayOutputStream baosSignedData = new
ByteArrayOutputStream();
                    ObjectOutputStream out = new ObjectOutputStream(baosSignedData);
                    out.writeObject(obj);
                    out.flush();
                    out.close();
                    byte[] signedData = baosSignedData.toByteArray();

                    if (sig!=null) {
                            sig.update(signedData);
                            realSig = sig.sign();
                    }

            } catch (Exception e) {
                    e.printStackTrace();
            }
            signatures.put(label, realSig);
```

```
                appendOnly.put(label, obj);

        }

        public Object getObject(String label) {
                String pubkey = getDSAPublicKey(label);
                try {

                        ByteArrayOutputStream baosSignedData = new
ByteArrayOutputStream();
                        ObjectOutputStream out = new ObjectOutputStream(baosSignedData);
                        out.writeObject(appendOnly.get(label));
                        out.flush();
                        out.close();
                        byte[] signedData = baosSignedData.toByteArray();
                        FileInputStream keyfis = new FileInputStream(pubkey);
                        byte[] encKey = new byte[keyfis.available()];
                        keyfis.read(encKey);
                        keyfis.close();
                        X509EncodedKeySpec pubKeySpec = new
X509EncodedKeySpec(encKey);
                        KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
                        PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

                        // Inicia o objeto do tipo Signature com a chave pública
                        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
                        sig.initVerify(pubKey);
                        sig.update(signedData);
                        boolean verifies = false;
                        if (((byte[]) signatures.get(label)) == null)
System.out.println("ASSINATURA NULA");
                        // Verifica se a chave pública confere com a assinatura
                        verifies = sig.verify((byte[]) signatures.get(label));

                        if (!verifies) throw new SignatureException("Signature do not" +
                        " match specified public key");
                } catch (Exception e) { e.printStackTrace(); }

                return appendOnly.get(label);
        }

        public String[] getObjectLabels() {
                Enumeration e = appendOnly.keys();
                int size = appendOnly.size();
                String st[] = new String[size];
                for (int i=0; i<size;i++)
                        st[i]= (String) e.nextElement();
                return st;
        }

        public String getDSAPrivateKey() {
                BufferedReader buffer = null;
```

```
		String privkey = null;
		try {
			buffer = new BufferedReader(new FileReader("dsa.private.key"));
			if (buffer==null)
				throw new FileNotFoundException("File dsa.private.key cannot
be found.");
		} catch (FileNotFoundException e) {
			System.out.println(e.getMessage());
		}
		try {

			StringTokenizer st;
			String temp;

			while((st=new StringTokenizer(buffer.readLine()))!=null) {
				temp=st.nextToken();
				if (!temp.substring(1).equals("#")) {
					privkey = temp;
					break;
				}
			}
			buffer.close();
			System.out.println("PRIVATE KEY: " + privkey);
			if (privkey == null) {
				throw new FileNotFoundException("Username/PublicKey was
not found.");
			}

		} catch(Exception e) {
			e.printStackTrace();
			System.out.println(e.getMessage());
		}
		return privkey;
	}

	public String getDSAPublicKey(String destination) {
		BufferedReader buffer = null;
		String pubkey = null;
		try {
			buffer = new BufferedReader(new FileReader("dsa.public.keys"));
			if (buffer==null)
				throw new FileNotFoundException("File dsa.public.keys cannot
be found.");
		} catch (FileNotFoundException e) {
			System.out.println(e.getMessage());
		}
		try {
			StringTokenizer st;
			String temp;

			while((st=new StringTokenizer(buffer.readLine()))!=null) {
				temp=st.nextToken();
```

```
                        if (!temp.substring(1).equals("#")) {
                            if (temp.equals(destination)) {
                                pubkey = st.nextToken();
                                break;
                            }
                        }
                    }
                    buffer.close();
                    if (pubkey == null) {
                        throw new FileNotFoundException("Server/PublicKey was not
found.");
                    }

                } catch(Exception e) {
                    System.out.println(e.getMessage());
                }
                return pubkey;
        }

        public String getRSAPublicKey(String destination) throws FileNotFoundException {
                BufferedReader buffer = null;
                String pubkey = null;
                buffer = new BufferedReader(new FileReader("rsa.public.keys"));
                if (buffer==null)
                    throw new FileNotFoundException("File rsa.public.keys cannot be
found.");


                StringTokenizer st;
                String temp;
                try {
                    while((st=new StringTokenizer(buffer.readLine()))!=null) {
                        temp=st.nextToken();
                        if (!temp.substring(1).equals("#")) {
                            if (temp.equals(destination)) {
                                pubkey = st.nextToken();
                                break;
                            }
                        }
                    }
                    buffer.close();
                } catch (Exception e ) { e.printStackTrace(); }
                if (pubkey == null) {
                    throw new FileNotFoundException("Server/PublicKey was not
found.");
                }
                System.out.println("Public Key: " + pubkey);
                return pubkey;
        }
}
```