

ALEX KALINOVSKY

# JAVA SECRETO

Técnicas de descompilação,  
patching e engenharia  
reversa

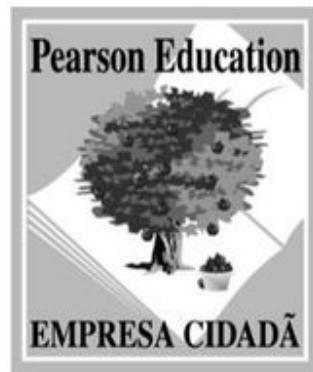


PEARSON

Makron  
Books

# **Java secreto**

## **técnicas de descompilação, patching e engenharia reversa**



# **Java secreto**

## **técnicas de descompilação, patching e engenharia reversa**

**Alex Kalinovsky**

Tradução

**Sandra Figueiredo & Carlos Schafranski**

Revisão técnica

**Leonardo Galvão**

Engenheiro de software da Rhealeza Informática  
e editor geral das revistas Clube Delphi e Java Magazine



São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha  
Guatemala México Peru Porto Rico Venezuela

© 2005 by Pearson Education do Brasil

Título original: **Covert Java: Techniques for Decompiling,  
Patching and Reverse Engineering**

© 2004 by SAMS Publishing

Tradução autorizada a partir da edição original em inglês,

Covert Java: Techniques for Decompiling, Patching and Reverse Engineering,  
publicada pela SAMS Publishing

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

*Diretor Editorial: José Martins Braga*

*Consultora Editorial: Docware Traduções Técnicas*

*Gerente de Produção: Heber Lisboa*

*Editor de Texto: Gustav Schmid*

*Designer de capa: Marcelo Françozo (sobre o projeto original de Gary Adair)*

*Editoração Eletrônica: Estúdio Castellani*

*Impressão: São Paulo – SP*

*Marcas comerciais*

Todos os termos mencionados neste livro conhecidos como marcas comerciais ou marcas de serviços estão em letras maiúsculas. A editora não pode atestar a exatidão dessas informações. A utilização de um termo neste livro não deve ser considerada como afetando a validade de qualquer marca comercial ou marca de serviço.

**Dados Internacionais de Catalogação na Publicação (CIP)**

**(Câmara Brasileira do Livro, SP, Brasil)**

---

Kalinovsky, Alex, 1974-

Java secreto: técnicas de descompilação, patching e engenharia reversa / Alex Kalinovsky; tradução Sandra Figueiredo & Carlos Schafranski; revisão técnica Leonardo Galvão. -- São Paulo: Pearson Education do Brasil, 2005.

Título original: Covert Java: techniques for decompiling, patching, and reverse engineering

Bibliografia.

ISBN 85-346-1539-X

1. Java (Linguagem de programação para computador).

I. Título.

04-5291

CDD-005.133

---

**Índices para catálogo sistemático:**

1. Java: Linguagem de programação: computadores: processamento de dados 005.133

2005

Direitos exclusivos para a língua portuguesa cedidos à

Pearson Education do Brasil,

uma empresa do grupo Pearson Education

Av. Ermano Marchetti, 1435

Cep: 05038-001 Lapa – São Paulo – SP

Tel.: (11) 3613-1222 – Fax: (11) 3611-0444

e-mail: vendas@pearsoned.com

# Sobre o livro Java secreto

*"Java secreto vai além dos tópicos avançados, discutindo coisas que só gurus de programação conhecem (descompilação, segurança, hacking de bytecode etc.). Gostaria de ter lido este livro no início de minha carreira. Ele teria me poupado horas a fio de aprendizado da maneira mais difícil.*

*Fico feliz que alguém finalmente teve a coragem de escrever um livro assim. O livro está repleto de informações controversas, não-triviais e muito úteis. Preenche um vazio no espaço de programação Java que nenhum outro livro conseguiu preencher."*

— Emmanuel Proulx, desenvolvedor Web; WebLogic 7 Certified Engineer

*"Java secreto fornece uma visão fascinante de táticas de desenvolvimento Java de bastidores, que são normalmente o domínio de veteranos. É definitivamente leitura recomendada para os que querem dominar a tecnologia Java."*

— Floyd Marinescu, autor, EJB Design Patterns; gerente geral & fundador do TheServerSide Communities

*"De vez em quando aparece um livro sobre Java que deve ser incluído em sua biblioteca. Java secreto é um desses livros.*

*Ele explora o tópico do hacking de código Java — para resolver problemas, não para criá-los. O autor explora várias técnicas de hacking, como descompilação de classes, patching e rastreamento de lógica, além de ferramentas (como FAR e JODE) que simplificam o trabalho de hacking. Uma variedade de 'Histórias das trincheiras' fornece experiências da vida real em que as técnicas de hacking foram necessárias para salvar projetos. Os questionários no final de cada capítulo ajudam a dominar a riqueza de informações valiosas e úteis do livro.*

*Este é um livro para os mocinhos, portanto, bandidos, cuidado. O autor fez um trabalho fantástico que explora um assunto delicado. Para todos os desenvolvedores Java, especialmente os envolvidos com sistemas J2EE e outros projetos Java pesados, o Java secreto é um livro obrigatório. Eu o recomendo sem reservas."*

— Jeff Friesen, autor e articulista da coluna "Java 101" do JavaWorld

*"O primeiro livro sobre hacking de software escrito especificamente para o desenvolvedor Java. Há 2.500 anos, o grande general Sun Tzu disse: 'Se você conhecer o inimigo e a si mesmo, não precisa temer o resultado de cem batalhas.' Se você for um desenvolvedor Java, este livro é um passo gigante para conhecer seus inimigos."*

— Kevin Bedell, editor-chefe da LinuxWorld; co-autor de Struts Kick Start, Programming Apache Axis

*"Java secreto mostra como chegar ao núcleo das aplicações em Java, da manipulação da VM à espionagem de chamadas JDBC. Um guia muito prático, fornece técnicas e ferramentas para entender como o Java funciona no nível de bytecode e por que isso é importante."*

— Craig Pfeifer, consultor técnico sênior, Impact Innovations Group

## O autor

Alex Kalinovsky nasceu na Ucrânia em 1974 e mudou-se para os Estados Unidos em 1997. Trabalha na área de TI há mais de 10 anos, tendo experiência que varia desde a escrita de aplicações em C e C++ até o desenvolvimento de soluções corporativas em Java. Desde 1997, Alex trabalha unicamente com Java e tem orgulho de ser um dos seus primeiros divulgadores. Ministrou mais de 15 cursos sobre as tecnologias Java corporativas e trabalhou como mentor de muitas equipes. Alex escreveu para várias publicações, incluindo *JavaWorld*, *Sun JavaSoft*, *Information Week* e *Washington Post*. Ele é consultor em Certified Enterprise Java Architect para importantes empresas que utilizam Java e J2EE. Também é o principal arquiteto do WebCream, um produto em Java revolucionário que faz a ponte entre o Swing e a HTML. No seu tempo livre, Alex gosta de viajar, ler, praticar windsurf, snowboard e musculação.

## **Dedicatória**

*Dedico este livro a meus pais, Stanislav e Lubov Kalinovsky, que me deram tudo o que puderam desde o primeiro dia de minha vida. Somente com o passar dos anos, uma pessoa começa a verdadeiramente entender e valorizar o impacto que a família tem sobre a sua vida, de modo que quero aproveitar esta oportunidade para agradecer a meus pais por todos os sacrifícios que fizeram e por toda a paciência que tiveram. Este trabalho também é um tributo a duas outras pessoas que tiveram uma influência enorme sobre minha vida: meu irmão Andrew Kalinovsky e meu segundo pai e mentor, Sergei Boiko. Obrigado e eu amo vocês todos.*

## **Agradecimentos**

Durante as longas horas que passei escrevendo este livro, muitas pessoas me ajudaram a realizar este projeto. Agradeço a meus grandes amigos, LaWanda Tetteh e Gleb Tulukin, por me darem suporte e estímulo quando precisei. Dedico um agradecimento especial a Amie Koker pela sua paciência e compreensão e a Tricia Riviere pelo seu senso de humor e perspicácia. Troy Davis e Yves Noel foram formidáveis em compartilhar suas técnicas e pontos de vista pessoais e em revisar o meu trabalho. Este livro não teria sido possível sem Todd Green, Sean Dixon e o restante da equipe na Sams Publishing, que compartilharam sua perícia e profissionalismo. Quero expressar minha admiração por todas as pessoas, aqui mencionadas ou não, que me ajudaram a completar esse objetivo.

**PÁGINA EM BRANCO**

# Sumário

<b>Introdução</b>	<b>xv</b>
<b>1 Iniciando</b>	<b>1</b>
Visão geral das técnicas – quando e por que utilizar cada método .....	1
Melhorando a produtividade com gerenciadores de arquivos .....	3
Funcionalidade e estrutura da aplicação de exemplo.....	7
Questionário rápido .....	8
Resumo .....	8
<b>2 Descompilando classes</b>	<b>9</b>
Determinando quando descompilar.....	9
Conhecendo os melhores descompiladores .....	10
Descompilando uma classe .....	12
O que torna a descompilação possível? .....	17
Problemas potenciais com código descompilado.....	18
Questionário rápido .....	21
Resumo .....	21
<b>3 Ofuscando classes</b>	<b>22</b>
Protegendo as idéias por trás do seu código .....	22
Ofuscamento como uma forma de proteção de propriedade intelectual.....	23
Transformações realizadas por ofuscadores .....	24
Conhecendo os melhores ofuscadores .....	28
Problemas potenciais e soluções comuns .....	29
Utilizando o Zelix KlassMaster para ofuscar uma aplicação de chat .....	32
Quebrando código ofuscado .....	36
Questionário rápido .....	36
Resumo .....	37
<b>4 Hackeando métodos não-públicos e variáveis de uma classe</b>	<b>38</b>
O problema do encapsulamento.....	38
Acessando pacotes e membros protegidos de classes .....	39
Acessando membros privados de classes.....	41
Questionário rápido .....	44
Resumo .....	44

<b>5 Substituindo e aplicando patches a classes de aplicações</b>	<b>45</b>
O que fazer depois de termos explorado cada possibilidade, mas falharmos? . . . . .	45
Localizando a classe em que é necessário aplicar um patching . . . . .	47
Exemplo de cenário que requer patching . . . . .	49
Aplicando um patch a uma classe para adicionar nova lógica . . . . .	53
Reconfigurando a aplicação para carregar e utilizar a classe com patch . . . . .	53
Aplicando patch a pacotes selados . . . . .	54
Questionário rápido . . . . .	55
Resumo . . . . .	55
<b>6 Utilizando rastreamento eficaz</b>	<b>56</b>
Introdução ao rastreamento . . . . .	56
Rastreamento como um método eficaz de conhecimento do software . . . . .	57
APIs e ferramentas de rastreamento e logging . . . . .	58
Rastreamento: recomendações e cuidados . . . . .	59
Questionário rápido . . . . .	60
Resumo . . . . .	60
<b>7 Manipulando a segurança Java</b>	<b>61</b>
Visão geral sobre a segurança em Java . . . . .	61
Driblando verificações de segurança . . . . .	63
Questionário rápido . . . . .	65
Resumo . . . . .	65
<b>8 Espionando o ambiente de execução</b>	<b>66</b>
A vantagem de dominar o ambiente de execução . . . . .	66
Propriedades de sistema . . . . .	67
Informações de sistema . . . . .	68
Informações de memória . . . . .	68
Informações de rede . . . . .	69
Acessando variáveis de ambiente . . . . .	70
Questionário rápido . . . . .	71
Resumo . . . . .	71
<b>9 Quebrando código com depuradores heterodoxos</b>	<b>72</b>
Entendendo o funcionamento interno de aplicações desconhecidas . . . . .	72
Depuradores convencionais e suas limitações . . . . .	73
Hackeando com um depurador onisciente . . . . .	73
Questionário rápido . . . . .	78
Resumo . . . . .	78
<b>10 Utilizando profilers para análise de aplicações em tempo de execução</b>	<b>79</b>
Por que e quando utilizar profiling . . . . .	79

Os melhores profilers para Java.....	80
Investigando o uso do heap e a freqüência da coleta de lixo para melhorar o desempenho.....	80
Pesquisando a alocação e referências de objetos para encontrar e corrigir vazamentos de memória .....	82
Investigando a alocação e a sincronização de threads.....	86
Identificando métodos de alto custo para melhorar o desempenho.....	89
Investigando uma aplicação em tempo de execução utilizando um dump de threads .....	90
Questionário rápido .....	92
Resumo .....	92
<b>11 Realizando testes de carga para encontrar e corrigir problemas de escalabilidade</b>	<b>93</b>
A importância do teste de carga .....	93
Testes de carga de servidores baseados em RMI com JUnit .....	95
Testes de carga com o JMeter .....	98
Questionário rápido .....	108
Resumo .....	108
<b>12 Aplicações de engenharia reversa</b>	<b>109</b>
Elementos e recursos da interface com o usuário.....	109
Hackeando textos .....	110
Hackeando imagens .....	111
Hackeando arquivos de configuração.....	113
Questionário rápido .....	113
Resumo .....	113
<b>13 Técnicas de eavesdropping</b>	<b>114</b>
Definição de eavesdropping .....	114
Eavesdropping em HTTP .....	115
Técnicas de eavesdropping no protocolo RMI .....	119
Eavesdropping do driver JDBC e de instruções de SQL .....	122
Questionário rápido .....	124
Resumo .....	124
<b>14 Controlando o carregamento de classes</b>	<b>125</b>
Funcionamento interno da JVM do ponto de vista do carregamento de classes .....	125
Escrevendo um class loader personalizado .....	129
Questionário rápido .....	133
Resumo .....	133

<b>15 Substituindo e aplicando patches a classes Java básicas</b>	<b>134</b>
Por que se incomodar? .....	134
Aplicando patch a classes Java básicas utilizando o classpath de inicialização .....	135
Exemplo do patching de <code>java.lang.Integer</code> .....	136
Questionário rápido .....	138
Resumo .....	138
<b>16 Interceptando o fluxo de controle</b>	<b>139</b>
Definição de fluxo de controle .....	139
Interceptando erros de sistema .....	140
Interceptando fluxos de sistema .....	140
Interceptando uma chamada a <code>System.exit</code> .....	142
Reagindo a uma desativação da JVM utilizando <i>hooks</i> .....	144
Interceptando métodos com um proxy dinâmico .....	144
A interface do profiler da máquina virtual Java .....	147
Questionário rápido .....	148
Resumo .....	148
<b>17 Entendendo e ajustando o bytecode</b>	<b>149</b>
Fundamentos de bytecode .....	149
Visualizando arquivos de classes com o visualizador de bytecode jClassLib ..	150
O conjunto de instruções da JVM .....	150
O formato de arquivos de classes .....	152
Instrumentando e gerando bytecode .....	158
Ajuste de bytecode em comparação com proxies dinâmicos e AOP .....	166
Questionário rápido .....	166
Resumo .....	167
<b>18 Controle total com patches de código nativo</b>	<b>168</b>
Por que e quando aplicar patch a código nativo .....	168
O uso do código nativo na máquina virtual Java .....	169
Abordagens genéricas para o patching de métodos nativos .....	173
Aplicando patch a código nativo na plataforma Windows .....	174
Aplicando patch a código nativo em plataformas Unix .....	181
Questionário rápido .....	182
Resumo .....	183
<b>19 Protegendo aplicações comerciais contra hacking</b>	<b>184</b>
Definindo objetivos para a proteção das aplicações .....	184
Tornando dados seguros com a Java Cryptography Architecture .....	185
Protegendo a distribuição da aplicação contra hacking .....	191
Implementando licenciamento para desbloquear funcionalidades de aplicações .....	199

---

Questionário rápido .....	208
Resumo .....	209
<b>A Licença de software comercial</b>	<b>210</b>
<b>B Recursos</b>	<b>216</b>
Utilitários e ferramentas .....	216
Descompilação .....	216
Ofuscamento .....	217
Rastreamento e logging .....	217
Depuração .....	217
Profiling .....	217
Testes de carga .....	218
Eavesdropping .....	218
Ajuste de bytecode .....	219
Aplicando patch a código nativo .....	219
Proteção contra hacking .....	220
<b>C Respostas do questionário</b>	<b>221</b>
Capítulo 1 .....	221
Capítulo 2 .....	221
Capítulo 3 .....	222
Capítulo 4 .....	222
Capítulo 5 .....	222
Capítulo 6 .....	223
Capítulo 7 .....	223
Capítulo 8 .....	223
Capítulo 9 .....	223
Capítulo 10 .....	224
Capítulo 11 .....	224
Capítulo 12 .....	225
Capítulo 13 .....	225
Capítulo 14 .....	226
Capítulo 15 .....	226
Capítulo 16 .....	226
Capítulo 17 .....	227
Capítulo 18 .....	227
Capítulo 19 .....	228
<b>Índice</b>	<b>231</b>

**PÁGINA EM BRANCO**

# Introdução

Há muitos bons livros escritos sobre Java. Às vezes me surpreende a quantidade de livros que você pode encontrar sobre um mesmo assunto. Pesquisar em [www.amazon.com](http://www.amazon.com) por um livro sobre Enterprise JavaBeans (EJB) retorna mais de 50 resultados. Convenhamos! EJB é uma tecnologia complexa e, hoje, todo desenvolvedor de respeito precisa tê-la no seu currículo, mas 50 livros? Portanto, que direito tenho eu de acrescentar mais um volume à coleção de publicações Java? Bem, acredito que há algumas técnicas menos divulgadas de desenvolvimento que, quando utilizadas corretamente, podem produzir resultados espantosos. A maioria dos métodos lida com conceitos e questões básicas sobre Java e, portanto, pode ser utilizada em uma variedade de aplicações. As técnicas apresentadas neste livro são soluções heterodoxas para problemas comuns no desenvolvimento em Java. Algumas delas são controversas e devem ser utilizadas com bastante cuidado, mas todas são métodos poderosos para você obter os resultados desejados. Aprendendo essas técnicas, você será capaz de destacar-se da maioria dos outros desenvolvedores apresentando uma solução quando todas as outras pessoas estão ainda tentando entender qual é realmente o problema. Talvez você já tenha utilizado algumas técnicas apresentadas neste livro. Se for o caso, parabéns, mas tenho certeza de que você encontrará alguns novos truques úteis ao examinar as recomendações aqui fornecidas.

Boa parte deste livro é dedicada a técnicas comumente consideradas como hacking. O termo *hacking* é utilizado bem livremente na mídia e, freqüentemente, com uma conotação negativa. Os hackers costumam ser retratados como geeks fanáticos por computador, que querem elevar sua auto-estima. Em alguns casos isso certamente é verdade. Entretanto, os métodos apresentados neste livro são voltados aos desenvolvedores de software profissionais, e cada técnica tem uma aplicação prática na vida real.

## Quem se beneficiará deste livro?

Desenvolvedores e arquitetos são os leitores que melhor poderão tirar proveito deste trabalho. Para aproveitar verdadeiramente os problemas e soluções apresentadas neste livro, você deve ter completado pelo menos algumas aplicações em Java significativas e trabalhado com código de terceiros. Isso não significa que desenvolvedores iniciantes não tenham nada a ganhar com este trabalho. Para manter este livro conciso e focado nos tópicos principais, pouca atenção é dada a assuntos que acreditamos ser bem conhecidos ou bem documentados. Por exemplo, ao discutir o processo de hacking de membros não-públicos de classes, este livro não explica as limitações impostas por cada modificador de visibilidade. Essas informações podem ser facilmente obtidas na Internet ou em livros que tratem de tais tópicos em detalhes. *Java secreto: Técnicas de descompilação, patching e engenharia reversa* trata de técnicas extremas que ultrapassam limites comumente esperados.

Vale destacar que as técnicas apresentadas aqui são bastante independentes umas das outras. Como a apresentação deste material segue uma ordem “primeiro os métodos mais comuns e mais simples”, sinta-se livre para pular capítulos e ir diretamente para aquele em que você está interessado. O Capítulo 1 inclui uma seção que descreve brevemente cada uma das técnicas e quando utilizá-las e, portanto, recomendo que você primeiro se familiarize com ele.

## Os aspectos morais e legais do hacking

A maioria dos capítulos é estritamente técnica, mas é extremamente importante entender que nem todas as técnicas podem ser aplicadas livremente ao trabalhar com aplicações. Nem todas as abordagens apresentadas neste livro são “hacking”, mas, se utilizadas sem antes verificar suas consequências legais, certamente poderão lhe trazer problemas. Vamos começar tentando oferecer uma definição ampla sobre hacking e então examinar como percorrer esse terreno traíçoeiro.

O dicionário *Merriam-Webster* apresenta a seguinte definição para o termo *hacker*: “(...) um especialista em programação e na solução de problemas com um computador”. Entretanto, há um outro significado logo após: “(...) pessoa que, ilegalmente, acessa e às vezes adultera informações em um sistema de computador”. Ser um especialista em programação é certamente algo excelente; mexer com coisas ilegais o colocará na cadeia. A breve mensagem é que este livro é voltado a pessoas bem-intencionadas e, se você for uma pessoa mal-intencionada, por favor pare de ler agora mesmo e procure um novo emprego na equipe de testes. Quaisquer informações ou descobertas podem ser utilizadas para o bem ou para o mal. Não são as informações, mas o uso delas que determina se o resultado é considerado positivo ou negativo.

Já houve diversos casos judiciais famosos relacionados com direitos autorais digitais, engenharia reversa e violações de patentes. Empresas e pessoas perdem milhões de dólares e às vezes também suas reputações. Embora as leis sejam complexas e os contratos de licenças sejam escritos por advogados para advogados, não é tão difícil evitar os problemas legais. Eis as duas regras básicas a serem seguidas:

- Se um autor espera que você pague pelo trabalho dele, pague;
- Se você estiver mexendo com algo, certifique-se de não prejudicar os interesses do autor.

Simples e eficaz. A primeira regra é muito fácil de entender, mas é a segunda que é importante lembrar ao aplicar os métodos apresentados neste livro. Por exemplo, se você fizer engenharia reversa do código de alguém para encontrar uma maneira de contornar um bug, é pouco provável que o autor o processe. Entretanto, se você fizer engenharia reversa do código de uma pessoa e criar um produto concorrente com base nos mesmos princípios únicos, há uma grande possibilidade de o autor processá-lo.

É importante lembrar que o software em que estamos trabalhando é escrito por pessoas como eu ou você e, como nós, essas pessoas têm contas para pagar. O Open Source é um

fenômeno diferente. Como o código está livremente disponível, você não precisa utilizar métodos extremos para aprender sobre o produto ou alterar algo nele. Mas a maioria dos softwares desenvolvidos hoje é comercial, e a maioria das inovações é feita por fornecedores comerciais. Hackear o software para evitar o pagamento de taxas de licença é contraproducente, porque isso prejudica o mercado de software e indiretamente prejudica os desenvolvedores. Furtar sorvetes de uma loja ao lado aumentará o preço do sorvete ou levará a sorveteria a fechar as portas. E, se você tiver uma padaria, o proprietário da sorveteria talvez comece a roubar os seus biscoitos.

As duas regras abrangem os aspectos morais do processo de hacking, mas o que geralmente abrange os aspectos legais são os direitos autorais, as leis de propriedade intelectual e os contratos de licenciamento do usuário final (End User Licence Agreements – EULAs). As leis são complexas e difíceis de interpretar, mas os EULAs são uma necessidade, porque geralmente são mais restritivos do que as leis. São escritos para oferecer ao autor a proteção que talvez não seja concedida adequadamente pelas respectivas leis, e geralmente é exigido que os usuários concordem explicitamente com os termos do acordo antes de utilizar um produto. Por exemplo, mesmo que a engenharia reversa não seja proibida por lei, a maioria dos produtos de software proíbe isso em um EULA. Portanto, é *imprescindível* analisar completamente o EULA antes de utilizar as técnicas descritas neste livro em um produto. Para evitar a repetição e manter o conteúdo deste livro estritamente técnico, o material dos capítulos não menciona os aspectos legais associados às técnicas. É *sua* responsabilidade assegurar a legalidade das suas ações.

## Características especiais do texto

Várias convenções tipográficas são utilizadas em *Java secreto: técnicas de descompilação, patching e engenharia reversa* para tornar o texto mais legível. O *itálico* é utilizado para enfatizar e indicar termos novos. Texto *monoespacado* é utilizado para trechos de código, nomes de arquivo e URLs. Texto *monoespacado em itálico* indica pontos a serem substituídos na sintaxe dos códigos.

Além disso, alguns elementos especiais são utilizados neste livro. “Histórias das trincheiras” descrevem minhas próprias experiências ao trabalhar com as várias técnicas descritas por todo o *Java secreto: técnicas de descompilação, patching e engenharia reversa* para ajudá-lo a entender como essas técnicas funcionam na prática. Cada capítulo termina com uma seção de “Resumo” com os principais pontos do capítulo, bem como uma seção de questionário para ajudá-lo a revisar o material.

**PÁGINA EM BRANCO**

# Iniciando

## Visão geral das técnicas – quando e por que utilizar cada método

A Tabela 1.1 apresenta uma breve visão geral das técnicas discutidas em mais detalhes nos capítulos correspondentes. Utilize essa tabela como um guia introdutório para este livro.

TABELA 1.1

### Visão geral sobre as técnicas

CAPÍTULO	TÉCNICA	ÚTIL PARA
2	Descompilar classes	Recuperar código-fonte perdido Aprender sobre a implementação de um recurso ou truque Solucionar problemas em códigos não-documentados Corrigir bugs urgentes no código de produção ou de terceiros Avaliar como seu código poderia ser alterado por hackers
3	Ofuscar classes	Proteger o bytecode contra descompilação Proteger a propriedade intelectual no bytecode Evitar que as aplicações sejam hackeadas
4	Hackear métodos e variáveis não-públicas de uma classe	Acessar funcionalidades existentes mas não expostas Alterar os valores das variáveis internas

# 1

## NESTE CAPÍTULO

- ▶ Visão geral das técnicas – quando e por que utilizar cada método 1
- ▶ Melhorando a produtividade com gerenciadores de arquivos 3
- ▶ Funcionalidade e estrutura da aplicação de exemplo 7
- ▶ Questionário rápido 8
- ▶ Resumo 8

**TABELA 1.1****Continuação**

CAPÍTULO	TÉCNICA	ÚTIL PARA
5	Substituir e aplicar patch a classes de aplicações	Alterar a implementação de uma classe sem ter de reconstruir toda a biblioteca Alterar a funcionalidade de uma aplicação ou framework de terceiros
6	Utilizar rastreamento eficaz	Criar aplicações de fácil manutenção e solução de problemas Aprender o funcionamento interno de uma aplicação Inserir informações de depuração em aplicações existentes para entender detalhes de implementação
7	Manipular a segurança em Java	Adicionar ou remover restrições no acesso a recursos críticos de sistema
8	Espionar o ambiente de execução	Determinar os valores de propriedades de sistema Determinar informações de sistema, tais como número de processadores e limites de memória Determinar uma configuração de rede
9	Quebrar o código com depuradores heterodoxos	Hackear aplicações sem um bom rastreamento Analizar o fluxo de controle de aplicações com múltiplos threads Quebrar aplicações ofuscadas
10	Utilizar profilers para análise da aplicação em tempo de execução	Investigar a utilização do heap e a freqüência da coleta de lixo para melhorar o desempenho Examinar a alocação e referências de objetos para encontrar e corrigir vazamentos de memória Investigar a alocação e a sincronização de threads para encontrar problemas de bloqueio e de concorrência no acesso a dados, para melhorar o desempenho Investigar uma aplicação em tempo de execução para entender melhor sua estrutura interna
11	Testes de carga para encontrar e corrigir problemas de escalabilidade	Criar scripts de testes automatizados que simulem uma determinada carga em um sistema Analizar como a aplicação atende aos requisitos de nível de serviço, como escalabilidade, disponibilidade e tolerância a falhas
12	Aplicações da engenharia reversa	Hackear os elementos da interface com o usuário, como mensagens, alertas, prompts, imagens, ícones, menus e cores
13	Técnicas de eavesdropping	Interceptar a comunicação HTTP entre o navegador e o servidor Web Interceptar a comunicação entre o cliente e o servidor RMI Interceptar instruções e valores SQL do driver JDBC

**TABELA 1.1****Continuação**

CAPÍTULO	TÉCNICA	ÚTIL PARA
14	Controlar o carregamento de classes	Implementar um <i>class loader</i> personalizado para controlar como e de onde classes são carregadas Usar o utilitário <i>class loader</i> personalizado para instrumentalizar o bytecode em execução Criar classes programaticamente em tempo de execução
15	Substituir e aplicar patch a classes Java básicas	Modificar as implementações das classes de sistema para alterar o comportamento básico Aprimorar a funcionalidade básica do JDK para atender às necessidades da aplicação Corrigir bugs na implementação do JDK
16	Interceptar o fluxo de controle	Reagir elegantemente a erros de sistema como falta de memória e estouro de pilha Capturar a saída para <code>System.out</code> e <code>System.err</code> Interceptar chamadas a <code>System.exit( )</code> Reagir ao encerramento da JVM Interceptar qualquer chamada de método, alocação de objetos ou evento de ciclo de vida de threads via JVMPPI
17	Entender e ajustar bytecode	Alterar a implementação de classes no nível de bytecode Gerar bytecodes programaticamente Instrumentar o bytecode para incluir nova lógica
18	Controle total com patches de código nativo	Aplicar patch à implementação de funções nativas Estender o comportamento da JVM no nível mais baixo
19	Proteger aplicações comerciais contra hacking	Proteger informações sigilosas com a criptografia Java Assegurar a integridade de dados com assinaturas digitais Implementar uma política de licença segura para desbloquear recursos de aplicações comerciais

## Melhorando a produtividade com gerenciadores de arquivos

As técnicas discutidas neste livro objetivam aumentar a produtividade no desenvolvimento. Ao final do dia, a qualidade e a produtividade são o que diferenciam programadores especialistas de programadores iniciantes e, como este livro é voltado a tornar os leitores especialistas, sinto que é meu dever apresentar algumas ferramentas de produtividade. Hacking e o desenvolvimento típico requerem a manipulação de arquivos e diretórios e obter a ferramenta certa pode tornar isso muito mais fácil. Obviamente, cabe a você decidir quando utilizar uma ferramenta. Você deve lembrar-se de que a maioria das

ferramentas requer investimento antecipado na instalação, configuração e aprendizagem – sem mencionar as possíveis taxas de licença. Mas como ocorre com a maioria das ferramentas, o investimento é recompensado muito rapidamente.

Vamos analisar duas substituições avançadas à combinação do Windows Explorer, Editor de Textos/Bloco de Notas e o CMD.EXE. Vamos focalizar o Windows, porque é nele que a maior parte do desenvolvimento em Java acontece, mas ferramentas de produtividade também podem estar disponíveis em outras plataformas. Talvez soe estranho o fato de iniciarmos um livro sobre Java avançado discutindo o Bloco de Notas e o CMD.EXE, mas muitos desenvolvedores ainda o utilizam, por isso apresento uma alternativa melhor.

O Windows Explorer é um shell simples e fácil de aprender, mas não ajuda nas tarefas que um programador precisa realizar. Um exemplo muito simples é criar e executar um arquivo .bat. Utilizando a interface padrão do Windows, você teria de pesquisar o diretório-alvo, clicar com o mouse em algumas caixas de diálogo para criar um novo arquivo e então abrir e editar esse arquivo no Bloco de Notas. Para executar o arquivo, você poderia dar um duplo-clique nele, mas qualquer saída ou erros seriam perdidos na janela CMD automaticamente aberta pelo Explorer. Portanto, uma maneira melhor é abrir um CMD.EXE, pesquisar o diretório e então executar o arquivo. Ao final, você vai ter que lidar com três janelas abertas não-sincronizadas ou relacionadas. Uma alternativa melhor é utilizar um software de gerenciamento de arquivos integrado que combine uma interface de navegação de diretórios com um editor de textos, uma linha de comando para executar scripts, o suporte a *archives* e uma variedade de recursos que tornem mais fáceis tarefas comuns.

## FAR e Total Commander

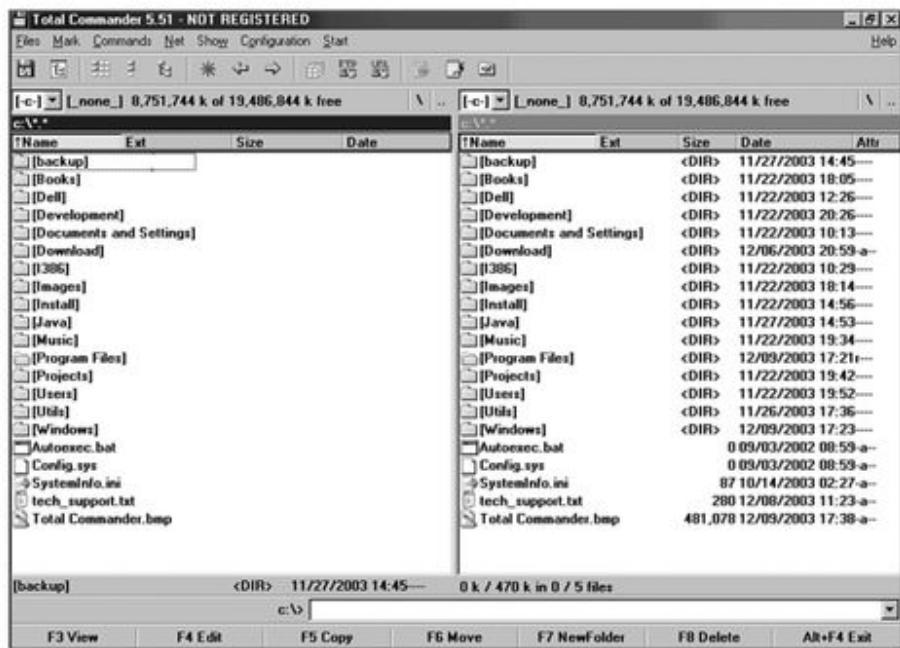
*File e Archive Manager* (FAR) e *Total Commander* são gerenciadores de arquivos avançados para Windows que se originam do DOS e do *Norton Commander*. Distribuídos como *shareware*, podem ser utilizados sem limite de tempo até que você decida registrá-los pagando um valor baixo. Eles incluem recursos para pesquisar arquivos, alterar atributos de arquivos e trabalhar com múltiplos arquivos e diretórios. Têm suporte a redes e FTP: apresentam sites FTP remotos em um painel semelhante a um diretório local. O FAR tem um poderoso editor integrado que pode ser configurado para exibir realce de sintaxe colorido. Os dois ambientes têm conjuntos extensos de atalhos de teclado, e o FAR suporta plug-ins. Ambas as ferramentas suportam navegação em repositórios de arquivos como JAR e ZIP, da mesma forma que se navega por subpastas. Isso torna um software como o WinZip desnecessário e, ainda melhor, o usuário não tem de lidar com diferentes diretórios de trabalho e interfaces com o usuário, como é o caso ao trabalhar com um software não-integrado. A Tabela 1.2 fornece uma lista de recursos e uma comparação lado a lado entre o FAR e o Total Commander.

**TABELA 1.2****FAR versus Total Commander**

RECURSO	FAR	TOTAL COMMANDER
Criar, copiar, visualizar, editar e excluir arquivos e pastas	Excelente	Excelente
Visualizador/editor interno e externo	Excelente	Bom (sem editor interno)
Navegação transparente pelo conteúdo do arquivo compactado (JAR, ZIP e assim por diante)	Excelente	Excelente
Personalização extensa dos recursos e da interface gráfica	Excelente	Bom
Aparência e comportamento como o do Windows	Fraco	Bom
Menus personalizáveis	Excelente	Excelente
Suporte embutido a redes Windows	Sim	Sim
Cliente FTP incluído	Sim	Sim
Atalhos de teclado	Excelente	Bom
Filtros de nomes de arquivos	Excelente	Excelente
Visualização rápida	Razoável	Excelente
Históricos de comandos, pastas, visualizações e edições	Excelente	Bom
Associações de arquivos personalizáveis	Excelente	Bom
Realce de sintaxe	Excelente	Excelente
Utilização de memória	4 MB –14 MB	4 MB – 14 MB
API de plug-ins e disponibilidade de vários plug-ins	Excelente (mais de 500 plug-ins)	Não disponível
Custo do registro de uma cópia	US\$ 25	US\$ 28
<i>Avaliação geral</i>	<i>Excelente</i>	<i>Bom</i>

Embora as duas ferramentas forneçam uma alternativa melhor ao Windows Explorer estendido com outros softwares, o FAR é mais poderoso. Mesmo no seu pacote padrão, ele fornece mais recursos e ganhos de produtividade que o Total Commander. Além disso, com mais de 500 plug-ins escritos por outros desenvolvedores, sua funcionalidade é praticamente ilimitada. A desvantagem do FAR é sua interface com o usuário um pouco desinteressante, embora seja algo com que você possa se acostumar. O Total Commander, mostrado na Figura 1.1, é mais parecido com o Windows Explorer; se você não precisar do máximo de personalização e funcionalidades, ele pode ser uma melhor escolha.

Independentemente das suas preferências, experimente um gerenciador de arquivos integrado, mesmo se você perceber que ele é de difícil de utilização. Com o tempo, valerá a pena!



**FIGURA 1.1** Total Commander.

## IDEs Java

A maioria das técnicas descritas neste livro não envolve uma grande quantidade de codificação e, com uma ferramenta como o FAR, você pode realizar facilmente todas as tarefas exigidas. Entretanto, ambientes de desenvolvimento integrados (IDEs – *Integrated Development Environments*) tornam a codificação muito mais fácil. Assim, esta seção apresenta uma breve visão geral sobre os IDEs importantes e uma recomendação de qual utilizar.

Hoje, quando se trata de IDEs, a questão não é se você deve ou não utilizá-los, mas qual IDE utilizar. Em muito, isso têm a ver com informações sobre o desenvolvimento e preferências pessoais, portanto não vou investir muito tempo discutindo sobre eles. Os dois IDEs gratuitos líderes de mercado são o Eclipse (<http://www.eclipse.org>), promovido pela IBM, e o NetBeans (<http://www.netbeans.org>), promovido pela Sun. Os dois são bons, embora o Eclipse tenha um pouco mais de força e de adeptos. Os melhores IDEs comerciais são o IntelliJ IDEA Borland Jbuilder e o Oracle JDeveloper.

Como você irá trabalhar com codificação e hacking de baixo nível, sua melhor aposta é um IDE flexível com uma pequena ocupação de memória. Meu favorito pessoal é o IDEA, por causa da sua flexibilidade, interface intuitiva e abundância de atalhos e recursos de refatoração. Ele não é gratuito; por isso, se você não puder adquirir uma licença, minha segunda recomendação é o Eclipse.

## Funcionalidade e estrutura da aplicação de exemplo

Em quase todo este livro, vamos trabalhar com a mesma aplicação de exemplo. Ela não é muito sofisticada, mas contém um conjunto básico de componentes encontrados na maioria dos programas Java independentes. Esta seção descreve a aplicação e sua implementação.

A aplicação Chat é uma implementação simples de chat TCP/IP em Java. Ela permite que usuários troquem mensagens instantâneas via rede. A aplicação Chat mantém um histórico da conversa e utiliza cores para diferenciar entre mensagens enviadas e recebidas. Tem uma barra de menus e uma caixa de diálogo About. A aplicação Chat pode ser iniciada utilizando-se o script `chat.bat` no diretório `distrib/bin`. A Figura 1.2 mostra a aplicação Chat em execução.

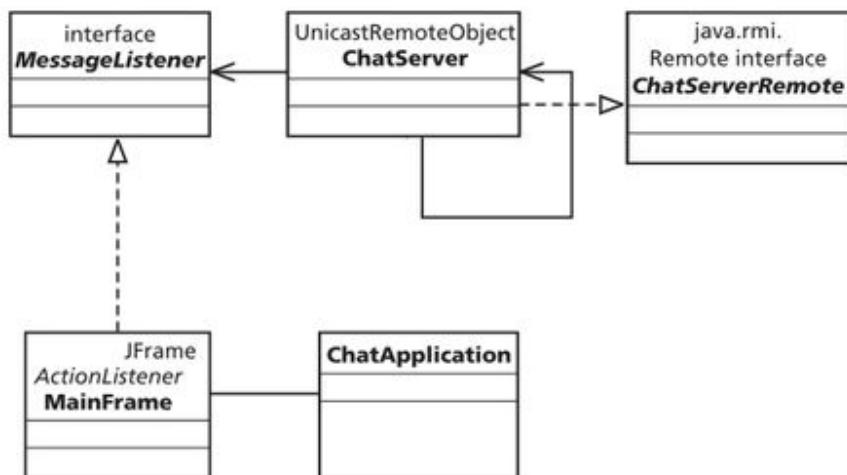


**FIGURA 1.2** A aplicação Chat.

A aplicação Chat é implementada utilizando-se Swing para a interface com o usuário e RMI para comunicação de rede. Quando em execução, cada instância da aplicação Chat cria um registro RMI *in-process*, que é utilizado por outras instâncias para enviar mensagens para o usuário. É necessário que os usuários insiram o hostname do usuário para o qual querem postar uma mensagem. Quando o usuário envia uma mensagem, a aplicação Chat pesquisa o objeto do servidor remoto e chama um método nele. Para fins de testes, mensagens podem ser enviadas a "localhost". Nesse caso a mesma mensa-

gem é adicionada à conversa da maneira como foi enviada e recebida.

O diagrama de classes UML para o Chat é mostrado na Figura 1.3.



**FIGURA 1.3** Um diagrama de classes para Chat.

A estrutura do diretório da aplicação Chat segue os padrões *de facto* para desenvolvimento de aplicações em Java. A pasta “home” para os diretórios da aplicação é chamada *CovertJava*. Os subdiretórios nela contidos estão listados na Tabela 1.3.

TABELA 1.3

**A estrutura do diretório da aplicação Chat**

NOME DO DIRETÓRIO	DESCRIÇÃO
bin	Contém os scripts de desenvolvimento e de testes
build	Contém o build.xml do Ant e outros arquivos relacionados ao build
classes	Diretório da saída do compilador para arquivos .class
distrib	Contém a aplicação na forma que será distribuída
distrib\bin	Contém scripts que executam a aplicação
distrib\conf	Contém arquivos de configuração como arquivos de políticas de segurança Java
distrib\lib	Contém bibliotecas utilizadas para executar a aplicação
distrib\patches	Contém patches para classes
lib	Contém bibliotecas utilizadas para criar a aplicação
src	Contém os arquivos-fonte da aplicação

Uma aplicação Chat pode ser criada pelo Ant através do build.xml no diretório criado.

## Questionário rápido

1. Quais técnicas podem ser utilizadas para aprender a implementação interna de uma aplicação?
2. Quais técnicas podem ser utilizadas para alterar a implementação interna de uma aplicação?
3. Quais técnicas podem ser utilizadas para capturar a comunicação entre um cliente e um servidor?
4. Quais aplicações Windows são substituídas pelo FAR e Total Commander?

## Resumo

- Várias técnicas apresentadas neste livro podem ser utilizadas para aprender sobre o funcionamento interno de implementações, para hackear uma aplicação ou para trabalhar com o JDK em nível de sistema.
- Aplicações de gerenciamento de arquivos integradas aumentam a produtividade e compensam o investimento.

# Descompilando classes

*"Quando tudo mais falhar, leia o manual."*

As Leis de Murphy aplicadas à tecnologia

## Determinando quando descompilar

Em um mundo ideal, a descompilação provavelmente seria desnecessária, exceto para aprender como outras pessoas que não gostam de produzir documentação de boa qualidade implementaram um certo recurso. No dia-a-dia, porém, freqüentemente há situações em que uma referência direta ao código-fonte pode ser a melhor, se não a única, solução. Eis algumas razões para descompilar:

- Recuperar código-fonte accidentalmente perdido
- Aprender sobre a implementação de um recurso ou truque
- Solucionar problemas de uma aplicação ou biblioteca sem boa documentação
- Corrigir bugs urgentes em códigos de terceiros para os quais não há código-fonte
- Aprender a proteger seu código contra hacking

A descompilação produz o código-fonte a partir do bytecode Java. Ela é um processo inverso à compilação, que é possível devido à estrutura bem-documentada e padronizada do bytecode. Assim como executar um compilador produz o bytecode a partir do código-fonte, você pode executar um descompilador para obter o código-fonte para um dado bytecode. A descompilação é um método poderoso para aprender a lógica de implementação na ausência da documentação e do código-fonte, razão pela qual muitos fornecedores de produtos proíbem explicitamente a descompila-

# 2

## NESTE CAPÍTULO

- ▶ Determinando quando descompilar 9
- ▶ Conhecendo os melhores descompiladores 10
- ▶ Descompilando uma classe 12
- ▶ O que torna a descompilação possível? 17
- ▶ Problemas potenciais com código descompilado 18
- ▶ Questionário rápido 21
- ▶ Resumo 21

ção e a engenharia reversa no acordo de licenciamento. Certifique-se de verificar o acordo de licenciamento ou obtenha permissão explícita do fornecedor, se não tiver certeza sobre a legalidade das suas ações.

Alguns podem argumentar que você não deveria recorrer a medidas extremas como a descompilação, e que deve contar com os fornecedores do bytecode para o suporte e a correção de bugs. Em um ambiente profissional, se você for o desenvolvedor de uma aplicação, será responsável pela sua funcionalidade ser sem falhas. Os usuários e a gerência não se importam se um bug está no seu código ou no código de terceiros. Eles querem que o problema seja corrigido, e o considerarão responsável por isso. Contatar o fornecedor do código de terceiros deveria ser uma maneira preferida. Entretanto, em casos urgentes, quando você precisa fornecer uma solução em questão de horas, ser capaz de trabalhar com o bytecode lhe trará vantagem adicional sobre os seus colegas, e talvez também um bônus.

## Conhecendo os melhores descompiladores

Para embarcar na tarefa da descompilação, você precisa das ferramentas certas. Um bom descompilador pode produzir código-fonte que será quase tão bom quanto o código original. Alguns descompiladores são gratuitos, e outros estão disponíveis comercialmente. Embora eu apóie os princípios por trás dos softwares comerciais, eles precisam oferecer um diferencial útil em relação aos seus concorrentes gratuitos para que eu os utilize. No caso

### HISTÓRIAS DAS TRINCHEIRAS

No Riggs Bank, estávamos nos preparando para colocar on-line uma aplicação J2EE imensa e muito importante, que seria implantada em um cluster de servidores de aplicações de um fornecedor J2EE líder de mercado. Várias equipes esperavam que o ambiente de produção estivesse pronto, mas por alguma razão estranha o servidor de aplicações não inicializava em alguns hosts. A mesma instalação funcionava em algumas máquinas, mas apresentava falhas em outras, com uma mensagem de erro indicando uma URL de configuração inválida. Para piorar as coisas, a URL na mensagem de erro não podia ser encontrada em nenhum arquivo de configuração, script de shell ou variável de ambiente.

Vários dias foram gastos em vão tentando-se corrigir o problema, e a situação estava prestes a explodir porque várias equipes estavam prestes a estourar um prazo crítico. Depois de copiar e reinstalar o servidor de aplicações com problemas, finalmente recorremos à localização da classe nas bibliotecas do servidor de aplicações que gerava a mensagem de erro. Descompilando essa classe e algumas outras que a utilizavam, descobrimos que a URL foi gerada automaticamente com base no diretório de instalação do servidor. O diretório de instalação havia sido determinado executando-se o comando `pwd` do Unix. Revelou-se que nos hosts com problemas não havia nenhuma permissão para executar o `pwd`, mas a mensagem de erro enganosa não tornou isso óbvio. Corrigir as permissões levou alguns minutos, e o processo inteiro, a partir do momento em que encontramos e descompilamos a classe, durou menos de uma hora. Assim, um desastre prestes a acontecer transformou-se em uma grande vitória para a equipe de TI.

dos descompiladores, não achei que faltassem recursos aos descompiladores gratuitos. Portanto minha recomendação pessoal é utilizar uma ferramenta gratuita como o JAD ou JODE. A Tabela 2.1 lista alguns descompiladores comumente utilizados, e inclui uma descrição curta destacando as qualidades de cada um. Os URLs apresentados talvez estejam desatualizados, portanto fazer uma pesquisa no Google é, em geral, a melhor maneira de encontrar a home page do descompilador e a versão mais recente para download.

Um critério muito importante é a maneira com que o descompilador suporta construções mais avançadas da linguagem, tais como classes internas e implementações anônimas. Mesmo que o formato do bytecode tenha sido muito estável desde o JDK 1.1, é importante utilizar um descompilador freqüentemente atualizado pelos seus autores. Os novos recursos da linguagem no JDK 1.5 exigirão uma atualização nos descompiladores, portanto certifique-se de verificar a data da distribuição da versão que você está utilizando.

**TABELA 2.1**

**Descompiladores**

FERRAMENTA/AVALIAÇÃO	LICENÇA	DESCRIÇÃO
JAD/excelente	Gratuito para uso não-comercial	O JAD é um descompilador sofisticado, confiável e muito rápido. Ele tem suporte completo a classes internas, implementações anônimas e outros recursos avançados da linguagem. O código gerado é limpo e os <i>imports</i> são bem organizados. Vários outros ambientes de descompilação utilizam a linha de comando do JAD como o mecanismo de descompilação.
JODE/excelente	Licença pública GNU	O JODE é um descompilador muito bom, escrito em Java com código-fonte completo disponível no SourceForge.net. É possível que não seja tão rápido e disseminado quanto o JAD, mas produz resultados excelentes, em alguns momentos mais claros do que o JAD. Ter o código-fonte para o próprio descompilador é fundamental para propósitos educacionais.
Mocha/razoável	Gratuito	O Mocha é o primeiro descompilador bem conhecido que gerou várias controvérsias legais, mas também uma onda de entusiasmo. O Mocha tornou óbvio o fato de que o código Java pode ser reconstruído quase na sua forma original, o que foi aplaudido pela comunidade de desenvolvimento, mas repudiado por departamentos jurídicos. O código público não foi atualizado desde 1996, embora a Borland o tenha aparentemente atualizado e integrado ao JBuilder.

Embora seja possível encontrar outros descompiladores no mercado, o JAD e o JODE certamente são suficientemente bons e, portanto, amplamente utilizados. Muitos produtos fornecem interfaces gráficas (GUIs – *Graphical User Interfaces*), porém contam com um descompilador para fazer o trabalho real. Por exemplo, Decafe, DJ e Cavaj são ferramentas gráficas que acompanham o JAD e que, portanto, não foram incluídas na revisão. No restante deste livro, iremos utilizar o JAD em linha de comando para produzir código-fonte. Na maior parte do tempo, o descompilador de linha de comando é tudo que você precisa, mas, se preferir utilizar uma GUI, certifique-se de que ela utiliza um bom descompilador como o JAD ou o JODE.

## Descompilando uma classe

Caso você ainda não tenha utilizado um descompilador, vejamos a qualidade do trabalho dessas ferramentas. Trabalharemos com uma versão ligeiramente melhorada da classe `MessageInfo`, utilizada pela aplicação Chat para enviar o texto da mensagem e atributos a um host remoto. `MessageInfoComplex.java`, mostrada na Listagem 2.1, tem uma classe interna anônima (`MessageInfoPK`) e um método `main()`, para ilustrar alguns casos mais complexos de descompilação.

---

### LISTAGEM 2.1 Código-fonte da `MessageInfoComplex`

---

```
package covertjava.decompile;
/**
 * MessageInfo é utilizada para enviar informações adicionais com cada mensagem na
 * rede. Ela contém atualmente o nome do host em que a mensagem
 * se originou e o nome do usuário que a enviou.
 */
public class MessageInfoComplex implements java.io.Serializable {

    String hostName;
    String userName;

    public MessageInfoComplex(String hostName, String userName) {
        this.hostName = hostName;
        this.userName = userName;
    }

    /**
     * @return nome do host que enviou a mensagem
     */
    public String getHostName() {
        return hostName;
    }
}
```

**LISTAGEM 2.1** Continuação

---

```
/*
 * @return nome do usuário que enviou a mensagem
 */
public String getUserName( ) {
    return userName;
}

/**
 * Método de conveniência para obter uma string que melhor identifique o usuário.
 * @return nome utilizado para identificar o usuário que enviou essa mensagem
 */
public String getDisplayName( ) {
    return getUserName( ) + " (" + getHostName( ) + ")";
}

/**
 * Gera o id da mensagem que pode ser utilizado para identificar essa mensagem
 * em um banco de dados
 * O formato é: <ID><UserName><HostName>. Nomes são limitados a 8 caracteres de
 * extensão
 * Exemplo: 443651_Kalinovs_JAMAICA seria gerado para Kalinovsky/JAMAICA
 */
public String generateMessageId( ) {
    StringBuffer id = new StringBuffer(22);

    String systemTime = "" + System.currentTimeMillis( );
    id.append(systemTime.substring(0, 6));

    if (getUserName( ) != null && getUserName( ).length( ) > 0) {
        // Adiciona o nome do usuário, se especificado
        id.append('_');
        int maxChars = Math.min(getUserName( ).length( ), 8);
        id.append(getUserName( ).substring(0, maxChars));
    }

    if (getHostName( ) != null && getHostName( ).length( ) > 0) {
        // Adiciona o nome do host, se especificado
        id.append('_');
        int maxChars = Math.min(getHostName( ).length( ), 7) ;
        id.append(getHostName( ).substring(0, maxChars));
    }
}
```

**LISTAGEM 2.1** Continuação

---

```
        return id.toString( );
    }

    /**
     * Inclui um exemplo de uma classe interna anônima
     */
    public static void main(String[ ] args) {
        new Thread(new Runnable( ) {
            public void run( ) {
                System.out.println("Running test");
                MessageInfoComplex info = new MessageInfoComplex("JAMAICA", "Kalinovsky");
                System.out.println("Message id = " + info.generateMessageId( ));
                info = new MessageInfoComplex(null, "JAMAICA");
                System.out.println("Message id = " + info.generateMessageId( ));
            }
        }).start( );
    }

    /**
     * Classe interna que pode ser utilizada como uma chave primária para
     * MessageInfoComplex
     */
    public static class MessageInfoPK implements java.io.Serializable {
        public String id;
    }
}
```

---

Depois de compilar `MessageInfoComplex.java` utilizando `javac` com as opções padrão, obtemos três arquivos de classes: `MessageInfoComplex.class`, `MessageInfoComplex$MessageInfoPK.class` e `MessageInfoComplex$1.class`. Como você talvez saiba, classes internas e classes anônimas foram adicionadas ao Java no JDK 1.1. Mas o objetivo era preservar a compatibilidade do formato do bytecode com versões anteriores do Java. É por isso que essas construções da linguagem resultam em classes bastante independentes, embora retenham a associação com a classe principal. O passo final do nosso teste é executar o descompilador sobre o arquivo de classe e depois comparar o código gerado com o original. Supondo que você fez o download e instalou o JAD e o adicionou ao *path*, poderá executá-lo utilizando o seguinte comando:

```
jad MessageInfoComplex.class
```

Concluída a execução, o JAD gera o arquivo `MessageInfoComplex.jad`. Este é renomeado para `MessageInfoComplex_FullDebug.jad`, conforme mostrado na Listagem 2.2.

#### **LISTAGEM 2.2** Código descompilado de `MessageInfoComplex`

---

```
// Descompilado pelo Jad v1.5.7g. direitos autorais 2000 de Pavel Kouznetsov.  
// Home page do JAD: http://www.geocities.com/SiliconValley/Bridge/8617/jad.html  
// Opções de descompilador: packimports(3)  
// Nome do arquivo-fonte: MessageInfoComplex.java  
  
package covertjava.decompile;  
  
import java.io.PrintStream;  
import java.io.Serializable;  
  
public class MessageInfoComplex  
    implements Serializable  
{  
    public static class MessageInfoPK  
        implements Serializable  
    {  
  
        public String id;  
  
        public MessageInfoPK( )  
        {  
        }  
    }  
  
    public MessageInfoComplex(String hostName, String userName)  
    {  
        this.hostName = hostName;  
        this.userName = userName;  
    }  
  
    public String getHostName( )  
    {  
        return hostName;  
    }  
  
    public String getUserName( )  
    {  
        return userName;  
    }  
}
```

**LISTAGEM 2.2** Continuação

```
public String getDisplayName( )
{
    return getUserName( ) + " (" + getHostName( ) + ")";
}

public String generateMessageId( )
{
    StringBuffer id = new StringBuffer(22) ;
    String systemTime = "" + System.currentTimeMillis( );
    id.append(systemTime.substring(0, 6));
    if(getUserName( ) != null && getUserName( ).length( ) > 0)
    {
        id.append('_');
        int maxChars = Math.min(getUserName( ).length( ), 8);
        id.append(getUserName( ).substring(0, maxChars));
    }
    if(getHostName( ) != null && getHostName( ).length( ) > 0)
    {
        id.append('_');
        int maxChars = Math.min(getHostName( ).length( ), 7);
        id.append(getHostName( ).substring(0, maxChars));
    }
    return id.toString( );
}

public static void main(String args[ ])
{
    (new Thread(new Runnable( ) {

        public void run( )
        {
            System.out.println("Running test");
            MessageInfoComplex info = new MessageInfoComplex("JAMAICA", "Kalinovsky");
            System.out.println("Message id = " + info.generateMessageId( ));
            info = new MessageInfoComplex(null, "JAMAICA");
            System.out.println("Message id = " + info.generateMessageId( ));
        }
    })).start( );
}

String hostName;
String userName;
```

---

Pare por alguns momentos para revisar o código gerado. Como você pode ver, o código é praticamente idêntico ao original! A ordem e a formatação das variáveis, métodos e declarações de classe interna são diferentes, mas a lógica é exatamente a mesma. Também perdemos os comentários, mas um código Java bem escrito como o nosso é auto-explicativo, não é?

Nosso caso produziu bons resultados porque todas as informações sobre a depuração são incluídas pelo javac quando é utilizada a opção `-g`. Se o código fosse compilado sem as informações sobre a depuração (usando a opção `-g:none`), o código descompilado perderia parte da clareza. Seriam perdidos, por exemplo, nomes de parâmetros de métodos e de variáveis locais. O código a seguir mostra o construtor e um método que utiliza variáveis locais para `MessageInfoComplex` sem informações de depuração incluídas:

```
public MessageInfoComplex(String s, String s1)
{
    hostName = s;
    userName = s1;
}

public String generateMessageId( )
{
    StringBuffer stringbuffer = new StringBuffer(22);
    String s = "" + System.currentTimeMillis( );
    stringbuffer.append(s.substring(0, 6));
    if(getUserName( ) != null && getUserName( ).length( ) > 0)
    {
        stringbuffer.append('_');
        int i = Math.min(getUserName( ).length( ), 8);
        stringbuffer.append(getUserName( ).substring(0, i));
    }
    if(getHostName( ) != null && getHostName( ).length( ) > 0)
    {
        stringbuffer.append('_');
        int j = Math.min(getHostName( ).length( ), 7);
        stringbuffer.append(getHostName( ).substring(0, j));
    }
    return stringbuffer.toString( );
}
```

## O que torna a descompilação possível?

Código-fonte Java não é compilado para o código de máquina binários como ocorre com código-fonte C/C++. Compilar o código-fonte Java produz bytecode intermediário, que é uma representação do código-fonte independente de plataforma. O bytecode pode ser interpretado ou compilado depois do carregamento, o que resulta em uma transformação de dois passos, da linguagem de programação de alto nível no código de máquina de

baixo nível. É esse passo intermediário que torna a descompilação do bytecode Java quase perfeita. O bytecode inclui todas as informações significativas encontradas em um arquivo-fonte. Mesmo que os comentários e a formatação sejam perdidos, obviamente, todos os métodos, variáveis e a lógica de programação são preservados. Como o bytecode não representa a linguagem de máquina de mais baixo nível, o formato do código assemelha-se ao do código-fonte. A especificação da JVM define um conjunto de instruções que correspondem a operadores e palavras-chave da linguagem Java. Assim, um fragmento de código Java como

```
public String getDisplayName( ) {  
    return getUserName( ) + " (" + getHostName( ) + ")";  
}
```

é representado pelo seguinte bytecode:

```
0 new #4 <java/lang/StringBuffer>  
3 dup  
4 aload_0  
5 invokevirtual #5 <covertjava/decompile/MessageInfoComplex.getUserName>  
8 invokestatic #6 <java/lang/String.valueOf>  
11 invokestatic #6 <java/lang/String.valueOf>  
14 invokespecial #7 <java/lang/StringBuffer.<init>>  
17 ldc #8 <(>  
19 invokevirtual #9 <java/lang/StringBuffer.append>  
22 aload_0  
23 invokevirtual #10 <covertjava/decompile/MessageInfoComplex.getHostName>  
26 invokevirtual #9 <java/lang/StringBuffer.append>  
29 ldc #11 <)>  
31 invokevirtual #9 <java/lang/StringBuffer.append>  
34 invokestatic #6 <java/lang/String.valueOf>  
37 invokestatic #6 <java/lang/String.valueOf>  
40 areturn
```

O formato do bytecode é explorado em detalhes no Capítulo 17, mas você pode notar a semelhança simplesmente examinando o bytecode. O descompilador carrega o bytecode e tenta reconstruir o código-fonte com base nas instruções do bytecode. Em geral, os nomes dos métodos e variáveis da classe são preservados, enquanto os nomes dos parâmetros do método e de variáveis locais são perdidos. Se as informações de depuração estiverem disponíveis, elas fornecerão ao descompilador os nomes dos parâmetros e os números das linhas – isso torna o arquivo-fonte reconstruído ainda mais legível.

## Problemas potenciais com código descompilado

Na maior parte das vezes, a descompilação produz um arquivo legível que pode ser alterado e recompilado. Entretanto, em algumas ocasiões, a descompilação não gera um ar-

quivo que possa ser recompilado. Isso pode acontecer se o bytecode foi ofuscado e os nomes atribuídos pelo ofuscador resultarem em uma ambigüidade na compilação. O bytecode é verificado quando carregado, mas as verificações pressupõem que o compilador tenha checado alguns erros. Dessa forma, os verificadores de bytecode não são tão rigorosos, e compiladores e ofuscadores podem tirar proveito disso para melhor proteger a propriedade intelectual. Por exemplo, esta é a saída do JAD na classe interna anônima do método `MessageInfoComplex main()` que foi ofuscado pelo Zelix ClassMaster:

```
static class c
    implements Runnable
{
    public void run( )
    {
        boolean flag = a.b;
        System.out.println(a("*4%p\002\026&j\016\0135"));
        b b1 = new b(a("2\000\006_\";\\0"), a("3 'w\\005\\02778u\\022"));
        System.out.println(a("5$8m\\n\\037$kw\\017X|k").concat(String.valueOf
➥(String.valueOf(b1.d( )))));
        b1 = new b(null, a("2\000\006_\";\\0"));
        System.out.println(a("5$8m\\n\\037$kw\\017X|k").concat(String.valueOf
➥(String.valueOf(b1.d( )))));
        if(flag)
            b.c = !b.c;
    }

    private static String a(String s)
    {
        char ac[ ];
        int i;
        int j;
        ac = s.toCharArray( );
        i = ac.length;
        j = 0 ;
        if(i > 1) goto _L2; else goto _L1
_L1:
    ac;
    j;
_L10:
    JVM INSTR dup2 ;
    JVM INSTR caload ;
    j % 5;
    JVM INSTR tableswitch 0 3: default 72
//          0 52
//          1 57
```

```
//          2 62
//          3 67;
    goto _L3 _L4 _L5 _L6 _L7
_L4:
    0x78;
    goto _L8
_L5:
    65;
    goto _L8
_L6:
    75;
    goto _L8
_L7:
    30;
    goto _L8
_L3:
    107;
_L8:
    JVM INSTR ixor ;
    (char);
    JVM INSTR castore ;
    j++;
    if(i != 0) goto _L2; else goto _L9
_L9:
    ac;
    i;
    goto _L10
_L2:
    if(j >= i)
        return new String(ac);
    if(true) goto _L1; else goto _L11
_L11:
}
}
```

Como se vê, ela é um grande fracasso, nem de longe se assemelhando ao código-fonte Java. E o que é mais perturbador: o JAD produziu um código-fonte que não pode ser compilado. Os outros dois descompiladores reportaram um erro no arquivo de classe. O JVM, é claro, reconhece e carrega sem problemas o bytecode em questão. O ofuscamento é estudado em detalhes no Capítulo 3.

Uma maneira eficiente de proteger a propriedade intelectual é codificar os arquivos de classes e aplicar um *class loader* personalizado para decodificá-los no carregamento. Dessa maneira, os descompiladores não podem ser utilizados em qualquer uma das classes da aplicação, exceto para o ponto de entrada e para o *class loader*. Embora não seja in-

quebrável, a codificação torna o hacking muito mais difícil. Um hacker primeiro teria de descompilar o *class loader* para entender o mecanismo de decodificação e então decodificar todos os arquivos de classes; somente então poderia ele prosseguir à descompilação. O Capítulo 19 fornece as informações sobre como melhor proteger a propriedade intelectual em aplicações Java.

## Questionário rápido

1. Quais são as razões para descompilar bytecode?
2. Quais e como opções do compilador afetam a qualidade da descompilação?
3. Por que o bytecode em Java descompilado é quase idêntico ao código-fonte?
4. Como você pode proteger o bytecode contra descompilação?

## Resumo

- A descompilação produz o código-fonte a partir do bytecode, que é quase idêntico ao original.
- A descompilação é um método poderoso de aprender a lógica da implementação na ausência da documentação e do código-fonte. Entretanto, a descompilação e a engenharia reversa talvez sejam explicitamente proibidas no acordo de licenciamento.
- A descompilação requer o download e a instalação de um descompilador.
- Descompilar classes Java é eficaz porque o bytecode é um passo intermediário entre o código-fonte e o código de máquina.
- Um bom ofuscador pode tornar o código descompilado muito difícil de se ler e entender.

# 3

## *NESTE CAPÍTULO*

- ▶ Protegendo as idéias por trás do seu código 22
- ▶ Ofuscamento como uma forma de proteção de propriedade intelectual 23
- ▶ Transformações realizadas por ofuscadores 24
- ▶ Conhecendo os melhores ofuscadores 28
- ▶ Problemas potenciais e soluções comuns 29
- ▶ Utilizando o Zelix KlassMaster para ofuscar uma aplicação de chat 32
- ▶ Quebrando código ofuscado 36
- ▶ Questionário rápido 36
- ▶ Resumo 37

# Ofuscando classes

*"Qualquer tecnologia suficientemente avançada é indistinguível da mágica."*

As Leis de Murphy aplicadas à tecnologia

## **Protegendo as idéias por trás do seu código**

Engenharia reversa e hacking existem desde os primeiros dias do desenvolvimento de software. Na verdade, roubar ou reproduzir as idéias de outras pessoas sempre foi a maneira mais fácil de criar produtos competitivos. Há, naturalmente, um método perfeitamente aceitável de criar baseando-se em descobertas anteriores de outras pessoas. Contanto que essas pessoas não se importem, o método funciona bem. Entretanto, a maioria dos inventores e pesquisadores prefere receber o crédito e, possivelmente, uma recompensa financeira pelo seu trabalho. Em termos mais simples, eles também têm contas a pagar e férias a tirar.

Uma boa maneira de proteger a propriedade intelectual é o autor obter os direitos autorais e patentes sobre os recursos exclusivos de seus produtos. Isso certamente é recomendável para invenções e descobertas importantes que exijam altos investimentos em pesquisa e desenvolvimento. Atribuir direitos autorais a um software é um processo bastante fácil e econômico, mas ele protege somente o código “original” da aplicação, não as idéias por trás dele. Outros não seriam capazes de se apoderar do código com direitos autorais e utilizá-lo nas suas aplicações sem a permissão do autor, mas se eles tiverem suas próprias implementações do mesmo recurso, utilizá-las não seria considerado uma violação. As patentes fornecem uma proteção muito melhor, pois abrangem as idéias e os algorit-

mos, em vez de uma implementação específica, mas o seu registro é caro e pode-se levar anos para consegui-lo.

O risco de a sua aplicação ser hackeada é real? Se você tiver boas idéias, certamente. No momento em que escrevia este livro, a maioria dos casos famosos de engenharia reversa não ocorria com produtos Java, mas aqui está um trecho de um fornecedor Java (DataDirect Technologies):

ROCKVILLE, MD., July 1, 2002 – DataDirect Technologies, Inc., um importante fornecedor da área de conectividade de dados processou a i-net Software GmbH alegando infração de direitos autorais e quebra de contrato. A DataDirect Technologies busca tanto reparos críticos preliminares como permanentes, para impedir que a i-net empregue outros esforços para disponibilizar e vender produtos que, a DataDirect Technologies acredita, tenham sido obtidos por meio de reengenharia reversa de seus produtos.

A DataDirect Technologies alega que um concorrente fez engenharia reversa de seu produto. Mesmo assim, até hoje seu produto só apresenta uma proteção mínima contra a descompilação.

Na vida real, atribuir direitos autorais ao código e patentear técnicas pode não fornecer a proteção adequada, se um concorrente ou hacker puder aprender facilmente a implementação a partir do código-fonte. As questões de proteção legal são discutidas em um capítulo específico, mas, por enquanto, vamos nos concentrar em maneiras de proteger a propriedade intelectual em aplicações Java.

## Ofuscamento como uma forma de proteção de propriedade intelectual

*Ofuscamento* é o processo de transformar o bytecode em uma forma menos legível por humanos, com o propósito de dificultar a engenharia reversa. Em geral, esse processo consiste, basicamente, em remover todas as informações de depuração, tais como tabelas de variáveis e números de linhas, e renomear pacotes, classes e métodos com nomes gerados automaticamente. Ofuscadores avançados vão mais longe, alterando o fluxo de controle do código Java, reestruturando a lógica existente e inserindo código falso que não funciona. Uma premissa do ofuscamento é que as transformações não comprometem a validade do bytecode e não alteram a funcionalidade exposta.

O ofuscamento é possível pelas mesmas razões que é possível a descompilação: o bytecode Java é padronizado e bem documentado. Ofuscadores carregam arquivos de classes Java, analisam seus formatos e então aplicam transformações com base nos recursos suportados. Quando todas as transformações tiverem sido aplicadas, o bytecode é salvo como um novo arquivo de classe. Esse novo arquivo tem uma estrutura interna diferente, mas comporta-se exatamente como o arquivo original.

Ofuscadores são especialmente necessários para produtos e tecnologias em que a lógica de implementação está disponível para o usuário. Esses são os casos de páginas HTML e JavaScript, que são distribuídas como código-fonte. Aplicações Java não apresentam comportamento muito melhor, pois, mesmo que geralmente sejam distribuídas

em bytecode binário, utilizar um descompilador como descrito no capítulo anterior possibilita produzir código-fonte muito próximo do original.

## Transformações realizadas por ofuscadores

Não existe um padrão para o ofuscamento, portanto o nível de proteção varia com base na qualidade do ofuscador. As seções a seguir apresentam alguns recursos comumente encontrados nos ofuscadores. Utilizaremos o método `sendMessage` de `ChatServer` para ilustrar como cada transformação afeta o código descompilado. O código-fonte original para `sendMessage` é mostrado na Listagem 3.1.

### **LISTAGEM 3.1** Código-fonte original de `sendMessage`

```
public void sendMessage(String host, String message) throws Exception {
    if (host == null || host.trim( ).length( ) == 0)
        throw new Exception ("Please specify host name");

    System.out.println("Sending message to host " + host + ":" + message);
    String url = "//" + host + ":" + this.registryPort + "/chatserver";
    ChatServerRemote remoteServer = (ChatServerRemote)Naming.lookup(url);

    MessageInfo messageInfo = new MessageInfo(this.hostName, this.userName);
    remoteServer.receiveMessage(message, messageInfo);
    System.out.println("Message sent to host " + host);
}
```

## Removendo informações de depuração

O bytecode Java pode conter informações inseridas pelo compilador que ajudam a depurar o código em execução. As informações inseridas pelo `javac` podem conter algumas ou todas as seguintes informações: números de linhas, nomes de variáveis e nomes de arquivos-fonte. As informações de depuração não são necessárias para executar a classe, mas são utilizadas pelos depuradores para associar o bytecode ao código-fonte. Os descompiladores utilizam essas informações para melhor reconstruir o código-fonte. Com todas as informações de depuração no arquivo de classe, o código descompilado é quase idêntico ao original. Quando as informações de depuração são removidas, perdem-se os nomes armazenados, de modo que os descompiladores têm de gerar seus próprios nomes. No nosso caso, depois da remoção, os nomes do parâmetro `sendMessage` apareceriam como `s1` e `s2` em vez de `host` e `message`.

## Desfiguração de nomes

Os desenvolvedores, claro, utilizam nomes significativos para pacotes, classes e métodos. A implementação do servidor da nossa aplicação de exemplo Chat é chamada `Chat-`

Server, e o método que envia uma mensagem a outro usuário é chamado sendMessage. Bons nomes são cruciais no desenvolvimento e na manutenção, mas eles não significam nada para a JVM. O Java Runtime Environment (JRE) não se importa se sendMessage tem o nome goShopping ou abcdefg; ele simplesmente o invoca e o executa. Ao substituir nomes significativos e inteligíveis para humanos por nomes sem sentido gerados pelo computador, os ofuscadores tornam a tarefa de entender o código descompilado muito mais difícil. O que era Chatserver.sendMessage transforma-se em d.a; e se houver muitas classes e métodos com os mesmos nomes, o código descompilado torna-se extremamente difícil de entender. Um bom ofuscador tira proveito da sobrecarga de métodos (*overloading*) para tornar as coisas ainda mais difíceis. Três métodos com nomes e assinaturas diferentes, realizando tarefas diferentes no código original podem ser renomeados com um mesmo nome no código ofuscado. Como as assinaturas desses métodos são diferentes, o código resultante não viola a especificação da linguagem Java, mas a mudança torna mais confuso o código descompilado. A Listagem 3.2 mostra um exemplo de sendMessage descompilado depois de um ofuscamento que removeu as informações de depuração e realizou a desfiguração de nomes.

---

**LISTAGEM 3.2** Método sendMessage descompilado depois da desfiguração de nomes

---

```
public void a(String s, String s1)
    throws Exception
{
    if(s == null || s.trim( ).length( ) == 0)
    {
        throw new Exception("Please specify host name");
    } else
    {
        System.out.println(String.valueOf(String.valueOf((
            new StringBuffer("Sending message to host ")
            .append(s).append(": ").append(s1))));

        String s2 = String.valueOf(String.valueOf((
            new StringBuffer("//").append(s).append(":")
            .append(b).append("/chatserver"))));
        b b1 = (b)Naming.lookup(s2);
        MessageInfo messageinfo = new MessageInfo(e, f);
        b1.receiveMessage(s1, messageinfo);
        System.out.println("Message sent to host ".concat(
            String.valueOf(String.valueOf(s))));
        return;
    }
}
```

---

## Codificando strings Java

Strings em Java são armazenadas como texto simples dentro do bytecode. A maioria das aplicações bem escritas incluem dados de rastreamento no código, que produzem logs de execução para depuração e auditoria. Mesmo que os nomes de classes e de métodos sejam alterados, as strings gravadas por métodos em um arquivo de log ou no console podem ir contra o objetivo do método. No nosso caso, `ChatServer.sendMessage` gera a saída de uma mensagem de rastreamento utilizando o seguinte:

```
System.out.println("Sending message to host " + host + ": " + message);
```

Mesmo que `ChatServer.sendMessage` seja renomeado para `d.a`, ao vermos um rastreamento como esse no corpo da mensagem descompilada, fica claro o que o método faz. Entretanto, se a string for codificada em bytecode, a versão descompilada da classe será parecida com o seguinte:

```
System.out.println(String.valueOf(String.valueOf((new  
StringBuffer(a("A\025wV6|\0279_:_a\003xU:2\004v\0227)\003m\022"))  
).append(s).append(a("P")).append(s1))));
```

Examinando de perto a string codificada: primeiro ela é passada para o método `a( )`, que a decodifica e retorna a string legível ao método `System.out.println( )`. A codificação de strings é um recurso poderoso que deve ser fornecido por um ofuscador de qualidade profissional.

## Alterando o fluxo de controle

As transformações apresentadas anteriormente tornam a engenharia reversa do código ofuscado mais difícil, mas não alteram a estrutura fundamental do código Java. Elas também nada fazem para proteger os algoritmos e o fluxo de controle do programa, que normalmente constitui a parte mais importante das inovações. A versão descompilada de `ChatServer.sendMessage` mostrada anteriormente continua razoavelmente compreensível. Você pode ver que o código primeiro verifica a entrada válida e emite uma exceção se ocorrer um erro. Pesquisa-se então o objeto no servidor remoto e aplica-se um método nele.

Os melhores ofuscadores são capazes de transformar o fluxo de execução do bytecode, inserindo condicionais e instruções `goto` falsas. Esse processo pode desacelerar um pouco a execução, mas isso talvez seja um custo baixo a ser pago por uma proteção maior da propriedade intelectual. A Listagem 3.3 mostra no que `sendMessage` se tornou depois de todas as transformações discutidas terem sido aplicadas.

---

### LISTAGEM 3.3 `sendMessage` descompilado depois de todas as transformações

---

```
public void a(String s, String s1)  
    throws Exception  
{  
    boolean flag = MessageInfo.c;
```

**LISTAGEM 3.3** Continuação

```

    s;
    if(flag) goto _L2; else goto _L1
_L1:
    JVM INSTR ifnull 29;
    goto _L3 _L4
_L3:
    s.trim( );
_L2:
    if(flag) goto _L6; else goto _L5
_L5:
    length( );
    JVM INSTR ifne 42;
    goto _L4 _L7
_L4:
    throw new Exception(a("\002)qUe7egDs1,rM6:*g06<$yQ"));
_L7:
    System.out.println(String.valueOf(String.valueOf((
        new StringBuffer(a("\001 zP\177<"4Ys!6uSsr1{\024-=6`\024"))
        .append(s).append(a("he")).append(s1))));

    String.valueOf(String.valueOf(
        (new StringBuffer(a("}j"))).append(s).append(":")
        .append(b).append(a("}{|Ub! fBs ")))));

_L6:
    String s2;
    s2;
    covertjava.chat.b b1 = (covertjava.chat.b)Naming.lookup(s2);
    MessageInfo messageinfo = new MessageInfo(e, f);
    b1.receiveMessage(s1, messageinfo);
    System.out.println(a("\037 gGw5 4Gs<14@yr-{Gbr").concat(String.valueOf
    -(String.valueOf(s))));

    if(flag)
        b.c = !b.c;
    return;
}

```

Isso é uma confusão total, mas poderosa! O método sendMessage é relativamente pequeno com pouca lógica condicional. Se o ofuscamento do fluxo de controle fosse aplicado a um método mais complexo com loops for, instruções if e variáveis locais, o ofuscamento seria ainda mais eficaz.

## Inserindo código corrompido

Inserir código corrompido é uma técnica um pouco duvidosa utilizada por alguns ofuscadores para impedir que classes ofuscadas sejam descompiladas. A técnica é baseada em uma interpretação vaga da especificação do bytecode Java pelo JRE. O JRE não impõe estritamente todas as regras de verificação do formato de bytecode. Isso permite que ofuscadores introduzam bytecode incorreto nos arquivos de classes. O código introduzido não impede que o original funcione, mas uma tentativa de descompilar o arquivo de classe resulta em uma falha – ou, na melhor das hipóteses, em um código-fonte confuso cheio de palavras-chave JVM INSTR. A Listagem 3.3 mostra como um descompilador poderia tratar código corrompido. O risco da utilização desse método é que o código corrompido talvez não funcione em uma versão da JVM que obedeça de maneira mais rigorosa à especificação. Mesmo que isso não seja hoje um problema para a maioria das JVMs, talvez venha a tornar-se um problema mais tarde.

## Eliminando código não-utilizado (encolhimento)

Como um benefício adicional, a maioria dos ofuscadores remove o código não-utilizado, o que resulta na redução do tamanho da aplicação. Por exemplo, se uma classe A tiver um método `m()` que nunca é chamado por nenhuma classe, o código para `m()` será removido do bytecode de A. Esse recurso é especialmente útil para o download de código via Internet, ou para código que seja instalado em ambientes inseguros.

## Otimizando bytecode

Um benefício adicional proclamado pelos ofuscadores é a potencial otimização do código. Os fornecedores sustentam que declarar métodos não-finais como finais sempre foi possível e que realizar aprimoramentos menores no código pode ajudar a acelerar a execução. É difícil avaliar os ganhos reais de desempenho; a maioria dos fornecedores não publica as métricas relacionadas. O que vale observar aqui é que, a cada nova versão, compiladores JIT estão se tornando mais poderosos. Portanto, recursos como tornar métodos *final* e eliminar código morto são, de toda forma, mais comumente realizados pelo JIT.

## Conhecendo os melhores ofuscadores

Há muitos ofuscadores disponíveis, e a maioria contém o mesmo conjunto de recursos básicos. A Tabela 3.1 inclui alguns dos produtos mais populares, tanto gratuitos como comerciais.

**TABELA 3.1****Ofuscadores populares**

PRODUTO	KLASSMASTER	PROGUARD	RETRO GUARD	DASH-O	JSHRINK
Versão	4.1	1.7	1.1.13	2.x	2.0
Preço	US\$199- US\$399	Gratuito	Gratuito	US\$895- US\$2995	US\$95
Removendo informações de depuração	Sim	Sim	Sim	Sim	Sim
Desfiguração de nomes	Sim	Sim	Sim	Sim	Sim
Codificação de strings Java	Sim	Não	Não	Não	Sim
Alteração do fluxo de controle	Sim	Não	Não	Não	Não
Inserção de código corrompido	Sim	Não	Não	Não	Não
Eliminação de código não-utilizado (encolhimento)	Sim	Sim	Não	Sim	Sim
Otimização de bytecode	Não	Não	Não	Sim	Sim
Flexibilidade da linguagem de scripts e controle sobre o ofuscamento	Excelente	Excelente	Bom	Não avaliado	Bom
Reconstrução de stack traces	Sim	Sim	Não	Não	Não

Para aplicações comerciais que contenham propriedade intelectual, recomendo o Zelix KlassMaster, principalmente pelo seu recurso exclusivo de ofuscamento de fluxo de controle. Essa técnica torna o código ofuscado realmente difícil de quebrar, portanto o produto vale seu preço. No momento em que eu escrevia este livro, o Zelix KlassMaster era o único ofuscador conhecido com esse recurso. O ProGuard está disponível gratuitamente em [sourceforge.net](http://sourceforge.net) e é a melhor escolha para usuários econômicos, trabalhando com aplicações que não requerem proteção de padrão comercial.

## Problemas potenciais e soluções comuns

O ofuscamento é um processo razoavelmente seguro que deve preservar a funcionalidade da aplicação. Entretanto, em alguns casos as transformações realizadas pelos ofuscadores podem inadvertidamente quebrar código que estava funcionando. As seções a seguir examinam os problemas comuns e as soluções recomendadas.

### Carregamento de classes dinâmicas

A renomeação de pacotes, classes, métodos e variáveis funciona bem, contanto que os nomes sejam alterados de maneira consistente em todo o sistema. Os ofuscadores garantem que quaisquer referências estáticas dentro do bytecode sejam atualizadas para

refletir os novos nomes. No entanto, se o código realizar o carregamento de classes dinâmicas utilizando `Class.forName()` ou `ClassLoader.loadClass()`, passando um nome original de classe (não-ofuscado), isso pode resultar em uma `ClassNotFoundException`. Ofuscadores modernos são excelentes para tratar esses casos, e tentam alterar strings para refletir os novos nomes. Mas se a string for criada em tempo de execução, ou lida a partir de um arquivo de propriedades, o ofuscador não será capaz de tratá-la. Bons ofuscadores produzem um arquivo de log com alertas indicando código com potenciais problemas de execução.

A solução mais simples é configurar o ofuscador para preservar os nomes das classes carregadas dinamicamente. O conteúdo das classes, tais como métodos, variáveis e o código, ainda pode ser transformado.

## Reflection

*Reflection* requer conhecimento em tempo de compilação dos nomes de métodos e campos, por isso também é afetada pelo ofuscamento. Certifique-se de utilizar um bom ofuscador e examine os alertas no arquivo de log. Assim como ocorre com o carregamento de classes dinâmicas, se erros de execução forem causados pelo ofuscamento, você deverá excluir do ofuscamento os nomes de métodos ou de campos referenciados em `Class.getMethod()` ou `Class.getField()`.

## Serialização

Dentre os objetos Java serializados, incluem-se dados de instâncias e informações sobre classes. Se a versão da classe ou a sua estrutura mudar, isso poderá resultar em uma exceção de desserialização. Classes ofuscadas podem ser serializadas e desserializadas, mas uma tentativa de desserializar uma instância de uma classe não-ofuscada por uma classe ofuscada não irá funcionar. Esse não é um problema muito comum; normalmente pode ser resolvido excluindo-se as classes serializáveis do ofuscamento ou evitando-se misturar classes serializadas.

## HISTÓRIAS DAS TRINCHEIRAS

O produto mais inovador da CreamTec é o WebCream, disponível para download gratuito na Web. A edição gratuita é limitada a cinco usuários simultâneos; para um número maior, é necessário adquirir uma licença comercial. Tendo sido criado na Ucrânia, encontrei muitas pessoas que preferiram violar o licenciamento para transformar a edição gratuita em uma edição ilimitada que normalmente valeria milhares de dólares. Na CreamTec, utilizamos um ofuscador simples e gratuito que não faz muito mais do que a desconfiguração de nomes. Achávamos isso suficiente até que um amigo meu, que vê a funcionalidade limitada dos softwares comerciais como um insulto pessoal, quebrou nosso código de licenciamento em menos de 15 minutos. A mensagem foi clara, e decidimos adquirir o Zelix KlassMaster para proteger o produto o melhor possível. Depois que utilizamos o ofuscamento agressivo do fluxo de controle e alguns outros truques, nosso amigo não foi capaz de quebrar o código de licenciamento com a mesma facilidade de antes – e como não quis gastar dias para descobrir, desistiu.

## Violão de convenções de nomeação

A renomeação de métodos pode violar padrões como os de Enterprise JavaBeans (EJB), em que se exige que o desenvolvedor forneça métodos com nomes e assinaturas específicos. Os métodos callback do EJB, como ejbCreate e ejbRemove, não são definidos por uma superclasse ou interface. Fornecer esses métodos com uma assinatura específica é uma convenção definida na especificação EJB e imposta pelo container. Alterar os nomes de métodos de callback viola essa convenção tornando os beans inutilizáveis. Você deve sempre se certificar de excluir os nomes desses métodos do ofuscamento.

## Dificuldades com manutenção

Por último, não se esqueça de que o ofuscamento dificulta a manutenção e a solução de problemas em aplicações. O tratamento de exceções em Java é uma maneira eficaz de isolar código com problemas, e a análise de rastreamentos de pilha (*stack traces*) geralmente pode nos dar uma boa idéia do que saiu errado e onde isso ocorreu. Guardar informações de depuração para nomes de arquivos-fonte e números de linha permite que operações em tempo de execução informem a localização exata em que o erro ocorreu. Se feito sem cuidado, o ofuscamento pode inibir esse recurso e tornar a depuração mais difícil, pois o desenvolvedor só verá os nomes de classes ofuscados, em vez dos nomes reais das classes e os números de linhas.

Você deve preservar pelo menos as informações de números de linhas no código ofuscado. Bons ofuscadores produzem um logging das transformações, incluindo o mapeamento entre nomes e métodos originais da classe e os nomes ofuscados correspondentes. Veja a seguir um trecho do arquivo de log gerado pelo Zelix KlassMaster para a classe ChatServer:

```
Class: public covertjava.chat.ChatServer => covertjava.chat.d
Source: "ChatServer.java"
FieldsOf: covertjava.chat.ChatServer
    hostName => e
    protected static instance => a
    messageListener => d
    protected registry => c
    protected registryPort => b
    userName => f
MethodsOf: covertjava.chat.ChatServer
    public static getInstance( ) => a
    public getRegistry(int) => a
    public init( ) => b
    public receiveMessage(java.lang.String, covertjava.chat.MessageInfo)
        => NameNotChanged
    public sendMessage(java.lang.String, java.lang.String) => a
    public setMessageListener(covertjava.chat.MessageListener) => a
```

Assim, se um rastreamento de pilha de exceção mostrar o método `covertjava.chat.d.b`, você pode consultar o log e descobrir que o método se chamava "init" em uma classe antes chamada de `covertjava.chat.ChatServer`. Se a exceção ocorreu em `covertjava.chat.d.a`, você não terá como saber com certeza o nome original do método, pois há múltiplos mapeamentos (o que comprova o poder do *overloading*). É por isso que os números de linhas são tão importantes. Utilizando o arquivo de log e o número de linha no arquivo-fonte original, você pode localizar rapidamente a área problemática no código da aplicação.

Alguns ofuscadores fornecem um utilitário que reconstrói os *stack traces*. É uma maneira conveniente de se obter o *stack trace* real a partir do correspondente ofuscado. Esse utilitário geralmente emprega o mesmo método que utilizamos anteriormente, mas automatiza o trabalho – então por que não poupar um pouco de tempo? Ele também permite embaralhar os números de linha como uma proteção adicional.

## Utilizando o Zelix KlassMaster para ofuscar uma aplicação de chat

Mesmo que cada ofuscador tenha seu próprio formato para configurar transformações, todos suportam um conjunto comum de recursos. A aplicação Chat não contém algoritmos de última geração ou invenções patenteadas, mas ela é muito importante para nós. Assim, utilizaremos o Zelix KlassMaster para protegê-la contra os olhos curiosos de hackers e ladrões.

Primeiro, obtemos e instalamos uma cópia do Zelix KlassMaster em uma máquina local. Lembre-se de que nos referimos ao diretório *home* da aplicação Chat como *CovertJava*. Em seguida, copiamos *ZKM.jar* no diretório de instalação do KlassMaster para o diretório *lib* do nosso projeto, de modo que possamos fazer um script fazendo referência a ele. A maneira mais fácil de criar o script de ofuscamento é com a GUI (interface gráfica) do KlassMaster. Utilizando o comando

```
java -jar ZKM.jar
```

no diretório *lib*, executamos a GUI. Em seguida, na caixa de diálogo auxiliar inicial que aparece, selecionamos a opção Set Classpath. Agora selecionamos as bibliotecas de tempo de execução do JDK que estamos utilizando e, na caixa de diálogo Open Classes, selecionamos *CovertJava/lib/chat.jar*. Depois disso, o KlassMaster deve carregar todas as classes da aplicação Chat. Assim poderemos visualizar as estruturas internas do bytecode. A tela deve ser semelhante à da Figura 3.1.

Ao trabalhar com a GUI, você pode facilmente perceber a flexibilidade do KlassMaster. Você pode alterar manualmente os nomes de classes, métodos e campos; modificar a visibilidade de classes ou métodos; tornar métodos finais; alterar strings de texto e fazer outras coisas úteis. O KlassMaster tenta propagar as alterações por todo o código carregado; por isso, se outras classes se referirem a um método e você alterar o nome dele, as classes referenciadas serão atualizadas para refletir a alteração. Depois de fazer todas as suas alterações, você pode salvar as classes como estão ou primeiro limpá-las e ofuscá-las. As classes carregadas no ambiente gráfico podem ser modificadas ainda depois do ofusca-



**FIGURA 3.1** Classes do Chat carregadas na GUI do KlassMaster.

mento, embora eu não veja por que alguém precisaria fazer isso. Para detalhes dos recursos do KlassMaster e como utilizá-lo, consulte o manual de usuário.

Uma aplicação Java bem escrita fornece scripts para construí-la (*build*), por isso vamos integrar o ofuscamento ao nosso script de build. Iniciamos utilizando a GUI do KlassMaster para criar o script de ofuscamento. Em seguida, atualizamos o script manualmente para torná-lo mais flexível. É possível escrever o script manualmente ou copiar e modificar um script de exemplo. Executamos a GUI e selecionamos ZKM Script Helper no menu Tools. Então, fazemos o seguinte:

1. Leia as instruções na Introductory Page e clique em Next.
2. Na página Classpath Statement, selecione rt.jar e clique em Next.
3. Na página Open Statement, navegue até CovertJava/distrib/chat.jar e clique em > para abri-lo. Precisamos apenas de um arquivo, porque todas as nossas classes da aplicação encontram-se nele. Clique em Next.
4. Na página TrimExclude Statement, as exclusões-padrão são predefinidas para excluir os casos em que o ofuscamento possivelmente resulta em um erro. Por exemplo, renomear métodos de uma classe de implementação EJB torna-a inutilizável, e dessa forma os EJBs são excluídos por padrão.
5. Na página Trim Statement, marque a caixa de seleção Delete Source File Attributes e a caixa de seleção Delete Deprecated Attributes para remover as informações de depuração; clique então em Next.
6. Na caixa de combinação Don't Change Main Class Name na página Exclude Statement, selecione covertjava.chat.ChatApplication para preservar seu nome. Assim as

entradas válidas do *manifest* do JAR se mantêm visíveis, tornando-se possível que os usuários continuem a invocar a aplicação Chat utilizando um nome inteligível para humanos.

7. Na página Obfuscate Statement, selecione Aggressive na caixa de combinação Obfuscate Control Flow. Selecione então Aggressive na caixa Encrypt String Literals e selecione Scramble na caixa de combinação Line Number Tables. Isso assegura uma proteção adequada ao código e permite converter stack traces mais tarde. Certifique-se de que Produce a Change Log File estejam marcados e clique em Next.
8. Na página SaveAll Statement, navegue pelo CovertJava/distrib e crie um subdiretório chamado obfuscated. Selecione o diretório recém-criado para a saída e clique em Next.
9. A próxima página deve mostrar o texto do script e permite salvá-lo em um diretório. Salve-o como obfuscate\_script.txt no diretório CovertJava/build e feche a GUI. O script resultante deve ser semelhante ao da Listagem 3.4.

#### **LISTAGEM 3.4** O script de ofuscamento gerado pela GUI

---

```
/****************************************/
/* Generated by Zelix KlassMaster 4.1.1 ZKM Script Helper 2003.08.13 17:03:43 */
/****************************************/

classpath "c:\java\jdk1.4\jre\lib\rt.jar"
           "c:\java\jdk1.4\jre\lib\sunrsasign.jar"
           "c:\java\jdk1.4\jre\lib\jsse.jar"
           "c:\java\jdk1.4\jre\lib\jce.jar"
           "c:\java\jdk1.4\jre\lib\charsets.jar";

open      "C:\Projects\CovertJava\distrib\chat.jar";

trim     deleteSourceFileAttributes=true
        deleteDeprecatedAttributes=true
        deleteUnknownAttributes=false;

exclude   covertjava.chat.^ChatApplication^ public static main(java.lang.String[ ]);

obfuscate changeLogFileIn=""
           changeLogFileOut="ChangeLog.txt"
           obfuscateFlow=aggressive
           encryptStringLiterals=aggressive
           lineNumbers=scramble;

saveAll    archiveCompression=all "C:\Projects\CovertJava\distrib\obfuscated";
```

---

Uma boa idéia seria substituir os caminhos absolutos por caminhos relativos, de modo que, em vez de abrir C:\Projects\CovertJava\distrib\chat.jar, o script abra dis-

trib\chat.jar. Por fim, integramos o ofuscamento ao processo de build declarando uma tarefa (*task*) personalizada e adicionando um alvo (*target*) que a chama. O KlassMaster é escrito em Java e pode ser chamado a partir de qualquer script de build. Convenientemente, ele fornece uma classe *wrapper* para integração ao Ant, e tudo o que precisamos fazer é adicionar o seguinte a build.xml da aplicação Chat:

```
<!-- Define uma tarefa que executará Zelix KlassMaster para ofuscar classes -->
<taskdef name="obfuscate" classname="ZKMTTask" classpath="${basedir}/lib/ZKM.jar"/>
...
<!-- Define um alvo que produz uma versão ofuscada de Chat -->
<target name="obfuscate" depends="release">
    <obfuscate scriptFileName="${basedir}/build/obfuscate_script.txt"
        logFileName="${basedir}/build/obfuscate_log.txt"
        trimLogFileName="${basedir}/build/obfuscate_trim_log.txt"
        defaultExcludeFileName="${basedir}/build/obfuscate_defaultExclude.txt"
        defaultTrimExcludeFileName="${basedir}/build/obfuscate_defaultTrimExclude.txt"
        defaultDirectoryName="${basedir}">
    />
</target>
```

Agora, podemos executar Ant no alvo ofuscado. Se o build for bem-sucedido, um novo arquivo (chat.jar) será criado em CovertJava/distrib/obfuscated. Esse arquivo contém a versão ofuscada da aplicação Chat que ainda pode ser invocada utilizando-se o comando java -jar chat.jar. Examine esse JAR e tente descompilar algumas classes.

Antes de encerrarmos o assunto sobre a utilização do KlassMaster, vamos fornecer alguns outros exemplos da sintaxe de arquivos de script para excluir classes e membros de classes do ofuscamento. O formato mostrado na Tabela 3.2 pode ser utilizado para instruções de um script de ofuscamento que recebe nomes como parâmetros. A linguagem de scripts ZKM suporta coringas como \* (qualquer seqüência de caracteres) e ? (qualquer caractere único), além de operações booleanas, como !! (ou) e ! (não). Para uma explicação detalhada e a sintaxe completa, consulte a documentação do KlassMaster.

**TABELA 3.2**

**Padrões de nomes comumente utilizados para KlassMaster**

SINTAXE	CORRESPONDE A
package1.package2.	Nomes de pacotes package1 e package2. Outros nomes de pacotes e filhos de package2 não serão correspondidos.
*	Todos os nomes de pacotes na aplicação.
Class1	O nome da classe Class1.
package1.Class1	O nome de Class1 no pacote package1, mas não o nome de package1.
package1.^Class1	Os nomes da Class1 e de package1.
package1.^Class1^ method1( )	Os nomes de package1, Class1 e method1, sem parâmetros.
package1.^Class1^ method1(*)	Os nomes de package1, Class1 e todas as versões sobrecarregadas ( <i>overloaded</i> ) de method1.

## Quebrando código ofuscado

Agora que investimos tanto tempo discutindo como proteger a propriedade intelectual por meio do ofuscamento, precisamos falar um pouco sobre a força dessa proteção. Um bom ofuscador torna difícil hackear uma aplicação? Certamente. Ele garante que a aplicação não será hackeada? De jeito nenhum!

A menos que o ofuscamento de controle de fluxo seja utilizado, ler e trabalhar com o código ofuscado não é tão difícil. O segredo é encontrar um bom ponto de partida para a descompilação. O Capítulo 2 apresenta várias técnicas para fazer a engenharia reversa em aplicações, mas o ofuscamento pode derrotar muitas delas. Por exemplo, a maneira mais eficaz de localizar um ponto de partida é pesquisar o texto dos arquivos de classes. Com a codificação de strings, a pesquisa não fornecerá nenhum resultado, pois as strings não são armazenadas como texto simples. Os nomes de pacotes e de classes não mais poderão ser utilizados para conhecer a estrutura da aplicação ou para escolher um bom ponto de partida. Tecnicamente, ainda será possível descompilar o ponto de entrada da aplicação e navegar pelo fluxo de controle até o ponto de entrada, para uma aplicação de tamanho razoável, mas isso não seria praticável.

Para código com o fluxo de controle ofuscado, o método mais sensato de conhecer a implementação da aplicação é utilizando um bom e velho depurador. A maioria dos IDEs possui recursos de depuração, mas nosso caso exigirá um depurador peso-pesado capaz de funcionar sem o código-fonte. Para localizar um bom ponto de partida para a descompilação, a aplicação precisa ser executada em modo de depuração. A plataforma Java tem uma API padrão para depuradores chamada Debugger API, com capacidades de depuração local e remota. A depuração remota permite ao depurador anexar-se a uma aplicação em execução no modo de depuração; é uma maneira popular de se quebrar uma aplicação. Bons depuradores exibem informações detalhadas sobre a execução de threads, pilhas de chamadas para cada thread, classes carregadas e objetos em memória. Eles permitem configurar um breakpoint e rastrear execuções de métodos. Uma técnica-chave para trabalhar com aplicações ofuscadas é utilizar a interface comum (GUI ou API de programação), para navegar por um recurso de interesse, e então contar com o depurador para conhecer as classes que implementam o recurso. Depois de essas classes serem identificadas, elas podem ser descompiladas e analisadas, como descrito no Capítulo 2. Trabalhar com depuradores será explorado em detalhes no Capítulo 9.

## Questionário rápido

1. Quais são os meios de proteger a propriedade intelectual em aplicações Java?
2. Quais transformações fornecidas pelos ofuscadores oferecem a melhor proteção?
3. Para cada um dos potenciais problemas listados neste capítulo, quais transformações podem causá-los?
4. Qual a maneira mais eficiente de analisar código ofuscado?

## Resumo

- O ofuscamento pode ser a melhor maneira de proteger a propriedade intelectual no bytecode Java.
- Os ofuscadores realizam algumas ou todas as transformações a seguir: remoção das informações de depuração, desfiguração de nomes, codificação de strings, alteração do fluxo de controle, inserção de código corrompido, eliminação de código não-utilizado e otimização de bytecode.
- O ofuscamento cria dificuldades de manutenção que podem ser minimizadas configurando o ofuscador.
- O código ofuscado continua legível, a menos que sejam utilizados o ofuscamento do fluxo de controle e a codificação de strings.

# 4

## *NESTE CAPÍTULO*

- ▶ O problema do encapsulamento 38
- ▶ Acessando pacotes e membros protegidos de classes 39
- ▶ Acessando membros privados de classes 41
- ▶ Questionário rápido 44
- ▶ Resumo 44

# Hackeando métodos não-públicos e variáveis de uma classe

*"Qualquer coisa pode ser forçada a funcionar se você mexer com ela. Se você mexer com ela por tempo suficiente, você vai quebrá-la."*

As Leis de Murphy aplicadas à tecnologia

## O problema do encapsulamento

O encapsulamento é um dos pilares da programação orientada a objetos. O propósito do encapsulamento é a separação da interface da implementação e a modularidade dos componentes de aplicação. Geralmente, é recomendável tornar os membros de dados privados ou protegidos (*protected*) e fornecer funções de acesso e modificação públicas (também conhecidas como funções *getter* e *setter* públicas). Às vezes, também é recomendável tornar privados ou públicos os métodos de implementação interna, para impedir que uma classe seja utilizada incorretamente. Seguir o princípio do encapsulamento ajuda a criar aplicações de maior qualidade, mas, ocasionalmente, isso pode ser um obstáculo a uma utilização não-prevista pelo desenvolvedor das classes.

Utilizaremos `java.awt.BorderLayout` em nossas experiências. Talvez em algum momento isso encoraje os engenheiros da "JavaSoft" (o departamento de software Java da Sun) a adicionar métodos públicos. Obteremos o código-fonte para `BorderLayout` de `src.jar` no diretório de instalação do JDK.

## Acessando pacotes e membros protegidos de classes

Iniciaremos demonstrando como acessar facilmente variáveis e métodos com visibilidade *package* (de pacote). Nossa exemplo utiliza uma variável com visibilidade de pacote, mas a técnica funciona da mesma maneira para variáveis *protected*. Uma variável ou método tem *visibilidade de pacote* quando nenhum modificador de visibilidade específico como *public*, *protected* ou *private* é utilizado na declaração. *BorderLayout* armazena o componente adicionado utilizando a restrição *BorderLayout.CENTER* em uma variável *center* declarada desta maneira:

```
package java.awt;

public class BorderLayout implements LayoutManager2, java.io.Serializable {
    ...
    Component center;
    ...
}
```

Lembre que membros com visibilidade de pacote são acessíveis à classe que os declarou e a todas as classes dentro do mesmo pacote. No nosso exemplo, qualquer classe no pacote *java.awt* acessa a variável *center* diretamente. Portanto, uma solução simples é criar uma classe auxiliar, *AwtHelper*, no pacote *java.awt* e utilizá-la para acessar membros com visibilidade de pacote das instâncias de *BorderLayout*. *AwtHelper* tem um método público que recebe uma instância de *BorderLayout* e retorna o componente para uma dada restrição de layout:

### HISTÓRIAS DAS TRINCHEIRAS

O WebCream é um produto que converte aplicações AWT Java e Swing em sites HTML interativos. Ele faz isso emulando um ambiente gráfico para a aplicação GUI, executando no lado do servidor e capturando e convertendo a janela atualmente em foco em uma página HTML. Para gerar o HTML, o WebCream itera por todos os containers e tenta simular layouts Java com tabelas HTML. Um dos layouts que o WebCream precisa suportar é *BorderLayout*. Para um container com *BorderLayout*, o módulo de renderização de HTML precisa saber qual componente filho foi adicionado à seção South, à seção North, e assim por diante. O *BorderLayout* armazena essas informações nas variáveis-membro *south*, *north* e assim por diante. Ele tem até um método *getChild()* que pode ser utilizado para obter o componente. O problema é que as variáveis são declaradas com visibilidade de pacote e o método *getChild* é declarado como privado. Para contornar a ausência de acesso público aos componentes secundários de *BorderLayout*, os engenheiros da WebCream tiveram de contar com as técnicas de hacking descritas neste capítulo.

```
package java.awt;

public class AwtHelper {

    public static Component getChild(BorderLayout layout, String key) {
        Component result = null;

        if (key == BorderLayout.NORTH)
            result = layout.north;
        else if (key == BorderLayout.SOUTH)
            result = layout.south;
        else if (key == BorderLayout.WEST)
            result = layout.west;
        else if (key == BorderLayout.EAST)
            result = layout.east;
        else if (key == BorderLayout.CENTER)
            result = layout.center;
        return result;
    }
}
```

Vamos escrever uma classe de testes chamada `covertjava.visibility.PackageAccessTest` que utiliza `AwtHelper` para obter a instância do *Split Pane* no *MainFrame* da aplicação Chat. É no trecho de código a seguir que estamos mais interessados:

```
Container container = createTestContainer();
if (container.getLayout() instanceof BorderLayout) {
    BorderLayout layout = (BorderLayout)container.getLayout();
    Component center = AwtHelper.getChild(layout, BorderLayout.CENTER);
    System.out.println("Center component = " + center);
}
```

Obtemos o layout do container e, se este for `BorderLayout`, utilizamos `AwtHelper` para obter o componente na posição central. O *MainFrame* da aplicação Chat tem o *Split Pane* no centro; portanto, se o código for escrito corretamente, devemos ver uma instância de `JSplitPane` no console do sistema. Executando `PackageAccessTest`, obtemos a seguinte exceção:

```
java.lang.SecurityException: Prohibited package name: java.awt
```

A exceção é lançada porque `java.awt` é considerado como um espaço de nomes de sistema, que não deve ser utilizado por classes comuns. Isso não teria acontecido se estivéssemos tentando hackear um membro com visibilidade de pacote de uma classe de terceiros, mas selecionamos intencionalmente uma classe de sistema para ilustrar um exemplo do dia-a-dia. O único problema em potencial com a utilização dessa técnica para um espaço de nomes que não seja de sistema, como `com.mycompany.mypackage`, ocorre se o pacote for selado (*sealed*). Adicionar uma classe auxiliar a um pacote selado requer a mesma técnica explicada para adicionar uma classe corrigida com um patch no Capítulo 5.

Adicionar classes de sistema é um pouco mais difícil, porque elas são carregadas e tratadas de maneira diferente das classes de aplicação. O Capítulo 16 apresenta uma discussão abrangente sobre as classes de sistema. Por enquanto, é suficiente dizer que, para adicionar uma classe ao pacote de sistema, essa classe deve estar no *classpath* de inicialização (*boot classpath*). Um diretório ou arquivo JAR pode ser prefixado ou acrescentado ao *classpath* de inicialização utilizando o parâmetro *-Xbootclasspath* na linha de comando de java. Como já temos um subdiretório patches para a aplicação Chat, iremos utilizá-lo também para as classes de sistema. Modificamos build.xml para mover o diretório java.lang.com.AwtHelper para distrib/patches e criar um novo script (package\_access\_test.bat) em distrib/bin desta maneira:

```
@echo off  
set CLASSPATH=..\lib\chat.jar  
java -Xbootclasspath/p:..\patches covertjava.visibility.PackageAccessTest
```

Executar package\_access\_test.bat produz a saída a seguir:

```
C:\Projects\CovertJava\distrib\bin>package_access_test.bat  
Testing package-visible access  
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

Ter de colocar as classes no *classpath* de inicialização do sistema torna a instalação (*deployment*) mais complicada, pois requer modificação do script de inicialização. Por exemplo, uma aplicação Web instalada em um container Web/J2EE, como o Tomcat ou o WebLogic, não poderá mais ser simplesmente instalada por meio de um console ou diretório de deployment de aplicação. O script que inicia o servidor de aplicações deve ser modificado para incluir o parâmetro *-Xbootclasspath*. Outra desvantagem dessa técnica é que ela não funciona com membros privados. Finalmente, não devemos esquecer que adicionar classes a pacotes pode violar o acordo de licenciamento do produto. Esse é o caso com BorderLayout, já que uma seção no acordo de licenciamento Java da Sun proíbe explicitamente a adição de classes a pacotes que iniciem com java. A seção a seguir apresenta uma alternativa que resolve alguns desses problemas.

## Acessando membros privados de classes

Membros privados estão acessíveis somente à classe que os declara. Essa é uma das regras fundamentais da linguagem Java que assegura o encapsulamento. Mas é realmente assim? Isso é realmente imposto sempre? Se você pensou “bem, esse cara está escrevendo sobre isso, portanto deve haver algum tipo de brecha”, você estava certo. O compilador do Java impõe a privacidade sobre membros privados em tempo de compilação. Portanto, não pode haver nenhuma referência estática em outras classes a variáveis e métodos privados de uma classe. Entretanto, o Java tem um mecanismo poderoso de *reflection* que permite consultar a instância e os metadados da classe e acessar os campos e métodos em tempo de execução. Como o reflection é dinâmico, verificações em tempo de compilação não são aplicáveis. Em vez disso, Runtime Java conta com um gerenciador de segurança (*security manager*) – se houver um – para verificar se o código chamador tem privilé-

gios suficientes para um tipo específico de acesso. O gerenciador de segurança fornece proteção suficiente, porque todas as funções da API de reflection são delegadas para ele antes da execução. O que subverte essa proteção é o fato de que o gerenciador de segurança freqüentemente não é ativado. Por padrão, o gerenciador de segurança não é ativado. A menos que o código instale explicitamente um gerenciador padrão ou personalizado, as verificações de controle de acesso em tempo de execução não são ativadas. Mesmo se um gerenciador de segurança estiver ativo, ele em geral é configurado por um arquivo de políticas de segurança que pode ser estendido para permitir acesso à API de reflection.

Se você examinou a classe `BorderLayout` cuidadosamente, pode ter notado que ela já tem um método retornando um componente filho com base na posição fornecida. Ele é chamado `getChild` e tem a seguinte assinatura:

```
private Component getChild(String key, boolean ltr)
```

Parece uma boa notícia, pois você não terá de escrever sua própria implementação. O problema é que o método é declarado como privado; não há nenhum método público que se possa utilizar para chamá-lo. Para aproveitar o código JDK existente, você deve chamar `BorderLayout.getChild( )` utilizando a API de reflection. Utilizaremos a mesma estrutura de testes da seção anterior. Agora, em vez de utilizar `AwtHelper`, a classe de testes faz a delegação para sua própria função auxiliar (`getChild( )`):

```
public class PrivateAccessTest {

    public static void main(String[ ] args) throws Exception {
        Container container = createTestContainer( );
        if (container.getLayout( ) instanceof BorderLayout) {
            BorderLayout layout = (BorderLayout)container.getLayout( );
            Component center = getChild(layout, BorderLayout.CENTER);
            System.out.println("Center component = " + center);
        }
        ...
    }

    public static Component getChild(BorderLayout layout, String key) throws Exception {
        Class[ ] paramTypes = new Class[ ]{String.class, boolean.class};
        Method method = layout.getClass( ).getDeclaredMethod("getChild", paramTypes);
        // Métodos privados não são acessíveis por padrão
        method.setAccessible(true);
        Object[ ] params = new Object[ ] {key, new Boolean(true)};
        Object result = method.invoke(layout, params);
        return (Component)result;
    }

    ...
}
```

A implementação de `getChild( )` é o coração da técnica. Ela obtém o objeto do método por meio de reflection e então chama `setAccessible(true)`. O valor `true` é definido para suprimir a verificação de controle de acesso e permitir a invocação do método. O restante do método é a utilização simples da API de reflection. Executar `covertjava.visibility.PrivateAccessTest` produz a mesma saída mostrada na seção anterior:

```
C:\Projects\CovertJava\distrib\bin>private_access_test.bat
Testing private access
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

Isso foi muito fácil. Talvez seja necessário um pouco mais de trabalho se um gerenciador de segurança for configurado utilizando-se `System.setSecurityManager` ou através da linha de comando, que é o caso na maioria dos servidores de aplicação e produtos de middleware. Se executarmos nosso teste passando `-Djava.security.manager` para o comando `java`, obteremos a exceção a seguir:

```
java.security.AccessControlException: access denied
(java.lang.RuntimePermission accessDeclaredMembers)
```

Para que nosso código funcione com um gerenciador de segurança instalado, temos que conceder permissões para o acesso a membros declarados, por meio de reflection, e suprimir as verificações de acesso. Fazemos isso adicionando um arquivo de políticas de segurança do Java que concede (com um *grant*) essas duas permissões ao nosso código:

```
grant {
    permission java.lang.RuntimePermission "accessDeclaredMembers";
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```

Por fim, criamos um novo script de testes (`private_access_test.bat`) no diretório `distrib\bin` que adiciona um parâmetro de linha de comando (`java.security.policy`) a fim de instalar nosso arquivo de políticas:

```
set CLASSPATH=..\lib\chat.jar
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.manager
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.policy=../conf/java.policy

java %JAVA_ARGS% covertjava.visibility.PrivateAccessTest
```

Se um arquivo de políticas já estiver instalado, nossa cláusula *grant* precisa ser inserida nele. Os arquivos de segurança do Java permitem incluir arquivos de políticas adicionais usando o atributo `policy.url.n`. Consulte o Capítulo 7 para uma discussão detalhada sobre arquivos de segurança e de políticas do Java.

A técnica que conta com a API de reflection pode ser utilizada para acessar membros com visibilidade de pacote e *protected*. Isso torna desnecessário inserir classes auxiliares em pacotes de terceiros. A desvantagem da API de reflection é que ela é notoriamente lenta, porque tem que lidar com informações de tempo de execução e talvez passar por

verificações de segurança. Quando a velocidade é importante, é preferível contar com classes auxiliares, para membros com visibilidade de pacote ou protegida. Contudo, uma alternativa é serializar uma instância em um stream de array de bytes e então analisar sintaticamente esse stream para obter os valores das variáveis-membro. Obviamente, esse é um processo entediante que não funciona para campos *transient*.

## Questionário rápido

1. Qual técnica pode ser utilizada para obter o valor de uma variável protegida?
2. Qual técnica pode ser utilizada para obter o valor de uma variável privada?
3. Quais são as vantagens e desvantagens de cada técnica?

## Resumo

- Métodos e variáveis não declarados como `public` podem ainda assim ser acessados.
- Um membro com a visibilidade `package` ou `protected` pode ser acessado inserindo-se uma classe auxiliar ao seu pacote ou utilizando a API de reflection.
- Um membro com visibilidade `private` pode ser acessado utilizando a API de reflection.
- Se um gerenciador de segurança estiver instalado, as políticas de segurança de Java precisam ser alteradas para permitir acesso irrestrito à API de reflection.

# **Substituindo e aplicando patches a classes de aplicações**

*"Quantos engenheiros de software são necessários para trocar uma lâmpada?"*

*"Nenhum. Iremos documentar isso no manual."*

*"Nenhum. É um problema de hardware."*

*"Um, mas ele só vai estar disponível em 2010."*

*"Dois. Um deles sempre sai no meio do projeto."*

*"Quatro. Um para projetar a troca, um para implementá-la, um para documentá-la e outro para mantê-la funcionando depois."*

## **O que fazer depois de termos explorado cada possibilidade, mas falharmos?**

Quase todos os desenvolvedores em algum momento utilizaram uma biblioteca ou um componente desenvolvido por outra pessoa. Quase todos os desenvolvedores em algum momento já ficaram frustrados com bibliotecas, ao ponto de desejarem encontrar a pessoa que decidiu tornar um método `private` e ensinar algumas coisas a ela. Bem, a maioria de nós não iria tão longe, mas seria ótimo poder alterar as coisas que atrapalham nossas vidas. Não que bibliotecas sejam escritas por pessoas más; é que mesmo os projetistas mais

# **5**

## ***NESTE CAPÍTULO***

- ▶ O que fazer depois de termos explorado cada possibilidade, mas falharmos? 45
- ▶ Localizando a classe em que é necessário aplicar um patching 47
- ▶ Exemplo de cenário que requer patching 49
- ▶ Aplicando um patch a uma classe para adicionar nova lógica 53
- ▶ Reconfigurando a aplicação para carregar e utilizar a classe com patch 53
- ▶ Aplicando patch a pacotes selados 54
- ▶ Questionário rápido 55
- ▶ Resumo 55

brilhantes não são capazes de prever todas as formas possíveis com que outros desenvolvedores usarão seus códigos.

Certamente, sempre é melhor resolver essas questões pacificamente. Se você puder fazer com que o fornecedor altere o código, ou se estiver em uma posição em que você mesmo possa fazê-lo, faça você mesmo. Mas o fato de você estar lendo este livro comprova que, no dia-a-dia, a abordagem convencional nem sempre funciona. É aí que as coisas tornam-se interessantes. Tendo dito isso, quando você deve recorrer à substituição e ao patching de classes? As várias situações a seguir exigem uma abordagem de hacker:

- ▶ **Você utiliza uma biblioteca de terceiros que tem a capacidade necessária, mas que não é exposta por meio de uma API pública** – por exemplo, até o JDK 1.4, o Swing não fornecia um método de obter uma lista de *listeners* para um *JComponent*. O componente armazenava os listeners em uma variável com visibilidade de pacote, sem acesso público, e assim não havia como descobrir, de forma automatizada, se um componente possuía listeners.
- ▶ **Você utiliza uma classe ou interface de terceiros, mas a funcionalidade exposta não é suficientemente flexível para sua aplicação** – Uma alteração simples na API pode poupar dias de trabalho ou pode ser a única solução para o seu problema. Nesse caso, você está satisfeito com 99% da biblioteca, mas o 1% remanescente impede que você a utilize com eficácia.
- ▶ **Há um bug no produto ou na API utilizada e você não pode esperar que o fornecedor o corrija** – O JRun 3.0, por exemplo, tinha um bug na detecção de versão da JVM no HP UX. Ao processar a string de versão informada pelo Runtime Java, ele erroneamente concluía que estava rodando sob uma versão mais antiga do JDK e se recusava a rodar.
- ▶ **Você precisa de uma integração muito forte com um produto, mas sua arquitetura não é suficientemente aberta para satisfazer seus requisitos** – Muitos frameworks separam interfaces da implementação. Internamente, as interfaces são utilizadas para o acesso às funcionalidades e classes concretas são instanciadas para fornecer a implementação. A maioria das bibliotecas básicas de Java permite especificar as classes de implementação por meio de propriedades de sistema. Esse é o caso do AWT e do parser SAX, para os quais as classes de implementação podem ser especificadas utilizando as propriedades de sistema `java.awt.Toolkit` e `org.xml.sax.driver`, respectivamente. O hacking seria exigido se você precisasse fornecer uma classe de implementação diferente para uma biblioteca que não fornece um meio de personalização.
- ▶ **Você utiliza código de terceiros, mas a funcionalidade esperada não funciona** – Você não está seguro se isso ocorre porque não está utilizando o código corretamente, ou porque existe um bug no código. A documentação não faz referência ao problema e não há como contorná-lo. Inserir temporariamente rastreamentos e mensagens de depuração no código de terceiros pode ajudá-lo a entender o que está acontecendo no sistema.

► **Você tem um problema urgente de produção que precisa ser corrigido**

– Você também não pode arcar com o risco de reinstalar o novo código no ambiente de produção. A solução para o problema exige uma pequena alteração no código que afeta somente algumas classes.

Se você trabalha com código de terceiros, poderia violar o acordo de licenciamento. Portanto, por segurança, certifique-se de ler esse acordo e submetê-lo à análise do seu departamento jurídico. Leis de direitos autorais podem ser aplicadas, pois alterar código de terceiros costuma ser ilegal. Obtenha permissão do fornecedor para implementar uma solução, em vez de assumir a responsabilidade pela modificação. A boa notícia é que, utilizando o método apresentado neste capítulo, você não faz alterações diretas na biblioteca ou no produto utilizado. Você não está adulterando o código; em vez disso, está substituindo funcionalidades com as quais não está satisfeito. De certa maneira, é como derivar sua classe da classe do fornecedor para sobrescrever um método, embora isso possa ser arriscado. Questões legais à parte, vamos ver como você pode fazer isso.

## Localizando a classe em que é necessário aplicar um patching

Primeiro, você tem de determinar o código em que você precisa aplicar patching. Às vezes, isso é bastante fácil e você saberá qual é a classe ou a interface específica imediatamente. Se

### HISTÓRIAS DAS TRINCHEIRAS

A AT&T Wireless estava atualizando seu sistema de entrada de pedidos para ativação via telefones celulares. Sendo escrito em Java, o sistema teve de ser migrado do JDK 1.2 para o 1.3. A atualização era crucial pois havia muitas correções de bugs no Swing e em outros pacotes Java, e porque os usuários esperavam uma opção melhor. A melhoria de desempenho e a otimização no consumo de memória do JDK 1.3 estavam entre outros fatores importantes relacionados à decisão pela atualização. O desenvolvimento foi feito em sistemas Windows, mas o ambiente de produção era HP UX. Quando todos os bugs foram corrigidos e o sistema foi testado sob o JDK 1.3, a nova versão do Java foi instalada numa plataforma de homologação HP UX, para iniciar o teste de integração formal.

Infelizmente, descobriu-se que o servidor de aplicações utilizado para serviços J2EE tinha um bug na detecção da versão do JDK. Havia um erro ao se fazer o parse do string de versão a partir das propriedades de sistema, mas, como esse string era diferente no Windows e no Unix, o bug não apareceu até a fase de testes de integração. O servidor de aplicações recusava-se a ser executado no HP, acreditando estar rodando em uma versão anterior do Java. Nesse momento, era muito tarde para voltar ao JDK 1.2 e não havia como corrigir o problema. Para salvar o projeto, os engenheiros recorreram à descompilação da classe do servidor de aplicações que fazia a verificação de versão, e eles mesmos corrigiram o bug. Depois de o patching ser implantado, o servidor foi iniciado e funcionou perfeitamente em produção. Infelizmente, nenhum dos engenheiros ganhou uma viagem à Jamaica – como havia sido prometido pela gerência –, mas a vida é assim mesmo.

você dispensa saber sobre como localizar o código, então pode pular para a seção que discute como aplicar patch. Caso contrário, sente-se, relaxe e aprenda as várias abordagens para alcançar o resultado.

## A abordagem geral

O método geral de localizar uma classe a que se deve aplicar um patch consiste em encontrar um ponto de partida e navegar pela seqüência de execução até obter o código que se quer alterar. Se você não encontrar o código a ser alterado em um local próximo desse ponto de partida, deve obter um novo ponto de partida e repetir o processo. Um bom ponto de partida é crucial para resultados rápidos. Às vezes, selecionar uma classe para começar é simples. Por exemplo, para patching de API ou de lógica, o ponto de partida seria a interface ou a classe a ser alterada. Se você quiser tornar um método privado de uma classe um método público, o ponto de partida é a classe em questão. Se precisar corrigir um bug que resulta em uma exceção Java, o ponto de partida é a classe no topo do *stack trace*.

Independentemente da situação, depois de estabelecer uma classe inicial você deve obter o código-fonte (descompilando o bytecode se for preciso) e, se necessário, percorrer o código até a classe que realmente precisa ser corrigida com patch. Esse processo é semelhante a percorrer um diagrama de seqüência, iniciando do método recém-descrito e examinando cada classe invocada no caminho. Em sistemas grandes com centenas de classes, você talvez tenha de identificar vários pontos de partida e selecionar o que fornece a rota mais curta para o código que precisa ser alterado.

## Procurando strings de texto

Um sistema grande e sofisticado possui dezenas de pacotes e centenas de classes. Se não houver um ponto de partida claro, você poderá facilmente se perder ao tentar percorrer a lógica da aplicação. Imagine o código de inicialização para um servidor de aplicações como o WebLogic. Durante a inicialização, o WebLogic realiza centenas de tarefas e utiliza muitos threads para executá-las – e mesmo com um estoque ilimitado de cafeína, eu não o aconselharia a tentar quebrá-lo percorrendo o código a partir da classe `weblogic.Server`.

A abordagem mais confiável nesses casos é uma pesquisa por strings que se sabe estarem próximos da classe-alvo. Bibliotecas e produtos bem escritos podem ser configurados para produzir informações extensas de depuração em um arquivo de log. Além dos benefícios óbvios na manutenção e na solução de problemas, isso torna mais fácil localizar o código responsável pela funcionalidade em questão. Quando você configura a aplicação para gravar um log detalhado da seqüência de execução e ocorre um problema em algum lugar, você pode utilizar a última mensagem de log bem-sucedida (ou a primeira errônea) para identificar o ponto de entrada. Como o bytecode armazena strings como textos simples, você pode pesquisar em uma substring todos os arquivos `.class` que você viu em um arquivo de log. Suponha que, ao utilizar o framework de segurança, uma exceção com o texto `Invalid username` seja lançada para alguns nomes. A razão para a rejeição desses nomes

e a solução para o problema são desconhecidas. A maneira mais fácil de chegar ao código se o *stack trace* não estiver disponível é procurando `Invalid username` em todos os arquivos.class do framework. Mais provavelmente, o erro estará em uma ou duas classes no código inteiro e, descompilando o arquivo de classe, você será capaz de entender a raiz do problema. Da mesma forma, você pode pesquisar em todos os arquivos de classes um nome de método ou de classe, um *label* de interface gráfica, uma substring de uma página HTML ou qualquer outra string que possa estar incorporada ao código Java.

### Trabalhando com código ofuscado

Um cenário pior é quando você tem de lidar com o código ofuscado. Um bom ofuscador renomeia pacotes, classes, métodos e variáveis. Os melhores produtos no mercado até codificam strings em Java, de modo que procurar uma mensagem de rastreamento pode não produzir resultado algum. Isso torna infernal a tarefa de entender a aplicação pedaço por pedaço. Aqui você tem de utilizar uma abordagem mais criativa; do contrário, é como tentar localizar uma agulha em um palheiro. Conhecer os princípios sobre ofuscamento pode ajudá-lo na navegação. Embora o ofuscador tenha a liberdade de alterar nomes de métodos e de classes da aplicação, ele não pode fazer isso para classes de sistema. Por exemplo, se uma biblioteca verificar a presença de um arquivo e lançar uma exceção indicando que o arquivo não está lá, fazer uma pesquisa binária na string de exceção talvez não produza nenhum resultado, se o ofuscador for suficientemente inteligente para codificá-la. Entretanto, fazer uma pesquisa em `File` ou em `FileInputStream` pode levá-lo ao código relacionado. De maneira semelhante, se a aplicação ler incorretamente a data ou hora do sistema, você poderá procurar pelo método `java.util.Date` ou `getTime` da classe `Calendar`. O maior problema é que classes ofuscadas nem sempre podem ser recompiladas depois da descompilação. Consulte o Capítulo 2 para informações adicionais.

## Exemplo de cenário que requer patching

Iremos modificar a aplicação Chat apresentada anteriormente para mostrar o nome de usuário e o hostname, em vez de somente o hostname, na janela de chat. Como você vai

lembra, a aplicação original exibe o hostname seguido por dois-pontos para cada mensagem recebida, como mostrado na Figura 5.1.

Isso facilita a implementação do utilitário, mas os usuários certamente irão preferir ver de quem as mensagens estão chegando, em vez de saber o computador utilizado para enviar as mensagens. A aplicação Chat é gratuita e aberta a aprimoramentos, mas sem código-fonte.

Como é comum com aplicações em Java, o bytecode é distribuído em um ou vários arquivos JAR; assim a primeira ta-



**FIGURA 5.1** Janela principal da aplicação Chat original.

refa é criar um diretório de trabalho e descompactar nele todas as bibliotecas dentro do JAR . Isso facilita a navegação e o acesso direto a arquivos.class, que serão o alvo da nossa pesquisa. Depois de criar um diretório de trabalho executando jar xf chat.jar, veremos os arquivos a seguir:

```
images
AboutDialog.class
ChatApplication.class
ChatServer.class
ChatServerRemote.class
MainFrame.class
MainFrame$1.class
MessageInfo.class
MessageListener.class
```

Vamos tentar todas as abordagens para localizar o ponto de partida apresentado anteriormente e ver qual funciona melhor para essa aplicação.

## Utilizando o nome de classe

Felizmente, o bytecode não está ofuscado, portanto podemos examinar os nomes de classes e escolher o vencedor. Um exame de cinco segundos deve levar à conclusão de que MainFrame é o melhor candidato para começar. Navegando pelo código descompilado, vemos que todas as gravações de conversas são realizadas com o método appendMessage e têm a seguinte aparência:

```
void appendMessage(String message, MessageInfo messageInfo) {

    if (messageInfo == null) {
        this.conversation.append("<font color=\"red\">");
        this.conversation.append("You");
    }
    else {
        this.conversation.append("<font color=\"blue\">");
        this.conversation.append(messageInfo.getDisplayName());
    }
    this.conversation.append(": ");
    this.conversation.append("</font>");

    this.conversation.append(message);
    this.conversation.append("<br>");
    this.txtConversation.setText(this.conversation.toString() +
        "</BODY></HTML>");
}
```

A implementação do método utiliza o método `getDisplayName()` da classe `MessageInfo` para obter o nome do remetente. Isso nos leva a descompilar a classe `MessageInfo` para obter a implementação de `getDisplayName` mostrada aqui:

```
public String getDisplayName() {  
    return getHostName();  
}
```

Bingo! Descobrimos que a interface gráfica da aplicação Chat depende de `MessageInfo` e que a implementação atual utiliza apenas o hostname. Portanto, nossa tarefa é aplicar um patch a `MessageInfo.getDisplayName()`, para que ele utilize tanto o hostname como o nome do usuário.

## Procurando strings de texto

Suponhamos que Chat seja uma grande aplicação com mais de 500 classes em muitos pacotes diferentes. Esperar que seja possível adivinhar a classe correta com base no seu nome é como esperar que seu código seja executado corretamente depois da primeira compilação. Você precisa utilizar um método mais confiável para obter um ponto de partida. O utilitário Chat grava mensagens de log de boa qualidade, e assim vamos tentar utilizá-lo. Depois de iniciá-lo, enviamos uma mensagem a outro usuário, obtemos uma resposta e recebemos a seguinte saída no console Java:

```
Initializing the chat server...  
Trying to get the registry on port 1149  
Registry was not running, trying to create one...  
ChatApplication server initialized  
Sending message to host JAMAICA: test  
Received message from host JAMAICA
```

Não é difícil concluir que acrescentar uma mensagem ao histórico de conversas ocorre quando uma mensagem é enviada ou recebida. Também é claro que informações como o host que enviou a mensagem ou foi o destinatário da mensagem não seria parte de uma string estática. Portanto, utilizaremos `Received message from host` como critério de uma pesquisa em todos os arquivos .class no diretório de trabalho. A pesquisa produz um arquivo, `ChatServer.class`, o qual prontamente descompilamos para obter `ChatServer.jad`. Pesquisando a string dentro do código-fonte descompilado, chegamos ao método `receiveMessage()`, que é o seguinte:

```
public void receiveMessage(String message, MessageInfo messageInfo)  
    throws RemoteException  
{  
    System.out.println("Received message from host " + messageInfo.getHostName());  
    if(messageListener != null)  
        messageListener.messageReceived(message, messageInfo);  
}
```

Procurando em ChatServer.jad por messageListener, descobrimos que ele é uma interface e que um método chamado setMessageListener( ) configura a instância do listener. Agora temos duas opções: uma é encontrar as classes que implementam MessageListener e verificar qual (se houver várias) está associada com ChatServer. Outra abordagem baseia-se no fato de nomes de métodos Java serem armazenados como texto dentro do bytecode. Como o código não está ofuscado, podemos procurar setMessageListener( ) em todos os arquivos de classes. Utilizaremos o último método e executaremos a pesquisa. No nosso caso, o método retorna duas classes, ChatServer e MainFrame. Concluímos que somente MainFrame atua como listener em ChatServer e prosseguimos para descompilá-lo. O restante da investigação é realizado exatamente como na seção anterior: utilizamos o nome de classe para localizar MainFrame. Na nossa aplicação de exemplo, adivinhar o ponto de partida a partir do nome da classe mostrou-se mais rápido, mas também contamos com a sorte. Utilizar mensagens de log é uma abordagem mais confiável, que funciona melhor para a maioria das aplicações do dia-a-dia.

## Utilizando a pilha de chamadas para navegar pela lógica da aplicação

Muitos problemas em Java se manifestam por meio de exceções. Exceções podem ser lançadas por classes do JRE ou pela própria aplicação e, é claro, a mensagem de erro fornecida pela exceção, juntamente com o tipo de exceção, é geralmente suficiente para resolver o problema. Mas a razão de este livro existir é porque na vida nem todas as coisas são simples. Pode ocorrer uma NullPointerException ou uma exceção sem mensagem de erro e, se você estiver lidando com código de terceiros, não terá pistas de como contornar isso. Contanto que a licença não impeça que você descompile o código-fonte ou, se tiver seu próprio código-fonte, você poderá começar uma pesquisa utilizando uma técnica bem menos trabalhosa.

A maneira mais fácil e a mais garantida de entender o que causa uma exceção é analisando a pilha de chamadas. Você deve estar ciente de que os sistemas operacionais utilizam uma pilha para monitorar as chamadas de método. Se o método A chamar o método B, as informações de A são colocadas na pilha. Se, depois, o método B chama C, as informações de B também são colocadas na pilha. À medida que cada método retorna, a pilha é utilizada para determinar o método que deve retomar a execução. De qualquer maneira, em Java você pode acessar a pilha de chamadas por meio de um depurador (chamando printStackTrace( ) na exceção) ou utilizando o método Thread.dumpStack( ). Normalmente, a utilização de um depurador mostra-se muito trabalhosa para aplicações servidoras; assim, para nosso exemplo, sugiro a utilização dos dois últimos métodos. O método printStackTrace( ) de Exception é amplamente utilizado e não deve surpreender ninguém. O método dumpStack de Thread não é tão comum, mas pode ser extremamente útil para ver quem está chamando um método em tempo de execução. Basta adicionar

```
Thread.dumpStack();
```

ao corpo do método que você está investigando e executar a aplicação. Toda vez que o método for chamado, você saberá exatamente como a execução chegou a ele e que outros métodos precisarão inspecionar.

A pilha de chamadas é abordada em mais detalhes no Capítulo 16.

## Aplicando um patch a uma classe para adicionar nova lógica

Agora que você sabe o que precisa ser corrigido, o trabalho real se torna relativamente fácil. Obtenha o arquivo-fonte diretamente da distribuição ou descompilando o arquivo .class (certifique-se de que a licença não proíbe isso). Para facilidade de manutenção, você deve manter todas as suas classes corrigidas com patch em um diretório separado – por exemplo, em patches. Recrie a estrutura de diretórios para que ela corresponda à estrutura de pacotes da classe e utilize seu IDE favorito, ou seu bom e velho editor de textos e compilador de linha de comando, para fazer as alterações na classe. Saiba que possivelmente será necessário atualizar a biblioteca em algum momento, e assim você deve escrever comentários para cada fragmento de código inserido. Dessa maneira, ao obter a nova versão da classe original, você poderá repetir os passos facilmente. Pela mesma razão, você deve isolar suas alterações e agrupá-las no arquivo. Se você estiver adicionando uma quantidade razoável de código, considere criar uma classe auxiliar e delegar para ela, de modo que seja feito o menor número possível de alterações no arquivo corrigido.

No nosso exemplo, devemos fornecer uma nova implementação para o método `getDisplayName()`, que utilizará o `UserName (HostName)` padrão. Em seguida, criamos um novo diretório chamado `patches` com um subdiretório chamado `covertjava.chat` no diretório de instalação da aplicação e copiamos `MessageInfo.jad` para ele, renomeando-o para `MessageInfo.java`. `MessageInfo` já contém o método `getUserName()`, e assim nossa tarefa é apenas concatenar as duas strings. Reescrevemos o método `getDisplayName()` para que ele se pareça com o seguinte:

```
public String getDisplayName() {
    return getUserName() + " (" + getHostName() + ")";
// *** lógica original corrigida por Alex Kalinovsky para atender a novos requisitos
//     return getHostName();
}
```

Então compilamos `MessageInfo.java`. Feito isso, agora podemos atualizar a aplicação para colocar em efeito a nova lógica. Devemos também documentar nossas alterações de modo que possamos dar manutenção no código no futuro.

## Reconfigurando a aplicação para carregar e utilizar a classe com patch

Essa é a tarefa que na verdade faz o truque funcionar. Precisamos assegurar que a JVM utilize a versão corrigida da classe que fornecemos em vez da versão antiga. Isso nos leva à pedra angular da programação Java, que é configurar o `CLASSPATH` correto. Sim, é simples assim. Você só precisa assegurar que o arquivo JAR ou ZIP, ou o diretório contendo a versão corrigida da classe, apareça no caminho de pesquisa de classe *antes* do arquivo ou diretório que contém a versão original. No arquivo de script que inicializa sua aplicação e configura seu `CLASSPATH`, você deve tornar esse arquivo ou diretório a primeira entrada. Agrupar todos os seus patches e mantê-los separados dos produtos e bibliotecas que eles

corrigem facilita a manutenção. Ao obter uma nova versão de uma biblioteca, você pode sobrescrever os arquivos JAR antigos na biblioteca, sem medo de perder suas correções. É claro que você deve realizar testes novamente e, como o patch não depende apenas de interfaces públicas, mas também de implementações privadas, você talvez precise atualizar seu patch de acordo com a nova biblioteca. Finalmente, se você precisar explicar a alguém o que fez com o sistema, poderá apontar para o diretório patches e para a documentação.

Como dizem: “As aparências enganam”. Às vezes você poderia *achar* que o sistema está carregando sua nova classe, mas, de fato, a versão antiga é carregada. A melhor maneira de garantir que a nova versão seja utilizada é adicionando um rastreamento de depuração ou mesmo um simples `System.out.println()` à nova classe. Ao executar a aplicação, certifique-se de que é possível ver o rastreamento; depois disso você pode removê-lo. No nosso código de exemplo, adicionamos um inicializador estático que imprime uma mensagem indicando que os patches estão em funcionamento, como mostrado no código a seguir:

```
static {
    // Registre em log o fato de que o patch está em efeito
    System.out.println("Loaded patched MessageInfo");
}
```

Para ver nossas alterações em ação, atualizamos o script inicial da aplicação Chat a fim de incluir o diretório patches. Copiamos bin/chat.bat para bin/chat\_patched.bat e abrimos esse arquivo em um editor. Então alteramos a inicialização do CLASSPATH, de modo que inclua o diretório patches antes do arquivo chat.jar da aplicação:

```
set CLASSPATH=..\patches;..\\lib\\chat.jar
```



**FIGURA 5.2** Janela principal da aplicação Chat após o patching.

Em seguida, salvamos e executamos o arquivo. Enviamos algumas mensagens de testes a localhost e, se tudo for feito corretamente, a nova tela deverá se parecer com a Figura 5.2.

Se a classe a que você estiver aplicando um patch for uma classe de sistema, o trabalho será mais difícil. A classe de sistema é aquela que é carregada do *boot classpath* – por exemplo, `java.lang.String`. Mesmo se você colocar a versão corrigida com um patch da classe em primeiro lugar no CLASSPATH, ela não substituirá a classe original. Aprenda a aplicar patches a classes de sistema no Capítulo 15.

## Aplicando patch a pacotes selados

O Java suporta a noção de pacotes *selados* (*sealed*). Se um pacote estiver selado, todas as classes nesse pacote devem ser carregadas do mesmo arquivo JAR. Para desenvolvedores de aplicações e fornecedores de ferramentas, isso é um recurso excelente concebido para pre-

venir o uso da técnica que você acabou de conhecer. Para selar um pacote, você deve especificar o atributo Sealed com true no arquivo de manifesto do JAR. Se o arquivo chat.jar da aplicação Chat for selado, executar bin\chat\_patched.bat produz a seguinte exceção:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
  at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
  at sun.reflect.NativeConstructorAccessorImpl.newInstance
  ↪(NativeConstructorAccessorImpl.java:39)
  at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
  ↪(DelegatingConstructorAccessorImpl.java:27)
  at java.lang.reflect.Constructor.newInstance(Constructor.java:274)
  ...
Caused by: java.lang.SecurityException: sealing violation: package covertjava.chat is
↪sealed
  at java.net.URLClassLoader.defineClass(URLClassLoader.java:225)
  at java.net.URLClassLoader.access$100(URLClassLoader.java:54)
  at java.net.URLClassLoader$1.run(URLClassLoader.java:193)
  at java.security.AccessController.doPrivileged(Native Method)
```

Infelizmente para os fornecedores e felizmente para os hackers e para pessoas inquisidoras como nós, pacotes selados costumam ser fáceis de quebrar. Tudo o que você precisa fazer é alterar o valor do atributo Sealed para false dentro do manifesto do JAR, ou extrair o conteúdo do JAR em um diretório de trabalho e modificar o script de inicialização para utilizar esse diretório em vez do arquivo JAR original. Esse é um exemplo de uma boa intenção que nunca foi implementada seriamente.

## Questionário rápido

1. Quais abordagens podem ser utilizadas para localizar uma classe a que deve ser aplicado um patch?
2. Por que é possível utilizar uma pesquisa textual para localizar uma classe, e quando isso não funcionaria?
3. Como você pode obter uma pilha de chamadas para um ponto arbitrário na aplicação (fora de um bloco catch)?
4. Quais são os requisitos para instalar um patching?

## Resumo

Para aplicar um patch à lógica da aplicação, você precisa:

- localizar a classe que contém a lógica;
- obter o código-fonte da classe;
- modificar o fonte para implementar a nova lógica;
- compilar e instalar o novo arquivo de classe;
- atualizar o ambiente de inicialização da aplicação para primeiro carregar a nova classe.

# 6

## NESTE CAPÍTULO

- ▶ Introdução ao rastreamento 56
- ▶ Rastreamento como um método eficaz de conhecimento do software 57
- ▶ APIs e ferramentas de rastreamento e logging 58
- ▶ Rastreamento: recomendações e cuidados 59
- ▶ Questionário rápido 60
- ▶ Resumo 60

# Utilizando rastreamento eficaz

*"Invariavelmente se descobre que um sistema complexo que não funciona evoluiu de um sistema simples que funcionava perfeitamente."*

As Leis de Murphy aplicadas à tecnologia

## Introdução ao rastreamento

O *Rastreamento* consiste em gravar mensagens de depuração em um stream de saída durante a execução de uma aplicação. Ele fornece uma gravação das operações realizadas pela aplicação em execução, em tempo real. O rastreamento não requer que a aplicação esteja rodando em modo de depuração. Além disso, como as mensagens de rastreamento geralmente são gravadas em um arquivo de log, o rastreamento fornece informações para analisar o comportamento da aplicação e solucionar problemas. O rastreamento é utilizado há vários anos, mas tornou-se especialmente popular na programação do lado do servidor. Se a aplicação for pequena, ela poderá facilmente ser carregada em um depurador e inspecionada passo a passo para investigar um problema. Sistemas distribuídos, no entanto, tendem a ser muito mais complexos no lado do servidor e costumam ser executados dentro de produtos middleware, como um servidor Web ou de aplicações. Isso torna a utilização de um depurador impraticável, e fazer uma inspeção passo a passo no código pode interferir com o multithreading. A depuração não é absolutamente uma opção para aplicações em produção, portanto nesses casos o rastreamento é o único método plausível de fornecer informações sobre o que está acontecendo no sistema.

Os rastreamentos são inseridos no código de forma similar a comentários; um bom rastreamento torna os

comentários desnecessários. Como o rastreamento resulta em chamadas de métodos e em mensagens gravadas, o processo pode tornar-se custoso – especialmente se colocado dentro de loops e em métodos chamados com freqüência. Frameworks de logging são projetados para adicionar o mínimo de overhead à execução do código, e podem ser configurados para reduzir o nível de mensagens de rastreamento gravadas. É importante perceber, porém, que, mesmo que o rastreamento não esteja bem otimizado, ele ainda é melhor que nenhum rastreamento. Se houver um problema em uma aplicação em produção que não tenha rastreamentos, a única fonte de informações será o feedback do usuário, que, como sabemos, não costuma ser confiável ou preciso. Uma boa utilização de rastreamentos permite monitorar a ação do usuário e a resposta da aplicação a ela, o que, na maioria dos casos, deve ser adequado para identificar a origem do problema. Você deve incluir algum tipo de identificação de usuário em cada rastreamento, de modo que se possa filtrar o arquivo de log para operações realizadas por um usuário específico.

O Java não tinha uma API padrão de logging até o JDK 1.4. É uma pena, dada a popularidade e a importância de logging em aplicações J2EE. Como resultado dessa padronização tardia, cada fornecedor e muitos desenvolvedores de aplicações tiveram de criar suas próprias versões de APIs de rastreamento e logging. Hoje é comum se encontrar várias versões de APIs de logging, às vezes até mesmo na mesma aplicação. Por exemplo, uma aplicação J2EE em execução dentro do WebLogic pode utilizar o Log4J para seus rastreamentos de depuração e utilizar a API de logging do WebLogic para informar erros críticos no nível de sistema. Todas as APIs de logging suportam o conceito de níveis de mensagens de log, que controla o número de mensagens gravado no arquivo de log.

## Rastreamento como um método eficaz de conhecimento do software

Já discutimos a importância do rastreamento na solução de problemas de grandes sistemas. As informações fornecidas nos logs de rastreamento também são valiosas para se entender o fluxo de execução de uma aplicação e para se identificar pontos de partida para a engenharia reversa. Como os rastreamentos são projetados para fornecer um histórico das operações realizadas por um produto, de forma legível por seres humanos, são mais fáceis de ler do que código Java descompilado. O arquivo de log apresenta um registro instantâneo do trabalho feito até o momento. Portanto, se um bug resultar em uma exceção sem *stack trace*, você poderá ativar o nível mais detalhado de rastreamento, replicar o erro e localizar a última mensagem que o produziu no arquivo de log antes da exceção. A mesma técnica pode ser utilizada para localizar o código em que é necessário aplicar um patch. Se a aplicação não fornecer rastreamento adequado, você pode adicionar mensagens de rastreamento e até dumps da pilha de thread a classes, utilizando patches. Embora trabalhosa, essa pode ser a única maneira efetiva de analisar código ofuscado, especialmente se o ofuscamento do fluxo de controle tiver sido utilizado. O uso do rastreamento não dá espaço a suposições, pois mensagens de rastreamento fornecem provas indiscutíveis de que um determinado trecho de código foi executado em um dado momento.

Se a aplicação não está imprimindo mensagens de log, inserir rastreamentos pode demorar, e – vamos admitir – não é o tipo de desafio que um bom desenvolvedor está pro-

curando. Uma maneira “rápida e suja” de examinar o estado da execução é imprimindo a pilha de chamadas do thread atual, a partir de um método sob investigação, como descrito no Capítulo 5. Lembre que adicionar uma chamada ao método `dumpStack()` de `java.lang.Thread` faz com que sejam impressos os nomes dos métodos que constituem a pilha de chamadas. Com apenas uma linha de código, você pode ter um bom entendimento das chamadas que resultaram na invocação do método em questão.

## APIs e ferramentas de rastreamento e logging

Hoje há duas APIs de logging dominantes. A primeira é a Log4J da Apache, iniciada em 1999 pelo grupo AlphaWorks da IBM. Ela percorreu um longo caminho e hoje é provavelmente a API de logging mais avançada e flexível. É gratuita e pode ser distribuída como um arquivo JAR independente, que é compatível com todas as versões do JDK a partir da 1.1. Devido ao reconhecimento da marca do grupo Apache e do excelente conjunto de recursos encontrado na Log4J, a biblioteca ganhou enorme popularidade entre os desenvolvedores Java e pode ser considerada um padrão de mercado para logging em Java.

O segundo framework é a Java Logging API da Sun, que foi adotada como o padrão oficial. Para melhor ou para pior, o JCP decidiu não utilizar o Log4J, mas criar um novo padrão. A Java Logging API também é poderosa e flexível, embora não tenha alguns recursos avançados encontrados na Log4J, como:

### HISTÓRIAS DAS TRINCHEIRAS

Há alguns anos, fui chamado pelo nosso principal cliente para fazer uma avaliação do estado atual do seu projeto e fornecer recomendações sobre aprimoramentos. As coisas não estavam indo bem no lado do cliente e tanto a gerência como a equipe de desenvolvimento estavam frustradas com o progresso lento e a incapacidade de corrigir defeitos rapidamente. Uma análise do código revelou que não havia utilização sistemática de rastreamentos. No melhor dos casos, os desenvolvedores utilizavam um `System.out` ocasional para imprimir o valor de uma variável, mas a aplicação em execução parecia um buraco negro – ninguém sabia o que estava acontecendo em um dado momento. Mesmo um problema simples era difícil de corrigir, porque as condições que resultaram nele, e às vezes mesmo a origem do problema, eram desconhecidas.

Minha primeira recomendação foi, portanto, investir três dias inserindo logging no código. Toda a equipe parou de fazer as tarefas atuais e começou a analisar o código, adicionando rastreamentos que imprimiam os nomes dos métodos sendo chamados e os parâmetros de negócio, como nomes de usuários, números de pedidos e IDs de registros no banco de dados. Esses três dias foram recompensados quase imediatamente. Com o rastreamento, identificar e corrigir defeitos tornou-se fácil, mesmo sem precisar depurar o código. Mesmo que parte da aplicação tivesse sido escrita por uma outra pessoa, os rastreamentos forneciam informações sobre o processo de execução. E o que é mais importante, quando o sistema foi colocado em produção, o arquivo de log foi a principal fonte de informações sobre o estado atual do sistema e das operações realizadas no dia.

- Logging em logs de sistema como o log de eventos do Windows NT e o Unix Syslog
- Rotação de arquivos de log com base em data
- Padrões de formatação como o printf da linguagem C
- Recarregamento do arquivo de configuração

O recarregamento de arquivo de configuração é um recurso importante para aplicações de alta disponibilidade. Em geral, a configuração é lida quando um framework é inicializado, o que normalmente ocorre na inicialização da aplicação. Alterações nos níveis e categorias de logging não entram em ação até que a aplicação seja reinicializada. Ao solucionar problemas de sistemas em produção, não são possíveis reinicializações freqüentes e, portanto, sem o recurso de recarregamento, o nível de logging tem de ser estabelecido antes de se iniciar a aplicação.

Para os propósitos de hacking e engenharia reversa, a biblioteca Log4J é claramente a melhor opção, pois pode ser utilizada com praticamente todas as versões do JDK. O download e documentação on-line estão disponíveis em <http://jakarta.apache.org/log4j>.

## Rastreamento: recomendações e cuidados

O rastreamento é extremamente importante para grandes aplicações distribuídas, mas, se não for utilizado cuidadosamente, poderá degradar significativamente o desempenho da aplicação. A seguir há uma série de regras simples para um rastreamento eficaz.

### Recomendações no rastreamento

- Utilize o rastreamento o máximo possível.
- Certifique-se de que cada rastreamento inclua a data/hora, classe e nome do método que o produziu.
- Escreva rastreamentos antes e depois de partes críticas do código, como chamadas de sistema externas, chamadas ao banco de dados e acesso a disco. Isso permite identificar facilmente o ponto de falha.
- Inclua valores dos parâmetros e variáveis de contexto que ajudarão a entender o contexto de execução no momento do rastreamento. Por exemplo, incluir o ID de usuário em cada rastreamento ajuda a filtrar o arquivo de log nas operações realizadas para um usuário específico.
- Gere o rastreamento de exceções incluindo o *stack trace* quando a exceção é capturada pela primeira vez.
- Certifique-se de utilizar diferentes níveis de rastreamentos ao gravar mensagens. Utilize o nível crítico para mensagens de erros e de sistema e o nível de depuração para mensagens que não são muito importantes na solução de problemas do sistema. Isso permite controlar o tamanho do arquivo de log e o overhead na aplicação.

## Cuidados no rastreamento

- Não utilize `System.out.println` para rastreamentos. Isso impõe overheads permanentes na aplicação e não permite flexibilidade. Utilize, em vez disso, um framework de logging.
- Não utilize `Exception.printStackTrace()` para gerar a saída de uma exceção, pois ele sempre grava a saída em `System.out`. Utilize métodos fornecidos pelo framework de logging.
- Não insira rastreamentos em loops com centenas ou milhares de repetições.
- Não insira rastreamentos em pequenos métodos chamados freqüentemente.

Outra prática a ser evitada é a utilização do operador `+` para concatenar strings ao passar parâmetros para o framework de logging. Mesmo que o nível de rastreamento seja aumentado, o tempo de execução ainda realizará a operação custosa de alocar um novo buffer para a string resultante e de copiar para ele as strings dos argumentos. Dessa forma, em vez de utilizar

```
logger.debug("Message received from host: " + host + ", text: " + text)
```

utilize

```
if (logger.isDebugEnabled() == true)  
    logger.debug("Message received from host: " + host + ", text: " + text)
```

O custo da execução do segundo trecho de código é muito mais baixo porque a concatenação de strings não acontece, a menos que os rastreamentos de depuração estejam ativados.

## Questionário rápido

1. Como o rastreamento difere da depuração?
2. Como o rastreamento pode ser utilizado para hackear uma aplicação?
3. Por que o recarregamento da configuração de logging em tempo de execução é importante?
4. Por que utilizar `+` para concatenação de strings é perigoso?

## Resumo

- O rastreamento consiste em gravar mensagens de depuração em um stream de saída durante a execução de uma aplicação.
- Os rastreamentos são inseridos como chamadas de APIs no código das aplicações.
- Rastreamentos adicionam overhead, mas isso é justificado pelo ganho na solução de problemas.
- Adicionar rastreamentos a uma aplicação ajuda a entender seu fluxo de execução.
- A Log4J e a Java Logging API são duas boas opções de frameworks de logging. A Log4J é mais avançada e pode ser utilizada com JDKs antigos, o que a torna uma melhor opção para hacking.

# Manipulando a segurança Java

*"Um especialista é aquele que aprende cada vez mais sobre cada vez menos, até saber absolutamente tudo sobre nada."*

As Leis de Murphy aplicadas à tecnologia

## Visão geral sobre a segurança em Java

Desde o início, o Java foi desenvolvido para ser seguro. Segurança inclui recursos de linguagem, como verificações de limites de índices de arrays, verificação de bytecode e controle de acesso a recursos críticos do sistema, como arquivos e informações de usuário. No início, o modelo de segurança de Java pressupunha que todas as classes instaladas localmente deveriam ter acesso a recursos locais, e que todo bytecode carregado através da rede deveria ter excluído o acesso a informações e operações sigilosas. À medida que o Java amadureceu, a distinção entre classes locais e remotas tornou-se menos definida. Servidores de aplicações e servidores Web dependem de um modelo de segurança para garantir que um componente, como uma aplicação Web, não viole outros componentes. Applets assinadas permitem que código baixado da rede acesse arquivos locais, a área de transferência e propriedades de sistema.

Várias classes básicas em tempo de execução utilizam o framework de segurança para verificar se o chamador pode realizar a operação solicitada. No centro do modelo de segurança há uma instância de `java.lang.SecurityManager` que funciona como uma fachada para o pacote `java.security`. A plataforma Java utiliza o conceito de *permissão* para representar o acesso a um recurso ou a informações de sistema. Por exemplo, `PropertyPermission` representa o acesso a propriedades

# 7

## NESTE CAPÍTULO

- ▶ Visão geral sobre a segurança em Java 61
- ▶ Driblando verificações de segurança 63
- ▶ Questionário rápido 65
- ▶ Resumo 65

de sistema. Antes de executar a lógica de métodos, classes básicas utilizam o gerenciador de segurança para verificar se são concedidas ao chamador as permissões apropriadas. Uma maneira fácil de entender como esse mecanismo funciona é examinando a implementação do método `System.setProperty()`, mostrada na Listagem 7.1.

#### **LISTAGEM 7.1** `System.setProperty`

```
public static String setProperty(String key, String value) {
    if (key == null) {
        throw new NullPointerException("key can't be null");
    }
    if (key.equals("")) {
        throw new IllegalArgumentException("key can't be empty");
    }
    if (security != null)
        security.checkPermission(new PropertyPermission(key, "write"));
    return (String) props.setProperty(key, value);
}
```

A implementação primeiro obtém o gerenciador de segurança do sistema. Se um gerenciador estiver instalado e os parâmetros forem válidos, o código então utiliza o gerenciador de segurança para verificar se a `PropertyPermission` para gravação na chave dada foi concedida. O método `checkPermission` do gerenciador lança uma exceção `SecurityException` se a permissão não for concedida. A propriedade de sistema é configurada com o novo valor somente se `checkPermission` retornar com sucesso. A Tabela 7.1 lista as operações administradas pelo gerenciador de segurança.

**TABELA 7.1**

#### **Operações protegidas pelo gerenciador de segurança**

TIPO	OPERAÇÕES PROTEGIDAS
Sistema	Acesso a pacotes Acesso a variáveis-membro e métodos de uma classe Carregamento de bibliotecas nativas Encerramento do sistema Inicialização e interrupção de threads Criação de <i>class loaders</i> personalizados
Entrada e saída, acesso à rede	Criação, remoção, leitura e escrita de arquivos Navegação por diretórios Criação de sockets Carregamento de classes de uma URL de rede
AWT/Swing	Acesso à área de transferência Acesso à fila de eventos de sistema Exibição de janelas sem uma mensagem de alerta Obtenção de um trabalho de impressão

Informações sobre quais permissões são concedidas a quais classes são armazenadas nos arquivos políticas de Java. O arquivo de sistema `java.policy` é carregado primeiro a partir do diretório  `${java.home}/lib/security`, onde  `${java.home}` é o diretório de instalação do JRE. Se ele existir, o arquivo  `${user.home}/.java.policy` é carregado em seguida. Um arquivo de políticas personalizado pode ser especificado na linha de comando usando o parâmetro `-Djava.security.policy`. Uma discussão detalhada sobre a segurança Java e permissões pode ser encontrada em <http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html>.

O framework de segurança Java inclui APIs para criptografia (JCE), autenticação e autorização (JAAS), suporte a sockets seguros (JSSE) e muitos outros. Essas APIs adicionais fornecem um meio poderoso de proteger informações, mas não são utilizadas diretamente em uma aplicação comum. Focalizaremos os recursos de segurança com os quais um desenvolvedor de aplicações provavelmente irá se confrontar.

## Driblando verificações de segurança

No contexto deste livro, estamos principalmente interessados em driblar as verificações de segurança realizadas pelo runtime Java. Os três cenários de configuração que determinam o nível de rigor do modelo de segurança são:

- o gerenciador de segurança não está instalado.
- o gerenciador de segurança está instalado com uma política padrão.
- o gerenciador de segurança está instalado com uma política personalizada.

### HISTÓRIAS DAS TRINCHEIRAS

Planejávamos utilizar uma tecnologia de distribuição independente para colocar nossa aplicação Swing em milhares de desktops de usuários. O produto que utilizariamos suportava atualizações automáticas, o que exigia que a aplicação fosse iniciada pelo lançador do produto, e não como uma aplicação independente. Para implementar a atualização automática via HTTP, o produto instalava seu próprio handler HTTP, que entrava em conflito com o handler HTTP com suporte a SSL utilizado pela nossa aplicação. A plataforma Java não suporta a alternância dinâmica dos handlers de protocolos, porque armazena e reutiliza as instâncias de implementação em um cache privado. A única maneira de contornar o problema era limpar o cache à força na inicialização da aplicação. Utilizamos a técnica de acesso a membros privados descrita no Capítulo 4, que requer um gerenciador de segurança e um arquivo de políticas personalizado. Distribuímos o arquivo de políticas, permitindo acesso irrestrito a membros de classes via a API de reflection, com nossa aplicação; depois de limpar o cache na inicialização, permitimos que o produto coexistisse com nosso código.

## O gerenciador de segurança não está instalado

Se o gerenciador de segurança não estiver instalado, as verificações simplesmente não serão realizadas. Como vimos na Listagem 7.1, as verificações de permissões ocorrem somente se a instância do gerenciador de segurança do sistema não for nula. Por padrão, aplicações Java (mas não applets ou aplicações Web Start) são executadas sem um gerenciador de segurança, o que significa que o código tem acesso praticamente irrestrito às informações de sistema e recursos. Por exemplo, sem um gerenciador de segurança, qualquer classe pode ler o nome da conta de usuário a partir da propriedade de sistema `user.name`. Portanto, não ter um gerenciador de segurança instalado é desaconselhável para aplicações que exigem um ambiente seguro.

## O gerenciador de segurança está instalado com uma política padrão

Instalar um gerenciador de segurança ativa as verificações de permissões para acesso a informações e recursos. Esse é o modelo padrão para applets e aplicações Web Start, que são executadas em um *sandbox*. O gerenciador de segurança pode ser instalado configurando a opção de linha de comando `-Djava.security.manager` no comando `java` ou chamando `System.setSecurityManager()`. Se nenhum arquivo de políticas personalizado estiver especificado, o arquivo de políticas padrão será carregado e utilizado. As políticas padrão liberam somente algumas permissões, resultando em um ambiente de execução relativamente seguro. A seguir, há uma versão reduzida do arquivo de políticas padrão, que mostra quais permissões são concedidas:

```
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};

grant {
    // Permite que qualquer thread pare a si próprio utilizando
    // o método java.lang.Thread.stop() sem argumentos.
    permission java.lang.RuntimePermission "stopThread";

    // permite a qualquer pessoa ouvir portas un-privileged
    permission java.net.SocketPermission "localhost:1024-", "listen";

    // propriedades "padrão" que podem ser lidas por qualquer pessoa
    permission java.util.PropertyPermission "java.version", "read";
    ...
}
```

Se uma permissão não for explicitamente concedida, ela não será liberada. Por exemplo, acessar um método privado por meio de reflection requer uma `RuntimePermission` do tipo `accessDeclaredMembers` e uma `ReflectPermission` do tipo `suppressAccessChecks`. Se você tentar executar o teste mostrado no Capítulo 4 com o gerenciador de segurança instalado, a aplicação falha indicando uma exceção de segurança.

Há várias maneiras de conceder permissões ao código. A mais fácil é criar um arquivo de políticas de segurança personalizado contendo todas as permissões exigidas e especificar esse arquivo na linha de comando, utilizando a propriedade `-Djava.security.policy`. Essa abordagem foi mostrada no Capítulo 4.

## O gerenciador de segurança está instalado com uma política personalizada

E se o gerenciador de segurança estiver instalado e um arquivo de diretiva personalizado já estiver especificado com a opção `-D`? Isso costuma ser o caso com servidores de aplicações e servidores Web, que dependem da segurança Java para isolar aplicações J2EE. Há ainda duas alternativas para conceder permissões adicionais. Obviamente, você pode simplesmente editar o arquivo de políticas personalizado do produto que executa seu código, ou o arquivo de política de sistema, mas essa não é uma solução limpa.

Uma maneira melhor é nomear seu arquivo de diretiva `.java.policy` e colocá-lo no diretório inicial (`home`) do usuário. O JRE primeiro carrega o arquivo de políticas de sistema de  `${java.home}` e então carrega o arquivo de políticas do usuário de  `${user.home}`. Com isso você não teria de alterar scripts ou arquivos existentes; isso isola suas alterações e facilita a manutenção.

Uma segunda alternativa é incluir uma referência a seu arquivo de diretiva no arquivo `java.security`, que pode ser encontrado no subdiretório  `${java.home}/lib/security`. Esse arquivo descreve a configuração de segurança e especifica quais, e em que ordem, arquivos de políticas devem ser carregados. Você encontra referências ao arquivo de políticas de sistema e de usuário nesse local; para incluir o seu arquivo de políticas, adicione a linha a seguir:

```
policy.url.3=file:/C:/Projects/CovertJava/distrib/conf/java.policy
```

Certifique-se de utilizar o caminho que corresponde à localização do arquivo no seu computador. “3” é o próximo índice disponível na lista de URLs do arquivo de políticas.

## Questionário rápido

1. Qual classe atua como fachada para o framework de segurança de Java?
2. Que papel desempenham as permissões?
3. Quais arquivos são carregados para determinar as permissões e a ordem em que devem ser concedidas?

## Resumo

- A segurança em Java controla o acesso a informações de sistema e a recursos por meio de permissões.
- O gerenciador de segurança está no centro do modelo de segurança. Se ele não estiver instalado, nenhuma verificação será realizada.
- As permissões são concedidas em arquivos de políticas Java.
- Os arquivos de políticas podem ser especificados na linha de comando ou no arquivo de configuração de segurança.

# 8

## NESTE CAPÍTULO

- ▶ A vantagem de dominar o ambiente de execução 66
- ▶ Propriedades de sistema 67
- ▶ Informações de sistema 68
- ▶ Informações de memória 68
- ▶ Informações de rede 69
- ▶ Acessando variáveis de ambiente 70
- ▶ Questionário rápido 71
- ▶ Resumo 71

# Espionando o ambiente de execução

*"A lógica é um método sistemático para chegar com confiança à conclusão errada."*

As Leis de Murphy aplicadas à tecnologia

## A vantagem de dominar o ambiente de execução

Hackear sistemas e solucionar problemas de aplicações freqüentemente requer a manipulação de vários parâmetros de sistema. Por exemplo, para instalar uma versão corrigida com um patch de uma classe no Capítulo 5, modificamos o valor da variável CLASSPATH. No Capítulo 7, você viu como utilizar uma propriedade de sistema para instalar o gerenciador de segurança. Entender o ambiente de execução e conhecer os valores exatos dos parâmetros de sistema permite que você baseie seu trabalho no conhecimento e não em suposições. Neste capítulo, você aprenderá a obter os valores de propriedades de sistema, a verificar se um gerenciador de segurança está instalado e a obter várias informações sobre a memória e a rede.

A maioria das informações úteis está acessível facilmente por meio de classes da API de Java, contanto que você saiba onde procurar. Por causa da característica multiplataforma de Java, as informações sobre o hardware e recursos físicos são limitadas. Neste capítulo, criaremos uma classe de investigação que consolida e imprime todas as informações úteis sobre o ambiente. Ela pode ser chamada a partir de uma aplicação em execução ou ser executada separadamente utilizando `covertjava.snoop.RuntimeInfo` como a classe principal, ou rodando `snoop_re.bat`. Por exemplo, servidores de aplicações normalmente são iniciados por uma série

de arquivos *batch* ou scripts de shell, que configuram variáveis de ambiente e parâmetros de aplicação. O código de inicialização do servidor pode alterar o estado padrão do ambiente, e a única maneira confiável de conhecer os valores atuais é com rastreamento.

## Propriedades de sistema

*Propriedades de sistema* são pares nome-valor que fornecem informações sobre o ambiente de execução. Elas incluem dados sobre o sistema operacional, sobre o fornecedor e a versão da JVM, caminhos para classes e carregamento de bibliotecas, diretórios iniciais (*home*) e atuais do usuário e outras propriedades úteis. O significado de cada propriedade pode ser encontrado na Internet; recomendo que você se familiarize com ele. Um bom lugar para começar é o JavaDoc da classe `java.lang.System`, que pode ser encontrado em <http://java.sun.com/j2se/1.4.1/docs/api/java/lang/System.html>. O método da Listagem 8.1 imprime todas as propriedades de sistema.

---

### **LISTAGEM 8.1** Código-fonte de `printSystemProperties`

---

```
public void printSystemProperties(PrintStream stream) {
    StringBuffer buffer = new StringBuffer(1000);
    Properties props = System.getProperties();
    for (Enumeration keys = props.keys(); keys.hasMoreElements(); ) {
        String key = (String)keys.nextElement();
        buffer.append(key);
        buffer.append("=");
        buffer.append(System.getProperty(key));
        buffer.append('\n');
    }
    stream.print(buffer.toString());
}
```

---

Executar esse método produz uma saída semelhante a esta:

```
java.vm.version=1.4.1-b21
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
java.vm.name=Java HotSpot(TM) Client VM
...

```

Propriedades de sistema podem ser alteradas utilizando o método `System.setProperty()`. Se um gerenciador de segurança estiver instalado, ele será utilizado para assegurar que o chamador tenha a permissão `write` para a propriedade requerida. Algumas propriedades são lidas somente na inicialização; dessa forma, se comportam como se fossem de leitura. Por exemplo, a propriedade `java.class.path` descreve o caminho utilizado para o carregamento de classes. Mesmo que você possa alterar o valor dessa propriedade, fazê-lo não produzirá o efeito desejado, pois o *class loader* de sistema vai ler a

classe somente uma vez, e não verá a alteração. Manipular a segurança Java foi discutido em detalhes no Capítulo 7.

## Informações de sistema

As propriedades de sistema abrangem muito bem as informações sobre o ambiente. Entretanto, mais alguns aspectos importantes do sistema merecem nossa atenção. Queremos verificar se um gerenciador de segurança está configurado para determinar o grau de segurança do ambiente. Também queremos saber qual *class loader* foi utilizado para carregar a classe para saber quanto espaço temos para a manipulação do carregamento de classes. Por fim, queremos ver quantas CPUs estão disponíveis, por questões de licenciamento e desempenho. A Listagem 8.2 mostra o código que obtém essas informações.

---

### **LISTAGEM 8.2** Código-fonte de printSystemInfo

---

```
public void printSystemInfo(PrintStream stream) {  
    StringBuffer buffer = new StringBuffer(200);  
    buffer.append("Security manager: ");  
    buffer.append(System.getSecurityManager() == null?  
        "null": System.getSecurityManager().getClass().getName());  
    buffer.append('\n');  
  
    buffer.append("Class loader for this class: ");  
    ClassLoader classLoader = this.getClass().getClassLoader();  
    buffer.append(classLoader == null?  
        "null": classLoader.getClass().getName());  
    buffer.append('\n');  
  
    buffer.append("Number of available processors to JVM: ");  
    buffer.append(Runtime.getRuntime().availableProcessors());  
    buffer.append('\n');  
  
    stream.println(buffer.toString());  
}
```

---

## Informações de memória

A JVM gerencia a memória disponível para uma aplicação em Java. Quando a aplicação solicita mais memória, a JVM primeiramente verifica se ela pode atender à solicitação usando a memória livre já alocada ao processo. Se não houver memória livre suficiente, a JVM pode forçar a realização de uma coleta de lixo, na tentativa de liberar a memória utilizada por objetos mortos. Se a coleta de lixo não conseguir liberar memória suficiente, a JVM tenta obter mais memória no sistema operacional, aumentando assim a memória total alocada. Ela continua obtendo mais memória do Sistema Operacional, até o limite

máximo permitido, 16 MB por padrão, mas esse valor pode ser modificado utilizando-se o parâmetro `-Xmx`.

O gerenciamento de memória é crucial à saúde e ao desempenho de aplicações Java. Os detalhes da coleta de lixo e o uso eficaz de memória estão além do tema deste livro, mas você aprenderá a obter o tamanho de vários pools de memória. A melhor maneira de monitorar o uso de memória da aplicação é via um profiler, como o JProbe ou o Optimizelt. Entretanto, essas ferramentas exigem que as aplicações sejam executadas em modo de depuração, o que é inaceitável para sistemas em produção. A nova família de ferramentas, como Wily Introscope e Borland Server Trace, pode monitorar e exibir o uso de memória de qualquer aplicação em execução com overheads mínimos. Elas, porém, contam com o mesmo método que utilizaremos e podem ser bastante caras para muitas aplicações Java. A Listagem 8.3 mostra a implementação de um método que imprime o perfil atual de memória da JVM.

#### **LISTAGEM 8.3** Código-fonte de printMemoryInfo

```
public void printMemoryInfo(PrintStream stream) {  
    StringBuffer buffer = new StringBuffer(200);  
    buffer.append("Maximum memory allowed for JVM: ");  
    buffer.append(toMb(Runtime.getRuntime().maxMemory()));  
    buffer.append(" Mb\n");  
    buffer.append("Memory currently allocated in JVM: ");  
    buffer.append(toMb(Runtime.getRuntime().totalMemory()));  
    buffer.append(" Mb\n");  
    buffer.append("Free memory in JVM: ");  
    buffer.append(toMb(Runtime.getRuntime().freeMemory()));  
    buffer.append(" Mb\n");  
    stream.println(buffer.toString());  
}
```

A função auxiliar `toMb` converte o valor dado em bytes para megabytes com dois pontos de precisão. Ela pode ser encontrada na classe `RuntimeInfo`.

## Informações de rede

Não há muito a aprender sobre o ambiente de rede utilizado por uma aplicação em Java. Java é uma linguagem de alto nível e multiplataforma, o que a torna uma escolha menos que ideal para implementar utilitários e drivers de baixo nível do sistema operacional. Praticamente, as únicas informações úteis que podem ser obtidas do runtime Java são o hostname local e o endereço IP. Essas informações podem ser utilizadas para propósitos de licenciamento, no qual licenças são emitidas para cada servidor e permanecem bloqueadas por nó (máquina). Informações adicionais sobre a configuração de rede e o hardware subjacente podem ser obtidas utilizando-se a JNI e bibliotecas C que fazem chamadas nativas ao sistema operacional. A Listagem 8.4 mostra como obter informações sobre o host.

**LISTAGEM 8.4** Código-fonte de printNetworkInfo

---

```
public void printNetworkInfo(PrintStream stream) {  
    StringBuffer buffer = new StringBuffer(200);  
    InetAddress localhost = null;  
    try {  
        localhost = java.net.InetAddress.getLocalHost();  
        buffer.append("Local host name: ");  
        buffer.append(localhost.getHostName());  
        buffer.append('\n');  
        buffer.append("Local host IP address: ");  
        buffer.append(localhost.getHostAddress());  
    }  
    catch (UnknownHostException ex) {  
        buffer.append("!!! Failed to detect network properties " +  
                    "due to UnknownHostException: ");  
        buffer.append(ex.getMessage());  
    }  
    stream.println(buffer.toString());  
}
```

---

## Acessando variáveis de ambiente

As variáveis de ambiente fornecem uma boa maneira de parametrizar uma aplicação. Por exemplo, o diretório de instalação da aplicação talvez seja diferente entre hosts, e o Java não tem uma maneira confiável de descobrir onde a aplicação foi instalada. Essa é a razão pela qual uma prática comum é definir a variável de ambiente APP\_HOME e passá-la para a aplicação Java. Você não deve utilizar variáveis de ambiente em excesso, pois fazer isso torna complexa a programação de arquivos batch e de shell scripts. Uma alternativa melhor é usar arquivos com o formato de propriedades Java ou XML, e uma variável de ambiente que especifica a localização dos arquivos de configuração. A maneira correta de acessar as variáveis de ambiente em Java é passando-as para a linha de comando utilizando a opção -D. Para clareza, costumo prefixar os nomes de variáveis de ambiente com env. O exemplo a seguir mostra como passar os valores das variáveis de ambiente TEMP e COMPUTERNAME no Windows para investigar o utilitário que estamos desenvolvendo neste capítulo:

```
set JAVA_ARGS=%JAVA_ARGS% -Denv.temp=%TEMP%  
set JAVA_ARGS=%JAVA_ARGS% -Denv.computername=%COMPUTERNAME%  
  
java %JAVA_ARGS% covertjava.snoop.RuntimeInfo
```

A aplicação Java pode então obter o valor da variável utilizando o método `System.getProperty( )` com o nome definido na linha de comando. No nosso exemplo, o valor das variáveis TEMP pode ser obtido utilizando `System.getProperty("env.temp")`.

## Questionário rápido

1. Quando você investigaria o ambiente de execução?
2. Quais informações sobre o ambiente de execução estão disponíveis por meio da API Java padrão?
3. Como informações específicas de hardware ou do sistema operacional não-disponíveis por meio de APIs padrão podem ser recuperadas?

## Resumo

- Obter um registro instantâneo dos valores das variáveis de ambiente é uma maneira confiável de entender a configuração de execução de uma aplicação em execução.
- As propriedades de sistema disponíveis pelo método `System.getProperty( )` fornecem a maioria das informações sobre o ambiente de execução.
- Um registro instantâneo de tamanhos de memória pode ajudar a entender a eficácia do gerenciamento de memória da aplicação no ambiente de produção.
- As informações sobre a rede são limitadas ao nome e ao endereço IP do host local.
- Os valores de variáveis de ambiente podem ser passados para uma aplicação Java utilizando-se o parâmetro de linha de comando `-D`.

# 9

## *NESTE CAPÍTULO*

- ▶ Entendendo o funcionamento interno de aplicações desconhecidas 72
- ▶ Depuradores convencionais e suas limitações 73
- ▶ Hackeando com um depurador onisciente 73
- ▶ Questionário rápido 78
- ▶ Resumo 78

# Quebrando código com depuradores heterodoxos

*“Qualquer teoria simples será escrita da maneira mais complicada.”*

As Leis de Murphy aplicadas à tecnologia

## **Entendendo o funcionamento interno de aplicações desconhecidas**

Escrever um capítulo sobre o uso de um depurador convencional é praticamente menosprezar a inteligência do leitor. Se você não consegue descobrir como utilizar um depurador por conta própria, deveria pensar em mudar de carreira. Em vez disso, o foco deste capítulo é sobre a utilização de uma ferramenta heterodoxa que oferece uma abordagem diferente para examinar aplicações em execução. Embora eu discuta sobre *depuração*, o que você na verdade vai aprender é o funcionamento interno de aplicações desconhecidas. Trabalhar com o código-fonte geralmente é a maneira preferida, mas em certos casos um depurador é insubstituível, incluindo quando:

- você está trabalhando em uma grande aplicação sem rastreamento eficaz;
- interpretar o código-fonte da aplicação não fornece um entendimento claro da lógica interna por causa de uma hierarquia sofisticada de objetos e práticas de programação pesadas;
- você está trabalhando com uma aplicação que foi ofuscada agressivamente.

## Depuradores convencionais e suas limitações

O Java foi projetado desde o início com a depuração em mente. A API padrão Java Debug API baseia-se no conceito de um visualizador remoto para o processo em depuração. Esse padrão permite que fornecedores implementem seus próprios depuradores com uma gama de recursos. Alguns deles não vão muito além de permitir ao desenvolvedor definir pontos de interrupções e fazer inspeções passo a passo no código de aplicações. Depuradores mais avançados fornecem recursos adicionais como pontos de interrupções condicionais que param a execução se uma variável tem um certo valor, ou se uma certa exceção é lançada.

Os depuradores convencionais ajudam na codificação e na correção de bugs, mas não são eficazes para a engenharia reversa ou na solução de problemas em uma aplicação complexa. O maior problema desses depuradores é que eles exibem as informações somente durante o momento atual e, depois disso, as informações são irremediavelmente perdidas. Para serem eficazes, os depuradores convencionais exigem pontos de interrupções estrategicamente posicionados por todo o código, o que obviamente é um processo entediante – especialmente se o código não for muito bem conhecido.

Por exemplo, digamos que você obteve a aplicação Chat e quer saber como ela processa as mensagens recebidas. Embora você possa carregá-la e executá-la em modo de depuração, como vai saber onde posicionar pontos de interrupção? Mesmo com uma pequena aplicação como a Chat, fazer uma inspeção passo a passo no código não vai ajudar, uma vez que as mensagens são entregues em um thread RMI, assincronamente. Somente a descompilação e uma análise cuidadosa do código pode levá-lo ao método `receiveMessage`, definido em `ChatServerRemote` e implementado em `ChatServer`. Se você tentar fazer engenharia reversa na versão da aplicação Chat ofuscada com o Zelix KlassMaster, essa tarefa irá se tornar muito difícil. Você precisa de uma ferramenta melhor para ajudá-lo nessa tarefa. Entra em cena o *Omniscient Debugger*.

## Hackeando com um depurador onisciente

A depuração onisciente permite gravar o estado do programa em execução e depois voltar no tempo e examinar os estados gravados. Isso é diferente da depuração convencional porque a aplicação do usuário não é executada em modo de depuração e, portanto, não pode ser pausada. A idéia por trás do depurador onisciente é registrar o maior número de informações possível sobre threads e variáveis, seus valores, streams de entrada e saída padrão e classes carregadas. Depois que as informações são gravadas, a aplicação pode ser parada ou fechada. O depurador pode então ser utilizado para rastrear a execução do início ao fim, ou começando a partir de um momento específico, identificado por uma chamada de método ou por uma mensagem gravada no fluxo de saída padrão. Com isso, a definição de pontos de interrupções deixa de ser necessária, e você poderá ter certeza de que não vai perder operações processadas assincronamente.

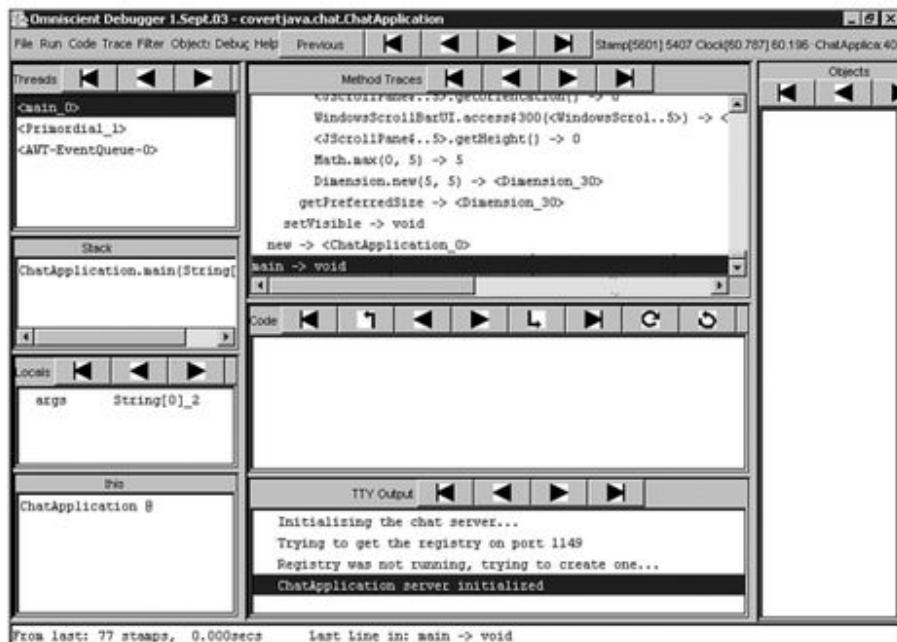
Atualmente, a única implementação dessa abordagem conhecida é o *Omniscient Debugger* (ODB) escrito por Bil Lambda. Ele é distribuído sob licença GPL, e o download pode ser feito em <http://www.lambdacs.com/debugger/debugger.html>. O ODB ainda precisa de

muito trabalho, especialmente na parte da interface gráfica, mas realiza muito bem os trabalhos de gravação e investigação. O ODB utiliza uma técnica de "decoração" de bytecode pela qual é inserido código que grava as informações necessárias para a depuração. Vamos supor que nada sabemos sobre a implementação interna da aplicação Chat e usaremos o Omniscient Debugger para aprender como o Chat processa as mensagens recebidas.

## Gravando a execução de um chat

Começamos registrando a execução da aplicação Chat no ODB. A distribuição do Covert Java inclui a versão do ODB mais recente, no momento de escrita. Ela é colocada no diretório lib e utilizada por padrão. Você pode utilizar uma versão mais recente do ODB, se encontrar uma. Execute debugChat.bat localizado no diretório distrib\bin; esse arquivo batch configura o ambiente do ODB e o instrui a executar covertjava.chat.ChatApplication na inicialização. Depois de ter sido carregado, o ODB abre sua janela principal e executa o Chat. Quando é solicitada a localização do código-fonte de ChatApplicaton, clicamos no botão No Source Available para restringir nosso trabalho ao bytecode. A Figura 9.1 mostra a janela do depurador depois do passo de inicialização.

A interface com o usuário do ODB não é muito intuitiva mas, depois de se acostumar com ela, você será capaz de navegar-a com destreza. À esquerda do menu na parte superior está a barra de ferramentas Stamp. O ODB grava a execução de programas numa seqüência de *timestamps* (registros de data/hora); a barra Stamp permite percorrer essa seqüência para frente ou para trás. Alguns painéis exibem as informações sobre o estado da aplicação:



**FIGURA 9.1** Janela do ODB depois da inicialização.

- ▶ **Threads** – Esse painel mostra todos os threads gravados. Se o nome de um thread estiver prefixado com -, isso significa que o thread ainda não havia sido criado no registro da data/hora atual. Se o thread estiver inativo, ele é mostrado como Dead.
- ▶ **Stack** – Esse painel mostra a pilha de chamadas para o thread atualmente selecionado.
- ▶ **Locals** – Esse painel mostra os nomes e valores das variáveis locais do método selecionado na pilha de chamadas ou no painel Method Traces.
- ▶ **This** – Esse painel mostra o resultado de uma chamada a `toString()` no objeto cujo método está atualmente selecionado.
- ▶ **Method Traces** – Esse painel mostra a seqüência de chamada de métodos, incluindo chamadas aninhadas, que aparecem indentadas.
- ▶ **Code** – Esse painel mostra o código-fonte, se disponível.
- ▶ **TTY Output** – Esse painel mostra as chamadas interceptadas para `System.out.println` (nenhum outro método de saída é atualmente suportado pelo ODB).
- ▶ **Objects** – Esse painel mostra as versões transformadas em strings dos objetos sendo observados.

Vamos tentar compreender as informações exibidas na Figura 9.1. Podemos ver que, depois da inicialização, a aplicação Chat tem três threads. Um é `<main_0>`, que está atualmente selecionado e não tem uma pilha de chamadas, pois ele terminou a execução; entretanto, podemos ver os métodos que são executados nesse thread no painel Method Traces. Rolando pelo painel Method Traces vemos o log completo dos nomes dos métodos e até mesmo os valores dos parâmetros. Clicar em um método no painel Method Traces nos leva ao momento em que o método foi executado e atualiza os outros painéis com as informações correspondentes. De maneira semelhante, clicar em uma mensagem no painel TTY Output permite ver por quem e quando a mensagem foi impressa. Para rastrear a execução do programa desde o início, clique no botão First Timestamp of Any Thread na barra de ferramentas Stamp (parecido com um botão Rewind); então clique no botão Next Timestamp (que parece um botão Play) para avançar passo a passo.

## Navegando pelo código de processamento de mensagens

Agora estamos prontos para quebrar o código de processamento de mensagens da aplicação Chat. Primeiro, devemos contornar um bug no ODB que o impede de capturar as informações sobre classes carregadas dinamicamente. Pressione Alt+Tab para ir à janela Debug Controller, que tem um botão Stop Recording e algumas checkboxes que especificam opções de gravação. Clique no botão Stop Recording; quando seu rótulo mudar para Start Recording, clique nele novamente. Em seguida, pressione Alt+Tab para abrir a janela da aplicação Chat. Digite `localhost` na combobox Host Name e `Hello` no campo de texto da mensagem. Pressione Enter para enviar a mensagem e certifique-se de que ela apareça no campo de conversa da aplicação Chat. Como executamos a aplicação Chat no depurador, nossa entrada e a resposta da aplicação a ela foi gravada, de modo que agora podemos voltar à janela do depurador para examinar os rastreamentos. Clique no botão Last

Timestamp Recorded (Any Thread) na barra de ferramentas Stamp para avançar rapidamente para o final da seqüência gravada. O painel TTY Output deve agora mostrar mais linhas. Clique na linha que informa Received message from host.... A janela do depurador deve se assemelhar à Figura 9.2.

A partir das informações exibidas nos painéis do depurador, podemos concluir que a mensagem foi impressa pelo método ChatServer.receiveMessage em execução em um thread de conexão RMI TCP . Isso faz sentido porque, como sabemos, a aplicação Chat utiliza a tecnologia RMI para enviar mensagens. Rolando pelo painel Method Traces, podemos ver exatamente o que ocorreu antes e depois da chamada a System.out.println. Vemos que o texto da mensagem é obtido da classe MessageInfo e acrescentado ao JEditorPane dentro de MainFrame. Mesmo sem o código-fonte, temos um bom entendimento sobre a lógica de processamento de mensagens da aplicação Chat.

Nossa tarefa foi facilitada porque vimos uma mensagem de rastreamento, Received message from host..., que nos leva diretamente ao método responsável pelo processamento de mensagens. E se não houvesse essa pista? Várias abordagens podem apresentar os mesmos resultados. Depois da gravação das ações, poderíamos primeiro examinar o painel Threads. Conhecer a arquitetura da plataforma Java ajuda a decidir quais threads devem ser examinados. Por exemplo, sabendo que a comunicação remota entre aplicações Java possivelmente utilizará RMI e que RMI processa chamadas recebidas no seu próprio thread, podemos procurar esses threads. Não é de surpreender que, depois de encaminhar rapidamente o depurador para o final do envio da mensagem, podemos ver um novo thread – RMI TCP Connection.... Clicar no thread no painel Threads exibe os métodos executados nele, e eis que mais uma vez vemos ChatServer.receiveMessage na parte superior do rastreamento.

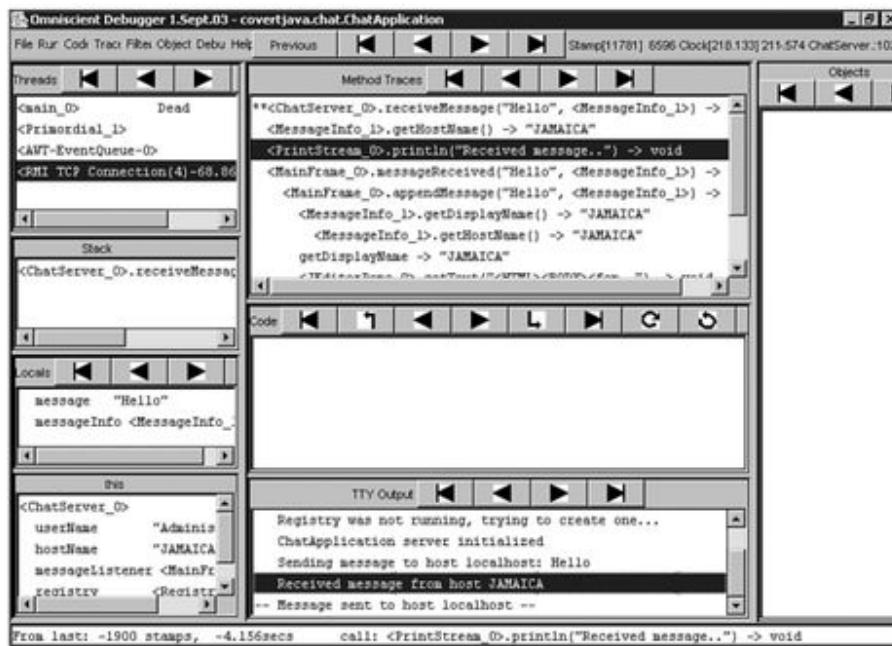


FIGURA 9.2 A janela do ODB depois do envio da mensagem.

Outra boa maneira de localizar a lógica do processamento de mensagens é utilizando o recurso Search do ODB. Embora não muito sofisticado, ele permite procurar uma string de testes em um painel de rastreamento. No nosso caso, sabemos que o texto da mensagem era `Hello`. Se não tivermos certeza sobre o thread utilizado para executar a lógica do processamento de mensagens, podemos ativar a opção Any Thread OK no menu Code do depurador. Então podemos selecionar Search no menu Trace e digitar `Hello`. Pressionar Enter instruirá o depurador a localizar e selecionar o rastreamento que contém a string de pesquisa, que, em nosso caso, é (você adivinhou): `ChatServer.receiveMessage`.

Depois que a classe e o método forem localizados, você pode utilizar o código-fonte ou o código descompilado para analisar a lógica de implementação. Um depurador convencional agora pode ser utilizado para executar a aplicação no modo de depuração; podemos ver dados “ao vivo” porque sabemos onde posicionar os pontos de interrupções.

## Utilizando ODB para quebrar a versão ofuscada do chat

No Capítulo 3 e no Capítulo 4, discutimos o ofuscamento e as dificuldades de quebrar aplicações ofuscadas agressivamente. Descompilar o código ofuscado pode ser entediano, e entender o funcionamento interno de uma grande aplicação pode mostrar-se impossível. Utilizar um depurador como o ODB é a melhor maneira de trabalhar com essas aplicações porque, em vez de tentar imaginar as seqüências das chamadas de métodos, você pode simplesmente analisar sua versão gravada. As mesmas abordagens utilizadas para localizar o ponto de partida para classes regulares podem ser utilizadas para código ofuscado. Para ilustrar isso, vamos fazer o mesmo exercício buscando entender o processamento de mensagens da aplicação Chat, mas desta vez na versão ofuscada.

Alteramos o CLASSPATH dentro de `debugChat.bat` a fim de utilizar `lib\ofuscard\chat.jar` em vez de `lib\chat.jar`. Então, executamos o depurador e seguimos os mesmos passos de gravação descritos nas seções anteriores. Depois que a mensagem é enviada, avançamos o depurador para o fim da gravação utilizando a barra de ferramentas Stamp. Comparando as informações exibidas nos painéis do depurador com as informações exibidas na aplicação Chat não-ofuscada, podemos perceber que ela é essencialmente a mesma. Embora os nomes das classes e métodos da aplicação Chat tenham mudado, as pilhas de chamadas e valores de parâmetro não mudaram. As informações sobre o sistema também não mudaram, por exemplo os nomes das classes Java básicas e de threads. A Listagem 9.1 mostra o conteúdo do painel Method Traces da aplicação Chat ofuscada.

### LISTAGEM 9.1 Conteúdo do painel Method Traces

```
**<d_0>.receiveMessage("Hello", <MessageInfo_1>) -> void
<MessageInfo_1>.a( ) -> "JAMAICA"
<PrintStream_0>.println("Received message..") -> void
<f_0>.a("Hello", <MessageInfo_1>) -> void
<f_0>.b("Hello", <MessageInfo_1>) -> void
<MessageInfo_1>.b( ) -> "JAMAICA"
<MessageInfo_1>.a( ) -> "JAMAICA"
```

**LISTAGEM 9.1** Continuação

---

```
b -> "JAMAICA"
<JEditorPane_0>.setText("<HTML><BODY><font...>") -> void
b -> void
a -> void
receiveMessage -> void
```

---

Podemos continuar a utilizar o depurador para descobrir qual método imprimiu a mensagem Received message from host... ou a linha de rastreamento de métodos que inclui a string Hello. Isso oferece uma grande vantagem ao localizar o ponto de partida para analisar a aplicação.

## Questionário rápido

1. Por que utilizar um depurador no processo de hacking?
2. Quais são as limitações dos depuradores convencionais?
3. Qual é o princípio por trás da implementação do ODB?
4. Quais abordagens podem ser utilizadas para localizar lógica de implementação no ODB?

## Resumo

- Um depurador pode fornecer um atalho para localizar lógica de implementação em uma grande aplicação ou em código ofuscado.
- As principais limitações dos depuradores convencionais são a necessidade de configurar pontos de interrupções em locais estratégicos e a falta de um histórico das alterações de estado e de chamadas de método.
- O ODB grava os registros de data/hora (*timestamps*) da aplicação em execução e permite pesquisar essas informações para entender a lógica interna.
- O ODB é a melhor abordagem para quebrar aplicações ofuscadas.

# Utilizando profilers para análise de aplicações em tempo de execução

*"Nenhum bug pode ser corrigido corretamente depois das 4:30 da tarde de uma sexta-feira. A solução correta se tornará óbvia às 8:15 da manhã da segunda-feira."*

As Leis de Murphy aplicadas à tecnologia

## Por que e quando utilizar profiling

O termo *profiling* é usado tradicionalmente para descrever o processo de medição de tempos de execução de métodos, para localizar e corrigir gargalos de desempenho. Entretanto, no universo Java, esse termo foi expandido para incluir a coleta de várias métricas e permitir a depuração de threads e objetos em tempo de execução. A seguir são apresentadas algumas razões para utilizar um profiler em aplicações Java:

- investigar o uso do heap e a freqüência da coleta de lixo, para melhorar o desempenho;
- pesquisar a alocação de objetos e referências para encontrar e corrigir vazamentos de memória;
- investigar a alocação e a sincronização de threads para encontrar problemas de bloqueio e de concorrência no acesso a dados, e para melhorar o desempenho;

# 10

## NESTE CAPÍTULO

- ▶ Por que e quando utilizar profiling 79
- ▶ Os melhores profilers para Java 80
- ▶ Investigando o uso do heap e a freqüência da coleta de lixo para melhorar o desempenho 80
- ▶ Pesquisando a alocação e referências de objetos para encontrar e corrigir vazamentos de memória 82
- ▶ Investigando a alocação e a sincronização de threads 86
- ▶ Identificando métodos de alto custo para melhorar o desempenho 89
- ▶ Investigando uma aplicação em tempo de execução utilizando um dump de threads 90
- ▶ Questionário rápido 92
- ▶ Resumo 92

- identificar métodos custosos, visando melhorar o desempenho;
- investigar uma aplicação em tempo de execução para entender melhor sua estrutura interna.

O profiling geralmente ocorre depois da fase de desenvolvimento e, se você ainda não utilizou um profiler, ele poderá ser surpreendente. Os três principais objetivos de alto nível na utilização de um profiler são: melhorar o desempenho de aplicações, corrigir bugs difíceis de localizar, e entender o que está acontecendo na aplicação enquanto ela executa a lógica de negócios.

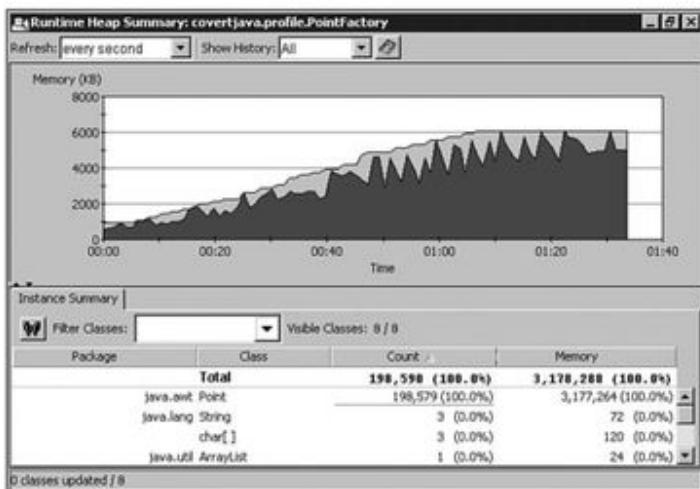
## **Os melhores profilers para Java**

Para ser eficaz, você precisa das ferramentas corretas. Os três profilers mais famosos para Java são o JProbe Suite da Quest Software (originalmente desenvolvido pela KLGGroup), o OptimizeIt Suite da Borland e o JProfiler da ej-technologies. Todos os três são bons e realizam seu trabalho quase com a mesma qualidade. Gosto do fato de o JProfiler integrar toda a funcionalidade em uma aplicação única, em vez de ter ferramentas separadas para profiling, depuração de memória e depuração de threads. O JProbe foi provavelmente a primeira ferramenta desse tipo e ainda é a melhor. Além de fornecer funcionalidades disponíveis em outras ferramentas, o JProbe fornece uma janela de grafo de heap, que mostra objetos como nós retangulares e permite navegar por grafos de objetos de diversas maneiras. As outras ferramentas mostram apenas o grafo de objetos como uma árvore, o que simplifica a exibição, mas não é tão visual. Esse e os outros recursos adicionais do JProbe tornam-no a minha ferramenta preferida para depuração de memória. Iremos utilizá-lo na maior parte deste capítulo, mas sinta-se livre para selecionar a ferramenta que você preferir, ou a que possui.

## **Investigando o uso do heap e a freqüência da coleta de lixo para melhorar o desempenho**

A coleta de lixo é um excelente recurso da plataforma Java que elimina a necessidade dos desenvolvedores de liberar explicitamente os objetos alocados. O custo dessa simplicidade é um overhead de desempenho quando a coleta de lixo é executada. As versões mais recentes da JVM utilizam a coleta de lixo generacional, que pode ser executada assincronamente; mesmo depois desses aprimoramentos, o overhead continua significativo. É fácil subestimar o efeito da coleta de lixo sobre o desempenho da aplicação.

O JProbe e outros profilers mostram o gráfico de resumo de heap de runtime para uma aplicação em execução. Ele permite monitorar o tamanho total da memória alocada e da memória livre disponível, como mostrado na Figura 10.1.



**FIGURA 10.1** Um gráfico de resumo de heap em runtime para um tamanho de heap não-otimizado.

Quando não há memória livre suficiente para satisfazer a solicitação de alocação, a JVM executa a coleta de lixo de primeira geração para liberar a memória ocupada por objetos não-utilizados. A coleta de primeira geração examina somente os objetos recém-alocados, a fim de evitar uma iteração demorada por todo o grafo de objetos. Se uma quantidade de memória suficiente não tiver sido liberada, uma coleta de lixo completa é executada. A coleta de lixo completa se inicia na raiz da árvore de objetos, que geralmente consiste nos objetos referenciados por threads ativos e variáveis estáticas, e identifica todos os objetos que podem ser alcançados a partir da raiz. Os objetos inacessíveis a partir da raiz (isto é, os que não fazem parte de um grafo de referência começando no objeto-raiz) são marcados para coleta de lixo. Se a coleta de lixo completa não conseguir liberar memória suficiente, a JVM verifica se pode alocar mais memória do sistema operacional. Se ela alcançar o limite máximo permitido ou o sistema operacional não conseguir fornecer mais memória, uma exceção `OutOfMemory` é lançada.

Como você pode perceber, criar muitos novos objetos pode resultar em operações complexas e demoradas. Em vez de realizar os processos que constituem a lógica de negócio, a JVM talvez gaste tempo de processador no gerenciamento de memória. O gráfico na Figura 10.1 mostra o uso de memória de uma aplicação sem o tamanho máximo de

## HISTÓRIAS DAS TRINCHEIRAS

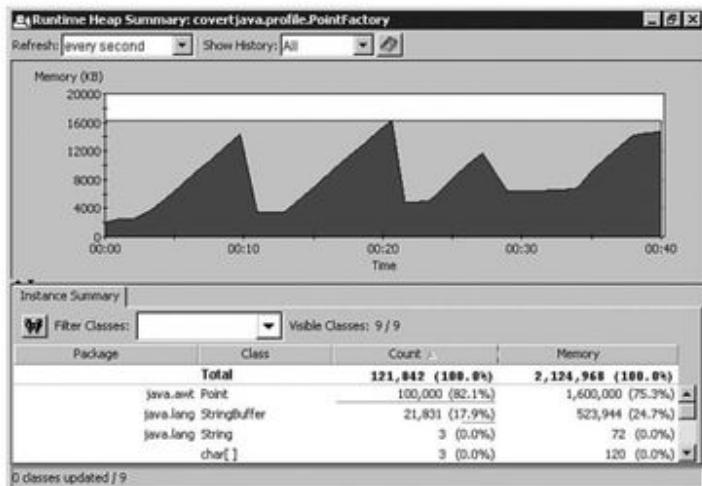
Uma das aplicações em Java que utilizamos no Riggs Bank não conseguia mais inicializar quando o volume de dados a ser exibido aumentou drasticamente. Os usuários esperavam até 30 minutos para que a aplicação fosse inicializada, mas, mesmo assim, o processo não terminava. Investigando a inicialização da aplicação, observamos sintomas de ciclos intermináveis de coleta de lixo – um alto uso de CPU sem atividade de disco ou tráfego de rede. Depois que o tamanho máximo do heap permitido foi aumentado de 128 MB para 256 MB, a aplicação passou a ser inicializada em menos de 30 segundos!

heap otimizado. Cada diminuição na quantidade de heap utilizada é o resultado de uma coleta de lixo. Você pode ver que a JVM tem de executar a coleta de lixo freqüentemente porque há pouca memória livre. O valor máximo do tamanho de heap é configurado utilizando o parâmetro `-Xmx` na linha de comando Java. Aumentar o tamanho máximo do heap de 5 MB para 16 MB produz o gráfico de resumo de heap mostrado na Figura 10.2.

O desempenho da aplicação melhorou significativamente porque a coleta de lixo foi executada somente 3 vezes, comparado a quase 30 vezes antes. Não há uma regra fixa que possa ser utilizada para determinar o tamanho ótimo do heap. Às vezes, executar vários ciclos de coleta de lixo pode ser quase tão rápido quanto executar um ciclo que tem de iterar por mais objetos. Experimentar valores para tamanhos iniciais e máximos do heap para uma aplicação específica é a única maneira confiável de encontrar a melhor configuração.

## Pesquisando a alocação e referências de objetos para encontrar e corrigir vazamentos de memória

Não é mais novidade o fato de que aplicações Java têm vazamentos de memória. Não são os mesmos tipos de vazamentos que ocorrem com linguagens de nível mais baixo como C++, mas são vazamentos, uma vez que a memória alocada não é liberada de volta ao pool livre. Como vimos anteriormente, a coleta de lixo Java libera a memória mantida por objetos inacessíveis. Se houver um referência a um objeto, este não estará elegível para coleta de lixo, mesmo que nunca mais seja utilizado. Por exemplo, se um objeto for colocado em um array e nunca for removido desse array, ele sempre permanecerá na memória. Esses objetos, às vezes chamados de *objetos persistentes* (*lingering objects*), com o passar do tempo podem consumir toda a memória livre disponível e resultar em uma exceção `OutOfMemory`. O problema é ampliado quando o objeto refere-se a uma grande árvore



**FIGURA 10.2** Um gráfico de resumo de heap em tempo de execução para um tamanho de heap otimizado.

de outros objetos, que costuma ser o caso em aplicações. Outro perigo são os objetos persistentes, que representam recursos como conexões de banco de dados ou arquivos temporários. Quanto mais tempo o objeto permanece na memória, mais custosa será a coleta de lixo, pois ela tem mais objetos a serem percorridos. Profilers oferecem o melhor meio de localizar objetos persistentes e identificar o que impede que sofram coleta de lixo.

Vamos examinar um exemplo simples, que produz objetos persistentes, e utilizar o JProbe para localizar a fonte do problema. Suponha que estejamos criando um editor gráfico e precisamos de uma fábrica que forneça uma camada de abstração para a criação de objetos Point. O trecho com a implementação da fábrica é mostrado na Listagem 10.1.

#### **LISTAGEM 10.1** Implementação de PointFactory

---

```
public class PointFactory {  
    protected ArrayList points = new ArrayList( );  
  
    public Point createPoint(int x, int y) {  
        Point point = new Point(x, y);  
        this.points.add(point);  
        return point;  
    }  
  
    public void removePoint(Point point) {  
        this.points.remove(point);  
    }  
}
```

---

O método `createPoint` cria instâncias de `Point` e as coloca em um array. `removePoint` remove um ponto do array e `PointFactory` inclui um método de testes que cria vários pontos por meio da fábrica. O código para o método de testes é mostrado na Listagem 10.2.

#### **LISTAGEM 10.2** Método que testa a criação de pontos

---

```
public void printTestPoints( ) {  
    for (int i = 0; i < 5; i++) {  
        Point point = createPoint(i, i);  
        System.out.println("Point = " + point);  
    }  
}
```

---

O método principal de `PointFactory` exibe uma caixa de diálogo com um botão Print Test Points. Clicar em um botão resulta em uma chamada ao método `printTestPoints( )` de `PointFactory`. A implementação de `printTestPoints` parece código Java perfeitamente válido, mas executá-lo resulta em vazamentos de memória. Se aumentarmos o tamanho do objeto que alocamos e o número de iterações no loop, podemos rapidamente exaurir

toda a memória disponível. Sem dúvida, você entendeu o que está causando o vazamento no nosso exemplo. Isso é fácil de perceber porque temos uma classe com menos de 50 linhas de código, mas, se tivéssemos centenas de classes com lógica complexa, o problema não seria tão evidente. É aí onde os profilers se tornam indispensáveis.

Para investigar o gerenciamento de memória do código de exemplo, devemos configurar `covert.java.profile.PointFactory` como uma aplicação independente no JProbe Memory Debugger. Depois de executá-la no JProbe, a visualização inicial mostra um resumo de heap de runtime e uma visualização de classes. Como a aplicação é pequena, vemos somente algumas classes na visualização de classes, mas, em uma grande aplicação, pode ser aplicado um filtro com base no nome da classe ou do pacote. Quando a caixa de diálogo da aplicação aparece, examinar as classes e a contagem de instâncias revela que nenhum objeto `java.awt.Point` ainda foi criado. Clicamos no botão Print Test Points na aplicação e retornamos à visualização de classes. Agora vemos que cinco instâncias de `Point` e cinco instâncias de `StringBuffer` foram criadas. Esse é o resultado óbvio de chamar o método `printTestPoints`. JProbe permite forçar a coleta de lixo; é a próxima coisa que fazemos. Depois do término da coleta de lixo, é possível observar que as instâncias `StringBuffer` desapareceram e que os pontos ainda estão na memória. Repetir o teste a partir do botão Print Test Points aumenta o número de instâncias de `Point` para 10. Isso confirma que há vazamentos de memória; agora vamos descobrir por que os pontos estão persistindo na memória.

Depois de identificar a instância que não é liberada, há várias opções para localizar o que está evitando que ela sofra a coleta de lixo: examinar o código-fonte manualmente, navegando pela árvore de referências em um profiler e localizando todos os caminhos para a raiz.

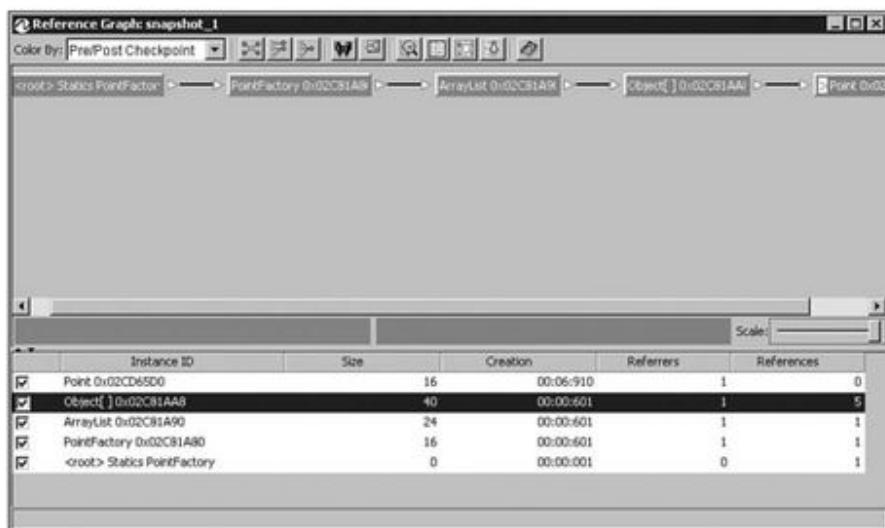
Examinar o código-fonte e localizar todos os lugares em que a instância é criada pode ajudá-lo a descobrir se algumas referências a ela não saem de escopo e não são tornadas nulas. Normalmente, profilers podem mostrar o ponto de alocação de uma instância, o que pode ajudar nessa tarefa. Essa é uma abordagem de força bruta que pode ser demorada. Uma maneira mais sofisticada, às vezes a única praticável, é capturar um *snapshot* (registro instantâneo, “fotografia”) do heap em tempo de execução e então examiná-lo investigando o objeto em questão. Em vez de monitorar manualmente as possíveis referências a uma instância no código-fonte, pesquisar um snapshot do heap permite navegar pelas referências em execução que estão mantendo o objeto na memória. A maioria dos profilers mostra *referentes* (objetos que apontam para o objeto em questão) e *referências* (objetos apontados pelo objeto em questão). Um recurso muito poderoso é a capacidade do profiler de mostrar todos os caminhos do objeto para a raiz. Os grafos de objeto no snapshot podem ser complexos e difíceis de pesquisar. Se estiver interessado somente em um objeto particular, você não precisará ver todo o grafo; precisa apenas ver quais objetos estão impedindo que ele sofra a coleta de lixo. Lembre que o coletor de lixo libera um objeto somente se ele for alcançável através das raízes. Portanto, se você puder identificar e eliminar todos os caminhos para a raiz a partir de um dado objeto, ele torna-se elegível para a coleta de lixo.

Antes de tirar um snapshot do heap, sempre é uma boa idéia solicitar uma coleta de lixo. Assim você remove os objetos que não são mais referenciados e facilita o trabalho de monitorar vazamentos de memória. No JProbe você pode fazer isso selecionando Re-

quest Garbage Collection no menu Program. Depois disso, tire um snapshot do heap selecionando Take Heap Snapshot no menu Program. Depois que o JProbe gerar o snapshot, selecione Class View no menu Snapshot. Agora, você pode ver o estado do heap e os grafos de objetos. No nosso caso, como estamos interessados nos objetos `java.awt.Point`, devemos clicar com o botão direito em Class View e selecionar Instance Detail View no menu pop-up. A nova tela exibe todas as instâncias de `Point` na memória e permite navegar pelas árvores de referentes e de referências. A árvore de referentes é composta por uma hierarquia de ancestrais que direta ou indiretamente têm a instância como um filho – isto é, todos os seus pais imediatos, pais de seus pais, pais dos pais dos seus pais, e assim por diante. Navegar pela árvore fornece uma boa idéia sobre quais referências são mantidas. No caso da nossa instância `Point`, podemos ver que ela é referenciada por um `ArrayList`, que é referenciado por um `PointFactory`.

Na verdade, o JProbe tem outra visualização que prefiro utilizar. Chamada Reference Graph, é uma visualização mais apropriada do grafo de objetos. Para visualizar a Reference Graph para um `Point`, clique com o botão direito em uma instância de `Point` e selecione Instance Referrers and References no menu pop-up. Isso exibe uma nova janela que mostra todos os objetos como retângulos, identificados com o objeto selecionado no centro. Como estamos tentando descobrir por que os pontos estão persistindo na memória, clique no ícone Paths to Root na barra de ferramentas do grafo. A janela resultante é mostrada na Figura 10.3.

Mais uma vez, podemos ver que o objeto `Point` é referido pelo `ArrayList` de `PointFactory`. Podemos concluir que a razão pela qual os pontos permanecem na memória é porque nunca são removidos do array. Examinando a implementação de `printTestPoint()`, podemos perceber que ela chama `createPoint`, que armazena o objeto `Point` instanciado no array de pontos. Para corrigir o problema, precisamos adicionar uma chamada a `removePoint` em `printTestPoint`, de modo que o método se pareça com o da Listagem 10.3.



**FIGURA 10.3** Um grafo que mostra caminhos para raiz de um ponto.

**LISTAGEM 10.3** Implementação corrigida do teste de criação de pontos

```
public void printTestPoints( ) {  
    for (int i = 0; i < 5; i++) {  
        Point point = createPoint(i, i);  
        System.out.println("Point = " + point);  
        removePoint(point);  
    }  
}
```

## Investigando a alocação e a sincronização de threads

Escrever aplicações com múltiplos threads apresenta riscos inerentes de problemas de sincronização. Em um thread simples, o código é executado na mesma ordem em que é escrito no arquivo fonte. Com múltiplos threads de execução, várias operações concorrentes podem ser realizadas ao mesmo tempo, e cada thread pode ser interrompido para permitir que o processador execute as operações do próximo thread. A ordem e os pontos de interrupção são praticamente aleatórios e não podem ser previstos. A maioria das aplicações Java são multithreaded e, embora o conceito de threads embutidos oculte as complexidades da alocação de threads, ele não apresenta salvaguardas contra problemas comuns com threads, como concorrência de dados, deadlocks e travamentos de threads.

Uma condição de *concorrência de dados (data race)* ocorre quando dois threads tentam acessar e modificar simultaneamente um recurso compartilhado. Um exemplo típico é alterar o saldo de uma conta bancária sem sincronização. Digamos que você tenha uma classe BankAccount com os métodos `getBalance()` e `setBalance()` para obter e configurar o saldo atual, respectivamente. Você também tem um método `deposit()` para adicionar ou retirar fundos da conta. A implementação do primeiro `deposit()` obtém o saldo atual, adiciona então o valor do depósito e, por fim, define o novo saldo. Se dois threads – `thread1` e `thread2` – estiverem fazendo um depósito ao mesmo tempo, o `thread1` pode ler o saldo atual e incrementar o valor, mas antes de ter oportunidade de configurar o novo saldo, ele pode ser interrompido pelo `thread2`. O `thread2` lê o saldo antigo e o incrementa antes de ser interrompido para a execução de `thread1`. O `thread1` atualiza o saldo de acordo com o valor que havia calculado, mas, depois disso, o `thread2` retoma a execução e sobrescreve o saldo definido pelo `thread1` com seu próprio valor. Nesse exemplo, o valor depositado pelo `thread1` seria perdido. Os clientes talvez gostem desse tipo de anomalia para saques, mas eles não tolerarão para depósitos em suas contas. Profilers como o JProbe Threadalizer podem capturar e informar os vários tipos de concorrência de dados. Uma solução típica para concorrências de dados é adicionar sincronização que proteja o acesso aos dados. Para corrigir a implementação de depósitos em contas bancárias, o método que faz o depósito deve ser declarado com uma palavra-chave: `synchronized`. A sincronização assegura que somente um thread possa obter o *lock* (“trava”) e qualquer outro thread que tente obter o mesmo lock seja bloqueado, até que o lock seja liberado pelo seu proprietário. Uma alternativa para criar métodos sincronizados é declarar um objeto de “lock” e utilizá-lo como um parâmetro para um bloco `synchronized` (sincronizado), como mostrado na Listagem 10.4.

**LISTAGEM 10.4** Protegendo o acesso a dados com um bloco *synchronized*

```
protected Object balanceLock = new Object( );
protected double balance;
...
public void deposit(double amount) {
    synchronized (balanceLock) {
        double balance = getBalance( );
        balance += amount;
        setBalance(balance);
    }
}
```

Utilizar sincronização adiciona overhead, e portanto isso deve ser aplicado somente quando necessário. Também observe que nem todas as concorrências de dados informadas por um profiler são problemas que precisam ser corrigidos. Por exemplo, mesmo que diversos threads leiam o saldo ao mesmo tempo, não há nenhum problema nisso; portanto, o método `getBalance()` não tem necessariamente de ser sincronizado. Em uma aplicação com múltiplas camadas, a sincronização pode ser deixada para outras camadas. Por exemplo, o nível de isolamento de uma transação de banco de dados pode ser utilizado para evitar leituras e gravações concorrentes dos mesmos dados.

Um *deadlock* ocorre quando um thread espera um lock adquirido por outro thread, mas que nunca é liberado. É um problema comum em aplicações com vários threads, que pode fazer com que a aplicação pare de funcionar completamente. Por exemplo, se o `thread1` obtiver `lock1` e então esperar o `lock2`, que no momento está com o `thread2`, ocorrerá um deadlock se o `thread2` primeiro tentar obter o `lock1` sem liberar `lock2`. Para ilustrar esse problema, vamos examinar a classe de exemplo com dois locks e dois métodos que utilizam esses locks para sincronizar o trabalho que realizam. Na Listagem 10.5, o trabalho real realizado não é importante, porque o foco está na lógica da sincronização.

**LISTAGEM 10.5** Código com possibilidade de deadlock

```
public Object lock1 = new Object( );
public Object lock2 = new Object( );

public void method1( ) {
    synchronized (lock1) {
        synchronized (lock2) {
            doSomething( );
        }
    }
}

public void method2( ) {
```

**LISTAGEM 10.5** Continuação

---

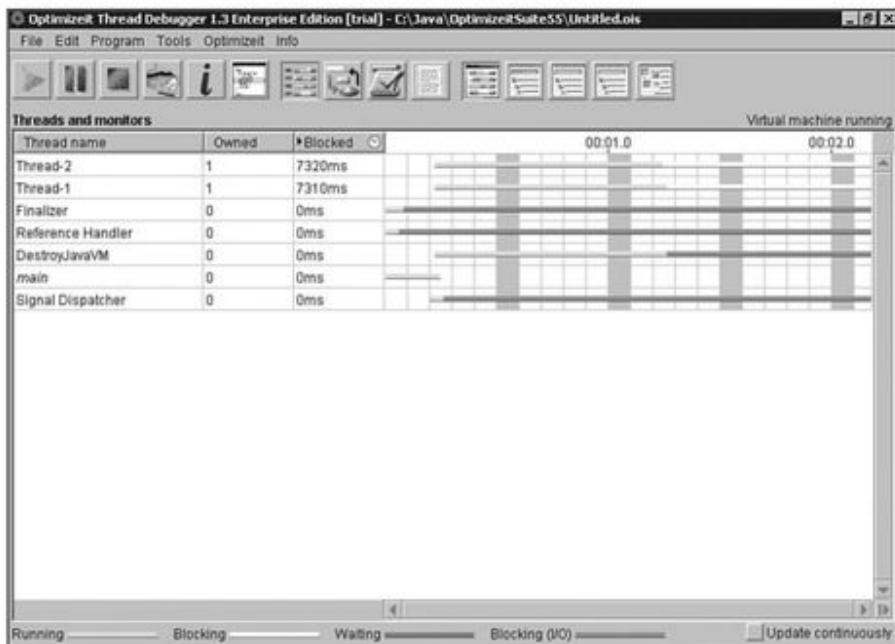
```
synchronized (lock2) {  
    synchronized (lock1) {  
        doSomething( );  
    }  
}  
}
```

---

Como você pode ver, os dois métodos utilizam os locks `lock1` e `lock2`, mas estes são obtidos em uma ordem diferente. Se executados em threads separados, `method1` poderia entrar no primeiro bloco sincronizado e obter o `lock1`. Entretanto, se esse método for interrompido pelo `method2`, `method2` pode entrar no seu próprio bloco sincronizado externo e obter `lock2`. Nesse ponto, o deadlock ocorre à medida que os threads continuam a execução, porque os dois ficam esperando por locks indefinidamente. A classe `ThreadTest` no pacote `covertjava.profile` fornece um exemplo desse cenário. Executá-la produz um deadlock que evita uma desativação elegante da JVM, porque os dois threads gerados pelo método `main()` nunca retornam.

No nosso exemplo, identificar o problema é fácil se você examinar atentamente o código de `ThreadTest`, mas em uma aplicação do dia-a-dia com centenas de classes isso não é tão simples. Bons profilers detectam deadlocks e fornecem informações sobre o que resultou neles. O JProbe Threadalyzer da Quest e o OptimizeIt Thread Debugger da Borland podem detectar deadlocks, mas o OptimizeIt também fornece uma linha de tempo visual da execução do thread, por isso iremos utilizá-lo no nosso trabalho. Execute o OptimizeIt e configure `ThreadTest` como uma aplicação com `covertjava.profile.ThreadTest` como a classe principal. Certifique-se de marcar a opção Auto-start Analyzer, na caixa de diálogo de configurações da aplicação, para começar a analisar automaticamente do programa em execução à procura de problemas. Inicie `ThreadTest` utilizando um comando menu ou um botão na barra de ferramentas; depois de ser executado por alguns segundos, o OptimizeIt deve exibir os threads, como mostrado na Figura 10.4.

A visualização exibe todos os threads e a linha de tempo de seus status de execução. Verde indica que o thread está em execução; amarelo, que o thread está bloqueado enquanto espera um lock que está com thread; vermelho indica que o thread está aguardando uma notificação; e roxo indica que o thread está bloqueado no código nativo. O `ThreadTest` gera dois threads que recebem os nomes padrão de `Thread-1` e `Thread-2`. Examinando a linha de tempo da execução dos dois threads, você verá que, depois de executarem por um curto período, permanecem bloqueados indefinidamente. Todos os outros threads são iniciados internamente pela JVM para realizar tarefas de sistema, portanto manteremos o foco nos threads criados pela aplicação. O OptimizeIt permite a visualização dos locks mantidos por cada thread e os locks que o thread está esperando. Como optamos por utilizar o Analyzer, tiramos proveito dele para obter uma dica sobre o problema. Alternamos para a visualização Analyzer utilizando o menu File. Em seguida, clicamos no botão Stop Recording; devemos ver um deadlock detectado pela ferramenta, com a descrição `Locking order mismatch`. Selecionar o item de impasse exibe as informações detalhadas sobre os threads envolvidos e os bloqueios mantidos. No nosso caso, o texto



**FIGURA 10.4** Visualização de threads do Optimizelt do ThreadTest em execução.

detalhado de descrição é Thread Thread-2 enters monitor java.lang.Object 0xabf9280 first and then java.lang.Object 0xabf72f0 while thread Thread-1 enters the same monitors in the opposite order [Thread Thread-2 insere primeiro o monitor java.lang.Object 0xabf9280 e depois java.lang.Object 0xabf72f0, enquanto o thread Thread-1 insere os mesmos monitores na ordem inversa]. Optimizelt mostra até mesmo as linhas de código em que os locks foram adquiridos. Isso facilita a correção do problema.

Para evitar deadlocks, a ordem em que os locks são adquiridos deve ser a mesma. No nosso caso, devemos alterar `method2` para primeiro obter `lock1` e depois `lock2`. A última coisa a ser lembrada é que, mesmo que o Analyzer possa detectar muitos erros comuns, capazes de provocar deadlocks, ele não pode detectar todas as possíveis situações. Um bom design e práticas sólidas de programação ajudam bastante a evitar problemas no futuro.

*Thread stall* é um termo utilizado para descrever threads que ficam esperando ser notificados, mas a notificação nunca chega. Isso acontece quando um thread chama o método `wait()` em um objeto, mas nenhum outro thread chama o método `notify()` no mesmo objeto. O *thread stall* é outro exemplo de problema em aplicações com múltiplos threads que pode ser detectado por um profiler.

## Identificando métodos de alto custo para melhorar o desempenho

A otimização de desempenho é um tema complexo que está muito além do escopo deste capítulo. O desempenho do Java evoluiu muito, e os JITs modernos apresentam otimizações de velocidade e de código excelentes. O fator mais importante que afeta a velocida-

de da execução final é a qualidade do design e da implementação da aplicação. Conhecer algumas armadilhas bem conhecidas e maneiras de contorná-las pode aumentar o desempenho em grande escala. Frequentemente, reescrever alguns métodos estratégicos pode resultar em aplicações que respondem dez vezes mais rápido que antes. Otimizações de desempenho começam com a identificação de gargalos. Naturalmente, o desempenho deve ser considerado até mesmo durante a fase de design, embora não seja recomendável sacrificar um design limpo para obter pequenos ganhos antecipados em desempenho. À medida que é desenvolvido e testado, o código deve ser analisado com um profiler para identificar os métodos que estão levando o maior tempo. Em geral, é nos testes de integração que é feita a maior parte do profiling.

O profiling coleta várias estatísticas de execução que fornecem uma idéia sobre onde o tempo é realmente gasto. Estas são as métricas mais úteis produzidas pelo profiler JProbe:

- ▶ **Method time** – tempo gasto executando um dado método, excluindo o tempo gasto nos métodos que ele chamou.
- ▶ **Method number of calls** – número de vezes em que o método foi chamado. Pode ser utilizado para identificar os métodos em que a otimização pode produzir o efeito mais drástico sobre o desempenho geral.
- ▶ **Average method time** – tempo médio que o método levou para ser executado. Isso é equivalente ao tempo de método, dividido pelo número de chamadas; pode ser utilizado para identificar os métodos mais lentos.
- ▶ **Cumulative time** – tempo total gasto ao executar um dado método, incluindo os métodos que ele chamou. Pode ser utilizado para identificar os métodos que utilizam a maior parte do tempo de processamento.
- ▶ **Average objects per method** – número médio de objetos criados dentro de um método. Como a coleta de lixo de Java afeta significativamente o desempenho das aplicações, reduzir a criação de objetos pode aumentar a velocidade da execução.

Depois que as estatísticas forem coletadas e analisadas, podemos proceder à otimização de partes da aplicação com desempenho ruim.

## Investigando uma aplicação em tempo de execução utilizando um dump de threads

Como executar uma aplicação em um profiler exige que a JVM seja iniciada em modo de depuração, isso não é praticável em aplicações de produção. Seria interessante poder obter um registro instantâneo de threads e o que eles fazem em uma aplicação qualquer, mesmo se essa aplicação fosse iniciada em um modo de execução normal. Felizmente, uma JVM pode ser instruída a produzir um dump de threads completo que exibe todos os threads com estados e pilha de chamadas. Para obter um dump de threads no Unix, você deve executar um comando `kill -3`, onde 3 é o ID de processo da JVM (`kill -QUIT` funciona no Solaris). No Windows isso pode ser feito pressionando-se Ctrl+Break na janela de console (prompt de comando). Um dump de threads para a aplicação Chat é mostrado na Listagem 10.6.

**LISTAGEM 10.6** Dump de thread parcial em tempo de execução para a aplicação Chat

```
Full thread dump:  
"RMI ConnectionExpiration-[10.241.11.244:14512]" daemon prio=7  
tid=0x8ad0b68 nid=0x748 waiting on monitor [0x119df000..0x119dfdbc]  
    at java.lang.Thread.sleep(Native Method)  
    at sun.rmi.transport.tcp.TCPChannel$Reaper.run(TCPChannel.java:522)  
    at java.lang.Thread.run(Thread.java:479)  
  
"TimerQueue" daemon prio=5 tid=0x8ac15d8 nid=0x8a0  
waiting on monitor [0x1177f000..0x1177fdb]  
    at java.lang.Object.wait(Native Method)  
    at javax.swing.TimerQueue.run(TimerQueue.java:228)  
    at java.lang.Thread.run(Thread.java:479)  
  
"AWT-EventQueue-0" prio=7 tid=0x8a1ac28 nid=0x148  
waiting on monitor [0x10f3f000..0x10f3fdb]  
    at java.lang.Object.wait(Native Method)  
    at java.lang.Object.wait(Object.java:415)  
    at java.awt.EventQueue.getNextEvent(EventQueue.java:255)  
    at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread...)  
    at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread...)  
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:88)  
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:80)  
  
"Signal Dispatcher" daemon prio=10 tid=0x802388 nid=0x2b0  
waiting on monitor [0..0]  
  
"Finalizer" daemon prio=9 tid=0x7fe368 nid=0x640  
waiting on monitor [0x8c4f000..0x8c4fdb]  
    at java.lang.Object.wait(Native Method)  
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:103)  
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)  
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:157)  
  
...
```

A Listagem 10.6 mostra alguns threads em execução dentro da JVM para a aplicação Chat. Podemos ver, por exemplo, que um thread de conexão RMI está fazendo um *sleep* dentro do método *run* da classe *sun.rmi.transport.tcp.TCPChannel\$Reaper*. O thread AWT-EventQueue espera o próximo evento, e o thread finalizador espera em um monitor utilizando o método *wait()*.

Dumps de threads podem ser muito úteis se um deadlock ocorrer em uma aplicação em execução, não sendo óbvio o que o causou. Conhecer quais threads estão esperando por quais monitores é a melhor pista que se pode obter. No Unix, o dump também inclui as variáveis de ambiente, informações sobre o sistema operacional, dump de monitores e muitas outras informações úteis.

## Questionário rápido

1. Quais são as razões para se fazer profiling em uma aplicação?
2. Por que a coleta de lixo afeta dramaticamente o desempenho de aplicações?
3. O que pode causar vazamentos de memória em uma aplicação Java?
4. Como você faria para localizar e corrigir vazamentos de memória?
5. Quais são os problemas comuns em aplicações com múltiplos threads?
6. Qual processo você utilizaria para melhorar o desempenho de uma aplicação?

## Resumo

- Os profilers fornecem um meio sofisticado de resolver problemas como vazamentos de memória, concorrência de dados, deadlocks e problemas de desempenho.
- Investigar o uso do heap e a freqüência da coleta de lixo traz conhecimento sobre como melhorar o desempenho.
- Pesquisar a alocação de objetos e referências a objetos em tempo de execução ajuda a localizar e corrigir vazamentos de memória.
- Investigar a alocação e a sincronização de threads ajuda a encontrar problemas de travamento de threads e concorrência de dados e melhora o desempenho.
- Identificar métodos custosos durante o profiling melhora o desempenho das aplicações.
- Investigar uma aplicação em um profiler fornece informações importantes sobre sua estrutura interna.
- Um dump de threads produzido pela JVM como resposta a um SIGQUIT fornece, para uma aplicação que não esteja rodando em modo de depuração, informações úteis sobre threads e monitores.

# Realizando testes de carga para encontrar e corrigir problemas de escalabilidade

*"Bugs de software só podem ser detectados pelo usuário final."*

As Leis de Murphy aplicadas à tecnologia

## A importância do teste de carga

Aplicações servidoras têm requisitos de nível de serviço que especificam a disponibilidade, a escalabilidade e a tolerância a falhas:

- ▶ **Disponibilidade** – Especifica requisitos de tempo de funcionamento (*up-time*), que descrevem por quanto tempo a aplicação deve ser executada sem precisar de reinicialização;
- ▶ **Escalabilidade** – Especifica a capacidade da aplicação de fornecer o mesmo nível de serviço à medida que aumenta o número de solicitações;
- ▶ **Tolerância a falhas (failover)** – Especifica a capacidade da aplicação de continuar a fornecer o mesmo nível de serviço se um dos componentes da aplicação apresentar falhas.

Um ciclo de desenvolvimento típico aloca tempo aos testes de unidade e de integração, que geralmente focam nas funcionalidades; mas nem sempre aloca

## 11

### *NESTE CAPÍTULO*

- ▶ A importância do teste de carga 93
- ▶ Testes de carga de servidores baseados em RMI com JUnit 95
- ▶ Testes de carga com o JMeter 98
- ▶ Questionário rápido 108
- ▶ Resumo 108

tempo para testes de carga. O propósito dos testes de carga é avaliar como o desempenho do sistema atende aos requisitos de nível de serviço, sob uma carga determinada. Obviamente, o tempo de resposta de cada sistema se degrada à medida que a carga aumenta, mas, contanto que sejam atendidos os requisitos especificados, o sistema será considerado como escalável.

Ignorar os testes de carga é uma prática arriscada, especialmente se for esperado que a aplicação atenda a centenas ou milhares de usuários. Com uma grande comunidade de usuários, um problema pequeno torna-se grande, porque afeta um grupo numeroso de pessoas. Algumas falhas não aparecem a menos que haja uma certa carga. Esse poderia ser o caso de operações que dependem de recursos como threads, conexões a bancos de dados e memória. A maioria dos problemas inerentes a aplicações com múltiplos threads ocorre quando há um número específico de solicitações concorrentes. Somente os testes de carga e execução do sistema por um período de tempo prolongado simulam o ambiente operacional de produção, e portanto é crucial realizar esses testes antes de um sistema ser colocado on-line. E não devemos esquecer que testar a carga de uma aplicação permite ver como ela responde a ataques de *denial of service* e tentativas de hacking. Os testes de carga não revelam como a aplicação é construída, e sim se ela foi construída para durar. Podem ser utilizados para tirar melhor proveito das técnicas apresentadas em outros capítulos. Por exemplo, fazer o profiling de uma aplicação sob carga produz um quadro diferente do que sem carga.

Simular uma carga no sistema não é uma tarefa trivial e, assim, é importante tirar proveito das ferramentas certas. Existem muitos produtos para testes de carga disponíveis no mercado, na sua maioria projetados para funcionar com sites Web que disponibili-

## HISTÓRIAS DAS TRINCHEIRAS

Estávamos construindo uma aplicação Web que seria utilizada por 5 mil usuários. Vários prazos finais de entrega haviam sido perdidos; assim, quando o desenvolvimento e os testes de unidade foram concluídos, não havia mais tempo para escrever testes automatizados ou realizar testes de carga com uma ferramenta de testes apropriada. Vários testadores simularam manualmente determinados volumes de carga e, cruzando os dedos, a implantação recebeu um “sinal verde”. A aplicação foi colocada em produção às 2 da manhã, aproveitando uma brecha para manutenção do servidor; exausta, a equipe de desenvolvimento foi para casa esperando contar com alguns dias de descanso. Entretanto, às 11 da manhã do dia seguinte, quando os usuários começaram a acessar o site, o servidor de aplicações parou de responder a solicitações. Reinicializá-lo ajudou por algumas horas, mas aí o deadlock ocorreu novamente. A gerência estava furiosa, fazendo ameaças de encontrar e demitir os responsáveis. Quando o problema foi finalmente encontrado, descobrimos que o culpado era um código com múltiplos threads que tratava de exceções e da tolerância a falhas. O código havia sido escrito de modo que, se uma exceção fosse detectada ao processar uma solicitação, ela seria analisada e, em alguns casos, seria tentada uma recuperação seguida por uma nova tentativa da operação. O problema era que, com múltiplas solicitações concorrentes, o código de análise/recuperação/nova-tentativa transformara-se em um loop infinito. Se testes de carga adequados tivessem sido feitos no início, o erro seria localizado e corrigido sem afetar milhares de usuários e a reputação da equipe de desenvolvimento.

zam conteúdo HTML via HTTP. Além de HTTP, aplicações servidoras em Java também funcionam com vários outros protocolos, como o JRMP para clientes RMI, e IIOP para clientes CORBA e RMI. As melhores ferramentas são bastante caras, mas algumas alternativas open source oferecem a maioria das funcionalidades básicas. Vamos realizar testes de carga na aplicação Chat, baseada em RMI, usando o framework livre JUnit. Posteriormente, utilizaremos outra ferramenta open source, o JMeter, para testar a carga do WebCream, uma aplicação Web com clientes no browser.

Os princípios por trás da simulação de cargas são praticamente os mesmos. Um caso de testes é criado, registrando a aplicação em execução ou programmaticamente. Depois de pronto, o caso de testes é utilizado para criar usuários ou clientes virtuais, que são então executados em múltiplos threads simultaneamente acessando o servidor. Para a aplicação servidora, os usuários virtuais aparecem como tráfego real; e monitorar a precisão e o tempo de resposta do servidor produz os resultados do teste.

## Testes de carga de servidores baseados em RMI com JUnit

O JUnit fornece um framework para escrever e executar testes de unidade. Ele promove a escrita de código de testes, que assegura a validade das funcionalidades das aplicações. Um caso de teste do JUnit é uma classe Java compilada e executada para testes de aplicações. A abordagem fornece os benefícios de novos testes automatizados com a facilidade de manter o código de testes em sincronia com o da aplicação. Além disso, os desenvolvedores escrevem código Java, em vez de clicar botões em depuradores e ferramentas de testes, o que deve ter contribuído muito para a popularidade do framework.

Para utilizar o JUnit, um desenvolvedor deve escrever um caso de teste que estenda `junit.framework.TestCase`, ou implemente `junit.framework.Test`. O caso de teste contém chamadas às classes sob testes, além de assertivas (*assertions*) de que os valores de retorno correspondam ao esperado. Por exemplo, um caso de teste para uma conta bancária pode fornecer o saldo atual, fazer um depósito e então verificar se o novo saldo corresponde ao saldo antigo, somado ao valor do depósito. A qualidade do caso de teste é diretamente proporcional ao zelo do desenvolvedor. A idéia é tentar abranger todos os possíveis cenários, incluindo os errôneos. Depois que os testes são escritos, eles são compilados e executados individualmente ou em grupos. O JUnit é um framework bem documentado e fácil de aprender; se você ainda não trabalhou com ele, não deixe de investir algumas horas na leitura do manual e dos exemplos (E não se esqueça de atualizar o seu currículo, pois bons gerentes irão considerar esse conhecimento como um indicador de um bom desenvolvedor.). O download do framework e da documentação relacionada pode ser feito gratuitamente em <http://www.junit.org>. O restante desta seção foca no desenvolvimento de testes de carga da aplicação Chat, baseados no JUnit.

A aplicação Chat certamente não foi construída para ser indestrutível, mas também não foi concebida para servir centenas de usuários. Contanto que ela possa lidar com um número entre três e seis usuários simultâneos, isso provavelmente será suficiente para satisfazer os requisitos de concorrência de uma aplicação de demonstração. Para qualquer teste de carga, você deve configurar uma carga máxima um pouco acima da prevista. Por-

tanto, no nosso exemplo, utilizaremos 10 como o número de clientes virtuais que devem ser suportados. Nossa objetivo é simular este número de clientes enviando simultaneamente mensagens à mesma aplicação Chat. Também queremos espaçar (*stagger*) as chamadas, para simular uma experiência real. *Espaçamento (staggering)* significa que, em vez de enviar as solicitações no mesmo instante, elas são enviadas aproximadamente ao mesmo tempo. Às vezes, o termo *simultâneo* é utilizado para descrever os clientes que enviam uma solicitação ao mesmo tempo, enquanto *concorrente* é utilizado para descrever clientes que mantêm uma conversa com o servidor, mas enviam solicitações em momentos próximos um do outro. Em um sistema com uma única CPU, não há na verdade nenhuma distinção entre execução concorrente e execução simultânea, porque não existe verdadeiro processamento paralelo; isso torna os dois termos intercambiáveis.

Começaremos desenvolvendo um caso de teste simulando um usuário que envia uma mensagem ao host de destino. Construiremos então um *harness* que utiliza o caso de teste para criar alguns usuários virtuais, que enviam mensagens repetidamente. Para sermos flexíveis, vamos permitir a parametrização dos testes especificando o número de usuários simultâneos a serem simulados, o número de repetições do teste e os intervalos para espaçar as chamadas.

O caso de teste é escrito em `covertjava.loadtest.ChatTestCase`. Ele estende `TestCase` e implementa sua lógica básica em seu método `testSendMessage()`, mostrado na Listagem 11.1.

#### LISTAGEM 11.1 Código-fonte de `testSendMessage`

```
public void testSendMessage( ) {
    logger.info("Sending test message...");
    try {
        StringBuffer message = new StringBuffer( );
        message.append("[ChatTestCase@");
        message.append(Integer.toHexString(this.hashCode( )));
        message.append("] Test ");
        message.append(messagesSent++);
        ChatServer.getInstance( ).sendMessage(this.host, message.toString( ));
        logger.info("Sent message successfully");
    }
    catch (Exception e) {
        e.printStackTrace( );
        assertTrue("Exception: " + e.getMessage( ), false);
    }
}
```

`ChatTestCase` cria uma mensagem de teste e utiliza `ChatServer` para enviá-la ao host de destino. A parte principal desse método é uma chamada a `assertTrue( )` na instrução `catch` com `false` como o segundo parâmetro, que informa ao JUnit que o teste falhou. Caso contrário, o método simplesmente retorna, o que significaria sucesso. Essa maneira de testar está certamente longe da ideal para assegurar que o servidor Chat processou a

mensagem corretamente. Ela não testa se foi feito o parse da mensagem e se ela foi adicionada apropriadamente à janela de histórico de conversas. Entretanto, fornece uma tática razoável de testar a comunicação de rede e o *throughput* do servidor remoto; portanto, será suficiente para ilustração.

O próximo passo é criar uma suite de testes que contenha instâncias de `ChatTestCase`. Isso é realizado na classe `ChatLoadTest`; o código é mostrado na Listagem 11.2.

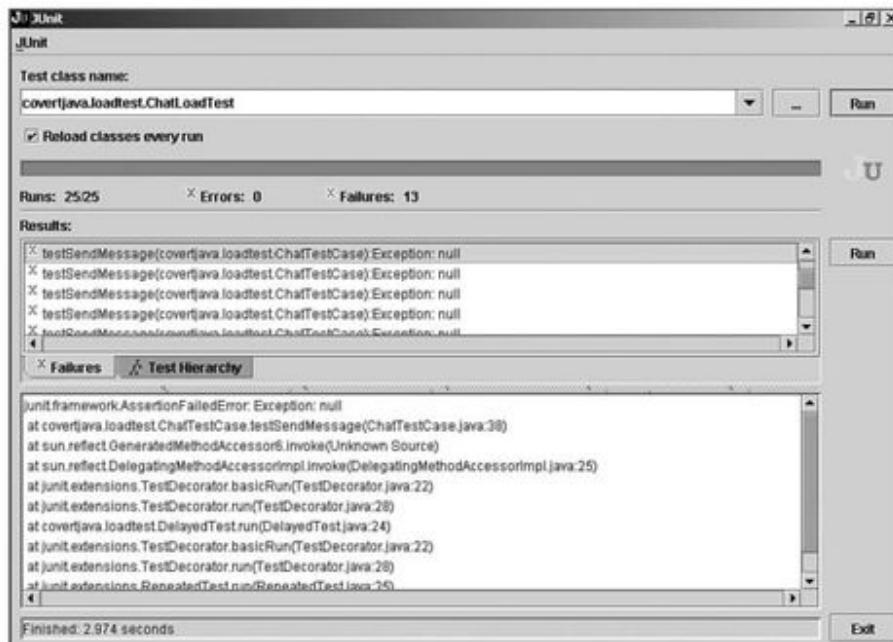
### **LISTAGEM 11.2** Criação de uma suite de testes

```
ActiveTestSuite suite = new ActiveTestSuite( );
for (int i = 0; i < clientsNumber; i++) {
    Test test = new ChatTestCase( );
    test = new DelayedTest(test, (int)(Math.random( )*lagTime));
    test = new RepeatedTest(test, repeatRuns);
    suite.addTest(test);
}
```

Parâmetros, como `clientsNumber` e `lagTime`, são lidos no arquivo de propriedades para suportar a personalização. Suites de testes do JUnit agregam casos de teste, simplificando a execução de múltiplos testes. `ActiveTestSuite` executa todos os testes simultaneamente e então espera que eles terminem, para depois retornar o resultado. O JUnit utiliza um padrão *decorator* para anexar funcionalidades adicionais aos testes. `RepeatedTest`, por exemplo, executa um dado thread repetidamente um número de vezes determinado. Para fornecer a execução com espaçamento para um teste, adicionamos a classe *decorator* `DelayedTest`, que executa um *sleep* por um certo tempo antes de executar o teste. O resultado final do código na Listagem 11.2 é um `clientsNumber` maior que o número de clientes que enviarão simultaneamente mensagens, depois de fazer um *sleep* por um tempo aleatório.

Você pode executar um teste com o JUnit de várias maneiras; utilizaremos a interface Swing para obter feedback visual sobre o progresso dos testes. Se uma instância da aplicação Chat ainda não estiver sendo executada no host local, podemos iniciá-la utilizando `CovertJava\distrib\bin\chat.bat`. Então usamos o arquivo `loadtestJUnit.bat` localizado no diretório `CovertJava\bin` para abrir a interface gráfica do JUnit e executar nossa suite de testes. Logo depois que os testes começarem a ser executados, a interface do JUnit deve estar semelhante à Figura 11.1.

A maioria dos testes falha; examinando o painel de resultados, podemos ver esta mensagem de erro: `testSendMessage(covert.java.loadtest.ChatTestCase): Exception java.lang.NullPointerException: null`. Examinar a aplicação Chat nos mostra que o componente de histórico de conversas é confuso e que várias `NullPointerExceptions` estão na janela de console. Adivinhou o que está acontecendo? Começamos a colher os benefícios dos testes de carga. Se reduzirmos o número de clientes para dois, os testes são executados com sucesso, e portanto o problema deve vir do uso de múltiplos threads. O Capítulo 9 e o Capítulo 10 forneceram as técnicas para localizar e corrigir problemas que surgem na execução concorrente. Você pode tentar aplicar essas técnicas para descobrir o que há de errado com a aplicação Chat.



**FIGURA 11.1** Interface gráfica do JUnit mostrando o progresso dos testes.

Para aquele leitor que já se considera suficientemente instruído, vou revelar que o problema vem da maneira como novas mensagens são acrescentadas ao histórico de conversas. O Swing não é *thread-safe*; assim geralmente é recomendável interagir com componentes Swing a partir do thread de envio de eventos (*event dispatch thread*) do AWT. Os projetistas do Swing sacrificaram a robustez pela velocidade; não podemos mesmo culpá-los, pois o desempenho do Swing tem estado há muito tempo sob exame rigoroso. A aplicação Chat recebe as mensagens recebidas em um thread RMI e delega o processamento das mensagens à classe MainFrame. O método appendMessage de MainFrame, que acrescenta uma nova mensagem ao JEditorPane de conversas, não utiliza sincronização. Portanto, se vários usuários tentarem enviar uma mensagem ao mesmo host simultaneamente, vários threads tentarão alterar o texto do mesmo JEditorPane. Esse é um problema clássico de corrupção de dados, que pode ser resolvido tornando-se o método appendMessage sincronizado. Depois de fazer essa alteração, executar novamente os testes de carga produz um resultado perfeito; podemos considerar o trabalho concluído.

O benefício da utilização do JUnit para testes de carga é que ele é simples e permite tirar proveito de casos de teste já escritos. É também um método eficaz de testar servidores baseados em RMI, porque a gravação automática de scripts por ferramentas de testes de carga nem sempre produz resultados confiáveis.

## Testes de carga com o JMeter

A seção anterior mostrou como fazer testes de carga de aplicações Java utilizando o JUnit. O desenvolvimento do teste foi simples e, embora não tenha resultado em gráficos ou diagramas sofisticados, o trabalho básico foi bem executado. Entretanto, essa abordagem é

um pouco limitada, porque requer que seja escrito um cliente virtual manualmente; a única coisa que o JUnit pode fazer é executar o caso de teste em vários threads. E se quiséssemos testar uma aplicação Web com um *frontend* HTML? Em geral, isso exigiria que os usuários executassem um navegador Web, e o servidor teria de usar Servlets e páginas JSP para implementar a interface com o usuário. O JUnit não seria uma opção nesse caso, uma vez que os usuários virtuais devem suportar funcionalidades de navegadores Web, como o gerenciamento de sessões, cookies, formulários e outros. Outra deficiência do JUnit é que ele não produz uma prova tangível do sucesso ou falha do teste. Um desenvolvedor experiente conhece o efeito de relatórios, diagramas e gráficos sobre a gerência. Resumindo, precisamos de uma ferramenta melhor.

Os testes de carga constituem um mercado enorme e lucrativo, e há muitos bons produtos disponíveis. Atualmente, esses produtos oferecem praticamente a mesma funcionalidade básica, que inclui a capacidade de gravar usuários virtuais automaticamente, criação de scripts de testes programaticamente, suporte a múltiplas linguagens e protocolos de comunicação e – você adivinhou – muitos gráficos e relatórios sofisticados. Frequentemente, são os gráficos que influenciam a decisão final sobre qual produto comprar. Destacarei dois produtos considerados como líderes de mercado: O Mercury Load Runner e o Rational Test Suite. O Load Runner é uma ferramenta excelente e comprovada pelo uso, que fornece o máximo em flexibilidade e tem praticamente todo recurso encontrado em ferramentas de testes de carga. Um fator importante é sua capacidade de gravar e personalizar scripts de usuários virtuais, o que pode tornar desnecessário escrever código. A ferramenta pode até mesmo gravar a operação de clientes HTTP e RMI no nível de protocolo. Como testes de carga têm reconhecidamente um alto valor (e com todo o direito), os preços das ferramentas podem ser bastante altos. Testar um cluster de servidores com centenas de usuários virtuais pode resultar em custos de licenciamento de milhares de dólares. Examinaremos o JMeter, uma alternativa open source que pode ser obtida gratuitamente do site da Apache. Embora nem de longe tão aperfeiçoado ou versátil, o JMeter oferece funcionalidade básica semelhante, e é suficiente para a maioria das aplicações Web. E não se preocupe: você vai aprender a produzir alguns gráficos e um relatório também.

## Visão geral do JMeter

O JMeter é uma ferramenta de testes de carga e de medição de desempenho de sites Web e aplicações Java. Suporta servidores que aceitam conexões HTTP e FTP, mas também pode ser utilizado para testar bancos de dados, scripts Perl e objetos Java. O JMeter suporta a gravação básica de scripts via um servidor proxy, e requer conhecimento aprofundado do protocolo subjacente e da implementação do servidor. Criar um plano de testes exige trabalho manual, mas, como ocorre com o JUnit, é o tipo de trabalho desafiador de que desenvolvedores gostam. Como a maioria das novas aplicações construídas hoje tem *thin clients* (clientes com interfaces leves, como as baseadas em HTML), vamos usar o JMeter para fazer testes de carga em um produto para Web chamado WebCream.

O JMeter tem uma interface gráfica que pode ser usada para criar planos de testes, e inclui um mecanismo que executa esse plano para gerar a carga desejada. O plano de testes

agrega elementos de configuração e controladores lógicos que representam definições e ações a serem realizadas. A criação dos testes é visual e o resultado final é uma árvore de elementos aninhados que descrevem os testes e sua execução. Utilizar um servidor proxy HTTP, que registra as ações de navegação, é uma boa maneira de criar rapidamente um plano de testes inicial. Independentemente de você iniciar gravando o plano de testes, ou criando-o manualmente, você vai precisar conhecer os elementos utilizados pelo JMeter para que o plano de testes funcione. Os seguintes nós podem ser adicionados a um plano de testes:

- **Thread Group** (Grupo de Threads) – Define o número de usuários virtuais que executarão os nós aninhados (abaixo do nó), de forma concorrente. Também permite especificar o tempo de *ramp-up* para espaçamento, além da duração do teste.
- **Listeners** – Podem ser adicionados para fornecer visualizações dos resultados dos testes e monitorar o progresso de sua execução. Por exemplo, para gerar um gráfico de tempos de resposta, um *graph results listener* pode ser adicionado a um plano de testes.
- **Configuration Elements** (Elementos de Configuração) – Utilizados para adicionar aos *Samplers* gerenciadores de protocolos e configurações padrão; os *Samplers* são discutidos adiante. Por exemplo, para que os clientes virtuais suportem sessões HTTP via cookies, adicione o elemento de configuração *HTTP Cookie Manager*.
- **Assertions** (Assertivas) – Utilizados para testar a validade de respostas do servidor. Obter uma resposta não é suficiente para informar se um teste foi concluído. As assertivas permitem verificar a resposta a uma determinada substring ou checar uma correspondência exata; se a verificação falhar, a assertiva falha. Assertivas que falharam podem ser visualizadas com um listener *assertion results*.
- **Preprocessors** (Pré-processadores) – Executados antes de uma operação ser realizada. Por exemplo, o pré-processador *user parameter* é utilizado para definir e inicializar variáveis para uma solicitação HTTP.
- **Post-processors** (Pós-processadores) – Estes são executados depois que uma operação é realizada. Por exemplo, um *regular expression extractor* (extrator baseado em expressões regulares) pode ser adicionado, para fazer o parse do título de uma página HTML retornada pelo servidor.
- **Timers** – Utilizados para acrescentar a execução agendada e a execução espaçada (*staggered*) de operações.

Um grupo de threads pode ter os seguintes nós adicionais:

- **Logic controllers** (Controladores lógicos) – Esses podem ser adicionados para especificar o fluxo de controle para nós aninhados. Por exemplo, adicionar um *loop controller* permite a execução dos nós aninhados em um loop por um número de vezes especificado.
- **Samplers** – Permitem o envio de solicitações à aplicação sendo testada. São os *Samplers* que realizam as chamadas ao servidor. Atualmente, o JMeter suporta solicitações FTP, HTTP, SOAP, Java, JDBC e LDAP.

Depois de criado, um plano de testes pode ser executado na máquina local. Você também pode configurar várias máquinas para atuar como servidores JMeter remotos, que podem ser controlados visualmente. Isso permite simular um número maior de clientes virtuais que apenas uma máquina pode atender.

## Visão geral do WebCream

O WebCream é uma ferramenta única para Java que automatiza a ativação para Web de aplicações e applets Java gráficos. O WebCream permite que desenvolvedores implementem um frontend gráfico com AWT e Swing e que, ao mesmo tempo, obtenham acesso por HTML à aplicação, automaticamente. De certa maneira, o WebCream pode ser considerado como um conversor dinâmico entre Java e HTML, que transforma instantaneamente janelas e caixas de diálogo em HTML. Ele então emula as ações nas páginas Web como eventos de interface gráfica, retendo a lógica original da aplicação. O WebCream é único pelo fato de que não requerer modificações nos formulários existentes, ou na lógica de negócio e por não exigir que programadores aprendam novas APIs. Como o WebCream utiliza uma interface num navegador Web, além de gerar conteúdo dinâmico e manter uma sessão, é uma boa escolha para testes de carga HTTP com o JMeter.

O WebCream vem com um servidor Tomcat embutido e uma aplicação de demonstração; vale a pena brincar um pouco para se familiarizar com o produto. Utilize a aplicação demo do WebCream e não deixe de examinar o código HTML que o produto gera para cada página. Tente entender os URLs utilizados pelo produto, e como eles passam os dados de volta ao servidor; procure também padrões comuns nas páginas geradas. A versão *standard* do WebCream é gratuita, mas é limitada a cinco usuários concorrentes, e portanto talvez seja necessário reiniciar o Tomcat durante seus testes se você não quiser esperar que as sessões expirem. A página principal do demo do WebCream exibe um quadro com três botões – Login Dialog, Tabs and Table e Tree Dialog (ver Figura 11.2).

Clicar no botão Login Dialog exibe a próxima página, que mostra uma caixa de diálogo permitindo inserir o nome de usuário e a senha, além de selecionar um domínio. Clicar em OK passa as informações de login para a página principal. Para propósito de testes, iremos nos limitar às funcionalidades dessa demo.

## Criando um plano de testes na Web

Agora é hora de trabalhar um pouco. Baixe e instale o JMeter e o WebCream. Execute a interface gráfica do JMeter utilizando `JMeter\bin\jmeter.bat`. Como vamos testar o WebCream, inicie o Tomcat embutido utilizando `WebCream\bin\startServer.bat`.

A árvore inicial na janela do JMeter mostra um plano de testes vazio e um WorkBench. Um WorkBench é simplesmente um local para nós temporários e testes; vamos nos concentrar no plano de testes. Como queremos simular múltiplos usuários simultâneos, precisamos adicionar um grupo de threads clicando com o botão direito no nó Test Plan e selecionando Add > Thread Group. Inicialmente, configuramos o número de threads como 1 para simplificar, porém, mais tarde, iremos alterá-lo para 5. Para simular interações reais dos usuários com o servidor, precisamos garantir que haja pausas entre



**FIGURA 11.2** Página HTML principal da aplicação demo do WebCream.

solicitações ao servidor. Cem usuários concorrentes no dia-a-dia não significa cem solicitações simultâneas ao servidor, pois usuários passam tempo interpretando respostas do servidor e preenchendo dados em formulários (às vezes até param para um café). O JMeter fornece o *Gaussian Random Timer* para a simulação de pausas dos usuários. Vamos adicioná-lo ao nosso grupo de threads e especificar um intervalo constante de 300 milissegundos com um desvio de 100 milissegundos. Isso significa que o JMeter pausará um thread por, pelo menos, 300 milissegundos antes de tentar executar a próxima fase do plano de testes.

Antes de começarmos a planejar as solicitações HTTP a serem simuladas, precisamos de alguns passos preparatórios. O WebCream mantém uma sessão HTTP baseada em cookies armazenados pelo navegador Web. Estamos simulando um cliente baseado em navegador Web, portanto adicionamos um *HTTP Cookie Manager* ao grupo de threads. Adicionar apenas esse elemento de configuração é suficiente para que o JMeter armazene cookies; não será preciso definir cookies manualmente.

Por fim, vamos utilizar o elemento de configuração *HTTP Request Defaults*. Ele é uma maneira conveniente de fornecer, apenas uma vez, informações comuns a todas as solicitações HTTP. Opções óbvias para inclusão no *HTTP Request Defaults* são o protocolo (HTTP), o nome de servidor (localhost), o caminho (/webcream/apps/WebCreamDemo) e o número da porta (8040). Se você examinou cuidadosamente o código-fonte do HTML gerado pelo Webcream, deve ter notado que há três parâmetros *hidden* em cada página, como mostrado na Listagem 11.3.

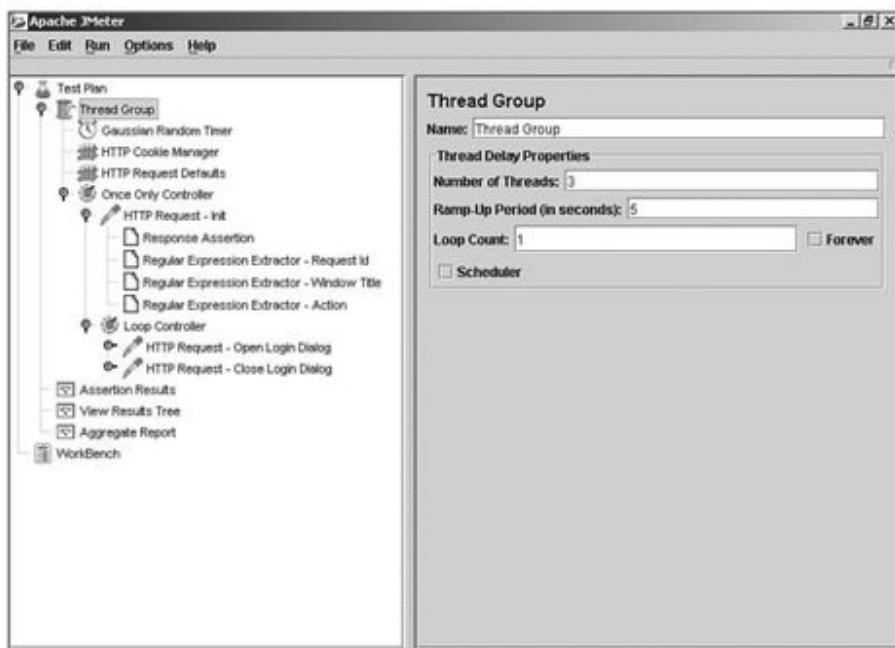
**LISTAGEM 11.3** Parâmetros comuns de formulários utilizados pelo WebCream

```
<input type="hidden" name="__RequestId" value="1">
<input type="hidden" name="__WindowTitle" value="WebCream Demo">
<input type="hidden" name="__Action" value="">
```

À medida que o usuário navega pelas páginas, os valores desses parâmetros mudam para refletir o número de solicitações consecutivas e o título da janela atual. O parâmetro `_Action` é utilizado para informar o servidor sobre a ação a ser realizada na aplicação. Por exemplo, para fechar uma janela, o código no cliente configura `_Action` como `close`. Como esses parâmetros são enviados com cada solicitação, é recomendável incluí-los no HTTP Request Defaults. Utilizamos a interface gráfica do JMeter para adicionar os três parâmetros, deixando os valores em branco, por enquanto. Vamos especificar esses valores depois de conhecer o JMeter um pouco melhor. A Figura 11.3 mostra a árvore do plano de testes que criamos até agora.

Vamos simular um usuário abrindo a aplicação demo do WebCream em um navegador, clicando em Login para ir à página de Login e depois em OK para voltar à página principal. Para garantir que não há vazamentos de memória, ou problemas de múltiplos threads no servidor, vamos simular o usuário abrindo e fechando a caixa de diálogo várias vezes.

Para manter o plano de testes conciso, adicionamos um Once Only Controller ao grupo de threads; este controlador permite agrupar as solicitações dos Samplers em uma sub árvore, para separá-los dos elementos de configuração. Em seguida, adicionamos um *HTTP Request Sampler* ao controlador. Vamos nomeá-lo como *HTTP Request - Init*. Como ele é a solicitação inicial à aplicação Web, especificamos o método GET e deixamos o resto em



**FIGURA 11.3** Árvore do plano de testes do JMeter, passo 1.

branco. As informações sobre o servidor Web (como o protocolo e o host) vêm dos padrões para as solicitações HTTP especificados anteriormente. Nesse ponto, teremos criado um passo que é equivalente a um usuário digitando

```
http://localhost:8040/webcream/apps/WebCreamDemo
```

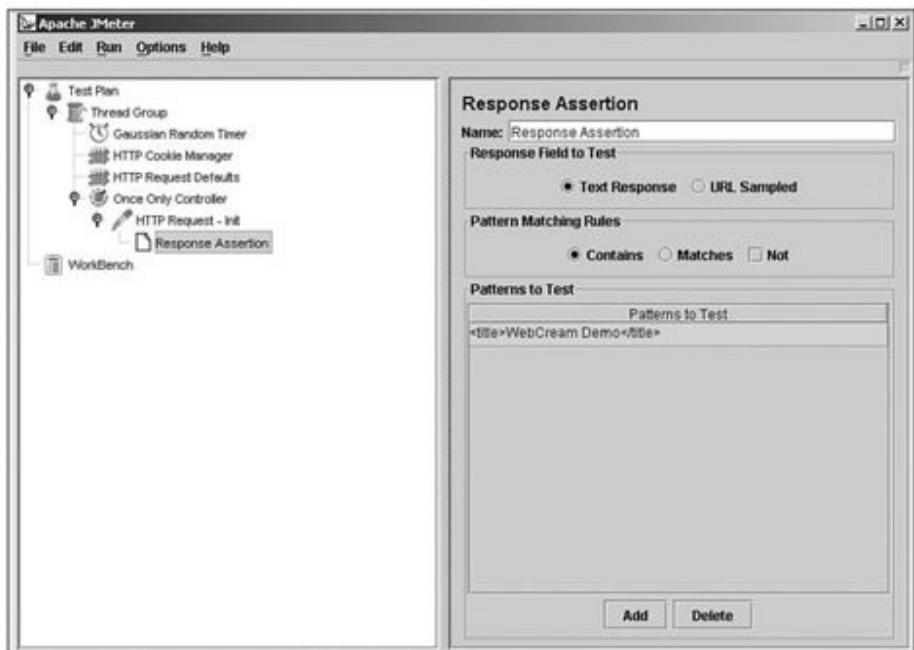
na barra de endereços do navegador e pressionando Enter. O servidor deve responder com a página principal (veja a Figura 11.2). Devemos verificar se obtivemos a resposta correta; fazemos isso adicionando uma assertiva Response ao nó HTTP Request - Init. Uma assertiva pode ser bem simples, como exigir que uma substring esteja presente na resposta, ou ser bastante sofisticada, como definir que expressões regulares produzam uma correspondência. Vamos simplificar e apenas testar se a resposta tem o título “WebCream Demo”, como vimos no navegador Web. Ao escrever o plano de testes, será de utilidade manter o navegador aberto com a página que você vai testar. Para completar a assertiva, especificamos que a resposta deve conter WebCream Demo. A Figura 11.4 mostra a árvore do plano de testes que criamos até agora.

Agora, estamos prontos para prosseguir para o teste que abre e fecha uma página com uma caixa de diálogo. Queremos fazer isso várias vezes; assim, primeiro adicionamos um controlador lógico Loop Controller ao Once Only Controller e especificamos 5 para o número de repetições do loop (*loop count*). A próxima tarefa é adicionar um HTTP Request Sampler, que simula o usuário clicando no botão Login Dialog. Para ver o que tem de ser enviado nessa solicitação, precisamos abrir o código-fonte da página HTML principal. Fazer uma busca por Login Dialog nos leva ao código mostrado na Listagem 11.4.

#### LISTAGEM 11.4 Código HTML para o botão Login

```
<form name="WebCreamForm" method="POST"
      action="http://localhost:8040/webcream/apps/WebCreamDemo"
      onSubmit="onSubmitForm( )"
>
...
<input type=button
       name="JButton31266642"
       value="Login Dialog"
       class=button
       OnClick=javascript:doSubmit('/button/JButton31266642')
       style="position:absolute;left:84;top:5;width:103;height:26"
>
...
</form>
```

Com algum conhecimento de HTML, você verá que, quando um usuário clica no botão, é invocado o código JavaScript `doSubmit`, passando '/button/JButton31266642' como parâmetro. Pesquisar por `doSubmit` na página e nos arquivos JavaScript incluídos produz o seguinte trecho de código, encontrado em `webcream_misc.js`:



**FIGURA 11.4** Árvore do plano de testes do JMeter, passo 2.

```
function doSubmit(action) {
    window.document.WebCreamForm._Action.value = action;
    onSubmitForm();
    window.document.WebCreamForm.submit();
}
```

Assim, concluímos que, quando um usuário clica no botão Login Dialog, é definido o parâmetro `_Action` como `/button/JButton31266642` e o formulário é submetido ao servidor. Acessando várias vezes a aplicação demo do WebCream com o navegador, podemos ver a string de ação mudando. Ela sempre se inicia com `/button/JButton`, mas os números restantes variam, de tempos em tempos, pois são gerados automaticamente. A maioria das páginas Web inclui alguma forma de conteúdo dinâmico, portanto a técnica que utilizamos para o WebCream também é útil para outras aplicações.

Para trabalhar com conteúdo gerado dinamicamente no JMeter, você vai precisar de variáveis e expressões regulares. O Regular Expression Extractor é um pós-processador que aplica uma expressão regular à resposta de um Sampler e coloca o valor extraído em uma variável. Expressões regulares são um mecanismo poderoso para trabalhar com textos; se você não estiver familiarizado com elas, recomendo conhecê-las. Há muitas referências e tutoriais na Web que abordam expressões regulares, e até um livro da O'Reilly, publicado nos Estados Unidos, chamado *Mastering Regular Expressions*.

Depois que uma variável é inicializada pelo extrator, seu valor pode ser passado para outros elementos de testes, como Samplers. Utilizaremos o extrator para obter os valores dos três parâmetros que definimos no `HTTP Request Defaults` (`_RequestId`, `_WindowTitle` e `_Action`). Clicamos com o botão direito no nó `Http Request - Init` e adicionamos um pós-processador chamado Regular Expression Extractor. Em seguida, acrescentamos - Re-

quest Id ao nome do extrator, pois ele será utilizado para recuperar o valor desse parâmetro. Na tela de configuração do extrator, especificamos `_RequestId` como nome de referência. Fazer isso informa o JMeter que o resultado da extração deve ser armazenado em uma variável chamada `_RequestId`. Examinando o código HTML na Listagem 11.3, que define os parâmetros `hidden` do formulário, chegamos a uma expressão regular que vai corresponder ao valor de `_RequestId`:

```
name="__RequestId" value="(\d+)"
```

Essa expressão utiliza caracteres estáticos para identificar unicamente a definição de `_RequestId`, e a máscara `\d+` para especificar um número qualquer de dígitos. Os parênteses especificam a parte da expressão a ser utilizada como resultado. Como estamos interessados somente no valor numérico, cercamos `\d+` com parênteses.

O template no extrator permite criar uma string a partir das correspondências encontradas através da aplicação da expressão regular. O template pode incluir texto estático e `$n$`, que representa a enésima correspondência da expressão regular. Estamos esperando apenas uma correspondência, e assim especificamos `$1$` como o template. Isso conclui a extração de `_RequestId`.

De maneira semelhante a `_RequestId`, devemos adicionar dois outros extratores, para as variáveis `_WindowTitle` e `_Action`. Este é um bom momento para treinar, descobrindo suas próprias expressões regulares e strings de templates. Mas vou tornar sua vida mais fácil: a expressão regular para `_WindowTitle` pode ser

```
name="__WindowTitle" value="(.)"
```

e o template pode ser `$1$`. A expressão regular `_Action` pode ser

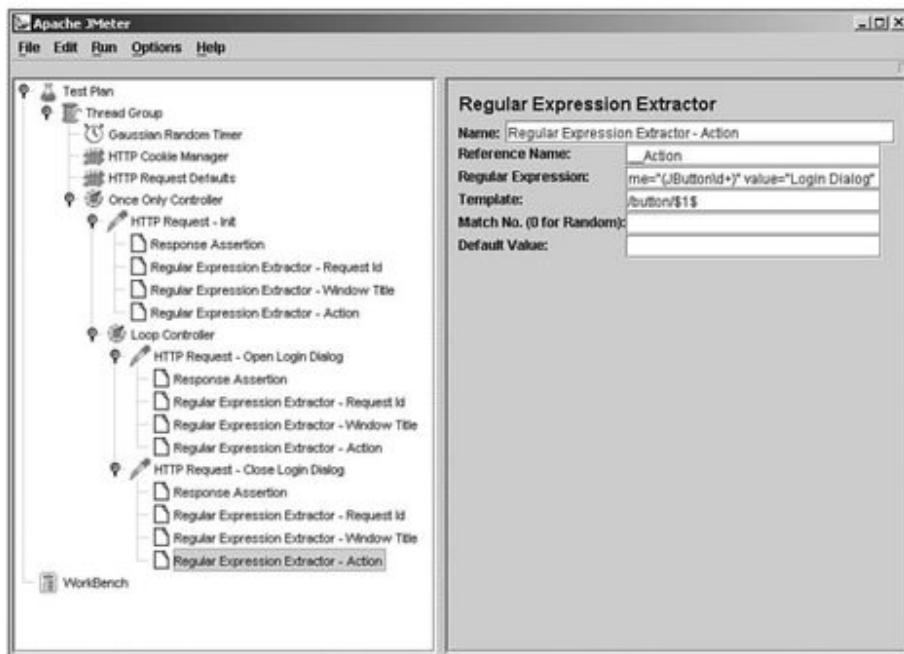
```
name="(JButton\d+)" value="Login Dialog"
```

e o template, `/button/$1$`. O segredo para chegar a uma expressão regular e um template é produzir uma string que possa ser usada como o valor de um parâmetro para a próxima solicitação. A Figura 11.5 mostra a árvore do plano de testes que criamos até agora.

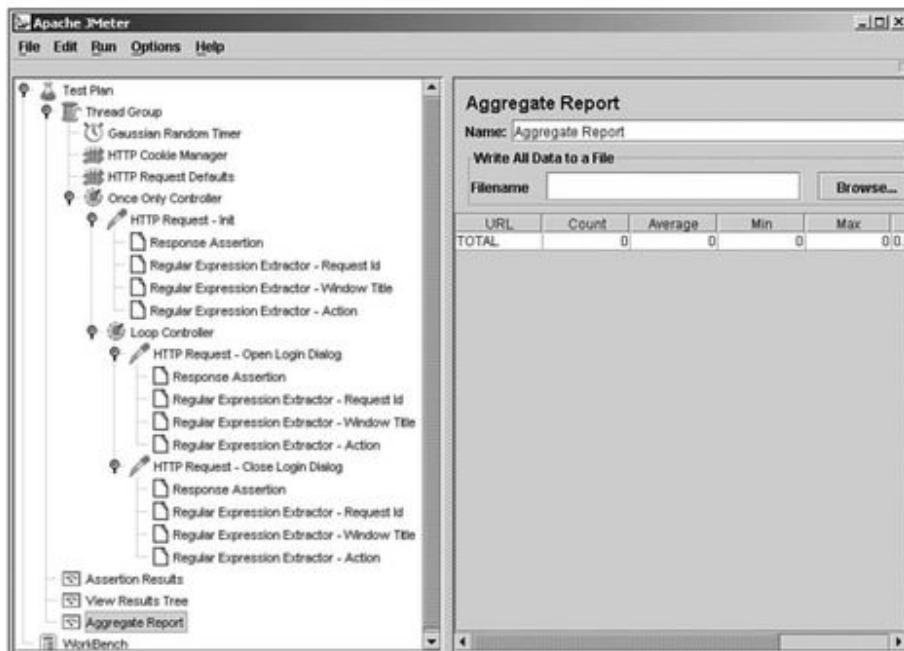
Ainda não estamos prontos para utilizar as variáveis, que contêm os valores extraídos, como parâmetros para solicitações HTTP. Já definimos os parâmetros no nó `HTTP Request Defaults`, assim vamos retornar a ele e especificar os valores. Para obter o valor de uma variável no JMeter, precisamos utilizar a sintaxe `${name}`, onde `name` é o nome da variável. Especificamos `_${_RequestId}` para o parâmetro `_RequestId`, `_${_WindowTitle}` para `_WindowTitle` e `_${_Action}` para `_Action`.

Estamos quase lá. Antes de executar os testes, precisamos adicionar alguns listeners para monitorar a execução. Os listeners mais úteis são `View Results Tree` e `Assertion Results`. O listener `View Results Tree` mostra cada solicitação com os seus dados e os da resposta; `Assertion Results` é uma maneira direta de identificar assertivas que falharam. Você não deve manter o `View Results Tree` na versão final do script de testes, devido ao overhead de desempenho associado com a coleta e o armazenamento de todos os dados. Por fim, adicionamos o listener `Aggregate Report` para produzir um resumo da execução e métricas de desempenho por solicitação.

Se você seguiu todos os passos corretamente, seu plano de testes deve estar como o da Figura 11.6.



**FIGURA 11.5** Árvore do plano de testes do JMeter, passo 3.



**FIGURA 11.6** Árvore do plano de testes do JMeter para o WebCream, concluída.

Certifique-se de que o Tomcat embutido no WebCream esteja em execução e acione o JMeter selecionando Start no menu Run. Depois de alguns minutos, o teste deve terminar, e você poderá ver o resultado nos listeners.

## Questionário rápido

1. Qual é o propósito dos testes de carga?
2. Qual é a diferença entre usuários simultâneos e usuários concorrentes?
3. Quais protocolos-cliente podem ser testados com o JUnit?
4. Como os casos de teste do JUnit asseguram a validade da resposta?
5. Quais protocolos-cliente podem ser testados com o JMeter?
6. Quais nós de configuração e de Samplers são utilizados em planos de testes do JMeter?
7. Como você monitora o progresso e os resultados de um plano de testes em execução no JMeter?
8. Quais são os benefícios e deficiências do JUnit e do JMeter?

## Resumo

- O propósito dos testes de carga é avaliar como o desempenho do sistema atende aos requisitos no nível de serviço sob uma carga determinada.
- Produtos comerciais e open source para testes de carga podem incluir a capacidade de gravar usuários virtuais automaticamente, criar scripts de testes programaticamente, suportar múltiplas linguagens de programação e protocolos de comunicação e gerar gráficos e relatórios.
- O JUnit é um framework open source que pode ser preparado para conduzir testes de carga simples em servidores RMI ou servidores Java comuns. Um caso de teste do JUnit é uma classe Java compilada e executada para o código de aplicações.
- O JMeter é uma ferramenta open source para testes de carga e medida de desempenho de sites Web e aplicações Java. Suporta servidores que aceitam conexões HTTP e FTP, mas também pode ser utilizado para testar bancos de dados, scripts Perl e objetos Java.
- O JMeter suporta conteúdo dinâmico com extractores e variáveis de expressões regulares.
- Os listeners do JMeter podem produzir relatórios e gráficos de desempenho.

**TABELA 12.1****Associações entre elementos e recursos da interface gráfica**

TIPO DO ELEMENTO	EXEMPLOS DE ELEMENTOS	RECURSO EM JAVA
Texto	Mensagem de diálogo, alerta, texto de exceção	String Java dentro de um arquivo de classe ou texto em um arquivo de configuração
Imagen	Tela de splash, ícone, imagem de fundo	Arquivos JPG ou GIF (ou outros formatos de imagem) em um diretório da aplicação ou dentro de um JAR
Elemento visual	Layout de janela, composição de menu, cores	Arquivo de classe Java contendo o elemento correspondente
Definição de configuração	Limites programáticos, como número máximo de conexões, data de expiração, número de threads	Um arquivo de classe Java se a definição for codificada diretamente; arquivos .properties ou .xml se a definição for configurável
Áudio	Música de fundo, efeito sonoro	Arquivos AIFF, WAV ou AU que representam o clipe

## Hackeando textos

A aplicação Chat utiliza o texto `Send message` no item de menu e no `ToolTip` do botão na barra de ferramentas. Embora isso deixe claro o significado da ação, o texto pode ser considerado sem graça pela geração mais jovem. Para satisfazer esse público, vamos hackear a aplicação Chat para que utilize `Unleash the fury` em vez de `Send message`. Vamos trabalhar com a versão distribuída (binária) da aplicação, e supor que não temos acesso ao código-fonte.

Como a aplicação Chat vem empacotada em um JAR, a primeira coisa a fazer é descompactar `CovertJava/distrib/lib/chat.jar` em um diretório de trabalho temporário. Precisamos então localizar as classes que definem as strings atualmente exibidas no Chat. Utilizando o FAR ou uma ferramenta de pesquisa, fazemos uma pesquisa binária por `Send message` em todos os arquivos no diretório de trabalho e em seus subdiretórios. O resultado da execução da pesquisa deve ser o arquivo `MainFrame.class`, que descompilamos como descrito no Capítulo 2. Pesquisar `Send message` no código-fonte descompilado nos leva ao método `jbInit`, um trecho que é mostrado na Listagem 12.1.

**LISTAGEM 12.1** Trecho de `jbInit` mostrando as strings de texto

```
private void jbInit() throws Exception {
    ...
    btnSend.setToolTipText("Send message");
}
```

**LISTAGEM 12.1** Continuação

```
btnHelp.setIcon(imageHelp);
...
menuEditSend.setText("Send message");
menuEditSend.addActionListener(this);
...
}
```

Utilizando a técnica de patching apresentada no Capítulo 5, podemos simplesmente substituir `Send message` por `Unleash the fury` no código-fonte descompilado e adicioná-la como um patch à aplicação Chat. Isso foi bastante fácil e, felizmente, na maioria das aplicações deve ser igualmente simples. As complicações podem surgir se o código da aplicação estiver ofuscado com codificação de strings, mas com o conhecimento adquirido nos capítulos anteriores, isso não deve nos impedir de localizar a classe a ser corrigida com um patch.

Boas aplicações não codificam diretamente as strings de texto no código. Em vez disso, as mensagens são armazenadas em pacotes de recursos ou em arquivos de configuração; isso simplifica a manutenção e a localização (adaptação de uma aplicação ao local onde está sendo utilizada). Contanto que as strings não estejam codificadas, a pesquisa binária realizada no conteúdo do diretório de trabalho continuará produzindo os resultados desejados. Quando o arquivo contendo a string for encontrado, ele pode ser atualizado com o novo texto. Em seguida, a aplicação precisa ser reempacotada para que a alteração tenha efeito.

## Hackeando imagens

Trabalhar com imagens é um pouco mais complexo do que com textos. Diferentemente de textos, não é possível simplesmente procurar por uma imagem, pois não sabemos o que utilizar como critérios de pesquisa. O primeiro passo, portanto, é encontrar o nome da imagem e determinar como ela é carregada pela aplicação, para que então possamos aplicar um patching. Não há uma maneira fácil e direta de localizar o nome da imagem. É mais provável que a aplicação carregue a imagem no código de interface gráfica, mas muitas vezes desenvolvedores utilizam um framework ou uma classe de gerenciamento para ajudar com o "trabalho braçal" necessário para carregar uma imagem em Java. Sugiro explorar a aplicação expandida (ou seja, não armazenada em um JAR), ou talvez procurar em todos os arquivos `.jpg` e `.gif`. Na maioria das vezes, o nome da imagem é o melhor indício sobre onde ela é utilizada. Por exemplo, uma tela de splash poderia se chamar `splash.gif` e o ícone para o botão na barra de ferramentas New talvez seja `new.gif`. Uma aplicação Java típica não tem muitas imagens, e assim você deve ser capaz de localizar facilmente a que está procurando. Obviamente, visualizar a imagem é a maneira de confirmar que você a encontrou. (Eu não precisava dizer isso, precisava?)

E se houvesse muitas imagens e o programador que codificou a aplicação tivesse utilizado nomes de arquivos como `img0045.gif`? Você ainda poderia adivinhar qual imagem é

a sua, examinando o tamanho, ou simplesmente percorrendo todas seqüencialmente. Lembre-se de que a aplicação talvez armazene imagens em um arquivo JAR diferente, e assim você tem de abrir todas ao fazer a pesquisa.

Agora veja um cenário de pior caso: você pode ver que a aplicação está exibindo uma imagem e precisa alterá-la desesperadamente, mas, depois de passar por todos os arquivos de imagens empacotados na aplicação, não consegue localizá-la. Há várias opções para determinar de onde vem a imagem. Como a maneira simples não funcionou, teremos de cavar mais fundo e chegar ao lugar no código em que a imagem é carregada. Mais uma vez, os capítulos anteriores abordaram técnicas de engenharia reversa, de modo que sugiro pesquisar os arquivos de classe .gif ou .jpg como um ponto de partida. Nunca vi uma aplicação que armazenasse arquivos GIF com uma extensão .txt e, seguramente, espero nunca viver para ver esse dia. Por exemplo, para alterar a imagem na caixa de diálogo About da aplicação Chat, podemos procurar todos os arquivos GIF e JPG no diretório de trabalho. Fazer isso nos leva ao diretório `covertjava.chat.images` que contém imagens. Como ele contém somente algumas imagens, podemos simplesmente visualizá-las para determinar qual é a imagem utilizada na caixa About. Como alternativa, poderíamos ter procurado por .gif, o que nos levaria novamente ao arquivo de classe `MainFrame`. Descompilando esse arquivo, iríamos descobrir que a aplicação Chat utiliza `saturn_small.gif` na caixa About.

Por fim, também podemos procurar pelo método `getImage` de `java.awt.Toolkit`, que é a maneira mais comum de carregar uma imagem. Veja a seguir alguns exemplos de código Java que carregam imagens:

```
/**  
 * Cria um ícone a partir de uma imagem no diretório "resources/images"  
 * dentro de um arquivo .jar  
 */  
public ImageIcon createImageIcon(String fileName) throws Exception  
{  
    String path = "/resources/images/" + fileName;  
    return new ImageIcon(getClass( ).getResource(path));  
}  
/**  
 * Carrega uma imagem a partir do diretório "images" no disco  
 */  
public Image loadImage(String fileName) throws Exception  
{  
    return Toolkit.getToolkit( ).getImage("images/" + filename);  
}
```

Depois de localizar o código que está carregando a imagem, você deve ser capaz de entender de onde ela vem. Se a imagem não estiver empacotada na aplicação, ela provavelmente está sendo acessada via um URL externo, semelhante a <http://mycompany.com/app/images/splash.gif>. Nesse ponto, você (o hacker instruído) sabe exatamente como atacar o problema. Utilize seu navegador Web para fazer o download da imagem a partir do URL, salve-a localmente, edite-a e empacote-a junto com a aplicação. Então corrija com

um patch a classe que carrega a imagem; em vez do URL HTTP, utilize `jar:file/URL` para carregá-la do seu JAR, ou `Toolkit.getImage( )` para carregá-la de um diretório. Pronto, você conseguiu!

## Hackeando arquivos de configuração

Arquivos de configuração são apenas outro tipo de recurso de aplicação, e a abordagem para fazer hacking deles é a mesma para textos e imagens. Não vamos tratar dos arquivos de configuração armazenados no diretório da aplicação, que podem ser facilmente modificados; vamos falar dos arquivos que podem estar dentro do `.jar` ou `.zip` e que não necessariamente foram planejados para serem editados. Como antes, vamos descompactar o JAR ou ZIP da aplicação, e examinar a estrutura de diretórios, para procurar arquivos suspeitos. Se a estrutura de diretórios for grande, você pode procurar por todos os arquivos com extensão `.properties` ou `.xml`. É bem provável que essa pesquisa simples produza o resultado desejado – caso contrário, você pode recorrer às técnicas de engenharia reversa descritas nos capítulos anteriores para localizar a parte do código que utiliza a configuração. Podemos então rastrear nosso caminho até o código que carregar a configuração; isso vai fornecer informações suficientes para determinar a estratégia de patching.

## Questionário rápido

1. Suponha que você esteja utilizando uma biblioteca que exibe uma mensagem `Unknown error` se for capturada uma exceção genérica. Como você pode alterá-la para, digamos, `Internal error, please contact technical support?`
2. Como alterar a caixa de diálogo `About` da aplicação Chat para exibir o seu nome na imagem?
3. Como descobrir se o número máximo de conexões concorrentes imposto pela aplicação que você utiliza é configurável?

## Resumo

- Hackear os elementos da interface gráfica requer a localização e o patching de recursos que foram utilizados para criá-la.
- Hackear textos exige localizar e modificar o arquivo `.class` ou o arquivo de configuração (`.properties`, `.xml` ou `.resource`) que o define.
- Hackear imagens requer encontrar o arquivo da imagem (normalmente JPEG ou GIF).
- Hackear arquivos de configuração envolve a edição de textos simples, depois que você localizar o arquivo que define as propriedades de interesse.

# Aplicações de engenharia reversa

*"Se engenheiros construissem prédios como programadores escrevem programas, um único pica-pau seria capaz de destruir a civilização."*

As Leis de Murphy aplicadas à tecnologia

## Elementos e recursos da interface com o usuário

No Capítulo 5, discutimos o patching de classes para alterar a lógica da aplicação. Neste capítulo, vamos discutir a alteração de elementos de interface com o usuário, tais como mensagens, alertas, prompts, imagens, ícones, menus e cores. Os mesmos princípios descritos no Capítulo 5 também se aplicam a este capítulo. A primeira tarefa é localizar um recurso que precisa ser alterado. Em seguida, uma alteração pode ser feita e a aplicação, atualizada para assegurar que o novo recurso seja utilizado no lugar do antigo. A Tabela 12.1 mostra a associação entre elementos da interface gráfica e os recursos a serem corrigidos com patch.

Aprender por meio de exemplos é uma das maneiras mais fáceis de digerir informações; assim vamos escolher uma aplicação e modificá-la de acordo com nossos caprichos. Você viu vários exemplos da aplicação Chat em todos os capítulos anteriores; mantendo a tradição, vamos hackeá-la para ilustrar cada uma das técnicas.

# 12

## *NESTE CAPÍTULO*

- ▶ Elementos e recursos da interface com o usuário 109
- ▶ Hackeando textos 110
- ▶ Hackeando imagens 111
- ▶ Hackeando arquivos de configuração 113
- ▶ Questionário rápido 113
- ▶ Resumo 113

# 13

## *NESTE CAPÍTULO*

- ▶ Definição de eavesdropping 114
- ▶ Eavesdropping em HTTP 115
- ▶ Técnicas de eavesdropping no protocolo RMI 119
- ▶ Eavesdropping do driver JDBC e de instruções de SQL 122
- ▶ Questionário rápido 124
- ▶ Resumo 124

# Técnicas de eavesdropping

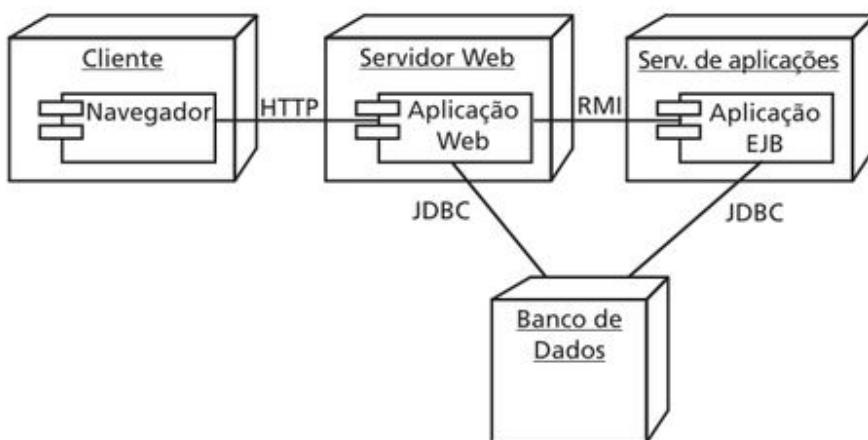
*"Você pode observar muito apenas olhando."*

Primeira lei de Berra

## **Definição de eavesdropping**

Os capítulos anteriores focaram principalmente no trabalho com bytecode e recursos de aplicações. Aplicações multicamadas, que são dominantes no lado do servidor, oferecem novas oportunidades para a engenharia reversa e hacking. É uma prática comum distribuir camadas da aplicação como processos separados, que se comunicam entre si via protocolos de rede. Por exemplo, um navegador Web que exibe um frontend HTML nas estações de trabalho dos usuários utiliza o Hypertext Transfer Protocol (HTTP) para se comunicar com o servidor Web. Por sua vez, o servidor Web, em geral, utiliza RMI ou IIOP para se comunicar com o servidor de aplicações. O servidor de aplicações utiliza JDBC para se comunicar com o banco de dados. Um diagrama de deployment clássico para aplicações Java multicamadas distribuídas é mostrado na Figura 13.1.

Este capítulo apresenta várias técnicas de *eavesdropping* (“espionagem”) que podem ser empregadas na comunicação entre as camadas distribuídas. *Eavesdropping* consiste em interceptar e registrar a troca de mensagens entre cliente e servidor. Isso pode facilitar a solução de problemas ou o ajuste de desempenho de sistemas distribuídos complexos, e fornecer dicas sobre o design de aplicações e princípios de comunicações.



**FIGURA 13.1** Diagrama de deployment de uma aplicação multicamadas.

## Eavesdropping em HTTP

Primeiro vamos examinar como fazer o eavesdropping em aplicações Web e Web Services, que utilizam HTTP como protocolo de transporte para se comunicarem com seus clientes. Mensagens HTTP consistem de um cabeçalho e o conteúdo, que é enviado pela rede via TCP/IP. Para que um cliente seja capaz de conversar com um servidor, ele precisa conhecer o hostname ou o endereço IP e a porta em que o servidor está ouvindo. Mensagens HTTP são enviadas como texto simples e, por isso, podem ser facilmente lidas e entendidas por pessoas.

Para fazer o eavesdropping da comunicação cliente/servidor, você deve interceptar a troca de mensagens. Uma das alternativas para isso é colocar um intermediário que rastreie e faça o tunelamento de mensagens HTTP entre o cliente e o servidor. Outra opção é monitorar a comunicação na camada de protocolo de rede, filtrando as mensagens trocadas entre o cliente e o servidor. Vamos examinar essas duas alternativas nas seções a seguir. Mais uma vez, utilizaremos o WebCream como a aplicação do lado do servidor, porque ele fornece conteúdo HTML estático e dinâmico via servlets e páginas JSP.

### Utilizando um túnel para capturar a troca de mensagens HTTP

Um *túnel*, nesse contexto, é um pseudo-servidor posicionado entre o cliente real e o servidor real, a fim de interceptar e rastrear a troca de mensagens. Em vez de conversar diretamente com o servidor, o cliente é reconfigurado para enviar mensagens ao túnel; o túnel é configurado para enviar solicitações ao servidor, fazendo o papel do cliente, e encaminhar as respostas do servidor de volta ao cliente. O túnel registra as mensagens trocadas, na tela ou em um arquivo, permitindo que um hacker ou desenvolvedor analise os detalhes da conversa. A natureza sem-estado e textual do HTTP torna-o uma boa escolha para o tunelamento.

Empregaremos o utilitário TCPMON, distribuído junto com o projeto Apache AXIS, para fazer o tunelamento de solicitações do navegador para o Tomcat do WebCream. Embora o TCPMON não seja nem de longe o software mais sofisticado de tunelamento, é

gratuito e faz um bom trabalho na maioria dos casos. Baixe o AXIS no site da Apache e, se você não tiver um script para iniciar o TCPMON, copie o subdiretório `CovertJava\bin\axis` para o diretório onde instalou o AXIS. O TCPMON recebe três parâmetros de linha de comando: a porta na qual escutar, o host e o número da porta do servidor para o qual fazer o tunelamento das solicitações. Como o WebCream executa o Tomcat na porta 8040, vamos rodar o TCPMON na porta 8000 e instruí-lo para encaminhar solicitações à porta 8040 do *localhost*.

Precisamos fazer algumas alterações simples de configuração no WebCream para que ele suporte o tunelamento através do AXIS. O WebCream gera formulários de páginas com URLs completos para a submissão, incluindo o host e o número de porta. Como queremos que todo o tráfego passe pelo TCPMON, precisamos fazer com que o WebCream sempre submeta os formulários à porta 8000, em vez da 8040, como ele faz por padrão. Podemos fazer isso adicionando as duas linhas a seguir ao arquivo `WebCreamconf\WebCream-Demo.properties` (consulte a documentação do WebCream para informações adicionais):

```
html.docsURL=http://localhost:8040/webcream  
html.submitURL=http://localhost:8000/webcream/apps/WebCreamDemo
```

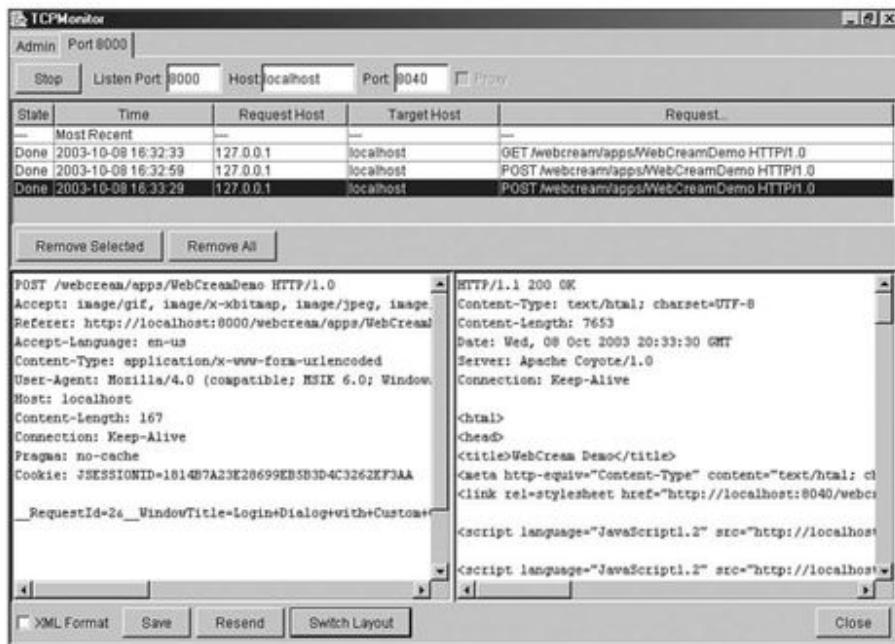
Inicie o Tomcat embutido no WebCream utilizando `WebCream\bin\startServer.bat`. Abra seu navegador Web e digite o seguinte na linha de endereços:

`http://localhost:8000/webcream/apps/WebCreamDemo`

Observe que utilizamos a porta 8000, que é a porta na qual o TCPMON está sendo executado, em vez de apontar o navegador diretamente para o Tomcat (que está rodando na porta 8040).

O navegador deve exibir a página principal da aplicação de demonstração do WebCream. Alterne para o TCPMON e certifique-se de que é possível ver a solicitação interceptada no painel superior (ou esquerdo). O TCPMON pode demorar um pouco para detectar que a solicitação foi completamente transmitida, e assim espere até que o status da solicitação no painel superior mude para `Done`. Volte ao navegador e clique no botão `Login Dialog` para abrir a próxima página; insira `Neo` como nome de usuário e `Wakeup` como senha e clique em `OK` (Sinta-se à vontade para selecionar seu próprio nome de usuário e senha.). Se você alternar para a tela do TCPMON, verá que ela está semelhante à Figura 13.2.

Agora, podemos examinar as solicitações e os dados interceptados pelo TCPMON. O painel superior mostra três mensagens, que correspondem ao número de páginas solicitadas pelo navegador. As informações mais interessantes são mostradas nos painéis inferiores esquerdo e direito. O painel esquerdo mostra os dados e o cabeçalho da solicitação; o painel direito mostra o cabeçalho e o conteúdo da resposta. Examinando os vários atributos e elementos de conteúdo, podemos entender melhor a troca de dados entre o navegador e o servidor. Por exemplo, selecionando a terceira solicitação no painel superior e examinando os dados da solicitação, podemos ver os valores reais do nome de login (`Neo`) e senha (`Wakeup`). No cabeçalho, vemos um cookie, `JSESSIONID`, que pode ser usado para "seqüestrar" a sessão do usuário.



**FIGURA 13.2** TCPMON mostrando solicitações interceptadas.

### Utilizando um sniffer de rede para capturar a troca de mensagens HTTP

O tunelamento é uma maneira simples e eficaz de ver o que é de fato transmitido entre o cliente e o servidor. Ele é apropriado para depurar e analisar aplicações Web, mas tem a desvantagem de requerer uma reconfiguração do cliente (e possivelmente até do servidor, como tivemos de fazer com o WebCream), para enviar solicitações ao agente de tunelamento, em vez de diretamente ao servidor. A segunda técnica que você vai aprender permite usar técnicas eavesdropping no nível de protocolo de rede, o que evita as limitações do tunelamento.

Todas as comunicações distribuídas passam por uma camada de protocolos suportados pela JVM, pelo sistema operacional e pelo driver de rede. Os protocolos são empilhados um sobre o outro, ou seja, os de nível mais alto baseiam-se nos de nível mais baixo para realizar tarefas mais básicas. Dessa forma, o HTTP utiliza o TCP, que por sua vez utiliza o IP. Uma única mensagem HTTP pode ser representada por vários pacotes IP de baixo nível, e o trabalho das camadas de rede é quebrar e depois remontar os pacotes. A maioria das redes físicas é composta de segmentos Ethernet de estações de trabalho interconectadas. Dentro de um segmento, pacotes em um dos nós são transmitidos para todos os nós, independentemente do endereço do nó de destino. Para se comunicarem com um computador fora do segmento, os roteadores redirecionam os pacotes para outros segmentos. O resumo dessa explicação muito simplificada é que, quando uma aplicação em um host comunica-se com uma aplicação remota em um host diferente, os pacotes de protocolos têm de atravessar outros hosts de rede antes de alcançar o alvo. O sniffing e o monitoramento de rede tiram proveito desse princípio para espionar comu-

nicações. Normalmente, um nó de rede aceita somente os pacotes destinados a ele, ignorando todos os outros. Entretanto, um nó pode estar em modo *promiscuo*, aceitando todos os pacotes independentemente do endereço de destino. Um engenheiro ou hacker pode examinar os pacotes e o seu conteúdo para obter informações sobre a comunicação das aplicações.

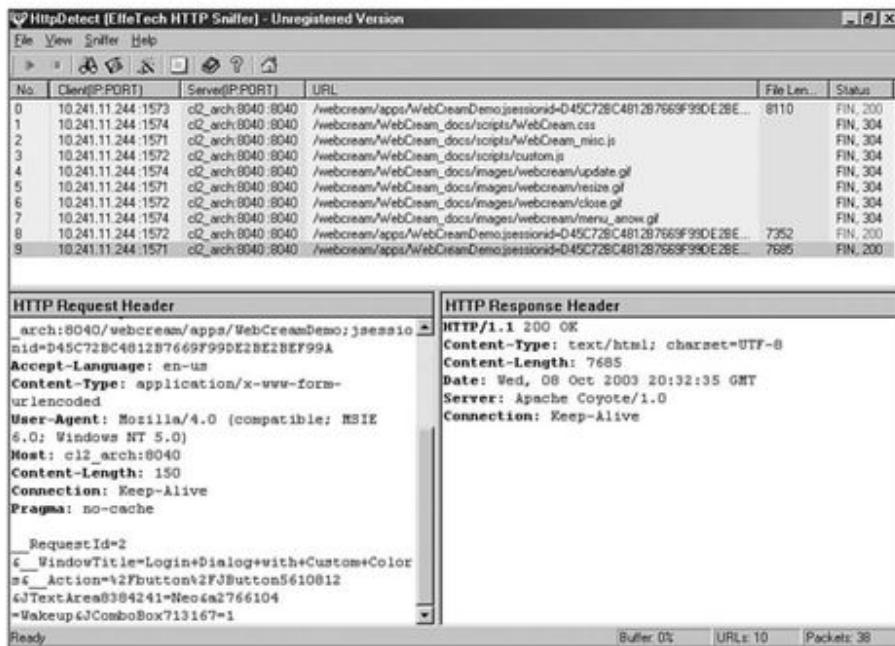
Trabalhar no nível de protocolo pode ser difícil e demorado. Como o HTTP é um protocolo muito comum, há produtos disponíveis que simplificam o eavesdropping em comunicações HTTP. Vamos examinar o HTTP Sniffer da EffeTech, uma aplicação shareware para Windows. Vamos tentar realizar a mesma tarefa de interceptar a comunicação entre o navegador e o Tomcat do WebCream, para ver se obtemos resultados diferentes dos obtidos com o TCPMON. Uma das desvantagens da utilização de um sniffer de rede é que ele não funciona quando o cliente e o servidor estão rodando no mesmo host. Nenhum pacote é passado para o driver de rede, portanto o sniffer não é capaz de capturar solicitações e respostas do protocolo. Assim, para esta seção e a seguinte, você precisa colocar o servidor Tomcat em execução em um host de rede diferente do host do navegador web. Se você não tiver acesso a outra máquina, poderá utilizar um site Web qualquer em vez da aplicação demo do WebCream.

Baixe e instale o HTTP Sniffer da EffeTech. Será instalada a WinPcap, uma biblioteca de captura de pacotes adicionada ao Windows como um driver de dispositivo para capturar dados brutos da placa de rede. Como o WebCream está executando o Tomcat na porta 8040, precisamos adicionar essa porta ao filtro do sniffer, utilizando o item Filter do menu Sniffer. Talvez você precise mudar o adaptador de rede atual no sniffer se a gravação não produzir nenhum resultado, mas, por enquanto, deixe-o com o valor padrão. Inicie o Tomcat do WebCream no servidor remoto e abra seu navegador Web. Observe que o sniffer pode ser executado na mesma máquina do navegador, na máquina do servidor Web ou em qualquer máquina conectada ao mesmo segmento Ethernet, dos hosts do navegador ou do servidor. Inicie a gravação usando o menu Sniffer ou a barra de ferramentas, e digite o URL do servidor no navegador. Se estiver testando com o WebCream, abra a tela principal, clique no botão Login Dialog e vá para a próxima página; insira **Neo** como nome de usuário e **Wakeup** como a senha, depois clique em OK. Depois de parar a gravação, a tela do HTTP Sniffer deve se parecer com a Figura 13.3.

A maneira como as informações são apresentadas e os tipos das informações interceptadas são quase idênticos ao TCPMON. Além de apresentar uma interface com o usuário mais limpa, o HTTP Sniffer mostra outros recursos obtidos pelo navegador a partir do servidor, como arquivos GIF e JavaScript. Mais uma vez, examinando os dados da solicitação podemos obter os valores dos parâmetros de formulários, como nome de usuário e senha. A beleza dessa abordagem é que não tivemos de fazer nada na aplicação-alvo, contudo capturamos um log completo da comunicação entre o cliente e o servidor.

## Protegendo aplicações Web contra eavesdropping

Em pouco tempo, você aprendeu a fazer o eavesdropping em interfaces Web. A facilidade com que você pode obter valores potencialmente sigilosos é bastante alarmante. Algumas precauções podem tornar muito mais difícil o trabalho de hackear uma aplicação Web. A maneira mais simples e eficaz de proteger a integridade dos dados e tornar a



**FIGURA 13.3** HTTP Sniffer mostrando solicitações interceptadas.

comunicação cliente/servidor segura é via o uso do Hypertext Transfer Protocol Secure (HTTPS). Esse protocolo obriga que um cliente estabeleça um canal seguro com o servidor utilizando o Secure Sockets Layer (SSL) antes de enviar quaisquer dados HTTP. Depois que o canal é estabelecido, a troca de mensagens ocorre da mesma forma que com HTTP. O benefício do HTTPS é que todos os dados são criptografados; portanto mesmo que alguém intercepte um pacote de rede, descriptografa-lo é praticamente impossível. O SSL traz um overhead que pode diminuir a velocidade do servidor, portanto, para aplicações de alto desempenho, realizar todas as comunicações via HTTPS talvez não seja praticável. Uma boa opção é utilizar o HTTPS seletivamente, ou continuar utilizando HTTP e criptografar o conteúdo sigiloso na camada de aplicação. Se a interface com o usuário for um navegador Web, funções JavaScript podem ser utilizadas para criptografar e descriptografar no lado do cliente.

## Técnicas de eavesdropping no protocolo RMI

O Java Remote Method Invocation (JRMI) utiliza o Java Remote Method Protocol (JRMP), específico para Java, ou o Internet Inter-ORB Protocol (IIOP), para enviar mensagens binárias a hosts remotos. O JRMP e o IIOP utilizam Transmission Control Protocol/Internet Protocol (TCP/IP) para transportar as mensagens pela rede, o que torna o canal de comunicação sujeito a sniffing de rede. Teoricamente, você poderia escrever um túnel que receberia e gravaria as mensagens antes de passá-las adiante para o receptor desejado, mas isso seria uma tarefa muito entediante. Portanto, usaremos o sniffing de rede para fazer o eavesdropping na comunicação RMI. Infelizmente, nenhuma ferramenta tem suporte nativo a protocolos Java de nível mais alto, tal como o suporte a HTTP pelo

HTTP Sniffer. Isso significa que devemos trabalhar em um nível mais baixo, analisando pacotes TCP e IP, que representam todas ou partes das chamadas RMI. Vamos treinar espiando a conversa entre dois usuários de aplicações Chat.

## O protocolo de transporte RMI

O RMI utiliza o conceito de stream para representar o formato da comunicação física. Internamente, a comunicação tem dois streams associados – de saída (*out*) e de entrada (*in*). Esses streams são mapeados para os streams de sockets correspondentes, e utilizados para enviar e confirmar o recebimento de mensagens. O stream de saída consiste em um cabeçalho seguido por uma seqüência de mensagens. Se for utilizado HTTP, o stream de saída será embutido em uma mensagem HTTP. O cabeçalho contém a identificação do protocolo e atributos que descrevem o tipo de protocolo utilizado. A parte essencial de uma chamada RMI está contida na sua seção de mensagem. Mensagens de saída podem ser chamadas de métodos, pings remotos, ou tráfego de coleta de lixo; mensagens de entrada podem ser o resultado de uma chamada, ou o reconhecimento de um ping.

Para executar uma chamada remota, o RMI utiliza o Java Object Serialization Protocol para formatar o nome de métodos e de valores de parâmetros numa estrutura binária, que é enviada pela conexão física. Portanto, todas as chamadas remotas seguem o mesmo formato, no nível binário. Não é necessário conhecer as especificidades do transporte para fazer eavesdropping em RMI, mas, se você quiser obter informações adicionais, visite <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmitoc.html>. Para nosso propósito é suficiente saber que, quando duas aplicações Java utilizam RMI para trocar mensagens, uma conexão TCP/IP de baixo nível, com streams de entrada e saída, é criada e utilizada.

## Utilizando um sniffer de rede para interceptar mensagens RMI

Uma das ferramentas mais populares de sniffing de rede é o Ethereal. Ele é gratuito e tem versões para Unix e Windows. Com o Ethereal, pode-se analisar praticamente qualquer protocolo concebível (mas não protocolos RMI) e, embora sua interface com o usuário seja um pouco limitada, ele será suficiente, por falta de uma ferramenta melhor. O Ethereal utiliza a WinPcap, a mesma biblioteca usada pelo HTTP Sniffer para capturar pacotes de rede. Baixe, instale e execute o Ethereal. Inicie a gravação do tráfego de rede, só para ter uma idéia sobre os tipos de informações que ele pode capturar. Abra o navegador e visite alguns sites; então pare de gravar e examine os itens exibidos no painel superior. Você verá vários pacotes. Nossa primeira tarefa é configurar a ferramenta para mostrar somente as informações em que estamos interessados.

Iniciamos abrindo a caixa de diálogo Capture Options, exibida ao selecionar Start no menu Capture. Desmarque Capture Packets no botão Promiscuous Mode, selecione Update List of Packets in Real Time e certifique-se de que todos os botões Stop Capture After estejam desmarcados. Como o sniffing de rede requer que o cliente e o servidor rodam em hosts diferentes, execute a aplicação Chat em duas máquinas. Clique em OK para confirmar as opções de captura e iniciar a gravação. Então alterne para a aplicação Chat e envie uma mensagem Hi Alex ao outro host (Não vou ficar ofendido se você utilizar seu próprio nome.). Envie mais uma mensagem e lembre-se do texto (minha segunda

mensagem é `IT jobs are becoming boring`); você precisará desse texto mais tarde. Então volte ao Ethereal e pare a captura.

Capturamos vários pacotes e, para ver a conversa da aplicação Chat, precisamos de uma maneira de filtrar os pacotes que transportam os fragmentos das mensagens textuais enviadas. Podemos iniciar procurando um pacote que transporta a string que sabemos ter sido enviada durante a gravação. Selecione Find Frame no menu Edit para exibir a caixa de diálogo Search. Digite uma parte da mensagem que você enviou (eu digitaria Alex, por exemplo) e selecione a opção String no grupo de botões Find Syntax. Isso instrui o Ethereal a procurar a string em qualquer parte do pacote. Clique em OK. O pacote que continha a string deve ser selecionado. Nos painéis inferiores, você deve ver o conteúdo do pacote, que deve incluir sua string de pesquisa dentro do conteúdo binário. Como uma chamada RMI pode ser quebrada em várias mensagens TCP/IP, geralmente ajuda utilizar o recurso Follow TCP Stream, que remonta o fluxo e mostra-o em uma janela separada. Clique com o botão direito no frame do pacote selecionado e escolha Follow TCP Stream.

Nesse ponto, estamos examinando a versão binária de um pacote RMI. Os dados do pacote são um pouco obscuros, mas com um pouco de paciência podemos obter algum conhecimento deles. O pacote inicia com o cabeçalho `JRMI`, indicando que o protocolo JRMI é utilizado para transporte no RMI. Em seguida, há o endereço IP do host de origem, o ID de objeto do servidor e informações gerais sobre a DGC (Distributed Garbage Collection, coleta de lixo distribuída). O conteúdo da mensagem está no final. No meu caso, posso ver a string `Hi Alex`, seguida por `covertjava.chat.MessageInfo`, que supomos ser o nome de classe da mensagem. Em seguida vêm outros parâmetros da mensagem, como o hostname e o nome de usuário.

Depois de examinar o stream TCP/IP que representa a mensagem RMI enviada pela aplicação Chat, podemos supor que as mensagens subsequentes terão o mesmo formato. Em outras palavras, mesmo que o texto da mensagem mude, o cabeçalho e o formato da mensagem permanecerão os mesmos. Se essa suposição estiver correta, devemos ser capazes de procurar mensagens da aplicação Chat com base em uma substring, no cabeçalho ou no formato da mensagem. Para testar essa teoria, vamos tentar pesquisar todos os frames `covertjava.chat.MessageInfo` utilizando a caixa de diálogo Find Frame. Se a string de filtragem estiver configurada na parte inferior da tela, redefina o filtro. Vá então para o primeiro frame e execute a pesquisa. O primeiro frame que encontrei tem a string `Hi Alex`; depois de procurar o próximo frame, encontrei uma mensagem com `IT jobs are becoming boring` no conteúdo. Isso confirma a suposição de que a aplicação Chat permite acompanhar as conversas entre usuários.

Uma abordagem semelhante pode ser utilizada com outras aplicações. O Ethereal funciona em um nível muito baixo, mas, com atenção e análise, você pode fazer *eavesdropping* da comunicação de rede entre quaisquer aplicações.

## Protegendo aplicações RMI contra eavesdropping

Com base no que aprendemos sobre sniffing de rede, podemos concluir que não se pode realmente evitar a interceptação das comunicações de rede. No final, os dados têm mesmo de viajar pelos cabos e passar por vários nós da rede, o que os torna suscetí-

veis à interceptação. A única maneira de proteger os dados é criptografá-los. O SSL oferece recursos padronizados para tornar canais de rede seguros e pode ser utilizado para a comunicação RMI. A Java Secure Sockets Extension (JSSE) é um conjunto de APIs que permite a aplicações em Java tirarem proveito do SSL, para a criptografia de dados, autenticação e garantia de integridade de mensagens. O RMI permite que aplicações fornecam fábricas de sockets personalizadas para objetos exportados, que são utilizados em vez dos sockets TCP/IP padrão. Utilizar fábricas SSL do JSSE como fábricas de sockets personalizadas, tanto para o cliente como para o servidor, torna o canal mais seguro. Um exemplo de casamento entre RMI e SSL é fornecido no site Web da JavaSoft em:

<http://java.sun.com/j2se/1.4.1/docs/guide/rmi/socketfactory/SSLInfo.html>

A comunicação SSL introduz overhead que é possivelmente desnecessário para todas as possíveis trocas de dados cliente/servidor. Uma melhor opção talvez seja continuar a utilizar fábricas de sockets padrão e criptografar as partes sigilosas dos dados utilizando JSSE. Um exemplo da utilização da criptografia em Java é mostrado no Capítulo 19, “Protegendo aplicações comerciais contra hacking”.

## Eavesdropping do driver JDBC e de instruções de SQL

A maioria das aplicações servidoras utiliza um banco de dados para armazenar e recuperar dados. Entender como a aplicação interage com o banco de dados e quais instruções SQL ele utiliza pode ser uma excelente ajuda no ajuste de desempenho, ou para fazer engenharia reversa. A tecnologia dominante para persistência é o JDBC, e isso não mudará tão cedo. Para utilizar JDBC, uma aplicação deve identificar e carregar um driver, obter uma conexão e então executar instruções para realizar atualizações e consultas. A lógica mais interessante, do ponto de vista de ajuste do desempenho e da engenharia reversa, é a estrutura e os parâmetros das instruções SQL. É de conhecimento geral que o desempenho dos bancos de dados depende muito da eficácia das instruções SQL nas aplicações. Analisar o SQL e os tempos de execução pode levar a aprimoramentos no lado da aplicação, no lado do banco de dados, ou em ambos.

Teoricamente, você pode analisar o bytecode ou o código-fonte da aplicação e coletar todas as instruções SQL armazenadas como strings. Isso pode, é claro, ser problemático para aplicações com um grande número de instruções, ou que montam instruções SQL dinamicamente. Você também pode usar o próprio banco de dados para obter um log de instruções SQL. Embora essa seja definitivamente uma opção mais adequada do que examinar o código da aplicação, ela requer privilégios administrativos no banco de dados; além disso, pode ser difícil se várias aplicações estiverem compartilhando o mesmo banco de dados.

A API JDBC fornece um método para gravar um log das operações de banco de dados no nível do driver, com o método `setLogWriter` de `DriverManager`. Ele gera informações como o registro de drivers, URLs utilizados para criar conexões a bancos de dados e nomes de classes de conexão. O problema com o logging do `DriverManager` é que ele não fornece as informações mais importantes, como as instruções e valores SQL passados para o banco de dados. Redirecionei o logging do driver JDBC para `System.out` e fiz testes com um programa sim-

bles, que registra um driver, obtém uma conexão ao banco de dados e executa algumas instruções SELECT com parâmetros. Examinando-se a saída do logging do driver na Listagem 13.1, vemos que nenhum rastreamento das instruções SELECT executadas é apresentado.

#### **LISTAGEM 13.1** Saída de logging do driver JDBC

```
DriverManager.initialize: jdbc.drivers = null
JDBC DriverManager initialized
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.deregisterDriver: com.sybase.jdbc2.jdbc.SybDriver@1d4c61c
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.getConnection("jdbc:sybase:Tds:helunx142:2075/pslwrkdb1")
    trying driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
getConnection returning driver[className=
com.sybase.jdbc2.jdbc.SybDriver...]
```

Para fornecer uma maneira confiável de interceptar chamadas ao banco de dados, você deve substituir o driver JDBC por um *wrapper* que faça o logging de instruções e então delegue para o driver “real”. Há uma máxima antiga que afirma que uma boa tecnologia ou produto torna “fáceis coisas simples, e possíveis coisas complexas”. O P6Spy, que utilizaremos para logging de JDBC, demonstra exatamente isso. Em menos de 10 minutos e com alterações mínimas de configuração, ele permite o registro de chamadas ao banco de dados a partir de aplicações existentes, sem nenhuma alteração no código. O download do P6Spy, uma aplicação livre, pode ser feito do SourceForge.

Depois de baixar e instalar o P6Spy, os arquivos p6spy.jar e spy.properties devem ser colocados no CLASSPATH da aplicação-alvo. Para ativá-lo, a classe de driver real utilizada pela aplicação deve ser substituída pela classe de driver do espião. O nome do driver real deve ser configurado em spy.properties, de modo que o espião possa delegar para ele. O restante da configuração relacionada ao banco de dados para a aplicação não precisa de alterações. Por exemplo, se a aplicação conectar-se a um banco de dados Oracle utilizando o driver padrão, o nome da classe do driver real será oracle.jdbc.driver.OracleDriver. Para ativar o espião, esse nome deve ser substituído por com.p6spy.engine.spy.P6SpyDriver no arquivo de configuração da aplicação; em spy.properties, o driver real deve ser configurado como a seguir:

```
realdriver=oracle.jdbc.driver.OracleDriver
```

Reiniciar a aplicação produz um log que contém as chamadas ao banco de dados interceptadas. Por exemplo, depois de configurar o P6Spy com o TableExample, da aplicação demo padrão do JDK, e digitar algumas instruções SELECT na tabela, obtive o log a seguir:

```
1066073757578|16|0|statement||SELECT * FROM TAB
1066073780718|0|0|statement||SELECT * FROM TAB where tname='Alex'
```

O P6Spy utiliza o Log4J e é bastante personalizável. Consulte a documentação do produto para mais detalhes.

## HISTÓRIAS DAS TRINCHEIRAS

No Riggs Bank, começamos a utilizar o Wily Introscope para monitorar o desempenho de um cluster de servidores. O Introscope é uma aplicação em Java que coleta métricas de desempenho em tempo de execução e permite armazená-las em um banco relacional, para análise posterior. Embora baseado em boas idéias, faltava refinamento ao Introscope e, com a quantidade de dados de desempenho que tínhamos, a ferramenta não estava funcionando com os dados históricos. As tabelas utilizadas pelo Introscope cresciam para milhões de registros; suspeitávamos que a causa do problema era o projeto de banco de dados e instruções SQL ineficientes. O suporte técnico tinha poucas sugestões, a não ser “Mantenha menos dados ou atualize a máquina”. Para ressuscitar o produto, decidimos espionar o código SQL utilizado para executar as consultas. Depois de conectar e analisar as instruções SQL, pudemos adicionar índices a tabelas, otimizar o projeto do banco de dados e reconfigurar a persistência dentro do produto. Isso decuplicou o desempenho das consultas, e foi uma grande vitória para a equipe de desenvolvimento.

## Questionário rápido

1. Quais são as abordagens gerais para fazer eavesdropping?
2. Por que é fácil espionar a comunicação HTTP?
3. Como a comunicação HTTP pode ser protegida?
4. O que torna o sniffing de rede possível?
5. Como se pode espionar a comunicação RMI?
6. Qual é a maneira mais confiável de interceptar instruções SQL em JDBC?

## Resumo

- *Eavesdropping* consiste em interceptar e registrar a troca de mensagens entre cliente e servidor. É possível porque a comunicação tem de atravessar os limites do processo ou da máquina.
- *O tunelamento* é uma técnica baseada na colocação de um intermediário entre um cliente e o servidor, e na reconfiguração do cliente para enviar as mensagens a esse intermediário. O intermediário despacha as solicitações para o servidor e encaminha a resposta de volta ao cliente, fazendo o logging da comunicação no processo.
- O eavesdropping em HTTP é fácil devido à sua natureza textual, e porque há uma ampla disponibilidade de ferramentas.
- O sniffing de rede é possível porque os pacotes de protocolo de rede precisam tráfegar entre os nós. Um nó de rede promíscuo opta por receber todos os pacotes, independentemente do endereço de destino. Isso permite a sniffers interceptarem mensagens que atravessam a rede.
- Fazer eavesdropping em RMI é possível com um sniffer de rede como o Ethereal. Isso requer análise de pacotes de rede de nível mais baixo, os quais transportam o conteúdo binário das mensagens RMI.
- O eavesdropping em JDBC pode ser feito facilmente colocando wrappers em volta dos drivers reais e dos objetos de conexão. Os wrappers registram todas as instruções antes de encaminhá-las às classes dos drivers reais.

# Controlando o carregamento de classes

*“Ninguém nota quando as coisas vão bem.”*

Lei das reclamações de Zimmerman

Os *class loaders* (“carregadores de classe”) carregam e inicializam classes e interfaces na JVM. Este capítulo fornece uma visão geral desse processo e mostra como desenvolver um *class loader* personalizado, que permite a decoração do bytecode carregado.

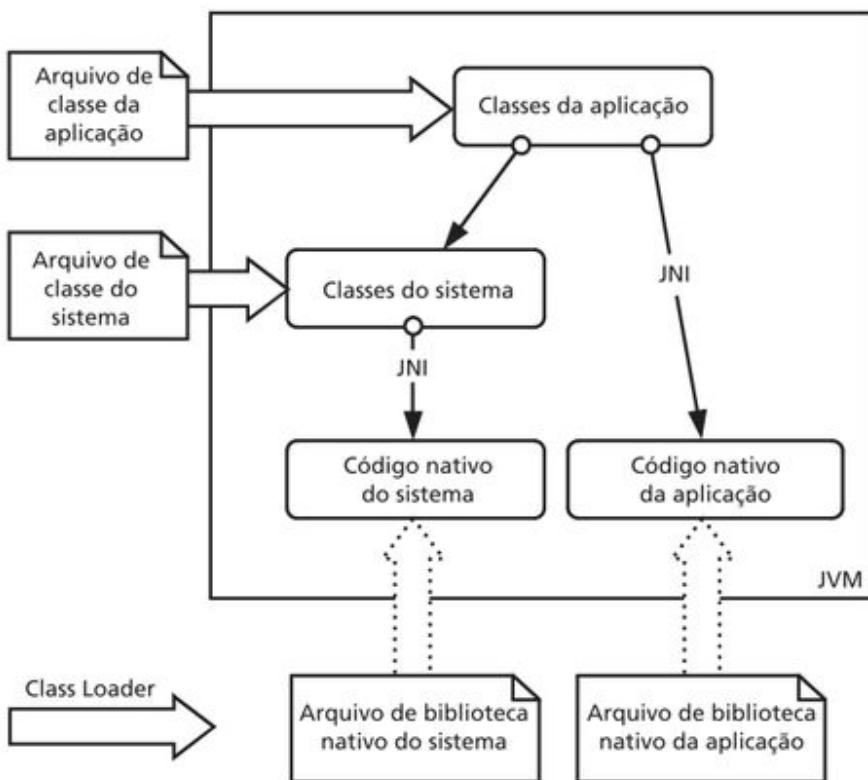
## Funcionamento interno da JVM do ponto de vista do carregamento de classes

Executada na parte superior do sistema operacional (SO), a máquina virtual Java fornece um nível de abstração para as aplicações escritas em Java. A especificação da linguagem e APIs padronizadas permitem o desenvolvimento e a implantação em diversas plataformas, porque as aplicações não fazem chamadas dependentes de plataforma diretamente ao SO. As APIs básicas de Java são, na sua maioria, elas próprias escritas em Java, incluindo apenas uma pequena parte de código nativo, acessado via JNI. Aplicações Java podem conter bytecode e bibliotecas dinâmicas nativas. As bibliotecas nativas são distribuídas como bibliotecas de vínculo dinâmico (.dll) para Windows e bibliotecas compartilhadas (.so) para Unix. Para executar uma aplicação Java, a JVM utiliza um *class loader* para carregar e inicializar o sistema e as classes de aplicações, o que podem resultar no carregamento de bibliotecas nativas. A JVM, da perspectiva de carregamento de classes, é mostrada na Figura 14.1.

# 14

## NESTE CAPÍTULO

- ▶ Funcionamento interno da JVM do ponto de vista do carregamento de classes 125
- ▶ Escrevendo um class loader personalizado 129
- ▶ Questionário rápido 133
- ▶ Resumo 133



**FIGURA 14.1** JVM a partir da perspectiva de carregamento de classes.

Embora uma aplicação típica não precise se preocupar com o carregamento e a inicialização, a capacidade de controlar esse processo é explorada por técnicas como instrumentação e proteção de integridade de bytecode.

O processo de carregamento de aplicações se inicia com a classe inicial fornecida ao lançador Java (tipicamente passada como parâmetro de linha de comando para o runtime Java). Todas as classes pai e quaisquer classes referenciadas pelo método `main()` da classe inicial são carregadas com *lazy loading* e inicializados à medida que são feitas referências elas. A menos que explicitamente especificado, o class loader para a classe referente é utilizado para carregar a classe referenciada. Se class loaders Java carregam classes, o que seria utilizado para carregar class loaders? A resposta é o class loader de inicialização (*bootstrap class loader*), que é implementado com código nativo e utilizado para carregar as classes básicas de Java como `java.lang.Object` e `java.lang.ClassLoader`. O class loader de extensões é utilizado para carregar as bibliotecas de extensão, em geral a partir do diretório `lib/ext` do JRE; arquivos JAR colocados nesse diretório ficam automaticamente disponíveis para aplicações Java. O class loader de *aplicação* é criado internamente pelo lançador da JVM para carregar classes do `CLASSPATH` padrão. Por fim, o class loader de *sistema* é geralmente o mesmo que o de aplicação.

Os class loaders são organizados em uma cadeia, na qual um filho deixa o pai localizar a classe antes de tentar ele mesmo localizá-la. Geralmente, um class loader verifica primeiro se a classe já foi carregada e inicializada. Se a classe ainda não tiver sido carregada, ele tenta criá-la ou carregá-la e, se for localizada, inicializá-la na JVM. O class loader de

inicialização é a raiz da hierarquia de class loaders e corresponde ao valor `null`; já o class loader de sistema é o pai da ordem de delegação-padrão para novos class loaders. A Tabela 14.1 lista os class loaders e as informações sobre as origens de seus dados.

**TABELA 14.1****Class loaders e as origens das suas classes**

CLASS LOADERS	ORIGEM DAS CLASSES
Inicialização (bootstrap)	Diretórios e arquivos JAR listados na propriedade de sistema <code>sun.boot.class.path</code> , que por padrão inclui as classes básicas de runtime no <code>rt.jar</code> e alguns outros JARs padrão. Ele pode ser manipulado passando <code>-Xbootclasspath</code> para o launcher de Java.
Extensões	Diretórios listados na propriedade de sistema <code>java.ext.dirs</code> , que por padrão aponta para o diretório <code>lib/ext</code> do JRE. Entretanto, pode ser especificado explicitamente na linha de comando com <code>-Djava.ext.dirs=&lt;path&gt;</code> .
Aplicação	Diretórios e JARs listados na propriedade de sistema <code>java.class.path</code> , por padrão configurada a partir da variável de ambiente <code>CLASSPATH</code> .

Para examinar a hierarquia de class loaders em tempo de execução, escreveremos uma classe simples que imprime o class loader utilizado para carregá-la, além de sua cadeia de “pais”. O código da classe é mostrado na Listagem 14.1.

**LISTAGEM 14.1** Código-fonte de PrintClassLoaders

```
public class PrintClassLoaders {

    public static void main(String[ ] args) {
        System.out.println("System class loader = " +
            ClassLoader.getSystemClassLoader( ));
        System.out.println("Thread context class loader = " +
            Thread.currentThread( ).getContextClassLoader( ));

        System.out.println("Class loader hierarchy for this class:");
        String padding = "";
        ClassLoader cl = PrintClassLoaders.class.getClassLoader( );
        while (cl != null) {
            System.out.println(padding + cl);
            cl = cl.getParent( );
            padding += "    ";
        }
        System.out.println(padding + "null (bootstrap class loader)");
    }
}
```

A classe PrintClassLoaders está localizada no pacote `comvertjava.classloader`. Executar essa classe produz a seguinte saída:

```
System class loader = sun.misc.Launcher$AppClassLoader@12f6684
Thread context class loader = sun.misc.Launcher$AppClassLoader@12f6684
Class loader hierarchy for this class:
sun.misc.Launcher$AppClassLoader@12f6684
    sun.misc.Launcher$ExtClassLoader@f38798
        null (bootstrap class loader)
```

Podemos concluir a partir da saída que a hierarquia de class loaders corresponde à ordem em que o runtime tenta localizar e carregar uma classe. A instância AppClassLoader retornada pela chamada a `getClassLoader()` a partir da classe inicial é o class loader associado a PrintClassLoaders. A mesma instância de class loader é utilizada como o class loader de sistema retornado por `ClassLoader.getSystemClassLoader()`; seu pai imediato é o class loader de extensões, ExtClassLoader. O class loader de extensões tem `null` como pai, que representa o class loader de inicialização. Pelo fato de os class loaders delegarem o carregamento a seu pai, antes de eles próprios tentarem carregar classes, as classes localizadas no classpath de inicialização (*boot classpath*) sempre são carregadas pelo carregador de inicialização. Se a classe não for localizada no classpath de inicialização, o classpath de extensões é verificado em seguida. Somente se uma classe não for localizada nestes dois classpaths, será verificado o classpath de aplicação. Se a classe também não for localizada pelo class loader de aplicação, é lançada a famosa `ClassNotFoundException`.

Vários pontos importantes sobre o carregamento de classes precisam ser bem entendidos:

- O class loader que carregou e definiu um processo de classe fica associado a ele e é referenciado como *class loader definidor*.
- A classe carregada é identificada não só pelo seu nome, mas pelo par composto de seu nome e do class loader definidor. Duas classes que compartilham o mesmo nome, mas que tenham diferentes class loader, são consideradas diferentes.
- Se a classe A faz referência à classe B, que ainda não foi carregada, o class loader definidor de A será utilizado para carregar B.

Portanto, se uma classe é carregada a partir do classpath de inicialização pelo class loader de inicialização, ela não pode se referir a classes do classpath de aplicação, a menos que passe explicitamente pelo carregador de sistema ou por um carregador personalizado. Um thread tem um class loader associado, que pode ser obtido utilizando `getContextClassLoader()`. Em geral, ele é configurado como o carregador da classe que iniciou o thread. Às vezes, você precisa utilizar o class loader de contexto de threads, em vez do class loader definidor, para acessar classes de uma origem diferente. Esse pode ser o caso para servlets carregados utilizando um carregador personalizado, do container Web. O class loader do container talvez seja configurado para carregar somente classes da aplicação da Web, o que significa que classes no classpath de sistema ficam inacessíveis.

## Escrevendo um class loader personalizado

O Java fornece controle às aplicações sobre como classes são criadas ou carregadas, por meio de class loaders personalizados ou definidos pelo usuário. Por exemplo, os servidores Web e de aplicações fornecem, para cada aplicação instalada, seu próprio class loader; isso isola melhor as aplicações lógicas executadas na mesma JVM. Lembre que, mesmo que duas classes sejam carregadas a partir do mesmo arquivo de classes, elas são consideradas diferentes se forem definidas por class loaders diferentes. Tendo cada aplicação seu próprio class loader, as variáveis estáticas utilizadas tão freqüentemente para implementar padrões singleton e armazenar dados globais tornam-se variáveis estáticas em nível de aplicativo em vez de nível da JVM. Class loaders personalizados também permitem que servidores Web e servidores de aplicações recarreguem classes sem reiniciar a JVM. Outra técnica poderosa é utilizar um class loader personalizado para criar classes em tempo real, ou para alterar a estrutura binária de classes antes de defini-las no runtime de Java.

Nesta seção, vamos criar um class loader personalizado chamado `DecoratingClassLoader`, no pacote `covertjava.classloader`, que permite a modificação de dados binários das classes carregadas, em tempo real. Depois de codificada, essa classe pode ser utilizada para suportar arquivos de classes encriptados ou a decoração de bytecode, discutidos mais adiante. Por enquanto, vamos focar carregar classes a partir de um classpath definido pelo usuário, e invocar um método callback, que terá a oportunidade de examinar e modificar os dados binários. O código será inspirado na implementação de class loaders do JUnit. Todos os class loaders devem estender a classe abstrata `java.lang.ClassLoader` além de, no mínimo, redefinir o método `findClass`, para carregar ou criar a classe com um nome dado. A implementação do `DecoratingClassLoader` de `findClass` é mostrada na Listagem 14.2.

### HISTÓRIAS DAS TRINCHEIRAS

O WebCream converte aplicações Java em sites Web HTML, em tempo real. Um dos recursos essenciais desse produto é a capacidade de executar múltiplos clientes virtuais dentro da mesma JVM. O recurso de multithreading torna fácil a execução de várias instâncias de uma aplicação Java em uma JVM, mas as aplicações acabam compartilhando as classes. Isso leva a erros, porque inicializadores estáticos só são executados para o primeiro cliente e membros estáticos são comuns a todos os clientes. Para fornecer uma separação clara entre os clientes virtuais, o WebCream utiliza um class loader personalizado. Cada cliente virtual recebe sua própria instância de carregador, que carrega todas as classes de aplicação. Como a JVM utiliza o class loader junto com o nome de classe para identificar classes, cada cliente virtual recebe sua própria classe, além de dados estáticos.

**LISTAGEM 14.2** Implementação de `findClass`

```
protected Class findClass(String name) throws ClassNotFoundException {
    System.out.println("DecoratingClassLoader loading class " + name);
    byte[ ] bytes = getClassBytes(name);

    if (getDecorator( ) != null)
        bytes = getDecorator( ).decorateClass(name, bytes);

    return defineClass(name, bytes, 0, bytes.length);
}
```

---

O método `findClass` só é chamado se a classe não for localizada por nenhum class loader pai na hierarquia. Dessa maneira, o carregamento de classe básicas como `java.lang.Object` é feito pelo class loader de inicialização. Depois que os dados binários da classe são carregados, o decorador tem a oportunidade de incrementar o bytecode. Então, os bytes da classe são passados para o método `defineClass` de `java.lang.ClassLoader`, que converte a representação binária da classe em um objeto `Class` interno. A leitura de dados binários de classes é realizada no método `getClassBytes( )`, mostrado na Listagem 14.3.

**LISTAGEM 14.3** Implementação de `getClassBytes`

```
protected byte[ ] getClassBytes(String className)
    throws ClassNotFoundException
{
    byte[ ] bytes = null;
    for (int i = 0; i < classPathItems.size( ); i++) {
        String path = (String)classPathItems.get(i);
        String fileName = className.replace('.', '/') + ".class";
        if (isJar(path) == true) {
            bytes = loadJarBytes(path, fileName);
        }
        else {
            bytes = loadFileBytes(path, fileName);
        }
        if (bytes != null)
            break;
    }

    if (bytes == null)
        throw new ClassNotFoundException(className);

    return bytes;
}
```

---

O método itera pelos itens do classpath configurado tentando localizar e carregar os dados de um arquivo .class no diretório correspondente. Se os dados da classe não forem localizados, `ClassNotFoundException` é lançada. `DecoratingClassLoader` recebe o classpath como um parâmetro no seu construtor; o construtor padrão utiliza o valor da propriedade de sistema `decorate.class.path` para inicializar seu classpath. O código-fonte completo de `DecoratingClassLoader` está no diretório `CovertJava/src/covertjava/classloader`.

Para utilizar o class loader personalizado, este deve ser especificado explicitamente para o carregamento de uma classe que esteja inacessível aos class loaders pai. Isso pode ser feito passando-se uma instância do class loader para o método `Class.forName()`, ou chamando diretamente o método `loadClass()` do carregador personalizado. Lembre que class loaders delegam o carregamento à cadeia de pais; se um class loader pai puder localizar a classe, o método `findClass()` do carregador filho não será chamado. Isso significa que a maneira mais direta de permitir que um class loader personalizado carregue uma classe é assegurando que a classe não esteja localizada no classpath de sistema, de inicialização nem de extensão. Essa é uma escolha comum para implementações de servidores Web e servidores de aplicações, que sugerem que classes de aplicação não sejam colocadas no classpath de sistema. Se o controle total do carregamento de classes for necessário, o método `loadClass(none da string, booleano resolve)` pode ser redefinido. Esse método é chamado para iniciar o carregamento de classes; sua implementação padrão permite ao pai tentar carregar a classe primeiro. Redefinindo-se esse método, o class loader personalizado será chamado por todas as classes no sistema, independentemente de onde os dados de classes estejam localizados. Uma alternativa é utilizar o class loader de inicialização como um pai, passando `null` para o construtor de `ClassLoader`. Entretanto, remover do classpath de sistema os JARs e diretórios com classes que devem ser carregadas com um carregador personalizado é a melhor alternativa, porque leva à separação de classes personalizadas e de sistema.

Para testar a delegação do `DecoratingClassLoader`, adicionaremos um decorador chamado `PrintingClassDecorator`, que simplesmente imprime o nome da classe em que é invocado (ver Listagem 14.4).

#### **LISTAGEM 14.4** Implementação de `PrintingClassDecorator`

```
public class PrintingClassDecorator implements ClassDecorator {  
  
    public byte[ ] decorateClass(String name, byte[ ] bytes) {  
        System.out.println("Processed bytes for class " + name);  
        return bytes;  
    }  
}
```

Para ver nosso `DecoratingClassLoader` em ação, vamos experimentá-lo na classe `PrintClassLoaders` que examinamos anteriormente. Precisaremos de uma classe launcher que instancie nossos class loaders personalizados e então carregue a classe de teste que o utiliza. `DecoratingLauncher`, no pacote `covertjava.classloader`, utiliza o nome da classe de teste a

ser executada, e utiliza o DecoratingClassLoader para carregá-la. Então, ele invoca a função main( ) da classe de teste via a API de reflection. O método main( ) de DecoratingLauncher é mostrado na Listagem 14.5.

#### **LISTAGEM 14.5** Método DecoratingLauncher main( )

---

```
public static void main(String[ ] args) throws Exception {
    if (args.length < 1) {
        System.out.println("Missing command line parameter <main class>");
        System.out.println("Syntax: DecoratingLauncher " +
            "[-Ddecorate.class.path=<path>] <main class> [<arg1>, [<arg2>]...]");
        System.exit(1);
    }

    DecoratingClassLoader decoratingClassLoader = new DecoratingClassLoader( );
    decoratingClassLoader.setDecorator(new PrintingClassDecorator( ));
    Class mainClass = Class.forName(args[0], true, decoratingClassLoader);

    String[ ] mainArgs = new String[args.length - 1];
    System.arraycopy(args, 1, mainArgs, 0, args.length - 1);
    invokeMain(mainClass, mainArgs);
}
```

---

Note que estamos passando a instância de nosso carregador para o método forName( ). Antes de executar o teste, precisamos assegurar que a classe PrintClassLoaders tenha sido removida de CLASSPATH e colocada em um diretório que será apontado pela propriedade de sistema decorate.class.path. Teremos de modificar a versão de destino em nosso build.xml para mover covertjava.classloaders.PrintClassLoaders para o diretório CovertJava/lib/classes, antes de criarmos um JAR. Além disso, precisamos criar um novo arquivo batch no diretório bin, que invocará o launcher e passará PrintClassLoaders como um parâmetro. O conteúdo do arquivo batch é mostrado na Listagem 14.6.

#### **LISTAGEM 14.6** Código de classLoaderTest.bat

---

```
@echo off
rem Demonstration of using custom class loader on PrintClassLoaders class

call setEnv.bat
set JAVA_OPTS=-Ddecorate.class.path=..\lib\classes %JAVA_OPTS%

java %JAVA_OPTS% covertjava.classloader.DecoratingLauncher
covertjava.classloader.PrintClassLoaders
```

---

Executar `classLoaderTest.bat` produz a saída a seguir:

```
Processed bytes for class covertjava.classloader.PrintClassLoaders
System class loader = sun.misc.Launcher$AppClassLoader@12f6684
Thread context class loader = sun.misc.Launcher$AppClassLoader@12f6684
Class loader hierarchy for this class:
covertjava.classloader.DecoratingClassLoader@ad3ba4
    sun.misc.Launcher$AppClassLoader@12f6684
        sun.misc.Launcher$ExtClassLoader@f38798
        null (bootstrap class loader)
```

Analizando a saída, podemos ver que o class loader associado à classe `PrintClassLoaders` é uma instância de `DecoratingClassLoader`. `DecoratingClassLoader` obteve o class loader de sistema como pai; o restante da hierarquia é o mesmo.

## Questionário rápido

1. Qual é a função de um class loader?
2. O que é o class loader de inicialização (*bootstrap class loader*) e quais classes ele carrega?
3. O que é o class loader de extensões e quais classes ele carrega?
4. Quais classes são consideradas as mesmas dentro da JVM?
5. Para que um class loader personalizado pode ser utilizado?
6. Um class loader personalizado pode ser utilizado para carregar *todas* as classes (inclusive classes Java básicas)?
7. Por que `DecoratingClassLoader` utiliza seu próprio classpath?

## Resumo

- Class loaders carregam e inicializam classes e interfaces na JVM.
- O processo de carregamento de aplicações se inicia com a classe inicial passada para o *launcher* (lançador). Todas as classes pai e quaisquer classes referenciadas pelo método `main( )` da classe inicial são carregadas com *lazy loading* e inicializadas à medida que são feitas referências a elas.
- O class loader é implementado com código nativo e utilizado para carregar classes básicas de Java, como `java.lang.Object` e `java.lang.ClassLoader`.
- Os class loaders são organizados em uma cadeia, em que um filho permite que o pai localize uma classe, antes de ele próprio tentar localizá-la.
- Uma classe carregada é identificada não só por seu nome, mas pelo par composto do nome e do class loader definidor.
- Class loaders personalizados permitem que aplicações Java assumam o controle do carregamento de classes. Eles são utilizados para implementar o recarregamento de classes, a separação de aplicações lógicas dentro da mesma JVM, e a decoração ou a criação de bytecode em tempo real.

# 15

## *NESTE CAPÍTULO*

- ▶ Por que se incomodar? 134
- ▶ Aplicando patch a classes Java básicas utilizando o classpath de inicialização 135
- ▶ Exemplo do patching de `java.lang.Integer` 136
- ▶ Questionário rápido 138
- ▶ Resumo 138

# Substituindo e aplicando patches a classes Java básicas

*"Um caminho sem obstáculos provavelmente não leva a lugar algum."*

Defalque

## **Por que se incomodar?**

No Capítulo 5, conversamos sobre o patching de classes Java, para alterar ou estender sua lógica. As técnicas apresentadas neste capítulo funcionam para classes de aplicação e de bibliotecas, carregadas pelo class loader de sistema ou por um carregador personalizado. Entretanto, tentar aplicar as técnicas aplicar patches a classes básicas em um pacote cujo nome inicia com `java` não tem efeito, porque a versão original da classe continua a ser utilizada. No Capítulo 14, fornecemos uma discussão detalhada sobre como classes são carregadas, e podemos ver porque classes de sistema (básicas) exigem uma abordagem diferente. Lembre que classes de sistema são carregadas pelo class loader de inicialização (*bootstrap class loader*), que não utiliza a variável de ambiente `CLASSPATH`. Embora a abordagem geral para o patching de classes de sistema seja semelhante à do patching de classes de aplicação, há algumas diferenças sutis, que são o assunto deste capítulo.

Há realmente necessidade de aplicar patches a classes básicas? Na minha carreira, tive de aplicar patches a classes de aplicação com muito mais freqüência do que a classes básicas. Uma das razões talvez seja por que as classes básicas foram bem projetadas e já estão

maduras o suficiente para a atender à maioria dos desenvolvedores. Entretanto, de vez em quando, você pode se deparar com uma deficiência em uma classe básica, sem uma boa maneira de contornar o problema.

Definitivamente não é aconselhável aplicar patches a classes Java básicas como uma solução permanente. Isso tem consequências jurídicas (a licença do JDK proíbe modificações nessas classes) e pode requerer trabalho adicional para migrar para uma nova versão de JDK. Entretanto, a técnica fornece muito mais controle ao desenvolvedor. Ela pode ser utilizada para inserir rastreamentos no código do JDK e alterar temporariamente a implementação da lógica básica, para adaptá-la às necessidades das aplicações. Além disso, é um recurso fantástico, e estarmos munidos com essa técnica poderosa não nos fará mal. Só não deixe de ler o acordo de licença antes de ir por esse caminho.

## Aplicando patch a classes Java básicas utilizando o classpath de inicialização

Como mencionei, a abordagem para corrigir classes básicas é semelhante à utilizada para corrigir classes de aplicação. É necessário obter o arquivo-fonte da classe que exige o patching. O JDK é convenientemente distribuído com o código-fonte (obrigado, Sun!), então, a maior parte do tempo, você pode simplesmente obter o código de `src.jar`. Observe que algumas classes de sistema são distribuídas sem código-fonte; isso é verdade para as classes dentro do pacote `sun` e de outros pacotes não-públicos. Você pode descompilar os arquivos de classes, como descrito no Capítulo 2, embora o acordo de licença deva ser observado.

### HISTÓRIAS DAS TRINCHEIRAS

Trabalhei em um produto chamado WebCream, que é capaz de executar múltiplos clientes Swing virtuais dentro da mesma JVM. Durante o teste, foi observado que, depois de executar por algum tempo, a JVM travava e nenhum novo cliente conseguia mais ser inicializado. Uma análise de dumps de threads da JVM, como descrito no Capítulo 10, revelou que o travamento estava ocorrendo em uma chamada do método `sgetTreeLock()` de `java.awt.Component`. A implementação de `getTreeLock()` simplesmente retorna uma variável, declarada em `Component` da seguinte maneira:

```
static final Object LOCK = new AWTTreeLock();
```

Portanto, o AWT utiliza um lock global, que é compartilhado por todos os componentes; se um thread não conseguir liberar o monitor do lock num tempo hábil, nenhum outro thread poderá realizar operações AWT e Swing. Isso foi feito por projetistas do Java para evitar problemas de concorrência ao refazer layouts, mas ele acaba com a escalabilidade de um produto como o WebCream. Uma solução imediata naquela época era aplicar um patch à classe `java.awt.Component` para que utilizasse um lock específico ao cliente virtual, em vez de um lock global. Com o patch aplicado, o bloqueio de clientes virtuais deixou de acontecer.

Depois obter o código-fonte, você pode inserir a nova lógica. Compile a classe exatamente como compilaria qualquer outra e certifique-se de não adicionar bugs. Agora que você tem uma nova versão do bytecode, resta instruir a JVM sobre como utilizá-la, em vez do bytecode original. Isso pode ser feito manipulando o classpath de inicialização, como explicado no Capítulo 14. O class loader de inicialização utiliza o classpath de inicialização para localizar as classes básicas. Por padrão, ele é configurado para incluir somente rt.jar e possivelmente algumas outras bibliotecas de sistema. O arquivo rt.jar, localizado em JRE\_HOME\lib, contém a maioria das classes básicas, então, se não houver nenhum código-fonte para uma classe e você quiser localizar seu bytecode, verifique primeiro em rt.jar. O classpath de inicialização pode ser definido passando o parâmetro -Xbootclasspath para a linha de comando do launcher de Java. Executar java -X exibe as seguintes informações de ajuda:

```
C:\CovertJava\java -X
-Xbootclasspath:<directories and zip/jar files separated by ;>
    set search path for bootstrap classes and resources
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
    append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
    prepend in front of bootstrap class path
...
...
```

Utilizando o parâmetro de linha de comando, podemos configurar ou estender o classpath de inicialização. Como estamos interessados em substituir uma classe existente, utilizamos -Xbootclasspath/p: para inserir o diretório que contém os patches na frente do caminho padrão. Executar a JVM com esse parâmetro resulta na utilização da classe com patch em vez da original.

## Exemplo do patching de java.lang.Integer

Para colocar a teoria na prática, vamos escrever um patch simples para java.lang.Integer. Por motivos desconhecidos à comunidade Java, o objeto Integer é imutável. Depois que o valor está configurado, ele não pode ser alterado. A idéia era provavelmente fazer objetos Integer comportarem-se como objetos String, porque, se for preciso alterar um valor representado por um Integer, você deveria criar uma nova instância e utilizá-la, em vez de usar o antigo valor. O problema com essa abordagem é que ela resulta no uso ineficiente de memória para aplicações que precisam de coleções dinâmicas de inteiros. O Java não fornece classes de coleção para tipos primitivos, então a única maneira de obter um array dinâmico de inteiros é utilizar um java.util.Array de instâncias de Integer. Se o valor do inteiro armazenado precisar mudar, você deve criar uma nova instância de Integer e coloca-la no array em que o antigo valor estava. Naturalmente, as alocações e coletas de lixo subsequentes produzem um overhead considerável. Uma abordagem muito melhor é alterar o valor interno do objeto Integer. Entretanto, como java.lang.Integer é imutável, a única maneira legítima de contornar o problema é criar e utilizar sua própria classe que simula o Integer e fornecer a ela um método setValue( ).

Contudo, vamos aplicar um patch à classe `java.lang.Integer` existente e conceder a ela um método `setValue( )`. Faremos isso com interesse puramente acadêmico e para praticar o que recomendamos, porque não queremos violar o acordo de licença de Java. Examinar o código-fonte de `java.lang.Integer` revela que o valor do objeto está armazenado em um campo privado, `value`. Assim, devemos copiar o arquivo fonte para o diretório `CovertJava\src\java\lang` e inserir um método chamado `setValue` (veja a Listagem 15.1).

#### **LISTAGEM 15.1** O código-fonte do método `setValue( )`

---

```
public void setValue(int value)
    this.value = value;
}
```

---

O próximo passo é criar uma classe de teste, `CorePatchTest`, que acessa o método `setValue( )` recentemente inserido. O código da classe de teste é mostrado na Listagem 15.2.

#### **LISTAGEM 15.2** Utilizando `java.lang.Integer` corrigida com patch

---

```
package covertjava.patching;

public class CoreClassTest {
    public static void main(String[ ] args) {
        Integer i = new Integer(10);
        System.out.println("Old value = " + i);
        i.setValue(100);
        System.out.println("New value = " + i);
    }
}
```

---

Compilar as classes que utilizam as versões corrigidas com patch das classes básicas pode ser um pouco trabalhoso se a interface pública da classe básica tiver sido alterada. Tentar executar o `javac` na nossa classe de teste resulta em um erro, porque a implementação de `Integer` no JDK não inclui `setValue( )`. Por isso, não podemos utilizar o Ant para compilar o `java.lang.Integer` corrigido com patch. A maneira mais fácil de contornar o problema é compilar nosso `Integer` corrigido com patch utilizando manualmente o `javac`, e então copiar o arquivo de classe para o diretório `CovertJava/distrib/patches`. Agora, podemos configurar o compilador para utilizar a versão de `Integer` corrigida no nosso projeto. Fazemos isso colocando a classe corrigida com patch no classpath de inicialização, antes da versão original. O `javac` aceita o parâmetro `-bootclasspath`, que permite redefinir o classpath de inicialização padrão, da mesma maneira que é feito na tarefa `javac` do Ant. Entretanto, se tentarmos redefinir o classpath de inicialização do `javac`, devemos especificar a localização de `rt.jar` e de todas as outras bibliotecas de sistema. Isso torna os scripts de build dependentes do diretório de instalação do JDK, ou de variáveis de

ambiente. Uma maneira mais simples é passar `-Xbootclasspath/p:` para a JVM que executa o Ant, para que, em vez de redefinir o caminho padrão, adicionemos apenas um item na frente dele. O script `ant.bat` utiliza a variável de ambiente `ANT_OPTS` para passar opções de linha de comando ao Java. Vamos tirar proveito disso, adicionando a seguinte linha a `CovertJava\bin\build.bat`:

```
ANT_OPTS=-Xbootclasspath/p:..\distrib\patches
```

Agora podemos utilizar o Ant para construir (*build*) o projeto e das bibliotecas de distribuição. Nossa tarefa final depois de construir o projeto é criar um arquivo de lote chamado `corePatchTest.bat` no diretório `CovertJava\bin` que executa `CorePatchTest`. Mais uma vez, para garantir que seja utilizada a versão corrigida com patch de `Integer`, passamos o parâmetro `-Xbootclasspath` para `java`. O código-fonte relevante de `corePatchTest.bat` é mostrado na Listagem 15.3.

#### **LISTAGEM 15.3** Testando o patch de uma classe básica

---

```
set JAVA_OPTS=-Xbootclasspath/p:..\distrib\patches  
java %JAVA_OPTS% covertjava.patching.CoreClassTest
```

---

`corePatchTest.bat` produz a seguinte saída:

```
Old value: 10  
New value: 100
```

Pronto! Adicionamos mais uma técnica à nossa caixa de truques.

## Questionário rápido

1. Você pode imaginar um caso em que gostaria de aplicar um patch a uma classe básica?
2. O quanto o processo de patching de classes básicas é diferente do de corrigir classes de aplicação?
3. Por que precisamos alterar o classpath de inicialização?

## Resumo

- O patching de classes básicas Java pode ajudar a depurar e entender o funcionamento da JVM.
- Classes básicas são sempre carregadas pelo class loader de inicialização, que utiliza o classpath de inicialização para localizar o bytecode.
- Para aplicar um patch a uma classe básica, a nova versão deve ser colocada no classpath de inicialização, antes da versão antiga.
- Para que seja possível compilar uma classe corrigida com um patch que utiliza a versão corrigida com um patch da classe com sua interface pública alterada, você deve especificar a versão corrigida no classpath de inicialização do compilador Java.

# 16

## NESTE CAPÍTULO

- ▶ Definição de fluxo de controle 139
- ▶ Interceptando erros de sistema 140
- ▶ Interceptando fluxos de sistema 140
- ▶ Interceptando uma chamada a `System.exit` 142
- ▶ Reagindo a uma desativação da JVM utilizando *hooks* 144
- ▶ Interceptando métodos com um proxy dinâmico 144
- ▶ A interface do profiler da máquina virtual Java 147
- ▶ Questionário rápido 148
- ▶ Resumo 148

# Interceptando o fluxo de controle

*"Nada é tão simples ao ponto de não poder ser mal-entendido."*

Lei de Freeman

## Definição de fluxo de controle

O *fluxo de controle* é uma sequência de execuções de métodos e instruções por um thread. A JVM executa instruções de bytecode Java na ordem em que são encontradas no arquivo de classe. O fluxo de controle pode ser programado utilizando instruções condicionais como `if`, `else` e `for`, ou invocando-se um método. Interceptar o fluxo de controle inclui o conhecimento da instrução ou do método que está executando, e a capacidade de alterar o fluxo em tempo de execução. Por exemplo, você poderia querer interceptar uma chamada a `System.exit()` para impedir a JVM de se desligar. Antes que você fique animado(a) demais com as possibilidades, deixe-me limitar as expectativas. Não existe uma maneira direta de interceptar qualquer instrução ou chamada de método em uma JVM em execução – a menos que a JVM tenha sido iniciada em modo de profiling. A execução de métodos é feita pelo JIT; não existe uma API de Java padrão que possa ser utilizada para adicionar um listener ou um *hook* (“gancho”) para as chamadas de métodos. Entretanto, veremos várias abordagens indiretas para interceptar o controle em cenários comuns. Também examinaremos a interface de profiler da JVM, que pode ser usada para interceptar qualquer chamada no modo de depuração.

## Interceptando erros de sistema

Erros de sistema são informados pela JVM quando ocorrem condições anormais, que estão presumivelmente além do controle das aplicações. Eles são lançados como subclasses de `java.lang.Error`, portanto, não são declarados; isso significa que podem ser lançados por qualquer método, mesmo que a assinatura do método não declare esses erros explicitamente. Erros de sistema incluem erros da máquina virtual, como `OutOfMemoryError` e `StackOverflowError`, erros de linkagem, como `ClassFormatError` e `NoSuchMethodError`, além de outras falhas. Convencionalmente, os programadores de aplicações só devem capturar instâncias de `java.lang.Exception`, o que significa que uma condição do tipo `OutOfMemoryError` não é detectada pela lógica de tratamento de erros das aplicações. Para a maioria das aplicações do dia-a-dia, isso não é desejável porque, mesmo que nada possa ser feito quando ocorrer um erro, a aplicação deve geralmente registrá-lo em um arquivo de log e tentar liberar recursos alocados. Uma boa solução é ter um bloco `try-catch` no topo da pilha de chamadas nos threads principais das aplicações, capturando `java.lang.Error` ou `java.lang.Throwable` e delegando a um método que analisa a condição de erro, registra-a e procurar realizar um desligamento limpo. Aqui está um exemplo:

```
public static void main(String[ ] args) {
    try {
        // Executa lógica da aplicação
        runApplication( );
    }
    catch (Throwable x) {
        // Registra erro em log e tenta um desligamento limpo
        onFatalError(x);
    }
}
```

Nas situações em que a JVM estiver sem memória, ou em que uma classe não for localizada, a aplicação pode tentar restabelecer a JVM, liberando o conteúdo de caches ou desativando recursos afetados pelas classes ausentes. Qualquer coisa é melhor que um encerramento repentino sem deixar rastros.

## Interceptando fluxos de sistema

Antes de o logging ter se tornado um requisito obrigatório em aplicações Java, era comum utilizar `System.out.println` para gerar a saída de rastreamentos de depuração. As desvantagens dessa abordagem são muitas e óbvias. Uma vez escritos, tais rastreamentos não podem ser ativados ou desativados sem alterar o código. Mesmo que o stream de saída da aplicação possa ser redirecionado para um arquivo, não há *rollover* e, como o arquivo é mantido aberto, ele não pode ser excluído até a aplicação ser desligada (por isso o tamanho aumenta tanto). Ao lidar com o código Java legado, cheio de chamadas a `System.out.println( )`, um problema comum é convertê-las em chamadas para um framework de logging (veja o Capítulo 6, para uma discussão sobre logging e rastreamento).

Também é importante capturar o stream de erros padrão, que recebe a saída de métodos como `printStackTrace()` de `Exception`. Uma das boas soluções para isso é interceptar a saída para `System.out` e `System.err`, e enviá-la para o arquivo de log. A técnica utiliza o fato de que o stream de saída de sistema pode ser redirecionado para um `PrintStream` personalizado, utilizando-se o método `setOut` de `java.lang.System`. `PrintStream` é uma classe decoradora em torno de uma instância de `OutputStream`, que é responsável pela saída real. Precisamos, portanto, desenvolver um `OutputStream` de redirecionamento, que escreve em um arquivo de log em vez de gravar na saída padrão do processo, e então atribuir o `System.out` a ele.

Vamos desenvolver uma classe chamada `LogOutputStream`, que estende `java.io.OutputStream` e grava sua saída em um arquivo de log utilizando o Apache Log4J. O framework de entrada e saída de Java é muito bem-projetado, e todos os métodos de `OutputStream` no final delegam para um único método – `write()` –, que aceita um parâmetro inteiro. `LogOutputStream` utiliza um `StringBuffer` para acumular caracteres, obtidos com o método `write(int)`; quando um separador de linha é detectado, o buffer inteiro é gravado no disco utilizando o Log4J. A única parte difícil da implementação é detectar o final de uma linha. Como você sabe, no Unix o final de linha é marcado por um único caractere: `\n` (nova linha). No Windows, o final de linha é indicado por uma combinação de dois caracteres: `\r` e `\n` (retorno de carro e nova linha). Para escrever código verdadeiramente multiplataforma em Java, você deve usar uma propriedade de sistema chamada `line.separator`. Como essa propriedade é uma `String`, a implementação deve fazer uma pesquisa de `substring`, em vez de uma comparação de caractere. Nossa implementação é otimizada para utilizar primeiro a comparação de caractere para verificar o *possível* final de uma linha, e então utilizar uma pesquisa de `substring` para garantir que o que encontramos é realmente um final de linha. O método `write()` é mostrado na Listagem 16.1.

#### **LISTAGEM 16.1** Método `write()` de `LogOutputStream`

```
public void write(int b) throws IOException {
    char ch = (char)b;
    this.buffer.append(ch);
    if (ch == this.lineSeparatorEnd) {
        // Verifica, char por char, visando velocidade
        String s = buffer.toString();
        if (s.indexOf(lineSeparator) != -1) {
            // A string de separadores inteira é gravada
            logger.info(s.substring(0, s.length() - lineSeparator.length()));
            buffer.setLength(0);
        }
    }
}
```

O logger aqui é uma referência para uma variável estática de tipo `org.apache.log4j.Logger`, declarada em `LogOutputStream`, assim:

```
static Logger logger = Logger.getLogger(LogOutputStream.class.getName());
```

Portanto, toda a saída para `System.out` é redirecionada para o framework Log4J, na forma de mensagens com nível *INFO-level*, da classe `LogOutputStream`. Para ver nossa classe em ação, temos de configurar um "anexador de arquivos" (*file appender*) em `log4j.properties` e instalar o interceptador (*interceptor*), como mostrado na Listagem 16.2.

#### **LISTAGEM 16.2** Instalando a interceptação de `System.out`

---

```
public static void main(String[ ] args) {  
    System.out.println("Installing the interceptor...");  
    PrintStream out = new PrintStream(new LogOutputStream( ), true);  
    System.setOut(out);  
    System.out.println("Hello, world");  
    System.out.println("Done");  
}
```

---

Executar o método `main( )` de `LogOutputStream` exibe uma mensagem `Installing the interceptor...` no console, mas grava as mensagens `Hello, world` e `Done` no arquivo de log. O mesmo interceptador pode ser instalado para o stream `System.err`. Por flexibilidade, ele pode ser parametrizado a fim de receber o nível de logging e o nome do stream no construtor. De modo semelhante, `System.In stream` pode ser configurado programaticamente com `System.setIn( )` para alimentar uma entrada desejada para uma aplicação.

## **Interceptando uma chamada a `System.exit`**

O processo da JVM normalmente termina quando não houver mais threads ativos. Threads executados como *daemons* (`Thread.isDaemon( ) == true`) não impedem a JVM de ser desligada. Em aplicativos com múltiplos threads, que incluem interfaces gráficas Swing e servidores RMI, não é fácil conseguir um desligamento limpo, que permita que todos os threads sejam encerrados suavemente. Freqüentemente, uma chamada para `System.exit( )` é feita forçosamente, sem desativar a JVM e terminar o processo. Usar `System.exit( )` tornou-se uma prática comum, mesmo em programas sem muita sofisticação; embora isso facilite a vida do desenvolvedor de aplicações, pode criar problemas em produtos *middle-tier*, como servidores Web e servidores de aplicações. Uma chamada a `System.exit( )` por uma aplicação Web, por exemplo, pode derrubar o processo do servidor Web e possivelmente impedir que usuários acessem outras aplicações Web e mesmo páginas HTML estáticas. Essa não é a maneira certa de fazer amizade com os administradores de sistema, e todo bom desenvolvedor sabe o valor que tem um relacionamento saudável com essa equipe.

Esta seção mostra uma maneira simples de interceptar uma chamada a `System.exit( )` e evitar o desligamento da JVM. Essa técnica pode ser descoberta examinando-se o código-fonte do método `exit( )` em `java.lang.System`. A primeira coisa que o método faz é verificar se um gerenciador de segurança (*security manager*) está instalado. Se estiver, o

método verifica se o chamador tem permissão para fechar a JVM. Nossa tarefa, portanto, é instalar um gerenciador de segurança personalizado (ou modificar a política de segurança, se um gerenciador já estiver instalado); esse gerenciador impede a saída da JVM até que ela seja explicitamente permitida. A classe InterceptingSecurityManager localizada no pacote `covertjava.intercept`, estende a classe `SecurityManager` e redefine o método `isExitAllowed()` para controlar o desligamento da JVM. Ela utiliza um flag interno que pode ser configurado com o método `setExitAllowed()`, para determinar se é permitido à JVM desligar-se. Se a saída não for permitida, uma `SecurityException` não-verificada é lançada para alterar o fluxo de controle. O método `main()` mostrado na Listagem 16.3 mostra como instalar o gerenciador de segurança de interceptação, e como ele afeta o fluxo de execução.

### **LISTAGEM 16.3 Interceptando `System.exit()`**

---

```
public static void main(String[ ] args) {
    InterceptingSecurityManager secManager = new InterceptingSecurityManager( );
    System.setSecurityManager(secManager);
    try {
        System.out.println("Run some logic...");
        System.exit(1);
    }
    catch (Throwable x) {
        if (x instanceof SecurityException)
            System.out.println("Intercepted System.exit( )");
        else
            x.printStackTrace( );
    }
    System.out.println("Run more logic...");
    secManager.setExitAllowed(true);
    System.out.println("Finished");
}
```

---

Para manter o exemplo simples, a lógica de negócio real, que normalmente seria invocada dentro do bloco `try`, foi substituída por uma mensagem `Run some logic...`. A chave é capturar a classe `Throwable` em vez da `Exception` normal, porque a chamada a `System.exit()` interceptada é informada como uma exceção não-verificada. Executar o método `main()` mostrado na Listagem 16.3 produz a seguinte saída:

```
Run some logic...
Intercepted System.exit( )
Run more logic...
Finished
Process terminated with exit code 0
```

Em vez de encerrar a JVM depois de uma chamada para `System.exit()` dentro do bloco `try`, o programa continua a executar até que a saída seja permitida.

## Reagindo a uma desativação da JVM utilizando *hooks*

A seção anterior mostrou como interceptar uma tentativa programada de desligar a JVM chamando `System.exit( )`. Às vezes, o desligamento da JVM é iniciado por um usuário via um comando `kill` no Unix ou um sinal `Ctrl+C` no Windows. A JVM também pode ser desligada, porque o usuário está efetuando logoff ou o SO está sendo desligado. Um programa Java pode interceptar o sinal de desligamento? Ele não pode interceptar esse sinal, mas pode reagir a ele. Desde o JDK 1.3, uma aplicação pode instalar um *hook* (gancho) de desligamento utilizando o método `addShutdownHook( )` de `java.lang.Runtime`. *Ganchos de desligamento (shutdown hooks)* são instâncias de `java.lang.Thread`, que são inicializadas mas não executadas. Quando a JVM estiver sendo desligada, todos os threads de ganchos de desligamento são iniciados para executar concorrentemente com os outros threads na JVM. Esses ganchos têm acesso à API Java inteira, mas devem levar em conta o delicado estado da JVM. Os threads de gancho não devem realizar operações demoradas e devem ser *thread-safe*. Não se deve ter expectativas quanto à disponibilidade de serviços de sistema, porque eles mesmos poderiam estar no seu processo de desligamento. Uma boa utilização para um gancho de desligamento é gravar uma entrada em um arquivo de log antes de fechá-lo e liberar todos os outros recursos, como conexões a banco de dados e arquivos abertos. Um exemplo de instalação de um gancho de desligamento é mostrado na Listagem 16.4.

### **LISTAGEM 16.4** Instalando um gancho de desligamento

---

```
public static void main(String[ ] args) {
    Runtime.getRuntime( ).addShutdownHook(new Thread( ) {
        public void run( ) {
            handleJVMShutdown( );
        }
    });
}

public static void handleJVMShutdown( ) {
    // Registrar desligamento e fechar todos os recursos
```

---

## Interceptando métodos com um proxy dinâmico

Às vezes você precisa fazer pré-processamento e pós-processamento de uma chamada de método. Isso pode incluir: rastrear o nome do método e os valores de seus parâmetros, medir o tempo de execução, ou mesmo fornecer uma implementação alternativa. Suponha que você está desenvolvendo um editor de desenhos que utiliza interfaces como `Line`, `Circle`, `Rectangle` e `Curve` para representar as formas básicas. Para adicionar rastrea-

mento a todos os métodos nessas interfaces, você tem várias opções. Você pode e inserir meticulosamente chamadas de rastreamento em cada método ou pode codificar uma classe proxy, implementando cada interface que imprime o rastreamento e então delega à implementação original. Essa é uma abordagem mais limpa, porque mantém o código de depuração separado da implementação, mas requer uma grande quantidade de codificação sem graça. Uma alternativa interessante e um pouco desconhecida é um proxy dinâmico, que utiliza reflection para interceptar chamadas de métodos. O pacote `java.lang.reflection` oferece a interface `InvocationHandler` e uma classe (proxy) que, juntas, podem ser utilizadas para criar dinamicamente uma instância que implementa múltiplas interfaces especificadas em tempo de execução. Essa abordagem não requer uma definição em tempo de compilação das interfaces que o proxy implementa. Uma vez instanciado, pode ser feito um cast do proxy para qualquer uma das interfaces especificadas na criação, e qualquer chamada para um método definido por essas interfaces é encaminhada a um único método (`invoke`) do proxy. O único requisito para a classe de proxy dinâmico é que ela implemente a interface `InvocationHandler`, a qual define o método `invoke`.

Vamos desenvolver um proxy dinâmico para o Chat, que rastreia as invocações do listener de mensagens. Lembre que o Chat utiliza a interface `MessageListener` para associar o frame principal com o servidor RMI. Mesmo que o `MessageListener` tenha apenas um método, ele será suficiente para ilustrar o conceito. Colocaremos o proxy dinâmico entre a instância de `MainFrame` e a de `ChatServer`, para adicionar o rastreamento das chamadas de métodos. Criaremos uma classe `TracingProxy` no pacote `covertjava.intercept` e faremos implementar a interface `InvocationHandler`. O proxy vai delegar as invocações de métodos para o objeto real, por isso vamos codificar o construtor de forma que receba o objeto-alvo como um parâmetro. A declaração da classe `TracingProxy` e o seu construtor são mostrados na Listagem 16.5.

#### **LISTAGEM 16.5 Declaração de TracingProxy**

```
public class TracingProxy implements InvocationHandler {  
  
    protected Object target;  
  
    public TracingProxy(Object target) {  
        this.target = target;  
    }  
    ...  
}
```

Note que o proxy de rastreamento recebe como alvo um `java.lang.Object`. Esse é o ponto principal, pois a classe proxy não está amarrada a `MessageListener`, portanto, pode ser utilizada em *qualquer* interface.

Agora temos de codificar o método `invoke( )` da interface `InvocationTarget`. Ele recebe três parâmetros – o próprio objeto proxy, o método, e o array de parâmetros do método. Nossa implementação imprime o nome do método e então delega a invocação para o alvo que foi passado para o construtor do proxy. A Listagem 16.6 mostra esse código.

**LISTAGEM 16.6** Implementação de invoke( )

```
public Object invoke(Object proxy, Method method, Object[ ] args) throws Throwable {  
    Object result;  
    try {  
        System.out.println("Entering " + method.getName( ));  
        result = method.invoke(target, args);  
    }  
    catch (InvocationTargetException e) {  
        throw e.getTargetException( );  
    }  
    finally {  
        System.out.println("Leaving " + method.getName( ));  
    }  
    return result;  
}
```

---

Agora o proxy está pronto para um “test drive”. Para vê-lo em ação, vamos criar uma instância de TracingProxy inicializada com uma instância MainFrame do Chat como o alvo. Então, vamos criar um objeto java.lang.reflect.Proxy, que implementa a interface MessageListener dinamicamente e delega chamadas à instância do proxy de rastreamento. Por fim, passaremos o proxy de reflection para o servidor do Chat, fazendo um cast para a interface MessageListener. A Listagem 16.7 mostra o código Java correspondente.

**LISTAGEM 16.7** Utilizando um proxy dinâmico

```
public static void main(String[ ] args) throws Exception {  
    ChatServer chatServer = ChatServer.getInstance( );  
    chatServer.setMessageListener(new MainFrame(false));  
  
    TracingProxy listener = new TracingProxy(chatServer.getMessageListener( ));  
    Object proxy = Proxy.newProxyInstance(  
        chatServer.getClass( ).getClassLoader( ),  
        new Class[ ] {MessageListener.class},  
        listener  
    );  
    chatServer.setMessageListener((MessageListener)proxy);  
    MessageInfo messageInfo = new MessageInfo("localhost", "alex");  
    chatServer.receiveMessage("Test message", messageInfo);  
    System.exit(0);  
}
```

---

Executar o método `main( )` de `TracingProxy` gera a saída mostrada aqui:

```
C:\Projects\CovertJava\classes>java covertjava.intercept.TracingProxy
Received message from host localhost
Entering messageReceived
Leaving messageReceived
```

Portanto, fomos capazes de interceptar uma chamada para o método `messageReceived`, sem implementar a interface `MessageInfo`. Proxies dinâmicos também são úteis para o desenvolvimento de frameworks e ferramentas, quando você precisa interfacear com classes cujos tipos são desconhecidos em tempo de compilação. Em vez de ter de gerar e compilar classes proxy estáticas em Java, frameworks podem usar proxies dinâmicos como "cola" entre os componentes.

## A interface do profiler da máquina virtual Java

Uma evolução promissora é a introdução do Java Virtual Machine Profiler Interface (JVMPPI), que padroniza a interação entre um profiler e a JVM. O recurso foi exposto pela primeira vez no JDK 1.2.2 e estendido no JDK 1.4. A API é uma interface de duas vias que especifica como uma máquina virtual deve notificar um agente de profiler sobre os eventos dentro da VM, tais como inicializações de threads, chamadas de métodos e alocações de memória. Ela também especifica o meio de um profiler obter informações sobre o estado da JVM e de configurar eventos em que ele estiver interessado. O agente de profiler é executado dentro da JVM, e todos os métodos da API são funções no estilo C, invocadas via JNI. Para acesso a essa API, a JVM tem de ser iniciada com o parâmetro `-Xrun ProfilerLibrary`, onde `ProfilerLibrary` é o nome da biblioteca nativa a ser carregada. Infelizmente, não existe uma interface Java para a JVMPPI, e entrar nos detalhes de implementações C e JNI está fora do escopo deste livro. Entretanto, inclui uma lista dos eventos mais interessantes que podem ser interceptados:

- `JVMPPI_EVENT_CLASS_LOAD` – enviado quando uma classe é carregada.
- `JVMPPI_EVENT_CLASS_LOAD_HOOK` – enviado depois que os dados de classes são carregados pelo class loader, mas antes de a representação interna da classe ser criada. Isso fornece a capacidade para o profiler decorar ou instrumentar o bytecode.
- `JVMPPI_EVENT_METHOD_ENTRY` – enviado ao entrar em um método.
- `JVMPPI_EVENT_METHOD_EXIT` – enviado ao sair de um método.
- `JVMPPI_EVENT_THREAD_START` – enviado quando um thread é iniciado.
- `JVMPPI_EVENT_THREAD_END` – enviado quando um thread é encerrado.

Uma referência completa sobre JVMPPI pode ser localizada em

<http://java.sun.com/j2se/1.4.2/docs/guide/jvmpci/jvmpci.html>

## Questionário rápido

1. Por que e onde em uma aplicação é importante utilizar `java.lang.Throwable`?
2. Como a saída para o stream de erro de sistema pode ser redirecionada para um banco de dados?
3. Como uma chamada a `System.exit( )` pode ser interceptada?
4. Como uma aplicação Java, rodando como um serviço, pode fechar todas as conexões de banco de dados quando a máquina estiver se desligando?
5. Que eventos podem ser recebidos pela JVMPPI?

## Resumo

- Não existe uma boa maneira de interceptar o fluxo de controle em Java. A JVMPPI fornece recursos para interferir na execução, mas requer programação JNI.
- Erros de sistema são informados como erros não-declarados e podem ser capturados como objetos `java.lang.Throwable`.
- Os streams padrão de saída e de erro de sistema podem ser redirecionados programaticamente para um `PrintStream` personalizado.
- Uma chamada a `System.exit( )` pode ser interceptada instalando-se um `SecurityManager` personalizado, que proíbe a saída da JVM até que ela seja explicitamente permitida.
- Aplicações podem executar código no desligamento da JVM utilizando ganchos de desligamento (*shutdown hooks*). Ganchos são threads que são iniciados pela JVM quando um sinal de desligamento é recebido.
- A JVMPPI fornece um controle extenso sobre o ambiente de execução, o carregamento de classes e a execução de métodos.

# Entendendo e ajustando o bytecode

*"Cada solução gera novos problemas."*

Quinto corolário de Murphy

## Fundamentos de bytecode

O Capítulo 2 apresentou uma breve visão geral sobre o bytecode e de seu propósito em Java. Como você sabe, o bytecode é o passo intermediário entre o código-fonte e o código de máquina, que permite a execução multiplataforma de programas Java. O bytecode é definido pela Java Virtual Machine Specification (<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>), que também descreve os conceitos da linguagem, o formato de arquivos de classes, os requisitos da Java Virtual Machine (JVM) e outros aspectos importantes da linguagem Java. A adesão rigorosa à especificação assegura a portabilidade e a execução onipresente de aplicações compiladas para bytecode. A JVM, que executa sobre o sistema operacional, é responsável por fornecer o ambiente de execução e por converter as instruções de bytecode Java em instruções nativas de máquina.

A maioria das técnicas de hacking apresentadas anteriormente neste livro pressupõe obter e manipular o código-fonte para alterar o comportamento de uma aplicação. Neste capítulo, vamos trabalhar no nível de bytecode, em vez de no nível de código-fonte. Vamos descobrir como visualizar as estruturas de dados de arquivos de classes, instrumentar (aprimorar) o bytecode existente e gerar novas classes programaticamente. Eis alguns benefícios de fazer alterações no nível de bytecode:

# 17

## NESTE CAPÍTULO

- ▶ Fundamentos de bytecode 149
- ▶ Visualizando arquivos de classes com o visualizador de bytecode jClassLib 150
- ▶ O conjunto de instruções da JVM 150
- ▶ O formato de arquivos de classes 152
- ▶ Instrumentando e gerando bytecode 158
- ▶ Ajuste de bytecode em comparação com proxies dinâmicos e AOP 166
- ▶ Questionário rápido 166
- ▶ Resumo 167

- Você não precisa obter o código-fonte ou descompilar o bytecode e depois recompilar o fonte.
- O bytecode pode ser gerado ou instrumentado por um class loader em tempo real, enquanto as classes são carregadas em uma JVM.
- É mais fácil e mais rápido automatizar a geração de bytecode do que a de código-fonte, porque há menos passos e o compilador não precisa ser executado. Por exemplo, o Hibernate gera código de persistência para classes Java em tempo de execução.
- Ferramentas podem usar a instrumentação de bytecode para introduzir lógica adicional que não precisa estar nos arquivos-fonte. Algumas implementações de AOP (Aspect Oriented Programming), por exemplo, inserem atributos personalizados no bytecode e instrumentam os métodos para suportar AOP.

As duas próximas seções apresentam uma breve introdução aos aspectos da especificação da JVM relacionados com o bytecode. Embora seja útil se familiarizar com a maneira como a JVM opera e com o formato de arquivo de classes, esse conhecimento não é estritamente necessário para implementar as técnicas apresentadas neste capítulo. Se você não costuma ser paciente e se, para você, ler coisas como especificações é como escrever documentação para o usuário final, sinta-se à vontade para pular as duas próximas seções e ir direto para a seção “Instrumentando e gerando bytecode”.

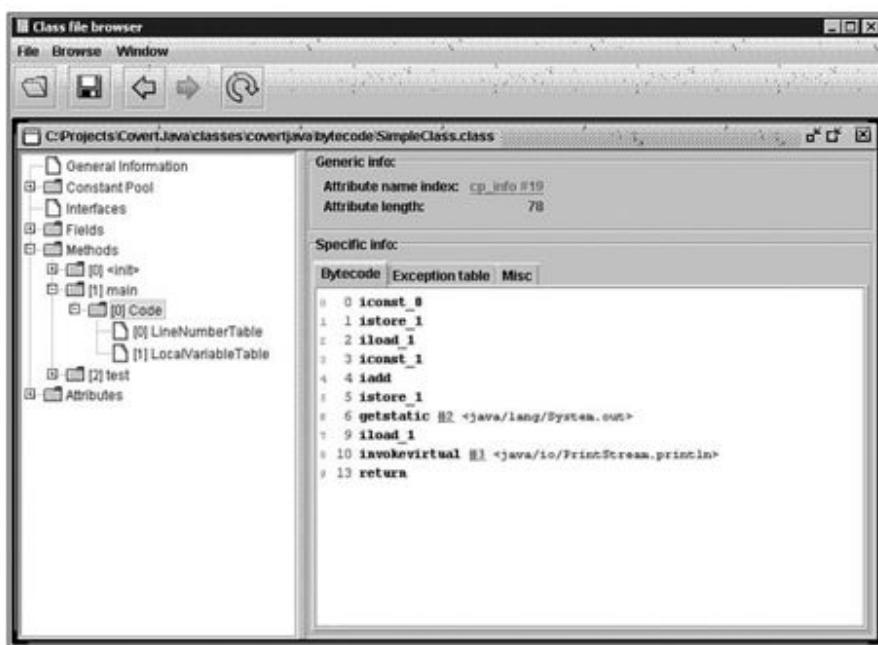
## Visualizando arquivos de classes com o visualizador de bytecode jClassLib

O Bytecode Viewer, distribuído com a biblioteca livre jClassLib, é um excelente utilitário gráfico que permite navegar pelo conteúdo de arquivos de classes. Ele mostra uma visualização hierárquica da estrutura de arquivos no painel esquerdo e o conteúdo do elemento selecionado no painel direito. A Figura 17.1 mostra o jClassLib exibindo o conteúdo de `SimpleClass`, no pacote `covertjava.bytecode`.

O jClassLib Bytecode Viewer não permite modificações do arquivo de classe, mas é excelente para visualizar as estruturas que são apresentadas nas próximas seções. Uma maneira útil de entender o bytecode é comparando as instruções de bytecode com instruções e operadores no código-fonte. Esse visualizador também pode ser utilizado para depurar a geração e a instrumentação do bytecode, que realizaremos no final deste capítulo.

## O conjunto de instruções da JVM

As fontes Java são compiladas em arquivos de classes binários, que têm um formato específico. A lógica de cada método Java é representada como um conjunto de instruções primitivas da JVM, definidos na especificação da JVM. As instruções da JVM são comandos básicos, semelhantes a código de máquina. Cada instrução consiste em um código de operação (*opcode*), seguido por zero ou mais operandos, que representam os parâmetros da operação. No arquivo de classe, as instruções são armazenadas como um stream biná-



**FIGURA 17.1** O jClassLib Bytecode Viewer

rio, que representa o atributo `Code` de um método. O opcode é armazenado como um byte, que pode ser seguido pelos bytes que representam dados de operandos. Por exemplo, o código-fonte mostrado na Listagem 17.1 é representado pelo conjunto de instruções mostrado na Listagem 17.2.

#### **LISTAGEM 17.1** Exemplo de código-fonte Java

---

```

int i = 0;
i = i + 1;
System.out.println(i);

```

---

### **HISTÓRIAS DAS TRINCHEIRAS**

O Hibernate é um serviço livre de alto desempenho de persistência objeto/relacional e consultas para Java. Um dos pontos altos do Hibernate é sua capacidade de persistir objetos Java de forma transparente. Em vez de codificar chamadas JDBC tediosas, os desenvolvedores escrevem um arquivo XML de mapeamento de objetos para um esquema de banco de dados, e o Hibernate cuida de todo o trabalho de infra-estrutura. O serviço de persistência se baseia em reflection e na geração de bytecode em tempo de execução, garantindo que não haja impacto sobre a depuração em IDEs ou na compilação incremental. O Hibernate defende que a Byte Code Engineering Library (BCEL) do Apache – e posteriormente a biblioteca de geração de bytecode CGLIB – para manipulação de bytecode, permite evitar o overhead da API de reflection de Java.

**LISTAGEM 17.2** Representação em bytecode do exemplo de código-fonte

```
0  iconst_0
1  istore_1
2  iinc 1 by 1
5  getstatic #21 <java/lang/System.out>
8  iload_1
9  invokevirtual #27 <java/io/PrintStream.println>
12 return
```

A maioria das instruções é muito simples e é fácil rastrear as instruções para o código-fonte que elas representam. Por exemplo, `iconst_0` define uma constante inteira com um valor 0 e `istore_1` armazena o valor do topo da pilha (0 em nosso caso) em uma variável local especificada por um índice (`i` no caso). Um cenário mais interessante é uma chamada de método. Como você pode ver nas listagens, o nome do campo de classe estático (`System.out`) e o valor do parâmetro (`i`) são primeiros colocados na pilha de operandos, antes de o método `println` ser invocado. Informações detalhadas sobre as instruções podem ser obtidas na especificação da JVM, mas isso está além do escopo deste livro. É útil se familiarizar com as instruções e seus operandos, mas iremos utilizar um framework que fornece uma camada de abstração para o bytecode. A instrumentação e a geração de bytecode requerem construir programaticamente conjuntos de instruções; portanto, pelo menos um entendimento básico do conjunto de instruções e do modo como as instruções são mapeados para código Java é essencial.

## O formato de arquivos de classes

O formato de arquivo de classe binário é determinado pela especificação da JVM. Ele é descrito por uma série de estruturas de dados representando a própria classe, seus métodos, e seus campos e atributos. Para manipular o bytecode, você precisa saber as convenções de atribuição de nomes utilizada para vários elementos, e o formato das principais estruturas de dados.

### Descritores de campos e de métodos

O Java suporta métodos sobrecarregados (*overloaded*), através da união do método com o descritor, que é criado com base nos parâmetros que o método recebe. Dessa maneira, internamente, `print(int i)` e `print(char ch)` são armazenados como dois métodos separados. A desfiguração de nomes segue uma convenção determinada pela especificação da JVM, e como o bytecode armazena os nomes desfigurados, você pode obter uma idéia de como isso é feito aqui.

Os descritores de campos e de métodos são codificados com base em seus tipos. A Tabela 17.1 mostra o tipo Java declarado e o tipo de descritor de campo correspondente utilizado no bytecode.

**TABELA 17.1****Códigos de tipo de campo**

TIPO DECLARADO	TIPO DE DESCRIPTOR
byte	B
char	C
double	D
float	F
int	I
long	J
short	S
boolean	Z
Instância de InClassname	L<Classname>;
[ ] (uma dimensão de array)	[

A Tabela 17.2 mostra alguns exemplos de declarações em Java e seus descritores no bytecode.

**TABELA 17.2****Exemplos de tipos de descritor**

DECLARAÇÃO DE TIPO	TIPO DE DESCRIPTOR
int number;	I
int[ ][ ] numbers;	[[I
Object reference;	Ljava.lang.Object;

Os descritores de métodos são criados com o seguinte formato:

([<param1>[...<paramN>]])<return>

onde

- <param1> ... <paramN> são descritores opcionais de tipos de parâmetros.
- <return> é o descritor de tipo de retorno ou V, se o método for void.

Por exemplo, um método declarado como

Integer getIntProperty(String propertyName, int defaultValue)

teria o descritor de método

(Ljava.lang.String;I)Ljava.lang.Integer;

Certos métodos especiais têm nomes predefinidos. Os inicializadores estáticos são chamados <clinit> e os inicializadores de instância e construtores são chamados de <init>.

## Estrutura de arquivos de classes

Cada classe Java é definida por um stream binário, tipicamente armazenado em um arquivo de classe, consistindo em bytes de 8 bits. O conteúdo do stream é descrito por uma pseudo-estrutura dada na especificação da JVM e copiada aqui na Listagem 17.3. Embora isso possa parecer informação demais, as estruturas apresentadas nesta e nas próximas seções nos ajudarão posteriormente a entender a geração e a instrumentação de bytecode.

### **LISTAGEM 17.3 Estrutura de ClassFile**

---

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

---

Para maior clareza, a especificação da JVM define os pseudo-tipos `u1`, `u2` e `u4`, que representam tipos de 1, 2 e 4 bytes sem sinal, respectivamente. A Tabela 17.3 lista todos os campos da estrutura de `ClassFile` e os seus significados.

**TABELA 17.3**

#### **Campos de ClassFile**

CAMPO	DESCRIÇÃO
Magic	Marcador de formato de arquivo de classe. Tem sempre o valor 0xCAFEBABE.
Minor_version, major_version	Versão da JVM para a qual o arquivo de classe foi compilado. JVMs podem suportar versões principais menores, mas não executam em versões principais maiores.

**TABELA 17.3****Continuação**

CAMPO	DESCRIÇÃO
constant_pool_count	Número de itens no array do pool de constantes ( <i>constant pool array</i> ). O primeiro item do pool de constantes é reservado para utilização interna pela JVM, portanto os valores válidos para constant_pool_count são 1, ou maior.
constant_pool[ ]	Array de estruturas de comprimento variável, representando constantes de strings, nomes de classes e de campos, e outras constantes.
access_flags	Máscara de modificadores utilizada nas declarações de classes ou interfaces. Os modificadores válidos são ACC_PUBLIC, ACC_FINAL, ACC_SUPER, ACC_INTERFACE e ACC_ABSTRACT.
this_class	Um índice do item do array constant_pool que descreve essa classe.
super_class	Zero ou um índice do item do array constant_pool que descreve a superclasse para essa classe. Para uma classe, um valor 0 indica que a superclasse é java.lang.Object.
interfaces_count	Número de superinterfaces dessa classe ou interface.
interfaces[ ]	Array de índices de itens constant_pool, que descreve as superinterfaces dessa classe.
fields_count	Número de itens no array fields.
fields[ ]	Array de estruturas de comprimento variável que descreve os campos declarados nessa classe.
Methods_count	Número de itens no array methods.
Methods[ ]	Array de estruturas de comprimento variável que descreve os métodos declarados nessa classe, incluindo o bytecode dos métodos.
attributes_count	Número de itens no array attributes.
Attributes[ ]	Array de estruturas de comprimento variável que declara atributos desse arquivo de classe.  Dentre os atributos padrão, estão SourceFile, LineNumberTable e outros. A JVM é obrigada a ignorar os atributos que não conhece.

O pool de constantes merece um pouco mais de atenção, porque é usado freqüentemente por outras estruturas. Qualquer string em uma classe Java, independentemente de sua natureza, é armazenada no mesmo pool de constantes. Isso inclui o nome da classe, nomes de campos e de métodos, nomes das classes e métodos que a classe invoca, e as strings literais utilizadas no código Java. Sempre que um nome ou string precisar ser utilizado, ele referencia o pool de constantes por índice. O pool de constantes é um array de estruturas cp\_info; seu formato geral é mostrado na Listagem 17.4.

**LISTAGEM 17.4 Estrutura de itens do pool de constantes**


---

```
cp_info {
    u1 tag;
    u1 info[ ];
}
```

---

Os verdadeiros itens armazenados no pool seguem a estrutura que corresponde ao tag. Por exemplo, uma string é definida, usando uma estrutura `CONSTANT_String` e uma referência a um campo, usando `CONSTANT_Fieldref`. A lista de estruturas e seus conteúdos estão na especificação da JVM.

A estrutura `ClassFile` utiliza três outras estruturas: `field_info`, `method_info` e `attribute_info`. `field_info` é semelhante a `method_info`, assim mostraremos apenas a estrutura `method_info` na Listagem 17.5.

**LISTAGEM 17.5 Estrutura de method\_info**


---

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

---

Os significados dos campos `method_info` são fornecidos na Tabela 17.4.

**TABELA 17.4****Campos de method\_info**

CAMPO	DESCRIÇÃO
<code>access_flags</code>	Máscara de modificadores que descreve a acessibilidade e propriedades de métodos, incluindo <code>static</code> , <code>final</code> , <code>synchronized</code> , <code>native</code> e <code>abstract</code> .
<code>name_index</code>	Índice do array <code>constant_pool</code> , que representa o nome do método.
<code>descriptor_index</code>	Índice do array <code>constant_pool</code> , que representa o descritor do método.
<code>attributes_count</code>	Número de itens no array <code>attributes</code> .
<code>attributes[ ]</code>	Array de atributos do método. Atributos definidos na especificação da JVM incluem <code>Code</code> e <code>Exceptions</code> . Atributos não reconhecidos pela JVM são ignorados.

## Atributos

Atributos são utilizados nas estruturas `ClassFile`, `field_info`, `method_info` e `Code_attribute` para fornecer informações adicionais que dependem do tipo de estrutura. Por exemplo, atributos de classes incluem o nome de arquivo de origem e informações de depuração; atributos de métodos incluem o bytecode e exceções. A Listagem 17.6 mostra a estrutura de `attribute_info`; a Tabela 17.5 lista seus campos.

**LISTAGEM 17.6** Estrutura de `attribute_info`

---

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

---

**TABELA 17.5**

**Campos de `attribute_info`**

CAMPO	DESCRIÇÃO
<code>attribute_name_index</code>	Índice em <code>constant_pool</code> que representa o nome do atributo
<code>attribute_length</code>	Comprimento do array <code>attribute_info</code> em bytes
<code>attribute_info</code>	Conteúdo binário do atributo

---

Compiladores e pós-processadores têm permissão de definir e nomear novos atributos, contanto que isso não afete a semântica da classe. Por exemplo, implementações de AOP podem utilizar atributos de bytecode para armazenar aspectos definidos para uma classe.

## Verificação de bytecode

Quando um compilador compila código-fonte Java em bytecode, ele faz verificações extensas na sintaxe, uso de palavras-chave, operadores e outros possíveis erros. Isso garante que o bytecode gerado seja válido e que seja seguro para execução. Quando a classe é carregada em uma JVM, executa-se um subconjunto simplificado de verificações para garantir que o arquivo de classe tenha o formato correto, e que não foi modificado indevidamente. Por exemplo, o verificador de bytecode (*bytecode verifier*) verifica se os primeiros 4 bytes contêm o número mágico e se os atributos têm o comprimento adequado. Ele garante que classes *final* não tenha sido estendidas e que campos e métodos tenham referências corretas no pool de constantes; e realiza várias outras verificações.

## Instrumentando e gerando bytecode

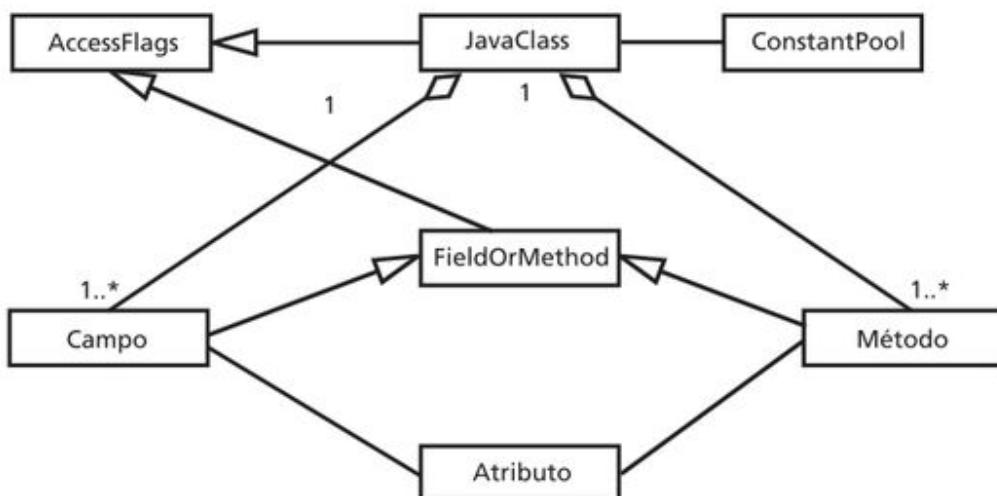
Chegamos ao ponto em que você pode finalmente pôr a mão no teclado e fazer algumas coisas interessantes. Agora que você sabe o suficiente sobre o bytecode, você pode implementar os dois métodos de manipulação de bytecode mais comuns. Obviamente, trabalhar diretamente com o conteúdo do arquivo de classe binário é uma tarefa tediosa. Para tornar nosso trabalho mais fácil, utilizaremos uma biblioteca livre da Apache chamada Byte Code Engineering Library (BCEL).

### Visão geral da BCEL

A home page da BCEL está localizada em <http://jakarta.apache.org/bcel>, onde você pode baixar a distribuição binária, o código-fonte e o manual. A biblioteca fornece uma API orientada a objetos para trabalhar com as estruturas e os campos que compõem uma classe. Ela pode ser utilizada para ler um arquivo de classe existente e representá-lo com uma hierarquia de objetos; para transformar a representação da classe, adicionando campos, métodos e código binário; e para gerar programaticamente novas classes a partir do zero. A representação da classe pode ser salva em um arquivo, ou passada para a JVM como um array de bytes, para suportar a instrumentação e a geração em tempo real. A BCEL vem até com um class loader que pode ser utilizado para instrumentar dinamicamente ou criar classes em tempo de execução. O diagrama de classes das principais classes da BCEL é mostrado na Figura 17.2.

A Tabela 17.6 fornece descrições breves das principais classes que vamos utilizar. Informações detalhadas estão disponíveis no JavaDoc da BCEL.

Como você pode ver, a maioria das classes é um mapeamento direto para os termos e estruturas de dados definidos na especificação da JVM.



**FIGURA 17.2** Diagrama de classes das classes principais da BCEL.

**TABELA 17.6****Principais classes da BCEL**

CLASSE DA BCEL	DESCRIÇÃO
JavaClass	Representa uma classe Java existente. Contém campos, métodos, atributos, o pool de constantes e outras estruturas de dados de classes.
Field	Representa a estrutura field_info.
Method	Representa a estrutura method_info.
ConstantPool	Representa um pool de constantes contido na classe.
ClassGen	Cria dinamicamente uma nova classe. Pode ser inicializada com uma classe existente.
FieldGen	Cria dinamicamente um novo campo. Pode ser inicializada com um campo existente.
MethodGen	Cria dinamicamente um novo método. Pode ser inicializada com um método existente.
ConstantPoolGen	Cria dinamicamente um novo pool de constantes. Pode ser inicializada com um pool de constantes existente.
InstructionFactory	Cria instruções a serem inseridas no bytecode.
InstructionList	Armazena uma lista de instruções de bytecode.
Instruction	Representa uma instrução, como iconst_0 ou invokevirtual.

**Instrumentando métodos**

*Instrumentar* é inserir novo bytecode ou estender o bytecode existente de uma classe. Os produtos que produzem métricas de desempenho de aplicações Java em tempo de execução utilizam a instrumentação para coletar esses dados. Para uma experiência prática, vamos desenvolver um framework que produz um log de invocações de métodos em tempo de execução. O Omniscent Debugger, discutido no Capítulo 9, utiliza uma técnica semelhante para gravar a execução de programas, então podemos vê-lo posteriormente. Gravar as invocações de métodos em tempo de execução fornece o benefício de gerar um log detalhado do código executado pela JVM.

Para testar a implementação, utilizaremos uma classe chamada SimpleClass definida no pacote `covertjava.bytecode`, com um método `main`, que é mostrado na Listagem 17.7.

**LISTAGEM 17.7** Método main( ) de SimpleClass

```
public static void main(String[ ] args) {
    int i = 0;
    i = i + 1;
    System.out.println(i);
}
```

Para manter o exemplo simples, não vamos escrever todo o framework de logging de invocações. Em vez disso, limitaremos a implementação à classe `InvocationRegistry` com um método `static`, como mostrado na Listagem 17.8.

#### **LISTAGEM 17.8** Ponto de entrada do framework de logging de métodos

---

```
public static void methodInvoked(String methodName) {
    System.out.println("**** method invoked " + methodName);
}
```

---

`methodInvoked( )` é o ponto de entrada do framework de logging de métodos e é utilizado para registrar em log uma invocação de método. Para cada thread, ele pode armazenar uma pilha de chamadas de métodos, que pode ser salva ou impressa ao final da execução da aplicação. Por enquanto, a implementação só imprime o nome de método para indicar que o framework foi chamado para esse método.

Com os princípios básicos prontos, podemos passar à implementação da classe, que fará a instrumentação de bytecode de métodos. Vamos chamar `MethodInstrumentor` e faremos seu método `main( )` receber o nome da classe e os métodos que queremos instrumentar na linha de comando. Quando executado, o `MethodInstrumentor` carrega a classe dada, instrumenta os métodos cujos nomes correspondem à expressão regular dada (adicionando uma chamada a `InvocationRegistry.methodInvoked( )`), e depois salva a classe com um novo nome. Se executarmos a nova versão da classe, devem ser registradas suas invocações de métodos no Registry. `MethodInstrumentor` está localizado no pacote `covertjava.bytecode`; utilizaremos uma abordagem de cima para baixo (*top-down*) para desenvolvê-lo. O método `main( )` de `MethodInstrumentor` é mostrado na Listagem 17.9.

#### **LISTAGEM 17.9** Método `main( )` de `MethodInstrumentor`

---

```
public static void main(String[ ] args) throws IOException {
    if (args.length != 2) {
        System.out.println("Syntax: MethodInstrumentor " +
                           "<full class name> <method name pattern>");
        System.exit(1);
    }
    JavaClass cls = Repository.lookupClass(args[0]);
    MethodInstrumentor instrumentor = new MethodInstrumentor();
    instrumentor.instrumentWithInvocationRegistry(cls, args[1]);
    cls.dump("new_" + cls.getClassName( ) + ".class");
}
```

---

Depois de verificar a sintaxe da linha de comando, o `MethodInstrumentor` tenta carregar a classe dada, utilizando a classe `Repository` da `BCEL`. `Repository` utiliza o classpath de aplicação para localizar e carregar a classe; esta é apenas uma das muitas alternativas para carregar uma classe com a `BCEL`. Por alguma razão inexplicável, a `BCEL` retorna `null` em

condições de erro em vez de lançar uma exceção, mas para manter a clareza do código não vamos verificar isso. Depois de a classe ser carregada, uma instância de `MethodInstrumentor` é criada, e seu método `instrumentWithInvocationRegistry()` é chamado para realizar as transformações. Ao terminar, a classe é salva em um arquivo com um novo nome. Vejamos a implementação de `instrumentWithInvocationRegistry` mostrada na Listagem 17.10.

#### **LISTAGEM 17.10** Implementação de `instrumentWithInvocationRegistry`

---

```
public void instrumentWithInvocationRegistry(JavaClass cls,
                                            String methodPattern) {
    ConstantPoolGen constants = new ConstantPoolGen(cls.getConstantPool());
    Method[] methods = cls.getMethods();

    for (int i = 0; i < methods.length; i++) {
        // Instrumenta todos os métodos que correspondem aos critérios dados
        if (Pattern.matches(methodPattern, methods[i].getName())) {
            methods[i] = instrumentMethod(cls, constants, methods[i]);
        }
    }
    cls.setMethods(methods);
    cls.setConstantPool(constants.getFinalConstantPool());
}
```

---

Como vamos adicionar a invocação de um método de uma classe diferente, devemos referenciá-la por nome. Lembre que todos os nomes são armazenados no pool de constantes, o que significa que teremos de adicionar novas constantes ao pool existente. Para adicionar novos elementos às estruturas na BCEL, devemos usar as classes geradoras, que têm um sufixo `Gen` em seus nomes. O código cria uma instância de `ConstantPoolGen`, que é inicialmente preenchida com as constantes do pool existente; depois itera sobre todos os métodos, usando expressões regulares para testar quais métodos devem ser instrumentados. Quando todos os métodos forem processados, a classe é atualizada com os novos métodos e com o novo pool de constantes. O verdadeiro trabalho de instrumentação é feito em `instrumentMethod()`, como mostrado na Listagem 17.11.

#### **LISTAGEM 17.11** Implementação de `instrumentMethod()`

---

```
public Method instrumentMethod(JavaClass cls, ConstantPoolGen constants,
                               Method oldMethod) {
    System.out.println("Instrumenting method " + oldMethod.getName());
    MethodGen method = new MethodGen(oldMethod, cls.getClassName(), constants);
    InstructionFactory factory = new InstructionFactory(constants);
    InstructionList instructions = new InstructionList();

    // Acrescenta duas instruções que representam uma chamada de método
```

**LISTAGEM 17.11 Continuação**

```
instructions.append(new PUSH(constants, method.getName( )));
Instruction invoke = factory.createInvoke(
    "covertjava.bytecode.InvocationRegistry",
    "methodInvoked",
    Type.VOID,
    new Type[ ] {new ObjectType("java.lang.String")},
    Constants.INVOKESTATIC
);
instructions.append(invoke);

method.getInstructionList( ).insert(instructions);
instructions.dispose( );
return method.getMethod( );
}
```

Como você pode ver, `instrumentMethod( )` cria programaticamente instruções de bytecode que correspondem a uma chamada de método. A maneira mais fácil de selecionar as instruções de JVM corretas e seus parâmetros é escrever primeiro o código em Java, compilá-lo e depois utilizar algo como o visualizador jClassLib para examinar como o código é convertido em bytecode. Então o bytecode correspondente pode ser construído utilizando objetos da BCEL.

A primeira coisa que `instrumentMethod( )` faz é instanciar um objeto `MethodGen`, que é utilizado para armazenar o novo bytecode. Em seguida, são geradas uma fábrica e uma lista para armazenar as instruções criadas. Se você prestou atenção neste capítulo e executou o jClassLib Bytecode Viewer, vai lembrar que uma chamada de método Java é representada por várias instruções de bytecode. Primeiro, os parâmetros de métodos devem ser colocados na pilha de operandos; então a instrução `invokevirtual` é emitida para transferir o controle para o método (consulte a Listagem 17.2 para um exemplo de bytecode de chamada de método). Isso é exatamente o que temos de inserir no código do método antes de seu bytecode existente. Se estivéssemos trabalhando diretamente com o bytecode, teríamos de inserir duas constantes no pool de constantes: `covertjava.bytecode.InvocationRegistry` para o nome de classe e o `methodInvoked` para o nome de método. Felizmente, a BCEL faz isso para nós, porque estamos utilizando classes de alto nível como `InstructionFactory` e `PUSH`, que automaticamente adicionam constantes ao pool. Depois de criadas as instruções, elas são acrescentadas à lista de instruções. Quando a parte de geração de código for concluída, a lista é inserida nas instruções do método geradas, e a estrutura do método é retornada.

Para testar se a instrumentação funciona, compile as classes e execute `MethodInstrumentor` em `SimpleClass.class` utilizando a seguinte linha de comando:

```
java covertjava.bytecode.MethodInstrumentor covertjava.bytecode.SimpleClass .*
```

Um novo arquivo de classe chamado `new_covertjava.bytecode.SimpleClass.class` deve ser criado no diretório atual. Copie esta classe para o diretório de classes, sobrescrevendo o

arquivo SimpleClass.class existente; execute então o método SimpleClass main( ). Se tudo funcionar bem, você deve ver o seguinte no console:

```
C:\Projects\CovertJava\classes>java covertjava.bytecode.SimpleClass
*** method invoked main
1
```

Como você pode ver, a classe instrumentada começa chamando InvocationRegistry, que gera a primeira linha como saída; depois executa seu próprio corpo, gerando 1 como saída.

## Gerando classes

Nossa segunda tarefa é aprender a gerar uma nova classe programaticamente. Como mencionamos, isso é útil para produtos middleware e frameworks que querem evitar a geração de código-fonte. Em nosso exemplo, criaremos um gerador de *value objects*, contendo todos os campos de uma classe dada, mas nenhum método. O value object é um padrão de projeto comum, utilizado em aplicações distribuídas para enviar dados pela rede. Nosso gerador vai produzir uma versão muito limitada de value objects, mas o tornaremos mais interessante garantindo que só sejam gerados campos cujos valores devem ser retidos.

Mais uma vez, utilizaremos SimpleClass como cobaia em nossa experiência. SimpleClass define cinco campos, como mostra a Listagem 17.12.

### LISTAGEM 17.12 Campos de SimpleClass

```
public int number;
protected String name;
private Thread myThread;
static String className;
transient String transientName;
```

Vamos escrever uma classe ClassGenerator, no pacote covertjava.bytecode, que aceita dois parâmetros de linha de comando – um nome de classe completo e um padrão de expressão regular para os nomes dos campos a serem copiados. O método main( ) de ClassGenerator é mostrado na Listagem 17.13.

### LISTAGEM 17.13 Método main( ) de ClassGenerator

```
public static void main(String[ ] args) throws IOException {
    if (args.length != 2) {
        System.out.println("Syntax: ClassGenerator " +
                           "<full class name> <field name pattern>");
        System.exit(1);
    }
```

**LISTAGEM 17.13** Continuação

---

```
JavaClass sourceClass = Repository.lookupClass(args[0]);
ClassGenerator generator = new ClassGenerator();
JavaClass valueClass = generator.generateValueObject(sourceClass, args[1]);
valueClass.dump(valueClass.getClassName() + ".class");
}
```

---

Assim como em `MethodInstrumentor`, a implementação verifica a sintaxe da linha de comando, carrega a classe e, em seguida, chama o método `generateValueObject()`, que é mostrado na Listagem 17.14.

**LISTAGEM 17.14** Método `generateValueObject()` de `ClassGenerator`


---

```
public JavaClass generateValueObject(
    JavaClass sourceClass,
    String fieldPattern)
{
    String newName = sourceClass.getClassName() + "Value";
    ClassGen classGen = new ClassGen(
        newName,
        "java.lang.Object",
        newName,
        Constants.ACC_PUBLIC | Constants.ACC_SUPER,
        new String[] { "java.io.Serializable" });
    Field[] fields = sourceClass.getFields();
    for (int i = 0; i < fields.length; i++) {
        if (Pattern.matches(fieldPattern, fields[i].getName())) {
            int skipFlags = Constants.ACC_STATIC | Constants.ACC_TRANSIENT;
            if ((fields[i].getAccessFlags() & skipFlags) == 0) {
                fields[i].setAccessFlags(Constants.ACC_PUBLIC);
                addField(classGen, fields[i]);
            }
        }
    }
    return classGen.getJavaClass();
}
```

---

A implementação cria primeiro uma instância de `ClassGen` para representar a classe sendo gerada. A classe tem o mesmo nome da classe de parâmetro, mas tem o sufixo `Value`. Ela estende `java.lang.Object` e implementa `java.io.Serializable`. Em seguida, a implementação itera pelos campos da classe de parâmetro, procurando nomes que correspondam aos critérios dados. Utilizando uma máscara de bits, a implementação filtra os campos estáticos e `transient` e copia os campos escolhidos para a classe sendo gerada. O

modificador de acesso do campo gerado é definido como público, por simplicidade. Depois que a geração for concluída, a representação da classe é retornada para o chamador, que o faz persistir no disco.

Executar `ClassGenerator` em `SimpleClass` produz um arquivo chamado `covertjava.bytecode.SimpleClassValue.class` no diretório atual. A Listagem 17.15 mostra a versão descompilada da classe.

#### **LISTAGEM 17.15** Versão descompilada da classe `SimpleClassValue`

```
package covertjava.bytecode;
import java.io.Serializable;

public class SimpleClassValue
    implements Serializable
{
    public int number;
    public String name;
    public Thread myThread;
}
```

Pronto! Todos os campos apropriados de `SimpleClass` foram gerados para `SimpleClassValue`.

## Biblioteca ASM

Um novo projeto open source que está ganhando ímpeto é a biblioteca de manipulação de bytecode ASM, hospedada em <http://asm.objectweb.org/>. Ela foi projetada com os mesmos objetivos da biblioteca de BCEL, mas tem desempenho significativamente melhor, por causa de uma abordagem de implementação diferente. A BCEL cria uma árvore de objetos completa, que representa um arquivo de classe binário, descendo até as instruções de bytecode individuais. Portanto, pode haver centenas de objetos criados para um arquivo de classe, levando a uma possível degradação de desempenho. Embora seja conveniente ter um objeto para cada atributo de arquivo de classe, essa abordagem pode tornar-se custosa para a manipulação de bytecode em tempo de execução, se milhares de classes forem instrumentadas.

A ASM utiliza um padrão de projeto *visitor* para evitar que sejam instanciados objetos desnecessariamente. Um analisador de classes fornecido pelo framework invoca uma classe visitor definida pelo usuário, passando como parâmetros informações de métodos e campos. Para a maioria dos parâmetros, a implementação do padrão *visitor* simplesmente os passa para o próximo *visitor*, mantendo os dados na forma binária. Para os campos ou métodos que precisam ser alterados, a implementação obtém a representação do objeto a partir do framework e depois manipula esse objeto. Dessa maneira, a maior parte do bytecode permanece em forma binária e o overhead de desempenho é mínimo.

Se for importante ter o mínimo overhead de desempenho com a instrumentação, o ASM é uma escolha melhor que a BCEL. Se a clareza e a simplicidade da implementação forem de prioridade mais alta, recomendo a BCEL.

## Ajuste de bytecode em comparação com proxies dinâmicos e AOP

Agora que aprendeu a ajustar o bytecode, você pode comparar essa técnica com outras abordagens para estender a funcionalidade em tempo de execução. O Capítulo 16 apresentou proxies dinâmicos que permitem interceptar métodos de qualquer interface, sem uma implementação estática dessa interface. Embora os proxies dinâmicos sejam simples de escrever e fáceis de utilizar, sua desvantagem principal é que só funcionam com interfaces (e não com classes); além disso, requerem a instanciação explícita no código chamador. Portanto, para utilizar um proxy dinâmico com o Chat, temos de chamar o método `setMessageListener( )` de `ChatServer` para instalar o proxy. Se não tivéssemos o código-fonte para o Chat, isso não teria sido possível sem fazer uma descompilação. Alterar o código da aplicação é aceitável durante o desenvolvimento, mas não é uma solução adequada para a integração de código de terceiros ou em tempo de execução. Diferentemente do proxy dinâmico, o ajuste de bytecode não requer alterações em tempo de compilação nos códigos sendo ajustados.

AOP, uma tecnologia emergente para adicionar propriedades *cross-sectional* (“ortogonais a funcionalidades”) a objetos e métodos, é um aprimoramento simples e bem estruturado da programação tradicional. Utilizando aspectos, você pode facilmente adicionar funcionalidades como o rastreamento de chamadas de métodos ou pré-processamento e pós-processamento. A AOP separa de maneira limpa a implementação da lógica de programas de tarefas de infra-estrutura, como rastreamento, profiling, segurança e outras. Aspectos são definidos em arquivos separados, que são compilados e processados junto com o código das aplicações. As implementações de AOP utilizam a instrumentação de bytecode para inserir o comportamento adicional. Nisso, elas são mais semelhantes ao ajuste de bytecode que examinamos neste capítulo do que a proxies dinâmicos. A AOP é uma abordagem de alto nível que não possui a flexibilidade oferecida pela engenharia direta de bytecode. Quando apropriados, aspectos podem ser a maneira mais fácil de adicionar lógica de ocultar a uma aplicação existente.

## Questionário rápido

1. Quais são as razões de manipular o bytecode?
2. O que é opcode e como os operandos são passados para uma instrução de bytecode?
3. Como seria um descritor de um método Java chamado `getCount( )`, declarado como `public Object[ ] getCount(String name, char type)`?
4. De que estruturas o arquivo de classe é composto?

5. Quais são as principais classes da BCEL utilizadas para instrumentar ou gerar uma classe?
6. Que atributo de um método precisa ser alterado para instrumentar seu bytecode?

## Resumo

- A manipulação de bytecode é útil para a geração de código, instrumentação de classes existentes e aprimoramento do comportamento de classes, sem alterar seu código-fonte.
- O formato dos arquivos de classes Java e as instruções suportadas são definidos na especificação da JVM.
- A lógica de cada método Java é representada por um conjunto de instruções primitivas da JVM, que são comandos básicos bastante semelhantes a código de máquina.
- O formato binário de um arquivo de classe é representado por pseudo-estruturas definidas na especificação da JVM, que incluem dados na classe, campos, métodos, atributos e outras propriedades.
- A Byte Code Engineering Library (BCEL) da Apache fornece uma API orientada a objetos para trabalhar com as estruturas e campos que compõem uma classe.
- Instrumentar é inserir novo bytecode ou estender o bytecode existente de uma classe.

# 18

## *NESTE CAPÍTULO*

- ▶ Por que e quando aplicar patch a código nativo 168
- ▶ O uso do código nativo na máquina virtual Java 169
- ▶ Abordagens genéricas para o patching de métodos nativos 173
- ▶ Aplicando patch a código nativo na plataforma Windows 174
- ▶ Aplicando patch a código nativo em plataformas Unix 181
- ▶ Questionário rápido 182
- ▶ Resumo 183

# Controle total com patches de código nativo

*"Todo homem tem um plano que não vai funcionar."*

Lei de Howe

## **Por que e quando aplicar patch a código nativo**

Vimos várias técnicas para substituição, patching e engenharia reversa de classes Java. Todas essas técnicas exigem trabalhar no nível de código-fonte ou de bytecode e isso confinou nossas capacidades ao mundo de Java de alto nível. A JVM (Java Virtual Machine) integra-se com o sistema operacional (SO) via bibliotecas nativas, o que significa que todas as operações de baixo nível não são codificadas em Java e, portanto, não podem ser manipuladas pelas técnicas apresentadas. Por exemplo, `System.currentTimeMillis()` é um método nativo, e todos os métodos de `ClassLoader` delegam sua definição na classe para seu método nativo, chamado `defineClass()`. Embora o patching de uma classe Java seja geralmente mais fácil e mais limpo, em alguns casos não resta outra opção senão aplicar um patch ao código nativo. Este capítulo apresenta várias técnicas de baixo nível para o patches de código nativo que, juntamente com as técnicas anteriores, fornecem controle total sobre a JVM.

Gostaria de apresentar dois pontos importantes antes de sujarmos as mãos com o patching nativo. O primeiro tem a ver com a legalidade do trabalho que estamos prestes a realizar. Como discutido anteriormente neste livro, é de sua responsabilidade verificar se a engenharia reversa e o patching não são proibidos por um acordo de licença do produto com o qual você está trabalhando. Além de ser ilegal, roubar propriedade

intelectual de outras pessoas é antiético, por isso encorajo-o veementemente a utilizar as técnicas apresentadas somente por uma boa causa. O segundo ponto é que trabalhar com código nativo exige um sólido conhecimento da linguagem C, um entendimento básico das instruções de máquina e familiaridade com os formatos de arquivos binários. Os arquivos binários têm diferentes formatos em diferentes plataformas e até dois compiladores na mesma plataforma podem produzir arquivos executáveis diferentes. Por exemplo, os arquivos-objeto compilados por um compilador C da Microsoft diferem dos criados por um compilador C da Borland. O patching código binário requer a inserção de instruções de máquina no código de máquina existente, além da manipulação do arquivo binário. Isso é como se aventurar em águas inexploradas, então esteja preparado para lidar com desafios, e não espere que tudo funcione de primeira. A ausência de um formato comum bem-definido e a complexidade de lidar com instruções de máquina brutais resultam na falta de boas ferramentas, como as que tanto nos ajudaram anteriormente. Por exemplo, nenhum descompilador pode produzir código C a partir de um executável binário.

A seguir está uma lista de pré-requisitos para este capítulo:

- Conhecimento da linguagem C.
- Capacidade de escrever e compilar bibliotecas nativas para a plataforma de destino.
- Conhecimento básico de instruções de máquina e de linguagem assembly.
- Alguma familiaridade com a Java Native Interface (JNI).

## O uso do código nativo na máquina virtual Java

A maior parte do código executado dentro da JVM, inclusive as classes básicas, é escrita em Java. Isso faz perfeito sentido, porque código Java é limpo, seguro e independente de plataforma. Entretanto, em algum ponto, a JVM precisa interagir com o hardware; para fazer isso, ela utiliza os recursos do SO. As operações de baixo nível, como a leitura de um bloco de bytes de um disco rígido, ou a criação de um socket de rede, são delegadas a bibliotecas nativas, que fazem chamadas específicas do SO. A Figura 14.1 no Capítulo 14 mostrou em um diagrama simplificado o carregamento de classes e de código nativo pela JVM. Na maior parte do tempo, as bibliotecas nativas simplesmente delegam a chamada para o sistema operacional, de maneira dependente de plataforma. As bibliotecas nativas de Java só podem ser escritas na linguagem C e acessadas via JNI.

### Visão geral de JNI

Para ser compatível com diversas plataformas, Java precisa utilizar uma camada de abstração entre ela mesma e o sistema operacional. Esse nível de abstração é implementado em um conjunto de bibliotecas nativas, que são acessadas pela JNI. A JNI é uma especificação que descreve como definir métodos nativos em Java e como fornecer a implemen-

tação desses métodos em bibliotecas C. Em outras palavras, a JNI fornece um contrato entre classes e bibliotecas nativas de Java.

O lado Java do contrato é simples: para declarar um método nativo, você simplesmente adiciona uma palavra-chave (`native`) à declaração do método e termina a declaração com um ponto-e-vírgula. Vamos supor que um programa Java precise descobrir parâmetros de memória, como a quantidade total e disponível, de memória física e virtual, na máquina local. A classe `java.lang.Runtime` pode fornecer apenas informações sobre os parâmetros de memória da JVM, mas não as propriedades de memória total, portanto, temos de recorrer a fazer uma chamada nativa ao SO. Para isso, escrevemos uma classe Java chamada `OSMemoryInfo`, com um conjunto de métodos nativos. Esta é a declaração do método que retorna a memória física total:

```
public native static long getPhysicalTotal();
```

Depois de o método ser declarado, ele pode ser compilado e utilizado por outras classes Java. Uma tentativa de executar esse método resulta em `java.lang.UnsatisfiedLinkError`, porque nenhuma implementação ainda foi oferecida para `getPhysicalTotal()`. Para executar os métodos nativos, a classe Java que os declara deve carregar uma biblioteca nativa com a implementação dos métodos. Bibliotecas nativas são dependentes do SO, o que significa que uma versão diferente da biblioteca deve ser escrita para cada plataforma em que a aplicação será executada. A biblioteca é carregada apenas pelo nome, porque a extensão é dependente de plataforma. No Windows, nomes de arquivos de bibliotecas terminam com `.dll`; no Unix, com `.so`. A Listagem 18.1 mostra como carregar uma biblioteca chamada `OSMemoryInfo`.

---

#### **LISTAGEM 18.1** Carregando uma biblioteca nativa numa classe Java

---

```
public class OSMemoryInfo {  
    static {  
        try {  
            System.loadLibrary("OSMemoryInfo");  
        } catch (Exception x) {  
            System.err.println("Error while loading native library");  
            x.printStackTrace(System.err);  
            System.exit(1);  
        }  
    }  
    ...  
}
```

---

A biblioteca é carregada por um inicializador estático, que é executado quando a classe é carregada pela primeira vez em uma JVM. Esse passo completa o contrato no lado Java; podemos passar para o lado do código nativo.

Para executar a classe `OSMemoryInfo`, deve ser fornecida à JVM uma biblioteca contendo as implementações de todos os métodos nativos. A localização da biblioteca é

determinada por um caminho de pesquisa (*search path*) específico à plataforma. No Windows, o caminho de pesquisa inclui o diretório atual e os diretórios especificados pela variável de ambiente PATH. No Unix, o caminho de pesquisa é determinado por uma variável de ambiente, cujo nome depende da versão do Unix. Por exemplo, em Solaris o nome desse caminho é LS\_LIBRARY\_PATH, em HP UX, é SH\_LIB\_PATH. O nome da biblioteca nativa também é específico ao SO. No Windows, nossa biblioteca nativa seria chamada OSMemoryInfo.dll; no Unix seria OSMemoryInfo.so. O requisito para a biblioteca é exportar as funções que correspondem ao nome e à sintaxe de declaração dos métodos nativos definidos na classe Java. A JNI especifica o mapeamento de tipos entre tipos C e tipos Java, e fornece extensos mecanismos para acessar objetos Java, lançando exceções e manipulando tipos de dados. Por exemplo, uma função C que implementa o método getPhysicalTotal( ) de Java, mostrado anteriormente, deve ser declarada assim:

```
JNIEXPORT jlong JNICALL  
Java_covertjava_nativecode_OSMemoryInfo_getPhysicalAvail(JNIEnv *, jclass );
```

## Exemplo de implementação com JNI

Aprender com exemplos é a maneira mais eficiente, então vamos trabalhar com a classe OSMemoryInfo apresentada na seção anterior. Lembre que a classe foi projetada para utilizar JNI para obter informações do sistema operacional sobre a memória. Ela possui quatro métodos nativos, retornando as quantidades total e disponível de memória física e virtual. Todos os métodos têm a sintaxe mostrada na Listagem 18.1; a fonte completa da classe está em CovertJava/src/covertjava/nativecode/OSMemoryInfo.java.

A maneira mais fácil de localizar a sintaxe correta para as funções C que correspondem aos métodos nativos de Java é usando o utilitário javah. O javah gera um arquivo de cabeçalho (*header*) C, baseado no arquivo de classe Java fornecido. Para cada método nativo encontrado na classe Java, javah cria uma assinatura da função no arquivo de cabeçalho C gerado como saída. Executar javah na classe covertjava.bytecode.OSMemoryInfo gera um arquivo, covertjava\_nativecode\_OSMemoryInfo.h, que também está no diretório CovertJava/src/covertjava/nativecode. Pare um instante para examinar as declarações da função e como os tipos de dados Java são mapeados para tipos C.

O próximo passo é codificar os corpos das quatro funções declaradas em covertjava\_nativecode\_OSMemoryInfo.h. Para manter o exemplo conciso, veremos somente a implementação para Windows, porque a implementação para Unix difere apenas na chamada de função feita ao SO. Todas as quatro funções utilizam a mesma função da API Win32 – GlobalMemoryStatusEx – que retorna muitas informações sobre a memória do SO. Os corpos da função são codificados em OSMemoryInfo.c, que está no diretório CovertJava/src/covertjava/nativecode. A Listagem 18.2 mostra a implementação de Java\_covertjava\_nativecode\_OSMemoryInfo\_getPhysicalTotal( ).

**LISTAGEM 18.2** Implementação nativa de getPhysicalTotal( )

```

JNIEXPORT jlong JNICALL
Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal
(JNIEnv *env, jclass cls)
{
    MEMORYSTATUSEX memStat;
    memStat.dwLength = sizeof (memStat);
    if (GlobalMemoryStatusEx(&memStat) == 0 && (*env) != 0) {
        jclass exceptionCls = (*env)->FindClass(env, "java/lang/Exception");
        char msg[100];
        sprintf(msg,
                "Failed to get memory information from the OS, error code %li",
                (long)GetLastError());
        if (exceptionCls != 0) /* Levanta exceção Java */
            (*env)->ThrowNew(env, exceptionCls, msg);
        return -1;
    }
    return (jlong) (int) memStat.ullTotalPhys;
}

```

Não há nada complicado aqui – apenas uma chamada a uma função da API Win32, a verificação de um erro e o retorno do resultado. Seguindo o espírito Java, a função C que criamos lança uma `java.lang.Exception` se a chamada à API Win32 falhar.

Com os corpos das funções codificados, podemos construir uma DLL (Dynamically Linked Library) do Windows. Vou utilizar o compilador MSVC, que é distribuído gratuitamente com o Windows SDK e o .Net SDK. O `makefile` que constrói a DLL está no diretório `CovertJava/build` e o arquivo `CovertJava/bin/build_native.bat` pode ser utilizado para rodar `nmake.exe`. Você está livre para escolher o compilador e o método de construção preferido, mas recomendo utilizar o compilador da Microsoft, por motivos explicados mais adiante. Se você quiser reconstruir as bibliotecas nativas, não deixe de atualizar todos os caminhos dentro `build_native.bat`.

Agora podemos executar o método `main()` da classe `OSMemoryInfo`, que gera como saída os valores recebidos dos métodos nativos. Ao executar o arquivo `CovertJava/bin/OsMemoryInfo.bat`, que invoca o método `main()`, é gerada a seguinte saída na minha máquina:

```
C:\Projects\CovertJava\bin>OsMemoryInfo.bat
Total      Physical Memory: 535121920
Available Physical Memory: 199958528
Total      Virtual   Memory: 2147352576
Available Virtual   Memory: 1960931328
```

Agora temos uma implementação JNI funcional, com a qual podemos fazer experiências.

## Abordagens genéricas para o patching de métodos nativos

Conhecendo os princípios de como o código Java interage com código nativo e a arquitetura da JNI, agora podemos mostrar os métodos para redefinir as funções nativas. Assim como com o patching de bytecode, o objetivo é interceptar uma invocação de um método nativo e fornecer nossa própria implementação. O patching deve ser transparente ao chamador, não exigindo alterações no código cliente em Java. Vamos examinar as três abordagens, cada uma tendo suas vantagens e desvantagens.

### Aplicando patch a uma declaração de método Java

A solução mais fácil é aplicar um patch à classe Java que declara o método, removendo a palavra-chave `native` e substituindo-a por uma implementação Java. A implementação pode delegar para uma classe auxiliar que fornece a lógica real do método. Embora seja simples, esse é o método mais eficiente e deve ser sua primeira escolha. Como todas as alterações são feitas no nível Java, você não precisa se aprofundar na programação em C e na manipulação de arquivos binários. Uma complicação nessa abordagem surge quando você quer fazer uma chamada nativa mas precisa alterar alguma parte de sua lógica. Suponha que você tenha um novo requisito: que o usuário do SO seja criado no grupo `Users` em vez de no `Administrators`. Aqui você não vai evitar chamar um método nativo que interage com o SO. Mas, mesmo nesse caso, você pode aplicar um patch ao método original Java para ser não-nativo e então fazê-lo chamar um método nativo. O método nativo é então implementado em uma biblioteca nativa personalizada, com um nome alternativo, que cria um usuário no nível do SO. O único caso em que o patching da declaração não pode ser utilizado é quando um acordo de licença proíbe a engenharia reversa de classes Java, mas não restringe as modificações de bibliotecas nativas.

### Substituindo bibliotecas nativas

A segunda abordagem é substituir a biblioteca nativa original por uma alternativa que exporte as mesmas funções que a biblioteca original. As funções substitutas delegam às funções originais, a menos que uma implementação alternativa seja necessária. A biblioteca substituta atua como um proxy inteligente para a biblioteca original, capaz de pré-processar, pós-processar e redefinir completamente as chamadas de métodos. Essa abordagem funciona bem se a biblioteca tiver poucas funções, ou se forem necessários patches na maioria dos métodos exportados pela biblioteca. Como todo o trabalho pode ser feito em C, essa é uma abordagem relativamente simples, que não requer alterações nas classes Java ou no código de máquina binário. Se for grande o número de funções exportadas, a codificação da biblioteca substituta pode tornar-se cansativa. Assim como com o patching da declaração de métodos Java, um problema pode surgir na tentativa de manter alguma lógica do método nativo original. Isso é semelhante a uma abordagem tudo ou nada – ou você delega ao método original ou não delega.

## Aplicando patch a código nativo

Você se lembra de uma das perguntas que contemplamos no Capítulo 15? A pergunta era, “o que fazemos quando tentamos todos os caminhos, e falhamos?” Não espero que isso seja citado na Internet, mas de certa maneira tem tudo a ver com este livro. As duas abordagens anteriores fornecem soluções relativamente limpas e simples para o patches de código nativo, mas não cumprem a promessa de “controle total”. Para obter o controle total, devemos ser capazes de hackear as bibliotecas nativas e aplicar patch ao código de forma semelhante ao que fizemos com o bytecode. A terceira abordagem faz exatamente isso: Ela utiliza a exploração do formato binário da biblioteca, localizando o código de máquina a ser alterado, e aplica-lhe um patch com a nova lógica. Não é um caminho fácil, por isso recomendo utilizar as duas primeiras abordagens antes de recorrer a esta. O patches de código nativo é específico à plataforma e exige um entendimento completo do formato de arquivos executáveis, além do conhecimento de assembly e de endereçamento de processadores. Mas a recompensa é grande. A técnica que estudaremos aqui pode ser utilizada em qualquer executável, não só em bibliotecas da JNI. Ela também fornece uma idéia sobre os formatos de arquivo executáveis e sobre como o sistema operacional carrega e executa programas. As seções a seguir exploram o patches de código nativo nas plataformas Windows e Unix.

## Aplicando patch a código nativo na plataforma Windows

Entender esta seção requer um conhecimento básico da linguagem assembly e alguma familiaridade com o formato Portable Executable (PE). Hacking e patching são um assunto bastante popular entre jogadores de videogames e alunos de faculdade; isso gerou uma abundância de utilitários que simplificam muito essa tarefa no Windows. Em vez de editar manualmente o código binário e inserir novas instruções de máquina, podemos usar utilitários e bibliotecas para fazer o patching de baixo nível.

### Formato de executável portável

O formato Portable Executable (PE) do Windows é inspirado no Common Object File Format (COFF) do Unix. Ele descreve a estrutura binária de um arquivo executável que pode ser executado em qualquer SO compatível com Win32. Dentre os arquivos executáveis, estão EXE, DLL, SCR, VxD e outros tipos. Estruturalmente, um arquivo PE é muito semelhante a um JAR ou ZIP, que contém outros arquivos ou seções. Um arquivo PE contém um cabeçalho DOS, um cabeçalho PE e uma tabela de seções, seguida por seções que representam vários recursos como textos, dados e recursos de interface gráfica. A Tabela 18.1 mostra a estrutura de um arquivo PE.

**TABELA 18.1****Estrutura de arquivos PE**

<b>ELEMENTO</b>	<b>DESCRIÇÃO</b>
Cabeçalho DOS MZ	Fornecido para compatibilidade retroativa, para que o arquivo seja reconhecido como um executável válido quando executado sob o MS-DOS.
Stub DOS	Pequeno programa embutido que normalmente gera uma linha de texto indicando que o arquivo deve ser executado em um sistema Win32.
Cabeçalho PE	Contém várias informações sobre a parte PE do arquivo, como o número de seções e os endereços de ponto de entrada.
Tabela de seções	Array de estruturas descrevendo cada seção. As estruturas contêm informações como o atributo de seção, o offset de arquivo e o offset virtual.
seção .text	Contém o código binário do programa.
seção .data	Contém dados inicializados.
seção .idata	Contém a tabela de importação.
seção .edata	Contém a tabela de exportação.
Símbolos para depuração	Várias informações de depuração, como números de linhas.

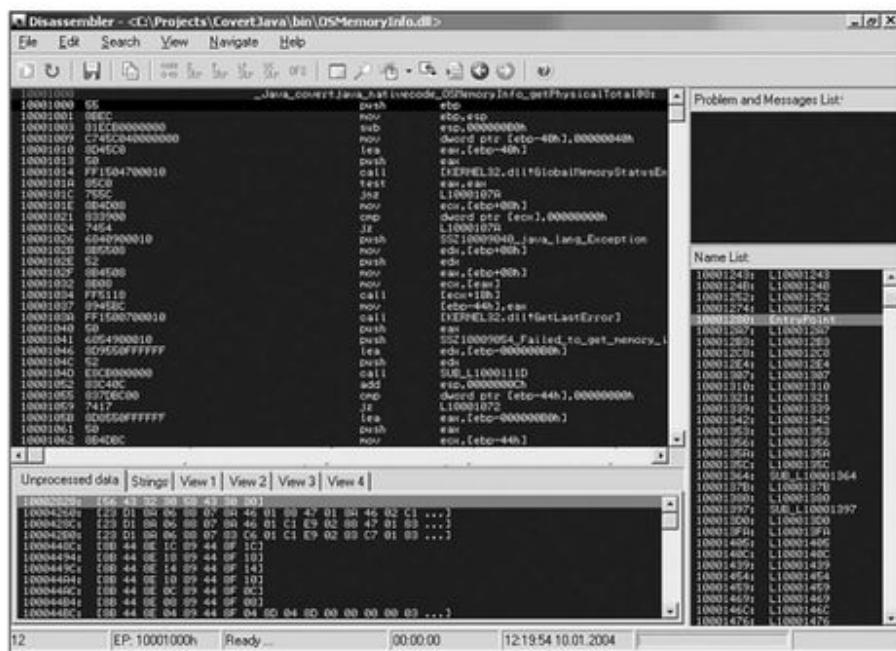
Uma excelente maneira de explorar a estrutura interna de um executável portável é abri-lo no utilitário PE Explorer. Este é um programa shareware bem escrito que exibe em uma janela gráfica cabeçalhos, seções e conteúdos das seções de um PE conhecidas. O PE Explorer também inclui um disassembler que pode ser utilizado para estudar o código de máquina dentro do arquivo. O download do PE Explorer pode ser feito, para avaliação gratuita, a partir de <http://www.heaventools.com>. Por exemplo, carregar o arquivo OSMemoryInfo.dll que criamos anteriormente no PE Explorer permite ver as seções e as exportações dessa DLL. Visualizar as exportações revela que a DLL expõe quatro funções com nomes desfigurados. Podemos ver que `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal` é exportado como `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal@8`. O compilador C acrescentou automaticamente um sinal @ seguido pelo número de bytes que os parâmetros recebem na pilha para todas as funções, seguindo a convenção `_stdcall`.

Como estamos interessados no patching da lógica de funções, precisamos ser capazes de ver seu código de máquina correspondente. Código-fonte C é compilado diretamente para código binário de máquina. Ao contrário do bytecode Java, que precisa ser depois compilado ou interpretado pela JIT, o código de máquina é diretamente executado pelo processador. Uma implicação direta disso é que o executável compilado pode ser executado somente na arquitetura para a qual foi construído. Uma implicação indireta é que não há uma maneira fácil de descompilar o código de máquina de volta ao código-fonte. Os dois são muito diferentes; não há padrão para a maneira de representar construções da linguagem C como instruções de máquina; e cada compilador faz otimizações diferentes, que complicam ainda mais a descompilação. Portanto, a única maneira de fazer a

engenharia reversa dos executáveis binários é trabalhando no nível da linguagem assembly. A linguagem assembly é uma representação, inteligível para seres humanos, das instruções de máquina. É muito primitiva, mas seu código corresponde diretamente à maneira como o processador o executará. Não vamos escrever código assembly, mas, se quiser aprender mais a respeito, escolha um livro da Amazon.com ou simplesmente leia a documentação on-line. Para arquiteturas da Intel, recomendo *Assembly Language for Intel-Based Computers* de Kip R. Irvine (Prentice Hall, ISBN: 0130910139).

Vamos tentar localizar o código da função `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal` dentro do arquivo binário, utilizando o PE Explorer. Se não fez isso ainda, faça download do PE Explorer, instale-o e execute-o; então carregue `OSMemoryInfo.dll` nele. Examine as exportações para ver os nomes das funções expostas pela DLL. Em seguida, execute o Disassembler a partir do menu Tools, com as configurações padrão. Você verá uma tela azul exibindo painéis com várias informações. O painel principal mostra o código desassembled para o ponto de entrada na DLL. Como estamos interessados no código `getPhysicalTotal()`, utilizaremos o recurso de pesquisa para localizá-lo rapidamente. Selecione Find no menu Search e, na caixa de diálogo Find, digite `getPhysicalTotal` no campo de texto. O painel Name List deve destacar um item chamado `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal`, e seu código desassembled deve ser exibido no painel principal, como mostrado na Figura 18.1.

Com um entendimento básico da linguagem assembly, você deve ser capaz de discernir que a função começa salvando o ponteiro de pilha (*stack pointer*) e alocando espaço na pilha para as variáveis locais. Então ela chama `GlobalMemoryStatusEx` do módulo `KERNEL32.dll` e verifica se o valor de retorno, é 0. Se o resultado for 0, a função verifica se o parâmetro `env` de `getPhysicalTotal()` é 0; se não for, ela formata uma mensagem de erro e



**FIGURA 18.1** O PE Explorer mostrando o código desassembled de `getPhysicalTotal()`.

chama uma subrotina para lançar uma exceção. Caso contrário, utiliza o valor de uma estrutura local preenchida por GlobalMemoryStatusEx como o valor de retorno. Em seguida, ela restaura o ponteiro da pilha e retorna. O que vemos é uma correspondência praticamente um-para-um para o código C, porque getPhysicalTotal utiliza apenas operações primitivas, como comparações e chamadas de função. Agora estamos prontos para aplicar um patch a esse código com uma nova lógica.

## Aplicando patch a uma função nativa com o utilitário de substituição de funções

Como indicado anteriormente, o processo de patching de uma função nativa envolve localizar o código binário da função e substituir parte dele pelo novo código, ou por um desvio para o novo código. O desvio pode ser uma instrução assembly JMP simples, para o endereço em que as novas instruções começam, ou uma parte de código que carrega uma biblioteca dinâmica e chama um procedimento dela. O patch deve ser aplicado cuidadosamente, para evitar corromper o estado dos registradores e da pilha de chamadas. Outra questão delicada é o que acontece com o código que foi sobrescrito com o código de desvio. Se você não precisa executar o código original, o código de correção pode ser escrito por cima das instruções originais. Entretanto, se o patching adicionar lógica à lógica original, fazendo pré ou pós-processamento, o código original deve ser realocado para um espaço diferente, antes de ser substituído pelo desvio. Como você pode ver, o patching binário é um processo bastante complexo e frágil, que exige uma análise completa do estado do chamador e do código sendo chamado. Essa é a razão pela qual recomendo aplicar um patch à declaração do método Java ou substituir a biblioteca inteira como primeira opção.

Nenhuma ferramenta confiável poder fazer o patching binário com segurança. O único utilitário razoável que encontrei e utilizei com pouco sucesso (ele não funcionou no JDK 1.4), é um Function Replacer escrito por um membro do grupo de codificação Execution, com o bombástico nome Death. Você pode fazer download dele no site do grupo Execution, hospedado em <http://execution.cjb.net>. A idéia por trás do utilitário se adapta perfeitamente aos nossos requisitos. O Function Replacer substitui uma função exportada de uma DLL Win32 por uma função exportada por outra DLL. A função de substituição deve ter o mesmo número de parâmetros e o mesmo estilo de chamada, para preservar o estado da pilha. Usaremos esse utilitário para aplicar um patch ao método getPhysicalTotal( ) de OSMemoryInfo.dll com um stub de outra DLL, que é escrito diretamente no código para retornar sempre um valor de 10. A Listagem 18.3 mostra o código-fonte do patching.

### LISTAGEM 18.3 Código-fonte do patching getPhysicalTotal( )

```
JNIEXPORT jlong JNICALL Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal
(JNIEnv *env, jclass cls)
{
    return (jlong) (int) 10;
}
```

A DLL com o patching é chamada OSMemoryInfoPatch.dll e foi pré-criada para este livro. Ela pode ser reconstruída utilizando-se o script CovertJava/bin/build\_native.bat, contanto que você instale um compilador C e ajuste o script de build para utilizá-lo. Faça uma cópia de backup do OSMemoryInfo.dll e execute o Function Replacer. Na interface gráfica do Function Replacer, especifique OSMemoryInfo.dll como a DLL a ser corrigida com um patch e selecione Java\_covertjava\_nativecode\_OSMemoryInfo\_getPhysicalTotal@8 (o segundo item na caixa de listagem) como a função a substituir. Especifique OSMemoryInfoPatch.dll como a Replacer DLL e selecione Java\_covertjava\_nativecode\_OSMemoryInfo\_getPhysicalTotal@8 como a função pela qual substituir. Clique no botão Replace Function e certifique-se de que o utilitário não informa nenhum erro. Agora tente executar a aplicação Java e veja se o patch funcionou. Certifique-se de que o JDK atual é 1.2 ou 1.3 e execute CovertJava/bin/OSMemoryInfo.bat. Em minha máquina obtive a seguinte saída:

```
C:\Projects\CovertJava\bin>OsMemoryInfo.bat
Total Physical Memory: 10
Available Physical Memory: 318607360
Total Virtual Memory: 2147352576
Available Virtual Memory: 1992871936
```

Em vez de imprimir 535121920, que é o valor real da memória física total na minha máquina, o método nativo Java agora retorna 10. O patch funcionou, então vamos investigar a mágica por trás dela. O Function Replacer escreve o código de inicialização (*bootstrap*) por cima do código original do método, e inserindo uma chamada ao procedimento de substituição. O código de inicialização escrito no início do código original da função, carrega a DLL de patch utilizando uma chamada de API LoadLibrary( ) e localiza a função de substituição usando GetProcAddress( ). Essa é uma maneira padrão de carregar dinamicamente uma DLL na plataforma Win32. Depois que a função de substituição é localizada, o controle é transferido para ela via uma instrução JMP. O código assembly de inicialização é mostrado na Listagem 18.4.

---

**LISTAGEM 18.4** Código assembly corrigido com um patch de getPhysicalTotal( )

---

```
push    esi
call    osmemory.10001006
pop    esi
sub    esi,401005
lea     eax,dword ptr ds:[esi+40102c]
push    eax
call    dword ptr ds:[<&kernel32.LoadLibraryA>]
push    ebx
lea     ebx,dword ptr ds:[esi+401042]
push    ebx
push    eax
call    dword ptr ds:[<&kernel32.GetProcAddress>]
pop    ebx
```

**LISTAGEM 18.4** Continuação

```
pop    esi  
; OSMemoryInfoPatch._java_covertjava_nativecode_osmemoryinfo_GetPhysicalTotal@8  
jmp    eax  
; Define strings for library and patch function name  
db    ...
```

Como o controle é transferido via uma instrução JMP, o procedimento de substituição retorna diretamente para o chamador, em vez de voltar para o código de inicialização. Analisando o código, podemos entender as limitações do Function Replacer. O tamanho do código de inicialização depende do comprimento da DLL de patch e do nome de função, portanto a abordagem não funciona para funções nativas muito pequenas. Como o código de partida sobrescreve o código original, a função original não pode ser chamada.

Outro problema com o Function Replacer é que ele trava a JVM quando o patching está rodando no JDK 1.4.2. Mesmo que o código assembly seja válido e a DLL corrigida com um patch possa ser carregada por programas C sem problemas, ele parece interferir no estado interno da JVM. O Function Replacer facilita o patching, mas o utilitário não é confiável. Portanto veremos uma abordagem alternativa, de utilizar uma biblioteca poderosa para implementar e instalar o patching manualmente.

## Aplicando patch manualmente utilizando a biblioteca Microsoft Detours

Detours é uma biblioteca da Microsoft para o trabalho com arquivos PE em nível binário e para interceptar funções em tempo de execução. É um framework sólido e bem escrito, que pode ser utilizado em programas C. A seguir são apresentados os principais recursos da biblioteca Detours:

- **Interceptação de funções em tempo de execução** – as funções são interceptadas na memória, em tempo de execução, não em disco. Essa é uma abordagem mais limpa, que também pode ajudar a sobrepujar certas restrições de licenciamento.
- **Invocação da função original** – a Detours preserva o código da função corrigida com um patch. Ao contrário do Function Replacer, a biblioteca Detours salva as instruções de máquina do código original da função em uma entidade chamada *trampoline* (“trampolim”), antes de sobrepor-las com o código de desvio. Isso permite adicionar lógica de pré e pós-processamento em volta da função original.
- **Pequena ocupação de memória do desvio** – o desvio é implementado como um JMP para a lógica do patching; isso requer apenas 4 bytes, portanto, funciona também para funções muito curtas.
- **Edição de tabelas de importação para inserção de DLLs** – o Detours inclui funções para editar a tabela de importação de um executável PE. Isso é útil para in-

serir uma DLL que implementa e instala um patching como um desvio para uma função-alvo. Modificações de importações são salvas em um arquivo no disco.

- **API C limpa e de alto nível** – a biblioteca é bem projetada e relativamente fácil de utilizar. Ela ainda requer um entendimento da arquitetura Win32, mas torna a codificação em assembly desnecessária. O patching e o desvio são codificados como funções C, e a interceptação pode ser instalada com apenas algumas linhas de código.

O download da biblioteca Detours pode ser feito gratuitamente de <http://research.microsoft.com/sn/detours>. Ela vem com uma boa documentação e muitos exemplos, e como este livro é centrado em Java, não vamos perder tempo escrevendo código C. A Listagem 18.5 mostra alguns trechos importantes de um exemplo que aplica um patch a uma função Win32 Sleep e mede o tempo total que um programa gasta em modo de espera.

#### **LISTAGEM 18.5** Passos-chave na utilização da biblioteca Detours

---

```
/* Declara um trampolino Sleep( ) utilizando uma macro do Detours */
DETOUR_TRAMPOLINE(VOID WINAPI UntimedSleep(DWORD dwMilliseconds), Sleep);

/* Ponto de entrada da DLL que instala e remove um desvio para Sleep */
BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    if (dwReason == DLL_PROCESS_ATTACH) {
        printf("slept.dll: Starting.\n");
        Verify((PBYTE)Sleep);
        printf("\n");
        fflush(stdout);
        DetourFunctionWithTrampoline((PBYTE)UntimedSleep, (PBYTE)TimedSleep);
    }
    else if (dwReason == DLL_PROCESS_DETACH) {
        DetourRemove((PBYTE)UntimedSleep, (PBYTE)TimedSleep);
        printf("slept.dll: Removed trampoline, slept %d ticks.\n", dwSlept);
        fflush(stdout);
    }
    return TRUE;
}

/* Isto é um patch para Sleep( ), que mede o tempo total gasto "dormindo" */
VOID WINAPI TimedSleep(DWORD dwMilliseconds)
{
    DWORD dwBeg = GetTickCount();
    UntimedSleep(dwMilliseconds);
    DWORD dwEnd = GetTickCount();

    InterlockedExchangeAdd(&dwSlept, dwEnd - dwBeg);
}
```

---

O código na Listagem 18.5 instala um desvio (patch) chamado `TimedSleep()`, na função `Sleep()`. A função `Sleep()` original ainda pode ser invocada via o trampolino chamado `UntimedSleep()`. Para utilizar a Detours para uma função JNI, uma função de substituição com a mesma assinatura da função-alvo precisa ser escrita e colocada dentro de uma DLL. A função `DllMain()` dessa DLL deve instalar um desvio utilizando `DetourFunctionWithTrampoline()`; então a DLL precisa ser inserida como a primeira importação para a DLL ou o EXE que contém a função JNI sendo corrigida com um patch.

## Aplicando patch a código nativo em plataformas Unix

O patching de binários no mundo Unix é uma tarefa muito mais difícil, se comparada com a plataforma Windows. Como o Unix é uma plataforma diversificada, com múltiplas arquiteturas de hardware e padrões de software, um trabalho de baixo nível, como desassembrar um arquivo executável e editar o código de máquina, requer diferentes implementações para diferentes arquiteturas. Por exemplo, as arquiteturas comuns de processadores para Unix incluem SPARC, utilizada pela Sun Solaris, PA-RISC ou Itanium, utilizadas pela HP UX, R/6000 ou PPC, utilizadas pela IBM AIX, e Intel, utilizada pelo Linux. Cada processador tem um conjunto de instruções diferente, portanto, os arquivos binários não são portáveis entre as arquiteturas. Isso significa que nenhum disassembler único pode converter o código de máquina em assembly em todas as plataformas. Disassemblers gratuitos e comerciais estão disponíveis para cada plataforma, mas a qualidade e a facilidade de utilização variam muito. Um dos melhores utilitários é o IDA Pro (<http://datarescue.com>), que suporta muitos tipos de processadores. Ele só roda no Windows, mas afirma ser capaz de desassembrar os binários da maioria das arquiteturas de hardware comuns.

A situação com padrões de software não é muito melhor. Existem muitos padrões para formatos de arquivos executáveis, sendo o Common Object File Format (COFF) e o Executable and Linking Format (ELF), as duas opções com maior destaque hoje. O formato COFF era tradicionalmente utilizado em sistemas Unix. Ele tem certas limitações e não possui flexibilidade, razão pela qual vem sendo gradualmente substituído pelo formato ELF, mais moderno. Tanto o COFF como o ELF são semelhantes ao formato PE da Microsoft. A Tabela 18.2 mostra uma estrutura de alto nível do formato ELF, da visualização de linking.

O patching de binários requer leitura e gravação. O trabalho com arquivos ELF pode ser simplificado utilizando-se a biblioteca `libelf`. A `libelf` fornece um conjunto de funções C de alto nível que manipulam arquivos executáveis, bibliotecas compartilhadas, arquivos-objeto e outros arquivos que seguem o formato ELF. A `libelf` está disponível para Solaris, HP-UX, AIX e Linux; também é provável que haja versões para outras versões do Unix. Como `libelf` é uma biblioteca de uso geral, ela não fornece funções para o patching que encontramos na biblioteca Detours da Microsoft. A `libelf` oferece uma maneira conveniente de localizar o código a ser corrigido com um patch e de atualizar o arquivo executável com as alterações, mas a tarefa de inserir instruções assembly e possivelmente implementar um *trampoline* tem de ser feita manualmente.

**TABELA 18.2****Estrutura de arquivo ELF**

ELEMENTO	DESCRIÇÃO
Cabeçalho ELF	Contém várias informações sobre o arquivo, como o número de seções e os endereços de ponto de entrada.
Tabela de cabeçalho do programa (opcional)	Fornece a localização e a descrição de segmentos.
Seção 1	Dados específicos da seção 1. Podem ser instruções de máquina, dados, uma tabela de símbolos e assim por diante.
Seção N	Dados específicos da seção N.
Tabela de cabeçalhos de seções	Um array de estruturas que descreve os atributos de cada seção, como o nome, o tipo e o endereço de início da seção, e como as informações devem ser interpretadas.

A abordagem para aplicar patches a bibliotecas compartilhadas Unix que contêm código nativo é idêntica ao trabalho que fizemos no Windows. O código nativo para a função-alvo tem de ser localizado e desassemblado. Em seguida, ele pode ser sobreescrito com o novo código ou com uma instrução JMP para o novo código. A nova lógica também pode ser implementada em uma biblioteca compartilhada, que é dinamicamente carregada pelo patch. Contanto que a assinatura da função e a convenção de chamada sejam as mesmas, a passagem dos parâmetros e o retorno ocorrem corretamente. Para criar o código assembly específico, consulte a documentação do processador de destino.

## Questionário rápido

1. Que papel o JNI desempenha na arquitetura Java?
2. Que passos precisam ser executados para implementar e executar um método nativo?
3. Liste as vantagens e desvantagens de cada uma das três abordagens para o patching de métodos nativos.
4. Que seção do arquivo PE precisa ser acessada para obter o código de máquina?
5. Por que o utilitário Function Replacer não funciona para funções nativas que contenham apenas algumas instruções de código de máquina?
6. Ao implementar um desvio no código assembly, pode-se transferir o controle para o patch via uma CALL em vez de um JMP? Explique por quê.
7. Que vantagens a biblioteca Detours oferece sobre o Function Replacer?
8. Quais são os formatos dominantes para arquivos executáveis no Unix?
9. Como você corrigiria uma função nativa no Unix?

## Resumo

- O patching de código nativo fornece o controle máximo sobre a JVM, porque permite alterar o comportamento no nível mais baixo. Ela utiliza a exploração do formato binário da biblioteca, localizando o código de máquina a ser alterado e aplicar-lhe um patch com a nova lógica.
- A JNI é uma especificação que descreve como definir métodos nativos em Java e como fornecer a implementação desses métodos em bibliotecas nativas.
- Os métodos nativos Java requerem o desenvolvimento de uma biblioteca dinâmica (compartilhada) na linguagem C, que é carregada pela JVM em tempo de execução.
- A abordagem mais fácil para o patching nativo é corrigir a classe Java que declara o método, removendo a palavra-chave `native` e fornecendo uma nova implementação do método, em Java.
- Uma segunda alternativa para o patches de código nativo é substituir uma biblioteca nativa por um proxy de delegação. A biblioteca substituta é implementada em C, sem alterações nas classes Java. A biblioteca original é renomeada, e a nova biblioteca recebe o nome da biblioteca original.
- Na plataforma Windows, um utilitário como o Function Replacer pode ser utilizado para aplicar um patch a uma função exportada por uma DLL com uma função exportada por outra DLL. O Function Replacer é fácil de utilizar, mas tem limitações e problemas de confiabilidade.
- A Microsoft Detours é uma biblioteca para trabalhar com arquivos PE no nível binário e para interceptar funções em tempo de execução. Ela é um framework sólida e bem escrita, que pode ser utilizada em programas C para fazer patching manual.
- Arquivos executáveis no Unix em geral aderem ao formato COFF ou ao ELF. A abordagem geral para o patching bibliotecas Unix é semelhante à do Windows.
- A `libelf` é uma biblioteca comumente utilizada para a manipulação de arquivos executáveis no formato ELF no Unix.

# 19

## *NESTE CAPÍTULO*

- ▶ Definindo objetivos para a proteção das aplicações 184
- ▶ Tornando dados seguros com a Java Cryptography Architecture 185
- ▶ Protegendo a distribuição da aplicação contra hacking 191
- ▶ Implementando licenciamento para desbloquear funcionalidades de aplicações 199
- ▶ Questionário rápido 208
- ▶ Resumo 209

# Protegendo aplicações comerciais contra hacking

*"Murphy era um otimista."*

Postulado de Beck

## **Definindo objetivos para a proteção das aplicações**

Por todos os capítulos deste livro, vimos uma variedade de técnicas para engenharia reversa, hacking, espionagem e cracking. Em muitos dos capítulos eu estava recorrendo à consciência e à boa ética do leitor, para que não abusasse dos direitos intelectuais dos autores de software. Mesmo que a maioria dos usuários tenha valores morais justos, é a minoria que pode causar muitos danos. Este capítulo oferece conselhos práticos sobre como proteger aplicações Java contra hacking e implementar um modelo de distribuição para produtos de software comercial.

Uma aplicação Java típica é oferecida como um pacote integrado (a maior parte do tempo como um arquivo JAR ou ZIP, mas às vezes como um instalador executável), que contém bibliotecas Java e bibliotecas nativas, arquivos de configuração, documentação e vários arquivos de recursos. Hoje é uma prática comum oferecer uma versão do software sem sofisticações, para a utilização pública e gratuita, deixando o conjunto de recursos completo disponível apenas para versões licenciadas. Outra estratégia para atrair potenciais compradores é permitir um período de avaliação com tempo limitado, durante o qual toda a funcionalidade fica disponível. Depois do período de avaliação, os recursos comerciais da aplicação são desativados até

uma licença ser comprada. A Borland utiliza essa estratégia para distribuir o JBuilder X. Inicialmente, ele funciona como uma versão de avaliação chamada Enterprise Edition durante 30 dias e então torna-se o JBuilder X Foundation que tem funcionalidade limitada. Muitos fornecedores de software corporativo oferecem seus produtos gratuitamente para o desenvolvimento, ou contam com a honestidade e o medo de processos judiciais dos usuários para encorajar a compra da licença correta. Independentemente da escolha do modelo de licença e de distribuição, cada fornecedor está interessado vitalmente em cobrar a taxa de licença para gerar faturamento. As técnicas simples demonstradas neste capítulo fornecem uma boa garantia de que o modelo de licença seja seguido.

É importante entender que praticamente não há maneira de obter proteção absoluta contra hackers, especialmente quando o download da aplicação pode ser feito a partir da Internet. Mesmo que o mais forte algoritmo de segurança seja utilizado para produzir e encriptar dados sensíveis como um número serial, um bom hacker com acesso à aplicação pode aplicar um patch ao código para pular toda a verificação. Os capítulos anteriores deste livro mostraram o quanto é fácil localizar e invadir classes Java; até o código nativo pode ser quebrado se houver grandes interesses envolvidos. Portanto, a chave para um mecanismo bem sucedido de proteção é dificultar bastante a invasão para os 95% dos usuários típicos e forçar os 5% restantes dos hackers experientes a gastar uma quantidade significativa de tempo na invasão. Em outras palavras, o objetivo é tornar a licença mais barata do que o tempo que os hackers gastariam para quebrar sua proteção. Outro aspecto vital é dificultar a redistribuição na rede de versões adulteradas por hackers e impossibilitar que estes gerem suas próprias licenças para o produto.

Começaremos examinando os principais aspectos de segurança e criptografia. Utilizando APIs Java de criptografia, você aprenderá, por exemplo, a encriptar e desencriptar informações com cifras, proteger a integridade de dados com um *message digest* ("resumo de mensagem") e a implementar um mecanismo de licença robusto, com pares de chaves assimétricas. Além dessas medidas, examinaremos várias técnicas que protegem os arquivos básicos das aplicações contra hacking e patching.

## Tornando dados seguros com a Java Cryptography Architecture

A palavra *criptografia* é baseada nas palavras *kryptos* (oculto) e *graphein* (escrever), do grego antigo. A *criptografia* fornece um meio de converter informações legíveis, em código incompreensível que pode ser transmitido abertamente e então ser revertido à sua forma original. *Encriptação* é o processo de codificar informações legíveis no código, e *decriptação* é o processo de extrair as informações legíveis a partir do código. Outro serviço de criptografia comumente utilizado é produzir um *hash*, ou um resumo de uma mensagem, para verificar se a mensagem não foi modificada desde que deixou o remetente. Vários algoritmos matemáticos são utilizados para implementar os serviços criptográficos. Os algoritmos podem ser agrupados em três categorias principais, pelo tipo de serviços que fornecem: resumo de mensagem, encriptação/decriptação e assinatura.

Algoritmos de *resumo de mensagem* não modificam o conteúdo da mensagem; em vez disso, produzem um *hash* único baseado no conteúdo da mensagem e em uma chave se-

creta. A chave pode ser qualquer coisa – um número passado como um parâmetro para um algoritmo computacional, uma string de caracteres utilizada como uma senha, ou uma seqüência de bytes. O remetente de uma mensagem sigilosa computa o resumo utilizando uma chave secreta e o envia junto com a mensagem. O receptor utiliza a mesma chave secreta para computar o resumo da mensagem recebida e, se esse resumo não corresponder à mensagem de resumo do remetente, o conteúdo é considerado comprometido. Um terceiro que intercepta a mensagem pode visualizar seu conteúdo e modificá-lo, mas, na ausência da chave secreta, não pode recalcular o resumo. Portanto, a integridade da comunicação é preservada.

Os algoritmos de *encriptação/decriptação* têm o propósito de proteger informações sensíveis passíveis de interceptação por terceiros. O conteúdo da mensagem é modificado utilizando uma chave secreta, produzindo uma saída que é praticamente impossível de ser convertida no conteúdo original. Um terceiro que interceptar a mensagem não poderá decifrar seu conteúdo sem a chave.

Há duas categorias de algoritmos de encriptação/decriptação: de chaves simétricas e de chaves assimétricas. Algoritmos *simétricos* exigem que o remetente e o receptor tenham a mesma chave exata para realizar a encriptação e a decriptação. Os algoritmos simétricos são, às vezes, referidos como algoritmos de *duas vias* porque a mesma chave é utilizada para a encriptação e para a decriptação. A força de proteção obviamente depende da qualidade da proteção das chaves contra o acesso de terceiros. Os algoritmos *assimétricos* utilizam pares de chaves para transformações. Um par de chaves consiste em uma chave pública e uma privada. Esse tipo é referido como algoritmo de *uma via*, porque as informações encriptadas com uma chave pública só podem ser decriptadas com uma chave privada, e as informações encriptadas com a chave privada exigem a chave pública. Geralmente, a chave *pública* é livremente disponível, enquanto a chave *privada* é mantida em segredo pelo proprietário. Para aplicações cliente/servidor e aplicações servidoras, isso fornece mais segurança que os algoritmos simétricos, porque apenas a chave pública precisa estar incluída na distribuição da aplicação cliente.

Um exemplo típico do uso de algoritmo assimétrico é um navegador Web que precisa estabelecer uma comunicação segura com um servidor Web. O navegador recebe a chave pública para codificar as informações enviadas ao servidor Web. O servidor Web decripta as informações com sua chave privada armazenada secretamente, mas, se um terceiro interceptar a mensagem, ele não pode decriptá-la com a chave pública. O servidor utiliza a chave privada para encriptar as informações enviadas para o navegador, portanto, a comunicação inteira fica segura. Os algoritmos simétricos são muito mais rápidos do que os assimétricos, razão pela qual os dois são freqüentemente utilizados em conjunto. Por exemplo, a implementação de SSL estabelece uma sessão utilizando um algoritmo assimétrico. Quando o canal seguro é criado, as chaves simétricas são geradas e trocadas para encriptação dos dados transmitidos.

Assinar refere-se ao processo de gerar uma assinatura digital relativamente curta baseada em dados de tamanho arbitrário, utilizando uma chave privada de um algoritmo assimétrico. A assinatura é produzida pelo remetente e transmitida com a mensagem. O receptor utiliza a chave pública e a assinatura para verificar a integridade da mensagem. Da mesma forma que com resumos de mensagem, a assinatura é matematicamente única para os dados fornecidos, então, se os dados foram modificados, a assinatura

não corresponderá ao conteúdo. A autenticidade é assegurada pelo uso de um algoritmo assimétrico, em que somente o remetente tem a chave privada. Para impedir um terceiro de forjar uma chave pública e afirmar que ela é a chave do remetente, certificados digitais são comumente utilizados. Um *certificado digital* contém a chave pública do remetente, que está assinada por uma chave pública de um Certificate Authority (CA) confiável. Por exemplo, os navegadores são pré-configurados para confiar na Verisign (<http://www.verisign.com>) como uma CA. Uma empresa que deseja permitir que usuários estabeleçam um canal de comunicação seguro com seu servidor Web deve enviar sua chave pública para a Verisign, a fim de obter um certificado digital. Quando o certificado é obtido, ele é instalado no servidor Web para ser passado para o navegador na inicialização de comunicação. O navegador verifica a autenticidade do certificado utilizando a chave pública da Verisign e só estabelece uma conexão segura se a verificação for bem-sucedida.

## Visão geral sobre a arquitetura da criptografia em Java

A Java Cryptography Architecture (JCA) fornece uma implementação completa e robusta de algoritmos e serviços de criptografia. Como a maioria das APIs Java, a JCA fornece interfaces que definem como uma aplicação pode interagir com serviços, de forma independente de fornecedor. A JCE (Java Cryptography Extensions), que já foi um módulo separado, é parte do J2SE desde o JDK 1.4. A JCE vem com um provedor da Sun que implementa os algoritmos mais comumente utilizados, como HmacSHA1, para hashing seguro e DES, para assinaturas e geração de pares de chaves e. Por causa das restrições de exportação do governo dos EUA, alguns algoritmos como RSA não são incluídos no JDK. Além da JCE da Sun, outros pacotes open source excelentes implementam um rico conjunto de algoritmos. A Bouncy Castle (<http://www.bouncycastle.org>) e a Cryptix (<http://www.cryptix.org>) fornecem implementações Java; podem ser baixadas e utilizadas gratuitamente.

As classes JCA básicas estão no pacote javax.crypto; as classes e interfaces para trabalhar com resumos de mensagem ficam no pacote java.security. A home page de segurança Java (em <http://java.sun.com/j2se/1.4.2/docs/guide/security>) é um bom ponto inicial para obter detalhes, focados em Java, sobre vários tópicos de segurança. A home page da JCE em <http://java.sun.com/products/jce/index-14.html> inclui uma visão geral da JCE, e links para páginas relacionadas, tais como o guia de referência. Se você quiser se aprofundar, recomendo comprar um livro sobre segurança em Java, porque o assunto é vasto e interessante. O livro *Java Security Handbook* de Jamie Jaworsky e Paul Perrone (Sams Publishing, ISBN: 0672316021) oferece uma cobertura abrangente de vários tópicos sobre segurança. O foco das próximas seções é na utilização prática da segurança para proteger aplicações Java contra hacking.

## Tornando mensagens de chat seguras com JCA

Mais uma vez, vou utilizar nossa conhecida aplicação Chat para ilustrar os métodos mais úteis de proteger a privacidade dos usuários e a propriedade intelectual do autor. Como o Chat envia mensagens através da rede, a conversa dos usuários está propensa à interceptação.

tação e à espionagem por terceiros. O passo inicial mais óbvio para tornar a aplicação Chat segura é, portanto, a proteção do conteúdo da mensagem transmitida.

Lembre que a aplicação Chat utiliza RMI sobre TCP/IP para trocar mensagens entre instâncias que executam em hosts diferentes. Isso não é tão mal quanto HTML sobre HTTP, porque é muito mais difícil espionar streams TCP/IP binários do que o HTTP, que é textual. Além disso, como você viu no Capítulo 13, com as ferramentas certas, um hacker pode ouvir a conversa e ler o conteúdo da mensagem. A principal razão de o eavesdropping (“espionagem”) ser possível é que as strings dentro dos objetos Java serializados permanecem como texto. Portanto, para tornar as mensagens da aplicação Chat seguras, é necessário encriptar as strings. Quase todo tipo de encriptação vai funcionar para o Chat, porque as mensagens são binárias e, contanto que as strings não sejam reconhecíveis por humanos, elas não vão se destacar no corpo da mensagem (consulte o Capítulo 13). Até mesmo a simples operação de XOR dos caracteres funciona. Na teoria, a maneira mais segura de proteger o canal de comunicação RMI é utilizar factories de sockets personalizados que criam soquetes SSL. Entretanto, como estamos interessados em aprender um método genérico para a proteção de dados, codificaremos com a Java Cryptography API.

A primeira decisão de projeto é escolher o algoritmo. Os algoritmos assimétricos geralmente oferecem melhor proteção porque a chave privada não está disponível para o público em geral. Entretanto, no caso do Chat, um algoritmo assimétrico não é a melhor solução para a encriptação de mensagens. A aplicação Chat instalada em um computador deve ser capaz tanto de encriptar as mensagens que envia como de decriptar as que recebe. Utilizar um algoritmo assimétrico significaria distribuir tanto chaves privadas como públicas com a aplicação Chat. Isso elimina a proteção adicional que se obteria com o algoritmo assimétrico, portanto você deve utilizar a encriptação simétrica, porque ela tem melhor desempenho e é mais fácil de escrever.

A segunda decisão de projeto é escolher o provedor de segurança a ser utilizado. O provedor oferece uma implementação concreta de um algoritmo específico. Para evitar ter de redistribuir bibliotecas adicionais com o Chat, vamos primeiro verificar os algoritmos implementados pela JCE da Sun, porque ela é distribuída junto com a JRE. A JCE da Sun suporta os seguintes algoritmos de cifra: Data Encryption Standard (DES), DESede e PBEWithMD5AndDES. O DES é um padrão largamente utilizado que foi adotado pelo governo dos EUA. Mesmo que existam maneiras conhecidas de quebrá-lo, usando muito poder de processamento, ele fornece proteção adequada para a maioria das aplicações. O DESede, também conhecido como *DES múltiplo*, utiliza múltiplas chaves DES para obter mais força de criptografia. O PBEWithMD5AndDES utiliza uma combinação de algoritmos que inclui uma encriptação baseada em senha definida no padrão PKCS#5, e um resumo de mensagem dos algoritmos MD5 e DES. Por causa da padronização especificada pela JCA, o código cliente que determina esses algoritmos é praticamente independente do algoritmo utilizado. Selecionamos PBEWithMD5AndDES porque, dos três, ele oferece a proteção mais forte.

Uma imagem vale mais que mil palavras e, no mundo da programação, o código-fonte vale mais do que mil imagens. Todo o código-fonte com que vamos trabalhar neste capítulo está localizado no pacote `covertjava.protect`. Vamos começar examinando uma classe que fornece os serviços de encriptação da aplicação Chat. A Listagem 19.1 mostra o construtor de `covertjava.protect.Encryptor`.

**LISTAGEM 19.1** Preparando cifras para encriptação e decriptação

```
import javax.crypto.*;
import javax.crypto.spec.*;

public Encryptor(char[ ] password) throws Exception {
    PBEKeySpec keySpec = new PBEKeySpec(password);
    SecretKeyFactory keyFactory =
        SecretKeyFactory.getInstance("PBEWithMD5AndDES");
    secretKey = keyFactory.generateSecret(keySpec);

    PBEParameterSpec paramSpec = new PBEParameterSpec(this.keyParams, this.iter_count);
    this.encCipher = Cipher.getInstance("PBEWithMD5AndDES");
    this.encCipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);
    this.decCipher = Cipher.getInstance("PBEWithMD5AndDES");
    this.decCipher.init(Cipher.DECRYPT_MODE, secretKey, paramSpec);
}
```

Vamos dissecar o código-fonte e entender o que está sendo feito. O construtor `Encryptor` recebe uma senha como um array de caracteres. O algoritmo `PBEWithMD5AndDES` utiliza três parâmetros: Ele passa o *salt* e a contagem de iterações para inicializar o algoritmo DES e passa a senha utilizada para encriptação com PKCS#5. A classe JCE que representa um algoritmo de encriptação é `javax.crypto.Cipher`. Um programa obtém uma instância de uma cifra chamando o método `Cipher.getInstance()`, que aceita o nome do algoritmo (há um método sobrecarregado que também pode aceitar o nome do provedor).

Os algoritmos de segurança exigem freqüentemente que os parâmetros sejam fornecidos pelo código do cliente. Os parâmetros são utilizados em cálculos matemáticos realizados durante a encriptação e representam uma *semente (seed)* secreta ou uma senha, que é requerida para decriptar os dados posteriormente. Embora a maioria dos algoritmos que requer parâmetros possa utilizar os valores-padrão fornecidos pelo provedor, é altamente recomendado inicializá-los com valores personalizados.

Há duas maneiras de inicializar nossa cifra. Uma é fornecer parâmetros de algoritmo como o *salt* e a contagem de iterações. Outra é fornecer uma chave já gerada. Se fôssemos escolher fornecer uma chave, teríamos de distribuí-la com o Chat, o que torna mais fácil para os hackers extrair a chave. Os parâmetros do algoritmo, que são números comuns, podem ser escritos diretamente no código Java e colocados em classes diferentes. O ofuscamento torna a leitura do código muito difícil, de modo que optaremos por fornecer os parâmetros em vez da chave. A Listagem 19.2 mostra a declaração dos parâmetros do algoritmo dentro da classe `Encryptor`.

**LISTAGEM 19.2** Parâmetros `PBEWithMD5AndDES` utilizados por `Encryptor`

```
public class Encryptor {
    private static byte[ ] keyParams = {
        (byte)0x10, (byte)0x15, (byte)0x01, (byte)0x04,
```

**LISTAGEM 19.2** Continuação

```
(byte)0x55, (byte)0x06, (byte)0x72, (byte)0x01  
};  
private static int iter_count = 20;  
  
...  
}
```

---

Em uma aplicação real, seria melhor colocar os parâmetros em uma classe diferente ou gerá-los utilizando um gerador de números aleatórios com a semente escrita diretamente no código. Isso tornaria o hacking da aplicação mais difícil, mas manteremos as coisas simples. Voltando à Listagem 19.1, você pode ver que o primeiro bloco de código cria uma especificação de chave baseada na senha fornecida. Essa especificação é uma forma intermediária de dados da chave, que são utilizados pela fábrica para gerar uma chave secreta. Como todas as instâncias do Chat utilizam os mesmos parâmetros de algoritmo e a mesma senha, as chaves geradas são idênticas. Isso significa que as mensagens encriptadas por uma instância do Chat podem ser decriptadas por outra instância. Após a chave secreta ser gerada, são obtidas duas instâncias da cifra. Uma delas é inicializada para encriptação; a outra, para decriptação.

Depois do Encryptor ser construído, ele está pronto para realizar a encriptação e decriptação. A maioria dos algoritmos de criptografia lida com bytes brutos. A classe de cifra tem dois métodos – `update( )` e `doFinal( )` – que podem ser utilizados para encriptar um array de bytes. Por exemplo, se tivéssemos uma cifra inicializada chamada `cipher` e um array de bytes chamado `data`, os dados poderiam ser encriptados da seguinte maneira:

```
byte[ ] encryptedData = cipher.doFinal(data);
```

A JCE tem uma classe utilitária chamada `SealedObject`, que contorna qualquer objeto serializável e utiliza uma cifra fornecida para encriptar ou decriptar o objeto empacotado durante a serialização. Como a aplicação Chat envia mensagens como objetos, `SealedObject` é uma escolha melhor do que bytes brutos, pois fornece uma API de mais alto nível para a encriptação. Os dois métodos fornecidos por `Encryptor` para encriptar e decriptar instâncias de `java.io.Serializable` são mostrados na Listagem 19.3.

**LISTAGEM 19.3** Métodos que implementam a encriptação e decriptação

```
public Serializable encryptObject(Serializable object) throws Exception {  
    return new SealedObject(object, this.encCipher);  
}  
  
public Object decryptObject(Serializable object) throws Exception {  
    return ((SealedObject)object).getObject(this.decCipher);  
}
```

---

Como você pode ver, SealedObject torna a implementação trivial. A classe Encryptor que discutimos pode ser utilizada agora na aplicação Chat. Em vez de passar instâncias de covertjava.chat.MessageInfo de um pro outro, o Chat chamará Encryptor para obter a versão encriptada da mensagem antes de enviá-la. Quando uma nova mensagem for recebida, o Chat utilizará Encryptor para extrair o objeto MessageInfo a partir do objeto selado recebido. Esse código teria de ser colocado nos métodos sendMessage( ) e receiveMessage( ) do ChatServer, mas não vamos fazê-lo pois queremos economizar tempo e espaço. O método main( ) na classe Encryptor mostra um auto-teste que grava e lê uma string em um arquivo.

## Protegendo a distribuição da aplicação contra hacking

Encriptar os dados transmitidos protege as informações contra a espionagem no nível de protocolo. Isso protege as informações do usuário, mas não a propriedade intelectual no software, como algoritmos, padrões de projeto e código. Muitas técnicas de engenharia reversa apresentadas neste livro podem ser facilmente utilizadas para quebrar um produto comercial e desbloquear a funcionalidade que, de outro modo, exigiria a compra de uma licença. Para licenças que são emitidas com base no número de hosts em que o software é instalado, outra ameaça potencial pode vir de uma organização aétnica que compra a licença mais barata para um host e então a replica para um grande número de hosts. Esta seção discute várias técnicas que protegem a distribuição da aplicação contra hacking e assegura que os valores sejam pagos de acordo com o modelo de licenciamento.

### Protegendo bytecode contra descompilação

O Capítulo 2, mostrou como você pode obter facilmente o código-fonte a partir do bytecode Java e que, na maioria dos casos, o código descompilado é praticamente uma correspondência um-para-um com o código-fonte original. O Capítulo 3 forneceu detalhes sobre como o bytecode pode ser protegido da descompilação. Deve ter ficado óbvio que a força da proteção total seja tão forte quanto o código que a implementa. Você pode utilizar o algoritmo mais forte para encriptar os dados, mas se o código pode ser descompilado e corrigido um patch em meia hora, o código de encriptação pode ser simplesmente desativado colocando-o como comentários.

Ofuscamento, ofuscamento, ofuscamento. Esta é a única maneira confiável de proteger o bytecode e, portanto, a propriedade intelectual de uma aplicação. O ofuscamento de fluxo de controle, discutido no Capítulo 3, é crucial para alcançar os melhores resultados. A contramedida mais poderosa contra a descompilação de bytecode é compilar a aplicação Java em um executável nativo. Examinamos a complexidade de fazer engenharia reversa e o patching do código nativo e, independentemente da qualidade do ofuscador de bytecode, o código nativo é muito mais difícil de quebrar. Infelizmente, agora a maioria dos fornecedores que oferecia Java para compiladores de código nativo

faliu ou parou de suportar ativamente seus produtos. As melhorias nos JITs e as crescentes velocidades de processamento fornecem desempenho suficiente para aplicações Java, eliminando a necessidade de compilar em código nativo. As empresas TowerJ e Excelsior têm provavelmente as melhores implementações para Windows, mas aconselho precaução e testes completos das aplicações compiladas, para assegurar que todos os recursos estão funcionando adequadamente. Para a maioria das aplicações Java, é provável que utilizar um ofuscador agressivo como o Zelix KlassMaster seja uma escolha melhor do que compilar o código em um binário nativo.

## Protegendo bytecode contra hacking

Independentemente da qualidade do trabalho realizado pelo ofuscador, o bytecode ainda pode ser descompilado. E, se pode ser descompilado, pode também ser modificado e a aplicação pode ser corrigida com um patch. Para fortalecer a proteção, revisaremos algumas idéias sobre verificações de segurança que protegem as classes contra patching.

Por todo o livro, desenvolvemos várias técnicas para hacking e patching. Discutimos como descompilar e então aplicar patches a classes inteiras, formas de acessar métodos protegidos e privados e como trabalhar com classpath de inicialização de sistema. Utilizadas pelas razões erradas, essas técnicas podem prejudicar a propriedade intelectual dentro de uma aplicação Java. Aqui examinamos as técnicas que tornam o hacking muito mais difícil, e as medidas de defesa contra cada técnica.

### Hackear métodos não-públicos e variáveis de uma classe (Capítulo 4)

A solução mais fácil para isso é selar a aplicação JAR. Selar um JAR garante que todas as classes em um pacote venham do mesmo código-fonte. Isso significa que um hacker não pode colocar classes personalizadas nos pacotes fornecidos pelo JAR. A selagem de JARs é feita adicionando a seguinte linha ao arquivo de manifesto:

```
Sealed: true
```

O próprio JAR precisa ser protegido contra modificações. Assim como você pode facilmente selá-lo, um hacker pode facilmente remover esse selo. Tudo que o hacker teria de fazer é descompactar o conteúdo do arquivo JAR em um diretório temporário, remover o atributo Sealed e montar o JAR novamente. O Java suporta a noção de JAR assinado, que pode proteger seu conteúdo de modificações atribuindo a cada classe uma assinatura digital. Isso funciona bem para applets assinados cujo download e verificação são feitos pelo navegador. O problema é que o próprio JAR assinado não está protegido, então mais uma vez um hacker pode descompactar o arquivo JAR, remover o manifesto com as assinaturas digitais e recriar o JAR. Mesmo que o JAR não fosse mais considerado autêntico e procedente de seu verdadeiro fornecedor, ele poderia ser executado e utilizado com facilidade. Portanto, você precisa de uma maneira de assegurar que o conteúdo da distribuição da aplicação não seja modificado; trataremos disso na seção a seguir.

### Substituindo e aplicando patches a classes de aplicações (Capítulo 5)

Selar um JAR também é uma defesa contra essa técnica de hacking. Para obter mais proteção, você pode adicionar uma verificação que assegure que uma classe de fato seja carregada a partir do JAR de distribuição da aplicação e não de um JAR de terceiros. A implementação desse método simples é fornecida em `covertjava.protect.IntegrityProtector`. A Listagem 19.4 mostra o código-fonte de `assertClassSource( )`.

#### LISTAGEM 19.4 Assegurando o JAR fonte da classe

---

```
public void assertClassSource(Class cls, String jarName) {
    // O class loader não deve ser nulo
    if (cls.getClassLoader() == null)
        throw new InternalError(BOOT_CLASSLOADER);

    String name = cls.getName();
    int lastDot = name.lastIndexOf('.');
    if (lastDot != -1)
        name = name.substring(lastDot + 1);
    URL url = cls.getResource(name + ".class");
    if (url == null)
        throw new InternalError(FAILED_TO_GET_URL);

    name = url.toString();
    if (name.startsWith("jar:") == false || name.indexOf(jarName + "!") == -1)
        throw new InternalError(UNEXPECTED_JAR);
}
```

---

A primeira instrução `if` assegura que o class loader de classes dado não é o carregador de inicialização, comparando-o com um `null`. Essa afirmação pode ser feita porque você sabe que nenhuma das classes da aplicação Chat é colocada no classpath de inicialização, o que significa que as classes devem ser carregadas com o class loader do launcher padrão da aplicação.

O resto do código do método obtém um URL para a fonte do arquivo CLASS utilizado para criar a classe dada. Este é o URL retornado por `Class.getResource( )` para a classe `MessageInfo`:

```
jar:file:/C:/Projects/CovertJava/distrib/lib/chat.jar!/covertjava/chat/MessageInfo.class
```

O URL indica que a classe foi carregada do arquivo `chat.jar` localizado no diretório `C:/Projects/CovertJava/distrib/lib`. Depois de obter o URL, `assertClassSource( )` assegura que ela se inicia com `jar:` e que contém o nome do arquivo JAR passado como parâmetro do método. Um `InternalError` é lançado para abortar a execução se a afirmação apresentar falha. Essa pode não ser uma verificação à prova de falhas, mas deve ser suficientemente útil para resistir à maioria das tentativas no patching. Para tirar proveito dessa proteção, o Chat deve invocar `assertClassSource( )` nas classes principais, que são os principais can-

didatos para o patching. Adicionaremos uma nova classe, `ProtectedChatApplication`, como ponto de entrada alternativo para a aplicação Chat. `ProtectedChat` estenderá `covertjava.chat.ChatApplication` e utilizará vários mecanismos de proteção desenvolvidos neste capítulo. O código na Listagem 19.5 mostra uma parte do método `main( )` de `ProtectedChatApplication` que assegura a origem da classe `LicenseManager`.

#### **LISTAGEM 19.5 Assegurando a origem de LicenseManager**

---

```
public static void main(String[ ] args) throws Exception {
    LicenseManager licenseManager = new LicenseManager("conf/chat.license");
    IntegrityProtector protector = new IntegrityProtector( );
    protector.assertClassSource(licenseManager.getClass( ), "/lib/chat.jar");
    ...
}
```

---

O método `main( )` assegura que a classe `LicenseManager` seja carregada a partir do arquivo `chat.jar` localizado no subdiretório `lib`. Devemos inserir verificações como essas em outras classes do Chat. Quanto mais verificações utilizarmos, mais trabalho um hacker terá para quebrar a aplicação.

#### **Manipulando a segurança Java (Capítulo 7)**

Manipular a segurança Java permite que os hackers obtenham acesso a pacotes protegidos e a membros privados de uma classe, e pulem outras verificações de segurança normalmente impostas por um gerenciador de segurança. Se uma aplicação instala um gerenciador de segurança ou utiliza um arquivo de políticas personalizado, você deve inserir verificações que assegurem que o gerenciador está instalado e que o arquivo de políticas correto seja utilizado. O gerenciador de segurança pode ser obtido com `System.getSecurityManager( )`; para verificar o arquivo de políticas original, você pode checar o valor da propriedade de sistema `java.security.policy`. O próprio arquivo de políticas pode ser protegido contra modificações, com a técnica de proteção de conteúdo de aplicações descrita mais adiante neste capítulo.

#### **Aplicações de engenharia reversa (Capítulo 12)**

Dependendo do tipo de recurso, a proteção requer a proteção de bytecode, ou a proteção do conteúdo da aplicação. Recursos como strings de itens de menu e mensagens de erro são, com freqüência, escritos diretamente no bytecode; já recursos como imagens e arquivos de mídia em geral são armazenados em um diretório separado ou dentro de um JAR. A proteção do bytecode foi revisada anteriormente; a proteção do conteúdo da aplicação é apresentada na próxima seção.

#### **Controlando o carregamento de classes (Capítulo 14)**

Class loaders personalizados fornecem muito poder porque podem manipular o bytecode em tempo real. Certamente, não é uma técnica comum para hackear aplicações, mas,

se você quiser proteger uma classe de aplicação contra manipulações de bytecode em tempo de execução, você pode instalar e utilizar um class loader personalizado predefinido, em vez do class loader de sistema. Então, em vários lugares do código da aplicação, pode-se verificar se uma classe foi carregada com o class loader esperado.

### Entendendo e ajustando bytecode (Capítulo 17)

O ajuste de bytecode requer um class loader personalizado, que realiza o ajuste em tempo real, ou faz modificações estáticas nos arquivos CLASS da aplicação. Discutimos como impedir o uso de um class loader de terceiros no parágrafo anterior; a próxima seção descreve como proteger os arquivos de distribuição da aplicação.

## Protegendo o conteúdo da aplicação contra hacking

O *conteúdo da aplicação* aqui se refere aos arquivos distribuídos com a aplicação. Isso inclui as bibliotecas, como arquivos JAR e ZIP, imagens, arquivos de configuração e outro conteúdo. Assegurar que os arquivos principais da aplicação não sejam modificados é crucial para a proteção de sua integridade, porque a maioria das técnicas de hacking exige a alteração de algum arquivo. O tipo de arquivo mais importante que precisa de proteção de integridade é o JAR. Já vimos um truque que pode assegurar que a classe seja carregada a partir do JAR esperado. Agora vamos desenvolver uma classe que permite que uma aplicação assegure que nenhum de seus arquivos foi mexido.

A maneira mais simples e direta de verificar a integridade do conteúdo da aplicação é iterar pelos arquivos da distribuição verificando atributos como tamanho e hora de modificação. Para o máximo de proteção, a aplicação pode produzir um checksum do conteúdo do arquivo e confrontá-lo com o checksum dos arquivos originais recebidos no momento de criação da distribuição. Vamos desenvolver uma classe chamada `IntegrityProtector` no pacote `covertjava.protect` e a utilizaremos para proteger nossa aplicação Chat. Para manter o exemplo conciso, limitaremos a verificação a tamanhos de arquivos, embora ela possa ser facilmente estendida para incluir outros atributos de arquivos, além do hash do conteúdo. `IntegrityProtector` itera por uma lista dos principais arquivos da aplicação, gera um tamanho total de arquivos em bytes e depois calcula um checksum utilizando o algoritmo de resumo de mensagem. Vamos então adicionar ao Chat um arquivo de configuração que armazena a versão e o resumo de mensagem (*digest*) da distribuição da aplicação. Armazenar o resumo de mensagem em vez do comprimento total dos arquivos torna o hacking muito mais difícil. Por fim, na inicialização da aplicação Chat vamos usar `IntegrityProtector` para assegurar que o checksum atual dos arquivos da distribuição corresponde ao checksum original fornecido no arquivo de configuração.

Nossa primeira tarefa é decidir quais arquivos na aplicação Chat devem ser protegidos contra modificações. Não podemos simplesmente incluir todos os arquivos, porque alguns arquivos são feitos para serem alterados pelo usuário final. Por exemplo, `bin/setenv.bat` pode ser modificado para fornecer um diretório home específico para o Chat, ou para executá-lo em uma porta diferente. `conf/log4j.properties` pode sofrer alterações se um usuário ajustar os níveis de logging. Entretanto, arquivos como `lib/chat.jar` e `conf/java.policy` nunca devem diferir das versões originais (a menos que

desejássemos enviar um patching aos clientes; nesse caso os novos checksums podem ser fornecidos com o patching). Nesse exemplo, protegeremos apenas os arquivos básicos da distribuição do Chat:

```
conf\java.policy  
lib\chat.jar  
lib\log4j-1.2.8.jar
```

Para flexibilidade, vamos ler a lista de arquivos básicos de um arquivo de configuração chamado `Chatfilelist.class`. Para confundir os hackers, fornecemos ao arquivo de lista uma extensão `.class`, embora seu conteúdo seja textual. Durante o desenvolvimento, esse arquivo será mantido no diretório `CovertJava/conf`, mas modificaremos o arquivo `build.xml` para copiar `ChatFileList.class` no diretório `covertjava/protect`, junto com as classes do pacote `covertjava.protect`. A Listagem 19.6 mostra a tarefa `<copy>` que foi adicionada ao `build.xml`.

#### **LISTAGEM 19.6** Copiando a lista de arquivos para o diretório da distribuição

---

```
<copy todir="classes/covertjava/protect">  
    <fileset dir="conf">  
        <include name="ChatFileList.class"/>  
    </fileset>  
</copy>
```

---

Vale a pena o esforço de gastar mais 15 minutos para disfarçar a lista de arquivo como um arquivo de classe comum, porque isso torna um método de proteção trivial menos óbvio para um hacker. Agora podemos prosseguir com o desenvolvimento da classe `IntegrityProtector`, no pacote `covertjava.protect`. Primeiro, devemos escrever alguns métodos auxiliares, que leem o conteúdo de um arquivo-texto (como `ChatFileList.class` de Chat) e fazem o parse desse conteúdo, gerando um array de strings. Se você abrir o arquivo `IntegrityProtector.java` no diretório `src/covertjava/protect`, verá as implementações dos métodos auxiliares: `readFilePathsFromResource()`, `readFilePathsFromFile()` e `readFilePathsFromString()`. Agora podemos codificar um método que produz um checksum para uma lista de caminhos (*paths*) de arquivos. Em seguida, adicionamos o método `getFilesCheckSum()` ao `IntegrityProtector` e o implementamos como mostrado na Listagem 19.7.

#### **LISTAGEM 19.7** Calculando um checksum para uma lista de arquivos

---

```
public String getFilesCheckSum(String[ ] paths, char separator,  
                               String installPath) throws Exception {  
    long totalSize = 0;  
    for (int i = 0; i < paths.length; i++) {  
        String path = paths[i];  
        if (separator != File.separatorChar)  
            path = path.replace(separator, File.separatorChar);
```

**LISTAGEM 19.7** Continuação

---

```

        path = installPath + File.separatorChar + path;
        totalSize += new File(path).length( );
    }

    byte[ ] checkSum = toByteArray(totalSize);
    MessageDigest sha = MessageDigest.getInstance("SHA-1");
    checkSum = sha.digest(checkSum);
    BASE64Encoder encoder = new BASE64Encoder( );
    return encoder.encode(checkSum);
}

```

---

O método itera pelo array de nomes de arquivos, adicionando o tamanho de cada arquivo em bytes ao tamanho total. Depois de o tamanho total ser calculado, ele é convertido em um array de bytes que utiliza um método auxiliar chamado `toByteArray()`. `getFilesChecksum()` então obtém uma instância de um algoritmo de resumo de mensagem SHA-1 (Secure Hash Algorithm, fornecido pelo JCE da Sun) e obtém um hash do tamanho total. Como o checksum tem de ser armazenado em um arquivo-texto, precisamos converter os bytes em caracteres ASCII inteligíveis para humanos. Não podemos simplesmente fazer o cast de variáveis `byte` para `char`, porque isso produz caracteres não imprimíveis (por exemplo, o valor de byte 7 produziria um “bip” e o valor de byte 8 produziria um backspace). A solução padrão para esse problema é a *codificação base64*. A codificação base64 utiliza um subconjunto do código ASCII que contém apenas 64 caracteres imprimíveis. Esse subconjunto inclui caracteres de A-Z e de a-z, numerais 0-9 e alguns outros caracteres seguros, como sinais de pontuação. Como menos caracteres são utilizados, a base64 aloca 6 bits por caractere em vez dos 8 bits utilizados para caracteres ASCII. Conseqüentemente, 3 bytes de dados de entrada são codificados em 4 bytes de dados de saída. `IntegrityProtector` utiliza a classe `Base64Encoder`, localizada no pacote `sun.misc`, para obter uma representação imprimível do checksum de um arquivo.

Agora que somos capazes de obter o checksum do arquivo, vamos usá-lo para verificar a integridade da instalação da aplicação Chat. Codificaremos o método `main()` do `IntegrityProtector` para gerar como saída o checksum para uma dada lista de arquivos. A Listagem 19.8 mostra o corpo do método `main()`.

**LISTAGEM 19.8** Gerando como saída o checksum de um arquivo

---

```

public static void main(String[ ] args) throws Exception {
    if (args.length != 1) {
        System.out.println("Syntax: IntegrityProtector " +
            "[-Dhome=<path to home>] <file list path>");
        System.exit(1);
    }
}

```

**LISTAGEM 19.8** Continuação

---

```

IntegrityProtector protector = new IntegrityProtector();
String[ ] paths = protector.readFilePathsFromFile(args[0]);

String homePath = System.getProperty("home", ".");
String checksum = protector.getFilesCheckSum(paths, '\\', homePath);
System.out.println("Checksum = [" + checksum + "]");
}

```

---

O método `main()` recebe um parâmetro que especifica o arquivo com a lista (`conf/ChatFia 1elist.classe`, em nosso caso), e um parâmetro opcional que fornece o diretório `home` dos arquivos (`distrib` em nosso caso). Por conveniência, incluímos um arquivo batch `getChatChecksum.bat` no diretório `CovertJava/bin`, que utiliza `IntegrityProtector` para gerar como saída o checksum da aplicação Chat. Executar `getChatChecksum.bat` depois de construir a distribuição do Chat com a tarefa de release do Ant produz a seguinte saída:

`Checksum = [gLmBOKQe88gLrC9vaSjBarf2Rfw=]`

Toda vez que um arquivo básico do Chat mudar (por exemplo, quando você reconstruir `lib/chat.jar`), o checksum será diferente. Mas se os tamanhos de arquivo não mudarem, o checksum permanece o mesmo, o que potencialmente abre uma brecha na proteção. Essa é a razão pela qual é mais seguro obter um checksum baseado no conteúdo real dos arquivos; entretanto, a maioria dos hackers não vai se preocupar de manter o tamanho do arquivo inalterado, então até o nosso mecanismo simples funcionaria para a aplicação Chat.

Agora podemos adicionar um arquivo de configuração chamado `chat.properties` ao diretório `conf` do Chat. Dentro desse arquivo, armazenaremos o checksum da distribuição do Chat como o valor da propriedade `chat.versionInfo`. Mais uma vez, evitamos utilizar um nome intuitivo para a propriedade, a fim de dificultar ainda mais o hacking. Nossa tarefa final é assegurar que, no início, a aplicação Chat compare o checksum atual de seus arquivos com o checksum lido do arquivo de configuração. A parte do método `main()` do `ProtectedChatApplication` que faz isso é mostrada na Listagem 19.9.

**LISTAGEM 19.9** Confrontando o checksum atual com o checksum da distribuição

---

```

public static void main(String[ ] args) throws Exception {
    String homePath = System.getProperty("chat.home");
    String propPath = homePath + File.separator + "conf" +
                      File.separator + "chat.properties";
    AppProperties props = new AppProperties(propPath);
    String checkSum = props.getProperty("chat.versionInfo");
    IntegrityProtector protector = new IntegrityProtector();
    String[ ] paths = protector.readFilePathsFromResource("ChatFileList.class");
    protector.assertFilesIntegrity(paths, '\\', homePath, checkSum);
}

```

---

Depois de ler o checksum original e a lista de arquivos protegidos, o método utiliza `assertFilesIntegrity( )`, de `IntegrityProtector`, para assegurar a integridade do conteúdo. `assertFilesIntegrity( )`, mostrado na Listagem 19.10, simplesmente invoca `getFilesChecksum( )` para lista de arquivos fornecida e lança um `InternalError` se o checksum calculado não corresponder ao checksum original.

---

**LISTAGEM 19.10** Implementação de `IntegrityProtector.assertFilesIntegrity( )`

```
public void assertFilesIntegrity(String[ ] paths, char separator,
    String installPath, String checkSum) throws Exception {
    String installCheckSum = getFilesChecksum(paths, separator, installPath);
    if (installCheckSum.equals(checkSum) == false)
        throw new InternalError("Some of the installation files are corrupt");
}
```

---

Com toda a codificação concluída, podemos testar nossa proteção. Os arquivos de Chat fornecidos neste livro são distribuídos com o checksum correto. Você deve ser capaz de executar o Chat utilizando `chat_protected.bat` do diretório `distrib/bin`. Verifique se você pode abrir janela Chat principal na sua máquina. Agora vamos fingir que estamos hackeando o Chat modificando `java.policy` no diretório `distrib/conf`. Abra esse arquivo, adicione uma nova linha e salve-o. Certifique-se de que o comprimento do arquivo foi alterado e tente executar `chat_protected.bat` novamente. Você deve ver a seguinte exceção:

```
Exception in thread "main" java.lang.InternalError:
Some of the installation files are corrupt
at covertjava.protect.IntegrityProtector.assertFilesIntegrity(...)
at covertjava.protect.ProtectedChatApplication.main(...)
```

Como o comprimento do arquivo mudou, o checksum calculado não corresponde mais ao checksum original e `IntegrityProtector` lança um erro. Se fôssemos distribuir a implementação protegida de Chat, não iríamos, naturalmente, distribuir o arquivo `getChatChecksum.bat` e removeríamos `distrib/bin/chat.bat` junto com o método `main( )` de `ChatApplication`. Isso asseguraria que a única maneira de carregar Chat é pela classe `ProtectedChatApplication`. Para permitir ao Chat protegido ser executado com a nova versão de arquivos, teríamos de obter o novo o checksum e defini-lo com o valor da propriedade `chat.versionInfo` no arquivo `chat.properties`.

## Implementando licenciamento para desbloquear funcionalidades de aplicações

Esta seção examina a maneira como as aplicações são licenciadas hoje e discute como desenvolver um framework de licenciamento para uma aplicação distribuída comercialmente.

## Modelos modernos de licenciamento de software

Vários modelos populares de licenciamento controlam a distribuição de software moderna. Em geral, os termos de distribuição e de utilização são escritos no EULA (*User License Agreement* –Contrato de Licença do Usuário Final) que vem com o produto. Embora cada fornecedor tenha a escolha de escrever os termos de licença, os modelos de licença podem ser agrupados nas três categorias a seguir.

### Software comercial de código-fonte fechado

Esse é o modelo tradicional para distribuição de software comercial. Ele inclui produtos proprietários como o Microsoft Windows; software que pode ser baixado para avaliação gratuita, como o IntelliJ IDEA; e produtos que têm uma edição gratuita com funcionalidade limitada, como JBuilder da Borland e o WebLogic da BEA. Oferecer uma edição gratuita com funcionalidade limitada está se tornando cada vez mais popular entre fornecedores Java, porque os desenvolvedores gostam de conhecer bem um produto antes de se decidir pela compra. Quando um produto é bem escrito, os usuários se acostumam a ele e ao final freqüentemente decidem comprar a versão com funcionalidade integral.

### Software comercial open source (de código-fonte aberto)

Esse modelo de licença emergente está ganhando popularidade; permite aos usuários finais não só fazerem o download e utilizar o software, mas também obterem o código-fonte. Em geral, os termos de utilização permitem o desenvolvimento e a implantação gratuitos, mas podem requerer taxas de documentação, suporte técnico ou recursos avançados. Exemplos de destaque nessa categoria são o servidor de aplicações JBoss e o banco de dados MySQL.

### Software livre open source (de código fonte aberto)

O software nesta categoria está disponível gratuitamente ao público sem nenhuma restrição. A licença mais comum utilizada para software gratuito open source é a GPL (General Public License), que permite o uso do software e do seu código-fonte comercialmente ou não-comercialmente. Há variações da GPL e de outras licenças de open source, como a Licença Apache, que podem impor certas restrições na utilização do software, mas todos enfatizam a idéia básica de que o software é livre para todos.

## Implementando licenciamento para desbloquear recursos comerciais

Quando o código-fonte é fornecido com um produto, é óbvio que ter restrições via programação para impor o modelo de licença não faz sentido. Qualquer usuário pode remover facilmente as restrições de funcionalidade modificando o código-fonte e reconstruindo o produto. Entretanto, a maioria dos produtos hoje não vem com o código-fonte, então a imposição via programação da política de licenciamento pode ajudar a gerar

vendas. Vamos desenvolver uma classe `LicenseManager` que produz números seriais seguros baseados no ambiente do cliente e no tipo de licença. A classe vai usar um algoritmo assimétrico para assegurar que apenas o fornecedor do software possa emitir os números seriais. Mesmo se não precisar implementar o gerenciamento de licenças, você vai se beneficiar da leitura desta seção, pois ela demonstra uma utilização prática da segurança Java e de APIs de criptografia.

Decidir sobre que plano de licenciamento utilizar requer a consideração da maneira mais efetiva de impedir o abuso da política de licenciamento, sem sacrificar a experiência do cliente com o produto. Por exemplo, é perigoso emitir uma licença que desbloqueia os recursos do produto comercial sem vinculá-la ao ambiente do usuário final. Este tipo de licença pode ser mais fácil de emitir, porque não exige que o usuário envie as informações para o fornecedor do produto, mas pode transformar a distribuição em um pesadelo, se alguém colocar a licença na Internet. Você deve anexar a licença a um parâmetro no ambiente do cliente, como o nome de host, o endereço IP, ou o nome de domínio. Dessa maneira, mesmo que a seja disponibilizada na Internet, ela não funcionará em um ambiente para o qual ela não foi emitida. Outra consideração importante é a capacidade de habilitar recursos restritos do produto através do arquivo de licença, sem ter de criar e manter múltiplas versões do software. Incorporar o tempo de expiração ao arquivo de licença pode ser útil se o fornecedor quiser emitir uma licença temporária para avaliação do produto.

Por exemplo, se o Chat tivesse potencial comercial, poderíamos ter distribuído uma edição gratuita, com funcionalidade limitada, que permitiria enviar mensagens de textos simples para um usuário de cada vez. Então poderíamos ter implementado recursos adicionais como texto HTML, cores, listas de amigos, emoticons e suporte a imagens. Teoricamente, poderíamos ter construído duas instâncias do Chat – uma limitada e uma completa com todos os recursos. Mas, na prática, é muito difícil manter esse tipo de código, então nós, como a maioria dos fornecedores, preferimos manter uma base de código da versão completa, com todos os recursos. Para limitar o acesso a uma funcionalidade comercial, devemos inserir verificações em certas partes chave do código, para testar se um arquivo de licença existe e se ele permite ou não tal funcionalidade. Se a verificação falhar, a funcionalidade é desativada. Então poderíamos oferecer o Chat sem um arquivo de licença, como um download gratuito. Se um usuário quisesse se divertir com os recursos avançados de nossa maravilhosa aplicação, ele teria de comprar uma licença. Depois de efetuado o pagamento, emitiríamos uma licença baseada no nome do host do usuário e enviariamós o arquivo de licença para ele. A próxima vez que o usuário executasse o Chat, a aplicação lerá as informações de licença e habilitaria as funcionalidades compradas.

As informações sobre quais recursos habilitar e quais desativar podem ser encriptadas no arquivo de licença, mas isso dificulta a leitura e a manutenção desse arquivo. É mais fácil armazenar os parâmetros de licença como texto simples, mas isso seria como pendurar um pedaço de frango na frente de um crocodilo faminto. Por exemplo, uma licença pode ser emitida para um host específico, mas até mesmo o menos sofisticado dos usuários poderia copiar o arquivo de licença para outra máquina e alterar o valor do nome do host. A solução mais limpa é produzir uma assinatura digital segura baseada nos parâmetros de licença e armazená-la no arquivo de licença, junto com os parâmetros em texto simples. A assinatura então é gerada por um utilitário de licenciamento, usan-

do uma chave privada de um algoritmo assimétrico. Na inicialização, a aplicação utilizaria uma chave pública do algoritmo e a assinatura, para verificar a autenticidade das informações a serem lidas a partir do arquivo de licença. Somente se a verificação fosse bem-sucedida, as restrições da licença seriam removidas.

### Criando um arquivo de licença

Vamos prosseguir com o desenvolvimento de um gerenciador de licenças genérico, e depois utilizá-lo com o Chat. Para simplificar, utilizaremos o formato de arquivo de propriedades de Java para o arquivo de licença. O licenciamento será baseado em três parâmetros: o nome do host, o endereço IP e a data de expiração. Como não quis gastar tempo escrevendo todos esses recursos lucrativos para o Chat, esse exemplo simples é bom o suficiente para ilustrar a abordagem. Crie um novo arquivo chamado `chat.license` em `CovertJava/distrib/conf`, que é semelhante ao mostrado na Listagem 19.11.

#### **LISTAGEM 19.11** Arquivo de licença da aplicação Chat

```
host=localhost
ip=172.24.109.159
expires=2005/1/1
serial=
```

Utilize seu nome de host e seu endereço IP e deixe o valor da propriedade `serial` em branco por enquanto – chegaremos a ela mais tarde. Devemos codificar duas classes, uma para a geração da licença e outra para a verificação. Não queremos distribuir o códí-

## HISTÓRIAS DAS TRINCHEIRAS

A WebCream é uma ferramenta popular para Java que permite a conversão dinâmica de aplicações Swing e applets em sites Web interativos. É um produto comercial distribuído pela CreamTec. Decidimos tornar a edição padrão do WebCream gratuita para persuadir os desenvolvedores a avaliá-la. A edição padrão é uma versão com todos os recursos, que pode ser convertida na versão completa. Para estimular a compra de licenças comerciais, foram impostas limitações no número de usuários concorrentes e em alguns recursos avançados de personalização.

Inicialmente, estávamos construindo três edições diferentes a partir de uma base de código ligeiramente diferente. Com múltiplas plataformas e versões de instaladores, a construção das versões estava levando mais de um dia. Para simplificar a manutenção e atender a prazos curtos, decidimos manter a mesma base de código para as edições comerciais e gratuitas. Os recursos comerciais simplesmente foram bloqueados na edição gratuita, só sendo desbloqueados se fosse localizado um arquivo de licença. O arquivo de licença contém informações encriptadas sobre os hosts permitidos pela licença, o número de usuários concorrentes, além do acesso às funcionalidades comerciais. Essa abordagem simplificou muito a distribuição e o gerenciamento de múltiplas edições.

go de geração de licença com a aplicação, por isso duas classes são necessárias. Como as duas classes precisam ler as informações da licença a partir de um arquivo, faria sentido uma estender a outra. Vamos começar com a classe `LicenseManager`, no pacote `covertjava.protect`. Vamos definir os campos membro para as propriedades da licença (ver Listagem 19.12).

#### **LISTAGEM 19.12 Declaração de LicenseManager**

```
public class LicenseManager {  
    private String host;  
    private String ip;  
    private Date expires;  
    ...  
}
```

Depois vamos fornecer um construtor que recebe o nome do arquivo de licença como parâmetro e preenche os campos internos com as informações da licença, como mostrado na Listagem 19.13.

#### **LISTAGEM 19.13 Inicialização de LicenseManager**

```
public LicenseManager(String licenseFileName) throws Exception {  
    this.licenseProps = new AppProperties(home+File.separator+licenseFileName);  
    this.host = licenseProps.getProperty("host");  
    this.ip = licenseProps.getProperty("ip");  
    String expiresString = licenseProps.getProperty("expires");  
    this.expires = this.dateFormat.parse(expiresString);  
    ...  
}
```

Para proteger os parâmetros da licença contra modificações, precisamos gerar uma assinatura digital. Todos os algoritmos do JCE trabalham com arrays de bytes, então adicionaremos um método `getLicenseString()`, que retorna uma representação unificada de todas as propriedades da licença. O código-fonte para esse método é mostrado na Listagem 19.14.

#### **LISTAGEM 19.14 Representação unificada das propriedades da licença**

```
protected String getLicenseString() throws Exception {  
    return this.host + this.ip + this.expires;  
}
```

Por enquanto, podemos deixar a classe LicenseManager e começar a trabalhar no LicenseGenerator. O LicenseGenerator deve estender o LicenseManager e fornecer métodos para gerar a assinatura digital. Utilizaremos a assinatura digital codificada com base64 como o número serial. Para gerar uma assinatura, também precisamos de um par de chaves para uso com o algoritmo assimétrico. As chaves podem ser geradas com o utilitário *keytool* do JDK, ou usando as APIs de segurança de Java. Com o utilitário *keytool*, as chaves podem ser geradas e exportadas com apenas alguns comandos, mas adotaremos a abordagem de programação, por interesse acadêmico. Primeiro, precisamos decidir o algoritmo e o comprimento da chave a serem usados. As escolhas padrão para a encriptação assimétrica são os algoritmos DSA e RSA. Ambos os algoritmos oferecem proteção adequada, para o tamanho de chave correto, mas utilizaremos o DSA porque ele é suportado nativamente pela JCE da Sun, que é distribuída com a JRE. O tamanho da chave afeta diretamente a complexidade da encriptação: Quanto maior a chave, maior é a dificuldade de quebrá-la. Cada bit dobra o tempo necessário para a quebra. Enquanto chaves de 16 bits podem ser quebradas por uma CPU moderna em questão de minutos, considera-se impossível quebrar chaves de 1024 bits porque, mesmo utilizando toda a potência de silício na Terra, o tempo necessário para quebrar uma chave levaria milhões de anos (pelo menos é o que dizem). Como não estamos fazendo decriptação em tempo real, utilizaremos o tamanho de chave de 1024 bits. O código na Listagem 19.15 mostra um método de LicenseGenerator que gera um par de chaves.

#### **LISTAGEM 19.15** Geração de chaves para o algoritmo DSA

```
public void generateKeys( ) throws Exception {
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
    random.setSeed(System.currentTimeMillis( ));
    keyGen.initialize(1024, random);
    KeyPair pair = keyGen.generateKeyPair();

    String publicKeyPath = home + File.separator + "conf"
        + File.separator + "key_public.ser";
    byte[ ] bytes = pair.getPublic( ).getEncoded( );
    FileOutputStream stream = new FileOutputStream(publicKeyPath);
    stream.write(bytes);
    stream.close( );
    ...
}
```

A implementação obtém primeiro uma instância do gerador DSA de pares de chaves. O gerador precisa ser inicializado com o tamanho da chave (1024 bits) e um provedor de números aleatórios (utilizamos a classe SecureRandom da Sun). Depois de inicializar o gerador, as chaves são geradas via uma chamada a *generateKeyPair()*. Após o par ter sido gerado, falta apenas salvar as chaves públicas e privadas no disco. A Listagem 19.15 mos-

tra como a chave pública foi salva no arquivo `key_public.ser`, no diretório `Conf`. A parte restante do método, que não é mostrada na Listagem 19.15, salva a chave privada no arquivo `key_private.ser`, da mesma maneira.

Obviamente, queremos gerar as chaves apenas uma vez. `LicenseGenerator` recebe um método `main()` que chama `generateKeys()` ou `generateSerialNumber()`, dependendo dos parâmetros de linha de comando. Já vimos a implementação da geração de chaves, então vamos examinar a geração do número serial. Como mencionamos, o número serial é gerado como uma assinatura digital, codificada com `base64` para as propriedades unificadas da licença. O método `generateSerialNumber()` mostrado na Listagem 19.16 faz exatamente isso.

#### **LISTAGEM 19.16** Gerando um número serial

---

```
public String generateSerialNumber() throws Exception {  
    String licenseString = getLicenseString();  
    byte[ ] serialBytes = licenseString.getBytes(CHARSET);  
    serialBytes = getSignature(serialBytes);  
    BASE64Encoder encoder = new BASE64Encoder();  
    return encoder.encode(serialBytes);  
}
```

---

Os bytes da string do número serial são passadas para o método `getSignature()`, cuja saída é então convertida em uma string utilizando a classe `BASE64Encoder`. Isso nos leva à implementação de assinatura digital com o algoritmo DSA, que é mostrada na Listagem 19.17.

#### **LISTAGEM 19.17** Assinatura digital com DSA

---

```
private byte[ ] getSignature(byte[ ] serialBytes) throws Exception {  
    String privateKeyPath = home + File.separator + "conf" +  
                           File.separator + "key_private.ser";  
    FileInputStream stream = new FileInputStream(privateKeyPath);  
    byte[ ] encodedPrivateKey = new byte[stream.available( )];  
    stream.read(encodedPrivateKey);  
  
    PKCS8EncodedKeySpec pubKeySpec= new PKCS8EncodedKeySpec(encodedPrivateKey);  
    KeyFactory keyFactory = KeyFactory.getInstance("DSA");  
    PrivateKey key = keyFactory.generatePrivate(pubKeySpec);  
  
    Signature dsa = Signature.getInstance("SHA1withDSA");  
    dsa.initSign(key);  
    dsa.update(serialBytes);  
    return dsa.sign();  
}
```

---

Estamos assinando as informações de licença utilizando a chave privada para assegurar que mais ninguém possa gerar licenças. A chave privada é lida a partir de um arquivo em um array de bytes (`encodedPrivateKey`). Então convertemos a representação binária em uma representação ASN.1 interna, utilizando a classe `PKCS8EncodedKeySpec` do pacote `java.security.spec`. A representação de chave então é convertida em uma instância de `PrivateKey` utilizando o fabrica de chaves DSA. Tendo os bytes da chave e do número serial, obtemos uma instância do hash seguro com o algoritmo DSA (`SHA1withDSA`), fornecemos seus parâmetros e geramos a assinatura digital utilizando o método `sign()`. Executar o script de geração de licença `licenseGenerator.bat`, para o arquivo de configuração `chat.license` no diretório `distrib/conf`, gera a seguinte saída:

```
C:\CovertJava\bin>licenseGenerator.bat -serial distrib/conf/chat.license
License information read:
host=localhost
ip=172.24.109.159
expires=Sat Jan 01 00:00:00 EST 2005
Serial=[MCOCFBiEzKka0pnEQS1DyKxbHy+gE1+zAhUA1xP1WyAXcCDcoWSRY/Kk/xAkvTQ=]
```

Agora precisamos copiar o número serial gerado e colá-lo como o valor da propriedade `serial` no arquivo `chat.license`. A geração de licença está completa.

### Verificando o arquivo de licença

Temos de aprimorar a aplicação Chat para ler o número serial e verificar se os parâmetros da licença foram mexidos. Como o número serial que utilizamos para Chat é na verdade uma assinatura digital dos parâmetros, precisamos codificar um método que utilize a chave pública do par de chaves gerado para verificar essa assinatura. Vamos adicionar um método chamado `verifySerialNumber()` à classe `LicenseManager` que codificamos anteriormente. Para verificar a assinatura digital gerada com a chave privada, o método deve utilizar a chave pública. As propriedades da licença e o número serial foram lidos a partir do arquivo de licença no construtor `LicenseManager` e armazenadas nas variáveis membro. O código-fonte do `verifySerialNumber()` é mostrado na Listagem 19.18.

#### LISTAGEM 19.18 Verificando o número serial

---

```
public void verifySerialNumber(String keyFileName) throws Exception {
    String keyFilePath = this.home + File.separator + keyFileName;
    FileInputStream stream = new FileInputStream(keyFilePath);
    byte[ ] encodedPubKey = new byte[stream.available( )];
    stream.read(encodedPubKey);
    X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encodedPubKey);
    KeyFactory keyFactory = KeyFactory.getInstance("DSA");
    PublicKey publicKey = keyFactory.generatePublic(pubKeySpec);
```

**LISTAGEM 19.18** Continuação

---

```

byte[ ] licenseData = getLicenseString( ).getBytes(CHARSET);
String encodedSig = this.licenseProps.getProperty("serial");
if (encodedSig == null || encodedSig.length( ) == 0)
    throw new InternalError("Serial number is missing");
BASE64Decoder decoder = new BASE64Decoder( );
byte[ ] serialSig = decoder.decodeBuffer(encodedSig);

Signature signature = Signature.getInstance("SHA1withDSA");
signature.initVerify(publicKey);
signature.update(licenseData);
if (signature.verify(serialSig) == false)
    throw new InternalError("Invalid serial number");
}

```

---

O método lê primeiro o conteúdo do arquivo da chave pública e utiliza a classe X509EncodedKeySpec com o factory de chaves DSA para converter a representação binária da chave em um objeto do tipo PublicKey. Então a representação unificada dos parâmetros da licença retornada por `getLicenseString()` é convertida em um array de bytes. O número serial é lido como o valor da propriedade `serial` a partir do arquivo de licença e, se não estiver ausente, o número é decodificado a partir da codificação base64 utilizando a classe `BASE64Decoder`. Uma instância do algoritmo de assinatura `SHA1withDSA` é obtida e é fornecida com a chave pública e os dados da licença. Por fim, uma chamada ao método `verify()` do algoritmo de assinatura é utilizada para testar se os dados do número serial são uma assinatura digital correta para os dados de licença. Se a verificação falhar, uma exceção é informada utilizando `InternalError`.

Para integrar a verificação de licença com o Chat, precisamos adicionar a invocação a `verifySerialNumber()` ao método `main()` de `ProtectedChat`. Como já temos uma instância de `LicenseManager` em `main()`, simplesmente adicionamos o bloco de código mostrado na Listagem 19.19.

**LISTAGEM 19.19** Invocando a verificação de licença

---

```

public static void main(String[ ] args) throws Exception {
    ...
    // Verifica as informações de licença
    licenseManager.verifySerialNumber("conf/key_public.ser");
    if (licenseManager.isHostAllowed( ) == false)
        throw new Exception("Host is not allowed by the license");
    if (licenseManager.isLicenseExpired( ) == true)
        throw new Exception("The license is expired");
}

```

---

Se o método `verifySerialNumber( )` de `LicenseManager` não lançar uma exceção, são realizadas a verificação do nome do host e da data de expiração da licença. A verificação do nome do host é uma comparação de strings simples entre o nome do host lido do arquivo de licença e o nome do host que executa o Chat. A verificação da data de expiração é uma comparação igualmente simples entre a data de sistema atual e a data de expiração da licença lida. Apenas se as duas verificações forem bem-sucedidas, os recursos comerciais de Chat são habilitados. Contanto que o código em `LicenseManager` e `ProtectedChat` não seja hackeado, temos um mecanismo muito seguro de licenciamento.

Uma abordagem interessante para inserir as verificações de licenciamento é utilizando a instrumentação de bytecode, descrita no Capítulo 17. Em vez de invocar manualmente os métodos de `LicenseManager` por todas as classes da aplicação, um utilitário de pós-processamento pode ser desenvolvido para decorar os métodos chave da aplicação com o código de verificação de licença. O utilitário executaria depois de o código-fonte ser compilado, mas antes de ser colocado em um JAR de distribuição. O bytecode inserido lançaria uma exceção ou retornaria um erro se a licença fosse inválida, ou se o recurso não fosse permitido. Isso fornece uma separação limpa entre a lógica da aplicação e código de licenciamento.

## Registro de licenças e ativação na Web

Utilizar o mecanismo de licenciamento descrito na seção anterior fornece uma grande proteção contra pirataria. Entretanto, se a verificação de licença for hackeada, a proliferação do produto comprometido pode ser difícil de monitorar, especialmente se ele aparecer na Internet. Uma boa estratégia é duplicar a invocação dos métodos de verificação de licença por todo o código da aplicação. No código de exemplo, a aplicação Chat só instancia e utiliza `LicenseManager` no método `main( )` de `ProtectedChat`. Para mais proteção, você deve chamar os mesmos métodos no código `MainFrame` ou `ChatServer`. Outra medida de proteção é incluir ativação e registro via Web.

A idéia por trás do registro via Web é que toda vez que a aplicação for executada, ela se conecte ao site Web do fornecedor e verifique se as informações de licença ainda são válidas. Isso permite que o fornecedor monitore o número de versões instaladas e desative as licenças ou builds que se sabe terem sido hackeados. Estabelecer uma conexão on-line com o fornecedor traz benefícios adicionais, como a possibilidade de atualizações automáticas e a coleta de estatísticas de uso.

Mas a conexão on-line não deve ser vista como o método principal de ativação e registro. Os produtos podem ser instalados e executados em um ambiente isolado, que esteja atrás de firewalls corporativos que bloqueiam o acesso à Internet. Enviar informações de um cliente para o site Web do fornecedor também pode levantar questões de privacidade.

## Questionário rápido

1. Quais as diferenças entre os algoritmos de resumo de mensagem (*message digest*), encriptação e de assinatura?
2. Qual é a diferença entre algoritmos simétricos e assimétricos?

3. Como você protegeria o conteúdo de um e-mail via Internet? Que classes da JCE precisariam ser utilizadas?
4. Quais medidas podem ser tomadas para proteger o conteúdo da aplicação contra hacking?
5. Utilizamos um algoritmo de resumo de mensagem para computar o checksum de arquivos da distribuição da aplicação Chat. Seria mais seguro utilizar um algoritmo simétrico ou um assimétrico para o checksum? Por quê?
6. Quais são os passos lógicos necessários para obter uma assinatura digital utilizando um algoritmo simétrico?
7. Quais são os passos lógicos necessários para obter uma assinatura digital utilizando um algoritmo assimétrico?

## Resumo

- A criptografia fornece um meio de converter informações legíveis em código incompreensível, que pode ser transmitido abertamente e depois ser revertido à sua forma original.
- A encriptação é o processo de codificação de informações legíveis em código, e a decriptação é o processo de extrair as informações legíveis a partir do código.
- Os algoritmos de resumo de mensagem não modificam o conteúdo da mensagem. Em vez disso, produzem um hash único baseado no conteúdo de mensagem e em uma chave secreta.
- Os algoritmos de encriptação/decriptação ocultam informações sigilosas que podem ser interceptadas por terceiros. O conteúdo da mensagem é modificado utilizando uma chave secreta, gerando uma saída que é praticamente impossível de reverter para o conteúdo original sem a chave.
- O *signing* (assinatura) refere-se à geração de uma assinatura digital relativamente curta, baseada em dados de tamanho arbitrário, utilizando uma chave privada de um algoritmo assimétrico. A assinatura é produzida pelo remetente e transmitida com a mensagem. A autenticidade é assegurada pelo uso de algoritmos assimétricos em que somente o remetente tem a chave privada.
- A JCA (Java Cryptography Architecture) fornece uma implementação completa e robusta de serviços de criptografia e algoritmos.
- Algoritmos de encriptação, de resumo de mensagem e de assinatura podem ser utilizados para proteger a comunicação entre as camadas de uma aplicação distribuída.
- A maioria dos algoritmos de segurança exige parâmetros como uma senha ou chaves. Em geral, os algoritmos operam sobre dados binários.
- Depois do ofuscamento de bytecode, assegurar a integridade de arquivos da distribuição é a medida mais importante para proteger a aplicação contra hacking.
- A maneira mais eficiente de implementar o licenciamento que desbloqueia os recursos comerciais é fornecendo um arquivo-texto de licença, com uma assinatura digital produzida por um algoritmo assimétrico.

# A

## Licença de software comercial

### **CONTRATO DE LICENÇA DO PRODUTO DE SOFTWARE WEBCREAM DA CREAMTEC**

**IMPORTANTE. LEIA COM ATENÇÃO:** Este Contrato de Licença do Usuário Final da WebCream (“Licença” ou “Contrato”) é um contrato legal entre você e a CreamTec (“CreamTec”) para o WebCream da CreamTec (“Software” ou “Produto”), que inclui software de computador e pode incluir materiais relacionados com mídia, impressos e documentação “on-line” ou eletrônica. Na instalação do produto de software, você concorda em obedecer aos termos desta Licença. Qualquer instalação ou utilização do produto WebCream significará a aceitação e seu consentimento em obedecer a esta Licença. Se você não concordar com os termos desta Licença, não instale nem utilize o Produto e, se recebeu o Produto por um meio diferente do eletrônico, retorno-o imediatamente para a CreamTec.

### **LICENÇA DA CREAMTEC**

WebCream significa o produto atual ou futuro WebCream da CreamTec e quaisquer módulos adicionais, se houver, licenciados para você da CreamTec, que estão instalados em computador(es) que atua(m) como servidor(es). Os componentes adicionais de software podem ter sido distribuídos para você junto com o Produto. Exceto se o contrário for especificamente declarado em um contrato de licença separado fornecido com quaisquer componentes, tais componentes adicionais estão sujeitos a esta Licença. O Produto é protegido pelas legislações de direitos autorais e tratados

de direitos autorais internacionais, bem como outras leis e tratados de propriedade intelectual. O Produto é licenciado, não-vendido.

**1. CONCESSÃO DE LICENÇA.** Esta Licença concede os seguintes direitos perpétuos, não-exclusivos e intransferíveis:

- a) Instalação e utilização – Sujeito ao servidor de testes, direitos de backup e recuperação após desastre declarados em outra parte desta Licença, você pode instalar o Produto onde a aplicação está localizada com base no número de usuários concorrentes por aplicação de acordo com restrições da licença comprada.
- b) Utilização comercial – Uma vez instalado de acordo com esta Licença, você pode utilizar o Produto somente na condução de seu próprio negócio ou de seus Afiliados e não pode, direta ou indiretamente, utilizar o Produto para realizar o trabalho para terceiros. “Afiliados” significa qualquer entidade controlada por você ou sob seu controle comum, indivíduo ou entidade que compra esta Licença.
- c) Outras restrições na utilização – Seus direitos sob esta Licença não incluirão o direito de conceder sublicenças ou transferir (incluindo a transferência por aluguel ou locação) o Produto ou qualquer parte deste. Qualquer tentativa de conceder sublicenças ou transferir quaisquer direitos será considerada uma infração a este Contrato. Você não pode criar trabalhos derivados com base no produto, fazer engenharia reversa dele, descompilá-lo ou desassemblá-lo, exceto à extensão da restrição expressa anteriormente proibida por lei aplicável.
- d) Recuperação após desastre e backup – Você pode manter o Produto em um site separado de recuperação após desastre, desde que a instalação seja unicamente para os propósitos de backup e utilização de emergência. Além disso, depois da instalação do Produto de contrato com esta Licença, você pode manter a mídia original em que o Produto foi fornecido unicamente para propósitos de arquivamento ou para a reinstalação do Produto de acordo com os termos desta Licença.

**2. VERSÕES SUBSEQÜENTES.** Um Produto rotulado como uma versão subseqüente (ou termo semelhante) substitui e/ou suplementa o produto originalmente licenciado e, seguindo a liberação subseqüente, você pode utilizar o Produto resultante apenas de acordo com os termos desta Licença. Essas versões incluem aprimoramentos e correções e modificações do Produto e adendos. As versões também incluem versões superiores do Produto. Durante o primeiro ano deste contrato e mediante pagamento da taxa anual de manutenção a partir desta data em diante, você receberá para sua utilização todas as versões publicadas pela CreamTec. A utilização dessas versões será regulada e sujeita aos termos deste Contrato que se relacionam à reprodução e utilização do Produto.

**3. POSSE.** O produto é licenciado, não vendido. O título e os direitos autorais dentro de e para o Produto, materiais impressos que acompanham o produto e quaisquer cópias que você tenha direito de fazer aqui são propriedade da CreamTec.

**4. SOFTWARE DE MÍDIA DUAL.** Você pode receber o Produto em mais de uma mídia. Independentemente do tipo ou tamanho de mídia recebido, você pode utilizar somente a mídia apropriada aos seus dispositivos de hardware. Você não pode emprestar, alugar, alocar ou, de outra forma, transferir qualquer mídia não-utilizada para outro usuário.

**5. CONTROLES DE EXPORTAÇÃO.** Você concorda e certifica que nenhum dos dados técnicos recebidos da CreamTec, nem o produto direto deles, será distribuído, transferido ou exportado, direta ou indiretamente, para qualquer país violando qualquer lei aplicável, inclusive a Lei de Administração de Exportação dos Estados Unidos e regulamentos sob essa norma (Export Administration Act).

**6. TÉRMINO.** Você pode terminar esta Licença destruindo ou retornando o Produto e todas as cópias deste à CreamTec. Se não conseguir cumprir qualquer condição deste Contrato, e cada uma das quais for considerada a essência deste Contrato, a CreamTec pode, imediatamente, encerrar este Contrato se não forem pagas as taxas de manutenção na data de vencimento, ou se forem infringidas quaisquer condições desta Licença, e não reparadas essa infração dentro de trinta (30) dias de notificação pela CreamTec. Mediante término do Contrato, você imediatamente cessará a utilização do Produto e, na opção de CreamTec, retornará prontamente à CreamTec todas as cópias do Produto em sua posse ou destruirá todas essas cópias e certificará por escrito que todas as cópias foram retornadas ou destruídas.

**7. GARANTIA LIMITADA.** A CreamTec não tem controle sobre as condições em que você utiliza o Produto e atualizações subsequentes, e não garante e não pode garantir os resultados obtidos por essa utilização.

a) GARANTIA LIMITADA. Além de garantir o direito de conceder a licença contida neste Contrato, a CreamTec garante que a mídia em que o Produto é fornecido e quaisquer manuais de usuário a serem locados sob os termos deste Contrato são de material e confecção sem defeitos sob utilização normal durante um período de trinta (30) dias depois da expedição. A CreamTec fornece garantias adicionais de que o Produto e quaisquer atualizações subsequentes sejam realizadas substancialmente de acordo com os materiais escritos que o acompanham como aquelas especificações contidas no manual do usuário ou documentação fornecida em vigor como da data deste Contrato, durante um período de trinta (30) dias da data de recebimento. A CreamTec não garante que as funções contidas no Produto ou em qualquer atualização subsequente atenderão aos seus requisitos ou que a operação do Produto será ininterrupta ou livre de erros. Esta Garantia Limitada não cobre nenhuma cópia do Produto ou atualização ou qualquer manual de usuário que tenha sido alterado de alguma maneira, ou se a falha do Produto tiver resultado de acidente, abuso ou utilização incorreta. A CreamTec não é responsável por problemas causados por alterações ou modificações nas características operacionais de qualquer hardware de computador ou sistema operacional para o qual o Produto foi adquirido, nem é responsável por problemas que ocorram como resultado do uso do Produto juntamente com software ou hardware incompatível com o Produto. À extensão máxi-

ma permitida pela legislação aplicável, garantias implícitas sobre o Produto, se houver, são limitadas ao prazo de trinta (30) dias. Algumas estados/jurisdições não permitem limitações na duração de uma garantia implícita, então a limitação acima pode não se aplicar a você.

b) INDENIZAÇÕES AO CLIENTE. Toda a responsabilidade da CreamTec e a única indenização a você facultadas será a substituição de qualquer mídia magnética ou manual de usuário que não compra a "Garantia Limitada" da CreamTec. Além disso, embora não assegurando funcionamento ininterrupto ou livre de erros sob nenhum sentido, a CreamTec envidará os melhores esforços para lhe fornecer versões corrigidas do Produto, através de atualizações para corrigir qualquer erro que você localizar no Produto durante o período de garantia e que impeça o Produto de executar, substancialmente, de acordo com o contrato, a forma que é descrita nos materiais que o acompanham. Qualquer substituição do Produto será garantida ao restante do período de garantia original ou trinta (30) dias, o que for maior. Fora dos Estados Unidos, esses reparos e quaisquer Serviços de Suporte ao Produto oferecidos pela CreamTec não estão disponíveis sem a prova de aquisição de uma fonte autorizada. Você deve notificar a CreamTec de qualquer infração de garantia dentro do período de garantia, para que tenha o direito a reparos.

c) À EXTENSÃO MÁXIMA PERMITIDA PELA LEGISLAÇÃO APLICÁVEL E À EXTENSÃO CONTIDA NESTE CONTRATO, OU ANEXO A ESTE CONTRATO, A CREAMTEC E SEUS DISTRIBUIDORES ISENTAM-SE DE TODAS OUTRAS GARANTIAS E CONDIÇÕES, SEJA EXPLÍCITA OU IMPLÍCITAMENTE, INCLUINDO, MAS NÃO LIMITANDO-SE A, GARANTIAS IMPLÍCITAS OU CONDIÇÕES DE COMERCIABILIDADE, ADEQUAÇÃO A UM PROPÓSITO PARTICULAR, TÍTULO E NÃO-INFRAÇÃO, COM REFERÊNCIA AO PRODUTO E A CLAÚSULA DE OU FALHA PARA FORNECER SERVIÇOS DE SUPORTE. A garantia limitada acima fornece a você direitos legais específicos. Você pode ter outros direitos, que variam entre jurisdições. As garantias contidas na Subseção a) desta Seção são feitas no lugar de todas as outras garantias expressas, sejam orais ou escritas. Somente um funcionário autorizado da CreamTec pode fazer modificações nesta garantia ou garantias de vinculação adicionais sobre a CreamTec, e tais modificações ou garantias devem ser por escrito. De maneira análoga, declarações adicionais como as feitas em anúncios publicitários ou apresentações, sejam elas orais ou escritas, não constituem garantias da CreamTec e não devem ser consideradas como tais.

d) Quaisquer declarações feitas por um negociante, ou qualquer outro terceiro que não a CreamTec, não são garantias e não podem ser consideradas. A CreamTec não será responsável por qualquer não-conformidade do Produto de Software reivindicada sob o Artigo 35(2) da Convenção das Nações Unidas em Contratos para a Venda Internacional de Mercadoria, mesmo se essa Convenção fosse determinada aplicável a esta Licença e a transações subjacentes.

e) LIMITAÇÃO DE RESPONSABILIDADE. À EXTENSÃO MÁXIMA PERMITIDA POR LEGISLAÇÃO APLICÁVEL, EM NENHUM CASO DEVE A CREAMTEC OU SEUS DISTRIBUIDORES SER RESPONSÁVEIS POR QUALQUER DANO ESPECIAL, ACIDENTAL, INDIRETO OU CONSEQUENTE, QUAISQUER QUE SEJAM ELES (INCLUSIVE, MAS NÃO

LIMITANDO-SE A, DANOS POR PERDA DE LUCROS, INTERRUPÇÃO DE NEGÓCIO, PERDA DE INFORMAÇÕES DE NEGÓCIO ou QUALQUER OUTRA PERDA PECUNIÁRIA) DECORRENTE DO USO OU DA INCAPACIDADE DE UTILIZAR O PRODUTO OU DA FALHA EM FORNECER SERVIÇOS DE SUPORTE, MESMO SE A CREAMTEC TENHA SIDO ADVERTIDA DA POSSIBILIDADE DESTES DANOS. EM QUALQUER CASO, A INTEIRA RESPONSABILIDADE DA CREAMTEC SOB QUALQUER PROVISÃO DESTA LICENÇA ESTARÁ LIMITADA À QUANTIDADE REALMENTE PAGA POR VOCÊ PELO PRODUTO. Como algumas jurisdições não permitem a exclusão ou limitação de responsabilidade, a limitação acima pode não se aplicar a você.

**8. RESTRIÇÃO DE DIREITOS DO GOVERNO DOS ESTADOS UNIDOS.** O Produto e a documentação são fornecidos com RESTRIÇÃO DE DIREITOS. A utilização, duplicação ou revelação pelo Governo está sujeita a restrições estabelecidas no subparágrafo (c)(1)(ii) dos Direitos sobre Dados Técnicos e cláusula de Software de Computador em DFARS 252.227-7013 ou subparágrafos (c)(1) e (2) do Software Comercial de Computador – direitos restringidos em 48 CFR 52.227-19, quando aplicáveis.

**9. TERMO.** Este Contrato é efetivo a partir da data de sua execução a uma data de um ano a partir desta data, a menos que encerrado anteriormente por qualquer parte, por ausência de outra parte em qualquer dever sob este Contrato.

**10. TAXA ANUAL DE MANUTENÇÃO.** Na data de aniversário deste Contrato, você pagará uma taxa anual de manutenção. O pagamento das taxas dá a você direito à utilização contínua do Produto, bem como atualizações de produto, versões e suporte técnico. A falha no pagamento da taxa de manutenção anual constitui quebra deste Contrato e será a base para o encerramento imediato deste Contrato.

**11. RENOVAÇÃO DE LICENÇA.** A Licença concedida sob este Contrato e os termos deste Contrato serão renovados automaticamente com o pagamento continuado da taxa de manutenção anual, a menos que qualquer parte notifique por escrito à outra uma intenção de encerrar ou um pedido de modificar os termos, pelo menos sessenta (60) dias antes da data de expiração deste Contrato. No caso de nenhum aviso de terminação ou pedido de modificação ser enviado por qualquer parte, este Contrato, sua licença e seus termos, serão renovados para o termo de um ano. Se qualquer parte notificar à outra da decisão de término sessenta (60) dias antes da data de expiração do Contrato, ou se as partes não puderem entrar em acordo com as modificações propostas, a Licença terminará na data de expiração do Contrato.

**12. IMPOSTOS.** Você deve pagar todos os impostos locais, estaduais e federais (mas excluindo taxas impostas na renda de CreamTec) cobrados ou impostos por motivo das transações contempladas neste Contrato. Você pagará prontamente à CreamTec uma quantidade igual a qualquer imposto(s) realmente pago(s) ou necessário a ser arrecadado pela CreamTec.

**13. INDENIZAÇÃO.** A CreamTec, por custo próprio, indenizará, defenderá e sustentará a inocência do Licenciado quanto a todos os danos, custos e prêmios decorrentes de qualquer ação de terceiros à extensão que seja baseada em uma reivindicação de que o Produto, ou qualquer atualização subsequente utilizada dentro do escopo deste Contrato, infrinja qualquer patente, direitos autorais, licença, segredo comercial, ou outro direito de propriedade, contanto que sejamos imediatamente notificados por escrito dessa reivindicação. A CreamTec, por despesa própria, defenderá qualquer ação trazida contra o Licenciado ou a CreamTec à extensão de que seja baseada em uma reivindicação de que o Produto ou qualquer atualização subsequente utilizada dentro do escopo deste Contrato infrinja qualquer patente, direitos autorais, licença, segredo comercial ou outro direito de propriedade, contanto que sejamos imediatamente notificados por escrito dessa reivindicação. A CreamTec terá o direito de controlar a defesa de todas essas reivindicações, processos e outros procedimentos. Em nenhum caso, você determinará qualquer reivindicação, processo ou procedimentos sem aprovação escrita prévia da CreamTec. A CreamTec não terá nenhuma responsabilidade por qualquer reivindicação sob esta Seção se uma reivindicação de patente, direitos autorais, licença ou infração de segredo comercial estiver baseada no uso de uma versão suplantada ou alterada do Produto, se tal infração tiver sido evitada pelo uso da última versão inalterada do Produto disponível como uma atualização.

**14. ARBITRAGEM.** Se você adquiriu esse produto nos Estados Unidos, este Contrato é governado pela legislação da Virgínia. Se este produto foi adquirido fora dos Estados Unidos, a legislação local pode ser aplicável. Exceto pelo direito de qualquer parte aplicar-se a um tribunal de jurisdição competente para uma ordem restringente temporária, uma injunção preliminar ou outro alívio equitativo para preservar o estado atual ou impedir dano irreparável, qualquer controvérsia ou reivindicação decorrente ou relacionada a este Contrato será determinada vinculando a arbitragem administrada pela American Arbitration Association (Associação Americana de Arbitragem) e visando às suas regras e o julgamento sobre a sentença administrada em tal arbitragem pode ser inserido em qualquer tribunal de jurisdição competente.

**15. GERAL.** Este Contrato terá efeito para o benefício de CreamTec, seus sucessores e cessionários. Cada parte reconhece que leu este Contrato e que o entende e concorda em estar limitado por seus termos, e, concordando adicionalmente que eles são a declaração completa e exclusiva do contrato entre as partes que suplanta e combina todas as propostas anteriores, entendimentos e todos outros contratos, orais e escritos, entre as partes relacionadas com este Contrato. Se qualquer cláusula do Contrato for considerada inválida por uma corte de jurisdição competente, tal cláusula será imposta à extensão máxima permitida e o restante permanecerá em força total.

Se você tiver qualquer pergunta relacionada a esta Licença, ou se desejar entrar em contato com a CreamTec por qualquer razão, entre em contato conosco: CreamTec, 2400 Clarendon Blvd. #406, Arlington, VA 22201, ou envie um e-mail para: [sales@creamtec.com](mailto:sales@creamtec.com).

# B

## *NESTE APÊNDICE*

- ▶ Utilitários e ferramentas 216
- ▶ Descompilação 216
- ▶ Ofuscamento 217
- ▶ Rastreamento e logging 217
- ▶ Depuração 217
- ▶ Profiling 217
- ▶ Testes de carga 218
- ▶ Eavesdropping 218
- ▶ Ajuste de bytecode 219
- ▶ Aplicando patch a código nativo 219
- ▶ Proteção contra hacking 220

# Recursos

## Utilitários e ferramentas

### **Nome: FAR**

URL: <http://www.rarsoft.com/>

Licença: Shareware (versão 1.70)

Descrição: Gerenciador de arquivos e de arquivos compactados que substitui uma combinação de Windows Explorer + Bloco de Notas + CMD.EXE.

### **Nome: Total Commander**

URL: <http://www.ghisler.com/>

Licença: Shareware (versão 6.02)

Descrição: Gerenciador de arquivos e de arquivos compactados que substitui uma combinação de Windows Explorer + Bloco de Notas + CMD.EXE.

### **Nome: WebCream**

URL: <http://www.creamtec.com/webcream/>

Licença: Comercial (versão 5.0.0)

Descrição: Converte aplicações gráficas em Java em sites Web HTML interativos, em tempo real.

## Descompilação

### **Nome: JAD**

URL: <http://kpodus.tripod.com/jad.html>

Licença: Freeware (versão 1.5.8e2)

Descrição: Descompilador rápido de arquivos de classes Java escritos em C.

### **Nome: JODE**

URL: <http://jode.sourceforge.net/>

Licença: GPL (versão 1.1)

Descrição: Biblioteca Java contendo um descompilador e um otimizador para código Java.

## Ofuscamento

### Nome: Zelix KlassMaster

URL: <http://www.zelix.com/klassmaster/>

Licença: Comercial (versão 4.1)

Descrição: Ofuscador muito poderoso que suporta ofuscamento do fluxo de controle.

### Nome: ProGuard

URL: <http://proguard.sourceforge.net/>

Licença: GPL (versão 2.1)

Descrição: Ofuscador de arquivos de classes Java.

### Nome: RetroGuard

URL: <http://www.retrologic.com/retroguard-main.html>

Licença: GPL (versão 1.1)

Descrição: Ofuscador de arquivos de classes Java.

## Rastreamento e logging

### Nome: Framework Log4J

URL: <http://logging.apache.org/log4j/>

Licença: Apache (versão 1.2.8)

Descrição: Framework para a geração de mensagens de log e gerenciamento de arquivos de log.

## Depuração

### Nome: Omniscent Debugger

URL: <http://www.lambdacs.com/debugger/debugger.html>

Licença: GPL (versão de 6 de setembro de 2003)

Descrição: Através do registro de cada alteração de estado na aplicação-alvo durante a execução, permite ao desenvolvedor navegar para trás no tempo para examinar valores de variáveis e objetos.

## Profiling

### Nome: JProbe

URL: <http://www.quest.com/jprobe/>

Licença: Comercial (versão 5.0.0)

Descrição: Conjunto completo para tuning de código Java (profiler, threadalizer, depurador de memória).

**Nome: Optimizelit**

URL: <http://www.borland.com/optimizelit/>

Licença: Comercial (versão 5.5)

Descrição: Conjunto completo para tuning de código Java (profiler, threadalizer, depurador de memória).

**Nome: JProfiler**

URL: <http://www.ej-technologies.com/products/jprofiler/overview.html>

Licença: Comercial (versão 2.4)

Descrição: Profiler, threadalizer e depurador de memória Java.

## Testes de carga

**Nome: JUnit**

URL: <http://www.junit.org/>

Licença: Licença pública comum (versão 3.8.1)

Descrição: Framework simples para escrever testes de unidade em Java.

**Nome: JMeter**

URL: <http://jakarta.apache.org/jmeter/>

Licença: Apache (versão 1.9.1)

Descrição: A aplicação Java desktop para realização de testes de carga funcionais e de medidas de desempenho de aplicações servidoras (web e J2EE).

**Nome: LoadRunner**

URL: <http://www.mercuryinteractive.com/products/loadrunner/>

Licença: Comercial (versão 6)

Descrição: Ferramenta avançada de testes de carga que faz previsões de comportamento e desempenho em nível de sistema.

## Eavesdropping

**Nome: TCPMon (Apache Axis)**

URL: <http://ws.apache.org/axis/>

Licença: Apache (versão 1.1)

Descrição: Utilitário gráfico de tunelamento que mostra o conteúdo de mensagens. Pode ser utilizado para fazer o *eavesdropping* de protocolos baseados em HTTP.

**Nome: HTTP Sniffer**

URL: <http://www.effetech.com/>

Licença: Comercial (versão 3.5)

Descrição: Ferramenta poderosa para monitorar e analisar tráfego na Internet e informações avançadas dentro de pacotes de vários protocolos, como HTTP, FTP, SMTP, POP3 e Telnet.

**Nome: Ethereal**

URL: <http://www.ethereal.com/>

Licença: GPL (versão 0.9.13)

Descrição: Utilizado para solucionar problemas de rede, análise e desenvolvimento de software e protocolos de rede e aprendizado. Pode ser utilizado para fazer o *eavesdropping* de praticamente todos os protocolos de comunicação.

**Nome: P6Spy**

URL: <http://www.p6spy.com/>

Licença: Apache (versão 1.2)

Descrição: Framework Open Source para aplicações que interceptam e opcionalmente modificam instruções de bancos de dados. Pode ser utilizado para *eavesdropping* JDBC.

## Ajuste de bytecode

**Nome: jClassLib Bytecode Viewer**

URL: <http://sourceforge.net/projects/jclasslib/>

Licença: GPL (versão 1.2)

Descrição: Ferramenta que visualiza todos os aspectos de arquivos de classes do Java compilados e o bytecode neles contido.

**Nome: BCEL**

URL: <http://jakarta.apache.org/bcel/>

Licença: Apache (versão 5.1)

Descrição: A Byte Code Engineering Library foi projetada para analisar, criar e manipular arquivos de classes Java binários.

**Nome: ASM**

URL: <http://asm.objectweb.org/>

Licença: BSD (versão 1.4.2)

Descrição: Framework de alto desempenho para manipulação de bytecode Java.

## Aplicando patch a código nativo

**Nome: PE Explorer**

URL: <http://www.heaventools.com/>

Licença: Comercial (versão 1.94)

Descrição: Utilitário gráfico que permite visualizar, analisar, editar, corrigir e reparar as estruturas internas de arquivos PE visualmente.

**Nome: Function Replacer**

URL: <http://execution.cjb.net/>

Licença: AS-IS (versão 1.0)

Descrição: Utilitário que substitui uma função exportada em uma DLL por uma função exportada por outra DLL.

**Nome: IDA Pro**

URL: <http://www.datarescue.com/>

Licença: Comercial (versão 4.6)

Descrição: O IDA Pro é o principal disassembler de sistema interativo, multioperacional e multiprocessado.

**Nome: Detours Library**

URL: <http://research.microsoft.com/sn/detours/>

Licença: Licença de pesquisa da Microsoft (versão 1.5)

Descrição: Detours é uma biblioteca para instrumentalizar funções Win32 arbitrárias em máquinas x86. A biblioteca Detours intercepta funções Win32 através da reescrita de imagens de funções-alvo.

**Nome: OllyDbg**

URL: <http://home.t-online.de/home/01lydbg/>

Licença: Grátis para uso; é necessário registro (versão 1.09)

Descrição: Depurador de 32 bits de análise em nível de assembler para Microsoft Windows. A ênfase na análise de código binário o torna particularmente útil nos casos em que a código-fonte não está disponível.

**Nome: libelf**

URL: <http://www.gnu.org/directory/libs/misc/libelf.html>

Licença: LGPL (a versão varia dependendo da plataforma)

Descrição: Permite ler, modificar ou criar arquivos ELF de maneira independente de arquitetura.

## Proteção contra hacking

**Nome: Bouncy Castle JCE**

URL: <http://www.bouncycastle.org/>

Licença: Gratuito, com licença AS-IS (versão 1.22)

Descrição: Biblioteca Java que fornece uma implementação de vários algoritmos de segurança e encriptação.

**Nome: Cryptix JCE**

URL: <http://www.cryptix.org/>

Licença: Livre com licença AS-IS (a versão varia dependendo do subpacote)

Descrição: Biblioteca Java que fornece uma implementação de vários algoritmos de segurança e encriptação.

# C

# Respostas do questionário

## Capítulo 1

1. Descompilando classes, utilizando rastreamento eficaz, quebrando código com depuradores, utilizando profilers para análise de aplicações em tempo de execução, eavesdropping e engenharia reversa.
2. Descompilar classes, hackear variáveis e métodos não-públicos, substituir e aplicar patches a classes de aplicações, manipular a segurança Java, hackear recursos de aplicações e elementos de interface gráfica, controlar o carregamento de classes, substituir e corrigir classes Java básicas, interceptar o fluxo de controle, entender e ajustar bytecode, e controle total com a substituição de código nativo.
3. Substituir e aplicar patches a classes de aplicações, utilizar rastreamento eficaz e eavesdropping.
4. Windows Explorer, Notepad/TextPad, CMD.EXE, cliente FTP e WinZip e outros arquivadores.

## Capítulo 2

1. Recuperar código-fonte que foi acidentalmente perdido, aprender sobre a implementação de um recurso, solucionar problemas de uma aplicação ou biblioteca sem boa documentação, corrigir bugs urgentes em software de terceiros sem código-fonte e aprender a proteger o código contra hacking.

2. Opções de depuração especificadas utilizando `-g`. Quanto mais informações de depuração forem incluídas, melhor será o código descompilado.
3. Porque o código-fonte Java é compilado no bytecode intermediário, em vez de no código de máquina, e porque existe um mapeamento bem definido entre os operadores de código-fonte e entre palavras-chave e o bytecode gerado.
4. Não há nenhuma maneira de proteger o código contra a descompilação, mas o ofuscamento pode tornar o código descompilado quase impossível de entender.

## Capítulo 3

1. Além da proteção jurídica oferecida por direitos autorais e patentes, o ofuscamento de bytecode fornece um meio eficiente contra a descompilação.
2. Desfiguração de nomes, codificação de strings Java e alteração do fluxo de controle.
3. Descompilar e utilizar um bom depurador.

## Capítulo 4

1. Criar uma classe auxiliar no pacote da classe que tem o membro com visibilidade `protected` ou de pacote; utilizar a Reflection API com um gerenciador de segurança.
2. Configurar um gerenciador de segurança que concede as permissões necessárias e em seguida acessar o membro privado com a Reflection API.
3. A classe auxiliar funciona bem para classes que não são de sistema, mas precisa estar no classpath de inicialização para classes de sistema; essa técnica não pode ser utilizada para membros privados. A Reflection API não requer a manipulação do classpath de inicialização e pode acessar membros de dados privados, mas é mais lenta e precisa de certas permissões.

## Capítulo 5

1. Navegar pelas classes, começando do ponto de entrada, pesquisa textual de uma string conhecida ou de um nome de classe e pilha de chamadas de uma exceção, ou um dump de thread.
2. Porque strings em Java são armazenadas como texto simples dentro do bytecode binário. A pesquisa textual não funciona para constantes string se as strings tiverem sido codificadas por um ofuscador.
3. Utilizando o método `dumpStack()` da classe `java.lang.Thread`.
4. As classes corrigidas com patches devem estar localizadas antes das classes originais e serem carregadas com o mesmo class loader.

## Capítulo 6

1. O rastreamento não exige que uma aplicação esteja executando em modo de depuração. As mensagens de rastreamento são inseridas permanentemente no código-fonte.
2. Como os rastreamentos são projetados para fornecer um histórico inteligível para seres humanos de operações realizadas pela aplicação, sua leitura é mais fácil do que a de código Java descompilado. Examinar rastreamentos fornece um entendimento da implementação e do fluxo de controle da aplicação.
3. Recarregar o arquivo de configuração em tempo de execução é importante porque evita que se tenha de reinicializar a aplicação.
4. Cada utilização do operador + em strings Java resulta na operação custosa de alocar um novo buffer e copiar as strings de argumento para ele.

## Capítulo 7

1. `java.lang.SecurityManager`.
2. Uma permissão representa o acesso a informações de sistema ou a um recurso.
3. Primeiro, o arquivo `java.policy` é carregado a partir do diretório `lib/security` do diretório de instalação do JRE. Em seguida, o arquivo `.java.policy` é carregado a partir do diretório `home` do usuário.

## Capítulo 8

1. A espionagem permite conhecer os valores exatos dos vários parâmetros de ambiente de execução. Ela elimina a adivinhação.
2. Valores de propriedades de sistema, o gerenciador de segurança instalado e várias informações sobre memória e rede.
3. Aplicações nativas não têm as mesmas restrições da JVM, portanto podem obter informações mais detalhadas sobre o sistema local. Aplicações Java podem fazer interface com módulos nativos utilizando JNI ou arquivos de configuração simples.

## Capítulo 9

1. Ao trabalhar com aplicações grandes que não utilizam rastreamento, quando o código-fonte de aplicação não fornece um entendimento claro da lógica interna, ou quando a aplicação tiver sido agressivamente ofuscada.
2. Depuradores convencionais só exibem as informações do momento atual; logo depois, essas informações são irremediavelmente perdidas. Para serem eficazes, os

depuradores convencionais exigem pontos de interrupção (*breakpoints*) inseridos estrategicamente em vários pontos do código.

3. A depuração onisciente permite registrar os estados do programa em execução e depois voltar no tempo para examinar os estados gravados. A idéia por trás do depurador onisciente é registrar o máximo possível de informações sobre os threads, variáveis e seus valores, a entrada padrão e streams de saída e classes carregadas.
4. A lógica pode ser localizada navegando-se para o código a partir de um ponto inicial conhecido. O ponto inicial pode ser uma chamada para `System.out`, um início de um thread ou o nome de um método. Uma pesquisa textual pode ser utilizada para localizar um ponto inicial com base em uma string conhecida.

## Capítulo 10

1. Investigar o uso do heap e a freqüência de coleta de lixo; localizar e corrigir vazamentos de memória; localizar problemas de travamentos e de concorrência de dados em aplicações com múltiplos threads; e investigar a aplicação em execução para entender melhor sua estrutura interna.
2. A coleta de lixo completa começa a partir das raízes da árvore de objetos e identifica todos os objetos que podem ser alcançados a partir da raiz. Objetos inacessíveis a partir da raiz são marcados para coleta de lixo. Aplicações com grandes árvores de objetos requerem muito tempo de processamento.
3. Objetos persistentes impedem que a memória seja recuperada na coleta de lixo. Enquanto existir uma referência a um objeto, ele não está elegível para a coleta de lixo, mesmo se nunca for utilizado novamente.
4. Utilizar um profiler é a maneira mais eficiente de localizar e corrigir vazamentos de memória. Isso pode ser feito navegando pela árvore de referências ou localizando todos os caminhos para a raiz em um instantâneo (*snapshot*) do heap.
5. Os dois problemas mais comuns são condições de concorrência de dados e deadlocks.
6. Executar uma aplicação em um profiler que coleta estatísticas de execução, como tempos de execução e número de chamadas de métodos, tempo cumulativo e número médio de objetos por método.

## Capítulo 11

1. O propósito dos testes de carga é avaliar como o desempenho de sistema atende aos requisitos de nível de serviço sob carga.
2. *Simultâneo* significa que os clientes estão enviando uma solicitação ao mesmo tempo; *concorrente* significa que os clientes mantêm uma conversa com o servidor mas enviam solicitações aproximadamente ao mesmo tempo.
3. RMI.

4. O método `assertTrue( )` precisa ser utilizado com um parâmetro `false` para indicar ao JUnit que um teste apresentou falhas.
5. HTTP e FTP. O JMeter também pode ser utilizado para testar bancos de dados, scripts Perl e objetos Java.
6. Um plano de testes pode ter grupos de threads, listeners, elementos de configuração, assertivas, pré-processadores, pós-processadores e timers. Um grupo de threads pode ter ainda controladores lógicos e amostras.
7. Adicionando listeners como View Results Tree e Assertion Results.
8. O JUnit é um framework simples, que requer a programação dos testes. Ele não oferece automação nem suporte para testes de aplicações Web complexas. É bom para testes de unidade de baixo nível. O JMeter é uma ferramenta com interface gráfica sofisticada e vários elementos para a construção de planos de teste. Ele pode automatizar o teste de aplicações Web, mas trata a aplicação como uma caixa-preta.

## Capítulo 12

1. Desempacote todas as classes de biblioteca e faça uma pesquisa textual por `Unknown error` em todos os arquivos. Se a string for escrita diretamente no código em um arquivo `CLASS`, descompile a fonte, altere o texto da string, recompile a fonte e instale o patching. Se a string estiver localizada em um arquivo de configuração, altere-a nesse arquivo.
2. Descubra qual arquivo GIF ou JPEG é utilizado na caixa de diálogo `About`, desempacotando a distribuição do Chat e visualizando os arquivos de imagens. Edite o arquivo da imagem, salve-o e reempacote a aplicação Chat.
3. Primeiro, verifique todos os arquivos de configuração, para identificar se a opção é configurável. Se não for, pesquise nos arquivos de classes a mensagem exibida quando o limite de conexões concorrentes for alcançado. Se um arquivo de classe for localizado, descompile-o e o utilize como ponto de partida para localizar a classe que impõe o limite. Depois que a classe for localizada, examine o código para ver se o limite é escrito diretamente no código.

## Capítulo 13

1. As abordagens gerais para eavesdropping requerem interceptar a troca de mensagens entre cliente e servidor. Isso pode ser feito colocando-se um intermediário que rastreia a comunicação, ou ouvindo protocolos de rede do tipo *broadcast*.
2. Porque HTTP é baseado em texto. A adoção ampla do HTTP resultou em uma proliferação de ferramentas de tunelamento e sniffing.
3. A comunicação HTTP pode ser protegida executando-se HTTP sobre SSL (HTTPS).

4. O sniffing de rede é possível executando-se um host no modo promíscuo, pelo qual ele aceita todos os pacotes que atravessam a rede, independentemente de seu destino.
5. Espionando o protocolo de rede subjacente utilizado para transportar os dados de objetos serializados.
6. Instalar um proxy de logging ou um wrapper em volta do driver JDBC real.

## Capítulo 14

1. Um class loader carrega e inicializa classes e interfaces na máquina virtual Java.
2. Um class loader de inicialização (*bootstrap classloader*) é implementado no código nativo e utilizado para carregar classes Java básicas, como `java.lang.Object` e `java.lang.ClassLoader`.
3. Um class loader de extensões é utilizado para carregar bibliotecas de extensão, em geral, a partir do diretório de JRE `lib/ext`. Arquivos JAR nesse diretório tornam-se automaticamente disponíveis para aplicações Java.
4. Classes que têm o mesmo nome e que foram carregadas pelo mesmo class loader.
5. Para o recarregamento de classes e decoração de bytecode em tempo de execução, e para fornecer uma separação limpa entre componentes lógicos (como aplicações Web) executados na mesma JVM.
6. Não, devido a considerações de segurança.
7. Porque `DecoratingClassLoader` redefine o método `findClass()`, que só é chamado se a classe não for localizada pela hierarquia de class loaders. Se a classe for localizada em `CLASSPATH`, ela é carregada pelo class loader de aplicação.

## Capítulo 15

1. Para responder a essa pergunta, pense no seu trabalho profissional e veja se você teve de implementar uma maneira trabalhosa de contornar um problema ou um bug nas classes básicas.
2. O patching de classes básicas requer a manipulação do classpath de inicialização, porque as classes básicas são sempre carregadas pelo carregador de inicialização.
3. Porque o classpath de inicialização é o local em que o class loader de inicialização obtém a lista de caminhos de onde carregar as classes de sistema.

## Capítulo 16

1. Recomenda-se capturar `java.lang.Throwable`, porque os erros de sistema são informados como exceções não-verificadas. Uma boa solução é ter um bloco de

*try-catch* no topo da pilha de chamadas, nos principais threads de aplicação, que registre em log exceções verificadas e não-verificadas.

2. Um stream personalizado pode ser codificado para persistir os dados em um banco de dados. Então o stream de erros do sistema pode ser redirecionado para esse stream personalizado, utilizando `System.setErr()`.
3. Instalando-se um gerenciador de segurança que lança uma exceção de segurança a partir de seu método `checkExit()`.
4. A aplicação Java pode adicionar um gancho de desligamento (*shutdown hook*). Quando o gancho for chamado, a aplicação fecha todas as conexões.
5. Eventos como carregamento de classes, entrada e saída de métodos, início e parada de threads, entre outros.

## Capítulo 17

1. Não há nenhuma dependência de se ter o código-fonte, capacidade de gerar/strumentar bytecode em tempo de execução e automação mais fácil.
2. Um opcode identifica uma instrução de JVM no bytecode. Para passar os parâmetros, os valores são colocados na pilha de operandos.
3. `(Ljava.lang.String;C)[Ljava.lang.Object]`.
4. `cp_info`, `field_info`, `method_info`, e `attribute_info`.
5. `JavaClass`, `Field`, `Method`, `ConstantPool`, `ClassGen`, `FieldGen`, `MethodGen`, `ConstantPoolGen`, `InstructionFactory`, `InstructionList` e `Instruction`.
6. Atributo de código.

## Capítulo 18

1. A JNI permite a invocação de métodos a partir de uma biblioteca nativa carregada dinamicamente. Ela fornece uma camada de abstração entre a JVM e o SO.
2. O método precisa ser declarado como `native` em uma classe Java, e a biblioteca nativa deve ser carregada no inicializador estático da classe. O utilitário `javah` deve ser utilizado para gerar um arquivo de cabeçalho (*header*) C; depois, o corpo do método deve ser codificado em um arquivo de implementação em C. O arquivo de implementação em C deve ser compilado e vinculado a uma DLL, que deve ser colocada no caminho binário do programa Java.
3. O patching de uma declaração de método Java é simples de usar e é portável, mas talvez não funcione se algum código da biblioteca nativa ainda precisar ser chamado. A substituição de bibliotecas nativas é tediosa, embora não seja muito difícil e dispense patching de baixo nível. Entretanto, ela pode não ser eficiente para DLLs com um grande número de funções exportadas, e pode não fornecer flexibili-

lidade suficiente ao chamar o código nativo original. O patching de código nativo oferece o máximo de controle e flexibilidade, mas é difícil de escrever porque o trabalho é feito no nível mais baixo; além disso, ela não é portável através de plataformas.

4. Seção .text.
5. Porque o código de máquina que implementa o patch é escrito pelo utilitário, sobrecrevendo o código da função original. O tamanho do código depende dos comprimento dos nomes e pode ser de até de 100 bytes, o que é maior do que muitas funções simples em C.
6. Sim, CALL pode ser utilizado, mas nesse caso os parâmetros precisam ser colocados na pilha novamente, para que o retorno da função de substituição seja executado corretamente. Utilizar um CALL em vez de um JMP também requer uma instrução RETN no código de desvio para devolver o controle ao chamador.
7. A Detours é mais confiável e deve funcionar em todas as versões de JDK e arquivos PE. Ela oferece muito mais flexibilidade, através da interceptação de funções na memória e a capacidade de chamar a função original via um *trampoline*.
8. COFF e ELF.
9. Utilize a mesma abordagem que a usada no Windows. Localize o código binário no arquivo executável baseado no formato de arquivo. Desassemble o código da função e implemente um desvio na linguagem assembly. O desvio pode delegar para a nova lógica acrescentada no mesmo arquivo, ou a uma função de uma biblioteca compartilhada carregada dinamicamente.

## Capítulo 19

1. Um resumo de mensagem (*message digest*) só protege a integridade da mensagem; a encriptação protege o conteúdo e a assinatura protege a integridade da mensagem e assegura a autenticidade do remetente por uma autoridade Certificate Authority (CA).
2. Algoritmos simétricos utilizam a mesma chave para encriptação e decriptação; algoritmos assimétricos utilizam um par de chaves (pública e privada) para fornecer encriptação unidirecional.
3. Utilize a encriptação para converter o corpo do e-mail em um conteúdo irreconhecível e em seguida aplique a codificação base64 para converter bytes em caracteres imprimíveis. Você deve utilizar `javax.crypto.Cipher` para o algoritmo, uma classe concreta que implementa `java.security.spec.KeySpec` para a cifra selecionada e `javax.crypto.SecretKeyFactory` para converter a especificação da chave em uma chave.
4. Ofuscamento, selagem dos arquivos JAR de aplicações, garantia da origem de classes críticas, e proteção dos arquivos de distribuição verificando o checksum.

5. Utilizar um algoritmo assimétrico é mais seguro, porque um checksum pode ser calculado usando a chave privada e verificado com a chave pública. Dessa maneira, ninguém, exceto o fornecedor, pode emitir o checksum. Na proteção do Chat de exemplo, o algoritmo de resumo de mensagem é fornecido junto com os parâmetros que podem ser obtidos a partir do código descompilado.
6. Um par de chaves precisa ser gerado ou obtido para o algoritmo especificado. Gerar uma assinatura requer uma instância de `PrivateKey` para a chave privada, e uma instância de `Signature` para o algoritmo especificado. A instância da classe `Signature` recebe a chave e os dados a serem assinados; em seguida, o método `sign()` é chamado para obter a assinatura.
7. Verificar a assinatura requer uma instância da classe `PublicKey` e uma classe `Signature` para o algoritmo dado. A instância do algoritmo recebe a chave pública e os dados, então seu método `verify()` é chamado, recebendo os dados da assinatura como parâmetro.

**PÁGINA EM BRANCO**

# Índice

## Símbolos

+ (sinal de adição), operador  
concatenação de string, 60

## A

**About, caixa de diálogo, 7**

### acesso

JavaDoc, 67  
membros de classe privados, 41, 42, 43, 44  
membros de classe protegidos, 39, 40, 41  
pacotes, 39, 40, 41  
variáveis de ambiente, 70

**Add, comandos de menu**

Thread Group, 101

**ajustando bytecode, 195**

### algoritmos

assimétricos  
aplicação Chat segura, 188  
assimétricos (uma via), 186  
cifra  
aplicação Chat segura, 188, 189, 190, 191  
encriptação/decriptação, 186  
resumo de mensagem, 185, 195  
SHA-1 (Secure Hash Algorithm), 197  
segurança, 189  
simétricos (duas vias), 186

**alocação**

objetos  
profiling, 82, 83, 84, 85, 86  
threads, 86, 87, 88, 89

**ambiente de tempo de execução, 66, 67**

ambiente de memória, 68  
gerenciamento de memória, 69  
informações de rede, 69, 70  
informações de sistema  
localizando, 68  
propriedades de sistema, 67, 68  
variáveis  
acessando, 70

**ambientes de desenvolvimento integrado (IDEs), 6**

**análise de aplicação em tempo de execução**

profilers, 2

**AOP (Aspect Oriented Programming)**

versus bytecode, 166

**Apache Log4J, API de logging, 58, 59**

### APIs

API de reflection  
membros de classe privados, acessando, 41, 42, 43, 44  
APIs de logging, 58, 59  
Debugger API, 36  
Java Debug, 73  
JVMP (Java Virtual Machine Profiler Interface), 147  
logging, 57  
segurança, 63

**aplicação, lógica da**

fornecendo  
patching de classes, 53  
navegando  
patching de aplicações Chat, 52

**aplicações**

AWT Java  
convertendo em HTML, 39  
Chat  
dump de thread, 90, 91  
ofuscamento, ofuscador Zelix KlassMaster, 32, 33, 34, 35  
patching, 49  
patching, navegação lógica da aplicação, 52  
patching, nomes de classe, 50, 51  
patching, pesquisa de string de texto, 51, 52  
class loader, 126, 127  
configurando  
carregamento de classes corrigidas com patch, 53, 54  
demo  
aplicação WebCream, 101

- desconhecidas
  - depuração, 72
  - depuração, convencional, 73
  - depuração, Java Debug API, 73
  - depuração, onisciente, 73, 74, 75, 76, 77, 78
- investigando durante tempo de execução
  - dump de thread, 90, 91
- JAR
  - assinaturas digitais, 192
- lado do servidor
  - requisitos no nível de serviço, teste de carga, 93, 94
- manutenção
  - solução de problemas (ofuscamento), 31
  - solucionando problemas (ofuscamento), 32
- P6Spy, 123
- proteção
  - HTTP (Hypertext Transfer Protocol), 118
  - licenciamento, ativando, 208
- protogendo
  - arquivos de licença, criando, 206
- RMI (Remote Method Invocation)
  - proteção contra espionagem, 121, 122
- solucionando problemas. *Ver também* rastreando, 56
- Swing
  - convertendo em HTML, 39
- WebCream
  - testando, 101, 103, 104, 105, 106, 107
  - tests, 102
- aplicações comerciais**
  - proteção contra hacking, 3
- aplicações.** *Ver também Chat, aplicação (aplicação de exemplo), 7*
- armazenando**
  - classes corrigidas com patch, 53
- arquivando**
  - FAR (File and Archive Manager), 4, 5
- arquivos**
  - classe, 152
    - atributo, 157
    - campo, 154, 155, 156
    - descritores de campo, 152, 153
    - descritores de método, 152, 153
    - estrutura, 156
    - fluxo binário, 154
  - verificação de bytecode, 157
  - visualizando, jClassLib Bytecode Viewer, 150, 151
- CMD.EXE, 4
- configuração
  - hacking (aplicação Chat), 113
- executáveis
  - formato PE (Executable Portable), 174
- JAR
  - selando, 192
- licença
  - criando, 206
- licenciamento
  - criando, 202, 203, 204, 205
  - verificando, 206, 207, 208
- bat , 4
- segurança
  - arquivos de diretiva, 43
- arquivos de diretiva**
  - segurança, 43
- ASCII**
  - codificação base64, 197
- Aspect Oriented Programming (AOP)**
  - versus bytecode, 166
- assembly, linguagem, 175**
- Assertions, nó (planos de teste JMeter), 100**
- assinando**
  - assinaturas digitais, 186
- assinaturas digitais**
  - aplicações JAR, 192
  - assinando, 186
  - licenciamento, 201, 203, 204, 205, 206
- ataques**
  - recusa de serviço
    - teste de carga, 94
- ataques de recusa de serviço**
  - teste de carga, 94
- ativando**
  - licenças, 208
- atributos**
  - arquivos de classe, 157
- autenticação**
  - APIs de segurança, 63
- autorização**
  - APIs de segurança, 63

**B****bancos de dados**

drivers JDBC (Java Database Connectivity)  
eavesdropping, 122, 123

**barras de ferramentas**

Stamp (Omniscient Debugger), 74

**barras de menu****bat**

(bibliotecas compartilhadas), 125  
(bibliotecas de vínculo dinâmico), 125  
arquivos, 4  
aplicação Chat, 7

**BCEL (Byte Code Engineering Library), 158**

site Web, 219

**biblioteca de jClassLib**

Bytecode Viewer  
depurando bytecode, 150  
visualizando arquivos de classe, 151

**bibliotecas**

BCEL (Byte Code Engineering Library), 158  
compartilhadas (.so), 125  
dinamicamente vinculadas (.dll), 125

DLL (Dynamically Linked Library)  
criando, 172

**extensão**

carregamento, 126

**jClassLib**

Bytecode Viewer, depurando  
bytecode, 150  
Bytecode Viewer, visualizando arquivos  
de classe, 150, 151

Microsoft Detours, 179, 180, 181

**nativas**

carregamento, 170  
métodos nativos, patching, 173

**bloqueando**

clientes virtuais, 135

**Borland**

profiler OptimizeIt Suite, 80

**Bouncy Castle, site Web, 220****Byte Code Engineering Library (BCEL), 158****Byte Code Engineering Library (site Web  
BCEL), 219****bytecode, 3**

arquivos de classe, 152  
atributos, 157  
campos, 154, 155, 156  
descritores de campo, 152, 153

descritores de método, 152, 153

estruturas, 156

fluxos binários, 154

verificação de bytecode, 157

**BCEL (Byte Code Engineering Library), 158**

benefícios, 149, 150

classes

gerando, 163, 164, 165

depurando

Bytecode Viewer, 150

descompilando, 9, 17, 18, 191, 192

hacking, 192

ajustando, 195

arquivos JAR, 192

class loaders personalizados, 194

classes, 193, 194

gerenciadores de segurança, 194

recursos, 194

**jClassLib Bytecode Viewer**

visualizando arquivos de classe, 150, 151

**JVM (Java Virtual Machine)**

conjunto de instruções, 150, 152

métodos

instrumentando, 159, 160, 161, 162, 163

ofuscamento, 19, 20

versus AOP (Aspect Oriented

Programming), 166

versus proxies dinâmicos, 166

visão geral, 149, 150

**Bytecode Viewer**

arquivos de classe

visualizando, 150, 151

bytecode

depurando, 150

**C****caixas de diálogo**

About, 7

Capture Options, 120

Find, 176

Find Frame, 121

Open Classes, 32

Search, 121

**caixas.** Ver também caixas de diálogo, 32

**campos**

arquivos de classe, 154, 155, 156

descritores

arquivos de classe, 152, 153

**Capture Options, caixa de diálogo, 120**

**Capture, comandos de menu**

Start, 120

**carregadores. Ver class loaders, 129**

**carregamento**

bibliotecas de extensão, 126

bibliotecas nativas, 170

class loader, 126

classe dinâmica

solucionando problemas  
(ofuscamento), 29

classes

controlando, 3

classes corrigidas com patch, 53, 54

classes dinâmicas

solucionando problemas  
(ofuscamento), 29

DLL, 178

**chamadas**

espaçamento de chamadas (teste de carga),  
96

interceptando

método System.exit(<|>), 142, 143

**Chat, aplicação**

(aplicação de exemplo), 7, 8

caixa de diálogo About, 7

carregamento, 7

diagrama de classe, 7

diretórios, 8

dump de thread, 90, 91

hacking elementos/recursos de UI (user interface)

arquivos de configuração, 113

imagens, 111, 112, 113

texto, 110, 111

menu barra, 7

ofuscamento

ofuscador Zelix KlassMaster, 32, 33,  
34, 35

patching, 49

navegação lógica da aplicação, 52

nomes de classe, 50, 51

pesquisa de string de texto, 51, 52

teste de carga, 95, 96, 97, 98

tornando segura

JCA (Java Cryptography Architecture),  
187, 188, 189, 190, 191

**Chat, aplicações**

código de processamento de mensagem

(depuradores oniscientes), 75, 76, 77

mensagens entrantes

processando (depuradores oniscientes),  
74, 75

ofuscamento (depuradores oniscientes),  
77, 78

**chaves privadas**

algoritmos assimétricos, 186

**chaves públicas**

algoritmos assimétricos, 186

**class loader**

JVM (Java Virtual Machine), 125, 126

**class loaders**

aplicação, 126, 127

BCEL (Byte Code Engineering Library), 158

carregamento, 126

definidores, 128

extensões, 126, 127, 128

JVM (Java Virtual Machine), 127, 128

hierarquia, 126, 127, 128

partida (bootstrap), 126, 127, 128

localizando classes básicas, 136

personalizados

delegação, 131

gravação, 129, 130, 131, 132, 133

hacking do bytecode, 194

hierarquia, 131

instanciando, 131, 132, 133

leitura de dados binários, 130

WebCream, 129

sistema, 126

**Class View, comando (menu**

**Snapshot), 85**

**classes**

aplicação

patching, 2

aplicando patch/substituindo

decidindo quando, 45, 46, 47

auxiliando

membros visíveis pelo pacote,

acessando, 39, 40

básicas

localizando, 136

patching, 3, 134, 135

patching, classpath de inicialização,  
135, 136

patching, exemplo java.lang.Integer,  
136, 137, 138

BCEL (Byte Code Engineering Library), 158

carregamento  
    controlando, 3  
classpath de inicialização  
    patching de classes básicas, 135, 136  
corrigidas com patch  
    armazenando, 53  
    carregamento, 53  
    loading, 54  
dados binários  
    leitura, 130  
descompilando, 1, 12, 13, 14, 15, 16, 17  
diagramas  
    aplicação Chat, 7  
dinâmica  
    carregamento, solucionando problemas  
        (ofuscamento), 29  
gerando (bytecode), 163, 164, 165  
hacking, 1  
hacking do bytecode, 193, 194  
java.lang.Integer  
    patching, 136, 137, 138  
localizando para aplicar patch, 47  
    abordagens gerais, 48  
    códigos ofuscados, 49  
    pesquisa de string de texto, 48, 49  
membros privados  
    acessando, 41, 42, 43, 44  
membros protegidos  
    acessando, 39, 40, 41  
nomes  
    patching de aplicações Chat, 50, 51  
ofuscando, 1  
patching  
    fornecendo lógica da aplicação, 53  
percorrendo, 48  
sistema  
    adicionando a pacotes, 41  
snoop  
    criando, 66  
teste  
    criando, 137

**CLASSPATH**  
    classes corrigidas com patch  
        carregamento, 53, 54

**classpaths de inicialização**  
    patching de classes básicas, 135, 136

**clientes**  
    concorrentes, 96  
    simultâneos, 96

virtual  
    bloqueando, 135

**CMD.EXE, arquivo, 4**

**code**  
    bytecode  
        descompilando, 9

**Code, painel (Omniscient Debugger), 75**

**codificação**  
    base64, 197  
    strings  
        ofuscamento, 26

**codificação base64, 197**

**codificando**  
    strings  
        ofuscamento, 25, 26

**código**  
    bytecode, 3  
        descompilando, 17, 18  
        ofuscamento, 19, 20  
    corrompido  
        inserindo (ofuscamento), 28  
    depuração, 2  
    encolhendo  
        ofuscamento, 28  
    máquina  
        formato PE (Executable Portable), 175  
    ofuscado  
        cracking, 36  
    ofuscador  
        cracking, 36  
    ofuscamento  
        patching de classe, 49  
    operação (opcode)  
        JVM (Java Virtual Machine), instruções, 150  
    otimizando  
        ofuscamento, 28  
    patching, 3  
    patching (rastreando), 57  
    processamento de mensagem  
        aplicações Chat, 75, 76, 77

**código nativo**  
    patching, 168, 169  
        JVM (Java Virtual Machine), 169  
        JVM (Java Virtual Machine),  
            implementação do JNI, 171, 172  
        JVM (Java Virtual Machine), visão geral  
            do JNI, 169, 171  
        métodos nativos, 173

- métodos nativos, bibliotecas nativas, 173, 174  
métodos nativos, declarações, 173  
Unix, 181, 182  
Windows, 174  
Windows, biblioteca Microsoft Detours, 179, 180, 181  
Windows, formatos Portable Executable (PE), 174, 175, 177  
Windows, utilitário Function Replacer, 177, 178, 179
- código.** Ver também **código nativo**
- bytecode
  - código nativo, 12
- código-fonte**
- MessageInfoComplex, 12, 13, 14
- código-fonte aberto, software comercial, 200**
- códigos ofuscados**
- cracking, 36
  - patching de classe, 49
- COFF (Common Object File Format), 174, 181**
- coleta de lixo**
- gerenciamento de memória, 68
  - profiling, 80, 81, 82
  - solicitando, 84
- comandos**
- Capture menu
    - Start, 120
  - menu Edit
    - Find Frame, 121
  - menu Program
    - Take Heap Snapshot, 85
  - menu Search
    - Find, 176
  - menu Snapshot
    - Class View, 85
  - menu Sniffer
    - filtro, 118
  - menu Tools
    - ZKM Script Helper, 33

- patching, 173
- declarando**  
métodos nativos, 170
- decompilando**  
limitações, 19, 20
- criptação, 185**  
algoritmos de encriptação/decriptação, 186
- delegação**  
class loaders personalizados, 131
- depuração**  
código, 2
- depuração local**  
Debugger API, 36
- depuração remota**  
Debugger API, 36
- depuração, mensagens. Ver também rastreando, 56**
- depuradores**  
aplicações desconhecidas, 72  
código ofuscado pelo fluxo, 36  
convencionais, 73  
Java Debug API, 73  
Omniscient, 73  
oniscientes  
    aplicações Chat, código de processamento de mensagem, 75, 76, 77  
    aplicações Chat, ofuscamento, 77, 78  
    aplicações Chat, processando mensagens entrantes, 74, 75
- depuradores convencionais, 73**
- depuradores oniscientes**  
aplicações Chat  
    código de processamento de mensagem, 75, 77  
    ofuscamento, 77, 78  
    processando mensagens entrantes, 74, 75
- depurando**  
bytecode  
    Bytecode Viewer, 150
- DES (Data Encryption Standard), 188**
- descompilação**  
determinando, 9, 10
- descompiladores, 10, 12**  
GUIs (graphical user interfaces), 12
- descompilando**  
bytecode, 9, 17, 18, 191, 192  
classes, 12, 13, 14, 15, 16, 17  
direitos, 9  
hackers, 21
- limitações, 18, 21
- descompilando classes, 1**
- descritores**  
campo  
    arquivos de classe, 152, 153
- método  
    arquivos de classe, 152, 153  
    criando, 153
- DESede (multiple DEScipher algorithm, 188)**
- desempenho**  
otimização, 89, 90
- desligamento**  
ganchos  
    JVM (Java Virtual Machine), 144  
    JVM (Java Virtual Machine), 142, 143  
    ganchos de desligamento, 144
- destruição de nome**  
ofuscamento, 24
- destruição de nomes**  
ofuscamento, 24
- Detours Library, site Web, 220**
- Detours, biblioteca (Microsoft), 179, 180, 181**
- direitos**  
descompilando, 9
- diretivas**  
padrão  
    instalações de gerenciador de segurança (pulando verificações de segurança), 64, 65  
personalizadas  
    instalações de gerenciador de segurança (pulando verificações de segurança), 65
- diretivas padrão**  
instalações de gerenciador de segurança (pulando verificações de segurança), 64, 65
- diretivas personalizadas**  
instalações de gerenciador de segurança (pulando verificações de segurança), 65
- diretórios**  
aplicação Chat, 8  
classes corrigidas com patch  
    armazenando, 53
- distribuição**  
proteção de aplicação, 191  
conteúdo de aplicação, 195, 196, 197,

- 198, 199  
descompilando bytecode, 191, 192  
hacking do bytecode, 192  
hacking do bytecode, ajustando, 195  
hacking do bytecode, arquivos JAR, 192  
hacking do bytecode, classes, 193, 194  
hacking do bytecode, gerenciadores de segurança, 194  
hacking do bytecode, personalizando class loaders, 194  
hacking do bytecode, recursos, 194  
proteção de aplicações  
hacking do bytecode, classes, 193
- DLL (Dynamically Linked Library)**  
carregamento, 178  
criando, 172
- drivers**  
JDBC (Java Database Connectivity)  
eavesdropping, 122, 123
- DSA, gerador de pares de chaves, 204**
- dump de thread**  
investigação de aplicação durante tempo de execução, 90, 91
- Dynamically Linked Library (DLL)**  
criando, 172
- E**
- eavesdropping**  
drivers JDBC (Java Database Connectivity), 122, 123  
HTTP (Hypertext Transfer Protocol), 115  
proteção de aplicação, 118  
sniffers de rede, 117, 118  
túneis, 115, 116  
instruções de SQL, 122, 123  
protocolo RMI (Remote Method Invocation), 119  
fluxos, 120  
proteção de aplicação, 121  
sniffers de rede, 120, 121  
RMI (Remote Method Invocation)  
fluxos, 120  
proteção de aplicação, 122  
visão geral, 114
- Eclipse, site Web, 6**
- Edit, comandos de menu**  
Find Frame, 121
- EffeTech**  
HTPP Sniffer, 118
- Ehtereal, ferramenta de sniffing de rede, 120, 121**
- ej-technologies**  
JProfiler, 80
- elementos**  
UI (user interface)  
hacking, 109  
hacking, arquivos de configuração, 113  
hacking, imagens, 111, 112, 113  
hacking, texto, 110, 111
- ELF (Executable and Linking Format), 181**
- encapsulamento**  
limitações, 38
- encolhendo**  
código  
ofuscamento, 28
- criptação, 185**  
DES (Data Encryption Standard), 188
- criptação/decriptação, algoritmos, 186**
- End User License Agreement (EULA), 200**
- engenharia**  
reversa, 22, 23  
API de logging do Apache Log4J, 59
- engenharia reversa**  
API de logging do Apache Log4J, 59  
proteção contra, 22, 23
- env, prefixo**  
variáveis de ambiente, 70
- EP Explorer, site Web, 219**
- erros**  
linkagem  
interceptando, 140  
máquina virtual  
interceptando, 140  
sistema  
interceptando, 140
- escalabilidade**  
teste de carga, 2, 93, 94, 95  
aplicação Chat, 95, 96, 97, 98  
baseados em servidores RMI, 97, 98  
ferramenta JMeter, 98, 99  
ferramenta JMeter tool, planos de teste, 106  
ferramenta JMeter, aplicação  
WebCream, 101  
ferramenta JMeter, GUI, 101  
ferramenta JMeter, planos de teste, 99,

- 101, 102, 103, 104, 105, 106, 107  
ferramenta JMeter, visão geral, 99, 100  
ferramentas, 94  
JMeter ferramenta, visão geral, 101  
Mercury Load Runner, 99  
Rational Test Suite, 99  
servidores baseados em RMI, 95, 96
- escrevendo**  
rastreamentos, 59
- espaçamento de chamadas (teste de carga), 96**
- estados**  
tempo de execução  
pilhas de chamadas, 58
- estruturas**  
arquivos de classe, 156
- Ethereal, site Web, 219**
- EULA (End User License Agreement), 200**
- exceções**  
OutOfMemory, 82  
rastreando, 59  
saída (rastreando), 60
- Executable and Linking Format (ELF), 181**
- executáveis, arquivos**  
formato PE (Executable Portable), 174
- expressões**  
regulares  
JMeter, 105
- extensões**  
bibliotecas  
carregamento, 126  
class loaders, 126, 127, 128  
JCE (Java Cryptography Extensions), 187  
JSSE (Java Secure Sockets Extension), 122
- F**
- FAR (File and Archive Manager), 4, 5**
- FAR Web site, 216**
- ferramentas**  
JMeter, 98, 99  
aplicação WebCream, 101  
GUI, 101  
planos de teste, 99, 101, 102, 103, 104, 105, 106, 107  
visão geral, 99, 100, 101
- File and Archive Manager (FAR), 4, 5**
- Filter, comando (menu Sniffer), 118**
- Find Frame, caixa de diálogo, 121**
- Find Frame, comando (menu Edit), 121**
- Find Replacer, utilitário, 177**
- Find, caixa de diálogo, 176**
- Find, comando (menu Search), 176**
- fluxo de controle, 3**  
interceptando, 139  
chamadas System.exit(<>), 142, 143  
erros de sistema, 140  
fluxos de sistema, 140, 141, 142  
JVM (Java Virtual Machine),  
desligamento, 144  
JVMP (Java Virtual Machine Profiler  
Interface), 147  
métodos, 144, 145, 146, 147
- ofuscamento**  
programas, 27  
programando, 139  
programas  
ofuscamento, 26, 27  
visão geral, 139
- fluxos**  
binário  
arquivos de classe, 154  
protocolo RMI (Remote Method  
Invocation)  
eavesdropping, 120  
sistema  
interceptando, 140, 141, 142
- fluxos de entrada (input stream)**  
protocolo RMI (Remote Method  
Invocation)  
eavesdropping, 120
- fluxos de saída**  
protocolo RMI (Remote Method  
Invocation)  
eavesdropping, 120
- fluxos de sistema**  
interceptando, 140, 141, 142
- formatos de arquivo**  
COFF (Common Object File Format), 174, 181  
ELF (Executable and Linking Format), 181  
PE (Portable Executable), 174, 175, 176, 177
- formatos. Ver formatos de arquivo, 174**
- funções**  
getter, 38  
setter, 38  
trampoline, 179

**Function Replacer, site Web, 220**

**Function Replacer, utilitário, 177, 178, 179**

## G

**ganchos**

ganchos de desligamento

JVM (Java Virtual Machine), 144

**gargalos**

otimização de desempenho, 90

**General Public License (GPL), 200**

**gerando**

classes (bytecode), 163, 164, 165

**gerenciadores**

segurança

instalando (JVM), 143

operações protegidas, 62

pulando verificações de segurança, 64

pulando verificações de segurança,

diretivas padrão, 64, 65

pulando verificações de segurança,

diretivas personalizadas, 65

**gerenciamento de arquivo, 3, 4**

FAR (File and Archive Management), 4, 5

IDEs (integrated development environments), 6

Total Commander, 4, 5

Windows Explorer, 4

**getter, funções, 38**

**GPL (General Public License), 200**

**gráfico de resumo de heap de tempo de execução, 80, 82**

**gráficos**

Reference Graphs, 85

resumo de heap de tempo de execução, 80, 82

**gravação**

class loaders personalizados, 129, 130, 131, 132, 133

**grupos de thread**

adicionando, 101

**GUIs (graphical user interfaces)**

descompiladores, 12

JMeter, 101

JUnit, 97

## H

**hacker**

descompilando, 21

**hacking**

API de logging do Apache Log4J, 59

bytecode, 192

ajustando, 195

arquivos JAR, 192

class loaders personalizados, 194

classes, 193, 194

gerenciadores de segurança, 194

recursos, 194

classes, 1

elementos/recursos de UI (user interface), 109

arquivos de configuração, 113

imagens, 111, 112, 113

texto, 110, 111

proteção contra, 22, 23

proteção de aplicação comercial, 3

**hash, 185**

**Hibernate, 151**

**hierarquia**

class loaders personalizados, 131

**hierarquias**

class loaders, 126, 127, 128

**HTML**

aplicações Java AWT

convertendo, 39

aplicações Swing

convertendo, 39

**HTTP Sniffer, 118**

**HTTP (Hypertext Transfer Protocol)**

eavesdropping, 115

proteção de aplicação, 118

sniffers de rede, 117, 118

túneis, 115, 116

**HTTP Sniffer, 118**

**HTTP Sniffer, site Web, 218**

**HTTPS (Hypertext Transfer Protocol Secure), 119**

**Hypertext Transfer Protocol (HTTP)**

eavesdropping, 115

proteção de aplicação, 118

sniffers de rede, 117, 118

túneis, 115, 116

tunnels, 116

**Hypertext Transfer Protocol Secure**

**(HTTPS), 119**

- I**
- IDA Pro, site Web**, 181, 220
  - IDEs (Integrated Development Environments)**, 6
  - IIOP (Internet Inter-Orb Protocol)**, 119
  - imagens**
    - hacking
      - aplicação Chat, 111, 112, 113
      - pesquisando, 111, 112, 113
  - impasse**
    - threads, 87, 88, 89
  - implementação**
    - licenciamento
      - arquivos de licença, criando, 206
  - implementando**
    - licenciamento, 200, 201, 202
      - arquivos de licença, criando, 202, 203, 204, 205
      - arquivos de licença, verificando, 206, 207, 208
  - imprimindo**
    - propriedades de sistema, 67
  - informações de depuração**
    - removendo
      - ofuscamento, 24
  - informações de sistema**
    - localizando
      - ambiente de tempo de execução, 68
  - inicialização**
    - classpath
      - patching de classes básicas, 136
  - instalações**
    - gerenciadores de segurança (JVM), 143
  - instanciando**
    - class loaders personalizados, 131, 132, 133
  - instantâneos do heap em tempo de execução**
    - alocação de objetos
      - navegando, 84, 85
  - instruções**
    - JVM (Java Virtual Machine), 150, 152
  - instruções de SQL**
    - eavesdropping, 122, 123
  - instrumentando**
    - métodos (bytecode), 159, 160, 161, 162, 163
  - interceptando**
- fluxo de controle**
  - chamadas System.exit(<|>), 142, 143
  - erros de sistema, 140
  - fluxos de sistema, 140, 141, 142
  - JVM (Java Virtual Machine), 144
  - JVMPPI (Java Virtual Machine Profiler Interface), 147
  - métodos, 144, 145, 146, 147
- interface com usuário. Ver UI (user interface), elementos**, 109
- interfaces gráficas com usuário. Ver GUIs (graphical user interfaces)**
- Internet Inter-Orb Protocol (IIOP)**, 119
- IP (intellectual property)**
  - proteção contra, 23, 24
- J**
- JAAS**
    - API de segurança, 63
  - JAD, descompilador**, 11
  - JAD, site Web**, 216
  - janelas**
    - Debug Controller
      - acessando, 75
  - JAR, aplicações**
    - assinaturas digitais, 192
  - JAR, arquivos**
    - selando, 192
  - Java AWT, aplicações**
    - convertendo em HTML, 39
  - Java Cryptography Architecture (JCA)**, 185, 186, 187
    - aplicação Chat, 187, 188, 189, 190, 191
    - visão geral, 187
  - Java Cryptography Extensions (JCE)**, 187
  - Java Database Connectivity (JDBC)**,
    - drivers**
      - eavesdropping, 122, 123
  - Java Debug API**, 73
  - Java Logging API (Sun)**, 58, 59
  - Java Remote Method Protocol (JRMP)**, 119
  - Java Secure Sockets Extension (JSSE)**, 122
  - Java Virtual Machine (JVM)**, 169
    - class loaders, 125, 126, 127, 128
      - hierarquia, 126, 127, 128
    - conjunto de instruções, 150, 152
    - desligando, 142, 143

- ganchos de desligamento, 144
- JNI**  
implementando, 171, 172  
visão geral, 169, 171  
utilitário Function Replacer, 179
- Java Virtual Machine Profiler Interface (JVMPi), 147**
- java.lang.Integer, classe**  
patching, 136, 137, 138
- JavaDoc**  
acessando, 67
- javah, utilitário, 171**
- JBuilder X, 185**
- JBuilder X Foundation, 185**
- JCA (Java Cryptography Architecture), 185, 186, 187**  
aplicação Chat, 187, 188, 189, 190, 191  
visão geral, 187
- JCE**  
API de segurança, 63
- JCE (Java Cryptography Extensions), 187**
- jClassLib Bytecode Viewer, site Web, 219**
- jClassLib, biblioteca**  
Bytecode Viewer  
visualizando arquivos de classe, 150
- JDBC (Java Database Connectivity), drivers**  
eavesdropping, 122, 123
- JMeter, ferramenta, 98, 99**  
aplicação WebCream, 101  
GUI, 101  
planos de teste, 99, 101, 102, 103, 104, 105, 106, 107  
visão geral, 99, 100, 101
- JMeter, site Web, 218**
- JNI**  
implementando, 171, 172  
visão geral, 169, 171
- JODE, descompilador, 11**
- JODE, site Web, 216**
- JProbe**  
profiler  
otimização de desempenho, 90  
site Web, 217
- JProbe Suite, profiler, 80**
- JProfiler, 80**
- JProfiler, site Web, 218**
- JRMP (Java Remote Method Protocol), 119**
- Jshrink, ofuscador, 29**
- JSSE (Java Secure Sockets Extension), 122**  
API de segurança, 63
- JUnit**  
aplicação Chat  
teste de carga, 95, 96, 97, 98  
limitações, 98  
servidores baseados em RMI  
teste de carga, 95, 96, 97, 98
- JUnit GUI, 97**
- JUnit, site Web, 218**
- JVM (Java Virtual Machine), 169**  
class loaders, 125, 126, 127, 128  
hierarquia, 126, 127, 128  
conjunto de instruções, 150, 152  
desligando, 142, 143  
ganchos, 144  
gerenciamento de memória, 68, 69  
implementando, 171, 172  
utilitário Function Replacer, 179  
visão geral, 169, 171
- JVMPi (Java Virtual Machine Profiler Interface), 147**
- L**
- leitura, propriedades de sistema, 67**
- libelf, site Web, 220**
- licenciamento, 199, 200**  
arquivos de licença  
criando, 202, 203, 204, 205, 206  
verificando, 206, 207, 208  
ativando, 208  
GPL (General Public License), 200  
implementando, 200, 201, 202  
registro, 208  
software comercial de código-fonte aberto, 200  
software comercial de código-fonte fechado, 200  
software gratuito de código-fonte aberto, 200
- linguagens**  
assembly, 175
- linkagem**  
erros  
interceptando, 140
- Listeners, (planos de teste JMeter), 100**
- Load Runner (Mercury), 99**

- loaders.** Ver **class loaders**, 129  
**LoadRunner, site Web**, 218  
**Locals, painel (Omniscient Debugger)**, 75  
**Log4J framework, site Web**, 217  
**Log4J, API de logging (Apache)**, 58, 59  
**logging, APIs**, 57  
**Logic controllers, nó (planos de teste JMeter)**, 100  
**lógica**  
 aplicação  
 fornecendo, patching de classes, 53  
 navegação, patching de aplicações Chat, 52  
**loops**  
 rastreando, 60
- M**
- manutenção**  
 aplicações  
 solucionando problemas (ofuscamento), 31, 32
- máquina virtual**  
 erros  
 interceptando, 140
- melhoras de desempenho**  
 coletas de lixo/uso de heap, 80, 81, 82
- membros de classe privados**  
 acessando, 41, 42, 43, 44
- membros de classe protegidos**  
 acessando, 39, 40, 41
- memória**  
 gerenciamento  
 JVM, 68  
 gerenciando  
 JVM, 69
- mensagem, algoritmo de resumo**, 195
- mensagens**  
 depuração. Ver também rastreando, 56
- Mercury Load Runner**, 99
- MessageInfoComplex, código descompilado**, 15, 16
- MessageInfoComplex, código-fonte**, 12, 13, 14
- Method Traces, painel (Omniscient Debugger)**, 75
- métodos**  
 descritores  
 arquivos de classe, 152, 153
- classe arquivos, 153  
 criando, 153  
 instrumentando (bytecode), 159, 160, 161, 162, 163  
 interceptando  
 proxy dinâmico, 144, 145, 146, 147  
 nativos  
 declarando, 170  
 patching, 173  
 patching, bibliotecas nativas, 173, 174  
 patching, declarações, 173  
 pilhas de chamadas  
 navegação lógica da aplicação, 52  
**System.exit(<>)**  
 interceptando chamadas, 142, 143
- Microsoft Detours, biblioteca**, 179, 180, 181
- Mocha, descompilador**, 11
- multiple DES (DESede), algoritmo de cifra**, 188
- N**
- navegação**  
 lógica da aplicação  
 patching de aplicações Chat, 52
- navegando**  
 alocação de objetos, 82, 83, 84, 85, 86
- NetBeans, site Web**, 6
- nó de grupo de thread (planos de teste JMeter)**, 100
- nomes**  
 classes  
 patching de aplicações Chat, 50, 51
- nós**  
 planos de teste  
 ferramenta JMeter, 100
- O**
- Objects, painel (Omniscient Debugger)**, 75
- objetos**  
 alocação  
 profiling, 82, 83, 84, 85, 86  
 persistentes, 82, 83
- objetos persistentes**, 83
- ODB (Omniscient Debugger)**, 73  
 aplicações Chat

- código de processamento de mensagem,  
     75, 76, 77  
 ofuscamento, 77, 78  
 processando mensagens entrantes, 74,  
     75
- Ofuscador ProGuard, 29**
- ofuscadores, 28, 30**
- Zelix KlassMaster
  - aplicação Chat, 32, 33, 34, 35
  - padrões de nomes, 35
  - personalizando, 32
- ofuscamento**
- aplicação Chat
  - ofuscador Zelix KlassMaster, 32, 33, 34,  
     35
  - aplicações Chat, 77, 78
  - bytecode, 19, 20
    - protegendo, 191
  - código
    - encolhendo, 28
    - otimizando, 28
  - código corrompido
    - inserindo, 28
  - destruição de nome, 24
  - engenharia reversa
    - proteção contra, 22, 23
  - fluxo de controle do programa, 26, 27
  - informações de depuração
    - removendo, 24
  - IP (Intellectual Property)
    - proteção de, 23, 24
  - recursos, 24
  - solucionando problemas
    - carregamento de classe dinâmica, 29
    - convenção para atribuição de nomes, 31
    - manutenção de aplicação, 31, 32
    - reflection, 30
    - serialização, 30
  - strings
    - codificando, 25, 26
  - Zelix ClassMaster, 19, 20
- ofuscamento de fluxo de controle**
- bytecode
    - protegendo, 191
- ofuscando classes, 1**
- OllyDbg, site Web, 220**
- Omniscient Debugger (ODB), 73**
- aplicações Chat
  - código de processamento de mensagem,  
     76, 77  
     ofuscamento, 77, 78  
     processando mensagens entrantes, 74, 75
- Omniscient Debugger, site Web, 217**
- Omniscient, depuradores, 73**
- Open Classes, caixa de diálogo, 32**
- operadores**
- + (sinal de adição)
  - concatenação de string, 60
- Optimizelt Suite, profiler, 80**
- Optimizelt Web site, 218**
- otimização**
- desempenho, 89, 90
  - rastreando, 57
- otimizando**
- código
    - ofuscamento, 28
- OutOfMemory, exceção, 82**

## P

**P6Spy, aplicação, 123**

**P6Spy, site Web, 219**

**pacotes**

- acessando, 39, 40, 41
- classes de sistema
- adicionando, 41

**pacotes selados**

- patching, 54, 55

**padrões de nomes**

- ofuscador Zelix KlassMaster, 35

**painéis**

- Omniscient Debugger, 74, 75

**parâmetros**

- rastreando, 59

**pares de chaves**

- algoritmos assimétricos, 186

**partida (bootstrap)**

- class loader, 127

**partida (bootstrap), class loader, 126, 128**

- classes básicas

- localizando, 136

**patching**

- aplicação Chat, 49

- navegação lógica da aplicação, 52

- nomes de classe, 50, 51

- pesquisas de string de texto, 51, 52

- classes

- decidindo quando, 45, 46, 47

- fornecendo lógica da aplicação, 53  
localizando, 47  
localizando, abordagens gerais, 48  
localizando, códigos ofuscados, 49  
localizando, pesquisa de string de texto, 48, 49  
classes básicas, 3, 134, 135  
    classpath de inicialização, 135, 136  
    exemplo java.lang.Integer, 136, 137, 138  
classes de aplicação, 2  
código, 3  
código (rastreando), 57  
código nativo, 168, 169  
    JVM (Java Virtual Machine), 169  
    JVM (Java Virtual Machine),  
        implementação do JNI, 171, 172  
    JVM (Java Virtual Machine), visão geral  
        do JNI, 169, 171  
métodos nativos, 173  
métodos nativos, bibliotecas nativas, 173, 174  
métodos nativos, declarações, 173  
Unix, 181, 182  
Windows, 174  
Windows, biblioteca Microsoft Detours, 179, 180, 181  
Windows, formato Portable Executable (PE), 174, 175, 176, 177  
Windows, utilitário Function Replacer, 177, 178, 179  
pacotes selados, 54, 55
- PBEWithMD5AndDES, algoritmo de cifra, 188, 189**
- PE (Executable Portable), formato, 174, 175, 176, 177**
- PE Explorer, utilitário, 175, 176**
- percorrendo**  
    classes, 48
- perfis**  
    análise de aplicação em tempo de execução, 2
- permissões, 61, 62, 63**  
    descompilando, 9  
    diretivas padrão  
        gerenciador de segurança, 64, 65  
    diretivas personalizadas  
        gerenciador de segurança, 65
- persistentes, objetos, 82**
- pesquisas**  
    imagens, 111, 112, 113  
    strings de texto  
        patching de aplicações Chat, 51, 52  
        patching de classe, 48, 49
- pilhas**  
    chamadas de método  
        navegação lógica da aplicação, 52  
    pilhas de chamadas  
        estados de tempo de execução, 58
- pilhas de chamadas**  
    estados de tempo de execução, 58  
    navegação lógica da aplicação, 52
- planos de teste**  
    JMeter, 99, 101, 102, 103, 104, 105, 106, 107
- Portable Executable (PE), formato, 174, 175, 176, 177**
- Post-processors, nó (planos de teste JMeter), 100**
- Preprocessors, nó (planos de teste JMeter), 100**
- profilers**  
    JVMPI (Java Virtual Machine Profiler Interface), 147
- profiling**  
     alocação de objetos, 82, 83, 84, 85, 86  
     alocação de threads, 86, 87, 88, 89  
     coleta de lixo, 80, 81, 82  
     investigação de aplicação durante tempo de execução  
        dump de thread, 90, 91  
     JProfiler, 80  
     objetos persistentes, 82, 83  
     otimização de desempenho, 89, 90  
     profiler JProbe Suite, 80  
     profiler Optimizelt Suite, 80  
     sincronização de threads, 86, 87, 88, 89  
     uso de heap, 80, 81, 82  
     visão geral, 79, 80
- Program, comandos de menu**  
    Request Garbage Collection, 84  
    Take Heap Snapshot, 85
- programação**  
    AOP (Aspect Oriented Programming)  
        versus bytecode, 166
- programando**  
    fluxo de controle, 139
- programas**

- fluxo de controle
  - ofuscamento, 27
- ProGuard, ofuscador, 29**
- ProGuard, site Web, 217**
- propriedade intelectual (IP)**
  - proteção contra, 23, 24
- propriedades**
  - sistema
    - ambiente de tempo de execução, 67
- proteção de aplicação**
  - arquivos de licença
    - arquivos, criando, 202
  - distribuição, 191
    - conteúdo de aplicação, 195, 196, 197, 198, 199
    - descompilando bytecode, 191, 192
    - hacking do bytecode, 192
    - hacking do bytecode, ajustando, 195
    - hacking do bytecode, arquivos JAR, 192
    - hacking do bytecode, classes, 193, 194
    - hacking do bytecode, gerenciadores de segurança, 194
    - hacking do bytecode, personalizando class loaders, 194
    - hacking do bytecode, recursos, 194
  - JCA (Java Cryptography Architecture), 185, 186, 187
    - aplicação Chat, 187, 188, 189, 190, 191
    - visão geral, 187
  - licenciamento, 199, 200
    - arquivos de licença, criando, 202, 203, 204, 205
    - arquivos de licença, verificando, 206, 207, 208
    - ativando, 208
    - implementando, 200, 201
    - registro, 208
    - software comercial de código-fonte aberto, 200
    - software comercial de código-fonte fechado, 200
    - software gratuito de código-fonte aberto, 200
    - objetivos, 184, 185
- proteção de aplicações**
  - licenciamento
    - arquivos de licença, criando, 206
- protetendo aplicações**
  - distribuição, 191
- conteúdo de aplicação, 195, 196, 197, 198, 199
  - descompilando bytecode, 191, 192
  - hacking do bytecode, 192
  - hacking do bytecode, ajustando, 195
  - hacking de bytecode, arquivos JAR, 192
  - hacking do bytecode, classes, 193, 194
  - hacking do bytecode, gerenciadores de segurança, 194
  - hacking do bytecode, personalizando class loaders, 194
  - hacking do bytecode, recursos, 194
- JCA (Java Cryptography Architecture), 185, 186, 187
  - aplicação Chat, 187, 188, 189, 190, 191
  - visão geral, 187
- licenciamento, 199, 200
  - arquivos de licença, criando, 202, 203, 204, 205
  - arquivos de licença, verificando, 206, 207, 208
  - ativando, 208
  - implementando, 200, 201, 202
  - registro, 208
  - software comercial de código-fonte aberto, 200
  - software comercial de código-fonte fechado, 200
  - software gratuito de código-fonte aberto, 200
  - objetivos, 184, 185
- protocolos**
  - HTTP (Hypertext Transfer Protocol)
    - eavesdropping, 115
    - eavesdropping, proteção de aplicação, 118
    - eavesdropping, sniffers de rede, 117, 118
    - eavesdropping, túneis, 115, 116
  - HTTPS (Hypertext Transfer Protocol Secure), 119
  - IOP (Internet Inter-Orb Protocol), 119
  - JRMP (Java Remote Method Protocol), 119
  - RMI (Remote Method Invocation)
    - eavesdropping, 119
    - eavesdropping, fluxos, 120
    - eavesdropping, proteção de aplicação, 121, 122

- eavesdropping, sniffers de rede, 120, 121
- TCP/IP (Transmission Control Protocol/Internet Protocol), 119
- proxy dinâmico**
  - interceptando métodos, 144, 145, 146, 147
  - métodos
    - interceptando, 144, 145, 146, 147
    - versus bytecode, 166
- pulando verificações de segurança, 63**
  - instalações de gerenciador de segurança, 64
  - diretivas padrão, 64, 65
  - diretivas personalizadas, 65
- Q**
  - Quest Software**
    - profiler JProbe Suite, 80
- R**
  - rastreamentos**
    - escrevendo, 59
    - inserindo, 56
  - rastreando, 2**
    - APIs de logging, 58, 59
    - exceções, 59
    - logging APIs, 57
    - loops, 60
    - níveis, 59
    - otimização, 57
    - parâmetros, 59
    - patching de código, 57
    - regras, 59, 60
    - saída de exceção, 60
    - software de aprendizagem, 57
    - variáveis, 59
    - visão geral, 56, 58
  - Rational Test Suite, 99**
  - recursos**
    - hacking do bytecode, 194
    - UI (user interface)
      - hacking, 109
      - hacking, arquivos de configuração, 113
      - hacking, imagens, 111, 112, 113
      - hacking, texto, 110, 111
  - rede**
    - informações
      - localizando, ambiente de tempo de execução, 70
  - rede, sniffers**
    - HTTP (Hypertext Transfer Protocol) eavesdropping, 117, 118
    - RMI (Remote Method Invocation) eavesdropping, 120, 121
  - redes**
    - informações
      - localizando, ambiente de tempo de execução, 69
  - Reference Graphs, 85**
  - reflection**
    - proxy dinâmico
      - interceptando métodos, 144, 145, 146, 147
    - solucionando problemas (ofuscamento), 30
  - registrando**
    - licenças, 208
  - regras**
    - rastreando, 59, 60
  - Remote Method Invocation (RMI), protocolo**
    - eavesdropping, 119
    - fluxos, 120
    - proteção de aplicação, 121, 122
    - sniffers de rede, 120, 121
  - removendo**
    - informações de depuração
      - ofuscamento, 24
  - Request Garbage Collection, comando (menu Program), 84**
  - Retro Guard, ofuscador, 29**
  - RetroGuard, site Web, 217**
  - RMI (Remote Method Invocation), protocolo**
    - eavesdropping, 119
    - fluxos, 120
    - proteção de aplicação, 121, 122
    - sniffers de rede, 120, 121
  - S**
    - saída**
      - exceções (rastreando), 60
    - Samplers, nó (planos de teste JMeter), 100**
    - Search, caixa de diálogo, 121**
    - Search, comandos de menu**

- Find, 176
- segurança**
- APIs, 63
  - aplicações comerciais
    - proteção contra hacking, 3
  - HTTPS (Hypertext Transfer Protocol Secure), 119
  - manipulando, 2
  - operações protegidas, 62
  - permissões, 61, 62, 63
  - proteção de aplicação
    - HTTP (Hypertext Transfer Protocol), 118
  - protetendo aplicações
    - distribuição, 191
    - distribuição, conteúdo de aplicação, 195, 196, 197, 198, 199
    - distribuição, descompilando bytecode, 191, 192
    - distribuição, hacking do bytecode, 192, 193, 194, 195
    - JCA (Java Cryptography Architecture), 185, 186, 187
    - JCA (Java Cryptography Architecture), aplicação Chat, 187, 189, 190, 191
    - JCA (Java Cryptography Architecture), visão geral, 187
    - licenciamento, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208
    - objetivos, 184, 185
  - protetendo aplicações
    - JCA (Java Cryptography Architecture), aplicação Chat, 188
    - licenciamento, registro, 208
  - pulando verificações, 63
    - instalações de gerenciador de segurança, diretivas padrão, 64, 65
  - senhas
    - sementes (algoritmos de segurança), 189
    - visão geral, 61, 62, 63
- segurança, gerenciadores**
  - hacking do bytecode, 194
  - instalando (JVM), 143
  - membros de classe privados
    - acessando, 41, 43, 44
  - operações protegidas, 62
  - pulando verificações de segurança, 64
    - diretivas padrão, 64, 65
- diretivas personalizadas, 65

- OllyDbg, 220  
Omniscient Debugger, 217  
OptimizeIt, 218  
P6Spy, 219  
PE Explorer, 219  
ProGuard, 217  
RetroGuard, 217  
TCPMon (Apache AXIS), 218  
Total Commander, 216  
Verisign, 187  
WebCream, 216  
Zelix KlassMaster, 217
- Snapshot, comandos**  
Class View, 85
- Sniffer, comandos de menu**  
filtro, 118
- sniffers de rede**  
HTTP (Hypertext Transfer Protocol)  
eavesdropping, 117, 118  
RMI (Remote Method Invocation)  
eavesdropping, 120, 121
- snoop, classe**  
criando, 66
- software**  
aprendendo via rastreamento, 57  
comercial de código-fonte aberto, 200  
comercial de código-fonte fechado, 200  
gratuito de código-fonte aberto, 200  
TCPMON, 115, 116
- solucionando problemas**  
aplicações. *Ver também rastreando*, 56  
escalabilidade  
teste de carga, 93, 94, 95  
teste de carga, aplicação Chat, 95, 96, 97, 98  
teste de carga, ferramenta JMeter, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107  
teste de carga, ferramentas, 94  
teste de carga, Mercury Load Runner, 99  
teste de carga, Rational Test Suite, 99  
teste de carga, servidores baseados em RMI, 95, 96, 97, 98  
ofuscamento  
carregamento de classe dinâmica, 29  
convenção para atribuição de nomes, 31  
manutenção da aplicação, 31, 32  
reflection, 30  
serialização, 30
- threads  
impasse, 87, 88, 89  
vazamentos de memória, 82, 83, 84, 85, 86
- soquetes seguros**  
APIs de segurança, 63
- SourceForge**  
aplicação P6Spy, 123
- Stack, painel (Omniscient Debugger), 75**
- Stamp, barra de ferramentas (Omniscient Debugger), 74**
- Start, comando (menu Capture), 120**
- strings**  
codificando  
ofuscamento, 25, 26  
concatenação (operador +), 60  
texto  
pesquisando, patching de classe, 48, 49  
pesquisas, patching de aplicações Chat, 51, 52
- substituindo**  
classes  
decidindo quando, 45, 46, 47
- Sun Java Logging API, 58, 59**
- Swing, aplicações**  
convertendo em HTML, 39
- System.exit(), método**  
interceptando chamadas, 142, 143
- T**
- Take Heap Snapshot, comando (menu Program), 85**
- TCP/IP (Transmission Control Protocol/Internet Protocol), 119**
- TCPMon (site Web do AXIS Apache), 218**
- TCPMON, software de tunelamento, 115, 116**
- tempo de execução**  
investigação de aplicação  
dump de thread, 90, 91
- tempo de execução, ambientes, 2**
- tempo de execução, estados**  
pilhas de chamadas, 58
- testando**  
instrumentação de bytecode, 162  
teste de carga  
ataques de recusa de serviço, 94  
escalabilidade, 2  
espacamento de chamadas, 96

- teste de carga (escalabilidade), 93, 94, 95  
  aplicação Chat, 95, 96, 97, 98  
  ferramenta JMeter, 98, 99  
  ferramenta JMeter, aplicação  
    WebCream, 101  
  ferramenta JMeter, GUI, 101  
  ferramenta JMeter, planos de teste, 99,  
    101, 102, 103, 104, 105, 106, 107  
  ferramenta JMeter, visão geral, 99, 100,  
    101  
  ferramentas, 94  
JMeter tool, planos de teste, 106  
Mercury Load Runner, 99  
Rational Test Suite, 99  
servidores baseados em RMI, 95, 96, 97,  
  98
- teste de carga**  
ataques de recusa de serviço, 94  
escalabilidade, 2  
espaçamento de chamadas, 96
- teste de carga (escalabilidade), 93, 94, 95**  
aplicação Chat, 95, 96, 97, 98  
ferramenta JMeter, 98, 99  
  aplicação WebCream, 101  
  GUI, 101  
  planos de teste, 99, 101, 102, 103, 104,  
    105, 106, 107  
  visão geral, 99, 100, 101  
ferramentas, 94  
JMeter tool  
  planos de teste, 106  
Mercury Load Runner, 99  
Rational Test Suite, 99  
servidores baseados em RMI, 95, 96, 97, 98
- testes**  
classes  
  criando, 137
- texto**  
hacking  
  aplicação Chat, 110, 111
- This, painel (Omniscient Debugger), 75**
- Thread Group, (planos de teste JMeter), 100**
- Thread Group, comando (menu Add), 101**
- thread stall, 89**
- threads**  
alocação, 86, 87, 88, 89  
concorrências de dados, 86, 87  
execução, 88, 89
- impasse, 87, 88, 89  
sincronização, 86, 87, 88, 89
- Threads, painel (Omniscient Debugger), 75**
- Timers, nó (planos de teste JMeter), 100**
- Tomcat, servidores Web**  
aplicação WebCream, 101  
Carregamento, 101
- Tools, comandos de menu**  
ZKM Script Helper, 33
- Total Commander, 4, 5**
- Total Commander Web site, 216**
- trampoline, 179**
- Transmission Control Protocol/Internet Protocol (TCP/IP), 119**
- TTY Output, painel (Omniscient Debugger), 75**
- túneis**  
HTTP (Hypertext Transfer Protocol)  
  eavesdropping, 115, 116
- U**
- UI (User Interface)**  
elementos/recursos  
  hacking, 109  
  hacking, arquivos de configuração, 113  
  hacking, imagens, 111, 112, 113  
  hacking, texto, 110, 111
- Unix**  
bibliotecas compartilhadas (.so), 125  
patches de código nativo, 181, 182
- uso de heap**  
profiling, 80, 81, 82
- V**
- variáveis**  
ambiente  
  acessando, 70  
  rastreando, 59
- vazamentos de memória**  
solucionando problemas, 82, 83, 84, 85, 86
- verificando**  
arquivos de licença, 206, 207, 208  
bytecode, 157
- Verisign, site Web, 187**
- visualizando**  
arquivos de classe  
  jClassLib Bytecode Viewer, 150, 151

**W****Web site**

FAR, 216

**WebCream**

aplicações

convertendo em HTML, 39

class loaders personalizados, 129

contrato de licença, 211, 212, 213, 214,  
215

licenciamento, 202

**WebCream, aplicação**

testando, 101, 103, 104, 105, 106, 107

testes, 102

**WebCream, ofuscador, 30****WebCream, site Web, 216****Windows**

bibliotecas de vínculo dinâmico (.dll),

125

gerenciamento de arquivo

FAR (File and Archive Manager), 4, 5

Total Commander, 4, 5

patches de código nativo, 174

biblioteca Microsoft Detours, 179, 180,  
181

formato Portable Executable (PE), 174,  
175, 176, 177

utilitário Function Replacer, 177, 178,  
179

**Windows Explorer**

gerenciamento de arquivo, 4

**Z****Zelik KlassMaster, ofuscador, 29**

aplicação Chat, 32, 33, 34, 35

padrões de nomes, 35

personalizando, 32

**Zelix ClassMaster, ofuscador, 19, 20****Zelix KlassMaster, site Web, 217****ZKM Script Helper, comando (menu  
Tools), 33**

**PÁGINA EM BRANCO**