CATÁLOGO DE PRÁTICAS DE PROGRAMAÇÃO SEGURA EM JAVA

Adriana Jacoto Unger
Divisão de Ciência da Computação
12228-901 São José dos Campos – SP - Brazil
ajacoto@terra.com.br

Clovis Torres Fernandes
Divisão de Ciência da Computação
12228-901 São José dos Campos – SP - Brazil
clovis@ita.br

RESUMO

A adoção de práticas de programação segura é um fator fundamental para o desenvolvimento de softwares que atendam requisitos de segurança. Práticas gerais de programação segura têm sido divulgadas, porém sua aplicação específica à linguagem Java nem sempre é pertinente. Por outro lado, as informações disponíveis sobre segurança na linguagem Java são dispersas e não sugerem a aplicação prática de técnicas de programação segura. Este artigo tem como objetivo compilar e classificar um conjunto de práticas de programação em Java na forma de um Catálogo, que permita a sua aplicação prática imediata tanto num contexto educacional e de treinamento, quanto num contexto profissional,

ABSTRACT

Adopting secure programming practices is a fundamental factor to develop software that attends security requirements. General secure programming practices have been published, but are not always applicable to Java language. On the other hand, available information about security of the Java language is dispersed and doesn't lead to practical application of secure programming techniques. The objective of this paper is to compile and categorize a set of Java programming practices as a catalogue which may allow immediate practical application both in educational and training settings and in professional settings.

1 INTRODUÇÃO

Programas podem ser corretos e confiáveis e ainda assim terem vulnerabilidades que os tornam suscetíveis de acidentes e ataques. Vulnerabilidade é um defeito ou imperfeição existente em estado latente em um sistema que, associado a certas condições ambientais e de contexto, pode levar a acidentes ou permitir ataques. Em Kirwan [19], relata-se que a maioria das vulnerabilidades conhecidas e exploradas em ataque hoje em dia são fruto de codificação e testes pobres e aplicação desleixada de princípios e técnicas de engenharia de software.

Atualmente existem normas e metodologias de desenvolvimento de software que permitem produzir programas visando reduzir a ocorrência de vulnerabilidades [1, 3, 5, 6, 9]. Algumas são práticas gerais, independentes de linguagem, enquanto outras são específicas a uma dada linguagem [4, 8, 9, 11, 16].

A linguagem Java possui mecanismos de segurança que possuem muitas vantagens em relação a outras linguagens atuais ou desenvolvidas no passado. Porém, possui também recursos que, usados de forma incorreta ou desatenta, podem levar ao desenvolvimento de um software inseguro ou cheio de vulnerabilidades. As informações disponíveis atualmente a respeito da produção de software seguro com o uso da linguagem Java são muitas [4, 7, 8, 9, 10, 11, 12, 13, 14, 16].

Contudo, nem sempre as práticas gerais aplicáveis à linguagem de programação Java e as práticas específicas relativas à linguagem Java são facilmente acessíveis ou são de conhecimento dos interessados. Nem sempre também objetivam a aplicação prática na forma de técnicas de programação segura. Ou seja, há uma dificuldade do interessado em encontrar numa fonte só as boas práticas de programação segura, em especial as aplicáveis e referentes à linguagem de programação Java.

Este trabalho é o resultado de um estudo da literatura com o objetivo de condensar, organizar e compilar um Catálogo de práticas de programação segura em Java, cuja aplicação direta tem o potencial de reduzir a introdução de vulnerabilidades e aumentar a segurança do software desenvolvido.

Um fator fundamental é a adoção de práticas de programação segura durante todas as fases do ciclo de desenvolvimento de software, com o objetivo de evitar que o software desenvolvido apresente vulnerabilidades, em especial as de conhecimento geral [18]. Daí se adotar o uso neste trabalho de "programação segura" em vez de "codificação segura", que se aplica apenas à fase de codificação.

A classificação das técnicas de programação segura compiladas nesse Catálogo permite que o mesmo seja utilizado como material didático para a capacitação de profissionais, com o objetivo de contribuir com a produção de programas mais seguros.

O artigo está organizado da seguinte forma. A Seção 2 apresenta um conjunto de práticas gerais de programação segura, que são também aplicáveis à linguagem Java. A Seção 3 apresenta alguns aspectos de segurança e insegurança da linguagem Java. A Seção 4 apresenta a estrutura do Catálogo e exemplifica o uso de uma prática. A Seção 5 apresenta a conclusão do trabalho e o Apêndice apresenta o Catálogo de acordo com a estrutura delineada na Seção 4.

2 PRÁTICAS GERAIS DE PROGRAMAÇÃO SEGURA APLICÁVEIS A JAVA

Existem muitas práticas gerais de programação segura, porém serão listadas a seguir somente aquelas aplicáveis à linguagem Java.

Controle de Race Conditions

As *race conditions* ocorrem quando há a concorrência de mais de um processo por um determinado recurso. Esse comportamento pode causar uma falha de segurança se um programa não estiver preparado para lidar com interferências causadas por outros processos [4].

Minimização de Privilégios

Minimizar privilégios é uma das técnicas mais efetivas de segurança, pois, além de reduzir o número de interações com dados ou programas sensíveis, contribui para limitar o alcance em caso de ocorrência de problemas [4]. Nessa técnica, deve-se fornecer a um programa ou usuário o mínimo de privilégio possível, bem como se devem criar diferentes permissões para diferentes funções ou perfis.

Utilização de Várias Camadas de Segurança

Devem-se utilizar várias camadas de segurança e nunca confiar em apenas um mecanismo de segurança, pois, por melhor que ele seja, pode ser passível de falha e, nesse caso, mecanismos adicionais serão necessários [1].

Validação das Variáveis de Entrada

Muitas falhas de segurança surgem porque um atacante envia um argumento inesperado ou de formato inesperado para um programa ou uma função de um programa. Se o método ou programa não estiver preparado para lidar com tais valores de entrada, isso pode acarretar perda de controle do sistema, falhas gerais de proteção e outros tipos de falhas catastróficas. Por isso, é importante verificar todos os argumentos, com atenção extra aos argumentos passados para o programa na linha de comando [2].

Uma boa prática é a programação de métodos intrinsecamente seguros, ou seja, que tratam todas as variáveis de entrada com não confiáveis, verificando sua validade antes de usá-las. Isso significa que cada método deve verificar se a variável de entrada está dentro de valores aceitáveis e não contém caracteres estranhos ou possui má-formação antes de qualquer uso. Sempre que o sistema receber dados, seja através da entrada direta de dados pelo usuário, seja pelo recebimento de dados de outro sistema, inclusive do próprio sistema operacional hospedeiro, estes devem passar por uma verificação de integridade e consistência.

Tratamento de exceções

Uma boa prática de programação consiste em sempre verificar os códigos de erro ou exceções retornados por uma função, principalmente nas chamadas a funções do sistema operacional. Mesmo códigos com pouca chance de erro podem ser explorados por atacantes para quebrar a segurança do sistema, se não for implementado um tratamento da exceção adequado.

É importante implementar o tratamento de cada exceção adequadamente, pois quase todas as funções possuem modos de falha. O não tratamento de uma exceção pode ser explorado por um usuário mal-intencionado e utilizado para causar um erro catastrófico e interromper o funcionamento de um sistema. Também se deve tomar o cuidado de implementar o lançamento de exceções sempre que algum método estiver sendo implementado e algum problema for detectado na entrada ou durante a execução da função.

Utilização de Verificadores de Código

Os verificadores de código são programas que vasculham o código fonte visando descobrir vulnerabilidades que podem estar presentes, devido à não adoção de boas práticas de programação segura ou não [1].

3 ASPECTOS DE SEGURANÇA E INSEGURANÇA DA LINGUAGEM JAVA

Nesta seção discutem-se alguns aspectos de segurança e insegurança da linguagem Java. Ela possui tanto construções que podem facilitar a produção consciente ou inconsciente de vulnerabilidades nos programas codificados com ela, quanto construções que facilitam a produção de programas seguros.

O mecanismo de controle de acesso da plataforma Java protege os recursos do sistema de acessos não autorizados através da verificação de que o código de chamada possua as permissões apropriadas para acessar um determinado recurso [8]. A plataforma Java permite a definição de permissões de acesso no nível de uma operação específica, como, por exemplo, a permissão para abrir um determinado arquivo. Esse recurso vai de encontro ao princípio de segurança de minimização de privilégios [4].

A linguagem Java permite o uso de palavras-chave de acesso que definem quais objetos de classes podem acessar os atributos e métodos de cada classe. O modificador *public* permite que atributos e métodos de objetos da classe sejam acessados externamente a eles. O modificador *private* permite que atributos e métodos de um dado

objeto da classe sejam acessados somente internamente ao objeto da classe. E, por fim, o modificador *protected* permite que atributos e métodos de um dado objeto da classe sejam acessados somente por objetos de subclasses da classe ou por objetos de classes que compartilham o mesmo package da classe.

A linguagem Java possui tipagem estática forte, o que significa que não é possível utilizar ponteiros arbitrários, aumentando a segurança da aplicação. Essa característica também permite a verificação dos tipos das variáveis de entrada ainda em tempo de compilação. Porém a validade do valor das variáveis, no que diz respeito ao domínio da aplicação do método, deve ser verificada no escopo do código de implementação. Além disso, o uso de herança pode acrescentar variáveis de entrada herdadas de superclasses que devem ser verificadas, de acordo com o princípio de segurança de validação de todas as entradas [4].

O recurso de herança deve sempre ser tratado com o máximo de cuidado, pois ao herdar métodos inseguros de superclasses ou interfaces fatalmente uma vulnerabilidade será introduzida no código [4].

A linguagem Java não possui ponteiros explícitos, apenas implícitos. Por causa disso, não permite o acesso do programador aos valores de endereço de memória que os ponteiros possuem. Este fato, aliado ao gerenciamento dinâmico da memória *heap* que a linguagem possui, evita que ocorram erros como o estouro de buffer [4]. As características da linguagem permitem que o estouro de *buffer* possa, às vezes, ser detectado ainda em tempo de compilação e sempre evitado em tempo de execução.

A linguagem Java permite, através da definição de exceções, a especificação de fluxos alternativos de execução para casos em que estados inválidos ocorram durante a chamada de um método. Desse modo, um método que realize a divisão de dois argumentos inteiros, por exemplo, pode lançar uma exceção no caso do denominador ser igual a zero. Essa exceção lançada pode, por sua vez, ser capturada pelo código cliente do método, evitando que o programa seja interrompido, ou que um valor inválido seja retornado pelo método.

Além disso, os seguintes mecanismos da linguagem Java são considerados robustos e o uso deles pode constituir-se em boa prática de programação segura:

- Linhas de execução e métodos sincronizados – Podem ser utilizados de forma combinada para evitar *race conditions* [10; 12]
- Assinatura de código e políticas – Podem ser usados de forma combinada para permitir assinatura de código e conseqüentemente permitir a execução de código somente se eles forem assinados por pessoas selecionadas
- *Java Sandbox* – É um mecanismo de segurança de Java que contém o seguinte: verificador de *bytecode*, que verifica se o código objeto compilado de uma classe é válido; carregador de classe, que é responsável por carregar as classes e evita que classes não confiáveis sejam carregadas; gerente de segurança, que define a política de segurança que deve ser usada [11]. Cabe notar que este mecanismo tem sido explorado para ataques, de modo que não se deve confiar totalmente nele se a aplicação for crítica.
- JCA (*Java Cryptography Architecture*) e JCE (*Java Cryptography Extension*) – São APIs que contêm algoritmos criptográficos.
- *CodeSource* e *ProtectionDomain* – Permitem a definição de uma política de segurança com alta granularidade.
- GuardedObject – Permite a proteção individual de uma instância do objeto.
- JSSE (*Java Secure Socket Extension*) – Implementa o SSL.
- JAAS (*Java Authentication and Authorization Service*) que provê métodos de autenticação

4 CATÁLOGO DE PRÁTICAS DE PROGRAMAÇÃO SEGURA EM JAVA

Os itens compilados no Catálogo de práticas de programação segura em Java encontram-se no Apêndice. São 21 práticas identificadas até agora. Elas foram categorizadas de acordo com os seguintes critérios:

- Tipo da Prática:
  - FAÇA - Indica as práticas que informam o que se deve seguir sempre.
  - EVITE - Indica as práticas que informam o que se deve deixar de seguir, sempre que possível.
  - NÃO FAÇA - Indica as práticas que informam o que não se deve seguir em hipótese alguma.
- Fase de desenvolvimento:
  - (M) Fase de Modelagem - Define se a prática pode ser aplicada já na fase de modelagem abstrata, a saber, durante análise ou projeto.
  - (C) Fase de Codificação – Define se a prática somente pode ser aplicada durante a fase de codificação da aplicação.
- Aplicação:
  - Direta - Define se a aplicação da prática é direta, através da substituição de recursos da linguagem, por exemplo.
  - Indireta - Requer uma reestruturação da implementação.
- Palavra-chave – Identifica-se, se aplicável, a palavra-chave da linguagem Java à qual a prática está associada. *N.A* indica a não

ocorrência de palavra-chave da linguagem Java que seja aplicável.

A fim de ilustrar uma aplicação de prática de programação segura do Catálogo, a prática de número 20 detalhada no Apêndice – **Nunca compare nomes de classes** – é mostrada a seguir, na forma de um exemplo prático [9, 4].

Considerando um programa que deve executar uma determinada operação apenas se um objeto for instância de uma dada classe, por exemplo classe Admin, uma implementação inadequada possível seria a seguinte:

```
 if (obj.getClass().getName().equals("Admin"))
{
        // executa aqui a operação privilegiada
}
```

Porém, sabe-se que esse código constitui uma vulnerabilidade que pode ser explorada a partir da codificação de uma nova classe Admin, por um programador malicioso, cujo objeto poderá ser enviado para o método acima. Para evitar essa vulnerabilidade, basta refatorar o código da seguinte maneira, supondo que o objeto "a" corresponda a uma instância da classe Admin:

```
  if (obj.getClass() == a.getClass()) {
        // executa aqui a operação privilegiada
}
```

Desse modo, no código, em vez de se comparar se as classes possuem o mesmo nome, verifica-se se os dois objetos pertencem à mesma classe.

Contudo, caso seja imprescindível saber se um determinado objeto é realmente instância de uma dada classe, deve-se assegurar do espaço de nomes da classe corrente. Dessa forma, o seguinte código deve ser usado:

```
if (obj.getClass() ==
   this.getClassLoader().loadClass("Admin")) {
        // executa aqui a operação privilegiada
}
```

5 CONCLUSÃO

O trabalho apresenta um conjunto de 21 práticas de programação segura em Java, destacando, para cada uma delas, as vulnerabilidades de segurança que podem ser eliminadas através do seu uso. A aplicação do conjunto dessas práticas durante o desenvolvimento do software pode levar à geração de uma aplicação mais segura. O Catálogo visa a auxiliar na formação dos programadores Java, quanto aos aspectos de segurança que devem ser considerados durante a implementação.

Este artigo descreve apenas um resultado inicial da pesquisa bibliográfica, e não é exaustivo no que diz respeito às práticas de programação. Além disso, destaca-se que algumas vulnerabilidades têm como causa a própria arquitetura e composição lingüística da linguagem Java, que vem evoluindo ao longo dos últimos anos e continua em desenvolvimento [11].

Outros problemas de segurança não podem ser resolvidos através da adoção de técnicas de programação, devendo ser tratados no nível de análise e projeto. Esses requisitos de segurança podem ser atendidos através do uso de padrões de segurança, cuja aplicação pode ser explorada em trabalhos futuros.

Como trabalho futuro, pretende-se também desenvolver cartilha hipermídia de treinamento e conscientização do programador Java. Nessa aplicação, cada uma das práticas seria justificada e exemplificada com código Java real, onde se mostrariam como as vulnerabilidades poderiam ser exploradas e como o código seguro que segue o que aqui foi prescrito poderia dificultar a ocorrência de acidentes ou ataques.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ALBUQUERQUE, Ricardo; RIBEIRO, Bruno. Segurança no desenvolvimento de software. Rio de Janeiro: Campus, 2002.

[2] PEREIRA, Felipe. Programas seguros: vulnerabilidades comuns e cuidados no desenvolvimento. In: Simpósio de Segurança em Informática, 2000.

[3] WHEELER, David A. Secure Programming for Linux and Unix HOWTO. 2003. Disponível em http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf. Acesso em 01/06/2004.

[4] WHEELER, David A. Java Security. 2000. Disponível em http://www.dwheeler.com/javasec/javasec.pdf. Acesso em 14/07/2004.

[5] PETEANU, Razvan. Best Practices for Secure Development. 2001. Disponível em http://members.rogers.com/razvan.peteanu. Acesso em 27/09/2004.

[6] National Cyber Security. Processes to Produce Secure Software. 2004. Disponível em http://www.cigital.com/papers/download/secure_software_process.pdf. Acesso em 29/09/2004.

[7] SUN MICROSYSTEMS INC. Security Code Guidelines. 2000. Disponível em http://java.sun.com/security/seccodeguide.htm. Acesso em 02/02/2005.

[8] McGRAW Gary; FELTEN Edward W. Securing Java. 1999. Disponível em http://www.securingjava.com/toc.html. Acesso em 03/12/2004.

[9] McGRAW Gary, FELTEN Edward W. Twelve Rules for developing more secure Java code. Javaworld, 1998. Disponível em http://www.javaworld.com/javaworld/jw-12-

1998/jw-12-securityrules.html. Acesso em 03/12/2004.
[10] HANSEN, Brinch. 1999. Java's Insecure Parallelism. Disponível em http://www.brinch-hansen.net/papers/1999b.pdf. Acesso em 22/06/2005.
[11] GONG, Li. 1997. Java Security: Present and Near Future. In: IEEE Micro, 1997. Disponível em http://www.cs.ucla.edu/~miodrag/cs259-security/gong97java.pdf. Acesso em 22/06/2005.
[12] BALFANZ, Dirk, GONG, Li. Experience with Secure Multi-Processing in Java. 1997. Disponível em http://www.windowsecurity.com/uplarticle/14/icdcs.pdf. Acesso em 22/06/2005.
[13]BAUER, Lujo, APPEL, Andrew W., FELTEN, Edward W. Mechanisms for secure modular programming in Java. In: Software Practice and Experience, 2003. Disponível em http://www.ece.cmu.edu/~lbauer/papers/jms-spe03.pdf. Acesso em 22/06/2005.
[14] YELLIN, Frank. Low Level Security in Java. 1995. Disponível em http://www.net.uom.gr/Books/Manuals/llsij.pdf. Acesso em 22/06/2005.
[15] STERBENZ, Andréas. An Evaluation of the Java Security Model. 1996.
[16] ZAIDMAN, Marsha. Teaching Defensive Programming in Java. In: The Journal of Computing in Small Colleges, 2004.
[17] HARTEL, Pieter. Formalizing the Safety of Java, the Java Virtual Machine, and Java Card. In: ACM Computing Surveys, 2001. Disponível em http://www.gamboas.org/ruben/classes/2002-fall/cosc5000/java-safety.pdf. Acesso em 22/06/2005.
[18]NOOPUR, Davis. Developing Secure Software. Disponível em http://www.softwaretechnews.com/stn8-2/noopur.html. Acesso em 10/08/2005.
[19] KIRWAN, Mary. The Quest for Secure Code. Disponível em http://www.globetechnology.com/servlet/story/RTGAM.20041001.gtkirwanoct1/BNStory/Technology/. Acesso em 18/08/2005.

APÊNDICE – Catálogo de Práticas de Programação Segura em Java

Tipo da Prática: FAÇA

Fase	Aplicação	#	Prática	Palavra-chave
M	Direta	1	Sempre classifique os atributos de uma classe como *private* [4].	*private public protected*
		2	Sempre defina métodos de acesso (métodos *accessors* e *modifiers*) para obter ou modificar os valores das variáveis *private* [4].	*N.A.*
		3	Em princípio, sempre defina os métodos de acesso (métodos *accessors* e *modifiers*) como *protected* e somente acesse ou modifique os valores dos atributos de objetos da classe ou subclasse através desses métodos. Evite definir os métodos de acesso a variáveis como *public*, a menos que faça sentido que eles sejam definidos como tal,	*private public protected*
		4	Declare sempre os métodos como *private*, a não ser em caso de justificada necessidade, quando serão declarados como *public*. Apenas métodos de acesso são declarados como *protected*. Métodos *public* devem conter em seu código proteção contra variáveis de entrada maliciosas [9, 4].	*private public protected*
	Indireta	5	Defina políticas para detalhar os privilégios de acesso a objetos de classes ou a *applets* específicos com base em arquivos fonte e/ou assinaturas.	N.A.
		6	Tome cuidado ao utilizar herança, pois ao herdar métodos de superclasses, interfaces ou superclasses de interfaces, há riscos de se incluir vulnerabilidades no código [16]	*extend*
C	Direta	7	Declare todas as classes como *final*, a não ser em caso de justificada necessidade. Isso evita que as classes sejam usadas de forma maliciosa através de uma subclasse que modifique o seu comportamento [4].	*final*
		8	Se precisar assinar o código, coloque tudo num único *archive*. Isso evita que um atacante use classes assinadas junto com classes maliciosas [4].	N.A.

		9	Defina todas as classes como *uncloneable*, pois isso evita que elas sejam instanciadas sem a execução do construtor [4].	*uncloneable*
		10	Defina todas as classes como *unserializeable*, pois isso evita que o conteúdo dos objetos, incluindo atributos *private*, possa ser acessado através de métodos de serialização [4].	*unserializeable*
		11	Defina todas as classes como *undeserializable*, pois isso evita que o conteúdo de um objeto possa ser obtido através de métodos de desserialização [4].	*undeserializable*
	Indireta	12	Configure o ambiente de desenvolvimento em Java levando em consideração aspectos de segurança.	N.A.
		13	Use classes de bibliotecas seguras. Não confie em classes reusáveis de fontes externas.	N.A.

Tipo da Prática: EVITE

Fase	Aplicação	#	Prática	Palavra-chave
M	Direta	14	Evite o uso de *privileged blocks*, pois esse recurso permite que um código tenha privilégios de realizar chamadas a serviços mesmo que o código de chamada não possua as permissões para acesso a esses serviços [16]	*privileged block*
C	Indireta	15	Evite ao máximo o uso de atributos *static*. Esse tipo de atributo é alocado à classe e não ao objeto e desse modo pode ser localizado por qualquer outra classe [4].	*static*

Tipo da Prática: NÃO FAÇA

Fase	Aplicação	#	Prática	Palavra-chave
M	Direta	16	Não use o mecanismo de *package* para controle de acesso, pois a grande maioria dos pacotes é aberta. Ou seja, é possível criar uma classe maliciosa e adicioná-la ao mesmo *package*, obtendo, desse modo, acesso privilegiado [4].	*package*
		17	Não use o tipo String para armazenar senhas, mesmo temporariamente, pois esse tipo de variável é *immutable*, o que significa que seu valor será mantido na memória até a próxima operação de *garbage collection*. Use o tipo char[] e sobrescreva o valor da variável assim que possível, a fim de evitar ataques à memória para a obtenção de senhas. [4]	*String*
	Indireta	18	Nunca armazene senhas ou outras informações secretas diretamente no código. Usando JVMs maliciosos é possível acessar esses dados mesmo sem acesso ao código fonte [4].	N.A.
C	Direta	19	Não utilize objetos *mutable* como entrada ou retorno de métodos, pois esse tipo de objeto pode ser modificado [4]. Esses objetos podem, por exemplo, passar pela validação de dados de entrada, sendo depois modificados intencionalmente para um valor inválido. Objetos do tipo *array* são *mutable*, de modo que se deve ter o máximo cuidado ao usá-los!	*mutable*
		20	Nunca compare nomes de classes. Nada impede que uma classe com o mesmo nome possa ser codificada e usada de forma maliciosa [9, 4].	*getClass getName*
	Indireta	21	Não use classes aninhadas. Sem aviso, essas classes quando compiladas se tornam acessíveis a qualquer outra classe do *package*. Além disso, os atributos *private* da classe que contém a classe aninhada se tornam *public* automaticamente, para permitir o acesso dos mesmos pela classe aninhada [4].	N.A.