ヒント: ファイナライザーによる脆弱性からコードを保護する

無効なクラスが作成されないようにするためのパターン

Neil D. Masson
Java Support Engineer
IBM

2011年 8月 05日

皆さんの Java コードはファイナライズ処理を悪用した攻撃に対して脆弱な可能性があります。ファイナライズ処理を悪用した攻撃の仕組み、またそうした攻撃を受けないようにコードを変更する方法について学びましょう。

オブジェクトを作成する際にファイナライザーが実装されると、Java コードに脆弱性が生じる可能性があります。この脆弱性を悪用した攻撃は、ファイナライザーを使ってオブジェクトを復活させるという、よく知られた攻撃手法のバリエーションです。`finalize()` メソッドを持つオブジェクトが到達不能になると、そのオブジェクトは後で処理されるためのキューに置かれます。今回のヒントでは、ファイナライズ処理を悪用した攻撃の仕組み、またそうした攻撃を受けないようにコードを保護する方法について説明します。この記事で紹介するコード・サンプルはすべてダウンロードすることができます。

ファイナライザーとは、オペレーティング・システムに返す必要のある任意のネイティブ・リソースを Java メソッドによって解放できるようにするためのものです。残念なことに、ファイナライザーでは任意の Java コードを実行することができるため、リスト 1 のようなコードもあり得ます。

リスト 1. 復活可能なクラス

```
public class Zombie {
  static Zombie zombie;

  public void finalize() {
    zombie = this;
  }
}
```

上記コードでは、`Zombie` のファイナライザーが呼び出されると、そのファイナライザーはファイナライズ処理の対象となるオブジェクト (`this` によって参照されています) を取得し、そのオブジェクトを静的な `zombie` 変数に格納します。すると、そのオブジェクトは再度到達可能となり、ガーベッジ・コレクションの対象ではなくなります。

このコードを不正な形に変更すると、完全な形で作成されていないオブジェクトを復活させることさえできてしまいます。リスト 2 のように、イニシャライザーによる基準では不適切と判断されるオブジェクトであっても、ファイナライザーによって作成することができてしまいます。

リスト 2. 不正なクラスを作成する

```
public class Zombie2 {
  static Zombie2 zombie;
  int value;

  public Zombie2(int value) {
    if(value < 0) {
      throw new IllegalArgumentException("Negative Zombie2 value");
    }
    this.value = value;
  }
  public void finalize() {
    zombie = this;
  }
}
```

リスト 2 では `value` 引数に対してチェックを行っていますが、`finalize()` メソッドがあることにより、そのチェックをしないのと同じことになっています。

攻撃はどのように行われるか

もちろん、リスト 2 のようなコードを作成する人はいないでしょうが、このクラスをリスト 3 のようにサブクラス化すると、脆弱なクラスになります。

リスト 3. 脆弱なクラス

```
class Vulnerable {
  Integer value = 0;

  Vulnerable(int value) {
    if(value <= 0) {
      throw new IllegalArgumentException("Vulnerable value must be positive");
    }
    this.value = value;
  }
  @Override
  public String toString() {
    return(value.toString());
  }
}
```

リスト 3 の `Vulnerable` クラスは、`value` の値が正ではない値に設定されることがないように設計されています。そのように設計したつもりであっても、リスト 4 の `AttackVulnerable()` メソッドによって、その設計が台無しにされてしまいます。

リスト 4. `Vulnerable` クラスの設計を台無しにするクラス

```
class AttackVulnerable extends Vulnerable {
  static Vulnerable vulnerable;

  public AttackVulnerable(int value) {
    super(value);
  }

  public void finalize() {
```

```
    vulnerable = this;
  }

  public static void main(String[] args) {
    try {
      new AttackVulnerable(-1);
    } catch(Exception e) {
      System.out.println(e);
    }
    System.gc();
    System.runFinalization();
    if(vulnerable != null) {
      System.out.println("Vulnerable object " + vulnerable + " created!");
    }
  }
}
```

`AttackVulnerable` クラスの `main()` メソッドでは、新しい `AttackVulnerable` オブジェクトをインスタンス化しようとしています。`value` の値は正ではない範囲外の値なので、例外がスローされ、`catch` ブロックでキャッチされます。`System.gc()` と `System.runFinalization()` を呼び出すことで、ガーベッジ・コレクション・サイクルとすべてのファイナライザーが実行されるように、VM に推奨しています。これらの呼び出しがなくても攻撃を成功させることはできますが、これらを呼び出すことで、攻撃による最終的な結果、つまり無効な値を持つ `Vulnerable` オブジェクトが作成されるという結果を示すことができます。

このテスト・ケースを実行すると、以下のような出力が結果として得られます。

```
java.lang.IllegalArgumentException: Vulnerable value must be positive

Vulnerable object 0 created!
```

なぜ、上記の結果では `Vulnerable` の値が -1 ではなく 0 になっているのでしょう？ `Vulnerable` のコンストラクター (リスト 3) では、引数のチェックが終わるまで `value` への代入が行われないことに注意してください。そのため、`value` の初期値である 0 が出力されているのです。

この種の攻撃は、明示的なセキュリティー・チェックをバイパスする目的にも使用されます。例えば、リスト 5 の `Insecure` クラスは、`SecurityManager` の下で実行された場合で呼び出し側がカレント・ディレクトリーへの書き込み権限を持っていない場合、`SecurityException` をスローするように設計されています。

リスト 5. `Insecure` クラス

```
import java.io.FilePermission;

public class Insecure {
  Integer value = 0;

  public Insecure(int value) {
    SecurityManager sm = System.getSecurityManager();
    if(sm != null) {
      FilePermission fp = new FilePermission("index", "write");
      sm.checkPermission(fp);
    }
    this.value = value;
  }
  @Override
  public String toString() {
    return(value.toString());
  }
}
```

先ほどと同じ方法でリスト 5 の `Insecure` クラスを攻撃することができます。それを示したものがリスト 6 の `AttackInsecure` クラスです。

リスト 6. `Insecure` クラスを攻撃する

```
public class AttackInsecure extends Insecure {
  static Insecure insecure;

  public AttackInsecure(int value) {
    super(value);
  }

  public void finalize() {
    insecure = this;
  }

  public static void main(String[] args) {
    try {
      new AttackInsecure(-1);
    } catch(Exception e) {
      System.out.println(e);
    }
    System.gc();
    System.runFinalization();
    if(insecure != null) {
      System.out.println("Insecure object " + insecure + " created!");
    }
  }
}
```

リスト 6 のコードを `SecurityManager` の下で実行すると、以下の出力が得られます。

```
java -Djava.security.manager AttackInsecure
java.security.AccessControlException: Access denied (java.io.FilePermission index write)
Insecure object 0 created!
```

攻撃を防ぐ方法

Java SE 6 で JLS (Java Language Specification: Java 言語仕様) の第 3 版が実装されるまで、上記の攻撃を防ぐための方法としては、`initialized` フラグを使用する方法、サブクラス化を禁止する方

法、`final` ファイナライザーを作成する方法しかなく、どれも満足できる方法ではありませんでした。

`initialized` フラグを使用する方法

攻撃を防ぐための 1 つの方法は `initialized` フラグを使用する方法です。この方法では、オブジェクトが適切に作成された後で `initialized` フラグを `true` に設定します。そのクラスのすべてのメソッドは、`initialized` がセットされているかどうかを最初にチェックし、セットされていない場合には例外をスローします。この種のコードは作成するのが面倒であるだけでなく、忘れられがちであり、しかも攻撃者がメソッドをサブクラス化するのを防ぐことはできません。

サブクラス化を防ぐ方法

クラスを作成する際に `final` と宣言して作成することができます。final と宣言すると、誰もそのクラスのサブクラスを作成することができなくなるため、攻撃を防ぐことができます。ただしこの方法では、そのクラスを継承して特殊化したり機能を追加したりする柔軟性がなくなります。

`final` ファイナライザーを作成する方法

クラスを作成する際に、そのクラスのファイナライザーを作成し、そのファイナライザーを `final` と宣言することができます。これはつまり、そのクラスのサブクラスがファイナライザーを宣言することはできない、ということです。この方法の欠点は、ファイナライザーが存在することにより、ファイナライザーが存在しない場合よりも長期間、そのオブジェクトが存続してしまうことです。

新しくて、より良い方法

コードや制約を追加しなくてもこの種の攻撃を容易に防げるように、Java 設計者達は JLS を変更し (「参考文献」を参照)、`java.lang.Object` が作成される前にコンストラクターの中で例外がスローされた場合にはそのメソッドの `finalize()` メソッドは実行されない、という仕様にしました。

しかし、`java.lang.Object` が作成される前に例外をスローするためにはどうすればよいのでしょう。結局のところ、どのようなコンストラクターであれ、そのコンストラクターの最初の行で `this()` または `super()` を呼び出す必要があります。コンストラクター内にそうした明示的な呼び出しがない場合には、`super()` への呼び出しが暗黙的に追加されます。そのため、オブジェクトを作成する前に、同じクラスの別のオブジェクト、または同じクラスのスーパークラスの別のオブジェクトを作成する必要があります。そのため、最終的に `java.lang.Object` 自体を作成する必要があり、その後ですべてのサブクラスを作成し、作成されたメソッドのコードを実行します。

`java.lang.Object` が作成される前に例外がどのようにスローされるのかを理解するためには、オブジェクトを作成する際の正確なシーケンスを理解している必要があります。JLS には、このシーケンスが明確に記述されています。

オブジェクトを作成する場合、JVM は以下の処理を行います。

1. そのオブジェクトのためのスペースを割り当てます。

2. そのオブジェクトのすべてのインスタンス変数をデフォルト値に設定します。これらのインスタンス変数には、そのオブジェクトのスーパークラスのインスタンス変数も含まれます。
3. そのオブジェクトにパラメーター変数を割り当てます。
4. コンストラクターに対する明示的または暗黙的な呼び出し (コンストラクター内の `this()` または `super()` への呼び出し) をすべて処理します。
5. そのクラスの変数を初期化します。
6. コンストラクターの残り部分を実行します。

重要なポイントは、コンストラクター内のどのコードが処理されるよりも前に、コンストラクターのパラメーターが処理されるという点です。これはつまり、パラメーターが処理される間に検証を行うと、(例外をスローすることで) クラスがファイナライズされるのを防ぐことができるということです。

このことを利用すると、リスト 3 の `Vulnerable` クラスをリスト 7 のような新しいバージョンに変更することができます。

リスト 7. `Invulnerable` クラス

```
class Invulnerable {
  int value = 0;

  Invulnerable(int value) {
    this(checkValues(value));
    this.value = value;
  }

  private Invulnerable(Void checkValues) {}

  static Void checkValues(int value) {
    if(value <= 0) {
      throw new IllegalArgumentException("Invulnerable value must be positive");
    }
    return null;
  }

  @Override
  public String toString() {
    return(Integer.toString(value));
  }
}
```

リスト 7 で、`Invulnerable` の public コンストラクターは private コンストラクターを呼び出し、この private コンストラクターが `checkValues` メソッドを呼び出して Invulnerable のパラメーターを作成しています。checkValues メソッドが呼び出されるのは、private コンストラクターがスーパークラス (`Object` のコンストラクター) を作成するための呼び出しを行う前です。そのため、`checkValues` 内で例外がスローされると、`Invulnerable` オブジェクトはファイナライズされません。

リスト 8 のコードは `Invulnerable` を攻撃しようとしています。

リスト 8. `Invulnerable` クラスを攻撃しようとする試み

```
class AttackInvulnerable extends Invulnerable {
  static Invulnerable vulnerable;
```

```
  public AttackInvulnerable(int value) {
    super(value);
  }

  public void finalize() {
    vulnerable = this;
  }

  public static void main(String[] args) {
    try {
      new AttackInvulnerable(-1);
    } catch(Exception e) {
      System.out.println(e);
    }
    System.gc();
    System.runFinalization();
    if(vulnerable != null) {
      System.out.println("Invulnerable object " + vulnerable + "
created!");
    } else {
      System.out.println("Attack failed");
    }
  }
}

with the addition of

    } else {
      System.out.println("Attack failed");
```

古いバージョンの JLS に従って作成された Java 5 では、`Invulnerable` オブジェクトを作成することができます。

```
java.lang.IllegalArgumentException: Invulnerable value must be positive
Invulnerable object 0 created!
```

Java SE 6 (Oracle の JVM と IBM の JVM SR9 の GA (General-Availability: 一般ユーザー向け) リリース以降) では、最新の仕様に従っているため、Invulnerable オブジェクトを作成することはできません。

```
java.lang.IllegalArgumentException: Invulnerable value must be positive
Attack failed
```

まとめ

ファイナライザーは Java 言語が持つ残念な機能です。Java オブジェクトで使用されなくなったメモリーはガーベッジ・コレクターによって自動的に回収されますが、ネイティブ・メモリーやファイル記述子、ソケットなどのネイティブ・リソースを再利用するためのメカニズムはありません。そうしたネイティブ・リソースとインターフェースを取るために、Java に用意された標準ライブラリーには通常、適切なクリーンアップを実行できる `close()` メソッドがあります。しかしそれらのメソッドも、オブジェクトが適切に閉じられない場合にリソースのリークが発生しないようにするためにファイナライザーを使用しています。

上記以外のオブジェクトの場合、一般的にファイナライザーを避けた方が無難です。ファイナライザーは最終的に実行される場合ですら、いつファイナライザーが実行されるのかに関する保証はありません。ファイナライザーがあるということは、到達不能なオブジェクトはファイナライ

ザーが実行されるまでガーベッジ・コレクションが行われず、そのオブジェクトによってさらに別のオブジェクトも存続しているかもしれないということです。そのため、存続するオブジェクトの数が増加し、従って Java プロセスに使用されるヒープも増加します。

ガーベッジ・コレクションが行われるはずのオブジェクトをファイナライザーによって復活できるという機能は、ファイナライズ・メカニズムによる動作の結果としてそうなっているにすぎないことは明らかです。現在は、JVM の新しい実装を利用することで、ファイナライザーによって生じるセキュリティーの危険性からコードを保護できるようになっています。

ダウンロード

内容	ファイル名	サイズ
Code samples for this tip	j-fv.zip.zip	4KB

著者について

Neil D. Masson

Neil Masson は長年 Java 言語の開発とサポートに従事しています。彼は現在、Java リリースの品質とセキュリティーの改善に焦点を絞った業務を行っています。