Javaセキュアコーディングセミナー東京

第1回
オブジェクトの生成とセキュリティ

演習の解説

2012年9月9日(日)
JPCERTコーディネーションセンター
脆弱性解析チーム
戸田 洋三

演習[1]

演習[1]

```
class Dog {
  public static void bark() {
    System.out.print("woof");
  }
}

class Bulldog extends Dog {
  public static void bark() {}
}

public class Bark {
  public static void main(String args[]) {
    Dog d1 = new Dog();
    Dog d2 = new Bulldog();
    d1.bark();
    d2.bark();
  }
}
```

A. どのような出力が得られるか?
B. bark()メソッドがstatic宣言されていない場合の出力は?
C. メソッドがどのように実行されているか説明せよ

ヒント
Java言語仕様 §15.12 Method Invocation Expressions

演習[1]

A. どのような出力が得られるか?

```
$ java Bark
woofwoof$
```

B. bark()メソッドがstatic宣言されていない場合の出力は?

```
$ java Bark
woof$
```

C. メソッドがどのように実行されているか説明せよ

コンパイラおよびVMによる処理

```
class Dog {
  public static void bark() {
    System.out.print("woof");
  }
}

class Bulldog extends Dog {
  public static void bark() {}
}

public class Bark {
  public static void main(String args[]) {
    Dog d1 = new Dog();
    Dog d2 = new Bulldog();
    d1.bark();
    d2.bark();
  }
}
```

(2) コンパイル時のチェック
Dogのメソッドbark()は
static宣言されている

(3) 実行時処理
Dogのstaticメソッド
bark()を呼び出そう!

(1) コンパイル時のチェック
- d1,d2どちらもDog型の変数
- DogのメソッドbarK()があるのでOK

コンパイラおよびVMによる処理(staticメソッドでない場合)

```
class Dog {
  public void bark() {
    System.out.print("woof");
  }
}

class Bulldog extends Dog {
  public void bark() {}
}

public class Bark {
  public static void main(String args[]) {
    Dog d1 = new Dog();
    Dog d2 = new Bulldog();
    d1.bark();
    d2.bark();
  }
}
```

(2) コンパイル時のチェック
Dogのメソッドbark()は
virtual呼び出しだ

(3) 実行時処理
d1,d2が参照している
インスタンスをチェック!

(4) 実行時処理
それぞれのインスタンスから辿って
メソッドを呼び出そう!

(1) コンパイル時のチェック
- d1,d2どちらもDog型の変数
- Dogのメソッドbark()があるのでOK

演習[2]

演習[2]

A. どのような出力が得られるか?
B. なぜこのような出力が得られたのか説明せよ.
C. 適切な出力が得られるようにコードを修正せよ.

```
class Point {
  protected final int x, y;
  private final String name;
  protected String makeName() { return "[" + x + "," + y + "]"; }
  public final String toString() { return name; }
  Point(int x, int y) {
    this.x = x; this.y = y;
    this.name = makeName();
  }
}
public class ColorPoint extends Point {
  private final String color;
  protected String makeName() { return super.makeName() + ":" + color; }
  ColorPoint(int x, int y, String color) {
    super(x, y);
    this.color = color;
  }
  public static void main(String[] args) {
    System.out.println(new ColorPoint(4, 2, "purple"));
  }
}
```

ヒント: Java言語仕様 §15.12 Method Invocation Expressions

演習[2]

A. どのような出力が得られるか?

```
$ java ColorPoint
[4,2]:null
$
```

B. なぜこのような出力が得られたのか説明せよ.

→Point.toString() から出力
→name の値
→name には makeName() の返り値が代入されている

どの makeName() ?

ColorPoint の makeName()

つまり...
サブクラスのインスタンスが初期化される前にスーパークラスのコンストラクタがサブクラスのメソッドを呼び出した

実行の様子

```
class Point {
  protected final int x, y;
  private final String name; // Cached at construction time
  protected String makeName() { return "[" + x + "," + y + "]"; }
  public final String toString() { return name; }
  Point(int x, int y) {
    this.x = x; this.y = y;
    this.name = makeName();
  }

public class ColorPoint extends Point {
  private final String color;
  protected String makeName() { return super.makeName() + ":" + color; }
  ColorPoint(int x, int y, String color) {
    super(x, y);
    this.color = color;
  }
  public static void main(String[] args) {
    System.out.println(new ColorPoint(4, 2, "purple"));
  }
}
```

(3) ColorPointの makeName() が呼び出される

(4) 初期化前の color にアクセス

(2) Pointのコンストラクタ呼出し

(5) ここでようやくcolorの初期化

(1) ColorPointのコンストラクタ呼出し

演習[2]

このコードの問題点:
初期化される前のcolorにアクセスしている
(その結果, デフォルト値の null が出力されている)

元々の意図は?

初期化された後のcolorの値を出力したい

C.適切な出力が得られるようにコードを修正せよ.

どのように修正するか?

```
class Point {
  protected final int x, y;
  private final String name; // Cached at construction time
  protected String makeName() { return "[" + x + "," + y + "]"; }
  public final String toString() { return name; }

  Point(int x, int y) {
    this.x = x; this.y = y;
    this.name = makeName();
  }
}

public class ColorPoint extends Point {
  private final String color;
  protected String makeName() { return super.makeName() + ":" + color; }
  ColorPoint(int x, int y, String color) {
    super(x, y);
    this.color = color;
  }
  public static void main(String[] args) {
    System.out.println(new ColorPoint(4, 2, "purple"));
  }
}
```

どのように修正するか?

```
class Point {
  protected final int x, y;
  private String name; // 遅延初期化 (最初に使われたときにキャッシュされる)
  protected String makeName() { return "[" + x + "," + y + "]"; }
  public final synchronized String toString() {
    return (name == null ? name = makeName() : name);
  }
  Point(int x, int y) {
    this.x = x; this.y = y;
  }
}

public class ColorPoint extends Point {
  private final String color;
  protected String makeName() { return super.makeName() + ":" + color; }
  ColorPoint(int x, int y, String color) {
    super(x, y);
    this.color = color;
  }
  public static void main(String[] args) {
    System.out.println(new ColorPoint(4, 2, "purple"));
  }
}
```

コンストラクタからの makeName() 呼び出しを避ける

演習[3]

演習[3]

```
class Purse {
  private int i;

  public Purse(int arg) {
      i = arg;
  }
  public int get_i() { return i; }
  public void set_i(int iarg) { i = iarg; }
}

class User {
  private Purse p;
  public User(Purse arg) {
      p = arg;
  }
  public Purse get_p() {
    return p;
  }
}
```

A. User クラスのインスタンスを生成し，その private フィールド p が参照する Purse インスタンスの持つ値を変更する攻撃コードを書け.
B. A. でつくった攻撃コードが動作しないように元のコードを修正せよ.

攻撃コード例

こんなことすると……

```
class dc {
  public static void main(String[] args){
    Purse newp = new Purse(9999);
    User u = new User(newp);
    System.out.println(u.get_p().get_i());

    Purse p = u.get_p();
    p.set_i(1099);
    System.out.println(u.get_p().get_i());

    newp.set_i(999);
    System.out.println(u.get_p().get_i());
  }
}
```

get_p() の返り値を使って u の private フィールドをいじることができる

コンストラクタに渡したオブジェクトを使って u の private フィールドをいじることができる

修正例

```
class User {
  private Purse p;
  public User(Purse arg) {
      p = new Purse(arg.get_i());
  }
  public Purse get_p() {
    return new Purse(p.get_i());
  }
}
```

受け取ったオブジェクトはコピーを作って使う.

外に渡すオブジェクトはコピーを作って使う.

defensive copy : デフェンシブコピー

修正例(2) Purseクラスを公開する必要がなければ...

```
class User {

  private class Purse {
    private int i;
    Purse(int arg) { i = arg; }
    int get_i() { return i; }
    void set_i(int iarg) { i = iarg; }
  }

  private Purse p;
  public User(int iarg) {
      p = new Purse(iarg);
  }
  public int get_i() { return p.get_i(); }
  public void set_i(int i) { p.set_i(i); }
}
```

Purse クラスを入れ子クラスとして定義する. Purse の中身へのアクセスは必ず User クラスのメソッド経由とする.

演習[4]

演習[4]

```
class Authlet {
  int i;
  Authlet(int i0){
    if (checkarg(i0)) { this.i = i0; }
  }
  boolean checkarg(int i) throws IllegalArgumentException {
    if (i<0 || 100<i) {
      throw
        new IllegalArgumentException("arg should be positive < 100.");
    }
    return true;
  }
}
```

```
class Auth {
  private static Authlet a0;

  public static void checkAuth(Authlet a){
    if (a0 == null){
      if (a == null){
        System.out.println("invalid Authlet!");
        System.exit(1);
      }
      a0 = a;
    }
  }
}
```

演習[4]

```
class useAuth {
  public static void main(String[] args){
    Authlet au;
    try {
        au = new Authlet(Integer.parseInt(args[0]));
    } catch(IllegalArgumentException ex){
        au = null;
    }
    Auth.checkAuth(au);
    System.out.println("Authenticated!");
  }
}
```

訂正!
SecurityException ではなく
IllegalArgumentException

A. checkarg() による入力値チェックを回避する攻撃コードを書け.
B. A.で書いた攻撃コードに対する対策を行え.

プログラムの動作を確認する

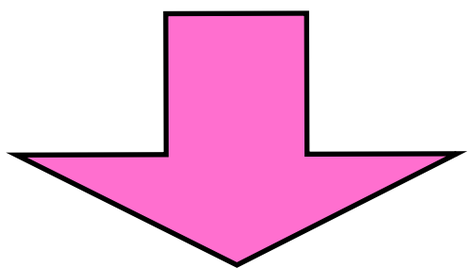
```
$ javac Authlet.java Auth.java useAuth.java
$ java useAuth
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at useAuth.main(useAuth.java:5)
$
```


```
$ java useAuth 101
invalid Authlet!
$
```


```
$ java useAuth 50
Authenticated!
$
```

Auth と Authlet の問題点

- Authlet のサブクラスをつくり, checkarg() の検査の後でフィールド i の値を更新可能
- Auth.checkAuth()は Authlet のインスタンスの中身をチェックしていない



- Authlet のサブクラスをつくり, フィールド i に任意の値を設定したインスタンスを作成可能
- 作成したインスタンスをAuth.checkAuth()にそのまま渡せば認証される

攻撃コード

```
class exploitAuthlet extends Authlet {
  exploitAuthlet(int e){
    super(10);
    this.i = e;
  }

  public static void main(String[] args){
    exploitAuthlet eal = new exploitAuthlet(102);
    Auth.checkAuth(eal);
    useAuth.main(new String[]{“1000”});
  }
}
```

checkarg() の後で i の値を再設定

不正な値でAuthletを作成, 登録

元のmainメソッドにはダミーの引数を渡す

exploitAuthlet の実行

```
$ javac exploitAuth.java
$ java exploitAuth
Authenticated!
$
```

値102のAuthletでパスしてしまう

対策
- Authlet のサブクラスをつくれないようにする
- フィールド i の値を再設定できないようにする
- Auth.checkAuth()で渡されたAuthletが持っている値をチェックする

修正例

```
final class Authlet {
  int i;
  Authlet(int i0){
    if (checkarg(i0)) { this.i = i0; }
  }
  boolean checkarg(int i) throws IllegalArgumentException {
    if (i<0 || 100<i) {
      throw
        new IllegalArgumentException("arg should be positive < 100.");
    }
    return true;
  }
}
```

```
class Auth {
  private static Authlet a0;

  public static void checkAuth(Authlet a){
    if (a0 == null){
      if (a == null){
        System.out.println("invalid Authlet!");
        System.exit(1);
      }
      a0 = a;
    }
  }
}
```