Javaセキュアコーディングセミナー東京
第2回
数値データの取扱いと入力値の検証
演習解説

2012年10月14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

Hands-on Exercises

- サンプルコード Unzip を修正しよう
- サンプルコード AltConst を修正しよう

Hands-on Exercise(1)

サンプルコード Unzip を
修正しよう

ZipBombの影響を受けるコード例

```
class Unzip {
  static final int BUFFER = 512;

  public static void main(String[] args) throws FileNotFoundException,IOException {
    BufferedOutputStream dest = null;
    ZipInputStream zis =
        new ZipInputStream(new BufferedInputStream(new FileInputStream(args[0])));
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null){
       System.out.println("Extracting: " + entry);
       int count;
       byte data[] = new byte[BUFFER];
       FileOutputStream fos = new FileOutputStream(entry.getName());
       dest = new BufferedOutputStream(fos, BUFFER);
       while ((count=zis.read(data,0,BUFFER)) != -1){
          dest.write(data, 0, count);
       }
       dest.flush();
       dest.close();
    }
    zis.close();
  }
}
```

コード例 Unzip の問題点

(A) 解凍後のサイズをチェックしていない

(B) 例外 ArrayIndexOutOfBoundsException が発生する可能性がある

(C) 既存ファイルを上書きする可能性がある

(D) その他?

これらの問題点を解決せよ!!

コード例 Unzip の問題点

(A) 解凍後のサイズをチェックしていない

(A)の対応
ZipBombの説明のなかで紹介済み

コード例 Unzip の問題点

(A) 解凍後のサイズをチェックしていない

(B) 例外 ArrayIndexOutOfBoundsException が発生する可能性がある

(B)の対応
例外 ArrayIndexOutOfBoundsException の発生原因は?
⇒ コマンドライン引数にファイル名が与えられることを想定しているため, 引数がない場合に例外が発生する
⇒ 引数の数をチェックしましょう

コード例 Unzip の問題点

(A) 解凍後のサイズをチェックしていない

(B) 例外 ArrayIndexOutOfBoundsException が発生する可能性がある

(B)の対応
例外 ArrayIndexOutOfBoundsException の発生原因は?
⇒ コマンドライン引数にファイル名が与えられることを想定しているため, 引数がない場合に例外が発生する
⇒ 引数の数をチェックしましょう

あるいは…
引数の数だけ処理を繰り返す
(引数がなければそのまま終了)

コード例 Unzip の問題点

(A) 解凍後のサイズをチェックしていない

(B) 例外 ArrayIndexOutOfBoundsException が発生する可能性がある

(C) 既存ファイルを上書きする可能性がある

(C)の対応
ファイル出力の前に，すでにファイルが存在していないかどうかチェックする

修正コード例(1/2)

```
class Unzip {
  static final int TOOBIG = 10000000;
  static final int BUFFER = 512;

  public static void main(String[] args) throws FileNotFoundException,IOException {
    if (args.length <= 0) return;
    BufferedOutputStream dest = null;
    ZipInputStream zis =
        new ZipInputStream(new BufferedInputStream(new FileInputStream(args[0])));
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null){
        if (entry.getSize () > TOOBIG) {
            throw new IllegalStateException(
                                “uncompressed size too huge: “ + entry.getName());
        }
        if (entry.getSize() == -1){
            throw new IllegalStateException(“uncompressed size unknown”);
        }
```

(…次のページに続く…)

修正コード例(2/2)

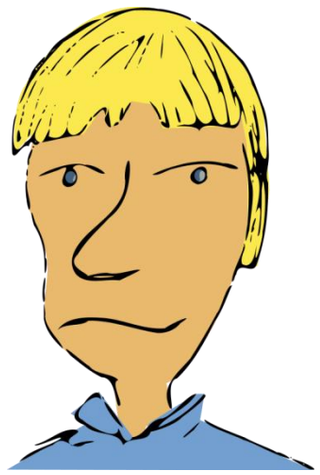
(… 前のページから…)

```
        System.out.println(“Extracting: “ + entry);
        int count;
        byte data[] = new byte[BUFFER];
         Path path = FileSystems.getDefault().getPath(entry.getName());
         Files.createFile(path); // exception if file already exists
        FileOutputStream fos = new FileOutputStream(entry.getName());
        dest = new BufferedOutputStream(fos, BUFFER);
        while ((count=zis.read(data,0,BUFFER)) != -1){
            dest.write(data, 0, count);
        }
        dest.flush();
        dest.close();
    }
    zis.close();
  }
}
```

ZipBomb対策に対するコメント

ところでZipEntry クラスの getSize() メソッドってどうやって展開後のサイズ調べてるのかな?

ZIP ファイルのヘッダに書いてある値をそのまま使ってるんじゃないのかなぁ? 標準ライブラリのコードちゃんと追っかけてみないと分かんないけど…



ZipEntryクラスの実装がどうなってるかは知らないが, ZIPファイルに記載されているメタ情報を信頼できないのであれば, 自分で圧縮データを展開して確認するというアプローチは必要になるんじゃないかな.

Hands-on Exercise(2)

サンプルコード AltConst を
修正しよう

Q. 以下のコードの問題点は何か?

```
class AltConst {
  int i;
  AltConst(){ this(10); }
  AltConst(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;
  }
}
```

(A) コンパイルエラーになる
(B) **checkarg()** による引数のチェックは役に立たない
(C) コンストラクタが二つ定義されている
(D) フィールド **i** に初期化子がない

A. 以下のコードの問題点は何か?

```
class AltConst {
  int i;
  AltConst(){ this(10); }
  AltConst(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;
  }
}
```

(A) コンパイルエラーになる

(B) **checkarg()** による引数のチェックは役に立たない

(C) コンストラクタが二つ定義されている

(D) フィールド **i** に初期化子がない

A. 以下のコードの問題点は何か?

AltConst のサブクラスをつくることにより, checkarg() によるチェックの後でフィールド i の値を変更できる.

```
class attack extends AltConst {
  attack(int arg) {
    super();
    this.i = arg;
  }
  public static void main(String[] args) {
    AltConst a = new attack(101);
    System.out.println("attack.i: " + a.i);
  }
}
```

A. 以下のコードの問題点は何か?

AltConst のサブクラスをつくることにより, checkarg() によるチェックの後でフィールド i の値を変更できる.

```
class attack extends AltConst {
  attack(int arg) {
    super();
    this.i = arg;
  }
  public static void main(String[] args) {
    AltConst a = new attack(101);
    System.out.println("attack.i: " + a.i);
  }
}
```

サブクラス化による攻撃への対策を行え!!

サブクラス化による攻撃への対策

A) サブクラスをつくれないように final クラスにする
B) 初期化後の i フィールドを変更されないよう final 宣言あるいは private 宣言する

B)の場合, checkarg() もオーバーライドされないように final 宣言すべし

Altconst クラスに対するコメント

そもそも checkarg() メソッドって検証成功で true を返し, 検証失敗では例外を投げるってなんかアンバランスだよね.

検証結果を true/false で返すか, void型のメソッドにして検証失敗のときは例外を投げるってしたほうがきれいだよね.

例外を投げるタイプだと, オブジェクトの初期化が途中終了しちゃうからファイナライザ攻撃の危険あるかもね.