マルチメディア通信と分散処理 111－15
コンピュータセキュリティ 20－15
（２００３．２．２７）

セキュアな Java プログラム作成のためのソースコード監査支援ツール

児島 尚* 中山 裕子* 川島 悟** 藤川 亮子***

*㈱富士通研究所 **㈱富士通インフォソフトテクノロジ ***富士通㈱

Web アプリケーションが一般的になるにつれ、それらに対する攻撃も増加している。特に、コード署名された Java アプレットは、悪用されるとユーザに直接被害が及ぶ可能性が高く、その安全性を確保することが重要である。アプレットが悪用される原因として、危険な実行パスの存在などいくつかの典型的なコーディング上の欠陥が知られている。このような欠陥を防止するにはソースコードの監査が有効だが、人手でおこなうには網羅性・効率の面で問題があり、支援するツールも不足しているのが現状である。そこで我々は、Java アプレットが悪用される問題に対処するためのソースコード監査支援ツールを開発した。

An audit-assisting tool for writing secure Java code

Hisashi Kojima* Yuko Nakayama* Satoru Kawashima** Ryoko Fujikawa***

* FUJITSU Lab. Ltd. ** FUJITSU INFO SOFTWARE TECHNOLOGIES LIMITED *** FUJITSU LIMITED

As web applications become common, the number of security breaches continues to grow steadily. Particularly, security of signed Java applets are quite important because users will suffer directly if they are exploited, and it is known that there are some typical coding defects that allow such exploitation. Although Source code auditing is a good way to find such defects, it is incomprehensive and inefficient to do it manually, and there is no good assisting tool suitable for our purpose. To cope with this problem, we have developed a tool to assist in auditing Java applet source code in security aspects.

1. はじめに

近年、Web を利用したシステム構築が一般的になっており、システムを構成するアプリケーションは Web アプリケーションと呼ばれている。Web アプリケーションは、サーバ側で実行されるものと、クライアント側の Web ブラウザ上で実行されるものにより構成されることが多い。一方、Web アプリケーションに対する攻撃も増加している。中でも、クライアント側アプリケーションへの攻撃は、ユーザが直接被害を受けることになるので、早急な対策が必要である。

我々は安全なクライアント側アプリケーションを構築するためのプラットフォームとして Java アプレットを選び、安全な Java アプレット作成技術について現状の問題点を検討した。そして、問題を解決する技術として、ソースコード監査支援ツールを開発したので、これを報告する。

2. 安全な Java アプレット作成技術の必要性

ここではまずクライアント側アプリケーションの実行モデル、特にコード署名モデルについて説明する。その中でよく使われる、署名付き Java アプレット、署名付き ActiveX コントロールについて、信頼されたアプリケーションが悪用されてしまう問題の重要性を指摘し、安全性の向上が必要であることを示す。そして、安全な実行プラットフォームとしては Java アプレットの方が優れていることから、我々は安全な Java アプレットの作成技術の確立を目指すべきであることを示す。

2.1. クライアント側の実行モデル

Web ブラウザ上で動作するアプリケーションには、Java アプレット、ActiveX コントロール、JavaScript などがある。これらのアプリケーションは Web サーバから自動的にダウンロードされて実行され、ユーザは特別なインストール作業を必要としないため、利便性が高い。しかし、中にはユーザに被害をおよぼすような悪意のあるものも存在するので、無制限に実行を許すことは非常に危険である。そこで、こういった自動的にダウンロードされて実行されるアプリケーションは実行を厳しく制限されており、ユーザに被害をおよぼす可能性のある動作、例えばローカルファイルへのアクセスなどは基本的に禁止されている。

2.2. コード署名モデル

しかしシステムによっては、ユーザのローカルファイルへのアクセスなど、危険性のある動作をクライアント側でおこないたい場合がある。そこで、ユーザが信頼できると判断したものについては、危険な動作の実行を許可する仕組みが用意されている。その信頼の拠り所としては、ダウンロードされてきたサイトの URL によって、「どこからきたか」で判断する方法や、アプリケーションに作成者が電子署名することによって、「誰が作成したか」で判断する方法がある。

このような判断をする際には、どの URL を許可するのか、どの作成者を許可するのか、ということを決める必要がある。これらをあらかじめ Web ブラウザに設定しておく方法があるが、これだとユーザに設定の手間が生じることになる。一方、アプリケーションの電子署名に基づく方法では、あらかじめ設定しておくほかに、ダウンロードされてきた時点でユーザに署名者の証明書情報を示し、実行の許可を判断してもらう、という仕組みが用意されている。この場合、あらかじめ設定する手間が省けるので、利便性の面から利用されることが多い。ここではアプリケーションへの電子署名をコード署名と呼び、上記のような動的な実行を許可する仕組みを、コード署名モデルと呼ぶ。よく使われるものには、署名付き Java アプレット、署名付き ActiveX コントロールがある。署名付き Java アプレットには、Microsoft Internet Explorer の JavaVM で動作するもの、Netscape

Communicator 4.x に付属の JavaVM で動作するもの、そして Sun が提供している、Web ブラウザへのプラグインである Java2 Plug-In で動作するものがある。一方、署名付き ActiveX コントロールは Microsoft Internet Explorer でのみ動作する。

2.3. 署名付きアプリケーションの悪用

しかし、コード署名モデルに基づいて信頼できると判断されたアプリケーションには、悪意を持った他のアプリケーションからその機能を悪用されてしまう、という問題がある。

過去に、Netscape 社の署名付き Java アプレットが悪用されて、任意のローカルファイルが読み出されるという問題や、Microsoft 社の署名付き ActiveX コントロールが悪用されて、任意のプログラムをダウンロードされて実行されるという問題が実際に起こった[1]。最近では、主要ベンダのパソコンにインストールされていた ActiveX コントロールに脆弱性が発見された[2]。

これはサーバ側ではみられない、クライアント側特有の問題といえる。まず、ユーザの環境が破壊されるなど、ユーザが直接的な被害を受けてしまうという特徴がある。また、一部のアプリケーションはインストールされるが、そのことをユーザにあまり意識させない。その結果、対策として修正パッチを配布しようとしても、ユーザにパッチの必要性を認識させることが難しくなる。さらに、アプリケーションの配布を停止したとしても、すでに流出した脆弱性のあるアプリケーションが、悪意のあるサーバから再配布される恐れがあり、その場合は開発者側でのコントロールが難しくなる。

2.4. Java と ActiveX コントロールの安全性比較

上記の問題の他に、ActiveX コントロールには Java に比べて安全性に問題がある部分がいくつか存在し、安全なアプリケーションを実行するためのプラットフォームとしては適していないことが指摘されている[3][4]。具体的には、メモリ保護機能のない C 言語などで作成されることが多く、バッファオーバーフローなどの脆弱性が発生しやすいこと、細かいアクセス制限ができないことなどがあげられる。

以上のことから、我々は、特に安全な署名付き Java アプレットを作成するための技術を確立することを目指した。

3. Java アプレットの安全性を脅かす問題

前述のような事例から得られた知見から、問題と対策を経験則としてまとめたガイドラインがいくつか発表されている。そうした経験則をまとめたガイドラインは、これまでにいくつか発表されている。Felten らによる[5]では、Java の安全性に関する、基本的な 12 ヶ条のアンチパターンを提唱している。Java の設計者である Sun 自らも、安全なプログラムを書くためのガイドラインとして[6]を公開しており、Java のセキュリティ設計者 Gong により[7]も出版されている。また、IPA/ISEC のセキュアプログラミング講座[8]も参考になる。さらに我々も過去に、これらを元に参照性・網羅性などを向上させたガイドラインを作成した[9]。

これらのガイドラインを分析・整理した結果、Java アプレットの安全性を脅かす問題として、主に以下のものがあげられることが分かった。

a) 危険な実行パス
b) プログラミングインターフェイス
c) native メソッド
d) オブジェクト直列化
e) その他

特に、 a)の危険な実行パスの問題は、Java アプレットの安全性を考える上で必須だと考える。以下では、まず危険な実行パスの問題について詳しく解説し、残りの問題については概略を述べる。

3.1. 危険な実行パスの問題

ここでは、危険な実行パスが原因となった、Netscape 社の事例[1]を使い、問題の本質を示す。そのためには、まず Java におけるアクセス制御の仕組みを理解しておく必要があるので、これを説明する。

3.1.1. 危険なメソッドとアクセス権

Java では、ファイルアクセス等の危険な動作は、すべて特定のメソッドに対応付けられている。そして、それらのメソッドを呼び出すには対応するアクセス権が必要であり、呼び出し元がアクセス権を持たない場合は基本的にセキュリティ例外が発生する。表 1 は Netscape JavaVM における危険なメソッドの例である。

プログラムに与えられるアクセス権はシステムの設定ファイル等で定義され、プログラムのロード元の URL、プログラムの作成者の証明書、などによって許可するアクセス権を設定する。また、2.2 で述べたコード署名モデルをサポートするシステムでは、プログラムがロードされた時にユーザがアクセス権の許可を判断する方法も可能である。

表 1 Netscape JavaVM における危険なメソッドの例

危険な動作	危険なメソッド	必要なアクセス権
ファイル読込	*java.io.FileInputStream* FileInputStream(String name)	UniversalFileRead
VM 終了	*java.lang.Runtime* System.exit(int i)	UniversalExitAccess

3.1.2. コールスタック検査

呼び出し元がアクセス権を持っているかどうかの検出は、コールスタック検査という手法で実現されている。危険なメソッドが呼び出されると、システムはその時点のコールスタックを走査し、各呼び出し元が対応するアクセス権を許可されているかどうか検査する。もし許可されていない呼び出し元があった場合、セキュリティ例外を発生させて危険なメソッドは実行されない。図 1 にその概要を示す。

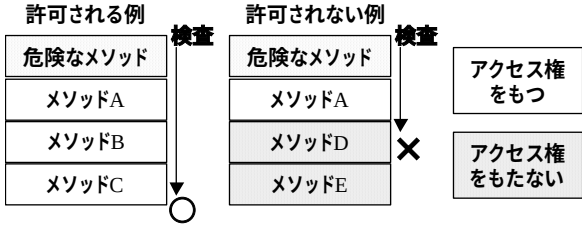

図 1 コールスタック検査

3.1.3. 特権コード

上記のコールスタック検査では、危険なメソッドのすべての呼び出し元が必要なアクセス権を持っていなければならないので、柔軟性を欠くことがあった。例えば、アクセス権を持つメソッド A が自分の責任のもとで、アクセス権を持たないメソッド D・E にも限定された機能を提供したい時には、あらかじめ設定ファイルなどによって、対応するアクセス権

をメソッド D･E に許可しておく必要がある。これは対象が不特定多数になると制御が非常に難しくなってくる。そこでこのような用途のため、特権コードという仕組みが用意されている[14][15][16]。

メソッド A は自らを特権コードとして実装することができる。すると、コールスタック検査では特権コードが見つかると、必要なアクセス権をメソッド A が持っていれば、その時点で検査に合格したとみなし、それ以降の呼び出し元は検査されないようになる。したがって、アクセス権を持たないメソッド D･E から間接的に危険なメソッドを呼び出せるようになる。図 2 はその概要である。

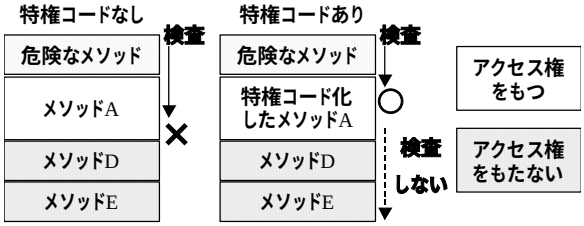

図 2　特権コードによるコールスタック検査の拡張

この仕組みにより、メソッド A は柔軟なアクセス制御を自ら行うことができるが、その反面、アクセス権制御という重要なセキュリティチェックの責任を負うことになる。万が一、メソッド A のセキュリティチェックに不備があった場合、悪意を持ったメソッド、例えばメソッド D･E などにその機能を悪用されるという可能性が出てくる。

3.1.4.　Netscape 社の事例

Netscape 社の事例ではまさに上記の特権コードを使用しており、セキュリティチェックに不備があったケースである。図 3 は事例の報告を元にした類似のソースコードである。

```
public class Victim extends Applet {
  // 読み込むファイル名
  private String file;

  // 読み込むファイル名を指定
  public void setFile(String file) {
    this.file = file;
  }

  // setFile()で指定したファイルをオープン
  public InputStream openFile()
                    throws IOException {
    // これ以降のコードを特権化する宣言
    // ファイル読込アクセス権を提供
    PrivilegeManager.enablePrivilege(
        "UniversalFileRead");
    // ファイルをオープンする危険なメソッド
    return new FileInputStream(file);
  }
}
```

図 3　脆弱性のあるアプレットの類似ソースコード

上記の例では、openFile()が特権化されたメソッドである。PrivilegeManager.enablePrivilege()は Netscape JavaVM における特権コードの宣言方法であり、このメソッドの呼び出し場所から openFile()の終了場所までが特権コードとして認識される。"UniversalFileRead"はファイル読込のためのアクセス権を示し、openFile()がこのアクセス権を呼び出し元に提供することをシステムに伝えている。そして特権化されたコード内で、ファイルを読み込む危険なメソッド、FileInputStream()が呼び出されており、InputStream が最終的に返される。ここで渡される引数 file は、入力された値がそのまま渡されており、何らセキュリティチェックがされていない。よって、このアプレットは図 4 のようにして容易に悪用することができる。

```
public class BadLot extends Applet {
  :
  Victim victim = new Victim();
  // "/etc/passwd"を指定
  victim.setFile("/etc/passwd");
  // 特権化されたメソッドを呼び出す
  // "/etc/passwd"の InputStream を取得
  InputStream is = victim.openFile();
  // "/etc/passwd"をサーバに送る処理・・・
  :
}
```

図 4　攻撃者のアプレット BadLot のソースコード

3.1.5.　危険な実行パス

上記の事例を考えてみると、攻撃者の望みは FileInputStream()を呼び出して"/etc/passwd"を読み出すことであるが、直接 FileInputStream()を呼び出しても例外が発生してしまう。しかし openFile()のような特権コード化されたメソッドを経由することにより、攻撃が可能になるのである。したがって、攻撃者が狙うターゲットは、public メソッドのような他のプログラムから呼び出せる場所から、特権コードを経由して、最終的に危険なメソッドを呼び出すような実行パスである。我々はこの実行パスを、危険な実行パスと呼び、最も注意すべき場所であると考えている。

また、Microsoft や Netscape の JavaVM では、署名付きアプレットでファイルアクセス等の拡張的なアクセス権を使う場合には、特権コードの実装が必要になる。そのため危険な実行パスが生じやすく、悪用される危険性が高い。

以上のことから、危険な実行パスを作りこまないことが重要であるといえる。

3.2.　その他の問題

その他の問題について概略を説明する。

b) プログラミングインターフェイス ･･･ クラス･インターフェイス･メソッド･フィールドなど、他のプログラムからアクセスできるインターフェイス。これらが不必要に多いと悪用される可能性が高まる。

c) native メソッド ･･･ native メソッドは Java の保護機能が働かず、バッファオーバーフロー等が起こりやすい。

d) オブジェクト直列化 ･･･ 直列化されたバイト列から private フィールドが漏洩したり、改ざんされたバイト列により不正な値をセットされる恐れがある。

e) その他 ･･･ その他一般的な問題。

詳しくは第 3 章のはじめで紹介したガイドラインを参照してほしい。

4.　有用な監査ツールの不足

上記のガイドラインにより、Java アプレットの安全性における注意点は明らかになっている。しかし、ガイドラインによる開発者への教育効果だけでは不十分であり、開発されたプログラムが上記のガイドラインに従っているかどうか、ソースコードを監査して確かめる必要がある。

一般に、ソースコードは膨大な量であることが多く、監査を支援するツールを用いた方が効率的である。Java アプ

レットの安全性監査のためのツールには、[5]の 12 ヶ条の規則に基づいたツールが発表されているが[10]、ツール自体が公開されていない。規約チェックや性能や品質などの監査のためのツールは多く製品化されており、Borland 社の Together ControlCenter[11]、(株)東陽テクニカの J.Taster[12]、当社の SIMPLIA/JF Kiyacker[13] (V20L10、本稿執筆時点での製品版) などがあるが、安全性の監査機能が十分でない。

以上のことから、第 3 章で述べたような問題を有効に監査するためのツールが必要であるといえる。

5. Java アプレットの安全性監査のためのソースコード監査支援ツール

上記の問題点を解決するために、我々は Java アプレットの安全性のためのソースコード監査を支援するツールを作成した。本ツールは当社の製品 SIMPLIA/JF Kiyacker[13] 次版 (V20L20) (以下、Kiyacker) の追加機能として実装されている。ここでは、本ツールの機能について説明し、3.1.4 で述べた Netscape 社のソースコードを実際に監査することによって、本ツールで効果的な監査が可能であることを示す。

5.1. 背景となる技術

本ツールでは Kiyacker の持つ以下のソースコード解析機能を利用している。

精度の高いコードパターン検出機能

ソースコードを構文解析して型解決等をおこなったモデル化されたオブジェクトを扱うことにより、単純なパターンマッチングとは異なる精度の高い検出が可能である。

コールチェーン検出機能

指定されたメソッドについて、そのメソッドを呼び出しているメソッドを再帰的におおもとの呼び出し元メソッドまでさかのぼり、メソッド呼び出し列 (コールチェーンと呼ぶ) として検出することができる。ただし、3.1.2 で述べたような実行時のコールスタック検出に比べ、静的な検出では呼び出しているかの判定があいまいになる場合がある。よって、呼び出している可能性のあるメソッドはすべて検出するようにしている。

一方、Kiyacker はデータフロー解析の機能は持っていない。したがって、変数に対する外部入力による影響、外部への漏洩については、本ツールではコールチェーン情報を提供する程度で、人手による監査が必要になる。

ソースコード解析機能の他には、検出結果の GUI 表示、GUI 画面の検出項目からソースコードの該当場所へのタグジャンプなど、監査作業を支援する機能を持っている。

5.2. 監査を支援するツールとしての位置付け

前述の通り、本ツールを使った監査では、データフロー解析は監査者が手動でおこなう必要がある。また、問題の種類によっては、ツールは監査すべき場所を示す程度で、監査者による詳細な確認が避けられない場合がある。そこで本ツールでは、人手でおこなう作業まで考慮し、監査者の作業を全体的に支援することも重要な目的としている。

具体的には、各監査に必要な手順を検討し、本ツールを用いて監査者がどのように監査を進めればよいかを明確にした。また、報告書の作成など、監査とは直接関係ないが、手動では手間がかかる作業も効率化している。

以下では、3.1 で述べた、最も重要である危険な実行パスの問題について、その監査の支援方法を詳しく説明する。そして、3.2 で述べたその他の問題の監査支援、および監査作業全体の支援について概略を述べる。

5.3. 危険な実行パスの問題の監査支援

既に述べた通り、危険な実行パスとは、他のプログラムから呼び出せる場所から、特権コードを経由して、最終的に危険なメソッドを呼び出す実行パスである。まずこのパスをすべて検出する必要がある。

5.3.1. 危険な実行パスの検出

実行パスとはコールチェーンのことであり、これは Kiyacker のコールチェーン検出機能により容易に検出できる。しかし、通常のコールチェーンとは異なり、間に特権コードが存在するものだけを検出する必要がある。そこで我々は、「外部～特権コード」、「特権コード～危険なメソッド」という二種類のコールチェーンを検出し、両者で特権コードが同一のものを組み合わせて、危険な実行パスとして検出するようにした。そして、危険な実行パスの監査の単位は、「特権コード～危険なメソッド」によって識別することにした。

外部から呼び出せる場所とは、基本的には public または protected のメソッドである。しかし、Microsoft JavaVM、Netscape JavaVM、Sun JRE 1.2.1 以前の JavaVM では、デフォルトアクセスの扱いに問題があり、デフォルトアクセスのメソッドにも他のプログラムが容易にアクセスできる[1]。よって、これらの JavaVM 向けの監査ではデフォルトアクセスも定義に含めている。

また、特権コード化の方法も JavaVM によって異なる。Microsoft JavaVM および Netscape JavaVM では、図 3 のように特権コードの開始を示すシステムメソッドを呼び出した場所から、メソッドの終了場所までが特権コードになる。一方、Sun Java2 では、匿名クラスにメソッドを実装して、システムメソッドからコールバックすることにより、メソッド全体が特権コードになる。本ツールでは JavaVM ごとに特権コードの違いを考慮した検出が可能である。

5.3.2. 手順(1)～(6) 危険な実行パスの監査

我々は、安全性の網羅的・効率的な確認方法として、以下の手順を定義した。

(1) 特権コードは簡潔な記述になっているか

3.1.3 で述べたように、特権コードではコード自らがセキュリティチェックをおこなう必要がある。しかし、この部分の記述が長すぎると、一般的にコーディングミスが発生しやすいので、簡潔に記述することが望ましい。そこで、簡潔かどうかの基準を特権コードから危険なメソッドへのメソッド呼び出しの数とし、5 以上は冗長であるとした。ツールではメソッド呼び出しの数を出力する。

(2) 不必要なアクセス権が与えられていないか

Microsoft や Netscape の JavaVM では、特権コードの開始メソッドで有効にするアクセス権を宣言する。そこでは要求される機能に最低限必要なアクセス権を宣言すべきである。宣言されたアクセス権と、危険なメソッドの呼び出しに最低限必要なアクセス権は等しくなければならない。ツールは両方のアクセス権を出力する。

(3) 呼び出されるだけで危険なメソッド

System.exit(int)などは、引数に関わりなく JavaVM を強制終了できるので、呼び出されるだけで危険である。ツールはこれに該当するメソッドを出力する。

(4) 危険な実行パスはできるだけ少なくする

危険な実行パスの数が多いと、悪用できるパスが存在する可能性が高くなる。ツールは「特権コード～危険なメソッド」ごとの、「外部メソッド～特権コード」の数を出力する。

(5) 危険なメソッドへの入力

ほとんどの危険なメソッドは引数にその動作を依存している。この引数に外部からの不正な値が入力されていると危険である。基本的にこの部分はデータフロー解析が必要であり、人手で監査することになるが、本ツールではコールチェーン情報を提供することで簡単な支援をおこなう。

(6) 危険なメソッドの出力

危険なメソッド呼び出しの戻り値は機密情報を含んでいることが多い。この戻り値が外部に漏洩しないかどうか確認する必要がある。(5)と同様にデータフロー解析を必要とするので、人手で監査し、ツールは簡単な支援をおこなう。

上記の手順にしたがうことにより、網羅的・効率的な監査が期待できる。

5.4. その他の問題の監査支援

その他の問題の監査支援について概略を述べる。

b) プログラミングインターフェイス ・・・ 各インターフェイスについて、実行に最低限必要なアクセススコープを解析し、不必要に大きくないかどうか監査する等。

c) native メソッド ・・・ 他のプログラムから native メソッドを呼び出せる実行パスを検出し、危険な実行パスとほぼ同様に監査する等。

d) オブジェクト直列化 ・・・ java.io.Serializable を実装しているクラスを検出する等。

e) その他 ・・・ java.applet.Applet を継承しているクラスを検出する等。

5.5. 監査作業全体の支援

監査作業全体の効率化のためには、監査に直接関係がない作業についても支援することが重要である。通常、こうした安全性の監査には専門的な知識が必要だが、我々は一般の開発者でも監査がおこなえるように、それらのノウハウを分かりやすいガイドラインにまとめた。また、最終的に報告書の作成作業の手間を省くため、Excel 形式の定型化された報告書を自動的に生成できるようにした。

5.6. ツールを使用した監査例

ここでは典型的な脆弱性のある危険な実行パスとして、3.1.4 で述べた Netscape 社の事例[1]の類似ソースコードを本ツールで実際に監査することにより、本ツールが脆弱性を有効に監査できることを示す。監査は図 3 の類似ソースコードを対象とし、危険な実行パスについてのみおこなった。

5.6.1. 危険な実行パスの検出

図 3 のソースコードを本ツールで解析した結果、1 つの危険な実行パス、および各監査項目の補助情報が検出された。表 2 にその抜粋を示す。

表 2 ツールの出力の抜粋

外部メソッド	*Victim*.openFile()
特権コード	[Netscape JavaVM 方式]
危険なメソッド	new FileInputStream(String)
(1)特権コードの長さ	1
(2)最低限必要なアクセス権	最低限必要：UniversalFileRead 実際に取得：UniversalFileRead
(3)引数に関係なく危険か	危険ではない
(4)実行パスの数	1

5.6.2. (1)-(6) 各項目の監査

上記の出力結果をもとに、(1)-(6)の各項目の監査をおこなう。

(1) 1 は目安の 5 よりも少ないので問題ない。
(2) UniversalFileRead は実際に取得しているものと最低限必要なものが同じアクセス権なので問題ない。
(3) 危険ではないので問題ない。
(4) パスが存在するので、なくせるかどうか検討が必要。
(5) new FileInputStream(String)の引数は、コールチェーン情報を参考に手動でデータフロー解析すると、Victim.setFile(String)によって外部から設定できることが分かる。これは問題である。
(6) new FileInputStream(String) の戻り値は、openFile()の戻り値として外部に漏洩する。これは問題である。

以上より、(4)、(5)、(6)について問題があることが分かる。対策としては、まず、(4)で危険な実行パスをなくす方法が考えられる。この場合、setFile()、openFile()を private にすればよく、(5)、(6)も解決できる。単体のアプレットであればこの対策が最も確実である。Victim クラスが他のプログラムから呼び出されることを目的としている場合は、(5)に関連して、指定できるファイル名を一時ディレクトリに限定する、ファイルダイアログを表示してユーザの選択したファイルのみ指定するといった対策が考えられる。そうした対策が取れない場合は、脆弱性を無視することはできないので、設計の見直しが必要になるだろう。

このようにして、本ツールにより、Netscape の事例でみられたような、典型的な脆弱性のある危険な実行パスを有効に監査できることが分かる。

6. 評価

本ツールについて、社内資産の Java アプレットによる評価をおこなった。ここでは表 3 のような Java アプレットを対象とし、網羅性・監査効率について評価した。

表 3 評価対象のアプレット

番号	対象 JavaVM	行数	署名	既知の重大な脆弱性
A	Netscape	340	あり	・ファイル改竄・漏洩
B	Microsoft	5k	あり	・VM が強制終了 ・証明書情報が漏洩
C	Java2	80k	あり	なし

上記で使用した資産は開発途中のものであり、製品版

ではこれらの脆弱性は修正されていることを明記しておく。

6.1. 網羅性の評価

上記の Java アプレットについて、作成したツールによる検出をおこなった結果、表 4 のような結果が得られた。

表 4 網羅性の評価結果

番号	既知の重大な脆弱性	その他の注意点
A	全て検出された	7 個
B	全て検出された	166 個
C	なし	6721 個

いずれの場合も、既知の重大な脆弱性について正しく検出することができた。また、直接的には重大な脆弱性とはならないが、安全性の上で問題があると考えられる注意点（プログラミングインターフェイスの問題等）についても、網羅的に監査することができた。ステップ数が大きくなると検出される脆弱性の数も大きくなるが、これは監査ツールとしての要件として、誤検出よりも検出漏れを防ぐことに重点を置いているためである。

6.2. 監査効率の評価

ツールの目的の一つである、監査効率の向上についても評価をおこなった。社内の評価者により、監査にかかった時間について表 5 のような効果が報告されている。

表 5 監査効率の評価結果

番号	ツールなし	ツールあり
A	1 時間	30 分
B	8 時間	2 時間
C	50 時間	4 時間

上記の結果から、ツールを使用することによって数倍の監査効率の向上が見込まれることが分かる。特に、大規模なプログラムにおいて大幅な向上がみられる。

以上の評価結果より、我々のツールが、網羅性、監査効率の点で効果が得られることが確かめられた。

7. まとめ

我々は、クライアント側アプリケーションの安全性向上を目指し、特に署名付き Java アプレットを対象とした、ソースコード監査支援ツールを開発した。

本ツールを用いると、Java アプレットの安全性に関する問題点を網羅的･効率的に監査できる。特に重大な問題である危険な実行パスについても、すべての危険な実行パスを検出し、それらのパスを、過去の知見から得られた 6 つの観点から監査できる。監査作業を全体的に効率化するため、監査に必要なノウハウを集約した解説書や、監査報告書の自動生成機能なども開発した。過去の脆弱性事例に類似のアプレットと、社内資産を用いて評価した結果、実際に問題点を検出できることを確かめた。また、監査に必要な工数を大幅に短縮できることも確認できた。なお、本ツールは今春発売予定の SIMPLIA/JF Kiyacker V20L20 に搭載される予定である。

今後の課題として、サーバ側 Java アプリケーションの監査支援機能の強化があげられる。サーバ側で重要な監査事項の一つに、入出力データのチェックに関するものがある。これを怠るとコマンドインジェクションやクロスサイトスクリプティングなどの脆弱性につながる可能性があるため、入出力データの効果的なチェック方法についても検討をおこなっていきたい。

謝辞

富士通（株）の直田 繁樹氏、中島 哲氏、（株）富士通インフォソフトテクノロジの上野 良造氏をはじめ、SIMPLIA/JF Kiyacker V20L20 の開発に携わったすべての方々に感謝いたします。

参考文献

[1] 児島尚, 丸山宏, “Java のコード署名に関する議論”, 第 1 回インターネットテクノロジーワークショップ, 1998.
[2] “CyberSupport のセキュリティに関する重要なお知らせ”, http://www.cybersupport.justsystem.co.jp/sony/index.html, （株）ジャストシステム, 2002.
[3] E. W. Felten, “Security Tradeoffs: Java vs. ActiveX,” http://www.cs.princeton.edu/sip/faq/java-vs-activex.html, the Princeton Secure Internet Programming Team, 1997.
[4] “Results of the Security in ActiveX Workshop,” http://www.cert.org/reports/activeX_report.pdf, CERT Coordination Center, 2000.
[5] G. McGraw, E. W. Felten, “Securing Java™,” Wiley Computer Publishing, 1999.
[6] “Security Code Guidelines,” http://java.sun.com/security/seccodeguide.html, Sun Microsystems, Inc., 2000.
[7] L. Gong, “Inside Java™2: Platform Security,” Addison Wesley, 1999.
[8] “セキュア Java プログラミング”, http://www.ipa.go.jp/security/awareness/vendor/programming/a03.html, IPA/ISEC, 2002.
[9] 児島尚, 鳥居悟, ”セキュアな Java ソースコードの作成とその監査手法”, 第 15 回 CSEC 研究会, 2001.
[10] J. Viega, et al, “Statically Scanning Java Code: Finding Security Vulnerabilities,” IEEE Software 17(5), 2000.
[11] “Borland Together ControlCenter™,” http://www.borland.com/together/controlcenter/index.html, Borland Software Corporation, 2002.
[12] “J.Taster”, http://www.toyo.co.jp/ss/jtaster/index.html, （株）東陽テクニカ, 2002.
[13] “SIMPLIA/JF Kiyacker”, http://software.fujitsu.com/jp/simplia/syoukai/jf-kiyacker_pc.html, 富士通（株）, 2002.
[14] L. Gong, “Java Security: Present and Near Future,” IEEE Micro 17(3), 1997.
[15] “Microsoft SDK for Java 4.0 Documentation,” http://www.microsoft.com/java/sdk/, Microsoft Corporation, 2002.
[16] “Introduction to the Capabilities Classes,” http://developer.netscape.com/docs/manuals/signedobj/capabilities/index.html, Netscape Communications Corporation, 1997.