Javaアプレット保護ツールCozilet のJava Web Start および署名なしプログラムへの適用

兒島 尚[†] 金谷 延幸[†]

RIA（Rich Internet Application）とよばれるWebアプリケーションのクライアント側技術が広く利用されているが，信頼されたプログラムが攻撃者に悪用されるという問題がある．我々は過去に，RIAの1つであるJavaアプレットについて，再構成攻撃という攻撃により署名付きJavaアプレットが悪用される危険性を指摘した．そして，開発者の注意だけでは対策が難しいことから，Coziletと称する対策ツールを作成した．本ツールは対象のアプレットをカプセル化することにより再構成を防ぐ．また，対象アプレットおよびJava Plug-in実行環境に対して透過的なので容易に適用できる．しかし，本ツールの対象は署名付きアプレットに限定されていた．再構成攻撃は署名付きアプレットだけの問題ではなく，アプレットに似たJavaベースのRIAであるJava Web Startアプリケーション，そして，署名されずにポリシファイルでアクセス権を与えられるプログラムもこの攻撃の対象となりうる．我々はこれらのプログラムに対しても再構成攻撃が可能であることを示し，本ツールを改良することで対処した．本論文ではその詳細について説明する．

Support of Java Web Start and Unsigned Programs in Cozilet: A Protection Tool for Java Applets

Hisashi Kojima[†] and Nobuyuki Kanaya[†]

Client-side technologies of Web applications, called RIA (Rich Internet Application), are widely used, but an attacker may abuse trusted RIA components. In the past, we showed that a Java applet which is a kind of RIA was vulnerable to an attack which we called a malicious recomposition attack. It is difficult for developers to prevent this attack only through their careful design and programming. Therefore, we developed a prevention tool called Cozilet. The tool prevents recomposition of applet components by encapsulating them, and is easily applicable to applets because it is transparent not only to applets, but also to Java Plug-in implementations. However, the tool was applicable only to signed applets. The attacks affect not only signed applets, but also Java Web Start which is Java based RIA like applets and unsigned trusted Java programs which are granted permissions by access control lists called policy files. We showed that these programs were also vulnerable to the attacks, and remedied them by improving the tool. In this paper, we describe details of the attacks and countermeasures.

1. はじめに

近年，RIA（Rich Internet Application）とよばれる技術が広く利用されている．RIAとは，Webアプリケーションにおいて，従来のHTMLベースであるユーザインタフェースの操作性や表現力を向上させるための技術の総称である．たとえば，Javaアプレット，Java Web Start [10]，ActiveXコントロールなどであり，最近ではAjax（Asynchronous Javascript And XML）もその1つとしてあげられる．RIAは自動的にダウンロードされて実行されるという特徴があり，事前に実行環境の準備が必要な場合もあるが，エンドユーザへの負担が少ないのが利点である．しかし，安全のため，基本的にプログラムには厳しいアクセス制限を課し，電子署名などにより出所が保証された場合に限りアクセス制限を緩める方式がとられている．特に，電子署名がある場合，実行時に署名者をエンドユーザに提示してアクセス制限の解除を尋ねる方式がある．この方式は事前にエンドユーザがセキュリティ設定などを変更する必要がないのでしばしば利用されている．たとえば，電子申請系のシステムではスマートカードを使って申請文書に電子署名を施す場合があるが，通常のRIAではスマートカードのようなローカルデバ

† 株式会社富士通研究所
Fujitsu Laboratories Ltd.

イスにアクセスすることは許されていない．そこで，署名付き Java アプレットを利用することが多い．

しかし，電子署名されたプログラムは攻撃者に再利用されてその機能を悪用される恐れがある．我々は過去に署名付きアプレットを構成する署名付き JAR ファイルを，攻撃者の用意した JAR ファイルと組み合わせることで，攻撃者が署名付きアプレットの機能を悪用できることを示した[5]．我々はこの攻撃を再構成攻撃とよぶ．この攻撃はアプレットだけではなく，再利用可能な署名付き部品を利用した RIA 一般にあてはまる．また，サーバ側の脆弱性とは異なり，被害の拡大を抑えるのが難しいという特徴がある．なぜなら，サーバ側の脆弱性についてはサーバを停止すれば被害の拡大を防げるが，RIA では攻撃者が入手した署名付き部品を自由に再配布できるからである．再構成攻撃への対策は急務といえる．そこで，署名付きアプレットを再構成攻撃から守るため，我々は Cozilet と称するツールを作成した[5]．本ツールは，Sun のアプレット実行環境である Java Plug-in のアプレット配備方式の脆弱さを補うものであり，再構成が難しい安全な配備方式をアプレットに強制する．また，対象のアプレットや既存の Java Plug-in 実行環境を変更することなく適用できるので，開発者やエンドユーザに負担を強いることがない．本ツールにより署名付きアプレットへの再構成攻撃を容易に防ぐことができる．

だが，本ツールには署名付きアプレットにしか適用できないという制限があった．再構成攻撃の対象となるのは署名付きアプレットだけではない．まず 1 つ目は Java Web Start（以降，JWS）アプリケーションである．近年，JWS はアプレットとともに一般的になりつつある．JWS はアプレットと同様に Sun が提供する Java ベースの RIA 実行環境であるが，アプレットが Web ブラウザ上で動作するのに対し，JWS アプリケーションはスタンドアロンなデスクトップアプリケーションとして動作するという違いがある．また，アプレットがつねにオンラインでの動作を前提としているのに対し，JWS アプリケーションは初回のダウンロード以降はオフラインでも動作できる．そこで，通常のオンライン申請ではアプレットを提供するが，オフラインでも利用したいエンドユーザのために JWS アプリケーションを提供する場合がしばしばみられる．我々は過去に JWS アプリケーションに対する再構成攻撃について検討し，アプレットと同様の攻撃が可能であることを示した[6]．また，アプレットと JWS はいずれも JAR ファイルを部品として利用するが，アプレットとして提供した JAR ファイルが JWS アプリケーションとして悪用される恐れがある．また，その逆も考えられる．アプレットの安全性を高めるためにも JWS アプリケーションに対する再構成攻撃への対策は欠かせないといえる．

2 つ目は，署名はされていないが信頼されたものとして扱われるアプレットおよび JWS アプリケーションである．ここではこれらを署名なしプログラムとよぶ．Java Plug-in および JWS では，ポリシファイルとよばれるアクセス制御リストを適切に設定することにより，JAR ファイルに署名がなくても配備された URL などに基づきアクセス制限を緩める仕組みがある[11]．たとえばイントラネットに配備された社内システムではこの仕組みを利用することが多い．攻撃者にとって，攻撃を成功させるために必要なことは署名の有無ではなくアクセス制御を緩められるかどうかであり，これらの署名なしプログラムも再構成攻撃の対象となりうる．よって，これらへの対策も必要である．

以上のことから，アプレットや JWS アプリケーションへの再構成攻撃を防ぐため，我々は本ツールを改良してこれらのプログラムにも適用できるようにした．本論文では再構成攻撃と対策の詳細について述べる．

本論文の構成は次のとおりである．まず，2 章で再構成攻撃の脅威をアプレットの場合を例に示し，3 章で本ツールがこの攻撃をどのように防ぐのかを説明する．そして，4 章において，本ツールの課題であった，JWS アプリケーションと署名なしプログラムへの再構成攻撃の詳細を説明し，5 章で我々が行った本ツールの改良について説明する．最後に，6 章で関連研究について説明し，7 章でまとめを述べる．

2. 再構成攻撃

まず再構成攻撃とはどのような攻撃なのかについて，アプレットの場合を例に説明する[5]．再構成攻撃とは，信頼されたプログラムの部品を攻撃者の部品と組み合わせることで，信頼されたプログラムの機能を悪用する攻撃である．アプレットの場合，部品は JAR ファイルであり，組み合わせ方法を決定するのは HTML ページのアプレットタグである．HTML ページはだれでも用意でき，攻撃者は攻撃対象の署名付き JAR ファイルなどを入手し，どこかに用意した Web サイト上で自らが作成した JAR ファイルと組み合わせることにより，様々な攻撃を仕掛けることができる．

再構成攻撃には典型的な攻撃手法として，特権コードの悪用，クラス置き換えという 2 つの手法がある．前者は，攻撃者のコードがアプレットとして動作し，攻撃対象のコードをライブラリとして悪用する手法である．一方，後者は，アプレットとして動作させるの

は攻撃対象のコードであるが，一部のクラスやリソースを攻撃者のものと置き換えることにより，データフローなどを制御して悪用する手法である．2.1 節で特権コードの悪用について説明し，2.2 節でクラス置き換えについて説明する．

2.1 特権コードの悪用

1 つ目は攻撃対象の JAR ファイルをライブラリとして利用し，特権コードを悪用する手法である．攻撃対象の JAR ファイルをライブラリとして利用することは容易である．たとえば，次のような署名付きアプレットがあったとする．

```
<applet code="app.PurchaseApplet"
 archive="purchase.jar" ...
```

ここで，攻撃者は攻撃対象の JAR ファイル `purchase.jar` と自らの JAR ファイル `evil.jar` を次のように組み合わせれば，信頼されたクラスのメソッドやフィールドにアクセスできる（`EvilApplet` は攻撃者のクラス）．

```
<applet code="EvilApplet"
 archive="evil.jar,purchase.jar" ...
```

ただし，Java にはスタック検査による呼び出し元のチェック機能があり，攻撃者のクラスから信頼されたクラスのメソッドを単純に呼び出しただけでは，このチェック機能によって拒否されてしまう[5]．

そこで攻撃者は特権コードを狙う．特権コードとは前述のチェック機能をバイパスするための仕組みであり，信頼されたプログラムの機能を信頼されていないプログラムにも限定的に利用させるために利用される．以下に例を示す．

```
public String loadConf() {
 Reader r = (Reader) // 特権コード開始
  AccessController.doPrivileged(
    new PrivilegedAction() {
     public Object run() {
       try {
       // ファイルダイアログを開く
        :
       // 選択されたファイルを開く
       return new FileReader(name);
      } catch (Exception e) {
       e.printStackTrace();
       return null;
 }}});       // 特権コード終了
 // ファイルを読み込み文字列として返す
  :
}
```

`loadConf()` はファイルダイアログをエンドユーザに表示し，選択されたファイルを読み込んで文字列として返すメソッドである．ファイル読み込みという危険な操作を実行する箇所が特権コードになっている．この例ではファイル名はユーザに選択されるが，もしファイル名がメソッドの引数で与えられていた場合は，このメソッドを呼び出せばだれでも任意のファイルが読み込めることになる．

よって，特権コードを利用する際には，攻撃者に呼び出されることを考慮して，細心の注意を払って実装しなければならない．しかし，前述のような脆弱性のある特権コードを実装してしまう場合がしばしばみられる．攻撃者はそうした脆弱な特権コードを探し出して悪用するのである．

2.2 クラス置き換え

2 つ目は攻撃対象の一部のクラスを攻撃者のクラスに置き換える手法である．前述の特権コードを悪用する方法では，対象のプログラムが特権コードを実装している必要があるが，多くのプログラムでは特権コードを実装していないので，悪用の可能性はそれほど高くはない．しかし，ここで説明する手法は特権コードを必要とせず，より成功する可能性が高い．

この手法はアプレットのクラスのロード方法を利用する．アプレットでは，複数の JAR ファイルに同じパスのファイルが存在する場合，`archive` 属性で先に指定された JAR ファイルを優先する．これを利用して，攻撃者は攻撃対象の JAR ファイルに含まれるファイルと同じパスのファイルを作成して，JAR ファイルに含めて `archive` 属性で先に指定することで，ロードされるファイルを置き換えることができる．

ここで，前述の `purchase.jar` が次のような構成だったとする．

```
purchase.jar
  +- app/PurchaseApplet.class
  +- util/MyFileDialog.class
```

ここで攻撃者は `MyFileDialog.class` の置き換え用のクラスファイルを作成し，`evil.jar` に含める．

```
evil.jar
  +- util/MyFileDialog.class
```

そして，次のように `evil.jar` を `purchase.jar` より先に指定する．

```
<applet code="app.PurchaseApplet"
 archive="evil.jar,purchase.jar" ...
```

これで `MyFileDialog.class` は攻撃者のものに置き換わって実行される．ただし，置き換わったクラスは署名付きクラスとしては扱われないので，そのクラ

スから危険な動作を行うメソッド（Runtime.exec() など）を直接呼び出すことはスタック検査によって禁止される．そこで，攻撃者はユーティリティクラスなどの，危険な動作を行うメソッドを直接呼び出さないが，様々な動作に影響を与えるようなクラスを置き換える．そうした構成のプログラムは多いと考えられ，攻撃が成功する可能性は高いといえる．

ただし，Java には *same-package-same-signer* とよばれる保護機能があり，あるパッケージのクラスはすべて同一の署名者に署名されている必要がある[5]．よって，対象の署名付きアプレットが単一のパッケージで構成されている場合は置き換えできない．しかし，複数のパッケージで構成されている場合，すべてのパッケージが同一の署名者である必要はないので，ユーティリティクラスが含まれるパッケージだけ置き換えることが可能である．また，この保護機能はクラスのみが対象でリソースは対象外である．よって，この保護機能の下でも攻撃は十分に成功するといえる．

この攻撃手法は幅が広く，開発者はいろいろな可能性を考慮してコーディングしなければならないが，セキュリティの専門知識のない一般の開発者にとってそれは非常に難しい．この攻撃手法の存在が，我々が対策ツールを作成するに至った主な動機である．

3. Cozilet

署名付きアプレットに対する再構成攻撃，特にクラス置き換えを防ぐために，我々は過去に Cozilet と称する対策ツールを作成した[5]．**図 1** に概要を示す．本ツールは Java Plug-in で動作するアプレットを対象とし，対象アプレットをカプセル化することで再構成攻撃を防ぐ．また，対象アプレットや Java Plug-in 実行環境に対して透過的であり，適用が容易であることが特徴である．3.1 節でカプセル化の仕組みを，3.2 節で透過性について説明する．

3.1 カプセル化

再構成攻撃はアプレットを構成する部品の結び付きが弱いことを利用して，これらをばらばらに悪用する．そこで，それらの結び付きを強固にして再構成を防ぐというのが我々の考えである．これをカプセル化とよぶ．カプセル化はクラスローダによる隔離とダウンロード元チェックという 2 つの仕組みにより実現され，前者は JAR ファイルを，後者は JAR ファイルと HTML ページをカプセル化する．

(1) クラスローダによる隔離

まず，JAR ファイルのカプセル化のために，複数の JAR ファイルをまとめて格納できるような特殊な形式の JAR ファイルを導入し，専用のクラスローダでロードするようにした．この特殊形式 JAR ファイルの内部に含まれる JAR ファイルは専用のクラスローダでなければ認識できない．特殊形式 JAR ファイルは署名されており，専用のクラスローダはその署名を検証し，改ざんや置き換えがないことを確認できた場合に限り，クラスやリソースをロードする．また，クラスローダを別にすることにより，他のアプレットと名前空間が分離されるので，実行時の攻撃者からのアクセスを防ぐことができる．

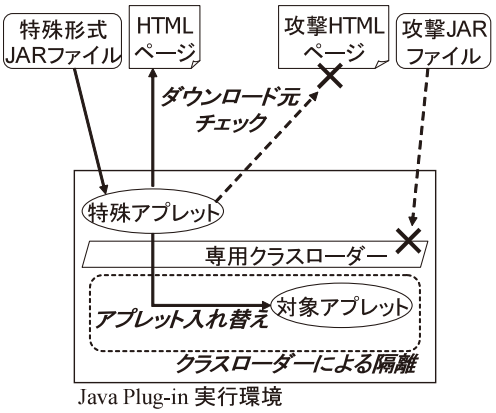

図 1 Cozilet による保護
Fig. 1 Protection by Cozilet.

(2) ダウンロード元チェック

次に，JAR ファイルとアプレットタグを含む HTML ページのカプセル化も必要である．HTML ページにはアプレットタグだけでなく，アプレットの初期パラメータやアプレットと連携する JavaScript などが記述される場合がある．しかし，初期パラメータにより不正な値を入力されたり，悪意のある JavaScript によりアプレットのメソッドが悪用されたりする恐れがある．HTML ページはアプレットを構成する重要な部品の 1 つであり，攻撃者が用意した HTML ページから起動されることを防ぐ必要がある．

そこで，アプレットの起動を許可する HTML ページの URL のリストをあらかじめアプレットに埋め込んでおき，実行時に実際に起動された HTML ページの URL と一致するかどうかチェックするようにした．アプレットを起動した HTML ページの URL は getDocumentBase() というメソッドにより取得できる[12]．

上記の仕組みにより JAR ファイルと HTML ページがカプセル化され，再構成攻撃を防ぐことができる．

3.2 透 過 性

カプセル化は透過的であることが望ましい．すなわ

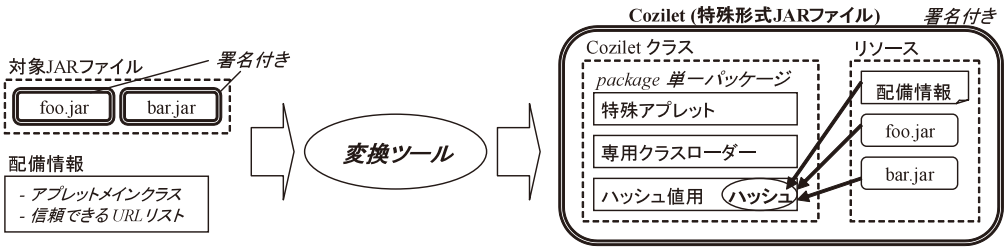

図 2 Cozilet による変換
Fig. 2 Translation by Cozilet.

ち，保護対象のアプレットに改変を強いず，かつ，標準の Java Plug-in で動作することが求められる．ここでは透過性を実現する 2 つの仕組みとして，(1) 項でアプレット入れ替え，(2) 項でツールによる自動変換について説明する．ただし，透過性の実現のために生じるデメリットもある．その対策は (3) 項で説明する．

(1) アプレット入れ替え

透過性の実現のため，本ツールではアプレット入れ替えという仕組みを導入している．これは，まずカプセル化を強制するための特殊なアプレットが動作し，その後で保護対象のアプレットに入れ替わるというものである．

この特殊アプレットは Java Plug-in からみると通常の署名付きアプレットにみえるので，アプレットとして実行される．そして，特殊アプレットは実行されるとすぐに専用のクラスローダを生成し，特殊形式 JAR ファイルに含まれる対象アプレットの JAR ファイルに改ざんや置き換えがないことを確認したのち，対象アプレットに制御を移す．

Java Plug-in において，アプレットは GUI 部品として扱われており，アプレットごとに用意される親パネルに登録されている．この親パネルには最初は特殊アプレットが登録されているが，自らを登録解除して代わりに対象アプレットを登録することで，GUI 機能を簡単に入れ替えることができる．

ただし，アプレットには GUI 機能のほかに，ライフサイクルの管理や，JavaScript との連携などの機能があり，これらも入れ替える必要がある．この入れ替えは委譲で実現する．具体的には，特殊アプレットにおける関連するすべてのメソッドの呼び出しを対象アプレットに委譲している．

これらの仕組みにより，Java Plug-in の実行環境を変更することなく，透過的にカプセル化を実現することができる．

(2) ツールによる自動変換

さらに，カプセル化を容易にするために，対象アプレットを構成する JAR ファイル群を前述の特殊アプレットに自動変換するコマンドラインツールを提供している（**図 2**）．これを変換ツールとよぶ．

変換ツールは対象アプレットの JAR ファイル群のほかに配備情報を入力とする．配備情報には，アプレットを構成する JAR ファイルの情報，アプレットとして実行されるクラスの名前（`code` にあたる），そしてダウンロード元チェックのための信頼できる URL のリストなどを記述する．変換ツールはこれらの情報をもとに特殊形式の JAR ファイルを生成する．これらの配備情報は特殊アプレットが実行される際に参照される．

対象アプレットの JAR ファイル群は，特殊形式 JAR ファイルの中にリソースとして保持されており，特殊アプレットの実行時に専用クラスローダによりロードされる．変換の際に元の署名は削除されるので，JAR ファイル群が取り出されても悪用される恐れはない．これらの完全性は特殊形式 JAR ファイルへの署名で保証される．

開発者からみた場合はこの変換ツールだけを意識すればよく，配備情報を与えて変換し，すでに配備済みのアプレットを変換されたものと置き換えるだけでよい．

(3) 透過性にともなうデメリットへの対処

しかし，本ツールの透過性は，自らが署名付きアプレットとして悪用されうるというデメリットをもたらす．そこで，次の 3 つの防衛策により対処している．

まず，特殊アプレットや専用クラスローダなど，本ツールを構成するクラスはすべて単一のパッケージとして定義し，全体を署名することで *same-package-same-signer* の仕組みにより保護されるようにしている．public である必要がないクラス・メソッド・フィールドはすべてデフォルトアクセスか，最低限必要なアクセス範囲として定義されており，攻撃されるエントリポイントをできるだけ少なくしている．public なメソッドについても，基本的に Java Plug-in 実行環境から呼び出せれば十分なので，スタック検査により `AllPermission` [13] を持つコード（すなわちシステムのコード）以外からの呼び出しを防いでいる．

また，クラスの置き換えは *same-package-same-signer* で保護されるものの，リソースの置き換えは自動的には保護されない．特殊形式 JAR ファイルでは対象アプレットの JAR ファイルをリソースとして保持しており，置き換えを防ぐ必要がある．そこで，対象 JAR ファイルのハッシュ値をハッシュ値格納用クラスの定数フィールド値として格納することにより，実行時に置き換えを検出している．

さらに，アプレットの直列化機能を悪用し，アプレットのフィールド変数の値を改ざんしたり盗み出したりされる恐れがある．そこで，直列化関連のメソッドをすべてオーバライドしてつねに例外を発生させることで，直列化を悪用した攻撃を防いでいる．

これらの防衛策により，本ツール自身を悪用することは困難である．

以上，これらのカプセル化と透過性の仕組みにより，本ツールを利用することで再構成攻撃を効果的に，そして容易に防ぐことができる．

4. 課　　題

Cozilet を適用すれば再構成攻撃を容易に防ぐことができるが，対象となるのはアプレットのみで，さらに署名付きのものだけという制約があった．しかし，最初に述べたように，Java Web Start（JWS）[10] の利用も一般的になりつつあり，JWS アプリケーションに対してもアプレットと同様の再構成攻撃が可能であることを，我々は過去に示している[6]．また，署名せずにポリシファイルによりアクセス制御を緩める方式があり，署名付きでなくても攻撃される恐れがある．ここではこれらの 2 つの課題について，4.1 節で JWS 上での再構成攻撃，4.2 節で署名なしプログラムへの再構成攻撃について説明し，対策の必要性を示す．

4.1 JWS 上での再構成攻撃

ここではまず JWS アプリケーションの構成について (1)，(2) 項で説明し，その後，JWS 上での再構成攻撃の仕組みについて (3) 項で説明する．ただし，JWS には再構成攻撃の防止に役立つ仕組みとして JNLP ファイルへの署名機能がある．しかし，我々はこの機能は十分な対策とならないことを示した．JNLP ファイルへの署名機能については (4) 項で説明する．

(1) JWS アプリケーションの構成

JWS アプリケーションは主に JAR ファイルによって構成される．JAR ファイルにはクラスファイルや，画像や音声などのリソースファイルを含めることができる．JAR ファイルの構成や最初に実行すべきクラスなどのアプリケーションの実行に必要な情報は，JNLP ファイルというファイルに記述される[3]．

JNLP ファイルは XML 形式のファイルであり，JAR ファイルなどとともに Web サイトに配備される．JNLP ファイルは大きく分けると，名前や作成者などの情報（`<information>`），セキュリティの情報（`<security>`），必要な JAR ファイルなどの情報（`<resources>`），そして JWS アプリケーションの性質の情報（`<application-desc>` など）で構成される．以下に例を示す．

```
----- purchase.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<information>
  <title>MyApp</title>
  <vendor>Hisashi</vendor>
</information>
<security><all-permissions/></security>
<resources>
  <j2se version="1.5"/>
  <jar href="purchase.jar"/>
</resources>
<application-desc
 main-class="app.PurchaseApp"/>
</jnlp>
```

上記の例では，これがアプリケーション型で，`MyApp` という名前を持ち，`Hisashi` によって作成され，完全なアクセス権を必要とし，J2SE のバージョン 1.5 を対象とし，`purchase.jar` という単一の JAR ファイルによって構成され，`app.PurchaseApp` というクラスが最初に実行すべきメインクラスであることを示している．以降では簡単のため `<information>` と `<j2se>` は省略する．

(2) アプリケーションとエクステンション

JWS アプリケーションには大きく分けてアプリケーション型とエクステンション型の 2 種類の性質がある．性質は排他的であり，1 つの JWS アプリケーションは 1 つの性質しか持つことができない．

アプリケーション型は単体で実行可能である．JNLP ファイルで `<application-desc>` を記述するとデスクトップアプリケーションとして扱われる．また，`<applet-desc>` と記述するとアプレットとして実行される．ただし，JWS でのアプレットはいわゆる `appletviewer` [15] で動作させたのと同様であり，Java Plug-in のように Web ブラウザと密に連携することはできない．

一方，エクステンション型は基本的に単体では実行されず，アプリケーション型からの利用を意図したものである．JNLP ファイルで `<component-desc>` を記述すると単純なライブラリとして扱われ，`<installer-desc>` を記述するとインストーラ付き

のライブラリとして扱われる．

アプリケーションからエクステンションを参照する場合には，<resources> に <extension> を記述する．エクステンションの署名状態はアプリケーションと独立して扱われるので，たとえばアプリケーションには署名するがエクステンションには署名しないといったことが可能である．

(3) JWS 上での再構成攻撃

JWS の場合でも，攻撃手法はアプレットの場合と基本的には変わらない．最も単純な方法は，アプレットで HTML ページを用意するのと同様に，攻撃対象の JAR ファイルと攻撃者の JAR ファイルを組み合せるための JNLP ファイルを用意することである．以下は 2.1 節で述べたような特権コードを悪用する例である．

```
----- evil.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<security><all-permissions/></security>
<resources>
  <jar href="evil.jar"/>
  <jar href="purchase.jar"/>
</resources>
<applicatoin-desc main-class="EvilApp"/>
</jnlp>
```

しかし，アプレットと異なり，JWS で署名付きアプリケーションとして扱われるためには，JNLP ファイルから <jar> で参照された JAR ファイルがすべて同一の署名者に署名されている必要がある．上記の例では署名付きではない攻撃者の JAR ファイル evil.jar が混在しており，署名付きとして扱われないため攻撃は成功しない．

そこで，攻撃者は <extension> を利用する．すでに述べたように，エクステンションとして参照する場合，署名状態は独立して扱われる．以下に例を示す．

```
----- evil.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<resources>
  <jar href="evil.jar"/>
  <extension href="purchase.jnlp"/>
</resources>
<application-desc main-class="EvilApp"/>
</jnlp>

----- purchase.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<security><all-permissions/></security>
<resources>
  <jar href="purchase.jar"/>
</resources>
<component-desc/>
</jnlp>
```

evil.jnlp，purchase.jnlp のいずれも攻撃者が用意したものである．evil.jnlp はアプリケーション型として，purchase.jnlp はエクステンション型として定義されており，evil.jnlp はエクステンションとして purchase.jnlp を参照している．purchase.jnlp は署名付きの JAR ファイルで構成されており，署名付きとして扱われるので攻撃者に悪用の機会がある．

また，2.2 節で述べたクラス置き換え攻撃を行うには次のようにすればよい．

```
----- purchase.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<security><all-permissions/></security>
<resources>
  <extension href="evil.jnlp"/>
  <jar href="purchase.jar"/>
</resources>
<application-desc
 main-class="app.PurchaseApp"/>
</jnlp>

----- evil.jnlp -----
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="...">
<resources>
  <jar href="evil.jar"/>
</resources>
<component-desc/>
</jnlp>
```

先の例とは逆に，purchase.jnlp がアプリケーション型で，evil.jnlp がエクステンション型になっている．purchase.jnlp から evil.jnlp をエクステンションとして参照しているが，purchase.jar よりも先に参照されていることに注意してほしい．こうすることで evil.jar が purchase.jar よりも優先され，クラスやリソースを置き換えることができる．

(4) JNLP ファイルへの署名

アプレットと異なり，JWS では JNLP ファイルを署名することで改ざんを検出できる[3]．再構成攻撃の大きな原因の 1 つは，攻撃者が HTML ページや JNLP ファイルを自由に作成できることにあるので，JNLP ファイルへの署名は効果的な対策である．しかし，以下に述べる理由から十分とはいえない．

まず，JNLP ファイルへの署名は必須ではないことである．JNLP ファイルに署名してもしなくても，署名付き JWS アプリケーションの動作には影響を与えない．よって，多くの場合は JNLP ファイルに署名せずに運用されている可能性が高い．

次に，JNLP ファイルは保護されるものの，参照されている署名付き JAR ファイルは依然として悪用されうる点があげられる．署名付き JAR ファイルは部

品として他の JWS アプリケーションに流用できるし，アプレットに使われる恐れもある．

さらに，JWS 実行環境による JNLP ファイルの署名検証アルゴリズムに不備があり，場合によっては JNLP ファイルを偽造されることがある．JWS では JNLP ファイルへの署名が存在するかどうかを，メインクラスを含む JAR ファイル，もしくは JNLP ファイルで最初に指定されている JAR ファイルに，`JNLP-INF/APPLICATION.JNLP` というエントリが存在するかどうかで判断する．そこで，このエントリを含む JAR ファイルの指定順序を変えれば，JNLP ファイルへの署名がないと JWS 実行環境に思わせることができる．我々の検証では，対象のアプリケーションが複数の JAR ファイルで構成されていれば，エクステンション型に再定義することで特権コードの悪用が可能であることが分かっている[6]．ただし，クラス置き換え攻撃はできない．

以上のことより，アプレットの場合と同様に JWS アプリケーションにも再構成攻撃の危険がある．JNLP ファイルへの署名は対策としては不十分であり，別の対策が必要である．

4.2 署名なしプログラムへの再構成攻撃

これまでの説明はすべて JAR ファイルに署名されていることを前提としていた．しかし，最初に述べたように，ポリシファイルを設定することにより，署名しなくても，たとえば特定の URL に配備された JAR ファイルなどに対するアクセス制御を緩めることができる[11]．攻撃者からみると，アクセス制御を緩められるかどうかが重要であり，署名がなくても悪用の対象になることには変わりはない．

ここで，`https://intra.example.com/` というイントラネット上のサイトに配備された次のようなアプレットがあるとする．

```
-- https://intra.example.com/に配備 --
<applet code="intra.MyApplet"
 archive="intra.jar"></applet>
```

このアプレットを構成する JAR ファイルには署名はないが，次のようにポリシファイルでアクセス権を与えられている．

```
grant codebase
      "https://intra.example.com/-" {
 permission java.io.FilePermission
              "<<ALL FILES>>", "read";
}
```

ここでは，イントラネット上のサイトは安全に運用されており，攻撃者が不正な JAR ファイルを配備できないものとする．よって，上記のアクセス権が攻撃者の JAR ファイルに与えられることはない．しかし，`intra.jar` が配備された URL が分かっていれば，次のように絶対 URL を指定することで再構成攻撃を仕掛けることができる．

```
-- 攻撃者のサイトに配備 --
<applet code="AttackApplet"
 archive="evil.jar,
  https://intra.example.com/intra.jar">
</applet>  // 絶対URLを指定
```

よって，署名なしプログラムでも再構成攻撃は可能である．また，絶対 URL には file URL を指定することも可能であり，ローカルに配備された JAR ファイルでさえも攻撃対象となりうる．署名せずにポリシファイルを設定する方式はイントラネットだけで動作する社内システムではしばしばみられ，再構成攻撃への対策が必要である．

以上のことから，アプレットだけではなく JWS アプリケーションや署名なしプログラムに対しても再構成攻撃が可能である．我々はこれらのプログラムにも Cozilet を適用できるように改良することで対処した．次章ではその詳細について説明する．

5. Cozilet の改良

ここでは，Cozilet を JWS アプリケーションや署名なしプログラムに対応させるために我々が行った改良について述べる．まず JWS への対応方法について 5.1 節で説明し，次に署名なしプログラムへの対応方法について 5.2 節で説明する．

5.1 JWS への対応

JWS は Java ベースなのでアプレットと仕組みがよく似ている．どちらもプログラムの実体は JAR ファイルであり，それらの組み合わせ方法を，アプレットではアプレットタグ，JWS では JNLP ファイルで，それぞれ指定する．よって，カプセル化や透過性といった本ツールの基本的な仕組みはほとんどそのまま適用できる．

しかし，アプレットと JWS では実行時の挙動に違いがあり，JWS 特有の仕組みに対応する必要がある．まず，JWS ではアプレットとは最初に実行されるクラスの種類やメソッドの呼び出し順序などが異なるため，JWS アプリケーションに対応させるためにはアプレット入れ替えの仕組みを変更する必要がある．これを (1) 項で説明する．

また，4.1 節で述べたように，JWS ではアプリケーションだけではなくアプレットも動作させることがで

きる．ここではこれを JWS アプレットとよぶ．この JWS アプレットへの対応は (2) 項で説明する．

さらに，JWS では実行に必要な情報が JNLP ファイルで完結しており，起動元のサイトが意識されないため，ダウンロード元チェックを行うことができないという問題がある[3]．この問題の解決策について (3) 項で説明する．

(1) JWS アプリケーションへの対応

JWS アプリケーションはアプレットに比べて実行環境との結び付きが強くないので，対応は比較的容易である．アプレットの場合，特殊アプレットは次のように動作する[5]．

i) まず，Java Plug-in 実行環境が特殊アプレットのクラスをロードし，インスタンスを生成する．特殊アプレットは静的初期化子において配備情報が格納されたリソースをロードする．その際にリソースの改ざんや置き換えがないかどうかを 3.2 節の (3) 項で述べたハッシュ値により検証する．
ii) 次に，Java Plug-in 実行環境は特殊アプレットの `init()` を呼び出す．ここではまず，配備情報に含まれる信頼できる URL のリストを取得し，ダウンロード元チェックを行う．
iii) チェックに問題なければ，専用クラスローダのインスタンスを生成する．専用クラスローダはリソースとして格納された対象アプレットの JAR ファイルをロードする．この際，i) と同様に改ざんや置き換えがないかどうか検証する．
iv) インスタンス生成に成功したら，対象アプレットのアプレットクラスを専用クラスローダによりロードし，アプレット入れ替えを行い，対象アプレットの `init()` を呼び出す．
v) その後，特殊アプレットは Java Plug-in 実行環境からの呼び出しを対象アプレットに委譲し続ける．

一方，JWS アプリケーションの場合，JWS 実行環境が行う処理は対象アプリケーションのメインクラスの `main()` を呼び出すことだけである[3]．よって，入れ替えは容易であり，次のようにすればよい．

i) アプレットの場合の特殊アプレットにあたる，JWS 実行環境によって最初に実行されるクラスを特殊メインクラスとよぶことにする．まず，JWS 実行環境が特殊メインクラスの `main()` を呼び出す．以降のすべての処理は `main()` 内で行われる．
ii) 次に，アプレットの場合の iii) と同様に，専用クラスローダのインスタンスを生成する．
iii) 成功したら，対象アプリケーションのメインクラスをロードし，`main()` を呼び出す．

ただし，最初に述べたように JWS ではダウンロード元チェックが行えないことに注意してほしい．その対処については (3) 項で述べる．

(2) JWS アプレットへの対応

4.1 節の (2) 項で述べたように，JWS では JNLP ファイルにおいて `<application-desc>` の代わりに `<applet-desc>` を記述することにより，アプレットを動作させることができる．アプレットとしての動作は基本的には Java Plug-in の場合と同様なので，JWS 特有の処理は必要ない．しかし，JWS におけるアプレットには `appletviewer` と同程度の機能しか提供されず，Java Plug-in で動作させたときのように Web ブラウザと密に連携することはできない．よって，JavaScript との連携処理などを提供する必要はない．

(3) ダウンロード元チェックへの対応

JWS では，JNLP ファイル，そしてそこから参照される JAR ファイルだけで処理は完結しており，アプレットのようにアプレットタグを含む HTML ページの JavaScript などと連携することはない．よって，Cozilet を利用しているならば，JNLP ファイルが改ざんされたとしても，アプレットに比べると脅威は小さいといえる．

しかし，JNLP ファイルにはアプリケーションやアプレットの起動時に渡す初期パラメータが記述される場合がある[3]．これらの値はプログラムの動作に大きな影響を与えることが多く，攻撃者が自由に設定できると危険である．よって，JNLP ファイルの改ざんを防ぐ必要がある．4.1 節の (4) 項で述べたように，JWS には JNLP ファイルへの署名機能が用意されている．そこで，JNLP ファイルに署名すればよい．JWS の署名検証アルゴリズムの不備が原因で，複数 JAR ファイルで構成されているプログラムは署名を無効化される恐れがあるが[6]，Cozilet の変換ツールはつねに単一の JAR ファイルを作成するので問題はない．

以上の仕組みにより，署名付き JWS アプリケーション，JWS アプレットのいずれについても，改良した Cozilet により再構成攻撃を防ぐことができる．

5.2 署名なしプログラムへの対応

ここではもう 1 つの課題である，署名なしプログラムへの対応方法について説明する．Cozilet はその仕組み上，特殊形式 JAR ファイルの完全性が保証されている必要がある．よって署名がない場合はこの完全性の保証が問題となるが，署名なしプログラムが運用されるような環境では，プログラム自体の改ざんは防止されていると考えてよい．なぜなら，署名しない場

合は JAR ファイルが配備された URL だけが信頼性の根拠であり，JAR ファイルが改ざんされるようなことがあれば信頼性が崩れてしまうからである．したがって，本ツールは 4.2 節で説明したような再構成攻撃の防止だけを考慮すればよく，署名なしプログラムの完全性は運用で保証されているという前提を置くことができる．ゆえに，本ツールの仕組みの多くは署名なしプログラムに適用できる．

しかし，署名がないことにより安全性が保てなくなる部分がある．それらは個別に対応する必要があり，パッケージの保護，JNLP ファイルの完全性，適切なアクセス権の付与が該当する．

まず，本ツールは自分自身を攻撃者から保護するために *same-package-same-signer* を利用してパッケージを保護しているが，この機能は署名がないと働かないという問題がある．この対処について (1) 項で説明する．

次に，JNLP ファイルの完全性が問題になる．JNLP ファイルは初期パラメータなどでプログラムの挙動に影響を及ぼすので，攻撃者による改ざんを防がなければならない．この対処について (2) 項で説明する．

最後に，適切なアクセス権を付与する必要がある．本ツールではつねに `AllPermission`（完全なアクセス権）を前提にしていたが，ポリシファイルでは柔軟なアクセス権の設定が求められる．この対処について (3) 項で説明する．

(1) パッケージの保護

3.2 節の (3) 項で述べたように，本ツールでは自分自身を攻撃者から守るために，パッケージを保護しデフォルトアクセスのクラス・メソッド・フィールドへのアクセスを防ぐ必要がある．本ツールでは *same-package-same-signer* を利用してパッケージを保護していた[5]．この仕組みは署名とパッケージを結び付け，署名が偽造されない限り，パッケージに攻撃者がアクセスすることを防ぐことができる．しかし，この仕組みは署名がないと有効にならないという問題があり，署名なしの場合は他の手段でパッケージを保護しなければならない．

そこで，本ツールでは JAR ファイルの `Sealed` 属性という仕組みを利用する[16]．`Sealed` 属性は JAR ファイルのマニフェストファイルに追加される属性であり，この属性を追加された JAR ファイル内のすべてのパッケージは，同一 JAR ファイル内のクラス以外からはアクセスできなくなる．よって，本ツールにより生成される特殊形式 JAR ファイルにこの属性を追加することで，パッケージを保護することができる．以下にマニフェストファイルの例を示す．

```
Manifest-Version: 1.0
Sealed: true
```

`Sealed` 属性は *same-package-same-signer* と同様にパッケージの保護を目的とする仕組みである．しかし，JAR ファイルの仕様では署名付き JAR ファイルの署名を損なわずに攻撃者がファイルを追加できるため，`Sealed` 属性による同一 JAR ファイルという制限が回避されるという問題があった[7]．だが，本節の最初で述べたように，ここでは JAR ファイルは改ざんされないという前提を置くことができるため，`Sealed` 属性によるパッケージの保護は有効である．

(2) JNLP ファイルの完全性

5.1 節の (3) 項で述べたように，本ツールが JWS で動作する場合，JNLP ファイルから渡される初期パラメータの安全性を，ダウンロード元チェック以外の手段で保証する必要がある．JNLP ファイルへの署名で実現することが最も効果的だが，署名がない場合は他の手段を考えなければならない．

そこで，署名がない場合は，JWS 実行環境に代わり，本ツールが JNLP ファイルの改ざんを検証することにした．検証の方法は JWS 実行環境が行う方法と同様で，JNLP ファイルを `JNLP-INF/APPLICATION.JNLP` という名前のエントリとして特殊形式 JAR ファイルに格納し，実行時に JWS 実行環境によって実際にロードされたファイルと比較することで実現される．すでに述べたように，JAR ファイルの完全性は運用で保証されるという前提を置けるので，署名がない場合でも JNLP ファイルの改ざんを防ぐことができる．

(3) 適切なアクセス権の付与

署名付きプログラムの場合，与えられるアクセス権は基本的に `AllPermission`，すなわち完全なアクセス権である．しかし，ポリシファイルで設定する場合，プログラムが動作するのに十分な適切なアクセス権を指定することが望ましい．従来，本ツールでは署名付きプログラムを前提としていたため，対象アプレットにはつねに `AllPermission` が与えられていた．

そこで，ポリシファイルの設定情報を配備情報ファイルに含めることにより，対象アプレットには指定されたアクセス権だけが与えられるようにした．これにより適切なアクセス権が付与される．ただし，本ツールの構成部品である特殊アプレットや専用クラスローダなどのクラスについては，クラスローダの生成などの危険と見なされる処理を行うので，依然として `AllPermission` が必要である．これらはシステムク

ラスと同じような役割といえ，悪用を防ぐように十分に安全性を考慮して設計されているので許容できると我々は考える．

以上，これまで述べたように，本ツールに改良を施すことにより，署名付きアプレットだけではなく，JWS アプリケーション（および JWS アプレット），そして署名せずにポリシファイルを設定することでアクセス権を緩めるようなプログラムにも本ツールを適用できるようになった．本ツールを利用することで，再構成攻撃の脅威を防ぎ，アプレットや JWS アプリケーションを安全に利用することができる．

6. 関連研究

我々のアプローチは再構成攻撃を効果的に防ぐだけでなく，既存のプログラムや Java Plug-in および JWS の実行環境に対して透過的でもあり，我々の知る限り同様の研究は知られていない．

他の再構成攻撃への対策としては，開発者に注意を促すためのいくつかのガイドラインが知られている．文献 9) では署名付きアプレットの開発者が注意すべき 12 個のアンチパターンを提唱している．Java Plug-in や JWS の提供元である Sun 自身からも，文献 2) および 14) が出版および公開されている．また，これらのガイドラインに基づいた監査ツールも公開されている[1),8),17)]．しかしながら，再構成攻撃は巧妙な攻撃手法であり，セキュリティの専門知識を持たない一般の開発者にとって，攻撃手法を理解し対策を施すことは非常に難しいというのが我々の主張である．

ほかに，再構成攻撃への対策を目的とするものではないが，我々が注目している技術として，JSR 121 Java Application Isolation API がある[4)]．これは Java VM 上で動作するアプリケーションについて，ヒープメモリ，native メモリ，他の VM 上のリソースなどをアプリケーションごとに分離することを目的としており，一般的な OS がプロセスに対して行っているような処理を Java で実現するものである．Cozilet の重要な保護機能であるクラスローダによる分離では，実装が容易である反面，システムクラスなどのクラスローダ間で共有されるクラスを完全に分離できないという問題がある．Isolation API を使えばこの問題を解決できる可能性がある．現在，Isolation API は仕様が完成したところであり，Java VM 実装へ導入されたならば Cozilet での活用を検討したい．

7. まとめ

本論文では，JWS 上で動作するプログラム，および，署名せずにポリシファイルを設定することでアクセス権を緩めるようなプログラムへの再構成攻撃の脅威を示し，その対策として，我々が過去に作成した対策ツール Cozilet の改良を行った．再構成攻撃はアプレットや JWS アプリケーションに対する大きな脅威であり，開発者の注意に頼る方法では防ぐことは難しい．本ツールは対象のプログラムを再構成できない形式にカプセル化し，かつ，既存のプログラムや Java 実行環境に対して透過的であるので，再構成攻撃を効果的に防ぐことができる．従来，本ツールが適用できるのは署名付きアプレットのみであったが，本論文で示したように JWS や署名のないプログラムも再構成攻撃の対象であり，署名付きアプレットと同様の対策が必要である．今回，本ツールを改良したことにより，JWS や署名のないプログラムにも適用できるようになった．これにより JWS や署名のないプログラムの安全性を向上させることができる．また，アプレットと JWS は部品に互換性があり，アプレットの安全性を高めても，JWS 上で悪用される恐れがあった．今回，JWS にも対応したことにより，アプレットの安全性の向上にも効果があるといえる．

本ツールはすでに実システムで稼動する署名付きアプレットに適用されており，システムの安全性向上に効果をあげている．今後は，これまで適用できなかった，JWS アプリケーションや署名のないプログラムにも適用していきたい．

参考文献

1) Curphey, M.: codespy. http://www.securityfocus.com/archive/107/349071/2004-01-02/2004-01-08/0
2) Gong, L., Ellison, G. and Dageforde, M.: *Inside Java 2 Platform Security: Architecture, API Design and Implementation*, 2/E, Addison-Wesley (2003).
3) Java Community Process: JSR 56: Java Network Launching Protocol and API. http://jcp.org/en/jsr/detail?id=56
4) Java Community Process: JSR 121: Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121
5) Kojima, H., Morikawa, I., Nakayama, Y. and Yamaoka, Y.: Cozilet: Transparent Encapsulation to Prevent Abuse of Trusted Applets, *Proc. 20th ACSAC*, pp.146–155 (2004).
6) 兒島 尚，鳥居 悟：信頼された Java Web Start アプリケーションの悪用に対する一考察，研究報告「コンピュータセキュリティ」，No.2006-CSEC-033 (2006).

7) 児島 尚，丸山 宏：Java パッケージシステムのセキュリティ，コンピュータセキュリティシンポジウム'98 論文集 (1998).
8) 児島 尚，中山裕子，川島 悟，藤川亮子：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，研究報告「コンピュータセキュリティ」，No.2002-CSEC-020 (2003).
9) McGraw, G. and Felten, E.W.: *Securing Java: Getting Down to Business with Mobile Code*, Wiley (1999).
10) Sun Microsystems, Inc.: Java Web Start. http://java.sun.com/j2se/1.5.0/docs/guide/javaws/
11) Sun Microsystems, Inc.: Default Policy Implementation and Policy File Syntax. http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html
12) Sun Microsystems, Inc.: Java 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api/
13) Sun Microsystems, Inc.: Permissions in the Java 2 Standard Edition Development Kit (JDK). http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html
14) Sun Microsystems, Inc.: Security Code Guidelines. http://java.sun.com/security/seccodeguide.html
15) Sun Microsystems, Inc.: appletviewer — The Java Applet Viewer. http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/appletviewer.html
16) Sun Microsystems, Inc.: JAR File Specification. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html
17) Viega, J., Mutdosch, T., McGraw, G. and Felten, E.W.: Statically Scanning Java Code: Finding Security Vulnerabilities, *IEEE Software*, Vol.17, No.5, pp.68–74 (2000).

(平成 18 年 11 月 21 日受付)
(平成 19 年 3 月 1 日採録)

兒島 尚（正会員）

1975 年生．1998 年東京工業大学工学部情報工学科卒業．2000 年同大学大学院理工学研究科計算工学専攻修士課程修了．2001 年（株）富士通研究所入社．現在，同社 IT コア研究所セキュアコンピューティング研究部に勤務．Java を中心としたソフトウェアセキュリティ，Web アプリケーションセキュリティの研究に従事．

金谷 延幸

1967 年生．1992 年群馬大学工学部情報工学専攻修士課程修了．同年（株）富士通研究所入社．現在，同社 IT コア研究所セキュアコンピューティング研究部に勤務．ソフトウェアセキュリティ，Web アプリケーションセキュリティの研究に従事．