オンラインセキュリティ実験基盤

若森拓馬[†] 井出真広[†]
中田晋平[†] 倉光君郎[†,††]

近年，ソフトウェアの不正な使用による個人情報の流出被害が増加している．これを防ぐために，適切なアクセス制御を施した安全なソフトウェアを設計することが重要になっている．アクセス制御は，OS レベルやアプリケーションレベルで行われている．我々は，Web 上にスクリプト実行環境 Aspen を構築したが，セキュリティホールが存在したために被害が発生してしまった．そこで，プログラミング言語からアクセス制御を行うことで，安全なソフトウェアを開発できると考えた．我々は，Web 上でプログラミングを行える安全なスクリプト実行環境を実現するために，スクリプト言語の実行エンジンによってアクセス制御する手法を提案する．我々は，提案機構を KonohaScript へ実装し，Aspen 上でのアクセス制御の実例を用いて，提案するアクセス制御手法が有用であることを示した．

An Online Scripting Infrastructure for Security Experimentation

Takuma Wakamori,[†] Masahiro Ide,[†] Shinpei Nakata[†]
and Kimio Kuramitsu[†,††]

Recently, security breaching caused by illegal use of a software is increasing. To prevent this, it is important to perform access control on a software. OS and Application have their own access controller because OS level subjects that can perform actions in the system differ from that of application level. We have constructed an online scripting environment named Aspen. However, Aspen had a security risk, server filesystems have suffered some attacks. To construct a safe online scripting environment, we propose a scripting language engine based access control method. We implemented our proposal system into KonohaScript. To evaluate our system, we show some access control examples that performed on Aspen.

1. はじめに

近年，管理ミスや不正アクセスによる個人情報の流出被害が増加している．そのため，機密情報を扱うソフトウェアを開発する際には，ソフトウェアを正しく動作させるだけでなく，不正な利用や攻撃を防ぐためのセキュリティ機構を実装することが重要である．ソフトウェアのセキュリティは，動作の主体が，許可された権限を越えて動作することのないように，適切なアクセス制御を行うことで向上する．

SELinux を代表とするセキュア OS には，このようなアクセス制御を行うアクセス制御機構が備えられており，OS のユーザがシステムにアクセスする際にアクセス制御が行われている．セキュア OS は，アクセス制御のルールをセキュリティポリシーとして管理している．一方，Web アプリケーションや顧客管理システム等のソフトウェアは，アクセス制御の対象となるユーザが OS とは異なるため，OS とは別にアクセス制御を行なっている．これらのアプリケーションの開発者は，アプリケーションフレームワークのもつ機能を利用し，ユーザ情報をデータベースで管理するなどして，アクセス制御を実装している．

我々は，Web ブラウザでプログラミングを行うために，スクリプト実行環境 Aspen を Web アプリケーションとして構築した．しかし，ユーザが記述したコードを評価する際に，アプリケーション側で適切なアクセス制御を行わなかったために，サーバ側のファイルシステムを不正に操作されてしまうという問題が発生した．

我々は，スクリプト言語 KonohaScript の設計・開発を行なっている．そこで，アプリケーションではなく，

† 横浜国立大学大学院工学府
Graduate School of Engineering, Yokohama National University
†† 日本科学技術振興機構/CREST
Japan Science and Technology Agency/CREST

プログラミング言語の実行環境にアクセス制御機構を実装することで，不正な操作が防止できると考えた．

我々は，Web からプログラミングを行える安全なスクリプト実行環境の構築を目的に掲げた．

本論文では，スクリプト言語の実行エンジン上で動作するアクセス制御機構を提案する．そして，提案機構を KonohaScript に実装し，Aspen を用いて行ったセキュリティ実験の事例を用いて，提案機構を評価する．

本論文の構成は以下のとおりである．第 2 節では，OS・アプリケーション上に実装されている，セキュリティモデルやアクセス制御の関連技術を紹介する．第 3 節では，Aspen の概要を説明し，Aspen で発生したセキュリティの問題について述べる．第 4 節で，提案するスクリプト言語ベースのセキュリティ機構の概要を述べる．第 5 節では，KonohaScript における実装の詳細について述べる．第 6 節では，Aspen 上で行ったアクセス制御の事例と，発表時に行ったアクセス制御のデモ内容を示す．第 7 節で関連研究を紹介し，第 8 節で本論文を総括する．

2. アクセス制御

アクセス制御は，動作の主体（Subject）の，システム内のリソース（Object）に対する，読み込みや書き込みなどの動作（Action）の可否を制御することで，セキュリティ目的を達成するための技術である．本節では，ソフトウェアに実装されているセキュリティ機構の関連技術として，OS レベル，アプリケーションレベルで実装されているアクセス制御機構についてそれぞれ説明する．

2.1 OS によるアクセス制御

OS は，任意アクセス制御（DAC）によって，ファイルやディレクトリへのアクセス権をファイルごとに管理している．例えば Linux では，ファイルの所有者がユーザ，グループ，その他の全ユーザという属性に応じて，読み込み，書き込み，実行などの権限を設定できる．しかし DAC では，管理者ユーザの権限を不正に奪われると，システムを保護することができなくなってしまう．そこで，SELinux などのセキュア OS は，Subject からの Object に対する全ての参照に対して，アクセス権が正しく設定されているかどうかを検査するリファレンスモニタと呼ばれるアクセス制御機構を備えている[7]．

図 1 は，リファレンスモニタの概念を表す．リファレンスモニタを正確に動作させるために必要なのは，「どの Subject がどの Object に，どのような Action が行えるのか」を正確に表現したセキュリティポリシーを厳格に定めることである．SELinux では，セキュリティコンテキストと呼ばれるラベルを Subject と Object に付与し，その組み合わせをセキュリティポリシーとして管理している．

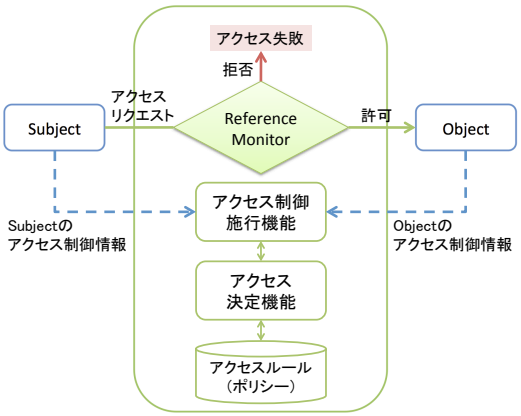

図 1 リファレンスモニタ

2.2 アプリケーションによるアクセス制御

アプリケーションは，言語のフレームワークを利用するなどして，アクセス制御を行なっている．例えば，PHP のフレームワークである Zend Framework には，アクセス制御リスト（ACL）と権限管理を提供する Zend_Acl という機能がある[5]．Zend Framework 上で開発されたアプリケーションは，ACL の設定によって，要求してきた Subject（ロール）が制限された Object（リソース）へのアクセスを認められているかどうかを制御することができる．ここでロールとは，guest や member，admin など，アプリケーションのユーザが属する役割を表す．

3. Aspen とセキュリティ問題

我々は，静的型付けスクリプト言語 KonohaScript[4] の設計と開発を行なっている．本節では，KonohaScript で構築した Web アプリケーション Aspen の概要と，Aspen で発生したセキュリティ問題について述べる．

3.1 Aspen

Aspen☆は，本学の学生のプログラミング教育を目的として，我々が開発した Web アプリケーションである．Aspen は，ユーザが Web ブラウザ上のテキストエリアに入力した KonohaScript のコードをサーバ

☆ http://konoha.ubicg.ynu.ac.jp/aspen/ にて公開中

側で実行するプログラミング環境を提供する．図 2 は，Aspen のユーザインタフェースを表す．画面の上半分は，コードの実行結果をテキストとして表示する部分であり，画面の下半分は，コードを入力するためのエディタ（テキストエリア）である．Aspen のユーザである学生は，学籍番号とパスワードを入力してログインした後，コードを入力する．入力したコードは，画面上部の Run ボタンを押すと，サーバ側の CGI スクリプト内で実行される仕様となっている．

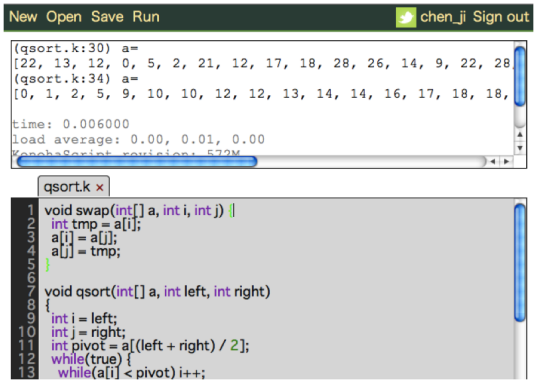

図 2　Aspen のメイン画面

Aspen がユーザの入力したコードを実行する手順は以下のとおりである．

(1) Run ボタンを押した時，テキストエリアに入力されている文字列をファイルに書き込み，サーバ側に保存する．
(2) 保存したファイルを KonohaScript で実行するシェルコマンドを実行する．
(3) 実行したプロセスの標準出力の結果をエスケープしてブラウザに表示する．

3.2 セキュリティ問題

第 2 節で述べたように，アクセス制御は OS・アプリケーションレベルで行われている．

Aspen は，ユーザの入力したスクリプトコードを実行するという性質上，スクリプトコードに任意の OS コマンドを実行するコードが含まれる可能性がある．以下は，OS コマンドを実行する KonohaScript の構文である．

```
System.exec("rm -rf /");
```

このコードが実行されてしまうと，サーバ上のファイルシステム上で，CGI の実行権限と同様の所有者権限をもつファイルが全て消去されてしまう．そのため，ユーザの入力したコードを安全に実行するためには，このような不正な操作を制限しなければならない．

従来は，サーバアプリケーションに対して，CGI を実行するための最小の権限を与え，CGI をシステムから保護された領域で動作させるなどの，OS レベルでのアクセス制御が行われてきた[3]．しかし Aspen では，アプリケーションのユーザに合わせて，行える操作を柔軟に変更する必要がある．なぜなら，Aspen には，学部生・大学院生・管理者などのロールにあわせて，言語やアプリケーションの機能を制限する用途があるからである．

4. 提　　案

本研究の目的は，Web からプログラミングを行える安全なスクリプト実行環境を構築することである．この目的を達成するために，我々は，スクリプト言語ベースでアクセス制御を行う手法を提案する．

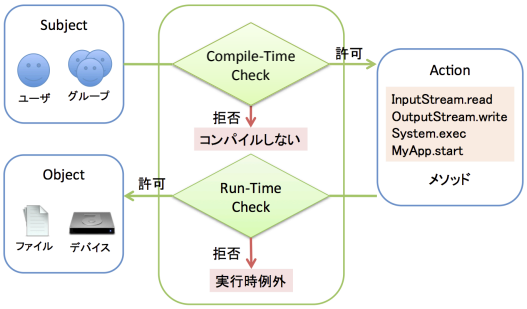

図 3　システムアーキテクチャ

図 3 は，提案機構の全体像を表す．我々は，スクリプト言語のプログラムが実行時にコンパイルされる特性を生かして，アクセス制御をプログラムのコンパイル時と実行時の二段階に分けて行う方針をとる．

4.1 コンパイル時のメソッドレベルアクセス制御

オブジェクト指向プログラミング言語では，オブジェクトの動作はメソッドに抽象化されている．そこで我々は，Subject の Action を制御するために，メソッド単位でアクセス許可を記述する手法を提案する．この手法の特徴は，アクセス制御の対象が Object ではなく，Action になっていることである．このような方式を採用することで，「誰が，どのような動作をするか」という形でポリシーファイルを定義することができるため，アクセス制御機構の汎用性が向上すると考えられる．

4.2 実行時のパラメータレベルアクセス制御

コンパイル時にメソッドの実行が許可された場合，メソッドは通常の定義で実行される．しかし実際には，

ユーザの入力値など，プログラムの実行時に決定する値に基づいて，アクセス制御を行いたい場合が考えられる．これに対応するため，我々は実行時のパラメータレベルアクセス制御を提案する．これは，「動作の対象がどのようなものか」といった，Action と Object のつながりを表現している．

5. 設計と実装

我々は，提案機構を KonohaScript に実装するにあたり，以下の設計方針をとった．

- アクセス制御機構の有効・無効にかかわらず，プログラムが動作するようにする．
- アクセス制御のための記述を，ソフトウェア本体と分離して記述する．

本節では，設計した 2 つのアノテーションとポリシーファイルの記法，及びアクセス制御の手順について述べる．

5.1 @Restricted アノテーション

まず我々は，アクセス制御対象のメソッドを指定するために，@Restricted アノテーションを設計した．@Restricted アノテーションの付加されたメソッドは，コンパイル時にポリシーファイルに許可があるかどうかチェックされ，許可のある場合のみコンパイルされる．図 4 に示すように，文字列の検索など，アクセス制御の必要がないと判断されるものについては，@Restricted を付けずにメソッドを定義することで，アクセス制御の対象から除外する．このようにアクセス制御対象のメソッドを明示的に指定することで，実行に権限が必要なメソッドを明確にするとともに，ポリシーファイルの記述量を減らす事ができる．

アクセス制御機構を無効としたとき，このアノテーションは単に無視されるため，変更前と同様に動作する．

```
/* アクセス制御の対象から除外する */
int String.indexOf(String s);
/* アクセス制御の対象とする */
@Restricted String System.exec();
@Restricted this Socket.new();
@Restricted void OutputStream.write();
```

図 4 @Restricted アノテーション

5.2 @Around アノテーション

我々は，メソッドの前後にセキュリティチェックを挿入する構文として，@Around アノテーションを実装した．@Around アノテーションをつけてメソッドを再定義した時，元のメソッドの中身を `proceed` 構文で呼び出すことができる．このような再定義を行った時，アスペクト指向におけるコードの織り込み（weaving）と同様に，すべてのメソッド呼び出しに対してコードが挿入される．図 5 は，@Around アノテーションを付加して `InputStream.new` メソッドを最定義する例を表す．再定義の後，`InputStream.new` メソッドが呼び出される際には，引数として与えられた文字列 `urn` が`"/home"`という文字から始まる場合のみ実行され，それ以外の場合はセキュリティ例外となる．

アスペクト指向を用いることで，ソフトウェアのメインのロジックと，セキュリティチェックのコードを明確に区別して記述することができるため，関心の分離が行える．

```
@Around this
InputStream.new(Path urn, String mode)
{
  if (urn.startsWith("/home"))
    proceed(urn, mode);
  else
    throw new Security!!();
}
```

図 5 @Around アノテーション

5.3 アクセス制御の手順

KonohaScript にアクセス制御を適用する手順は次の通りである．

まず，アクセス制御の対象となるユーザと，そのユーザによる実行を許可するメソッド群の組み合わせをポリシーファイルとして用意しておく．アクセス制御は，KonohaScript のプロセス起動時にユーザを指定することで適用される．例えば，ユーザ `Student` に対するアクセス制御を有効にして，スクリプト `test.k` を実行するには，KonohaScript を以下のコマンドで起動する．

```
$ konoha --enforce-security=Student test.k
```

ここで，スクリプトのコンパイル時にポリシーファイルを参照し，メソッドの実行の可否が判断される．実行が許可されていない場合は，そのメソッドはコンパイルされず，実行時に許可されていないメソッドを呼び出した場合は，セキュリティ例外が発行される．

6. Aspen を用いたセキュリティ実験

我々は，設計したアクセス制御機構を用いて，ユーザの記述したコードを安全に実行することができるか

どうか確かめるために，Aspen にアクセス制御を実装し，セキュリティ実験を行った．本節では，二つのアクセス制御事例と，発表時に行ったデモの内容を紹介し，提案機構の有用性について考察する．

図 6 は，アクセス制御を実装した Aspen のアーキテクチャを表す．Aspen は，ユーザの入力したコードとユーザ情報をもとに，ポリシーファイルを参照して，入力コードが許可されているかどうか判断したのち，サーバ側でコードを実行する．

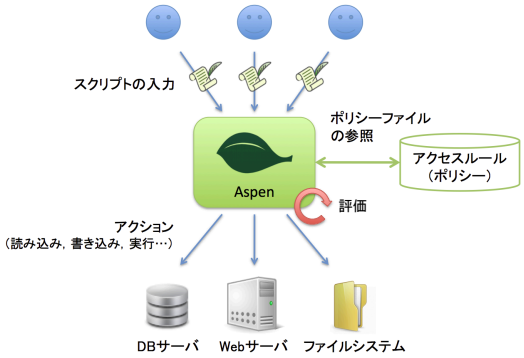

図 6 Aspen のアーキテクチャ

Aspen を安全な Web アプリケーションにするためには，以下のセキュリティ要件がある．

- ユーザの入力したコードを実行することで，サーバ側の機密ファイルが読み書きされたり，OS コマンドを実行して危険な操作が行われてはならない．
- 管理者でないユーザが，管理画面を表示できてはならない．

まず，サーバ側のファイルシステムを不正に操作することがあってはならない．従来このような操作を制限するためには，第 2.1 節で述べたとおり，OS レベルでアクセス制御を記述する必要がある．

また，管理者のみが管理画面を表示するなど，Web アプリケーションで想定されるユーザに対するアクセス制御を適切に行う必要がある．

6.1 OS コマンドのアクセス制御

OS コマンドによる攻撃を防ぐには，パラメータレベルのアクセス制御が有効である．例えば，nmap を用いたポートスキャンを防ぐためには，OS コマンドを実行する `System.exec` メソッドに対して，図 7 のようなセキュリティチェックのコードを記述する．

同様に，他の OS コマンドの実行を制限するには，大きく二つの方針がある．

- `System.exec` コマンドのセキュリティチェックのコードを追加していく．

```
@Around String
System.exec(String cmd) {
  if (cmd.indexOf("nmap") != 0)
    throw new Security!!
      ("nmap is not allowed");
  else
    proceed(cmd);
}
```

図 7 nmap コマンドの禁止

- `System.exec` メソッドを直接呼び出すことを禁止し，各 OS コマンドの実行をメソッド化しておく．

ひとつ目は，図 7 と同様に，セキュリティチェックのコードを禁止したいコマンドについてそれぞれ追加していく方針である．禁止したい操作が少ない場合は，記述量を少なくできるため，この方針が良いと思われる．

ふたつ目は，実行を許可するコマンドをそれぞれメソッド化しておき，ポリシーによってアクセス制御を記述する方針である．この方針を用いる利点としては，禁止の条件が複雑であったり，許可する操作がパラメータによらず一定である場合に，コンパイル時のアクセス制御で賄えることである．また，許可するメソッドをそれぞれポリシーファイルに記述するため，ポリシーファイルの汎用性が向上すると考えられる．

6.2 ユーザ定義メソッドのアクセス制御

次に，アプリケーションに応じたアクセス制御を行う場合を考える．この場合は，開発者がメソッドを記述する際に，図 8 に示すように，単に@Restricted アノテーションを付加すれば良い．あわせて，`gotoAdmin` メソッドの実行を許可する対象となる Subject を，ポリシーファイルとして作成する．

```
@Restricted void MyApp.gotoAdmin() {
  /* original code */
}
```

図 8 gotoAdmin メソッドの禁止

6.3 アクセス制御のデモ

本節では，発表時に行ったアクセス制御デモの内容を報告する．

まずはじめに，以下のスクリプトを用意し，アクセス制御対象とした．

```
$ cat demo.k
ous = new OutputStream("result", "w");
ous << System.exec("ls -l") << EOL;
ous.close();
```

ポリシーファイルには JSON 形式で以下のものを用意した．

```
[
  {"name": "Bachelor", "permission":
   ["System.exec"]},
  {"name": "Graduate", "permission":
   ["System.exec", "OutputStream.new"]}
]
```

デモでは，以下の手順でアクセス制御の実例を示した．

(1) アクセス制御を行わない
(2) コンパイル時アクセス制御を行う
(3) 実行時アクセス制御を行う

はじめに，アクセス制御を行わずにスクリプトを実行すると，正常に実行され，ファイルが生成されることを確認した．

```
$ ls
demo.k policy
$ konoha demo.k
$ ls
demo.k policy result
```

次に，コンパイル時アクセス制御の例として，ロール”Bachelor”として，同様のスクリプト実行した．”Bachelor”にはファイルの生成が許可されていないため，ファイルを生成するメソッドがコンパイルされず，エラーとなった．

```
$ konoha --enforce-security=Bachelor demo.k
 - (demo.k:2) (error) konoha.System.exec is
 not allowed
```

最後に，実行時アクセス制御の例として，System.exec メソッドの引数の文字列をチェックして，許可されている場合は実行するコードを記述した．そして，KonohaScript のインタラクティブシェルにて正しく制御が行われることを示した．

```
$ konoha --enforce-security=Graduate
>>> @Around String System.exec(String cmd) {
  if (cmd.startsWith("ls")) {
    return proceed(cmd);
  } else {
    throw new Security!!();
  }
}
>>> exec("ls");
demo.k
policy
result
test.k
>>> exec("rm -rf /");
((eval):1) Security!!: rm -rf / is not
 allowed
  at ((eval):1) System.exec(cmd="rm -rf /")
```

7. 関 連 研 究

本節では，アクセス制御機構を備えたプログラミング言語の関連研究を述べる．

プログラミング言語にセキュリティ機構を導入する手法は，Java で発展してきた．Java のセキュリティアーキテクチャ[2] では，ネットワークから取得した信頼できないコードを実行するためのサンドボックスを JVM 内に構築し，システムを安全に保つ方針がとられている．プログラムの実行時にセキュリティポリシーとの照合を行うためのクラスとして，AccessController クラスが定義されている．AccessController.checkPermission メソッドを実行すると，実行時にクリティカルなシステム資源へのアクセス可否が決定される．しかし，AccessController を呼び出す方針では，セキュリティチェックのコードを挿入しなければならず，既存のソフトウェアに変更を加える必要があった．そこで Erlingsson ら[1] は，プログラムの実行時に信頼できないコードに対してセキュリティチェックを挿入する Inline Reference Monitor (IRM) を提案した．我々の提案するアクセス制御機構は，アスペクト的にセキュリティチェックを挿入することができるため，IRM と同様，セキュリティチェックのコードを本体と区別して記述することができる．しかし IRM とは異なり，プログラムのコンパイル時と実行時のアクセス制御を明確に区別している．

スクリプト言語での実装としては，Ruby のオブジェクトの汚染とセーフレベルがある[6]．Ruby では，ユーザの入力などの信頼できない値を汚染（taint）し，危険な操作の引数として使えないようにすることができる．また，4 段階のセーフレベルを独自に定義しており，ユーザの入力した文字列を評価するなどの目的にあわせて，スレッドごとにセーフレベルを上げることができる．我々の提案方式は，スレッド単位ではなくプロセス単位でアクセス制御を行う設計になっている．また，セーフレベルの確認などのセキュリティチェックを，アスペクトにより分離して記述することができる．

8. お わ り に

安全なアプリケーションを開発するためには，セキュリティ機構の実装が不可欠である．本研究では，Web からプログラミングを行える安全なスクリプト実行環境を構築するため，スクリプト言語の実行エンジンにアクセス制御機構を実装する手法を提案した．

スクリプトのコンパイル時と実行時の二段階でアク

セス制御を行う提案機構を KonohaScript に実装☆し，Aspen でアクセス制御実験を行った．二つのアクセス制御事例をもとに，実用的なアクセス制御を行えることを示した．

今後は，Java などの既存言語のセキュリティ機構とセキュリティチェックの記述性やパフォーマンスについて比較し，提案機構の有用性について評価したい．

質疑応答

Q ポリシー違反のメソッドをコンパイル時に未定義にしてしまうと，ポリシーによって KonohaScript の言語仕様が変わることになる．実行時エラーにした方がいいのではないか．(NTT 中山様)

A 発表時は，見やすさのためにコンパイルエラーの扱いにしていたが，現在はアクセス制御された警告を表示するようにしている．

Q ラッパー関数は遅延コンパイルなのか．(座長)

A 遅延コンパイルではない．遅延コンパイルは型が不明の場合に行われる．ライブラリ系は型が分かっているため，事前にコンパイルされる．

Q 呼び出した時のエラーを，未定義ではなくて，ポリシー違反のエラーにするのはどうか．(座長)

A 今回はデモとして，型としてセキュリティがチェックできるということを見せるために，無いように見えたほうが良いと判断した．普段は disallowed のエラーを出すようにしている．

Q アノテーションの付いていないメソッドの内部で，read/write などの禁止された操作が呼ばれてしまっている場合は，禁止された操作が行えてしまうのではないか．そのような場合は，アノテーションの付け忘れとして処理するしかないのか．(質問者不明)

A 現状はそのように処理するしかない．しかし，ライブラリ設計時に注意深くアノテーションを付加するかどうか判断すれば，大部分は防げると考えている．

Q 制御の対象は，最終的にすべてシステムコールになるのか．それとも，システムコールを使用しないメソッドが対象になることもあるのか．(座長)

A どちらも対象となる．Aspen 上でスクリプトを安全に実行するという目的に対しては，危険だと思われる操作はすべてシステムコールになっている．

Q 単にシステムコールを制御するのであれば，KonohaScript をサンドボックス上で動作させれば，KonohaScript 内部に手を入れなくても，スクリプトの挙動を制御できる．なぜそのようなアプローチを取らなかったのか．サンドボックス化する方が，より汎用的なのではないか．(座長)

A Web アプリケーションでスクリプトを安全に実行するという利用ケースでは，サンドボックスで十分かもしれない．しかし今回は，Web アプリケーションだけでなく，あらゆるアプリケーションに適用可能な汎用的なアクセス制御機構を設計した．

Q Ruby ではスレッド単位でアクセス制御をしているという話だったが，KonohaScript ではプロセスを新たに起動してアクセス制御を行なっている．プロセスレベルならば，サンドボックスで事足りてしまう．長期間生存しているインスタンスがいて，KonohaScript 自身のオブジェクトを操作させたくないといった場合には，アクセス制御を導入する意義はある．今後どのような方針で行なっていくのか．(東京大学 荒川様)

A 今後はスレッドによる実行にも対応する予定だが，Aspen の仕組みをスレッドで行なってしまうと，スレッド間の共有変数の操作などを含む，大部分の操作が危険な操作に該当してしまうため，今回のようにプロセスで起動する方法が適している点もあると考えている．

Q Web のインタフェースからスクリプトを実行させる場合，無限ループを防ぐことができないという問題が考えられるが，Aspen ではどのようになっているのか．(NTT 中山様)

☆ http://konohascript.org より入手可能

A Aspen はプロセス単位で実行しているため，alarm シグナルを利用して長期間応答のないプロセスを kill する形で対策している．

Q パスの指定の例に”/home”のようなコードが挙げられていたが，相対パスやシンボリックリンクなどの設定をしないと制御から外すことができてしまう．詰めていくと，結局 OS レベルのアクセス制御リストを記述するのと同じように厳密にやらなければならない．OS コマンドの一部を許可するといった例であれば，@Around でアスペクト的に操作するのは有効だと思う．この辺りのバランスはどう考えているのか．（東京大学 荒川様）

A ファイル単位での細かい制御を行う場合は，適宜 OS のアクセス制御を利用する方が良い．しかしその方法は，アプリケーションのユーザの権限に応じてアクセス制御するには適していない．OS の保護機構と今回設計したアクセス制御機構は，同時に利用するのが良いと考えている．

謝　　辞

本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．

参 考 文 献

1) Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *In IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
2) Li Gong and Gary Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2nd edition, 2003.
3) Poul-Henning Kamp and Robert Watson. Building systems to be shared, securely. *Queue*, 2:42–51, July 2004.
4) Kimio Kuramitsu. Konoha: implementing a static scripting language with dynamic behaviors. In *Workshop on Self-Sustaining Systems*, S3 ’10, pages 21–29, New York, NY, USA, 2010. ACM.
5) R. Lerdorf, K. Tatroe, and P. MacIntyre. *Programming PHP*. O’Reilly Series. O’Reilly, 2006.
6) Y. Matsumoto. *Ruby in a nutshell: a desktop quick reference*. A Nutshell handbook. O’Reilly, 2002.
7) Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: system support for diverse security policies. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.