独自セキュア OS Set Based Access Control の設計及び実装

内藤 陽介
e-mail: jouxsuke@gmail.com

概要

既存の Linux 用セキュア OS は、複雑過ぎて理解しずらい、運用方法が繁雑で使いずらい、又は、表現力が貧弱で用件を満たせない、といういずれかの問題を持っている。従って、これらの問題を克服した独自セキュア OS を設計し、その設計に基づいて実装を行った。この独自セキュア OS により、Linux 用セキュア OS に対するユーザの選択肢の幅が大きく広がることが期待出来る。

1 序論

セキュア OS の必要性が高まるにつれ、様々な Linux 用セキュア OS が提案されている。代表的な Linux 用セキュア OS として、AppArmor、SELinux、PitBull、TOMOYO Linux、LIDS、RSBAC 等が上げられる。

これらのセキュア OS はそれぞれ独自のアクセス制御方法を採用している。例えば、SELinux は全てのプロセスとアクセス対象オブジェクトにそれぞれドメインとタイプと呼ばれる属性を付与し、あるドメインがあるタイプに対してどのような権限を持つのかを定義することによりアクセス制御の設定を行う。このアクセス制御を行う方法を、本論文ではアクセス制御モデルと呼ぶ。

これらの Linux 用セキュア OS は設計思想に基づいて大きく二つに分類される。軍事的な利用を念頭に置いたセキュリティ至上主義派と、容易に設定し利用出来るべきと考えているカジュアルセキュリティ派である [7]。SELinux、PitBull、及び RSBAC は前者に分類され、AppArmor、TOMOYO Linux、及び LIDS は後者に分類される。両派とも自分達の目標を達成しているという点においては、これらの Linux 用セキュア OS を評価することが出来る。ところが、両派にはそれぞれ異なる問題点が存在する。

セキュリティ至上主義派の Linux 用セキュア OS のアクセス制御モデルは複雑化する傾向がある。また、その Linux 用セキュア OS を導入したシステムの運用方法は繁雑化する傾向がある。アクセス制御モデルが複雑であれば、初歩的なセキュリティ管理者がそれを理解してシステムのセキュリティポリシーを表現するのは非常に困難である。また、システムの運用方法が繁雑であれば、そのシステムの運用中にシステム管理者がその Linux 用セキュア OS の機能を無効にしてしまうことが多くなる。

一方、カジュアルセキュリティ派の Linux 用セキュア OS のアクセス制御モデルはセキュリティポリシーの表現力が非常に貧弱である。セキュリティポリシー表現力が貧弱であれば、複雑な実社会で必要とされるセキュリティポリシーを表現することが出来ず要件を満たせないため、他の Linux 用セキュア OS を導入することが多くなる。

この様に、既存の Linux 用セキュア OS は以下のいずれかの問題を持っている。

1. アクセス制御モデルが複雑過ぎて理解しづらい。
2. 運用方法が繁雑で使いづらい。
3. セキュリティポリシー表現力が貧弱で要件を満たせない。

本研究では、容易に理解し運用出来、しかも、セキュリティポリシー表現力を確保している、という両派の中間的な Linux 用セキュア OS を開発することを目的にしている。本論文では、この Linux 用セキュア OS を SBAC(Set Based Access Control) と呼ぶことにする。

2 セキュア OS の評価指標と既存の Linux 用セキュア OS の評価

序論において既存の Linux 用セキュア OS はアクセス制御モデルが複雑過ぎたりセキュリティポリシー表現力が貧弱であることを述べたが、ここでアクセス制御モデル複雑度とセキュリティポリシー表現力を定義し、それらの定義を既存の Linux 用セキュア OS に適用する。しかし、全ての既存の Linux 用セキュア OS を評価することは出来ないため、セキュリティ至上主義派の Linux 用セキュア OS の代表としての SELinux とカジュアルセキュリティ派の Linux 用セキュア OS の代表としての AppArmor のみ評価する。

2.1 アクセス制御モデル複雑度

アクセス制御モデル複雑度を定量的に扱う方法として、アクセス制御モデルを ER 図として表現し、その図に対して ER 図の複雑度の定義式を適用する、という方法も考えられる [5]。しかし、アクセス制御モデルが複雑な場合はそれを ER 図として表現するのは困難であり、また、複雑度を定量的に比較するよりも具体的なセキュリティポリシーを実際に表現したほうが複雑度をより実感することが出来るため、ここでは複雑度を定性的に比較することにする。

ここで、一般ユーザである foo ユーザのみが/bin/date コマンドを実行出来る、という非常に単純なセキュリティポリシーを考える。特定のセキュリティ制御モデルがこのセキュリティポリシーをどれほど簡単に表現出来るかをそのアクセス制御モデル複雑度として定義する。

このセキュリティポリシーを SELinux のアクセス制御モデルを使用して表現すると表 1 の様になる。

表 1 SELinux のアクセス制御モデルによるセキュリティポリシー表現

アクセス制御

domain	type	object_class	permission
user_domain	date_type	ファイル	読み込み
user_domain	date_type	ファイル	実行

タイプ付与

object	type
/bin/date	date_type

ドメイン遷移

access_domain	type	transition_domain
user_domain	date_type	date_domain

RBAC(ユーザとロール)

user	role
foo	date_role

RBAC(ロールとドメイン)

role	domain
date_role	date_domain

タイプ遷移に対する設定は不要である。全てのユーザはログイン時に user_domain ドメインを持つ/bin/bash プロセスを起動すると仮定する。このプロセスが date_type タイプを付与された/bin/date ファイルを実行すると[*1]、ドメイン遷移の設定により/bin/date プロセスには date_domain ドメインが付与される。一方、ロールとドメインの設定により date_role ロールを持つユーザのみが date_domain ドメインを付与されたプロセスを実行することが出来るが、ユーザとロールの設定により foo ユーザのみが date_role ロールを持っている。従って、foo ユーザのみが/bin/date ファイルを実行出来ることになる。以上の説明により、SELinux のアクセス制御モデルを使用すると、これ程単純なセキュリティポリシーを表現するのでさえも様々な項目を考慮する必要があることが分かる。従って、SELinux のアクセス制御モデル複雑度はかなり高いと定性的に考えられる。

一方、AppArmor のアクセス制御モデルにはユーザという概念が存在しないため、このセキュリティポリシーを表現することは出来ないが、そのアクセス

[*1] この行為はアクセス制御の設定により許可されている

制御モデルはアクセス主体、アクセス種別、及びアクセス対象の三要素の一つの関係のみから構成される。従って、そのアクセス制御モデル複雑度が最も低いことは明らかである。

2.2 セキュリティポリシー表現力

今日に至るまで様々なアクセス制御モデルが提案されているが、大まかには MLS(MultiLevel Security)、RBAC(Role Based Access Control)、及び DBAC(Domain Based Access Control) に分類される [4]。従って、これらの三つのアクセス制御モデルに基づいたセキュリティポリシーを表現出来るか否かをセキュリティポリシー表現力として定義する。
SELinux のアクセス制御モデルは MLS 及び RBAC を含んでいる。しかし、SELinux のドメインの概念はお互いに包含関係が存在しないため、DBAC に基づいたセキュリティポリシーを表現しようとすると冗長になってしまう。従って、DBAC との親和性が高くはないが、SELinux のセキュリティポリシー表現力は一般に高いと言える。
一方、AppArmor のアクセス制御モデルにはユーザの概念が存在しないため、これらのアクセス制御モデルに基づいたセキュリティポリシーを表現出来ない。従って、セキュリティポリシー表現力が非常に低いことになる。

3 SBAC の設計

3.1 SBAC の設計の基本方針

SBAC は Set Based Access Control の略であり、セットという概念をアクセス制御モデルの中心に据えた Linux 用セキュア OS である。本研究は、容易に理解し運用出来、しかも、セキュリティポリシー表現力を確保している、という Linux 用セキュア OS を開発することを目的としていた。その目的を達成するために、SBAC の設計の基本方針を以下の様に定義する。

1. アクセス制御モデルで扱う項目の数を出来る限り少なくする。
2. 但し、それによってアクセス制御モデルのセキュリティポリシー表現力が著しく低下する場合、必要な項目は導入する。
3. たとえセキュリティが向上する手法であったとしても、それがシステムを容易に運用することを妨げる場合、その手法は採用しない。

以上の基本方針に基づいて SBAC の設計を行うことにする。

3.2 アクセス制御モデル設計

3.2.1 SBAC のアクセス制御モデルの概要

全てのユーザとオブジェクトは必ず一つのセットに所属することになる。但し、一つのセットには複数のユーザとオブジェクトが所属することが出来る。あるユーザがあるオブジェクトに対してアクセスするためには、そのユーザが所属しているセットが、そのオブジェクトが所属しているセットに対して、そのアクセスを行う権限を持っていなければならない。セットは階層構造を持っており、セットは複数の親セットを持つことが出来る。セットはその親セットの権限を継承することが出来る。これはオブジェクト指向における多重継承と同じ考えである。
これらのユーザ、オブジェクト、アクセス制御、及びセットの四つの関係について以下にそれぞれ詳細に説明する。

3.2.2 アクセス制御

SBAC におけるアクセス制御は、特定のアクセス、そのアクセス主体であるユーザが所属するセット、及びそのアクセス対象であるオブジェクトが所属するセット、これらの三つの関係で表現される。これらの三つに関係が存在する時に限りそのアクセスが許可される。
例えば、user1 が/bin/date ファイルを読み込み実行出来るという設定は以下の様になる。
ここでは、user1 及び/bin/date ファイルは user_set

access set	permission	target set
user_set	読み込み	date_set
user_set	実行	date_set

及び date_set にそれぞれに所属すると仮定する。そして、user_set が date_set を読み込み実行出来る様に設定されている。このアクセス制御の設定は root ユーザに対しても適用されるため、root ユーザがアクセス要求を行う場合、root ユーザが所属するセットにそのアクセス権限が与えられていなければ、root ユーザといえどもその行為を行うことは出来ない。従って、SBAC はセキュア OS の必要条件の一つである強制アクセス制御機能を備えていることになる。

3.2.3 セット

セットは親セットを持つ、即ち階層構造を形成している。
あるセットが特定の親セットを持っている場合、そのセットはその親セットが持っている権限を継承することが出来る。
セットは二つ以上の親セットを持つことが出来る。SBAC の設計の基本方針の一つ目のみを考慮した場合、セットは一つの親セットしか持つことが出来ないようにすべきである様に思われる。ところが、基本方針の二つ目を考慮した結果、セットは二つ以上の親セットを持つことが出来るように決定した。
例えば、set3 が set1 と set2 の二つのセットを親セットに持っているとする。また、set1 が set1 を実行する権限を与えられ、set2 が set2 を読み込む権限を与えられているとする。この場合、set3 は set1 が持つ権限と set2 が持つ権限の両方の権限を継承しており、set2 を実行する権限と set3 を読み込む権限を与えられていることになる。

3.2.4 ユーザ

ユーザは特定のセットに所属することが出来る。しかし、ユーザは一つのセットにしか所属することが出来ない。SBAC の設計の基本方針の一つ目を考慮した結果、ユーザは二つ以上のセットに所属することが出来ないことに決定した。

3.2.5 オブジェクト

オブジェクトは特定のセットに所属することが出来る。しかし、オブジェクトは一つのセットにしか所属することが出来ない。これは、ユーザは二つ以上のセットに所属することが出来ないことに決定したのと同じ理由による。

3.2.6 SBAC のアクセス制御モデルの表による表現

以上の四つの関係で表現される SBAC のアクセス制御モデルを表形式で表現すると以下の様になる。

$$acl(access\_set, permission, target\_set) \quad (1)$$

$$set(set, parent\_set) \quad (2)$$

$$user(user, set) \quad (3)$$

$$object(object, set) \quad (4)$$

これらの式を SBAC の設定ファイルの内容を設計する際に用いる。

3.3 アクセス制御モデル以外の設計

3.3.1 アクセス制御対象オブジェクト種別及びアクセス種別

Linux のファイルは、通常ファイル、ディレクトリ、キャラクタスペシャルデバイス、ブロックスペシャルデバイス、FIFO、ソケット、そして、シンボリックリンクに分類される。この中で SBAC のアクセス制御対象となるオブジェクトは通常ファイル、キャラクタスペシャルデバイス、そして、ブロックスペシャルデバイスのみとする。つまり、ディレクトリに対するアクセス制御は行わないことにする。これは、あるディレクトリ配下の全てのオブジェクトに対してあるセットを付与している場合、それらのオブジェクトは動的に変化する可能性があるため、どれがファイルでどれがディレクトリかは予め決定することは出来ず、適切なセキュリティポリシーを作成することが出来ないと考えたからである。例えば、あるユーザと/usr/local/apache2 ディレクトリ配下の全てのオブジェクトにまとめて一度

に httpd セットを付与したとする。httpd セットが httpd セットを実行する権限を与えない場合、httpd セットに所属するユーザは/usr/local/apache2 ディレクトリ配下にあるディレクトリ内のファイルやディレクトリにアクセスすることが出来なくなってしまう。逆に、httpd セットが httpd セットを実行する権限を与えた場合、httpd セットに所属するユーザは/usr/local/apache2 ディレクトリ配下の全てのファイルを実行することが出来てしまうが、これは明らかにセキュリティポリシーに反する。
また、ポートに対するアクセス制御は行わないことにする。これは、iptables と呼ばれるファイアウォールを使用することによってポートに対するきめ細かなアクセス制御が既に可能であると考えたためであるが、もし要望があれば今後検討するつもりである。
また、ファイルに対するアクセス種別は読み込み、書き込み、実行、そして、削除の四つとする。ファイルに対する通常の権限種別である読み書き実行に加えて削除権限を権限種別に加えたのは、SBAC においてはディレクトリに対するアクセス制御を行わないことに決定したため、本来ディレクトリに対するアクセス制御によって行うファイル削除権限の制御が行えなくなってしまったからである。
アクセス種別として更に Linux ケイパビリティを加える。本来 Linux ケイパビリティは、従来 root ユーザに無条件に付与されていた特権を細分化し、その細分化された各特権を選別して root ユーザに付与する機構である。例えば、root ユーザから CAP_CHOWN 特権を剥奪すると、root ユーザは自分の所有するファイル以外のファイルの UID や GID を自由に変更出来なくなる。SBAC ではこの細分化された特権を root ユーザのみでなく一般ユーザにも付与出来るようにする。Linux ケイパビリティは必要最小限の特権を各ユーザに付与することを可能にするため、SBAC はセキュア OS の必要条件の一つである最小特権機能を備えていることになる。

3.3.2 設定ファイル

Linux のアプリケーションの設定ファイルは/etc/(アプリケーション名) 配下に配置されることが多い。この慣習に倣って SBAC の設定ファイルを/etc/sbac ディレクトリ配下に配置することにする。
1 式から 4 式までの結果を反映して、/etc/sbac/acl.conf ファイル、/etc/sbac/set.conf ファイル、/etc/sbac/user.conf ファイル、
及び/etc/sbac/object.conf ファイルの四つの設定ファイルを用意する。
各設定ファイルはカンマ区切りの CSV 形式で記述する。CSV 形式の各区は 1 式から 4 式までの表の各列に対応している。
各設定ファイル内においてナンバーサイン (#) が現れた場合、そのナンバーサインから次に現れる改行までをコメントとみなす。また、スペースは無視される。
/etc/sbac/acl.conf 内においては半角英数字とアンダースコア (_) とハイフン (-) のみ使用することが出来る。アクセス種別を表現する文字列を表 2 にまとめる。この表に記述されているアクセス種別以外にも、[1] に記述されている Linux ケイパビリティもアクセス種別として指定することが出来る。

表 2　**行為の種類を表現する文字列の情報**

行為の種類を表現する文字列	行為の種類
read	ファイルの読み込み
write	ファイルの書き込み
execute	ファイルの実行
remove	ファイルの削除

アクセス種別に Linux ケイパビリティを指定する場合、そのアクセス対象オブジェクトのセットとして null という文字列を指定することにする。これは、Linux ケイパビリティにはアクセス対象オブジェクトが存在しないためである。/etc/sbac/acl.conf の設定例を以下に記述する。

```
#access set, permission, target set
set1,read,set1
set2,CAP_CHOWN,null
```

第一行はコメントであり、/etc/sbac/acl.conf ファイルの記述形式を表現している。第二行は、set1 に所属するユーザが set1 に所属するオブジェクトを読

み込む権限を与えられていることを意味している。第三行は、set2 に所属するユーザが CAP_CHOWN ケイパビリティを与えられていることを意味している。

/etc/sbac/set.conf 内においては半角英数字とアンダースコア (_) とハイフン (-) のみ使用することが出来る。また、あるセットが親セットを持たない場合、その親セットとして null という文字列を指定することにする。/etc/sbac/set.conf の設定例を以下に記述する。

```
#set,parent set
set1,null
set2,set1
```

第一行はコメントであり、/etc/sbac/set.conf ファイルの記述形式を表現している。第二行は、set1 がどのセットにも所属していないことを意味している。第三行は、set2 が set1 を親セットに持つことを意味している。

/etc/sbac/user.conf 内においては半角英数字とアンダースコア (_) とハイフン (-) のみ使用することが出来る。/etc/sbac/user.conf の設定例を以下に記述する。

```
#user,set
user1,set1
```

第一行はコメントであり、/etc/sbac/user.conf ファイルの記述形式を表現している。第二行は、user1 が set1 に所属することを意味している。

/etc/sbac/object.conf 内においては半角英数字とアンダースコア (_) とハイフン (-) とスラッシュ (/) とドット (.) とアスタリスク (*) のみ使用することが出来る。最初にディレクトリ名が現れた後、スラッシュが続き、次にアスタリスクが二つ連続して続き、最後にカンマが現れた場合、その最初に現れたディレクトリ配下の全てのファイルに対してセットを付与することを意味する。ディレクトリ配下の全てのファイルにはそのディレクトリ配下にあるディレクトリ内のファイルも含まれている。/etc/sbac/object.conf の設定例を以下に記述する。

```
#object,set
/bin/cat,set1
/home/user2/**,set2
```

第一行はコメントであり、/etc/sbac/object.conf ファイルの記述形式を表現している。第二行は、/bin/cat ファイルが set1 に所属することを意味している。第三行は、/home/user2 配下の全てのファイルが set2 に所属することを意味している。

3.3.3 ファイルに対するセット付与方法

ファイルに対してセットを付与する方法として、ファイル名に付与する方法、i-node に付与する方法、及びファイルシステムの拡張領域にラベルとして付与する方法の三つの方法が考えられる。

i-node の方法を採用した場合、新規に作成されたファイルに対して再度セットを付与する必要がある。また、ラベルの方法を採用した場合、セキュア OS の機能を無効にした後再びセキュア OS の機能を有効にした際に全てのファイルに対して再度セットを付与する必要がある。従って、i-node の方法やラベルの方法を用いた場合、運用が繁雑になってしまう。

一方、ファイル名の方法を採用した場合、以下で述べるハードリンクによる脆弱性の問題を考慮する必要がある。従って、ファイル名の方法を用いた場合、セキュリティが低下する。

それぞれの方法一長一短であるが、SBAC の設計の基本方針の三つ目を考慮した結果、ファイル名にセットを付与する方法を採用することにする。

3.3.4 ハードリンクとシンボリックリンクの扱い

SBAC においてはファイルが所属するセットを i-node ではなくファイル名に付与しているため、あるファイルに対してハードリンクを張るとそのファイルに対する本来のアクセス制御が回避されてしまう。例えば、/bin/date ファイルに date_set セットが付与されており、date_set セットに所属するユーザのみが date_set セットに所属するファイルを実行出来るようアクセス制御されていたとしても、/home/user1 配下の全てのファイルが set1 に

所属している場合、user1 ユーザによって/bin/date ファイルに対して/home/user1 配下にハードリンクが張られてしまうと、date_set セットに所属しない user1 も/bin/date を実行することが出来てしまう。従って SBAC においては、ハードリンク元とハードリンク先が同じセットを持つ場合を除いては、ハードリンクを張ることを一切禁止することにする。
但し、たとえシンボリックリンクからあるファイルへのアクセスを試みられたとしても、最終的にはリンク先のファイル名に対してアクセス制御が行われるため、シンボリックリンクにはハードリンクの持つ問題が存在しない。従って、シンボリックリンクは許可することにする。

4 SBAC のプロトタイプの実装

4.1 実装環境

実装は以下の環境で行ったが、LSM を持った Linux カーネルソースであれば、必ずしも以下の環境でなくとも実装可能である [6][3]。

表 3 **実装環境の情報**

項番	項目	値
1	カーネルバージョン	2.6.20.16
2	gcc バージョン	4.1.2

4.2 LSM とは

LSM は Linux Security Module の略であり、Linux において様々なアクセス制御モデルを容易に実現するのを可能にするために開発されたフレームワークである。
LSM のインターフェースは、

```
int register_security
(struct security_operations *ops);
int unregister_security
(struct security_operations *ops);
int mod_reg_security
(const char *name,
 struct security_operations *ops);
int mod_unreg_security
(const char *name,
 struct security_operations *ops);
```

の 4 つの関数から構成されている [2]。
全ての引数に現れている struct security_operations 型は、特定のアクセス要求に対応して自動的に呼び出されるフック関数へのポインタから構成されている。フック関数の例としては、オープンされたファイルに対する読み込み時や書き込み時に自動的に呼び出される file_permission 関数等が挙げられる。
register_security 関数は、security_operations を登録することにより、特定のアクセス制御モデルを登録する。
既に他の security_operations が登録されている場合には register_security 関数は 0 を返すため、その場合には mod_reg_security 関数を用いてアクセス制御モデルを入れ替えなければならない。
register_security 関数を用いて security_operations を登録した場合には、unregister_security 関数を用いて登録を解除しなければならない。一方、mod_reg_security 関数を用いて security_operations を登録した場合には、mod_unreg_security 関数を用いて登録を解除しなければならない。
実装を出来るだけ容易に行うため、本研究では LSM を使用して SBAC のプロトタイプの実装を行った。

4.3 処理の流れ

LSM のインタフェースを利用すると、SBAC のフローチャートは図 1 の様になる。
SBAC モジュールをロードすると同時に初期化処理が行われる。まず、/etc/sbac ディレクトリ配下の設定ファイルの内容をメモリに登録する。次に、security_operations を登録する。
初期化処理が終了すると、SBAC はユーザからのアクセス要求を待ち続ける。SBAC がアクセス要求を受け取ると、そのアクセス要求に対応したフック関

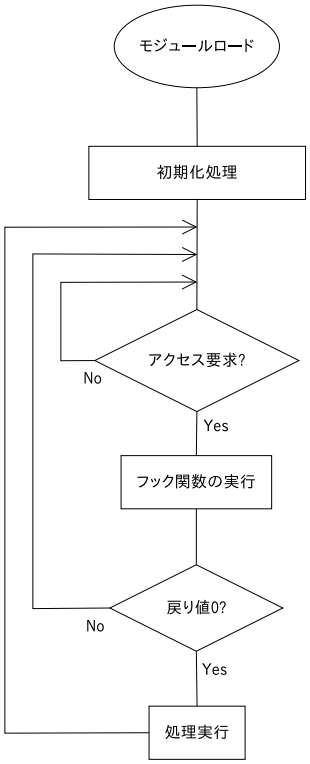

図 1　SBAC のフローチャート

数が呼び出される。
呼び出されたフック関数は引数として与えられた情報を基にそのアクセス要求を許可するか否か判定する。そのアクセス要求を許可する場合は戻り値として 0 を返し、許可しない場合は 0 以外の値を返す。そのフック関数からの戻り値が 0 である場合、ユーザから要求されたアクセスを実行し、その後ユーザからのアクセス要求待ち状態に戻る。その戻り値が 0 以外である場合、そのアクセスを実行せずに、そのままユーザからのアクセス要求待ち状態に戻る。
以下に各処理を簡潔に説明する。

4.3.1　初期化処理

LSM を利用したモジュールをロードすると、最初に security_initcall 関数の引数に与えた関数ポインタが指す関数を実行することになっている。従って、sbac_init 関数へのポインタを引数として security_initcall 関数を実行することにより、sbac_init 関数に記述された初期化処理を実行させることにする。

■設定ファイルの内容のメモリへの登録　sbac_init 関数において、まず register_sbac_config を実行することにより、設定ファイルの内容をメモリに保存する。register_sbac_config の実行が成功するとそれは 0 を返し、次の初期化処理である security_operations の登録処理が実行される。逆にそれが失敗すると 0 以外の値を返し、初期化処理と同時に SBAC モジュールのロードを中断する。
register_sbac_config 関数において、/etc/sbac ディレクトリ配下の acl.conf、set.conf、user.conf、及び object.conf のカンマ区切りの CSV 形式で記述された設定内容をそれぞれ構造体の双方向リストとしてメモリに登録する。
/etc/sbac/acl.conf ファイル内の一データを表現する sbac_acl 構造体を以下の様に定義する。

```
struct sbac_acl
{
    char set_name[SET_NAME_LENGTH];
    char permission[PERMISSION_LENGTH];
    char object_set_name[SET_NAME_LENGTH];
    struct sbac_acl *next_acl;
    struct sbac_acl *previous_acl;
};
```

set_name はアクセス要求を行ったプロセスの実行ユーザのセットである。permission はそのアクセス種別である。object_set_name はそのアクセス対象オブジェクトのセットである。SET_NAME_LENGTH 及び PERMISSION_LENGTH はそれぞれセット名の最大バイト数及びアクセス種別を表す文字列の最大バイト数であり、本研究では 16 バイト及び 32 バイトとした。next_acl 及び previous_acl はそれぞれ双方向リストにおける次のデータへのポインタと前のデータへのポインタである。
/etc/sbac/set.conf ファイル内の一データを表現する sbac_set 構造体を以下の様に定義する。

```
struct sbac_set
```

```
{
    char set_name[SET_NAME_LENGTH];
    char parent_set_name[SET_NAME_LENGTH];
    struct sbac_set *next_set;
    struct sbac_set *previous_set;
};
```

set_name 及び parent_set_name はそれぞれセット名及び親セット名である。next_set 及び previous_set はそれぞれ双方向リストにおける次のデータへのポインタと前のデータへのポインタである。
/etc/sbac/user.conf ファイル内の一データを表現する sbac_user 構造体を以下の様に定義する。

```
struct sbac_user
{
    unsigned long uid;
    char user_name[USER_NAME_LENGTH];
    char set_name[SET_NAME_LENGTH];
    struct sbac_user *next_user;
    struct sbac_user *previous_user;
};
```

uid、user_name、及び set_name はそれぞれユーザ ID、ユーザ名、及びセット名である。uid は/etc/passwd ファイルからユーザ名を基に取得する。USER_NAME_LENGTH はユーザ名の最大バイト数であり、本研究では 32 バイトとした。next_user 及び previous_user はそれぞれ双方向リストにおける次のデータへのポインタと前のデータへのポインタである。
/etc/sbac/object.conf ファイル内の一データを表現する sbac_object 構造体を以下の様に定義する。

```
struct sbac_object
{
    char object_name[FULL_PATH_NAME_LENGTH];
    char set_name[SET_NAME_LENGTH];
    struct sbac_object *next_object;
    struct sbac_object *previous_object;
};
```

object_name 及び set_name はそれぞれオブジェクト名及びセット名である。
FULL_PATH_NAME_LENGTH はオブジェクト名又はフルパス名の最大バイト数であり、本研究では 1024 バイトとした。next_object 及び previous_object はそれぞれ双方向リストにおける次のデータへのポインタと前のデータへのポインタである。

■security_operations の登録 sbac_init 関数において、register_sbac_config の実行に成功すると、次に security_operations 型へのポインタを引数として register_security または mod_reg_security を実行することにより、SBAC のアクセス制御モデルを表現するフック関数を登録する。register_security または mod_reg_security の実行のどちらか一方でも成功すると、初期化処理は成功して終了する。逆にそれらの両方の実行が失敗すると、初期化処理と同時に SBAC モジュールのロードを中断する。

4.3.2 フック関数の実行

■sbac_inode_permission sbac_inode_permission フック関数はあるファイル実行要求が行われた際に実行され、その要求を出したプロセスがそのファイルを実行する権限を持つか否かをチェックする。

■sbac_file_permission sbac_file_permission フック関数はあるファイルの読み書き要求が行われた際に実行され、その要求を出したプロセスがそのファイルを読み書きする権限を持つか否かをチェックする。

■sbac_inode_unlink sbac_inode_unlink フック関数はあるファイルの削除要求が行われた際に実行され、その要求を出したプロセスがそのファイルを削除する権限を持つか否かをチェックする。

■sbac_capable sbac_capable フック関数は特定の Linux ケイパビリティを必要とするアクセス要求が行われた際に実行され、そのアクセス要求を出したプロセスがそのケイパビリティを持っているか否かチェックする。

■sbac_inode_link　sbac_inode_link フック関数はハードリンクを張る際に実行され、ハードリンク元とハードリンク先が同じセットを持つか否かチェックする。

■その他のフック関数　LSM はダミーの security_operations を提供しており、各フック関数のデフォルトの振舞を定義している。上記以外のフック関数は SBAC とは無関係のため、そのダミーの security_operations のフック関数をそのまま採用する。

5　評価

5.1　アクセス制御モデル複雑度

第 2.1 小節で定義したセキュリティポリシーを SBAC のアクセス制御モデルを使用して表現すると表 4 の様になる。

表 4　SBAC のアクセス制御モデルによるセキュリティポリシー表現

アクセス制御

access set	permission	target set
date_set	読み込み	date_set
date_set	実行	date_set

セット

set	parent set
date_set	null

ユーザ

user	set
foo	date_set

オブジェクト

object	set
/bin/date	date_set

アクセス制御とオブジェクトの設定より、date_set に所属するユーザのみが date_set に所属する/bin/date を実行することが出来るが、ユーザの設定より、foo ユーザのみが date_set に所属している。従って、foo ユーザのみが/bin/date ファイルを実行することが出来る。
以上より、SBAC のアクセス制御モデルを使用すると、SELinux のアクセス制御モデルと比較してこのセキュリティポリシーが簡単に表現出来ることが分かる。一方、AppArmor のそれと比較すると明らかに複雑ではある。従って、SBAC のアクセス制御モデル複雑度は SELinux のそれと AppArmor のそれの中間に位置することになる。

5.2　セキュリティポリシー表現力

SBAC のアクセス制御モデルを適切に用いると、第 2.2 小節で述べた三つのアクセス制御モデルに基づいたセキュリティポリシーを表現することが出来る。
SBAC のアクセス制御モデルには機密度の高低によるアクセス制御という概念が存在しないため、MLS に基づいたセキュリティポリシーを表現しようとすると冗長になってしまうが可能ではある。
また、RBAC の Role を SBAC のアクセス制御モデルのユーザが属する Set に読み替えると、RBAC に基づいたセキュリティポリシーをそのまま表現することが出来る。
更に、DBAC の Domain の階層構造を上下逆転した構造を SBAC のアクセス制御モデルの Set の階層構造として再構成することにより、DBAC に基づいたセキュリティポリシーをそのまま表現することが出来る。これは、DBAC においては階層構造の上にある Domain がより権限を持つのに対して、SBAC のアクセス制御モデルにおいては階層構造の下にある Set がより権限を持つからである。
以上より、SBAC のアクセス制御モデルのセキュリティポリシー表現力は SELinux のそれとほぼ同等であることがわかる。一方、AppArmor のそれと比較すると明らかに表現力が豊かである。従って、アクセス制御モデル複雑度同様 SBAC のセキュリティポリシー表現力は SELinux のそれと AppArmor のそれの中間乃至 SELinux と同等に位置することになる。

5.3 設計実現度の評価

第 3 節で行った SBAC の設計内容が実現されていることを確認する。
評価対象システムとして、一般ユーザである foo ユーザのみが/bin/date コマンドを実行してシステム時刻を変更出来る、というセキュリティポリシーを持ったシステムを考える。
このセキュリティポリシーを表現するために、foo ユーザと/bin/date コマンドのみを admin セットに所属させ、admin セットに所属するユーザのみが admin セットに所属するファイルを実行出来、更に admin セットのみに CAP_SYS_ADMIN と CAP_SYS_TIME ケイパビリティを与えることにする。CAP_SYS_ADMIN ケイパビリティはそれぞれシステム管理用の操作を行うことが出来るケイパビリティであり、CAP_SYS_TIME はシステムクロックを変更出来るケイパビリティである。
セキュリティポリシー設計の内容の設定ファイルへの記述は本来セキュリティ管理者が行うべきであるが、本評価対象システムでは root ユーザがシステム管理者とセキュリティ管理者を兼任していることにする。従って、まず root ユーザが上記セキュリティポリシー設計の内容を設定ファイルへ記述する。
次に、SBAC モジュールをロードするため、root ユーザが以下の様に modprobe コマンドを実行する。

```
# modprobe sbac
```

modprobe コマンドを実行すると、設定ファイルの内容がコンソールに出力される。この出力内容と設定ファイルの内容を比較することにより、設定ファイルの内容が正しくメモリに登録されていることを確認することが出来た。
次に、root ユーザが以下の様に/bin/date を実行してシステム時刻を変更しようと試みる。

```
# /bin/date 070700002008
```

この時、以下の様に出力される。

```
-su: /bin/date: Operation not permitted
```

ところが、foo ユーザが/bin/date コマンドを実行すると何も出力されない。従って、アクセス制御モデルが正しく実装されていること、そして、ファイルに対するアクセス制御とケイパビリティの制御が正しく実装されていること、これらの二つを確認出来た。
最後に、root ユーザが/bin/date を不正に実行するために、以下の様に/bin/date へのハードリンクを/tmp ディレクトリ配下に作成することを試みる。

```
# ln /bin/date /tmp
```

この時、以下の様に出力される。

```
ln: creating hard link 'tmp/date' to
'bin/date': Operation not permitted
```

これは、admin セットに所属する/bin/date に対して admin セットに所属しない/tmp/date ハードリンクを張ることが出来ないという SBAC の設計を反映している。ところが、ln コマンドに s オプションを付加することにより、/bin/date へのシンボリックリンクを/tmp ディレクトリ配下に作成すると、何も出力されずに成功する。従って、SBAC のハードリンクとシンボリックリンクに対する設計が正しく実装されていることを確認出来た。
以上より、第 3 節において行った設計の内容を全て正しく実装していることを確認出来た。

6 SBAC のプロトタイプの問題点

本研究で作成した SBAC のプロトタイプは、SBAC の実現可能性を証明することを目的としていた。従って、このプロトタイプを実運用において使用することは出来ない。このプロトタイプを発展させて、実運用において使用可能な水準にまで高めるために必要なことを考察する。
3.3.2 小節で説明した通り、現在/etc/sbac/object.conf 内でオブジェクトを容易に指定するために使用出来る特殊記号は、あるディレクトリ配下の全てのファ

イルを意味する**のみである。ところが、実運用においてはより柔軟にオブジェクトを指定する方法が必要とされる。例えば、ログファイルのみを指定するために log という拡張子を持ったファイルのみを指定するという要件は頻繁に発生する。特に、正規表現を使用出来ることが望ましい。従って、文字列用のライブラリを追加することにより特殊記号の数を増やす必要がある。
このプロトタイプにおいては、設定ファイルの記述形式に関する異常系の処理を行っていない。例えば、user.conf ファイルにそのシステム上に存在しないユーザを指定した場合、SBAC モジュールのロードに失敗する。従って、設定ファイルのあらゆる記述ミスを考慮に入れた異常系の処理を追加する必要がある。

7 結論

本研究では、独自の Linux 用セキュア OS である SBAC の設計と実装を行った。
既存の Linux 用セキュア OS は、設計思想に基づいて軍事的な利用を念頭に置いたセキュリティ至上主義派と容易に設定し利用出来るべきと考えているカジュアルセキュリティ派の二つに大きく分類される。本研究では、容易に理解し運用出来、しかも、セキュリティポリシー表現力を確保している、という両派の中間的な Linux 用セキュア OS を開発することを目的としていた。
この目的を達成するための方針に基づいて SBAC の設計を行った。階層構造を持つセットという概念でユーザとアクセス対象オブジェクトをグループ化することにより、容易に理解し、しかも、セキュリティポリシー表現力を確保したアクセス制御モデルを設計することが出来た。また、セットという属性をラベルとして付与するのではなく、アクセス対象ファイルのパス名に付与することにより、運用方法が繁雑化するのを回避することが出来た。
設計に基づいて SBAC のプロトタイプの実装を行った。SBAC を使用して評価対象システムに対するセキュリティポリシー設計を行い、SBAC の実現可能性を証明することが出来た。
SBAC は運用方法が繁雑化する問題に対してはカジュアルセキュリティ派の立場を取っているが、アクセス制御モデルの複雑化とセキュリティポリシー表現力については両派の中間的立場を取っている。従って、SBAC によって両派の中間的セキュア OS の重要性が理解されれば、現在の Linux 用セキュア OS が二つの派閥に大きく分類されている現在の勢力図に影響を与え、Linux 用セキュア OS のユーザの選択肢の幅が大きく広がることが期待出来る。
今後の課題としては、本研究で作成した SBAC のプロトタイプを発展させて、実運用において使用可能な水準にまで高める必要があると考えられる。

参考文献

[1] Manpage of capabilities, http://www.linux.or.jp/jm/html/ldp_man-pages/man7/capabilities.7.html.

[2] Greg Kroah-Hartman. Using the kernel security module interface, http://www.linuxjournal.com/article/6279, Nov. 2002.

[3] Neil Matthew, Richard Stones. *Beginning Linux Programming*. Wrox Pr Inc, Dec. 2003.

[4] Mikel L. Matthews. Position paper. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, 2001.

[5] Yuan Zhao, Hee Beng Kuan Tan, Wei Zhang. Software cost estimation through conceptual requirement. *Quality Software, 2003. Proceedings. Third International Conference on*, pp. 141–144, Nov. 2003.

[6] 高橋浩和, 小田逸郎, 山幡為佐久. Linux カーネル 2.6 解読室. ソフトバンククリエイティブ, Nov. 2006.

[7] 才所秀明. 思想の数だけセキュア os は生まれる, http://www.atmarkit.co.jp /fsecurity/rensai/secureos01/secureos01.html.