1

OSへのセキュリティ脅威とLinuxの強制アクセス制御

海外浩平（日本電気(株) OSS推進センター）

あらゆる計算機資源がネットワークで相互接続され，世界中のどこの誰とも分からない相手と通信を行う可能性がある昨今，情報システムのセキュリティ強化はまさに喫緊の課題である．セキュリティ問題の難しいところは，『これさえやっておけば大丈夫』という性質のものではなく，個々の情報システムが直面する脅威を洗い出し，技術・運用の双方から実行可能な対策を考案する必要がある点である．中でも，情報システムのセキュリティを考える上で，OSの提供するセキュリティ機能は最も重要な要素の1つである．本特集では，SELinuxやTOMOYO Linuxなど，v2.6系列のLinuxがサポートするセキュリティ機能について紹介するが，はじめに，これらのセキュリティ機能がどのような思想のもとに設計され，どのような脅威に対処するのかを解説する．

セキュリティ技術の役割

あらゆるセキュリティ技術には，それが対抗しようとする脅威とその前提条件が存在する（少なくとも，存在すべきである）．脅威とは，脆弱性を利用してセキュリティが保全されている状態に対して損害を与え得る手段のことを意味する．たとえば，ファイルシステムのアクセス制御は，適切な権限を有しない利用者が機密情報を参照するという脅威に対抗する．だが，それは計算機が物理的な攻撃を受けないことが前提である．第三者が任意にディスクを着脱できる環境ではアクセス制御は無意味であるので，たとえば暗号化など，別の手段を用いる必要が出てくる．

情報セキュリティマネジメントの国際標準であるISO/IEC 27002（JIS Q 27002）の定義を用いると，情報セキュリティが保全されている状態であると主張するには，情報資産が以下の3つの性質を満たした上で管理されている必要がある．

- 機密性（Confidentiality）
  認可されていない個人，エンティティまたはプロセスに対して，情報を使用不可または非公開にする特性．
- 完全性（Integrity）
  資産の正確さおよび完全さを保護する特性
- 可用性（Availability）
  認可されたエンティティが要求したときに，アクセスおよび使用が可能である特性

本稿では，Linuxにおいて，どういった脅威が想定され，これらの性質がどのように保全されるのかを説明する．

大雑把に言ってOSの役割は，ハードウェアの計算機資源を抽象化し，利用者にシステムコールという形でこれら資源へのアクセス手段を提供することにある．これにより，利用者はファイルや仮想記憶，ネットワークといった抽象的な資源を直感的な方法で扱うことが可能となる．

一方，我々が機密性・完全性・可用性を保全しようと考える情報資産には，それ自身が実態を持たない（intangible）ため，何らかの媒体に格納する必要がある．それは紙や石版かもしれないし，ディスク装置や磁気テープかもしれない．重要なのは，これ

ら媒体の価値はそれ自身の格納している情報資産によって決定付けられるという点である．したがって，情報資産をOS管理下の計算機資源（ファイルシステム等）に格納する場合には，中身の情報資産に応じて計算機資源の機密性・完全性・可用性を保全する技術が必要となる．

Linuxの対抗する脅威

Linuxは延べ数千人規模の開発者が参加する巨大なOSSプロジェクトであり，特定の誰かが定めたセキュリティ仕様や脅威モデルに基づいて開発サイクルが動いているわけではない．だが，一部のLinuxディストリビューションはISO/IEC 15408の認証を取得しており，その過程で，OS向けのPP（Protection Profile）[☆1]に適合すべく開発作業が行われた．したがって，どのような脅威に対抗することを目的としてセキュリティ機能要件がPPに列挙されているのかを分析することで，便宜的に，Linuxが対抗しようとする脅威を洗い出すことができる．

ISO/IEC 15408のポータル（http://www.commoncriteriaportal.org/）を参照すると，認証済みOS製品の一覧を見ることができる．Linux製品としては，Red Hat Enterprise LinuxやSuSE Linux Enterprise Serverが複数のベンダにより認証取得されており，多くの認証において，CAPP（Controlled Access Protection Profile）や，その機能強化版であるLSPP（Labeled Security Protection Profile）に準拠したセキュリティ機能要件を満たしていることが分かる．

LSPPやCAPPはアクセス制御を主体とした要件集であり，物理的に安全な環境（例：データセンタ内など）でシステムを運用するという前提がある．LSPPによると，情報システムとは共有された情報資産を管理し，複数の利用者にアクセス手段を提供するものであると定義されている．利用者はシステムの利用に先立って認証を受けねばならない．すべての情報資産（を格納するシステム資源）には機密レベルが設定され，利用者のアクセスに際しては後述の強制アクセス制御ポリシーを適用することを求めている．加えて，利用者の行動を監査ログに記録することで，セキュリティ事故が発生した場合にはその原因を究明することを求めている．

したがって，これらのセキュリティ機能要件が対抗しようとする脅威は，次のように要約することができる．

- 不特定多数による情報資産へのアクセス
  例）パスワードクラック，アプリケーション脆弱性を利用した侵入
- 認証済み利用者による情報の漏えい
  例）不注意によって，または故意で本来は開示すべきでない情報を公開してしまう
- セキュリティ事故発生時の証跡
  例）攻撃を受けた事実に長期間気が付かない

言うまでもなく，LSPPやCAPPで列挙された機能要件が，Linuxにおけるセキュリティ機能のすべてではない．たとえば，dm-cryptによるディスクの暗号化は，システムに対する物理的な攻撃から情報資産の機密性を保全するが，LSPPやCAPPではシステムを物理的に安全な環境で運用することが前提であるので，機能要件には含まれていない．

リファレンスモニタとアクセス制御

OSであるLinuxは，我々の情報資産を計算機資源の中に格納し，これら計算機資源にアクセスする手続きを提供する．たとえば，利用者プログラムa.outがファイルXを参照するとき，open(2)およびread(2)システムコールという手続きを介してファイルXの内容を読み出す．読み出した情報はプロセスのメモリ空間に保持されるが，メモリ空間保護機構により他のプロセスはOSを介さずにこれを参照することはできない．

この話のポイントは，ファイルXにアクセスする際には常にOSが介在するという点である．言い換えれば，利用者がファイルXにアクセスを試み

☆1 PPとはOSやRDBMSといった特定領域のIT製品向けのセキュリティ機能・評価要件のセットである．

るとき，そのアクセスが妥当なものであるか否かを検査するには，システムコールの呼び出しを監視すれば必要十分であるといえる．より一般化すると，これは OS に限った話ではなく，たとえば RDBMS ではテーブルなど DB オブジェクトへのアクセスに SQL を用いる．通常，SQL が唯一のアクセス手段であるため，DB オブジェクトへのアクセスが妥当であるか否か，同様に SQL の実行を監視すれば必要十分であるといえる．

理屈の上では，必ずしもアクセス制御を OS や DBMS に集約する必要はない．たとえば，多くの Web アプリケーションは Apache/httpd プロセスの一部として動作し，OS 上のユーザ権限と Web アプリケーション上の利用者が関連付けられていないという実用上の理由により，Web アプリケーションはあらかじめ想定されたすべてのアクションを実行する権限を有して実行され，すべての意味あるアクセス制御は Web アプリケーションによって行われる．アプリケーションプログラムが正しく設計・実装されている限り，このアプローチは間違ってはいない．

しかし，一般的にはコード量の増加とともに，バグや脆弱性の入り込む可能性は高くなる．OS や DBMS の既存のアクセス制御とは別にアプリケーションが独自にアクセス制御を実施しようとする場合，アクセス制御モデルおよび実装が適切であることに加えて，コード上で利用者の権限をチェックすべき場所はすべての実行パスを網羅していなければならない．だが，これは“システムコールの呼び出し”ほどに自明でない場合が大半であり，加えて，アプリケーションのバグ・脆弱性を突かれた場合にアクセス制御をバイパスされる可能性を否定できない．

■リファレンスモニタ

ソフトウェアとバグ・脆弱性を切り離せない問題であるという前提に立ち，その上で，これらの脆弱性が顕在化するリスクを最小化しようとするアプローチの 1 つがリファレンスモニタ（Reference Monitor）コンセプトである．リファレンスモニタとは，利用者（プログラム）がシステム資源にアクセスすることが妥当であるか否か，事前に定義されたルールに基づいて意思決定を行うモジュールであり，以下の 3 つの要件を満たす必要がある．

- 対タンパ性を備えていること（Tamperproof）
- あらゆるリクエストに対して実行されること（Always Invoked）
- 十分に小さく検証可能であること（Small enough to be fully tested）

まず，リファレンスモニタそれ自身が改ざんから保護されている必要がある．改ざんされたモジュールがアクセス制御の意思決定を行ったとしても，そもそも信用できないからである．次に，OS が利用者からの要求（システムコール）を受け取ったときには，常にリファレンスモニタを呼び出してお伺いを立てねばならない．個々のサブシステムが『利用者は root 権限を有しているので以降のチェックは必要ない』などと，勝手に判断することは許されない．最後にユニークなのは，バグ・脆弱性が存在しないことを検証できる程度にリファレンスモニタ自身が十分に小さくシンプルであることが求められていることである．

Linux の場合，リファレンスモニタに相当するのはカーネル空間に配置されたアクセス制御モジュールである（図 -1）．ユーザ空間で動作するプログラムがカーネル空間を直接書き換えることはできない．したがって，適切な権限を持ったプログラムがシステムコールを呼び出す（もちろん，この呼び出しはリファレンスモニタによって検証される）以外には，アプリケーションプログラムがこれを改ざんすることはできない．また，利用者が計算機資源にアクセスするには必ずシステムコールを利用する必要がある．したがって，利用者がシステム資源にアクセスを試みる際の実行パスを網羅し，カーネル内の必要な個所すべてにおいてリファレンスモニタを呼び出すよう実装することは比較的容易である．OS は多種多様なサブシステムから構成されるが，リファレンスモニタを呼び出すことと，その結果，返戻されたアクセス制御の意思決定に従うことは各サブ

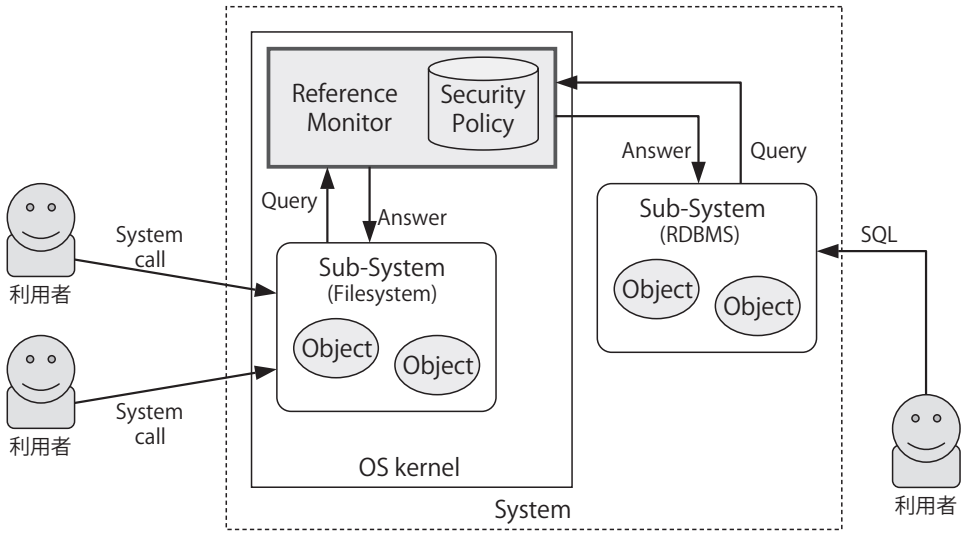

図-1
リファレンスモニタ

システムの責任であることに留意されたい．一方で，各々のアクセス制御モジュールがバグ・脆弱性を自明に検出できる程度に十分小さいかどうかは議論の余地がある．

利用者がOS管理下にある資源を利用するときには，必ずシステムコールを経由してOS内のサブシステムにリクエストを送出する．OS管理下にある資源にアクセスするには，システムコールが唯一の手段であることがポイントである．このとき，サブシステムはリファレンスモニタに対して要求されたアクセスの可否を問い合わせる．リファレンスモニタは「許可」または「禁止」の意思決定を行うので，サブシステムはそれに従って利用者からのリクエストの実行を制御する．システムコール以外にOS管理下の資源を利用する手段は存在しないので，実行パス上，この個所でアクセス制御を行うのが最も確実で間違いのない方法である．

一方で，Linuxは動的にカーネルモジュールをロードする機能を持っており，悪意のあるカーネルモジュールを(誤って)ロードした場合，原理的に，リファレンスモニタはカーネルモジュールがカーネル空間にアクセスすることを防ぐことはできない．したがって，カーネルモジュールをロードするシステムコールの時点で，信頼できない利用者の要求は禁止されるべきである．

この関係は，利用者がアプリケーション管理下にある資源を利用する場合にも容易に拡張できる．たとえば，RDBMS管理下にあるテーブルやスキーマにアクセスするには，SQLが唯一の手段である．このとき，RDBMSはリファレンスモニタに要求されたアクセスの可否を問い合わせ，その意思決定に従って利用者からのリクエストの実行を制御することができる．

なお，リファレンスモニタコンセプトを実装するハードウェアとソフトウェアの組合せはセキュリティカーネルと呼ばれる．

■アクセス制御

前節で説明した通り，リファレンスモニタは問合せに対して意思決定を行い，呼び出し元に要求されたアクセスの可否を返戻する．だが，どのように意思決定を行うかは(リファレンスモニタとして動作する)アクセス制御モジュールの実装に任されている．

アクセス制御とは，ある操作を実行する主体となるエンティティ（Subject）と，その操作の対象となるエンティティ（Object）との間に許可されている操作のセット(ルール)を定義し，そのルールに基づいて利用者のリクエストを許可または禁止することである．

情報システムの利用者とは本質的に人間のことで

ある．しかし，一般的には人間は自身の代理人であるソフトウェア（たとえばログインシェルや Web ブラウザが相当する）を介してのみ情報システムにアクセスすることができる．これらのソフトウェアは，何かの操作を実行する主体となるサブジェクト（Subject）と呼ばれ，実行時に誰の代理人として働くのかを示す属性を持つ．最も一般的な属性が利用者 ID（User Identifier）であり，システムの構成によってはケーパビリティ（Capability）や，強制アクセス制御に用いられるセキュリティラベル（Security Label）を持つこともある．利用者 ID も数多くのサブジェクト属性の 1 つであることに留意されたい．

一方，情報システムによって管理されている計算機資源で，利用者によるアクセスの対象はオブジェクト（Object）と呼ばれる．ファイルやディレクトリは代表的なオブジェクトであるが，その他，ネットワークソケット，セマフォや共有メモリセグメントなど，利用者の操作の対象すべてがオブジェクトである．これらのオブジェクトにも，それぞれオブジェクトの種類に応じた属性が設定されており，多くのオブジェクトは所有者 ID（Owner Identifier）や ACL（Access Control List）を持つ．ファイルであれば i ノード番号やパス名を持ち，システムの構成によってはセキュリティラベル（Security Label）が設定されていることもある．

アクセス制御メカニズムが特定のサブジェクトとオブジェクトの間にどのような操作が許可されているか意思決定を行う際，エンティティの識別にはこれらサブジェクト属性とオブジェクト属性が利用される．どの属性が利用されるかは，セキュリティモデルに依存する．

たとえば，伝統的な UNIX パーミッションに基づくファイルシステムアクセス制御では，サブジェクトはプロセスの利用者 ID によって識別され，オブジェクトは所有者 ID およびパーミッションビットによって識別される．これらの属性が定まれば，アクセス制御メカニズムは要求された操作（たとえば，ファイルへの書き込み）を許可すべきか，それとも禁止すべきか，意思決定を行うことができる．

一方，伝統的 UNIX パーミッションとは異なった観点から，利用者 ID/ 所有者 ID を用いることなく意思決定を行う方式も存在する．それが，後述の強制アクセス制御である．

また，伝統的な UNIX パーミッション機構では，ファイルの所有者は chmod コマンドを用いてアクセス権を変更することができる．ファイルに対する操作は，読み出し（Read）・書き込み（Write）・実行（eXecute），および所有者（ownership）権限を必要とするものに大別され，RWX の権限はそれぞれ所有者（Owner）・グループ（Group）・それ以外（Others）に対して設定することができる．利用者は Read/Write 権限なしに read(2) や write(2) システムコールを実行することはできない．また，アクセス権の変更は所有者権限を必要とする操作であり，ファイルの所有者は，誰にどのような操作を許可するかを正しく設定する責任を負う．

ここでのポイントは，誰がルールを設定するのかという点にある．たとえば，利用者 A（の ID を持ったプロセス）はファイル X に格納された情報資産の読み出しが許可されているとする．この場合，ファイル X の読み出しに続いて利用者 A は別のファイル Y を作成し，ファイル X から読み出した内容を書き込むことができる．伝統的な UNIX パーミッション機構においては，利用者 A は自身の所有するファイル Y のアクセス権を任意に変更することができるため，ファイル X の読み出しが利用者 A またはその他の利用者に許可されている限り，その内容が第三者に連鎖的に開示されることを防ぐ手段はない．

伝統的な UNIX パーミッション機構では，ファイルの所有者が任意にアクセス権を設定することができるために，任意アクセス制御（DAC ; Discretionary Access Control）と呼ばれる．一方，SELinux に代表される強制アクセス制御（MAC ; Mandatory Access Control）では，一元的に管理されたセキュリティポリシーがすべてのアクセス制御ルールを定める．したがって，たとえファイルの所有者であっても，あらかじめ許可された範囲を越えて情報を開示するこ

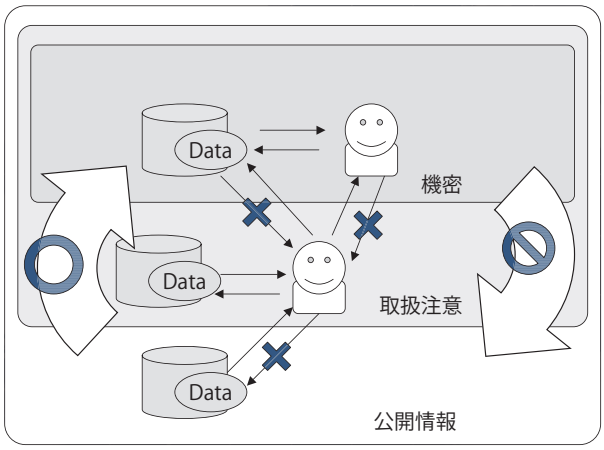

図 -2　情報フロー制御

とはできない．

たとえば，ファイルＸが『取扱注意』とラベル付けされており，これを読み出せるのは『機密』または『取扱注意』とラベル付けされた利用者（プロセス）であるとする（図 -2）．このとき，『取扱注意』とラベル付けされた利用者が新たにファイルＹを作成し，ファイルＸから読み出した内容を書き出すとする．任意アクセス制御とは異なり，ファイルＹの作成時にはデフォルトで利用者のラベル，すなわち『取扱注意』がファイルＹに付与され，これは所有者であっても変更することができない．したがって『取扱注意』のファイルＸから読み出した情報は，依然として『取扱注意』とラベル付けされたファイルＹに存在することとなり，元々のファイルＸの内容が連鎖的に第三者に開示されることはない．

このような枠組みを情報フロー制御と呼ぶ．一般的には機密度に応じて何段階かの階層が設定され，オブジェクトからの読み出し操作（オブジェクト→サブジェクトの情報の移動）においては，サブジェクトの機密度はオブジェクトと同じかより高く，オブジェクトへの書き込み操作（サブジェクト→オブジェクトの情報の移動）においては，サブジェクトの機密度はオブジェクトと同じかより低くなるよう，一元管理されたセキュリティポリシーによって強制される．この制約により，情報の移動する方向は常に機密度の低いドメインから高いドメインへと一方通行となる．したがって，たとえファイルの所有者であっても，セキュリティポリシーに反して，より機密度の低いドメインに情報を開示することはできない．

強制アクセス制御の配下においても，特定の利用者にファイルのラベル付け変更を許可するという形で，アクセス権の変更を許可することができる．だが，これは依然として一元管理されたセキュリティポリシーが許可する範囲において実行可能な操作（たとえば『取扱注意』→『機密』は可能でも，『取扱注意』→『公開』は不可など）であり，所有者が任意にアクセス権を設定できる任意アクセス制御の枠組みとは異なることに留意されたい．

LSM（Linux Security Modules）

我々がアクセス制御機能を実装するとしてまず考える必要があるのは，どのようにアクセス制御の意思決定を行うかという "How" の部分と，コード上のどこでアクセス制御の意思決定を行うかという "Where" の部分である．

Linux では，標準でファイルシステムに対する伝統的 UNIX パーミッション機構を備えているが，これは『アクセス制御』で説明した通り，利用者 ID とファイルパーミッション・所有者 ID の組合せを用いて意思決定を行うアクセス制御方式である．Linux カーネルでは inode_permission() 関数がこの判断を行い，open(2) システムコールの処理をはじめ，ファイルに対するアクセス権のチェックが必要な個所にこの関数呼び出しが埋め込まれている．仮に必要な場所でチェックが行われていなければ，それはバグである．

次に，この標準のアクセス制御の仕組みに加えて，別のアクセス制御の仕組みを追加する方法を考える．標準のアクセス制御と同様に "How" と "Where" の部分を実装する必要があるが，現状では任意アクセス制御における UNIX パーミッション機構のような，デファクトスタンダードと呼べるアクセス制御の仕組みは確立していない．したがって，利用者の目的や利用環境に応じてさまざまなセキュリティ機構が選択されることになるが，セキュリティ機

```
 :
 *
 * Security hooks for inode operations.
 *
 * @inode_create:
 *      Check permission to create a regular file.
 *      @dir contains inode structure of the parent of the new file.
 *      @dentry contains the dentry structure for the file to be created.
 *      @mode contains the file mode of the file to be created.
 *      Return 0 if permission is granted.
 :

int security_inode_create(struct inode *dir, struct dentry *dentry, int mode)
{
        if (unlikely(IS_PRIVATE(dir)))
                return 0;
        return security_ops->inode_create(dir, dentry, mode);
}
EXPORT_SYMBOL_GPL(security_inode_create);
```

図 -3　Linux における LSM の定義

構を 1 つ追加するたびに，Linux カーネル内の必要な個所すべてにアクセス制御関数を追加すると，コードが複雑になって保守性が下がったり，本来はチェックの必要な個所でのアクセス制御漏れが起きたりといったバグの誘発の要因となる（アクセス制御が必要なのはファイルシステムだけではなく，ネットワーク，プロセス間通信（IPC：Inter Process Communication）など幅広いサブシステムを含むことを留意されたい）．

注意が必要なのは，Linux は OSS であり，必ずしもセキュリティ／アクセス制御に関心の高い人ばかりが開発に参加しているわけではないということである．あるアクセス制御機構が特定のバージョンではアクセス制御の網羅性を満たしていたとしても，別のサブシステムで新機能が追加された結果，必要なアクセス制御をバイパスするコードパスが生まれてしまう可能性がある．

そこで，アクセス制御の "How" の部分は各モジュールでの作り込みが必要であるとしても，"Where" の部分を再実装するのは非合理であるので，アクセス権チェックの必要な場所でオプショナルなアクセス制御機構を呼び出すための共通のエントリーポイントが提供されることとなった．

これを LSM（Linux Security Modules）と呼び，v2.6 系列の Linux カーネルの標準機能である．LSM を用いることによって，開発者はアクセス制御のロジックの実装に注力できるほか，より多くの開発者が共通のフックを保守することで本体機能のバージョンアップに追従しやすくなる．

もし LSM が存在しなければ，SELinux 用のフック，Smack 用のフック，TOMOYO Linux 用のフック……とそれぞれ別個に保守しなければならないだろう．その場合，たとえばファイルシステムに修正を加えた結果，フックの場所を移動する必要があるとして，ファイルシステムの開発者が各々のセキュリティ機構の求めるフックの妥当性について正しく理解しなければならなくなってしまう．

実際には LSM とは巨大な関数ポインタ（フック）の集合であり，図 -3 は Linux カーネルにおける LSM 関数定義の一部である．v2.6.34 カーネルでは合計で 187 種類の LSM フックが定義されており，ソースコードには個々のフックの役割と期待される動作，引数として渡される情報が記述されている．

ここで定義された LSM 関数 security_inode_create() は，利用者がファイルを作成する権限を有するか否か意思決定を行う関数である．意思決定に必要な情報が渡されるが，それがどのように判断材料として利用されるのか，呼び出し元からは隠蔽されている．

たとえば，Linux カーネルがファイルを新規作成する処理である vfs_create() 関数（図 -4）は，利用者のファイル作成権限をチェックするために security_inode_create() 関数を呼び出す．この関数は LSM の inode_create フックを呼び出すが，vfs_create() 関数

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode,
               struct nameidata *nd)
{
        /* 1. UNIXパーミッションに基づくアクセス権チェック */
        int error = may_create(dir, dentry);

        if (error)
                return error;

        if (!dir->i_op->create)
                return -EACCES; /* shouldn't it be ENOSYS? */
        mode &= S_IALLUGO;
        mode |= S_IFREG;

        /* 2. LSMを通して、セキュリティ機構の意思決定を問い合わせる */
        error = security_inode_create(dir, dentry, mode);
        if (error)
                return error;

        /* 3.ファイルシステムのcreate()手続きを呼び出して、ファイルを新規作成 */
        error = dir->i_op->create(dir, dentry, mode, nd);
        if (!error)
                fsnotify_create(dir, dentry);
        return error;
}
```

図-4　Linuxにおける LSM の利用

から見ると，アクセス制御機構がどのように意思決定を行うかは重要ではなく，LSM の要求する情報を引数として与え，結果として『ファイルの新規作成』を許可するのか否かが明確になればよい．その結果，呼び出し元である vfs_create() は，アクセス制御機構の意思決定に従い，『禁止』であれば以降のファイル新規作成処理を中止して利用者にエラーを返却する．

Linux の強制アクセス制御まとめ

本特集で紹介する SELinux や TOMOYO Linux，その他，Linux カーネルに統合されている Smack や，今後に期待のかかる AppArmor は，すべて LSM フックの先でアクセス制御の意思決定を行い，それを呼び出し元に返却する構造を持っている．これは，Linux カーネルが利用者から何かリクエストを受けた際に，セキュリティモジュールが一元的にその可否を意思決定するという構造を持っていることを意味しており，まさに『リファレンスモニタ』で述べたものに相当するものである．

それゆえ，リファレンスモニタモデルに起因する長所（たとえば，root ユーザを含むあらゆる利用者からのシステムコール呼び出しを漏れなく捕捉可能であるとか，ファイルシステムとネットワークなど異なるサブシステムに対するセキュリティポリシーを一元管理できるであるとか）を持つ一方，リファレンスモニタモデルであるがゆえの制約も存在することを忘れてはならない．たとえば，第三者がシステムに物理的にアクセスできるような環境で運用されている場合には，SELinux や TOMOYO Linux を導入したとしても，その環境においてセキュリティが保全されているとは言えない．

これら Linux のセキュリティ強化機能には，従来の任意アクセス制御では実現が難しかった機能も含まれている．しかし，前提条件・運用環境・直面する脅威を検討の上で必要な対策を講じるという，セキュリティ対策の基本は不変である．どのようなセキュリティ対策を講じるのであっても，その点は留意しておかねばならない．

（平成 22 年 6 月 25 日受付）

■海外浩平　kaigai@ak.jp.nec.com

筑波大学大学院経営・政策科学研究科卒業．修士（ビジネス）．2003 年より日本電気（株）勤務，OSS/Linux の開発・サポート業務に従事する．主に SELinux を中心としたセキュリティ機能の開発に取り組み，現在は，RDBMS に強制アクセス制御を付加する SE-PostgreSQL のメインライン化に取り組んでいる．