TOMOYO Linux - タスク構造体の拡張によるセキュリティ強化Linux

原田 季栄　保理江 高志　田中 一男
(株)NTTデータ 技術開発本部
e-mail: {haradats, horietk, tanakakza}@nttdata.co.jp

概 要

コンピュータシステムのセキュリティを強化するための手段としてセキュリティ強化OSの導入が始まっている。多くのセキュリティ強化OSではユーザに対する役割分担及び資源に対するラベル付けに基づくアクセス制御を行っているが、そのためのアクセスポリシーが複雑になりがちであり、万人向けとは言い難い。本論文では、筆者らが開発したアクセスポリシーの自動定義機能を備えたTOMOYO Linuxの特長および実装について紹介する。TOMOYO Linuxでは、ラベル付けを必要とせず、またプロセスに対してアクセス制御を行うことにより、単純ながら効果的な強制アクセス制御を行うことが可能である。

1. はじめに

通常のLinux/UNIX におけるアクセス制御は、セキュリティビットと呼ばれるファイルシステム上の管理情報に基づいて行われ、ファイルやディレクトリの所有者はアクセス許可の設定を自由に変更できる。また、アクセスを要求したプロセスがシステム管理者権限を持つ場合には、アクセス許可の設定に関わらずアクセスが認められるため、バッファオーバーフロー等により外部からシステム管理者権限を持つプロセスの制御を奪われると、壊滅的な被害を受けてしまう点がセキュリティ上の問題点として指摘されていた[1,2]。これを解決する手法として、強制アクセス制御 (Mandatory Access Control) が考案され、SELinux[3,4]等のオープンソース実装が登場するようになった。またLinuxカーネルバージョン2.6には、セキュリティ拡張を容易とするフレームワークであるLSM (Linux Security Modules) [5]が標準で組み込まれ利用できるようになった。

ほとんどのLinux 強制アクセス制御の実装では、カーネル内でシステムコールの呼び出しを捕捉(フック)し、それが妥当なものかどうかを管理者が作成したアクセスポリシーに照らして正しいと判断した場合のみ実際の処理を行う形で実現されている。Linux においては全ての機能はカーネルが提供しているため、システムコールを捕捉する方式は確実に Linux のセキュリティを強化できる。しかし、それは全ての問題を解決するものではない。強制アクセス制御の導入には、その実装の粒度に応じた詳細なアクセスポリシーの策定及び管理運用が必要である。また、SELinux ではファイルやディレクトリ等の資源に対して「ラベル」と呼ばれる識別子が常に適切に付与されていなければならない。これらの結果、既存の Linux 強制アクセス制御の実装は、機能的には必要条件を満たしたものであっても、現実の導入や運用を考えると大きな障壁が残っていると考える。筆者らはこうした課題を解決するものとして、独自のセキュリティ強化 Linux、TOMOYO Linux を開発した。

Task Oriented Management Obviates Your Onus on Linux.
Toshiharu HARADA, Takashi HORIE and Kazuo TANAKA,
Research and Development Headquarters, NTT DATA CORPORATION

2. TOMOYO Linux

TOMOYO Linux とは"Task Oriented Management Obviates Your Onus on Linux"(タスク構造体に基づく制御は Linux に対するあなたの重荷を取り除きます)の略で、以下の特徴を備えたセキュリティ強化 Linux である。

- アクセスポリシーの自動定義機能を持つ
- タスク構造体の拡張による強制アクセス制御を実現
- ファイルやディレクトリに対するラベル付けが不要
- 改ざん防止 Linux[6]と組み合わせが可能
- プロセスが自発的に権限を放棄可能
- 初心者でも理解しやすいアクセスポリシー書式
- コンソール環境用アクセスポリシーエディタ付き

以下にそれぞれの特徴について説明する。

2.1. アクセスポリシーの自動定義機能

強制アクセス制御は、管理者が定めたアクセスポリシーに基づき厳格かつ詳細にアクセス要求の諾否を判断する。従って、参照する可能性のある全ての資源を把握し、許可を与えなければいけない点が現実の運用を困難にしている。例えば、httpサーバであるApacheが参照するファイルやディレクトリがApacheの設定ファイルに全て列挙されていればアクセスポリシーの策定は容易だが、実際にはその様にはなっていない。動的リンクライブラリやシンボリックリンク、ハードリンクにより異なった名前で発生するアクセス、あるいはCGIやperl, ruby等で使用するモジュール(およびそれらが必要とするアクセス許可)

を全て列挙することの困難さを想像すれば理解できるであろう。それを全てのアプリケーションに対して列挙しなければアクセスポリシーを策定できないのである。

筆者らはこうした課題に着目し、「プロセス実行履歴に基づくアクセスポリシー自動生成システム」[7]の技術を開発している。これは発生したアクセス要求をカーネル内で捕捉、記録することにより、許可したい操作を一通り実行するだけで、必要なアクセス許可の90%程度を定義することができるものである。TOMOYO Linuxはこの技術を発展させ、OS自体を「学習」モードで起動することにより、必要なアクセス許可を学習、結果をアクセスポリシーの原案として提供することができる。管理者は提供されたアクセスポリシーを確認、編集することにより無駄なく、もれのないアクセスポリシーを策定できる。アクセスポリシーの生成を支援する「アクセスポリシー自動学習モード」と、そのアクセスポリシーに基づいて実際にアクセス制御を行う「強制アクセス制御モード」の両方の機能を備えている点がTOMOYO Linuxの最大の特長である。

また、独自の「ドメイン」分割手法により、同一のプログラムに対しても状況に応じた必要最小限のアクセス許可を定義することができる。

2.2. タスク構造体の拡張による強制アクセス制御

Linuxではfork(), execve()のシステムコールの組み合わせにより、タスク構造体(task_struct)が親プロセスから子プロセスへ引き継がれるように設計、実装されている。筆者らはこの点に注目し、LSMを用いずに独自の強制アクセス制御を実現した。この方式を採用することにより、標準のカーネルに対する差分はごく小規模で、またプロセス実行履歴に基づくアクセス制御が可能となっている。

2.3. ファイルやディレクトリに対するラベル付けが不要

SELinuxを含めほとんどの強制アクセス制御の実装では、まずファイルやディレクトリに「ラベル(識別子)」を付与して、そのラベルに基づきアクセスポリシーを記述するようになっている。例えば/etc/passwdに対するアクセスを制御するためには、/etc/passwdおよびそれにアクセスする/usr/bin/passwd等のコマンドにラベルをつけなければならない。

これには2つの問題がある。ひとつは、ラベル付けの作業による管理運用の負荷と、もうひとつはラベルとファイル名との対応のずれによりアクセス制御が正しく行われない事態が発生し得る点である。TOMOYO Linuxではこれを解決するものとして、「正規化[1]されたファイル名」を「ラベル」の代わりとして利用しており、直感的なアクセス許可の指定が可能である。

2.4. 改ざん防止Linuxと組み合わせが可能

筆者らがLinux Conference 2003で発表した「書き換え不能メディアによる改ざん防止Linux」[6]の技術と組み合わせることにより、プログラムやファイルが改ざんされていないことを物理的にも保証することで、一層のセキュリティ強化を実現できる。

2.5. プロセスが自発的に権限を放棄可能

SELinuxを含めたセキュリティ強化Linuxでは、プログラム類は既存のものをそのまま利用し、アクセスポリシーによりその動作を律するという形になっている。TOMOYO Linuxも同様であるが、その他にプロセスが自発的に権限を放棄できる仕組みも備えている。具体的には、「新しいプログラムを実行する権利」や「一度放棄したシステム管理者権限を再び得る権利」等を自ら放棄するようなプログラムを僅かな修正で実現できる。例えばhttpdプロセス自身が新たなプロセス(例えば/bin/sh)に変化する必要がないことが明確であれば、httpdプログラムにexecve()の実行を放棄させるコードを1行追加することにより、httpdの欠陥を狙うシェルコード[2]が実行されてもhttpdプロセスの制御が奪われることを防止できる。そのためのアクセスポリシーは不要である。本機能は、強制アクセス制御で許可されているアクセス許可をさらに制限するために用いることも可能である。

3. 「ドメイン」に関する考察

3.1. ドメインとドメイン遷移図

強制アクセス制御では、全てのプロセスを「ドメイン」に分割し、各ドメインに属するプロセスがアクセス可能な資源を制限する形で実装されている。新しいプログラムを実行する等によりプロセスの状態が変化するが、その時にアクセスポリシーに記述された条件を満たすとそのプロセスは異なるドメインへ遷移する。遷移先を正しく制限することにより、正しいシステムの動作とセキュリティの確保が保証される。

ドメインの遷移を視覚的に把握するために、ドメイン遷移図と呼ばれるものが存在する。このドメイン遷移図の

---

[1] 本論文中での「正規化」とは、パス名に含まれるシンボリックリンクや「..」等をすべて解決し、/sbin/initプロセスの「/」ディレクトリから見た絶対パスに変換することを指す。対象プロセスの「/」ディレクトリから見た絶対パスでは無い。

[2] プログラムの脆弱性を突いてプロセスの制御を奪うためにクラッカーが使用する攻撃プログラム。

理解しやすさは、ドメイン遷移を正しく把握してアクセスポリシーを記述することができるかどうかに直結する。

以下に、3種類のドメイン遷移図を示す。

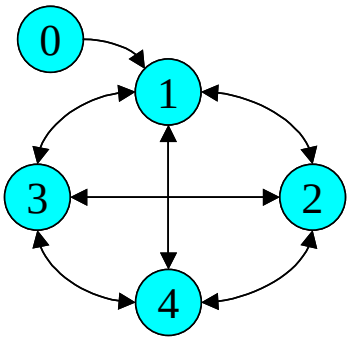

図1　ドメイン遷移の制限が無い場合のドメイン遷移

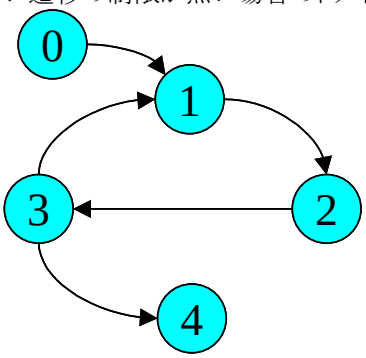

図2　現在のドメインだけを考慮したドメイン遷移

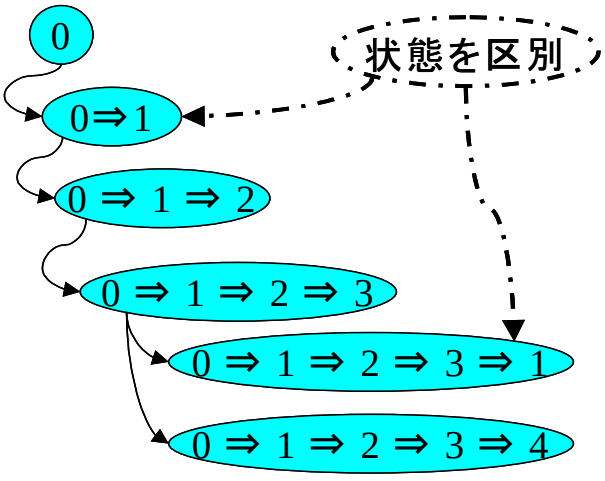

図3　過去のドメインも考慮したドメイン遷移

強制アクセス制御を導入することで、ドメインの遷移先を制限することができる。つまり、強制アクセス制御が導入される前は図1のようにプロセスは好きなドメインに自由に遷移することが可能であるのに対し、導入後は図2または図3のように遷移先が制限される。では、図2と図3とではどちらが理解しやすいか考えてみよう。

図2は、ドメインの数を少なくすることができるが、ドメイン遷移が無限ループになる場合がある。ドメインの数が少なければ図2は便利であるが、ドメインの数が増えると途端にドメイン遷移の流れを追いきれなくなる。また、ドメインを追加や削除したときの影響範囲が把握しづらいという問題も発生する。

図3は、全てのドメインをそれぞれ異なるドメインとして扱うので、ドメインの数は最も多くなる。しかし、ツリー構造で扱えることから、どんなにドメインの数が多くても流れを追うことが可能であり、ドメインを追加や削除したときの影響範囲も明白なので理解しやすい。図3においては、全てのドメインを記述する手間がかかるという問題が発生する。しかし、機械的にドメインを割り当てることが可能であるため、自動学習機能を使用することで記述の手間は大幅に削減できる。

3.2.　SELinuxにおけるドメイン

SELinuxにおいては、図2の方式のドメイン管理を行っている。そのため、細かく制御しようと思ってドメインを細かく分割すると、ドメイン遷移を視覚的に把握することが難しくなる。また、「ドメイン」や「ラベル」に使用することができる文字が限られているため、ファイル名をそのままドメイン名やラベル名に割り当てることができず、結果として「ドメイン名とプログラム名の対応」や「ラベルとファイル名の対応」が判りにくい場合がある。

3.3.　SubDomainにおけるドメイン

SubDomain[8]とは、Immunix社が開発した、強制アクセス制御を備えたLinuxである。主にファイルやディレクトリへのアクセスとプログラムの実行を制御することができる。SubDomainでは、図3の方式のドメイン管理を行っている。SELinuxほどのきめ細かな制御はできないが、構造が単純であり、SELinuxのように「ラベル」を用いずにファイル名で指定できるため、初心者でも理解しやすい。本論文で紹介するセキュリティ強化LinuxもSubDomainと似た機能を備えている。

3.4.　TOMOYO Linuxにおけるドメイン

TOMOYO Linuxでは、図3の方式のドメイン管理を行っており、ドメイン名にプログラムのファイル名をそのまま利用できる点が特長である。図3の方法により状態遷移を記述するため、視覚的に把握・編集することが容易である。さらに、図2の方法とは異なり、同じプログラムに対して、過去に実行されたプログラムの内容に応じて異なるアクセス許可を与えることができるため、状況に応じて必要最小限のアクセス許可を与えることが可能である。

4.　強制アクセス制御に関する実装内容

筆者らが開発した「読み込み専用マウントによる改ざん防止Linux」[6]では、クラッカーによってシステムを乗

っ取られたとしても、改ざんを防止することができる。しかし、改ざんを防止できたとしても、アクセス制御自体は強化されていないため任意のプログラムを実行したり任意のファイルを参照したりできてしまうという点がセキュリティ上の課題であった。TOMOYO Linuxでは、この手法を併用することで、改ざんを防ぎながら不要不適な資源へのアクセスも制限できる。以下、TOMOYO Linuxで実装した強制アクセス制御の概要について紹介する。

4.1. 正規化されたファイル名によるアクセス制御

強制アクセス制御の実装では、「ラベル」に基づくアクセス制御を行っているものが多い。この「ラベル」とは、ファイル名とは別に、強制アクセス制御を行うためだけに新しく追加された識別子である。しかし、ファイルやディレクトリについては本来「ラベル」を割り当てる必要は無い。何故なら、正規化されたファイル名はシステム上で一意であるため、識別子として利用可能だからである。

ファイル名を「ラベル」として利用することで、アクセスの要求元（プロセス）とアクセスが要求された資源（ファイル）の対応が直感的に把握できる。また、常に正しいラベル付けが行われていることが保証されているため、実装が簡単になる。これらは実際の運用では大きな利点となるであろう。以下、どのようにして正規化されたファイル名を導出しているか簡単に紹介する。

4.1.1. ユーザプログラムから見たファイルの指定方法

ユーザプログラムからは、ファイル名をヌル文字で終わる文字列として扱っている。人間にとって、ファイルを「人間が理解しやすい文字列」により指定できるということは重要である。

4.1.2. Linuxカーネル内部におけるファイルの指定方法

カーネルの内部では、ファイル名をヌル文字で終わる文字列ではなく、nameidataと呼ばれる構造体で扱っている。カーネルにとっては、文字列のままでは扱えないからである。文字列からnameidata 構造体へ変換するのがfs/namei.cの中で定義されている link_path_walk()である。nameidata構造体にはdentry構造体とvfsmount構造体が含まれており、前者にはファイルシステム内でのファイルの情報が、後者にはマウントポイントの情報が格納されている。

4.1.3. 正規化されたファイル名へ変換する方法

nameidata構造体はカーネル内でのファイル名の表現方法であるが、このnameidata構造体からヌル文字で終わる絶対パス名に逆変換するのがfs/dcache.cの中で定義されている__d_path()である。この関数を使うことで、nameidata構造体に変換されたファイル名を、シンボリックリンク等を含まない絶対パスへ変換することができる。

__d_path()は呼び出し元プロセスのルートディレクトリまでしか遡って解決しないため、呼び出し元プロセスがchrootによりルートディレクトリを変更していた場合、正規化されたファイル名を得ることができない。そこで、__d_path()を元に、正規化されたファイル名を返すように修正した特別な関数を用意した。

4.2. ドメインの遷移

このシステムにおいては、全てのプロセスは「ドメイン」に属しており、「ドメイン」毎にアクセスを許可するファイルやディレクトリを指定する。ただし、このシステムで使用している「ドメイン」の定義は、SELinuxでの「ドメイン」の定義とは異なるものである。

最初のドメインはカーネルプロセスであり、「<kernel>」として表現される。/sbin/initはカーネルによって実行されるので、/sbin/initのドメインは「<kernel> /sbin/init」として表現される。/sbin/initによって実行される/etc/rc.d/rcのドメインは「<kernel> /sbin/init /etc/rc.d/rc」として表現される。

このように、現在のプロセスに至るまでに実行された全てのプログラムの情報（パス名）が含まれているため、同じプログラムでもそれまでに実行されたプログラムが異なっていれば異なるドメインとして定義される。これにより、同じプログラムに対して、状況に応じた必要最小限のアクセス許可を与えることが可能になる。さらに、ドメイン遷移をツリー構造で表現できるため、ドメイン遷移の編集が容易になる。

カーネル内にはドメイン名の一覧を保持するドメインテーブルが存在し、ドメイン名とドメイン番号の対応付けを扱っている。プロセスから見ると、ドメインの遷移はドメイン番号が変化することである。

4.3. プログラムの実行に対するアクセス制御

プログラムの実行時には、実行権限のチェックを行う。

ファイル	関数名	権限のチェック位置
fs/exec.c	do_execve()	open_exec()

do_execve()の中では、以下の処理を行っている。

(1) 指定されたプログラム名を正規化されたファイル名に変換する。

(2) アクセスポリシー自動学習モードでは、(1)で取得したファイル名の実行許可をアクセスポリシーに追加する。強制アクセス制御モードでは、(1)で取得したファイル名の実行許可が与えられているかどうかアクセスポリシーを検査し、許可が与えられていなければエラーを返す。

(3) 現在のドメイン名に(1)で取得したファイル名を追加したものを新しいドメイン名とする。

(4) アクセスポリシー自動学習モードでは、新しいドメイン名をドメインテーブルの一覧に追加する。強制アクセス制御モードでは、新しいドメイン名をドメインテーブルの一覧から検索し、見つからない場合はエラーを返す。

(5) 通常のdo_execve()の処理を継続し、処理が正常終了した場合にはプロセスは新しいドメインに遷移する。do_execve()の処理では(1)で取得したファイル名を使用しない。その訳は、実行時の名前によって動作が異なるプログラムが存在するからである。例えば、/sbin/pidofは/sbin/killall5へのシンボリックリンクであるが、/sbin/pidofとして実行した場合と/sbin/killall5として実行された場合とでは動作が異なる。

4.4. 読み込みアクセスに対するアクセス制御

読み込みモードでのファイルのオープン時と共有ライブラリを読み込む時に、読み込み権限のチェックを行う。

ファイル	関数名	権限のチェック位置
fs/open.c	filp_open()	dentry_open()
fs/exec.c	sys_uselib()	read_lock()

filp_open()の中では、以下の処理を行っている。

(1) open_namei()を呼び出して指定されたファイル名をnameidata構造体に変換する。

(2) 指定されたファイル名の正規化されたファイル名を取得する。

(3) アクセスポリシー自動学習モードでは、(2)で取得したファイル名に対する読み込み許可をアクセスポリシーに追加する。強制アクセス制御モードでは、(2)で取得したファイル名に対する読み込み許可が与えられているかどうかアクセスポリシーを検査し、許可が与えられていなければエラーを返す。

(4) dentry_open()を呼び出して(1)で取得したnameidata構造体が示すファイルをオープンする。

sys_uselib()の中では、バイナリの種類を判別する処理を開始する前にチェックを行う。

4.5. 書き込みアクセスに対するアクセス制御

書き込みモードでのアクセス要求時にも、読み込みアクセスに対する制御と同様の処理を行っている。ただし、書き込みアクセスには、書き込みモードでのファイルのオープン以外にも、ファイルの作成、ディレクトリの作成や削除、ハードリンクの作成や削除、名前の変更、シンボリックリンクの作成、ファイル内容の切り詰めなどの操作が含まれる。そのため、書き込み権限のチェックは以下の箇所で処理を行っている。

ファイル	関数名	権限のチェック位置
fs/namei.c	open_namei()	vfs_create() vfs_truncate()
	sys_mknod()	vfs_create() vfs_mknod()
	sys_mkdir()	vfs_mkdir()
	sys_rmdir()	vfs_rmdir()
	sys_unlink()	vfs_unlink()
	sys_symlink()	vfs_symlink()
	sys_link()	vfs_link()
	do_rename()	vfs_rename()
fs/open.c	filp_open()	dentry_open()
	do_sys_truncate()	vfs_truncate()

いずれもvfs_で始まる関数を呼び出す直前にチェックを行うようにしている。その理由は、vfs_で始まる関数はファイルシステムを跨ぐ操作を対象としておらず、vfs_で始まる関数に渡されるパラメータにはマウントポイントの情報（vfsmount構造体）が含まれていない。その結果、vfs_で始まる関数の中からは正規化されたファイル名を取得することができないのである。

なお、上記以外にも、書き込み権限を必要とする操作として、所有者やパーミッションの変更、ファイルの更新時刻の修正等があるが、書き込み権限が1種類しか無いため、ファイルの内容が変化する操作だけをチェック対象とした。

チェックの手順は以下のように行う。

(1) 指定されたファイルをnameidata構造体に変換する。

(2) vfs_で始まる各関数の最初に実行される基本的なエラーチェックを行う。例えば、操作対象がファイルシステムを跨いでいないかのチェックや、実際の処理を行う関数が実装されているかのチェック、および、Linux標準のアクセス制御におけるパーミッションのチェック等を含む。この時点でエラーが発生したら通常のエラー処理を行って終了する。

(3) (1)で取得したnameidata構造体から正規化されたファイル名を取得する。

(4) アクセスポリシー自動学習モードでは、(3)で取得したファイル名に対する書き込み許可をアクセスポリシーに追加する。強制アクセス制御モードでは、(3)で取得したファイル名に対する書き込み許可が与えられているかどうかアクセスポリシーを検査し、許可が与えられていなければ

エラーを返す。

(5) 実際にvfs_で始まる各関数を呼び出す。

4.6. ポリシーの読み込みと保存

アクセスポリシーは/sbin/initが開始される直前にファイルから読み込まれる。また、自動学習モードでは、/etc/rc.d/init.d/haltスクリプトの最後（シャットダウンの直前）でファイルに保存される。自動学習モードでは、任意のタイミングでファイルに保存することが可能である。

4.7. アクセスポリシーの編集方法

このシステムにおけるアクセスポリシーは、空白と改行で区切られたASCIIテキストファイルとして構成されている(図5)ため、アクセス許可の修正はemacs等のテキストエディタを使用して編集することが可能である。また、ドメイン遷移の編集には、コンソール上で動作するCUIエディタ(図4)を使用可能である。ASCII文字コード表で32（SP）以下または127（DEL）以上の文字は「\ooo」という8進数で、「\」自身は「\\」として指定する。例えば、SP文字は「\040」と指定する。

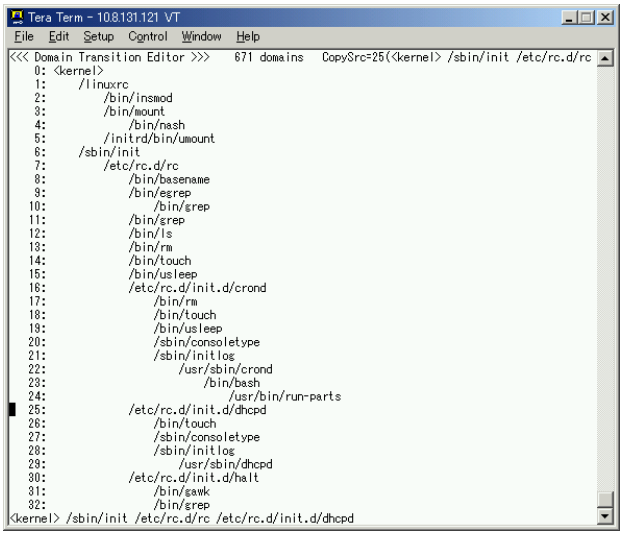

図4 ドメイン遷移の編集画面

図5 アクセスポリシーのサンプル

5. 自発的アクセス制限に関する実装内容

強制アクセス制御の機能とは別に、プロセスが自発的に権限を放棄できる仕組みも備えている。例えば、「新しいプログラムを実行する権利」や「一度放棄したシステム管理者権限を再び得る権利」等を放棄することができる。これにより、アクセスポリシーで許可されているアクセス許可をさらに制限することが可能である。

ここでは、とても簡単な実装でセキュリティを向上させることが可能な方法について紹介する。これらの実装内容の一部は、「読み込み専用マウントによる改ざん防止Linux」[6]にて実装されたものである。

5.1. タスク構造体を利用する理由

Linuxにおいて、全てのプロセスはそれぞれタスク構造体というデータベースを持っている(図6)。

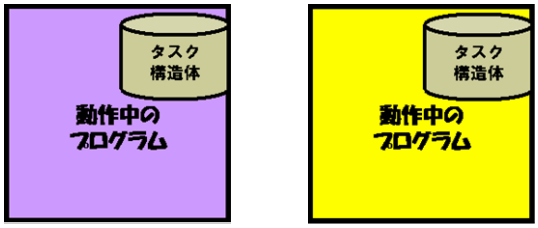

図6　タスク構造体

このデータベースには、プロセスid、プロセスの所有者のユーザid、使用しているメモリの量、プロセスのルートディレクトリ等の情報(図7)が格納されている。

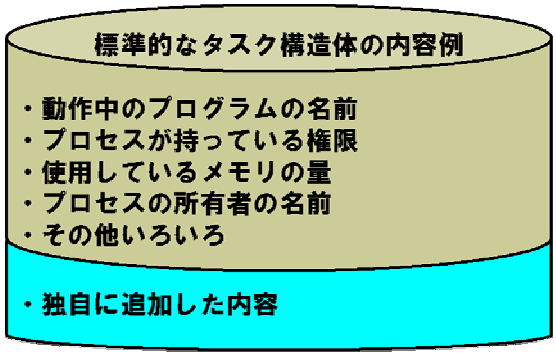

図7　タスク構造体の内容

このデータベースは、親プロセスが子プロセスを作成(kernel/fork.cで定義されているdo_fork()を呼び出し)したときに複製(図8)され、プロセスが新しいプログラムを実行(fs/exec.cで定義されているdo_execve()を呼び出し)した時にその内容の一部が更新(図9)される。

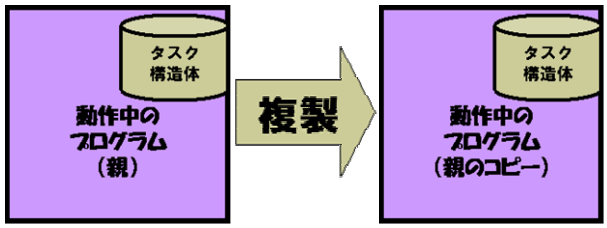

図8　タスク構造体の複製

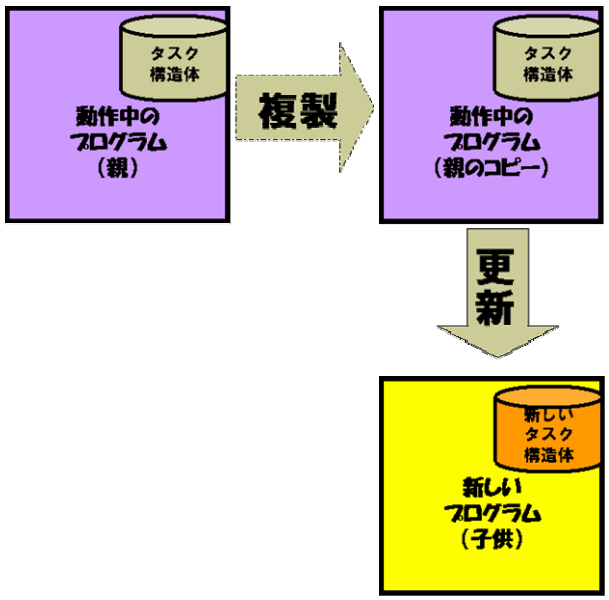

図9　タスク構造体の更新

全てのプロセスは最初に実行されるプログラム/sbin/initの子プロセスとして存在しており、/sbin/initが使用しているタスク構造体の内容の一部を継承することが可能である。

このデータベースに特別な情報を追加すると、追加された情報も子プロセスに引き継がれる。よって、親プロセスで不要なアクセス権限の一覧を記録しておくことで、それ以降に作成される子プロセスに対して親プロセスが宣言した不要なアクセス権限の一覧も引き継ぐことができる。

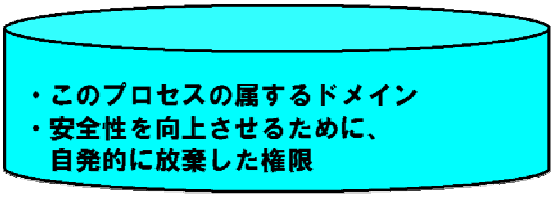

図10　独自に追加した内容

一度権限を放棄すると、再びその権限を取得することはできないように実装している。具体的には、タスク構造体に放棄したアクセス権限を記録する変数を追加(図10)し、権限を放棄可能なそれぞれのシステムコールに対して対応するビットをオンにする。適切な箇所でそのビットがオンかどうかをチェックし、オンになっていた場合はアクセスを拒否したりプロセスを強制終了させたりする。

「プログラムが必要な権限と不要な権限を1番良く知っているのは誰か？それはカーネルではなく、プログラム自身である。」

この「餅のことは餅屋に聞け」というのがこの自発的アクセス制限の原則である。「要らないものは最初に要らないと宣言してもらう」ことでセキュリティを高めている。これは、全てのアクセス制限をカーネルに行わせる強制アクセス制御とは正反対の手段である。

5.2. do_execve()権限の放棄

プロセスが新しいプログラムの実行を禁止することができる。親プロセスがこの権限を放棄すると、それ以降に作成された子プロセスも新しいプログラムを実行できなくなる。例えば、/bin/lsは他のプログラムを実行する必要は無いであろう。このような場合に、/bin/lsの実行開始直後にdo_execve()を呼び出す権限を放棄することで、仮に/bin/lsに脆弱性があって任意のコードを実行可能だとしても、/bin/shを起動されることが無い。シェルなどの一部のプログラムを除いて、自身の複製である子プロセスを作成（do_fork()を呼び出し）することはあっても、新しいプログラムを実行（do_execve()を呼び出し）する必要は無いであろう。do_execve()を呼び出す権限を放棄することにより、クラッカーが脆弱性を突いてシェルコードを実行させることを困難にできる。

5.3. sys_chroot()権限の放棄

プロセスがルートディレクトリの変更を禁止することができる。動作はdo_execve()権限の放棄と同様である。

プロセス単位での権限放棄とは別に、システム全体でsys_chroot()により移動可能なディレクトリを制限することも可能である。Linuxでは任意のディレクトリにchrootできるが、これはセキュリティ上好ましくない場合がある。「読み込み専用マウントによる改ざん防止Linux」[6]と組み合わせると、書き込み可能なディレクトリへのsys_chroot()の呼び出しを禁止することができ、クラッカーが設置したトロイの木馬を含むディレクトリツリーが使用されることを防止できる。

5.4. sys_pivot_root()権限の放棄

プロセスがルートディレクトリの交換を禁止することができる。動作はdo_execve()権限の放棄と同様である。

プロセス単位での権限放棄とは別に、システム全体でsys_pivot_root()の呼び出しを禁止することも可能である。

5.5. マウント権限の放棄

プロセスがファイルシステムをマウントすることを禁止することができる。動作はdo_execve()権限の放棄と同様である。

プロセス単位での権限放棄とは別に、システム全体でマウント可能なマウントポイントとデバイス名の組合せを制限することも可能である。これにより、書き込み可能なディレクトリツリーを作成するためにクラッカーがtmpfs等をマウントすることを禁止できる。

5.6. システム管理者権限を再取得する権限の放棄

Linuxにおいては、システム管理者権限での作業は必要最小限にすることが推奨されている。これは、システム管理者権限で作業していると、操作を誤った場合の影響が大きいからという理由がある。そのため、通常は一般ユーザ権限で作業を行い、必要なときだけ/bin/suを使ってシステム管理者権限を再取得するという流儀が定着している。

しかし、システム管理者が作業する場合を除いて、プロセスが一度システム管理者権限を放棄した後で再びシステム管理者権限を必要とすることはほとんど無いと考えられる。プロセスが再びシステム管理者権限を得る例としては、プロセスの制御を奪ったクラッカーがシステム管理者権限を得ることを試みた場合が挙げられる。

以上の仮定の下に、一度システム管理者権限を放棄した後に再びシステム管理者権限を得ることを禁止するという実装を行った。

システム管理者権限の再取得が行われたかどうかのチェックはfs/namei.cの中で定義されているlink_path_walk()の中で行っている。本来は権限の変更を制御するシステムコールの中でチェックを行うべきかも知れないが、敢えてlink_path_walk()の中で行うようにした。その理由は、以下の2点による。

- システムコール自身の脆弱性により、予期せぬ箇所でシステム管理者権限への昇格が発生することがある。
- システム管理者権限で動作するシェルを起動させるために、必ずlink_path_walk()が呼ばれる。

厳密にはシステム管理者権限を再取得することを禁止しているわけではない。しかし、ファイルを扱う殆どの操作においてlink_path_walk()が呼び出されるため、どこでシステム管理者権限を再取得したとしても、それを

直ちに検知することができる。

システム管理者権限の再取得を禁止されたプロセスが再びシステム管理者権限を得たことをlink_path_walk()の中で検知した場合、そのプロセスを直ちに強制終了させる。

この権限の放棄はシステム管理者権限を保持している間に行うこともでき、その場合はシステム管理者権限を放棄した後にlink_path_walk()が呼び出された時点でシステム管理者権限を再取得する権限の放棄が有効になる。

5.7. その他の放棄可能な権限の例

現時点ではファイルシステムに関する権限放棄しか実装していないが、その他の権限についても同様の方法で放棄することが可能である。例えば、TCP/IPを使用する権限を放棄したり、子プロセスを作成する権限を放棄したりするという実装が可能である。

また、権限を放棄するのとは逆に、システム管理者権限の一部だけを引き継がせるという応用も可能である。例えば、システム管理者権限で動作していないプロセスでもタスク構造体に「TCPポート80の利用を許可」という情報を持っていれば「TCPポート80を利用」できるようにカーネルを修正する。システム管理者権限で動作しているプロセスが自身のタスク構造体に「TCPポート80の利用を許可」という情報を登録した後、非システム管理者権限で動作するApacheを子プロセスとして実行させることで、非システム管理者権限で動作しているApacheが「TCPポート80を利用」できる。

5.8. 課題

これらの自発的権限放棄のための正式なインタフェースが存在しないことが課題である。現在は、引数として可変個のパラメータを渡すのに便利であるという理由からdo_execve()に特殊なキーワードを渡した時に権限放棄処理が行われるようになっている。

これらの機能を利用するためのソースコードの修正は僅かである。なぜなら、権限放棄処理を行う関数(この実装ではdo_execve())に、放棄したい権限とプロセスidを渡すだけなので、数行を追加するだけで利用できる。

「どのような権限を放棄できるようにするべきか」「どのようなインタフェースを用意するのが適切か」の議論は専門家にお任せしたい。本章で説明した内容は、強制アクセス制御とは別に、タスク構造体を使うことで簡単にセキュリティを強化させられるのではないかという筆者らの提案である。

6. 本システムで動作可能なアプリケーション例

「読み込み専用マウントによる改ざん防止Linux」[6]と本論文で紹介した強制アクセス制御機能を持つカーネルを組み合わせることで、以下のようなアプリケーションが読み込み専用メディアで動作可能であることを確認済みである。デモ環境として、これらの全てのアプリケーションを容量1GBのUSBフラッシュメモリの中に組み込んで運用している。

- WWW Server (httpd-2.0.40-21.11)
- LXR + glimpse (lxr-0.3.1 + glimpse-4.17.4)
- Another HTML-Lint
- WWW Server (Tomcat 4.1.30 + Java 1.4.2_04)
- FTP Server (vsftpd-1.1.3-8)
- DNS Server (bind-9.2.1-16 + caching-nameserver-7.2-7)
- DHCP Server (dhcp-3.0pl1-23)
- SAMBA Server (samba-2.2.7a-8.9.0)
- F-Secure AntiVirus for SAMBA (fsavsamba-4.51-04011901)
- Mail Server (sendmail-8.12.8-9.90)
- OpenSSH Server (openssh-server-3.5p1-11)
- Text Editor (emacs-21.2-33)

必要なアクセス許可の殆どは自動学習機能により定義することが可能である。読み込み専用メディアで運用することに向かないアプリケーションは存在するかもしれないが、TOMOYO Linux自体はどんなアプリケーションにも対応できる。

7. おわりに

セキュリティ強化Linuxの導入が進み、近い将来には強制アクセス制御を活用するのが当たり前になると考えられる。しかし、強制アクセス制御のためのアクセスポリシーを定義する作業は大変な負担であり、万人が利用できるものとは言い難い。

本論文で紹介したファイル名を使用した自動学習機能付きの強制アクセス制御システムでは、共有ライブラリや設定ファイル等に対して必要なアクセス許可を把握するという大変面倒な作業を自動化させることができ、導入も運用も容易である。

また、プログラムのソースコードに数行を加えて再コンパイルするだけで、強制アクセス制御によらないセキュリティ強化も可能であると考えている。

本論文で紹介した強制アクセス制御機能は、制御できる対象がファイルやディレクトリに限定されているが、単純明快な方法で強制アクセス制御を実装できることの一例として参考にしていただければ幸いである。

謝辞 プロトタイプ構築にいたるまで試行錯誤を繰り返しながら辛抱強く開発を支援いただいたNTTデータカスタマサービス半田哲夫氏に感謝します。

参考文献

[1] Peter A. Loscocco et al, *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computer Environments*

[2] 原田季栄「セキュアなシステムを作る」
日経システム構築2004年4月号 no.132「解説」

[3] P. Loscocco and S. Smalley. *Integrating Flexible Support for Security Policies into the Linux Operating System*. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01), June 2001.

[4] National Security Agency, *Security-Enhanced Linux*, http://www.nsa.gov/selinux/

[5] Stephen Smalley et al. *Implementing SELinux as a Linux Security Module*

[6] 原田季栄、保理江高志、田中一男「読み込み専用マウントによる改ざん防止Linuxサーバの構築」Linux Conference 2003

[7] 原田季栄、保理江高志、田中一男「プロセス実行履歴に基づくアクセスポリシー自動生成システム」Network Security Forum 2003

[8] Crispin Cowan et al, *SubDomain: Parsimonious Server Security*, 14th USENIX Systems Administration Conference (LISA 2000), December 2000.