Compartmented mode operating system

Yoshihiro Satoh†, Katsuyuki Yumoto†, Kazuya Iwasaki†,
Mike Wray‡, Richard Stock‡, Nigel Edwards‡

Hewlett Packard Japan†
Bristol Laboratory Hewlett Packard‡

ＯＳ（オペレーティングシステム）のデータセキュリティを強化するためのアーキテクチャとして、代表的なラベリング方式の実用面での問題点を考察し、それを解決するための新たな方式の検討を行なった。また、その方式を実際にＬｉｎｕｘ上に実装して実現性を確認した。さらに、実用面での検証を行い今後の課題を検討した。以上について報告する。

ＯＳの脆弱性

現在、情報システムを安全に保つ解決策としては、正しい設定を施した後、ＯＳやアプリケーションなどのソフトには、適時に最新のパッチをインストールすることで、常に、環境を安全な状態に自身で保つことが基本となっている。
これは技術的に正論であるが、大規模企業などでシステムの運用体制がある場合にはよいが、比較的小規模な企業や家庭内の環境、すなわちＳＯＨＯでは、大規模企業における場合と同等の適時性を維持できるとは限らない。そして、インターネットに接続する環境では、その遅延は安全を著しく脅かすことになる。
クライアント用ＰＣでは、ソフトのアップデートを自動的に行なう機能等も設けられ、よい方向に向かってはいる。しかし、これからは、ＳＯＨＯでもサーバを設置することも考えられよう。基本ソフト以外のものを組み合わせるサーバ環境での解決策は、技術を熟知した管理者のいないＳＯＨＯに対して現実的なものが示されているようには思われない。
これは、そもそも現在広く使われているＯＳが、ネットワーク接続に対して十分に安全ではないことを意味している。
もともと、ＯＳはアクセスを完全に拒否することに対しては強度を保つように考えられているが、ある程度のアクセスを許容しつつも、その安全を保つことに対して脆弱であると言える。さらには、そのアクセスの主体が、守りたい客体と同一の管理下にない不特定多数になると、非常に脆弱になると言える。
ＯＳの安全が脅かされる事態は、そのデータセキュリティを破られることから始まることが多い。そのため、ＯＳのデータセキュリティを強化することが必要である。

ラベリング方式による解決策

ＯＳのデータセキュリティを強化するには、1980年代に米国防総省が調達基準として制定したTCSEC[1] (Trusted Computing System Evaluation Criteria,通称：オレンジブック)による、BLS (Bレベルセキュリティ) がよく知られている。TCSECは、その後、CC (Common Criteria) となり、ISO/IEC 15408 (JIS X 5070) にも採用されている。BLS では、データセキュリティの強化を図るために、MAC (Mandatory Access Control,強制アクセス制御) という考え方に基づき、その実装方式として、ラベリングという手法を定めている。主体となるユーザやプログラムには、センシティビティレベルを、客体となるデータ等には、センシティビティラベルをラベリングすることで、レベルとラベルを突き合わせて、アクセス制御を行なう。ここではこれをラベリング方式と呼ぶことにする。いくつかの実装手法があるが、基本的な考え方は概ね同じである。[2]

ラベリング方式による問題点

ラベリングによるデータ保護は、それが軍用に耐えうることからも厳格で強力であることがうかがい知れるであろう。しかし、ディスク構造を標準のものと異なるものにするか追加の情報領域を管理しなければならない点と、ＯＳ内のファイル入出力処理を大きく変更する必要があるため、標準のＯＳとは異なるものとして供給する必要がある。このことは、ＯＳのセキュリティが強化される反面、ＯＳの安定性を保つためには、追加の品質保証が求められることを意味する。

当社は、ラベリング方式による BLS のＯＳの提供を長年数多く行なってきた。特に、インターネットバンキングを世界で最初に開始した 1995 年には、それまで軍用に供していた ラベリング方式を始めて商用に供した。[3]

日本における BLS のＯＳの導入は、当社のワールドワイドのビジネスにおいても大きな割合を占めており、国内有数の金融機関・通信業界他のインターネットサービスでのＷｅｂサーバのプラットフォームに用いられ、そのセキュリティを守っている。
それらのシステム構築を通じて、ラベリング方式の機能のうち、実際に利用される部分と、実際には最終的に利用されない部分があることをわれわれは経験した。
利用されない機能は、それがセキュリティの強度を高めることに有用であることは理解されつつも、その運用のコストが増大することや、市販アプリケーションソフトの使用に制限を与えるなどの理由により利用されない。つまり、そもそものサービス提供に供することのできる資源とのバランスで考えた場合に、過剰なセキュリティであるとみなされるのである。
軍用ではセキュリティ強度が優先され必須とされている機能が、商用では運用段階では結果的に現場から採用されないということを経験したのである。
仮に使われなくても、あっても邪魔にならない機能ではないかと思われるかもしれない。しかし、これらの機能を実現すると、標準のＯＳとの差分が多く、先述のように安定したＯＳを供給するには、品質保証などの工数を増大させても、なお、標準のＯＳに比べて品質が劣るか、あるいは、セキュリティ以外の最新の機能への追従が遅れることになる。

求められる機能

そこで、われわれは、実際には利用されない機能を省くことで、標準ＯＳとの差がより少なく、市販アプリケーションソフトの使用に制限を与えないような方式の開発に着手した。
以下に、そのモデルを説明する。理解を容易にするために、われわれが Linux 上に設計した実装を例に紹介するが、特にその実装に限定するものではない。

コンパートメントガード方式

アーキテクチャの基本となるのは、コンパートメントという考え方である。ＯＳの上に、同じリソースを共有できるコンパートメント（小部屋）を用意して、その中に、主体と客体を位置付ける。コンパートメント内の、及び、コンパートメント間のアクセスを閉じ込めるような形で制御する。この「閉じ込め」をコンテインメントと呼ぶ。コンテインメントは、２つの種類の資源を独立して制御できる実装が実用的であることが、ラベリング方式のシステム構築の経験からわかっている。

コンテインメント１：コミュニケーションフロー制御

コンテインメントの１種類目は、コンパートメント間の通信である。たとえば、Ｗｅｂサーバ・システムを考えることにする。そのシステムにおいて、Ｗｅｂサーバ・プロセス（デーモン・プロセス）を『ＷＥＢ』というコンパートメントに位置付けたとすると、その状態では、Ｗｅｂプロセスは、コンパートメント『ＷＥＢ』（以下、コンパートメント名を示すために『コンパートメント名』という表記を用いる）の中に閉じ込められ、外界からは隔離される。Ｗｅｂプロセスが、ＴＣＰ／ＩＰの８０番ポートに listen をしたとしても、そこには、『ＷＥＢ』に位置付けられたもの以外は、到達できない。Ｗｅｂサーバとしてサービスを行なうには、インターネットに接続するネットワーク・インターフェースからのＴＣＰ／ＩＰ８０番の通信を『ＷＥＢ』に許可するように、コミュニケーションフロー制御を設定する必要がある。
各プロセスが、どのコンパートメントに位置付けられるかは、親となるプロセスから継承される。したがって、コンパートメントの中から、外に出て行くことはできない。
最初に、どこかのコンパートメントに入るには、プロセス内の特権を有す必要がある。この特権は、init プロセスによってのみ与えられ、一度手放したら、再び獲得することはできない。
これらのモデルにより、コミュニケーションフロー制御によって許可された方法によってだけコンパートメントの外界と通信をするように、コンパートメントは閉じ込められる。
ＯＳそのものは、標準で『ＳＹＳＴＥＭ』というコンパートメント内に位置付ける。
先の例の『ＷＥＢ』では、例えば、その中で ps コマンドを実行しても、そこに位置付けられている httpd を確認できるが、init プロセスを含むあらゆるＯＳプロセスを確認することができない。すなわち、『ＷＥＢ』の内側から『ＷＥＢ』の外界を認識することができない。これは、アクセスが禁止されているというのではなく、存在を認識できないものとなる。
この部分は、ラベリング方式のそれと同等の強度を示すことになる。

コンテインメント２：ファイルアクセス制御

コンテインメントの２種類目は、ファイルシステ

ムに対するアクセスである。各コンパートメントに位置付けられたプロセスは、ファイルシステムに対するアクセスを制限される。
先例の『ＷＥＢ』であれば、httpd の動作に必要な設定ファイルと、ＨＴＭＬドキュメントのファイルを読み取り専用とし、アクセス・ログファイルを書き出すということだけが許可される。この状態では、『ＷＥＢ』に位置付けられたプロセスは、許可されたこと以外のファイルシステムへの操作をＯＳにより拒否される。たとえば、それらのファイル以外の参照や書き込みはできない。また、ＨＴＭＬドキュメントが読み取り専用であるため、それを『ＷＥＢ』の内側で書き換えることはできない。このことは、対象とするファイルのパーミッションビット（chmod コマンドで設定されるファイル保護）よりも先に評価されるため、仮に、それらのファイルのパーミッションがオープン（chmod 777）であったとしても書き換えられない。そのため、仮に、httpd プロセスが、バッファ・オーバ・ラン[4]などで制御を奪取されても、ホームページを改変されることはなくなる。
この挙動も、ラベリング方式のそれに等しい。異なるのは、われわれのアーキテクチャでは、ラベリングを用いないため、ファイルアクセスの制御は、客体であるファイルのラベルではなく、ファイルのパス名に関連付けられる点である。したがって、パスの変更を行なう場合には、コンテインメントのファイルアクセス制御の設定に反映させる必要がある。ラベリング方式では、その種の設定反映を必要としないが、この種の設定変更は、運用上受け入れられる範囲と考えられる。

これら２つのコンテインメント機能は、ユーザ ID=0（root ユーザ権限）の評価よりも優先させる。そのため、その種のユーザであっても、上記のコンテインメントによる制御を回避することはできない。これらの制御を回避できるのは、init プロセスにより特権を与えられるものだけである。
コンテインメント機能により、root ユーザであっても、ハードディスク上のファイルを改変できないという状態をＯＳだけで作り出すことができる。これは、BLS の最少特権（least privilege）機能よりも精度は荒いが、サーバ・システムで、外部から root アカウントとしてのプロセスの制御を奪取された場合の被害の最小化としては十分な強度を期待できる。

ラベリング方式よりも劣っている部分

客体をラベリングで保護していないため、コンパートメントに閉じ込めていない主体からのアクセスに対しては、標準より以上の保護は行なわれない。
ユーザとしての主体について、レベルを設けておらず、どのデバイスからログインしたかによって追加の特権を付与するため、安全かどうかは、ユーザが誰かよりも、むしろ、ログイン・デバイスに依存する必要がある。
ユーザに対する最少特権機構がない点と、ユーザ管理についてのデュアルロック機構がない点において、システム管理者の悪意に対しては、十分な保護を保てない。ラベリング方式では、それが可能である。

以上の事項は、セキュリティの技術上の観点で劣っているが、これまでのラベリング方式のシステム構築の経験から、これらは商用のサーバ・システムでは運用上必須とはされない場合が多い。特にＳＯＨＯにおいては、問題とならない事項と考えられる。

ラベリング方式よりも優れている部分

標準ＯＳからの変更を最少で済ますことができ、ＯＳのセキュリティ以外の安定性に関する品質保証を共通化することができる。
同様に、標準ＯＳ用のほとんどのパッチをそのまま適用することができる。
コンパートメントは、それぞれに独立しており、相互の関係は、コンテインメントによって自由に決めることができる。ラベリング方式は、ラベル間に上下の関係を持たせ、そのデータフローを一方向に配置する必要がある。
各コンパートメントは、通信とファイルシステムのそれぞれに対して、独立した制御を設定できる。ラベリング方式は、ラベルの下にすべての資源が一意に配備され、資源ごとに異なる制御を設定することができない。
これら２つの独立性により、既存の任意のアプリケーションを保護の対象にすることができる。ラベリング方式では、それらに独立性を持たせられないため、両者共通の制御に緩和せざるを得ない。また、複数のアプリケーションの相関が矛盾した場合には、そのアプリケーションの導入については完全に制御を開放せざるを得なくなり、その部分について BLS の強度は保たれなくなる。

コンパートメントガード方式の特徴

ラベリング方式が客体側において客体へのアクセスを保護するのに対して、コンパートメントガード方式の最大の特徴は、主体側ができることを制限することによって、結果的に客体を保護する点にある。

そのため、主体であるプログラムの誤動作や、その制御が奪取された場合にも、その主体から守られるべき客体を保護することができる。

コンパートメントガード方式の効用

コンパートメントガード方式のアーキテクチャは、ＯＳのセキュリティを強化することで、システムの安全性を高めるという技術的な効果がある。
その他に、アプリケーション・サービスにおける脆弱性を前提として、その歯止めをＯＳが担うことによる設計上の効果も期待できる。
従来、システム設計において、アプリケーションのデータフローの観点でのセキュリティの要件を明確に示すことには、あまり、工数が割かれているようには思えない。その理由を考えてみると、そのような要件を明確にしても、実際には、それを満たすかどうかは、設計どおりのプログラム開発、コーディングが行なわれるかに依存する場合が多い。その場合には、コーディング上の一般的なセキュリティへの配慮に左右され、実際のコーディングが設計どおりに行なわれたかを検証することが難しい。この問題を解決するには、ISO/IEC 15408 (JIS X 5070)による検証が極めて有用であるが、多くの個別システムにそれを適用するのは、費用面で限界がある。逆に、そこまでの検証がされないなら、セキュリティ要件を明確にする動機が失われるものと考えられる。
われわれのアーキテクチャのような機構がＯＳに存在すれば、データフローに関するセキュリティ要件を設計してあれば、それをＯＳに設定することができる。もしも、コーディングが設計どおりでない場合には、プログラムの暴走をＯＳがコンパートメント内に閉じ込めることができ、設計が無駄にならない。
したがって、この種の機構を搭載するＯＳが普及することは、安全性の技術的な向上に加えて、システム構築時のセキュリティ面でのデータフロー設計の動機付けの向上にも役立つものと考えている。

今後の改善

コンパートメントガード方式のアーキテクチャについては、Linux での実装を１年間公開し、実用面でのアーキテクチャの検証を行なった。そこで得られた情報を加味して、実装を改善している最中である。
たとえば、当初のコミュニケーションフロー制御においては、初期状態は完全な遮蔽であるが、コンパートメント内のプロセスが DNS の名前解決を求める場合には、ポート番号としての通信許可設定を必要とする。このような、ＯＳサービスの利用のために行なう許可設定とアプリケーションのための独自設定が、同じ書式で混在するのは紛らわしく、設定ミスによる問題を防ぐために、なんらかの書式の改善などが望ましいと考えている。
ここでのポイントは、セキュリティの解決策には実用面への配慮が重要な点である。いくらセキュリティの強度が上がるとしても、それがあまりに使いにくいものであったり、人の誤操作を招くものであったりすれば、結果的に機能しない。その場合に、それは技術の問題ではない。というような技術は、あっても意味がない。と考えている。
特に、セキュリティについては、最終的に機能するような配慮が必要である。また、技術的に意味があっても、実用面で過剰となるものは、費用を圧迫するだけで、やはり使われない。ＯＳレベルのセキュリティ技術については、特に費用に与える影響が大きいため、実用面で求められるものに限定すべきである。それを知るためには、セキュリティ上のアーキテクティングと、実用面での利用状況の把握を帰納法的に継続して改善していく必要がある。
そのような中から、ラベリング方式以外のセキュリティ・モデルとしてコンパートメントガード方式についても模索していきたい。
現在の実装は、ＬＳＭ[5]によらないが、カーネルの変更部分をより一般化したものとするため、ＬＳＭによる実装も検討したいと考えている。
できあがった実装例については、一部特許を取得しているものの、ソースコードについては、ＧＰＬ[6]に類するものとして広く普及をはかるための準備を進めたいと考えている。

参考文献・用語：

[1] TCSEC (Trusted Computer System Evaluation Criteria) DDS-2600-5502-87
CMWEC (Compartmented Mode Workstation E.C.) DDS-2600-6243-91

[2] http://www.ipa.go.jp/security/fy13/report/secure_os/secure_os.html

[3] http://ntrg.cs.tcd.ie/mepeirce/Project/Press/ca.html
http://www.liuc.it/didattica/econ/program/intra/progetti/g1/about.htm

[4] http://www.ipa.go.jp/security/awareness/vendor/programming/intro.html

[5] http://lsm.immunix.org/

[6] GPL: GNU General Public License