社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2004－ＣＳＥＣ－26　(33)
2004／7／21

マルチタスクOSにおけるメモリの保護
－データ改ざん検出手法の提案－

江頭 徹　稲村 雄　竹下 敦

㈱NTT ドコモ マルチメディア研究所 〒239-8536 神奈川県横須賀市光の丘 3-5

E-mail: {egashira, jane, takeshita}@mml.yrp.nttdocomo.co.jp

あらまし　マルチタスク OS においてプロセスのデータが改ざんされたことを検知する完全性検証機能に関して報告する。本機能により、不正プロセスが権限昇格などを通じて他のプロセスの重要なデータを改ざんすることにより攻撃者が隠れて他人の権限を継続的に利用することを防ぐことができる。本機能の実用化に向けた基本設計案は多数のデータを動的に検証対象として登録することができ、検証に必要となるメタデータについても改ざんを検知できることを特徴とする。基本設計案の１つである遅延検証方式ではさらに、データアクセスがあるまでは検証処理を省略することで完全性検証の処理負荷を削減する。

キーワード　メモリ保護, マルチタスク OS, 完全性検証, コンテクストスイッチ

Protecting Memories in Multitask OSes
－Tampering Detection－

Toru EGASHIRA　Yu INAMURA　and　Atsushi TAKESHITA

Multimedia Labs., NTT DoCoMo, Inc.　3-5, Hikarinooka, Yokosuka, Kanagawa, 239-8536, Japan

E-mail: {egashira, jane, takeshita}@mml.yrp.nttdocomo.co.jp

Abstract　This paper provides the design of an integrity check function, which detects in-process data tampering done by the other processes. The function prevents adversaries from hijacking processes to continuously impersonate a legitimate user and to attempt frauds. For practical applicability, both of two basic designs in this paper allow one to dynamically specify any number of data to be checked, and also to check metadata that are essential for the integrity checking. The lazy-checking method, one of the two designs, can reduce the processing overhead that might be a drawback, by delaying integrity checks until the protected data is actually accessed by the user application.

Keyword　Memory protection, Multitask OS, Integrity check, Context switch

1. はじめに

一般ユーザ権限のプロセスが特権ユーザと同等の権限を奪取する、いわゆる権限昇格を可能とするデーモン等アプリケーションや OS カーネルのバグが次々に発見されている。このような不正な権限昇格が可能であると、攻撃者がシステムをシャットダウンさせたり、ファイルを改ざんするなどしてシステム破壊に繋がる重大な問題が引き起こされる。しかし、そのような攻撃より深刻となりうる問題として筆者らが採り上げるのは、攻撃者がシステムの機能を維持しつつ、隠れて他人の権限を継続的に利用することである。これは例えば他のユーザの秘密鍵を窃取して攻撃者が他人の権限でサービスを利用したり、すでにサーバとユーザ認証がなされているクライアントプロセスの動作に働きかけて、攻撃者の意図に沿うように同プロセスのサービス利用に介入することによって行われる。

このようなりすまし攻撃への対策としては、たとえば秘密鍵の窃取に対抗する手段として、ファイル内に格納される鍵データをユーザのパスフレーズで暗号化する方法が広く用いられている。しかし、ファイル上で暗号化されていても鍵が実際にプログラムにより読み込まれて使われるときにはメモリ上において復号されているため、それをスワップファイルや ptrace システムコール、proc ファイルシステムを通じて読み出される可能性がある。さらに、同様の仕組みを通じて他人のクライアントプロセスのコマンドバッファ等を直接操作するような攻撃に対しては、ファイル上の鍵の暗号化だけでは対応できない。

それらの攻撃に対する対策としてはスワップアウトされたプロセス内データを暗号化することによる秘匿化[1]や、コンテクストスイッチ時に実行中断されたプロセスのデータを暗号化する Cryptographic Memory System (CMS)[2]が提案されている。特に CMS ではスワップファイルだけでなく、攻撃者が ptrace や proc フ

ファイルシステムを通じて他のプロセスのデータを参照もしくは改ざんしようとしても、対象のデータは暗号化されているため、メモリ上の秘密鍵の窃取や意図に沿う動作を行わせることは非常に困難となる。ただし、CMSの課題としては、意図に沿う動作をさせることは困難とするものの、データ改ざん行為により何らかの影響を及ぼすことは可能であった事と、コンテクストスイッチごとにデータの暗号化および復号を行うため、暗号化の対象となるデータの大きさによってはシステムの性能に与える影響が無視できないものとなる事が挙げられる。

本論文ではこのCMSの課題に対処するため、主にUNIXライクのマルチタスクOS環境をターゲットとしてコンテクストスイッチ前後でデータが変化していないか確認する完全性検証機能を導入することを提案し、その機能の概要について論ずる。完全性検証を行うことによりデータが改ざんされたことを検知することができるようになり、また、暗号化より比較的処理の軽いハッシュ関数等により完全性を検証するため、暗号化による秘匿性は必須ではないが改ざんは防ぎたいデータに関しては処理を軽量化することができる。

以下本論文では2章において完全性検証方式について説明し、3章では完全性検証機能の実用化に向けて満たすべき機能要件を提示する。4章と5章でUNIXライクOS向けの完全性検証機能の基本設計案を2つ示し、6章ではCMSとの連携や基本設計2案の昨日要件の適合性などについて考察する。最後に7章で総括する。

2. 完全性検証方式

本論文における完全性検証機能では、あるプロセスに対して割り当てられたタイムスロットが終わってから次にそのプロセスに割り当てられたタイムスロットが始まるまでの間に、そのプロセス内のデータが変化していないことを検証する。

この方式により改ざんを検知できる前提として、プロセスのデータの「改ざん」が行われるのはそのプロセスが実行中断状態(改ざん者プロセスが実行状態)である期間内であることを仮定している。すなわち、プロセスに実行コンテクストがあるときにそのデータが変化することは改ざんとは見なさない。ただし、この仮定で改ざんを検知できない環境が存在することも事実である。例えばptraceシステムコールを用いて他のプロセスのデータを改ざんする場合、4.4BSD-Encumberedなどの UNIX系OSではデータに対する操作が実際に行われるのは操作される側のプロセスに実行コンテクストが移った後であり[3]、前述の仮定によれば改ざんとは見なされない。また、対称型マルチプロセサシステムにおいては、改ざんされる側のプロセスに実行コンテクストがある時に別プロセスがデータを改ざんすることも可能であり、これも前述の仮定にそぐわない。一方で、4.4BSD-Liteから派生したOS[1]やLinuxではptraceでの改ざんやその他procファイルシステムを用いた改ざんも改ざんする側のプロセスの実行コンテクストで行われる。よって、前述の仮定を置くことはこれらのOSを用いている単一プロセサシステムにおいては妥当であると考える。

完全性検証機能では前述のようにデータがある期間をおいて変化していないことを検証する。注目するデータの現在時Tにおける内容がある過去の時刻T_0における内容と等しいことを検証するためには、時刻T_0においてデータをある変換関数を通した出力である「検証子」を安全に保存しておき、時刻Tにおいて同じ変換関数を再びデータに対して適用した結果が保存されている検証子と同一であるか比較すればよい。もし異なる場合は変化したという事であり、同じだった場合は、変換関数固有の確度で変化していないということができる。この確度を実用上十分に高めるためにはこの変換関数が満たすべき性質として、データAの検証子の値と検証子が同じ値となるような入力データBを見出すことが非常に困難であることが必要である。いわゆるセキュアハッシュ関数はこの性質を持つ。その他にもビット列を入力可能な単射であれば十分であり、検証子のデータ長を気にしなくて良いならば例えば入力をそのまま出力する関数でも良い。

完全性検証機能においては、注目するプロセスに割り当てられたタイムスロットが終わり実行中断する際にデータの検証子を生成し保存しておき、次にそのプロセスにタイムスロットが割り当てられ実行再開する際に再びデータの検証子を生成し、保存しておいた検証子と比較することで完全性の検証を行う(図1)。なお、これらの処理を行う完全性検証機能の実体は、プロセススケジューリングのタイミングに密接していることと一連の処理に対する攻撃を防ぐ意味とから、OSカーネルのプロセススケジューリング機能の一部として実装する。

3. 完全性検証機能の機能要件

完全性検証機能の基本原理はこれまで述べてきた通りであるが、それを具体的にシステムに適用し実用化するにあたってはさまざまな設計上の選択肢がある。次節以降において提案する基本設計案は以下に示す機能要件を念頭において検討したものである。

要件1: 検証対象データ(アドレス範囲、期間)を動的に指定できること

検証対象データはプログラム実行以前にアドレスの特定が可能な大域変数や静的変数に置かれるとは限らず、スタック上に置かれる自動変数や、ヒープ上に

[1] {Free, Net, Open}BSDなど。

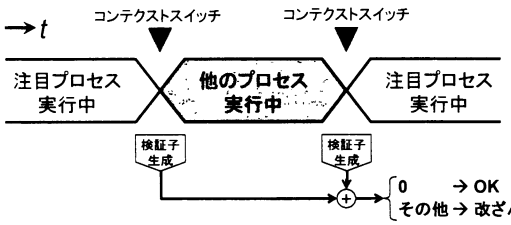

図1 完全性検証の原理

確保されるmalloc等のように、アドレスが実行時にならないと確定しない場合がある。よって、検証対象データのアドレス範囲は実行時に動的に指定できるようになっていなければならない。

また、あるデータを検証対象とすべき期間もプログラム実行全体というような静的なものではないため、動的に指定し得るようになっている必要がある。これは、データのライフサイクルに応じて必要な期間のみ動的に検証対象とすることにより不要な検証処理オーバーヘッドを削減することに繋がるため、後述の要件5の観点からも望ましい。

要件2: 多数のデータを検証対象として指定できること

例えば「プロセスあたり検証対象として指定できるデータ数は最大32個」のような制約があったとすると、仮に完全性検証機能が備わっていたとしても、プログラマーにとって使いづらい。その制約に収めるために無関係のデータを1つの連続した領域に納めて1つのデータとして検証するなどの無駄な労力が必要となり、結果として機能の利用が敬遠されかねない。よって、検証対象として指定できるデータ数は事実上無制限であることが望ましい。

要件3: どのデータが改ざんされたか分かること

これは実際にデータの改ざんがあり、それを検出した場合にどのような対処が可能となるかに関わる。もっとも単純な対処ポリシは、『指定したいずれかのデータが改ざんされた場合、改ざんされたプロセスを強制終了させる』であろう。しかし、データによっては改ざんされたことを検知することさえできれば、そのデータを単に初期化する、もしくは他のデータ(これらは改ざんされてないとして)から値を再計算するだけで安全に実行を継続することができるかもしれない。よって、改ざんを検知したときにどのデータに対して改ざんが起こったか、改ざんされたプロセスが知りうるようにし、その後の対処法を決めさせることが出来るようにすべきであると考える。

要件4: 検証メタデータへの改ざんも検知できること

検証メタデータとは、検証対象データのアドレス範囲や保存された検証子などデータを検証するに当たり必要なメタデータであり、これらに対し改ざんが発生すると、当然のことながら正常な検証を行うことができない。また、検証メタデータへの改ざんが可能であると、アドレス範囲を巨大な領域に設定することでシステムに負荷を与える新たな攻撃手段を用意してしまうことにもなりかねない。よって、検証メタデータの改ざんも防ぐか検知できなければならない。

要件5: システム性能への悪影響をできるだけ小さく

以上の要件を満たしつつも、システム性能への悪影響は小さくとどめたい。すなわち、完全性検証に必要な処理負荷およびメモリ使用量は少ない方が望ましい。

4. 基本設計案1 – 全検証方式

全検証方式ではコンテクストスイッチ時に実行中断されるプロセスの検証対象とする全てのデータについて検証子を作成し、実行再開されるプロセスの検証対象とする全てのデータについて検証する。この方式において用いる検証メタデータの構造についてまず説明し、次に完全性検証機能とユーザアプリケーションの動作について説明する。

4.1. データ構造

プロセスごとに用意される検証メタデータとしては図2に示すような構造を採用する。検証メタデータを構成する単位は検証子構造体配列(VSA: Verifier Structure Array)であり、それは検証子構造体(VS)を要素とする配列である。VSは検証子とAlteredフラグ、Chainフラグ、検証対象データ長、検証対象データポインタのフィールドを持つ構造体である。プロセスがあるデータに対し完全性検証機能を適用することを後述の方法により指示すると、このVSが1つ作成され、そのデータのポインタおよびサイズがそれぞれのフィールドに格納される。そして、そのプロセスが実行中断されるときにそのデータの検証子が生成され、同VSの検証子フィールドに格納される。プロセスが実行再開されるときには同様にデータの検証子が生成されるが、その時はVSには格納せず、検証子フィールドに格納されている検証子と新たに生成した検証子とを比較することで完全性を検証する。そしてもしも2つの検証子が一致しない場合は、Alteredフラグを立てることで改ざんされたことを記録する。なお、検証対象データ長フィールドが0の場合、そのVSは無視される。

検証子とそれを格納する検証子フィールドであるが、検証子の生成にはMD5を用いる。ただし、検証対象データのバイト長が16バイト以下、すなわちMD5の出力より小さいか同じ場合には、検証子として検証対象データをそのまま用いる。検証子が16バイト未満の場合であっても検証子フィールドは16バイト固定としてVS全体を固定長とする。MD5圧縮関数には衝突に関する懸念が指摘されている[4]が、本用途では衝

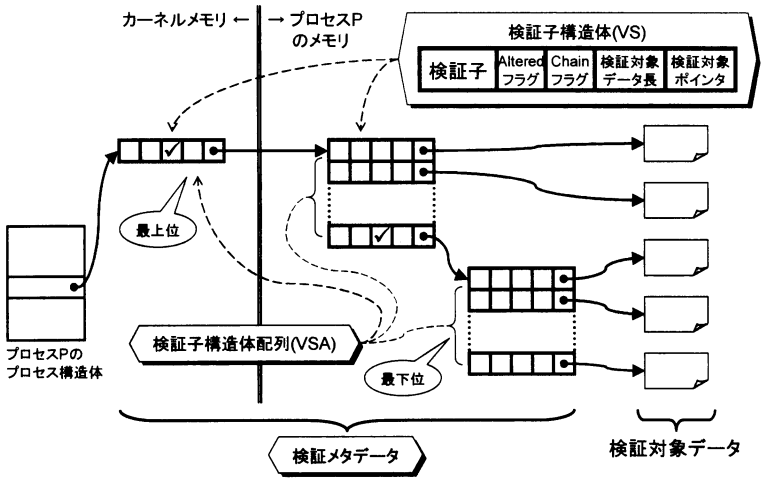

図2 全検証方式のデータ構造

突は問題とはならず、汎用一方向性の確保のみで十分であるため、処理速度の面で SHA-1 や RIPEMD-160 等より有利である MD5 がふさわしい。

Chain フラグの意味であるが、そのフラグの立っている VS は特別な意味を持っており、その検証対象データポインタフィールドが指す先には検証メタデータの一部を成す別の VSA が格納されていると解釈する。そして検証対象データ長フィールドはその VSA 全体のバイト長(配列の要素数と VS のサイズの積)を示すものとする。すなわち、Chain フラグの立っている VS は、VSA 間の連鎖関係を指定する。図 2の例では VSA が合計 3 つ、Chain フラグの作用により連鎖していることを示している。以降、連鎖関係にある 2 つの VSA "A"と"B"があるときに、A の要素の VS の Chain フラグの作用により B が連鎖されるとき、A を B に対し「上位」、B を A に対し「下位」であると言う。なお、Chain フラグによる連鎖により検証メタデータは VSA をノードとする木構造として構成することも可能であるが、筆者らはより単純な単方向リスト構造、すなわち VSA はたかだか 1 つの下位 VSA に連鎖する(たかだか 1 つの要素の VS の Chain フラグが立っている)構造を用いる。このようなリスト構造では、より上位の VSA が存在しない「最上位」VSA およびより下位の VSA が存在しない「最下位」VSA が定義される。

検証メタデータがそのように連鎖する VSA から成る時に、どの VSA の要素から扱うかについては、次のルールに従う:

検証子生成時 最下位の VSA の要素 VS 全て[2]について検証子の生成と VS への格納を行い、次にその上位の VSA について同様に行い、最上位に至るまで順次上位の VSA について同様に行う。

検証時 最上位の VSA の要素 VS 全てについて検証子の生成と VS に格納されている検証子との比較を行い、次にその下位の VSA で完全性が確認されたものについて、同様に行い、最下位に至るかまたは完全性が損なわれたものが現れるまで順次下位の VSA について同様に行う。

このようにすることにより、下位の VSA は上位の VSA により検証が行われるため、たとえ改ざんがあったとしても検知することが出来る。問題は最上位の VSA であり、それについては別の手段を用いて改ざんから護らなければならない。この手段としては、最上位の VSA をカーネルメモリに置き、特権プロセスであっても操作不可能とすることが現実的であると考える。最上位以外の VSA については改ざん検知が出来るためカーネルメモリに置く必要はなく、またカーネルメモリの節約の面からも置くべきでない。よってこれらのデータはプロセス内のヒープに配置する。

検証メタデータの大部分をこのようにユーザランドに置くことはカーネルメモリの節約の他に、システムコールの回数を削減することにも繋がる。すなわち、検証対象データが多数ある時でも、アプリケーションプログラムが直接(またはライブラリを通じて)検証メタデータを作成し、完成したら 1 回だけ最上位 VSA をカーネルメモリに置くためにシステムコールを行えばよい。それ以降、第 2 位(=最上位の下位)の VSA のアドレスや大きさに変化がない限り、VSA やその要素に編集を加えてもシステムコールを行う必要は無い。

検証メタデータを配列を要素とするリスト構造にすることは、検証対象データの登録および削除を動的に行う際にその柔軟性とメモリ効率、さらにアクセス

[2] 検証対象データ長が 0 の VS については前述のように無視する。以降特に断らなくてもそのように扱う。

性能の点で有利なトレードオフであると考える。そのように配列とリスト構造を共用するのではなく、配列のみ、またはリスト構造のみを用いることも考えられる。しかし、配列のみの場合は要素数をいくつにするかが問題となる。それを過大にするとメモリ利用効率が悪くなり、少ないと登録できない場合が発生する。要素数を使い果たしたときには realloc により拡張するという案もあるが、それによりアドレスが変化することが実装方式によっては管理上の問題(ハンドラとしてVSへのポインタが利用できなくなる)となり得る。また、リスト構造のみの場合には、検証メタデータがヒープ領域に広く分散する可能性がある。その場合、コンテクストが移って実行再開されるプロセスについて検証する際には検証メタデータのリストをたどるだけで大量のメモリページにアクセスすることになるため、特に物理メモリが不足気味のシステムにおいてはスワップインを引き起こす可能性が高まり、アクセス性能の面で不利である。一方、配列とリスト構造を併用する筆者らの案では、柔軟性の面では検証対象データを多数追加するときにはリストの末尾に VSA を次々に加えてゆけば対応でき、削除する際には対象のVSの検証対象データ長フィールドを0とし、もしも要素全てが削除されたVSAがある場合には、それをリストから外し、そのメモリ領域を解放することが出来る。さらに、VSAのサイズをメモリページサイズと同等とし単一ページに収まるようにすれば、検証メタデータをたどるだけで発生するスワップインを最小にすることができるため、アクセス性能的にも有利である。

以上のような検証メタデータは各プロセスに固有のものであるが、プロセスとその検証メタデータとの関連付けは図 2に示すようにプロセス構造体から検証メタデータの最上位の VSA にポインタを張ることで行う。この最上位 VSA の要素数は 1 で固定とし、プロセス構造体に含めるのはポインタのみで配列要素数やサイズの情報は含めない。もしこのポインタが null であるときにはそのプロセスには検証メタデータが指定されておらず、完全性検証は行わないことを意味する。

4.2. 動作

完全性検証機能と、それを利用するユーザアプリケーションの動作について説明する。

4.2.1. 完全性検証機能の動作

完全性検証機能側の動作について説明する。完全性検証機能は OS カーネル内部においてプロセススケジューリングと連動して動作し、プロセスのコンテクストスイッチ時に実行状態から実行中断されるプロセスと、実行待機状態から実行再開されるプロセスの2つのプロセスに対し作用する。以下、それぞれのプロセスに対する処理内容を概説する。

実行中断プロセスに対して

実行中断されるプロセスのプロセス構造体より検証メタデータの連鎖をたどり、まず最下位の VSA の要素 VS 全てについて検証対象データの検証子の生成と VS への格納を行う。次にその上位の VSA について同様に行い、最上位に至るまで順次上位の VSA について同様に行う。ただし、Altered フラグが立っている VS については無視し、Chain フラグと Altered フラグが両方立っている場合には、その VS で連鎖される下位の VSA から最下位のそれに至る全ての VSA について無視する。これは、下位の VSA が改ざんされているおそれがあるため、検証対象データ長を過大に設定し検証子生成に大量の時間を要するようにするなどの一種の DoS 攻撃を回避するためである。

実行再開プロセスに対して

実行再開されるプロセスのプロセス構造体より検証メタデータの連鎖をたどり、まず最上位の VSA の要素VS全てについて検証子の生成とVSに格納されている検証子との比較を行い、一致しない場合はその VS の Altered フラグを立てる。次にその下位の VSA について、同様に行い、最下位に至るまで順次下位の VSA について同様に行う。ただし、すでに Altered フラグが立っている VS については無視し、Chain フラグと Altered フラグが両方立っている場合には、その VS で連鎖される下位の VSA から最下位のそれに至る全ての VSA について無視する。

無視されない全ての VS を処理した後、それまでの一連の処理において新たに Altered フラグを立てた VS が存在する場合には、実行再開するプロセスに対しシグナルを送る。

4.2.2. ユーザアプリケーションの動作

一方のユーザアプリケーション側の動作について説明する。ユーザアプリは検証メタデータをヒープ上に作成する。ただし、最上位 VSA は 1 要素とし、それをシステムコールによりカーネルに登録する(カーネル内にコピーが作成される)。その時より完全性検証機能は動作する。完全性検証機能が動作している間も検証メタデータを変更することは可能であるが、ただし、最上位 VSA の内容を変更する場合にはそれのカーネルへの登録を再度行う必要がある。

完全性検証機能が改ざんを検知したときにはユーザアプリに対しシグナルが送られる。その時の処理はユーザアプリのポリシ次第であるが、改ざんされたデータにより処理を分岐する必要がある場合にはユーザアプリは検証メタデータを走査し、Altered フラグの立っている VS を特定することにより、改ざんされたデータを特定する。検証メタデータを走査する時には、第2位以下の VSA についてはヒープ上にあり直接アクセスできるが、最上位についてはカーネル内にあるコ

ピーが最新のものであり、それをアクセスするためにはシステムコールを用いる必要がある。ユーザアプリが検証メタデータを走査している時にもしも検証メタデータの一部が改ざんされたことが分かった場合、改ざんされたVSAを含め最下位までのVSAについては無視する。ただし、検証メタデータの一部が改ざんされた場合にはそれ以降の安全な実行継続は難しいと考えられるので、ユーザアプリはその場合は即時異常終了する事が適当であろう。

5. 基本設計案2 – 遅延検証方式

前述の全検証方式には大きく2つの無駄な処理があり、機能要件5の観点から改良の余地がある。その2つとは、

1. **参照していないデータについても検証している**
   改ざんされると不都合であるデータも、実際にユーザアプリによりそのデータが利用されない限りにおいて改ざんの影響はユーザアプリに及ばない。よって、実際に検証対象データがユーザアプリにより利用される直前まで完全性検証の時期は遅らせることができる。しかし、全検証方式では参照されるかどうかに関わらず、コンテクストスイッチの度に検証を行っている。
2. **変化していないデータの検証子も更新している**
   検証メタデータに格納する検証子は、ユーザアプリがメモリ上に書き込んだ正統な値である検証対象データの検証子であり、すなわちユーザアプリが検証対象データを変更しない限りは(検証対象データが改ざんされても)不変であるはずである。よって、ユーザアプリが検証対象データを変更するまでは検証メタデータに格納された検証子を更新する必要はない。しかし、全検証方式では検証対象データを変更したかどうかに関わらず、コンテクストスイッチの度に検証子を更新している。

遅延検証方式ではこれらの無駄な処理を削減する。具体的にはページフォールトを用いてユーザアプリのデータへのアクセスを監視し、必要な処理のみを行うようにする。

5.1. データ構造

遅延検証方式の検証メタデータの構造は全検証方式のそれとほとんど同じであり、図示は省略する。唯一の変更点はVSのメンバーにStaleフラグが追加されることである。Staleフラグは後述のようにユーザアプリにより検証対象データまたはその付近がアクセスされると立てられるフラグであり、それが立っている場合、そのVSは格納する検証子が古くて更新する必要があることを示している。

5.2. 動作

5.2.1. 完全性検証機能の動作

全検証方式と同様に、コンテクストスイッチ時に完全性検証機能が実行状態から実行中断されるプロセスと、実行待機状態から実行再開されるプロセスの2つのプロセスに対して行う処理内容について概説し、次に、本遅延検証方式に特徴的な処理であるページフォールトが発生したときの処理内容について概説する。

実行中断プロセスに対して

実行中断されるプロセスのプロセス構造体より検証メタデータの連鎖をたどり、まず最下位のVSAの要素VSのうちStaleフラグまたはChainフラグが立っているVS全てについて検証対象データの検証子の生成とVSへの格納を行い、Staleフラグを落とす。次にその上位のVSAについて同様に行い、最上位に至るまで順次上位のVSAについて同様に行う。ただし、Alteredフラグが立っているVSについては無視し、Chainフラグと Alteredフラグが両方立っている場合には、そのVSで連鎖される下位のVSAから最下位のそれに至る全てのVSAについて無視する。

Chainフラグが立っているVSについてはStaleフラグとは無関係に毎回検証子を更新することになるが、その理由については後述する。

実行再開プロセスに対して

実行再開されるプロセスのプロセス構造体より検証メタデータの連鎖をたどり、まず最上位のVSAについて、その要素VSのうちChainフラグが立っているVSについては検証子の生成とVSに格納されている検証子の比較を行い、一致しない場合はその VSのAlteredフラグを立てる。その他の要素全てについては検証対象データを含むページがアクセスされたらページフォールトが発生するようにページテーブルを操作する。次にその下位のVSAについて、同様に行い、最下位に至るまで順次下位のVSAについて同様に行う。ただし、すでにAlteredフラグが立っているVSについては無視し、Chainフラグと Alteredフラグが両方立っている場合には、そのVSで連鎖される下位のVSAから最下位のそれに至る全てのVSAについて無視する。

全検証方式と異なり、この時点では完全性検証を行うのは検証メタデータのみである。検証メタデータに対する検証はすなわち遅延検証ではなく、プロセスが実行再開されるたびに毎回確定的に検証されるが、これは検証メタデータは実行再開の度に必ず完全性検証機能が参照することが分かっているため、遅延検証とする意味が無いためである。前述の実行中断プロセスに対する処理において Chainフラグが立っている VSは毎回検証子を更新するのもほぼ同じ理由による。

検証メタデータ以外の検証対象データについては、

この時点ではページフォールトを起こすように設定するのみであり、検証処理は行わない。実際にそれを行うのはページフォールトが発生したときであり、それについては次に説明する。

ページフォールトが発生したとき

ページフォールトが発生した場合、カーネルはまずそれが完全性検証機能に由来するものであるかどうかを確認する。もしそれがそれ以外の例えばアクセス権違反であったり、スワップイン処理等に関わるものであった場合はそれらの処理を適切に行う。完全性検証機能によるページフォールトの場合には、まずページテーブルを操作し、そのページでページフォールトが発生しないようにする。次に、現在アクティブなプロセスのプロセス構造体より検証メタデータの連鎖をたどり、ページフォールトの発生したページに検証対象データの一部または全てが含まれる VS 全てについて検証子の生成と VS に格納されている検証子との比較を行い、一致しない場合はその VS の Altered フラグを立て、一致する場合はその VS の Stale フラグを立てる。そして新たに Altered フラグを立てた VS がある場合には、現在アクティブなプロセスに対しシグナルを送る。

Stale フラグは本来は検証対象データに対し書き込みが起こったときにのみ立てるべきであるが、あるページに対する初めての読み出しアクセスと初めての書き込みアクセスとをページフォールトの機能を通じてそのどちらが起こったのかを含めて両方とも検知するのは複雑であり、またページフォールトの頻度を増やすことは考察の項で述べるようにシステム負荷の増大に繋がるため、読み出しアクセスの場合であっても Stale フラグを立てるものとする。

5.2.2. ユーザアプリケーションの動作

ユーザアプリケーションの動作については全検証方式の場合とほぼ同等である。一点異なるのは Stale フラグの扱いであり、ユーザアプリが検証メタデータ中の VS の検証対象データポインタまたは検証対象データ長を書き換えた場合には、同時に Stale フラグを立てる必要がある。

6. 考察

6.1. CMS との連携について

完全性検証機能として示した基本設計案は CMS とは独立した機能として完全性検証機能を成立させている。これらを CMS と同時に用いる場合には、いくつかの問題がある。全検証方式の場合は比較的容易であり、次のような処理順序の制約を設けるのみである:

実行中断するプロセスに対して

まず CMS による暗号化を全ての秘匿化対象データに対し行い、それから完全性検証機能が全ての検証対象データの検証子を生成し、検証メタデータを更新する。

実行再開したプロセスに対して

まず完全性検証機能が全ての検証対象データの検証を行い、それから CMS が全ての秘匿化対象データの復号を行う。

この順序で行う必然性は、仮にこの逆で処理を行うと、検証子が平文のヒント情報となりうるからである。特に検証対象データ長が 16 バイト以下の時には平文がそのまま検証子となるため、暗号化を無意味にしてしまう。

一方の遅延検証方式では[2]のままの CMS では復号より検証が後に行われることになるため、上記の順序での処理が出来ない。よって CMS 側の改良や、完全性検証機能側で検証子から平文に関する情報を得にくくしたりするなどの対策が必要である。

6.2. 機能要件への適合について

基本設計案として示した2つの方式が機能要件1〜5を満たしているかどうか検証する。要件1については、両設計案ともに検証メタデータに動的に VS を追加・削除可能であり、また、その VS の指定する検証対象データも動的に変更できるため、要件を満たしている。要件2については、両設計案ともに検証メタデータの大部分をユーザランドに配したため、カーネルメモリの肥大化を危惧することなく事実上無制限の数の検証対象データを指定することができ、満たしている[3]。要件3については、両設計案ともに検証メタデータ中に検証対象データ個々についての検証子を用意しており、検証対象データそれぞれを独立して検証する。よってどの検証対象データに改ざんが起こったかが分かるため、満たしている。要件4については、両設計案ともに検証メタデータは内部的な相互検証による改ざん検出を行っており、改ざん検出できない最上位の VSA についてはカーネルメモリに置くことによる保護を行っているため、満たしている。要件5については、両設計案ともにメモリ使用量については検証メタデータ領域を必要に応じてページ単位で確保するため、ページサイズ以上の無駄な領域は確保しない。処理負荷については、両設計案ともに 16 バイト以下の検証対象データについては実データをそのまま検証子とすることでハッシュ値算出の負荷を軽減している。また、全検証方式に比較して遅延検証方式は不要な検証子生成処理や検証処理を行わないため軽量であると思われるが、これについては次項において考察する。

6.3. 遅延検証方式の隠れた負荷について

遅延検証方式は全検証方式に比べて不要な検証子生成処理や検証処理を行わないため、その点において

[3] ただし、物理メモリや limit、検証子生成処理負荷などによる別の制約は当然受ける。

処理負荷は軽減される。しかし一方で、遅延検証ではページフォールトを用いてユーザアプリのデータアクセスを監視するため、データアクセス時に必要となるカーネルモードへの移行のオーバーヘッドが隠れた負荷となる。よって、プロセスにコンテクストが移るたびに全ての検証対象データが必ずアクセスされるような極端な場合には、検証子生成省略などによる処理負荷軽減効果が全くなくなるため、遅延検証方式の方が確実に処理負荷が大きくなってしまう。

このことから、遅延検証方式が有利なのは検証対象データに対するアクセスの頻度がある程度小さい場合に限られることが分かる。遅延検証方式におけるページフォールトのオーバーヘッドを小さくする案としては、ページのスワップインの際に一般的に用いられているクラスタリングと同様に、ページフォールトが発生したページを含め前後の数ページ分についてもまとめて検証を行うことにより、ページフォールト頻度を抑制する方式が有効であると考えられる。

6.4. ページフォールトを起こす手段について

遅延検証方式ではページフォールトを用いてユーザアプリのデータアクセスを監視するが、そのページフォールトを具体的に発生させる手段としてはアクセス権違反やアドレス変換エラーなどを用いることができる。アクセス権違反は例えばページテーブルエントリを設定することにより対応するページをユーザモードではアクセス禁止となるようにすることにより、また、アドレス変換エラーは例えばページテーブルエントリの Valid(Present)フラグを落とすことにより、実現可能である。これらの手段を用いればユーザアプリのデータアクセスを監視することができるが、しかし、これらの手段はまた別の目的で OS により用いられている事が問題となる。アクセス権違反はデータの保護目的や copy-on-write 等で用いられており、アドレス変換エラーはスワップインのきっかけとして用いられている。よって、発生したページフォールトが完全性検証機能によるものであることが確実に識別できないと、データ保護機能などに悪影響を及ぼしセキュリティを低下させてしまうことになる。完全性検証機能によるページフォールトを確実に識別する手段はしかし、ハードウェアアーキテクチャに依存する部分もあるため一般的な方法として提示できないが、例えば IA-32 アーキテクチャにおけるページテーブルエントリには OS が自由に用いてよいビットが用意されているので[5]、それを用いて目的固有のマーキングを行い、ページフォールトハンドラからそれを確認するなどの方法が考えられる。

6.5. カーネルデータに対する改ざんについて

本論文の完全性検証機能は仮に特権が奪取されたところでカーネルメモリは安全であると仮定し、そこに検証メタデータの一部を置くことで検証メタデータの改ざんを防止、または改ざんの検知ができるようにしている。しかし例えば/dev/kmem を通じてカーネルメモリを直接書き換えできたり、任意のカーネルモジュールを追加できてそれを通じて書き換えできたりすると、これらの仮定は崩れる。よって、これらの機能は特権ユーザからであっても自由に使えないようになっている必要がある。例えば、これらの機能を排除したカーネルを用いたり、Mandatory Access Control 機能により制限する。

また、本論文で想定する特権の奪取は、スーパユーザ権限が奪取されることを意図しているが、より深刻な場合としてカーネル権限が奪取されることも考えられる。これまでにもカーネルメモリを好きなように書き換えることができるとされるカーネルの脆弱性がいくつか報告されており[4]、今後新たに発見されないとも限らない。もしこのようなカーネル権限奪取まで想定して完全性検証機能を構築しようとするならば、検証メタデータはカーネルメモリの保護ではもはや安全ではないため、より高い権限レベルの管理下におかなければならないが、その具体的手段については今後の検討課題である。

7. おわりに

攻撃者が他のプロセス内のデータを改ざんすることにより行うなりすまし攻撃を防止するために、プロセスに実行コンテクストが無い期間にデータが変化が起こったことを検知する完全性検証機能を提案し、その基本設計案を示した。これらの案は多数の検証対象データを動的に指定することが出来るデータ構造と管理方式を用いている。遅延検証方式案ではさらにユーザアプリケーションのアクセスに合わせて検証を行うことで処理を削減できるが、ページフォールトを用いることによるオーバーヘッドとのトレードオフとなることを述べた。両方式案を実装し、さまざまな状況下での処理負荷などを検証することが今後の課題である。

文　献

[1] N. Provos, "Encrypting Virtual Memory," Proc. 9th USENIX Security Symp., Denver, August, 2000.

[2] 稲村, "Cryptographic Memory System," 情処研報, vol. 2003, no. 74, July 2003.

[3] M.K. KcKusick et al., "Process Debugging," in The Design and Implementation of the 4.4BSD Operating System, section 4.9, 1996.

[4] H. Dobbertin, "The Status of MD5 After a Recent Attack," CryptoBytes, vol. 2, no. 2, Summer 1996.

[5] Intel, "Protected-Mode Memory Management," in IA-32 Intel Architecture Software Developer's Manual Vol. 3, Section 3, 2004.

[4] CAN-2002-1420 や CAN-2003-0961 等。