卒業論文　　2011年度（平成23年度）

汎用OSにおける動的なファイルアクセス制御機構の実装

慶應義塾大学 環境情報学部

氏名：相見 眞男

担当教員

慶應義塾大学 環境情報学部

村井 純

徳田 英幸

楠本 博之

中村 修

高汐 一紀

Rodney D. Van Meter III

植原 啓介

三次 仁

中澤 仁

武田 圭史

平成24年2月14日

卒業論文要旨 - 2011 年度 (平成 23 年度)

汎用 OS における動的なファイルアクセス制御機構の実装

近年, ユーザーが意図しないプログラムが実行された結果, ホスト上に保存されたファイルがネットワーク上に流出するといった情報漏えいのセキュリティインシデントの急増が問題となっている．こうしたインシデントに対して, 安全なファイル管理の仕組みを提供するファイルアクセス検知・制御機構の考案が求められている.

本論文では，WindowsNT 系 OS 上で動作するユーザーモードによるファイルアクセスを検知・制御するための手法を構築し, ユーザーが意図しないプロセスに対して, 保護対象とする一定範囲のファイルへのアクセスを制限するシステムを提案する．カーネル空間での Process Structure Routine を用いてプロセス生成時の情報を取得し，イベントオブジェクトを用いてユーザ空間でのファイルアクセス保護機構と通信することで，不明なプロセスの保護対象ファイルへのアクセスを無効化することを可能にする．イベントオブジェクトのシグナル化の際に, I/O 要求によるパケット通信を行うことで，カーネル空間での実装を最小限に留め，コンテキストスイッチの発生数を削減することで動作負荷を低減する．これにより，保護フォルダへのアクセスを常時監視することが可能となり，データ流出等の被害を防止することが可能となる.

キーワード:
1.DLL インジェクション, 2.API フック, 3.FilterDriver, 4.PsSetCreateProcessNortifyRoutine,

慶應義塾大学 環境情報学部

相見 眞男

Abstract of Bachelor's Thesis - Academic Year 2009

Implementation of dynamic file access controller on a versatile operating system

In recent years, security incidents of information leakage increased rapidly. In these incidents, files saved on hosts flow out onto network with uninted execution of programs. Therefore, the design of file access controller mechanism which offers a structure of secure file management need to be developed.

In this paper, we propose a method to detect and block unauthorized file access on Windows OS with reduced process load. Obtaining process generation event by using process structure routine in kernel space and communicating with file access control mechanism in userspace with event object. The proposed system blocks unknown process trying to access protected files. The load can be reduced by lowering number of context switch events by communicating with IO request packets at the time of event object signaling. With this method the system can monitor specific folder in realtime and can prevent data leakage caused by exposure viruses.

Keywords :
1.DLLinjection, 2.API hooking, 3.FilterDriver, 4.PsSetCreateProcessNortifyRoutine

Keio University, Faculty of Environment and Information Studies

Naoto Somi

目 次

図 目 次

表 目 次

第1章　序論

ヒューマンエラー, あるいはユーザーの意図しないプログラムが実行された結果, ホスト上に保存されたファイルがネットワーク上などに漏えいするセキュリティインシデントが問題となっている．本研究では, 急増するセキュリティインシデントに対して, 安全なファイル管理の仕組みを提供するフォルダ保護システムの実装と評価を行う.

1.1　はじめに

本研究では，コンピュータ上のファイルへのアクセスを全てブロックすることにより特定のフォルダからの情報漏えいを完全に防止するという発想から, ファイルアクセスそのものの監視に着眼している．そこで, Win32API フックの仕組みを取り入れたファイルアクセス検知・制御機構を提案する．この検知・制御機構は，ユーザーが意図しないプロセスに対して, 保護対象とする一定範囲のファイルへのアクセスを制限し, 重要なファイルの流出防止を可能にするものである．さらに, この機構を利用したフォルダ保護システムを実装し, システムの評価を行う.

汎用 OS としては WindowsNT 系 OS について言及し, NT シリーズの各 OS 間におけるカーネルの差異を吸収した実用性のあるシステムとなるような設計を行っている．これについては, 第 5 章で述べている．また, インターフェースを強化するための設計, Windows ファミリ間での移植性についても言及する.

さらに, フォルダ保護システムが常駐型として監視が行えるよう，オペレーションシステムへの負荷を最小限に留める工夫を行っている．本システムは, カーネル空間とユーザー空間に跨る実装の形をとっているが, これは両者を連携させることで, カーネル空間とユーザー空間における相互の欠点を補い合わせることにある．ユーザー空間のみを利用した場合, カーネル空間を利用した方法に比べて, Win32API のイベント観測粒度は粗いため, 公式に提供された機能をそのまま用いる実装では検知面での不安要素が残る可能性がある．逆に，カーネル空間のみで実装をすることも，カーネル空間で用いられる NativeAPI のドキュメントが一部非公開となっている理由により，正確なシステムの検証が行えない可能性がある．そこで, カーネル空間とユーザー空間の利点をできるだけ有効活用するため, 両空間に跨る実装の仕組みを本研究で新たなに構築した．ただし，実装においてユーザー空間とカーネル空間が混在することはコンテキストスイッチの増大も意味するため, ユーザーモードとカーネルモードの切り替えの手続きをできるだけ煩雑なものにしないように工夫した．精度が要求される部分についてのみカーネルモードで実装し, 実行速度を重視する部分については全てユーザーモードで実装するといった, 異なる空間での実装

の混在を最小限に留めることで, オペレーションシステム上の全プロセスを常時監視できる仕組みを提供する.

1.2 本研究の目的

情報漏えいのセキュリティインシデントは増加傾向にあり, 個人のみならず企業においても深刻な問題となっている. そのため, オペレーティングシステムの標準機能やウイルス対策ソフトを用いた様々な脅威への対策が行われているが, 現状はこれらの対策のスピードを上回る勢いで情報漏えいの拡大が続いている. 既存技術は多くの課題を抱えており, 万全な情報漏えい対策を行えているとは言い難い.

本研究の目的は, 既存技術の課題を克服するファイルアクセス検知・制御機構の構築を行い, この機構を利用したシステムを実装することで, 情報漏えいセキュリティインシデントの増加を食い止めることにある.

1.3 フォルダ保護システムとは

フォルダ保護システムは, 特定のフォルダへのファイルアクセスを制御する機能と, ファイルアクセスのログを出力する機能を有している. この機能の実装には, 本研究において提案するファイルアクセス検知・制御機構を用いた.

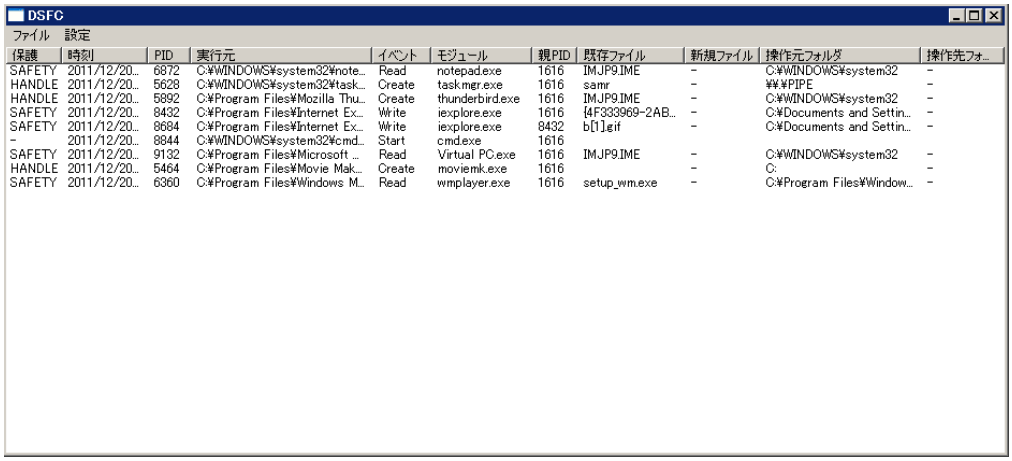

図 1.1: フォルダ保護システム

本システムは, WindowsNT 系 OS 上での Win32API コールの情報をリストビュー形式で表示している.(図 1.1) 具体的には, ファイルアクセスを担う CreateFile 関数, ReadFile 関数, WriteFile 関数, DeleteFile 関数, CopyFile 関数, MoveFile 関数といった Win32API の呼び出しをファイルアクセスを行ったプロセス (以下, ターゲットプロセス) をファイルの情報とセットにして表示することで，ターゲットプロセスの挙動を判別可能にする. さらに, ユーザーが保護対象としているフォルダに対してファイルアクセスがあった際に, 呼

び出したターゲットプロセスのアクセス制御を行うことも可能となっている.

さらに本システムはターゲットプロセスによるデータ転送の監視において, 将来的にフィルタドライバや NativeAPI フックの仕組みを用いることも検討する.

1.4 本論文の構成

本論文は, 全 8 章から構成される. 第 2 章では, 本研究を行うにあたっての研究背景を述べ, 第 3 章では情報漏えいへの既存の対策技術と関連研究について記す. 第 4 章では, フォルダ保護システムの提案を行い, 第 5 章では, ファイルアクセス検知・制御機構を用いた設計を記す. 第 6 章で, 本システムの性能評価を行い, 第 7 章では, それらに対する考察や本研究の課題点について述べていく. 最後に, 第 8 章において, 本研究のまとめと今後の展望を述べる.

第2章　背景

近年, コンピュータ上に保存されたファイルが漏えいするセキュリティインシデントが急増し, 問題となっている. 本章では, これらインシデントに対する様々な研究報告を纏めていく.

2.1　漏えいインシデントの急増

近年, 情報漏えいのセキュリティインシデントが問題となっている. NPO 日本ネットワークセキュリティ協会の調査によると, これらのセキュリティインシデント数は増加傾向にあり, 2002 年から 2010 年の報告件数だけをみても, 63 件から 1679 件へと, わずか 8 年で 24 倍近く増加している.[1][2]

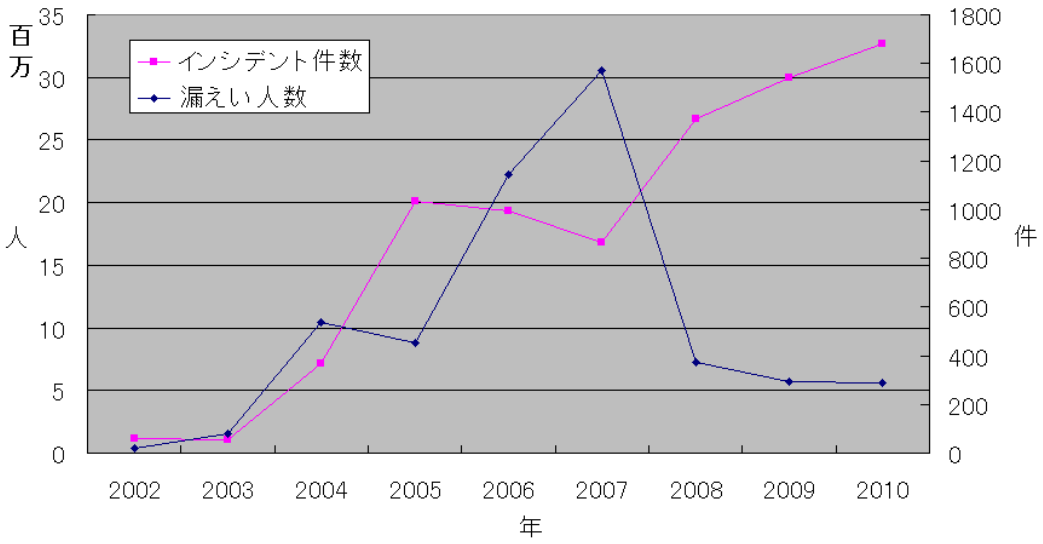

図 2.1: 漏えい人数とインシデント数の経年変化 [1][2]

漏えい人数は年によるばらつきがあるが, 2008 年からはほぼ横ばいとなっているが, インシデント数は今後も増加傾向にあることが読み取れる.(図 2.1)

2008 年より漏えい人数が横ばいのままインシデント数が増加していることは, セキュリティインシデント一件あたりの漏えい人数が減少していることを意味する. しかし, 依然としてインスデント数としての被害は増大傾向にあり, 早急な漏えい対策が求められている.

これら漏えいインシデントには, ウイルスといった不正なプログラムだけでなく, 誤操作, 紛失, 持ち出しやバグなど様々な要因も絡んでおり, そのためにコンピュータ上での安全なファイル管理をより一層難しくしている.

2.2 漏えい原因比率の複雑化

コンピュータ上の個人情報が含まれたファイルの漏えい原因としては, 一般的にウイルスや不正アクセスが知られているが, これらの漏えい原因は原因全体の割合としては小さい

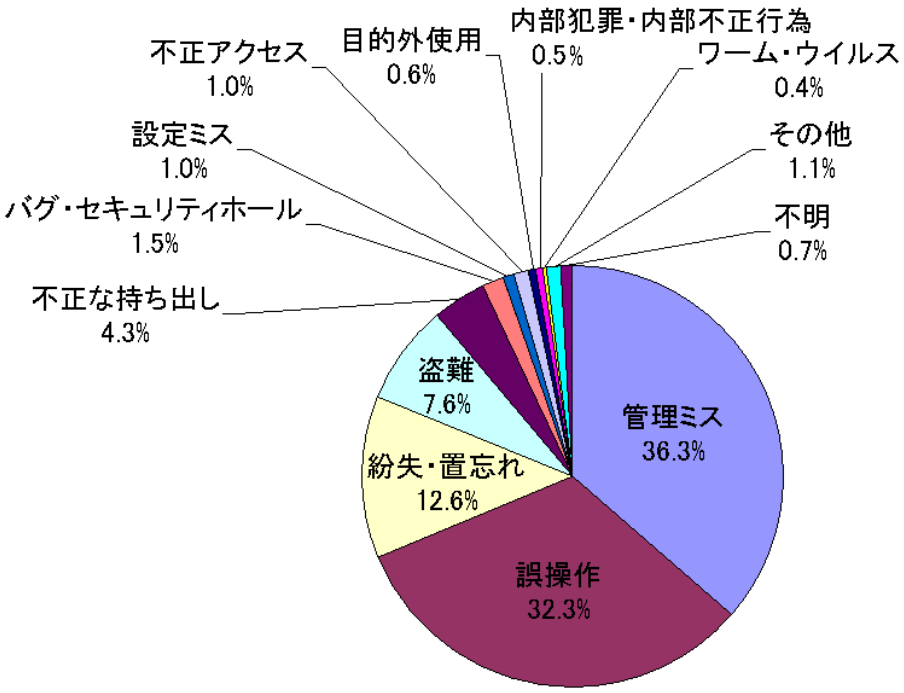

図 2.2: 漏えい原因比率 [2]

出典：JNSA 情報セキュリティインシデントに関する調査報告書

図 2.2 に示すとおり, 原因の約八割は管理ミス, 誤操作, 紛失・置忘れといったヒューマンエラーによるものであるため, ウイルス対策のみを行っていても漏えい対策としては不十分であることがわかる.

そこで本研究のフォルダ保護システムでは, 特定の原因に焦点をおいた漏えい防止策ではなく, コンピュータ上でのあらゆるファイルアクセスをブロックすることに着眼している. 不明なファイルアクセスに対して詳細な情報を取得し, その挙動の制御を行うことでフォルダからの漏えいを完全に防止することが可能となる.

2.3 セキュリティ・ベンダーによる対応の限界

セキュリティ事業会社マカフィーの調査によると, ”2010 年には約 1.4 秒に 1 件の割合で新種のマルウェアが発生”し, ネットワークを介して世界中に被害をもたらしている.[6] このため, ウイルス解析によって作成されるウイルス定義ファイル (以下, シグネチャ) を用いたパターンマッチングでは, セキュリティベンダによるタイムリーな対応が限界を迎えつつある.

また, ウイルスには標的型攻撃やウイルス亜種も存在し, これらに対するシグネチャ作成のための解析にはより多くの時間を必要とすることがある.

2.4 新たな脅威

本節では, 情報漏えいのセキュリティインシデントにおける新たな脅威と, その対策における課題点を述べる. 新たな脅威としては, 標的型攻撃とウイルス亜種を挙げる.

2.4.1 標的型攻撃

近年, 標的型攻撃と呼ばれる特定の企業や団体のユーザーをターゲットにした攻撃が増加している. 標的型攻撃には, 主に組織の関係者を装いウイルスを添付したメールの送信を行う手法が用いられており, ”実在する部署や社員の名前の記載によってターゲットを信用させ, 攻撃の成功率を高めている”.[7]

こうした標的型攻撃は特性上, 攻撃に用いられるウイルスが広い感染活動を行うことが少ない. もともと標的を定めている攻撃のため, ウイルスによる感染が発覚するころには既に手遅れとなってしまうことや, ウイルスに関する報告や検体の収集が不十分であることからウイルス解析がスムーズに進まず, 完全に攻撃を防ぎきることが難しい.

2.4.2 ウイルス亜種

個人や企業の情報が, Winny など P2P モデルのファイル共有ソフトを介して漏えいすることもある. こうした漏えいは, Winny 媒介型ウイルスなどの攻撃によって引き起こされている. Winny 媒介型ウイルスなどはネットワーク上で広く感染活動を行っていることが多く, 十分な検体が確保できることからウイルス対策ソフトによる対策が有効である. しかし, これらの対策を回避するために, ウイルスコードの一部をカスタマイズした亜種の存在も確認されている. こうした亜種はウイルス対策ベンダー側で網羅的に把握することが難しく, 亜種検体ごとの解析は不十分なものとなっている.

2.5 まとめ

本章では, 情報漏えいセキュリティインシデント数の経年変化からインシデントが増加傾向にあることを示した. また, 漏えい原因比率を詳細に示すことで, 漏えい対策がウイルス対策のみでは不十分であることを述べた. さらに, 新たな脅威として標的型攻撃やウイルス亜種について述べ, 対策の不十分さを述べた.

第3章　関連研究

特定のフォルダを安全に管理するための仕組みや, アクセス元のプログラムを除去する技術は既に存在しており, その幾つかは実用化されている. WindowsNT 系 OS にはオペレーションシステム自体に, アクセス権と呼ばれるフォルダ管理のためのセキュリティ機構が備わっている. また, ウイルス対策ソフトにはパターンマッチングを代表とするウイルス除去におけるセキュリティ機構が幾つか存在する. 本章では, それら関連研究による対策技術について述べていく.

3.1　オペレーションシステムによるフォルダ制御

WindowsOS は, Unix 系 OS におけるディレクトリ構造を GUI 上でフォルダとして扱うことにより, ユーザーに対して視覚的なファイル管理を提供している.

Unix 系 OS におけるディレクトリと, WindowsOS におけるフォルダの概念は, 基本的には同様のものとして捉えることが可能であるが, WindowsOS 上の特殊なフォルダの中には, コントロールパネルやデスクトップ, ごみ箱といった, ディレクトリツリーに一対一の関係で対応しないものも存在するため, 注意が必要である.

3.1.1　ディレクトリとフォルダ

図 3.1 に, Unix 系 OS のディレクトリ構造を示した.

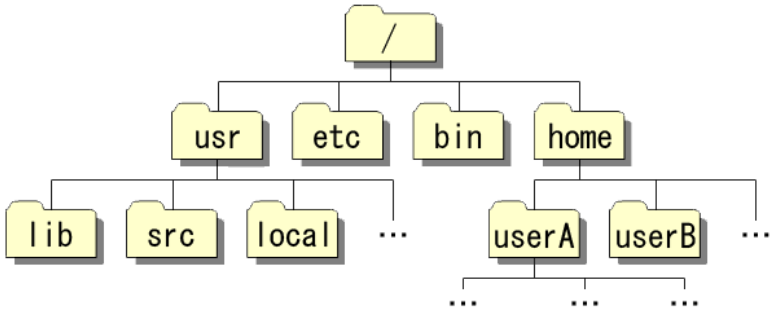

図 3.1: Unix 系 OS におけるディレクトリの概念

Unix 系 OS は, ルートディレクトリ「/」を最上層として, 全てのディスクを一つのディレクトリツリーにおいて管理しており, ディレクトリとファイルの関係が完全なツリー状

となっている. そのため, 全てのディレクトリには対応するパスが必ず与えられている.

次に, WindowsOS が扱うフォルダの概念である. 基本的な構造は Unix 系 OS と変わりないが, ルートディレクトリの変わりにドライブという概念が存在していることや, コントロールパネルのように, ディレクトリが存在しないものを特殊なフォルダとして扱う仕組みも存在するなど, 幾つかの例外が存在する. そのため, 全てのフォルダに一意のパスが定まるわけではない. こうした特殊なフォルダがどのように扱われているかを, 以下の対応表に纏めた.

特殊なフォルダ	実体
マイコンピュータ	アドレスによって管理される領域
ドライブ	パーティションごとに管理される論理ドライブ
デスクトップ	現在のユーザーに割り当てられる Windows 内のディレクトリ
コントロールパネル	System32 以下に保存された cpl ファイルの集合
ゴミ箱	論理ドライブごとに作成される特殊な領域

表 3.1: 特殊なフォルダの実体

このように, GUI で利用されることが前提の WindowsOS では利便性から, ディレクトリ構造と必ずしも一意に対応しない特殊なフォルダが存在している. また, WindowsOS では論理ドライブごとにフォルダが管理されているため, ディレクトリツリーの最上層にはドライブ文字が割り当てられている.

3.1.2 フォルダへのアクセス権

フォルダには Windows エクスプローラにより, 一定のセキュリティを設定することが可能である. 特定のフォルダに対するアクセス許可を付属することで成り立つこの機構は, アクセス制御リストをエントリ形式で提供している.(図 3.2)

このセキュリティモデルでは, ユーザーアカウントやグループといった単位にフォルダやファイルといったオブジェクトへの細かい権利を与えることが可能である. プロセス単位でのアクセス制御は行えないものの, オブジェクトには継承やコントロールを含め数多くの組み合わせを用いることができるため, ユーザーアカウントごとのファイル管理を厳密なルールのもとに行える利点がある.

しかし, コントロールの設定項目が複雑であり, アクセス制御エントリの適用先が継承フラグを含めると大変多く煩雑になりやすいため, この方法はファイルサーバを管理するようなシステム管理者レベルの知識を必要とする. さらに, フォルダへのファイルアクセスの把握をユーザーアカウント単位で行うため, プロセスが何らかの方法によりスーパーユーザーの特権を掌握した場合, セキュリティ機構がうまく機能できず, フォルダ内へのアクセスに対する制御を正常に行えないことがある.

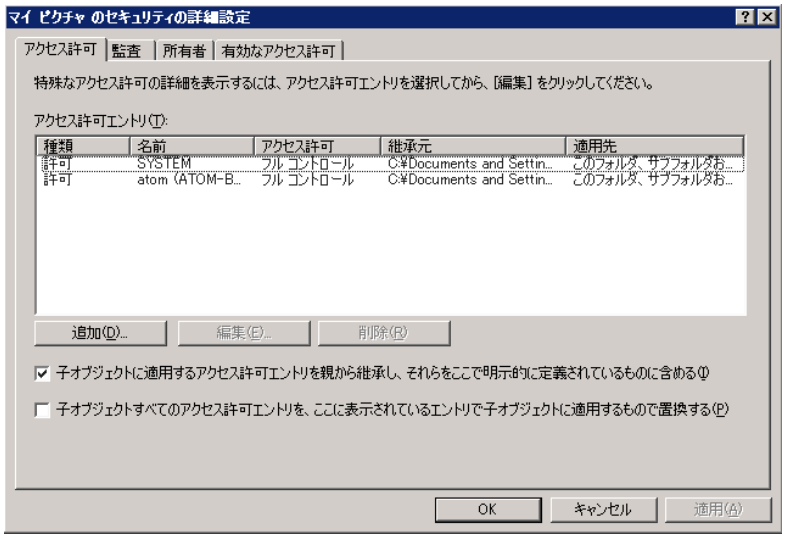

図 3.2: アクセス制御エントリ

3.2 ウイルス対策ソフトによるパターンマッチング

ウイルス対策ソフトにおけるパターンマッチングとは, あるウイルスに特徴的な部分をシグネチャコード化し, シグネチャパターンに合致するものをウイルスとして検出する手法のことである. 未知のウイルスが発見されると, セキュリティ・ベンダーによって動的解析や静的解析が行われ, ウイルスに特徴的な傾向を纏めたシグネチャコード群がウイルス定義ファイル (シグネチャ) の形式で提供される. このシグネチャを用いることで, 不正な実行可能ファイルをウイルスと判別し, 除去対象として検出することが可能になる.

3.2.1 静的解析と動的解析

ここでは, ウイルスに対する静的解析と動的解析の違いについて述べていく.

- 静的解析
  静的解析とは, ウイルス検体を実行させることなく, 逆アセンブルによってウイルスのプログラム構造を分析する手法である. 逆アセンブルしたプログラムコード中のサブルーチンの呼び出しを全てチェックできた場合には, ウイルスの挙動を完全に把握できることになるが, 時間的制約からこうしたことは困難である. また, 近年では自身のコードに圧縮・難読化処理を施すウイルスも登場しているため, 静的解析に要するコストが益々大きなものとなっている.

- 動的解析
  動的解析とは, ウイルス検体を他のシステムに影響を与えない隔離された環境下で実際に動作させることで, その挙動を明らかにする手法である. 自動化も可能であることや, 比較的短時間で検体の挙動をおおよそ把握することができる利点があ

るものの, 仮想環境下では動作しない検体や, 日付など特定の条件が満たされるまで潜伏している検体も存在し, ウイルスの解析結果が実際に動作する環境に左右されることも多い理由から, 厳密な解析が必要とされる場合には向かないこともある.

未知のウイルスに対する脅威にタイムリーに対応すべく, セキュリティ・ベンダーは, 手始めに自動化した動的解析を行うことが多い. 脅威の発覚後, いかに多くの情報を自動化した動的解析から収集し、手動による動的解析や静的解析に要する時間を削減できるかが, セキュリティ・ベンダーにとって重要な課題となっている.[4]

3.2.2 フォールスネガティブとフォールスポジティブ

パターンマッチングには, フォールスネガティブ (False Negative) とフォールスポジティブ (False Positive) における相関関係の問題がある. フォールスネガティブとは, ウイルスの見逃しであり, シグネチャの精度を上げることで発生率を抑えることができる. フォールスポジティブとは, 正規のプログラムをウイルスとして誤検出することであり, シグネチャの精度を粗くすることで発生率を抑えることができる.

脅威の分析においては, 両者の発生確率がゼロとなることが理想とされているが, 一般的にフォールスネガティブとフォールスポジティブは負の相関関係にあるとされるため, 実現は困難である.[3]

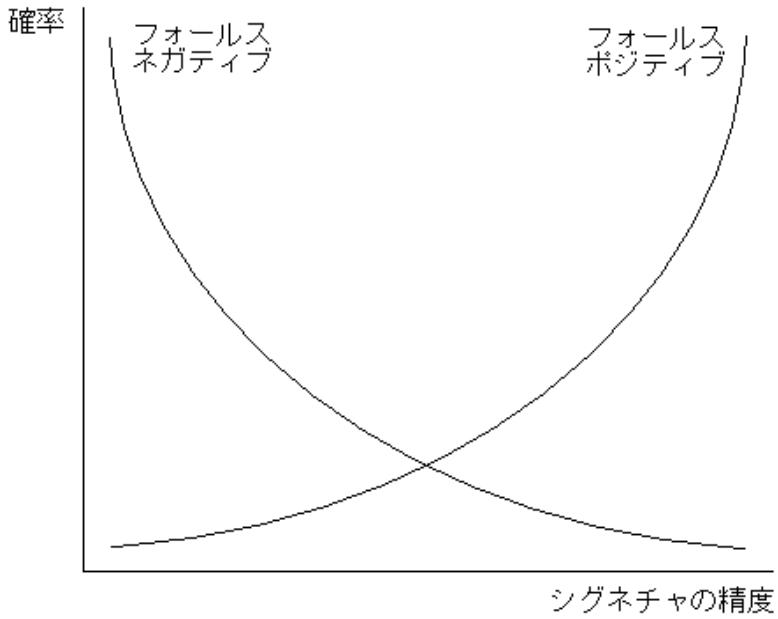

図 3.3: フォールスネガティブとフォールスポジティブの相関イメージ [3]

出典：川口洋のセキュリティ・プライベート・アイズ

そこで, 一般的なパターンマッチングは脅威を発見できないことを避けるため, フォールスネガティブの抑制に重点を置くことが多い.

3.2.3 難読化による検出回避

パターンマッチングによる検出がウイルス対策ソフトの主流であり, この検出を回避するため, 自身に様々な難読化処理を施すウイルスが登場している. 図 3.4 は, 発見されたウイルスのうち, パッキング (圧縮や難読化) が施されていたものの割合である.

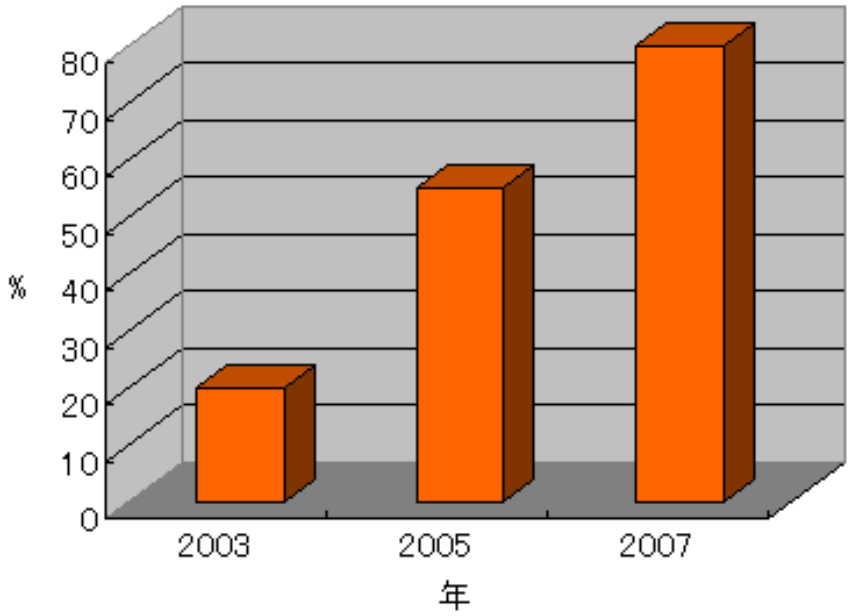

図 3.4: ウイルスのパッキングの割合 [4]

このように, ウイルスのパッキングの割合は年々高まりをみせている. 次に, 難読化が施されたウイルスについて, ポリモーフィック型とメタモーフィック型を例に挙げて説明する.

- ポリモーフィック型ウイルス
  ポリモーフィック型ウイルスは, 複製のたびに自身をランダムに暗号化することで, ファイルサイズやデータを変化させてパターンマッチングによる検出を不可能にする. 単純なモデルでは, 実行時に暗号化されたウイルスを復号するためのルーチンを持つため, このルーチンに対して, シグネチャを用いることで検出が可能となる. ミューテーションエンジンを持つポリモーフィック型ウイルスの場合, 復号ルーチンのコードを変化させて, 検出を回避するものもある.[8]

- メタモーフィック型ウイルス
  メタモーフィック型ウイルスは, 逆アセンブルによって自身のウイルスコードをメタ言語化する. その後, 分析やコードのモーフィング, さらに再アセンブルにより, 新たなコードを再生成することで, 暗号化を行うことなくシグネチャパターンを変化させる. ウイルスの機能は変化しないが, シグネチャパターンが変化しているため, パターンマッチングによる検出が困難になる.[8]

こうしたウイルスの登場により, パターンマッチングのみを用いたウイルス対策はセキュリティ・ベンダーにおいて, 限界を迎えつつある.

3.3 Windows 標準機能を利用した監視

マイクロソフトは, 公式に WindowsOS のイベントを監視するための API をいくつか公開している. FindFirstChangeNotification 関数や, SetWindowsHookEx 関数を用いることで, WindowsOS 上のイベントを監視を行うことができる.

3.3.1 ディレクトリに対するフィルタの利用

FindFirstChangeNotification 関数は, 指定したディレクトリやサブツリーにフィルタを付加し, 特定のイベント時に制御を返す関数である. この関数を用いることで, 指定したディレクトリに対するファイルアクセスを一部, 監視することが可能になる.

```
HANDLE FindFirstChangeNotification(
    LPCTSTR lpPathName,
    BOOL bWatchSubtree,
    DWORD dwNotifyFilter
);
```

FindFirstChangeNotification 関数によるフィルタがどのようなファイルの操作に対応可能かを, 表 3.2 に纏めた.

	保護フォルダ⇒通常フォルダ	通常フォルダ⇒保護フォルダ
読み込み	×	-
書き込み	×	○
削除	○	-
移動	○	○
コピー	×	○

表 3.2: FindFirstChangeNotification 関数でのファイル監視

この方法では指定したディレクトリにおけるファイルの変更を監視することができるが, ファイルの転送を API レベルで監視しているわけではない. このため, 保護フォルダから通常フォルダへデータ転送を行う漏えいには移動操作のみ, 通常フォルダから保護フォルダへデータ転送を行う改ざんへの対応は書き込み, 移動, コピー操作のみにしか対応せず, 脅威の一部に対応できないことから脅威への完全な対応が難しくなっている. また, 監視は行えるもののフォルダへのアクセス制御は一切行えないなど, 問題点も多い.

3.3.2 Windows メッセージフック機構の利用

Windows が公式に用意した SetWindowsHookEx 関数を用いて Windows メッセージをフックし, プロセスの監視を行う方法がある. この関数は, フックプロシージャをフックチェーン内にインストールすることができ, 特定のメッセージイベントを監視することが可能である. このイベントをターゲットプロセスの生成と結び付けることで, ターゲットプロセスに Win32API のフック処理を施すことが可能になる. API フックでは, 様々な API の呼び出しから, 不明なプロセスの挙動の詳細を把握することができる.

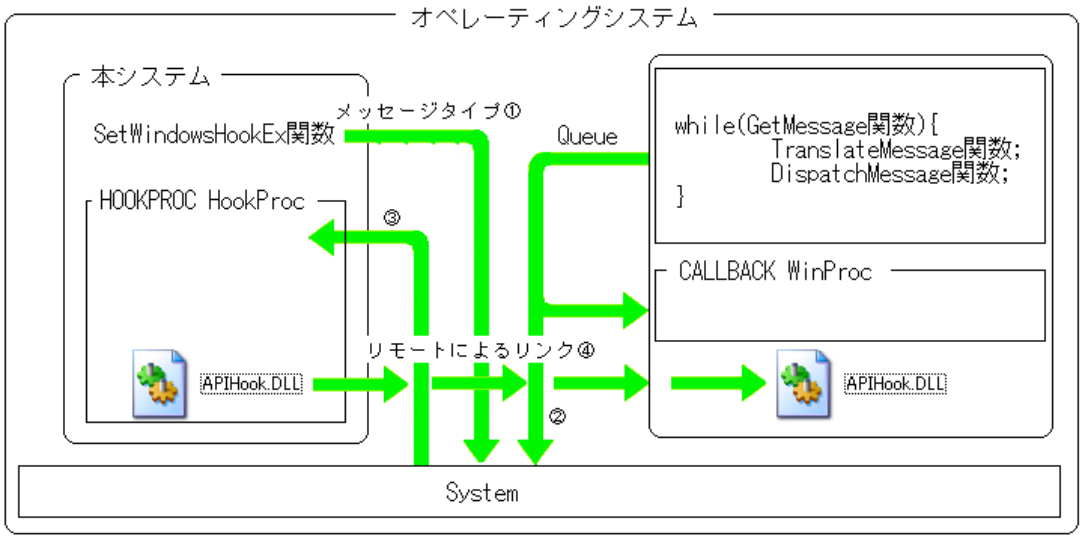

図 3.5: Windows メッセージフックの概要

ただし, この関数は Windows メッセージをフックするものであるため, CUI のようなウインドウを持たないアプリケーションには, Windows メッセージのフックを用いてもターゲットプロセスの正確な生成を捕捉することができない.

3.3.3 ハッシュ値によるファイルの同一性チェック

フォルダやファイルに対してチェックサムによる整合性検証を行うことで, フォルダに存在するファイルの持ち出しや改ざんをチェックする.

一般的に利用される方法ではあるが, この方法はアクセス制御まで行うことができない. また, 完全性を保つためハッシュ値の衝突問題を克服しなければならない.

3.4 フィルタドライバによるドライバ間の監視

フィルタドライバとは, カーネル空間における上位ドライバと下位ドライバにおけるデータ転送の監視が行えるデバイスドライバの一種である.

フィルタドライバを介することで, カーネル空間における様々なデータ転送の監視とアクセス制御が可能になる. また, カーネル空間に跨ることで, WindowsOS におけるイベントの観測粒度を高めることができ, ユーザー空間では捕らえきれなかったカーネルコードによるファイルアクセスの監視・制御も行うことも可能となる. WindowsOS においてユーザーモードで動作するプロセスも, デバイスにアクセスする際に I/O マネージャを介してドライバに対する要求を発行するため, これらドライバ間のデータ転送の監視を行うことで, ユーザーモードのプロセスのファイルアクセスにも対応可能である. ただし, 制御に関しては, ドライバにおけるカーネルコードから, ユーザーモードで動作するプロセスの Win32API への異なるアドレス空間における紐付けを行わなければならない.

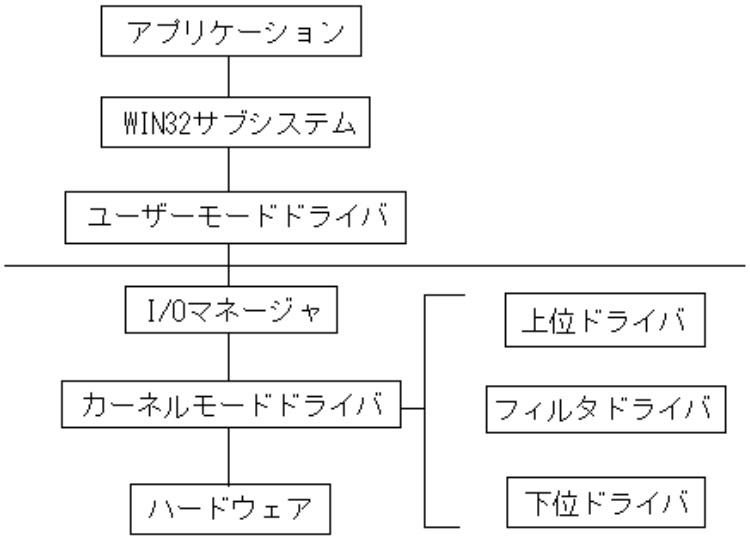

図 3.6: フィルタドライバの階層

WindowsNT 系 OS においては, ファイルは一般的に NTFS(NT File System) によって管理されている. そのため, 上位ドライバと最下位の NTFS ドライバにフィルタドライバを実装することで, WindowsNT 系 OS におけるファイルアクセスの監視・制御を網羅的に行うことが可能である.

3.5 NativeAPI フックによるカーネルコードの監視

WindowsNT 系 OS における動的なファイルアクセス制御の仕組みとしては, NativeAPI に着目したものがある. NativeAPI はカーネルモードで実行されるシステムコール郡であり, これらをフックすることで, Win32API フックよりもきめ細かいイベント観測粒度でプロセスの挙動を把握することが可能になる.

これまで, NativeAPI フックという優れた処理をプロセスの挙動を把握するために用いる研究は多くなされてこなかった. それは, NativeAPI 郡の中には, マイクロソフトから非公開扱いを受けて仕様が隠されている API が存在しており, NativeAPI の挙動を網羅的に

把握することが現実的に困難であったからである. さらに, NativeAPI フックの処理自体も OS に依存する部分が多く, Win32API フックという比較的実装難度の低いフック処理の代替的存在とはならなかった. しかし, 近年のウイルスの中にはカーネルモードで動作し, ユーザー空間での隠蔽活動を行うものも多く, 検出精度のより高いシステムを構築する際にはこのような手法を用いなければならないこともある.

そこで, 仮想計算機モニタ (VMM) による NativeAPI フックの手法も考案されている. VMM はゲスト OS よりも低い層で稼動しているため, OS におけるユーザーモードとカーネルモードのコンテキストスイッチの切り替えイベントを監視することが可能となる. この方法により, 仮想環境でカーネルモードで動作するウイルスの挙動を把握することが可能になっている.[9]

ただし, ネイティブ OS 上での網羅的な NativeAPI フックの実現は依然難解であり, NativeAPI フックによるファイルアクセス解析の研究には多くの課題が残されている.

3.6 まとめ

本章では, 特定のフォルダを安全に管理するための仕組みやパターンマッチングなど既存技術の紹介と, それらが抱える課題について述べた.

既存技術はアクセスに対する監視のみ対応し, 制御までの応用が利かないものも多い. また, 近年増加するパッキング処理の施されたウイルスには用いることができないなど課題が多く残っている. これらの難読化されたウイルスに対応するための研究が進められているが, その多くがオペレーションシステムのカーネル空間に依存する実装の形をとるため構造が難解なものとなり, 実用的なセキュリティ機構の構築に至っていないのが現状である.

第4章　アプローチ

本章では, 第 3 章で述べた既存技術の問題点を克服するために, フォルダ保護システムの提案を行う.

4.1　フォルダ保護システム

フォルダ保護システムは, 保護対象とするフォルダに対してファイルアクセスを制限し, 重要なファイルの流出防止を可能にするシステムである. 本システムは, 特定のフォルダへのファイルアクセスを制御する機能と, ファイルアクセスのログを出力する機能を有している.

4.2　要件定義

フォルダ保護システムがファイル漏えいのセキュリティインシデントへ柔軟に対応できるように, セキュリティインシデントをフェーズごとに分類し, それらに対して開発事項を列挙した.

- フェーズ
  - ローカルでの安全なファイルの閲覧と保存
  - フォルダに対するファイルアクセスの監視
  - ネットワークへのファイル漏洩の遮断

これらのフェーズにもとに, 対応する機能の開発事項を挙げていく.

- 開発事項
  - ファイルの暗号化と復号
  - ファイルアクセスのログ出力
  - ファイルアクセスのコントロール

本システムは, 特定のフォルダを保護フォルダとして設定し, 安全なファイル管理を行うシステムである. そのため, 開発要項に挙げたファイルの制御については, 予めユーザーが任意に設定した保護フォルダ内でのみ行う.

4.3 想定する利用者

本システムは複数のユーザーについて考慮して設計を行っている. このようにすることでユーザーの必要に応じて詳しい情報提供を可能にしつつ, 全てのユーザーへの対応が可能となる.

まずは, オペレーティングシステムやファイルシステムの仕組みについて詳しくない一般的な Windows ユーザーに対する設計について述べ, 次にウイルス解析を行う技術者向けの設計について述べる.

- 一般的な Windows ユーザー

  一般的な Windows ユーザーは, ファイルアクセスの仕組みを十分に理解しているとは言い難い. そこで, 特定のフォルダを保護する機能のみを利用することができるように, 最初の起動画面で選択肢を表示する.

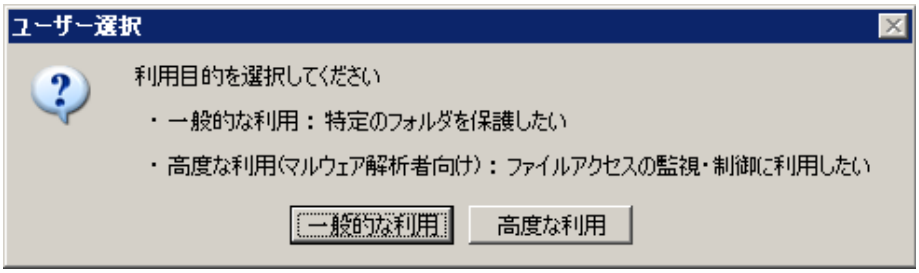

図 4.1: 起動画面におけるユーザー選択

  一般的な利用を選択した際には, ウインドウを表示せずバックグランドで本システムが起動する. これにより, ファイルアクセスの情報でユーザーの混乱が起ることがなく, 保護フォルダでのファイル管理に重点を絞った使用が可能となる.

- ウイルス解析を行う技術者

  ウイルス解析を行う技術者には, できるだけ多くの情報を表示する. また, ファイルアクセス以外の Win32API の解析を望む解析者に対しては, 特定のプロセスに任意の制御コードを施した DLL を注入することが可能な機能を追加する.

注入画面で, ターゲットプロセス名と, プロセス制御コードが記述された DLL を指定することで, 特定のウイルスの動作制御を行うことが可能となる.(図 4.7) この機能を利用することで, ターゲットプロセスの挙動を変化させることができ, 解析を有利に進めることが可能となる.

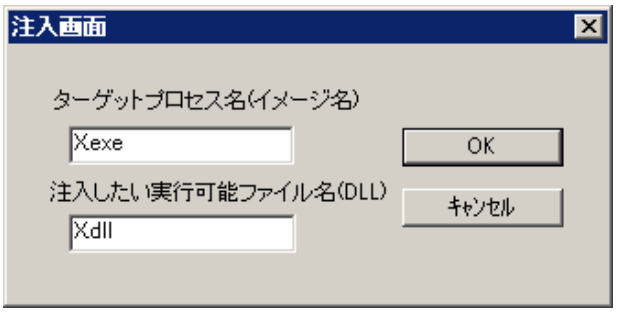

図 4.2: プロセス制御コードの注入画面

4.4 ファイルの暗号化と復号

悪意ある第三者にファイルを閲覧されない状態を維持したうえで, ユーザーが任意のタイミングでファイルを閲覧する機能を持たせる. こうしたファイル閲覧の制御に対しては, ファイル暗号化の仕組みを用いる. また, 決められた手順でのみ復号を行う機能も実装する. この機能はローカルでのファイル閲覧を安全なものにするだけでなく, ネットワークへのファイル漏えい時にも有効である. 一般的に, ネットワーク上に漏えいしたファイルを完全に消し去ることは極めて難しく, ファイルに含まれた個人情報からの二次災害を防ぐ役割の観点からも暗号化処理は重要な開発事項となる.

暗号化における手順では, 保護フォルダに対するファイルの移動及び複製時に, システムがファイルに対して毎回処理を施す. このようにすることで, 保護フォルダ内のファイルは全て暗号化された状態となり, 悪意ある第三者がファイルの中身を読み取ることを防ぐことができる.

復号における手順では, Windows エクスプローラにおいて, 保護フォルダから他の通常フォルダへ移動や複製を行った際や, ユーザーが指定したプログラムでファイルを閲覧した際に, システムが直前に処理を施す. Windows エクスプローラはユーザーのファイル移動及び複製を担っているため, これらのプロセスに対して復号処理を挟むことにより, ユーザーのドラッグアンドドロップによる保護フォルダからの安全なファイルの取り出しを可能にする.

4.5 ファイルアクセスのログ出力

本システムの重要な機能としてファイルアクセスの常時監視がある. ファイルアクセスの監視は, GUI 上で Win32API のやり取りをログ化することで行う. WindowsOS 上のプログラムは, 一般的に Win32API を用いて実装されているため, これら Win32API の呼び

出しを表示することで, ユーザーに対してどのようなファイルアクセスが行われているかを表示することが可能になる. また, 将来的には, 各 Win32API の呼び出し頻度や回数などの統計をグラフとして表示することで, 解析により適したシステムを目指す.

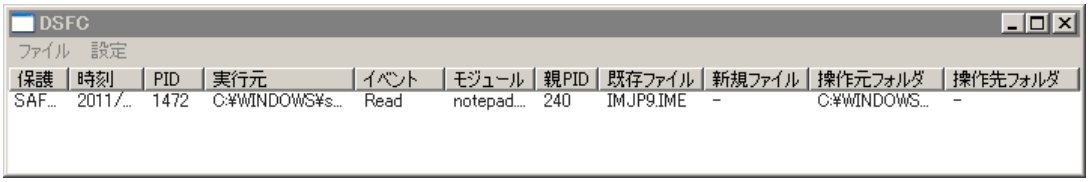

図 4.3: ファイルアクセスのリストビュー表示

ファイルアクセスはリストビューにより表示を行う. 図 4.3 に示すとおり, ファイルアクセスのログ視化項目には, 保護フォルダへのアクセスの有無, アクセス時刻のタイムスタンプ, 実行したプロセスの PID, プロセスの実行元のフルパス, プロセスがファイル操作, 親プロセスの PID, アクセスしたファイル (既存のファイル), 新規に作成されるファイル, アクセスしたフォルダ (操作元フォルダ), ファイルを移動する先のフォルダ (操作先フォルダ) があり, ファイルアクセスの具体的な詳細を見ることができる.

4.6 ファイル漏えいの遮断

ファイルアクセスの遮断は, ユーザーが指定した保護フォルダに対して行われる. 保護フォルダは, ユーザーが任意のフォルダに対して指定可能であり, 登録もコンテキストメニューから簡潔に行えるようにする. また, 保護フォルダへファイルアクセスがあった際に, ユーザーに対してそのファイルアクセスの制御を促すメッセージを表示する.

4.6.1 保護フォルダの登録

不明なファイルアクセスを検知し，保護対象とするフォルダを守るために, ユーザーは保護したいフォルダを登録設定する必要がある.

ユーザーは保護したいフォルダ上でコンテキスト・メニューを開き, PROTECT を選択することで, 任意のフォルダを保護フォルダに登録することが可能となる.(図 4.4)

4.6.2 ファイルアクセスのコントロール

ユーザーが設定した保護フォルダに対して, ファイルアクセスが検知された際には，警告メッセージを表示する．この時，ファイルアクセスを行ったプロセスはスリープ状態となり，ユーザーが警告メッセージを処理するまでは，動作を停止する．このようにすることで, ユーザーが任意のタイミングでアクセスを行ったプロセスに対してスリープからの

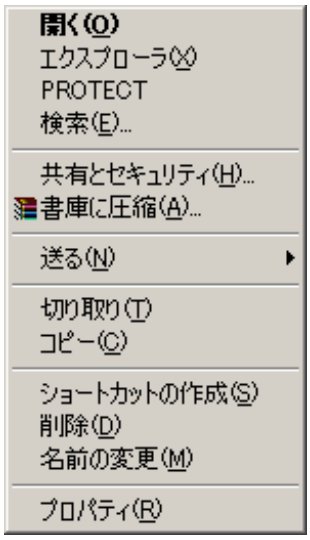

図 4.4: コンテキスト・メニューからの保護フォルダの登録

復帰, スリープの維持, 強制終了の選択を可能とする.

図 4.5 と図 4.6 は, 本システムのファイルアクセスに対する警告メッセージ表示の例である.

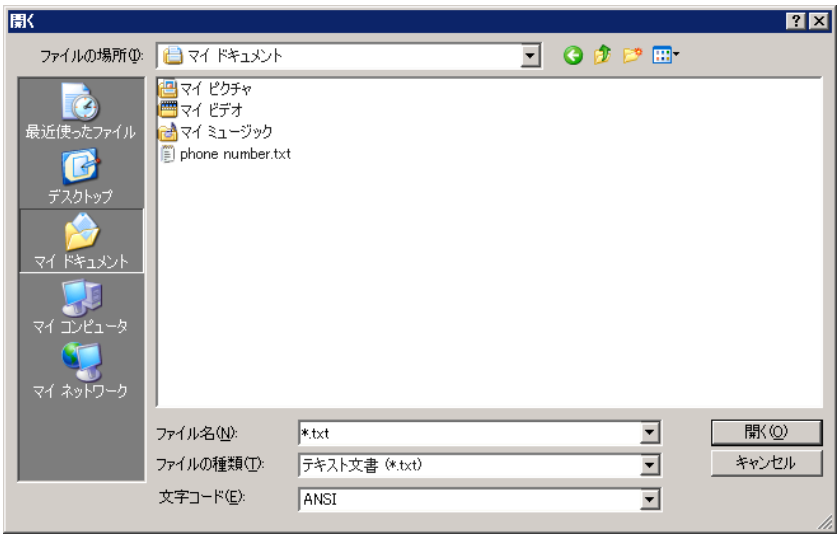

図 4.5: メモ帳によるファイル選択画面

マイドキュメントを保護フォルダに登録し, メモ帳によりテキストファイルのオープンを試みると, 図 4.6 のような警告メッセージが表示される.

この警告メッセージにより, ユーザーは保護フォルダに対するファイルアクセスを知覚することができ, 同時にファイルアクセスに対する詳細な情報を得たうえでアクセス制御を行うことが可能になる.

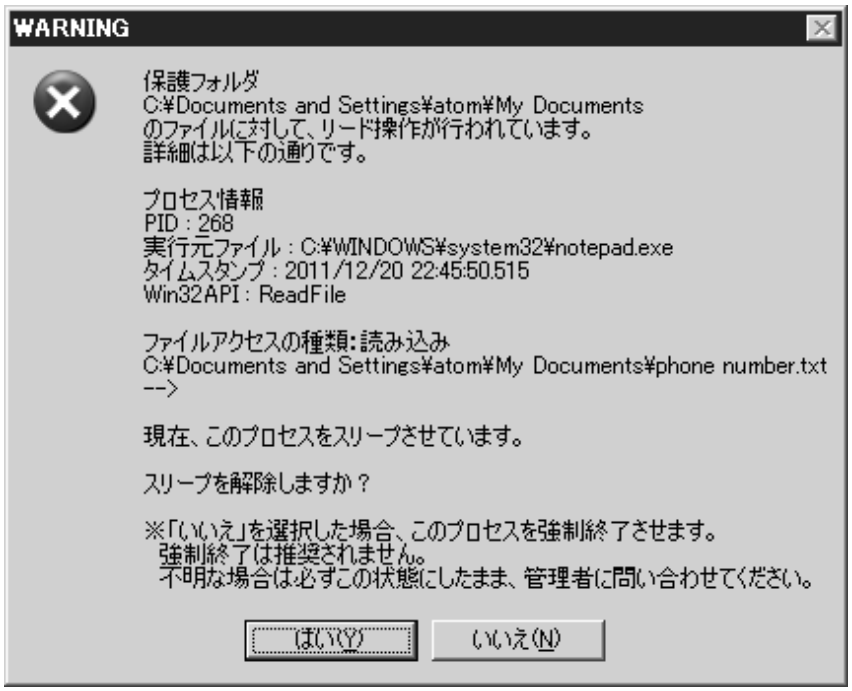

図 4.6: ファイルアクセスに対する警告メッセージ

4.7 拡張機能の追加

システムには拡張機能として, プロセス制御コードの注入, ロードモジュールリストの生成, USB ストレージの禁止設定が存在する. これらの機能によって, ファイルの漏えいをより効率的に防止することが可能となる.

4.7.1 任意のプロセス制御コードの注入

独自に作成したプロセス制御コードを任意のプロセスに DLL として注入する機能である. 本機能により, ファイルアクセスに関する Win32API 以外にも様々な Win32API のフックが可能となるほか, 様々な独自のコードをターゲットプロセスの仮想アドレス空間内で実行させることが可能となる.

4.7.2 ロードモジュールリストの生成

プロセスが読み込んだモジュール (主に DLL) を列挙し, リスト形式で表示する. ユーザー独自の DLL がターゲットプロセスに正常にリンクされているかを確認を行うための機能であるが, 将来的には明示的にリンクされた DLL のリンク時刻を把握することで, リンク時刻の傾向から不正な DLL を検出する用途に役立てる.

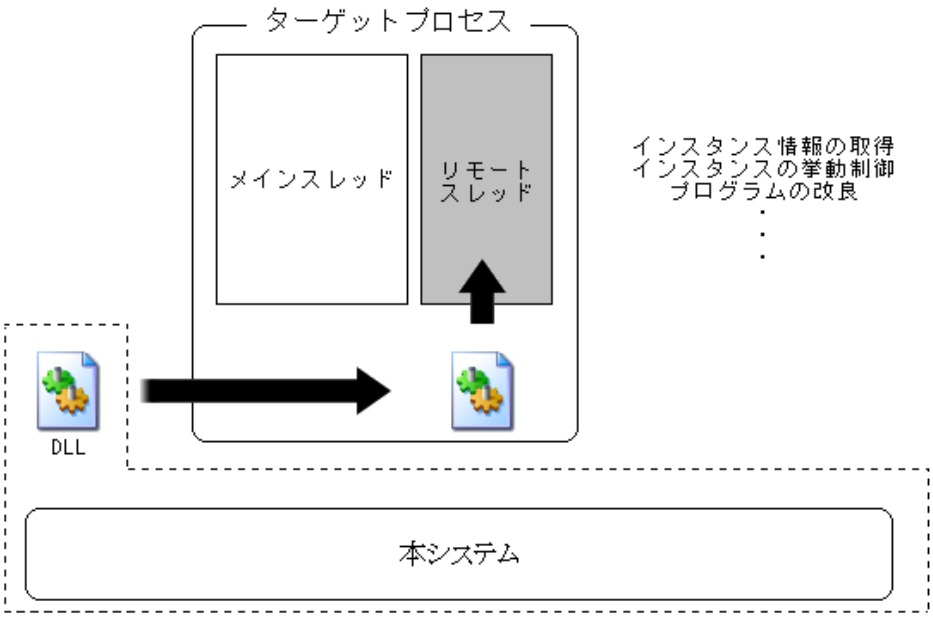

図 4.7: プロセス制御コードの注入

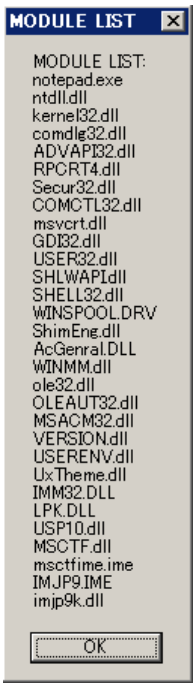

図 4.8: ロードモジュールリスト [5]

4.7.3 USB ストレージの禁止

USB ストレージを使用禁止にする機能である. この機能の目的は, USB メモリによる第三者のファイル持ち出しの防止にある.

近年, USB メモリを経由して感染するウイルスが増加しているが, これは USB メモリの

中に保存されたAutorun.infが, USB接続時にWindowsの自動実行機能 (Autorun) により, 悪意あるプログラムを自動で実行させるためである.[10] コンピュータ管理者が, WindowsのAutorunをオフにする方法もあるが, "Autorunの設定をWindowsが正しく解釈しないため, 無効化の設定をしたにもかかわらず, Autorunが動作するというマイクロソフトの報告もなされている".[11]

そこで本システムでは, レジストリにおけるUSBストレージの許可を変更することで, USBストレージの設定を正確に行えるようにする.

4.8 まとめ

本章では, フォルダ保護システムの提案を行い, その機能を詳細に述べた. また, 想定する利用者を複数に分けることで, Windows一般ユーザーとウイルス解析技術者の両者に対する要件を満たすシステムの提案を行った. 第5章で, このシステムの実装手法について解説する.

第5章　実装手法

本章では, 第 4 章で提案したフォルダ保護システムの仕組みを解説し, 実装手法を纏める.

5.1　フォルダ保護システムの処理フロー

フォルダ保護システムは，ユーザーが意図しない不明なプロセスによるファイルアクセスを検知し，保護対象とするフォルダ内のファイルを情報漏えいから守るシステムである.

本システムは, ターゲットプロセスの生成時よりプロセス情報の表示を行い, ユーザーに対して判断を問う. ターゲットプロセスがファイルアクセスを行った際には，リスト形式による詳細なアクセス情報を表示し, ユーザーが設定した保護フォルダに対するファイルアクセスが検知された際には，同時に警告メッセージボックスを表示する．この時，ファイルアクセスを行ったプロセスはスリープ状態となり，ユーザーが警告メッセージに対する応答を行うまでは，ターゲットプロセスの動作を一時停止させる. この一連のフローを図 5.1 に示した.

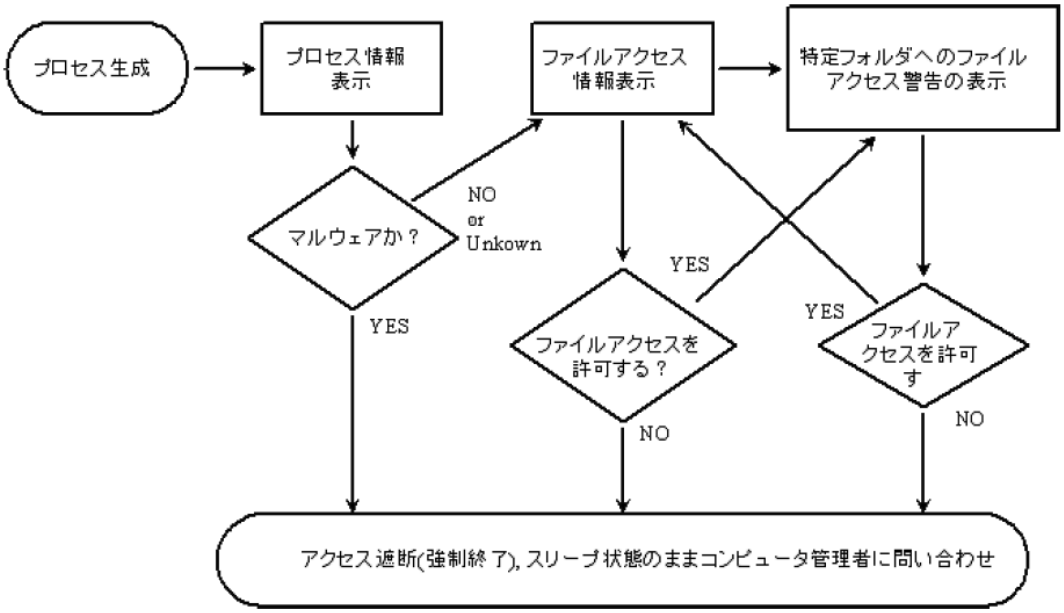

図 5.1: 提案システムの処理フロー

ターゲットプロセスは親プロセスや実行可能ファイルのパス情報などとセットで扱わ

れ, ユーザーはこれらの情報をもとにターゲットプロセスの生存許可の判断を下すことになる. 生存が許可されたターゲットプロセスがファイルアクセスに関する API を呼び出すと，アクセス対象とするフォルダやファイル, 呼び出した API に関する情報をユーザーに表示する. さらに, ユーザーの設定した保護フォルダにファイルアクセスがあった場合には，プロセスをスリープ状態にし，警告のメッセージボックスを表示することで，システムは, ユーザーにアクセス制御を促すことが可能になる. ユーザーがターゲットプロセスに対して行える挙動変更は幾つかあり, それらは第 5.8 節において述べる.

5.2 システムの構成

本システムの開発環境は, 表 5.1 に示すとおりである.

開発環境	開発要素	備考
OS	WindowsXP Professional	Service Pack 3
言語	C	Windows プログラミング
API	Win32API	ユーザーモード
	NativeAPI	カーネルモード
コンパイラ	Visual C++	アプリケーション統合開発
	WinDDK(7600.16385.0) x86 Free Build Environment	ドライバ開発

表 5.1: システムの開発環境

また, ドライバのビルドとセットアップの環境は図 5.2 のようになる.

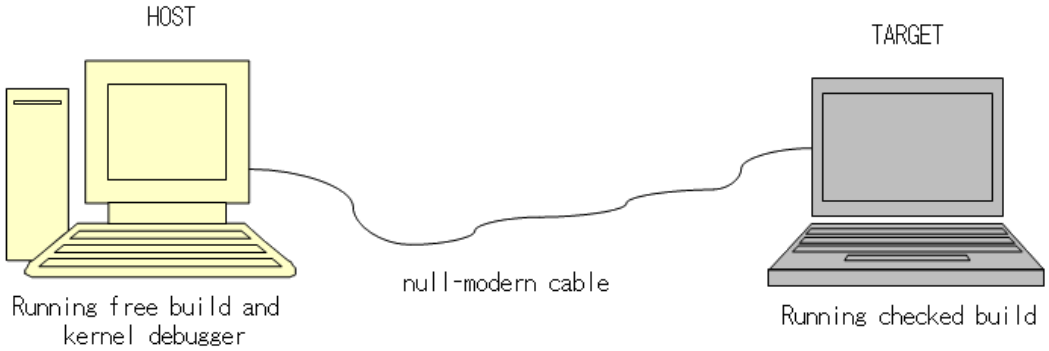

図 5.2: ビルドとセットアップ

出典：WindowXP ファイルドライバプログラミング 入門と実践

ドライバの開発には, Windows Driver Kits の WDK 7600.16385.0 を用いた. また, 表

5.5 に示したとおり, 本システムは将来的な改良の手間を考慮したため, 複数のコンポーネントに分割されている.

コンポーネント	ファイルフォーマット	役割
DSFC.exe	実行可能ファイル (GUI 形式)	ファイルアクセス情報の表示
APIHook.dll	ダイナミックリンクライブラリ	Win32API フック
PsCtl.dll		プロセス情報の収集
PSCPNR.sys	ドライバ	プロセス生成の検知
RgFldr.exe	実行可能ファイル (CUI 形式)	保護フォルダの登録
FlMpp.exe		メモリマップトファイルの作成
formatter.exe		保護フォルダの解除
PIMon.exe		モジュールリストの作成
FLIST.bin	バイナリファイル	保護フォルダリストの保存
WHITELIST.txt	テキストファイル	ホワイトリストの保存

表 5.2: フォルダ保護システムの構成コンポーネント

DSFC.exe は, ユーザーに対して GUI 形式によりターゲットプロセス情報を表示するメインアプリケーションとなる.(表 5.5) このアプリーケーションを通じて, ユーザーはコンピュータ上におけるファイルアクセスをログ化することが可能になる.

APIHook.dll と PsCtl.dll は生成されたターゲットプロセスに対してインジェクトされ, ターゲットプロセスの仮想メモリ空間上で動作するダイナミックリンクライブラリ (DLL) である. RgFldr.exe は保護フォルダの登録をリスト構造体の形で行い, formatter.exe はそれらの保護フォルダの登録解除を行う.

FlMpp.exe は各コンポーネント間の通信のためにメモリマップトファイルの作成を行う. FLIST.bin は保護フォルダの構造体情報を記録するためのバイナリファイルであり, WHITELIST.txt はホワイトリストを登録するためのテキストファイルである. PIMon.exe は, ターゲットプロセスに対してロードモジュールリストの作成や, ユーザーが独自に開発した DLL のリンクを行うためのツールである.

図 5.3 のように, フォルダ保護システムは複数のコンポーネントの連携によって, 成り立っている.

5.3 適用手法

ターゲットプロセスの生成は, ユーザー空間とカーネル空間の両者での検知が可能であるが, システムをどちらかの空間に絞って, 実装するのは得策ではない. ユーザー空間での検知システムは, 実装こそ容易であるものの, WindowsOS におけるイベント観測粒度が粗く, 厳密さが問われる本研究には, あまり向いていない. カーネル空間での検知は精度はよいが, システムをカーネル空間のみで実装することは NativeAPI の非公開ドキュメントの存在などから極めて困難である.

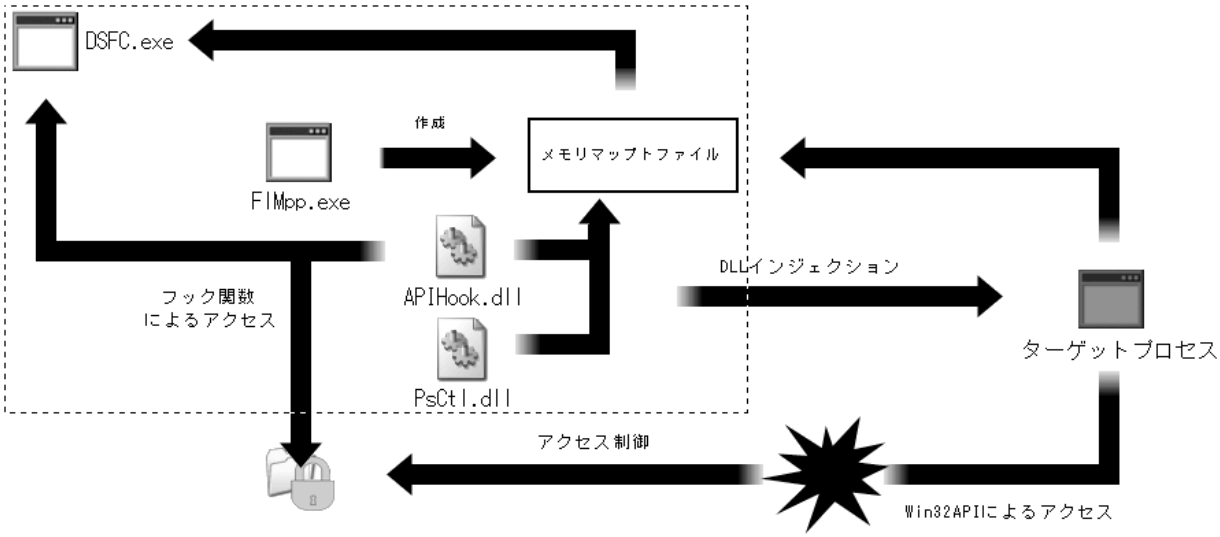

図 5.3: 構成コンポーネントの連携

そこでユーザー空間とカーネル空間に跨る実装としたが, ユーザーモードとカーネルモードの切り替え手続きが煩雑化するとコンテキストスイッチによる無駄な CPU サイクルの消費が考えられるため, 両者空間の通信に工夫が必要となる. 図 5.4 に，提案システムの処理手順を示した.

本システムは，常駐型である. ターゲットプロセスが生成されると, システムの構成コンポーネント PSCPNR.sys が生成を検知し, メインアプリケーションである DSFC.exe にプロセス情報を送信する. DSFC.exe は, メインスレッドの他に二つのスレッドを持つが, これらのスレッドを連携させてプロセス情報を元にした DLL インジェクションを行う. APIHook.dll にはターゲットプロセスの Win32API をフック関数に置換するための制御コードが記述されており, この処理によりターゲットプロセスの API フックが完了する. API フックが施されたターゲットプロセスは以後, ターゲットプロセスの発行する Win32API に関する情報を自ら DSFC.exe へと送信する. これにより, ターゲットプロセスが行うファイルアクセスの監視・制御が可能となる.

5.4 ユーザー空間でのプロセス生成検知

ユーザーモードによるターゲットプロセスの生成検知には, Windows の機能として提供されている SetWindowsHookEx 関数によるものと, タイマを用いたプロセス列挙による手法が存在する.

- WH_SHELL を用いた SetWindowsHookEx 関数
  SetWindowsHookEx 関数は, マイクロソフトが公式に提供する WindowsOS 上でのイベントをフックするための関数であり, フックプロシージャをフックチェーン

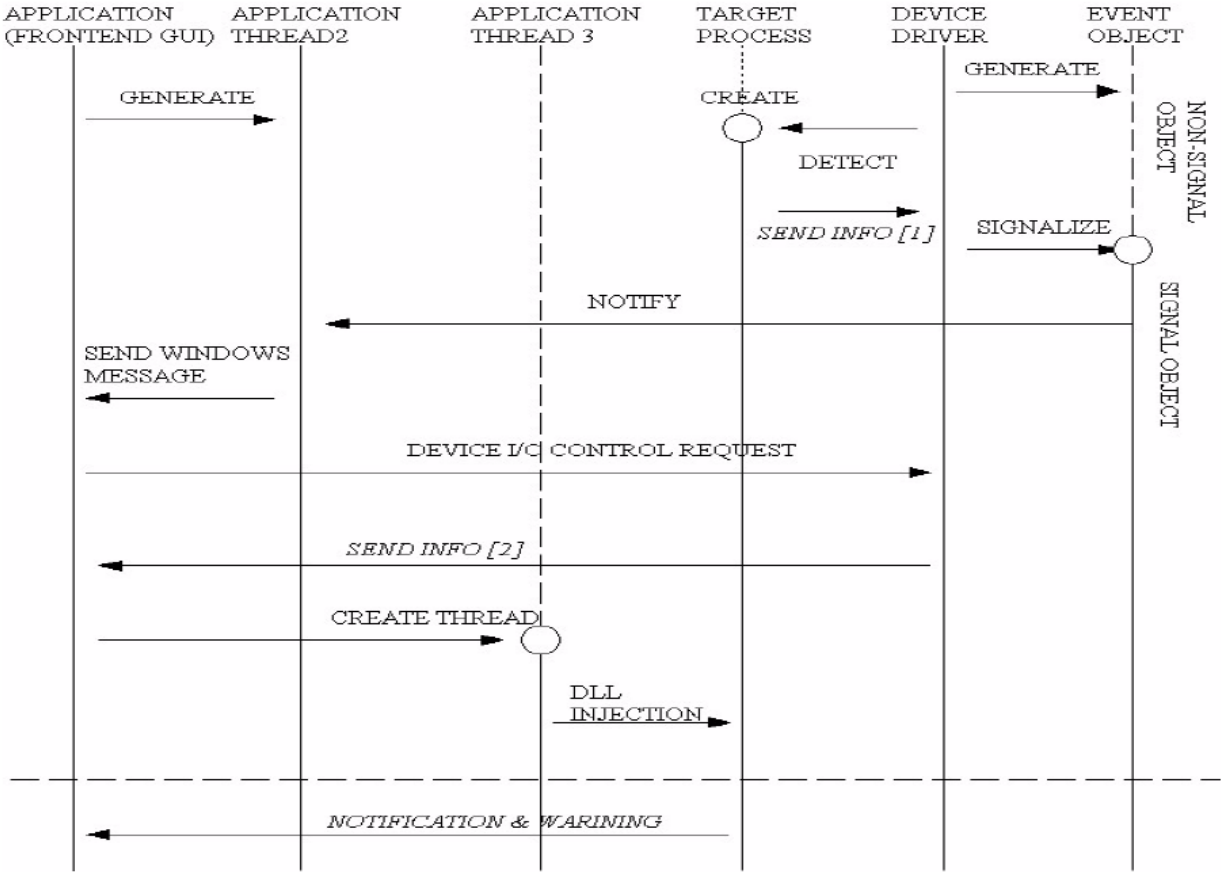

図 5.4: 本システムのシーケンス

にインストールすることで, Windows 上における特定のイベントを監視し, イベントの通知を受け取った際に, フックプロシージャに処理を横取りさせることが可能になる.[12]

```
HHOOK SetWindowsHookEx(
    int idHook, // フックタイプ
    HOOKPROC lpfn, // フックプロシージャ
    HINSTANCE hMod, // アプリケーションインスタンスのハンドル
    DWORD dxThreadId // スレッドの識別子
);
```

フックタイプにパラメータとして WH_SHELL をセットした場合, この関数はオペレーティングシステムからシェルイベントの通知を受け取った際にフックプロシージャに処理を渡す. また, WH_DEBUG をセットした場合には, デバッグの際に有用なフックプロシージャをインストールすることが可能である. そのため, これらのイ

ベントが発生した際に, 独自の処理を行うことができる. ただし, この関数はイベントの検知に Windows メッセージ機構を用いているため, ウインドウを持たない CUI アプリケーションなどには適切な処理を仲介させることができないことがある.

- タイマを用いたプロセス列挙
  プロセスの列挙をタイマを用いて常時行うことで, プロセスカウントの増減及びプロセス名のチェックし, プロセスの起動を検知する. Windows タスクマネージャが用いるなど, 一般的な手法である. プロセスの列挙には EnumProcesses 関数や CreateToolhelp32Snapshot 関数などを用いることができる.
  提案するシステムのようにターゲットプロセスが生成した瞬間を厳密に捉えなければらない場合には, タイマの間隔を分解能における最小限値に設定する必要があり, CPU に大きな負荷をかける恐れがあるため, 注意が必要である.

5.5 カーネル空間でのプロセス生成検知

カーネル空間でのターゲットプロセスの生成検知は, ドライバを用いて行う. PsSetCreateProcessNotifyRoutine はカーネルモードで動作する関数であり, ターゲットプロセスの生成を検知した際に, コールバック関数による処理を挟むことができる. 生成イベントを検知した際に, コールバック関数内でプロセス構造体にプロセス情報を代入し, イベントオブジェクトのシグナル化及び, I/O 要求を行うことで, ユーザー空間のアプリケーションにおいてもプロセスの生成の検知が可能となる.

5.5.1 Process Structure Routine

ターゲットプロセスをカーネル空間で検知し, ユーザー空間で動作する本システムのメインアプリケーションへと通知するまでの一連の流れを, 本論文では Process Structure Routine と呼ぶことにする. また, カーネル空間とユーザー空間を跨いだデータ転送に用いられる IRP のバッファに格納される構造体を, プロセス構造体と呼ぶ. プロセス構造体には, PID をはじめとした基本情報の他に, 様々な情報を格納する. ここでは, Process Structure Routine の構造を順を追って解説する.

5.5.2 コールバック関数の定義

PsSetCreateProcessNotifyRoutine 関数を実行することで，カーネル空間でターゲットプロセスの生成が検知可能となる. この関数は Ntddk.h において,

```
NTSTATUS PsSetCreateProcessNotifyRoutine(
    __in PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine,
    __in BOOLEAN Remove
);
```

として定義されており，ターゲットプロセスの生成が検知されると，第一引数に登録されたコールバック関数を呼び出す.

また, コールバック関数は, 以下のような定義となっている.

```
VOID (*PCREATE_PROCESS_NOTIFY_ROUTINE)(
    IN HANDLE ParentId,
    IN HANDLE ProcessId,
    IN BOOLEAN Create
);
```

このコールバック関数に渡される引数を, コールバック関数内の処理でチェックすることによりターゲットプロセスを把握することが可能となる.[12]

5.5.3 ドライバのインストール

ドライバは, PsSetCreateProcessNotifyRoutine 関数を用いて, ターゲットプロセスの生成及び終了の検知を行う. ターゲットプロセスの生成終了時, PsSetCreateProcessNority-Routine は, コールバック関数を呼び出す. コールバック関数の内部処理では, 生成されたプロセスを I/O 要求パケットして IRP のバッファの中に埋め込み, さらに検知を行ったサインとしてイベントの状態をシグナル化する. シグナルの合図は, カーネル空間とユーザー空間を跨ぐため, シグナル状態の変化を読み取った本システムのメインアプリケーションでは, このドライバに対して IRP の要求を行うタイミングを把握することが可能になる.

尚, カーネル空間とユーザー空間において, 共有メモリを用いたデータ転送を行う方法も存在するが, CreateSection 関数などの NativeAPI の仕様は完全に公開されているわけではなく, 実装が困難である.

ドライバは通信に対してイベントを作成するが, このイベントのアクセス権限 DACL, ACE とアクセスマスク, ZwCreateEvent のセキュリティ記述子の部分を正確に記述することは困難であるため, 同期を主目的としたイベントであることを明示するため, SYN-CHRONIZE のみを許可する. イベントの通知のみの使用目的であれば, SYNCHRONIZE のみの許可であっても, アクセス違反が発生することはない.

また, ドライバには自動でのロードを選択する. レジストリにおいて,

```
¥HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥
```

のキーにおける項目を変更し, system32¥drivers 以下にドライバを配置することで, オペ

レーティングシステム起動の際, アプリケーションの立ち上がり前にドライバをインストールすることが可能になる.

5.5.4 イベントオブジェクトを利用した通信

ターゲットプロセスの生成はカーネル空間で検出されたあと，イベントオブジェクトのシグナル化及び I/O 要求を用いて，ユーザー空間の本システムのメインアプリケーションへと通知される．イベントオブジェクトはシグナルのフラグであり，シグナルの状態を変更することで，ユーザー空間とカーネル空間の壁を越えてイベントの共有がなされる.

本システムのメインアプリケーションはセカンダリスレッドにおいて WaitForMultipleObjects 関数を用いてターゲットプロセス生成に関するイベントを待機しており, ドライバはターゲットプロセスが生成された際にデータ転送開始のためにこのイベントオブジェクトのシグナル化を行う. シグナル化によって, ユーザー空間へターゲットプロセス生成の検知が通知される.

5.5.5 Windows メッセージを用いた通信

第 5.5.2 項の検知によるプロセス構造体を, 本システムのメインアプリケーション内のプライマリスレッドへ伝達するために, セカンダリスレッドは Windows メッセージを発行する. Windows メッセージはプライマリスレッドで処理されるため, 最終的にはターゲットプロセスの生成がメインアプリケーションのプライマリスレッドに通知される.

メインアプリケーションはこの直後, プライマリスレッドにおいて DeviceIoControl 関数を用いてドライバへの I/O 要求を行う．この I/O 要求によって, ドライバからはプロセス構造体がバッファに格納された IRP が返信されるため，メインアプリケーションはプロセス構造体を取得することが可能になる．この情報をもとに，システムはターゲットプロセスへ対して DLL インジェクションを行う.

メインアプリケーションは, 事前にセカンダリスレッドを作成しておくことで, WaitForMultipleObjects 関数を用いてイベントの待機を行っている．セカンダリスレッドは, WaitForMultipleObjects 関数を呼び出したところ待機した状態となり, イベントのシグナル化で再び制御を返しコードを実行する. メインアプリケーションのプライマリスレッドではウインドウプロシージャが常時 Windows メッセージを処理しながら処理を行っているが, セカンダリスレッドはこのプロシージャに他のメッセージと競合することがない独自に定義したメッセージを送信する. プライマリスレッドでは, セカンダリスレッドの独自に定義されたメッセージを処理し, メッセージ内で I/O 要求を行う.

ただし, Windows メッセージ機構は First In First Out 方式である非同期のキューを用いているため, ターゲットプロセスが検知された順序でプライマリスレッドでの処理が行われていることが保証されない.

5.5.6 IRP を利用したプロセス構造体の受け渡し

IRP とは Windows オペレーティングシステムがユーザ空間におけるアプリケーションからカーネル空間におけるドライバへの I/O 要求を処理する際に用いるパケットである. 実体は, アプリケーションから I/O 要求が発生した際に, I/O マネージャが作成する構造体であり, それらは非ページプールメモリに割り当てられている. IRP は, ヘッダ部分とスタックロケーション部分から成り立っている.

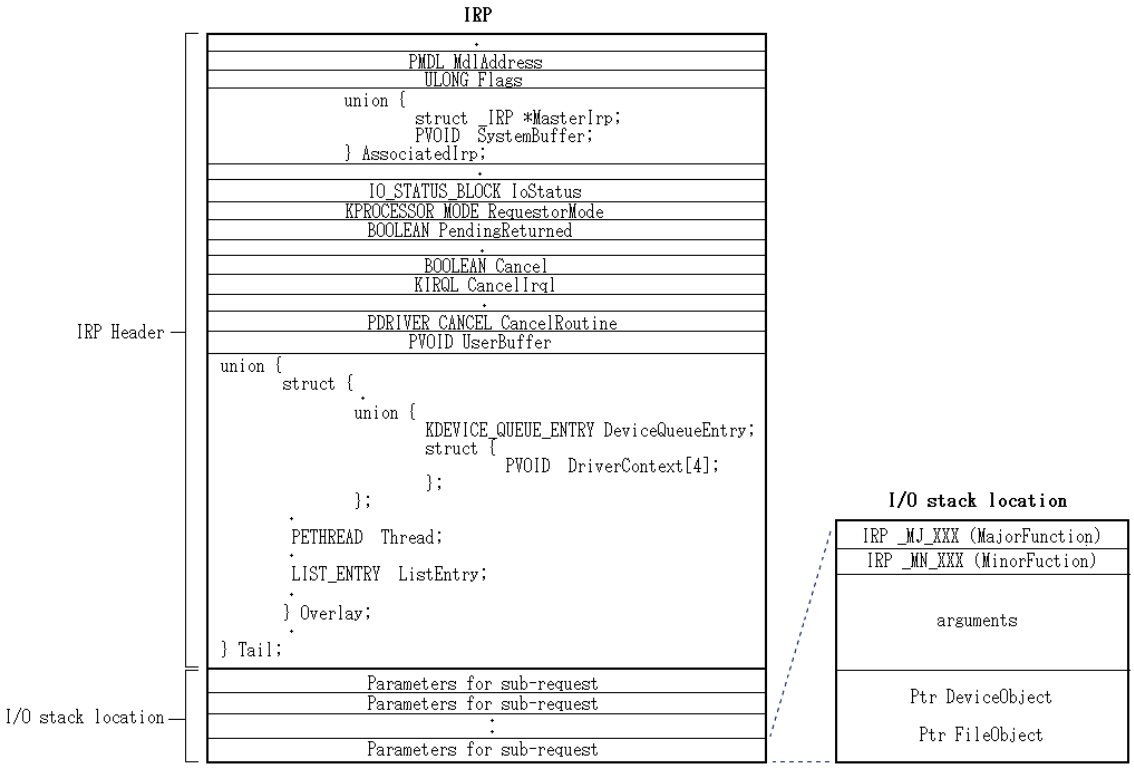

図 5.5: IRP の構造

IRP は, アプリケーションから I/O 要求がありドライバとのデータ転送を行う際に, データバッファを用いる. データバッファへのアクセス方法には, METHOD_IN_DIRECT(METHOD_OUT_DIRECT), METHOD_NEITHER, METHOD_BUFFERED の 3 つがある.[13]

- METHOD_IN(OUT)_DIRECT によるダイレクト I/O
  I/O 要求の際にロックされたユーザーバッファからメモリディスクリプタリスト (MDL) を作成し, MDL 構造体のアドレスをドライバが用いることで, ユーザー空間とカーネル空間のデータ転送を行う手法である. MDL 構造体は, ユーザーモードで動作するアプリケーションが用いる連続していないデータバッファの物理メモリ内のページへのポインタの集合体であるため, ドライバはユーザーモードのデータバッファを直接操作することが可能である. データ量が多い転送を行う際には有効である.

- METHOD_BUFFERED によるバッファI/O
  I/O 要求の際に, I/O マネージャがユーザーモードバッファと同じサイズのシステムバッファを非ページプールメモリに割り当て, ドライバが提供するデータをシステムバッファからユーザーバッファにコピーすることでデータ転送を行う方法である. ドライバはカーネルモードで動作するため, システムバッファのみにアクセスすることができる. システムバッファからユーザーバッファにコピーをとる手順が挟まれるため, アプリケーション側でユーザーバッファを固定長で定義する必要があり, ドライバでプロセス情報をリスト構造体で保持することができなくなるデメリットがある. 比較的容易に実装できることから, 本システムにはこの方法を採用している.

- METHOD_NEITHER によるニーザー I/O
  I/O マネージャが, 要求元のアプリケーションのデータバッファの仮想アドレスをドライバへ提供することでデータ転送を行う方法である. ただし, ドライバがカーネル空間において最上位にあるときのみに使える方法であり, 特殊なケースを除き使用を控えたほうがよい.

5.6 リモートスレッドによる DLL インジェクション

DLL インジェクションとは, 任意のプロセスに本来リンクされていない DLL をマッピングさせる手法である. 具体的には, ターゲットプロセスにリモートスレッドを作成し, その独自に作成したスレッド内で DLL を明示的にリンクさせることで実現する.(図 5.6)

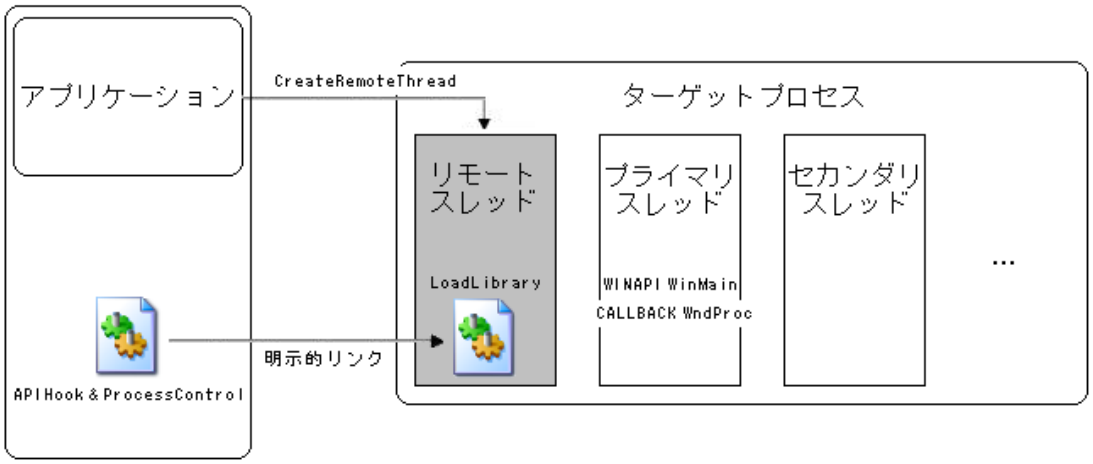

図 5.6: リモートスレッドによる DLL インジェクション

IRP に埋め込まれたデータバッファのプロセス構造体により, アプリケーションは生成されたプロセスに DLL インジェクションを行うことが可能になる. DLL インジェクションは ToolHelp32 関数を用いて行う.[5]

ところで, メインアプリケーションは生成されたターゲットプロセスに対して, Kernel32.dll や User32.dll といった既知の DLL がリンクされる前に DLL インジェクションを行ってはいけない. これらのリンク前にはアプリケーションの初期化がうまく行われておらず Win32API が IAT に格納されてないことから, API フックが失敗することがあるためである.

5.7 Win32API における API フック

ターゲットプロセスが Win32API を用いる場合, そのプロセスのインポートセクション内にあるインポートアドレステーブル (IAT) を書き換えることで, 本来呼び出される Win32API の代わりに異なる独自のフック関数を呼び出すことが可能になる. この手法は一般的に API フックと呼ばれている

API フックの処理をコンストラクタ内で記述した DLL をインポートさせると，DLL はターゲットプロセスへのアタッチ時に自動実行され, ターゲットプロセスの IAT を書き換える. API フックはターゲットプロセスの持つ仮想アドレス空間内の IAT を書き換えているため，プロセスが終了するまで有効である．この技術は特定の CPU の仕様に依存しないため，柔軟な利用が可能である.

5.8 警告表示とアクセス制御

API フックが成功した後は, ターゲットプロセス自ら呼び出した Win32API に関する情報を本システムへと送信するようになる. また, フック関数の中でパラメータのチェックを行い, 保護フォルダへのアクセスがみられた場合にはモーダル型のダイアログボックスを表示させることでユーザーに対してファイルアクセスの制御を促すことができる. モーダル型にすることで, アクセス制御を促すメッセージが表示されている間はターゲットプロセスのファイルアクセスを実行するスレッドの処理が一時停止する. このため, ユーザーはファイルアクセスの直前で制御の選択を行うことが可能になる. 表 5.3 にそれら選択肢と, その結果を纏めた.

選択肢	結果
実行	ファイルアクセスが実行される (本来の動作)
強制終了	ファイルアクセスが行われないまま終了する
スリープ	ファイルアクセスが行われる直前で一時停止する
遮断と続行 (一回)	ファイルアクセスを一回遮断し, 続行する
遮断と続行	ファイルアクセスを毎回遮断し, 続行する

表 5.3: ファイルアクセスに対する制御の選択肢

ただし, 本来行われるべきファイルアクセスを中断することは, ターゲットプロセスの動作だけでなく, オペレーションシステムに不具合をもたらす原因にもなりうるため, 慎重な判断のもとの制御が必要となる.

5.8.1 ターゲットプロセスとシステム間におけるデータ転送

第 5.6 節において, DLL インジェクションにより任意の DLL をターゲットプロセスに注入することができることを述べたが, この DLL とシステムのメインアプリケーション間においてデータ転送を行うことによって, 他プロセスからはアクセスの権限がないターゲットプロセスの仮想アドレス空間における様々な情報を取得することが可能になる.

本システムの構成コンポーネントの一つである FlMpp.exe は, このプロセス通信を実現するためのモジュールである. WindowsOS ではプロセス通信を実現するための仕組みが幾つか用意されているが, このモジュールはメモリマップトファイルを利用してプロセス間の通信を実現している.

5.8.2 メモリマップトファイルによるデータ転送

メモリマップトファイルとは WindowOS に用意されたファイルをあたかもメモリのように扱うための仕組みである. WindowsOS では, メモリ空間とファイル空間におけるデータの配置方法が同じであるため, メモリマップトファイルを用いるとファイルを仮想メモリとして扱うことが可能になる.(図 5.7)

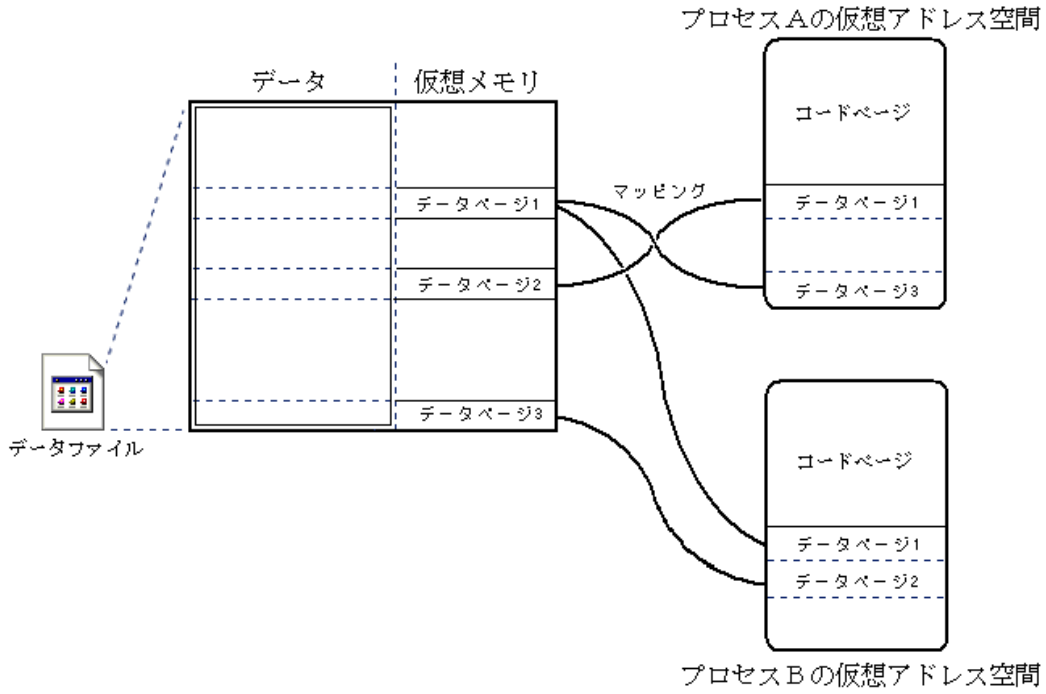

図 5.7: メモリマップトファイルの仕組み

メモリはアドレスを持っておりアドレスによりデータへのアクセスが可能になるが, ファイルはアドレスを持っていない. メモリマップトファイルはこの差異を取り払う仕組みを持つ.

メモリマップトファイルが複数のプロセス間で用いられる場合, データの読み書きに整合性をもたせるために, ミューテックスなどの同期機構を用いなければならない. 仮に非同期で仮想メモリに対して読み書きを許した場合, 複数間のプロセスによるやり取りの途中でデータが破損する恐れがある. しかし, ミューテックスの切り替え処理は同一プロセスにおけるクリティカルセクションなどに比べて実行速度がはるかに遅いので, 多様は避けるべきである.

5.9 保護フォルダの登録

保護対象のフォルダの登録を行うために, バイナリデータにフォルダのフルパスの構造体を埋め込む. テキストデータではなく, バイナリデータとしての形式を選択した理由は, 将来的な保護フォルダへの属性情報を構造体として扱いやすくするためであり. この形式により保護フォルダの登録データをシステムのメインアプリケーションから容易に参照することが可能になる.

今回は, Windows エクスプローラーが提供するコンテキストメニューから保護フォルダを登録させる形をとることで, ユーザーに対する負担を軽減する. コンテキストメニューからの実行可能ファイルの呼び出しはレジストリを変更することで実装する.

レジストリキー	種類	データ
¥HKEY_CLASSES_ROOT ¥Folder¥shell¥{*KEYNAME*}	文字列	エントリ名
¥HKEY_CLASSES_ROOT¥Folder ¥shell¥{*KEYNAME*}¥command		実行可能ファイルのパス

表 5.4: コンテキストメニュー追加のレジストリキー

5.10 ホワイトリストの活用

ホワイトリストは, テキストファイルに付加情報とともにプロセス名を列挙することで実装する. 登録されたプロセスは, ファイルアクセスの一部または全てにおいて API フックの影響を受けないなど, 予めユーザーが定義した挙動のもとで動作する. 証明書などにより安全性が確保されているプログラムについては既知のプログラムとして扱い, 警告表示やアクセス制御を行いことでユーザーの混乱を防ぐ目的がある. また, オペレーションシステムにプリインストールされているアプリケーションなど, 安全性が明らかであるものについては開発者側で予めリストへの登録を行っておく.

5.11 ターゲットプロセスのモジュール列挙

ターゲットプロセスがリンクしたDLLを一覧として列挙する機能は, CreateToolhelp32Snapshot関数を用いて実装する. この関数は, プロセスが使うモジュールのスナップショットなどを作成することが可能である.

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

この関数の dwFlags に TH32CS_SNAPMODULE を指定することで, th32ProcessID に指定したプロセスが読み込んだモジュール列挙が可能となる.

また, LoadLibrary フックにより, ターゲットプロセスが明示的に DLL をリンクした時刻をタイムスタンプとして生成しておき, モジュール名とともにリスト構造体として保持しておく. このようにすることで, ターゲットプロセスが扱う DLL の明示的リンクのタイミングを把握することができ, 潜伏期間を持つウイルスの活動開始タイミングの調査や, 他プロセスからの DLL インジェクションの有無の検査が可能になる.

5.12 システムの配布

本システムは, レジストリも活用するため, 各要素の個別のインストールをユーザーに任せるのは適切ではない. そこで, MSI 形式のファイルとして配布することで, ユーザーの手間を省いている.

要素	レジストリ
コンテキストメニューへの追加	¥HKEY_CLASSES_ROOT¥Folder¥shell¥
スタートアップへの登録	¥HKEY_LOCAL_MACHINE¥Software¥Microsoft ¥Windows¥CurrentVersion¥Run
ドライバの追加	¥HKEY_LOCAL_MACHINE¥System ¥CurrentControlSet¥Services¥

表 5.5: レジストリにおけるシステムの要素

5.13 レジストリによる USB ストレージの無効化

USB ストレージのドライバを無効化することで, USB 接続のストレージを介したファイル漏えいを防止する.

この無効化はレジストリの値を変更することで実現する. 本システムが該当のレジスト

リを変更すると, USB 接続の際にドライバがロードされずにハードウェアのインストールが失敗する. これにより, 以後 USB ストレージは認識されなくなる.

実装には, RegCreateKeyEx 関数などを呼び出し,

¥HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥UsbStor

の DWORD 値を変更する処理を記述すればよい.

5.14 漏えい時を想定した設計

漏えいの予防策だけでなく, 漏えい後のシナリオを想定した設計にすることで, より安全なファイル管理を行えるようにシステムを構築する.

一般的に, ユーザーがフォルダ間でファイルをやりとりする場合, ドロッグアンドドロップなどでは Windows エクスプローラがその役目を担うため, エクスプローラの実体である explorer.exe に対して特殊な DLL をインジェクションすることで, ユーザーのファイル操作とそれ以外のプロセスなどの操作を区別することが可能になる.

explore.exe にのみインジェクションする DLL には, ファイルの操作の直前で暗号化及び復号の処理を挟みこむ. このような実装にすることで, 保護フォルダ内のファイルは常に暗号化される状態となり, また, ユーザーがドロッグアンドドロップによる操作でファイルを取り出したときのみ復号の処理が行われるようになる. 何らかの原因によりファイルが漏えいした場合にも, 漏えいファイルが暗号化されているため, 容易に中身を読み取られることはない.

5.14.1 暗号化と復号

暗号化には, WindowsNT 系 OS におけるフォルダセキュリティ機構である Encrypting File System(EFS) を利用する. EFS は, NT File System(NTFS) が持つファイル暗号化機構であり, 通常はフォルダやファイルに対して, プロパティの詳細設定や Win32API を用いることで暗号化属性を付加することができる.

今回は, 物理ドライブとしてリムーバブルメディアである USB メモリ, 論理ドライブとしてプライマリハードディスク上に拡張パーティションとして作成したドライブを用いた. そのうえで, EFS による暗号化が施されたテキストファイルを対象ドライブ内のパラメータとして操作する際に, どのような許可がなされるかを調査し, 対応表に纏めた.

表 5.6 の結果から, EFS はパーティションで論理的に区切られたドライブに対してはファイルアクセスを行う Win32API に対して有効であることがわかるが, USB メモリなどのような物理的に区切られたドライブに対しては無効であることがわかる. また, EFS ファイルには READ/WRITE 操作が可能であるため, 直接これらの操作によりファイルの中身を USB メモリのファイルに書き出された場合には漏えいの可能があることにも注意しなければならない.

ドラッグアンドドロップで USB メモリにコピーや移動を試みると, 図 5.9 のような警告

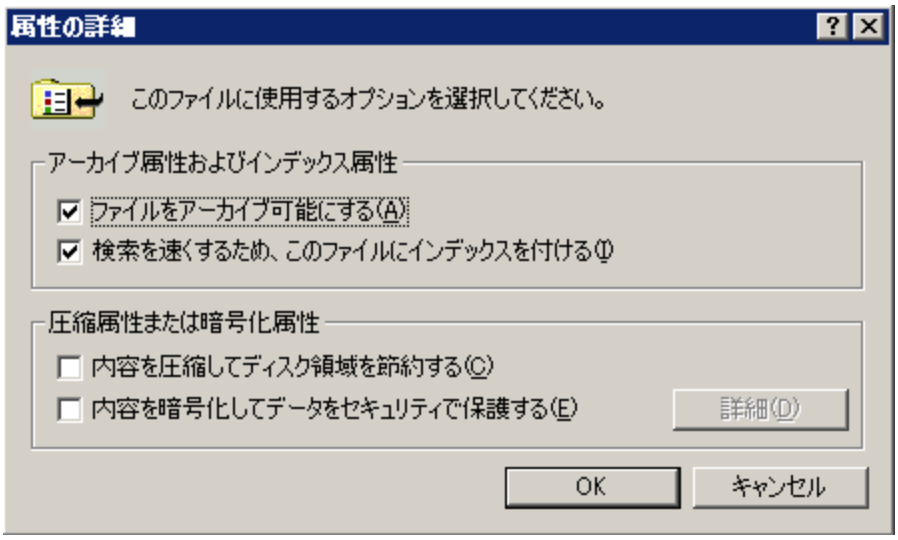

図 5.8: EFS(Encrypting File System) の設定画面

	プライマリ HDD	プライマリ HDD(拡張)	USB メモリ
CreateFile	○	○	-
ReadFile	○	○	-
WriteFile	○	○	○
DeleteFile	○	○	-
MoveFile	○	○	× (関数の失敗)
CopyFile	○	○	× (関数の失敗)
Drag and drop	○	○	△ (暗号化の解除)

表 5.6: EFS におけるファイル移行の可否

が表示される. 無視を選択することで暗号化の設定を解除することが可能なため, 第三者による物理的な持ち出しが可能となってしまうことがある.

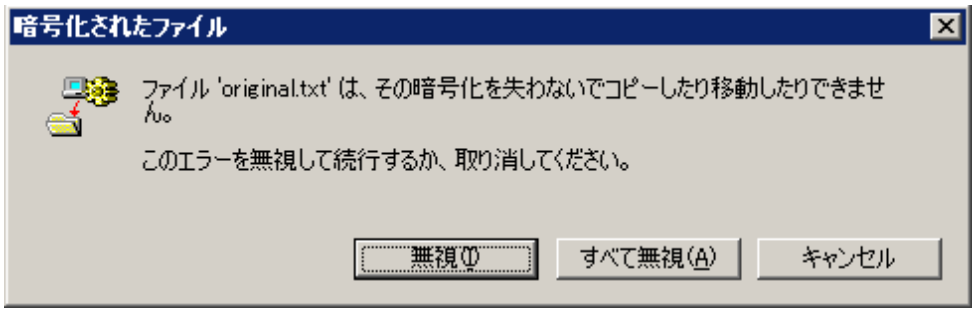

図 5.9: 暗号化の解除を伴うコピーと移動

そのため, EFS ファイルに関しては WriteFile 関数とドラッグアンドドロップによる, 論理

ドライブへの移動やコピーの監視のみを行わせる. 今回のシステムでは, explorer.exe が行うファイル操作にのみ, EFS による暗号化処理を施すことにする. 具体的には, explorer.exe に Win32API フックを行い, ファイル操作の処理の直前に EncryptFile 関数を挟みこむ. また, ファイルの取り出しの際には直前に DecryptFile 関数を挟むことで, 復号処理にあたる暗号化設定の解除を行うことを可能にする.

5.15 まとめ

本章では, 提案したシステムの設計をもとに具体的な実装を行い, その手法を纏めた. 本システムはユーザ空間とカーネル空間に跨るシステムである. また, 複数のコンポーネントの連携によって成り立つため, これらのコンポーネント間及びターゲットプロセスとの同期が重要な要素となっている. 第 6 章では本章で実装したシステムの性能評価を行う.

第6章　性能評価

第 5 章では, 本研究のファイルアクセス検知・制御手法を用い, フォルダ保護システムを実装した. 本章では, このシステムに対する性能評価を行う.

6.1　CPU への負荷の測定

	詳細
保護フォルダ	C:¥Documents_and_Setting(サブフォルダを含む)
ブラウザ	Safari.exe
動画サイト	YOUTUBE
計測時間	20 分
CPU	Intel Core 2 Quad Q6600 2.4Ghz
測定ツール	パフォーマンスモニタ (管理ツール)

表 6.1: CPU 負荷率の測定環境

フォルダ保護システムの CPU への負荷を測定するために, システムとブラウザを用いて対照実験を行った. 動画サイトの同一動画を再生し, 保護フォルダ以下の一時フォルダに対してブラウザにファイルアクセスを行わせることで, ファイルアクセス監視時の負荷を測定した.

6.1.1　低負荷でのシステム常駐

負荷測定には Windows 標準機能のパフォーマンスモニタを利用し, パフォーマンスモニタ上で CPU カウンタの追加を行うためにパフォーマンスオブジェクトに Processor を指定した. システムは測定中の 20 分間にブラウザによる複数回のファイルアクセスを検知しているが, これらのファイルアクセスは C:¥Documents_and_Setting 以下の Application ¥Data に対して行われ, 一時ファイルを対象にしていた.

パフォーマンスモニタの結果を図 6.1 及び図 6.2 に示した. 図 6.1 の測定グラフより, 測定中の 20 分, ブラウザ単体の CPU 負荷率は 5%未満に留まっており, 起動時以外にグラフに特徴的な部分が見られずほぼ一定の負荷率であることから, ファイルアクセス自体の CPU への負荷はほとんどないものと推測される.

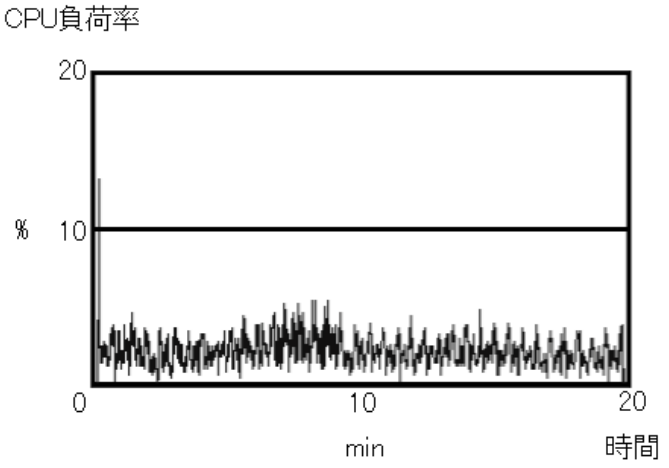

図 6.1: ブラウザを稼動させた時の CPU 負荷

また, 図 6.2 を, 図 6.1 のブラウザ単体の場合と比較しても, システムの常駐による CPU 全体への負荷率は大きな増加を見せていないことがわかった.

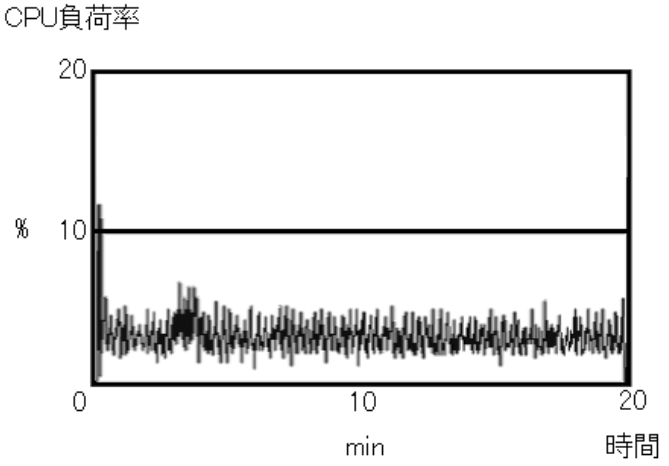

図 6.2: システムとブラウザを稼動させた時の CPU 負荷

このように, 本システムの CPU への負荷は極めて小さいことがわかり, システムがオペレーティングシステムの動作に影響なく, 常駐可能であることを確認した.

6.1.2 CPU への負荷に対する考察

本システムは Win32API にフック処理を行うことでターゲットプロセスが行うファイルアクセスの監視を行いつつ, 同時にフック関数にセットされるパラメータのチェックを行うことでファイルアクセスの制御も行う. このパラメータのチェックは, 文字列検索であるため, CPU に対する負荷はオペレーティングシステム全体の稼動に対して影響が少ないものと思われる.

フック関数に一度置き換えることで, 以後のターゲットプロセスのファイルアクセス関数呼び出した時には, 必ずフック関数が先に呼び出される. この処理はインポートアドレステーブルのアドレスの書き換えであるため, システムはターゲットプロセスの生成時の API フック時以外は例外を除きターゲットプロセスとやり取りを多く行わない. そのため, ターゲットプロセスやオペレーティングシステムに負荷をかけずに常駐させることが可能となる.

6.2 ターゲットプロセスへの影響

本システムはターゲットプロセスへ直接 DLL インジェクションを行いターゲットプロセスの挙動を制御することで, ファイルアクセスの監視と制御を行っている. そのため, Win32API がフック関数へと置き換えられたターゲットプロセスへの影響を考慮しなければならず, これらについての検証を行った. ターゲットプロセスへの影響としては, プロセスの実行速度や状態の変化が考えられる.

6.2.1 ターゲットプロセスの実行速度への影響

ターゲットプロセスの実行速度への影響を検証するために, 通常時とフック起動時における, 各 Win32API の処理速度の測定を行った.

各 Win32API の処理速度は非常に高速であるため, マイクロ秒単位での測定が行えるよう, 高分解能パフォーマンスカウンタを選定した. このカウンタを扱うために, QueryPerformanceFrequency 関数と QueryPerformanceFrequency 関数を用いている.(表 6.2)

関数	詳細
QueryPerformanceFrequency 関数	高分解能パフォーマンスカウンタの周波数を 64bit 値で取得
QueryPerformanceCounter 関数	高性能パフォーマンスカウンタの現在の値を 64bit 値で取得

表 6.2: 高分解能パフォーマンスカウンタを扱うための関数

この QueryPerformanceCounter 関数で処理速度を測定した各 Win32API 関数を挟み込み, 差分をとりシステムの周波数から実際の秒数を割り出すことで, 処理速度を計測している. 具体的な高分解能パフォーマンスカウンタによる処理速度の測定方法を, 表 6.3 に纏めた.

表 6.4 は, ファイルアクセスに関する各 Win32API について, その測定結果を纏めたものである.

関数	詳細
処理	QueryPerformanceFrequency(&nFreq); QueryPerformanceCounter(&nBefore); 処理速度を測定したい Win32API 関数; QueryPerformanceCounter(&nAfter);
計算式	$\frac{\text{(nAfter.QuadPart - nBefore.QuadPart) * 1000}}{\text{nFreq.QuadPart}}$

表 6.3: 処理速度の測定方法

	ターゲットプロセスの処理速度	
	通常時	フック時
CreateFile	25.7 回/ms	0.9 回/ms
ReadFile	170.1 回/ms	0.8 回/ms
WriteFile	69.1 回/ms	0.9 回/ms
DeleteFile	2.3 回/ms	0.4 回/ms
MoveFile	1.2 回/ms	0.6 回/ms
CopyFile	10.8 回/ms	0.2 回/ms

表 6.4: 各 Win32API の処理速度

6.2.2 実行速度への影響に対する考察

Win32API のそれぞれの処理速度は, 通常時とフック時において有意な差が見られており, フックにおける呼び出しの遅延には何らかの対策をとらなければならない. この遅延は, システムを用いた際にターゲットプロセスで稀に発生するアプリケーション初期化エラーの原因の一つとして考えられる.

6.2.3 ターゲットプロセスの状態への影響

ターゲットプロセスへの状態の影響を検証する. プリインストールされたプログラムに本システムを適用し, その起動から終了までの間にどのような影響があるかを纏めた.

6.2.4 状態への影響に対する考察

一般的なアプリケーションは, 開発者が Kernel32.dll や User32.dll をリンクしなくとも, オペレーティングシステムにより自動的に Kernel32.dll や User32.dll といった DLL をリンクする. これは, Kernel32.dll や User32.dll などの中に, 様々なオペレーティングの基本機能を呼び出す Win32API が含まれているためである. 本システムは, ターゲットプロセスの IAT を変更し, Win32API に対してフック処理を行うことで実現していたが, IAT に読み込まれた DLL のアドレスを置換する仕組み上, Kernel32.dll や User32.dll を既に読み

プロセスの状態	影響	備考
起動時	有り	大規模プログラム起動時, 稀にアプリケーション初期化エラー
走行中 (アイドル時)	無し	-
走行中 (ファイルアクセス時)	正常	小規模 CUI プログラムのファイルアクセス時, 稀にファイルアクセスの監視・制御が失敗
走行中 (負荷時)	無し	-
終了時	無し	-

表 6.5: ターゲットプロセスの状態への影響

込んだ状態のプロセスに対してのみ有効である. このため, ターゲットプロセスの生成時直後に闇雲な DLL インジェクションを行っても, Win32API がフック関数に置き換わらない状態でターゲットプロセスが立ち上がる可能性があり, フック処理の成功が保証されない.

第 6.2.3 項においてターゲットプロセスへの影響を纏めた. 小規模 CUI プログラムのファイルアクセス時に稀にファイルアクセスの監視や制御が失敗するのは, API フックの処理が遅れてしまい, フック処理の前にファイルアクセスが行われたか, あるいは API フックの処理が Kernel32.dll や User32.dll のリンク完了前に終了した結果, Win32API が置換されていないためと思われる.

また, 本システムを大規模プログラムに適用した際にもアプリケーションが強制終了や初期化エラーが見られることがあり, これらについては今後の更なる検証が必要とされている.

6.3 ファイルアクセスの検知率

本システムはフォールスポジティブを許容し, できる限り多くのプロセスにおけるファイルアクセスを検知する方式をとっている. このファイルアクセスの検知率を WindowsXP で動作するアプリケーションを用いて測定した. 検知率を表 6.6 に纏めた.

正規アプリケーションに対するファイルアクセスの検知率は 72.7%であった. 今後はこの検知率をできるだけ高めていくことが重要となる. また, 図 6.6 に NG と記載されているファイルアクセスが検知できなかったアプリケーションに対しては, そのエラー原因を特定できなかったため, 今後はこうした原因の究明も行わなければならない.

また, ハニーポットを用いて収集したウイルスから無作為に抽出したものに対するファイルアクセス検知率を本システムで検証した. 実験には 101 検体を用いたが, 100%のファイルアクセス検知率を達成した.

検知率	72.7%	33 件 (正規アプリケーション)
notepad.exe	OK	
sndrec32.exe	OK	
sndvol32.exe	NG	稀に起動不可
osk.exe	OK	
hypertrm.exe	OK	
msswchx.exe	OK	
utilman.exe	NG	アプリケーション初期化エラー
dwwin.exe	OK	
magnify.exe	OK	
charmap.exe	OK	
calc.exe	OK	
msmsgs.exe	OK	
wmplayer.exe	OK	
moviemk.exe	OK	
winmine.exe	OK	
wordpad.exe	NG	起動不可
mspaint.exe	OK	
freecell.exe	OK	
PINBALL.exe	OK	
mshearts.exe	OK	
sol.exe	OK	
spider.exe	NG	終了不可
taskmgr.exe	OK	
explorer.exe	OK	
cmd.exe	OK	
wab.exe	NG	表示に不具合
zClientm.exe	NG	不明なエラー
Rvsezm.exe	NG	強制バックグランド起動
msimn.exe	NG	稀に起動不可
thunderbird.exe	OK	
Safari.exe	NG	稀に起動不可
InternetExplorer.exe	OK	メニューバー表示に不具合
Virtual PC.exe	OK	

表 6.6: 本システムによるファイルアクセス検知率

6.4 まとめ

本章では, フォルダ保護システムへの様々な性能評価を行い, システムが常駐型として有効に活用できるかを確かめた. Win32API 単体の呼び出し速度では遅延が見られるなどシステムに改善の余地があることがうかがえるが, オペレーティングシステム全体では有意な負荷は見られなかった.

また, ウイルス 101 検体に対して 100%のファイルアクセスの検知率を達成したことで, ファイルアクセス検知・制御機構を利用したフォルダ保護システムの有効性を確認した.

第7章　考察

本章では本研究に残された課題に対して考察を行う.

7.1　フォルダ保護システムの自動化

フォルダ保護システムは, 特定の保護フォルダへのファイルアクセスを監視・制御するものであるが, 今後は使用目的に合わせた動的なフォルダ保護の仕組みを構築していく.

本システムは現状, 全てのファイルアクセスを監視・制御するものであるが, ユーザーアカウント単位のみならずコンピュータを使用する物理的なユーザーの識別を行ったうえで, プロセスやファイルをカテゴリ化し保護フォルダ内での効率的なファイル管理を可能にすることを目指す. また, ターゲットプロセスを正規のアプリケーションとウイルスに自動分類し, ユーザーに対しての負担を軽減させる仕組みを構築していく.

さらに, フィルタドライバや NativeAPI フックなどの処理機構を自動選択しターゲットプロセスの起動エラーを防いだり, より効率的なプロセス情報をユーザーに提示するためのコントロール機能も持たせていく.

7.2　クロスプラットフォーム化に対する考察

今回 WindowsXP においてシステムの構築を行ったが, 既に企業は Windows7 などへのオペレーティングシステムへと以降しているため, 将来的な利用を考えるとこれらに対応する必要がある.

また, WindowsNT 系 OS だけではなく, スマートフォンと呼ばれる高機能携帯に搭載された iOS や AndroidOS への対応も検討する. スマートフォンには個人情報のデータがパーソナルコンピュータよりも多く含まれる傾向にあり, 今後増加するであろうセキュリティインシデントに対して迅速な対応が望まれている.

こうしたクロスプラットフォーム化においては, 本システムの仕組みを用いるうえで他の NT 系 OS では DLL インジェクションやドライバとのデータ転送が開発者にどの程度許可されているかを調査しなければならない. また, AndroidOS や iOS は root 権限の取得を一般ユーザーに許可されていないことが多く, プロセスの挙動を監視するためのセキュリティ機構の構築が困難なものとなっており, こうした課題も克服していかなければならない.

7.3 システムにおける同期機構の形成

今回のシステムの実装ではProcess Structure Routineにおけるターゲットプロセスの生成検知の際に, メインアプリケーションとドライバの間でイベントシグナル化によるデータ転送以外の特別な処理を行っていない. このため実装が非同期となり, IRP による I/O 要求が行われる前に別のターゲットプロセスが起動した場合, 先に生成されたターゲットプロセスの情報が破棄される可能性がある. そこで, ターゲットプロセスの検知時に, ターゲットプロセスリストを自己参照構造体として保管しておくことで解決する. ただし, 自己参照構造体によって作られたプロセスリストを保管する場合, この構造体のサイズが固定長ではないことから, IRP の固定長バッファをオーバーフローしないための安全な設計が求められる.

7.4 フック関数置換における問題点

マイクロソフトではプロセスが生成された際にKernel32.dllやUser32.dllのリンクの完了をチェックを行うための関数を用意しておらず, これらのDLLのリンクのタイミングを把握することが困難になっている. そのうえ, LoadLibrary 関数自体をチェックすることではこの問題は解決しない. これは, LoadLibary 関数をフック関数に置き換え自体にAPIフックによる関数置換のズレの問題が発生するためである. そこで, ターゲットプロセスの仮想アドレス空間においてKernel32.dllやUser32.dllのリンクが完了したことを保証する新たな機構の構築が必要となる.

7.4.1 DLL のリンク保証機構

DLL のリンクが完了したことを保証するための方法としては, 以下のものが考えられる.

- ステップ実行を利用したリンクの監視
  APIHook.dllをアタッチした際, DLL_PROCESS_ATTACH内においてKernel32.dllもしくはUser32.dllの中で最も実行速度が速いと思われる関数を呼び出すことでリンクの完了を監視する. Kernel32.dllやUser32.dllのリンクが完了していない場合, これらのDLLからインポートされるWin32APIは実行されないため, Win32APIの実行直後にフック処理を行うことで, Kernel32.dllやUser32.dllのリンク完了時にフック処理を行うことが可能になるものと考えられる.

APIHook.dll 内の処理手順

1. エクスポート関数の呼び出し (Kernel32.dll, または User32.dll)
1. 呼び出し成功の確認
2. フック処理

この処理手順でオブジェクトコードが実行された場合は, フック処理が行われるときに必ず Kernel32.dll 及び User32.dll のリンクが保証されることになるものと思われる. ただし, コンパイル後のオブジェクトコードでは必ずしもソースコード上での処理手順の並びとならないこともあり, この手法が必ずしも有効であるとは言い難い.

- プロセスの仮想アドレス空間の探索による監視

  APIHook.dll がターゲットプロセスにアタッチされた直後から, ターゲットプロセスの仮想アドレス空間内を探索することで DLL のリンクを監視する. この処理は, DLL の完了が確認されるまでの間継続される. この手法はターゲットプロセスの仮想アドレス空間を探索しているため, DLL のリンクの有無に関しては正確な結果が得られると考えられるが, 探索を行うための処理速度を考慮すると, ターゲットプロセスの起動に負荷を与える恐れがあるため, ターゲットプロセスに何らかのオーバーヘッドや不具合が生じる可能性がある.

7.4.2 プロセスのサスペンドモードによる起動

今後, システムにはターゲットプロセスがアプリケーション初期化エラーにより起動不可となる状態を避けるための機能を実装する必要がある. この機能は, ターゲットプロセスが起動時にメモリへロードされる段階で, システムがターゲットプロセスをサスペンド状態にするものである. NativeAPI フックにおいて CreateProcess を利用し、

```
BOOL WINAPI CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

の dwCreationFlags を CREATE_SUSPENDED にセットすることで, 本機能は実装できるものと思われる.

7.5 全てのアカウント権限への対応

今回, ログオンしているユーザーアカウント権限で動作するターゲットプロセスに対しては正確に API フックを行えたが, SERVICE や SYSTEM のアカウント権限で動作するプロセスに対しては行えなかった. これらシステムのアカウント権限で動作するプロセスが悪意のある挙動をとることは考えにくいが, あらゆる権限で動作するターゲットプロセスへの対応は重要である. WindwosNT 系 OS では SERVICE においてはアクセス許可をコントロールすることが可能であり, SERVICE に対して対話の許可を出すことで SERVICE 権限のプロセスの Win32API に対してフック処理を行うことが可能になるものと思われる.

7.6 まとめ

本章では本研究に残された課題に対する考察を行った. フォルダ保護システムは現在, WindowsXP のみで動作確認を行っているが, 今後は Windows7 以降のオペレーションシステムにも対応する必要がある. また, システムは現在, 部分的に非同期の構造をとっているため, 早急に改善を行い完全な同期機構として実装しなければならない.

さらに, フォールスポジティブの確率を下げることにつとめ, そのためにウイルスによるファイルアクセスのみを自動で判断し制御するための仕組みを導入していく必要がある.

ターゲットプロセスに対する処理において初期化エラーが生じることがあり, これらのエラーに関しては原因の究明が急がれる. システムは現在, ファイルアクセスのコントロールを実現しているが, 今後は, ターゲットプロセスに適したフック及びフィルタ処理をシ

ステムが自動選別できる仕組みを実装し, ユーザーに対して負担を下げる工夫を行っていきたい.

第8章　結論

本章では本研究で解決した課題を纏め, さらに今後の展望を述べる.

8.1　まとめ

本研究で構築したファイル検知・制御機構の仕組みを用いたフォルダ保護システムにより, ユーザーは意図しないプロセスに対して, 保護対象とする一定範囲のファイルへのアクセスを監視・制御することが可能となった.

これにより, 従来のパターンマッチング型のウイルス対策ソフトでは困難であった未知のプロセスに対するファイルアクセスに対応し, 保護フォルダからのファイル漏えいを防ぐことが可能となった.

8.2　今後の展望

現在, マイクロソフトは次期 NT 系 OS である Windows8 を公表している. Windows の NT シリーズは今後もドライバモデルなどにおいて大きな変革を遂げていくと思われ, 本システムの将来的な移植のために, 各コンポーネントでの移植の可能性を検討していくことが重要となる. そのためにシステムをできるだけ細分化し, それぞれのコンポーネントの連携に対して汎用 OS に有効なセキュリティ機構を形成していきたい. 特にカーネル空間とのデータ転送を行う処理に関してはオペレーティングシステムの仕様変更に十分耐えうるモデルを構築しなければならない. そこで, SetWindowsHookEx 関数などのマイクロソフトが公式に提供する仕組みをできるだけシステムに取り入れ, それらのデメリットを補う実装にすることで移植性の向上に努めていく.

また, 今後は本システムを高度なウイルス解析の現場に導入するために, フィルタドライバや NativeAPI フックなどにおいても同様にファイルアクセス検知・制御機構を適用し, ターゲットプロセスが動作する空間に左右されることなく検知が行えるフォルダ保護システムの開発を目指す. さらに, リンクに関する初期化エラーや同期問題を改善しフック関数の処理速度を向上させることで, 常駐型として耐えうるシステムの構築を目指していきたい.

近年のウイルス対策における解析手法は目覚しい発展を遂げているが, ウイルスに起因しないセキュリティインシデントの対策は十分に行われていると言い難い. 今後はウイルス単位ではなく, 物理的なユーザー単位でのコンピュータ上でのファイル操作を自動管理し, 第 2.2 節で述べた複雑に絡み合う漏えい原因を自動でカテゴライズする仕組みの構築

を行っていきたい. セキュリティインシデントに対して的確なファイル管理の自動化を行うことで, 今後も増加していくと思われるセキュリティインシデントへの対応の限界という課題を克服しなければならない.

参考文献

[1] セキュリティ被害調査ワーキンググループ. 2002 年度 情報セキュリティインシデントに関する調査報告書. Technical report, NPO 日本ネットワーク協会, 3 2003.

[2] セキュリティ被害調査ワーキンググループ. 2010 年度 情報セキュリティインシデントに関する調査報告書〜個人情報漏えい編〜. Technical Report 第 1.4 版, NPO 日本ネットワーク協会, 8 2011.

[3] 川口 洋. 川口洋のセキュリティ・プライベート・アイズ (5), 7 2008. http://www.atmarkit.co.jp/fsecurity/column/kawaguchi/005.html.

[4] 丹田 賢. 仮想環境に依存せず詳細な解析能力を持つ動的解析システムの設計と実装. Technical report, フォティーンフォティ技術研究所, 7 2011.

[5] Jeffrey Richter. *Advanced Windows*. 日経 BP ソフトプレス, fifth edition, 8 2008.

[6] McAfee Labs. Mcafee 脅威レポート：2010 年第三四半期, 11 2010. http://www.mcafee.com/japan/media/mcafeeb2b/international/japan/pdf/threatreport/threatreport10q3.pdf.

[7] **日経パソコン用語事典** *2011*. 日経 BP 社, 9 2010.

[8] Noah Schiffman. メタモーフィック型マルウェアの脅威, 9 2007. http://techtarget.itmedia.co.jp/tt/news/0709/19/news01.html.

[9] 毛利公一. 仮想計算機モニタによるマルウェアの監視. **情報処理学会シンポジウム論文集**, 2010 巻 9 号:225–230, 10 2012.

[10] 独立行政法人 情報処理推進機構. コンピュータウイルス・不正アクセスの届出状況 [2008 年 11 月分] について. Technical report, 12 2008.

[11] 島田 広道. Windows の自動実行機能 (autorun) を完全に無効化してウイルス感染を防ぐ, 3 2009. http://www.atmarkit.co.jp/fwin2k/win2ktips/1139autrun/autrun.html.

[12] Microsoft Corporation. *MSDN Library*, 5 1997. http://msdn.microsoft.com/en-us/library/ms123401.aspx.

[13] Microsoft Corporation. *Handling IRPs: What Every Driver Writer Needs to Know*, 8 2006. http://msdn.microsoft.com/ja-jp/windows/hardware/gg487398.

[14] Gary Nebbett. *Windows NT/2000 Native API Reference*. Sams, first edition, 2 2000.

[15] Peter Szor. Attacks on win32 part u. Technical report, Symantec, 2000. Virus Bulletin Conference.

[16] 滝口 政光. *WindowXP* **ファイルドライバプログラミング 入門と実践**. 技術評論社, first edition, 2 2003.

謝辞

本論文の作成にあたり，ご指導頂いた慶應義塾大学環境情報学部教授 村井純 博士，同学部教授 徳田英幸 博士，同学部教授 中村修 博士，同学部准教授 楠本博之 博士，同学部准教授 高汐一紀 博士，同学部准教授 三次仁 博士，同学部准教授 植原啓介 博士，同学部専任講師 中澤仁 博士，同学部准教授 Rodney D.Van Meter III 博士，同学部教授 武田圭史 博士, 独立行政法人情報通信研究機構 安藤類央 博士に感謝致します.

特に武田圭史教授は，常に研究についての的確な助言をして下さったほか, 本研究で扱ったファイルアクセス検知・制御機構について多くの技術的なアドバイスをくださいました．また, お忙しい中幾度となく, 学会論文や卒業論文に対するレビューをしてくださいました. 情報セキュリティという分野に興味が持て, また Windows カーネルを扱うまでになれたのも武田圭史教授の指導のおかげでした．本当にありがとうございました.

そして本研究を進めていく上で，様々な激励と助言をいただきました，慶應義塾大学大学院政策メディア研究科 水谷正慶 博士，同研究科 上原雄貴 氏，同研究科 重松邦彦 氏，慶應義塾大学環境情報学部卒業生の 福岡英哲氏, 朝永愛子氏, Doan Viet Tung 氏, 梅田昂翔氏に感謝致します．水谷正慶氏には, 研究指導以外にも三年間様々な面でお世話になり, 大変充実した学生生活を送ることができました. 初めての RG の納会では初対面にも関わらず, 親身にいろいろ相談に乗っていただき, 本当にありがとうございました．上原雄貴氏と重松邦彦氏には，多忙な身にも関わらず，多くの相談に親身に乗っていただき，研究の方向性や研究発表など様々な点で面倒を見ていただきました. 本当に感謝致します.

研究に協力をしていただいた, 碓井利宣氏, 吉原洋樹氏, Nguyen Anh Tien 氏, Pham Van Hung 氏, Vu Xuan Duong 氏, 吉原大道氏, 関根冬輝氏, 中島明日香史，山本知典氏, 藤原龍氏, Nguyen Anh Tien 氏, 大矢崇央氏, 鴻野弘明氏, 三ツ木あかね史, 有馬怜文氏, 由井卓哉氏, 小松真氏と徳田・村井合同研究室の皆様に感謝致します．特に, 研究室で苦楽を共にした碓井利宣氏，吉原洋樹, Pham Van Hung 氏, Vu Xuan Duong 氏に感謝致します. 彼らと一緒に研究をすることで互いに切磋琢磨しあい，研究をより質の高い次元に高めることができました.

また, フォルダ保護システムのデバッグに協力して頂いた ISC の皆さんにも感謝致します. 彼らのおかげで，研究室でのモチベーションも高く維持し続けることができました.

最後に, 大学卒業までの間, あらゆる面で支えていただいた私の家族に心から感謝致します.