Know your enemy
Defense leadership

ランサムウェア「LockBit2.0」の内部構造を紐解く

October, 2021

三井物産セキュアディレクション株式会社

サイバーインテリジェンスグループ
吉川 孝志
菅原 圭

2021 年 10 月 22 日

はじめに

2021 年 10 月 22 日現在、すでに多くのニュースや公開記事で言及されているように、「LockBit2.0」はリークサイトを持つ暴露型ランサムウェア攻撃グループの中で現在(2021 年後半)最も活発である攻撃グループです。LockBit2.0 のリークサイト上では、いきなり窃取データが暴露されるのではなく、「被害組織名」とともに「暴露までの残り時間」をリアルタイムでカウントし被害組織に圧力をかけます。そのため、リークサイトに初めて掲載された時点においては被害組織と攻撃者間で金銭の支払いに関する交渉が行われているかもしくは交渉前の段階にあるケースが多いものと考えられます。

LockBit2.0 の開発者は自身のサイト上で、LockBit2.0 のランサムウェアが世界で最も暗号化速度が速く他のランサムウェアよりも優れていると、攻撃の実働部隊であるアフィリエイトに向け詳細にアピールしており、加えて他のランサムウェアには無い新しい技術も搭載しているという趣旨のコメントを掲載しています。(それを補足する関連情報として、海外で行われた LockBit2.0 の代表者へのインタビューでは「攻撃が速く実行されるほど、攻撃が撃墜されるリスクは少なくなる」と答えています。)

今回我々はこうした点を踏まえ、当該攻撃グループが使用するランサムウェア「LockBit2.0」本体に着目し詳細解析を実施、本記事でその全ての解析結果を共有します。

なお、LockBit2.0 ランサムウェアに関する解析記事については海外ベンダーなどからすでに幾らか公開されていますが、それらの記事において言及されていない事実や、解析が及んでいない挙動などが多くあることを確認しました。

本記事では、そうした他の解析記事では触れられていない詳細情報や、全世界初となる情報なども多数盛り込み、“世界で最も詳細な LockBit2.0 ランサムウェアの解析記事”(※)となるよう意識しました。
（※本記事執筆時点(2021 年 10 月)かつ公開されている範囲において）

また、海外のみでなく日本国内においても多くの被害が出ていると想定される状況下において、LockBit2.0 ランサムウェアに関する日本語オリジナルによる詳細解析記事自体が依然ほとんど存在しない状況であるため、今回公開した本記事が幾らか国内を中心とした皆様の、お役に立てば幸いです。

目次

※以降の文中において「LockBit2.0」と記載した場合、攻撃グループ名ではなくランサムウェアを指します。

■概要

はじめに LockBit2.0 の特徴をまとめると、以下のような項目があげられます。

- I/O 完了ポートの採用による暗号化処理の高速化
- ファイルの一部のみを暗号化することによる暗号化処理の高速化
- AES-NI が利用可能な CPU 認識＆活用による暗号速度の高速化
- 楕円曲線暗号（ECC）方式を採用したことによる暗号速度の高速化
- プリンタを利用した物理的な脅迫文の印刷（プリンター台当たり約 1 万枚）
- デスクトップ壁紙を利用したインサイダー（内部者）募集
- グループポリシーによるドメイン配下の全端末への自動展開
- 全ローカルドライブのネットワーク共有化
- 効果的な解析妨害（難読化、デバッガ検知、スレッドのステルス化など）
- あまり詳細が知られていない UAC Bypass による権限昇格
- 入念なフォレンジック対策（単純な削除ではなく自身をハードディスクから抹消）
- 入念な感染継続性（暗号化中に端末がシャットダウンされることに備えた再開処理）
- 隠しパーティションのスキャンと強制ドライブマッピング

など

LockBit2.0 が自身のサイト上で自負しているランサムウェアの高速性については、詳細解析の結果、実際にファイルの暗号化を高速化させる複数の高度なテクニックが実装されていることを確認しました。

また、様々な手口のアイデアが機能として盛り込まれており、特に、ドメインコントローラ感染時に特別に発生させる「グループポリシーによる自身の拡散」という仕組みは、暴露系ランサムウェアとしては史上初の新機能であり、従来のアフィリエイト（攻撃の実働部隊）が手動で行っていた作業を置き換えることで、攻撃能力に個人差があるアフィリエイトの手動作業をできるだけ自動化し攻撃の質を平準化させる試みが感じられます。

また今回の解析で、LockBit2.0 が世界的に知られていないグラフィカルなインターフェイス（GUI）である「隠しウインドウ」を実装していることや、LockBit2.0 が複数の致命的なバグを抱えており、検体や被害環境によっては多重暗号化が発生することで、たとえ復号ツールが入手できたとしても復号が困難となるシチュエーションが存在することなど、多くの新事実も発見しました。
これらを含む、LockBit2.0 詳細解析の全貌は以下の解説をご覧ください。

■表層解析

LockBit2.0 は Visual Studio 2017 で作成され、構造上は GUI アプリケーションとしてビルドされています。ただし実際には実行されても画面上には何も表示されずサイレントに感染が行われます。
また、単純な表層解析の範囲では一般的な GUI コンポーネントなどのリソースデータは EXE ファイルから確認することはできません。

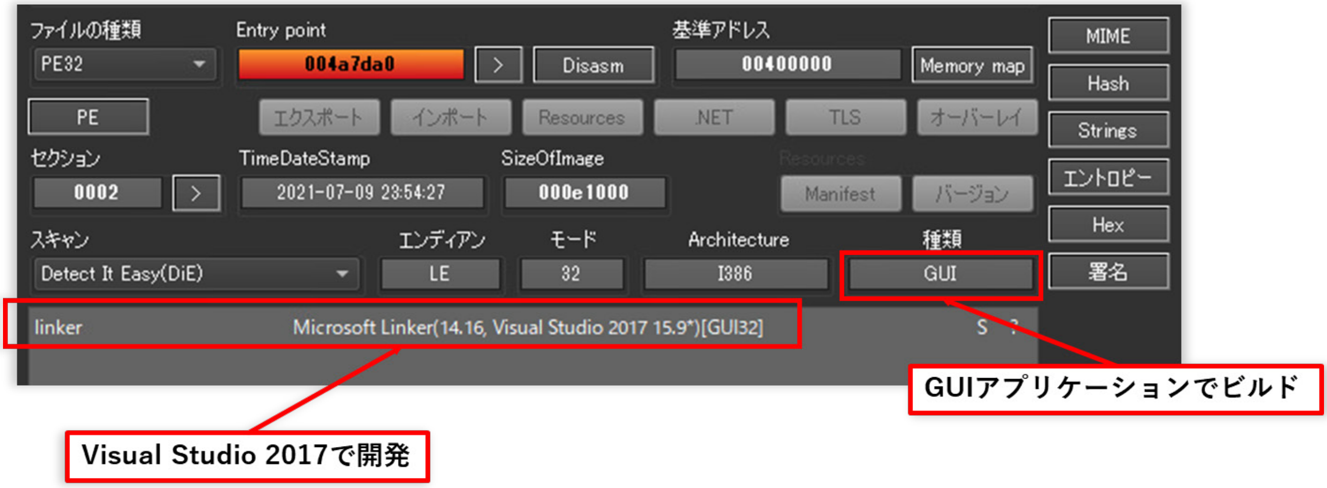


図 1　LockBit2.0 の開発情報

セクションとしては以下の 2 つの構造を持ち、一般的な GUI アプリケーションにみられるリソースセクションなども存在しません。

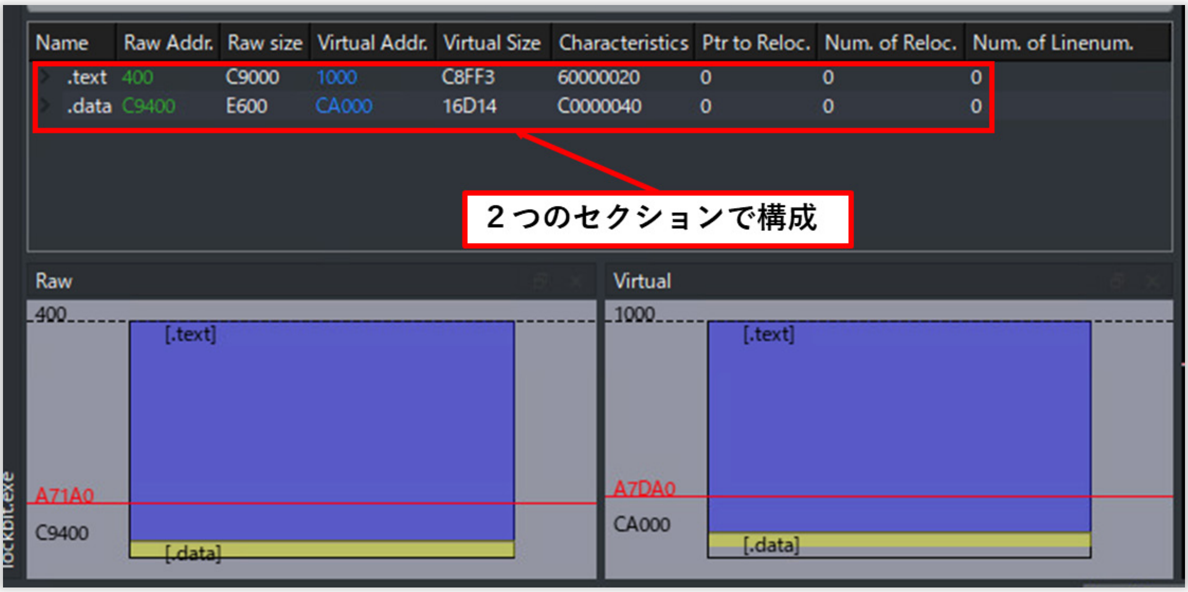


図 2 LockBit2.0 のセクション情報

アドレス空間配置のランダム化である ASLR は無効化された状態でビルドされています。

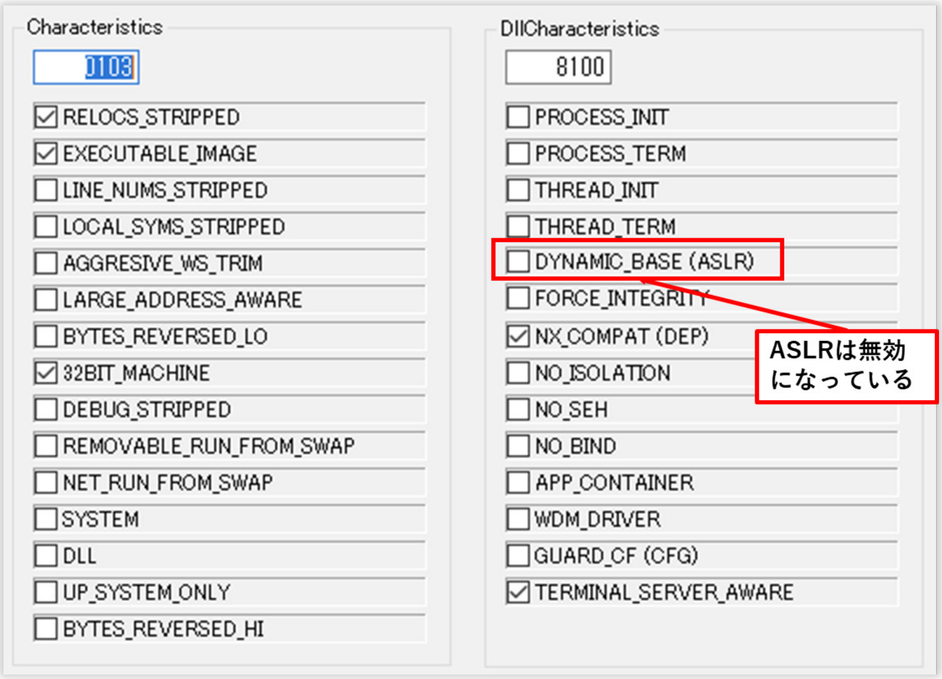


図 3 LockBit2.0 の EXE 設定情報

その他、デジタル署名やプロパティ情報などは基本的に付与されていません。

■耐解析機能（解析を妨害する機能）

LockBit2.0 は解析を困難とするためにいくつかの耐解析機能（解析妨害テクニック）を実装しています。

LockBit2.0 は起動すると、すぐに自身がデバッグされていないかどうかをチェックし、デバッグ中であると判断すると無限ループし動的解析を妨害します。
具体的には、プロセス内に存在する様々な情報が格納された PEB（Process Environment Block）という構造体の中の「NtGlobalFlag」という項目をチェックします。プロセスがデバッグ中の場合はこの NtGlobal Flag が 0x70 という値になるため、LockBit2.0 は NtGlobalFlag の値を取得し、「0x70」と比較することでデバッグ中であるかを判断します（下図参照）。

■ 起動直後に自身がデバッグされていないかどうかをチェック、デバッグ中と判断すると無限ループし動的解析を妨害する。

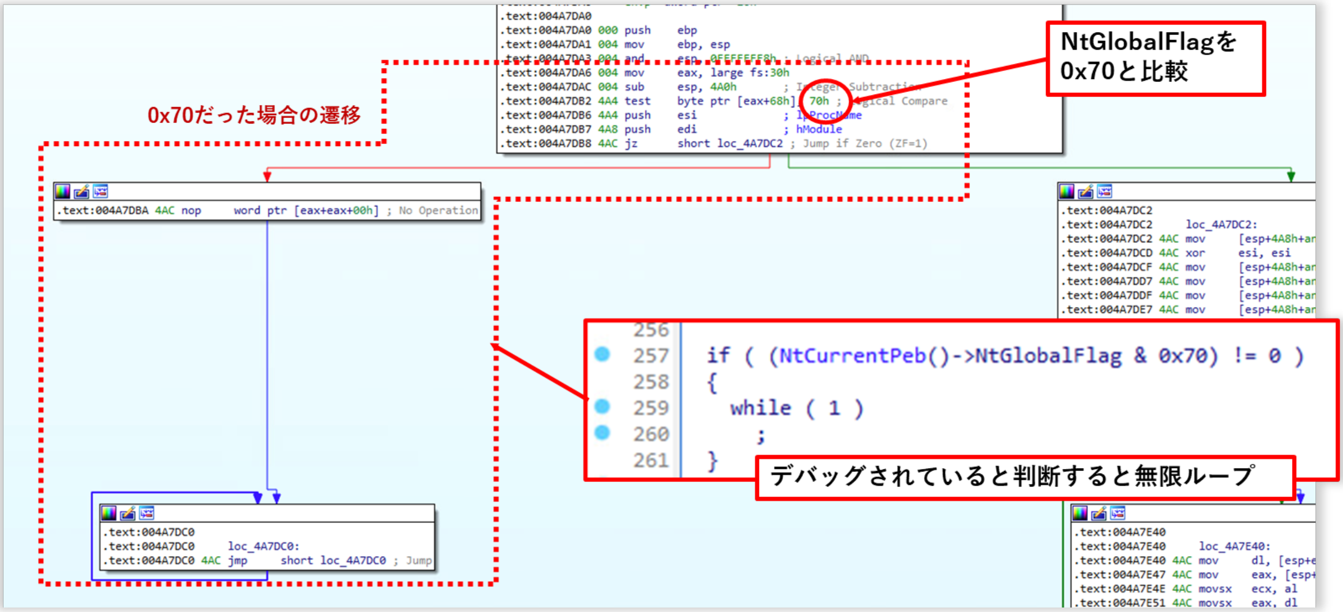


図 4 NtGlobalFlag による解析妨害

また、LockBit2.0 は動作中、複数のスレッド（マルチスレッド）により処理を行っていきますが、その際にデバッガからスレッドを解析できないよう、スレッドをステルス化することで解析を妨害します。

このテクニックは新しいものではありませんが、非常に効果的な解析妨害方法であり、このアンチデバッグテクニックは BlackMatter 等の高度なランサムウェアでも確認されています。

具体的には、NtSetInformationThread 関数を使用し、指定したスレッドに ThreadHideFromDebugger（0x11 という値）という特殊なフラグを設定する文書化されていない呼び出し方法により実現します。この値が設定されると、デバッガは該当フラグが設定されたスレッドからの情報を取得できなくなるため該当スレッド上のコードの解析ができなくなり、実質的にデバッガからスレッドを隠すのと同じ効果を得られます（該当スレッドを解析しようとしてもデバッガが素通りしてしまう上、無理やり解析を進めようとするとクラッシュします）（下図参照）。

■スレッドをデバッガから解析できないように、スレッド自体をステルス化（隠蔽）する。

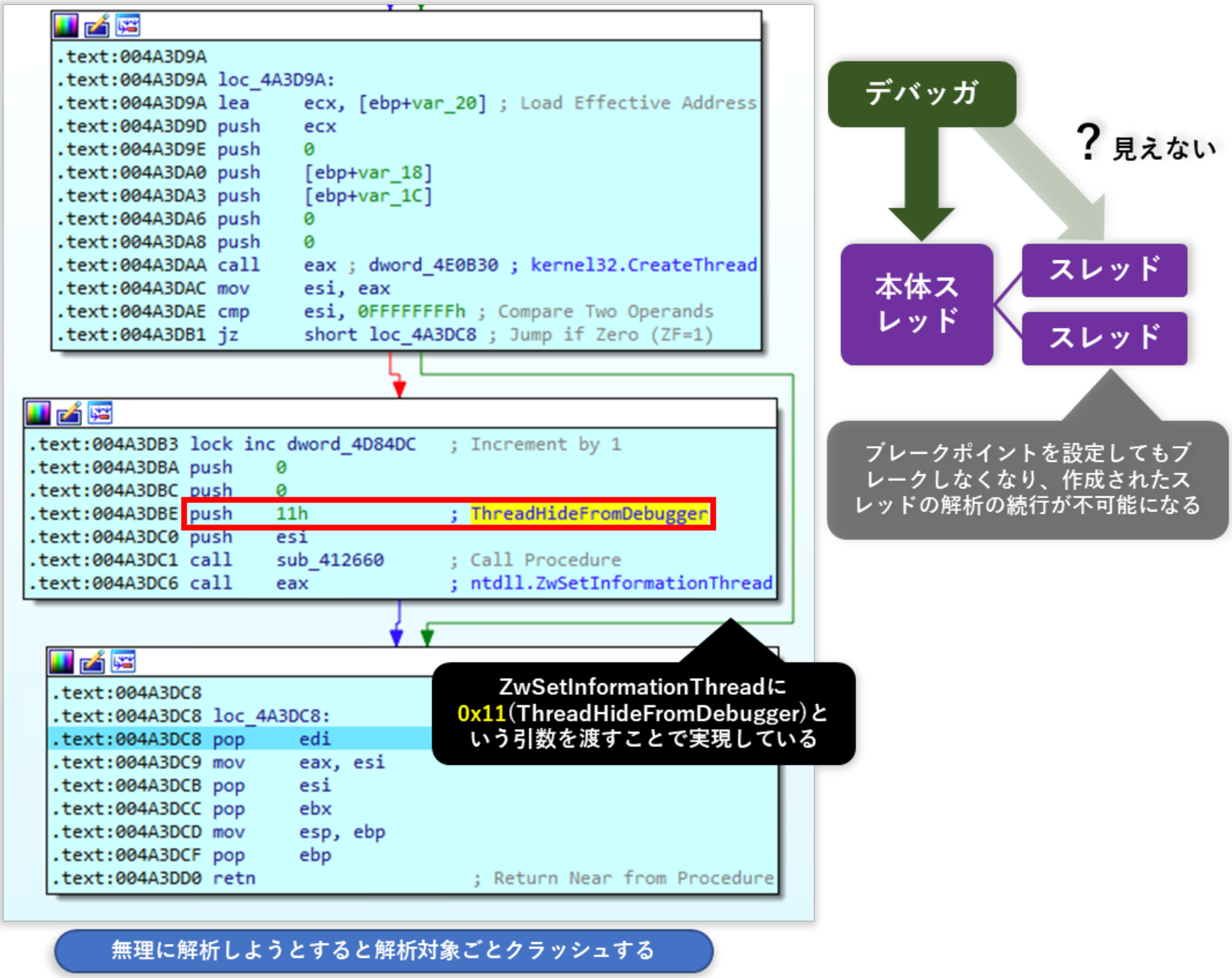


図 5　ThreadHideFromDebugger による解析妨害

また LockBit2.0 は、Windows API を名前ではなくハッシュ値で求める手法を採用し、Windows API を隠蔽することで静的解析を妨害するテクニックも実装しています（下図参照）。

■Windows APIをハッシュで求める手法を採用し、Windows APIを隠蔽することで静的解析を妨害する。

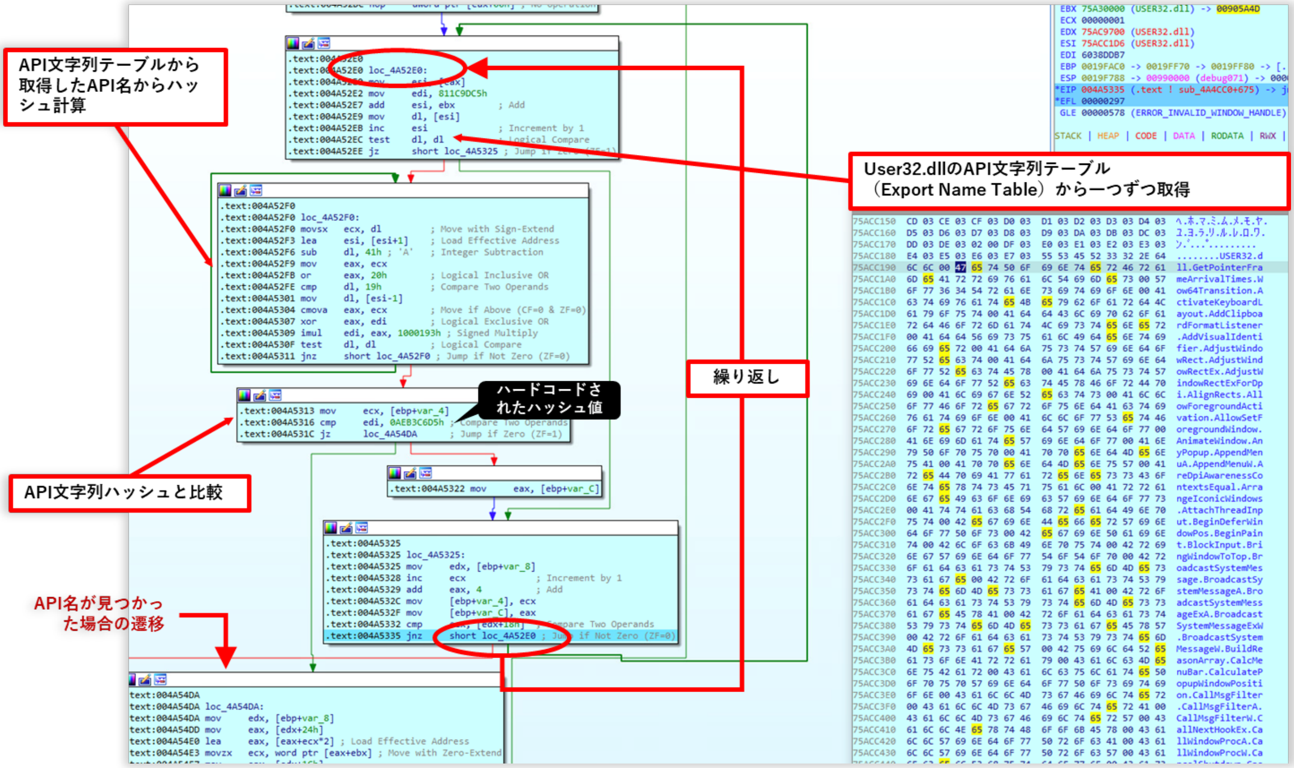


図 6　WindowsAPI のハッシュ比較による解析妨害

例えば、「Kernel32.dll」のアドレスは、以下のように PEB の InLoadOrderModuleList から辿りハッシュ値をチェックすることにより導き出します（下図参照）。

PEBのInLoadOrderModuleListから辿る「Kernel32.dll」のアドレス

```
FARPROC __stdcall Get_Addr_Kernel32DLL()
{
  _LIST_ENTRY *Flink; // edi
  _LIST_ENTRY *var_str_dllname; // eax
  int var_count; // ebx
  int var_Blink; // edx
  int var_Blink_1; // edi
  char str_one; // dl
  int var_xordata; // ecx
  int var_calc; // ecx
  _LIST_ENTRY *Flink_2; // eax
  _LIST_ENTRY *v10; // [esp+Ch] [ebp-10h]
  _LIST_ENTRY *Flink_1; // [esp+14h] [ebp-8h]
  unsigned int var_tmp; // [esp+18h] [ebp-4h]

  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v10 = Flink;
  Flink_1 = Flink;
  while ( 1 )
  {
    var_str_dllname = Flink[6].Flink;
    var_count = 0;
    var_Blink = LOWORD(Flink[5].Blink) >> 1;
    var_tmp = 0x811C9DC5;
    if ( var_str_dllname > (_LIST_ENTRY *)((char *)var_str_dllname + 2 * var_Blink) )
      var_Blink = 0;
    if ( var_Blink )
    {
      var_Blink_1 = var_Blink;
      do
      {
        str_one = (char)var_str_dllname->Flink;
        var_str_dllname = (_LIST_ENTRY *)((char *)var_str_dllname + 2);
        var_xordata = str_one | 0x20;
        if ( (unsigned __int8)(str_one - 65) > 0x19u )
          var_xordata = str_one;
        ++var_count;
        var_calc = 0x1000193 * (var_tmp ^ var_xordata);
        var_tmp = var_calc;
      }
      while ( var_count != var_Blink_1 );
      Flink = Flink_1;
      if ( var_calc == 0xA3E6F6C3 )
        break;
    }
    Flink_2 = Flink->Flink;
    Flink = Flink_2;
    Flink_1 = Flink_2;
    if ( Flink_2 == v10 || !Flink_2[3].Flink )
      return 0;
  }
  return (FARPROC)Flink_1[3].Flink;
}
```

InLoadOrderModule Listから辿る

ハードコードされたハッシュ値と比較

図 7　Kernel32.dll のアドレスの割り出し処理

例えば、「gdiplus.dll」という DLL を「LoadLibraryA」でロードする際の一連の処理は、以下のような流れになります（下図参照）。LoadLibraryA のアドレスがハッシュ値で導き出された後、難読化されていた「gdiplus.dll」という文字列が復号され、引数に渡されます。

つまり、API はアドレスをハッシュ値で参照し、API で使用する文字列などは毎回復号して使用します。

その他、動作中に使用する全ての文字列についても同様に難読化されており、都度復号して使用します。

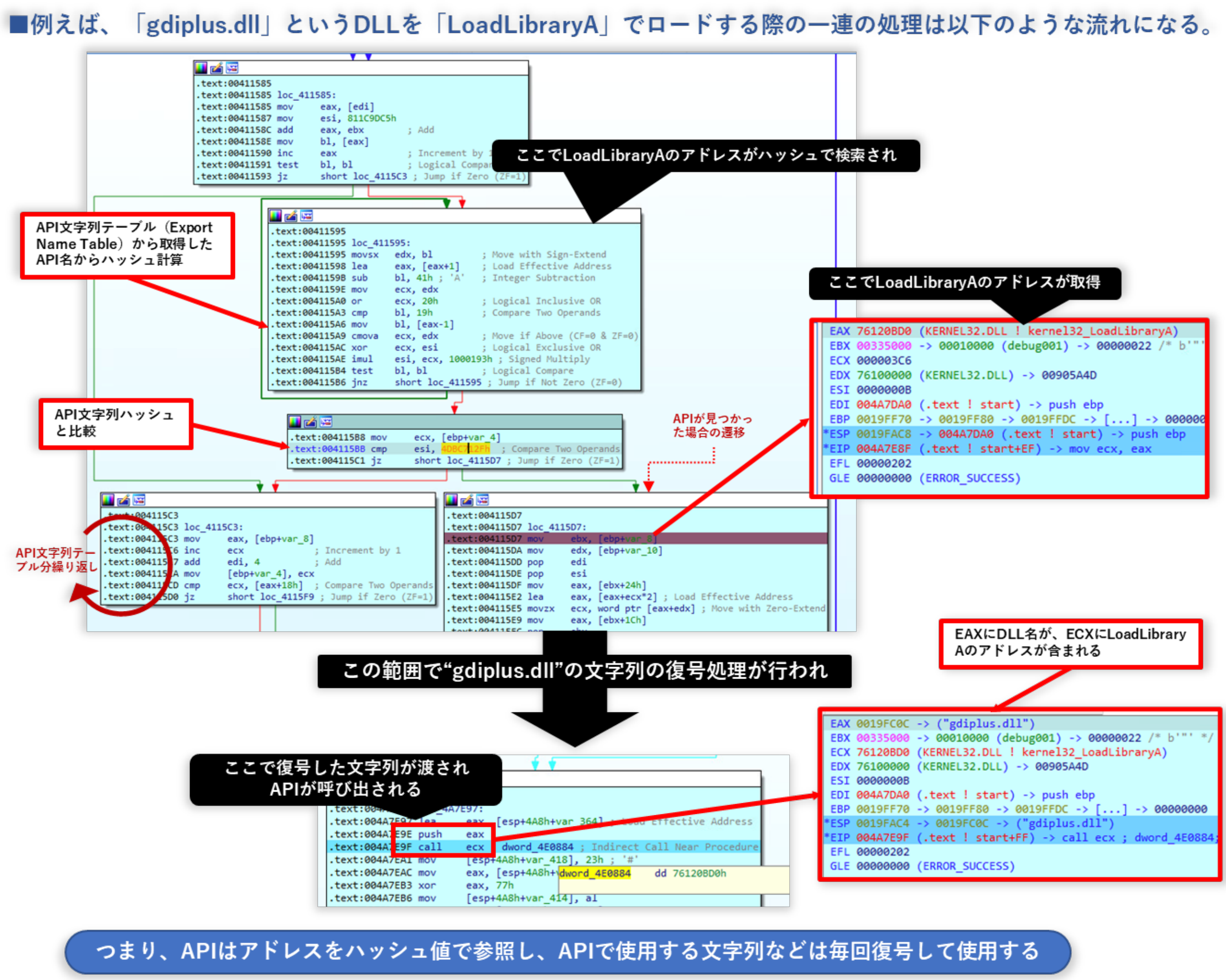


図 8 gdiplus.dll を LoadLibraryA でロードする際の処理例

文字列の復号は一つの関数で行われるわけではなく、文字列ごとにそれぞれ異なる復号方法を使用しています。以下の図は例として「Advapi32.dll」という DLL 名の復号と、「user32.dll」という DLL 名の復号処理の一部を抜粋したものです（下図参照）。

■文字列の復号は一つの関数ではなく、文字列ごとにそれぞれ異なる復号方法を使用している。以下は２つピックアップした例。

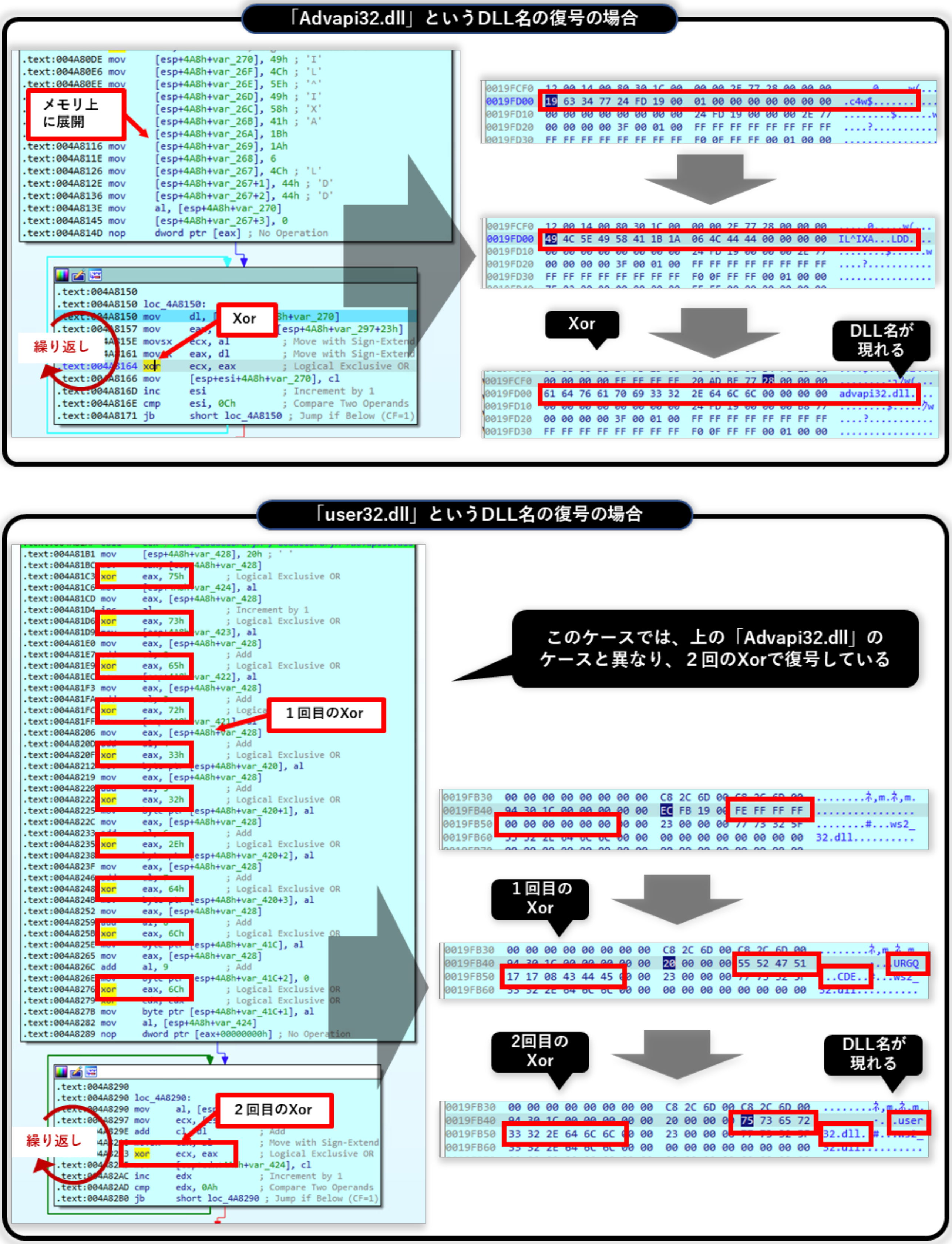


図 9　それぞれの文字列の復号は複数の異なる復号方法を使用する

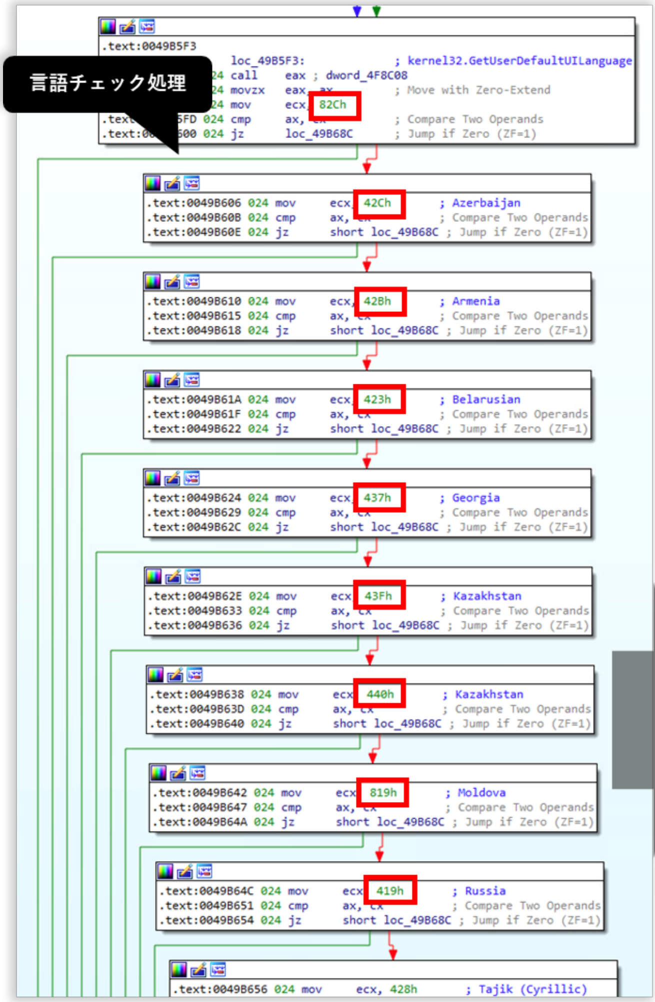
以上のような複数の対策により、動的解析・静的解析の両面の観点からの解析を妨害します。決して数多くの耐解析機能が実装されているわけではありませんが、少ない中でより効果的な耐解析テクニックを組み合わせて利用しているといえます。

■感染環境の言語チェック

LockBit2.0 は感染端末の言語設定を取得し、CIS（独立国家共同体）を含むロシアを中心とした旧ソ連諸国の言語が設定されていないかどうかをチェックし、設定されている場合は感染せずに終了します（下図参照）。この点については、開発者も自身のサイト上で”LockBit2.0 はポスト・ソビエト国家には感染しない”と明言しています。

言語チェックは、GetSystemDefaultUILanguage 関数と GetUserDefaultLangID 関数が用いられます。

■CIS（独立国家共同体）を含むロシアを中心とした旧ソ連諸国の言語が設定されていないかチェックし、設定されている場合は感染せずに終了する。



言語識別子	言語識別子に対応する言語
2092(0x82C)	アゼルバイジャン語（キリル文字）
1068(0x42C)	アゼルバイジャン語（ラテン文字）
1067(0x42B)	アルメニア語 - アルメニア
1059(0x423)	ベラルーシ語 - ベラルーシ
1079(0x437)	グルジア語 - グルジア
1087(0x43F)	カザフ語 - カザフスタン
1088(0x440)	キルギス語 - カザフスタン
2073(0x819)	ロシア語 - モルドバ
1049(0x419)	ロシア語 - ロシア
1064(0x428)	タジク語（タジキスタンのキリル文字）
1090(0x442)	トルクメニスタン
2115(0x843)	ウズベク語（キリル文字）- ウズベキスタン
1091(0x443)	ウズベク語（ラテン文字）- ウズベキスタン
1058(0x422)	ウクライナ語 - ウクライナ

図 10　特定の言語環境かどうかをチェックする処理とチェックする言語の一覧

なお一般的にあまり知られていませんが、こうした言語チェックを行わない LockBit2.0 の個体も一部存在することを今回の調査の中で確認しています。つまり、検体によっては言語チェックの機能自体が実装されていないケースがあり、ロシア語などの言語が設定された端末であっても以下の図のように感染してしまいます（下図参照）。

これは攻撃の実働部隊であるアフィリエイトがランサムウェアを攻撃対象組織に合わせチューニングしたうえで簡単に EXE を生成できるような仕組みを提供する RaaS(Ransomware as a Service)において、ランサムウェアに持たせる各機能の有効化/無効化をビルド時の設定で選択できる仕組みが背景にあるものと考えられ、そうした中で言語チェックを有効にしない状態でビルドされた検体が一部存在するものと推測しています。

■ただし、それらの言語チェックが行われないLockBit2.0の検体（異なる検体）も存在することが確認された

図 11　ロシア語端末で LockBit2.0 が感染した様子

■UAC Bypass による権限昇格

LockBit2.0 は、自身が管理者権限を持っていない状態で起動した場合、UAC(ユーザーアカウント制御)の昇格プロンプトを回避し管理者権限に昇格するテクニックである「UAC Bypass」を行います。

コード上では、以下のように管理者権限で実行された場合と、ユーザ権限で実行された場合で異なる遷移を取る分岐点があります（下図参照）。

■UAC Bypassの処理と終了処理の構造

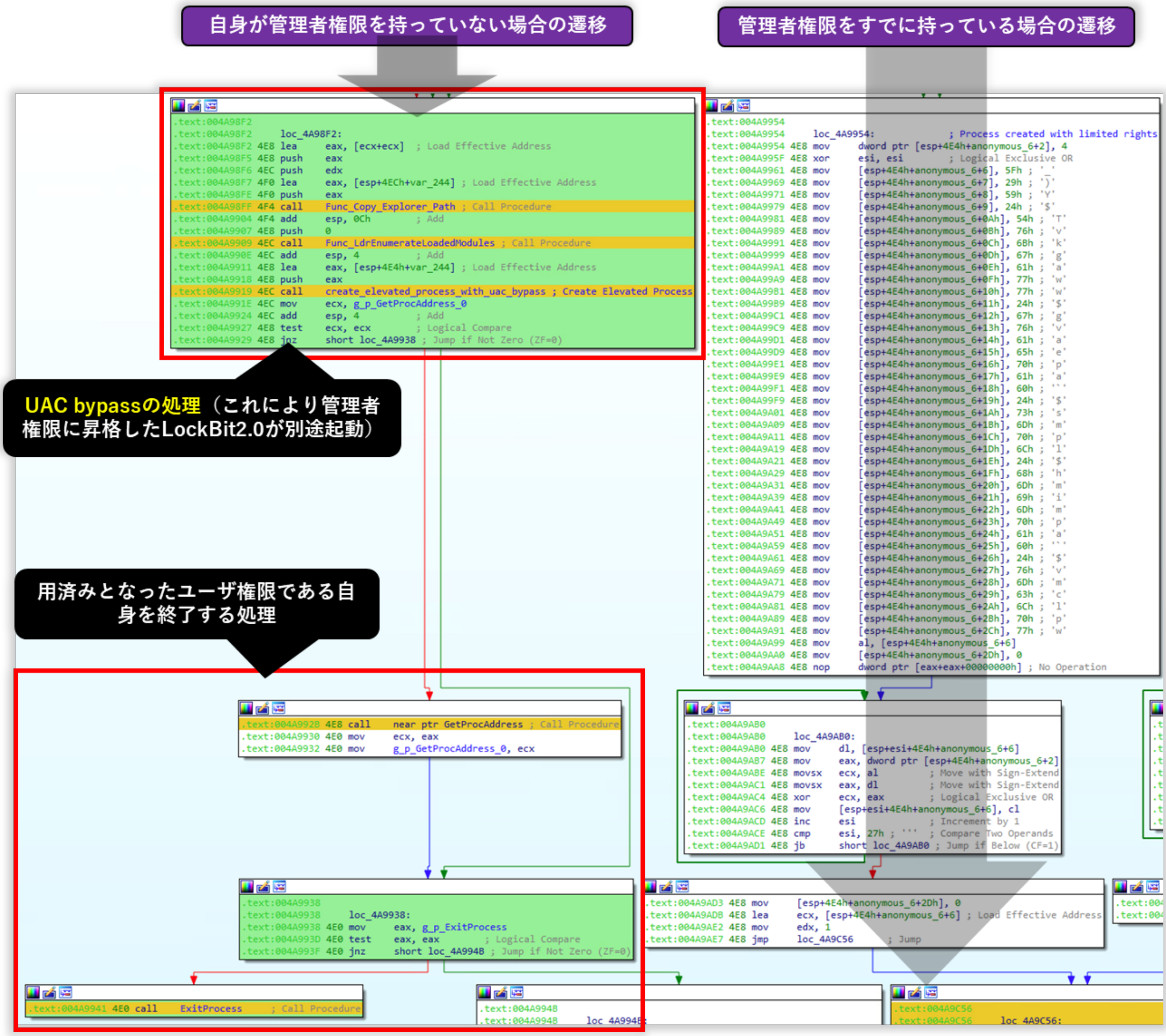


図 12　UAC Bypass に関わる処理の分岐箇所と終了処理の構造

なお、この分岐点を解析作業時に容易に見つけ出すためには、実行時に通過したコード領域の差分分析を行うカバレッジログ分析が有効です。

以下のように「管理者権限で動作したコード領域」と「ユーザ権限で動作したコード領域」の差分を取り、「ユーザ権限でのみ動作したコード領域」を抽出させることで、UAC Bypass のコード箇所が抽出され特定できます（下図参照）。

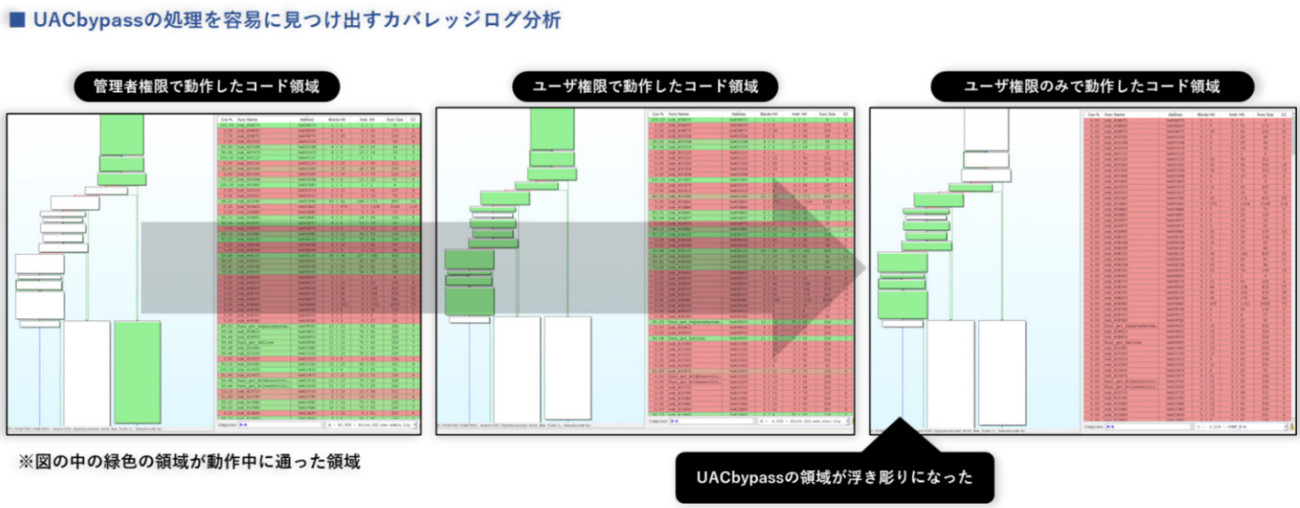


図 13　カバレッジログを用いた UAC Bypass 処理箇所の抽出と特定

LockBit2.0 は、上記のカバレッジログ分析において明らかとなった「ユーザ権限でのみ動作するコード領域」において、あまり広く詳細が知られておらず文書化されていない特定の COM(Component Object Model) インターフェイスを悪用する UAC Bypass テクニックを用います。

以降では、その UAC Bypass の流れを詳細解説します。

まず LockBit2.0 は自身が管理者権限で実行されているかどうかを確認します。
具体的には以下の方法で行います。
ZwOpenProcessToken 関数に TOKEN__QUERY(0x8)アクセス権を指定した上で NtQueryInformation Token 関数を呼び出しプロセストークンの構造体(TokenInformation)を取得、その際 TokenInformationCl ass の「TokenElevation(0x14=20)」を引数に指定することで TOKEN_ELEVATION 構造体を取得します。
その後、取得した TOKEN_ELEVATION 構造体のメンバ変数である「TokenIsElevated」の値が 0 かどうかをチェックすることで管理者権限かどうかを判断します（下図参照）。
（自身が管理者権限で実行されていると判断した場合、[Process created with admin rights]という難読化された文字列をメモリ上に復号したのち、メインの処理を継続します）

■管理者権限かどうかを確認するためにプロセストークンを取得

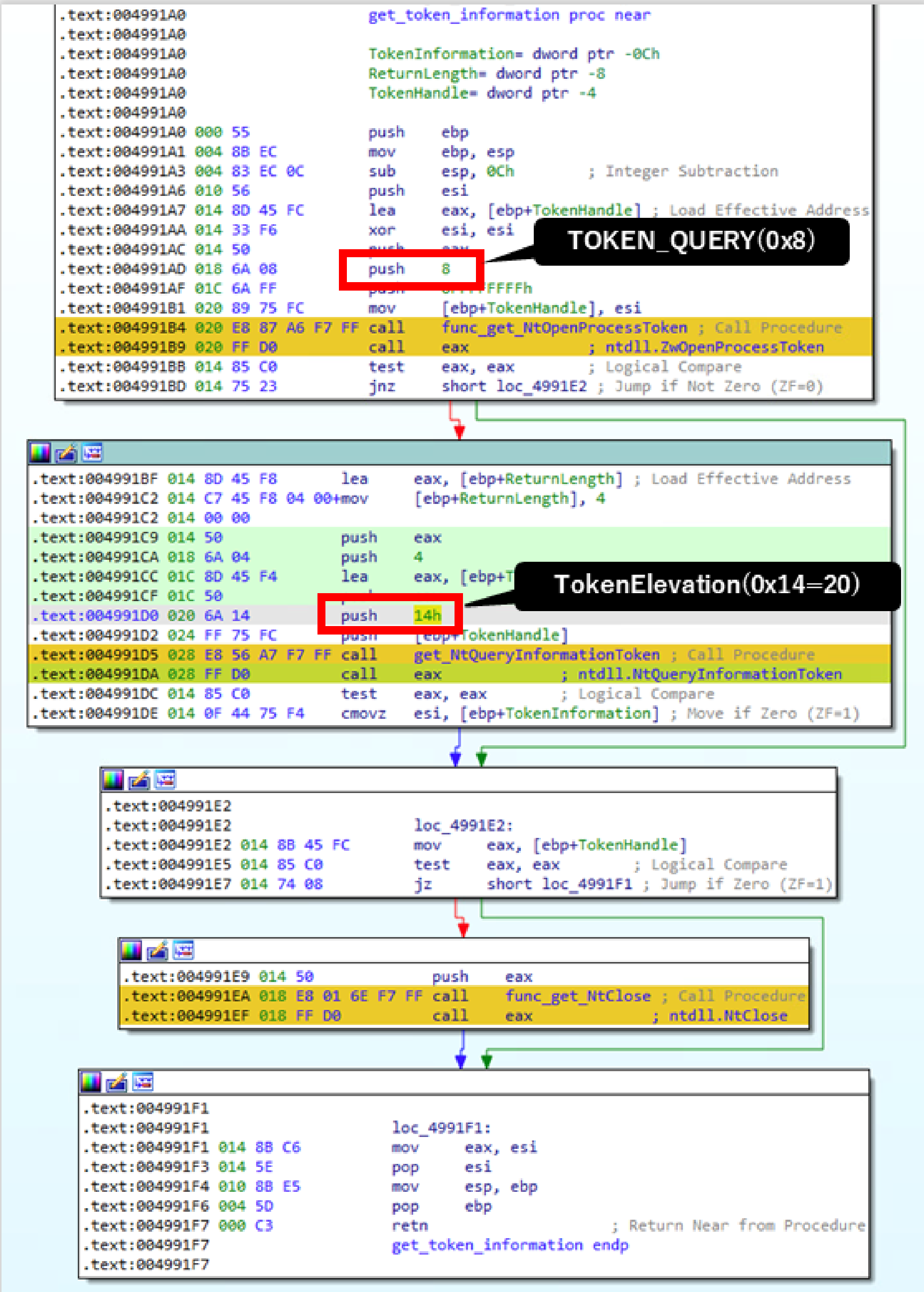


図 14 管理者権限かどうかを確認するためにプロセストークを取得

自身が管理者権限で起動していないと判断した場合、自身が UAC Bypass を行うことができる環境にあるかを確認する作業を行います。

まず先ほどと同様、ZwOpenProcessToken 関数に TOKEN_QUERY(0x8)アクセス権を指定した上で NtQueryInformationToken 関数を呼び出し、プロセストークンの構造体(TokenInformation)を取得します。

続いて CreateWellKnownSid 関数に「WinBuiltinAdministratorsSid」(0x1A=26)という引数を指定することで「BUILTIN\Administrators」グループに対応する SID(セキュリティ識別子)を取得(作成)したのち、C

heckTokenMembership 関数を利用して、呼び出し元のプロセストークン内で「BUILTIN\Administrators」グループの SID が有効になっているかどうかを確認することで、Administrators グループに属しているかどうかをチェックします（属していない場合は本処理を終了します）。
つまりこの時点で、現在のユーザが”管理者権限を持たず Administrators グループに属しているユーザである”と判断できることになります。
なお、Windows では Administarators グループに属しているユーザがログオンする際、対となる 2 つのトークン（管理者としてのフルアクセストークンと制限されたユーザのトークン）が発行されます。
この 2 つのトークンのうち、現在の対となるトークンを TokenLinkedToken 関数で取得することができます。
LockBit2.0 は自身のプロセスが管理者権限を持たず、かつ実行したユーザが Administrators グループに属していると判断すると、NtQueryInformationToken 関数の引数に「TokenLinkedToken」(0x13=19)を指定することで、対となるトークン、つまり、フルアクセストークンを取得できるか確認します。
(この処理が失敗した場合、つまり対となるフルアクセストークンが存在しない場合は「Process created with limited rights」という文字列を復号しメモリ上に展開し、UAC Bypass の処理を終了します)

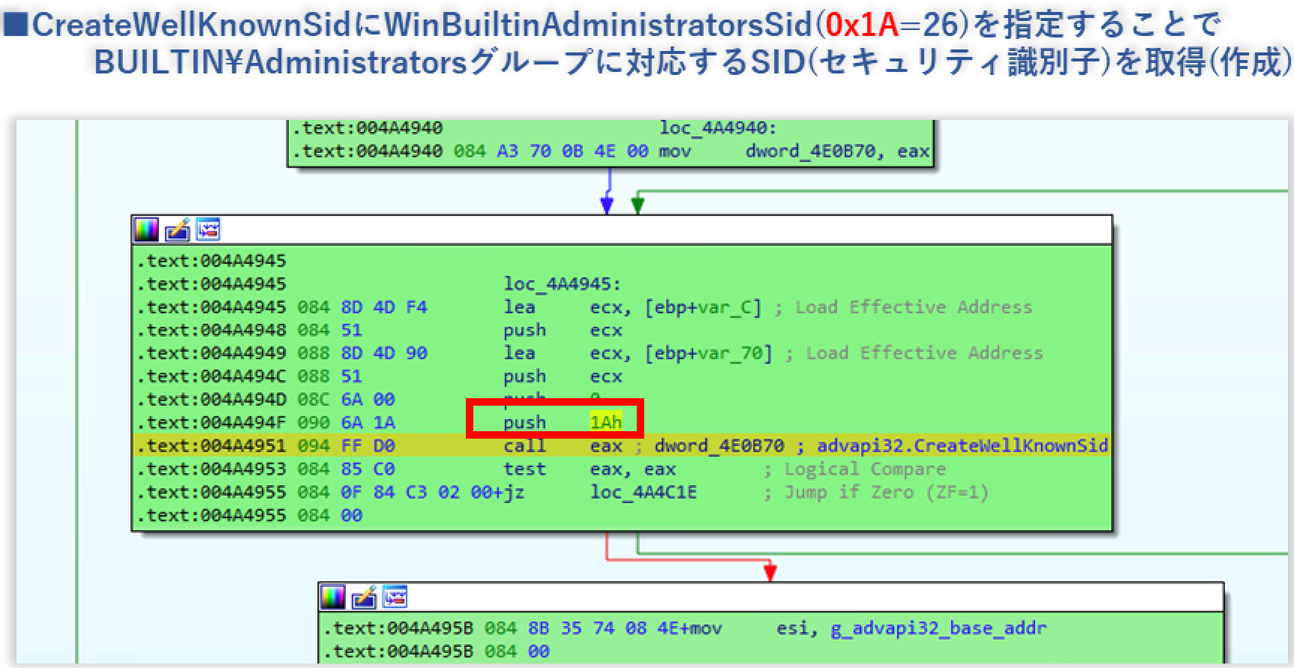


図 15　Administrators グループに属しているかどうかをチェック

次に、LockBit2.0 はこれから行う UAC　Bypass が利用できる OS バージョンであるかを確かめるため、GetVersion 関数で Windows のメジャーバージョンを取得し、「5」(Windows XP/2000/Server2003/Server2000)という値と比較、Windows のメジャーバージョンが「5」ではない場合のみ処理を継続します（下図参照）。

（もし「5」に該当した場合は「Process created with limited rights」という文字列を復号しメモリ上に展開し、UAC Bypass の処理を終了します）

■Windows OS のバージョンを比較する処理

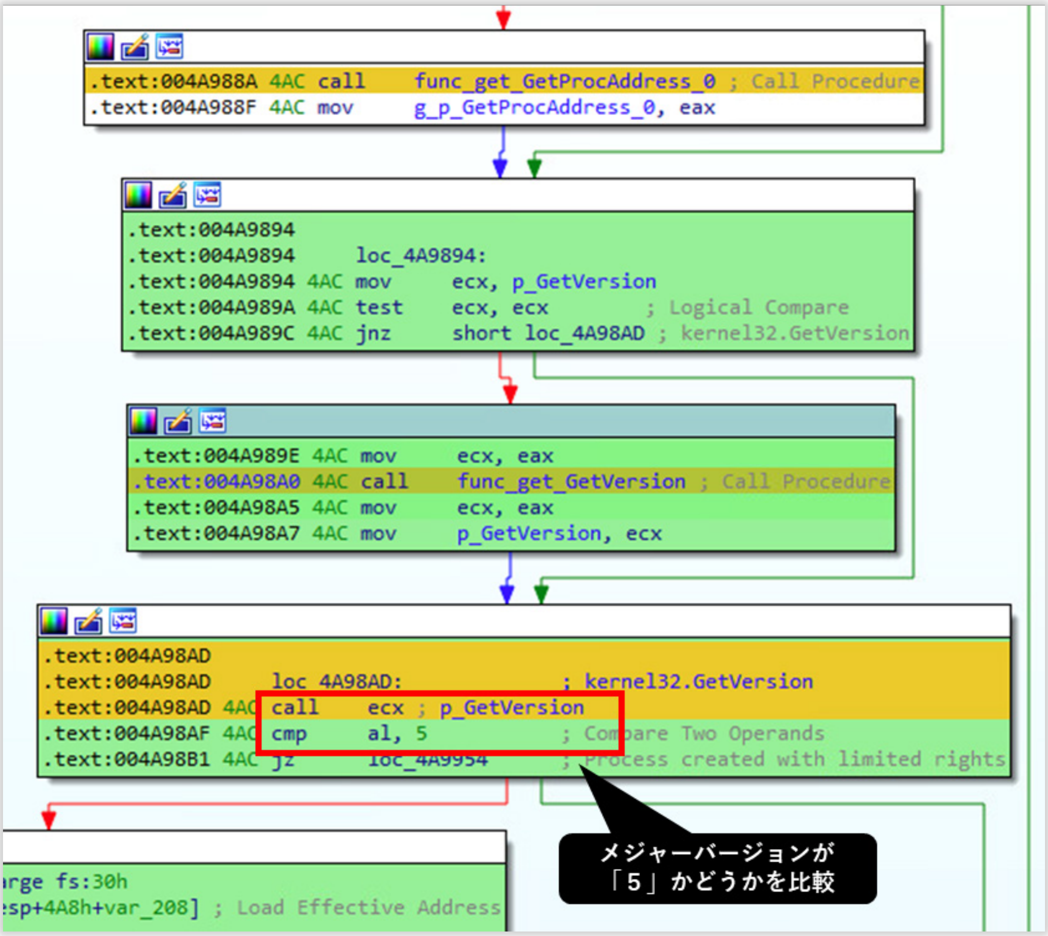


図 16　Windows のメジャーバージョンを「5」と比較

以上の処理により、LockBit2.0 は自身が UAC Bypass を行うことができる環境にあると判断し、UAC Bypass の主たる処理に遷移します。

今回 UAC Bypass にも利用される「COM」(Component Object Model)は、Process Status API（PSAPI）を使用して、プロセスの PEB という構造体の processParameters に含まれる実行パスに関する情報を読み取ることで COM の呼び出し元プロセスを識別します。
そのため前準備として、LockBit2.0 は自身のプロセスが持っている実行パス情報を、以降で解説する方法により信頼できるプロセスの一つである「Explorer」の実行パスに強制的に書き換えることで偽装します。
（こうしたプロセスの偽装処理を「マスカレード」と呼びます）

そのため、まず以下のように Explorer.exe のフルパス文字列を生成します（下図参照）。

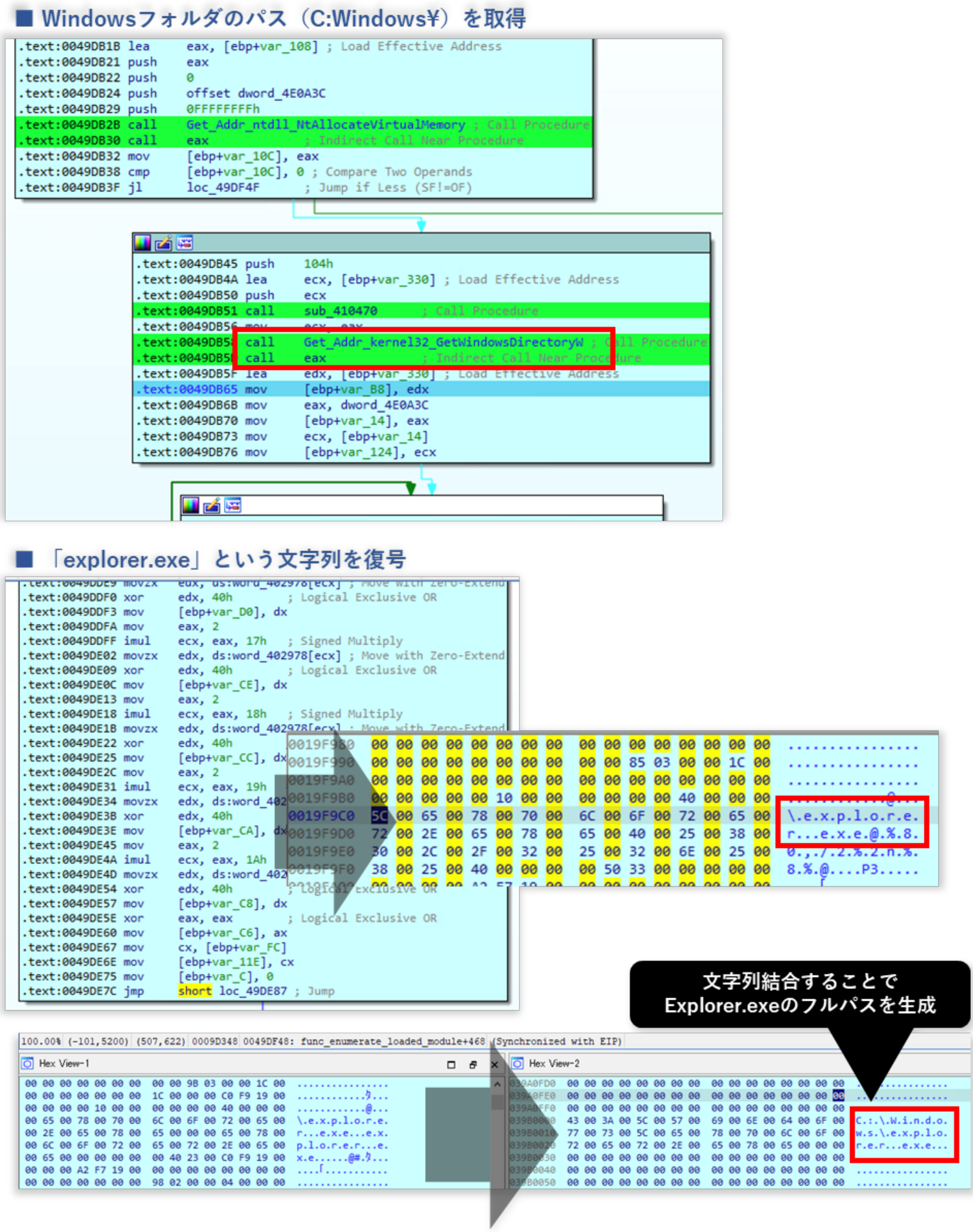


図 17　マスカレード処理で使用する Explorer のパスを生成

続いて、LockBit2.0 は RtlInitUnicodeString 関数を使用し、” C:¥Windows¥Explorer.exe” という文字列を自身の PEB の processParameters 項目にある ImagePathName に、また、 “Explorer.exe” という

文字列を自身の PEB の processParameters 項目にある CommandLine の項目に上書きします（下図参照）。

PEB の processParameters の各項目は UNICODE_STRING という構造体の形式を取りますが、この際、LockBit2.0 は PEB の書き換えに RtlInitUnicodeString 関数をそのまま使用することで UNICODE_STRING に変換しつつ直接書き換えるという二つの作業を同時に行います。

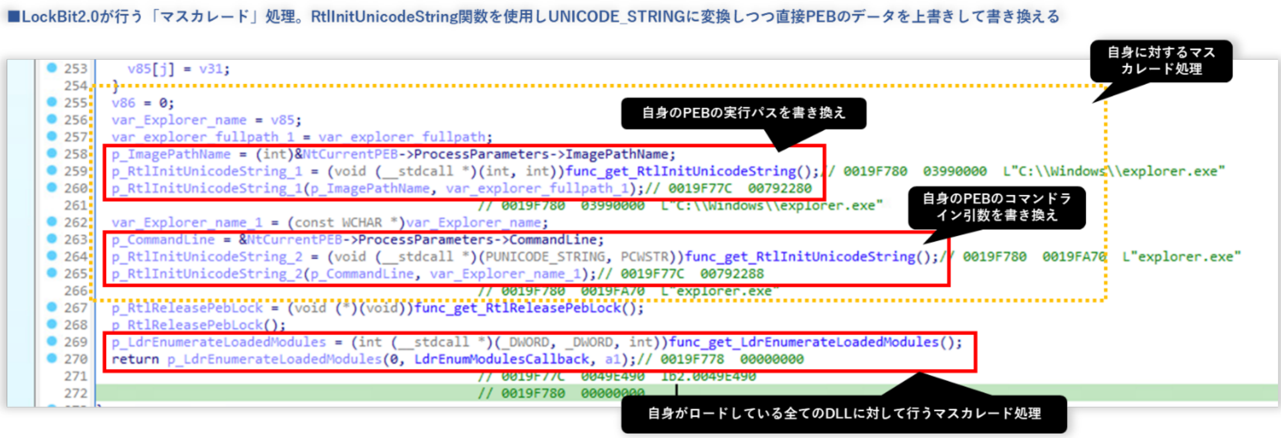


図 18　LockBit2.0 が行う「マスカレード」（プロセス情報偽装）処理

つまり、これにより以下のように LockBit2.0 の PEB にあるプロセスパラメータのパス情報が Explorer のパスに偽装（マスカレード）されます（下図参照）。
またこのマスカレード処理を、自身がロードしている全てのモジュール(DLL)に対しても同様に行います。

■起動中のLockbit2.0のプロセス情報から得られるパス情報が改ざんされる様子

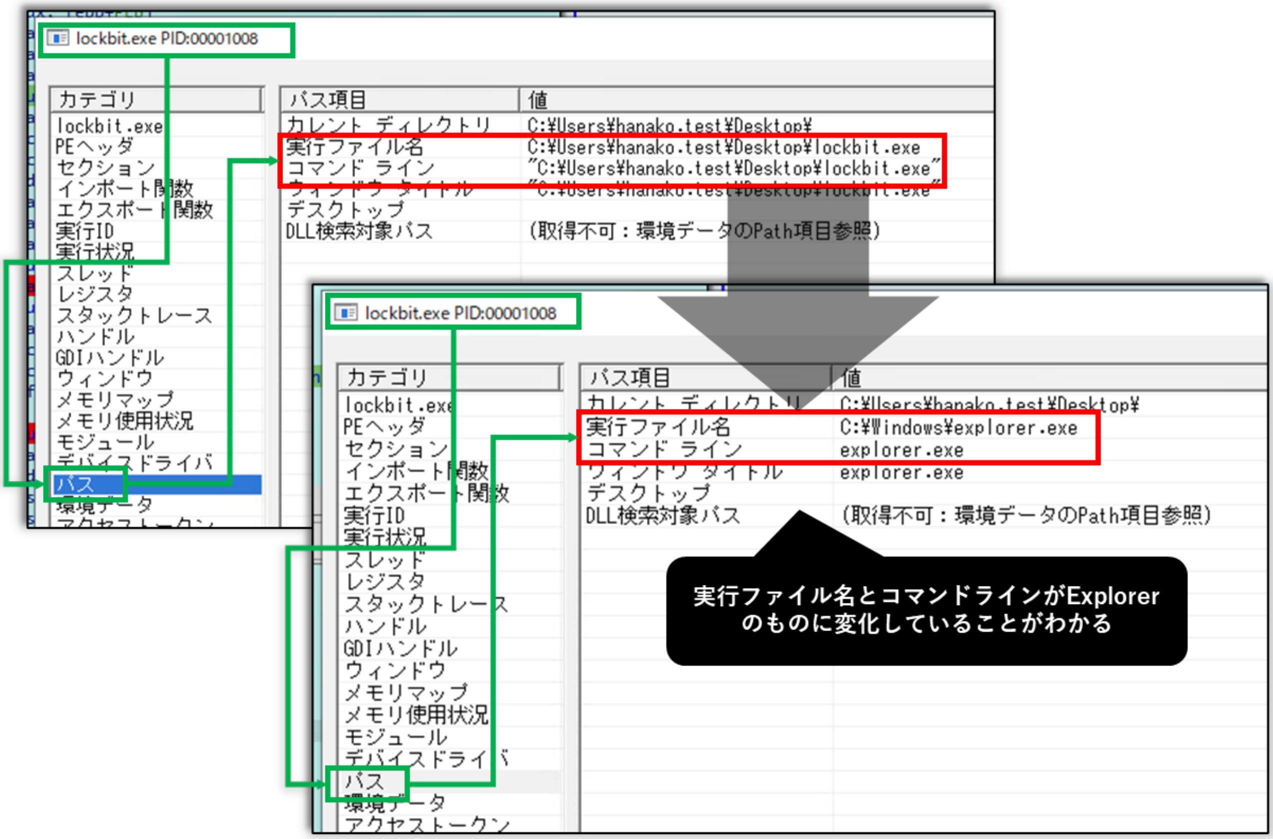


図 19　マスカレード処理によって改変され Explorer に偽装した LockBit2.0 のプロセス情報

続いて LockBit2.0 は、CoInitializeEx 関数で COM インターフェイスの使用準備をしたのち、CoGetObject 関数で以下 2 つの特定の CLSID を指定し、対応するオブジェクト名を作成します（下図参照）。

- 指定される CLSID：｛3E5FC7F9-9A51-4367-9063-A120244FBEC7｝
  - 作成されるオブジェクト名(COM 昇格モニカ)：" Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}"

- 指定される CLSID：｛D2E7041B-2927-42fb-8E9F-7CE93B6DC937｝
  - 作成されるオブジェクト名(COM 昇格モニカ)：" Elevation:Administrator!new:{D2E7041B-2927-42fb-8E9F-7CE93B6DC937}"

これらの CLSID は UAC Bypass の観点で脆弱であることが一部で知られている、それぞれ CMSTPLUA と IColorDataProxy という自動昇格が有効な COM インターフェイスに紐づいています。

図 20　COM 昇格モニカの作成

上記のオブジェクト名(COM 昇格モニカ)が CoGetObject 関数の引数に渡され呼び出されることで、COM のホストプロセス（COM サロゲートプロセス）である「Dllhost.exe」（正規プロセス）が以下の引数で起動します（下図参照）。

■DllHost.exe（COMサロゲートプロセス）が起動する様子

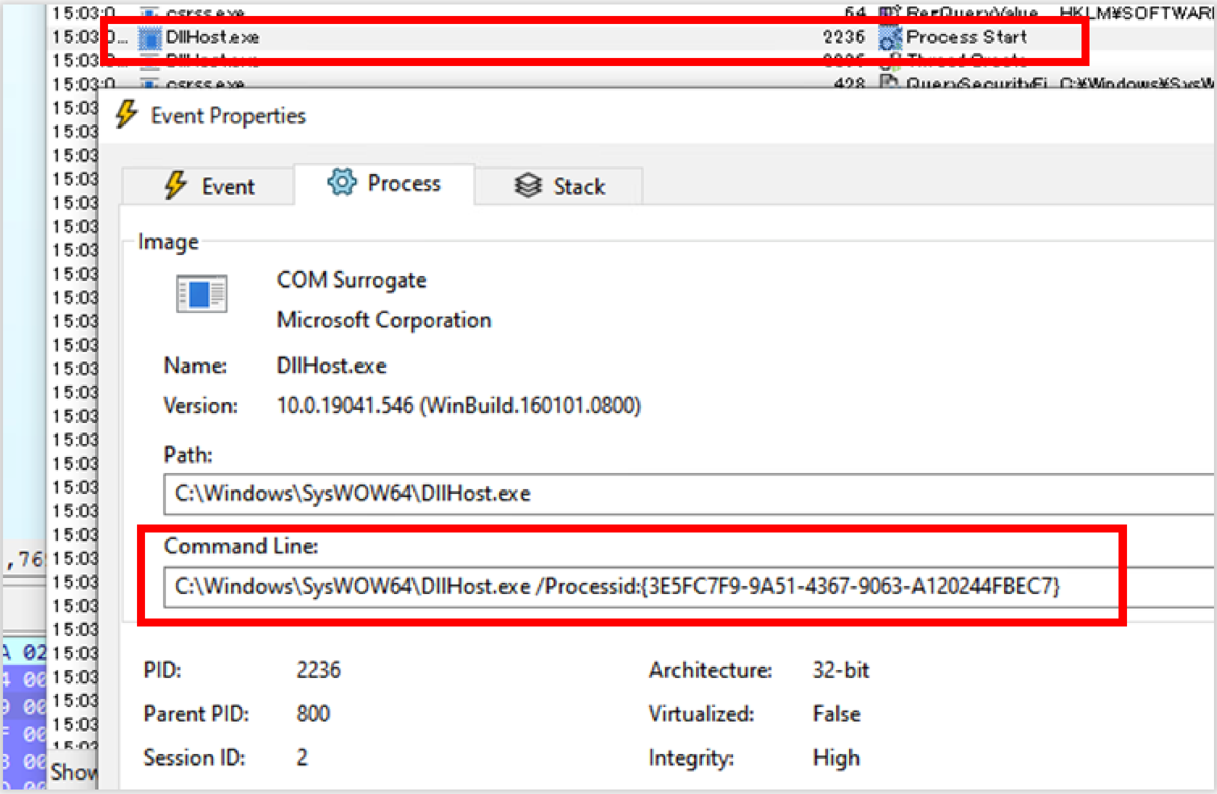


図 21　COM サロゲートプロセスが起動する様子

その後、LockBit2.0 は（Dllhost.exe を経由し）以下のレジストリ(今回利用する COM に関連した Display Calibrator というレジストリ)を書き換えることで、LockBit2.0 の EXE が UAC Bypass を行える状態にします（下図参照）。

■LockBit2.0が書き換えるレジストリ場所

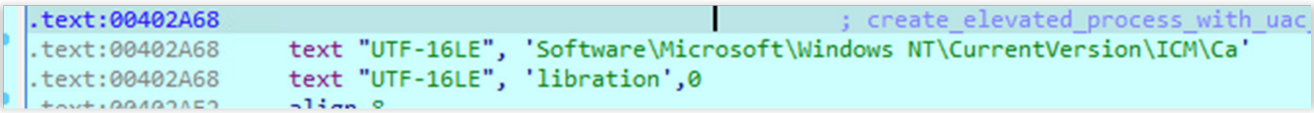


図 22　LockBit2.0 が UACbypass のために書き換えるレジストリ場所

以下は、書き換える処理と、書き換え前後の DisplayCalibrator のレジストリ値を比較した様子です（下図参照）。

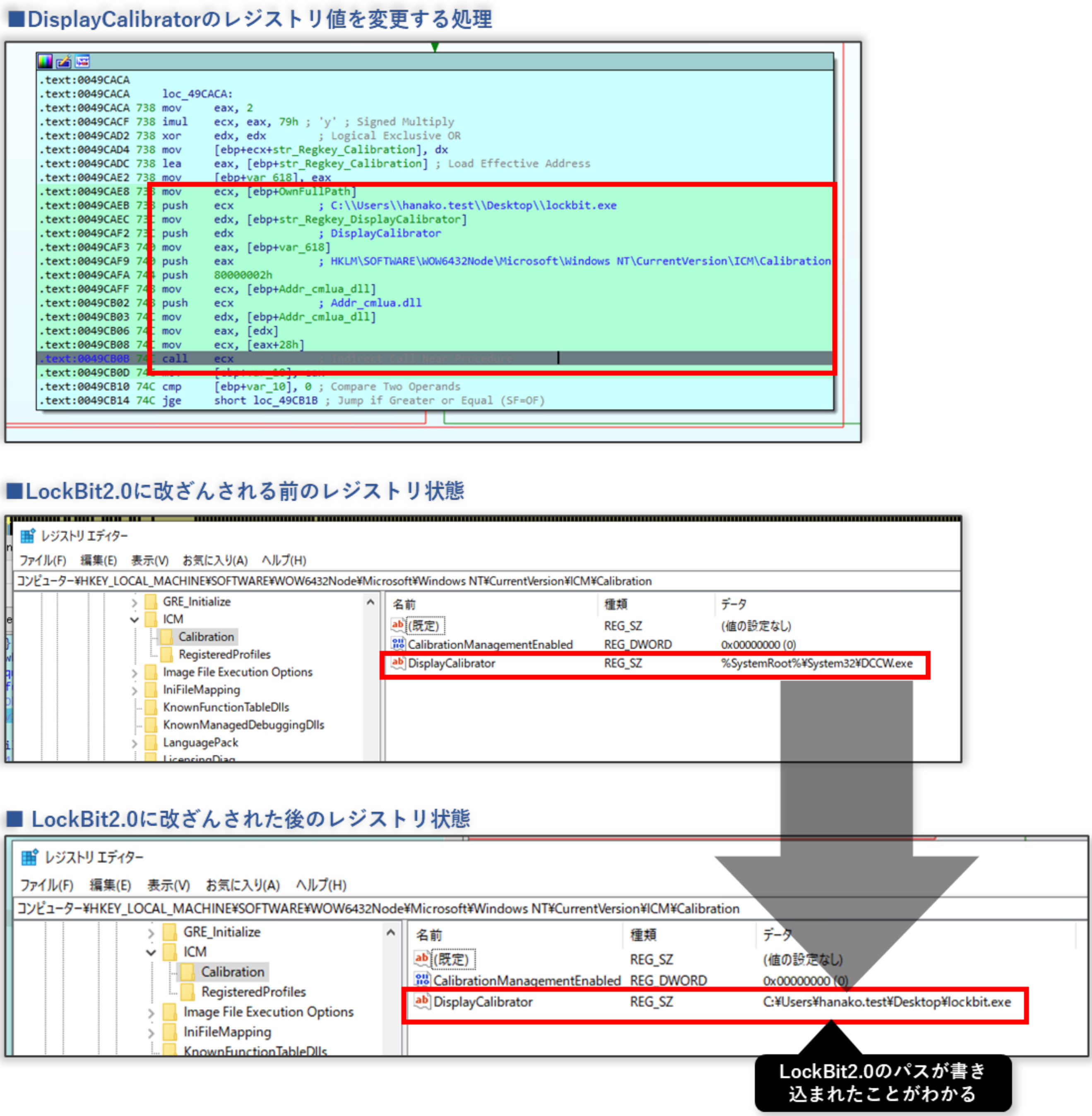
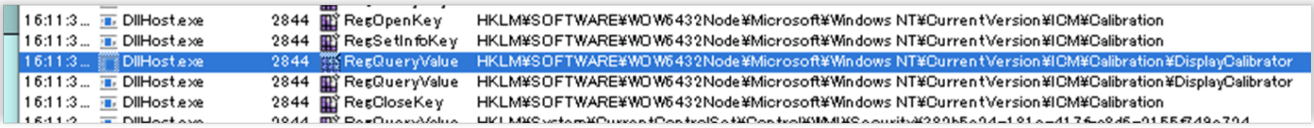


図 23　LockBit2.0 により書き換えられる前後のレジストリ値「DisplayCalibrator」

その後上記設定が反映される関連された COM 処理を呼び出すことで管理者権限に昇格した dllhost.exe が LockBit2.0 を起動させます（下図参照）。

■DllHost.exeがLockBit2.0に改ざんされたDisplayCalibratorのレジストリ値を読み取る様子



■その結果、DllHost.exeがLockBit2.0を管理者権限のプロセスとして起動する様子（継承による権限昇格）

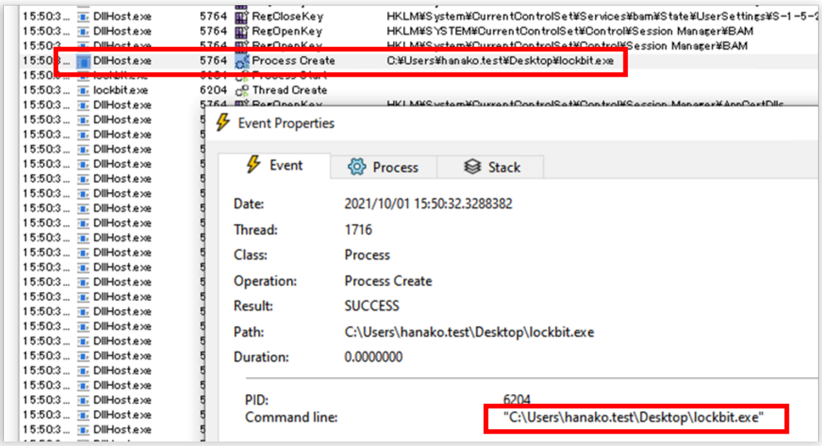


図 24 DllHost.exe が LockBit2.0 を起動する様子

その際管理者権限が継承され、結果的に LockBit2.0 が自動的に権限昇格した状態で新たなプロセスとして起動することになります（下図参照）。

■UAC Bypassにより権限昇格が行われた瞬間の様子

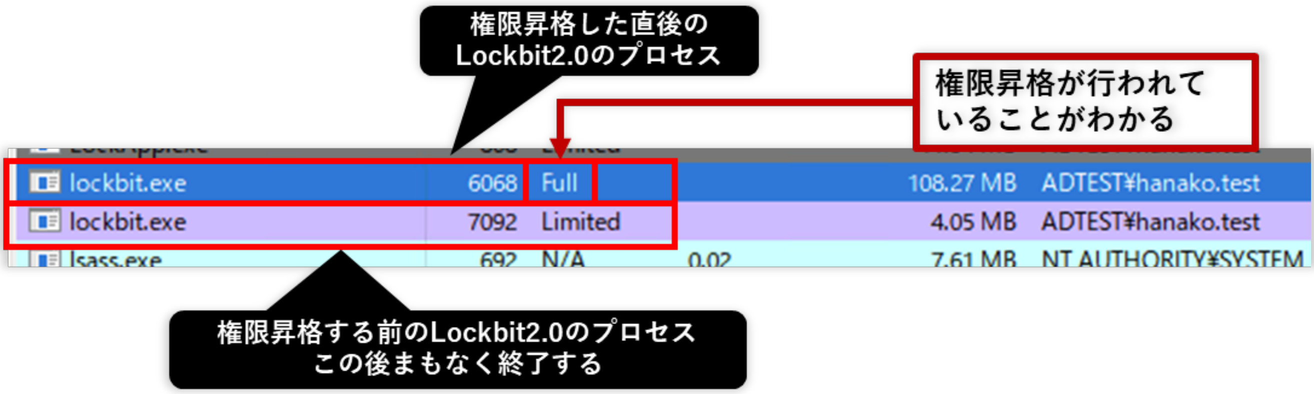


図 25 UAC Bypass により権限昇格が行われた瞬間の様子

以上が、LockBit2.0 が用いる、あまり広く詳細が知られておらず文書化されていない特定の COM インターフェイスを悪用した UAC Bypass のテクニックの仕組みです。

■多重起動を防ぐミューテックス（Mutex）の作成

LockBit2.0 は、アプリケーションが多重起動を防止する際によく使用するミューテックス（Mutex）を作成することで、自身のプロセスが多重に起動することを防ぎます（下図参照）。

■ミューテックス（Mutex）を作成することで自身の多重起動を防ぐ。作成できなければすでに起動済みと判断し自身を終了する。

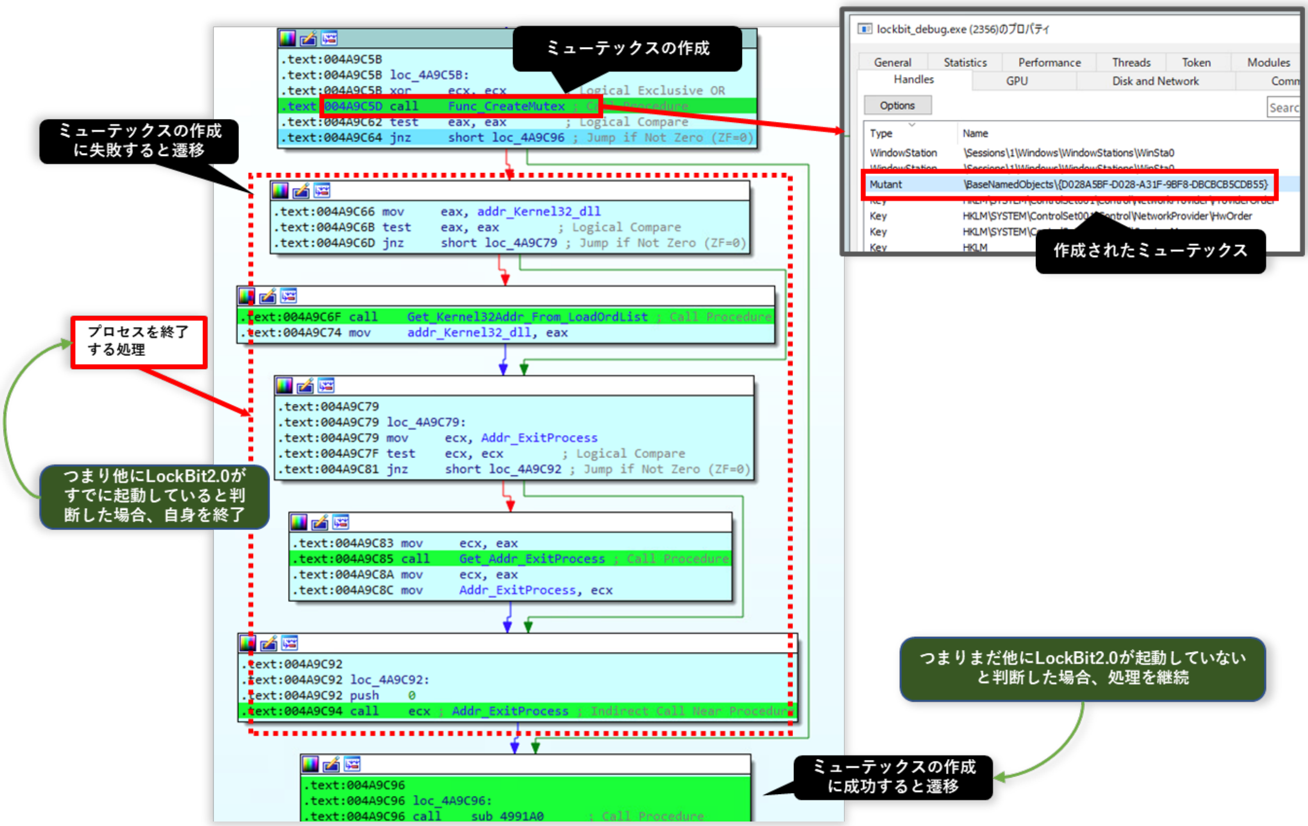


図 26　LockBit2.0 が作成するミューテックス

■不意のシャットダウンに備えた暗号化の再開手口

LockBit2.0 は、ユーザがインシデントに気付き端末を急いで強制的にシャットダウンするケースに備え、端末が再起動された際も暗号化が再開できるよう、レジストリにある HKCU の Run キーに自身のパスを登録することで、自動起動設定を行います。この自動起動のためのレジストリ設定は、暗号化がすべて完了すると削除されます。つまり、暗号化の途中でシャットダウンされても端末の再起動後に暗号化が再開できるようにしており、念を入れて執拗にすべてを暗号化しようとする攻撃者の意図が垣間見えます（下図参照）。

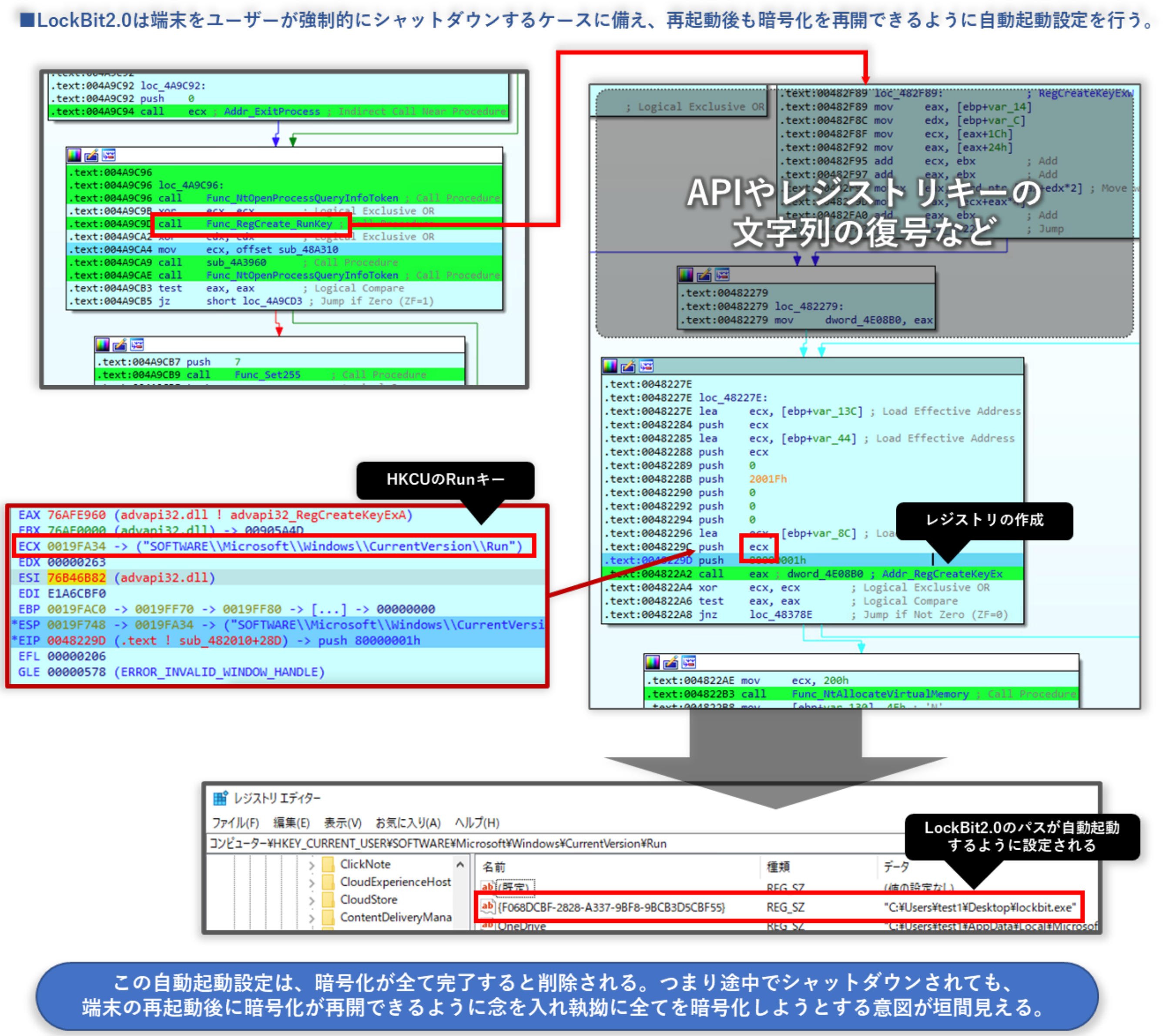


図 27　再起動時に再開できるように Run キーを一時的に設定

■シャットダウン時に最後まで居残る手口

また LockBit2.0 は、ユーザが端末をシャットダウンしようとした際に最後まで居残ることができるようにします。

具体的には、SetProcessShutdownParameters 関数を使用し、自身のプロセスがシャットダウン時に終了するプロセスの中で最後となるよう、シャットダウン時の終了順序を設定します。

これにより、端末のシャットダウン時に最後までシステムに居残ることで、できるだけ多くのファイルを暗号化しようとします。

■シャットダウンパラメータを変更し、シャットダウン時にシステムの最後まで居残るようにする

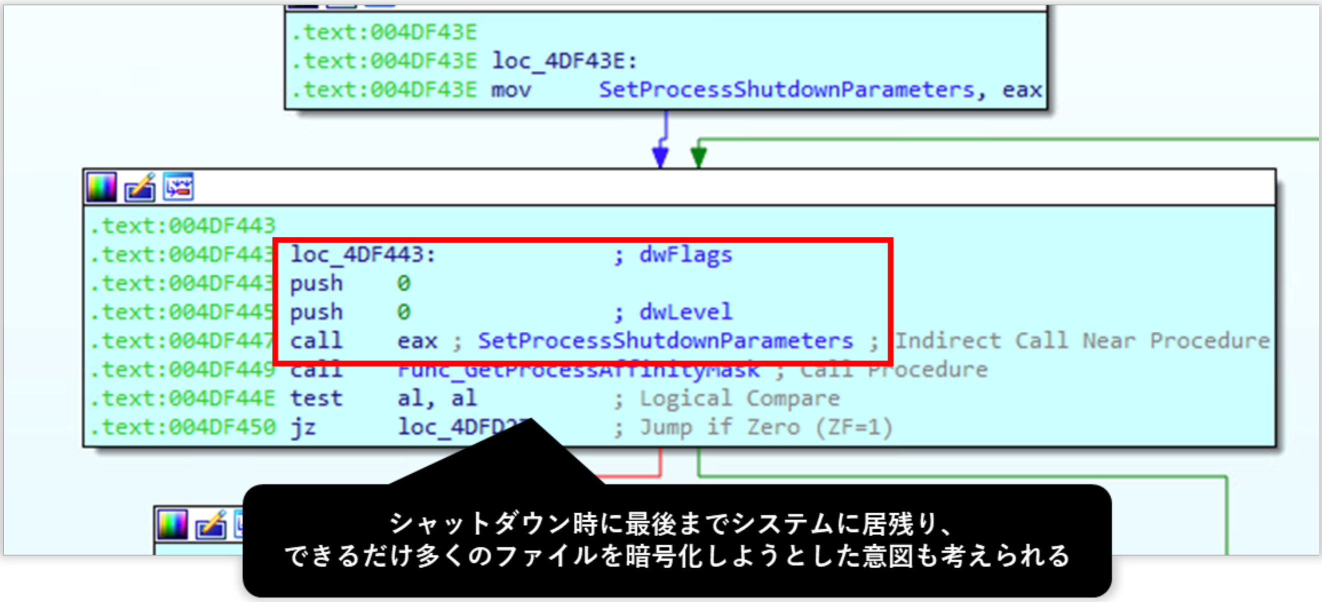


図 28　シャットダウンパラメータの変更

■プロセスの強制終了とサービスの停止

LockBit2.0 は動作中、繰り返し実行されるスレッドの中でプロセスの強制終了を定期的に行います。強制終了の対象プロセス名となる 137 個の文字列をメモリ上に復号/展開して用意します（下図参照）。

■強制終了対象プロセス名となる137個の文字列を復号/展開して用意し、それぞれの文字列ごとのアドレスを配列として格納し扱う

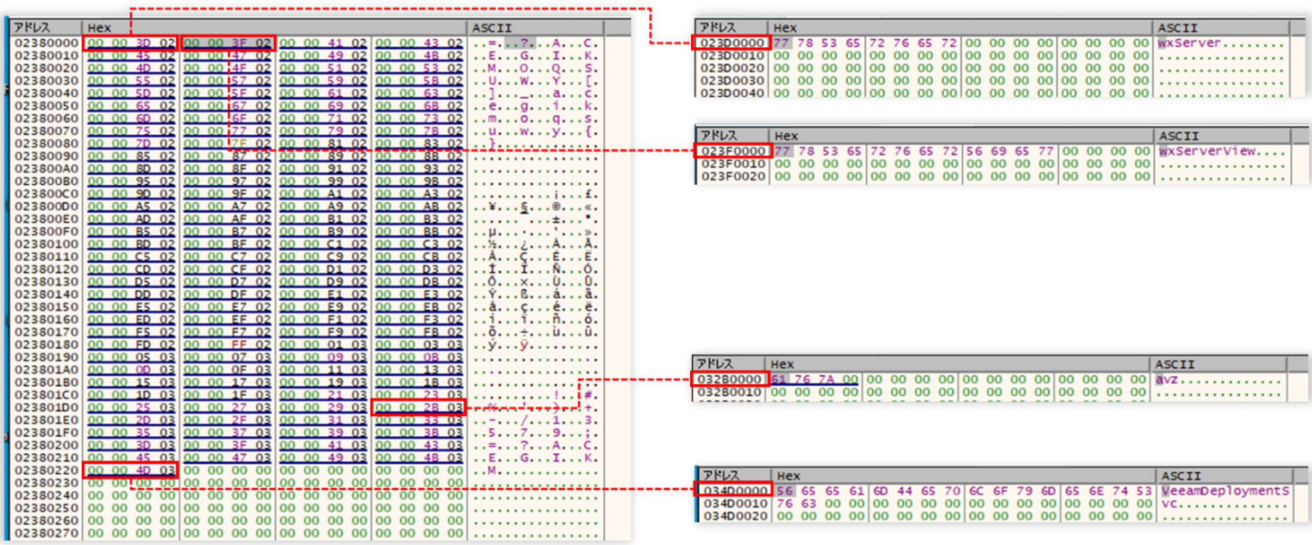


図 29　強制終了対象プロセスの展開

プロセスの強制終了は、ネイティブ API である ZwTerminateProcess が利用されます。

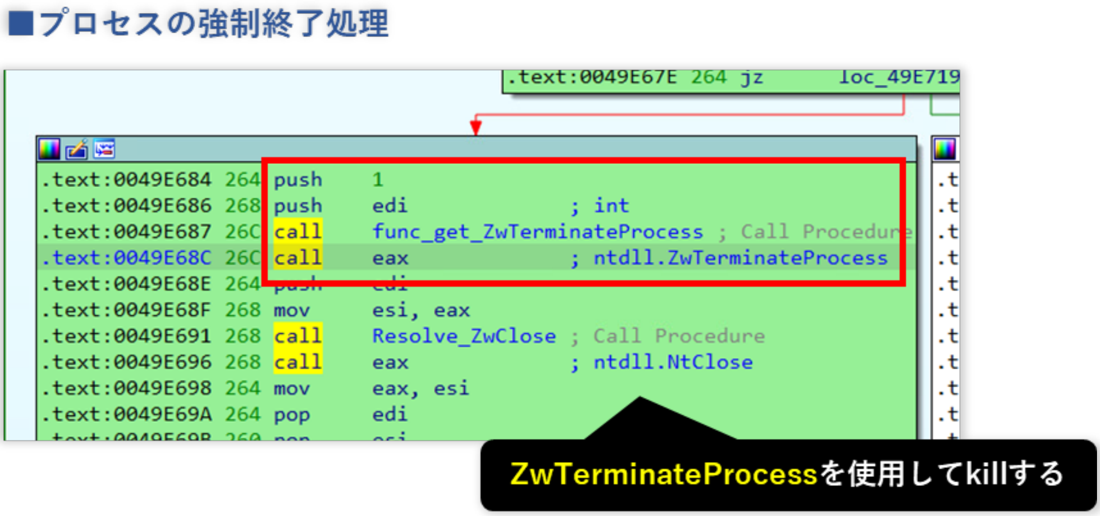


図 30　プロセスの強制終了処理

以下は、LockBit2.0 が強制終了対象とする 137 個のプロセス名一覧となります（下図参照）。
これらには、インシデント調査に使用されるプロセスや、業務アプリケーション、データベース系アプリケーションなどのプロセスが含まれており、これらのプロセスを強制終了させることで LockBit2.0 の暗号化作業がスムーズに行われるようにします。

■強制終了対象となる137個のプロセス名一覧(拡張子を省く)

wxServer	acwebbrowser	TitanV	outlook	j0gnjko1	Sqlservr
wxServerView	Creative Cloud	Ssms	powerpnt	Autoruns	RTVscan
sqlmangr	Adobe Desktop Service	notepad	steam	Autoruns64	sqlbrowser
RAgui	CoreSync	RdrCEF	thebat	Autoruns64a	tomcat6
supervise	Adobe CEF	sam	thunderbird	Autorunsc	QBIDPService
Culture	Helper	oracle	visio	Autorunsc64	notepad++
Defwatch	node	ocssd	wordpad	Autorunsc64a	SystemExplorer
winword	AdobeIPCBroker	dbsnmp	bedbh	Sysmon	SystemExplorerService
QBW32	sync-taskbar	synctime	vxmon	Sysmon64	SystemExplorerService64
QBDBMgr	sync-worker	agntsvc	benetns	procexp64a	Totalcmd
qbupdate	InputPersonalization	isqlplussvc	bengien	procmon	Totalcmd64
axlbridge	AdobeCollabSync	xfssvccon	pvlsvr	procmon64	VeeamDeploymentSvc
httpd	BrCtrlCntr	mydesktopservice	beserver	procmon64a	
fdlauncher	BrCcUxSys	ocautoupds	raw_agent_svc	ADExplorer	
MsDtSrvr	SimplyConnectionManager	encsvc	vsnapvss	ADExplorer64	
java	Simply.SystemTrayIcon	tbirdconfig	CagService	ADExplorer64a	
360se	fbguard	mydesktopqos	DellSystemDetect	tcpview	
360doctor	fbserver	ocomm	EnterpriseClient	tcpview64	
wdswfsafe	ONENOTEM	dbeng50	ProcessHacker	tcpview64a	
fdhost	wsa_service	sqbcoreservice	Procexp64	avz	
GDscan	koaly-exp-engine-service	excel	Procexp	tdsskiller	
ZhuDongFangYu	TeamViewer_Service	infopath	GlassWire	RaccineElevatedCfg	
QBDBMgrN	TeamViewer	msaccess	GWCtlSrv	RaccineSettings	
mysqld	tv_w32	mspub	WireShark	Raccine_x86	
AutodeskDesktopApp	tv_x64	onenote	dumpcap	Raccine	

図 31　LockBit2.0 が強制終了する対象プロセス名リスト

また、サービスの停止も行いますが、その際 ControlService 関数に SERVICE_CONTROL_STOP を引数として渡すことで実施します（下図参照）。

図 32　サービスの強制停止処理

LockBit2.0 が停止させるサービス名は以下の 79 個であり、データベース系アプリケーション、バックアップ系アプリケーション、セキュリティ製品などのサービスが含まれています（下図参照）。

■停止対象となる79個のサービス名一覧

wrapper	dbsrv12	SQLAgent$SHAREPOINT	BackupExecDiveciMediaService
DefWatch	dbeng8	QBFCService	BackupExecJobEngine
ccEvtMgr	MSSQL$MICROSOFT##WID	QBVSS	BackupExecManagementService
ccSetMgr	MSSQL$VEEAMSQL2012	YooBackup	BackupExecRPCService
SavRoam	SQLAgent$VEEAMSQL2012	YooIT	MVArmor
Sqlservr	SQLBrowser	vss	MVarmor64
sqlagent	SQLWriter	sql	stc_raw_agent
sqladhlp	FishbowlMySQL	svc$	VSNAPVSS
Culserver	MSSQL$MICROSOFT##WID	MSSQL	VeeamTransportSvc
RTVscan	MySQL57	MSSQL$	VeeamDeploymentService
sqlbrowser	MSSQL$KAV_CS_ADMIN_KIT	memtas	VeeamNFSSvc
SQLADHLP	MSSQLServerADHelper100	mepocs	AcronisAgent
QBIDPService	SQLAgent$KAV_CS_ADMIN_KIT	sophos	ARSM
Intuit.QuickBooks.FCS	msftesql-Exchange	veeam	AcrSch2Svc
QBCFMonitorService	MSSQL$MICROSOFT##SSEE	backup	CASAD2DWebSvc
msmdsrv	MSSQL$SBSMONITORING	bedbg	CAARCUpdateSvc
tomcat6	MSSQL$SHAREPOINT	PDVFSService	WSBExchange
zhudongfangyu	MSSQLFDLauncher$SBSMONITORING	BackupExecVSSProvider	MSExchange
vmware-usbarbitator64	MSSQLFDLauncher$SHAREPOINT	BackupExecAgentAccelerator	MSExchange$
vmware-converter	SQLAgent$SBSMONITORING	BackupExecAgentBrowser	

図 33　停止対象となる 79 個のサービス名一覧

■ゴミ箱を空にする

LockBit2.0 にはごみ箱を空にする挙動があり、全てのドライブにあるごみ箱を SHEmptyRecycleBin 関数で空にしていきます（下図参照）。

■全てのドライブのゴミ箱を空にする

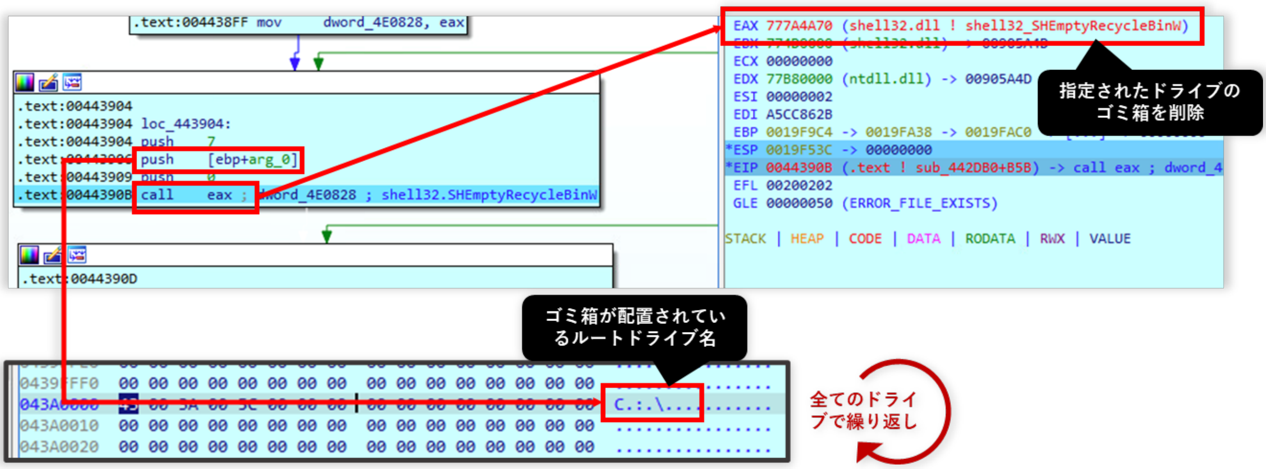


図 34　全てのごみ箱を空にする処理

■隠しパーティションのスキャンと強制ドライブマッピング

LockBit2.0 は、FindFirstVolumeW および FindNextVolumeW で、ローカルドライブにマッピングされていない隠しパーティションを含むコンピュータ上の全てのボリュームをスキャンし、マウントされていない場合は SetVolumeMountPointW で空きのあるドライブ名を指定しドライブマッピングしていきます。
これにより以下のように通常は論理ドライブとして見えない隠しパーティション領域も強制的に論理ドライブとしてマウントされてしまいます。この結果、隠しパーティションであっても LockBit2.0 の暗号化対象として認識されるようになります（下図参照）。

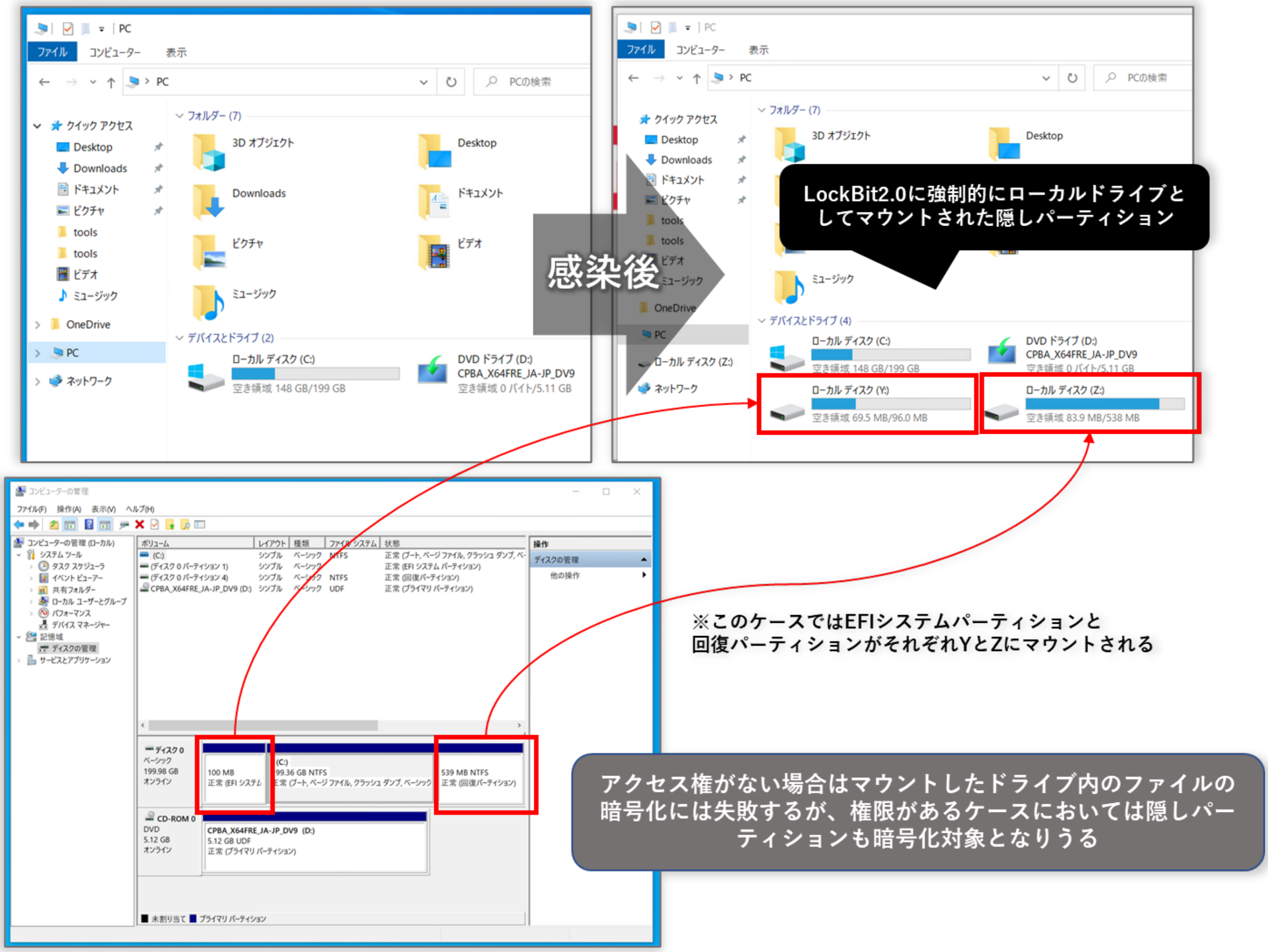


図 35　隠しパーティションの強制ドライブマッピング

■ファイル暗号化処理の高速化

冒頭で触れた通り、LockBit2.0 の開発者は自身のサイト上で、ほかのランサムウェアと比較し LockBit2.0 が世界で最も高速であると自負しアピールしています（下図参照）。

■ LockBit2.0の開発者は自身のサイト上で、他のランサムウェアと比較しLockBit2.0が最も高速であると自負している

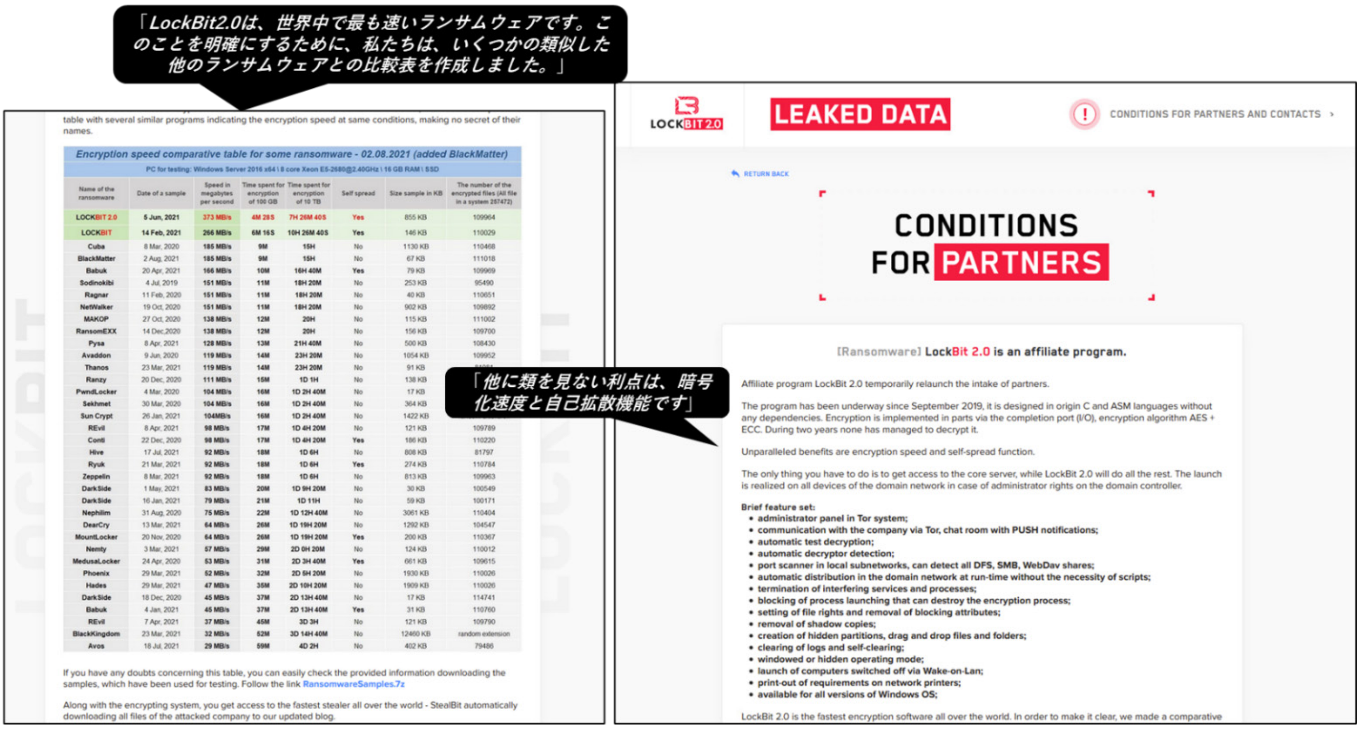


図 36　最も高速であると自負する LockBit2.0 のサイト上のアピール文

そのため、LockBit2.0 のファイルの暗号化に関わる処理を詳しく調べたところ、実際にファイルの暗号化処理を高速化するための複数のテクニックが実装されていることを確認しました。

■「I/O 完了ポート」の採用による処理の高速化

まず一つ目に挙げられるのは、「I/O 完了ポート」(I/O Completion Port, IOCP)の採用によるマルチスレッドの高速化です。

「I/O 完了ポート」とは、マルチスレッドの処理において、ファイルの読み書きなど CPU の範囲外での待機が発生するような、CPU から見ると時間を要する入出力処理(I/O)で CPU を無駄に待機させず、空き時間でほかの処理に効率的にリソースを割り振り管理するための仕組みで、サーバアプリケーションなどで採用されている高速化の仕組みです(下図参照)。

■ LockBit2.0で使用されている「I/O完了ポート」(I/O Completion Port, IOCP)の概念と仕組み

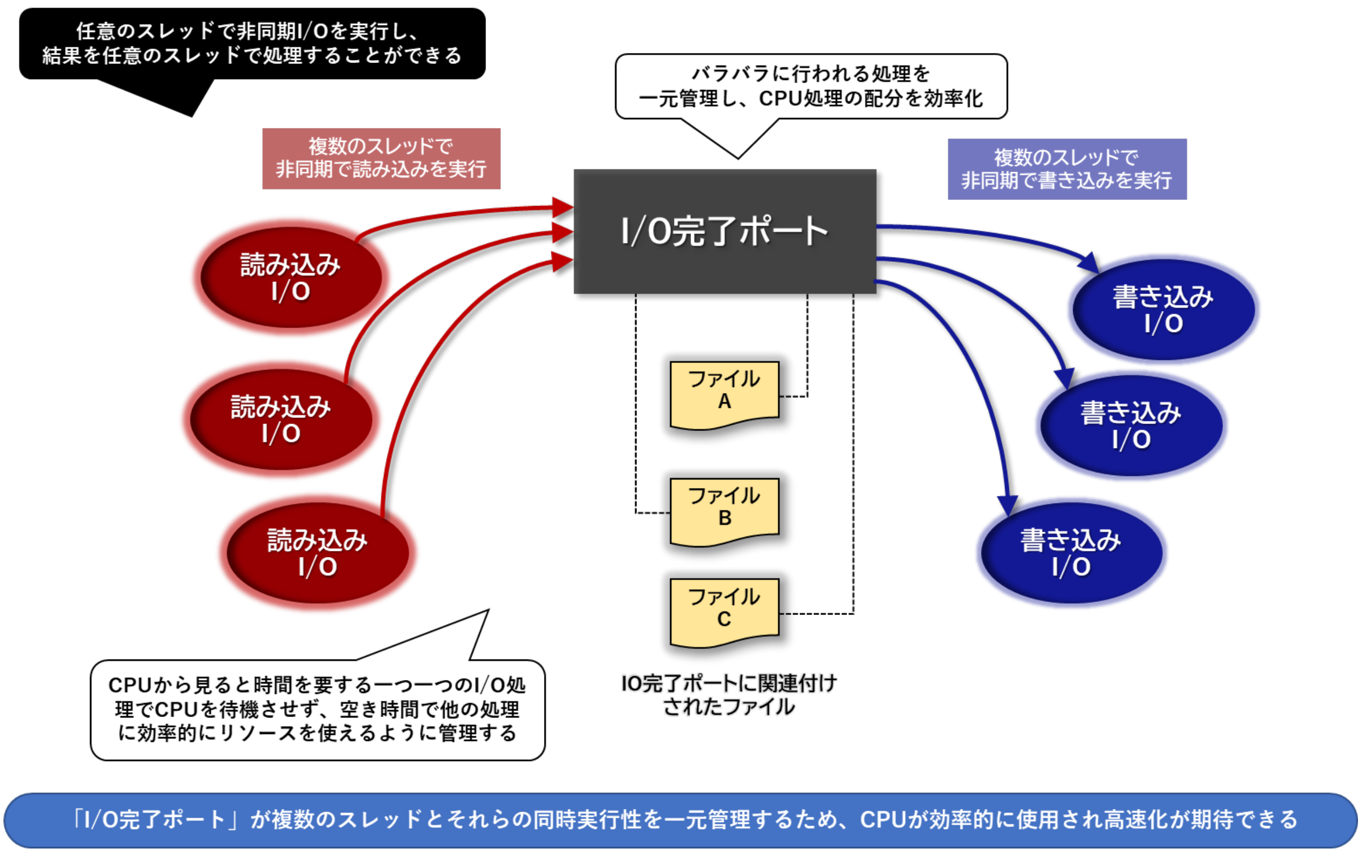


図 37 LockBit2.0 で使用されている「I/O 完了ポート」の概念と仕組み

従来の一般的なマルチスレッドによる並列方法ではスレッドが消費するリソースやスレッドの開始・終了に伴うコスト(OS にイベントが飛ぶなど)、スレッドの切り替えに伴うコストなどがあり、CPU に対する負担が大きく、また特に、CPU の世界よりも圧倒的に遅いハードディスクへのアクセスなどの入出力処理(I/O)において、処理を待機している時間 CPU を占領させることが非効率であるとも言えます。

そこで LockBit2.0 は、そうした観点が考慮された仕組みである「I/O 完了ポート」を採用することで効率化を図っています。I/O 完了ポートを利用することで、CPU の世界から見て時間の要する入出力処理（I/O）が完了するまで CPU リソースを他の処理のために一旦開放し、入出力処理（I/O）が完了したタイミングでスレッドプール（複数のスレッドを待機させせつつ空いたスレッドに適宜処理させる仕組み）に処理依頼(キュー)を渡すという流れで CPU に処理を再開させることができ、CPU の占有を抑えることで効率的に CPU リソースを活用したマルチスレッドを実現できます。また、同時にアクティブとなるスレッドの数を指定することができるため、プロセッサ数以上のスレッドを動作させないといった制御が可能です。

このため、多数の I/O を並行発生させることで高速化したい「ランサムウェア」というプログラムにとって I/O 完了ポートの採用は効果的な選択であると言えるでしょう。

具体的には、ネイティブ Windows API である NtCreateIoCompletion、NtSetInformationFile、ZwRemoveIoCompletion、ZwSetIoCompletion という関数群を呼び出すことで実現しています。

下図は、実際に LockBit2.0 がファイルを暗号化する際に使用する I/O 完了ポートの利用流れを図示したものです。ファイルを暗号化する際の入出力処理（I/O）において、I/O 完了ポートが一元管理することで CPU の処理が効率化されます（下図参照）。

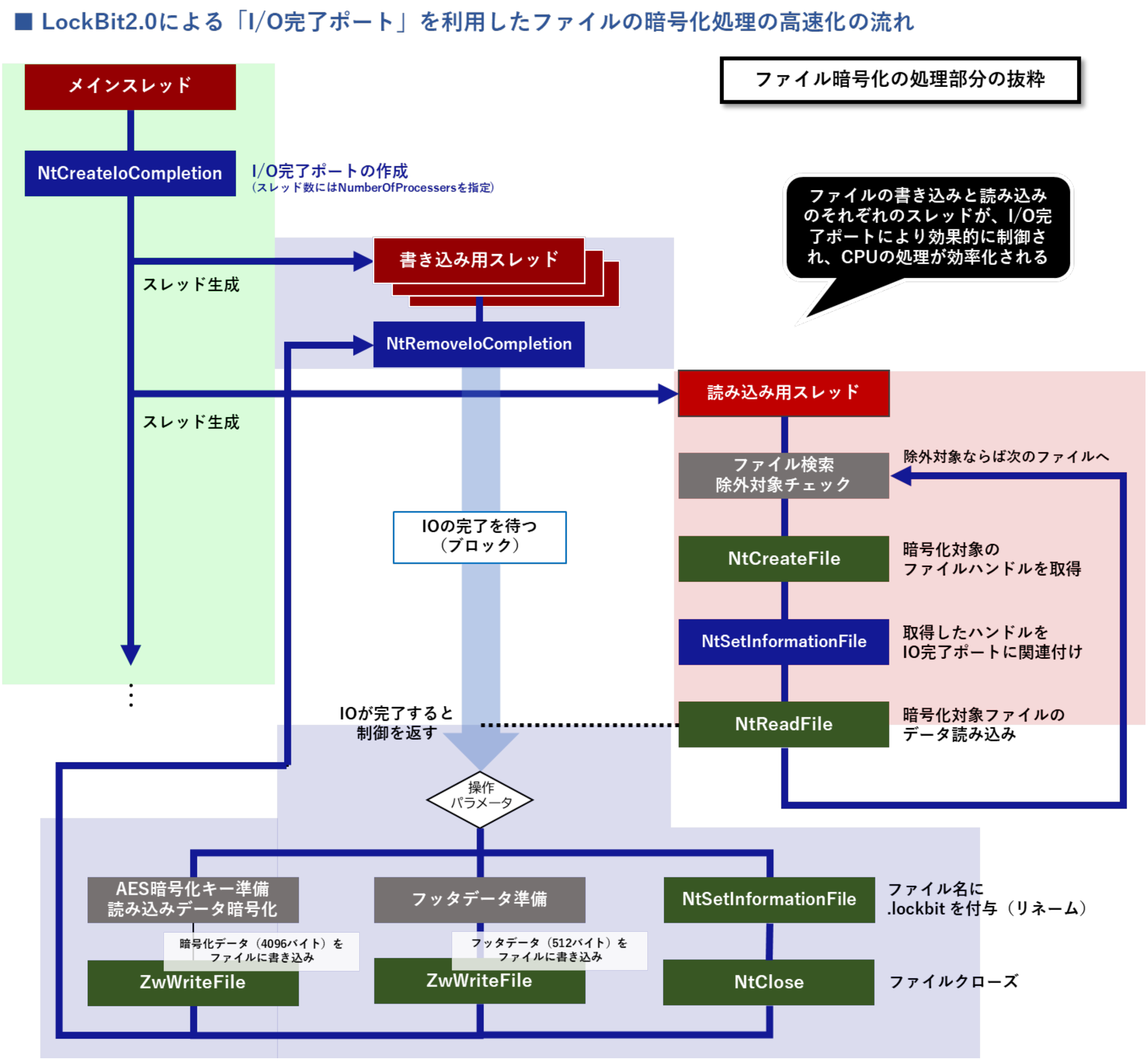


図 38　I/O 完了ポートを利用した LockBit2.0 のファイル暗号化処理の高速化の仕組み

つまり、I/O 完了ポートの採用により、実際に暗号化処理の高速化が期待できるといえます。
また、LockBit2.0 はファイルの暗号化処理だけでなくその他の機能においてもマルチスレッドでサブルーチン化しておりそれぞれにおいて I/O 完了ポートを活用した高速化を行なっています。

■「AES-NI」の活用による暗号化処理の高速化

インテルと AMD の一部の CPU には、AES の暗号処理を高速に行う為の「AES-NI」（Advanced Encryption Standard New Instructions）という仕組みが実装されています。AES-NI は、AES の暗号化/復号処理のために特別に用意された専用の命令セットをハードウェアに実装することで大幅な高速化が行われる機能です。

LockBit2.0 はファイルの暗号化に AES を使用するため、AES-NI を活用することでファイルの暗号化処理を高速化しています。

具体的には、LockBit2.0 はまず感染端末の CPU の種類を示す「CPUID」や CPU の製造ベンダーを示す「ベンダーID」を確認することで、AES-NI に対応した CPU であるかを確認します。さらに AES-NI をサポートしている CPU であるかどうかをチェックします。

これらのチェックの結果、AES-NI に対応していると判断した場合は、LockBit2.0 は AES-NI 専用に用意された命令セットである「aesenc」命令を使用して暗号化します（下図参照）。

■ LockBit2.0は「AES-NI」(Advanced Encryption Standard New Instructions)を活用することで暗号化処理を高速化

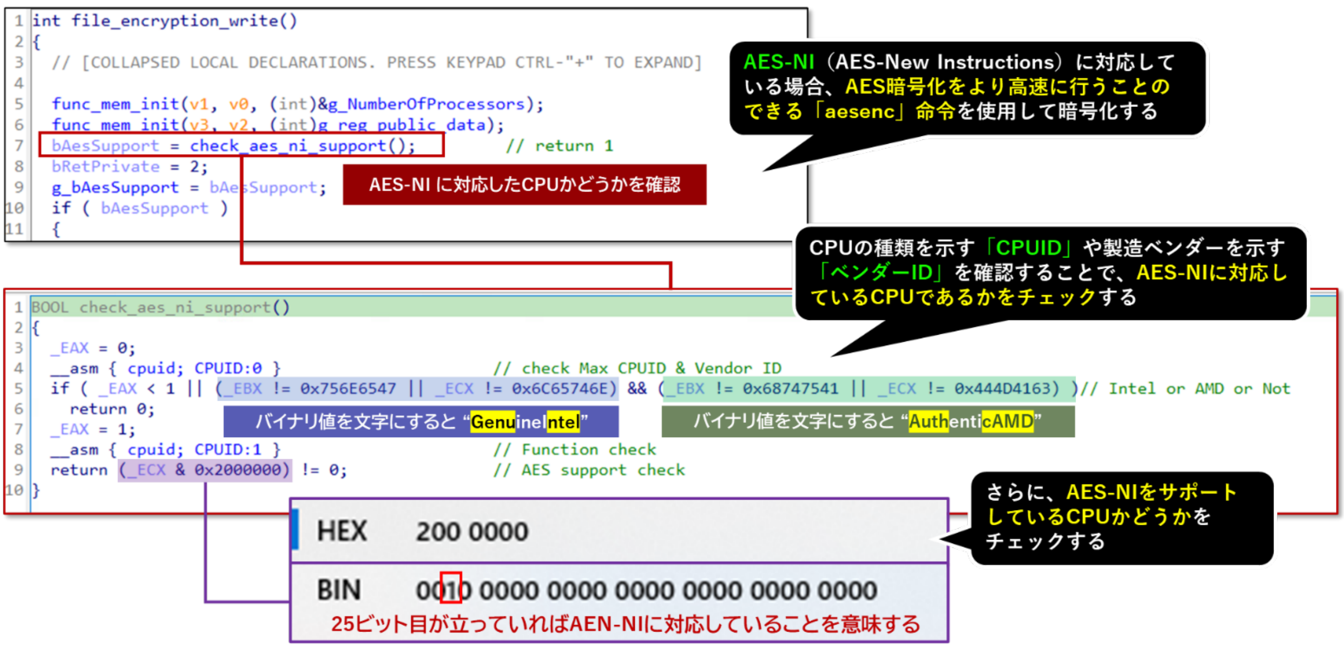


図 39　AES-NI を活用した LockBit2.0 のファイル暗号化処理の高速化

つまりこの AES-NI の利用により、実際に暗号化処理の高速化が期待できるといえます。

■ファイルの先頭部分のみの暗号化による高速化

LockBit2.0 は、暗号化する対象ファイルの中身全てを暗号化せず、ファイルの先頭部分 4KB のみを暗号化することでファイルの暗号化を高速化しています。以下の図は、LockBit2.0 に暗号化される前のファイルと暗号化された後のファイルの先頭部分を比較した様子ですが、先頭 4KB のみが暗号化されていることがわかります（下図参照）。

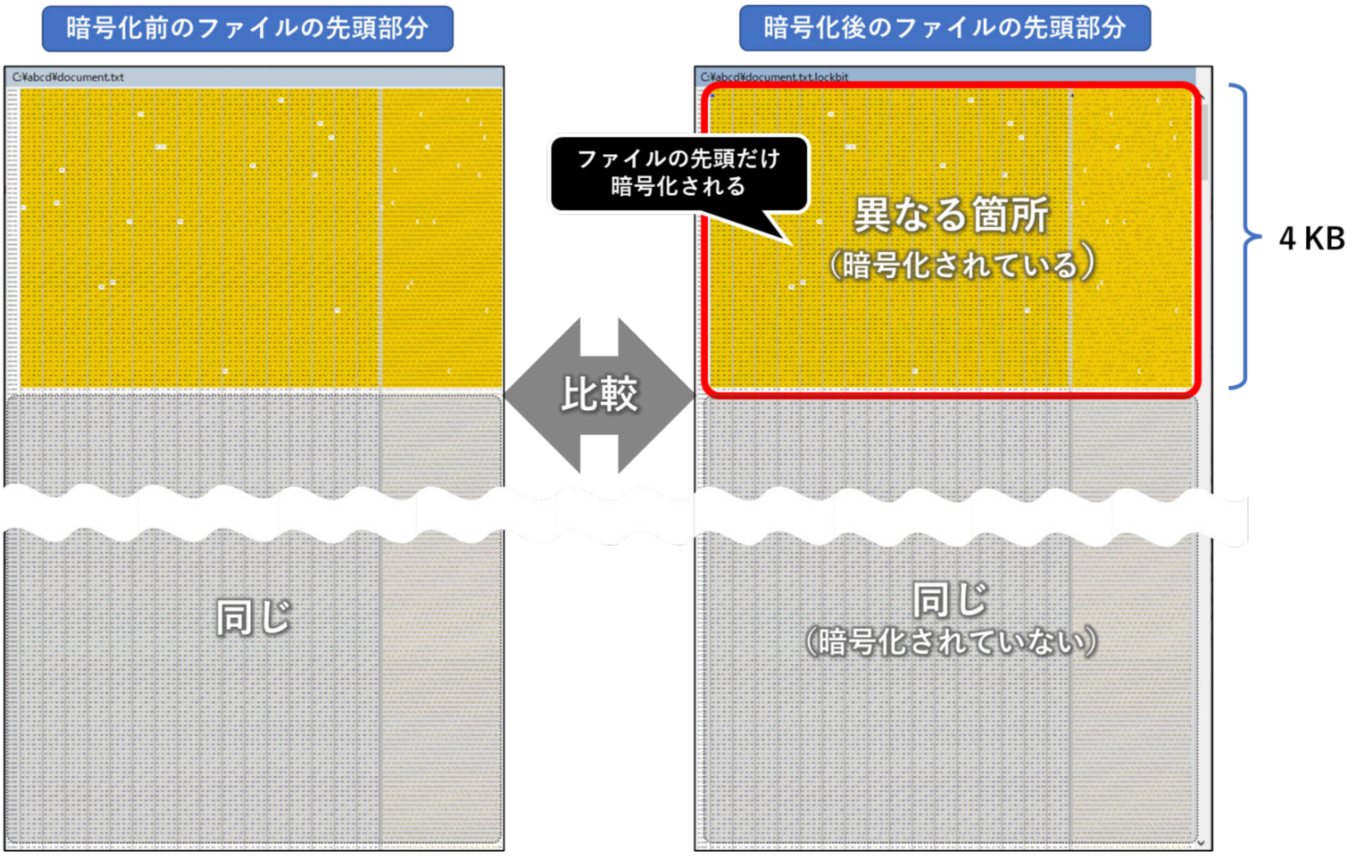


図 40　LockBit2.0 に暗号化される前後のファイルの先頭部分の比較

また別の例として 1GB のファイルサイズを持つ巨大なファイルを用意し実験した結果が以下ですが、LockBit2.0 によって暗号化された後のファイルを確認すると、先頭 4KB しか暗号化されていません（下図参照）。

■ LockBit2.0により暗号化された1GBのファイルの状態

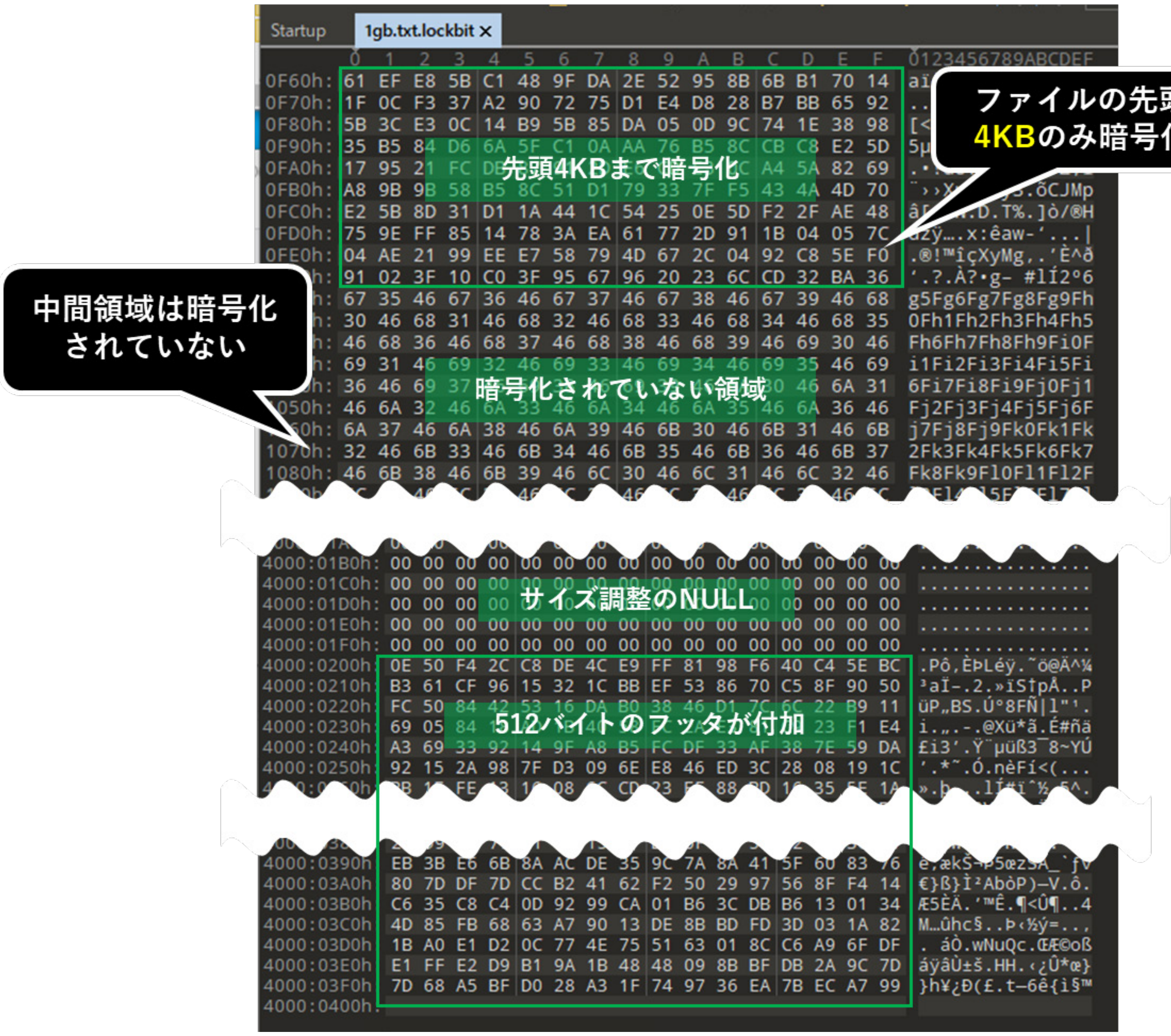


図 41　LockBit2.0 により暗号化された 1GB のファイルの状態

確かに、他のランサムウェアにおいて、多数の大きなサイズのファイルが暗号化されるようなシチュエーションでは、暗号化スピードが大幅に減速することをこれまでしばしば目にしてきました。Conti の解析記事（詳しくは [ここをクリック] してご覧ください）においても、ファイルの拡張子やサイズによって暗号化処理を分けていることが分かっており、ランサムウェアにとってこの減速が課題であることは明らかです。LockBit2.0 が採用したこの方法のように、ファイルの先頭 4KB のみを暗号化するという割り切りは、暗号化スピードの向上の観点においては非常に効果的なアイデアであると感じます。

また、4KB とはいえ、プレーンテキストの種類のファイルでない限り、ファイルの先頭 4KB を暗号化された場合は元の状態に復元することは非常に困難であり、結果的にファイル全体を暗号化された場合と同等の影響を被害組織に与えられる手口であるともいえます。

■LockBit2.0 のファイル暗号化ロジックの詳細

ここでは、LockBit2.0 が行うファイルの暗号化のロジックを詳細解説します。
LockBit2.0 のファイル暗号化のロジックを次の図にまとめましたので、図に沿って解説します（下図参照）。

まず、LockBit2.0 の攻撃者側は公開鍵暗号方式である「ECC」（楕円曲線暗号）鍵のペア（これを「マスターECC 公開鍵」と「マスターECC 秘密鍵」と呼びます）を生成し、「マスターECC 秘密鍵」を攻撃者側の手元で管理しています。
そして、LockBit2.0 の EXE 本体にこの「マスターECC 公開鍵」がハードコードされています。

次に、LockBit2.0 は感染時、感染端末ごとに一意となる別の ECC 鍵のペア（これを「セッション ECC 公開鍵」と「セッション ECC 秘密鍵」と呼びます）をローカルで生成します（※）。
（※）なお LockBit2.0 は ECC（楕円曲線暗号）の楕円曲線に「Curve25519」を使用しています。RSA より処理速度の早い ECC を採用している点で高速化を狙った意図等が考えられます。

一方で、LockBit2.0 は暗号化対象のファイルを暗号化する際 AES 暗号で暗号化しますが、その際に使用される AES 鍵（暗号鍵＆IV）はファイルごとに動的生成されます。つまり、AES 鍵（暗号鍵＆IV）はファイルごとに異なります。

ではファイルを暗号化する際のロジックを見ていきましょう。
LockBit2.0 はあるファイルを暗号化する際、そのファイルのフッタに、「セッション ECC 公開鍵」で暗号化した「AES 鍵（暗号鍵＆IV）」と、「マスターECC 公開鍵」で暗号化した「セッション ECC 秘密鍵」を追記して書き込みます。

したがって、「マスターECC 秘密鍵」があれば、対となる「マスターECC 公開鍵」で暗号化された「セッション ECC 秘密鍵」が抽出でき、さらにその「セッション ECC 秘密鍵」があれば、対となる「セッション ECC 公開鍵」で暗号化された「AES 鍵（暗号鍵＆IV）」を取り出すことができます。
つまり、ファイルの復号は「マスターECC 秘密鍵」を持った攻撃者のみが行えることを意味します。

■ LockBit2.0の暗号化のロジック図(1) 暗号化鍵が使用される流れと関係性

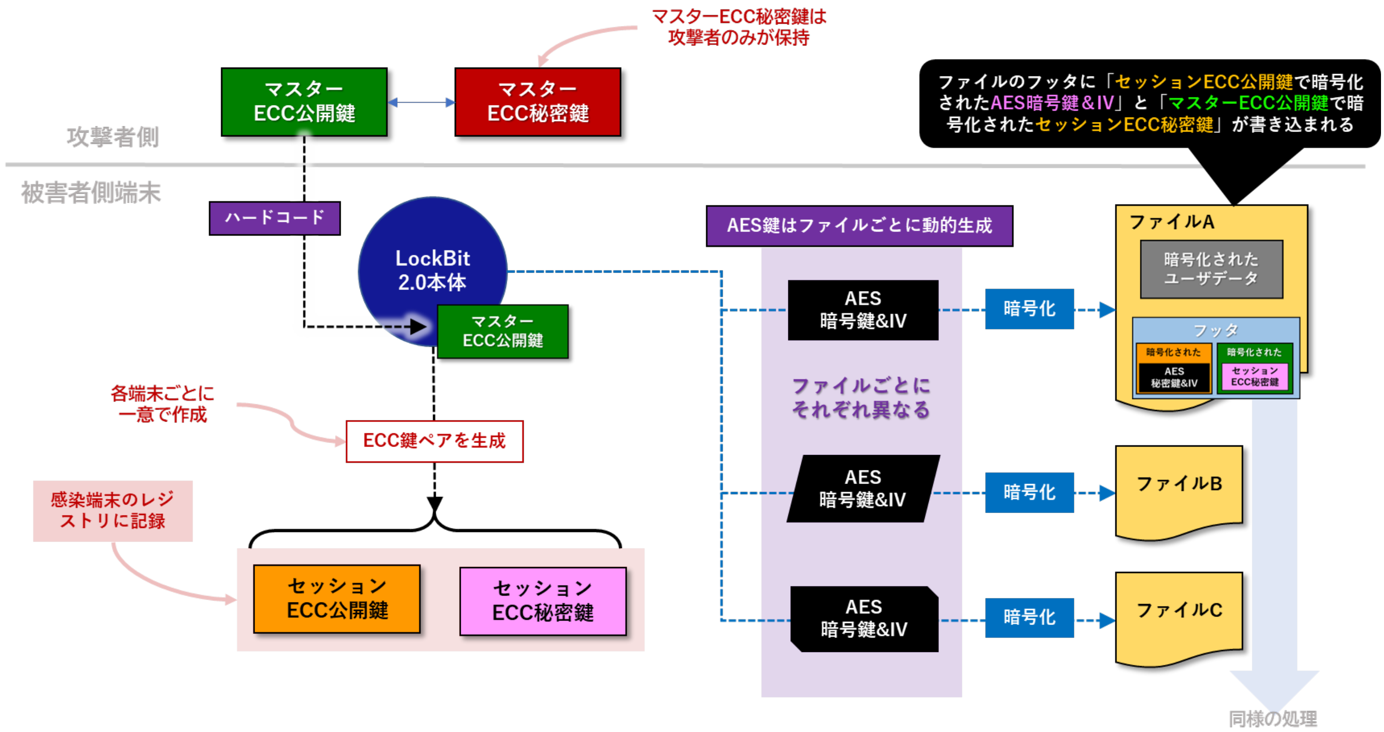


図 42　LockBit2.0 の暗号化ロジック図

上図にも一部記載していますが、LockBit2.0 はセッション鍵のペア（「セッション ECC 公開鍵」と「セッション ECC 秘密鍵」）を生成した際にレジストリに書き込むことで記録します（下図参照）。
なお、レジストリに記録されるセッション ECC 秘密鍵は暗号化された状態で書き込まれ、メモリ上に生成されていたセッション ECC 秘密鍵はすぐに 0xFF で上書きされ消去されます。

■ LockBit2.0はセッション鍵のペア（セッションECC公開鍵とセッションECC秘密鍵）をレジストリに書き込む

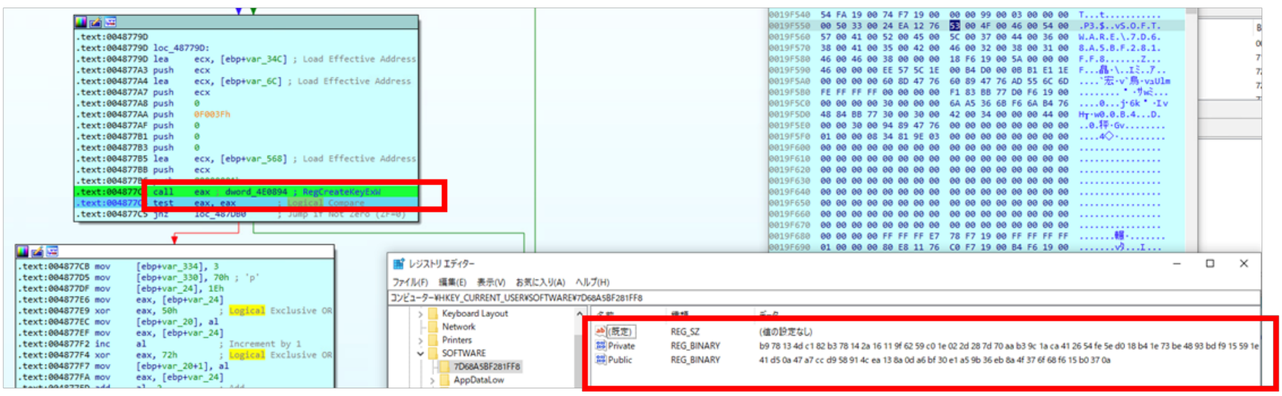


図 43　レジストリに書き込むセッション鍵のペア

この背景ですが、LockBit2.0 は起動した際、上記レジストリが存在した場合は前回の暗号化処理が途中であると判断し、これらのセッション鍵のペアを読み込むことで続きの暗号化に利用します。つまり、途中で端末がシャットダウンされるなどで中断された場合においても同じセッション鍵で暗号化が再開できるようにしていると考えられます（下図参照）。

■ **LockBit2.0は起動時にセッション鍵のペアをレジストリから取得することで、途中で端末がシャットダウンされるなどで中断された場合でも暗号化が再開できるようにしている**

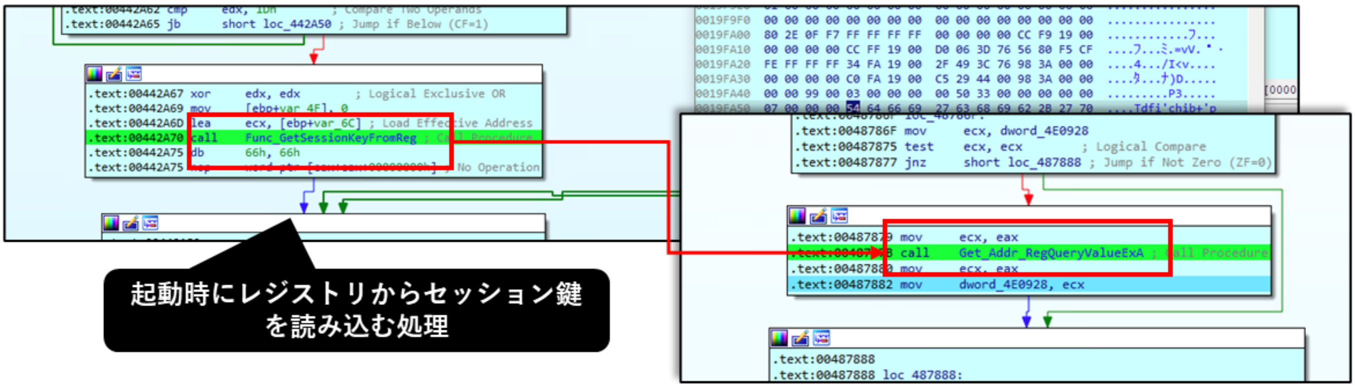


図 44　起動時にセッション鍵を読み込む処理

なお、ファイルを暗号化する際の AES 暗号鍵の作成方法ですが、BcryptGenRandom 関数を使用し 32 バイトのランダムなバイト列を生成します。この時点で鍵長が 32 バイト=256 ビットである可能性が考えられましたが、BcryptGenRandom を使用し生成した 32 バイトのランダムなバイト列のうち、最初の 16 バイトを IV(初期化ベクトル)として、残りの 16 バイトを AES 暗号鍵として使用することが確認され、この時点で 128 ビットの鍵長であることがわかりました（下図参照）。

■ 暗号化のロジック図(2) AES暗号鍵の作成方法

▼各ファイルを暗号化するために使用するAES暗号鍵は「BcryptGenRandom」関数を使用して作成する

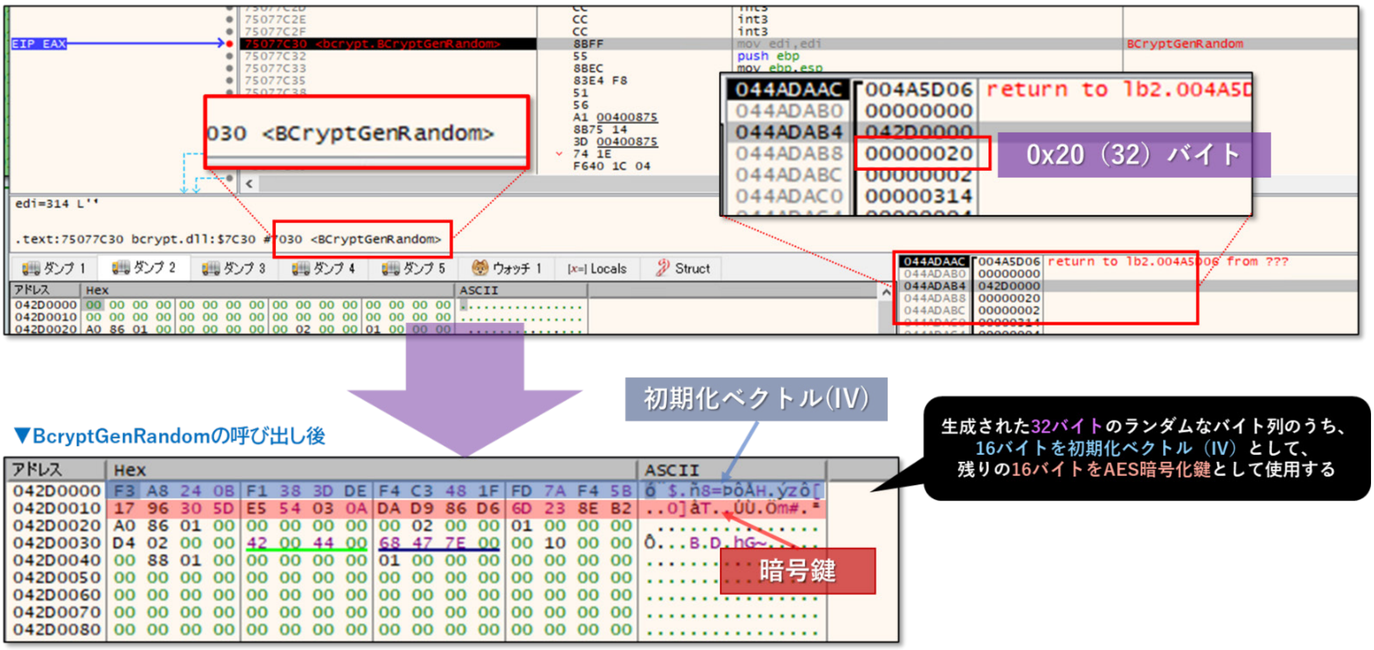


図 45　AES 暗号鍵の作成方法

さらにファイルの暗号化に使用する AES のアルゴリズムにおいて、10 個のラウンド定数による 10 回のラウンド処理が確認でき、これも鍵長が 128 ビット（128 ビットはラウンド 10 回）であることを意味します。また、前回暗号化した列との排他的論理和の処理が確認されるため、この特徴は CBC モードを意味します（後述）。暗号化するデータを 16 バイトごとに暗号化（異なる暗号化鍵で 10 回処理）し、それを 256 回繰り返します。この結果、16*256=4096 バイト暗号化することとなり、前述したとおりファイルの先頭を 4KB しか暗号化しないことを意味します。
初期化ベクトル（IV）としての利用部分と上記のラウンド処理が確認できたことから、暗号アルゴリズムは「AES128 ビット」「CBC モード」であると断定できます（下図参照）。

■ 暗号化のロジック図(3) ファイルの暗号化に使用するAESのアルゴリズム

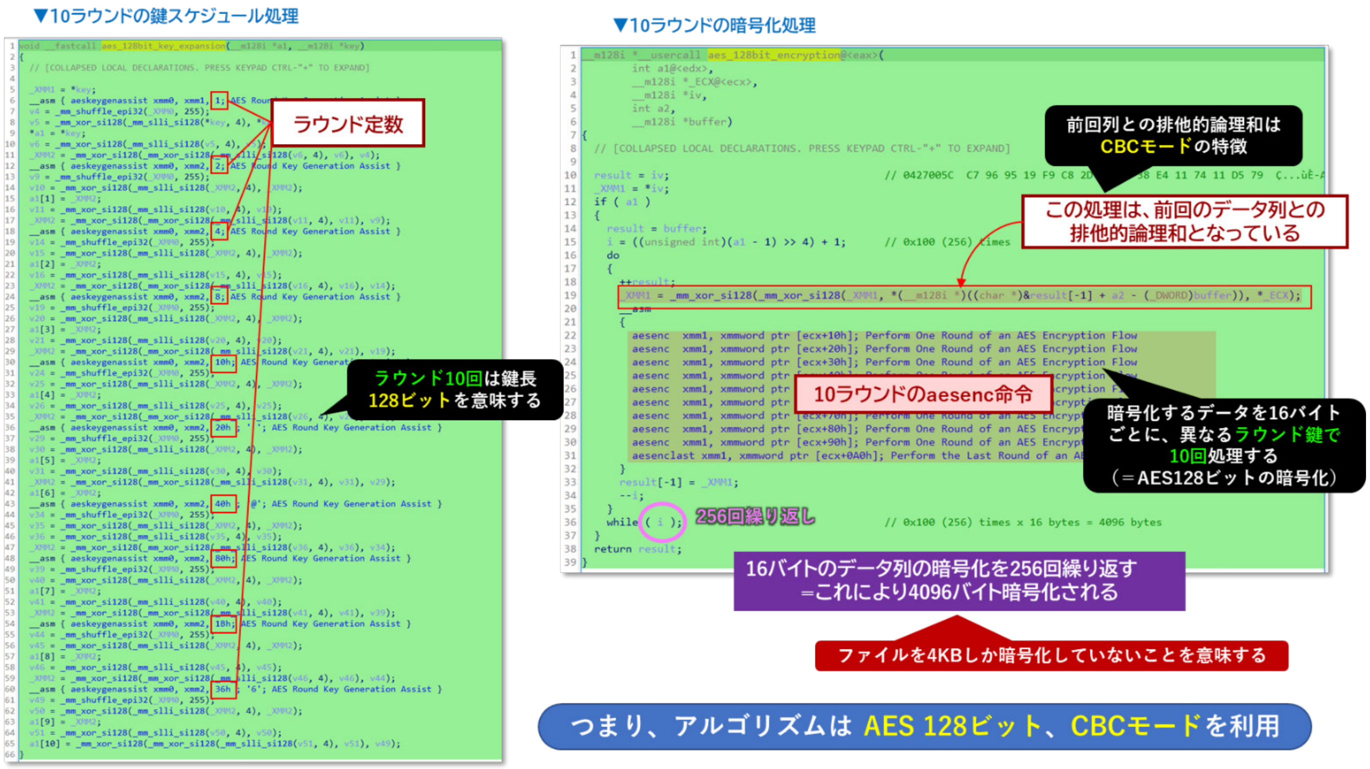


図 46　ファイルの暗号化に使用する AES のアルゴリズム

なお、上記で「前回暗号化した列との排他的論理和の処理」が AES 暗号における CBC モードの特徴であることに触れましたが、CBC モードの具体的な処理の流れは以下のようになります。暗号化するデータの最初の 16 バイトは、IV(初期化ベクトル)と暗号鍵が使用されますが、2 ブロック目以降の 16 バイトは、1 つ目の暗号化ブロックを IV(初期化ベクトル)の代わりに使用します（下図参照）。

このように前回暗号化した列を入力値とする排他的論理和の処理があれば、CBC モードと判別できる一つの特徴であるといえます。

■ AES暗号/CBCモードの特徴

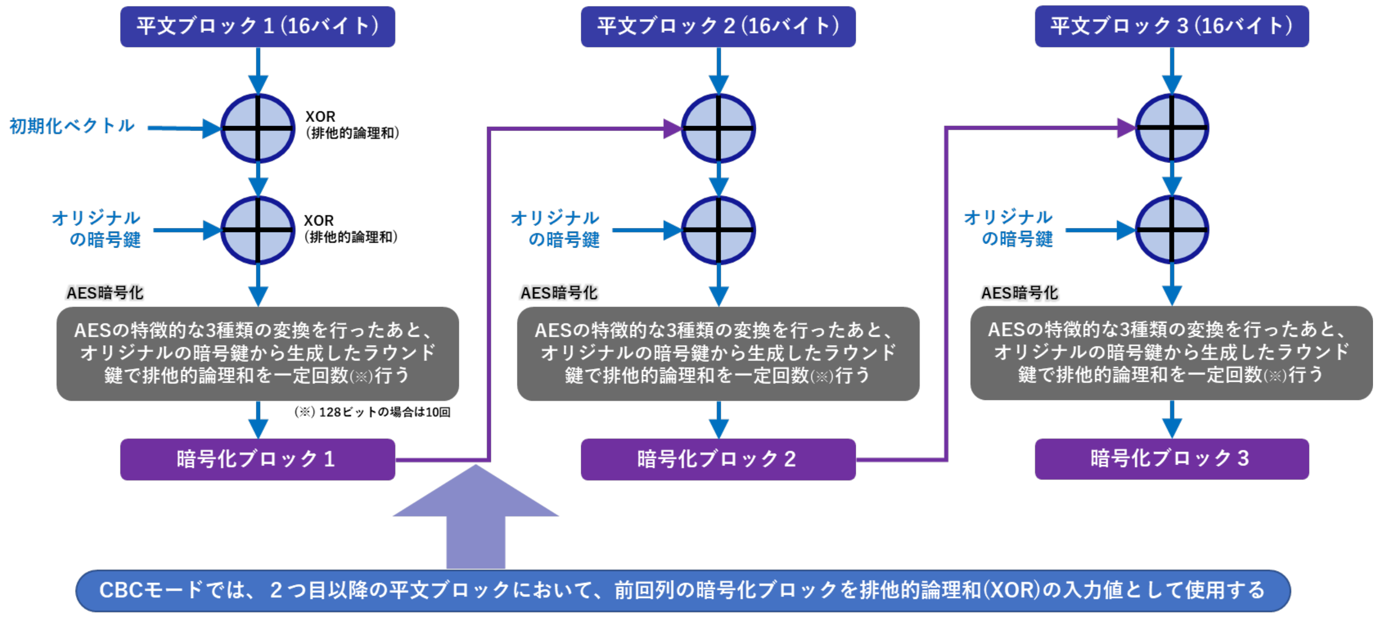


図 47　AES 暗号／CBC モードの仕組みと特徴

以上の暗号化処理により、LockBit2.0 が暗号化したファイルのフッタ構造は以下の図のようになります。512 バイトあるフッタ全体のうち、後方に「セッション ECC 公開鍵」で暗号化した「AES 鍵（暗号鍵＆IV）」と、「マスターECC 公開鍵」で暗号化した「セッション ECC 秘密鍵」を追記します（下図参照）。

■ LockBit2.0に暗号化されたファイルのフッタの全体構造

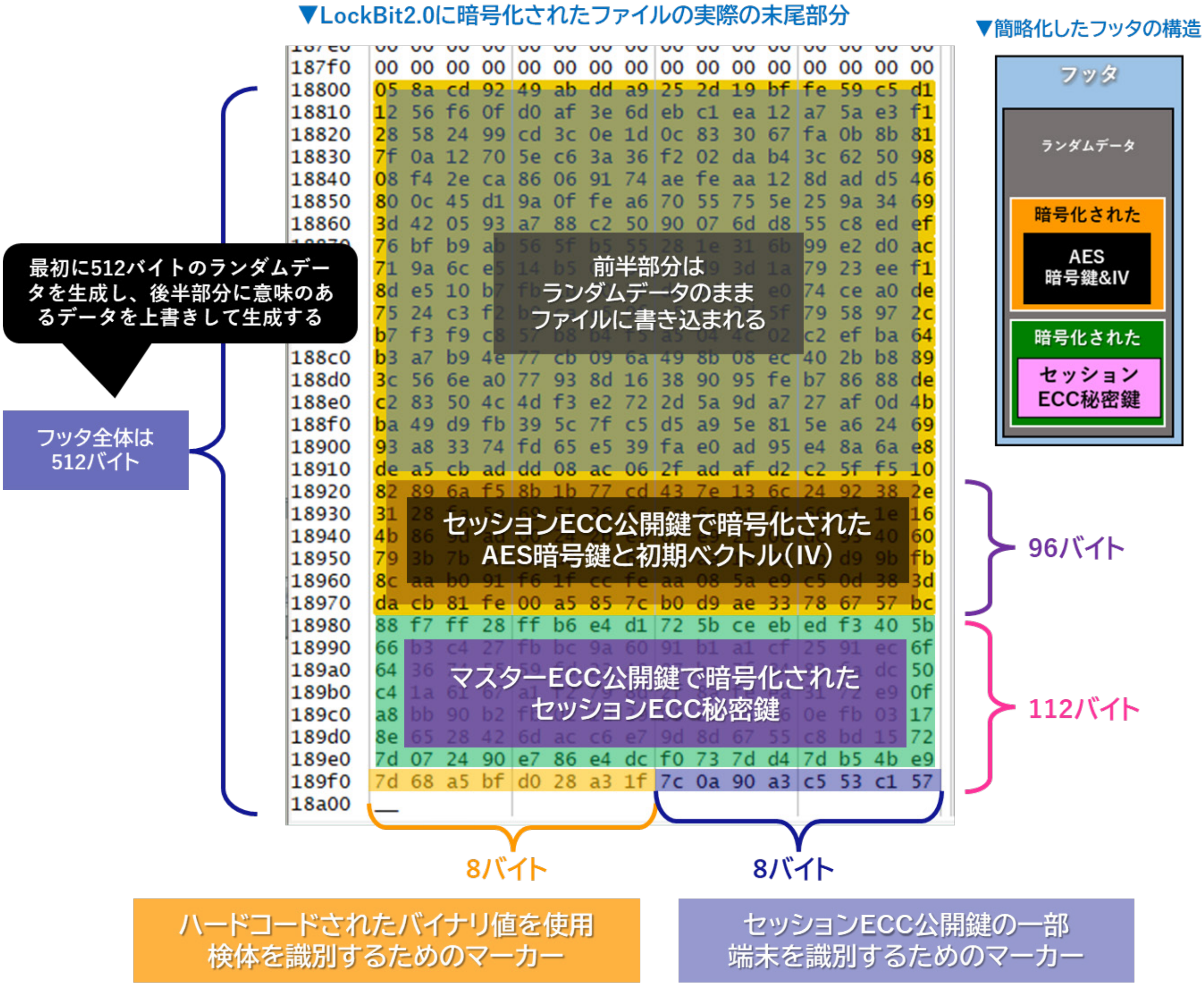


図 48　LockBit2．0 に暗号化されたファイルのフッタの全体構造

上記図にも記載した通り、フッタの末尾にはランサムウェアの「検体」を識別するための 8 バイトのマーカーが記載されます。この 8 バイトの値は LockBit2.0 の本体（EXE ファイル）のバイナリ部分にハードコードされており、検体に依存する値であるため、検体識別用に使用可能なデータであるといえます。使用した検体が特定できれば、攻撃者側で使用されているマスター鍵の識別が可能であると想定されます（下図参照）。

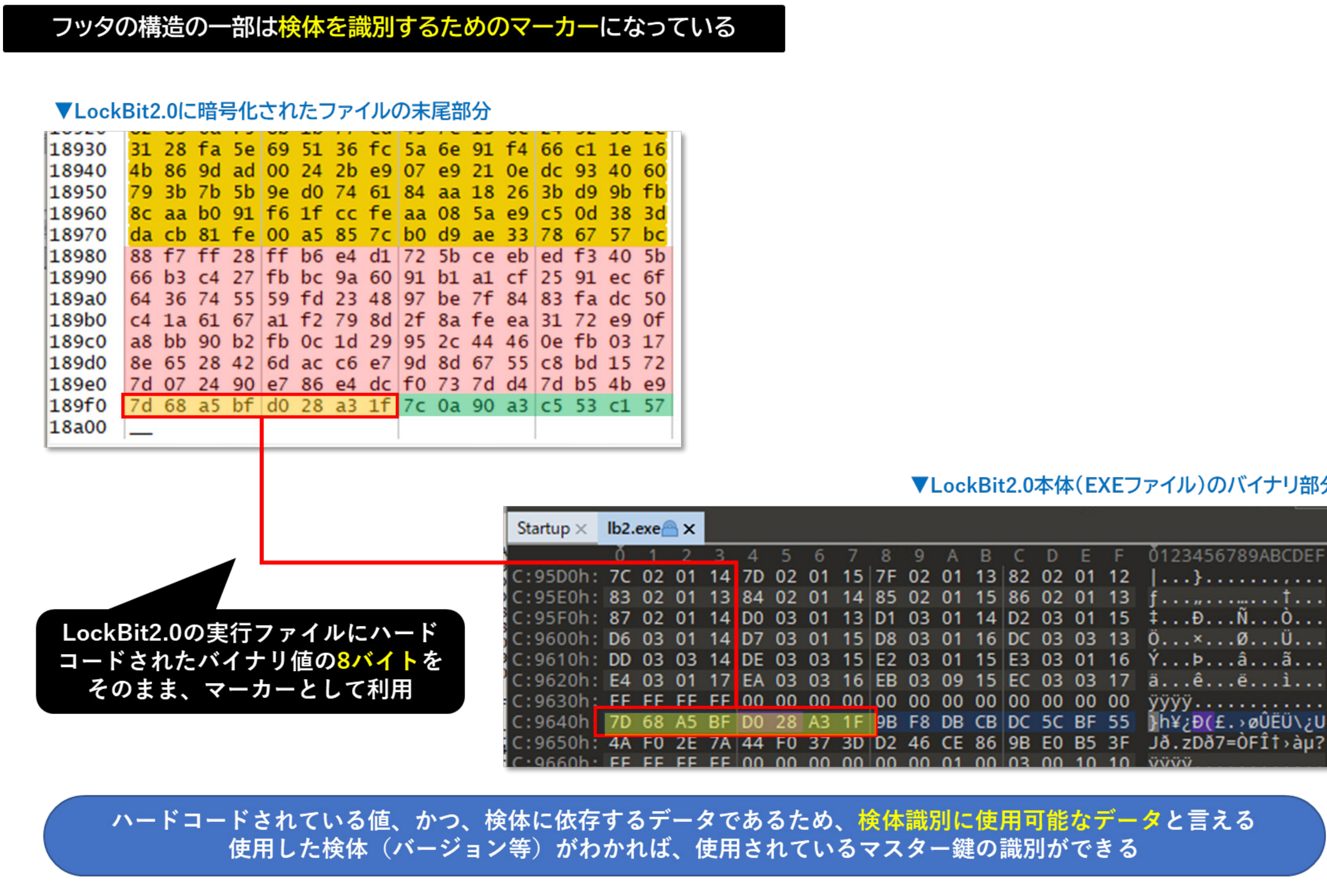


図 49　フッタの末尾に含まれる「検体」識別用マーカー

そしてもう一つ、フッタの末尾には「端末」を識別するための 8 バイトのマーカーが記載されます。この 8 バイトの値は、LockBit2.0 が作成したレジストリの「Public」に格納されたセッション ECC 公開鍵の先頭 8 バイトとなっており、セッション ECC 公開鍵は LockBit2.0 が最初に実行された際に生成されるものであるため、攻撃者側で端末を識別する際の目安として使用できるデータであるといえます（下図参照）。

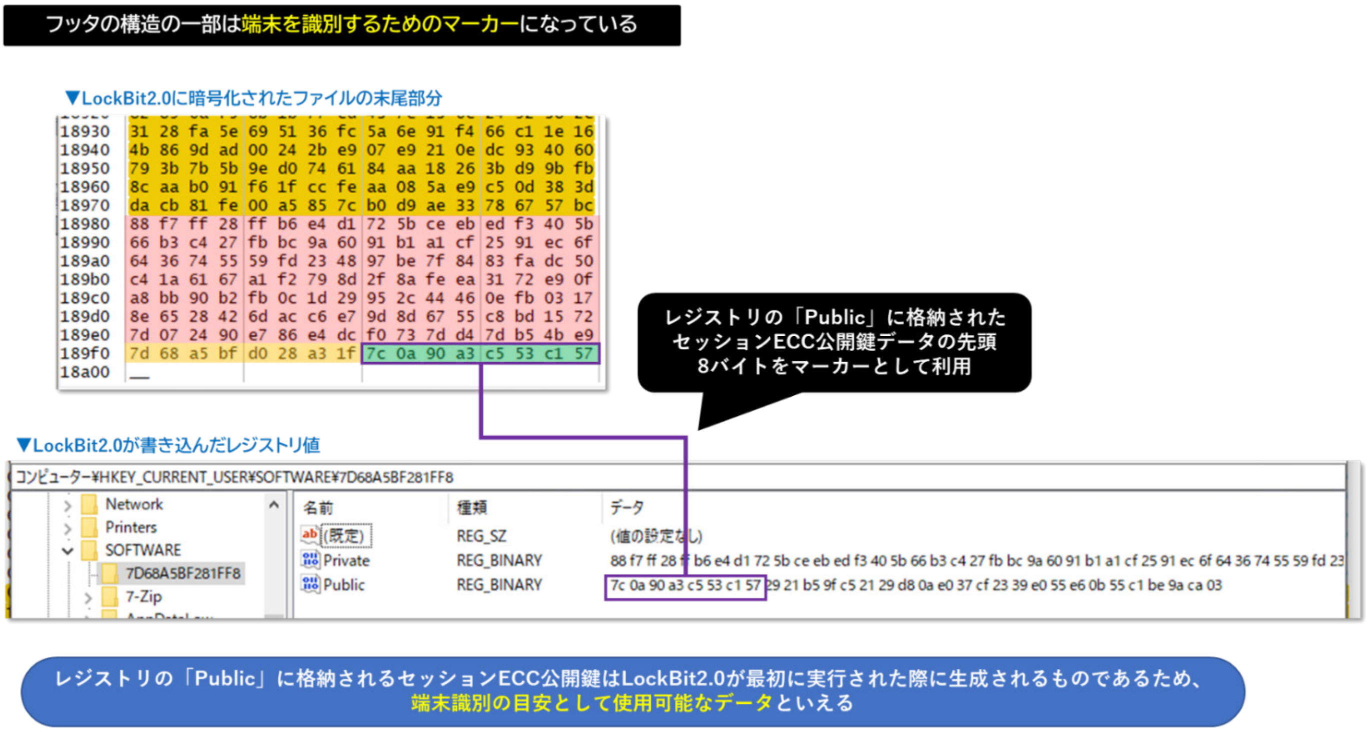


図 50　フッタの末尾に含まれる「端末」識別用マーカー

なお、上記レジストリキーの命名規則は、LockBit2.0 の実行ファイルにハードコードされたバイナリ値から特定の場所をピンポイントで抽出したものが使用されています（下図参照）。

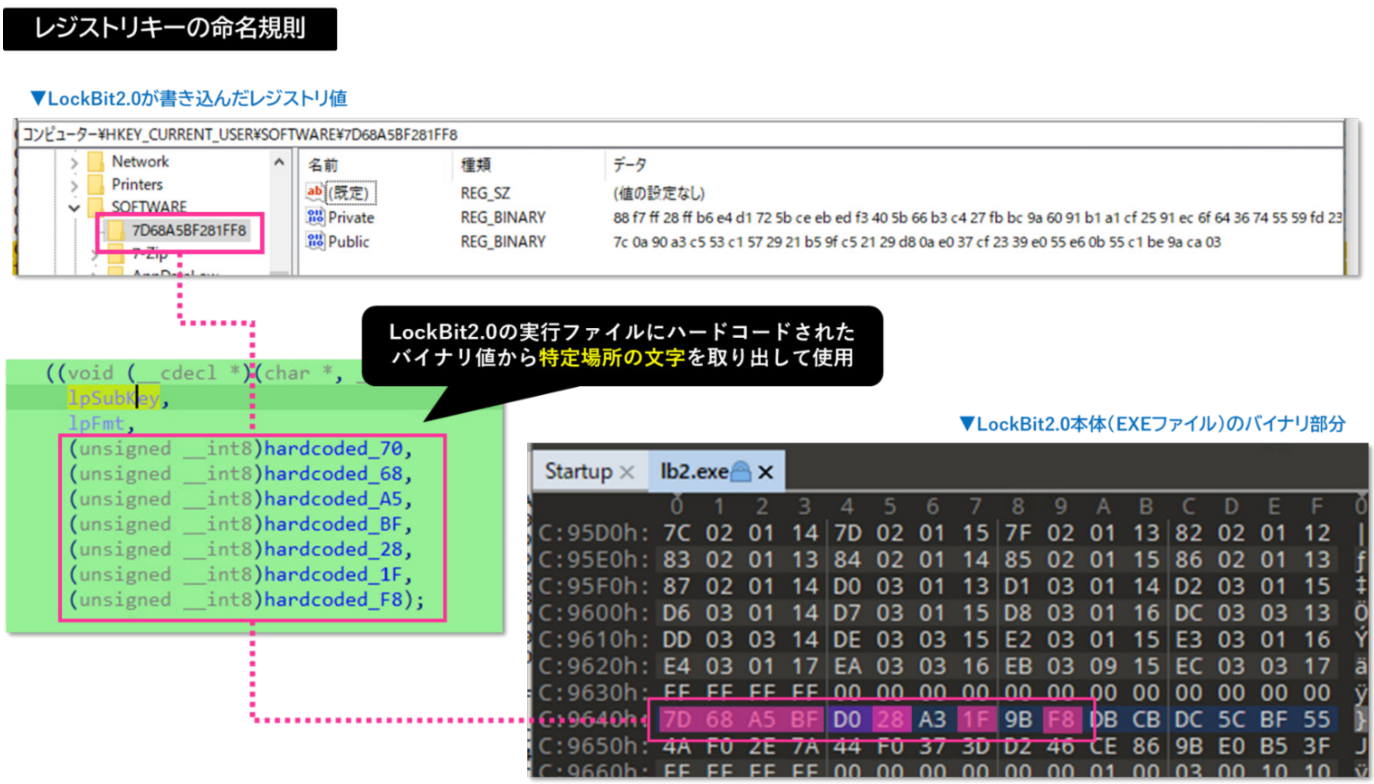


図 51　レジストリキーの命名規則

LockBit2.0 はこうしたファイルの暗号化を、GetDriveTypeW 関数で書き込み可能と判別した全てのドライブ配下のファイルに対して実施していきます（下図参照）。

図 52　書き込み可能な全ドライブのチェック

なお、LockBit2.0 はネットワーク共有に設定されたリソース先もアクセスし暗号化する能力を持っています。WNetOpenEnumW 関数および WNetEnumResourceW 関数を使用して感染端末からアクセス可能な全てのネットワーク共有を列挙し、発見されたネットワーク共有ごとに新しいスレッドを作成しリソース先に存在する全てのファイルに対し暗号化を行っていきます（下図参照）。また、WNetAddConnection2W 関数を利用したネットワーク共有へのアクセスについても試みます。

▼LockBit2.0によるネットワークリソースの暗号化処理

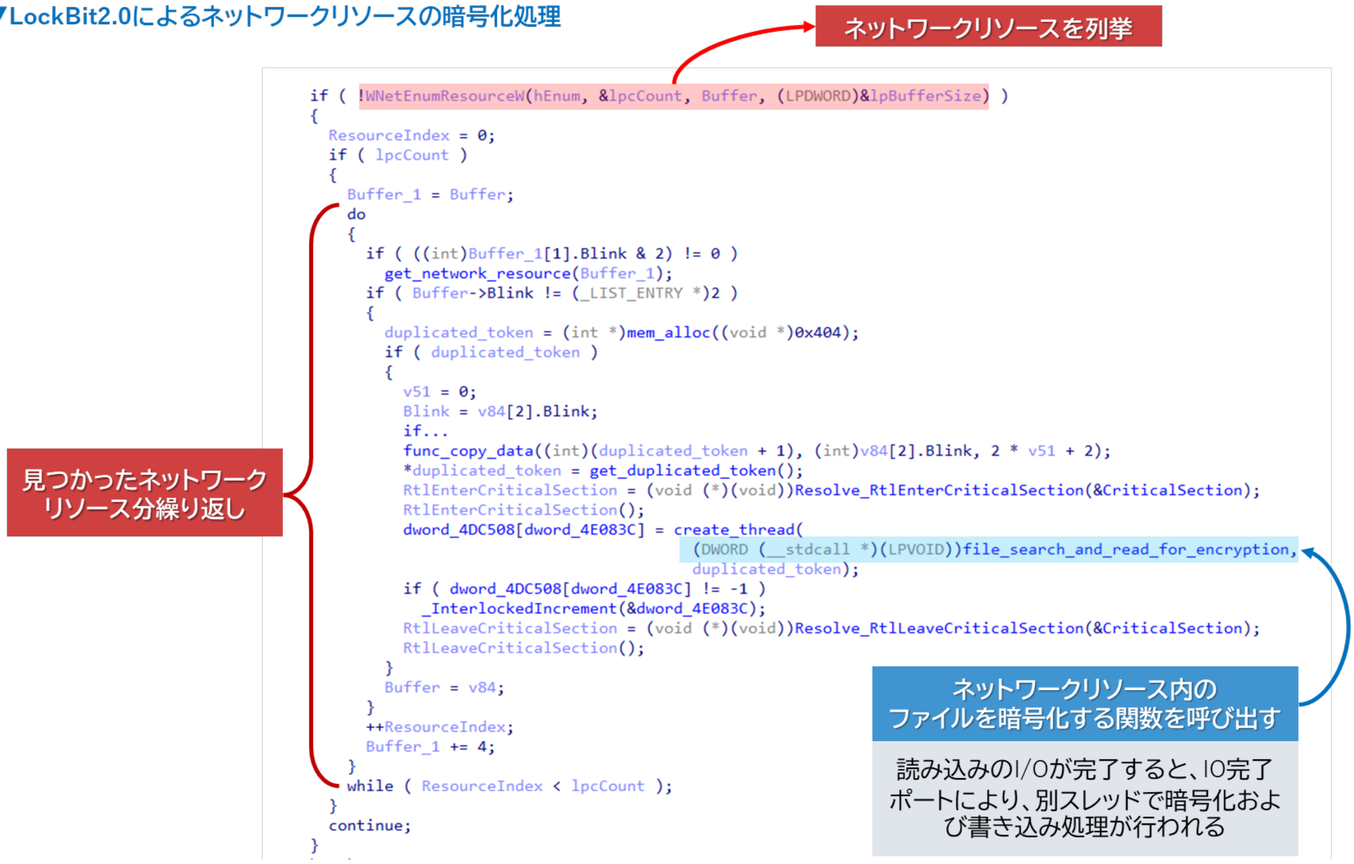


図 53　ネットワークリソースの暗号化処理

■書き込みが禁止されているファイルの属性変更

LockBit2.0 はできるだけ多くのファイルを暗号化するために、「読み取り専用」属性がついているファイル（つまり書き込みが禁止されているファイル）を発見した場合、該当の属性を削除します。

■ 「読み取り専用」属性が付いているファイルは該当の属性が削除される

図 54　読み取り専用属性が削除される様子

具体的には、個々のファイルを暗号化する際 GetFileAttributes 関数でファイルの属性をチェックし「読み取り専用」属性が設定されているファイルであると判断すると、SetFileAttributes 関数に「0x80=FILE_ATTRIBUTE_NORMAL」の引数を渡し、属性が設定されていない状態に変更します。

これによりたとえ「読み取り専用」属性が付与され書き込みが禁止されているファイルであっても、書き込み可能な状態にしたうえで暗号化を行うことが可能となっています。

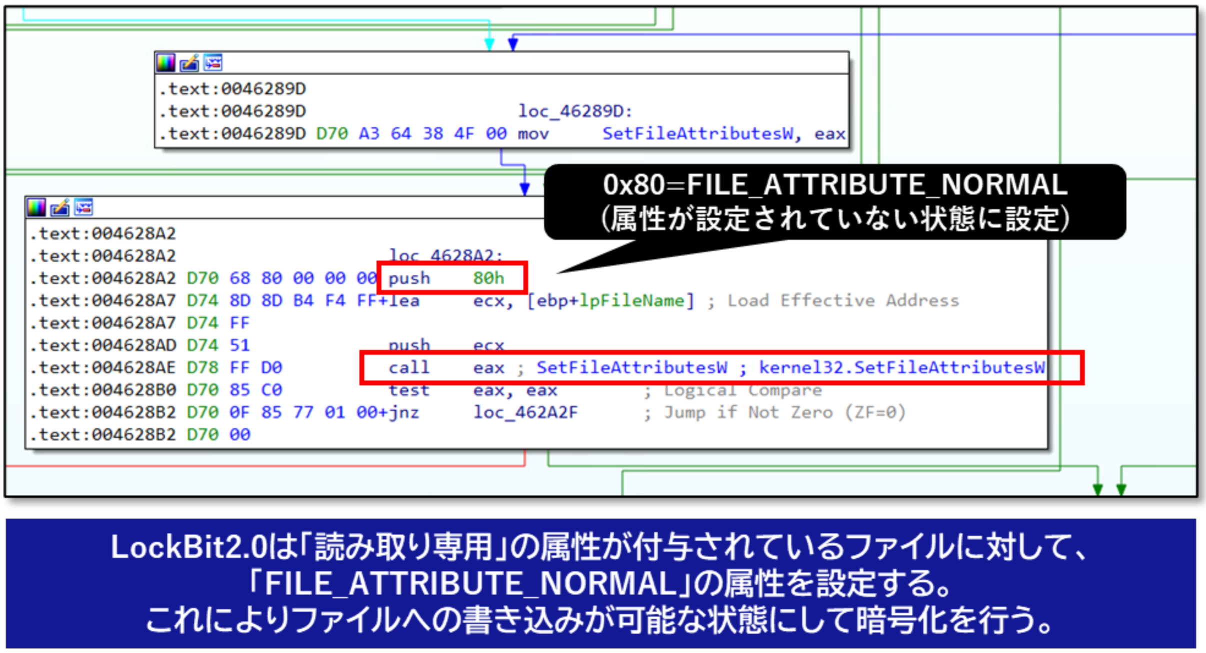


図 55　SetFileAttributes 関数による属性変更の処理

■ファイルの暗号化から除外する対象

ファイルを暗号化する際、以下に挙げる「ファイル名」「フォルダ名」「拡張子」を暗号化対象から除外します。これらにはシステムが動作するために最低限必要なファイルやフォルダ、および攻撃者とコミュニケーションするためのブラウザなどが含まれています（下図参照）。

■暗号化の対象から除外するファイル名

- restore-my-files.txt
- ntldr
- ntuser.dat.log
- bootsect.bak
- autorun.inf
- thumbs.db
- iconcache.db

■暗号化の対象から除外する拡張子

.386	.rtp	.exe	.ttf	.wad
.cmd	.rdp	.dll	.otf	.msu
.ani	.bin	.lnk	.fon	.icns
.adv	.hlp	.ico	.fnt	.lock
.msi	.shs	.hlp	.dmp	.lockbit
.msp	.drv	.sys	.tmp	.theme
.com	.wpx	.drv	.pif	.diagcfg
.nls	.bat	.cur	.wav	.diagcab
.ocx	.rom	.idx	.wma	.diagpkg
.mpa	.msc	.ini	.dmg	.msstyles
.cpl	.spl	.reg	.iso	.gadget
.mod	.msu	.mp3	.app	.woff
.hta	.ics	.mp4	.ipa	.part
.prf	.key	.apk	.xex	.sfcache

■暗号化の対象から除外するフォルダ名

- intel
- msocache
- $recycle.bin
- $windows.~ws
- tor browser
- boot
- system volume information
- perflogs
- google
- application data
- windows
- windows.old
- appdata
- windows nt
- msbuild
- microsoft
- all users
- mozilla
- microsoft.net
- microsoft shared
- internet explorer
- common files
- opera
- windows journal
- $windows.~bt

図 56　LockBit2.0 がファイルの暗号化から除外する対象

■LockBit2.0 に暗号化されたファイルのアイコン

LockBit2.0 は、以下のレジストリを改変することで「.lockbit」という拡張子（暗号化したファイルにつける拡張子）を持つファイルのデフォルトアイコンを、自身が作成した LockBit2.0 のデザインのアイコンに設定します。

■ 「.lockbit」という拡張子を持つファイルのデフォルトアイコンを登録

HKEY_LOCAL_MACHINE¥Software¥Classes¥.lockbit¥DefaultIcon

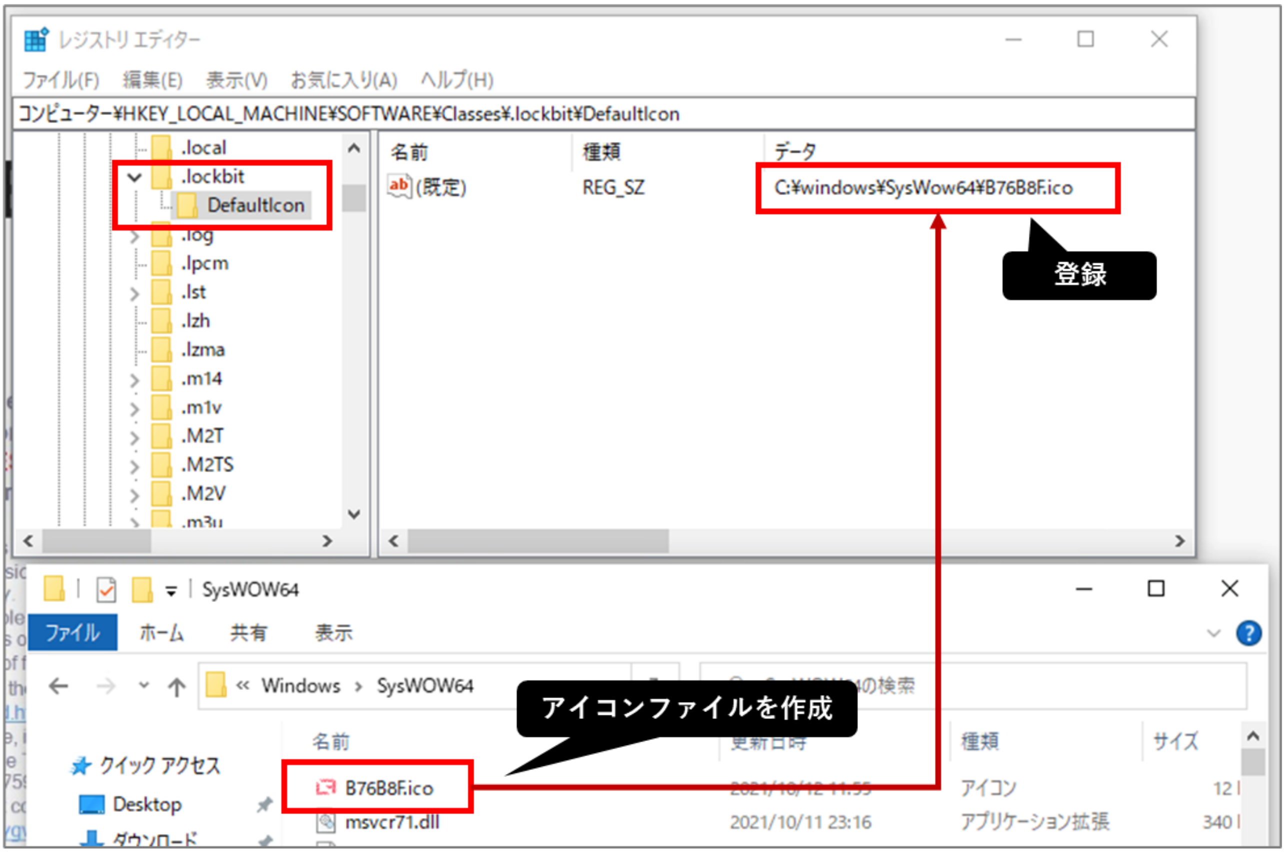


図 57　アイコンファイルの作成と設定

そして、設定したアイコンをすぐにシステムに適用させるため、SHChangeNotify 関数を呼び、システムのアイコン描画をリフレッシュさせます。

■ 「SHChangeNotify」で描画をリフレッシュし、設定したアイコンをシステムに即時反映

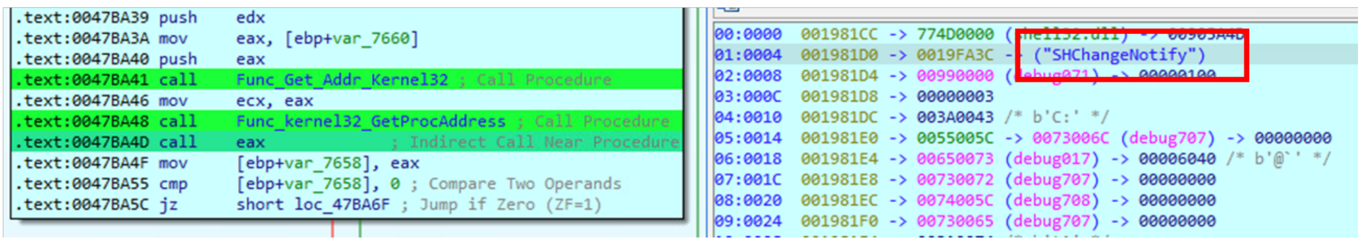


図 58　登録したアイコンのシステムへの即時反映の処理

その結果、LockBit2.0 に暗号化されたファイルは全てアイコンが以下のように変化します。

■ LockBit2.0に暗号化されたファイルの一覧

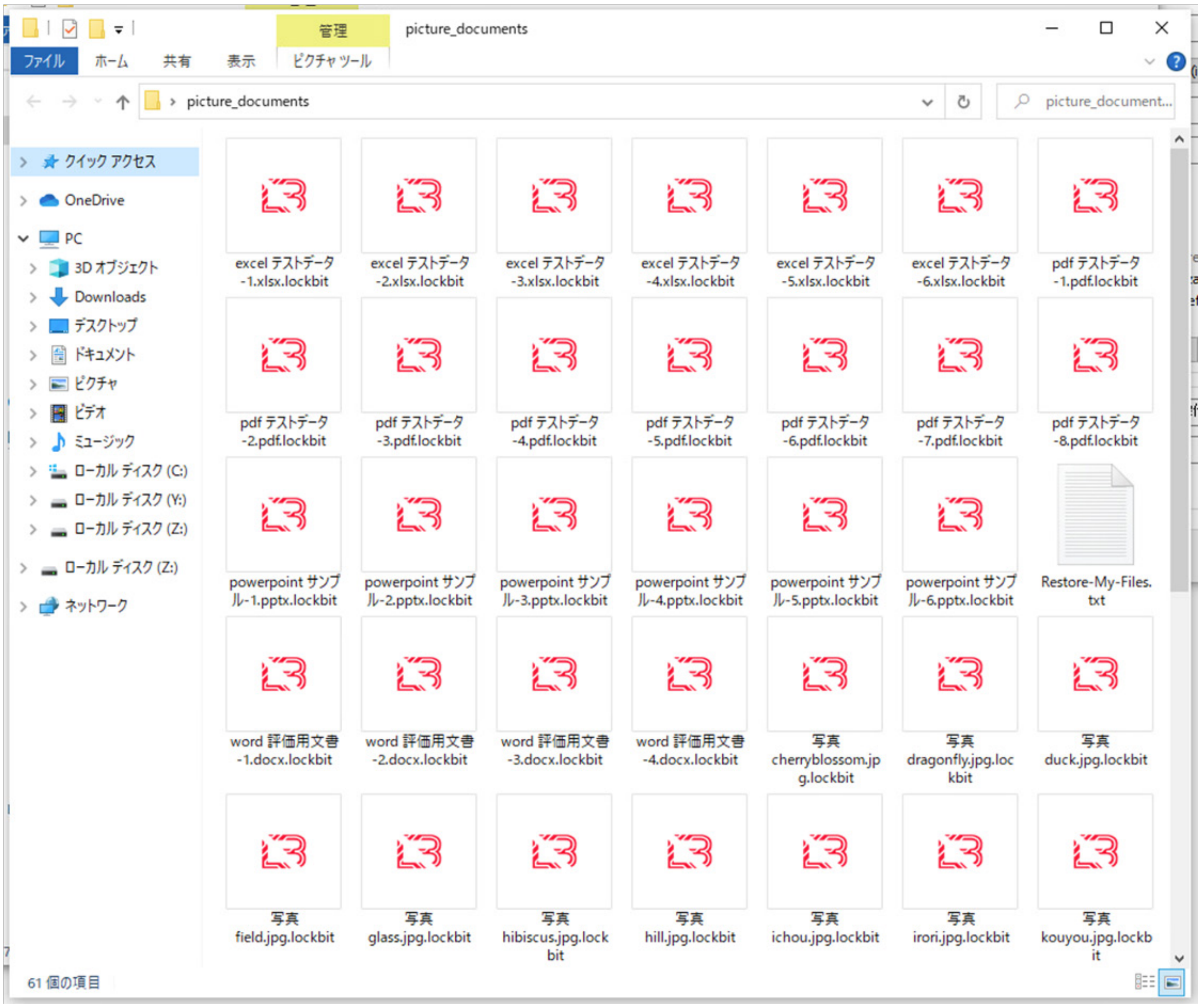


図 59　LockBit2.0 に暗号化されアイコンが変化したファイルの一覧

ただし、一部の検体において、アイコンが変化しないケースが存在することが複数の検体を解析することで明らかとなりました。そのため、LockBit2.0 の検体によっては、感染したアイコンが変化しない場合も存在します。

■LockBit2.0 のバグ（暗号化が途中で失敗するミス）

LockBit2.0 には、環境によってファイルの暗号化処理が途中で失敗してしまうバグを持つ検体が存在することが今回の調査の結果新たに判明しました。これは本検証時において最新の Windows Update を適用した後の Windows 10 で確認されており、LockBit2.0 がシステムに関わる特定の特殊フォルダへのアクセス権がないことで、該当のシステムフォルダへアクセスした時点でファイルの暗号化スレッドがクラッシュし無限ループに陥るか、LockBit2.0 のプロセス自体がクラッシュして終了します。

このシチュエーションに該当する環境では、ファイルが途中まで暗号化されているものの、LockBit2.0 の後半の挙動で行う処理である「デスクトップの壁紙変更」、「デスクトップ上の脅迫文(HTA)の作成」などが発生しません。

そのため、実際の被害組織の中にはデスクトップ壁紙が LockBit2.0 を示すわかりやすい画像に変更されず、また脅迫文も目が届くデスクトップなどに作成されないことから、LockBit2.0 という種類のランサムウェアに感染したこと自体を把握できていないケースがあるものと推測されます。

この場合、C ドライブ直下のアルファベット順で比較的前方にあるフォルダ内を確認し、ファイルの拡張子が「.lockbit」のものが存在するかどうかで、LockBit2.0 に感染し一部のファイルが暗号化されたかどうかを判別することが可能です。

■LockBit2.0 のバグ（多重暗号化を発生させるミス）

LockBit2.0 には、シチュエーションによってファイルを多重に暗号化してしまうバグを持つ検体が一部存在します。

LockBit2.0 は、暗号化済みのファイルかどうかをチェックする際、ファイル内部のフッタなどは確認せず単純に「.lockbit」という拡張子の有無でのみ判別しますが、ネットワーク共有フォルダなどネットワーク経由で暗号化を行う場合に限り、暗号化したファイルの拡張子を「.lockbit」に変更しないというバグを抱える検体が存在します。

そのため、感染端末が存在するローカルネットワーク上には、拡張子が「.lockbit」でないにも関わらず暗号化されたファイルが存在する可能性があり、状況によっては多重に暗号化される可能性があります（下図参照）。

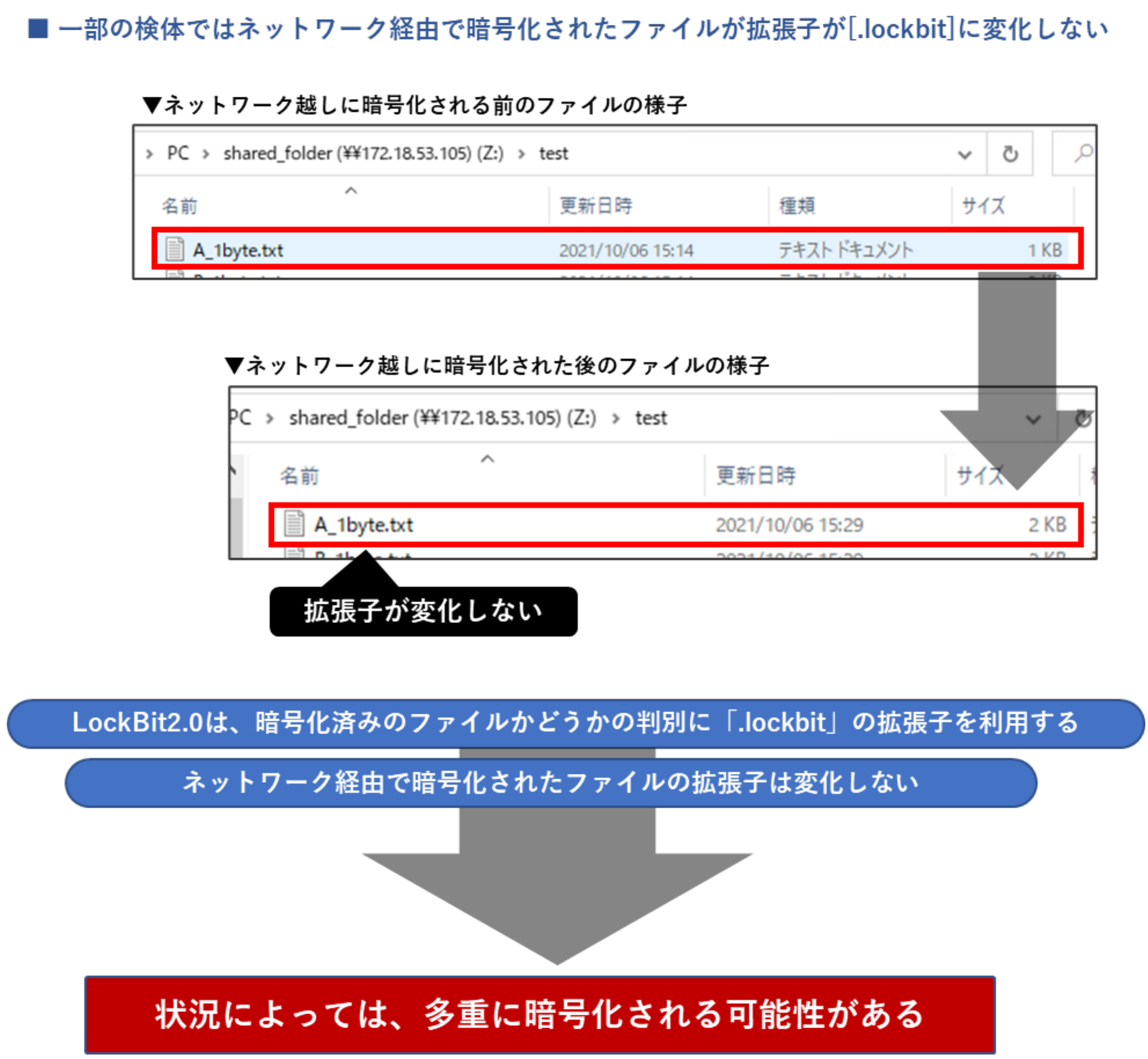


図 60　ネットワーク越しで暗号化されたファイルの拡張子は変化しない

多重に暗号化されることを確かめるため、実際に次の検証を行いました。

以下はネットワーク共有したフォルダを持つ端末α上で LockBit2.0 に感染させた際の検証結果を図示したものです。端末α上の共有フォルダには「ファイル B」が配置しており、「ファイル B」はすでに他の端末β上の”バグを持つ検体”により暗号化されている状況となります。上記で述べた通り、バグを持つ検体によりネットワーク経由で暗号化された「ファイル B」は、暗号化されているものの拡張子が「.lockbit」ではない状態になっています。

この状況下で、端末αを LockBit2.0 に感染させたところ、「ファイル B」は想定通り 2 重に暗号化されることが確かめられました（下図参照）。

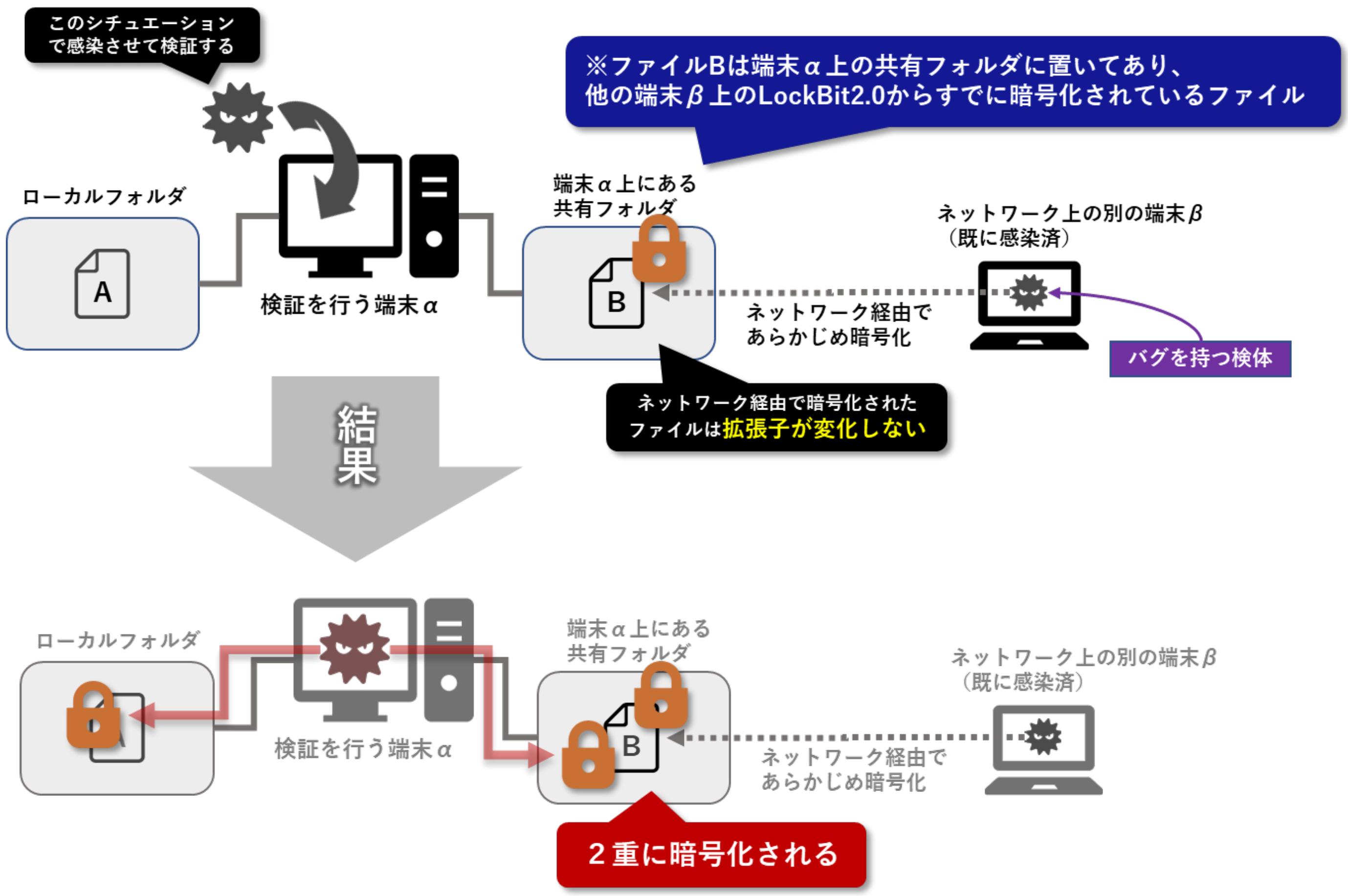


図 61　バグを持つ検体の動作

上記の検証結果を踏まえると、例えば以下のように、多数の感染端末からアクセス可能な共有フォルダが存在する状況下でバグを持つ LockBit2.0 に感染した場合、3 重 4 重と感染端末の数だけ多重に暗号化されてしまう可能性があるといえます（下図参照）。

■ 状況によっては以下のように一つのファイルが３重４重と多重に暗号化されてしまう可能性がある

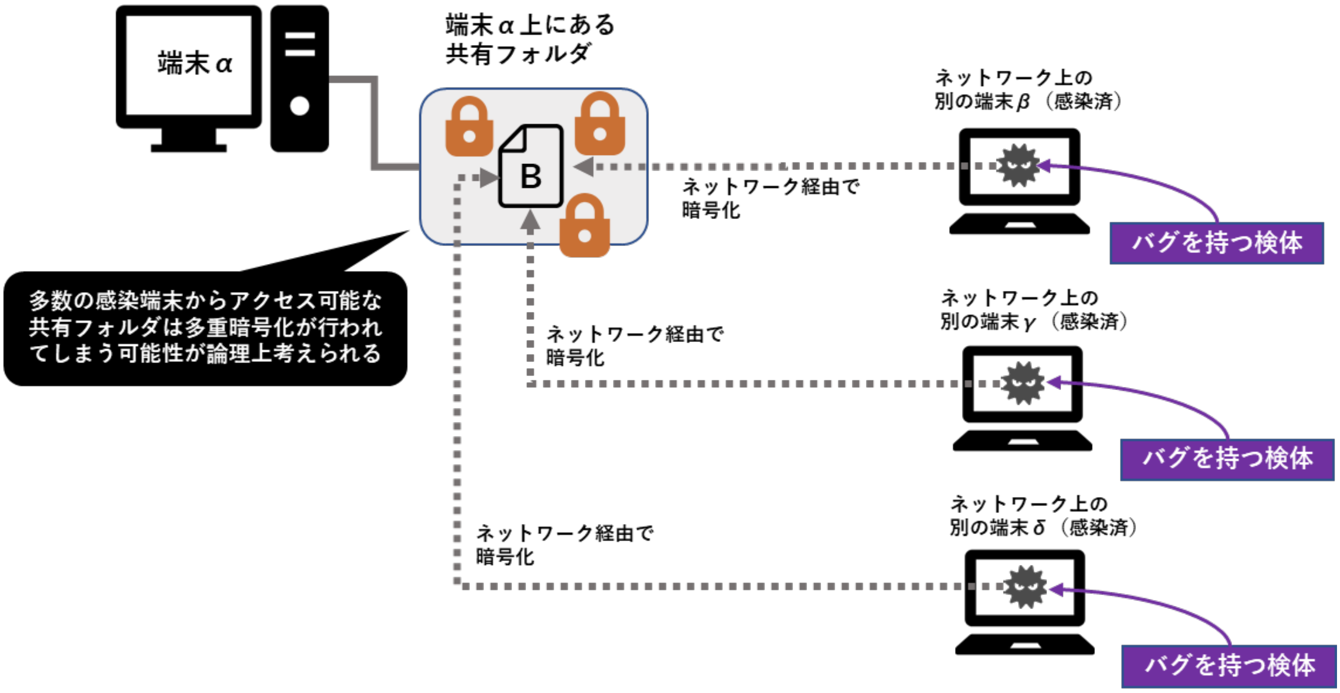


図 62　LockBit2.0 に多重暗号化される可能性とシチュエーション

このバグを抱える LockBit2.0 の個体に感染し、ファイルが多重に暗号化されてしまっていた場合、たとえ攻撃者から復号ツールを提供され復号しようとしても、実際は復号できないという状況が発生すると想定されます（復号ツールも拡張子で判別すると考えられるため）。
もちろん理論上、ファイルフッタの構造を都度判別し多重に暗号化されていることを想定して元のファイルになるまで繰り返し復号するツールを作れないわけではないですが、そこまで煩雑な復号ツールを作るのであれば、はじめから多重暗号化が発生しないようにランサムウェア側を実装したほうが現実的といえます。

つまり、ファイルが多重に暗号化されてしまうシチュエーションがあるというこの事実は、LockBit2.0 の攻撃者が想定しているとは考えられず、ファイルの復元が困難となる致命的なバグの一つであると考えられます。

ただし、比較的新しくビルドされた LockBit2.0 において、このバグが修正されている個体も存在することをその後確認しました。以下の図の通り、バグが修正された個体ではネットワーク経由で暗号化した場合も正常にファイルの拡張子を「.lockbit」に変更するように修正されているため、多重に暗号化されるという上記問題が解消されています。

■ 比較的新しいLockBit2.0の検体では、バグが修正されている個体があることがわかった

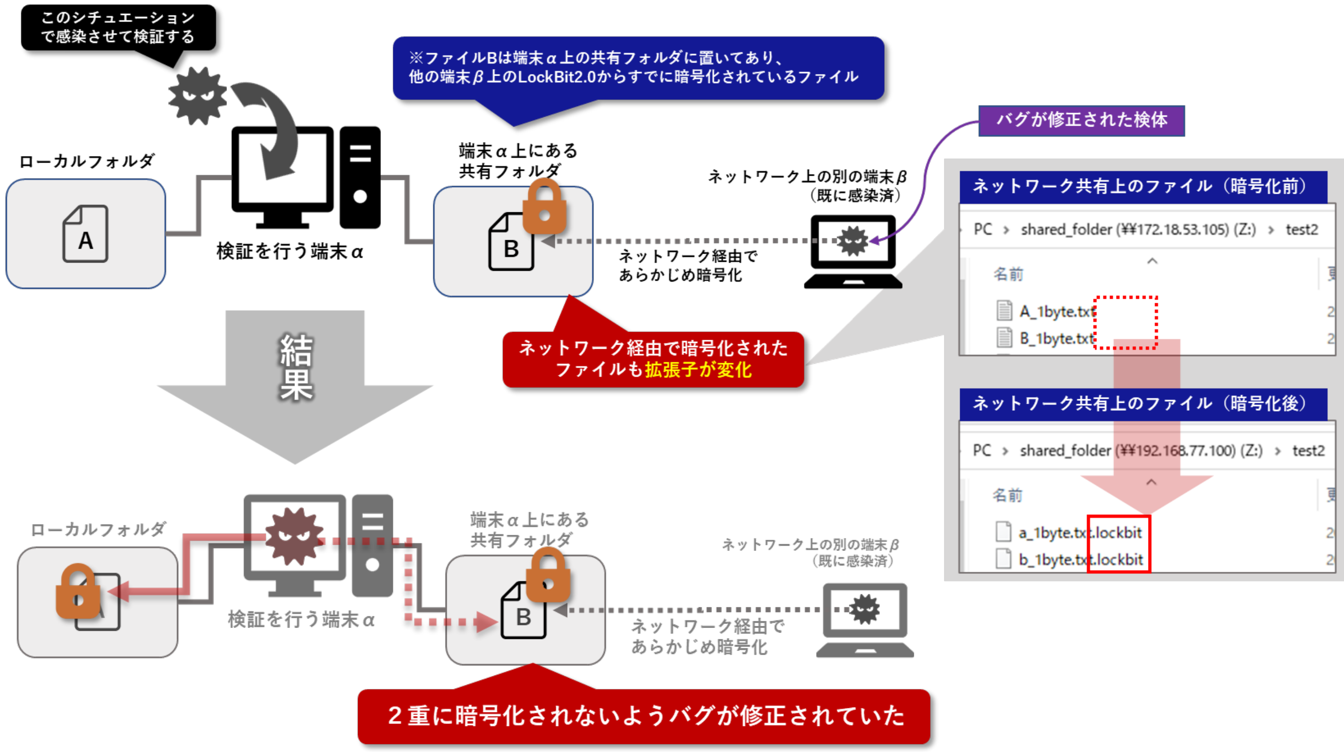


図 63　バグが修正された検体の動作

バグを持つ個体とバグが修正された個体はビルド日時が比較的近いため、現時点で LockBit2.0 に攻撃されている被害組織の中には、多重暗号化により復元を困難とさせてしまうバグを持つ個体が実際に使用されているケースも少なくないものと想定され、注意が必要です。

■LockBit2.0 のバグ（「.lock」ファイルのチェックミス）

LockBit2.0 は多重暗号化を防ぐため、プロセスの起動中はドライブ直下に「.lock」というファイルを作成し常に開いた状態にします。「.lock」のファイル名部分の文字列は「セッション ECC 鍵」のペアを記録したレジストリキー名の最初の４バイトが使用されます。
LockBit2.0 は起動時に該当の「.lock」ファイルが CreateFile で開けるか（ハンドルがとれるか）をチェックし、直前のエラーコードを「0xB7」(ERROR_ALREADY_EXISTS)と比較、該当した場合は自身を終了させます。

もしエラーコードが「0xB7」に該当しなかった場合は、処理を継続し「.lock」ファイルのハンドルを動作中は常に開いたままの状態にします。

ローカルにおける多重感染防止は前述のとおりミューテックスで実現されていることから、この「.lock」ファイルの役割は該当ドライブがネットワーク越しで暗号化される場合を想定した多重暗号化防止の役割を持つことが想定されます。つまり、「.lock ファイルのハンドルが開けない」＝「すでに該当の端末（ドライブ）では他の LockBit2.0 が動作し暗号化している最中である」と判断し、ファイルが多重に暗号化されることを防ぎたいのだろうと考えられます（下図参照）。

図 64　「.lock」ファイルのハンドルオープン処理

しかし、今回の解析の結果、ここにもバグが存在することが発見されました。

LockBit2.0 は直前のエラーコードを「0xB7」(ERROR_ALREADY_EXISTS)と比較していますが、「.lock」ファイルのハンドルが開かれた状態で感染させると、実際には OS から「0x50」(ERROR_FILE_EXISTS)というエラーコードが応答されます。そのため、実検証では「.lock」ファイルのハンドルが開かれた状態であっても処理は止まらず暗号化は継続されてしまいました。

また、一部の LockBit2.0 には前述した通り、ネットワーク越しで暗号化したファイルの拡張子を「.lockbit」に変更しないというバグも存在します。
つまり、これらのバグが組み合わさることで、他の端末上の LockBit2.0 からネットワーク越しで並行して暗号化が行われた場合、結果的に排他制御ができず、共有フォルダのみでなく共有化された各端末のドライブにおいても多重の暗号化が発生してしまう可能性があることを意味します。

この結果はコードの実装を見る限り開発者にとって想定外である可能性が高く、「多重暗号化を防ぎたい」という本来の意図を推し量ると LockBit2.0 の開発者はこの比較対象のエラーコードを「0xB7」ではなく「0x50」という値にしたかった可能性があります（下図参照）。

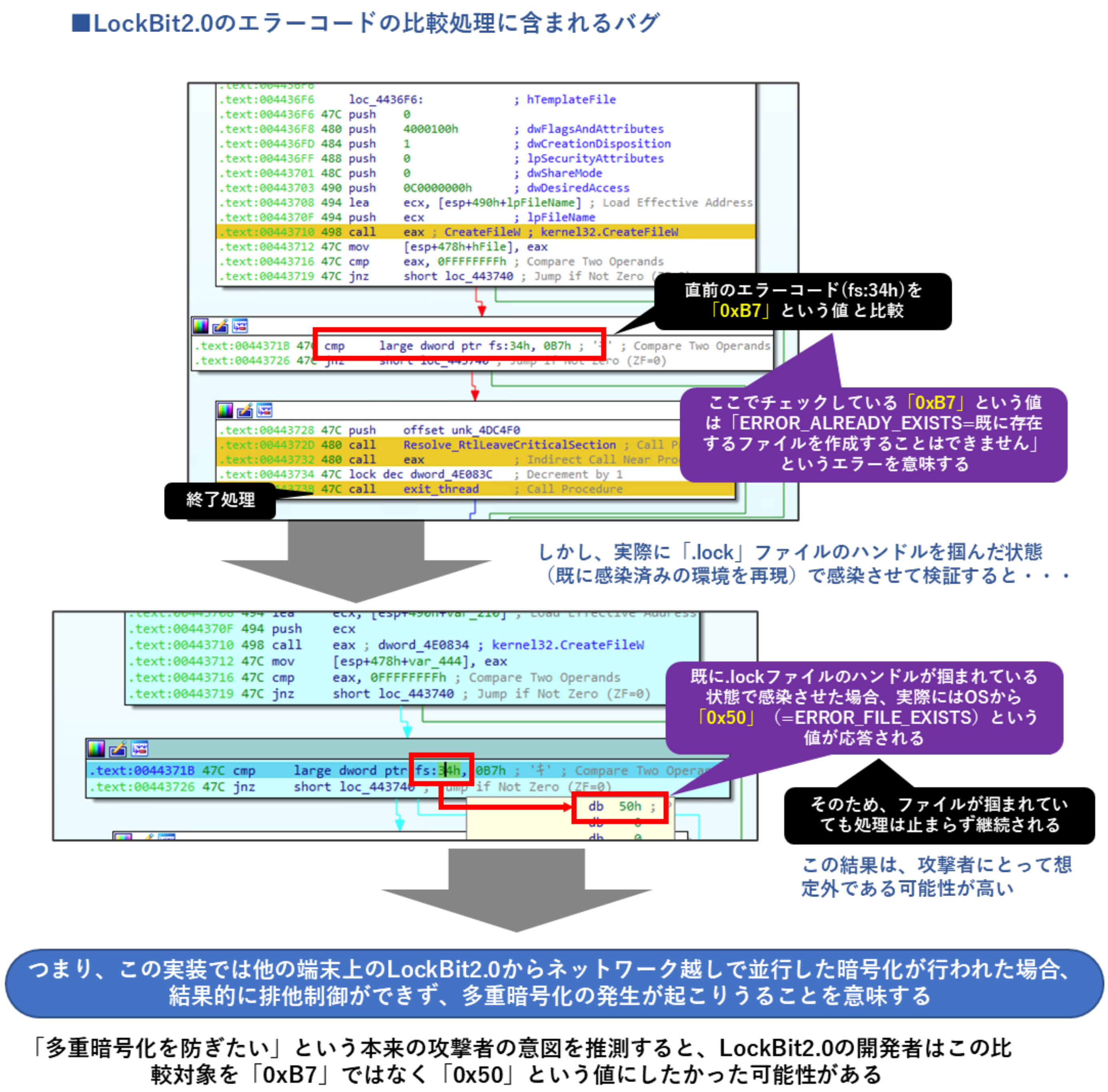


図 65　LockBit2.0 のエラーコードの比較処理に含まれるバグ

■ドメインコントローラでの動作

LockBit2.0 にはドメインコントローラで行う特別な挙動が存在します。以降ではその挙動を解説します。

LockBit2.0 は感染時、GetComputerName 関数を使用してコンピュータ名を取得したのち、NetGetDCName 関数によりプライマリドメインコントローラの名前を取得できるか確かめます。もしプライマリドメインコントローラの名前が取得できた場合、その名前の先頭から”¥¥”（バックスラッシュ）を 2 つ削除した文字列と、先ほど取得したコンピュータ名の文字列を比較することで、感染した端末がドメインコントローラであるかどうかをチェックします（下図参照）。

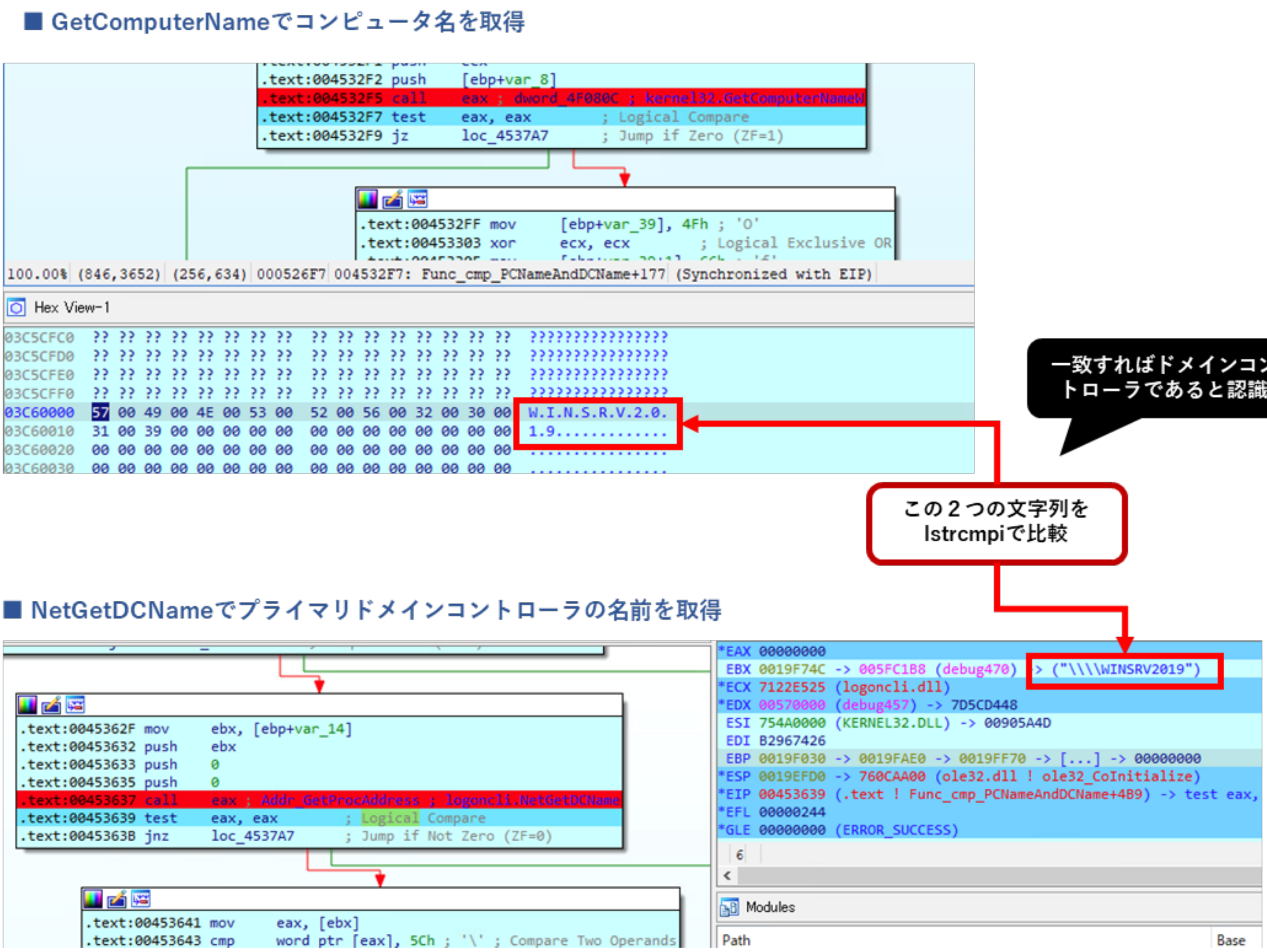
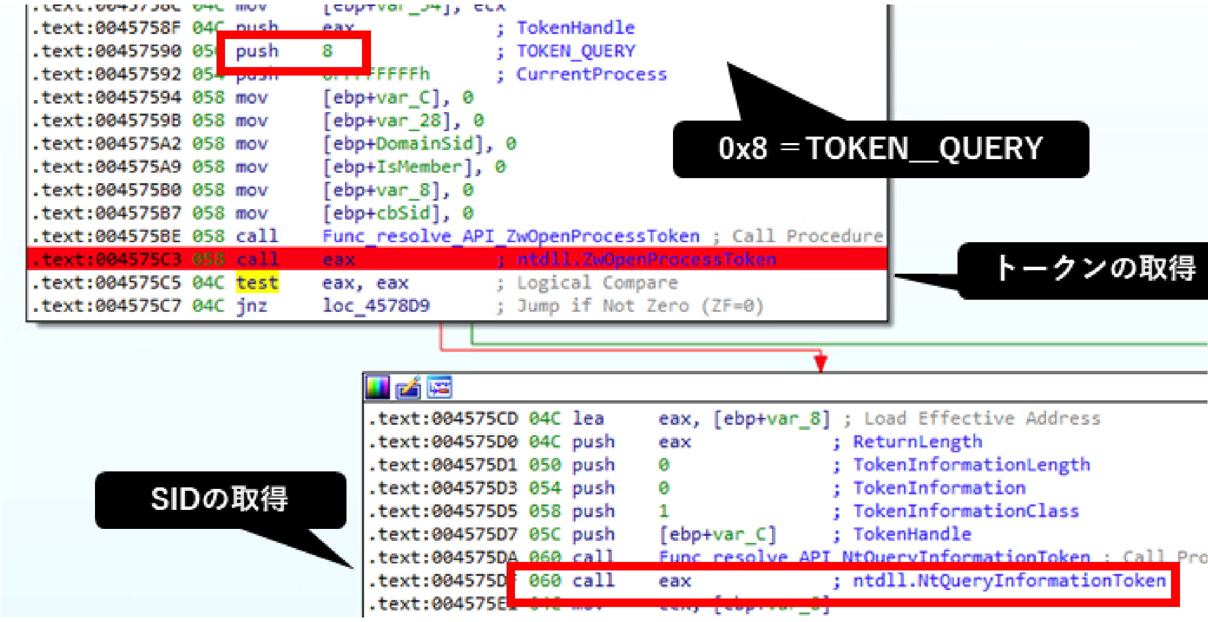


図 66　ドメインコントローラの名前を取得し判別する処理

ドメインコントローラであると判断した場合、以下の処理を継続します。

LockBit2.0 は自身のプロセスの実行アカウントが「DomainAdmins」(ドメイン管理者グループ)に属しているかどうかをチェックします。
具体的には次の方法で行います。
ZwOpenProcessToken 関数に TOKEN_QUERY(0x8)アクセス権を渡すことで現在のプロセスのトークンを取得したのち NtQueryInformationToken 関数で現在の SID を取得、取得した現在の SID から、GetWindowsAccountDomainSid 関数を使用してドメイン SID を取得します（下図参照）。

■ ZwOpenProcessTokenで現在のプロセスのトークンを取得 → SIDを取得



■ GetWindowsAccountDomainSid関数を使用してドメインSIDを取得

図 67　ドメイン SID を取得する処理

取得したドメイン SID と WellKnownSidType として知られる「WinAccountDomainAdminsSid」（0x29=41）という値を CreateWellKnownSid 関数に渡すことで DomainAdmins(ドメイン管理者グループ)の SID を取得(作成)、最後に CheckTokenMembership 関数を利用して、呼び出し元のトークン内で Domain Admins(ドメイン管理者グループ)の SID が有効になっているかどうかをチェックします（下図参照）。
(※もし DomainAdmins に属していないと判断すると「Don’t have admin rights」という文字列をログ用にメモリ上に作成します)

■ DomainAdmins（ドメイン管理者グループ）に属しているかチェック

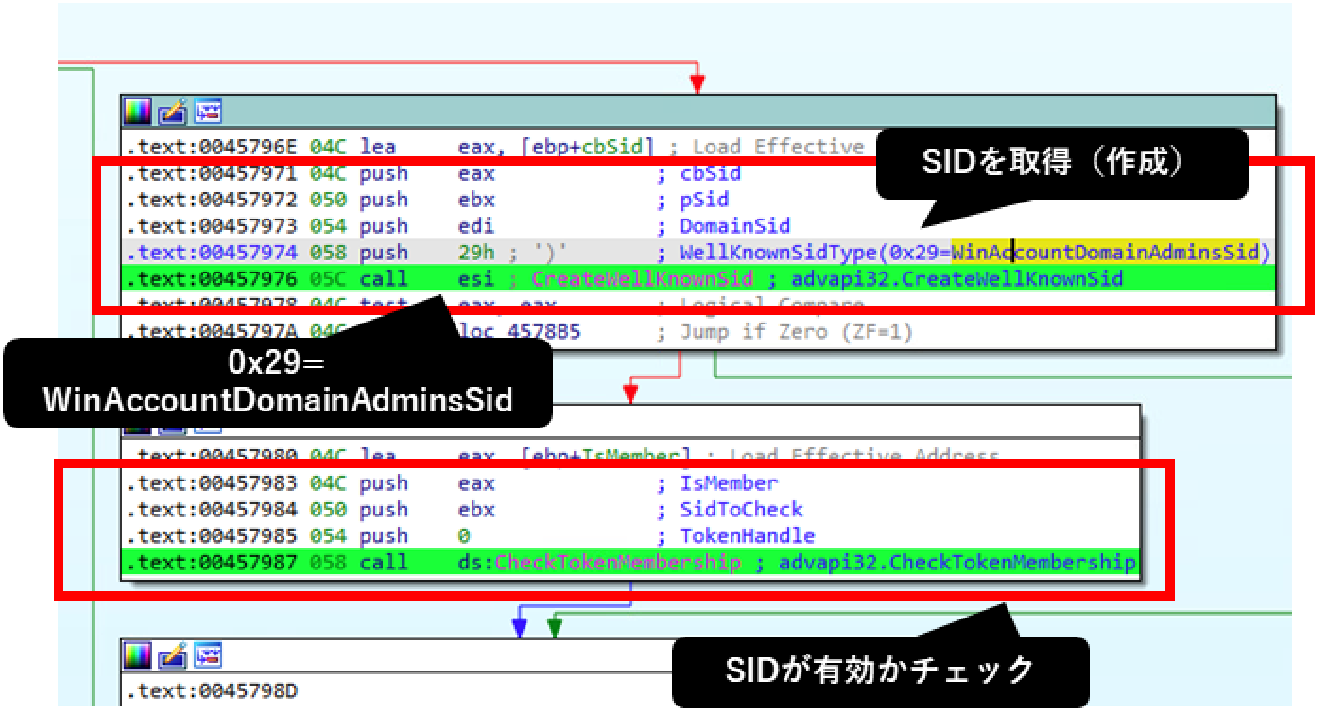


図 68　DomainAdmins に属しているかどうかのチェック処理

DomainAdmins に属していると判断すると、LockBit2.0 は DomainAdmins のアカウント名（ドメイン管理者）を取得します。
具体的には次の方法で行います。
先ほどと同じく ZwOpenProcessToken 関数に TOKEN_QUERY(0x8)アクセス権を渡すことで現在のプロセスのトークンを再度取得したのち、NtQueryInformationToken 関数で現在の SID を取得、LookupAccountSidW 関数を使用し、取得した SID に対するアカウント名（Administrator）とアカウント名が見つかったドメイン名（例：ADTEST）を取得します（下図参照）。

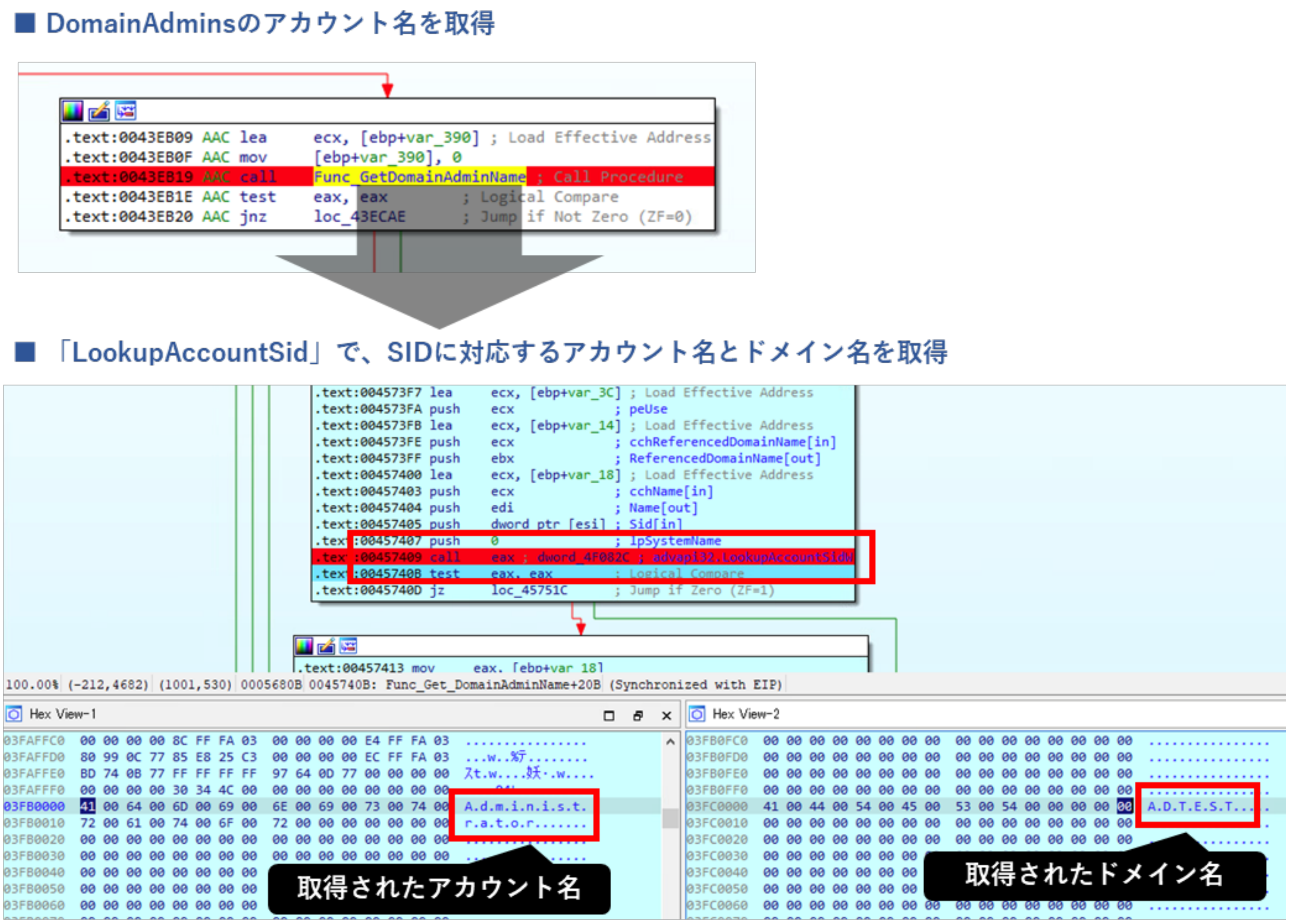


図 69　DomainAdmins のアカウント名とドメイン名を取得する処理

以上により、ドメインコントローラであることの確認およびドメインコントローラの情報を取得できた後、LockBit2.0 は、グループポリシーを作成し、ドメイン配下の全てのクライアント端末にランサムウェアを配布する挙動を行います。

このグループポリシーによる自身の拡散という仕組みは、暴露系ランサムウェアとしては史上初の新機能であると考えられます。

LockBit2.0 がドメイン内の各端末に配布するグループポリシーは以下に挙げる 6 つです（下図参照）。

LockBit2.0が配布するグループポリシーは以下の⑥つ

① 自身のEXEファイルを配布するポリシー　（コピーして取得させる）

② 配布したEXEファイルを実行するためのポリシー（スケジュールタスクを利用）

③ 指定した多数のプロセスを強制終了させるポリシー（スケジュールタスクを利用）

④ 指定した多数のサービスを強制停止させるポリシー

⑤ Windows Defenderの各種機能を無効化させるポリシー

⑥ 全てのローカルドライブを強制的にネットワーク共有に設定するポリシー

PowerShellコマンドにより強制的に配布され即時適用される

図 70　LockBit2.0 が配布するグループポリシー

それぞれについて、以降で詳細解説していきます。

LockBit2.0 は、まず CreateFile でグループポリシー設定ファイルである「GPT.ini」を作成します（下図参照）。

■ LockBit2.0により作成されるグループポリシー設定ファイル

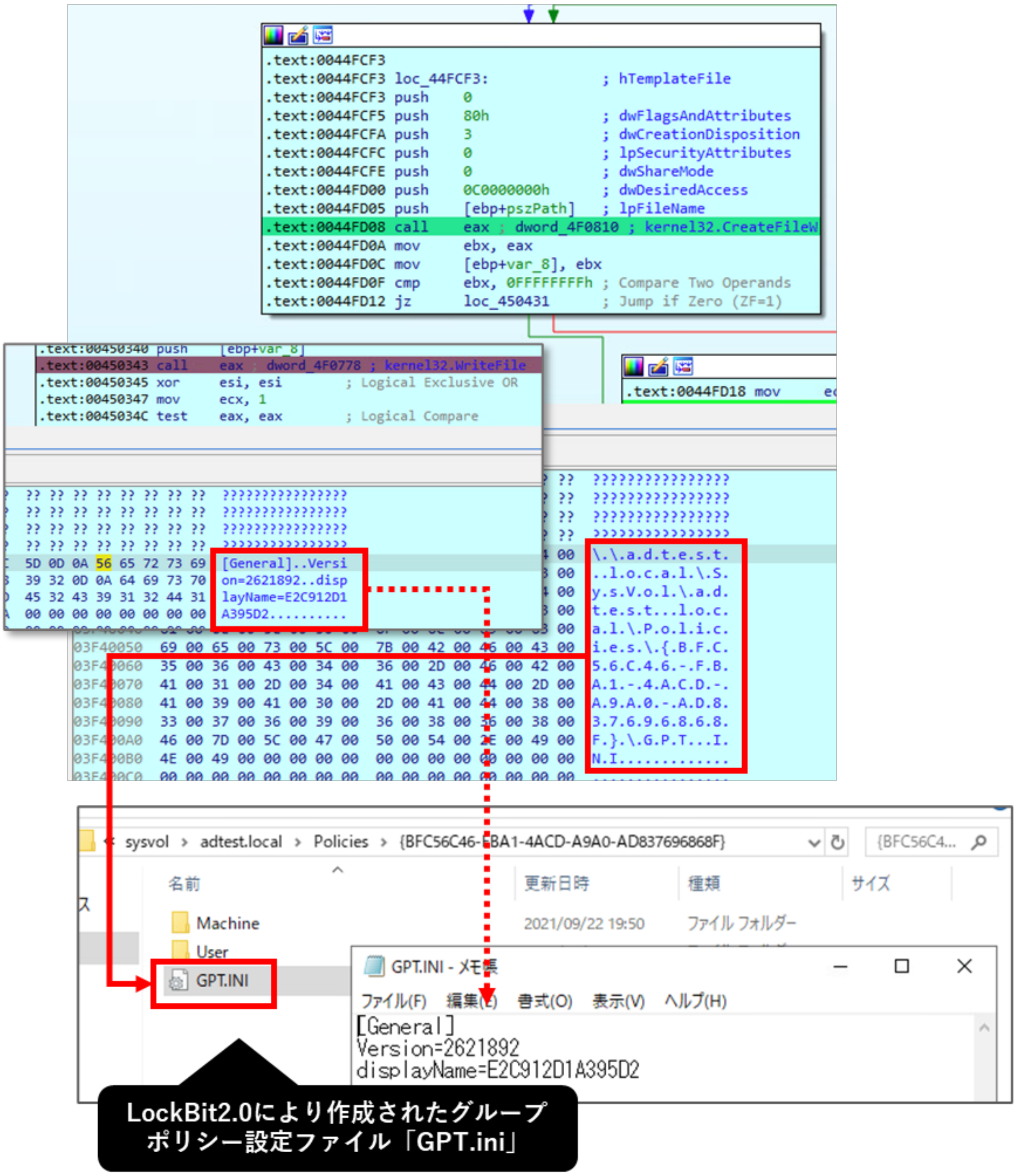


図 71　LockBit2.0 により作成される「GPT.INI」

続いて CreateGPOLink 関数により、指定した GPO（グループポリシーオブジェクト）と OU(Active Directory でアカウント管理の最小単位となるアカウント・コンピュータ・リソースの集合)をリンクさせグループポリシーの適応先を設定（下図参照）、ADSI 関数(Active Directory サービスインターフェイス)である ADSGetObject 関数を使用した上でグループポリシーの属性(gPCMachineExtensionNames や gPCUserExtensionNames などの拡張属性)の設定を行います。ADSGetObject 関数はアプリケーションが Active Di rectory(AD)にアクセスするための専用 API であり、この関数により AD 内のオブジェクトの検索・閲覧・編集・削除などが実行できるようになります。

■ **作成したグループポリシーをCreateGPOLinkを用いてOU(Organizational Unit)にリンク（適用）させる**

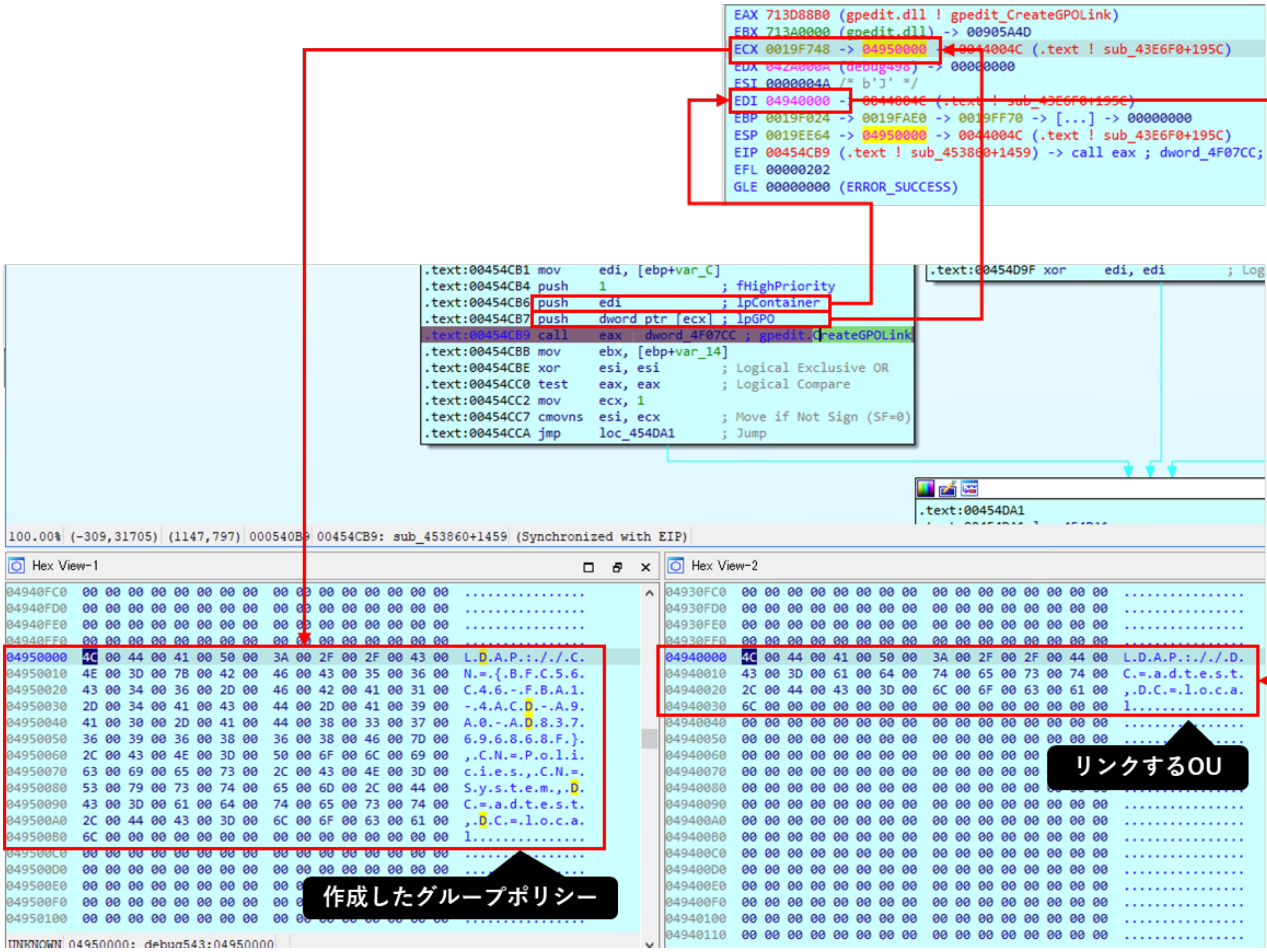


図 72 グループポリシーを OU にリンクさせる処理

その後、ドメイン内のクライアントに配布するための複数のグループポリシーファイルを作成していきます。

まずは、「LockBit2.0 本体をドメイン配下に一斉配布するグループポリシー」を作成します。
あらかじめ、ドメインコントローラにある SYSVOL フォルダ内 scripts フォルダに自身のコピーを作成した上で、「Files.xml」というグループポリシー設定ファイル(xml)を作成します。
作成された「Files.xml」には、ドメインコントローラの scripts フォルダに作成した LockBit2.0 の本体 EXE を、ドメイン配下の各クライアント端末のデスクトップ上にコピーするよう指示するポリシーが記述されています（下図参照）。

■ LockBit2.0により配布されるグループポリシー　< LockBit2.0本体をドメイン配下に一斉配布するポリシー >

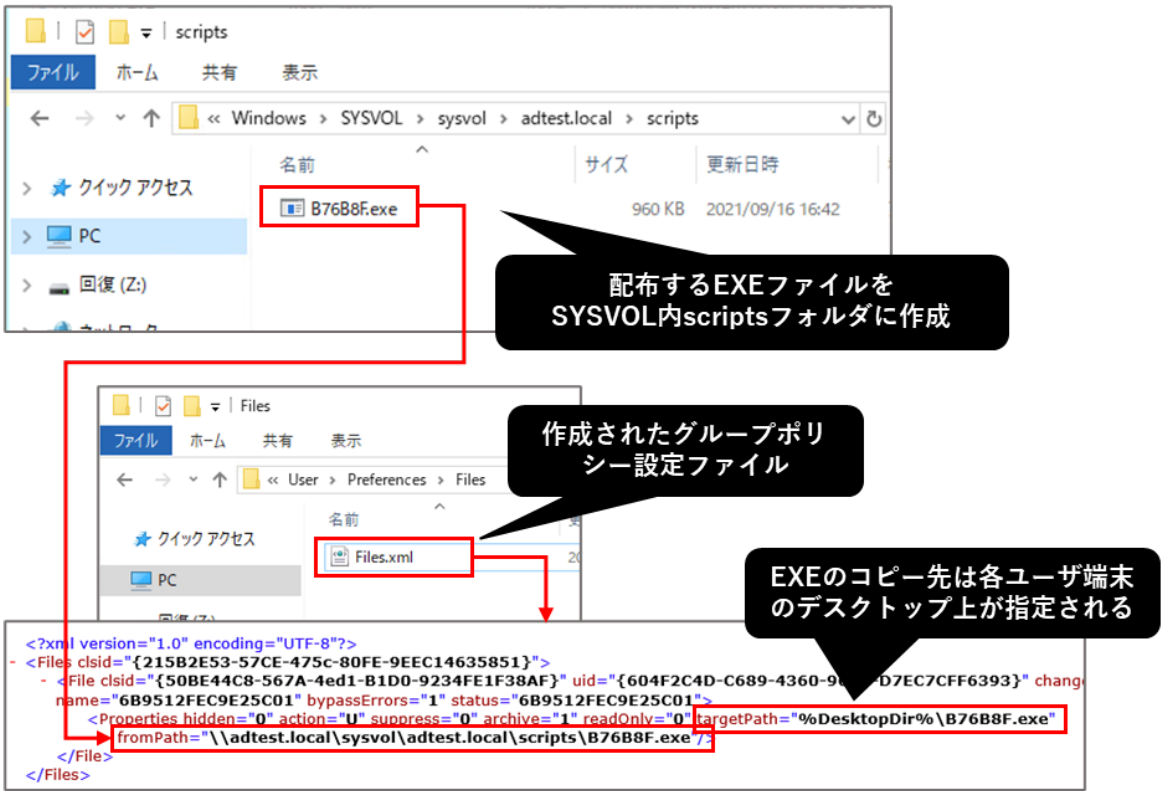


図 73　LockBit2.0 の本体をドメイン配下に一斉配布するポリシー

上記のグループポリシーをより分かりやすく表示し直したものが以下です（下図参照）。
つまり、この「File.xml」のグループポリシーが適用された瞬間に、ドメイン配下の全ての端末のデスクトップに LockBit2.0 の EXE ファイルがコピーされます。

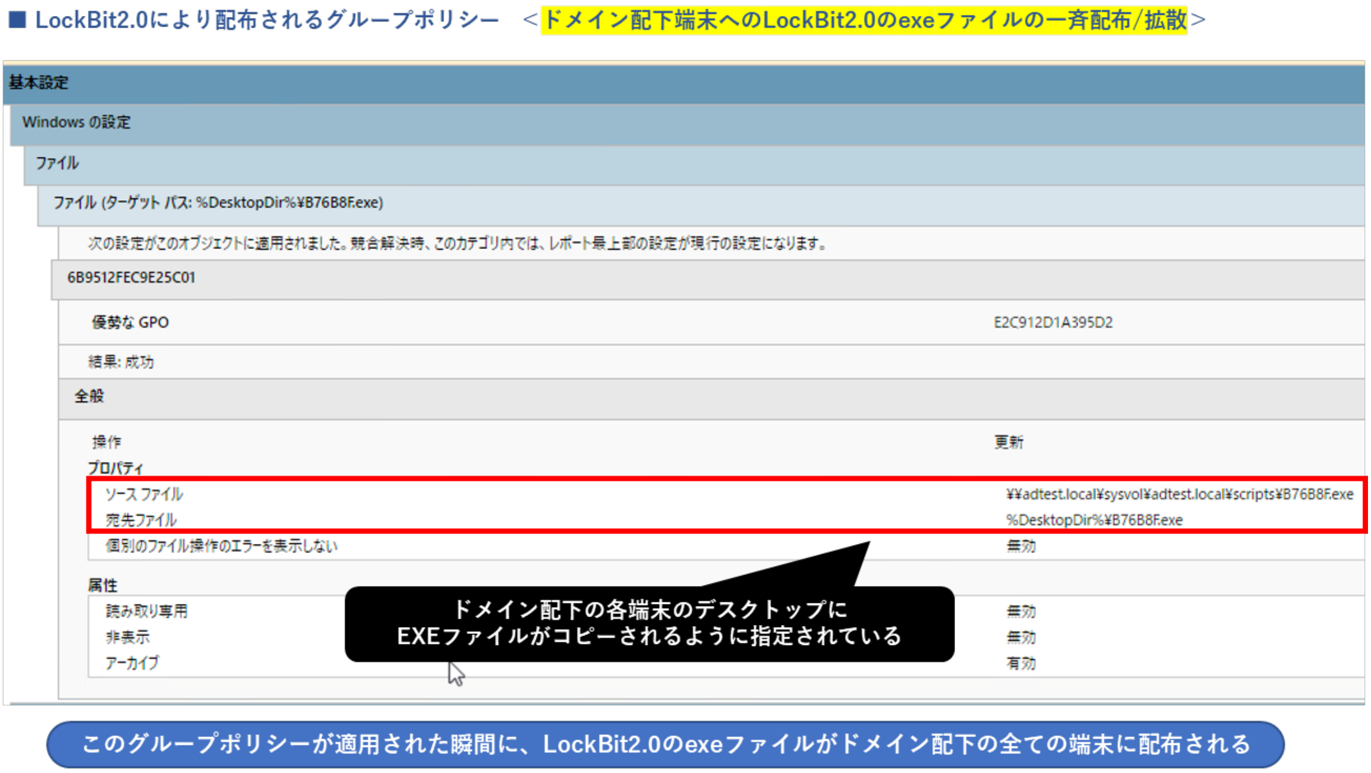


図 74　ドメイン配下端末への LockBit2.0 の EXE ファイルの一斉配布が指定されたポリシー

続いて、「ScheduledTasks.xml」というグループポリシー設定ファイル(xml)を作成します。
この「ScheduledTasks.xml」には、2 つのスケジュールタスクがまとめて記述されています。「ScheduledTasks.xml」の前方に「①プロセスの強制終了」に関するスケジュールタスクが、「ScheduledTasks.xml」の後方に「②EXE ファイルの実行」に関するスケジュールタスクが設定されています。

■ LockBit2.0により配布されるグループポリシー　＜スケジュールタスクポリシーの実体ファイル＞

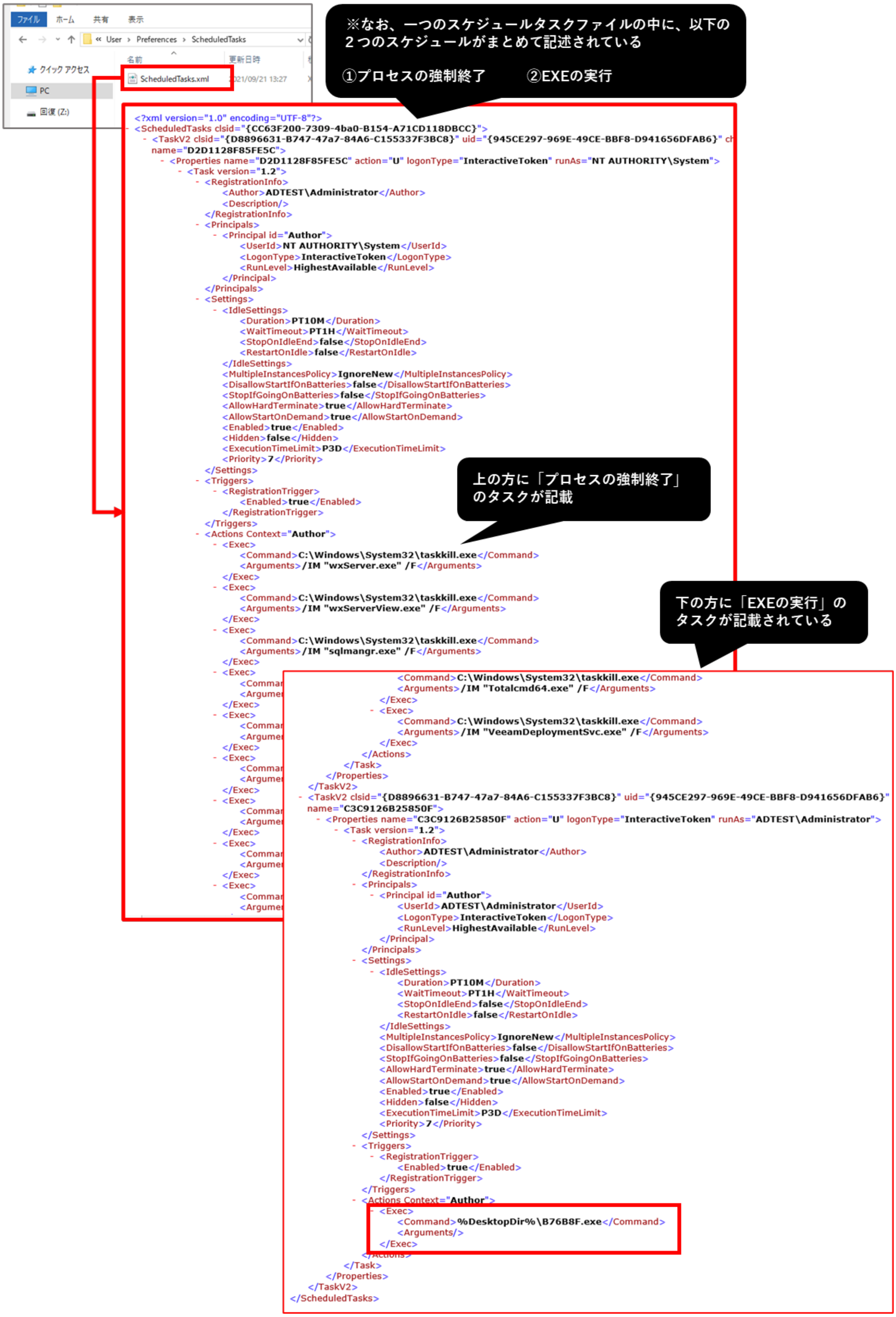


図 75　LockBit2.0 が作成する「ScheduledTasks.xml」

上記のグループポリシーをより分かりやすく表示し直したものが以下です。
「プロセスの強制終了」に関するスケジュールタスクには、インシデント調査ツールや、データベースアプリケーション、サーバアプリケーションなど、暗号化の邪魔となるプロセスの名前が並び、taskkill コマンドにより即時強制終了されるよう設定されていることがわかります（下図参照）。

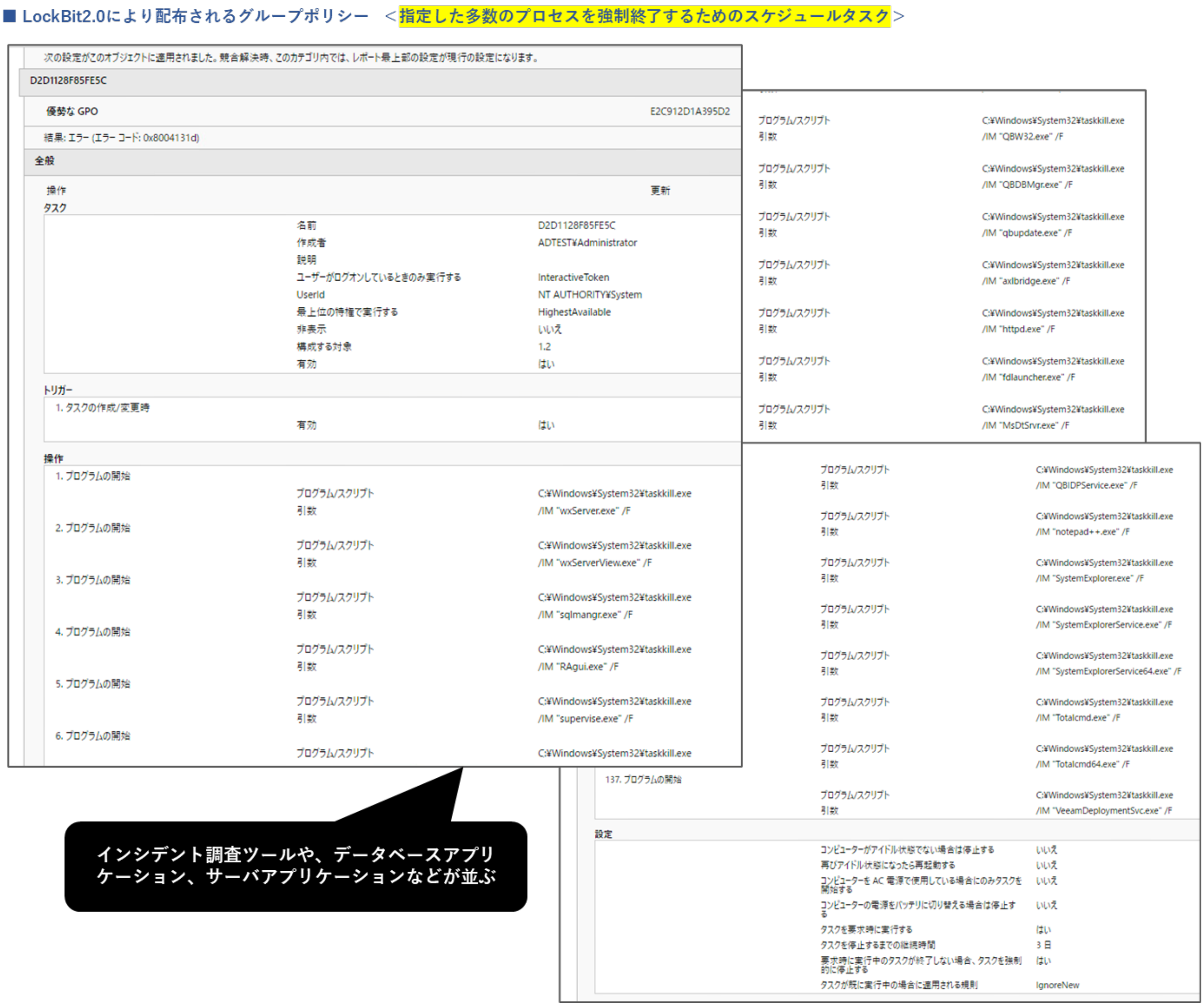


図 76 指定した多数のプロセスを強制終了させるスケジュールタスク

また、「ScheduledTasks.xml」にある「EXE ファイルの実行」に関するスケジュールタスクには、さきほどの「File.xml」によってあらかじめ作成されたデスクトップ上の EXE ファイル（LockBit2.0 本体のコピー）を即時実行するルールが記述されています（下図参照）。

■ LockBit2.0により配布されるグループポリシー

＜配布したLockBit2.0のexeファイルを実行させるためのスケジュールタスク＞

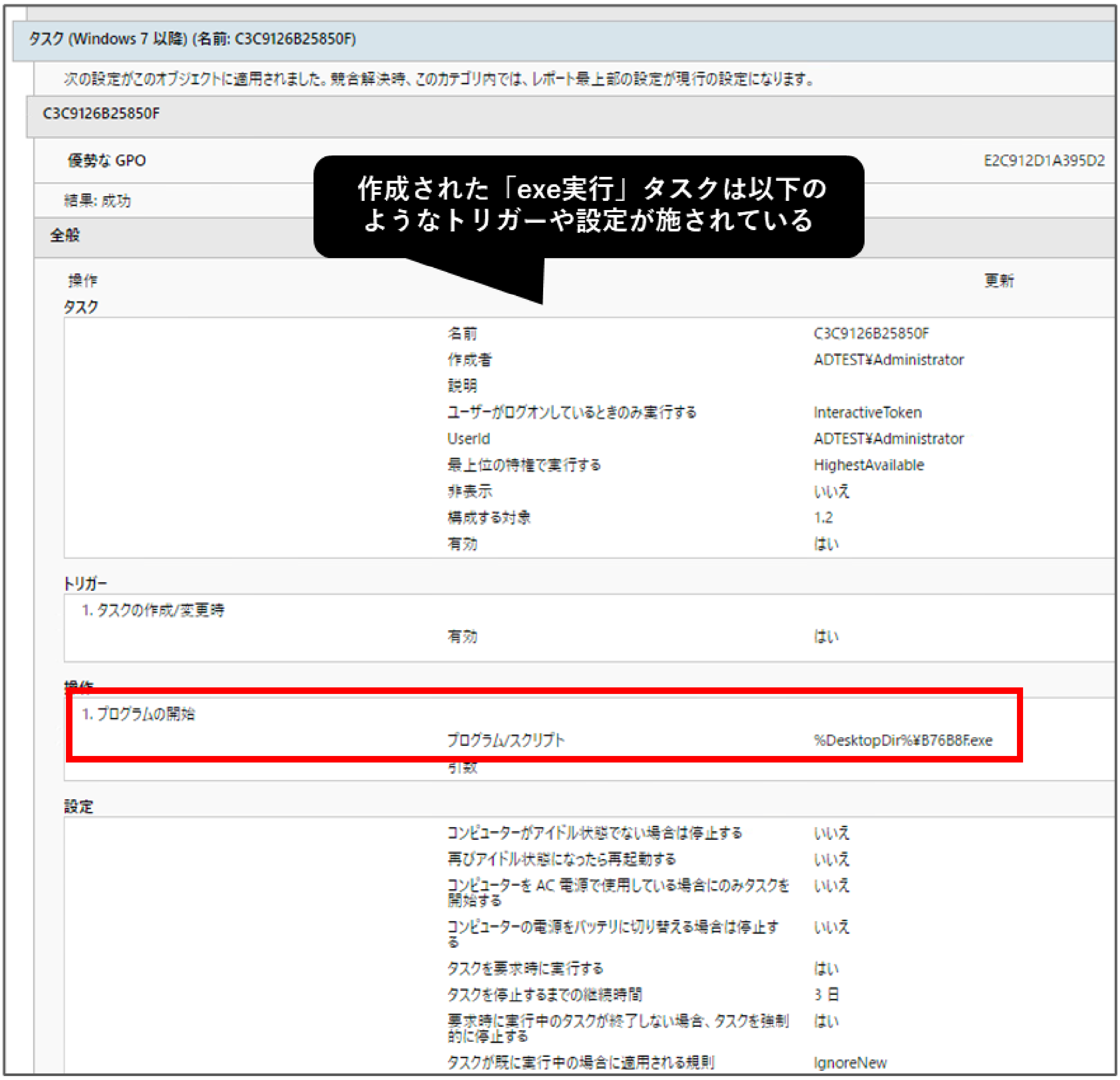


図 77　配布した EXE を実行させるためのスケジュールタスク

以下は、上記のグループポリシーにより、クライアント端末上に実際に配布されたスケジュールタスクの中身ですが、上記で解説したものと同じものがスケジュールタスクとして設定されていることがわかります（下図参照）。

■ グループポリシーが配布され、実際にユーザー端末上に作成されたスケジュールタスクとその中身

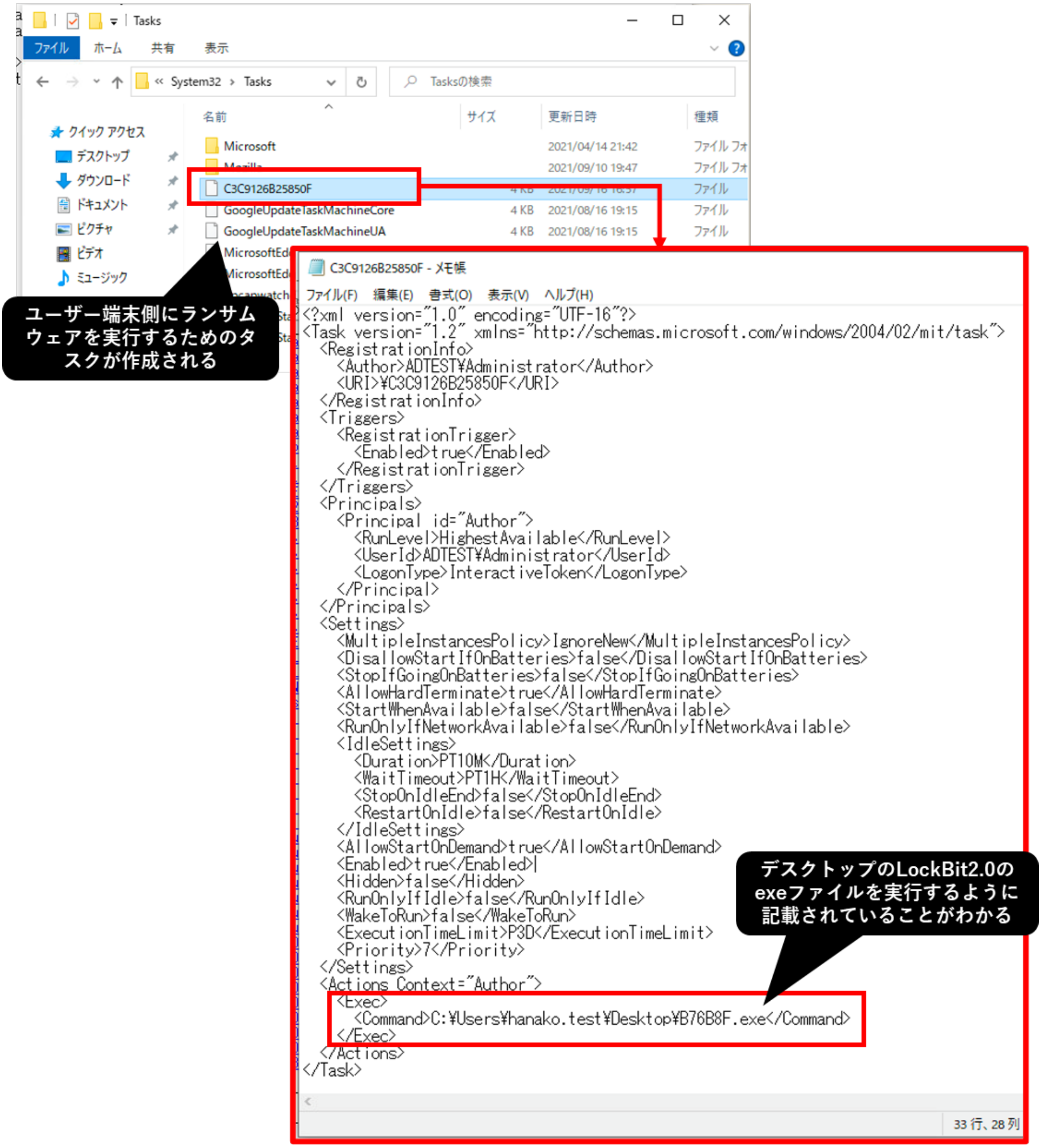


図 78　実際にクライアント端末側に作成されたスケジュールタスク

また、クライアント端末上のタスクスケジューラから確認したものが以下ですが、「プログラムの開始」としてデスクトップ上の LockBit2.0 の本体の EXE パスが設定されていることがわかります（下図参照）。

■ リモートから設定されたスケジュールタスクをユーザ端末上の「タスクスケジューラ」で確認した様子

図 79　タスクスケジューラで確認した様子

続いて、「Services.xml」と「NetworkShares.xml」という 2 つのグループポリシー設定ファイル(xml)を作成します（下図参照）。

■ LockBit2.0により作成されるグループポリシーの一部(サービスとネットワーク共有に関する設定)

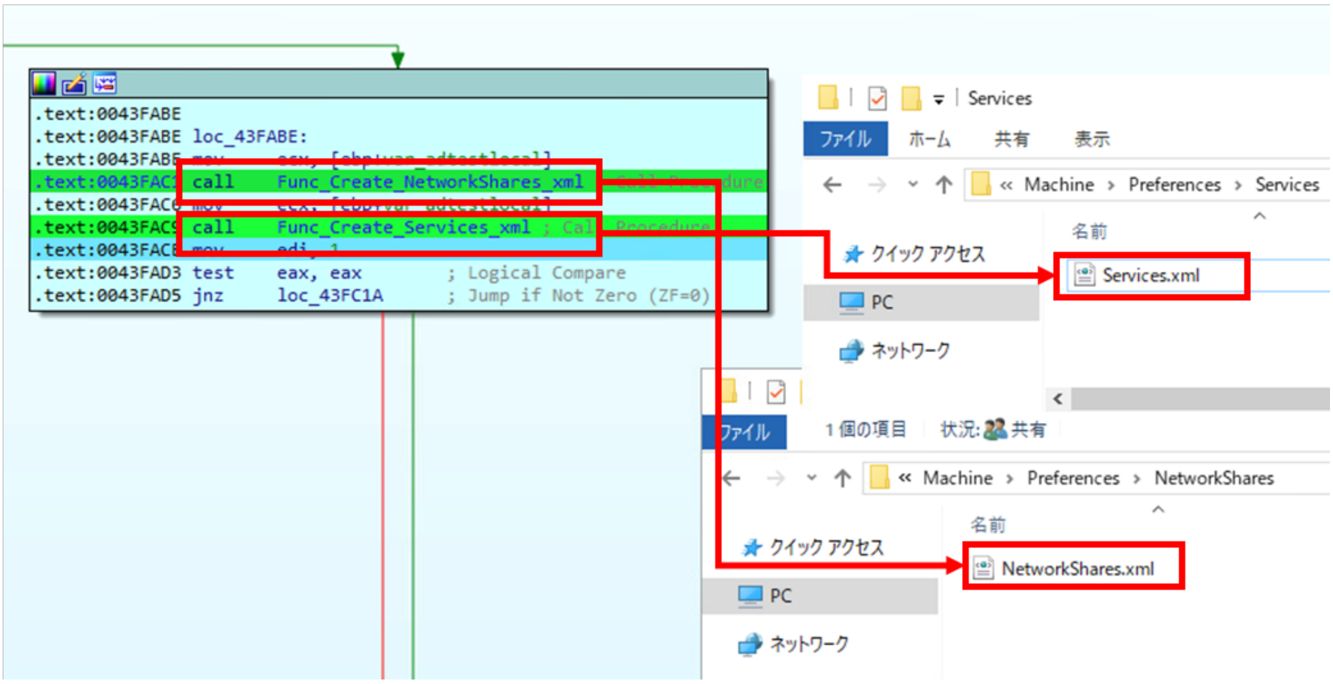
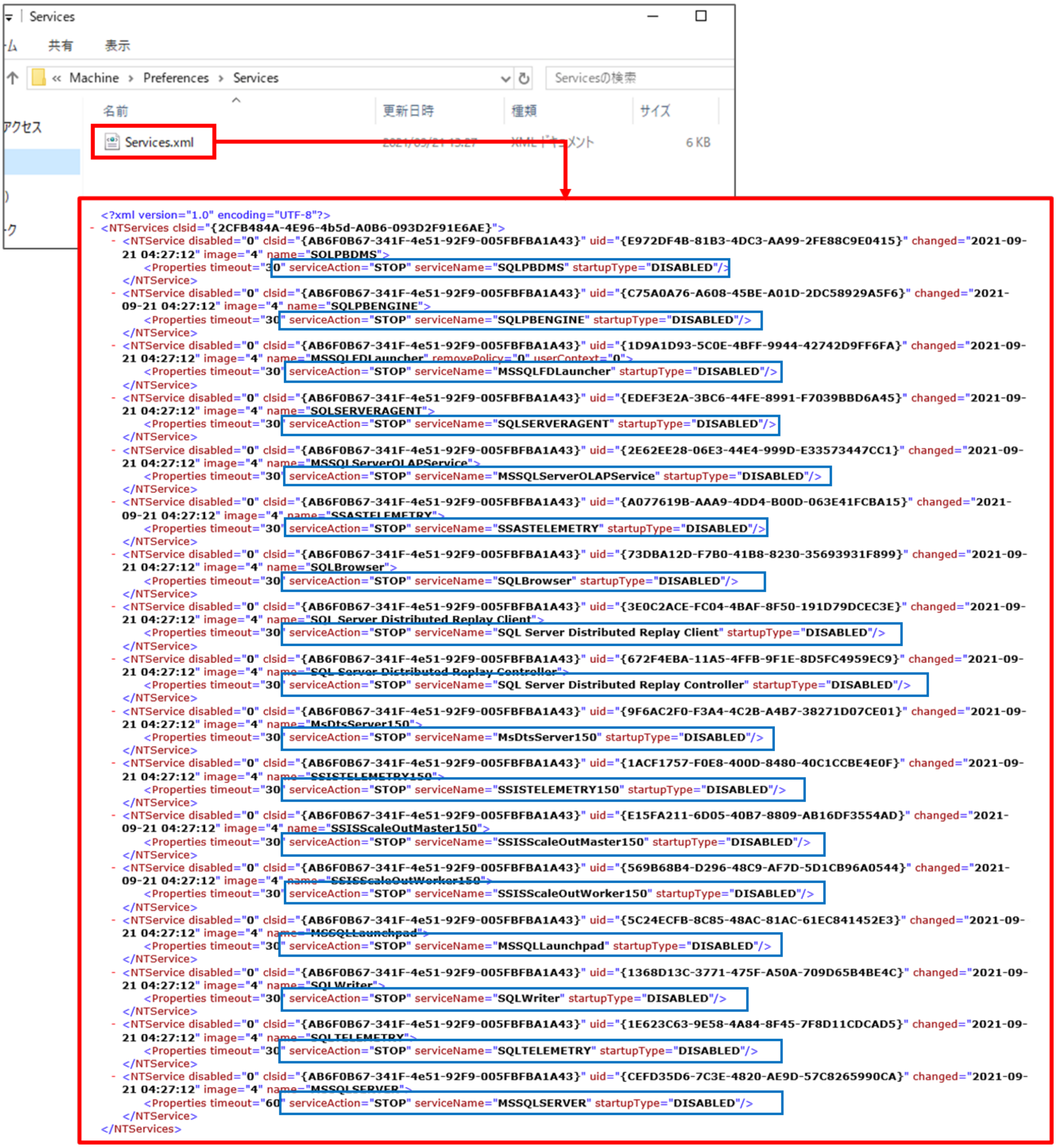


図 80　作成されるサービスとネットワーク共有に関するポリシー設定ファイル

上記で作成された「Services.xml」には、クライアント端末上の複数のサービス（暗号化の邪魔となるサービスなど）を停止させる処理が記述されています（下図参照）。

■ LockBit2.0により配布されるグループポリシー　＜サービスの無効化ポリシー＞




```xml
<?xml version="1.0" encoding="UTF-8"?>
- <NTServices clsid="{2CFB484A-4E96-4b5d-A0B6-093D2F91E6AE}">
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{E972DF4B-81B3-4DC3-AA99-2FE88C9E0415}" changed="2021-09-21 04:27:12" image="4" name="SQLPBDMS">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLPBDMS" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{C75A0A76-A608-45BE-A01D-2DC58929A5F6}" changed="2021-09-21 04:27:12" image="4" name="SQLPBENGINE">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLPBENGINE" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{1D9A1D93-5C0E-4BFF-9944-42742D9FF6FA}" changed="2021-09-21 04:27:12" image="4" name="MSSQLFDLauncher" removePolicy="0" userContext="0">
      <Properties timeout="30" serviceAction="STOP" serviceName="MSSQLFDLauncher" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{EDEF3E2A-3BC6-44FE-8991-F7039BBD6A45}" changed="2021-09-21 04:27:12" image="4" name="SQLSERVERAGENT">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLSERVERAGENT" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{2E62EE28-06E3-44E4-999D-E33573447CC1}" changed="2021-09-21 04:27:12" image="4" name="MSSQLServerOLAPService">
      <Properties timeout="30" serviceAction="STOP" serviceName="MSSQLServerOLAPService" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{A077619B-AAA9-4DD4-B00D-063E41FCBA15}" changed="2021-09-21 04:27:12" image="4" name="SSASTELEMETRY">
      <Properties timeout="30" serviceAction="STOP" serviceName="SSASTELEMETRY" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{73DBA12D-F7B0-41B8-8230-35693931F899}" changed="2021-09-21 04:27:12" image="4" name="SQLBrowser">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLBrowser" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{3E0C2ACE-FC04-4BAF-8F50-191D79DCEC3E}" changed="2021-09-21 04:27:12" image="4" name="SQL Server Distributed Replay Client">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQL Server Distributed Replay Client" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{672F4EBA-11A5-4FFB-9F1E-8D5FC4959EC9}" changed="2021-09-21 04:27:12" image="4" name="SQL Server Distributed Replay Controller">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQL Server Distributed Replay Controller" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{9F6AC2F0-F3A4-4C2B-A4B7-38271D07CE01}" changed="2021-09-21 04:27:12" image="4" name="MsDtsServer150">
      <Properties timeout="30" serviceAction="STOP" serviceName="MsDtsServer150" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{1ACF1757-F0E8-400D-8480-40C1CCBE4E0F}" changed="2021-09-21 04:27:12" image="4" name="SSISTELEMETRY150">
      <Properties timeout="30" serviceAction="STOP" serviceName="SSISTELEMETRY150" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{E15FA211-6D05-40B7-8809-AB16DF3554AD}" changed="2021-09-21 04:27:12" image="4" name="SSISScaleOutMaster150">
      <Properties timeout="30" serviceAction="STOP" serviceName="SSISScaleOutMaster150" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{569B68B4-D296-48C9-AF7D-5D1CB96A0544}" changed="2021-09-21 04:27:12" image="4" name="SSISScaleOutWorker150">
      <Properties timeout="30" serviceAction="STOP" serviceName="SSISScaleOutWorker150" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{5C24ECFB-8C85-48AC-81AC-61EC841452E3}" changed="2021-09-21 04:27:12" image="4" name="MSSQLLaunchpad">
      <Properties timeout="30" serviceAction="STOP" serviceName="MSSQLLaunchpad" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{1368D13C-3771-475F-A50A-709D65B4BE4C}" changed="2021-09-21 04:27:12" image="4" name="SQLWriter">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLWriter" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{1E623C63-9E58-4A84-8F45-7F8D11CDCAD5}" changed="2021-09-21 04:27:12" image="4" name="SQLTELEMETRY">
      <Properties timeout="30" serviceAction="STOP" serviceName="SQLTELEMETRY" startupType="DISABLED"/>
    </NTService>
  - <NTService disabled="0" clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" uid="{CEFD35D6-7C3E-4820-AE9D-57C8265990CA}" changed="2021-09-21 04:27:12" image="4" name="MSSQLSERVER">
      <Properties timeout="60" serviceAction="STOP" serviceName="MSSQLSERVER" startupType="DISABLED"/>
    </NTService>
</NTServices>
```


図 81 LockBit2.0 が作成した「Services.xml」

もう一つ作成される「NetworkShares.xml」には、クライアント端末上の全てのドライブドライブをネットワーク共有に設定するためのポリシーが記述されています（下図参照）。この結果、他の感染端末からネットワーク越しに、それらのネットワーク共有されたドライブ内のファイルが暗号化されてしまう状況に陥ります。

■ LockBit2.0により配布されるグループポリシー　＜ローカルドライブのネットワーク共有化ポリシー＞

図 82　LockBit2.0 が作成した「NetworkShares.xml」

続いて、「Registry.pol」というグループポリシーファイルを作成します（下図参照）。

この「Registry.pol」には、Windows Defender を無効化するための複数のレジストリ設定が記述されています。このポリシーが配布され適用されることにより、ドメイン配下の全てのクライアント端末上で Windows Defender が無効化される可能性があります。

■ LockBit2.0により配布されるグループポリシー　＜Windows Defednderの無効化ポリシー＞

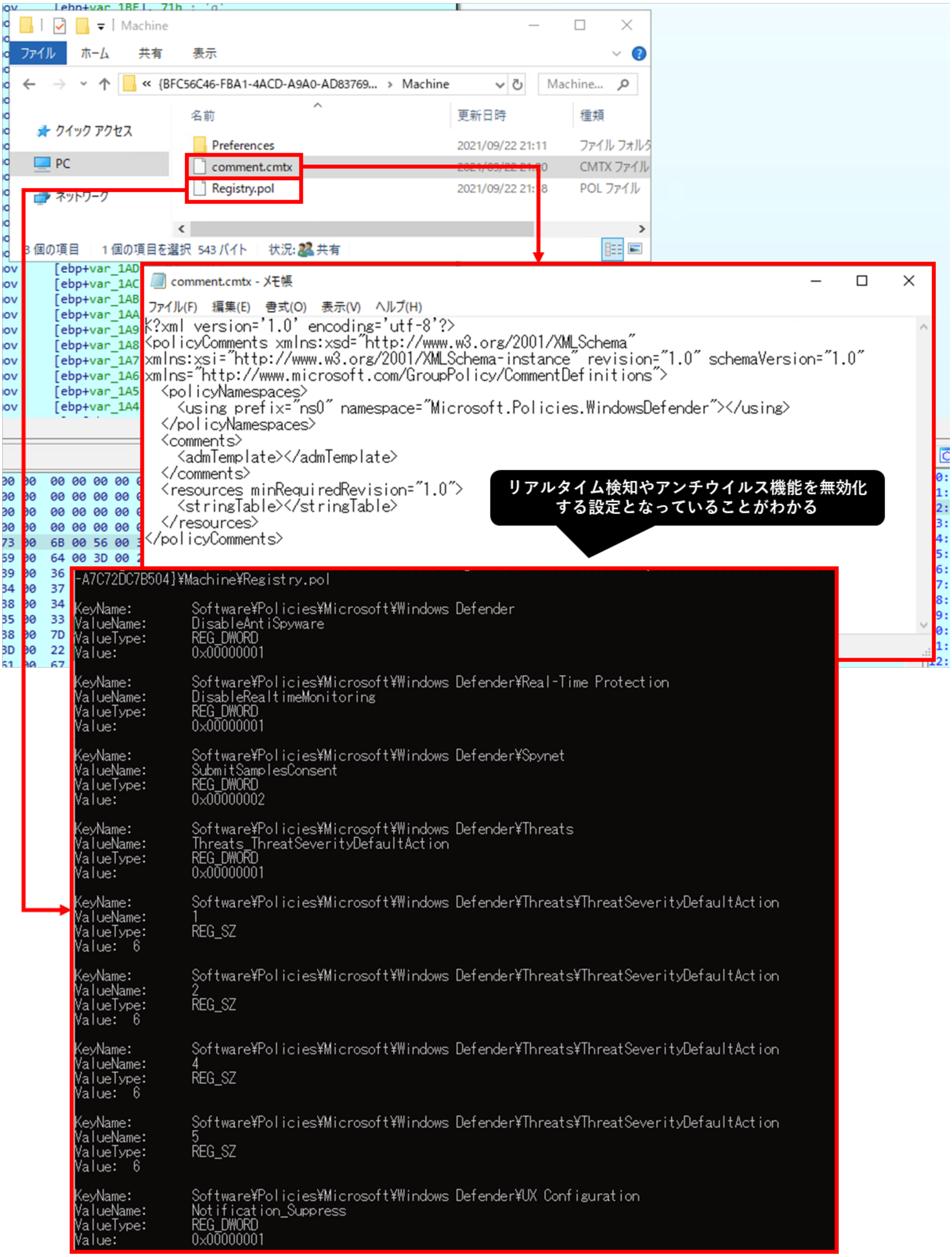


図 83　LockBit2.0 が作成した「Registry.pol」

そして最後に、LockBit2.0 は PowerShell コマンドにより、強制的にグループポリシーをドメイン配下の全端末へ一斉配布します（下図参照）。

■ PowerShellコマンドにより、強制的にグループポリシーをドメイン配下の全端末へ一斉配布する

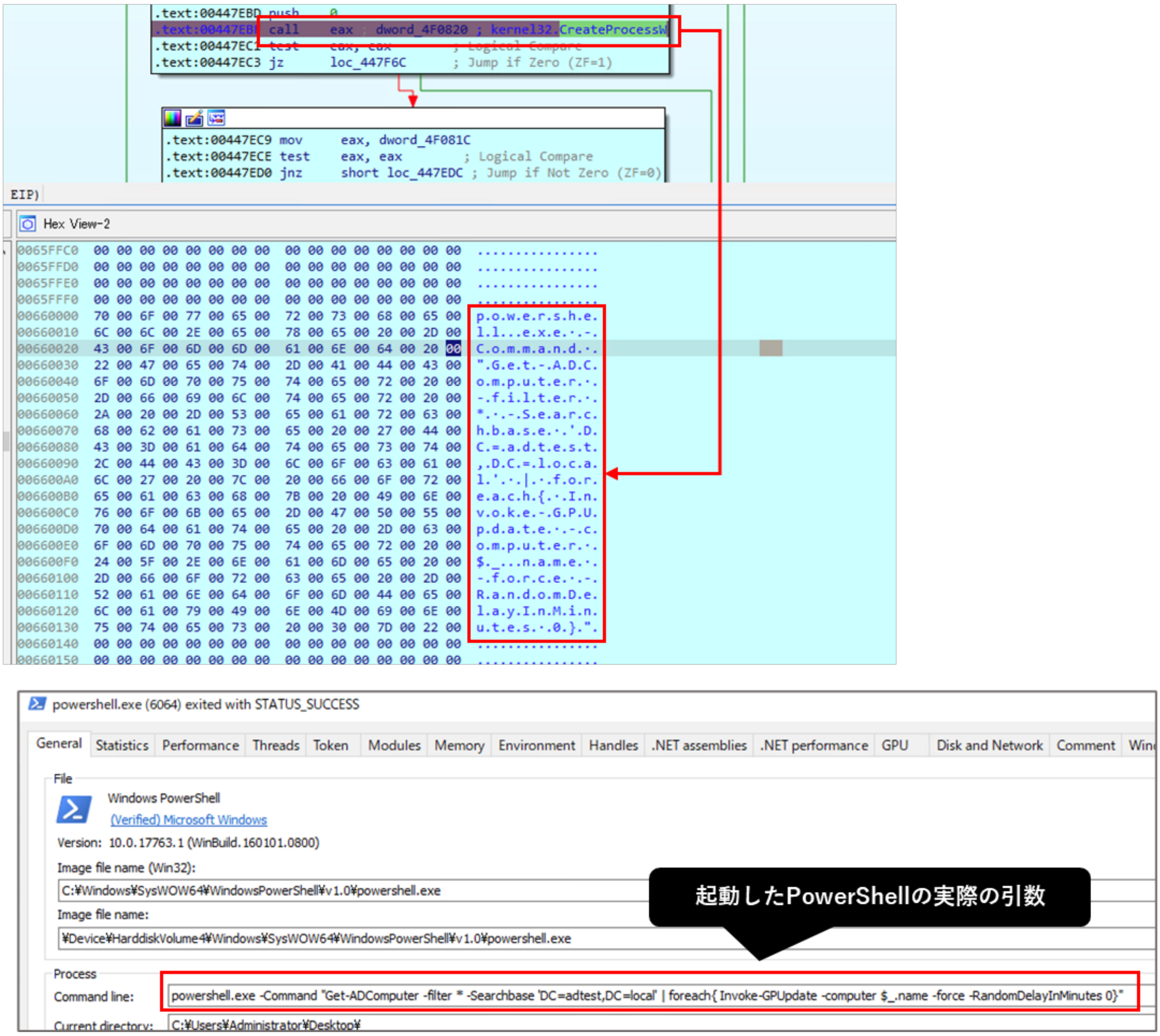


図 84　PowerShell コマンドによりグループポリシーが一斉配布される様子

上記の PowerShell のコマンドは以下のような構成となっており、「指定した OU に存在する全ての端末を取得」し、見つかった各端末に「グループポリシーを即時強制適用させる」コマンドであることがわかります（下図参照）。

■強制的にグループポリシーをドメイン配下の全端末へ一斉配布するPowerShellコマンドの構成

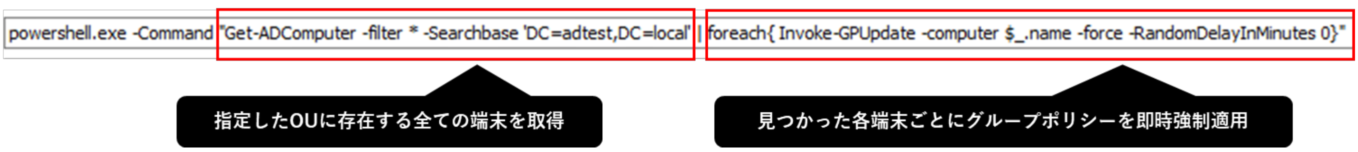


図 85　グループポリシーを一斉配布させる PowerShell コマンドの構成

上記の PowerShell が実行された瞬間に、ドメインコントローラ上ではコマンドプロンプトが開き、グループポリシーが更新される旨のメッセージが表示されます。またその直後、ドメイン配下にあるクライアント端末上でも同様にグループポリシーが更新されるプロンプトが表示されます（下図参照）。

（この際、LockBit2.0 は自身の内部で「Run on all domain(waiting 1 min)」というログをメモリ上に展開します。）

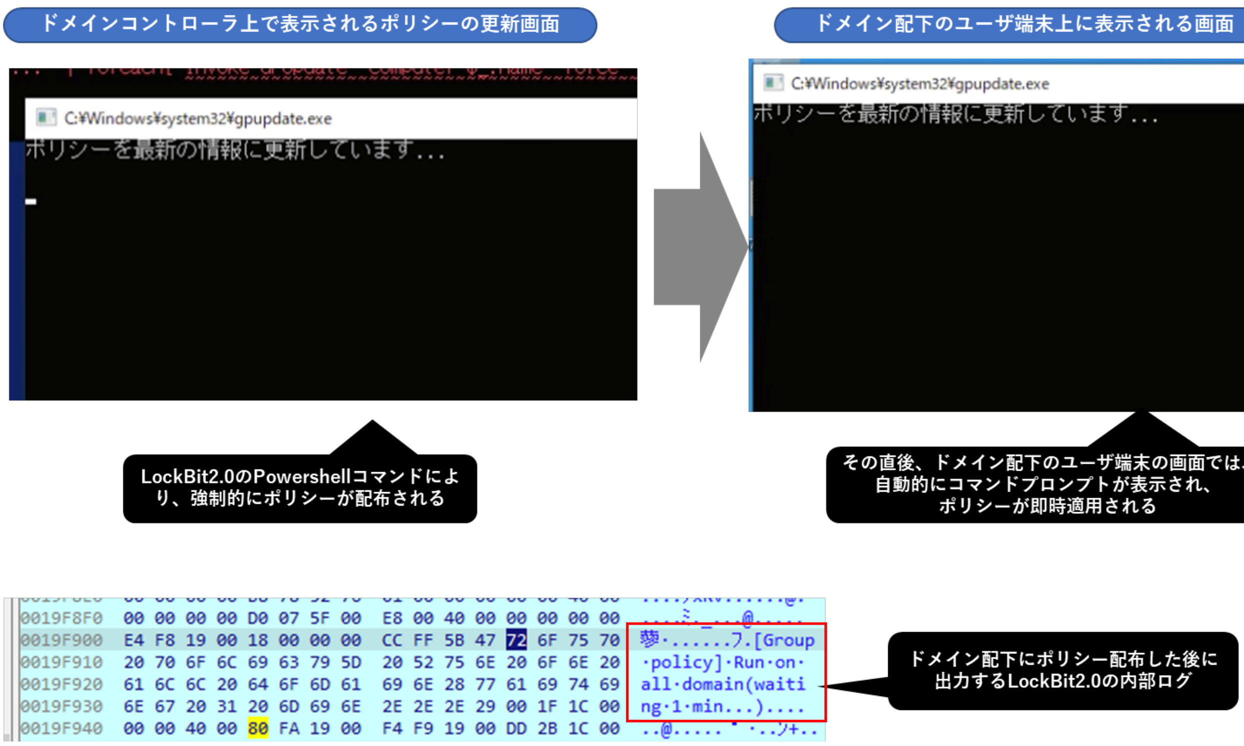


図 86　ドメインコントローラおよびクライアント端末上でグループポリシーが更新される様子

上記のポリシーが強制的に適用された瞬間、クライアント端末上のデスクトップにはコピーされた LockBit2.0 の EXE ファイルが作成され実行されます（下図参照）。

■ 上記の強制ポリシー適用の直後、ユーザ端末のデスクトップにLockBit2.0のexeファイルが作成される

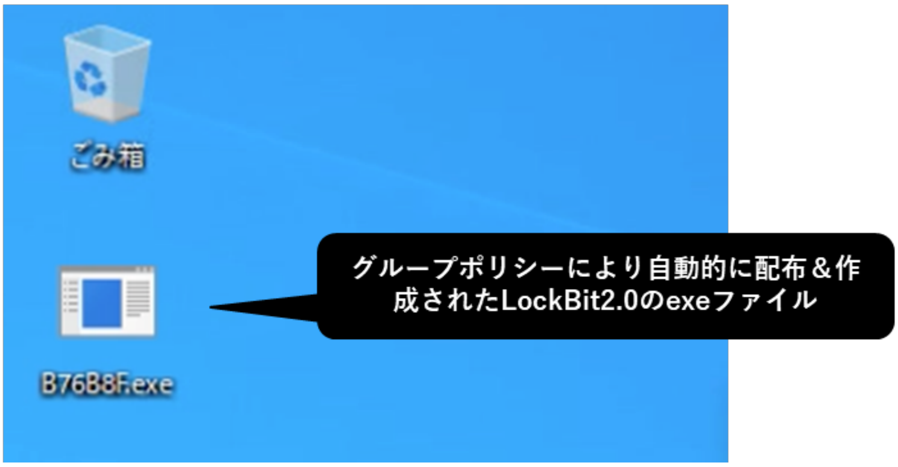


図 87　グループポリシーの適用後デスクトップ上に作成される EXE（LockBit2.0 の本体）

グループポリシーによる不正処理の一斉展開の仕組みは以上となります。

ドメインコントローラでのグループポリシーを利用したランサムウェアの展開の様子を動画に収録し、以下の YouTube で公開していますので併せてご覧ください。

https://www.youtube.com/watch?v=Co8mRfd0iPM

■LockBit2.0 のバグ（作成されたタスクのミス）

なお、「ScheduledTasks.xml」に記載されていたデスクトップ上の EXE ファイルを実行させるタスクですが、調査の結果、複数の検体において、グループポリシーで配布された EXE の実行が、クライアント端末側で失敗する個体が存在することを確認しました。

LockBit2.0 は、各クライアント端末に「ドメイン管理者アカウント」（Administrator）が前提条件として「ログオンしている必要がある」タスクを設定してしまっており、現実的にはこの起動条件が満たさせることは通常はあまり起こりえません。
ただし、このタスク内の権限設定が「ドメイン管理者アカウント」（Administrator）ではなく「SYSTEM アカウント」または「ユーザアカウント」が指定されていれば、攻撃者の意図する状況が成功していたものと考えられます。実際に、LockBit2.0 が作成した「ScheduledTasks.xml」の中の権限設定の記述を手動で書き換え、「SYSTEM アカウント」を指定したものを配布するようにした場合、ポリシーが配布された直後に LockBit2.0 がタスクにより実行され自動感染することが確認されました（下図参照）。

LockBit2.0のバグと想定される挙動

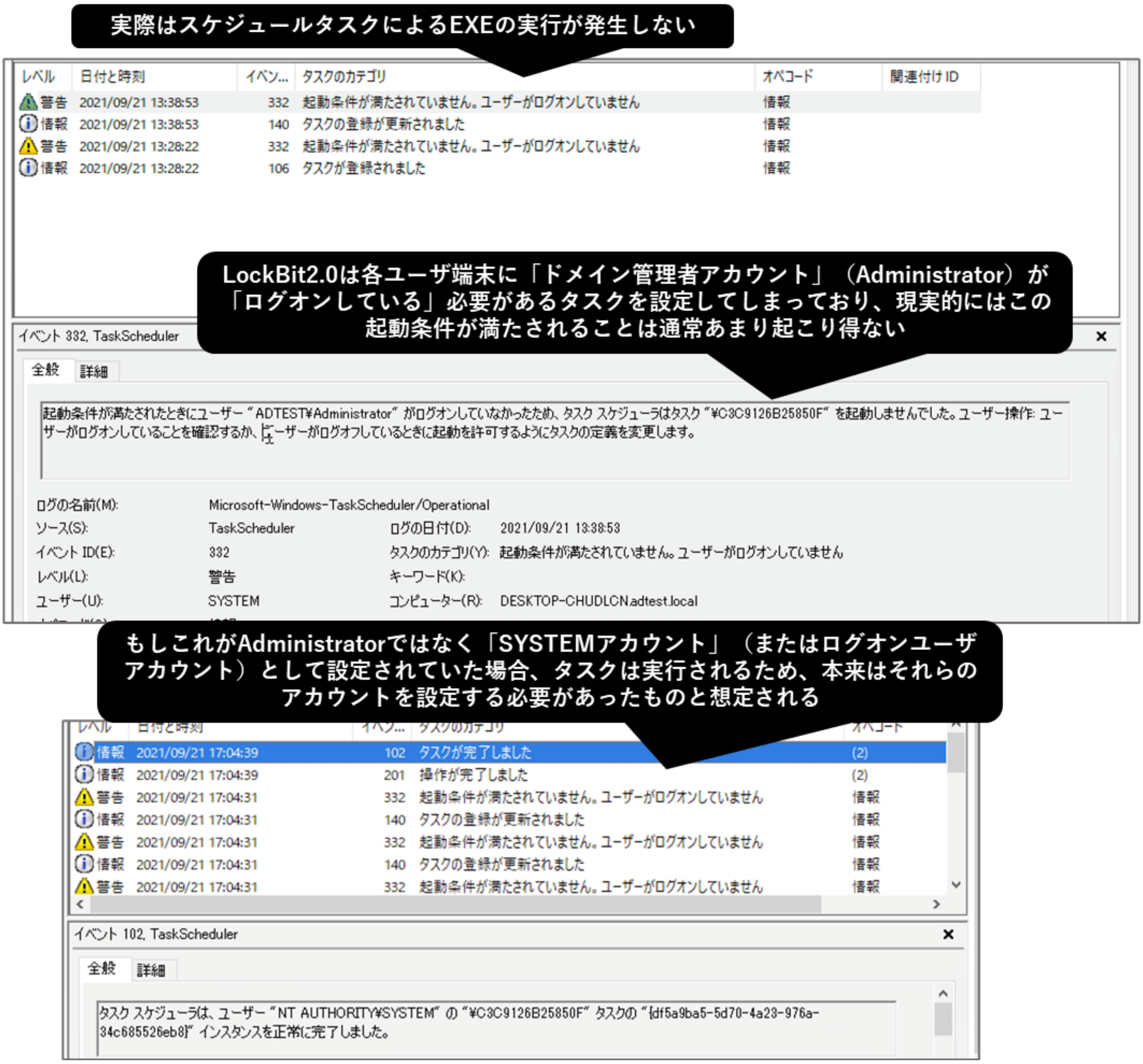


図 88　LockBit2.0 のスケジュールタスクに発見されたバグと想定される設定

ただし、一連の調査を進めた後、新しく出現した LockBit2.0 の検体において、このバグが修正されている個体を発見しました。

古い検体と新しい検体によって作成された「ScheduledTasks.xml」の記述をそれぞれ比較すると、上記でバグであると推測していた部分の記述が以下のようにしっかりとピンポイントで修正されていることがわかります（下図参照）。

そのため、この新しい LockBit2.0 の個体においては、ドメイン配下のクライアント端末上でスケジュールタスクによって EXE が正常に実行され、自動的に感染が発生することも確認されました。

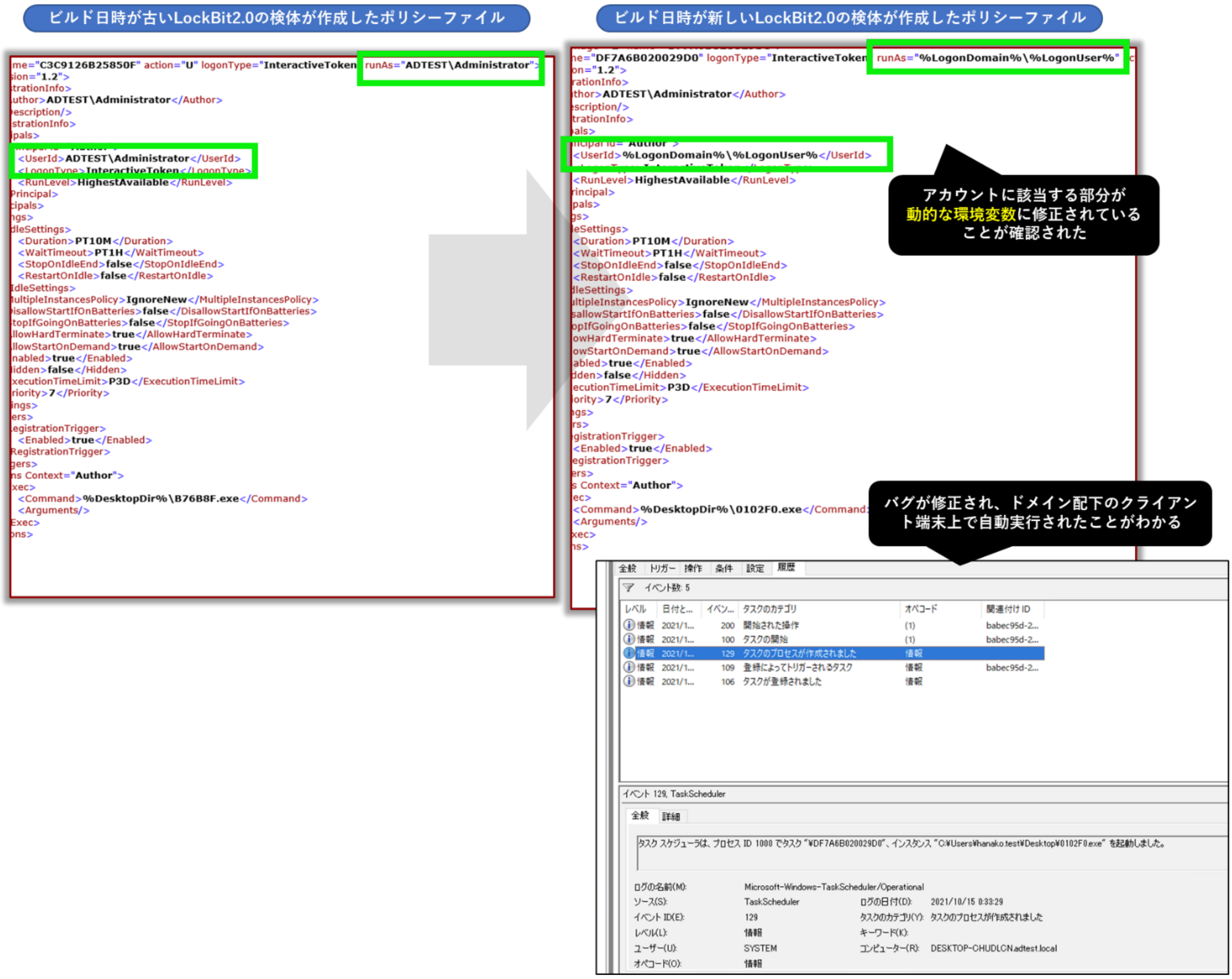


図 89　新しい LockBit2.0 の検体でタスクのバグが修正されていることが確認された様子

このように、LockBit2.0 の開発者はバグを随時修正しながら攻撃に使用していることが明確に確認できたため、検体の挙動における個体差（揺れ）があり、挙動にバグがあったとしても既に修正されたバージョンが出回っている可能性は高く、安心できる状況ではありません。

LockBit2．０ がグループポリシーで自身を拡散させる様子は動画に収録し以下の YouTube リンクで公開していますのでご覧ください。

https://www.youtube.com/watch?v=Co8mRfd0iPM

■デスクトップ壁紙の変更

LockBit2.0 は、感染した端末のデスクトップの壁紙を脅迫文が記載されたものに変更する挙動がありますが、他のランサムウェアにみられるように単純にあらかじめ用意された画像を取り出して使用するのではなく、感染端末のディスプレイサイズに合わせたデザインとなるように、わざわざ「Windows GDI+」（Graphics Device Interface）を用いて、一から画像を描画して壁紙を動的に生成します（下図参照）。

■単純に予め用意された壁紙画像を使うのではなく、感染端末のディスプレイサイズに合わせ、Windows GDI+(Graphics Device Interface)を用いて壁紙の画像を一から描画して生成する

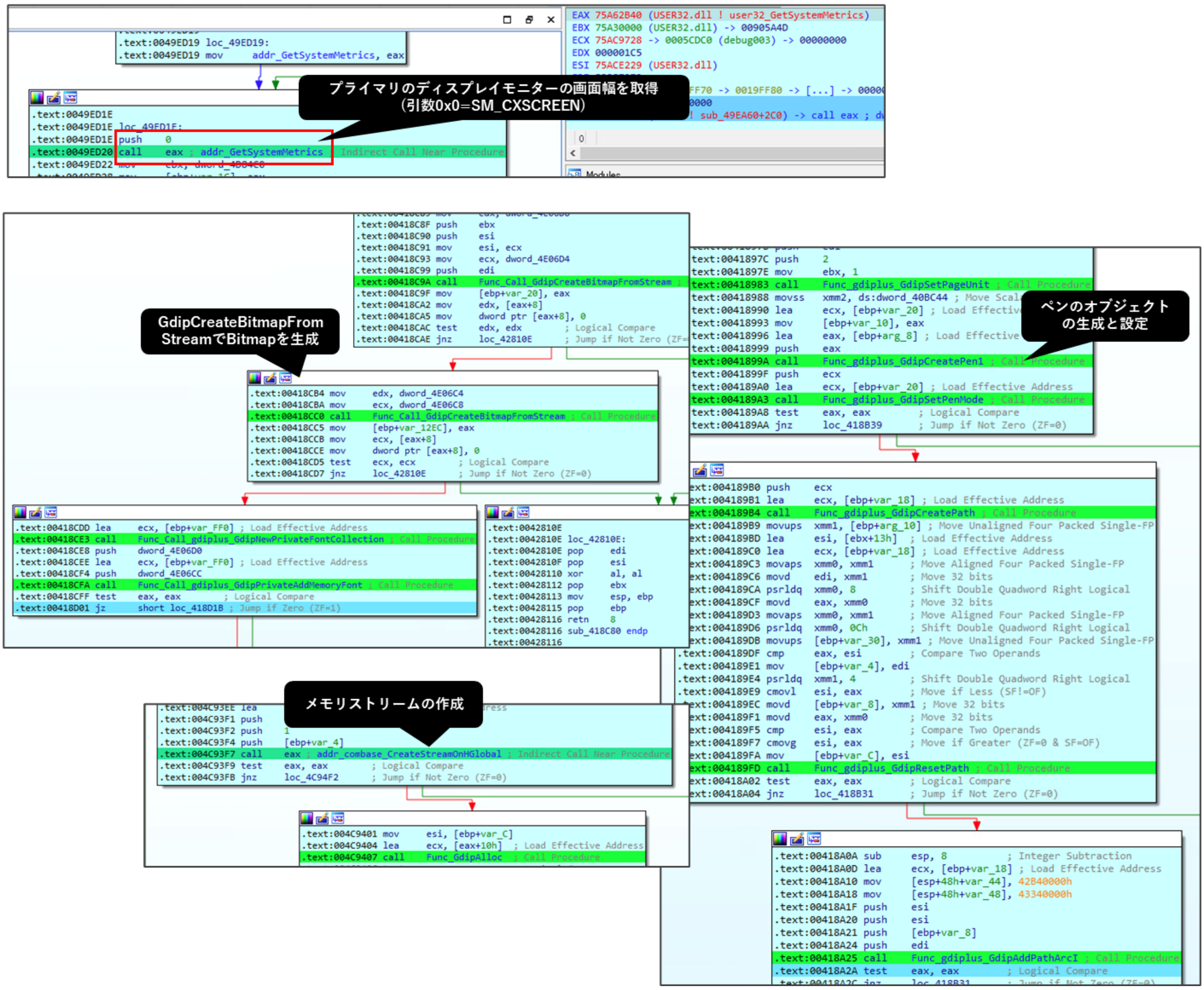


図 90　壁紙を一から描画して生成

以下のように壁紙に表示させる文字列も復号して用意し、Windows GDI+の API を用いてテキストとして描画します（下図参照）。

■壁紙に表示させる文字列を復号し用意、 Windows GDI+のAPIを用いて文字列としてテキストを描画

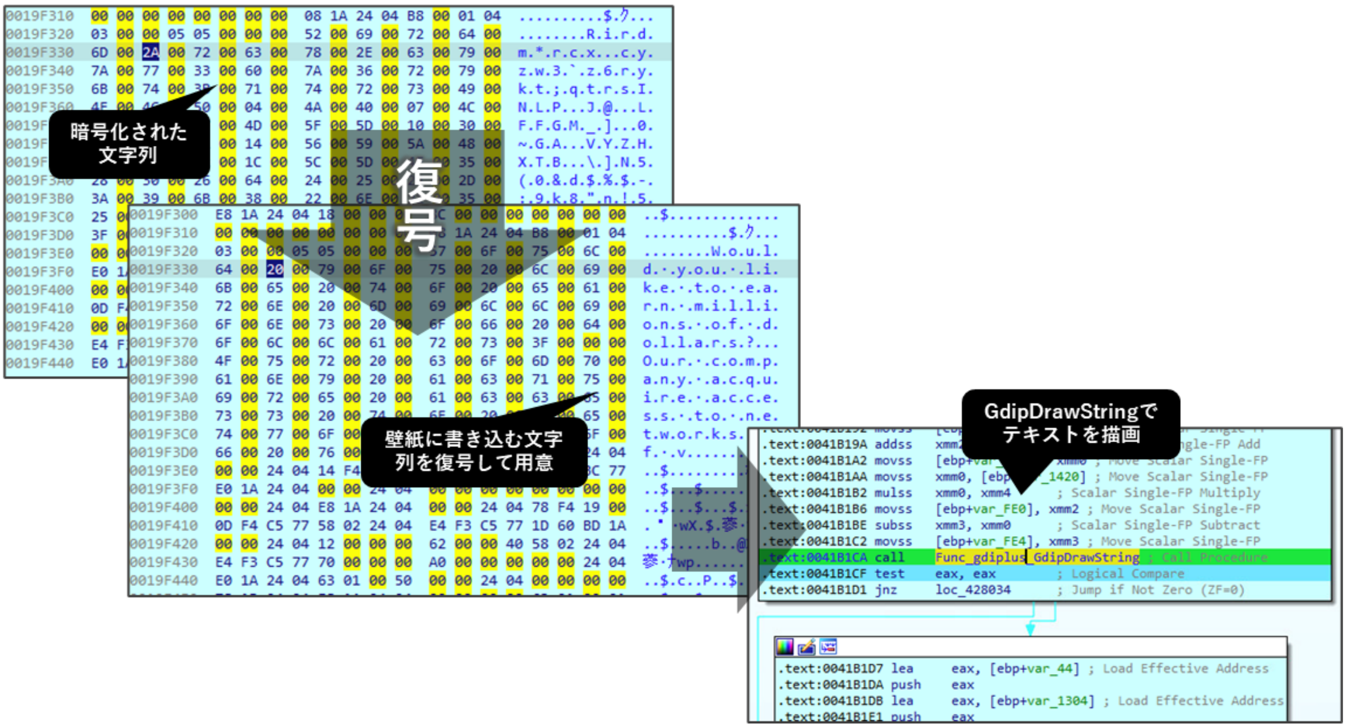


図 91　復号したテキスト文字列（脅迫文）を生成した画像に描画する処理

LockBit2.0 は一時フォルダ（%temp%）内に、上記の流れで生成した画像を一時ファイルとして作成します（下図参照）。

■壁紙に指定する画像ファイルの出力

図 92　壁紙に設定する画像の出力

作成した画像のフルパスを壁紙の設定に関わるレジストリ値を書き換えることで設定します。またその際、ディスプレイサイズに合わせて壁紙が見やすくなるように、いくつかの設定を行います（下図参照）。

■レジストリに壁紙の設定を書き込み

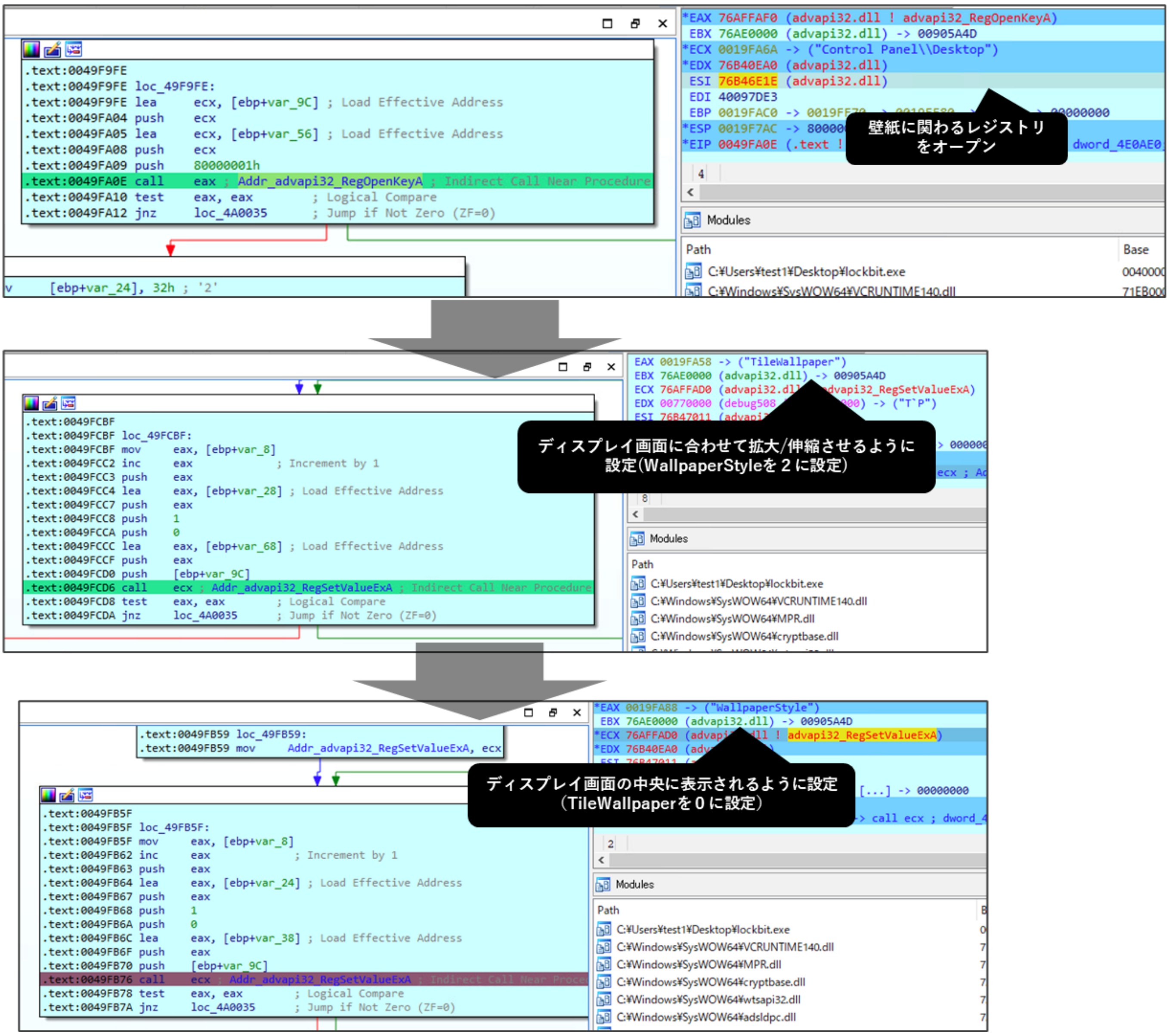


図 93　生成した画像パスなどをレジストリに設定する処理

上記の操作により、設定された壁紙に関わるレジストリ設定は以下となります（下図参照）。

■Lockbit2.0により改変された壁紙に関わるレジストリの設定

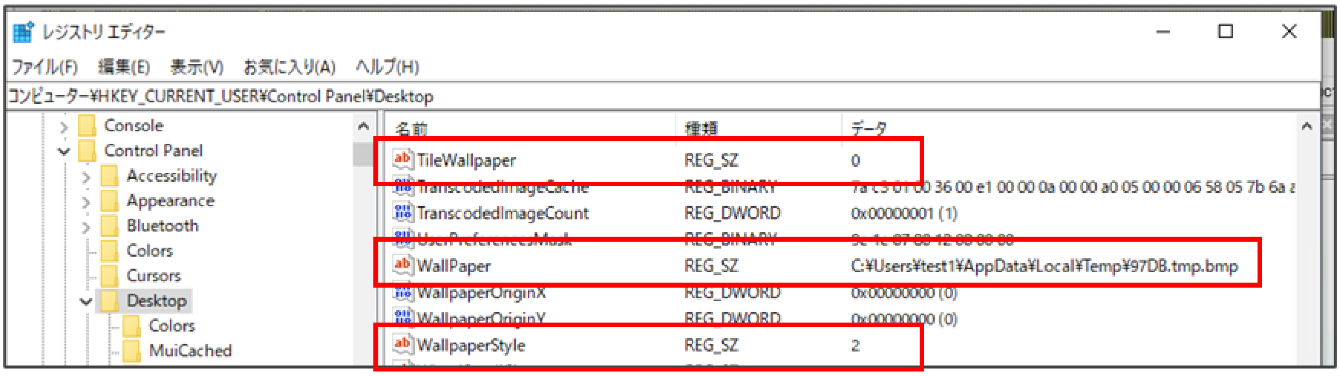


図 94 LockBit2.0 により改変された壁紙に関わるレジストリ設定

上記のレジストリを設定しただけではすぐに壁紙の変更がシステムに反映されないため、LockBit2.0 は SystemParametersInfoW 関数を使用することで即時壁紙の変更を反映させます（下図参照）。

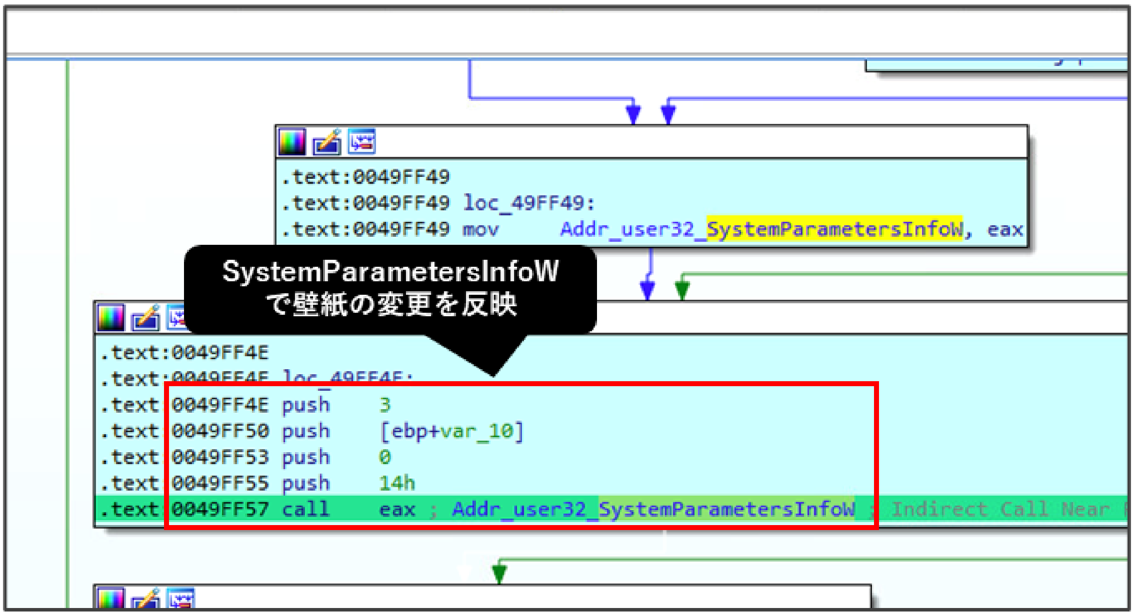


図 95 SystemParametersInfoW による壁紙変更の即時反映

以上により、動的に生成した壁紙がデスクトップに反映されます。

なお、このデスクトップの壁紙には、インサイダー（内部者）を募集する記載が書き込まれるタイプの検体が存在します（この壁紙を設定しない検体も存在します）。

インサイダーを募集する壁紙には「あなたも数百万ドルを稼ぎませんか？」と記載されており、この壁紙を見た人物に、どこかの会社の内部情報を知っている場合は攻撃者に提供すれば金銭を得られることを示唆する文章が並びます（下図参照）。

こうしてインサイダーを募集する壁紙を設定するという挙動は他のランサムウェアに見られないため、LockBit2.0 が持つ新規性の高い挙動の一つといえます。

これまでの RaaS においてはアフィリエイトなどが初期アクセスブローカーからの認証情報の購入に頼っていた側面もありますが、新しい攻撃手口のアイデアとしてインサイダーからの情報提供を初期アクセスのきっかけにしようとする企みが垣間見えます。この企みが成功した場合、攻撃者の侵入経路として非常に効果的な手段を提供する新たな脅威となる可能性があります。

■生成された画像がデスクトップ壁紙に反映された様子

図 96　インサイダーを募集する壁紙が設定された感染端末の様子

■脅迫文が記述されたテキストファイルの作成

LockBit2.0 は、暗号化したファイルが含まれるすべてのフォルダの中に、脅迫文が記述された「Restore-My-Files.txt」というテキストファイルを作成します。この脅迫文には、組織内のデータが盗まれてすでに暗号化されている旨と、身代金を支払わなければデータがダークウェブ上に暴露されるリスクがある点などが記述されています。

■メモリに暗号化された脅迫文を展開し、復号した上でテキストファイルとして生成する

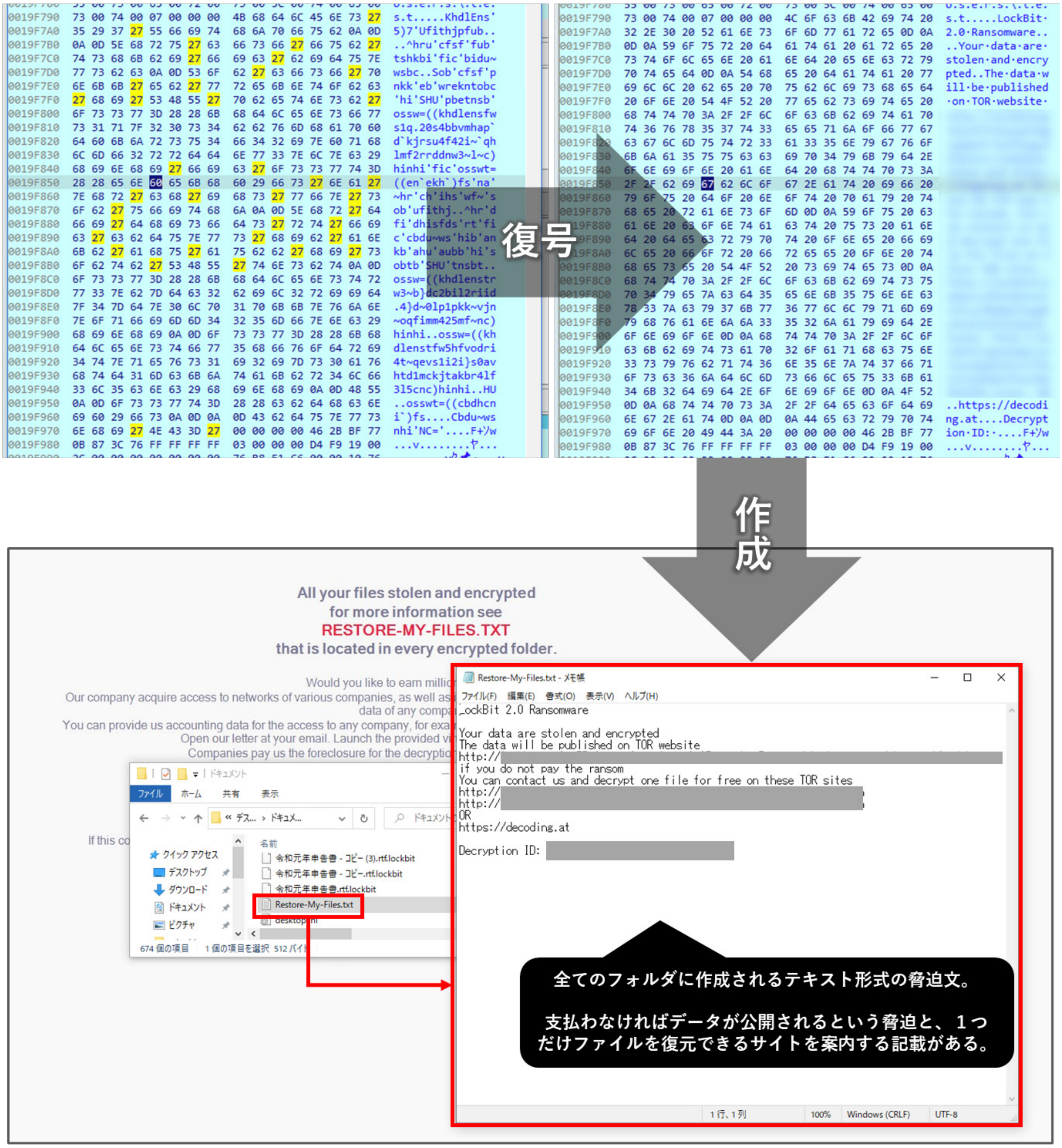


図 97　全てのフォルダに作成されるテキスト形式の脅迫文

■HTA ファイルによる脅迫文の表示

LockBit2.0 は、脅迫文を提示する別の方法として、デスクトップに HTML で記述されたアプリケーションである HTA（HTML Application）ファイルを作成し利用します（下図参照）。この HTA ファイルには”ファイルを復元するには攻撃者へコンタクトを取るしか道はない”という趣旨のコメントと、どのような手順でコンタクトを取れば良いかが記載されています。

■感染後に画面に表示させる「HTAファイル」を生成

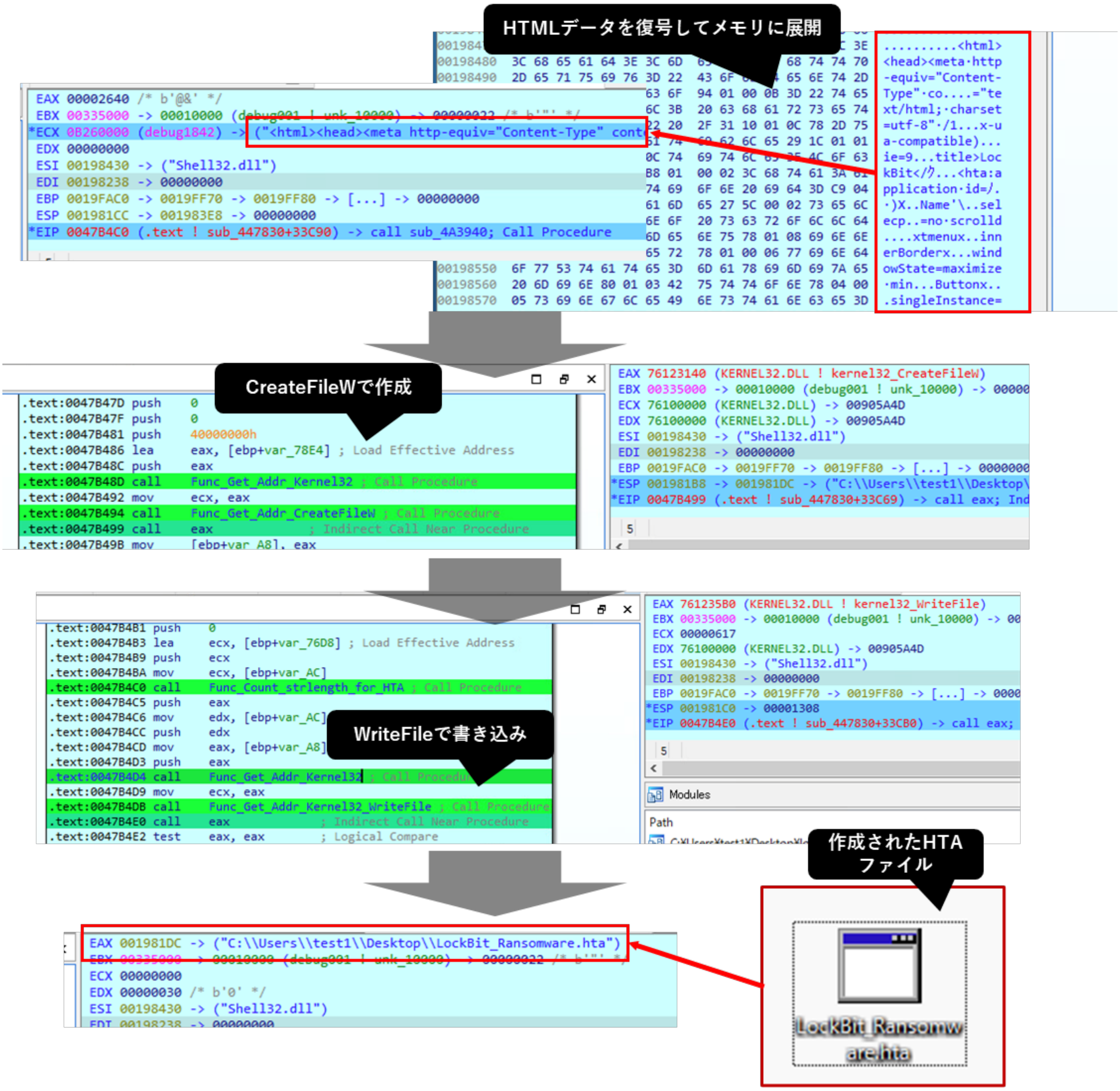


図 98　デスクトップに作成され脅迫文を表示する HTA ファイル

また、作成した HTA ファイルが端末の再起動時に常に自動的に起動するようにレジストリの Run キーに設定します（下図参照）。

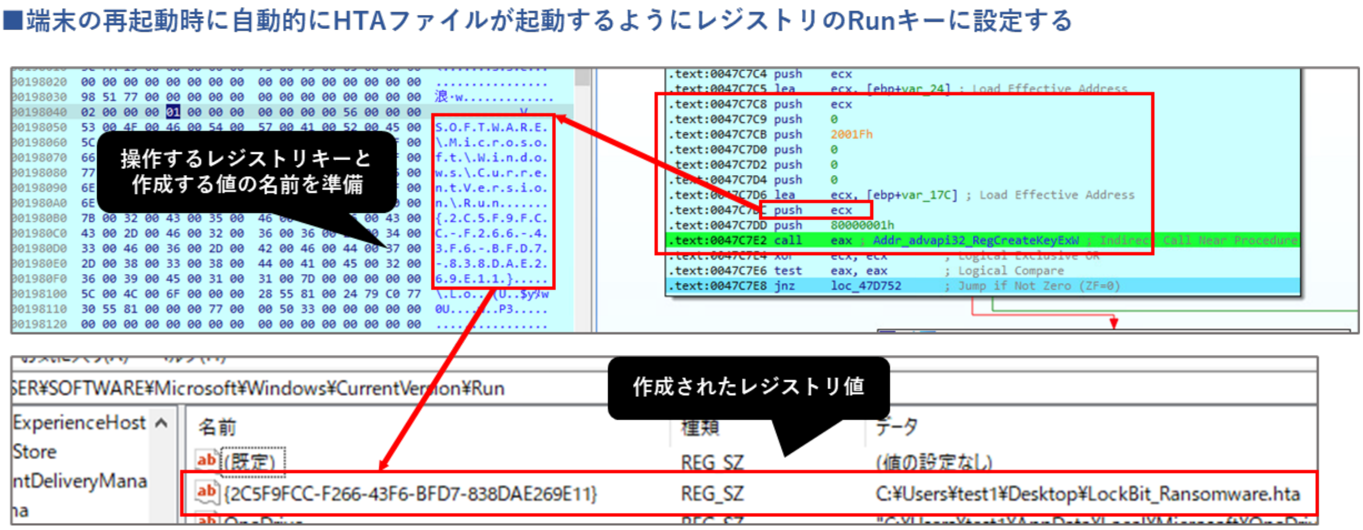


図 99　HTA ファイルを自動起動に関わるレジストリに設定する様子

続いて、「.lockbit」という拡張子を持つファイルをユーザが開いた場合、該当の HTA ファイルが自動的に開くようにレジストリに設定を行います（下図参照）。

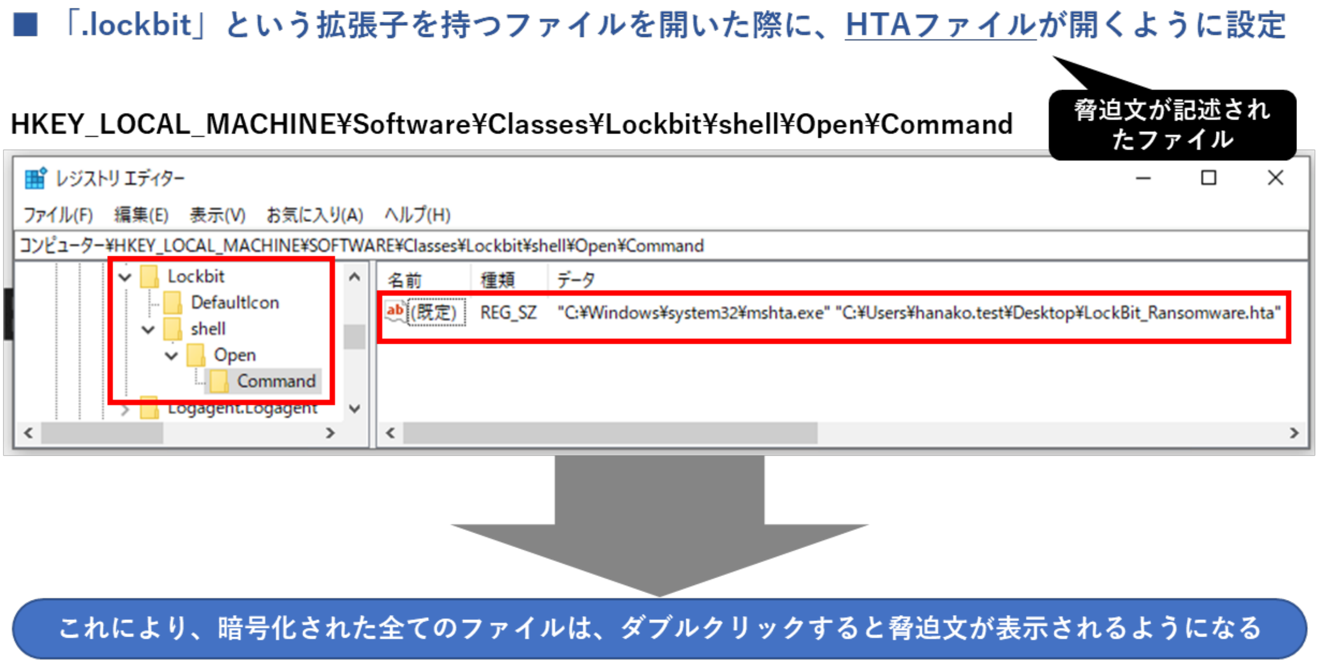


図 100　暗号化されたファイルを開くと HTA ファイルが開くように設定されたレジストリ

そして、ShellExecuteW で HTA ファイルを起動し表示させせます（下図参照）。

■ShellExecuteWでHTAファイルを起動し表示させる

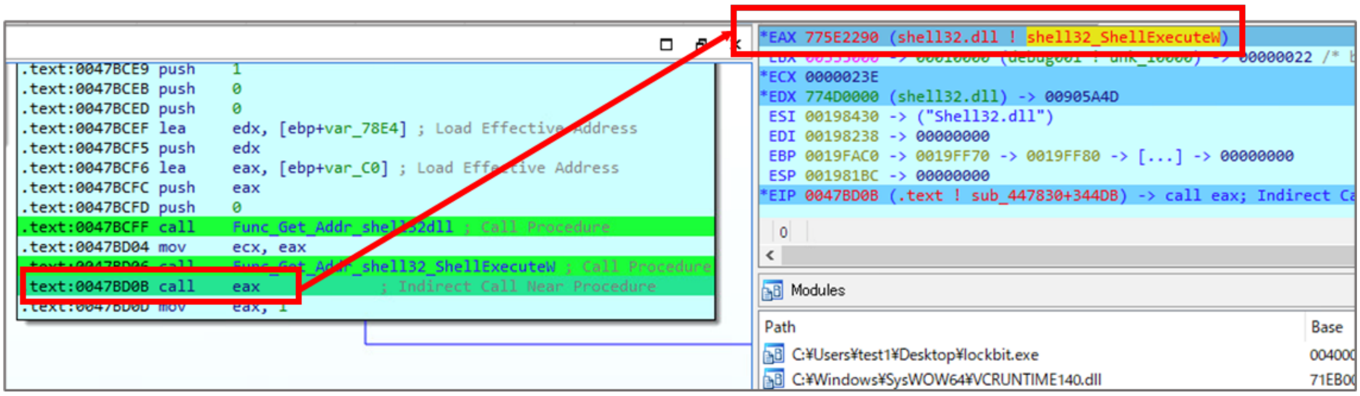


図 101　感染後に HTA ファイルを自動的に開く処理

これにより、以下のように自動的に HTA ファイルが開かれます。画面サイズいっぱいに開かれるため、一見デスクトップの壁紙に見えますが、これは HTA が表示するウインドウです（下図参照）。なお、このウインドウは[Alt]+[F4]キーで閉じることができます。

■感染後に自動的に実行されたHTAファイルにより表示されるウインドウ（これはデスクトップ壁紙ではなく、壁紙は別途異なる記載が掲載されている）

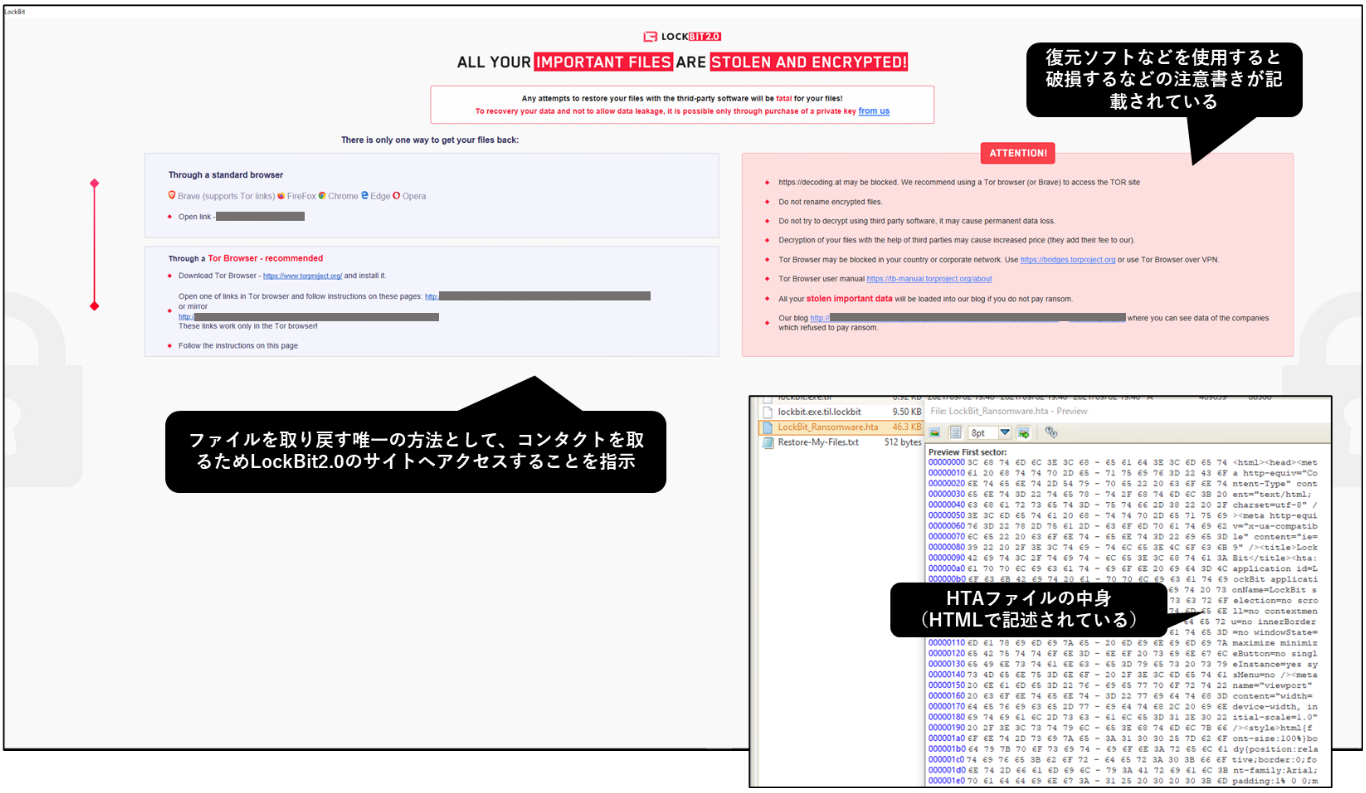


図 102　感染後にモニター上に表示される HTA ファイルが生成するウインドウ

■プリンタを利用した脅迫文の物理的な印刷（約一万枚）

LockBit2.0 の特徴的な挙動の一つが、プリンタを利用した脅迫文の物理的な「印刷」です。
印刷を行うランサムウェアの出現は初めてではなく Egregor という別のランサムウェアが過去に実装してはいたものの、ランサムウェア全体から見るとほとんど存在せず非常に珍しい挙動といえます。

LockBit2.0 は「あなたのデータは盗まれた上に暗号化された。身代金を支払わなければデータが公開される」という文言が並んだ脅迫文を、感染端末からアクセス可能なネットワークプリンタを含む全てのプリンタで印刷しようとします。

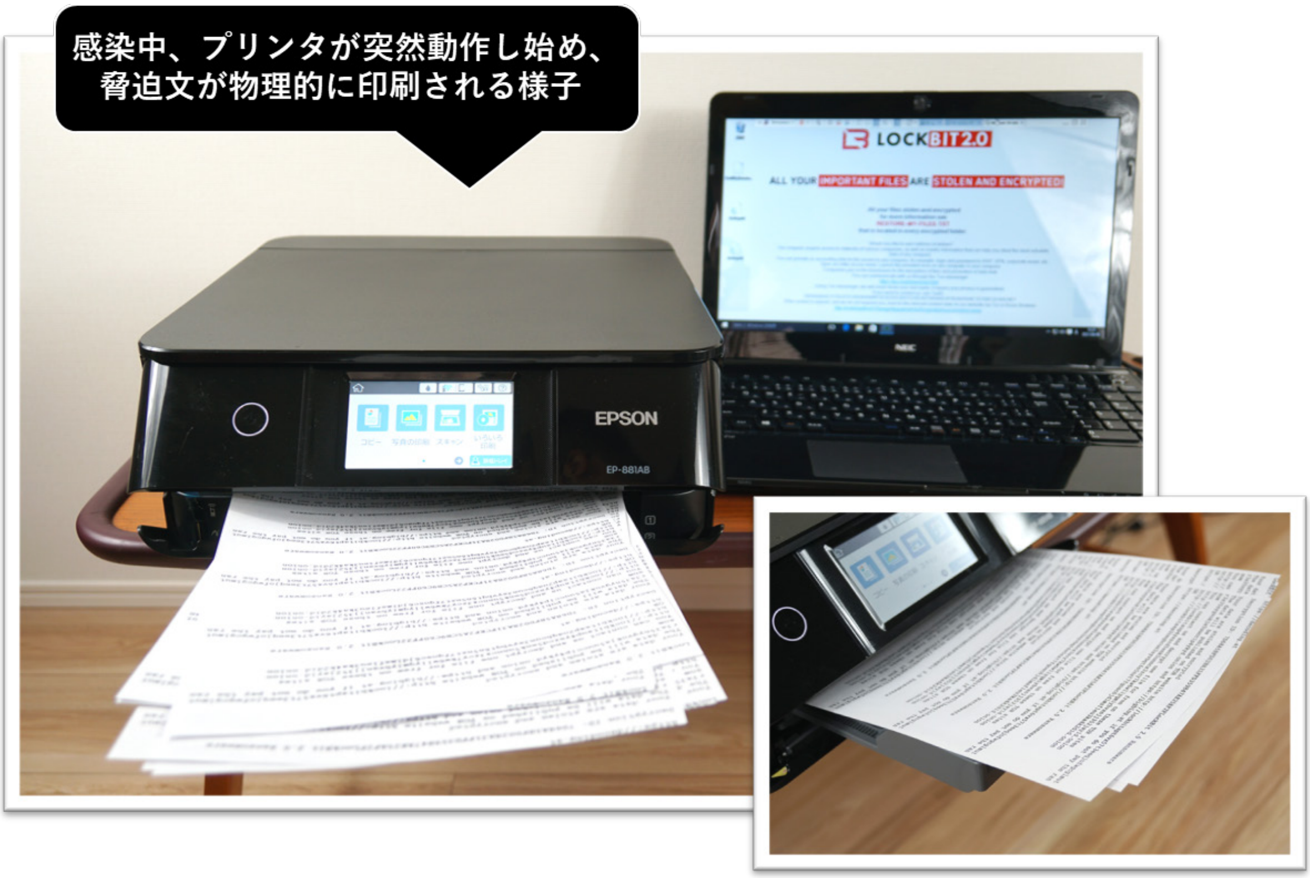


図 103　LockBit2.0 に感染し脅迫文をプリンタで印刷される様子

<余談>

以下の記事で解説していくように、ランサムウェアの解析結果を通した理論上においては、プリンタの印刷機能が特定され印刷機能を持っていることが把握できたのですが、実際にプリンタで印刷されるのか実検証したところ、想像以上に行き詰りました。

また、こうした物理的なデバイスに絡む挙動は仮想解析環境+仮想プリンタなどでは実機とは異なった結果となる場合があり、実際に「仮想プリンタ」を用いた検証では LockBit2.0 に感染させても印刷が発生しませんでした。
そのため、プリンタの実機を使用した検証を行うことにしました。プリンタ実機を感染端末に接続すればすんなり印刷されるものと想定していましたが実際には印刷が発生せず、複数台のプリンタを用いての数日間をまたいだ長い検証の結果、LockBit2.0 が送信する印刷データに対応しているプリンタ実機と、対応していないプリンタ実機があるという結論に達しました。
他社の解析記事においては LockBit2.0 の印刷機能に触れているもの自体が少ないですが、記載されている場合も「LockBit2.0 は脅迫文を印刷する」という程度にしか触れられておらず、実際には印刷されるプリンタとされないプリンタがあるという新たな事実が今回の検証を通して判明したことになります。
静的解析でコードを見ただけでは「印刷する」という挙動は判るものの、印刷されないプリンタが存在するという事実は実際に実機を用いて検証しないとわかり得ない結果であるといえます。以上の検証結果から、実際の被害組織においても印刷が発生した環境と、印刷が発生しなかった環境があるものと考えられます。

LockBit2.0 は、プリンタ関連の API を使用するため、それらの API を提供する「Winspool.drv」をロードします（下図参照）。

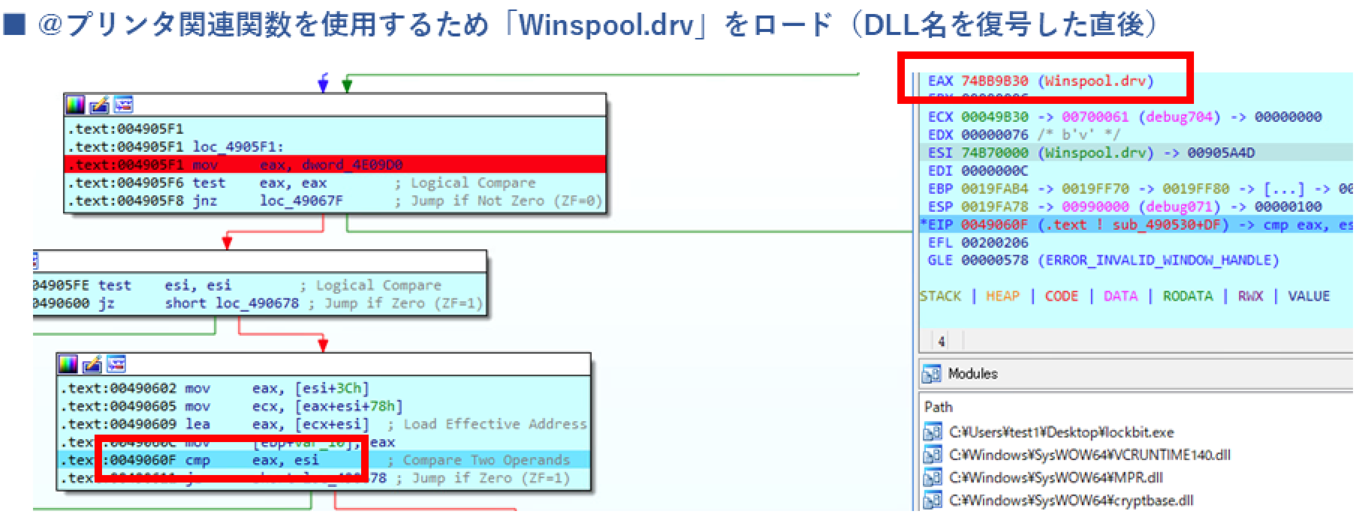


図 104　Winspool.drv をロードする処理の一部

その後、EnumPrinters 関数で感染端末からアクセス可能な全てのプリンタを列挙し取得します（下図参照）。

■ 「EnumPrinters」で感染端末上で利用可能な全てのプリンタを列挙

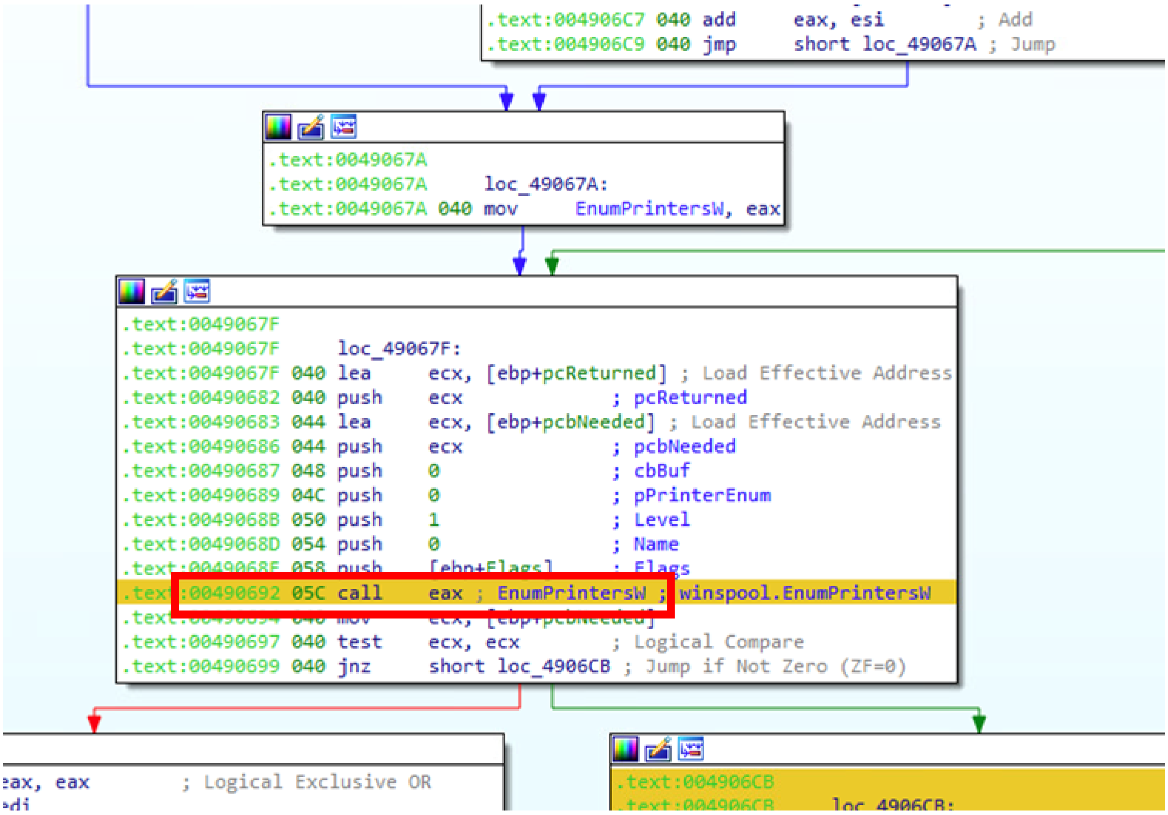
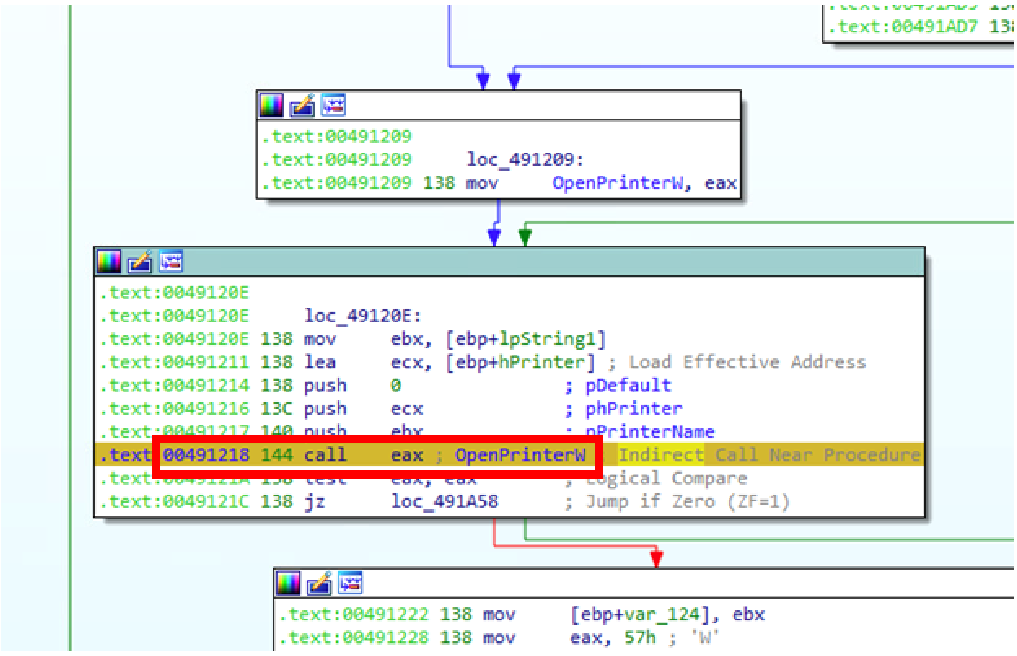


図 105　感染端末上で利用可能な全てのプリンタを列挙する処理

見つかった全てのプリンタに対し、それぞれ OpenPrinter 関数で開き操作可能な状態にした上で、StartDoc Printer 関数でページの印刷を通知します（下図参照）。

■ 「OpenPrinter」で指定したプリンタをオープン



■ 「StartDocPrinter」で指定したプリンタにページの印刷を通知

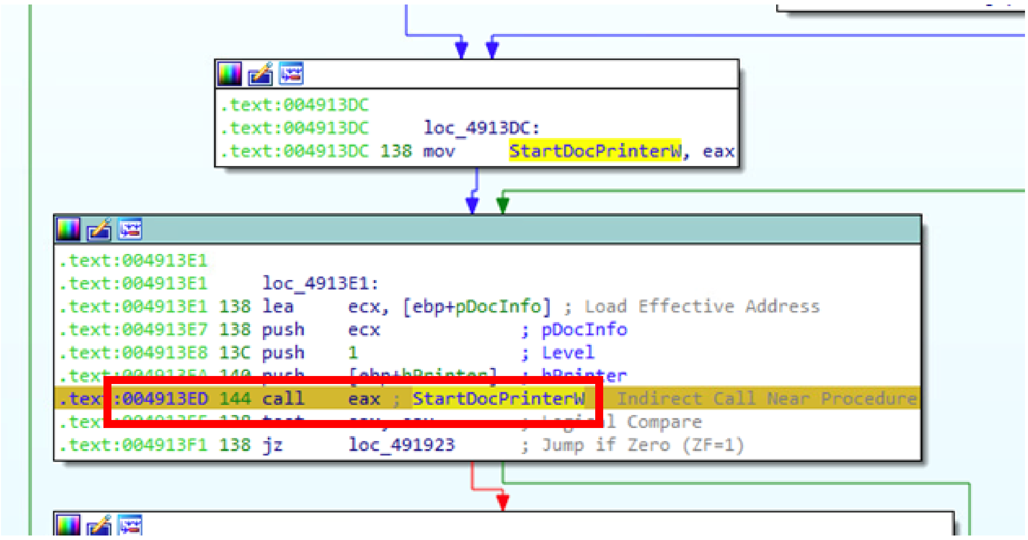


図 106 見つかったプリンタを開き印刷するページを通知する処理

その後、プリンタで印刷するための脅迫文の文字列を復号しメモリ上に展開します（下図参照）。

■ 全てのプリンタに印刷する脅迫文の文字列を復号

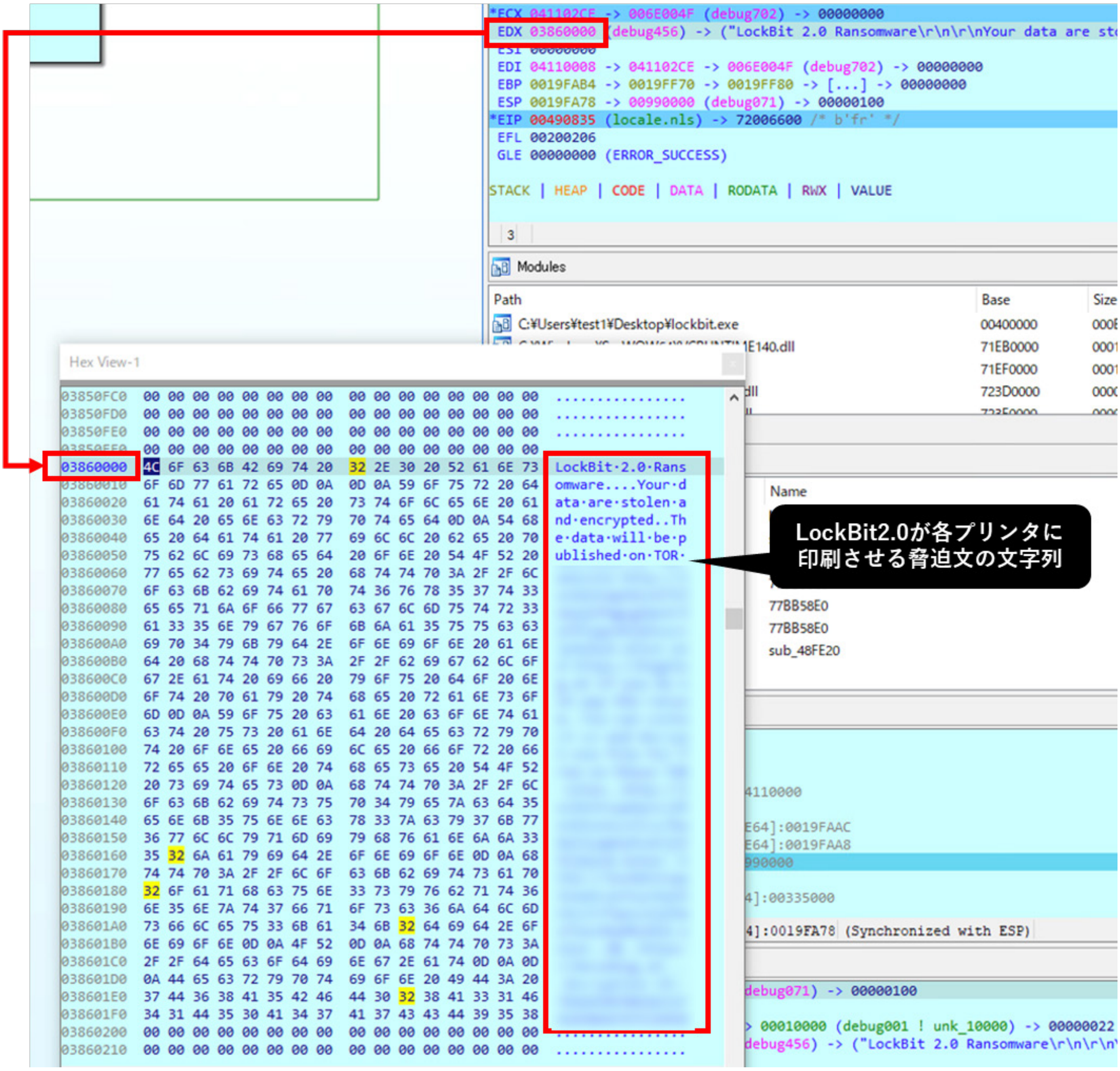


図 107　全てのプリンタに印刷させる脅迫文の文字列を復号する処理

そして、WritePrinter 関数で指定したプリンタに上記のデータを送信し印刷を指示します（下図参照）。

■ 「WritePrinter」で指定したプリンタにデータを送り印刷

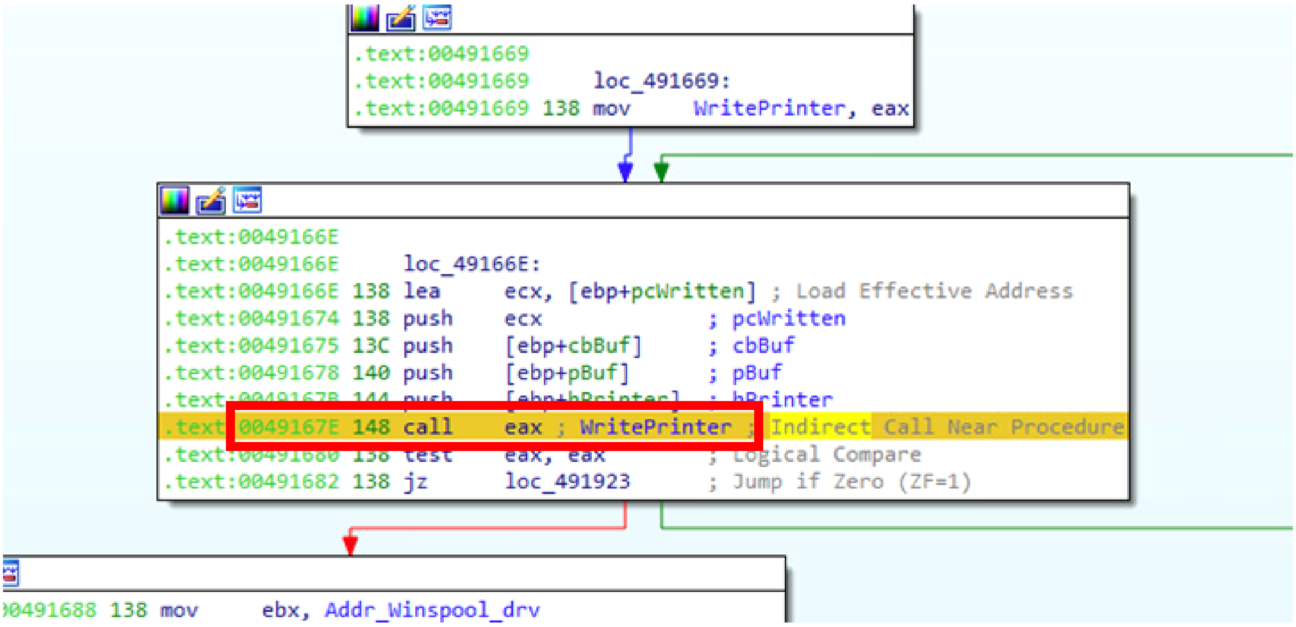


図 108　指定したプリンタにデータを送る処理

1 ページ分のデータの送信が完了したのち、EndDocPrinter 関数で指定されたプリンタの印刷ジョブを終了した上で、ClosePrinter 関数でプリンタオブジェクトを閉じます（下図参照）。

■ LockBit2.0の印刷処理の終了部分

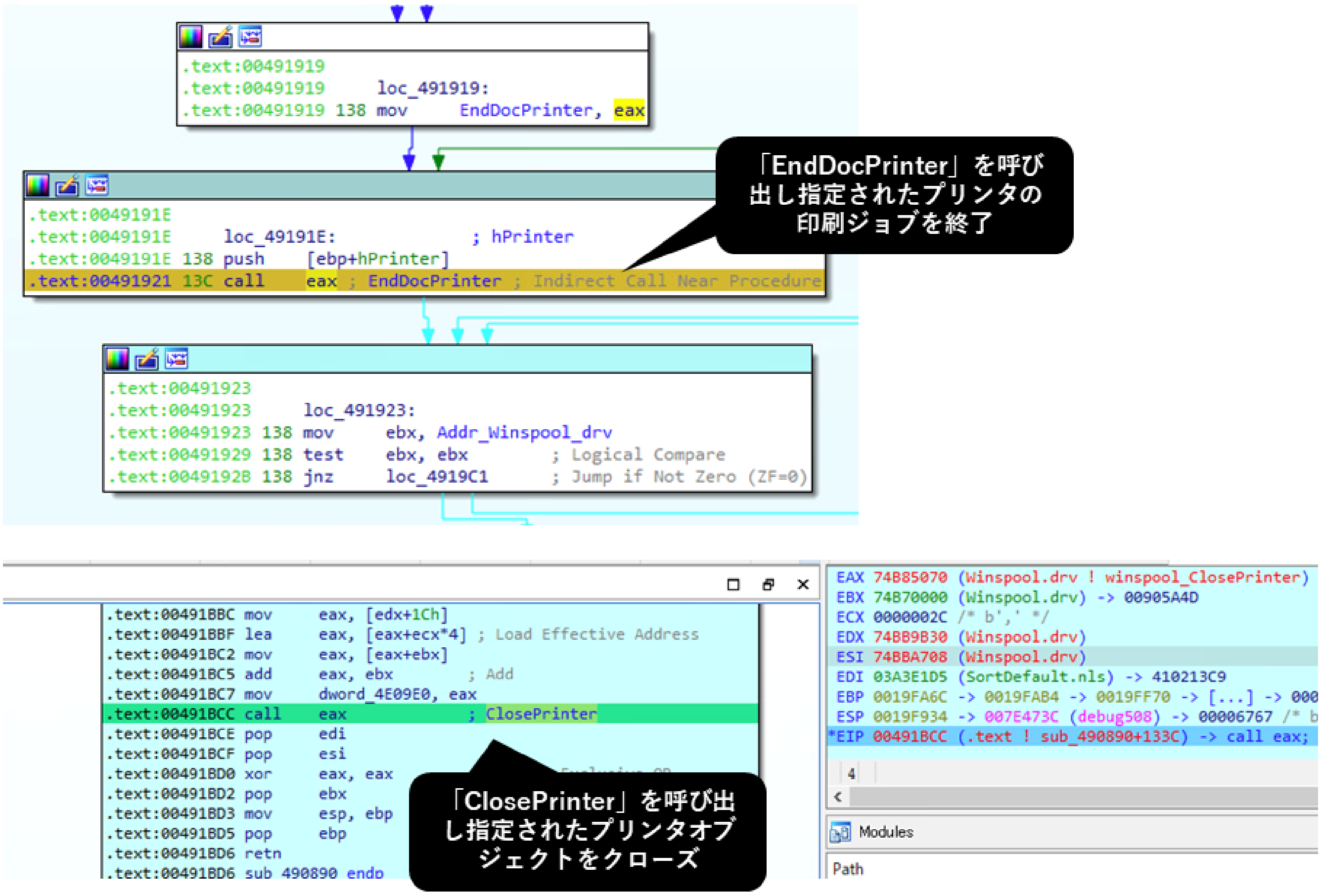


図 109 印刷終了の処理

以上が 1 ページ分の処理ですが、LockBit2.0 はこの処理を約 1 万枚（9,999 ページ）分繰り返します（下図参照）。

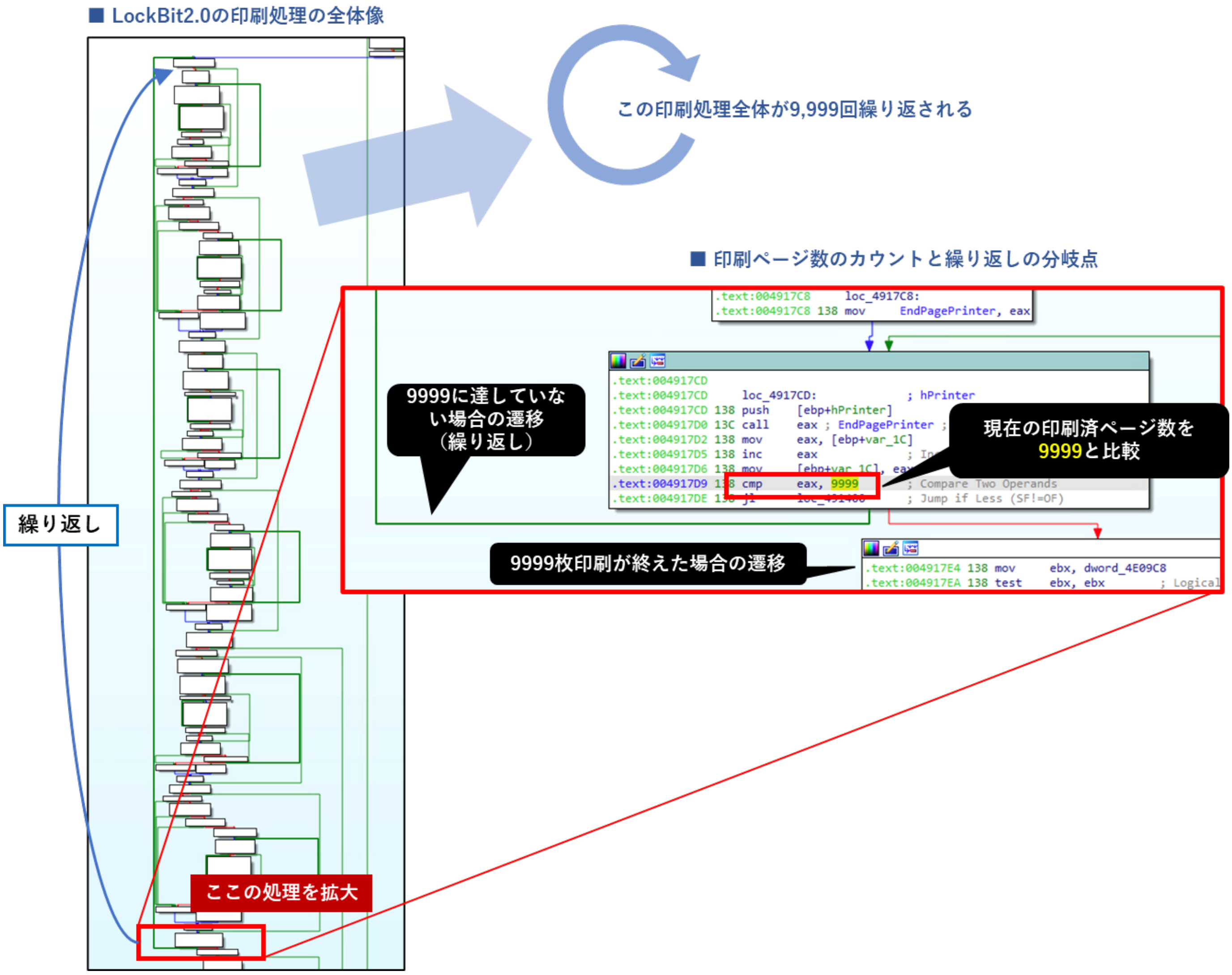


図 110　約一万枚印刷されるようにこれまで解説した一連の印刷処理を繰り返す

つまり、プリンター台当たり 9,999 枚印刷するよう指定されており、この操作を感染端末からアクセス可能なネットワークプリンタを含む全てのプリンタに対して行うため、途中でプリンタを急いで停止しない限り、延々と大量に脅迫文が印刷され続けてしまうことになります。

従来のランサムウェアが行ってきたようにモニター上で提示する脅迫文だけでなく、物理的なデバイスを用いて脅迫するというこの手法は、やはり被害組織に与える恐怖やインパクトはより大きいものになるだろうと想像します。

LockBit2.0 が実機プリンタで脅迫文を印刷する様子は動画に収録し、以下の YouTube で公開していますので併せてご覧ください。

https://www.youtube.com/watch?v=zfh-FqPdXI4

■インシデント対応や復旧作業を妨害する処理

LockBit2.0 は以下のコマンドを cmd の引数として実行することで、システムのバックアップ（ボリュームシャドウコピー）や、イベントログを削除し、インシデント対応や復旧作業を妨害する処理を行います。

なお、以下のうち上から４つ目のコマンドには実効性がありませんが、他の一部のランサムウェアやマルウェアでも同コマンドが確認されているため、開発者が参照した際に記述ミスのままコピーして使用しているものと推測されます。

/c vssadmin Delete Shadows /All /Quiet ➡全てのボリュームシャドウコピーを削除
/c bcdedit /set {default} recoveryenabled No ➡スタートアップ自動修復機能の無効化
/c bcdedit /set {default} bootstatuspolicy ignoreallfailures ➡すべての起動エラーを無視し Windows を開始
/c wmic SHADOWCOPY /nointeractive ➡ボリュームシャドウコピーの削除（記述ミスの可能性）
/c wevtutil cl security ➡イベントログ(security)の削除
/c wevtutil cl system ➡イベントログ(system)の削除
/c wevtutil cl application ➡イベントログ(application)の削除

図 111 LockBit2.0 がインシデント対応や復旧作業を妨害する処理

■フォレンジックを意識した自身の抹消処理

LockBit2.0 は全ての活動が終了すると、自身の EXE ファイルを削除しますが、その際、単純な削除（Delete）ではなく、LockBit2.0 の EXE ファイルが存在するハードディスクの位置の先頭から 512KB 分のハードディスク領域を直接 0 で埋め尽くすことでファイルを破損させ完全に破壊した上で削除します。具体的には、fsutil コマンドを使用し「Data offset」と「length」によってサイズ範囲を指定した上で 0 埋めした後 Del コマンドによってファイルを削除します。（※なお、これらのコマンド処理は外部プロセスである cmd.exe に依頼するため、実際には LockBit2.0 のプロセスが終了した後で実施されます）

このことから、フォレンジック調査時にランサムウェア本体が正常に復元されないよう意識し、念入りに自身を抹消していることがわかります（以下図参照）。

■自身のEXEファイルを削除する際、単純な削除（DELETE）ではなく、
ファイルの先頭から512KB(523,288 byte)分のハードディスク領域を直接0で埋め尽くすことで完全に破壊した上で削除する。

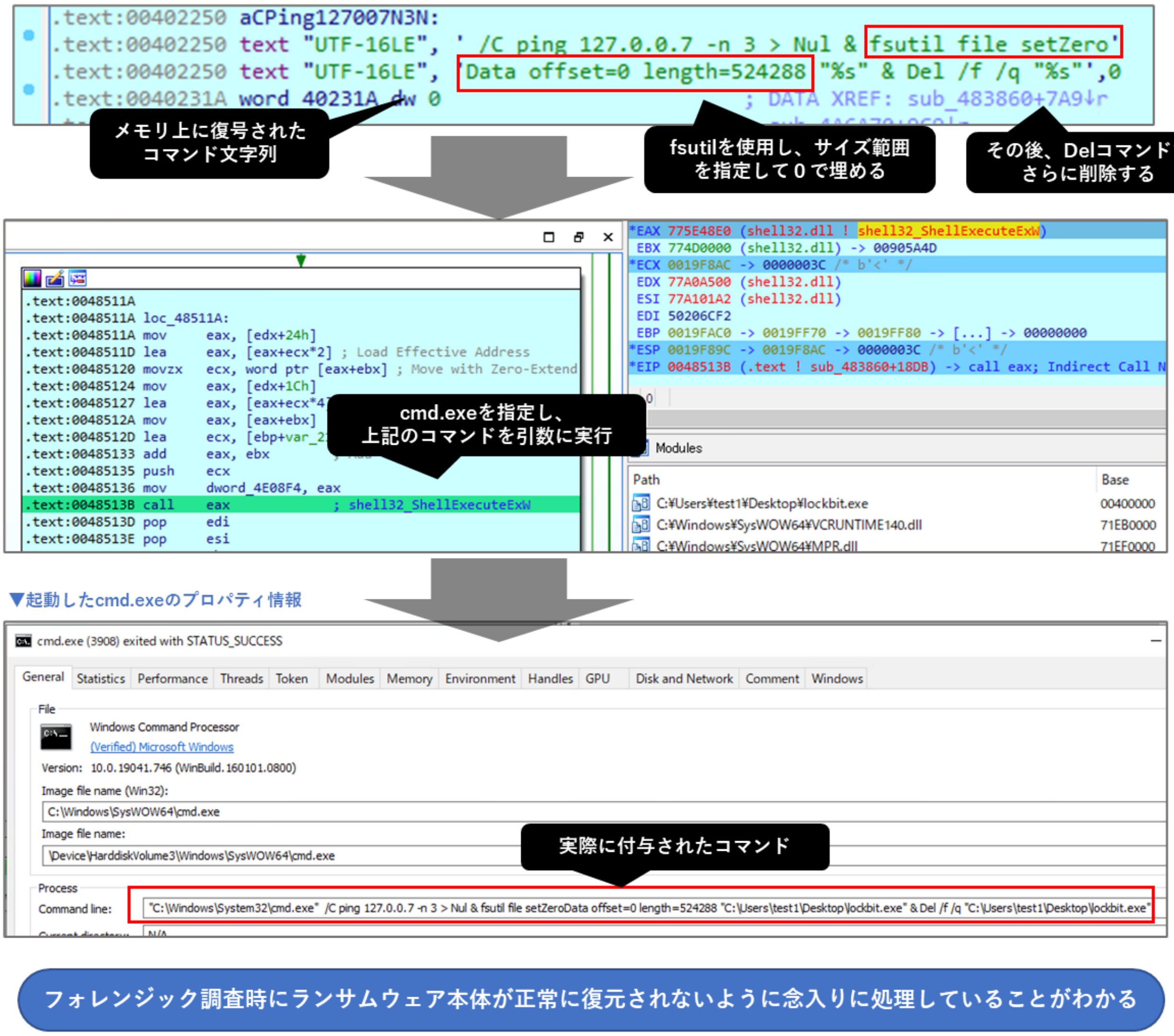
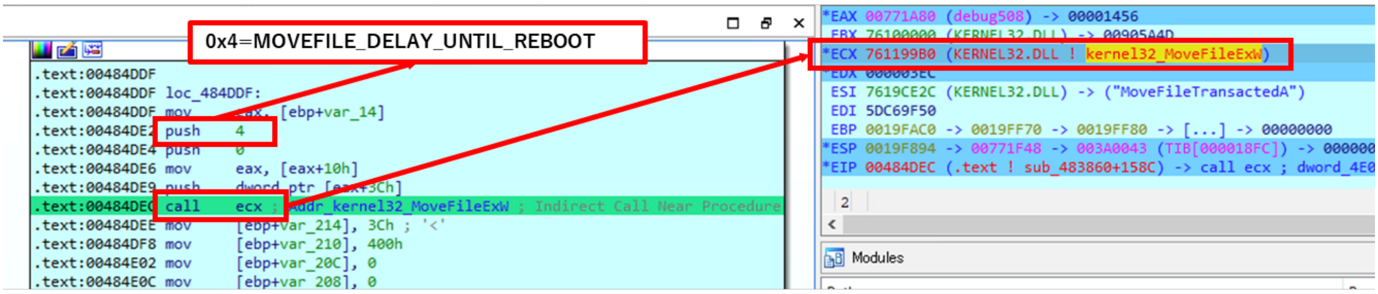


図 112　ハードディスク上から自身のファイルの一部を破壊した上で消去する LockBit2.0

加えて、上記の Del コマンドが失敗した場合に備え、Windows の再起動時に自動的に OS によって LockBit 2.0 の EXE ファイルが削除されるよう、システムに予約設定を行います。具体的には、MoveFileExW 関数に MOVEFILE_DELAY_UNTIL_REBOOT の引数を渡すことでシステムに予約を行います（以下図参照）。

■さらに、Windowsの再起動時に自動的にLockBit2.0のEXEをOSが削除するようシステムに予約設定



cmdによる自身の削除コマンド処理が失敗した場合を想定して執拗に自身を削除しようにしていることがわかる

図 113　システム起動時に自動的に OS によって削除されるように設定する LockBit2.0

これらの 2 つの方法で執拗に、不要になった自身の EXE を抹消しようとするランサムウェアは珍しい部類であるといえ、復旧作業でランサムウェアを入手させないように入念な隠蔽工作を行おうとする意図が感じられます。

これら一連の処理を行った後、LockBit2.0 は ExitProecss により自身の処理を終了させます（以下図参照）。

■それらの一連の処理を行なった後、最後に自身を終了する

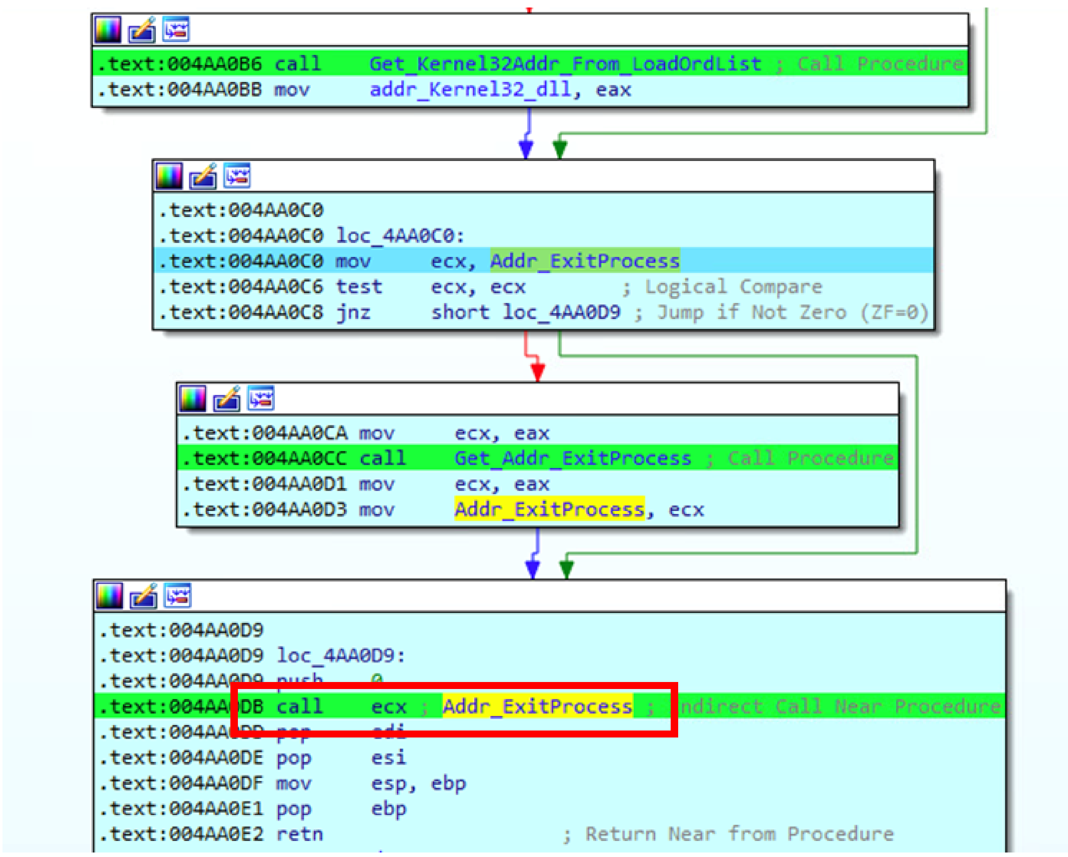


図 114　全ての処理を終えた後に自身を終了する処理

LockBit2.0 の挙動は以上となります。
以降では、LockBit2.0 が実装しているその他の機能をご紹介します。

■D&D（drag and drop）に対応

LockBit2.0 の EXE ファイルは、ファイルやフォルダの D&D(drag and drop)に対応しています。この点に触れた解析記事は確認した範囲においては存在せず、世界的にも知られていないようです。

LockBit2.0 は起動した際、CommandLineToArgW 関数を使用し実行引数の数が「2」であるかどうかをチェックします。引数の数が「2」である場合は D&D(drag and drop)の処理に遷移し、指定されたファイルまたは指定されたフォルダ内にあるすべてのファイルに対する暗号化処理のみを行います（以下図参照）。

つまり、LockBit2.0 は、自動的に動作するランサムウェアとしてだけでなく、攻撃者が指定した対象（ファイルやフォルダ）のみを暗号化したい、というシチュエーションでも活用できるよう、「暗号化ツール」として利用できるようにも開発されていることがわかります。

■ ファイルやフォルダのD&D(drag and drop)による暗号化ツールとしての利用用途にも対応している

図 115　D&D に対応した LockBit2.0 の EXE 本体

■LockBit2.0 が隠し持つ GUI(隠しウインドウ)の発見

冒頭の表層解析で解説した通り、LockBit2.0 の EXE が GUI アプリケーションとしてビルドされていたのにも関わらず、一般的な表層解析の範囲では GUI コンポーネントなどのリソースデータが確認できなかったことが筆者は気がかりになっており、その観点で LockBit2.0 の起動中のプロセスを念入りに調査しました。

その結果、起動中のプロセス内には表層解析では見えなかったリソースが出現しており、button や ListView などの GUI コンポーネントを持った非表示の”隠しウインドウ“が作成されていることを発見しました。隠しウインドウは「LockBit2.0 Ransom」という文字列を持っており、「LockBit_2_0_Ransom」というウインドウクラスを持つことがわかります（以下図参照）。

■ LockBit2.0のプロセスに隠されたGUIコンポーネントを発見

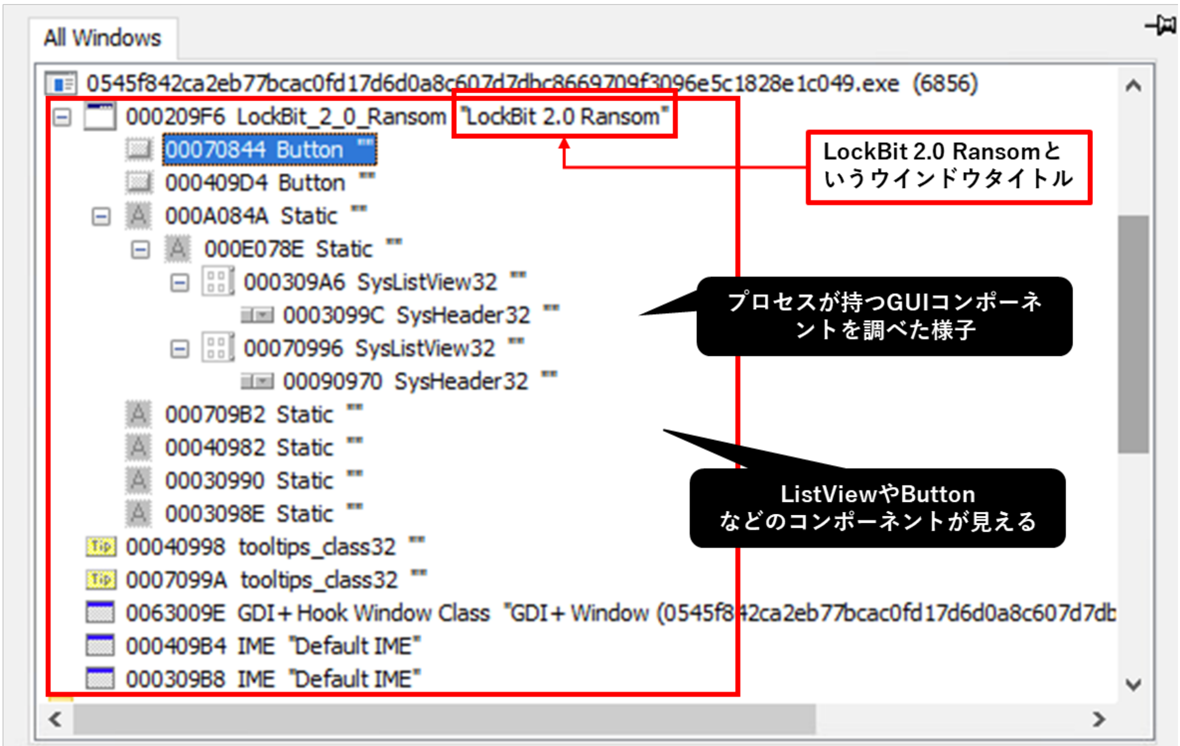


図 116　LockBit2.0 のプロセスに隠された GUI コンポーネントの発見

また、該当の隠しウインドウの状態を調べると Style が「4CF0000」(hidden,enabled)となっていることから、ウインドウの作成はされているが非表示になっているということがわかります（以下図参照）。

■ ウインドウが「隠しウインドウ」（hidden,enabled）になっていることが確認された

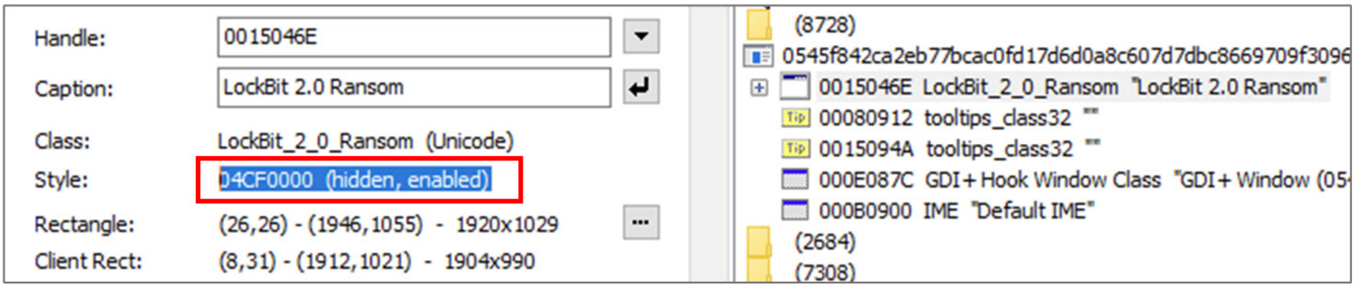


図 117　ウインドウが隠し状態に設定されている様子

この結果から改めて静的解析したところ、確かに CreateWindowExW 関数でウインドウを生成している処理が確認されました（以下図参照）。

■ 確かにCreateWindowExWでウインドウを生成している処理がある

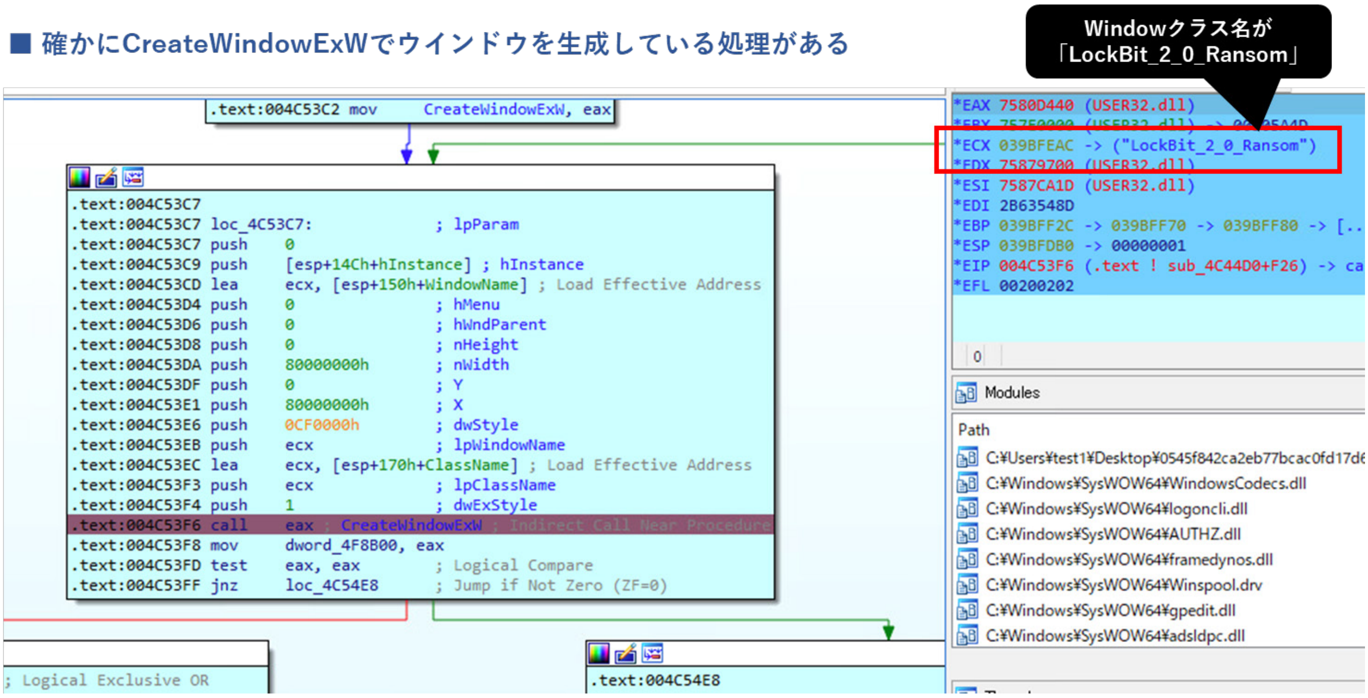


図 118　LockBit2.0 がウインドウを生成する処理

また、作成したウインドウを ShowWindow 関数で表示する際、引数として非表示を意味する「0」が渡されていることも確認できました（以下図参照）。つまり、ウインドウが生成されてはいるものの非表示になっているという先ほどの動的解析結果と一致することがわかります。

■ ShowWindowの引数が「0」（非表示）になっていることも確認された

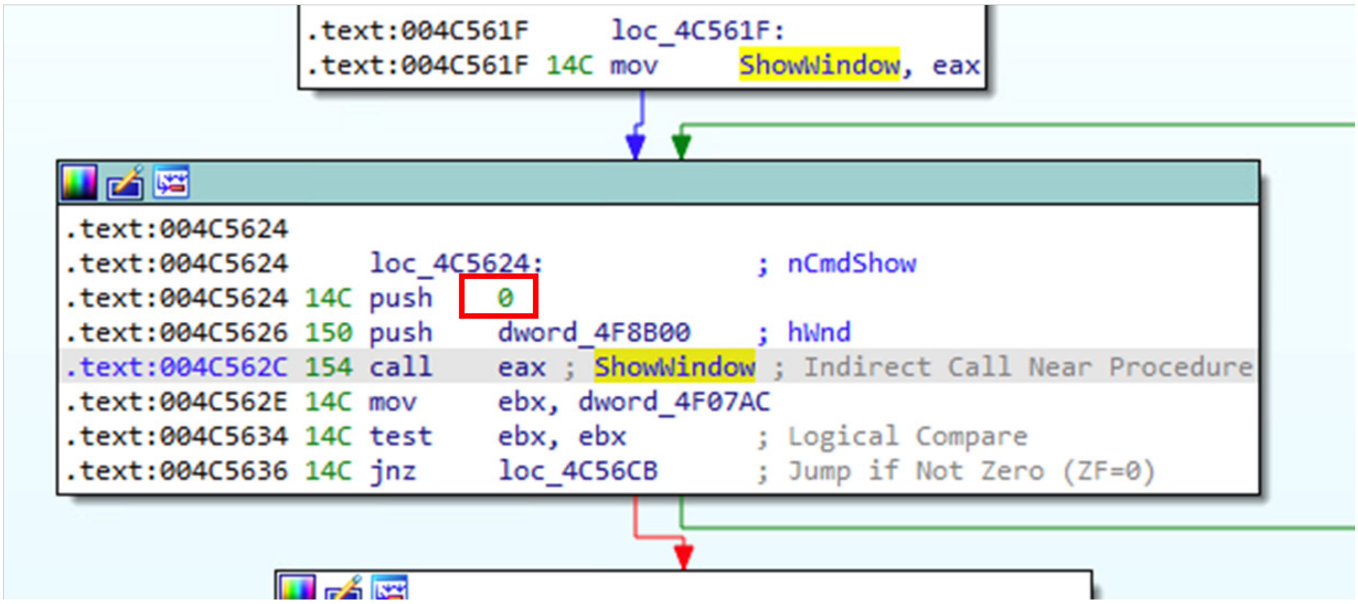


図 119　生成したウインドウを非表示状態で表示させる処理

そのため、非表示となっている隠しウインドウを無理やり表示させるよう LockBit2.0 ランサムウェアのバイナリを変更したところ、これまで世界的にも公に知られていなかった LockBit2.0 の隠しウインドウが表示され姿を現しました（以下図参照）。

表示されたウインドウは、複数のタブを持ち GUI コンポーネントが整然と並ぶグラフィカルなインターフェイスとなっており、感染端末上のドライブ一覧や現在暗号中の状況がリアルタイムに表示される様子がわかります。また、右クリックメニューも用意され、暗号化処理の停止や再開などが選択できるようになっています。

■ バイナリを変更し強制的に表示させた結果、出現したLockBit2.0の「隠しウインドウ」（Statisticsタブ）

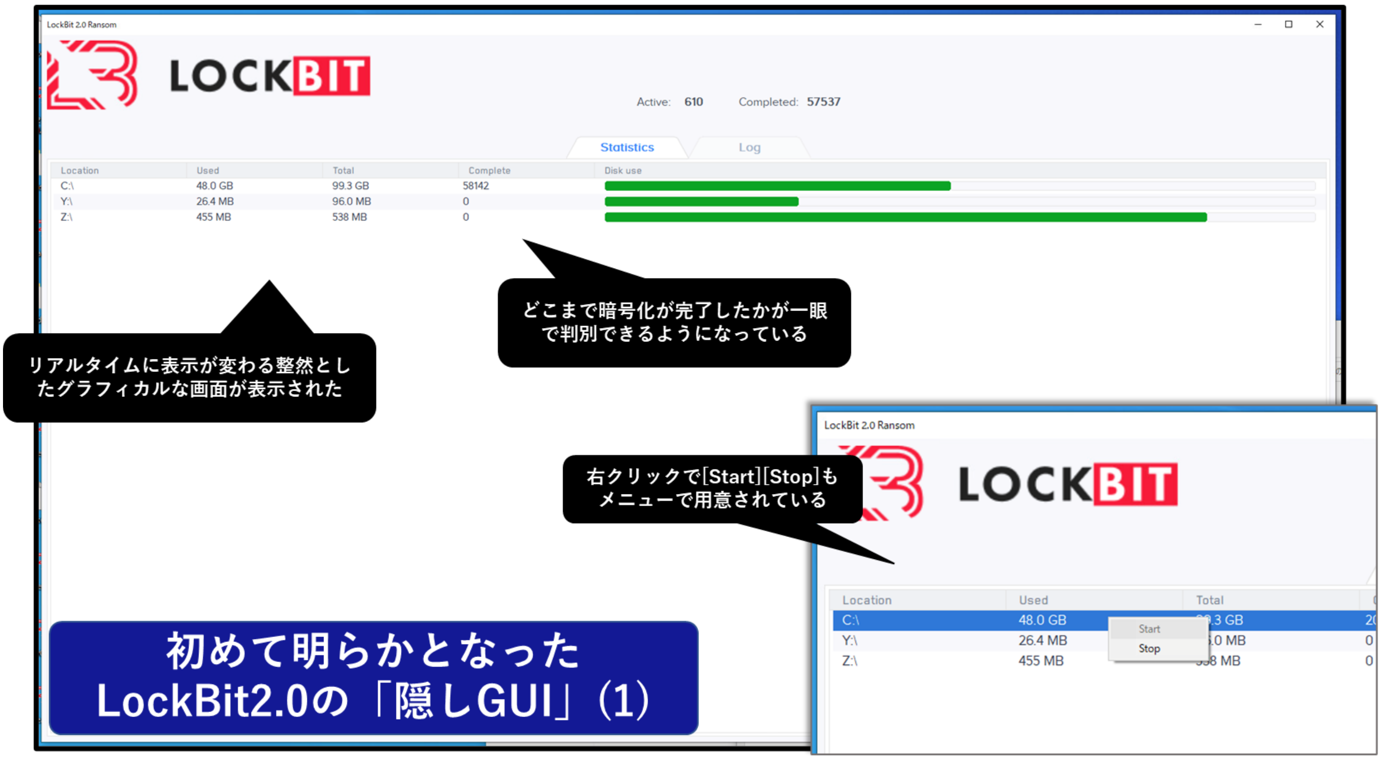


図 120　姿を現した LockBit2.0 の GUI （本来隠されたウインドウ）

以下は、「Log」タブを表示させた様子ですが、動作に関わる細かなログがリアルタイムに出力されていきます。

（これらのログ文字列は、該当する各処理の中で随時にメモリ上に展開され SendMessage 関数で送信されます）

■ バイナリを変更し強制的に表示させた結果、出現したLockBit2.0の「隠しウインドウ」（Statisticsタブ）

図 121 LockBit2.0 の GUI における Log タブの様子

先ほど解説した通り、ShowWindow 関数の引数は変数ではなく 0 で固定化(ハードコード)されているため、それは LockBit2.0 の EXE ファイルがコンパイルされた後に該当の値を動的に変更する仕組みを持たない可能性が高いということを意味します。つまり、RaaS(Ransomware as a Service)の中で、アフィリエイトに提供される LockBit2.0 のバイナリのビルド画面にこの GUI の表示に関わるオプションが用意されていると推測され、画面の表示/非表示があらかじめアフィリエイトによって設定された上でビルド(コンパイル)されているものと考えられます。

攻撃者がランサムウェアを被害組織内に展開し感染させる際は基本的に GUI 画面が表示されるべきではないため、この GUI オプションが有効になったバイナリが出回ることはほとんどないと想定され、そうした背景が LockBit2.0 の隠しウインドウの存在が一般的に知られていない理由の一つであるといえます。

以上、今回弊社の解析により公で初めて明らかとなった LockBit2.0 の GUI が実際に動作する様子は、動画で収録し以下の YouTube にて公開しています。

https://www.youtube.com/watch?v=-LEKTd4rJTU

その他、ドメインコントローラでのグループポリシーを利用したランサムウェアの展開の様子や、プリンタによる印刷動画についても、それぞれ動画に収録し、以下の YouTube で公開していますので併せてご覧ください。

https://www.youtube.com/watch?v=Co8mRfd0iPM
https://www.youtube.com/watch?v=zfh-FqPdXI4

※動画が記事の理解のお役に立てた際はぜひ「いいね!」を押してフィードバックいただけますと幸いです。

※本記事で解析に使用した全ての LockBit2.0 のハッシュ値:

0545f842ca2eb77bcac0fd17d6d0a8c607d7dbc8669709f3096e5c1828e1c049
0906a0b27f59b6db2a2451a0e0aabf292818e32ddd5404d08bf49c601a466744
4bb152c96ba9e25f293bbc03c607918a4452231087053a8cb1a8accb1acc92fd
4edbf2358a9820e030136dc76126c20cc38159df0d8d7b13d30b1c9351e8b277
bcbb1e388759eea5c1fbb4f35c29b6f66f3f4ca4c715bab35c8fc56dcf3fa621
717585e9605ac2a971b7c7537e6e311bab9db02ecc6451e0efada9b2ff38b474
73406e0e7882addf0f810d3bc0e386fd5fd2dd441c895095f4125bb236ae7345
a7591e4a248c04547579f014c94d7d30aa16a01bb2a25b77df36e30a198df108
acad2d9b291b5a9662aa1469f96995dc547a45e391af9c7fa24f5921b0128b2c
b3faf5d8cbc3c75d4c3897851fdaf8d7a4bd774966b4c25e0e4617546109aed5
d089d57b8b2b32ee9816338e96680127babc5d08a03150740a8459c29ab3ba78
f32e9fb8b1ea73f0a71f3edaebb7f2b242e72d2a4826d6b2744ad3d830671202
3cbdd9250d50cf9e5c9aa6c0ade4dde2995e1319e96a160ba6730e063e86f5bc
6edbe66b81f7168fa9d4b087c28409d6dfc98d560760369da4ed4ca2262abd33

三井物産セキュアディレクション株式会社
https://www.mbsd.jp