Contiランサムウェアの内部構造を紐解く | 技術者ブログ | 三井物産セキュアディレクション株式会社

Archived: 2026-04-06 00:52:15 UTC

はじめに

Contiランサムウェアは2020年５月に初めて確認され、現在全世界で多くの被害を出している標的型ランサムウェアであり、近年の流行に沿った二重脅迫を行う攻撃グループが用いるランサムウェアです。つまり、暗号化したファイルを復号するための身代金に関わる脅迫と、身代金の支払いに従わなかった場合にデータを流出させると脅し実際に徐々に公開するという二重の脅迫の手口と共に使用されます。

被害組織から盗み取った情報を公開する為に用意された攻撃グループのサイトをリークサイトと呼びますが、以下はContiランサムウェアのリークサイトのトップ画面であり、Contiランサムウェアの攻撃を受けた被害組織の情報が複数ページに渡り多数掲載されている状況が確認できます。

Contiランサムウェアのリークサイト

ダークウェブ上で攻撃した組織から盗み取ったデータを徐々に公開し脅迫する

▼ Contiランサムウェアのリークサイトのトップ画面

図1 Contiランサムウェアのリークサイト

現在全世界には同様に専用のリークサイトを持つランサムウェア攻撃グループが複数存在しますが、弊社が把握しているそれら約20種類程度の攻撃グループのリークサイトを一通り調査し、各攻撃グループにおける被害企業数(攻撃グループのリークサイトで公開されている組織数)の割合を示したものが以下のグラフです(2021年4月MBSD調べ)。

Contiランサムウェアの攻撃グループが全体の25%を占める結果となり、公開されている中では現在活動中の全てのランサムウェア攻撃グループの中で最も活発かつ被害数が多いと推測され、2021年以降だけでも、ひと月あたり平均30組織を上回る数の被害が継続して毎月確認されています。
Conti攻撃グループは顧客から盗み取ったデータの中からサイバー保険の契約書を見つけ出し、保険の補償額と照らし合わせることで支払い能力を指摘し交渉してくるような交渉術も持ち合わせており、以下の図の数字にもその洗練された能力の高さが現れています。

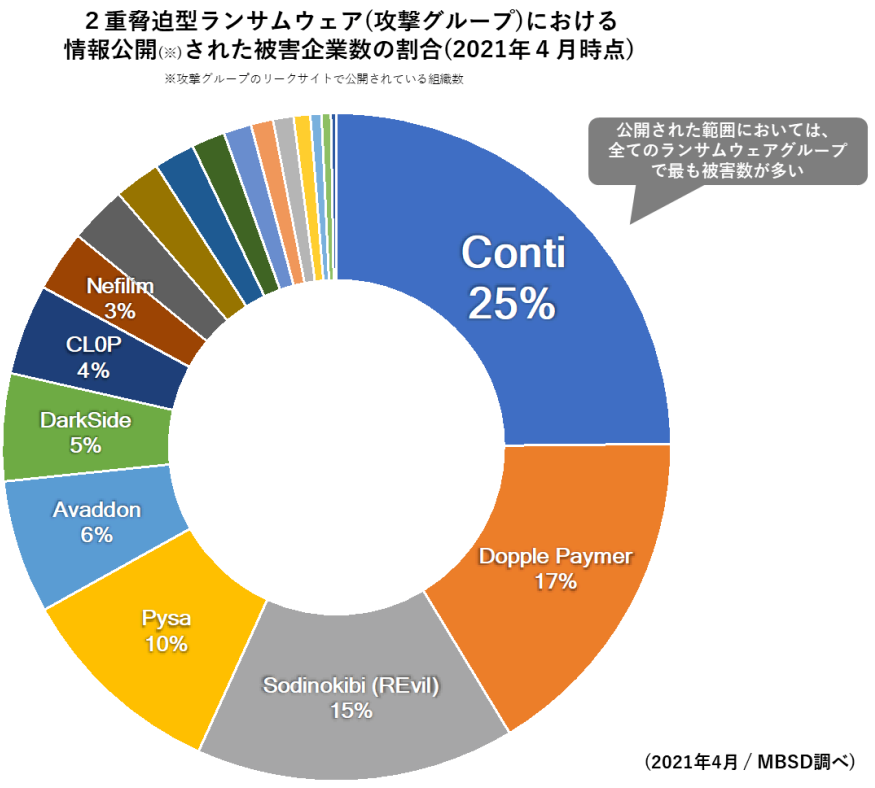


図 2　情報公開された被害企業数の割合

そのため、今回は現時点において最大のランサムウェア脅威の一つといえるContiランサムウェアに着目し詳細解析した結果を公開することにしました。

Contiランサムウェア本体に関する解析記事は既に世界各国のベンダー等からいくつか出てはいますが、それらを”ランサムウェア本体の理解”という一つの限られた視点で見た場合、挙動の一部しか記載されていなかったり、処理全体が把握できなかったりする記事が多くを占めていると感じました。
そのため、今回は世界で最も丁寧で詳細なContiランサムウェアの解析記事となることを心がけました。
また、今回併せてContiランサムウェアが解析妨害する処理の流れを明らかにする専用の自動解析スクリプトも新たに作成しGitHubで公開しました。（詳細は本文中程をご参照ください）

通常であれば解説を省略するような内容についても可能な限り詳細に解説していますので冗長な部分はありますが、本記事がContiランサムウェア本体の挙動の全貌を理解する一助になれば幸いです。

Contiランサムウェアの全体挙動と概要

Contiランサムウェアは出現以降、速いペースでバージョンアップを行っており、本執筆時点の最新バージョンはV3（バージョン３）となっています。そのため、本記事もV3のContiランサムウェアを対象に選定し解析を行いました。

まずはざっと全体を把握いただけるように、Contiランサムウェアの全体挙動の概要を一枚の図にしたものが以下の図です。ファイルレスとなるReflective PE Injection(詳細は後述)を複数使用した解析検知妨害や、動作中に使用する全ての文字列やAPIに暗号化を施すなど、一般的な他のランサムウェアと比較して非常に手の混んでいる作りとなっています。

以降では、これらの挙動一つ一つについて詳細に解説していきます。
（本記事はとてもボリュームが多いため、読み進める中でどこの解説であるかを見失った際は、この図に戻り一度俯瞰してみてください）

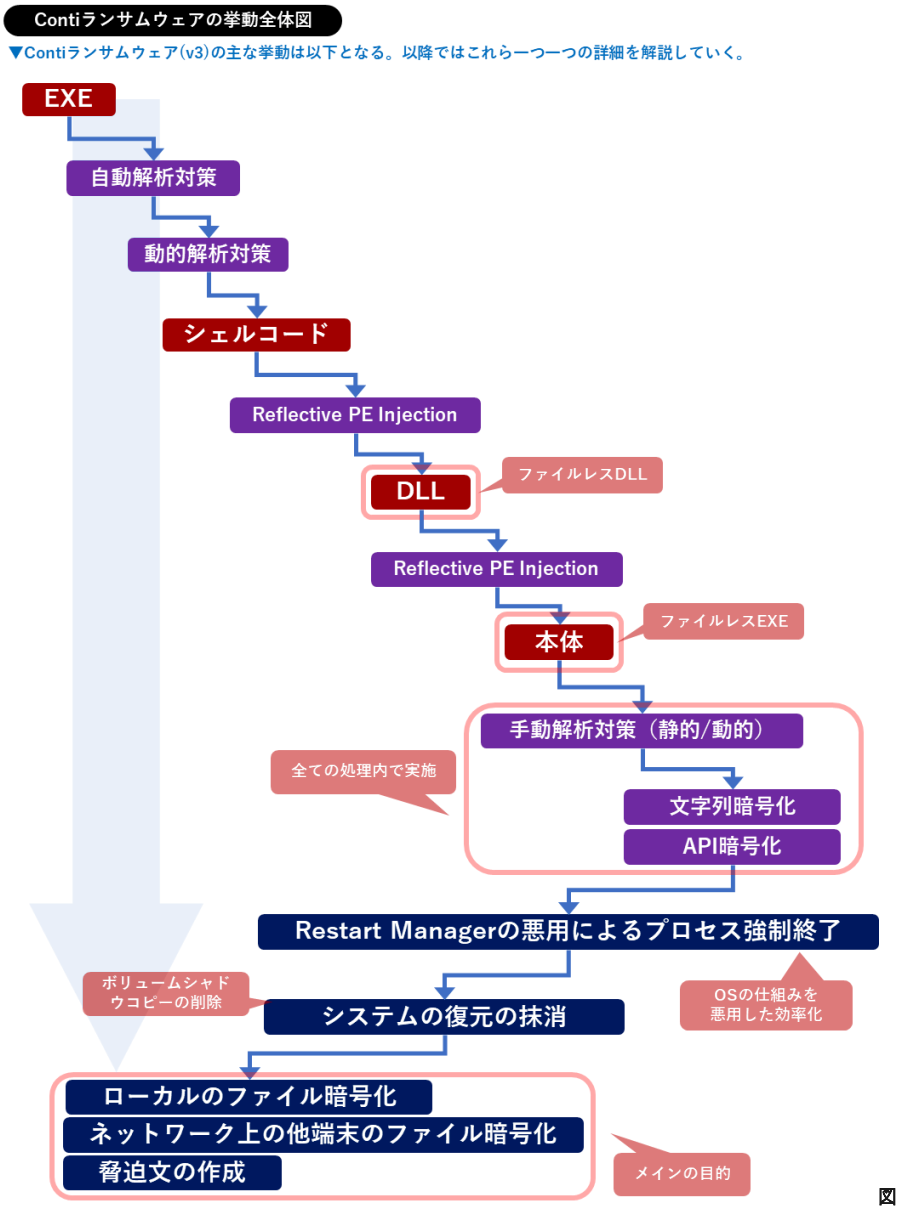


図 3

Contiランサムウェアの挙動全体図

表層解析

まずは表面的な構造から見ていきましょう。
Contiランサムウェアは EXE ファイルであり、リソースセクションに以下のようなダイアログを持っていますが、これらはダミーであり実行しても使用されません。こうした使用しないリソースを含むものはたまに見られますが、分析された際に正常な実行ファイルであると偽装する意図があるものと考えられます。

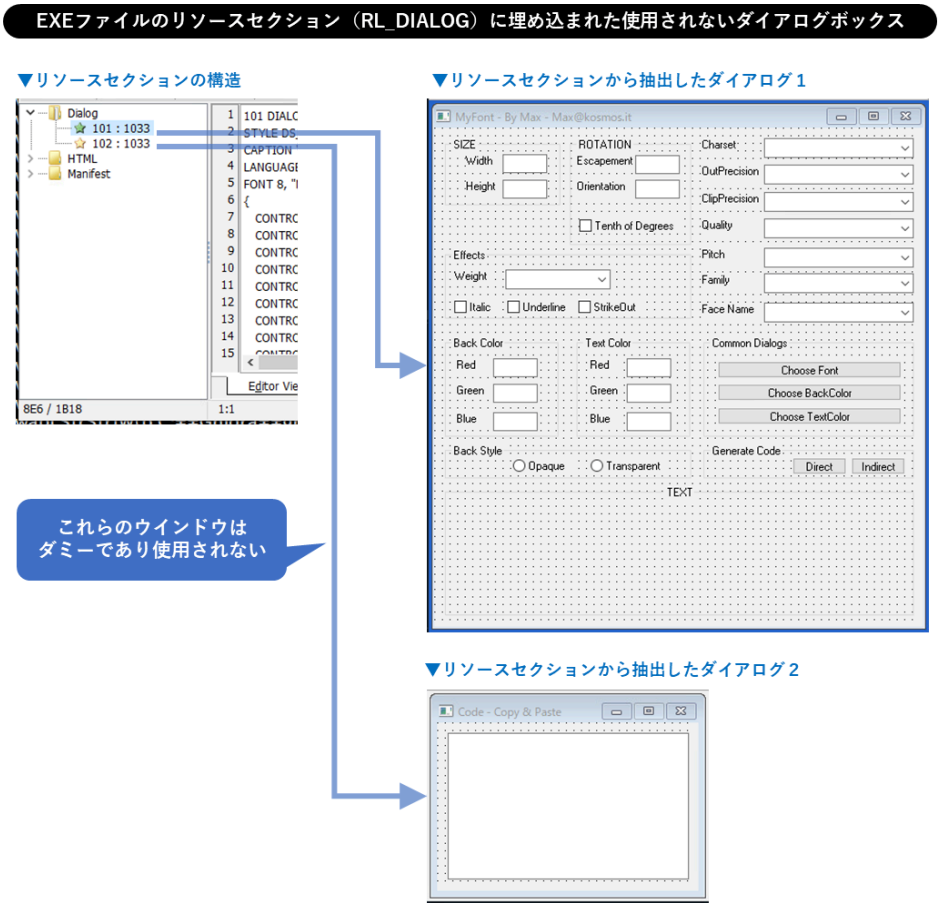


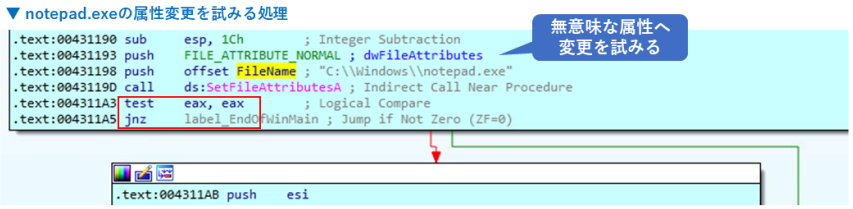
図 4リソースセクションに埋め込まれた使用されないダイアログボックス

耐自動解析

Contiランサムウェアは実行されるとすぐにWindowsのメモ帳である「C:\Windows\notepad.exe」の属性変更を試みます。このシステムファイルへの属性変更処理は通常のWindows環境であれば常に失敗する処理となります。この後の処理の分岐としては、属性変更処理が"成功した場合"のみ自身を終了し、"失敗した場合"のみ自身を継続する処理に遷移します。
つまり、「呼び出した関数が失敗すると動く」という不可思議なロジックになっており、そもそも失敗する前提で作られていることがわかります。
マルウェアの自動解析システムなどでは、マルウェアの処理の内部に何らかの関数の呼び出しがあり、その関数の成功有無によって変化する分岐が存在した場合、関数の結果が成功する遷移へ処理を強制変更させるというケースがありうるため、耐自動解析としてあえて正常な環境では必ず失敗するはずの分岐を挙動の冒頭に入れた可能性が考えられます。

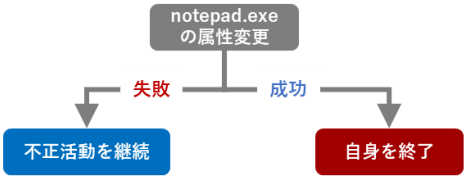
自動解析対策の可能性

Contiランサムウェアは起動してすぐにWindowsのメモ帳（notepad.exe）の属性変更を試みる。
このシステムファイルへの属性変更処理は通常のWindows環境であれば常に失敗する処理となる。



この後の処理の分岐としては、
上記の属性変更処理が成功した場合のみ自身を終了し、失敗した場合に自身を継続する処理に遷移する。

つまり失敗すると動くという不可思議なロジックになっており、上記の処理はそもそも失敗する前提で作られている。



マルウェアの自動解析システムなどでは、ある関数の成功有無によって変化する分岐が存在した場合、
関数の結果が成功する遷移へ処理を強制変更させるケースがありうるため、
あえて、正常な環境では必ず失敗するはずの処理の分岐を挙動の冒頭に入れた可能性が一つ考えられる。

図 5　notepad.exeの属性変更処理

再びEXEファイルの構造に話を戻しましょう。
最初に実行されるEXEファイルの表面的な構造と領域の比率を示したものが以下の図です。
以下の図からリソースセクション内に [RT_HTML->7765]という領域が存在することがわかりますが、214KB であるEXEファイル全体のサイズのうち、[RT_HTML->7765]のデータサイズは200KBと、EXEファイルにおけるサイズの大部分を占めていることがわかります（下図）。

ContiランサムウェアのEXEファイルのPE構造

▼リソースセクション内の[RT_HTML->7765]という名前の領域がEXEファイルサイズの大部分を占めていることがわかる

▼ContiランサムウェアのEXEファイルサイズは214KB

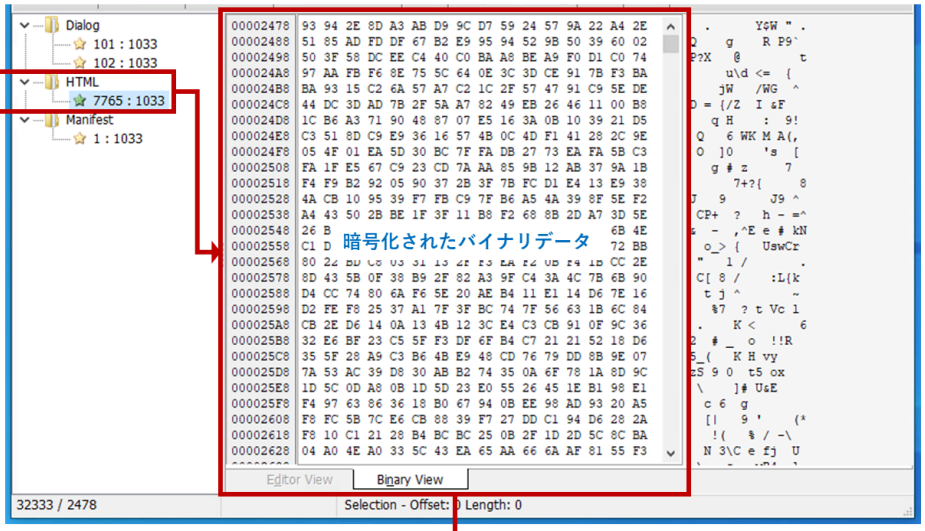
図 6 ContiランサムウェアのEXEファイルのPE構造

では上記リソースセクション内の [RT_HTML->7765]という領域には何が含まれているのでしょうか。[RT_HTML->7765]の領域をバイナリデータで表示させたものが以下の図です。[RT_HTML->7765]には判別不能な暗号化されたバイナリデータが格納されていることがわかります（下図）。

Contiランサムウェアのリソースセクションにおける[RT_HTML->7765]に含まれたデータ

▼以下の通り、[RT_HTML->7765]という名の領域には、200KBの（暗号化された）バイナリデータが含まれていることがわかる

▼EXEファイル内の [RT_HTML->7765]のリソース構造



▼Contiランサムウェアのファイル構造の概念図

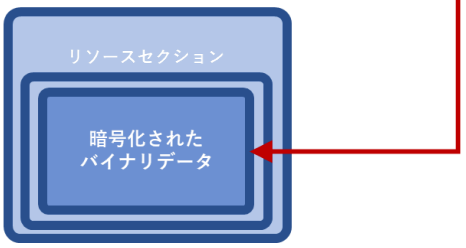


図 7　[RT_HTML->7765]に含まれたデータ

動き出したContiランサムウェアは、該当の [RT_HTML->7765]というリソース領域を指定し、新しく確保したメモリ領域（以降では、メモリ領域[α]と呼ぶ）にコピーします（以下図）。
この際、一般にあまり多くは使用されることのないネイティブAPIであるLdrFindResource_UとLdrAccessResourceを用いてリソースセクションから[RT_HTML->7765]を取り出します。

Contiランサムウェアの実行直後の挙動

実行されるとリソースセクションに埋め込まれたデータを、確保したメモリ領域［α］にコピーする

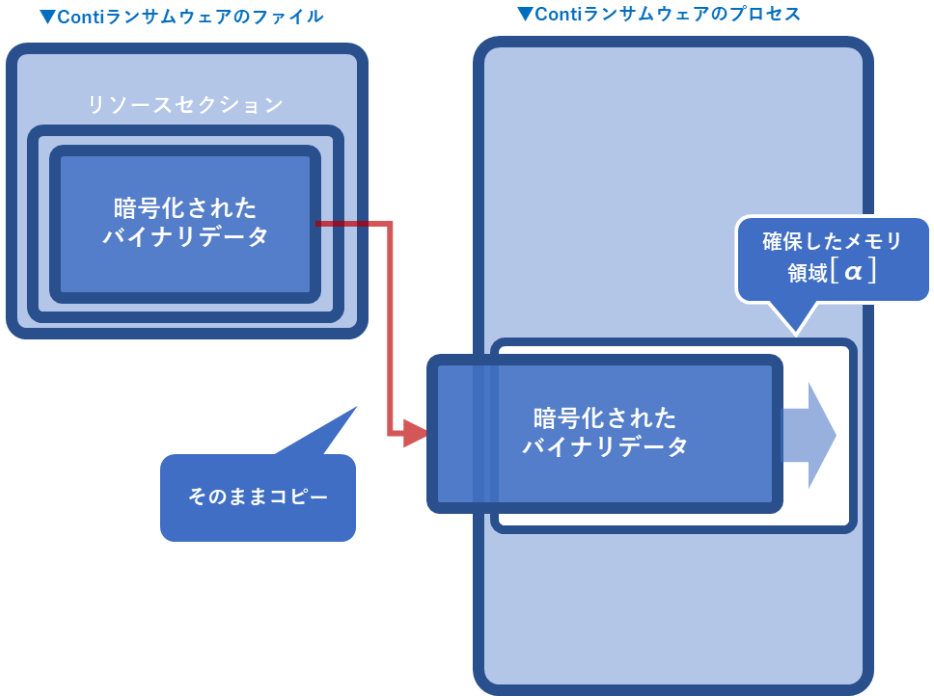
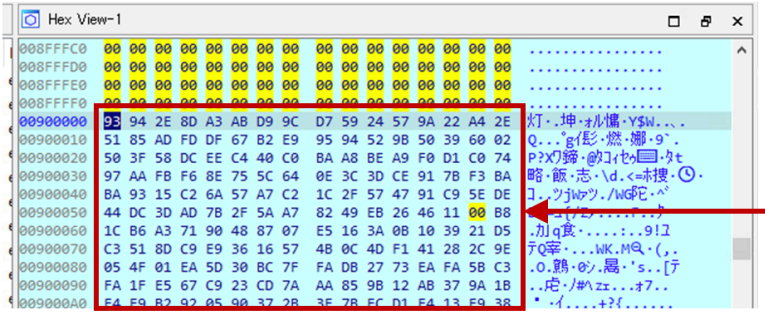


図 8　Contiランサムウェアの実行直後の挙動

VirtualAlloc関数でメモリ領域[α]を確保する際は、コードとして実行可能なアクセス権である実行権を持つ状態で作成します。
以下は実際のメモリ領域[α]に展開されたデータとリソースセクションの[RT_HTML->7765]を比較した様子ですが、同じバイナリデータがそのままメモリ上にコピーされていることがわかります。

Contiランサムウェアが実行後、メモリ上に展開するリソースデータ

▼VirtualAllocによって実行権が付与され確保されたメモリ領域［α］に展開されたデータ（画像は一部）



▼先ほど確認したリソースセクションの[RT_HTML->7765]と同じバイナリが並んでいることがわかる

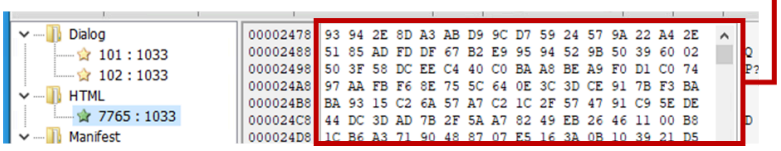


図 9 実行後メモリ上に展開するリソースデータ

続いてContiランサムウェアは、メモリ領域[α]に展開した該当のデータを復号する作業に入ります。その際、"0x3D"という1バイトの値と以下のランダムな文字列を用いて復号キーを生成します（以下図）。

"4lIzzSbq#>J1v*CSIr#ofX3Bh%)f$3CQSdkz!vnUspXDIu4RJYNXVaE#%uS)"

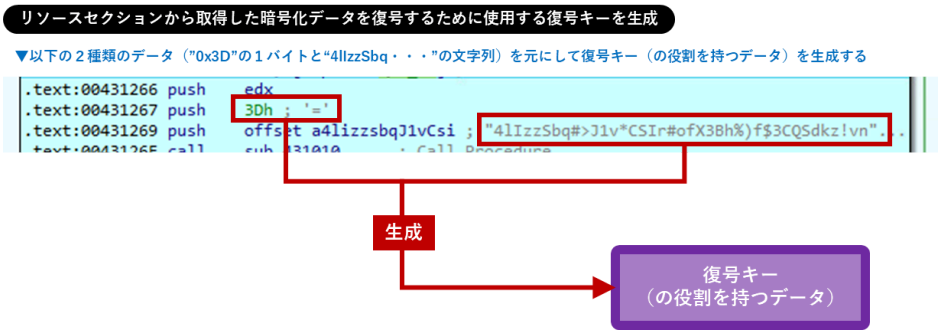


図 10　復号キーの生成

生成した復号キーを用いて、メモリ領域[α]上のデータを１バイトごとに復号したデータで上書きしていきます（下図）。

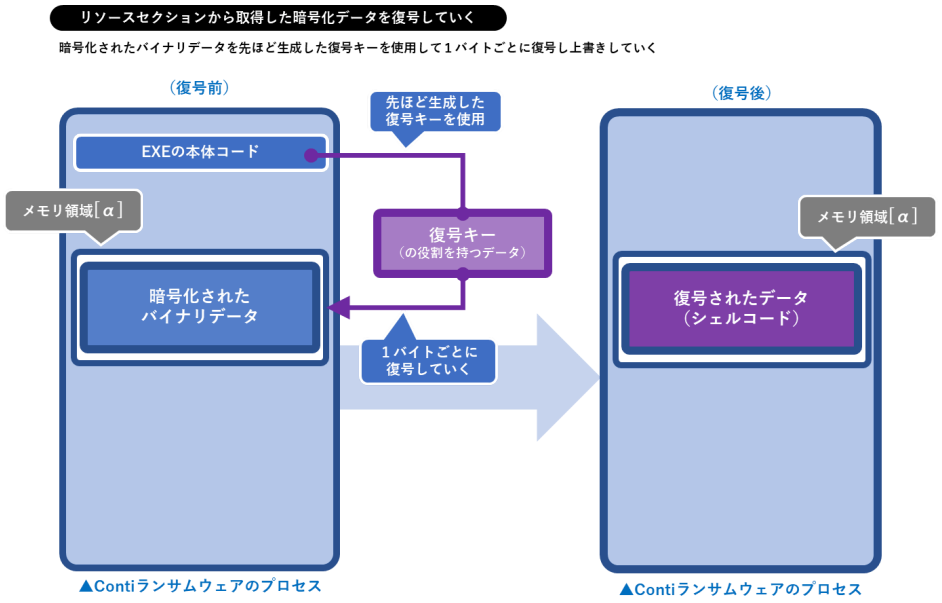


図 11　リソースセクションから取得したデータの復号

この際、１バイトごとのループにつき無意味なprintf関数を３回呼び出します。復号のループ処理は最大 205,619 回（0x32333回）行われるため、結果的に最大で約61万回(205,619回×３)のprintf関数の呼び出しが発生することになります。
Contiランサムウェアがこうした処理を入れている背景ですが、もし解析ツールや解析システムなどを用いてプロセスから呼び出される全てのAPIの呼び出しを動的に監視した場合、監視している分だけ実際よりも一つのAPIの呼び出しに遅延が発生します。大量の無駄なAPI呼び出しをわざと生み出すことで結果的に膨大な処理時間を発生させることになるため、これは動的解析に対する一種の解析妨害といえます。
実際に呼び出されるAPIを動的解析により監視した結果、大量のprintf関数でログが埋め尽くされ膨大な時間を要することがわかります（下図）。

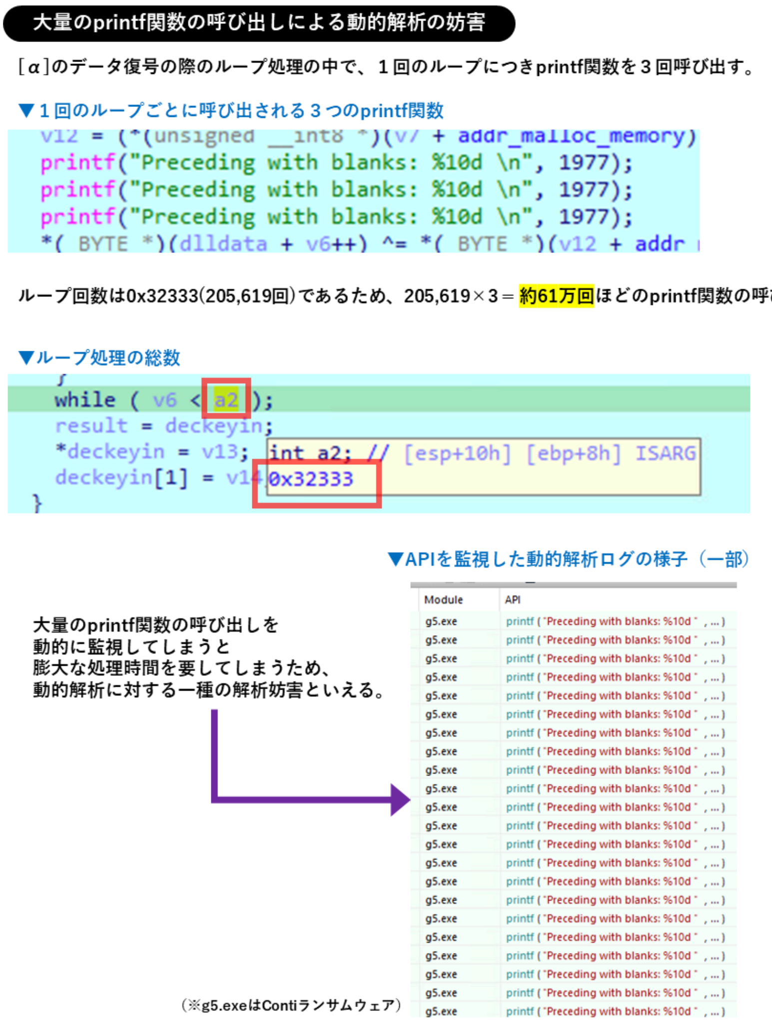


図 12　大量のprintf関数の呼び出しによる動的解析の妨害

シェルコードの解説

上記の処理が終わりメモリ領域[α]に復号されたデータはシェルコード（単独で動作するように機械語で作られた一連の不正コード）となっており、先頭アドレスがcall命令で呼び出されることで実行処理がEXEからシェルコードに遷移します（下図）。

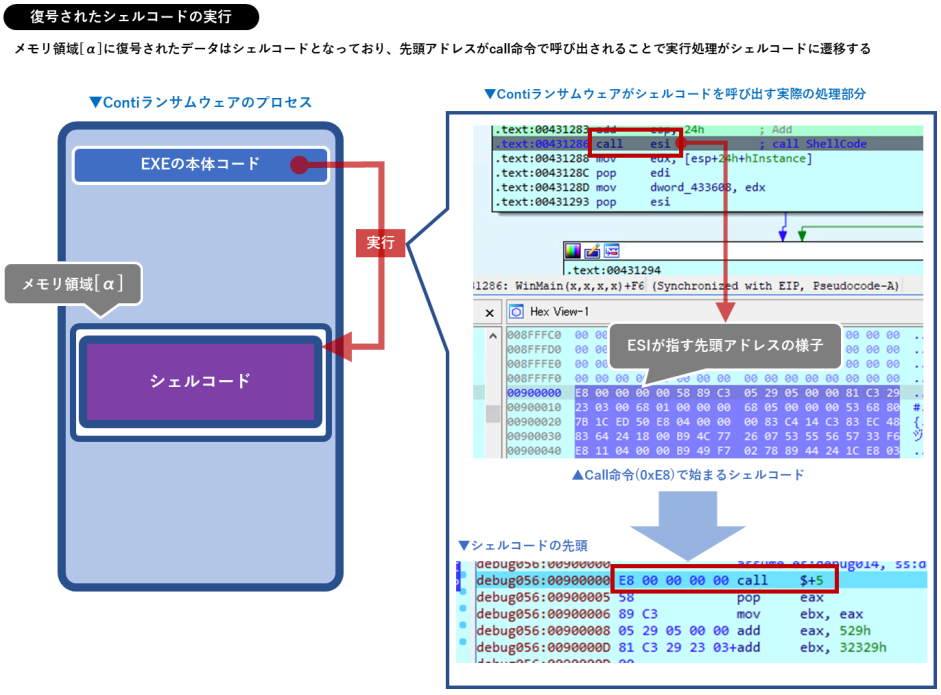


図 13　復号されたシェルコードの実行

実行が遷移したシェルコードは、WindowsAPIを使用するための準備に入ります。
自身のプロセスのPEBを辿っていき、InLoadOrderModuleListを参照することで、プロセス起動時にロードされた順の各DLLにおけるベースアドレスを取得していきます（下図）。

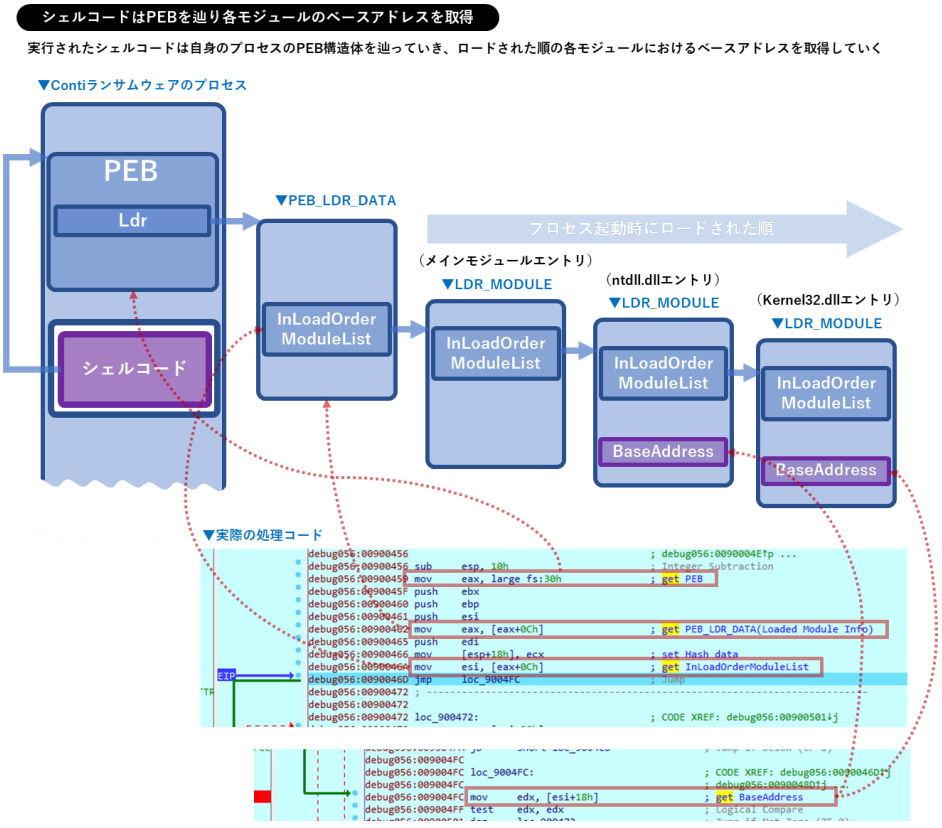


図 14　シェルコードがPEBを辿りモジュールのベースアドレスを取得

その後、各DLLが提供するAPI名を一つずつ取得していきます(下図)。

シェルコードは取得した各DLLのベースアドレスからエクスポート関数のリストを取得

それぞれのDLLが提供する関数（API）名を一つずつ取得していく

▼ntdll.dllがエクスポートしているAPIを参照している際の様子

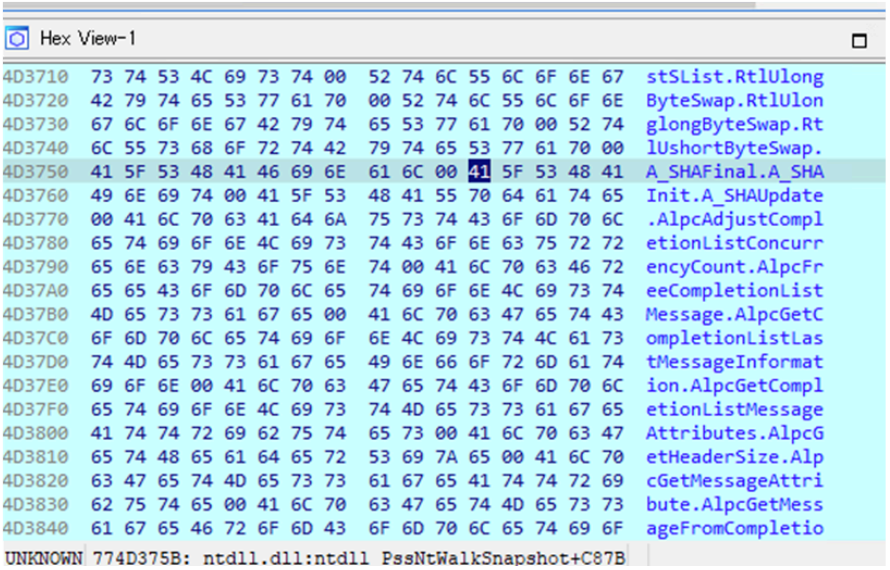


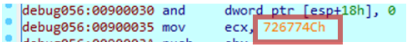
図 15　ベースアドレスを元にエクスポート関数のリストを取得

シェルコードの中には、使用したいWindowsAPI名がハッシュ値化した状態でハードコーディングされています（下図）。
シェルコードはDLLから取得した実際のAPI名の文字列をハッシュ計算し、ハードコーディングされているハッシュ値と比較していくことで、必要なAPIを特定していきます（下図）。

取得したAPI名をハードコーディングされているハッシュ値と比較

シェルコードのバイナリ内には複数のWindows API名をハッシュ化した値がハードコーディングされている

▼シェルコード内にハードコーディングされたハッシュ値（一例）



シェルコードはDLLから取得した実際のAPI名の文字列からハッシュ値を計算し、比較することで必要なAPIを特定していく

▼API名を取得しハッシュ計算、既存のハッシュ値と比較する処理

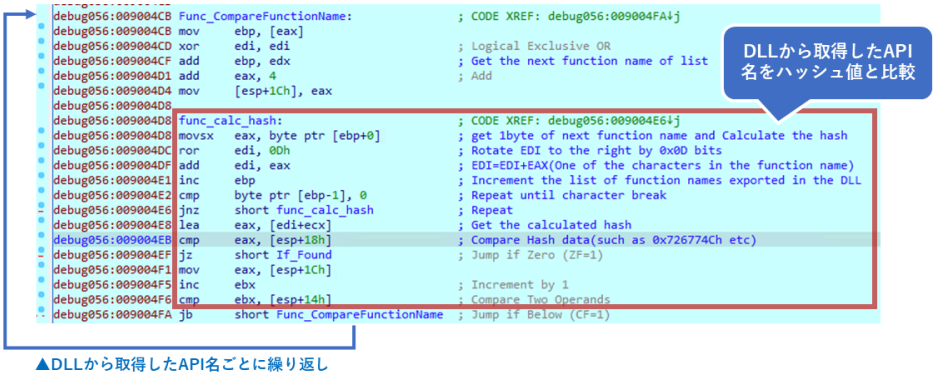


図 16　取得したAPI名をハードコーディングされているハッシュ値と比較

その結果、以下のように必要最低限のいくつかのWindowsAPIのアドレスを取得します（下図）。

ハッシュ値とそれに対応するWindowsAPI名

その結果、シェルコードは以下のようにいくつかの必要最低限のAPIのアドレスを取得する

▼ハッシュ値に対応するAPI名を割り出した様子

```
debug056:00900030 and     dword ptr [esp+18h], 0          ; Logical AND
debug056:00900035 mov     ecx, 726774Ch                   ; Kernel32_LoadLibraryA
debug056:0090003A push    ebx
debug056:0090003B push    ebp
debug056:0090003C push    esi
debug056:0090003D push    edi
debug056:0090003E xor     esi, esi                        ; Logical Exclusive OR
debug056:00900040 call    Get_API_Address                 ; Call Procedure
debug056:00900045 mov     ecx, 7802F749h                  ; Kernel32_GetProcAddress
debug056:0090004A mov     [esp+1Ch], eax
debug056:0090004E call    Get_API_Address                 ; Call Procedure
debug056:00900053 mov     ecx, 0E553A458h                 ; Kernel32_VirtualAlloc
debug056:00900058 mov     [esp+20h], eax
debug056:0090005C call    Get_API_Address                 ; Call Procedure
debug056:00900061 mov     ecx, 0C38AE110h                 ; Kernel32_VirtualProtect
debug056:00900066 mov     ebp, eax
debug056:00900068 call    Get_API_Address                 ; Call Procedure
debug056:0090006D mov     ecx, 945CB1AFh                  ; ntdll_NtFlushInstructionCache
debug056:00900072 mov     [esp+2Ch], eax
debug056:00900076 call    Get_API_Address                 ; Call Procedure
debug056:0090007B mov     ecx, 959E0033h                  ; kernel32_GetNativeSystemInfo
debug056:00900080 mov     [esp+30h], eax
debug056:00900084 call    Get_API_Address                 ; Call Procedure
```

図 17　ハッシュ値とそれに対応するWindows API名

ここで一度シェルコードの内部構造についてざっくりと解説しておきましょう。
Contiランサムウェアのシェルコードは以下の図のような3層構造になっています。
シェルコードにはDLLが含まれ、さらにそのDLLの内部にはEXEが埋め込まれています（下図）。

Contiランサムウェアは2回のReflective PE Injectionというテクニックを重ねて使用することでこれらのDLLやEXEをディスク上に書き込むことなくファイルレスで実行させます。
Reflective PE Injectionとは実行ファイルをハードディスクに書き込むことなく、つまりファイルレスでプロセスにロードさせ実行する技術です。この辺りの処理についてものちほど詳しく解説します。

Contiランサムウェアのシェルコードの構造

シェルコードは以下のような３層の構造になっている。
シェルコードにはDLLが含まれ、さらにDLLの内部にはEXEが埋め込まれている。

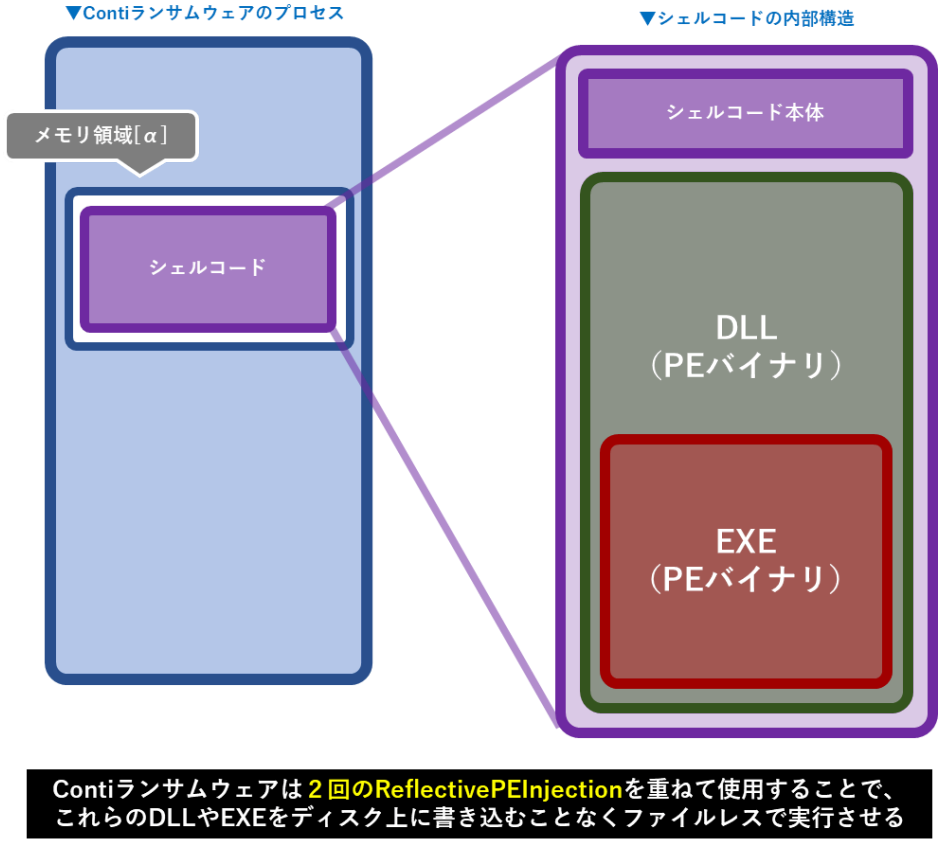


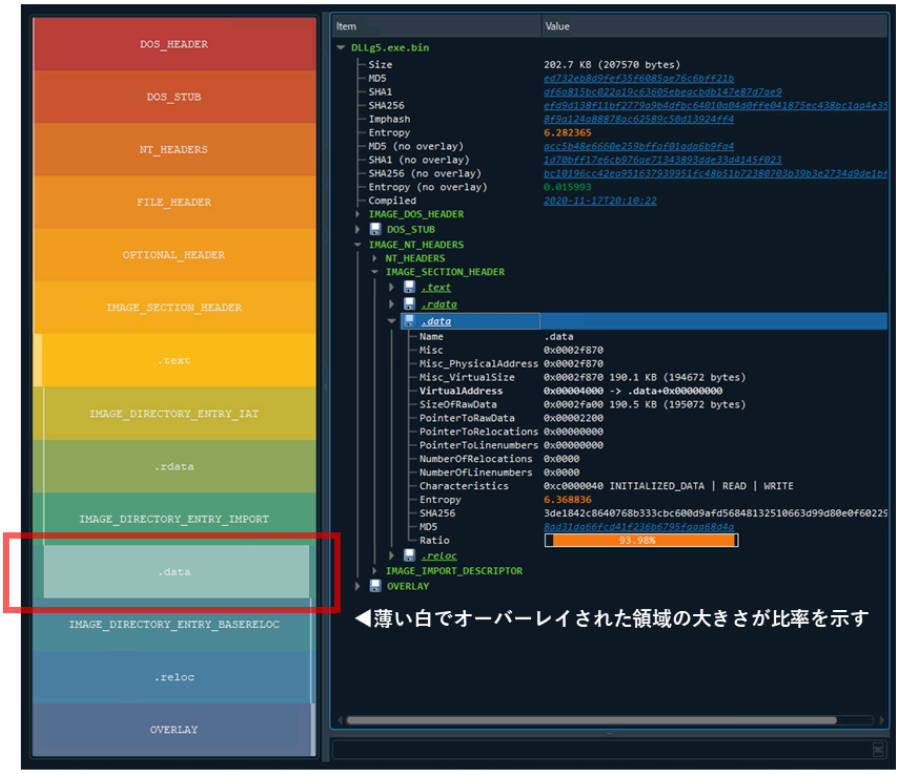
図 18　Contiランサムウェアのシェルコードの構造

上記の３層構造を実際に理解するため、シェルコード内に存在するDLL（以降ではReflective DLLと呼ぶ）の内部構造をさらに見てみましょう。
シェルコードから取り出したReflective DLLの構造を見ると、[.data]セクションがファイルサイズの大部分を占めていることがわかります（下図）。
さらに[.data]セクションに含まれるバイナリデータを確認するとPEファイルのMZヘッダの先頭を示す[4D 5A]の2バイトが確認できます。つまりこの結果から、DLLの中にさらにPEファイル(EXE)が含まれていることがわかります。

Contiランサムウェアのシェルコードに含まれるDLLの構造

シェルコードから取り出したDLLは.dataセクションがサイズ比率の大部分を占めることがわかる。

▼シェルコードから取り出したDLLの構造



.dataセクションの先頭から0x70の場所に、EXEのMZヘッダが確認できる。

▼DLLの.dataセクション部分のバイナリを表示させた様子

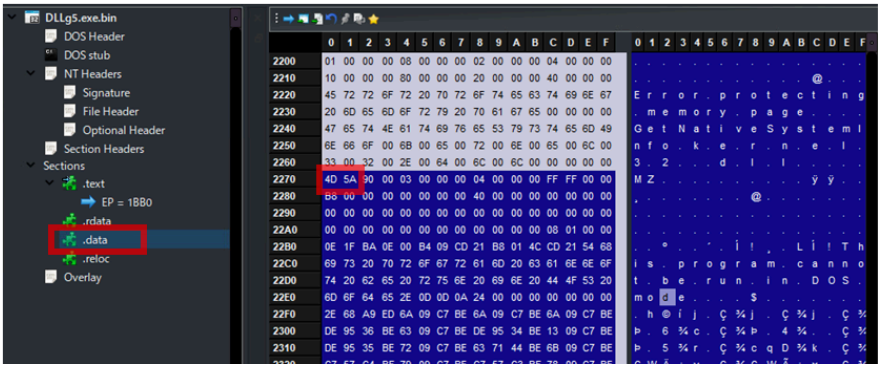


図 19　Contiランサムウェアのシェルコードに含まれるDLLの構造

では、話をシェルコードの動きに戻しましょう。
シェルコードは、Reflective DLLのヘッダ(Optional headers->SizeOfImage)を参照することでDLLの全体サイズを取得し、そのサイズ分のメモリ領域（以降ではメモリ領域[β]と呼ぶ）を確保して０で一度初期化します。この際にメモリ確保のために使用するVirtualAlloc関数では実行権を持たないRead/Writeのみのアクセス権でメモリを確保します。

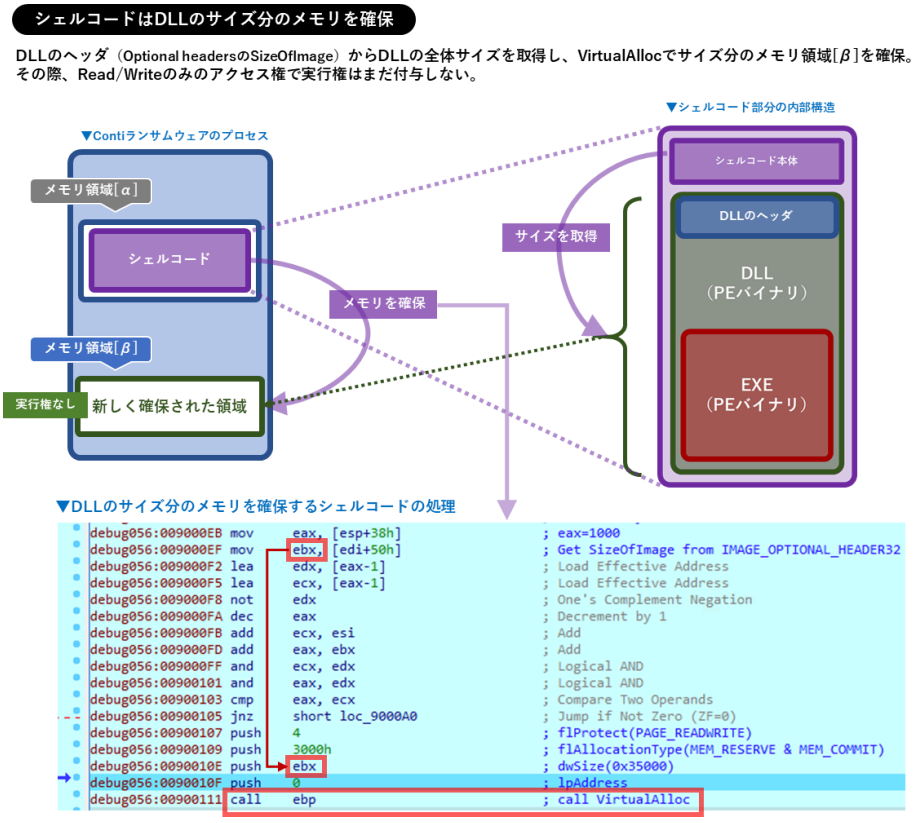


図 20　シェルコードはDLLのサイズ分のメモリを確保

ここからシェルコードによる1度目のReflective PE Injectionの処理に入っていきます。

シェルコードは確保したメモリ領域[β]にReflective DLLのヘッダの一部をコピーします。
以下の比較図の通り、メモリ領域[β]にコピーされたデータにはMZヘッダなど一部のデータが存在しないことがわかります（下図）。

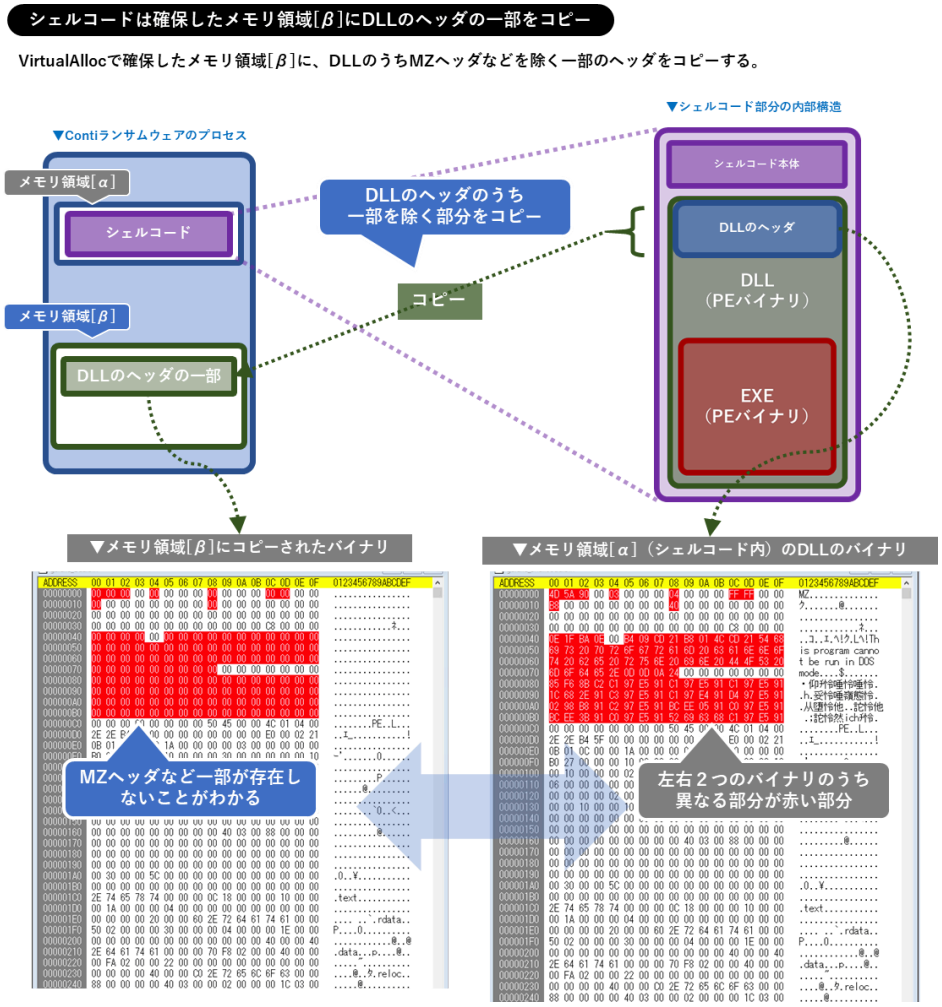


図 21　シェルコードは確保したメモリ領域[β]にDLLのヘッダの一部をコピー

上記のヘッダコピー処理では、具体的には以下に挙げる各情報がシェルコード内のReflective DLLからメモリ領域[β]へコピーされます。

- PEヘッダ（NT headers）へのオフセット (DOS Header の"File address of new exe header"の値)
- PEヘッダ（NT headers）のうち、Data Directories以外
- セクションテーブル

続いて、メモリ領域[β]の後方にReflective DLLの各セクションのデータをコピーしていきます（下図）。なお、以下の図に示したとおり、Reflective DLLの[.data]セクションに位置づけられる領域に、上で少し触れたEXE（以降ではReflective EXEと呼ぶ）が含まれています。

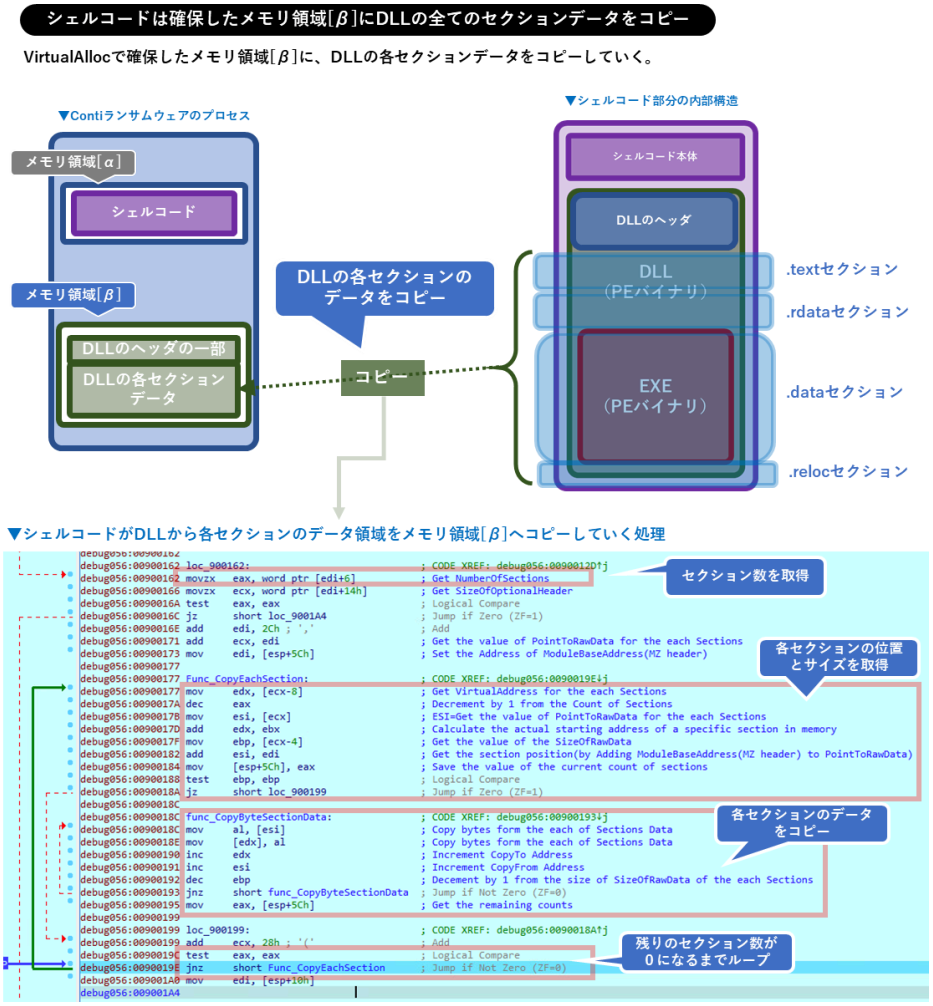


図 22　シェルコードは確保したメモリ領域[β]にDLLの全てのセクションデータをコピー

シェルコードはその後、当初実行権をつけずに確保したメモリ領域[β]にコピーしたReflective DLLにおいて、そのコードセクションにあたる[.text]セクションのメモリ領域を指定し実行権を付与することで、Reflective DLLのコードを実行できるようにするための準備を行います（下図）。

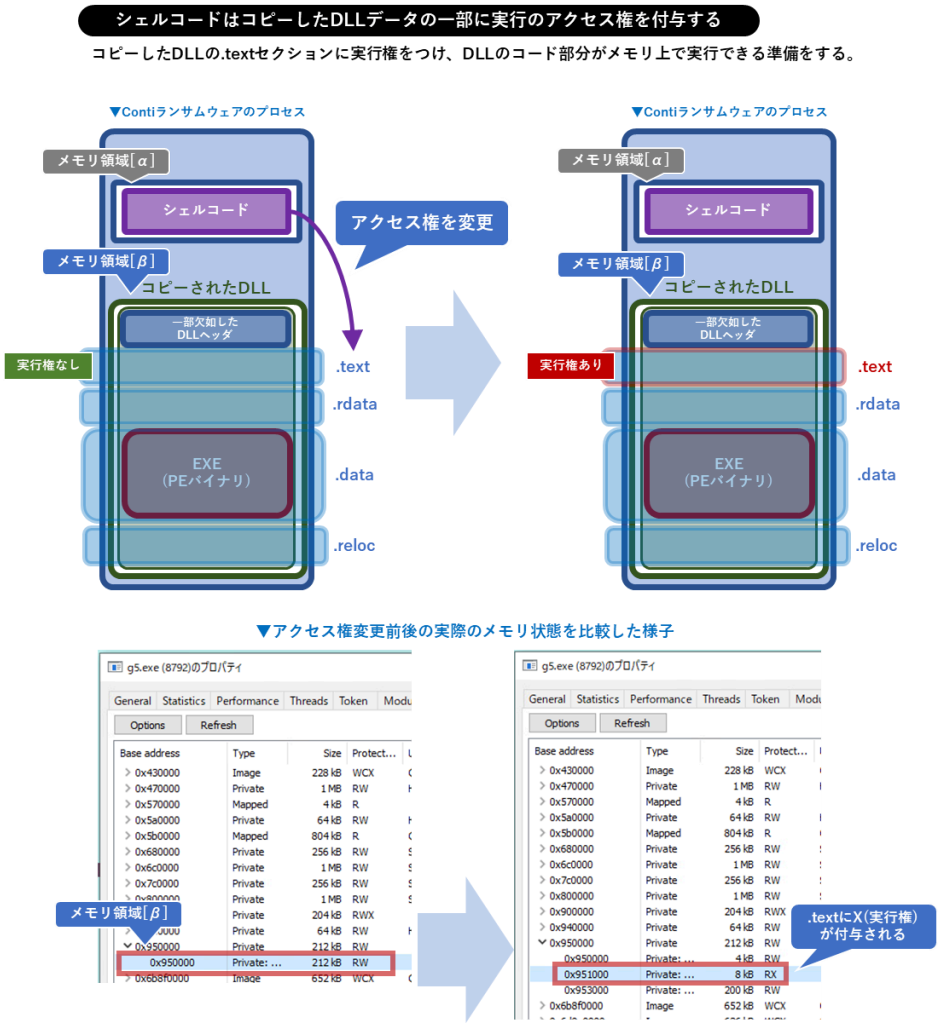


図 23　シェルコードはコピーしたDLLデータの一部に実行のアクセス権を付与

その後、シェルコードはReflective DLLの.textセクションをcall命令で呼び出すことで、実行処理がシェルコードからReflective DLLに遷移します（下図）。
この流れでおわかりのように、このReflective DLLはハードディスク上に作成されません。つまり、ファイルレスでDLLが実行されることになります。これがReflective PE Injectionの1回目に該当します。
なお、Reflective PE Injectionを用いて実行されたReflective DLLは、LoadLibraryなどの通常の手順でロードされていないため、プロセス調査ツール等を用いて調べてもプロセスのロード済みモジュール一覧などに表示されることはなく、存在に気づくことが困難となります。

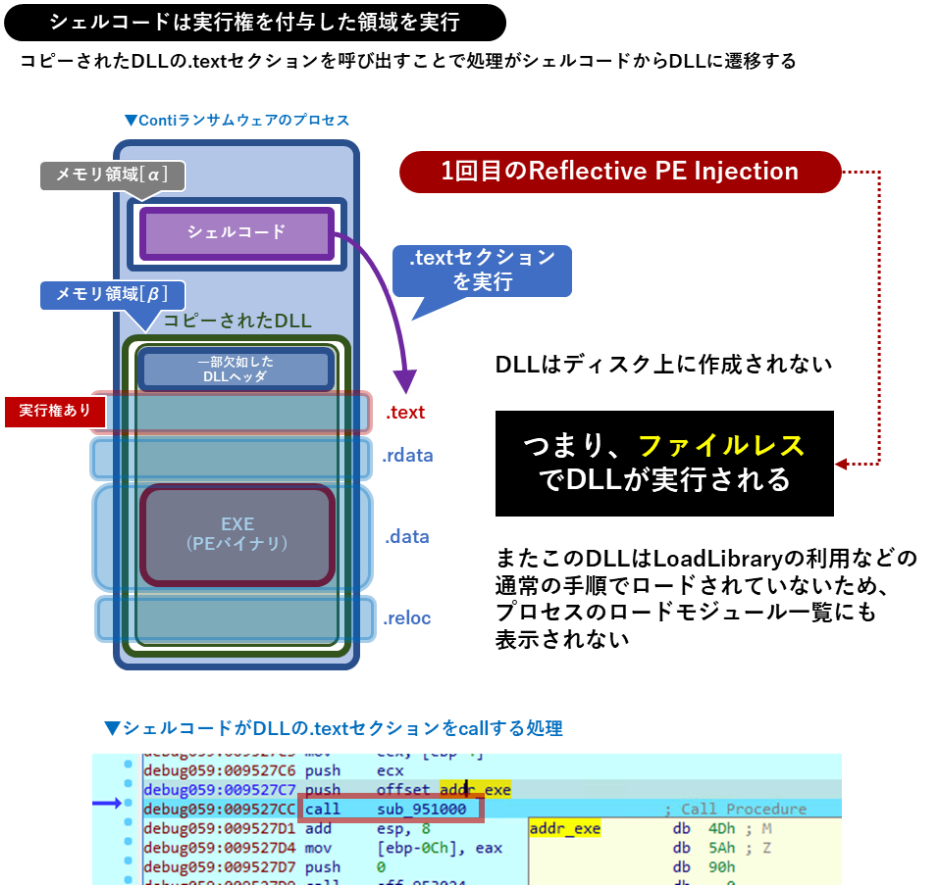


図 24　シェルコードは実行権を付与した領域を実行

シェルコードの処理は以上となり、これからReflective DLLの挙動に入っていきます。

Reflective DLLの解説

シェルコードによって実行されたReflective DLLは、自身の[.data]セクションの領域（つまりReflective EXEが含まれる領域）を参照し、新たに確保したメモリ領域（以降ではメモリ領域[γ]と呼ぶ）にコピーしていきます（下図）。
なお、この際確保したメモリ領域[γ]にはこれまで同様まだ実行権を付与しません。

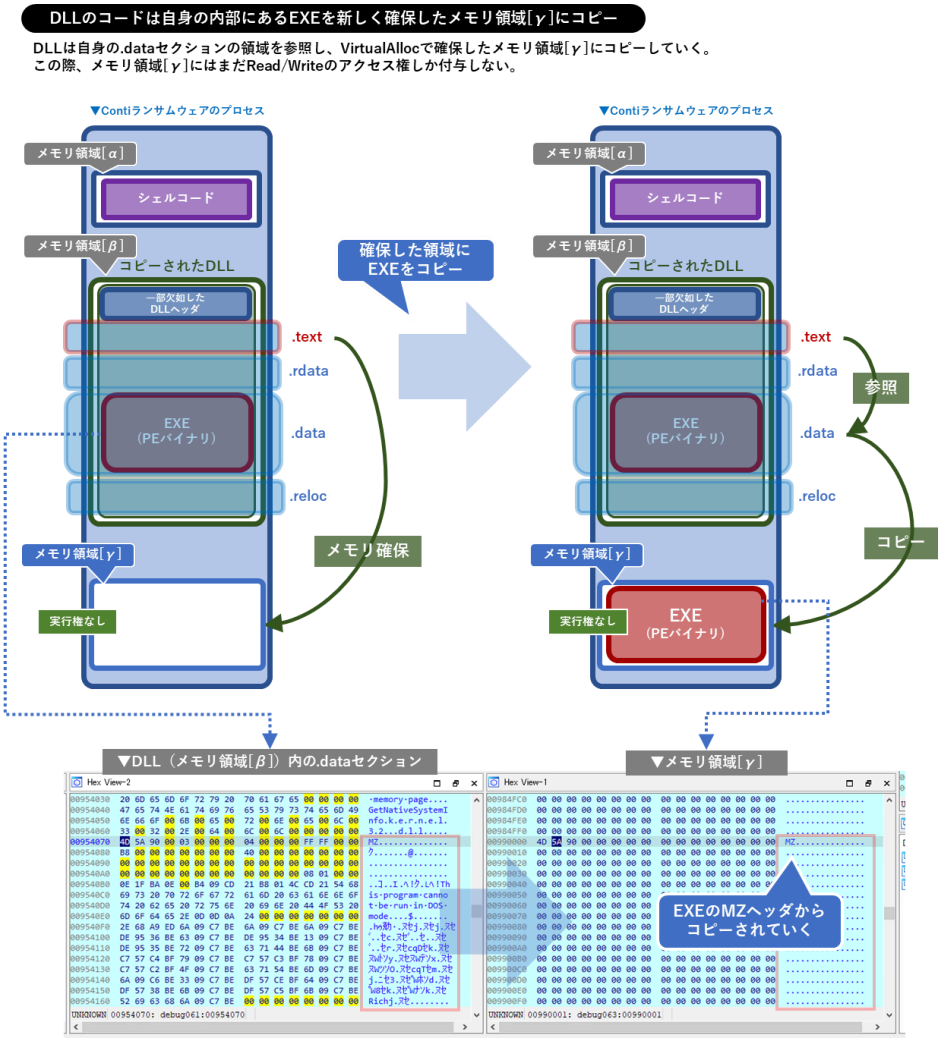


図 25　DLLのコードがEXEをメモリ領域[γ]にコピー

その後、Reflective DLLは、領域[γ]にコピーしたReflective EXEのコードセクションとなる[.text]セクションに実行権をつけることでコードとして実行できる状態にします。

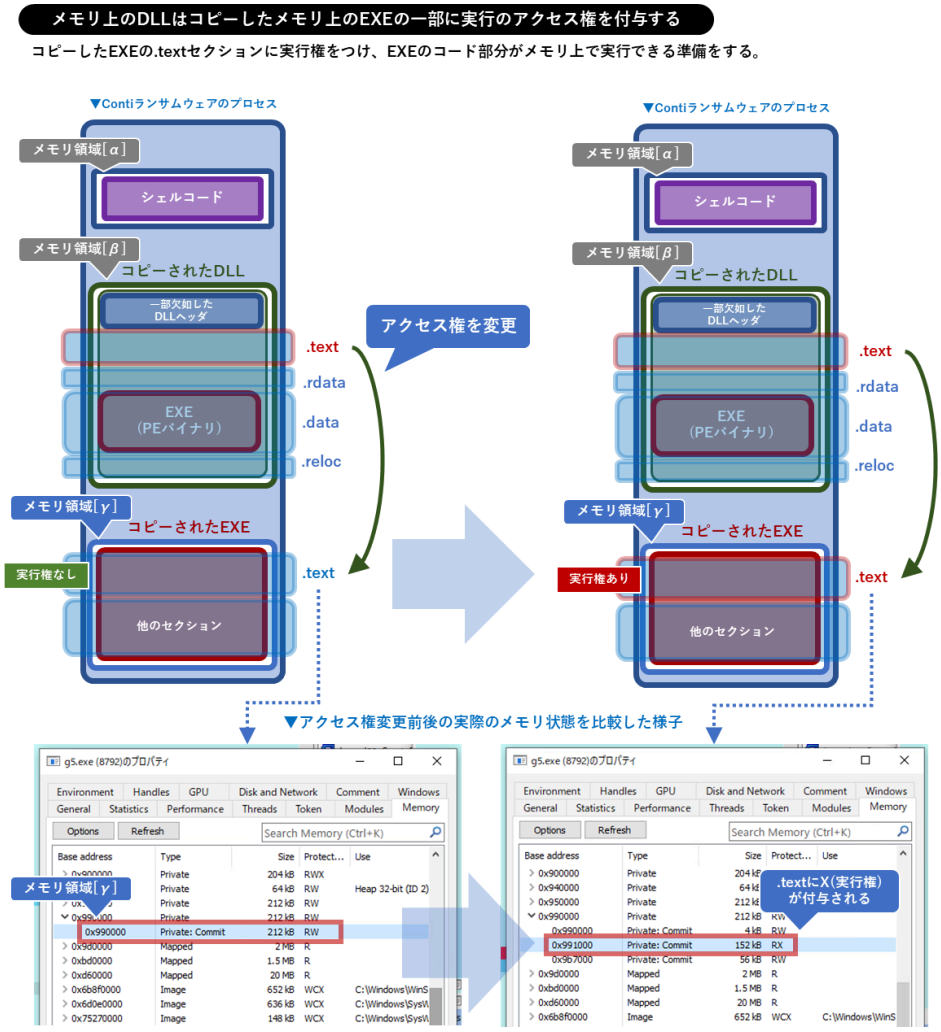


図 26　コピーしたメモリ上のEXEの一部に実行のアクセス権を付与

これでReflective EXEを実行できる状態になりました。
Reflective DLLは、Reflective EXEの[.text]セクションをcall命令で呼び出すことで実行し、実行処理がReflective DLLからReflective EXEに遷移します。
ここの流れでもおわかりの通り、Reflective EXEはハードディスク上に作成されず、ファイルレスで実行されることになります。つまり、これが2回目のReflective PE Injectionとなります。
なお、実行されたReflective EXEは通常の手順でプロセスとして実行されたわけではないため、プロセス調査ツールなどで見てもプロセス一覧には当然表示されません。

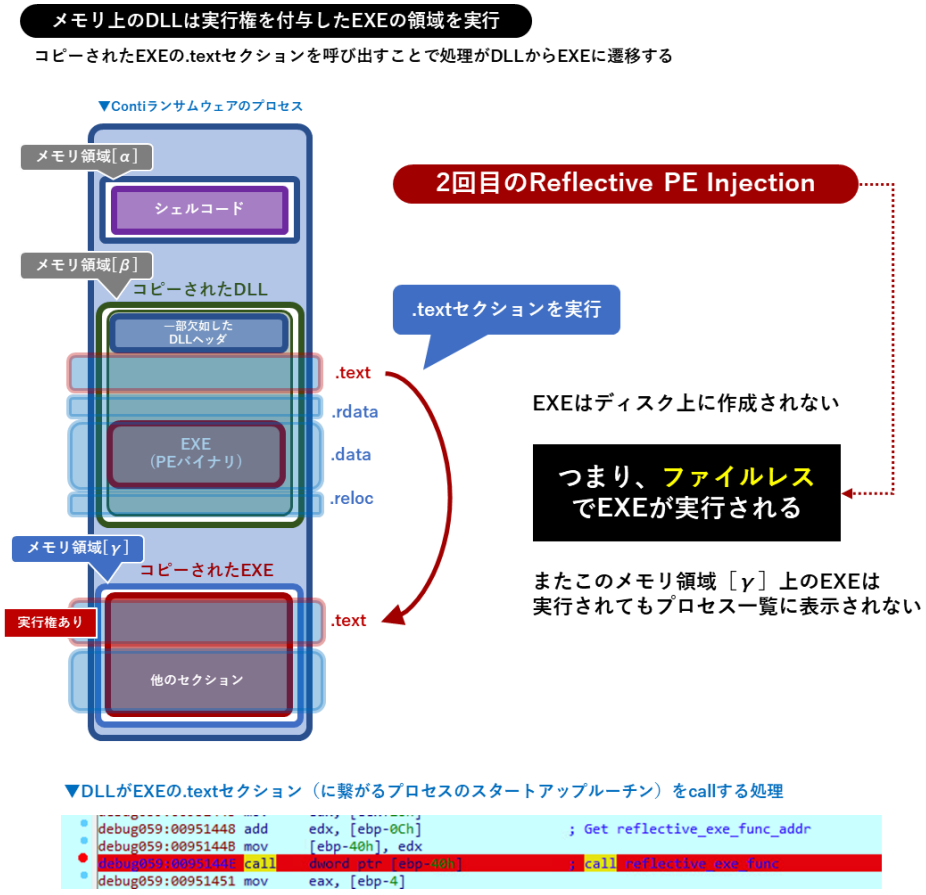


図 27　メモリ上のDLLは実行権を付与したEXEの領域を実行

このように、Contiランサムウェアは2回のReflective PE Injectionを重ねて利用することで、検知や解析から逃れつつ、ランサムウェアとしてのメイン機能であるReflective EXEの実行を最後に呼び出すわけです。

以上でDLLの役目は終了し、心臓部であるReflective EXEの挙動に入っていきます。

Windows APIの暗号化による解析妨害

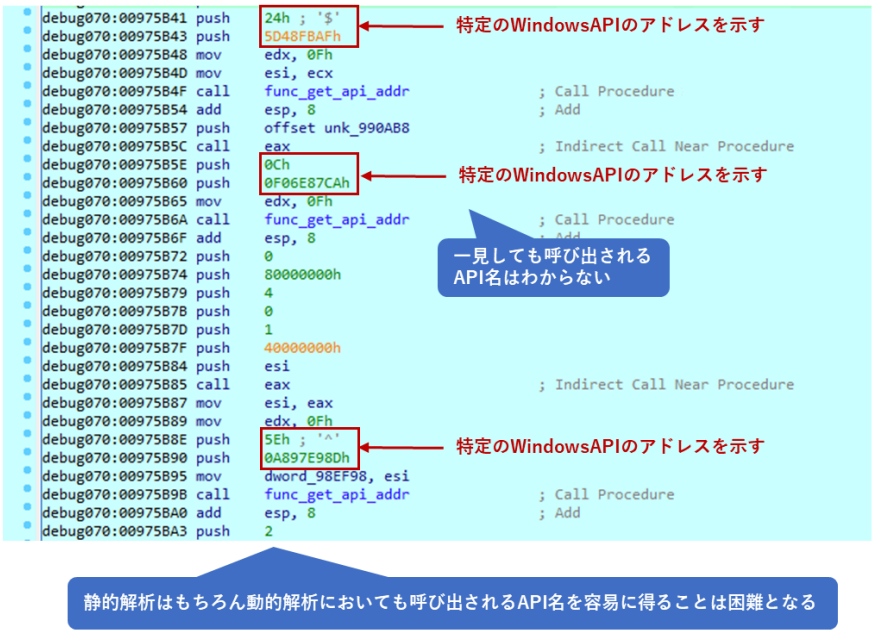
ここから、Contiランサムウェアのメイン機能となるReflective EXEの挙動を解説していきますが、その前にReflective EXEが行ういくつかの解析妨害機能をはじめにご紹介しましょう。

Reflective EXEは、動作中に使用する全てのWindows APIを暗号化しています。
具体的には以下の図のように固有のハッシュ値と1バイトキーを用いて、使用するたびに必要なWindows APIのアドレスを復号した上で呼び出します。全てのWindows APIが暗号化されているため、静的解析はもちろん動的解析においても呼び出されるWindows APIを容易に得ることは困難となります。
詳しくは後述しますが、さらにWindows APIに渡す引数の文字列も、全ての文字列をそれぞれ異なる暗号化関数で暗号化しており、文字列を使用するたびに復号します。

WindowsAPIを隠蔽する耐解析機能

Contiランサムウェアは使用する全てのWindowsAPIを固有のハッシュ値と１バイトキーで都度復号し利用する。

▼ContiランサムウェアがWindowsAPIを呼び出す際の処理例




```
debug070:00975B41 push    24h ; '$'
debug070:00975B43 push    5D48FBAFh
debug070:00975B48 mov     edx, 0Fh
debug070:00975B4D mov     esi, ecx
debug070:00975B4F call    func_get_api_addr             ; Call Procedure
debug070:00975B54 add     esp, 8                        ; Add
debug070:00975B57 push    offset unk_990AB8
debug070:00975B5C call    eax                           ; Indirect Call Near Procedure
debug070:00975B5E push    0Ch
debug070:00975B60 push    0F06E87CAh
debug070:00975B65 mov     edx, 0Fh
debug070:00975B6A call    func_get_api_addr             ; Call Procedure
debug070:00975B6F add     esp, 8
debug070:00975B72 push    0
debug070:00975B74 push    80000000h
debug070:00975B79 push    4
debug070:00975B7B push    0
debug070:00975B7D push    1
debug070:00975B7F push    40000000h
debug070:00975B84 push    esi
debug070:00975B85 call    eax                           ; Indirect Call Near Procedure
debug070:00975B87 mov     esi, eax
debug070:00975B89 mov     edx, 0Fh
debug070:00975B8E push    5Eh ; '^'
debug070:00975B90 push    0A897E98Dh
debug070:00975B95 mov     dword_98EF98, esi
debug070:00975B9B call    func_get_api_addr             ; Call Procedure
debug070:00975BA0 add     esp, 8                        ; Add
debug070:00975BA3 push    2
```

また、Contiランサムウェアは後述の通り、全ての文字列をそれぞれ異なる暗号化関数で暗号化しており、WindowsAPIの引数に渡す文字列についても例外なく個別に暗号化されている。

図 28　Windows APIを隠蔽する耐解析機能

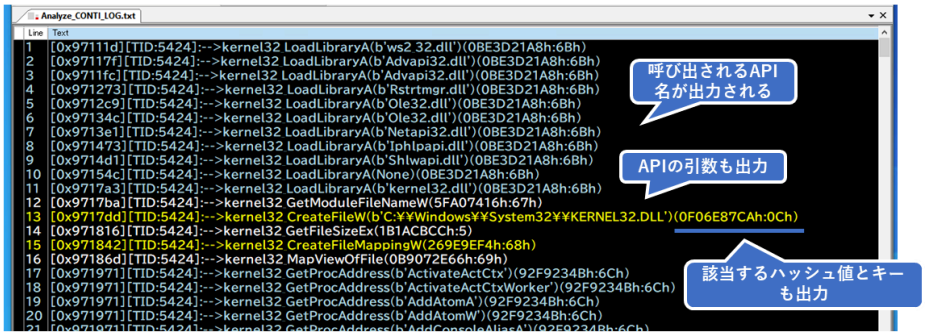
Contiランサムウェア自動解析スクリプト

そのため、今回Contiランサムウェアを解析するにあたり、Contiランサムウェアが呼び出すAPIの復号結果を列挙するオリジナルの自動解析スクリプトを新たに作成しました。
この自動解析スクリプトは、暗号化されたAPI名だけでなく、それぞれ個別に暗号化されている引数の文字列についても、復号後の文字列を一部併せて出力します。
さらに、それぞれのAPIに対応するハッシュ値と1バイトキーの組み合わせも末尾に出力します。
以下はその出力ログのサンプル画面です。

Contiランサムウェアが使用する暗号化されたAPIを明らかにする自動解析スクリプトを作成

本解析にあたり、Contiランサムウェアが呼び出すAPIの復号結果を列挙するオリジナル自動解析スクリプトを作成。
それぞれ個別に暗号化されているAPIの引数についても、復号後の文字列を一部併せて出力する。
またそれぞれのAPI名に対応するハッシュ値＆１バイトキーの組み合わせも出力する。

▼（ログ出力サンプル）自動解析スクリプトが出力するログのサンプル



（※画像のログは処理の種類に併せて別途ハイライトしている）

▼（ログ出力サンプル）検索したファイル／フォルダ名と除外対象の文字列を比較する処理部分

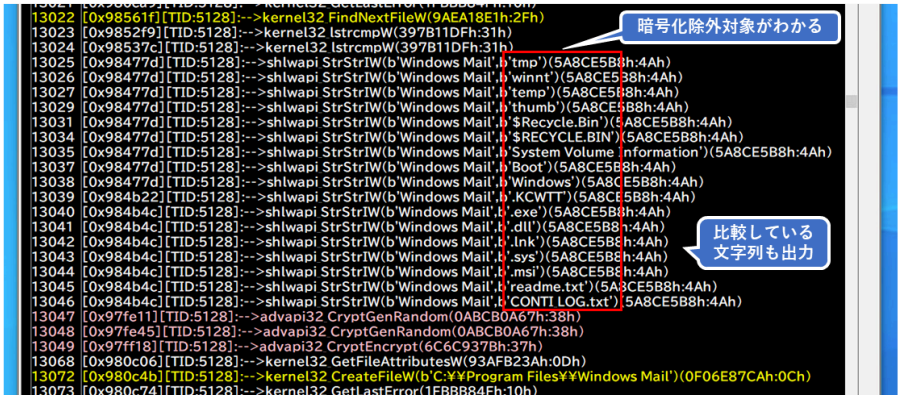


図 29　自動解析スクリプトのログ出力サンプル

また本自動解析スクリプトには補足機能として、静的解析ツールであるIDA Proの逆コンパイラ画面（IDB）の各ハッシュ値の右横に、復号後のDLL名とAPI名をコメントとして自動入力する機能もつけています。

補足機能として、IDAの逆コンパイラ画面(IDB)の各ハッシュ値の横に復号後のDLL名とAPI名をコメントとして自動入力する機能もつけている。

▼（例）自動解析スクリプトが自動コメントする情報

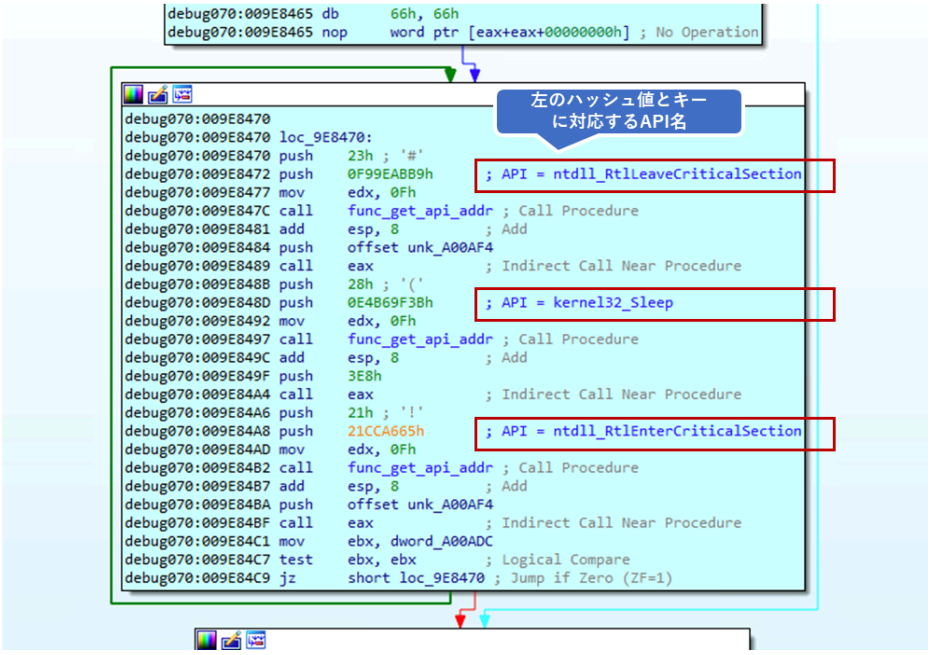


図 30　自動解析スクリプトが自動コメントする情報

以下は自動解析スクリプトを実際に動作させている様子ですが、Contiランサムウェアが呼び出すAPIとその引数が復号された状態でログとして出力されます（画面中央の黒い部分が出力ログ）。

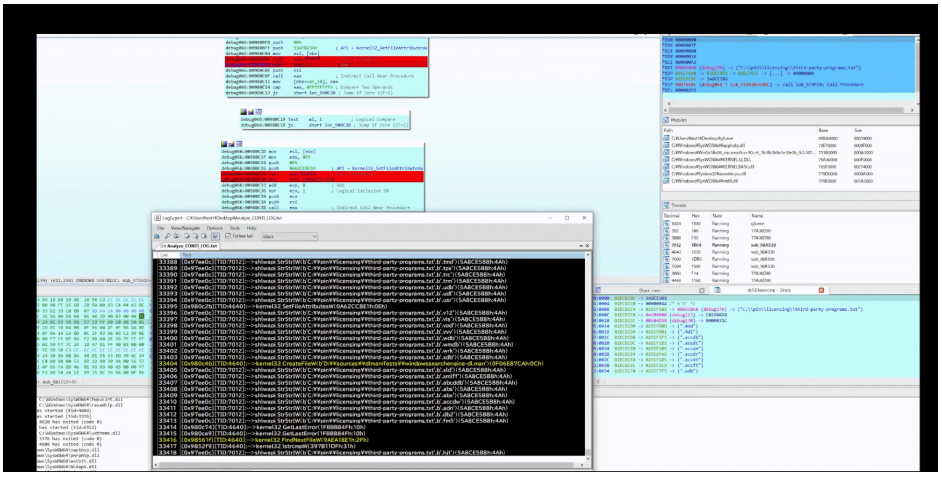


図 31　自動解析スクリプトの実行様子

上記の自動解析スクリプトは以下のGitHubにアップしています。

URL：https://github.com/AgigoNoTana/resolve_conti_API/blob/main/resolve_conti_API.py

文字列の暗号化による解析妨害

先ほど少し触れましたが、Contiランサムウェアは挙動の中で使用する全ての文字列をそれぞれ固有の計算により暗号化しており、文字列ごとに用意された関数で使うたびに復号して使用します。

では、その文字列の暗号化について少し踏み込んで詳しく解説しましょう。
以下の図は、ある一つの文字列（"explorer.exe"という文字列）を復号する処理をピックアップしたものですが、暗号化された文字列をその文字列専用に用意された個別の計算処理で1バイトごとに復号することを示しています。

下図におけるアセンブリ言語をよりわかりやすく書き直したものを下図右下に載せていますが、図に掲載した通りの計算に従うと、例えば0x7Cという値は計算され最終的には0x65となり、ASCIIの"e"というアルファベットになることがわかります。

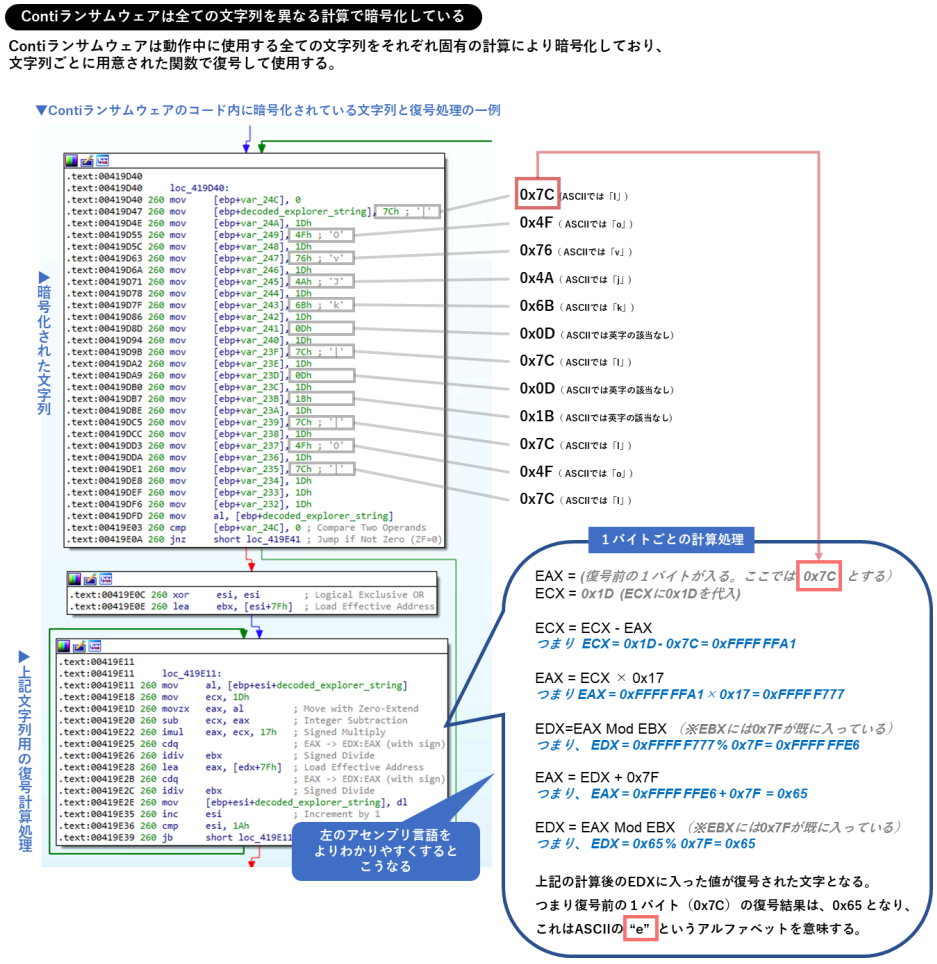


図 32 Contiランサムウェアによる文字列の暗号化例１

上記の文字列に対する計算を一つの式にした場合、以下の図のようになります。全てのバイトを同様の計算式で復号していくと、"explorer.exe"という文字列が表れます（下図）。
なお、復号した文字列は使用した後再利用されず必要となるたびに復号されるため、メモリ上の文字列検索やメモリダンプなどを取得してもContiランサムウェアが使用する文字列をまとめて得ることは出来ません。

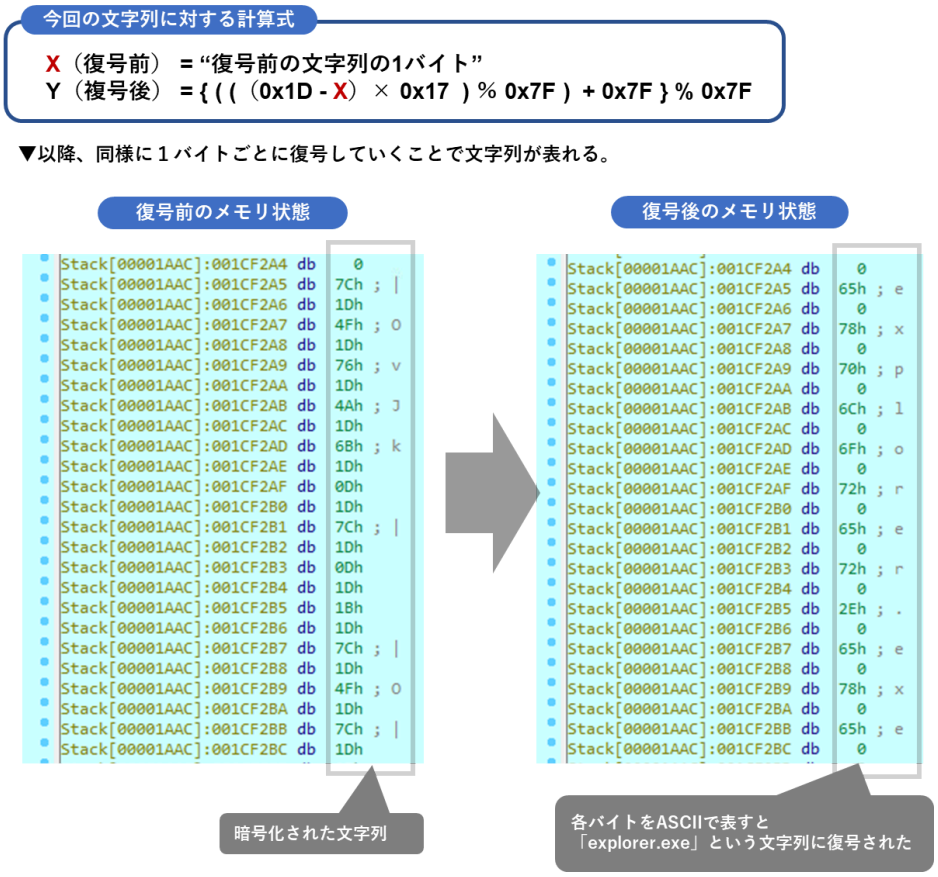


図 33　復号された前後の文字列の様子

上記でも述べた通り、Contiランサムウェアは文字列の暗号化に共通の関数を使用しておらず、全て異なる計算で暗号化しています。
例えば、別のある文字列では以下のような計算で暗号化と復号を行っています（下図）。
下図で割り出した計算式をさきほど上で解説した計算式と比較すると計算式が異なることがわかります。つまり、全ての文字列に対して個別の暗号計算を用いることで共通の復号ロジックを利用して割り出すような解析手法を妨害しており、非常に手が込んでいるといえるでしょう。
なお、以下の図の計算処理では「__ProviderArchitecture」という文字列に復号されます。

別の文字列の復号処理の例

別の文字列の復号計算が以下だが、先ほどとは異なる計算式を用いて文字列を復号していることがわかる。

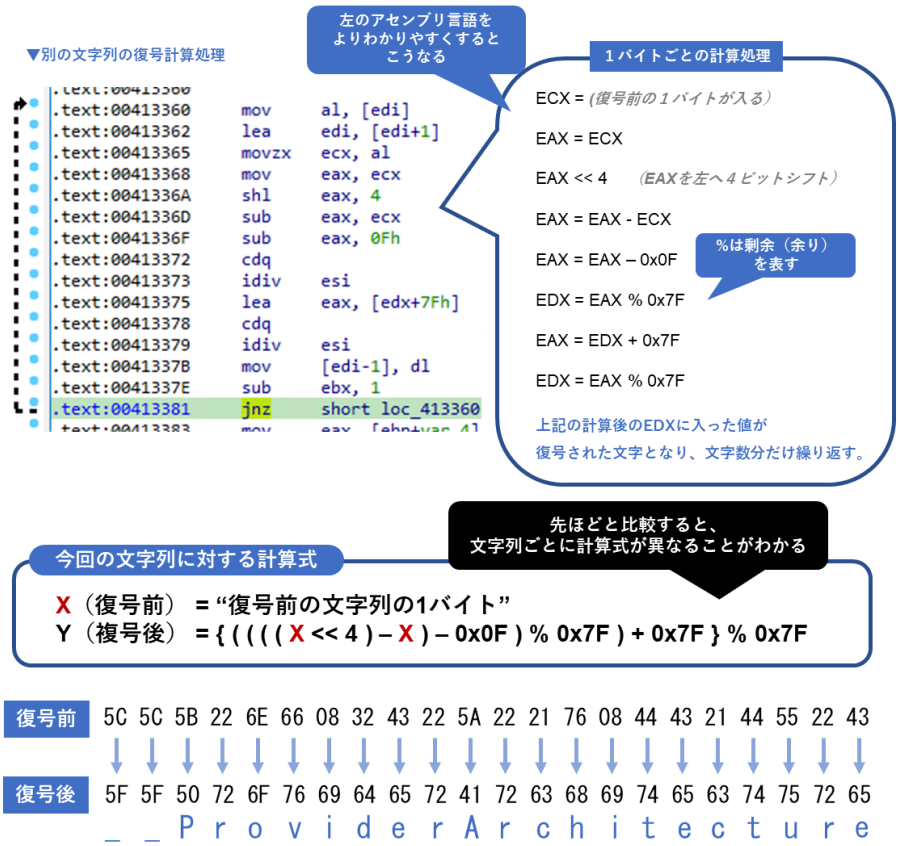


図 34　Contiランサムウェアによる文字列の暗号化例２

以上で解説したように、これから言及していくContiランサムウェアのReflective EXEは呼び出す全てのWindows APIと全ての文字列を暗号化しており、都度復号して使用しています。

Reflective EXEの解説

前置きが長くなりましたが、ここからReflective DLLによって最後に実行されるReflective EXEの処理の解説に入っていきます。

ここまでの解説をまとめると、ContiランサムウェアのEXEファイルは以下のような構造となっており、最終的にメモリ上でのみ実行されるReflective EXEがContiランサムウェアのメイン機能となります。

Contiランサムウェアの全体構造と主要本体の位置

**これまでの解説をまとめると、ContiランサムウェアのEXEファイルは以下のような構造になっており、
最終的にメモリ上で実行される内部のEXEがConriランサムウェアの主要本体（メインのプログラム）となる。**

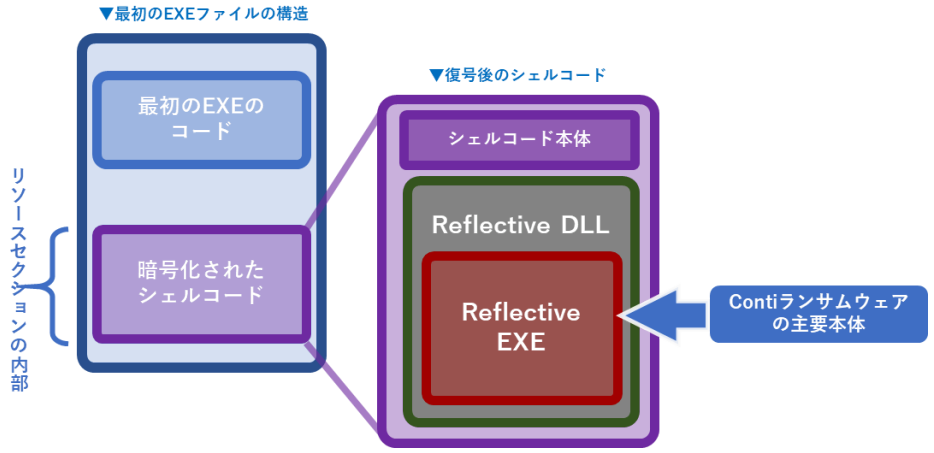


図 35　Contiランサムウェアの全体構造と主要本体の位置

Reflective EXEのバイナリデータからは以下の通り、ランサムウェア開発者の開発環境を示す情報が確認でき、
今回のContiランサムウェアがV3(バージョン３)として開発されたことがわかります。

Contiランサムウェアに含まれるバージョン情報

**メモリ上でのみ実行される最終的なEXEからは、ランサムウェア開発者の開発環境の情報が確認でき、
今回のContiランサムウェアはV3（バージョン３）として開発されたことがわかる。**

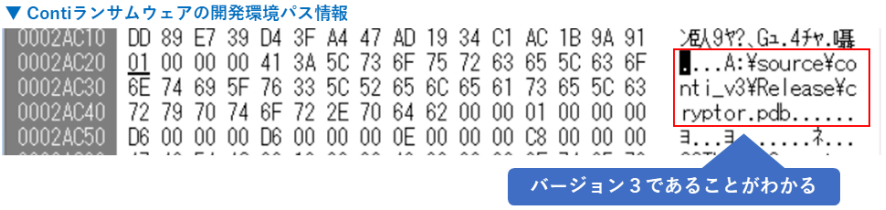


図 36　Contiランサムウェアに含まれるバージョン情報

Reflective EXEは実行されるとすぐにコマンドライン引数をチェックします。
以下はコマンドライン引数のチェック処理を抜粋したものですが、複数のコマンドライン引数により処理が細かく分岐します（以下図）。

コマンドライン引数の確認

以下の処理によりコマンドラインを確認する。
比較するコマンドライン引数の文字列もそれぞれ個別の暗号化処理により暗号化している。

▼ 起動時のコマンドライン引数のチェック処理

```
45  Argv = pCommandLineToArgvW(lpCmdLine, &pNumArgs);
46  copied_Argv = Argv;
47  if ( Argv )
48  {
49    encoded_string_hyphen_p[0] = 0;
50    encoded_string_hyphen_p[1] = 125;
51    encoded_string_hyphen_p[2] = 15;
52    encoded_string_hyphen_p[3] = 63;
53    encoded_string_hyphen_p[4] = 15;
54    encoded_string_hyphen_p[5] = 15;
55    encoded_string_hyphen_p[6] = 15;
56    str_hyphen_p = func_decode_string_hyphen_p(encoded_string_hyphen_p);// -p
57    v37 = 0;
58    qmemcpy(v38, "UPNPPP", sizeof(v38));
59    option_p_detail = (unsigned __int16 *)func_get_detailed_param(pNumArgs, copied_Argv, (int)str_hyphen_p);// -p additional_param
60    str_pyhen_m = func_decode_string_hyphen_m(&v37);// -m
61    encoded_string_hyphen_log[0] = 0;
62    encoded_string_hyphen_log[1] = 8;
63    encoded_string_hyphen_log[2] = 5;
64    option_m_detail = func_get_detailed_param(pNumArgs, copied_Argv, str_pyhen_m);// -m additional_param
65    encoded_string_hyphen_log[3] = 63;
66    encoded_string_hyphen_log[4] = 5;
67    encoded_string_hyphen_log[5] = 114;
68    encoded_string_hyphen_log[6] = 5;
69    encoded_string_hyphen_log[7] = 105;
70    encoded_string_hyphen_log[8] = 5;
71    encoded_string_hyphen_log[9] = 5;
72    encoded_string_hyphen_log[10] = 5;
73    str_hyphen_log = func_decode_string_hyphen_log(encoded_string_hyphen_log);// -log
74    encoded_string_hyphen_size[0] = 0;
75    encoded_string_hyphen_size[1] = 34;
76    encoded_string_hyphen_size[2] = 84;
77    encoded_string_hyphen_size[3] = 55;
78    encoded_string_hyphen_size[4] = 84;
79    encoded_string_hyphen_size[5] = 52;
80    encoded_string_hyphen_size[6] = 84;
81    encoded_string_hyphen_size[7] = 19;
82    encoded_string_hyphen_size[8] = 84;
83    encoded_string_hyphen_size[9] = 0;
84    qmemcpy(v32, "TTT", sizeof(v32));
85    option_log_filepath = func_get_detailed_param(pNumArgs, copied_Argv, str_hyphen_log);// -log additional_param
86    str_hyphen_size = func_ecode_string_hyphen_size(encoded_string_hyphen_size);// -size
87    encoded_string_hyphen_nomutex[0] = 0;
88    encoded_string_hyphen_nomutex[1] = 78;
89    encoded_string_hyphen_nomutex[2] = 107;
90    encoded_string_hyphen_nomutex[3] = 22;
91    option_size_detail = func_get_detailed_param(pNumArgs, copied_Argv, str_hyphen_size);// -size additional_param
92    encoded_string_hyphen_nomutex[4] = 107;
93    encoded_string_hyphen_nomutex[5] = 27;
94    encoded_string_hyphen_nomutex[6] = 107;
95    encoded_string_hyphen_nomutex[7] = 17;
96    qmemcpy(v27, "k9k4khkHkkk", sizeof(v27));
97    str_hyphen_nomutex = func_decode_string_hyphen_nomutex(encoded_string_hyphen_nomutex);// -nomutex
98    bOptionNomutex = func_check_option_nomutex_contains_or_not(str_hyphen_nomutex) != 0;// -nomutex is specified or not
99    if ( option_m_detail )
```

[-p]オプションの文字列復号処理
[-m]オプションの文字列復号処理
[-m]オプションの第２引数の処理
[-log]オプションの文字列復号処理
[-size]オプションの文字列復号処理
[-nomutex]オプションの文字列復号処理

図 37　コマンドライン引数の確認処理

上記の図にあるコマンドライン引数は以下の意味を持ちます。

- -p：特定のフォルダのみ暗号化する際にパスを指定
- -m：暗号化する範囲（ローカルやネットワーク等）の指定
- -log：失敗した処理のログ出力を指定
- -size：大きいファイルに対し分割して暗号化する際の割合に関する設定
- -nomutex：ミューテックスを作成しない

ここで注目すべきなのは本ランサムウェアにこうした手動操作による実行を示唆するコマンドライン引数が用意されている点です。つまり、単体で実行させるだけでなく、人の手による暗号化のための道具としての利用も考慮して開発されたランサムウェアであることがわかります。
これらのコマンドライン引数はあくまでオプションであるため、何も引数を渡さずに実行すると通常のランサムウェアとしての挙動を行います。

例えば、[-log]引数をつけて実行することで、以下の図のように実行ログが出力されます（下図）。
実行ログには主に暗号化に失敗したファイルが出力されるため、攻撃者がシステムに侵入しランサムウェアを展開した後、リアルタイムに作業成功状況を把握する目的などが垣間見えます。（もちろん[-log]オプションを付けない場合は何も出力されません。）

実行ログの出力に対応しているContiランサムウェア

[-log]引数を付けて実行することにより、以下のように実行ログが出力される。

▼ [-log]引数を付けた場合のみContiランサムウェアが出力する「CONTI_LOG.txt」

図 38　実行ログの出力に対応しているContiランサムウェア

以降では、何もコマンドライン引数が付与されずに実行された場合の挙動を解説していきます。

多重感染防止

Reflective EXEは特定のミューテックス（排他制御のための仕組み）を作成することでContiランサムウェアのプロセスが多重起動（多重感染）しないように防止します。
つまり、該当のミューテックスが既にシステムに作成されていた場合、Contiランサムウェアは何もせずに終了します。
なお、[-nomutex]の引数をつけて起動された場合はミューテックスを作成しません。

多重起動の防止

特定のミューテックスを作成することでContiランサムウェアのプロセスが多重起動しないように防止する。
つまり、該当のミューテックスが既にシステムに作成されていた場合はContiランサムウェアは何もせずに終了する。

もし[-nomutex]の引数をつけて起動された場合は、ミューテックスを作成しない。

▼ Contiランサムウェアがミューテックスを確認する処理

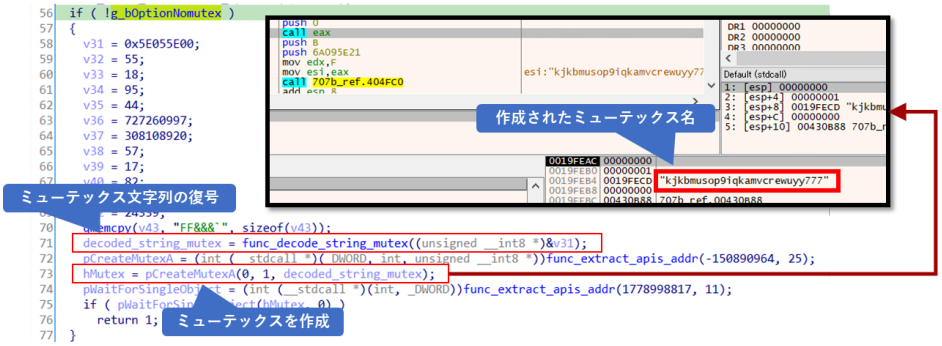


図 39　多重起動の防止処理

Reflective EXEによるファイル暗号化

ここからは、Contiランサムウェアのメイン機能であるReflective EXEによるファイル暗号化について解説していきます。

まずはファイル暗号化の全体処理の流れを把握いただくために一つの図にまとめました（下図）。
Contiランサムウェアは一つ一つのファイルに対し、以下の順序で処理を行うことでファイルを暗号化していきます。
下図に記載した個々の処理やキーワードについては、以降で詳細に解説していきます。

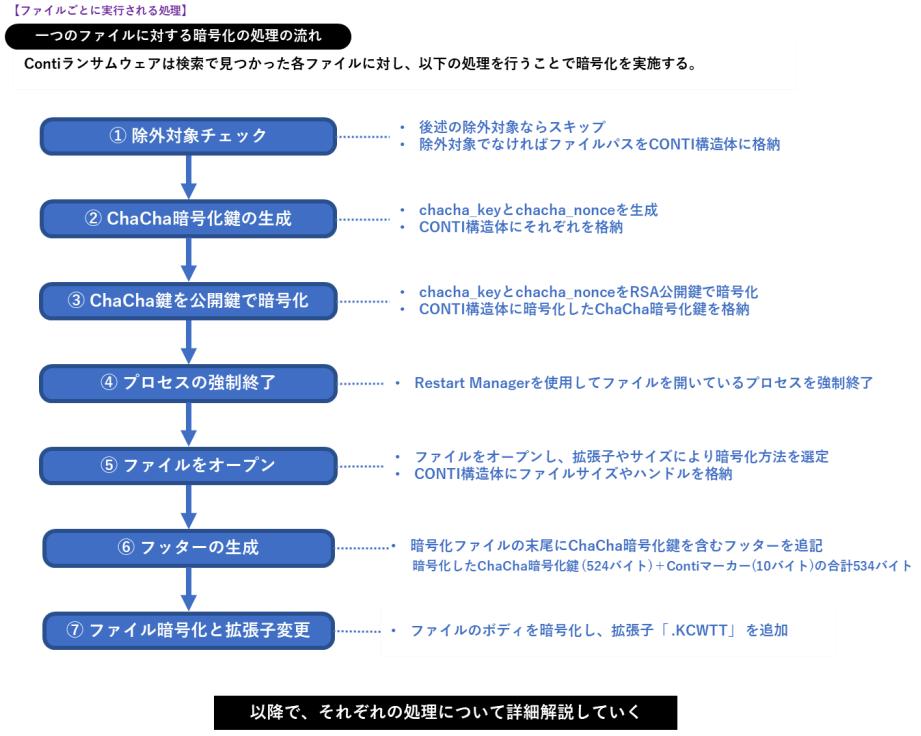


図 40　一つのファイルに対する暗号化の処理の流れ

まず、(上図)「①除外対象チェック」ですが、Contiランサムウェアは以下の図に記載した文字列を含むフォルダやファイルを除外対象とし暗号化を行いません。いずれの文字列もStrStrIW関数を用いて比較されるため、これらの文字列がファイルパスの一部に含まれるだけで除外されます（下図）。

暗号化から除外されるフォルダおよびファイル一覧

Contiランサムウェアは以下の文字列を含むフォルダやファイルを除外対象とし暗号化を行わない。

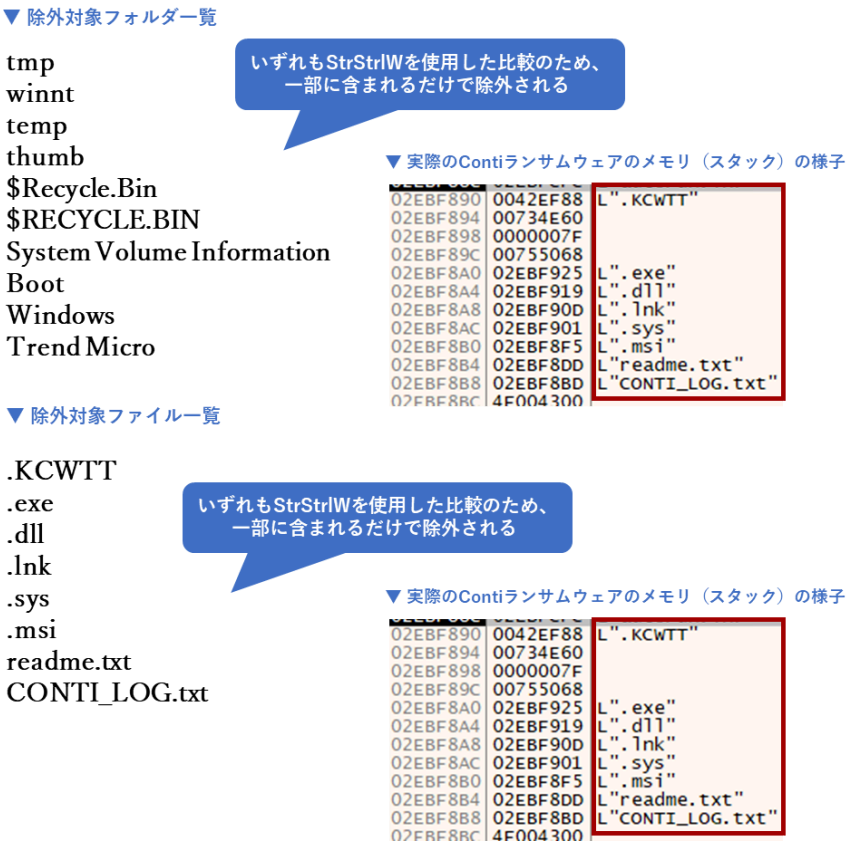


図 41　暗号化から除外されるフォルダおよびファイル一覧

のちほど詳しく触れますが、Contiランサムウェアはファイルの拡張子やサイズによって暗号化の手法を変化させる動きがあるため、それらの拡張子についても先にまとめてご紹介しておきましょう。

まず以下の図にあげる拡張子はどのようなサイズであってもサイズチェックなしに全体を暗号化します。これはつまり、Contiランサムウェアの攻撃者が明確に暗号化すべきだと考えている対象ファイルのリストと言い換えることができるでしょう。データベースに関連した拡張子が多いことから比較的大きな組織やファイルサーバ等の重要なデータが一元管理されている場所への攻撃を念頭に置いている事が伺い知れます。

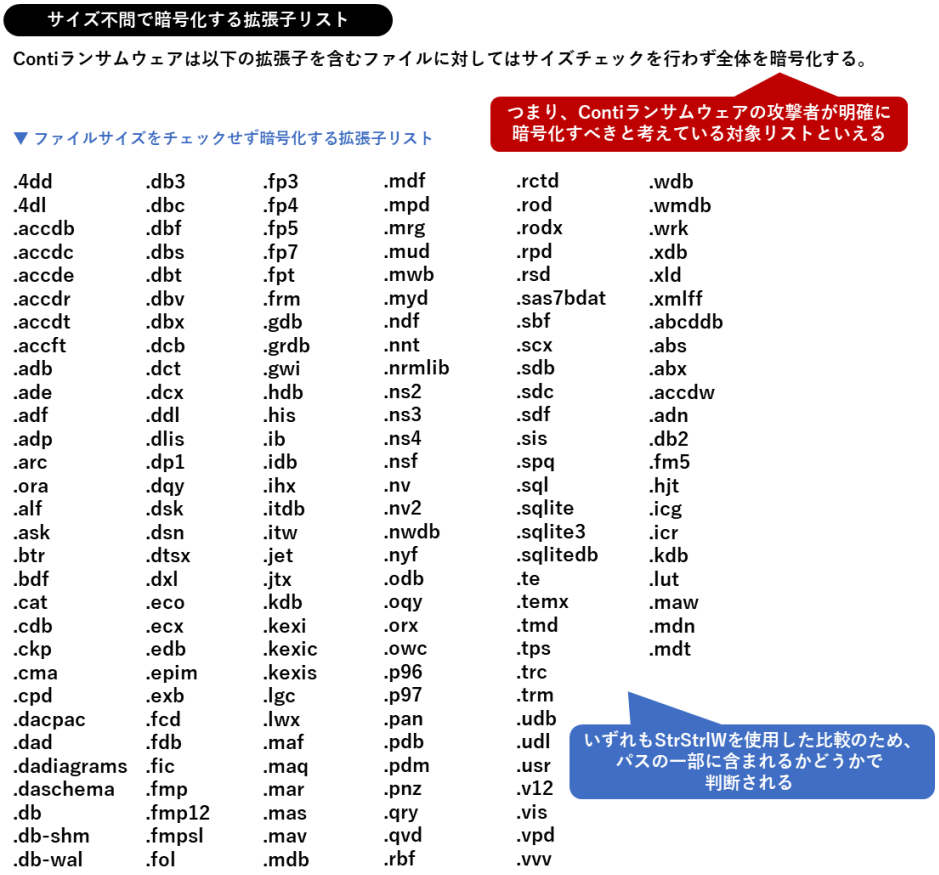


図 42　サイズ不問で暗号化する拡張子リスト

また以下に挙げる拡張子リストは、ファイルサイズをチェックせずとも無条件に拡張子だけでサイズが大きいと推定し、常にファイルサイズが大きい場合に用意された暗号化処理を行います。
後述しますがファイルサイズが大きい場合に用意された暗号化処理ではファイルの一部の暗号化を省略するため、場合によってはデータが全く暗号化されないケースがあります。

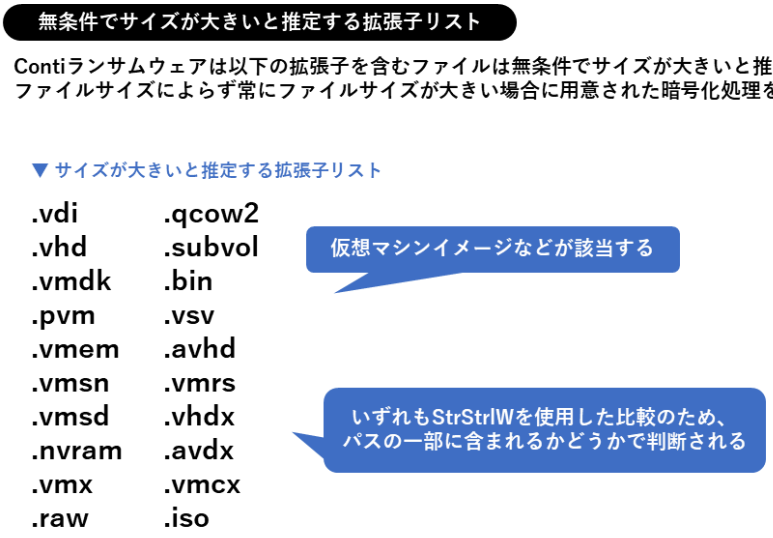


図 43　無条件でサイズが大きいと推定する拡張子リスト

そのケースの具体例をあげると、上記のリストに含まれる[.vmdk]という拡張子を、実際はファイルサイズが小さなテキストファイルにつけたとします。すると場合によっては、暗号化された後のファイルの中身を見ても、以下の図のように元のファイルデータが暗号化されずに残る場合があります。これはContiランサムウェアの開発者が”上記の拡張子を持つファイルはサイズが大きいだろう”と大雑把にコーディングしてしまった設計ミスとも言えるでしょう。ただし、確かに上記の拡張子を持つファイルにおいてサイズが小さいケースは実際稀であり、攻撃者にとって致命的と言えるようなバグではありません。

Contiランサムウェアの設計ミス

**Contiランサムウェアは一部のファイルを拡張子だけで判断してしまうため、
サイズが大きいと推定される拡張子を持つがサイズが小さいファイルがあった場合、
暗号化しないケースが稀に発生する。**

▼ vmdkの拡張子にしたサイズが小さなテキストファイルの暗号化前後の様子

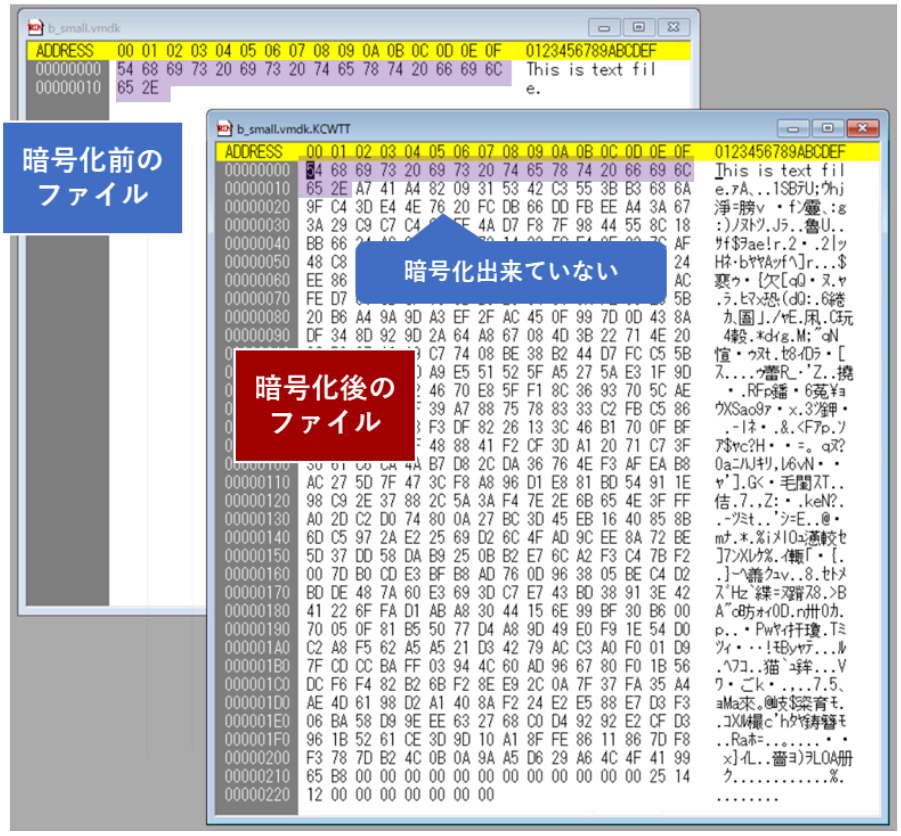


図 44　Contiランサムウェアの設計ミス

構造体で管理するファイル暗号化

Contiランサムウェアは、暗号化対象ファイルを独自の構造体（以降でCONTI構造体と呼ぶ）で扱うことで暗号化処理を効率化しています。
CONTI構造体には、一つのファイルを暗号化するために必要な様々な情報が格納され、各暗号化処理で必要に応じて参照されます。
以下の図は静的解析の結果から再現したCONTI構造体の定義とその役割となります（下図）。

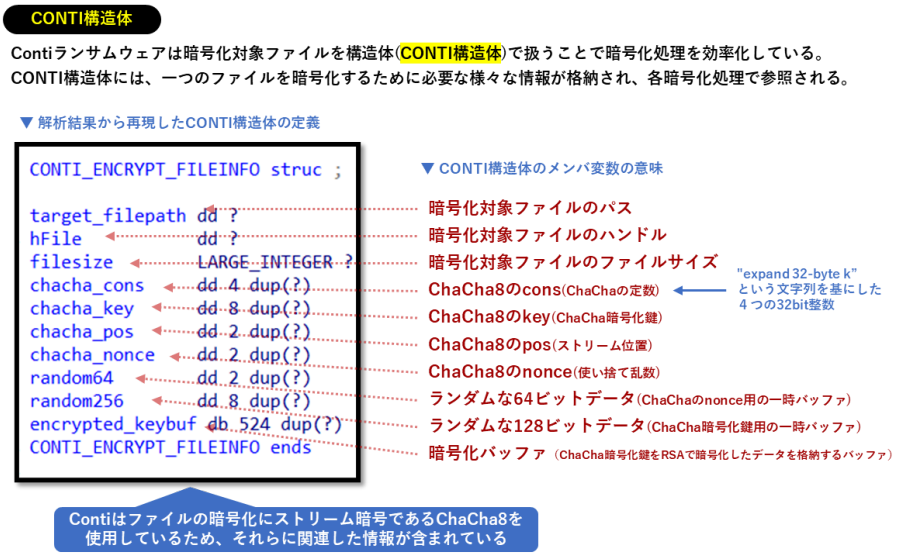


図 45 CONTI構造体

上のCONTI構造体を見ても分かる通り、Contiランサムウェアはファイルの暗号にChaCha暗号と呼ばれる暗号化方式を使用します。以下はChaCha暗号の処理に該当する処理部分ですが、ChaCha暗号の特徴であるローテート操作で使用される定数が確認できます。

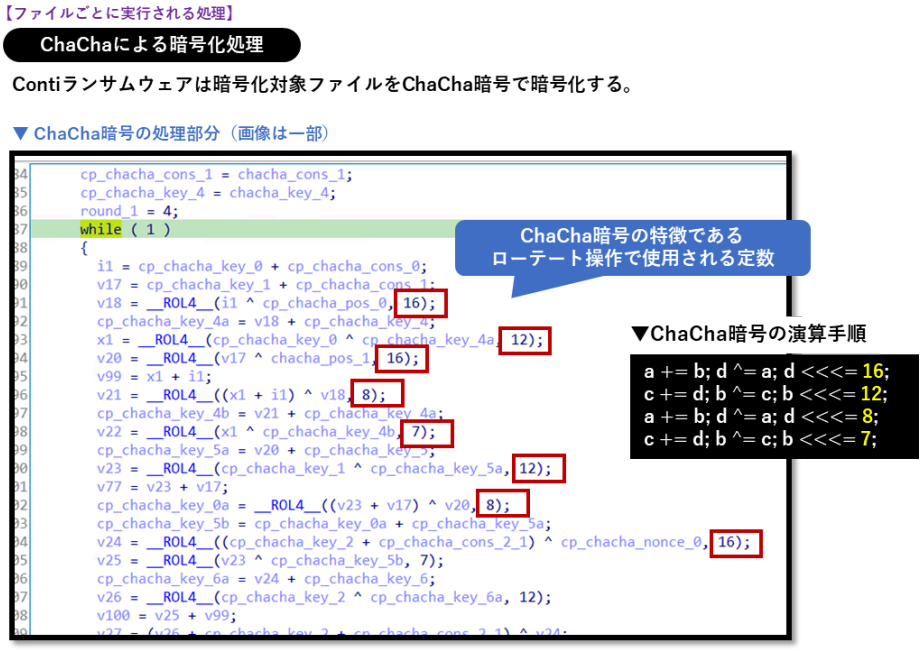


図 46　ChaChaによる暗号化処理

ChaCha暗号はラウンドと呼ばれる処理の回数により複数の方式が存在しますが、Contiランサムウェアは ChaCha８を使用します。
以下はContiランサムウェアのChaCha暗号の全体処理を示した図ですが、２つのラウンド処理（通称double-round と呼ばれる）が４回のループで処理されるため、2×4=8ラウンド、つまり、ChaCha8であることがわかります（下図）。

【ファイルごとに実行される処理】

暗号化にはChaCha８が使用されている

２つのラウンド処理(double-round)が４回のループで処理されるため、ChaCha8(2×4=8ラウンド)であることがわかる。

▼ contiランサムウェアによるChaCha8による暗号化における全体処理

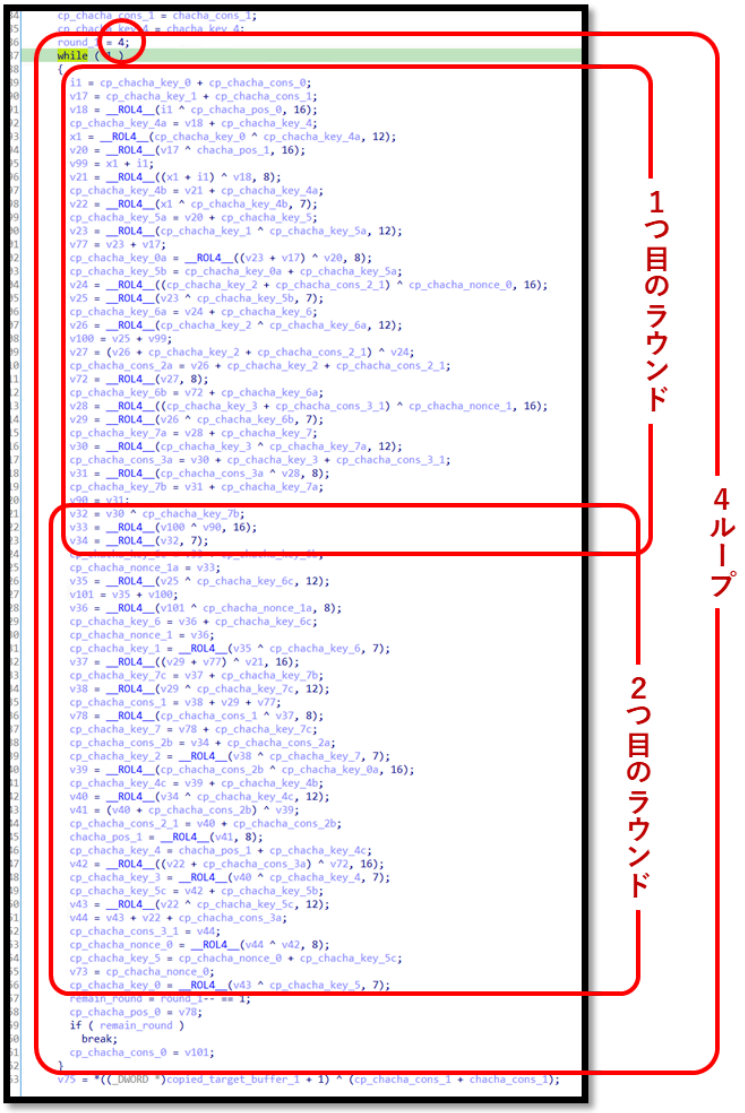


図 47

暗号化に使用されているChaCha8

Contiランサムウェアは暗号化対象ファイルを前述の通り、ChaCha暗号で暗号化し、その鍵であるChaCha暗号鍵をRSA公開鍵で暗号化して、暗号化したファイルの末尾に追加します（下図）。
そのため、ファイルを暗号化する前にRSA公開鍵を使用するための準備作業に入ります。

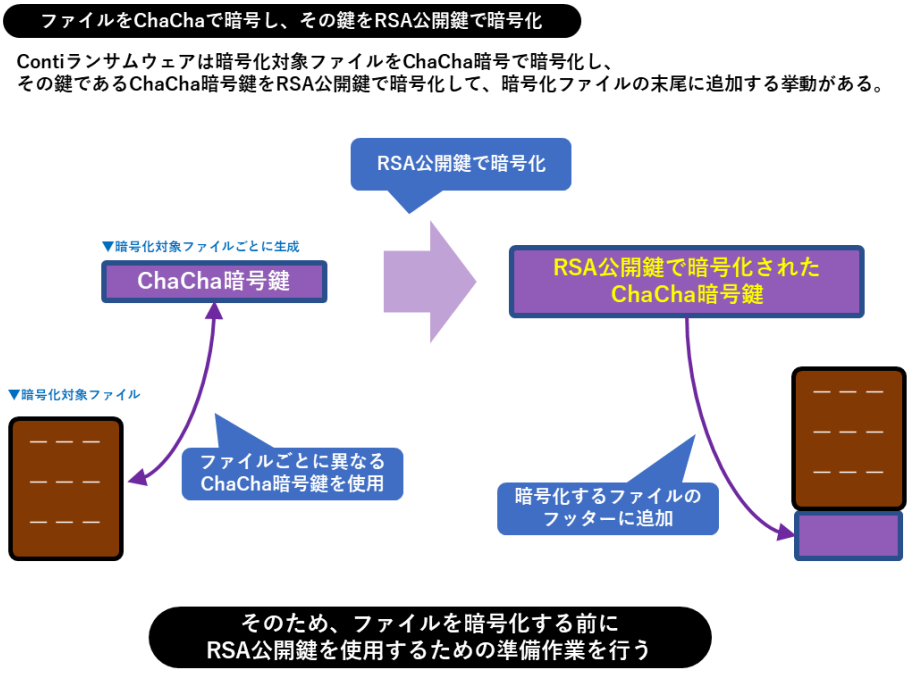


図 48 ファイルをChaChaで暗号化し、ChaCha鍵をRSA公開鍵で暗号化

RSA公開鍵を使用するためにContiランサムウェアは、CSP（暗号化サービスプロバイダ）と呼ばれるWindowsの暗号化エンジンを利用します。
その際、暗号化方式を指定する文字列（暗号化プロバイダー名）として「Microsoft Enhanced RSA and AES Cryptographic Provider」という文字列を使用しますが、例によってこの文字列は固有の暗号計算で暗号化されており復号して使用します（下図）。
復号した暗号化プロバイダー名をCryptAcquireContextA関数に渡すことで鍵の格納場所を示す「キーコンテナ」のハンドルを取得します。

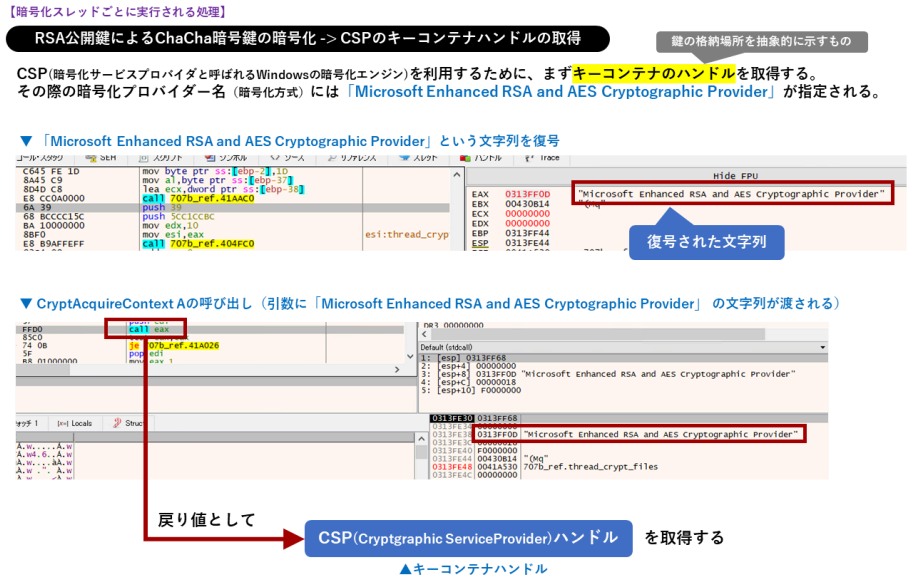


図 49 CSPのキーコンテナハンドルの取得

RSA公開鍵のバイナリデータ(0x1000バイト)はContiランサムウェアのReflective EXEにハードコーディングされています。CryptImportKey関数の引数にRSA公開鍵のバイナリデータを渡すことでインポートし、RSA公開鍵の「キーハンドル」を取得します（下図）。
これでRSA公開鍵を使用する準備が整いました。

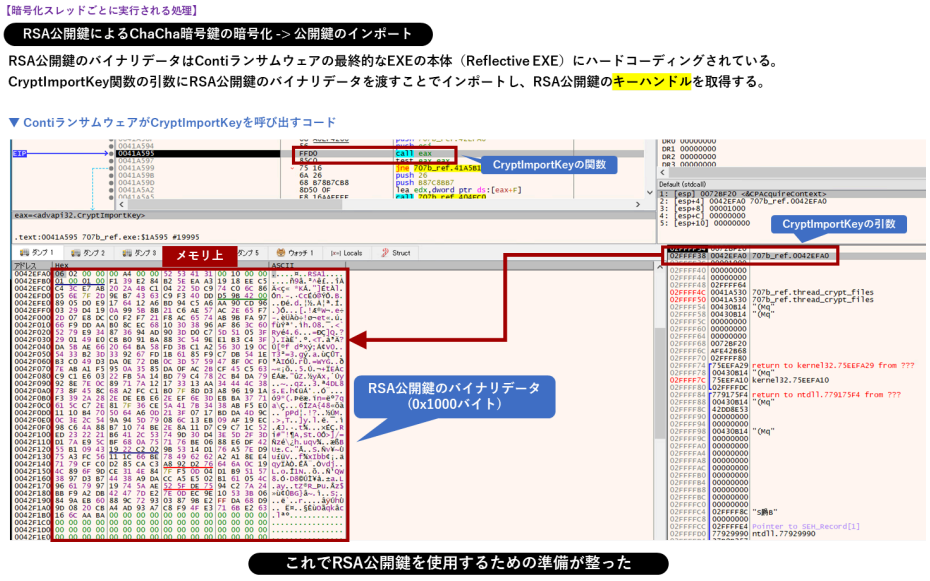


図 50　RSA公開鍵のインポート

Contiランサムウェアは暗号化するファイルパスを格納したCONTI構造体と、先程取得した各情報（RSA公開鍵のキーコンテナハンドルとキーハンドル）を引数にして、独自のファイル暗号化関数を呼び出します（下図）。

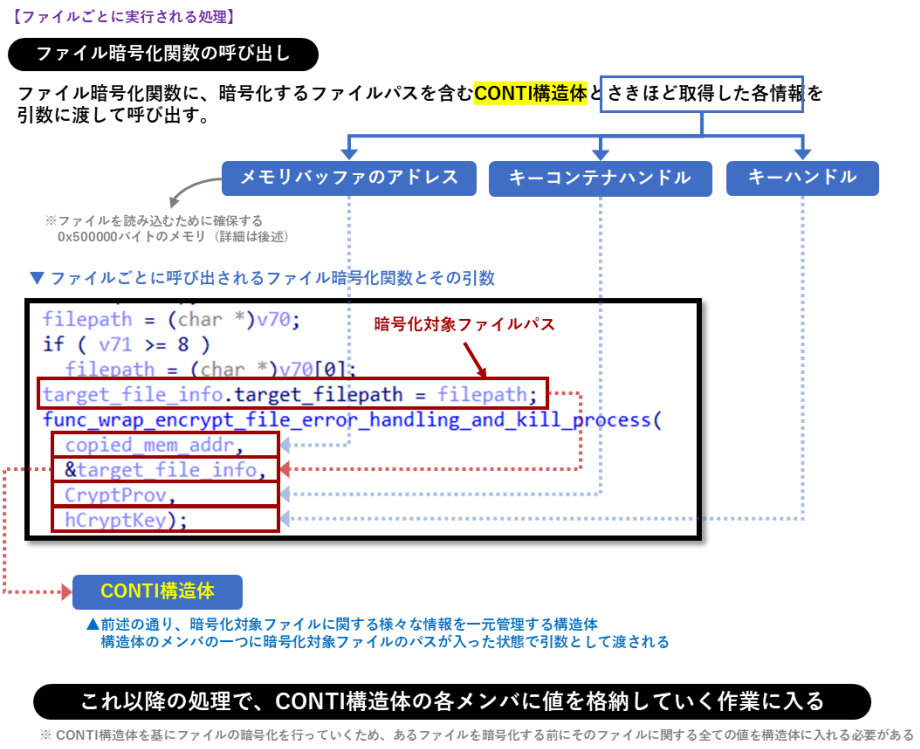


図 51　ファイル暗号化関数の呼び出し

この時点では、CONTI構造体のメンバ変数にはファイルパスのみが格納されている状況ですが、ファイルの暗号化は最終的にCONTI構造体のメンバ変数が全て埋まった状態で開始される必要があるため、これ以降で他のメンバ変数（ChaCha暗号鍵などの情報）を埋めていく作業に入ります。

Contiランサムウェアは一つ一つのファイルを個別のChaCha暗号鍵で暗号化するため、その作業に最も必要なChaCha暗号鍵に関わるデータを生成する処理に移っていきます。

まず、CryptGenRandom関数を用いて、64ビット、256ビットのランダムなデータを生成します。
生成したランダムデータは一度CONTI構造体のメンバ変数である[random64]と[random256]に一時的に格納された後、[chacha_nonce]と[chacha_key]というCONTI構造体のメンバ変数にそれぞれの値が格納されます。つまり、生成したこれらのランダムデータはそれぞれChaCha暗号の際に使用する"使い捨て乱数"と"ChaCha暗号鍵"に相当します。

さらに、ChaCha暗号で用いられる定数として知られる"expand 32-byte k"という文字列もCONTI構造体のメンバ変数である[chacha_cons]へ格納します（下図）。

図 52　CONTI構造体へのデータ格納１

その後、 [chacha_key]（ChaCha暗号鍵）と[chacha_nonce]（使い捨て乱数）をCONTI構造体のメンバである[encrypted_keybuf]に一時的にコピーした後、RSA公開鍵を使用してCryptEncrypt関数で[encrypted_keybuf]のデータを暗号化します。
つまり、個々のファイル暗号化で使用するChaCha暗号鍵をRSA公開鍵で暗号化していることを意味します（下図）。

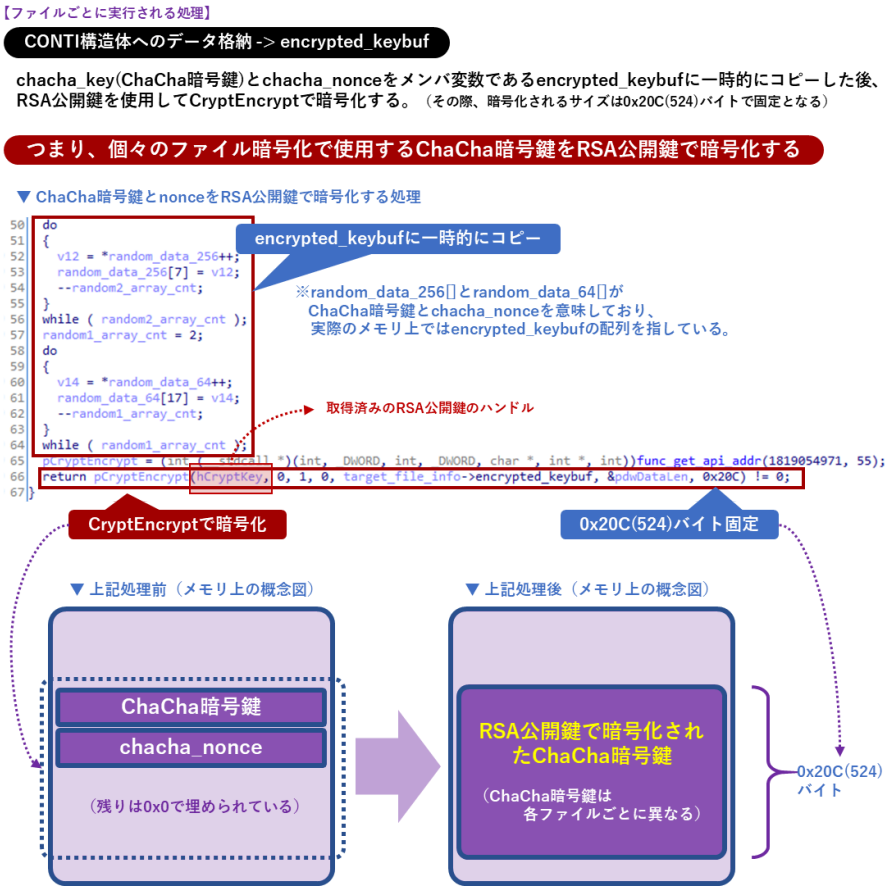


図 53　CONTI構造体へのデータ格納２

そして最後に、暗号化対象ファイルをCreateFileWで開きファイルハンドルをCONTI構造体のメンバ変数[hFile]に格納、GetFileSizeExで取得したファイルサイズを残るメンバ変数である[FileSize]に格納します（下図）。

ここまでの処理により、一つのファイルに対応するCONTI構造体の全ての値が埋まった状態となります。

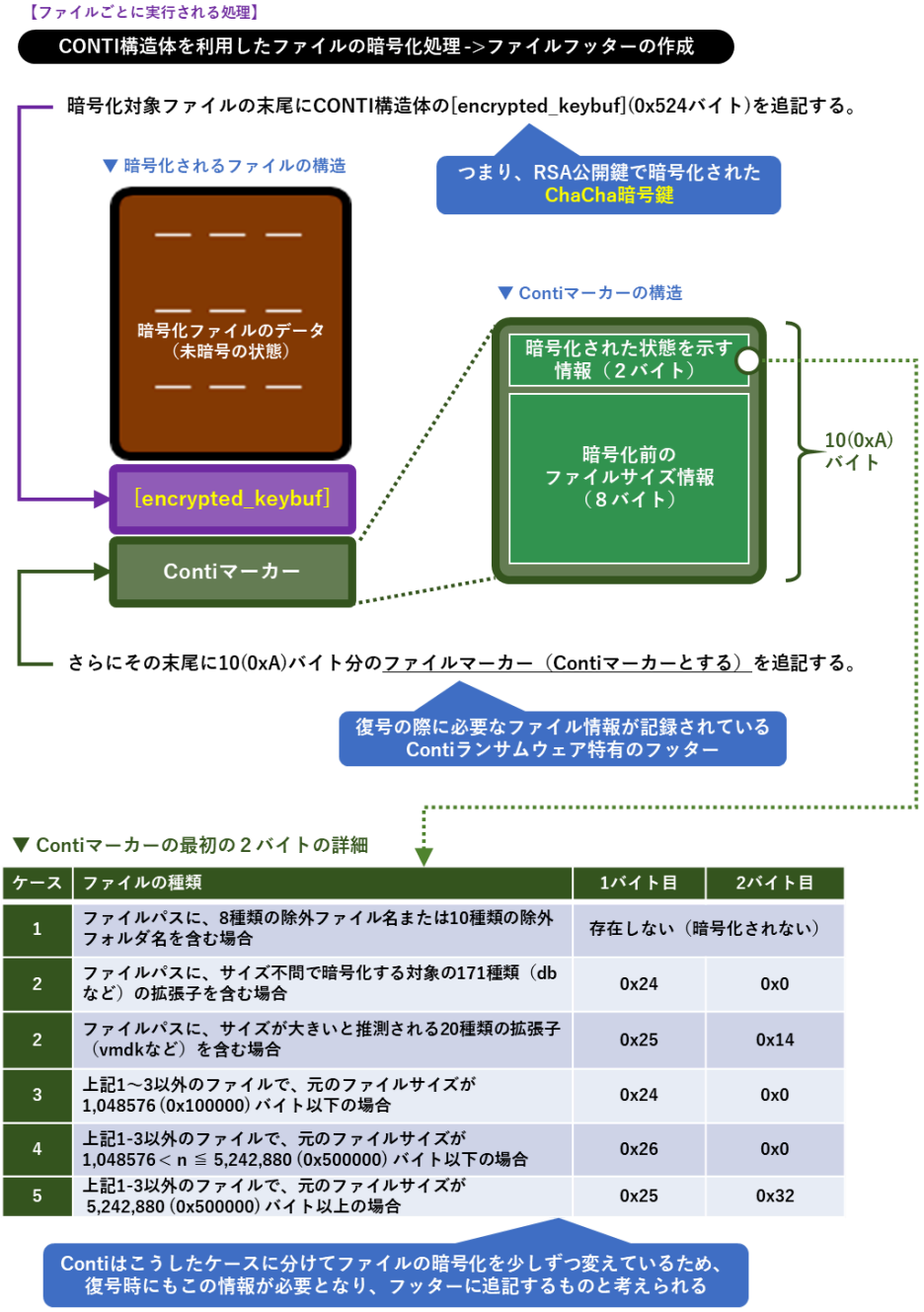
図 54　CONTI構造体へのデータ格納３

以上の処理で、一つのファイルを暗号化するための準備が整いました。
ここから実際の暗号化処理に入っていきます。

Contiランサムウェアははじめに、暗号化対象ファイルの末尾にCONTI構造体の[encrypted_keybuf]、つまりRSA公開鍵で暗号化したChaCha暗号鍵のデータを追記します。
そしてさらにその後ろに、復号の際に必要なファイル情報が記載されたContiランサムウェア特有のフッター（他のランサムウェアに見られるような固有の文字列は含みませんがConti特有であるため、以降ではこのフッターを"Contiマーカー"と呼びます）を追記します（下図）。

Contiマーカーは10(0xA)バイトの固定サイズであり、以下の図にまとめたように「暗号化された状態を示す情報」である２バイトと、「暗号化前のファイルサイズ情報」である８バイトで構成されています。

Contiランサムウェアは後述の通り、ファイルサイズや拡張子などで暗号化方式を選別しているため、ファイルを復号する際もどのようなロジックで暗号化されたファイルなのかを知る必要があります。そのために、CONTIマーカーの最初に２バイトにそれを示すデータを含ませているわけです（下図）。



ケース	ファイルの種類	1バイト目	2バイト目
1	ファイルパスに、8種類の除外ファイル名または10種類の除外フォルダ名を含む場合	存在しない（暗号化されない）	
2	ファイルパスに、サイズ不問で暗号化する対象の171種類（dbなど）の拡張子を含む場合	0x24	0x0
2	ファイルパスに、サイズが大きいと推測される20種類の拡張子（vmdkなど）を含む場合	0x25	0x14
3	上記1～3以外のファイルで、元のファイルサイズが1,048576 (0x100000) バイト以下の場合	0x24	0x0
4	上記1-3以外のファイルで、元のファイルサイズが1,048576 < n ≦ 5,242,880 (0x500000) バイト以下の場合	0x26	0x0
5	上記1-3以外のファイルで、元のファイルサイズが5,242,880 (0x500000) バイト以上の場合	0x25	0x32

図 55　ファイルフッターの作成

上でContiランサムウェアがファイルの種類やサイズにより暗号化方法を選別すると言及しましたが、具体的には以下のような状況に分かれます（下図）。
つまりはファイルサイズや種類などで分類して暗号化の効率（処理スピード）を上げていると考えられます。

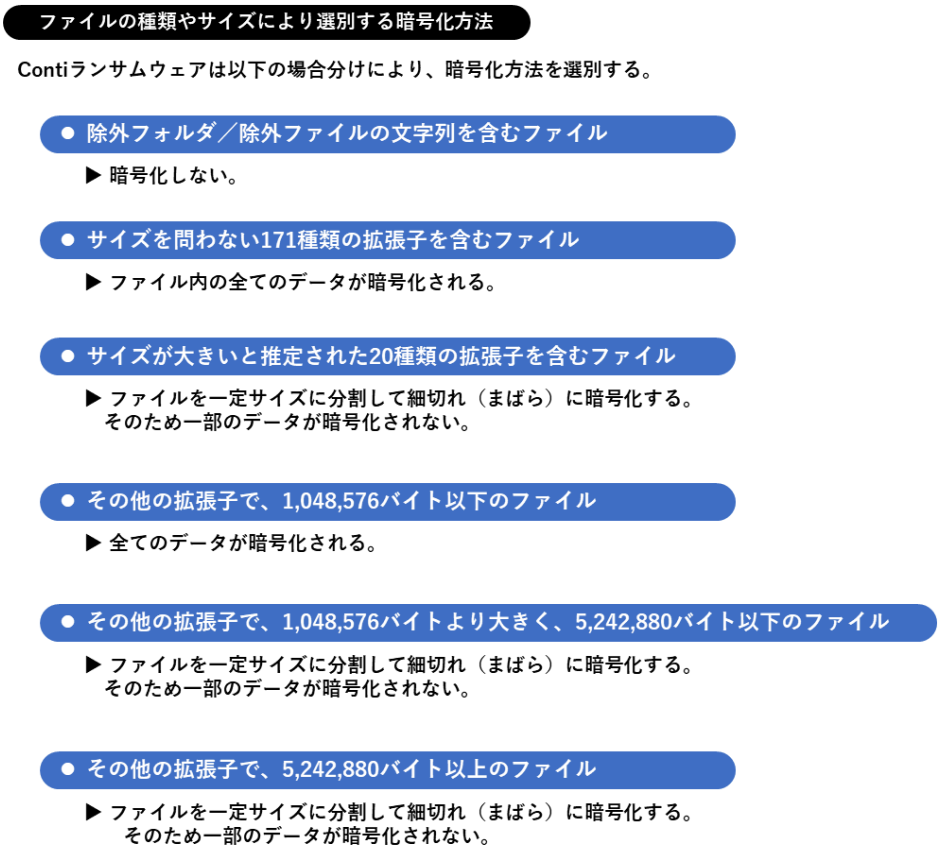


図 56　ファイルの種類やサイズにより選別する暗号化方法

ここまでの処理ではまだフッターが追記されただけでファイルのボディ部分は暗号化されていません。以降でファイルのボディ部分の暗号化に入ります。

まず、あらかじめVirtualAlloc関数により確保していたメモリ領域に、ReadFile関数で暗号化対象ファイルを読み込みます。

Contiランサムウェアは読み込んだメモリ領域（つまりファイルのボディ部分）に対し、CONTI構造体に含まれる各種情報を参照しながらChaCha暗号で暗号化していきます（下図）。
そして暗号化したメモリデータを実ファイルへ上書きすることにより書き込みます。

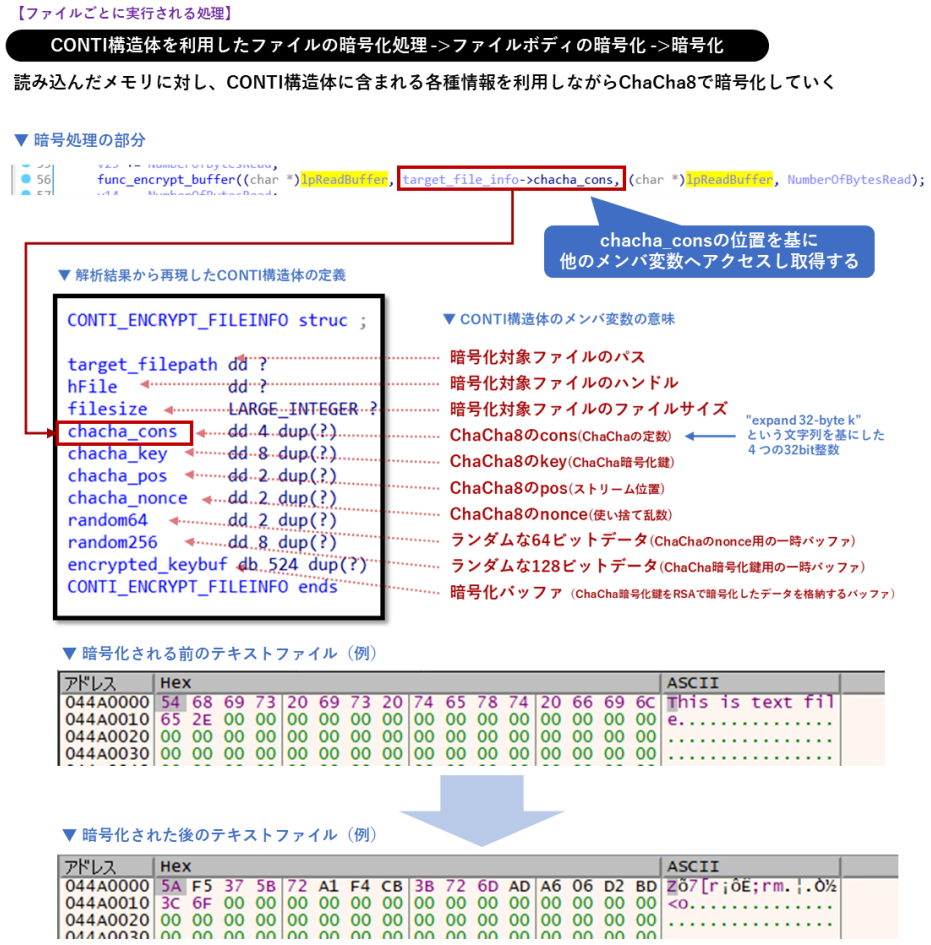


図 57　CONTI構造体を参照して行うファイルボディの暗号化

最後に、暗号化が完了したファイルの拡張子をMoveFileW関数により「＜元の拡張子＞.KCWTT」に変更します（下図）。

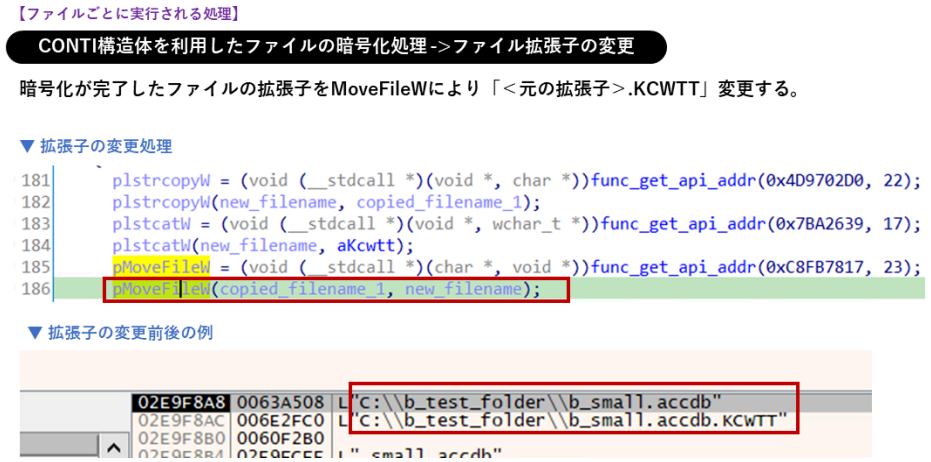


図 58　ファイル拡張子の変更

以上の処理により、Contiランサムウェアに暗号化されたファイルの内部構造は、暗号化前後で比較すると以下のようになります（下図）。
これまで解説した内容と以下の図からもわかる通り、Contiランサムウェアに暗号化されたファイルは元のファイルサイズから534バイト(524+10)分だけファイルサイズが増加します。

Contiランサムウェアにより暗号化されたファイルの内部構造

Contiランサムウェアにより暗号化されたファイルの構造は以下のようになる。

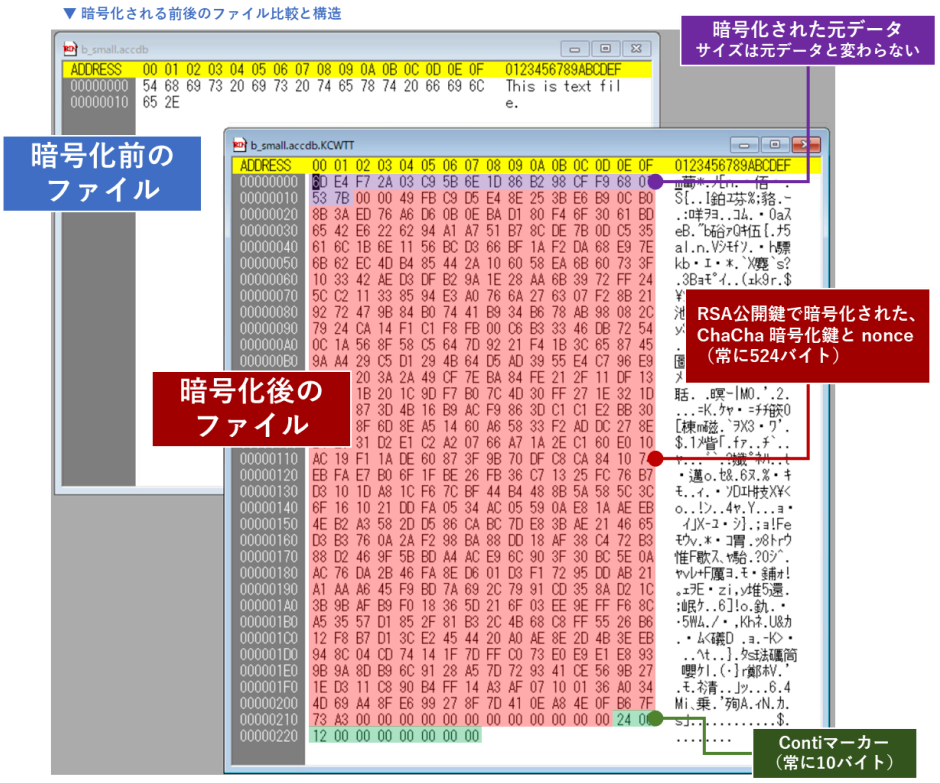


図 59　Contiランサムウェアにより暗号化されたファイルの内部構造

また、一つのファイルを暗号化する際のファイル操作の詳細は以下のような流れとなります(下図)。

【ファイルごとに実行される処理】

（補足）暗号化におけるファイル操作の詳細

前述の⑥と⑦の処理におけるファイル操作の詳細は以下となる。

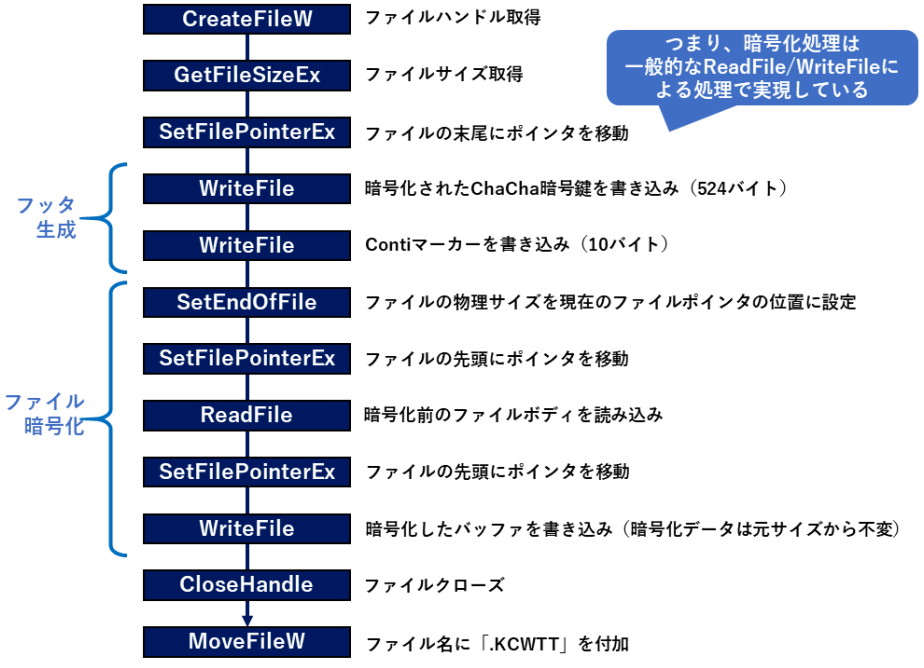


図 60　暗号化におけるファイル操作の詳細

ここまでで、一つファイルに対する暗号化の処理を詳しく解説してきましたが、同様にCONTI構造体を用いながらこのような処理を全てのファイルに対して行っていきます。

また、ファイルを暗号化する際ですが、感染端末のCPUにおけるプロセッサ数を取得し、２倍した数のスレッドを作成し、マルチスレッドで処理を行うことで暗号化作業を高速化します（以下図）。

マルチスレッドで高速化するファイルの暗号化

Contiランサムウェアは、感染PCのCPUにおけるプロセッサ数を２倍した数のスレッドを作成し、マルチスレッドで暗号化を高速化する。

▼ Contiランサムウェアがマルチスレッドを作成する処理

```
1  int __thiscall func_create_encryption_threads(void *this)
2  {
3    int *v1; // esi
4    unsigned int v2; // edi
5    int (__stdcall *pCreateThread)(_DWORD, _DWORD, int (__stdcall *)(int), int *, _DWORD, _DWORD); // eax
6
7    if ( this )
8    {
9      if ( this != (void *)1 )
10       return 0;
11     v1 = &g_allocated_mem_1;
12   }
13   else
14   {
15     v1 = &g_allocated_mem;
16   }
17   v2 = 0;
18   for ( v1[2] = 1; v2 < v1[1]; ++v2 )
19   {
20     pCreateThread = (int (__stdcall *)(_DWORD, _DWORD, int (__stdcall *)(int), int *, _DWORD, _DWORD))func
21     *(_DWORD *)(*v1 + 4 * v2) = pCreateThread(0, 0, func_crypt_files_for_thread, v1, 0, 0);
22   }
23   return 1;
```

プロセッサ数の２倍の数だけループ

スレッドとして作成

暗号化を行う関数

▼ Contiランサムウェアがプロセッサ数を取得して2倍している処理

```
78   if ( g_option_case != 0xE )                    // if "-p" option is Not specified
79   {
80     pGetNativeSystemInfo = (void (__stdcall *)(SYSTEM_INFO *))func_get_api_addr(0xDF1AF05E, 19);
81     pGetNativeSystemInfo(&SystemInfo);
82     unknown_const_val = 0x3C1A22;
83     unknown_const_val = (SystemInfo.wProcessorArchitecture + 3) * (unknown_const_val + 2);
84     while...
85     if ( g_option_case == 0xA )
86       num_of_threads = SystemInfo.dwNumberOfProcessors;
87     else
88       num_of_threads = 2 * SystemInfo.dwNumberOfProcessors;
89     unknown_const_val = 0x3C1A24 * (num_of_threads + 3);
90     while...
91     if ( g_option_case == 0xA || g_option_case == 0xB )
92     {
```

プロセッサ数の２倍

図 61　マルチスレッドで高速化するファイルの暗号化

Windows Restart Managerの悪用

一般的にランサムウェアがファイルを暗号化する際、ファイルが使用中で開かれていると暗号化が行えないため、そのファイルを開いているアプリケーションを閉じる必要がありますが、Contiランサムウェアはアプリケーションを終了させるために非常に効果的な手口であるWindows Restart Manager（再起動マネージャー）と呼ばれるOSの機能を悪用します（下図）。

暗号化時にアプリケーションを閉じる挙動は従来のランサムウェアにもありましたが、従来の手口はあらかじめ強制終了対象のプロセスやサービス名を並べた強制終了リスト（例えばWordやExcel、データベースソフトなど）をハードコーディングしており、それを用いて強制終了関数を呼び出すことで強制終了させる手口が一般的でした。しかし従来のその方法では、強制終了リストに含まれていないアプリケーションを終了することができず、またわかりやすい強制終了リストをランサムウェア内にハードコーディングする必要や、強制終了に関わるわかりやすいWindowsAPIを呼び出す必要があったため、シグネチャ検知や挙動検知のリスクがありました。

Contiランサムウェアが代わりに採用したWindows Restart Managerは、本来その名の通り、起動中のアプリケーションをWindowsOSの再起動時やシャットダウン時に自動的に閉じるために実装されたOSの正規機能です。このWindows Restart Managerを利用することで、ファイルを開いているアプリケーションを全て特定できるようになり、強制終了リストを持たなくてもOSの正規機能を用いて効率的にもれなく終了させることができるようになりました（下図）。
また、わかりやすい強制終了系のWindows APIを使用せずに実現できるメリットがあり、アプリケーションに対しては不審な強制終了ではなく"OSによる再起動もしくはシャットダウン時の終了命令"と誤解させることができるようになります。つまり、一般アプリケーションのみならずセキュリティ製品によっては、不審なプロセスからのTerminateProcessなどの明確なAPIによる強制終了はブロックしてもWindows Restart Mangerによる正規のアプリケーション終了を防御できない可能性が出てきます（ユーザーがWindowsを再起動したりシャットダウンしたりする際に自動的に終了できないアプリケーションは逆に珍しいかもしれません）。
Windows Restart Mangerを悪用するランサムウェアはContiランサムウェアが初めてではありませんが、最近徐々に目立ってきている印象があり、やっかいな手口であると言えるでしょう。

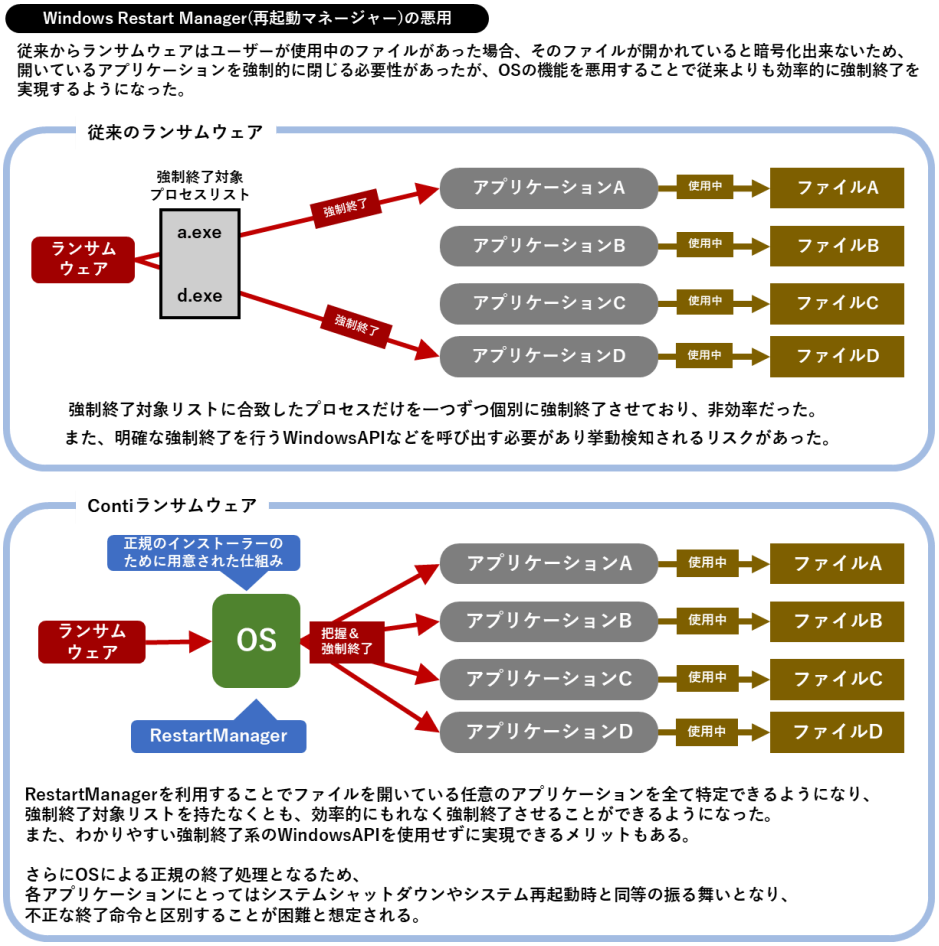


図 62　Windows Restart Managerの悪用

Contiランサムウェアによる Windows Restart Mangerの悪用手順について、さらにより詳細に解説したものが以下の図です。
RmStartSession関数で開始したRestart Managerセッションに、RmRegisterResources関数を用いて暗号化したいファイルのパスを登録します。次に、RmGetList関数を呼び出すことで該当ファイルを使用中のアプリケーション情報をWindowsOSから得ることができます。その後、Contiはファイルを使用中のアプリケーションがexplorer.exeの場合のみ例外的に強制終了から除外するためにexplorer.exe以外のプロセスを選別した後、ファイルを使用中のアプリケーションをRmShutdown関数を介して間接的に強制終了させます（下図）。

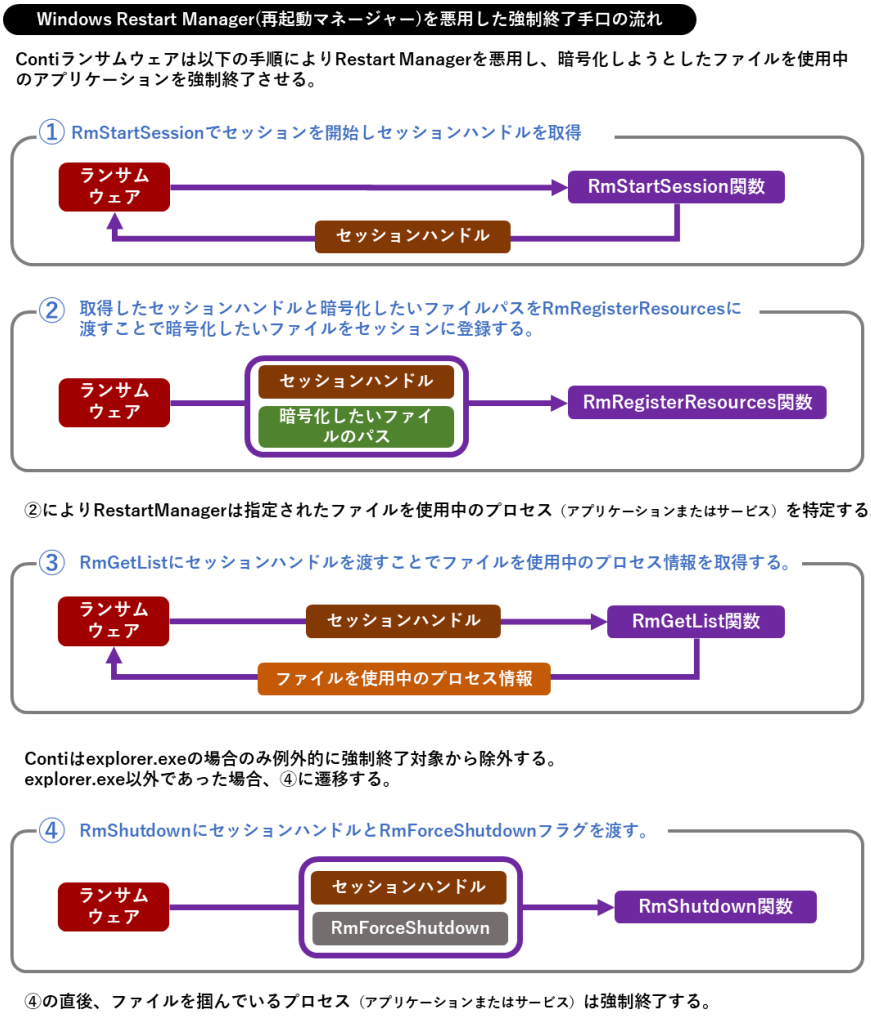


図 63　Windows Restart Managerの悪用の詳細

以下の図はさきほど触れた、explorer.exe以外のプロセスを検索してリスト化する処理の様子です。
Windows Restart Managerによる強制終了からexplorer.exeを除外している背景は、もしexplorer.exeが強制終了させられてしまうと、被害ユーザーがエクスプローラーを介した端末操作ができなくなってしまい脅迫文を開いたり攻撃者へコンタクトを取らせたりする際に障壁となってしまうため、explorer.exeのみWindows Restart Managerによる強制終了の対象から除外しているものと推測します。

Explorer.exe以外のプロセスを検索しリスト化

Contiランサムウェアは Windows Restart Managerによる強制終了対象からエクスプローラーのプロセスを除外するため、explorer.exeを除外したプロセスリストを作成する。

作成したプロセスリストはRmShutdown関数を使用する際に強制終了して良いプロセスかどうかのチェック時に参照される。

▼explorer.exe以外のプロセスを検索しリスト化する処理部分

```
pProcess32First = (int (__stdcall *)(int, PROCESSENTRY32 *))func_get_api_addr(-1844851069, 98);
if ( pProcess32First(copied_hSnapshot, &lppe) )
{
  do
  {
    decoded_explorer_string[0] = 124;
    decoded_explorer_string[1] = 29;
    decoded_explorer_string[2] = 79;
    decoded_explorer_string[3] = 29;
    decoded_explorer_string[4] = 118;
    decoded_explorer_string[5] = 29;
    decoded_explorer_string[6] = 74;
    decoded_explorer_string[7] = 29;
    decoded_explorer_string[8] = 107;
    decoded_explorer_string[9] = 29;
    decoded_explorer_string[10] = 13;
    decoded_explorer_string[11] = 29;
    decoded_explorer_string[12] = 124;
    decoded_explorer_string[13] = 29;
    decoded_explorer_string[14] = 13;
    decoded_explorer_string[15] = 29;
    decoded_explorer_string[16] = 27;
    decoded_explorer_string[17] = 29;
    decoded_explorer_string[18] = 124;
    decoded_explorer_string[19] = 29;
    decoded_explorer_string[20] = 79;
    decoded_explorer_string[21] = 29;
    decoded_explorer_string[22] = 124;
    decoded_explorer_string[23] = 29;
    decoded_explorer_string[24] = 29;
    decoded_explorer_string[25] = 29;
    for ( i = 0; i < 0x1A; ++i )
      decoded_explorer_string[i] = (23 * (29 - (unsigned __int8)decoded_explorer_string[i]) % 127 + 127) % 127;
    plstrcmpiW = (int (__stdcall *)(CHAR *, char *))func_get_api_addr(-684828759, 28);
    if ( !plstrcmpiW(lppe.szExeFile, decoded_explorer_string) )
    {
      array_process = malloc(0xCu);
      *(_QWORD *)array_process = 0i64;
      array_process[2] = 0;
      if ( !array_process )
        break;
      *array_process = lppe.th32ProcessID;
      array_process[1] = 0;
      array_process[2] = this[1];
      *(_DWORD *)this[1] = array_process;
      this[1] = array_process + 1;
    }
    pProcess32NextW = (int (__stdcall *)(int, PROCESSENTRY32 *))func_get_api_addr(-1448982970, 99);
  }
  while ( pProcess32NextW(copied_hSnapshot_1, &lppe) );
  pCloseHandle = (int (__stdcall *)(int))func_get_api_addr(0x
  hSnapshot = pCloseHandle(copied_hSnapshot_1);
```

プロセス列挙を開始

暗号化された"explorer.exe"という文字列

上記の暗号化された文字列を復号

explorer.exe以外の
プロセス情報を配列に格納

次のプロセスを検索

図 64　explorer.exe以外のプロセスをリスト化する処理

RmShutdown関数による強制終了の際、OS再起動やシャットダウン時において応答しないアプリケーションを強制的に終了させるために用意されたパラメーターである「RmForceShutdown」を渡すことで強制終了を実現させます（下図）。

Windows Restart Managerを悪用して強制終了させるContiランサムウェアの処理部分

▼ContiランサムウェアがRmShutdown関数にRmForceShutdown(0x1)を引数に渡して呼び出している処理

```
label_EndSession:
        v25 = pSessionHandle;
        pRmShutdown = (int (__stdcall *)(int, int, _DWORD))func_get_api_addr(19, 583759375, 66);
        v3 = pRmShutdown(v25, 1, 0) == 0;
        j_func_HeapFree(p_mem_malloc);
      }
      pSessionHandle_2 = pSessionHandle;
      pRmEndSession_2 = (void (__stdcall *)(int))func_get_api_addr(19, 2098544741, 62);
      pRmEndSession_2(pSessionHandle_2);
    }
```

ここでOSにより強制終了が行われる

▼RmShutdown関数のパラメータ(Microsoftのサイトから抜粋)

Parameters

`dwSessionHandle`

A handle to an existing Restart Manager session.

`lActionFlags`

One or more RM_SHUTDOWN_TYPE options that configure the shut down of components. The following values can be combined by an OR operator to specify that unresponsive applications and services are to be forced to shut down if, and only if, all applications have been registered for restart.

Value	Meaning
RmForceShutdown 0x1	Force unresponsive applications and services to shut down after the timeout period. An application that does not respond to a shutdown request is forced to shut down within 30 seconds. A service that does not respond to a shutdown request is forced to shut down after 20 seconds.
RmShutdownOnlyRegistered 0x10	Shut down applications if and only if all the applications have been registered for restart using the RegisterApplicationRestart function. If any processes or services

https://docs.microsoft.com/en-us/windows/win32/api/restartmanager/nf-restartmanager-rmshutdown

図 65　RmShutdownの処理

Windows Restart Mangerの悪用における具体例を一つご紹介しましょう。
以下の図は「MicrosoftEdgeUpdate.log」というログファイルが暗号化されようとする際にContiランサムウェアがRmGetList関数を呼び出した直後のメモリの様子ですが、暗号化しようとした「MicrosoftEdgeUpdate.log」というログファイルが「MicrosoftEdgeUpdate.exe」というアプリケーションによって開かれていることをContiランサムウェアが把握した瞬間を示しています。この直後、「MicrosoftEdgeUpdate.exe」というアプリケーションは強制終了されます。

Windows Restart Managerを悪用しているContiランサムウェアの実際の様子

▼ContiランサムウェアがRmGetList関数を呼び出した直後のメモリの状態（指定したファイルを使用中のアプリケーション情報が取得できている）

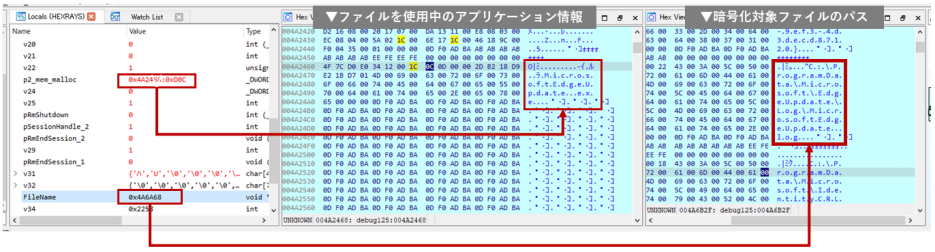


図 66　Windows Restart Managerが悪用された際の様子

なお補足となりますが、Windows Restart Managerを利用されることで偶発的にみられる分析者側の弊害として、状況によりマルウェア解析ツールなどがたまたま掴んでいるファイルが暗号化された際にマルウェア解析ツールが強制終了させられるという現象が稀に見られることがあります。ただしこれまでに解説したとおり、これは解析ツールを明確に妨害しようとして攻撃者が意図した挙動ではありません。例えば、以下の図は解析ツールであるIDA Proが「MSIGMSIZ.DAT」というシステムファイルをたまたま開いている際に、Contiランサムウェアが該当ファイルを含むフォルダの暗号化処理に入った際、RmGetList関数により該当ファイルを開いているIDA Proを認識しRmShutdown関数で強制終了させる際の様子です。当然、該当ファイルを開いていないタイミングであれば、IDA Proが強制終了されることはありません。これはその他のデバッグツールや調査ツールなども同様です。

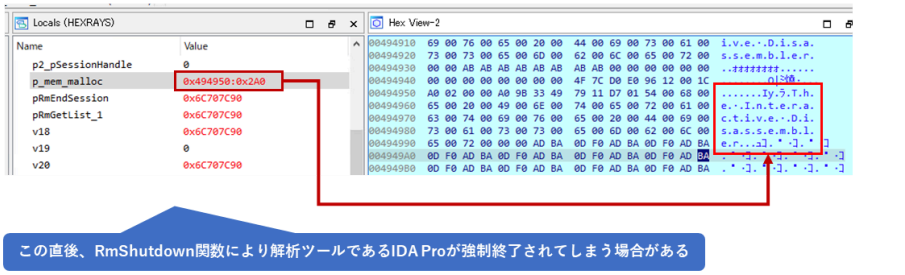
（補足）Windows Restart Managerを利用されることにより偶発的に見られる弊害現象

状況によりマルウェア解析ツールがたまたま掴んでいるファイルが暗号化された際、マルウェア解析ツールが強制終了させられるという現象の発生が稀に見られるが、これは解析ツールを妨害しようとして攻撃者が意図した挙動ではない。

▼(例)解析ツールであるIDA Pro(ida.exe)が「MSIGMSIZ.DAT」というファイルを偶然開いている状況下の様子

以下の通り、ContiがRmGetList関数により「MSIGMSIZ.DAT」を開いているアプリケーションを取得した際、IDA Proを意味するThe Interactive Disassemblerというアプリケーション名が取得されていることがわかる。

▼ContiランサムウェアがRmGetList関数を呼び出した直後のメモリの状態（指定したファイルを使用中のアプリケーション名が取得できている）



※上記は一例だが、他のデバッガや解析ツールでも同様の現象が起きる場合がある。

図 67　偶発的に見られる弊害現象

システムの復旧妨害

説明が少し前後してしまいましたが、Contiランサムウェアは暗号化の前にシステムの復元（ボリュームシャドウコピー）を削除する挙動があります。これによりユーザーはシステムを過去の状態に復元できなくなります。ボリュームシャドウコピーの削除処理は、WMI(Windows Management Instrumentation)を用いて行いますが、前述の通りこの際もWMIを使用する際に必要な全ての文字列は異なる暗号化が施されており、それぞれ個別の復号関数で復号されます（下図）。

システム復旧妨害

Contiランサムウェアは暗号化の前にシステムの復元（ボリュームシャドウコピー）を削除する。
これにより、ユーザーはシステムを過去の状態に復元できなくなる。

ボリュームシャドウコピーの削除処理は、WMI(Windows Management Instrumentation)を用いて行う。
なお、この際も文字列は全て異なる暗号化が施されており、それぞれ個別の復号関数で復号される。

▼WMIによるボリュームシャドウコピーの削除処理

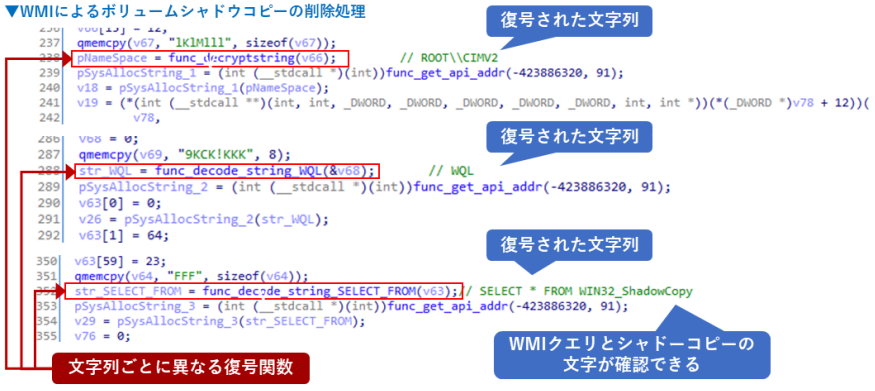


図 68　システムの復旧妨害

補足となりますが、上記のWMIによるボリュームシャドウコピーの削除操作は、イベントログの設定でWMIのイベントトレースを有効にしていた場合以下の図のようにイベントログに記録が残ります。

（補足）イベントログで把握する方法

イベントログの設定でWMIのイベントトレースを有効にしていた場合、以下のようにイベントログに記録が残る。

▼WMIによるボリュームシャドウコピーの削除が記録されたイベントログ

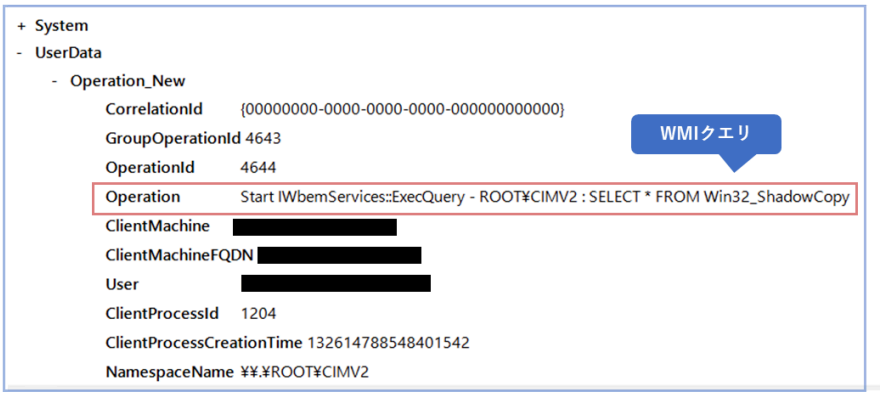


図 69　イベントログで補足する方法

脅迫文の作成

Contiランサムウェアは、暗号化と並行して全てのフォルダ配下に「readme.txt」というファイル名で脅迫文を作成していきます（下図）。

【フォルダごとに実行される処理】

全てのフォルダ配下に脅迫文の作成

脅迫文が記述された「readme.txt」というテキストファイルが全てのフォルダに作成されていく。

▼ Contiランサムウェアが Cドライブ直下に脅迫文を作成しようとしている様子（例）

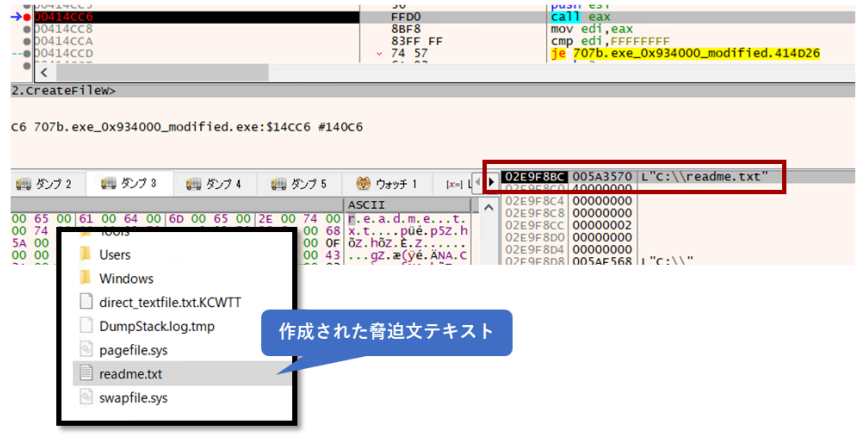


図 70　全てのフォルダ配下に脅迫文の作成

以下はContiランサムウェアが作成する脅迫文の本文ですが、脅迫文の中で二重の脅迫を示す窃取データの公開に関して言及していることがわかります。

Contiランサムウェアが作成する脅迫文

脅迫文の中で、二重の脅迫という特徴である窃取データの公開について言及していることがわかる。

▼ Contiランサムウェアが作成する脅迫文「readme.txt」の中身

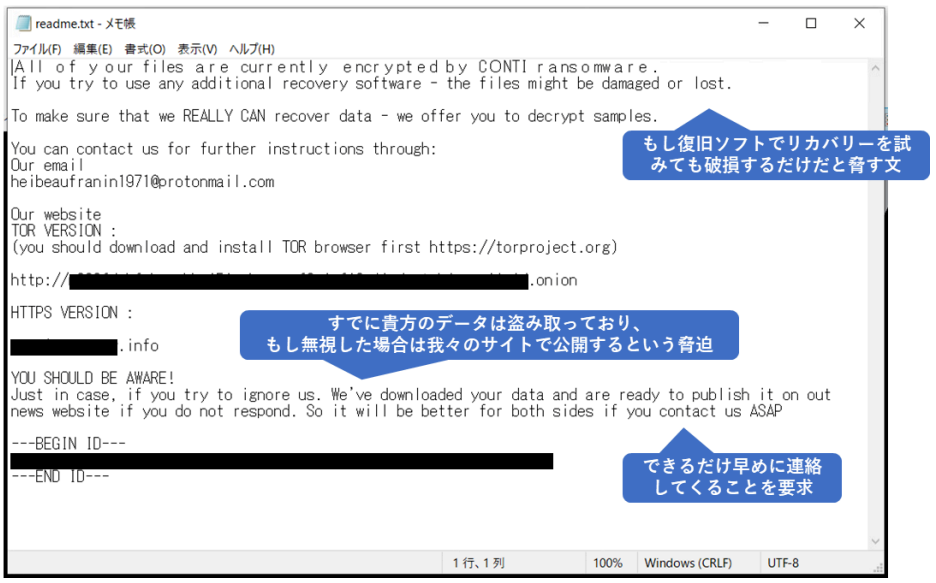


図 71　Contiランサムウェアが作成する脅迫文

ネットワーク挙動

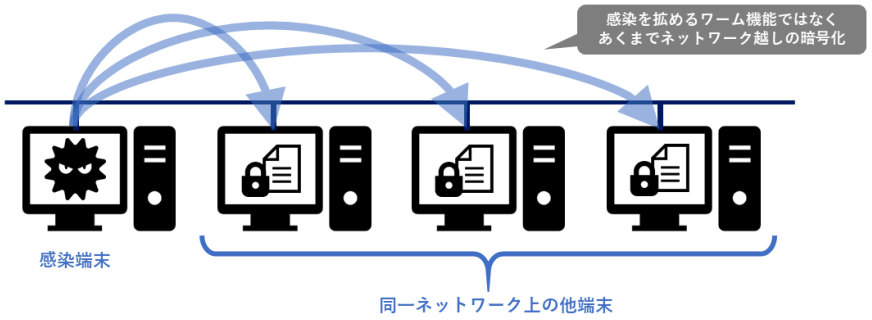
Contiランサムウェアは同一ネットワーク上の共有フォルダやネットワーク上の他の端末のファイルを暗号化する能力を持っている点が特徴的です。なおこのネットワーク挙動は、ランサムウェアそのものを拡散させるワーム機能ではなく、あくまでネットワーク越しの暗号化です（ただし結果的には端末内のファイルが暗号化されてしまうという点で同等の影響があると言えます）。

Contiランサムウェアは管理共有にSMBでアクセスすることで他の端末の中のファイルを暗号化していきますが、パスワードアタックなどの独自の認証突破機能は持っていないため、ネットワーク越しの暗号化は以下の図でまとめたような条件によって結果が異なります。あくまでContiランサムウェア本体が感染したサーバまたは端末から認証入力なしでアクセス可能な対象（端末や共有フォルダ）があればそれらが暗号化されていきます。ただし、例えば共有フォルダなどに認証が設定されていたとしても一度ユーザーが認証入力したなどで感染端末内に認証情報が記憶されている状況下であれば認証入力が不要となるため暗号化されてしまいます（下図）。

また、近年の攻撃ではドメインコントローラー(Active Directory)を狙って侵略するケースが少なくありませんが、デフォルト状態では管理共有が有効になっているため、ファイアウォールが適切に設定されていない環境では、ドメイン配下の全端末の管理共有に認証無しでアクセス可能となるケースがあり、その場合ドメインコントローラーを感染させることで、このネットワーク越しの暗号化機能によって結果的に配下全ての端末が暗号化されてしまう可能性があります。

ネットワークを介した暗号化の概要

Contiランサムウェアは同一ネットワーク上の共有フォルダや他の端末のファイルを暗号化する能力を持っている。



ネットワーク越しの暗号化は以下の条件によって異なる。

- **認証入力なしでアクセスできる共有フォルダ**
  - **▶ 共有フォルダ内の全ての対象ファイルが暗号化される。**
- **認証入力なしではアクセスできない共有フォルダ**
  - **▶ 暗号化されない。**
- **感染端末から管理共有に認証入力無しでアクセスできる他の端末**
  - **▶ 端末内の全ての対象ファイルが暗号化される。**
- **感染端末から管理共有に認証入力無しでアクセスできない他の端末**
  - **▶ 暗号化されない。**

認証突破機能は
持っていない

ただし、認証入力が必要な場合
でも認証状態が記憶されている
端末の場合は暗号化される。
（ネットワークドライブなど）

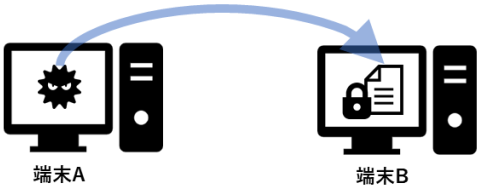
感染端末がAD
サーバの場合など

図 72　ネットワークを介した暗号化の概要

他端末を暗号化している際のランサムウェアのプロセス内部ではローカルファイルと同じようにアクセスし暗号化していきます。以下の図はContiランサムウェアがネットワーク上の別の端末のファイルを暗号化している際のプロセスのハンドル一覧の様子ですが、IPアドレスを含むファイルパスに対しファイルアクセスしている様子がローカルと同じように確認できます。

他端末の暗号化

Contiランサムウェアはネットワーク上の他の端末もローカルファイルと同じようにアクセスし暗号化する。



▼ Contiランサムウェアのプロセス(端末A上)がネットワーク上の別端末(端末B)のファイルを暗号化している様子

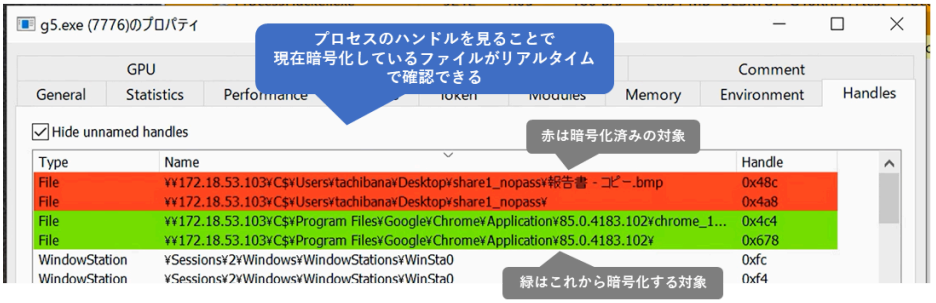


図 73　他端末の暗号化

上で触れたネットワーク上の他端末を探索する挙動についても深堀りして詳細を解説しましょう。
Contiランサムウェアは以下の図にあげた順で暗号化対象端末を選定し暗号化していきます（下図）。
まずGetIpTable関数によりARPテーブルの参照し、アドレスがプライベートIPアドレスであるものを探します。
なお、この際に比較する数値の文字列もこれまでに解説したとおり、一つずつ異なる計算で暗号化されています。条件に合致するIPアドレスが見つかった場合、該当IPの４オクテット目を0〜254までインクリメントしなが

らARPパケットを送信します。
ARP解決できるIPアドレスが存在した場合、445番ポートへSMBプロトコルで疎通確認を行い、アクセスできる端末やフォルダが見つかった場合、FindFirst/FindNextFileなどの一般的なファイル操作関数を用いてローカルファイルと同じように探索し暗号化していきます（下図）。

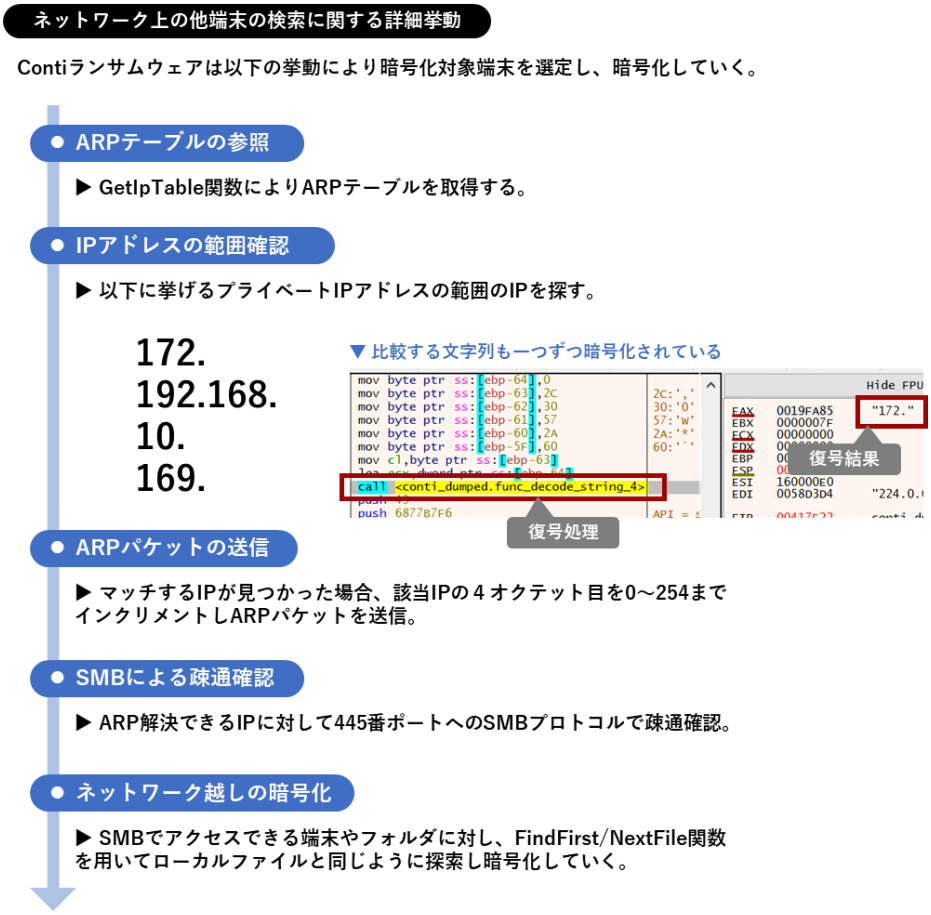


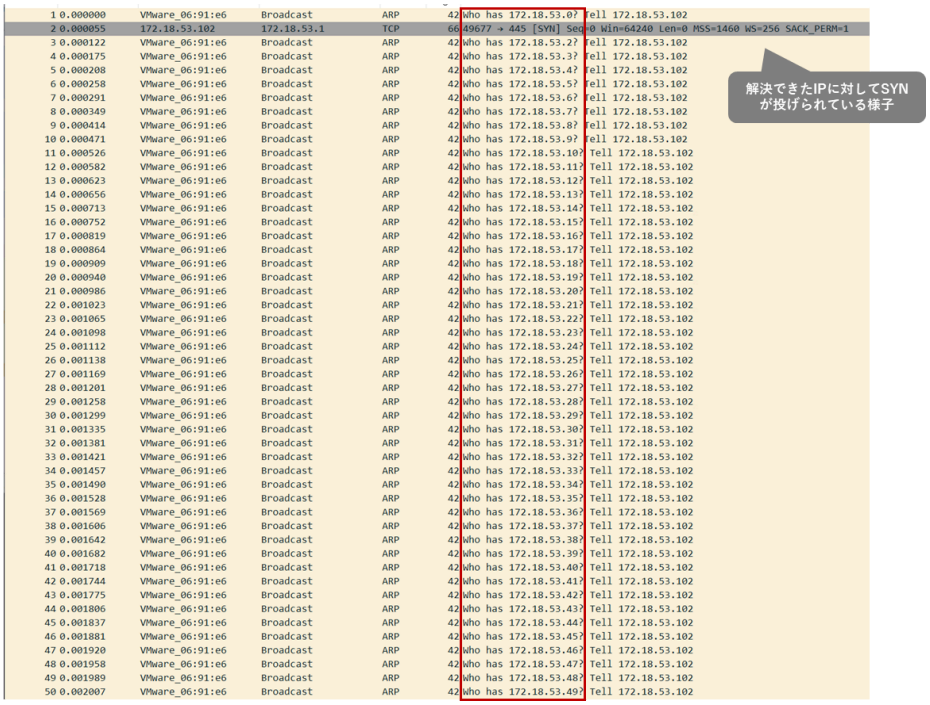
図 74　ネットワーク上の他端末の検索に関する詳細挙動

次の図は、Contiランサムウェアが同一ネットワーク上の他の端末を探索している様子ですが、ARPパケットが１IPアドレスごとにインクリメントされながら送信されていることがわかります（下図）。

ネットワーク上の他端末の検索

ContiランサムウェアはARP**パケットを用いて同一ネットワーク上の他の端末を検索する。**

▼ ContiランサムウェアがARPパケットで１IPアドレスごとに端末の存在有無（応答）を確かめる様子（パケットキャプチャ画面）



1	0.000000	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.0? Tell 172.18.53.102
2	0.000055	172.18.53.102	172.18.53.1	TCP	66	49677 → 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.000122	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.2? Tell 172.18.53.102
4	0.000175	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.3? Tell 172.18.53.102
5	0.000208	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.4? Tell 172.18.53.102
6	0.000258	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.5? Tell 172.18.53.102
7	0.000291	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.6? Tell 172.18.53.102
8	0.000349	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.7? Tell 172.18.53.102
9	0.000414	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.8? Tell 172.18.53.102
10	0.000471	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.9? Tell 172.18.53.102
11	0.000526	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.10? Tell 172.18.53.102
12	0.000582	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.11? Tell 172.18.53.102
13	0.000623	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.12? Tell 172.18.53.102
14	0.000656	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.13? Tell 172.18.53.102
15	0.000713	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.14? Tell 172.18.53.102
16	0.000752	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.15? Tell 172.18.53.102
17	0.000819	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.16? Tell 172.18.53.102
18	0.000864	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.17? Tell 172.18.53.102
19	0.000909	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.18? Tell 172.18.53.102
20	0.000940	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.19? Tell 172.18.53.102
21	0.000986	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.20? Tell 172.18.53.102
22	0.001023	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.21? Tell 172.18.53.102
23	0.001065	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.22? Tell 172.18.53.102
24	0.001098	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.23? Tell 172.18.53.102
25	0.001112	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.24? Tell 172.18.53.102
26	0.001138	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.25? Tell 172.18.53.102
27	0.001169	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.26? Tell 172.18.53.102
28	0.001201	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.27? Tell 172.18.53.102
29	0.001258	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.28? Tell 172.18.53.102
30	0.001299	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.29? Tell 172.18.53.102
31	0.001335	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.30? Tell 172.18.53.102
32	0.001381	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.31? Tell 172.18.53.102
33	0.001421	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.32? Tell 172.18.53.102
34	0.001457	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.33? Tell 172.18.53.102
35	0.001490	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.34? Tell 172.18.53.102
36	0.001528	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.35? Tell 172.18.53.102
37	0.001569	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.36? Tell 172.18.53.102
38	0.001606	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.37? Tell 172.18.53.102
39	0.001642	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.38? Tell 172.18.53.102
40	0.001682	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.39? Tell 172.18.53.102
41	0.001718	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.40? Tell 172.18.53.102
42	0.001744	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.41? Tell 172.18.53.102
43	0.001775	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.42? Tell 172.18.53.102
44	0.001806	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.43? Tell 172.18.53.102
45	0.001837	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.44? Tell 172.18.53.102
46	0.001881	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.45? Tell 172.18.53.102
47	0.001920	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.46? Tell 172.18.53.102
48	0.001958	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.47? Tell 172.18.53.102
49	0.001989	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.48? Tell 172.18.53.102
50	0.002007	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.49? Tell 172.18.53.102

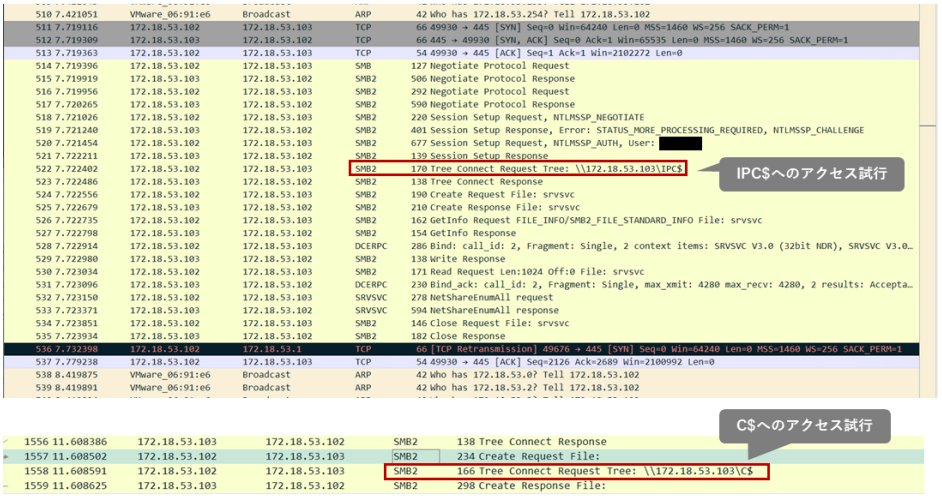
図 75　ネットワーク上の他端末の検索

また以下の図は、見つかった端末にSMBリクエストを送信しアクセスしている様子ですが、C$などの管理共有へアクセスしていることがわかります。

SMBを介したアクセス試行

Contiランサムウェアは見つかった端末にSMBリクエストを送信し、アクセスを試みる。

▼ ContiランサムウェアがSMBで他端末にアクセスを試みる様子（パケットキャプチャ画面）



510	7.421051	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.254? Tell 172.18.53.102
511	7.719116	172.18.53.102	172.18.53.103	TCP	66	49930 → 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
512	7.719300	172.18.53.103	172.18.53.102	TCP	66	445 → 49930 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
513	7.719363	172.18.53.102	172.18.53.103	TCP	54	49930 → 445 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
514	7.719396	172.18.53.102	172.18.53.103	SMB	127	Negotiate Protocol Request
515	7.719919	172.18.53.103	172.18.53.102	SMB2	506	Negotiate Protocol Response
516	7.719956	172.18.53.102	172.18.53.103	SMB2	292	Negotiate Protocol Request
517	7.720265	172.18.53.103	172.18.53.102	SMB2	590	Negotiate Protocol Response
518	7.721026	172.18.53.102	172.18.53.103	SMB2	220	Session Setup Request, NTLMSSP_NEGOTIATE
519	7.721240	172.18.53.103	172.18.53.102	SMB2	401	Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLENGE
520	7.721454	172.18.53.102	172.18.53.103	SMB2	677	Session Setup Request, NTLMSSP_AUTH, User: [redacted]
521	7.722211	172.18.53.103	172.18.53.102	SMB2	139	Session Setup Response
522	7.722402	172.18.53.102	172.18.53.103	SMB2	170	Tree Connect Request Tree: \\172.18.53.103\IPC$
523	7.722486	172.18.53.103	172.18.53.102	SMB2	138	Tree Connect Response
524	7.722556	172.18.53.102	172.18.53.103	SMB2	190	Create Request File: srvsvc
525	7.722679	172.18.53.103	172.18.53.102	SMB2	210	Create Response File: srvsvc
526	7.722735	172.18.53.102	172.18.53.103	SMB2	162	GetInfo Request FILE_INFO/SMB2_FILE_STANDARD_INFO File: srvsvc
527	7.722798	172.18.53.103	172.18.53.102	SMB2	154	GetInfo Response
528	7.722914	172.18.53.102	172.18.53.103	DCERPC	286	Bind: call_id: 2, Fragment: Single, 2 context items: SRVSVC V3.0 (32bit NDR), SRVSVC V3.0…
529	7.722980	172.18.53.103	172.18.53.102	SMB2	138	Write Response
530	7.723034	172.18.53.102	172.18.53.103	SMB2	171	Read Request Len:1024 Off:0 File: srvsvc
531	7.723096	172.18.53.103	172.18.53.102	DCERPC	230	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 4280 max_recv: 4280, 2 results: Accepta…
532	7.723150	172.18.53.102	172.18.53.103	SRVSVC	278	NetShareEnumAll request
533	7.723371	172.18.53.103	172.18.53.102	SRVSVC	594	NetShareEnumAll response
534	7.723851	172.18.53.102	172.18.53.103	SMB2	146	Close Request File: srvsvc
535	7.723934	172.18.53.103	172.18.53.102	SMB2	182	Close Response
536	7.732398	172.18.53.102	172.18.53.1	TCP	66	[TCP Retransmission] 49676 → 445 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
537	7.779238	172.18.53.102	172.18.53.103	TCP	54	49930 → 445 [ACK] Seq=2126 Ack=2689 Win=2100992 Len=0
538	8.419875	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.0? Tell 172.18.53.102
539	8.419891	VMware_06:91:e6	Broadcast	ARP	42	Who has 172.18.53.2? Tell 172.18.53.102

1556	11.608386	172.18.53.103	172.18.53.102	SMB2	138	Tree Connect Response
1557	11.608502	172.18.53.102	172.18.53.103	SMB2	234	Create Request File:
1558	11.608591	172.18.53.102	172.18.53.103	SMB2	166	Tree Connect Request Tree: \\172.18.53.103\C$
1559	11.608625	172.18.53.103	172.18.53.102	SMB2	298	Create Response File:

図 76　SMBを介したアクセス試行

その後、以下の図のようにネットワーク上の他の端末へのファイルアクセスもパケットキャプチャから確認することができます。

ネットワーク越しの暗号化の様子

以下のように感染端末(172.18.53.102)から他の端末へアクセスし暗号化している様子がわかる。

▼ Contiランサムウェアが他の端末のデスクトップのファイルを暗号化している際の様子（パケットキャプチャ画面）

No.	Time	Source	Destination	Protocol	Length	Info
101014	1234.513392	172.18.53.102	172.18.53.103	SMB2	298	Create Request File: Users\tachibana\Desktop\報告書の注意事項.pdf
101015	1234.513473	172.18.53.103	172.18.53.102	SMB2	298	Create Response File: Users\tachibana\Desktop\報告書の注意事項.pdf
101016	1234.513583	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101017	1234.513610	172.18.53.103	172.18.53.102	SMB2	182	Close Response
101018	1234.513722	172.18.53.102	172.18.53.103	SMB2	430	Create Request File: Users\tachibana\Desktop\報告書の注意事項.pdf
101019	1234.514601	172.18.53.103	172.18.53.102	SMB2	410	Create Response File: Users\tachibana\Desktop\報告書の注意事項.pdf
101020	1234.518377	172.18.53.102	172.18.53.103	SMB2	171	Read Request Len:1703 Off:0
101021	1234.518421	172.18.53.103	172.18.53.102	SMB2	1317	Read Response
101022	1234.518506	172.18.53.102	172.18.53.103	SMB2	162	SetInfo Request FILE_INFO/SMB2_FILE_ALLOCATION_INFO
101023	1234.518547	172.18.53.103	172.18.53.102	SMB2	124	SetInfo Response
101024	1234.518623	172.18.53.102	172.18.53.103	SMB2	162	GetInfo Request FILE_INFO/SMB2_FILE_NETWORK_OPEN_INFO
101025	1234.518646	172.18.53.103	172.18.53.102	SMB2	186	GetInfo Response
101027	1234.518726	172.18.53.102	172.18.53.103	SMB2	423	Write Request Len:1713 Off:0 File: Users\tachibana\Desktop\報告書の注意事項.pdf
101029	1234.518792	172.18.53.103	172.18.53.102	SMB2	138	Write Response
101030	1234.519016	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101031	1234.522087	172.18.53.103	172.18.53.102	SMB2	182	Close Response
101032	1234.522278	172.18.53.102	172.18.53.103	SMB2	430	Create Request File: Users\tachibana\Desktop\報告書の注意事項.pdf
101033	1234.522375	172.18.53.103	172.18.53.102	SMB2	378	Create Response File: Users\tachibana\Desktop\報告書の注意事項.pdf
101034	1234.522562	172.18.53.102	172.18.53.103	SMB2	258	SetInfo Request FILE_INFO/SMB2_FILE_RENAME_INFO NewName:Users\tachibana\Desktop\報告書の注意事項.pdf.KCWTT
101035	1234.522737	172.18.53.103	172.18.53.102	SMB2	124	SetInfo Response
101036	1234.522810	172.18.53.102	172.18.53.103	SMB2	162	GetInfo Request FILE_INFO/SMB2_FILE_NETWORK_OPEN_INFO
101037	1234.522833	172.18.53.103	172.18.53.102	SMB2	186	GetInfo Response
101038	1234.522917	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101039	1234.522981	172.18.53.103	172.18.53.102	SMB2	182	Close Response
101040	1234.523632	172.18.53.102	172.18.53.103	SMB2	306	Create Request File: Users\tachibana\Desktop\払い戻し制度について.pdf
101041	1234.523703	172.18.53.103	172.18.53.102	SMB2	298	Create Response File: Users\tachibana\Desktop\払い戻し制度について.pdf
101042	1234.523809	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101043	1234.523838	172.18.53.103	172.18.53.102	SMB2	182	Close Response
101044	1234.523947	172.18.53.102	172.18.53.103	SMB2	438	Create Request File: Users\tachibana\Desktop\払い戻し制度について.pdf
101045	1234.524960	172.18.53.103	172.18.53.102	SMB2	410	Create Response File: Users\tachibana\Desktop\払い戻し制度について.pdf
101046	1234.528335	172.18.53.102	172.18.53.103	SMB2	171	Read Request Len:1703 Off:0
101047	1234.528376	172.18.53.103	172.18.53.102	SMB2	1317	Read Response
101048	1234.528471	172.18.53.102	172.18.53.103	SMB2	162	SetInfo Request FILE_INFO/SMB2_FILE_ALLOCATION_INFO
101049	1234.528508	172.18.53.103	172.18.53.102	SMB2	124	SetInfo Response
101050	1234.528578	172.18.53.102	172.18.53.103	SMB2	162	GetInfo Request FILE_INFO/SMB2_FILE_NETWORK_OPEN_INFO
101051	1234.528600	172.18.53.103	172.18.53.102	SMB2	186	GetInfo Response
101053	1234.528674	172.18.53.102	172.18.53.103	SMB2	423	Write Request Len:1713 Off:0 File: Users\tachibana\Desktop\払い戻し制度について.pdf
101055	1234.528740	172.18.53.103	172.18.53.102	SMB2	138	Write Response
101056	1234.528889	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101057	1234.531960	172.18.53.103	172.18.53.102	SMB2	182	Close Response
101058	1234.532181	172.18.53.102	172.18.53.103	SMB2	438	Create Request File: Users\tachibana\Desktop\払い戻し制度について.pdf
101059	1234.532276	172.18.53.103	172.18.53.102	SMB2	378	Create Response File: Users\tachibana\Desktop\払い戻し制度について.pdf
101060	1234.532466	172.18.53.102	172.18.53.103	SMB2	262	SetInfo Request FILE_INFO/SMB2_FILE_RENAME_INFO NewName:Users\tachibana\Desktop\払い戻し制度について.pdf.KCWTT
101061	1234.532642	172.18.53.103	172.18.53.102	SMB2	124	SetInfo Response
101062	1234.532722	172.18.53.102	172.18.53.103	SMB2	162	GetInfo Request FILE_INFO/SMB2_FILE_NETWORK_OPEN_INFO
101063	1234.532749	172.18.53.103	172.18.53.102	SMB2	186	GetInfo Response
101064	1234.532833	172.18.53.102	172.18.53.103	SMB2	146	Close Request
101065	1234.532898	172.18.53.103	172.18.53.102	SMB2	182	Close Response

図 77　ネットワーク越しの暗号化の様子

なお、管理共有に関する詳細とその対策については、過去の記事(※)をご覧ください。
（※標的型攻撃ランサムウェア「Ryuk」の内部構造を紐解く：/research/20191211/ryuk/）

ネットワークによる暗号化（動画）

より直感的に感じていただくために、Contiランサムウェアに感染した端末がネットワーク上の他の端末を暗号化する際の様子を収めた動画を用意して公開していますので併せてご覧ください。

Contiランサムウェアの挙動は以上であり、全ての暗号化が完了すると終了します。

一般に誤解されている可能性がある点として、データを盗み取るのはランサムウェアではなく、ランサムウェアを拡散させる前に攻撃者が手動で行います。そのため、以上で解説してきた通り、ランサムウェアの役割はファイルの暗号化や脅迫文の提示、システム復旧妨害などが主となります。

まとめ

ContiランサムウェアはRyukの後継とされていますが、過去に弊社ブログで公開したRyukの解析結果と比較すると、ネットワークを跨いだ暗号化に関する挙動が類似しているようにも見える一方、それ以外の挙動は大きく異なるように見え、もし同じ攻撃者が開発しているのであれば明らかに手口が複雑かつ高度になっていると言えます。

加えて、上記で解説した通り、Contiランサムウェアはファイルごとに異なる暗号化鍵で暗号化していますが、最近よく見られるランサムウェアのファイル暗号化手法には以下の２種類があります。

1. 1. 全てのファイルを環境ごとに共通した鍵で暗号化しその鍵（共通鍵）を公開鍵で暗号化するタイプ
   2. Contiのようにファイルごとに異なる鍵で暗号化した上でそれぞれの鍵を公開鍵で個別に暗号化するタイプ

前者の場合、公開鍵で暗号化した共通鍵を攻撃者に送付させれば、攻撃者は秘密鍵をばらすことなく、復号した共通鍵（を含む復号ツール等）のみを被害者へ送付することで復号が実現でき、さらに全ての被害組織で共通の公開鍵を使用するため検体が使いまわせるメリットがありますが、その一方で、メモリから共通鍵を抽出された場合に全てのファイルを一括で復号されてしまうリスクも存在します。
後者の場合、ファイルごとに異なる鍵で暗号化しているため、前述のリスクは軽減されますが、ファイルを復号する際の攻撃者の選択肢は「暗号化鍵を含んだ全ての暗号化済みファイル（または全ての暗号化鍵）を送らせる」か「秘密鍵（を含む復号ツール等）を渡す」かのほぼ２択になります。全てのファイルを送らせることは現実的ではないため、実質的に「秘密鍵を渡す」という選択しかありませんが、もし秘密鍵を渡してしまった場合その秘密鍵が共有されることで他の被害組織にも使い回されてしまう可能性があります。
つまり、ファイルごとに異なる鍵を使用する手法を取る場合、被害企業ごとに秘密鍵と公開鍵のペアを生成する必要があり、それは言い換えれば攻撃者が標的ごとに鍵のペアを用意・管理していることを意味し、明確にターゲットを絞って攻撃を行う前提でランサムウェアを開発したということを示唆しています。

なお、弊社が持つ脅威インテリジェンスを用いてContiランサムウェアの被害を受けた企業を複数調査した結果、RDPやVPNの不備、サーバソフトウェアの脆弱性などが過去に存在していた事実が複数の被害企業において確認されました。実際にそうした経路から侵入されたケースや、TrickBot/QakBotなどのマルウェアによってRDP等の認証情報を取得された可能性がある事例なども一般には報告されており、これまでに多方面で言われてきたランサムウェア対策と考慮すべき観点に変わりはありません。

Contiランサムウェアに手動操作が行えることを示唆する実行引数が用意されていることからも分かる通り、ランサムウェアは攻撃グループにとってはただの一道具という側面があるため攻撃の全貌を知る上ではごく一部分の内容に過ぎませんが、また一方でメインとなるその"道具"の詳細挙動を理解することは攻撃を知る上で必要不可欠であるとも言えるでしょう。
今回本記事で詳細に解説してきた内容は、Contiランサムウェアのみならず他のランサムウェアを理解する上でも共通してご参考いただける情報が含まれていますので、こうした情報が少しでも何かのお役立てになれば幸いです。

- ハッシュ値：707b752f6bd89d4f97d08602d0546a56d27acfe00e6d5df2a2cb67c5e2eeee30
- Contiランサムウェアv3自動解析スクリプト：
  https://github.com/AgigoNoTana/resolve_conti_API/blob/main/resolve_conti_API.py
- 感染動画URL：
  https://youtu.be/PANnAIwct68
