Computer Security Symposium 2022
24 - 27 October 2022

マルウェア作成者のコーディング癖の収集例：メモリのゼロクリアの観点から

村上 弘和 [1,2,3,*]　西垣 正勝 [1]

概要：サイバー攻撃者のインテリジェンス収集のためのディープマルウェア解析のために、マルウェア作成者のコーディング癖の収集法の提案を行っている.マルウェア作成者のコーディング癖のうち、多くのソフトウェアで一般的に行われる処理からの収集は大きな役割を占める.本論文では、一般的に行われる処理からのコーディング癖の抽出の事例の一つとして、メモリの領域を特定の値で埋める「memset」を利用したメモリのゼロクリアの実装状況を複数のマルウェアから抽出し、比較検討した. その結果、比較的単純かつ一般的に用いられることが多い処理でも、実装に違いが出ていることを確認することができた.この結果から、一般的に行われる処理のうち、同一目的の処理を複数のマルウェアで比較することが可能であり、多くの処理に対してこの方法を適用することで、類似性を判定し得るのではないかと考えられる.

キーワード：ディープマルウェア解析，脅威インテリジェンス

An example of collecting malware coding fingerprints: Viewpoint of clearing memory to zero

Hirokazu Murakami [1,2,3,*]　Masakatsu Nishigaki [1]

Abstract: For deep malware analysis for cyber attacker intelligence collection, we are proposing a method for collecting coding fingerprints of malware authors. Among the coding fingerprints of malware authors, collection from common processes in many software plays a large role. In this paper, as an example of extracting coding fingerprints from commonly performed processing, we extract and compared the implementation status of memory zero clearing using "memset", which fills a memory area with a specific value, from several malware. As a result, it was confirmed that even relatively simple and commonly used processes show differences in implementation. From these results, it is considered that it is possible to compare the same purpose in common processes for multiple malware, and that similarity can be determined by applying this method to a large number of processes.

Keywords: Deep malware analysis, Threat intelligence

1. 序　論

サイバー攻撃者を追跡するためには、サイバー攻撃で使用されたマルウェアからのインテリジェンス情報の抽出は有用な情報源の一つになると考えられる[1]. マルウェアから作成者のインテリジェンスを抽出するとき、どのような点を抽出することに価値があるだろうか. 価値のある情報の一つに、マルウェア作成者の同一性、または非同一性を示すコーディング癖の収集があると考える. マルウェア解析で見つけることができる情報と出現が予想されるマルウェア作成者・攻撃者像の関係およびレイヤーのイメージを図 1 に示す. コーディング癖になり得る情報は様々ある[2]. その中で、特に出現頻度が高い「一般的に行われる処理」について、具体的にどのように差が表れているのか、どの程度の差が生じるのか、また差が生じる原因が何であるかについて、マルウェアからの抽出による観察、サンプルコードを用いた実験および考察を行う.

マルウェアのコーディング癖が生じる原因は、目的達成できるプログラム作成のためのコーディングにおいて、コード作成者に委ねられる部分があるためである. このような場合、実行結果が同じであれば、あとは作成者の好み次第というケースは多くあり、そこに作成者個々人の特徴が残る可能性があると考える. その特徴は、ソフトウェアそのものの目的を達成するための制御部という大きな観点から、各関数という小さな観点まで、コーディングが関わる全ての個所で特徴が残る可能性がある.

注意点として、特に関数といった一つの機能かつ処理単位が小さい処理の場合､ライブラリを利用することがある. この場合、コードの中から見つかる特徴はライブラリによるものになることに留意しておく必要がある. ただし、見つかったコードがライブラリによるものと判明した場合、「そのライブラリを使う傾向にある」という情報にはなる

1 静岡大学　創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University
2 東京電機大学　サイバーセキュリティ研究所
Cyber Security Lab, Tokyo Denki University
3 株式会社 CyCraft Japan
CyCraft Japan Corporation.
* murakami.hirokazu.22@shizuoka.ac.jp

© 2022 Information Processing Society of Japan

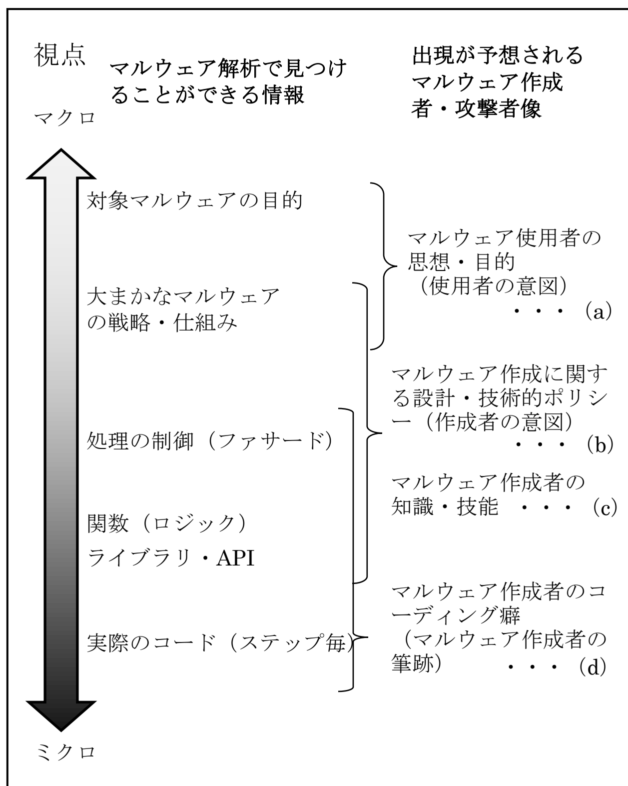

図 1　マルウェア使用者及び作成者の分析のレイヤーのイメージ

ため、情報量は 0 ではない．

一方、ファサードと呼ばれる制御部分は、目的に応じてコーディングする必要があるので、マルウェア作成の都度コーディングするということも多いとみられ、作成者の特徴が残りやすいのではないかと推測する．このような同じ目的の処理におけるコードの差分を、様々な処理で観察することにより、作成者により選択するコードのパターンに違いが生じることが予想される．このコードの選択パターンの観察から、マルウェア作成者の同一性、非同一性の指標情報の一つにできるのではないかと考える．

本論文では、「memset により特定の領域をゼロクリアする」という処理について、実際に複数のマルウェアから対応する処理を抽出し、どの程度違いがあるかについて示す．

2. 一般的に行われる処理からコーディング癖を抽出するための解析

現在のプログラミングでは、多くの異なる目的のアプリケーションであっても、普遍的に行われる処理が存在する．本論文では、そのような処理を一般的に行われる処理と定義する．具体例として、C/C++言語による Windows OS および Linux OS で動作する実行形式ファイルで一般的に行われる処理は、以下のような処理が挙げられる．

- メモリのクリア
- メモリのデータコピー
- 領域の確保・開放方法
- ファイルアクセス方法
- データ操作（文字列操作、編集方法など）

これらの処理は、目的が異なる処理であっても、ある程度以上の規模のアプリケーションでは利用されやすく、出現率が高いため、比較対象としても利用しやすい．これら以外の次点として、通信を伴う処理や暗号化・復号化を伴う処理であれば、それらの処理の特徴も比較対象にできると考えられる．このような抽出対象にすべき処理の発見、選定は、本論文の方法の有用性をより高めるための今後の課題である．

一般的に行われる処理では、最終的にその処理の目的を達成できれば、処理の流れがどのようになっていても問題はない．そのため、「全く同じ目的であっても、実装方法は複数あり、その違いが観察できる」ということが予想される．実装に違いがでる点として、以下が考えられる．

- 引数の順番、数、型、渡し方
- リターン値の意味、処理結果の渡し方
- 引数のエラーチェックの有無や方法
- 実装で使う命令
- 条件判定方法
- ループ処理の実装方法
- 特徴的、あるいは特殊な領域の使い方（ローカル変数とグローバル変数）

これらについて違いがあれば、ソースコードにも違いがあることが疑われる．このような点に着目し、それぞれの点について 2 つのアプリケーションを比較して、類似点が多ければ作成者が同一である可能性が高いという指標にできるのではないかと考える．

3. memset 処理の分析例

実際に使われたマルウェアから、一般的に行われる処理のうち実際に同じ目的の処理を抽出し、コーディング癖となるような特徴の違いが見られるかを検証する．本論文では、複数のマルウェアから memset 関数またはそれに相当する処理を抽出し、検証する．

memset 関数は、開始アドレスから、指定したサイズで、指定したデータでメモリの値を設定する関数である．多くの場合、指定したデータに 0x00 を指定することでメモリを 0 クリアする場合に利用される．これは、特に文字列処理でデータが入っていないことを明示的に示し、不用意なバグの発生を防ぐために利用される．また、必要無くなったデータを明示的に消去するためにも利用される．

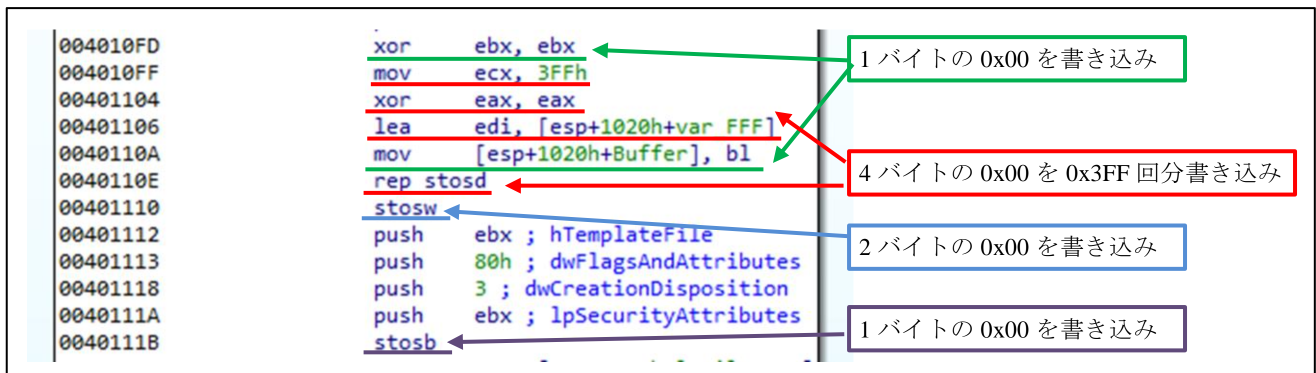

図 2 BlueNoroff から見つかった memset に相当する処理

3.1 マルウェア毎の memset に相当する処理

3.1.1 BlueNoroff の場合

BlueNoroff と呼ばれるマルウェアから見つかった 0 クリアを目的とした memset の処理を図 2 に示す．BlueNoroff は、Lazarus と呼ばれる APT グループが使用したワイパーマルウェアと言われている[3].

処理は、関数としてではなく、一つの処理の中に埋め込まれていた．ファイルを 0 のみで埋め尽くし破壊する処理に使用するために、4,096 バイトのメモリ領域を 0 クリアしている．0 クリアするためには、mov、stosb、stosw、stosd といった複数の命令を使用しているという特徴がある．mov、stosb 命令では、それぞれ 1 バイト 0 を書き込んでいる．stosw 命令では、2 バイト 0 を書き込んでいる．残りの 4,092 バイトは、4 バイトの書き込みを行う stosd 命令を rep で 0x3FF=1,023 回繰り返すことで 0 を書き込んでいる．繰り返しの命令や 1 命令で 4 バイトの書き込みができる命令の利用、4 種類の命令を使用している点に大きな特徴がある．このケースでは、処理が一つの処理の中に埋め込まれているため、関数の呼び出し時の引数や返り値は存在しない．また、0 クリアされる領域のアドレスはスタックのアドレス、設定値とサイズはハードコードされており、エラー値になることは無いため、チェックが行われていないことは妥当である．以上から、非常にコンパクトで不具合が無いコードを実現していることが分かる．

3.1.2 OlympicDestroyer の場合

OlympicDestroyer と呼ばれるマルウェアから見つかった 0 クリアを目的とした memset の処理を図 3 に示す．OlympicDestroyer は、2018 年の冬季オリンピックで運営の妨害を目的に使用された[4]．BlueNoroff のアーティファクトや機能を偽装することにより攻撃者を他者に擦り付ける偽旗作戦を行ったと言われている[3].

処理は、memset 関数の呼び出しをが一つの処理の中に埋め込まれていた．ファイルを 0 のみで埋め尽くし破壊する処理に使用するために、4,096 バイトのメモリ領域を 0 クリアしている．単純にライブラリにある memset 関数をコールしていた．このライブラリ関数の実体は同マルウェア内に含まれており、静的リンクで結合されたものとみられる．このケースでは、ANSI 準拠のライブラリの memset がコールされていることから、引数や返り値は ANSI の memset どおりである．また、memset で設定されるアドレスはスタックのアドレス、設定値とサイズはハードコードされており、エラー値になることは無いため、チェックが行われていないことは妥当である．以上から、メモリの 0 クリアは単純にライブラリの memset を使う選択をしたこ

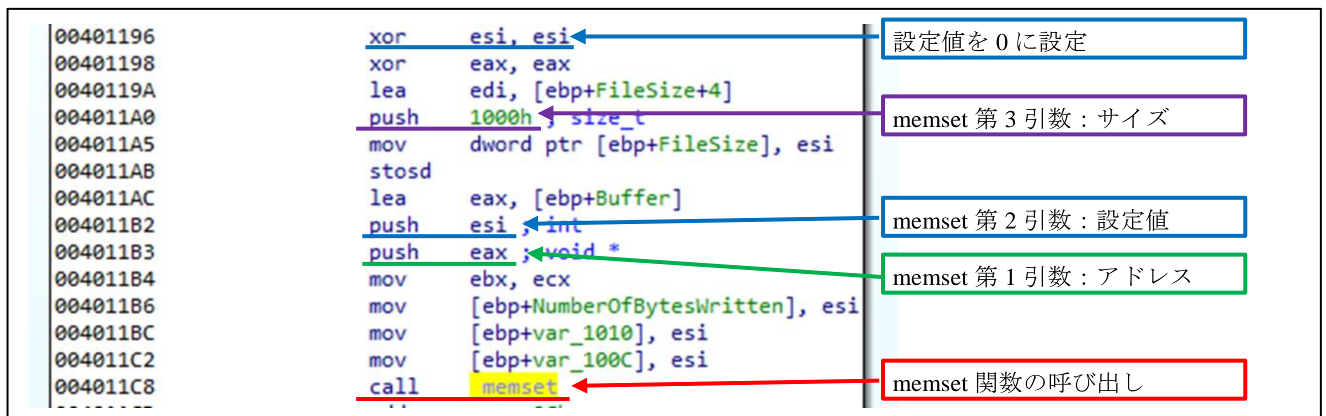

図 3 OlympicDestroyer から見つかった memset に相当する処理

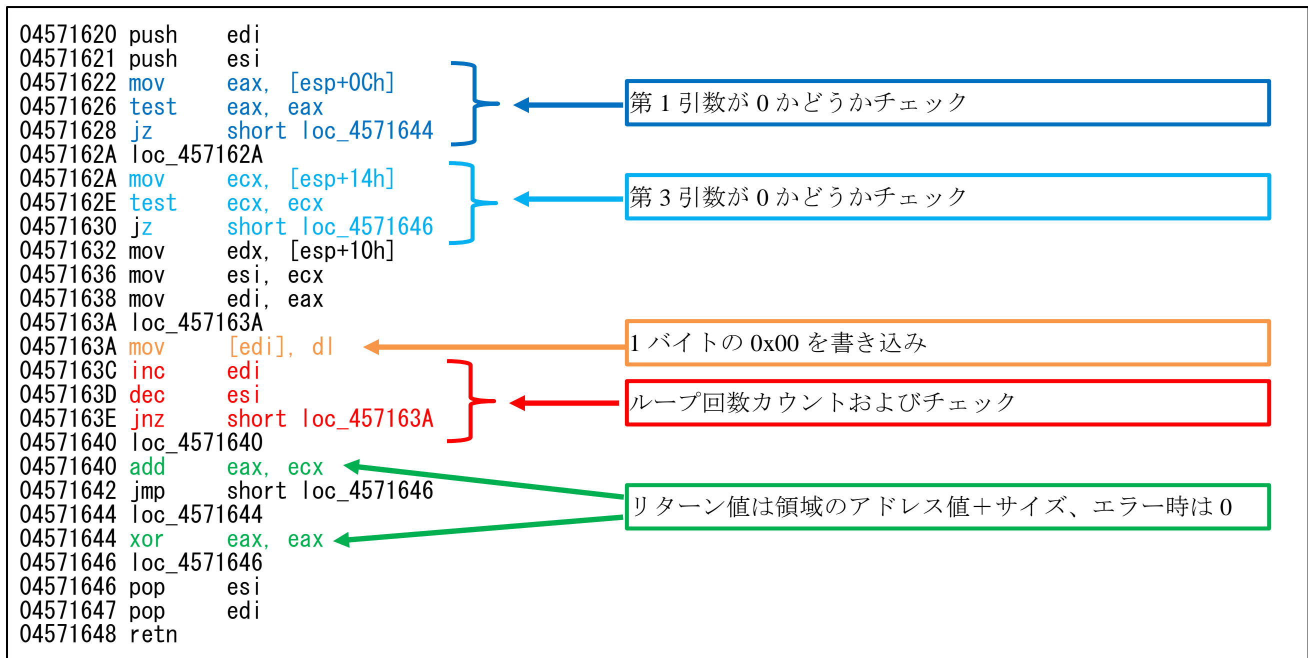

図 4 Emotet 2020 年春から見つかった memset に相当する処理

とが分かる．

3.1.3 Emotet 2020 年春の場合

Emotet と呼ばれるマルウェアのうち、2020 年春ごろの検体から見つかった 0 クリアを目的とした memset の処理を図 4 に示す．Emotet は、情報窃盗や追加のマルウェア感染を目的としているといわれるマルウェアと言われている [5]．

処理は、関数として実装されていた．第 1 引数にデータを設定する対象のアドレス、第 2 引数に設定する値、第 3 引数にサイズが設定されるように作られていた．これは、一般的な memset 関数と同じ引数の並びおよび内容である．このうち、第 1 引数のアドレスは NULL チェック、第 3 引数のサイズは 0 かどうかのチェックがされていた．このエラーチェックがあることも一つの特徴である．

実際の 0 クリアにあたる処理は、mov 命令で 1 バイトずつ第 2 引数の値を設定し、処理の度にアドレスを 1 加算、カウンタを 1 減算してカウンタが 0 になるまで繰り返している．この方法は、処理としては単純で分かりやすいものの、32 ビット向けのプログラムのコードとしてはやや非効率である．32 ビットのプログラムの場合、4 バイトのバスを使うことを前提としている．このとき、8 ビットの値の設定も 32 ビットの値の設定も概ね同じ処理コストになる．言い換えれば、設定値を 4 バイトに拡張して 32 ビット単位で処理をすれば、ループの回数は 4 分の 1 でよくなる．その分、処理時間の短縮が見込める．これは、memset する領域のサイズが大きくなるほど影響が大きくなる．一方、サイズが 4 の倍数でない場合も考慮する必要が生じる．サイズが 4 の倍数でない場合、その剰余のバイト数を別途処理する必要があり、そのための条件分岐と処理が必要になるため、コードはより複雑になる．この関数の作成者は、ライブラリを使うのではなくループによるハードコードすることを選択したが、処理の速度よりはコードの単純さを選択した、という特徴があると言える．

また、リターン値にも特徴がある．memset 関数は ANSI で規定されているが、その返り値は memset を行った領域の先頭アドレスである．これは、第 1 引数と一致する．しかし、この関数では第 1 引数と第 3 引数を加算した結果、つまり領域の末尾をリターン値として返すよう設定している．これは一般的な ANSI 準拠の memset の返り値と異なり、この関数を呼び出す側もリターン値の扱いにはその違いに留意する必要があることから、一つの強い特徴となっている．

3.1.4 Emotet 2020 年秋の場合

Emotet と呼ばれるマルウェアのうち、2020 年秋ごろの検体から見つかった 0 クリアを目的とした memset の処理を図 5 に示す．

処理は、memset 関数の呼び出しをが一つの処理の中に埋め込まれていた．データの格納領域として利用するために、スタック上の領域を 0 クリアしている．基本的には、Windows の ntdll.dll 内に含まれる memset API をコールしていた．ただし、解析者の解析を困難にするために、Import table に載らないよう API のアドレスを自力で取得し、自身で作成したアドレスマップに格納、再利用するように作られていた．このため、アドレスマップから memset API のアドレス取得、マップに無かった場合のアドレス取得のための独自処理の実行という、memset を行う前段階の処理に特

```
001F4C30 mov     eax, memset_dword_1FDBA4
001F4C35 sub     esp, 58h
001F4C38 push    esi
001F4C39 mov     esi, edx
001F4C3B push    edi
001F4C3C mov     edi, ecx
001F4C3E test    eax, eax
001F4C40 jnz     short loc_1F4C5D
001F4C42 mov     ecx, 0C41DA0BCh
001F4C47 call    GetProgramAddressByHash_sub_1F3F70
001F4C4C mov     edx, offset unk_773A0280
001F4C51 mov     ecx, eax
001F4C53 call    GetProcAddressByHash_sub_1F3ED0
001F4C58 mov     memset_dword_1FDBA4, eax
001F4C5D loc_1F4C5D
001F4C5D push    44h ; 'D'
001F4C5F lea     ecx, [esp+1Ch]
001F4C63 push    0
001F4C65 push    ecx
001F4C66 call    eax ; memset_dword_1FDBA4
001F4C68 mov     eax, CreateProcessW_dword_1FE16C
```

ntdll.dll に登録されている memset API のアドレスの読み込み試行。読み込めない場合、memset API アドレス取得処理へ。

32 ビットハッシュ値を用いて DLL、API のアドレスを取得する独自性の高い関数。
これにより、memset API のアドレスを得る。

memset 第 3 引数：サイズ

memset 第 2 引数：設定値

memset 第 1 引数：アドレス

ntdll.dll に登録されている memset API を使用

図 5　Emotet 2020 年秋から見つかった memset に相当する処理

徴が出ている．また、memset で設定されるアドレスはスタックのアドレス、設定値とサイズはハードコードされており、エラー値になることは無いため、チェックが行われていないことは妥当である．

OlympicDestroyer と似ているが、最も異なるのは memset の処理をアプリケーションそのものがスタティックリンクで内包しているか、OS の API を使っているか、という違いである．つまり、memset 関数の中身が、アプリケーション内に埋め込まれた静的ライブラリの処理か、OS の動的ライブラリによる処理かという違いが生じている．同じ memset 関数の利用でも、いずれを使うことを選択するかということもまた、作成者の特徴になるのではないかと考える．

3.2　ソースコードとバイナリコードの関連性の検証

バイナリコードを分析するにあたり、ソースコードはどのように対応するかについて、実際にソースコードを作成し、コンパイルして結果を検証した．ソースコードは、2 章で抽出したメモリを 0 クリアする目的の memset に相当するものとし、同じ処理を C/C++言語を用いて 3 種類の書き方で表現した．コンパイラは、Windows のアプリケーション作成においてある程度以上の利用頻度が見込まれる Microsofr Visual Studio の C/C++を使用した．ソースコードおよびコンパイル結果を図 6 に示す．

図 6 上部左のソースコードは、ループにより 1 バイトずつデータをサイズ分繰り返し設定するよう記載した．これは、図 4 のようなコードが表れることを期待したソースコードである．しかし、検証で使用した環境では、図 6 上部右のように memset 関数に置き換えられた．これは、コンパイラの最適化等の機能が、コードを解析した結果、memset 関数を使用したほうがよいと判断した結果であると考えられる．逆に考えれば、図 4 のようなコードを出力するには、図 6 上部左のようなコードにするだけでは足らず、コンパイラの環境を変えるか、アセンブラ言語でのハ

```
12    unsigned int *pointer;
20    // for文
21    for(i=0; i<4096; i++)
22    {
23        for_buf1[i] = 0x00;
24    }
25
26    // for文
27    pointer = (unsigned int *)for_buf2;
28    for(i=0; i<4096/4; i++)
29    {
30        pointer[i] = 0x00000000;
31    }
32
33    // memset文
34    memset(memset_buf, 0x00, 4096);
35
```

単純な 1 バイトコピーの for ループは、最適化で memset に置き換わる。

「rep stosd」を使わせるには、アセンブラで書くか、コードにトリックが必要。

ソースで memset を使うと、memset で出力。

```
push    1000h                       ; size_t
lea     eax, [ebp+Buffer]
push    0                           ; int
push    eax                         ; void *
mov     [ebp+NumberOfBytesWritten], 0
call    _memset
xor     eax, eax
mov     ecx, 400h
lea     edi, [ebp+DstBuf]
push    1000h                       ; size_t
rep stosd
push    eax                         ; int
lea     ecx, [ebp+var_3004]
push    ecx                         ; void *
call    _memset
```

＊Visual Studio 2010 使用。コンパイラオプションはデフォルトのまま。

図 6　ソースコードとコンパイル結果の比較

ードコードなどの工夫が必要で、図 4 のコードの作成者はそのような努力を行ったという特徴が残されていると言える．

図 6 中部左のソースコードは、型のキャストを利用することによって 4 バイトずつデータをサイズ分繰り返し設定するよう記載した．これは、図 4 のようなコードで 4 バイト単位の mov で設定する、または図 2 の rep stosd により 4 バイト単位で設定するようなコードが表れることを期待したソースコードである．この結果、図 6 中部右のように rep stosd を使用したコードが出力された．これは、図 2 のコードに類似している．ただし、図 2 では rep stosd のループの回数が 1 回少なく、1 バイトの mov および stosb、2 バイトの stosw による設定も行っていることから、仮に C/C++のコードで書いていたとしても図 6 中部左のループカウントより 1 回少なく、かつ 2 バイト、1 バイトでの値の設定処理が書かれていたはずである．また、アセンブラによるハードコードの可能性も残される．図 2 のコードがなぜ rep stosd のみで値を設定するようにしなかったかは不明であるが、この命令の使用と 1 バイト、2 バイトでの値の設定もあることが、一つの大きな特徴となっていることが分かる．

図 6 下部左のソースコードは、memset によりサイズ分設定するよう記載した．これは、明示的に memset 関数を使うことでコードでも memset を使用することを期待したソースコードである．この結果、図 6 下部右のように memset 関数を使用した．使用するコンパイラや設定によって変更される可能性はあるものの、ソースコードでライブラリの関数を使用している場合はそのままコードに埋め込まれることが概ね予想できるといえる結果といえる．なお、この検証で生成されたコードでは図 3 と同様にライブラリのコードを自身のコードにリンクで含めていた．Windows OS の memset API を使うようにするには、何らかの変更等が必要とみられ、それを行うかどうかもコード作成者の特徴となり得ると考える．

3.3 memset の分析および検証のまとめ

memset、或いは memset で 0 の設定に特化したメモリの 0 クリアは、多くのアプリケーションの内部処理で利用されている．この処理について、実装方法は複数存在する．今回の複数のマルウェア分析では、処理結果は同じであるが、コードに差が生じていることが確認された．

コードの差に差が生じる原因は、確実にソースコードが違うことを示すことができるものと、環境の影響による可能性が否定できないものの 2 種類があることが考えられる．引数の並び、引数の型や渡し方、引数の値チェック、返り値の内容の違いなどは、ソースコードが違わなければ発生し得ない．これらは、基本的にコンパイラや環境設定で勝手に変更されることが考えられない項目である．今回の分析でも、引数のチェックや返り値に明確に違いが見つかっている．一方、処理内容については、全く同じソースコードであっても、使用するコンパイラの種類や最適化等の設定により生成される実行コードが異なる可能性は十分考えられる．同様に、異なるソースコードでもコンパイル結果が同じであることがあるため、機械語のコードが同じであってもソースコードが同じとは言えない．これは、図 6 のコンパイル結果で memset を使用したコードの出力で再現を確認することができた．

マルウェアコードの分析から抽出したコードとソースコードをコンパイルした検証結果から、ソースコードでの単純な処理の記述やライブラリの使用では生成されないようなコードがマルウェア内から見つかることがあることが確認された．これは、作成者がカスタムで作っているか、特定のメンバーやコミュニティ内でのみ使われているライブラリを使用していることが考えられる．これは、マルウェア作成者に関する特徴量として大きなものになるのではないか考えられる．

4. 結　論

本論文では、一般的に行われる処理でコーディング癖がどの程度表れるかについて、memset 関数に相当する処理を対象に分析、検証した．その結果から、コーディング癖として利用できると考えられるものと問題点を整理する．

引数の並び、引数の型や渡し方、引数の値チェック、返り値の内容の違いといった特徴は、作成したコードが確実に違うであろうことが考えられる．また、一般的に行われる処理の場合、広く使われているライブラリを使う選択もあることが考えられる．ライブラリを使用しない場合は、どのように処理をしているかも特徴となる．また、本論文の分析では触れなかったが、これらの処理の呼び出し前後の処理にも癖となる違いが出ている可能性が予想される．例えば、今回例に挙げた memset 関数の場合は呼び出しの前にアドレスを NULL チェックする、サイズが 0 でないかをチェックするといった判定の有無や、エラー時の処理方法といった点である．

実装されている機械語のコードをコーディング癖とする場合の問題点としては、一般的なライブラリや API を使っている場合、そのコードの中身自体には差が出ないことには留意する必要がある．この場合、特定のライブラリや API を使う傾向にある、という情報のみに留まる．また、機械語のコードが同じであっても、ソースコードが同じであるとは限らないという点も留意する必要がある．さらに、同じソースコードでも、コンパイラそのものの違いや、最適化等のコンパイラの設定の違いにより、出力結果が異なる可能性があることに留意する必要がある．一方、研究により機械語コードからコンパイラや設定が推測できるように

なるならば、利用するコンパイラや設定の傾向という情報になり得る．

コーディング癖による作成者の同一性判定では、なりすましが可能かどうかという点は一つの大きな課題である．他のマルウェアのコードを詳細に分析し、そのコードを流用することによってなりすましをする可能性は考えられ、また理論上は不可能ではない．もしコードの作成者がコーディング癖を含めたなりすましをする場合、なりすまし対象の分析を行い、自身のコードに組み込む必要がある．このためには、それを可能とするだけの技術力とコスト、時間を必要とする．マルウェアの作成者によってもスキルの差はあり、これらを満たせるサイバー攻撃者はごく一部に限られると考えられる．ただし、可能性としては留意すべき点である．

マルウェアの目的が全く異なるものであっても、一般的に行われる処理として示したコードのようによく利用される処理は存在する．そして、それらが実装においては異なるコードになっていることがあることも確認できた．これらを抽出して比較することにより、同じマルウェア作成者が全く異なるマルウェアを作成したとしても、類似性が表れる可能性があると考える．マルウェア作成者が複数のマルウェアを作成していたことを判明させることができれば、マルウェア作成者やマルウェア使用者の関係性を示すインテリジェンスになることが期待できる．本論文の分析と検証は、そのようなインテリジェンスの情報源になり得る可能性を示すことができたと考える．

参考文献

[1] 村上弘和, 西垣正勝, “サイバー攻撃者のインテリジェンス収集のためのディープマルウェア解析”, 2022 年暗号と情報セキュリティシンポジウム(SCIS2022), 2022.

[2] 村上弘和, 西垣正勝, “マルウェア作成者のコーディング癖の収集法の提案”, コンピュータセキュリティシンポジウム2022(CSS2022), 2022.

[3] “The devil’s in the Rich header”, https://securelist.com/the-devils-in-the-rich-header/84348/, SECURELIST, 2018.

[4] 吉川 孝志、菅原 圭, “Olympic Destroyer の内部構造を紐解く”, https://www.mbsd.jp/blog/20180215.html, 三井物産セキュアディレクション, 2018.

[5] Emotet などのマルウェア感染に繋がるメールに引き続き警戒を, https://www.jpcert.or.jp/newsflash/2020122201.html, JPCERT/CC, 2020

付録

付録 A memset コンパイルテスト用サンプルソース

(Windows 用)

```
// memset_test.cpp : コンソール アプリケーションのエントリ ポイントを定義します。
//

#include "stdafx.h"
#include <memory.h>
#include <Windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
	char for_buf1[4096];
	char for_buf2[4096];
	unsigned int *pointer;
	char memset_buf[4096];
	int i;
	HANDLE FileHandle;
	DWORD WrittenSize = 0;

	// memset 関数と for 文等のメモリクリアがどのようにコンパイルされるかのテスト

	// for 文
	for(i=0; i<4096; i++)
	{
		for_buf1[i] = 0x00;
	}

	// for 文
	pointer = (unsigned int *)for_buf2;
	for(i=0; i<4096/4; i++)
	{
		pointer[i] = 0x00000000;
	}

	// memset 文
	memset(memset_buf, 0x00, 4096);

	// ダミーの処理
	// (実際にメモリを使わないと、最適化がコード自体をそもそも入れなくなる模様)
	FileHandle = CreateFile("test.txt",
							GENERIC_WRITE,
							0,
							NULL,
							CREATE_ALWAYS,
							FILE_ATTRIBUTE_NORMAL,
							NULL);
	sprintf_s(for_buf1, sizeof(for_buf1), "This is a test file.¥r¥nOutput for ¥"for loop normal¥"¥r¥n");
	WriteFile(FileHandle, for_buf1, strlen(for_buf1), &WrittenSize, 0);

	sprintf_s(for_buf2, sizeof(for_buf2), "This is a test file.¥r¥nOutput for ¥"for loop using unsigned int pointer¥"¥r¥n");
	WriteFile(FileHandle, for_buf2, strlen(for_buf2), &WrittenSize, 0);

	sprintf_s(memset_buf, sizeof(memset_buf), "This is memset buffer.¥r¥n");
	WriteFile(FileHandle, memset_buf, strlen(memset_buf), &WrittenSize, 0);

	CloseHandle(FileHandle);

	return 0;
}
```