PlugXローダーの進化

Published: 2022-11-30 · Archived: 2026-04-02 11:42:59 UTC

はじめに

ＦＦＲＩセキュリティ ソフトウェアエンジニアの松本です。 PlugX というバックドアは非常に古くから存在しており、現在でも利用されています。 今回はその PlugX のローダーの変遷について見ていきます。

PlugX は標的型攻撃に利用されている Remote Access Tool(以下 RAT と記載)であり、主に政府機関などを狙う標的型攻撃に利用されるツールです。 機能としては情報収集・窃取、侵入後の PC のコントロールなどがあります。

2012 年には既に TrendMicro の記事[1]があるほど PlugX は古くから存在します。 その一方、ここ数か月の間にも TA416 (通称 Mustang Panda[2]) というグループが利用しているという報告[3]が上がっております。

日本国内では株式会社 JTB が PlugX に感染し情報漏洩したという事例が存在します[5][6]。 また、株式会社ジャストシステムのワープロソフトである一太郎の脆弱性を利用し感染を狙う事例もありました[7][9]。

PlugX ローダーは世代に関わらず以下のような流れで動作しています。

1. ドロッパーを添付したメール経由などにより、対象組織内部でドロッパーを実行してもらう
2. ドロッパーが①正規 exe、②マルウェア dll、③バイナリファイルの 3 つを生成する
3. ドロッパーが①正規 exe を実行する

4. ②マルウェア dll は①正規 exe が利用する正規 dll を偽装しており、①正規 exe はそのまま②マルウェア dll を読み込む
5. ②マルウェア dll が③バイナリファイルを読み込む
6. ③バイナリファイルが子プロセスを生成し、次の段階に移る

読み込まれるバイナリファイルの内部に暗号化されたコードなどが入っているため、実際にはその複合など更に細かい動作が存在しますが、動作の流れは以下の概要図のようになります。

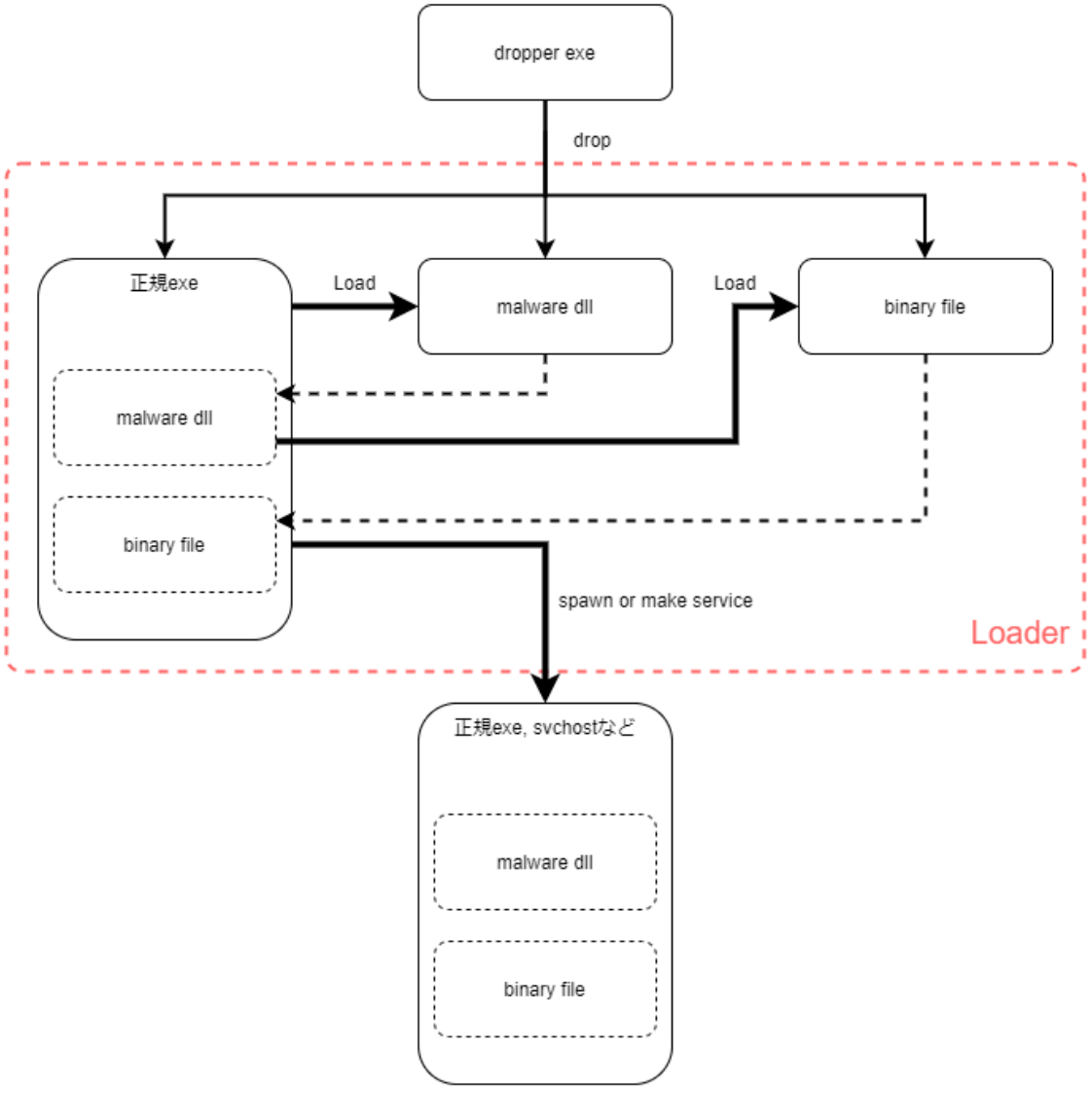


図1 PlugX動作の概要図

実際にドロップされる具体例としては下の図のようになります。 この図では RasTls.exe は正規の exe であり、RasTls.dll へのリンクが存在しています。 マルウェアバイナリにもアイコンが付いていますが、これは本来 msc が Microsoft 管理コンソールに紐付けられている拡張子であるためです。 実際の中身は全く関係無いバイナリファイルとなっています。

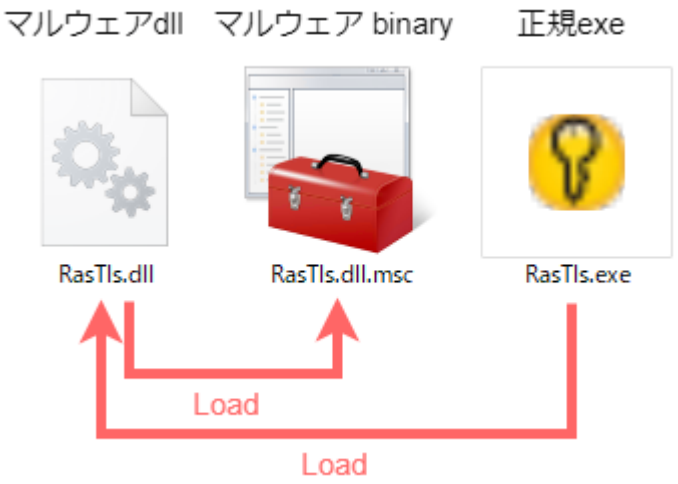
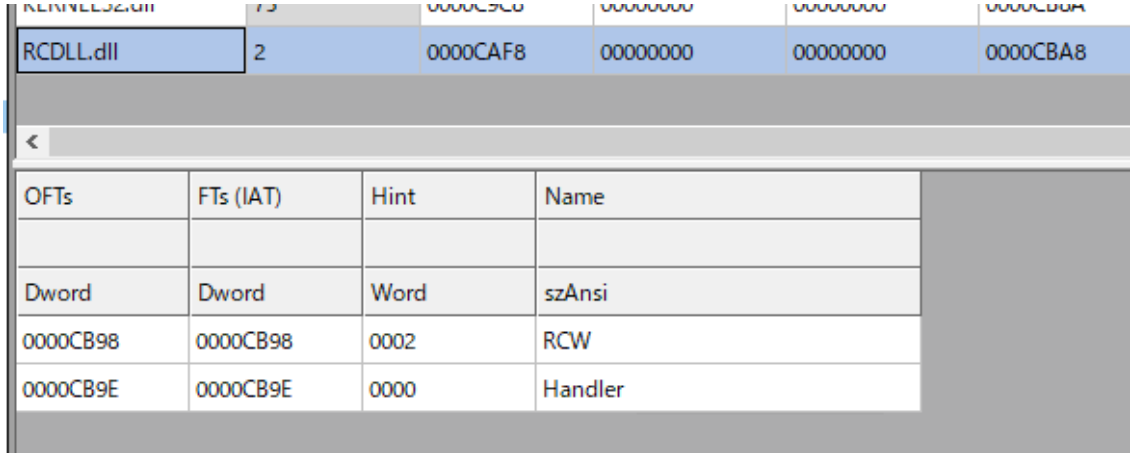


図2 PlugX動作例

exe が不正な dll を読み込んでしまう理由は、exe が dll を読み込む際に同じディレクトリの dll を優先する仕様のためです。



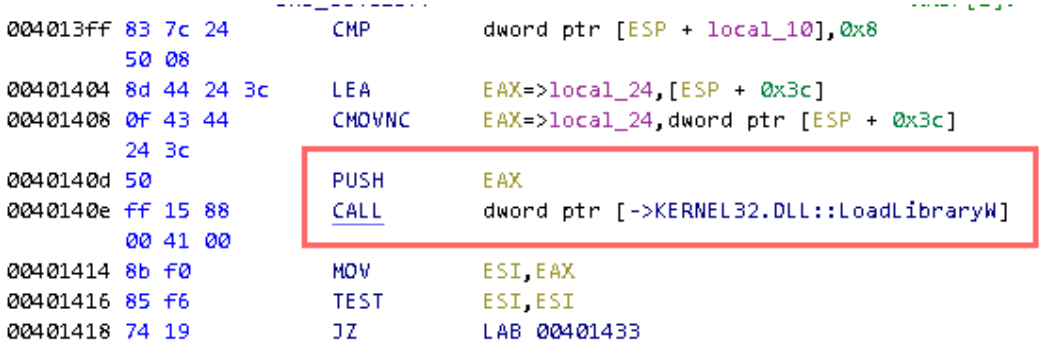
RCDLL.dll	2	0000CAF8	00000000	00000000	0000CBA8

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
0000CB98	0000CB98	0002	RCW
0000CB9E	0000CB9E	0000	Handler

図3 正規 exe のロードにより読み込まれる例

次の図は、LoadLibraryW 関数で同ディレクトリの dll を動的に読み込む箇所です。

このような動的な dll 読み込みの際、通称 exe は自身の現在のパスから読み込みたい dll のパスを生成しますが、 PlugX は同ディレクトリ内にファイルを生成するため、結局すり替えられた dll を読み込むことになります。



```
004013ff 83 7c 24        CMP         dword ptr [ESP + local_10],0x8
         50 08
00401404 8d 44 24 3c     LEA         EAX=>local_24,[ESP + 0x3c]
00401408 0f 43 44        CMOVNC      EAX=>local_24,dword ptr [ESP + 0x3c]
         24 3c
0040140d 50              PUSH        EAX
0040140e ff 15 88        CALL        dword ptr [->KERNEL32.DLL::LoadLibraryW]
         00 41 00
00401414 8b f0           MOV         ESI,EAX
00401416 85 f6           TEST        ESI,ESI
00401418 74 19           JZ          LAB 00401433
```

図4 正規 exe から LoadLibraryW 関数で読み込まれる例

この手法は DLL Side-Loading[4]と呼ばれ、デジタル署名された正規の exe を利用することで、アンチウイルスソフトの検知回避によく利用されます。

一般に dll 内部の悪意のあるコードは entry 関数か、exe から必ず呼ばれる export 関数のどちらかに実装されていますが、PlugX は entry 関数に悪意のあるコードが実装されています。 ここで dll における entry 関数は DllMain 関数であり、リンクなどで読み込まれた段階で必ず呼ばれる処理です。 そのため、exe の初期化処理である dll 読み込みの段階で PlugX ローダーが動作します。

次の段階では、子プロセスもしくはサービス登録によるサービスプロセスを生成します。 実行プロセスに管理者権限が付与されているときには、サービスプロセスを生成し、管理者権限がない時には、子プロセスを生成します。

さて、ここまでは全世代の PlugX 共通な動作の流れについて説明しましたが、次に世代によって変化してきた内容を説明します。

PlugXの変化

PlugX は時間と共に進化していきました。 その流れを簡潔に纏めると以下のようになります。

年代	世代	追加機能の例	引用
2012	第一世代	• RAT	[1]
2013	第二世代	• 自己解凍書庫の利用開始	[8][9]
2015	第三世代	• P2P機能の追加による、感染端末間の通信	[10]
2017	第四世代	• PoisonIvyのコード流用	[11]
現状	第五世代	• Go言語の利用、ローダーの多様化 • C2 サーバーからドロップ検体をダウンロード	[3][12]

第一世代

PlugX では前述通り、まずは正規 exe から dll が読み込まれ、dll からバイナリを読み込んだ上で次のプロセスを生成するという動作をします。 まず注目すべきは、マルウェアバイナリのコード内部でスタック上に文字列を生成する stack strings という手法が利用されている点です。 この手法はバイナリに文字列を直接埋め込まないため、解析をより困難にします。

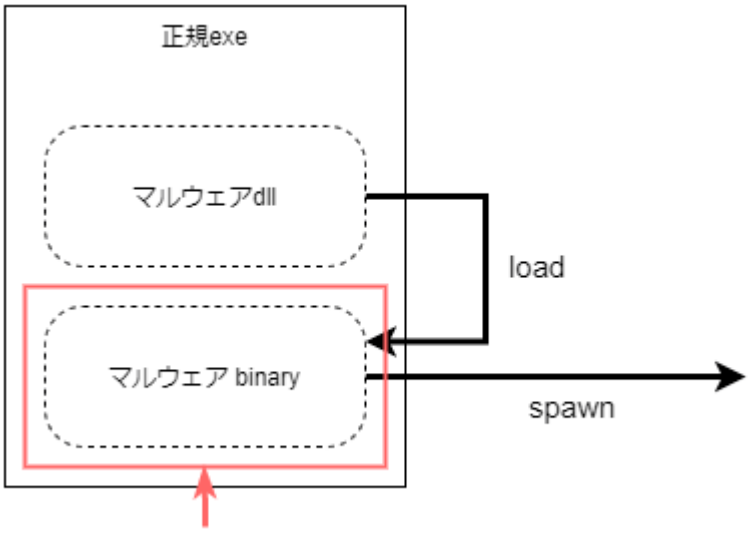


図5 stack strings が利用されている箇所

stack strings は x86 の mov 命令や lea 命令を利用してスタック上に文字列を手動で生成する方法です。

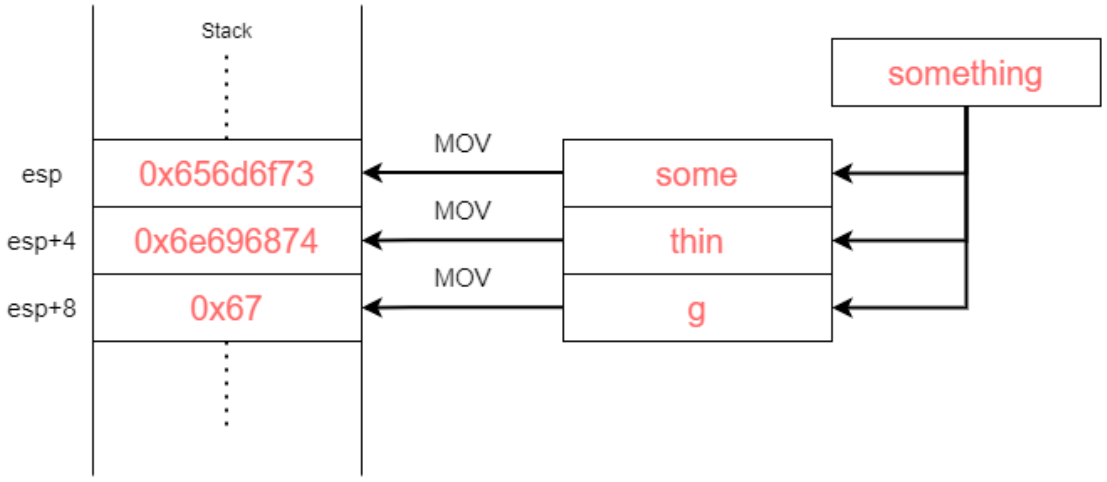


図6 stack strings の例

上の図であれば、4 byte 長の mov 命令を利用することでスタック上に文字列を構築できます。 構築後に esp レジスタが指している箇所を文字列として扱えば Windows API などで認識できる文字列になります。 上記の例では char 型でしたが、wchar_t 型でも同様に作成可能です。

```
02e7a446 c7 85 5c        MOV        dword ptr [EBP + local_a8],"daoL"
         ff ff ff
         4c 6f 61 64
02e7a450 c7 85 60        MOV        dword ptr [EBP + local_a4],"rbiL"
         ff ff ff
         4c 69 62 72
02e7a45a c7 85 64        MOV        dword ptr [EBP + local_a0],"Ayra"
         ff ff ff
         61 72 79 41
02e7a464 c6 85 68        MOV        byte ptr [EBP + local_9c],0x0
         ff ff ff 00
```

図7 LoadLibraryA を 4 byte ずつ生成

```
02e7a3c8 80 38 47        CMP        byte ptr [EAX],'G'
02e7a3cb 75 36           JNZ        LAB_02e7a403
02e7a3cd 80 78 01 65     CMP        byte ptr [EAX + 0x1],'e'
02e7a3d1 75 30           JNZ        LAB_02e7a403
02e7a3d3 80 78 02 74     CMP        byte ptr [EAX + 0x2],'t'
02e7a3d7 75 2a           JNZ        LAB_02e7a403
02e7a3d9 80 78 03 50     CMP        byte ptr [EAX + 0x3],'P'
02e7a3dd 75 24           JNZ        LAB_02e7a403
02e7a3df 80 78 04 72     CMP        byte ptr [EAX + 0x4],'r'
02e7a3e3 75 1e           JNZ        LAB_02e7a403
02e7a3e5 80 78 05 6f     CMP        byte ptr [EAX + 0x5],'o'
02e7a3e9 75 18           JNZ        LAB_02e7a403
02e7a3eb 80 78 06 63     CMP        byte ptr [EAX + 0x6],'c'
02e7a3ef 75 12           JNZ        LAB_02e7a403
02e7a3f1 80 78 07 41     CMP        byte ptr [EAX + 0x7],'A'
02e7a3f5 75 0c           JNZ        LAB_02e7a403
02e7a3f7 80 78 08 64     CMP        byte ptr [EAX + 0x8],'d'
02e7a3fb 75 06           JNZ        LAB_02e7a403
02e7a3fd 80 78 09 64     CMP        byte ptr [EAX + 0x9],'d'
02e7a401 74 13           JZ         LAB_02e7a416
```

図8 スタック上の GetProcAddress を 1 byte ずつ確認

スタック上に生成された文字列はバイナリ内部で利用する API のアドレスを動的に取得するために利用されます。

ここで LZNT1 復号に利用する RtlDecompressBuffer という関数も動的解決しています。 ここで得た LZNT1 復号と XOR 復号により、子プロセスの生成に進みます。

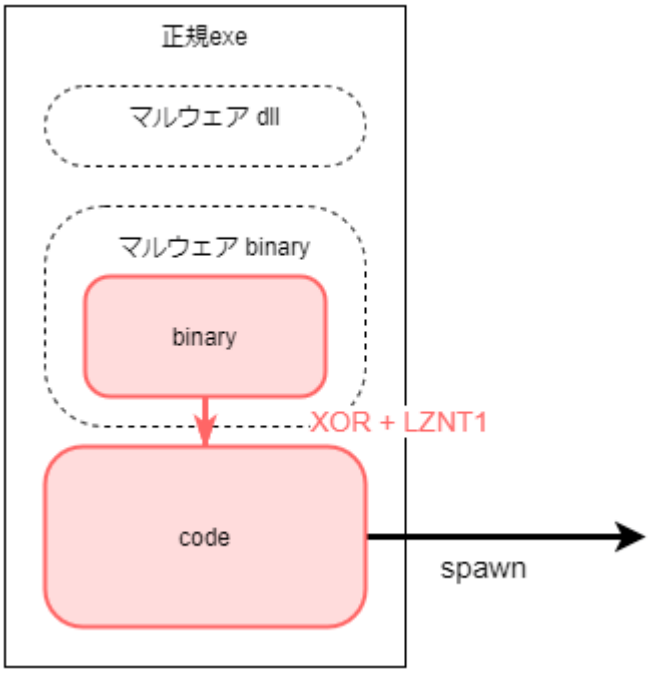


図9 バイナリをメモリ内部で復号

```
188       do {
189         uVar9 = uVar9 + 0xeeeeeeef + (uVar9 >> 3);
190         uVar15 = uVar15 + 0xdddddddd + (uVar15 >> 5);
191         local_c = local_c * -0x7f + 0x33333333;
192         next_payload_address = (short *)((int)next_payload_address * -0x1ff + 0x44444444);
193         *pbVar18 = (char)uVar15 + (char)uVar9 + (char)local_c + (char)next_payload_address ^
194                    pbVar18[local_18];
195         pbVar18 = pbVar18 + 1;
196         local_f8 = local_f8 + -1;
197       } while (local_f8 != 0);
```

図10 XOR 復号関数

```
00431b86 ff 70 0c        PUSH        dword ptr [EAX + 0xc]
00431b89 83 c0 10        ADD         EAX,0x10
00431b8c 50              PUSH        EAX
00431b8d 57              PUSH        buffer
00431b8e 56              PUSH        ESI
00431b8f 6a 02           PUSH        0x2     COMPRESSION_FORMAT_LZNT1                CC
00431b91 ff 55 b8        CALL        dword ptr [EBP + addr_RtlDecompressBuffer]       Rt
00431b94 85 c0           TEST        EAX,EAX
```

図11 RtlDecompressBuffer 呼び出しによる LZNT1 復号

この復号により、メモリ内部に PE ヘッダーを持つコード領域を展開することが確認できます。

```
0:000> db 0x10f0000
010f0000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
010f0010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
010f0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
010f0030  00 00 00 00 00 00 00 00-00 00 00 00 e0 00 00 00  ................
010f0040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
010f0050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
010f0060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
010f0070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

図12 マジックナンバーである MZ が確認できる

第二世代

第二世代は機能面の変化の他、メモリ内部で復号される PE ファイルのマジックナンバーが PE ではなくなっているといった変更点が報告されています[8][9]。 ドロッパーに自己解凍書庫が利用されているパターンもこの辺りから出てきます。 ここではローダー部分の変化について見ていきます。

マルウェア dll には上から順番にディスアセンブルすると、実際に実行される命令とは異なってしまうという難読化が施された関数も追加されています。

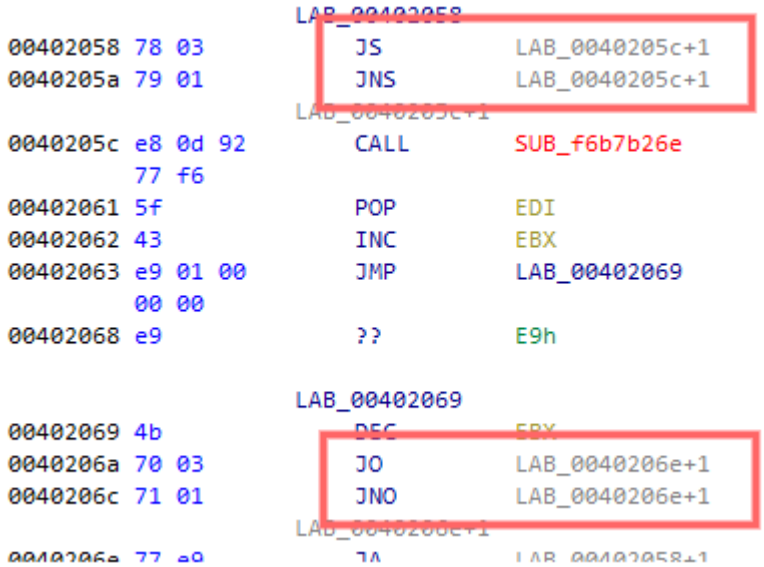


図13 不要な分岐

赤枠は以下のような役割を担っています。

1. 二行続けて同じ所にジャンプするように指定されているため、無条件ジャンプと同等
2. その直後の call 命令は、直前のジャンプを無条件ジャンプと解釈できなかった為に発生
3. e8 から命令を開始することで、意図的にコード外にジャンプする不正な call 命令と解釈させる

他にも、不要な XOR なども追加されており、処理を追いかけにくい構造になっています。

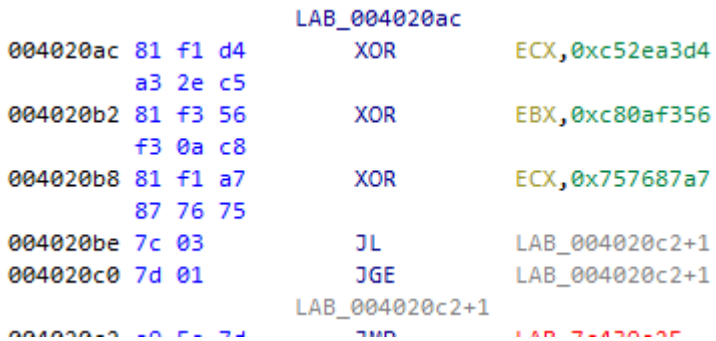


図14 不要な XOR

これらの難読化処理は Ghidra のデコンパイル結果にも影響しますが、除去整理することで以下のようにデコンパイル結果が綺麗になります。 デコンパイル結果から dll 内部のバイナリを復号するコードが出てきたことが確認できます。

```
{
  int iVar1;
  byte *pbVar2;
  int unaff_retaddr;

  pbVar2 = (byte *)(unaff_retaddr + 0x203);
  iVar1 = 0x17002;
  do {
    *pbVar2 = *pbVar2 - 0x2e;
    *pbVar2 = *pbVar2 ^ 0x4d;
    *pbVar2 = *pbVar2 + 3;
    pbVar2 = pbVar2 + 1;
    iVar1 = iVar1 + -1;
  } while (iVar1 != 0);
                    /* WARNING: Bad instruction - Truncating control flow here */
  halt_baddata();
```

図15 デコンパイル結果

この処理以降は第一世代と同じ stack strings を利用した API 動的解決の処理に入ります。
RtlDecompressBuffer も LZNT1 復号に利用されており、XOR 復号のコードは変わっているもののこの辺りのローダーとしての処理は第一世代と同じと見て良いでしょう。

第一世代と同じく PE ヘッダーを持つバイナリが復号されメモリに展開されますが、マジックナンバーが改変されている物となっております。

図16 XV ヘッダー

第三世代

第三世代の機能面における変化は P2P 機能の搭載などが報告されています[10]。

しかし、ローダー部分に関しては XV というマジックナンバーの出現個数の変化などの変更はあるものの、 XOR 復号はもちろん大まかな流れには変化が無いように見受けられました。

第四世代

PoisonIvy のコードを一部流用している[11]ということで第四世代として分類されるこの世代では、 ドロッパー自体は第三世代から引き続き自己解凍書庫であるものの、ローダー部分にも多くの変更が入っています。 例えば、簡単なものであればマルウェア dll の entry 関数から早速 NOP、DEC、INC 命令を使った無駄な処理を大量に配置しています。

```
                    entry
100046cb 55              PUSH        EBP
100046cc 8b ec           MOV         EBP,ESP
100046ce 83 7d 0c 01     CMP         dword ptr [EBP + param_2],0x1
100046d2 0f 85 d1        JNZ         LAB_10004aa9
         03 00 00
100046d8 8b 45 08        MOV         EAX,dword ptr [EBP + param_1]
100046db a3 00 60        MOV         [DAT_10006000],EAX
         00 10
100046e0 90              NOP
100046e1 90              NOP
100046e2 90              NOP
100046e3 90              NOP
100046e4 90              NOP
100046e5 48              DEC         EAX
100046e6 40              INC         EAX
100046e7 90              NOP
100046e8 90              NOP
100046e9 90              NOP
100046ea 90              NOP
100046eb 90              NOP
100046ec 90              NOP
100046ed 90              NOP
```

図17 entry 関数の NOP

ただし、Ghidra のデコンパイル結果には影響しておらず、比較的除去が簡単な処理になっています。

```
2  void __cdecl entry(undefined4 param_1,int param_2)
3
4  {
5    int iVar1;
6
7    if (param_2 == 1) {
8      DAT_10006000 = param_1;
9      GetModuleFileNameA((HMODULE)0x0,&DAT_10006008,0x104);
10     iVar1 = lstrlenA(&DAT_10006008);
11     if (*(int *)((int)&DAT_10006000 + iVar1 + 1) == 0x2e736c54) {
12       FUN_10004066();
13     }
14   }
15   return;
16 }
17
```

図18 entry 関数のデコンパイル結果

また、今までの世代では VirtualAlloc で取得したメモリへの展開は memcpy を利用していましたが、この世代からはバイナリファイルを直接 ReadFile で読み込む処理に変わっています。 ReadFile 経由の読み込みはユーザーランドにおける WinDbg のハードウェアブレークポイントでは止まらないので、 メモリアドレスへの書き込みイベントで止めることが出来ず、より解析が困難になっています。

WinDbg のクエリ結果から見ても、突然メモリに 0x3f の値が出てきていることが確認できます。

	(+) IP	(+) AccessType	(+) Value	(+) OverwrittenValue
[0x0]	0x10003f02	Write	0xe5	0x3f
[0x1]	0x10003f02	Read	0x3f	
[0x2]	0x10003f05	Read	0xe5	
[0x3]	0x10003f05	Write	0xc7	0xe5
[0x4]	0x10003f08	Read	0xc7	
[0x5]	0x10003f08	Write	0x90	0xc7
[0x6]	0x75bb66	Read	0x90	

図19 Time Travel Debugging で確認できるメモリ操作

上記のクエリ結果の中にある Write は復号処理であることが確認できます。

```
                    LAB_10003f02
10003f02 80 07 a6       ADD         byte ptr [EDI],0xa6

10003f05 80 37 22       XOR         byte ptr [EDI],0x22
10003f08 80 2f 37       SUB         byte ptr [EDI],0x37
10003f0b 47             INC         EDI
10003f0c 41             INC         ECX
```

図20 ReadFile で読み込んだメモリ領域を復号する箇所

また、子プロセス生成までに必要な API の取得において stack strings は利用しなくなり、代わりに随時ハッシュ値から関数アドレスを復元する形になっています。

```
::00733b1c 75 15        JNZ         LAB_decoded1_00733b33
::00733b1e 68 2a 19     PUSH        Hash_CreateServiceA
           75 00
::00733b23 e8 c6 f1     CALL        GetAdvapi32Address
           ff ff
::00733b28 50           PUSH        EAX
::00733b29 e8 80 dc     CALL        GetFunctionAddressFromHash
           01 00
::00733b2e a3 5c b6     MOV         [advapi32!CreateServiceA],EAX
           75 00

                                                    Hash_CreateServiceA
                                    5f b1 45 ad     uint        AD45B15Fh
                                    00              ??          00h
```

図21 CreateServiceA の呼び出し例

この関数アドレス取得の手法は子プロセス生成後にも沢山利用されております。 この手法は Poison Ivy と酷似していることが報告されています[11]。

API 解決に必要なモジュールのアドレス解決においては、対象のモジュール名がメモリ上にそのまま存在しています。

```
                    str_kernel32
:00751efe 6b 65 72        ds          "kernel32"
          6e 65 6c
          33 32 00 00
:00751f08 00              ??          00h
```

図22 メモリ上にある kernel32

現在

PlugX と呼ばれる検体は現在も変化し続けていますが、 基本的に正規 exe、マルウェアである dll、そしてバイナリファイルを同じディレクトリに配置するというドロップ方式は変わっていないようです。

一方で解析をより困難にするため C2 サーバーと接続出来なければ次の段階に進まないような改良が見られるようになりました。

図23 検体の C2 サーバーへの接続試行

また、Go 言語製の PlugX ローダーが存在していることも報告されています[11]。 この dll のインポート関数からも確認できます。

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	000783A0	0000	00136862	CEFProcessForkHandlerEx
00000002	00078700	0001	0013687A	_cgo_get_context_function
00000003	00078580	0002	00136894	_cgo_is_runtime_initialized
00000004	00078490	0003	001368B0	_cgo_maybe_run_preinit
00000005	00056330	0004	001368C7	_cgo_panic
00000006	00078410	0005	001368D2	_cgo_preinit_init
00000007	000783E0	0006	001368E4	_cgo_release_context
00000008	000787F0	0007	001368F9	_cgo_sys_thread_start
00000009	000512F0	0008	0013690F	_cgo_topofstack

図24 インポート関数の一部

この Go 言語製の dll も C2 サーバーと接続できなければ動作しないようになっていたため、 現在の主流は解析を困難にするため、 C2 サーバー側にドロップさせたい物を配置する形式になってきたと思われます。

終わりに

PlugX は長い活動期間を通して、機能の高度化以外にもローダー部分における難読化も施していることが確認できました。 こうした変化によりマルウェア解析用のスクリプトが正常に動作しないこともあり、検知を回避し、解析の妨害に一役買っています。

今回調査した PlugX の検体など多くのマルウェアは、以上のような検知回避や難読化などを行っておりますが、FFRI yarai は検知エンジンの新規開発や改善などを継続的に行うことで、検知可能となっておりますのでご安心ください。 (※ただし、全ての PlugX が検出できることを保証するものではありません)。

エンジニア募集

ＦＦＲＩセキュリティではマルウェアの解析などセキュリティの研究開発を行っています。採用に関しては こちら をご覧ください。

参考文献

[1] 標的型攻撃に利用されるPlugXの脅威とは | トレンドマイクロ 脅威データベース

[2] Mustang Panda, TA416, RedDelta, BRONZE PRESIDENT, Group G0129 | MITRE ATT&CK®

[3] The Good, the Bad, and the Web Bug: TA416 Increases Operational Tempo Against European Governments as Conflict in Ukraine Escalates | Proofpoint US

[4] Hijack Execution Flow: DLL Side-Loading, Sub-technique T1574.002 - Enterprise | MITRE ATT&CK®

[5] JTB、約793万人分の個人情報流出の恐れ - 有効パスポート番号4,300件含む | マイナビニュース

[6] 「PlugX」はどんなマルウェア? JTBを狙った標的型攻撃をファイア・アイが解説 | マイナビニュース

[7] 一太郎の脆弱性を突くマルウェア、人事情報装うメールで日本に「着弾」 | ITmedia エンタープライズ

[8] PlugX - The Next Generation | Sophos

[9] From the Labs: New PlugX malware variant takes aim at Japan | Naked Security

[10] マルウエアPlugXの新機能 (2015-01-22) - JPCERT/CC Eyes | JPCERTコーディネーションセンター公式ブログ

[11] Poison Ivyのコードを取り込んだマルウエアPlugX(2017-01-12) - JPCERT/CC Eyes | JPCERTコーディネーションセンター公式ブログ

[12] TA416 Goes to Ground and Returns with a Golang PlugX Malware Loader | Proofpoint US