Reversing Malware Loaders - The Matsnu-A Case

Author: Kyriakos Economou

Date: 21d/07m/2012

Blog: http://anti-reversing.com

Aknowledgements

This article, as always, is dedicated to my friends and family, as well as to my love Irene and to a very special person, Prof. George Prokopakis who has always been an inspiration to me and to others.

Various Vendors Detection Names:

Sophos: **Troj/Matsnu-A**

Symantec: **Trojan.Gen**

Kaspersky: **Trojan-Dropper.Win32.Injector.ddag**

McAfee: **PWS-Zbot.gen.ma**

ESET - NOD32: **Win32/Trustezeb.A**

Microsoft: **Trojan:Win32/Matsnu**

Note: All the numbers shown during the technical parts of this article are directly taken from the debugger through dynamic analysis; hence they are all hexadecimal numbers.

Introduction

The AV industry is growing every day along with the underground industry that produces all those types of malware, from simple file infectors to more sophisticated Trojan types, able to gather and send sensitive information to the bad guys.
The fight between AV companies and malware authors is getting bigger and bigger every single day. Both good and bad guys dedicate a lot of time into researching and implementing either ways to detect or ways to avoid detection, depending on which side these people are.
Most of the malware research is usually concentrated on the infection mechanisms of the malware, as well as the techniques used from the malware to communicate with his creator, completely ignoring sometimes the anti-virus evasion techniques used by the malware in the first place.
This article aims to dig inside the loader used by the **Matsnu** malware family, in order to deploy itself avoiding detection by AV products. Fortunately, at this point this variant is already detected by most AV vendors.
In my job as malware analyst, I hear very often talking about this kind of AV evasion technique as a 'packer'. In a very abstract way, this might be true, but in a technical way it really isn't.
From my experience with **packers** and manual unpacking, I expect that a packer will incorporate some compression algorithm and most probably an encryption algorithm custom or not. Furthermore, the behaviour of a packer is usually a lot different. A packer will usually decompress and decrypt the code of the original executable and then will jump to its original entry point (**OEP**).
On the other hand, I prefer calling these 'packers' used by more and more malware authors as **loaders**. This is because of the technical details. These loaders will usually launch a child process in suspended mode, will overwrite its memory with the decrypted code of the malware and then they will resume its main thread. Some of them, they might choose to allocate some extra memory on the child process, instead of overwriting its memory, and write there the decrypted viral code and then inject a thread to it with starting address the beginning of the allocated memory where the viral code is placed. Some others, they might overwrite themselves through a code stub written into an extra chunk of allocated memory and then jump back to the PE image address space.

In addition, very often the malware authors will choose to first compress the original viral code using a common packer such as UPX, PECompact etc…,and the encrypt it and incorporate it inside the loader.
From a technical point of view, it is quite fair to distinguish these two types of mechanisms, and even if we might keep calling them all as 'packers' for simplicity, it is necessary to understand the differences between them.
The final goal of this article is to manage to isolate a fully working executable of the original malware under the various anti-AV protection layers.

Self-Decryption Stage I

A big part of the code of the loader will be decrypted on run-time through a "slow" decryption algorithm which does a lot of operations in each loop, decrypting the code dword by dword.

The outer loop:

```
00401752    8B4D F0          MOV    ECX, DWORD PTR SS:[EBP-10]
00401755    83C1 01          ADD    ECX, 1
00401758    894D F0          MOV    DWORD PTR SS:[EBP-10], ECX
0040175B    817D F0 688E0    CMP    DWORD PTR SS:[EBP-10], 28E68 ← check counter

00401762    7D 5E            JGE    SHORT  004017C2 ← exit the loop once finished

…more code here

0040178F    E8 D7040000   CALL   00401C6B  ← call to the decryption routine

…more code here

004017C0    EB 90         JMP    SHORT  00401752 ← jump up to loop start
```

Inside the decryption routine:

Some more loops are taking place here, but the important instruction is the one that actually writes every time the result which is a dword stored in ECX register, to the memory location pointed by EAX register:

```
00401ED8    8908       MOV    DWORD PTR DS:[EAX], ECX ← Initial value in EAX is 00408584, it is incremented by a dword in each iteration.
```

Self-Decryption Stage II

When the outer loop mentioned above has finished, there is another one taking place a few instructions later.

```
004017DE  8B4D E0            MOV    ECX, DWORD PTR SS:[EBP-20]
004017E1  83C1 05            ADD    ECX, 5
004017E4  894D E0            MOV    DWORD PTR SS:[EBP-20], ECX
004017E7  817D E0 DF0C0      CMP    DWORD PTR SS:[EBP-20], 0CDF ← check
counter

004017EE  7D 77              JGE    SHORT    00401867 ← exit the loop

…more code here

00401862   E9 77FFFFFF      JMP    004017DE ← jump up to loop start
```

Self-Decryption Stage III

There is one more loop coming next during the self-decryption stage.

```
0040187E   BA 01000000       MOV    EDX, 1
00401883   85D2              TEST   EDX, EDX
00401885   0F84 D2000000     JE     0040195D
```

The three instructions above create a fake execution flow redirection. In fact since the value 1 is always passed to the EDX register, after performing the TEST instruction on the same register, the conditional JE jump that follows will never have any effect on the execution flow.

```
0040188B   817D F0 688E0   CMP    DWORD PTR SS:[EBP-10], 28E68 ← check
counter
00401892   0F85 A1000000   JNZ    00401939 ← if not equal jump to
increase_counter

…some more code here
```

increase_counter:

```
00401939    8B4D F0         MOV    ECX, DWORD PTR SS:[EBP-10]
0040193C    83C1 01         ADD    ECX, 1
0040193F    894D F0         MOV    DWORD PTR SS:[EBP-10], ECX
```

enter_next_decryprion_routine:

```
00401942    68 F7480700     PUSH    748F7
00401947    68 18194F00     PUSH    4F1918
0040194C    8B55 F4         MOV     EDX, DWORD PTR SS:[EBP-C]
0040194F    52              PUSH
00401950    E8 4A000000     CALL    0040199F   ← call decryption routine
00401955    83C4 0C         ADD     ESP, 0C
00401958    E9 21FFFFFF     JMP     0040187E ← jump to loop start
```

Inside the decryption routine:

Some more loops are taking place here, but the important instruction is the one that actually writes every time the result which is a dword stored in ECX register, to the memory location pointed by EAX register:

00401B70 8908 MOV DWORD PTR DS:[EAX], ECX ← **Initial value in EAX is 00408584. It is incremented by a dword in each iteration.**

Self-Decryption Stage IV

Going back to the loop outside the decryption function, since as we saw the condition which would normally signal the end of the looping process, it is fake, we need to examine it more carefully in order to locate the next step.

Indeed, when the conditions are correct the execution will reach a CALL instruction:

```
0040191D    E8 8FF8FFFF     CALL    004011B1
```

..and inside this function it is located the CALL to the beginning of the previously encrypted code:

004013B6 FF15 108B4000 CALL NEAR DWORD PTR DS:[408B10] ← **value stored in this address is 00408584**

Once we enter the function at address 00408584 we see the following:

```
00408584    E8 07000000         CALL        00408590
00408589    75 3A               JNZ     SHORT   004085C5
```

Note the obfuscation trick in the first instruction which confuses the disassembling engine. In fact the CALL instruction will bring the execution in the end of the instruction starting at address 0040858B, which means that all those bytes in between are junk bytes in this case.

```
0040858B   03A0 21D64F5B    ADD    ESP, DWORD PTR DS:[EAX+5B4FD621]
00408591   81EB 05103A00     SUB    EBX, 3A1005
00408597   8DB3 2E103A00     LEA    ESI, DWORD PTR DS:[EBX+3A102E]
0040859D   B9 8B020000       MOV    ECX, 28B
004085A2   66BF 7592         MOV    DI, 9275
004085A6   66313E            XOR    WORD PTR DS:[ESI], DI
004085A9   6683C7 02         ADD    DI, 2
004085AD   83C6 02           ADD    ESI, 2
004085B0   E2 F4             LOOPD  SHORT  004085A6
004085B2   FC                CLD
004085B3   7E 2A             JLE    SHORT  004085DF
004085B5   1B95 CFF6215C     SBB    EDX, DWORD PTR SS:[EBP+5C21F6CF]
004085BB   8745 92           XCHG   DWORD PTR SS:[EBP-6E], EAX
004085BE   D7                XLAT   BYTE PTR DS:[EBX+AL]
004085BF   1F                POP    DS
004085C0   30D5              XOR    CH, DL
004085C2   94                XCHG   EAX, ESP
```

This is what we see once we execute the CALL instruction:

```
00408590   5B                POP    EBX
00408591   81EB 05103A00     SUB    EBX, 3A1005
00408597   8DB3 2E103A00     LEA    ESI, DWORD PTR DS:[EBX+3A102E] ← starts from address 004085B2
0040859D   B9 8B020000       MOV    ECX, 28B ← loop counter
004085A2   66BF 7592         MOV    DI, 9275 ← decryption key
004085A6   66313E            XOR    WORD PTR DS:[ESI], DI ← decrypt by XORing with 9275, one word in each iteration.
004085A9   6683C7 02         ADD    DI, 2
004085AD   83C6 02           ADD    ESI, 2
004085B0   E2 F4             LOOPD  SHORT  004085A6
```

The above decryption algorithm will decrypt an extra portion of code starting from the instruction located immediately after the LOOPD.
So, at this point we saw the various steps used by this loader to decrypt the next parts of the code. Now it's time to continue with the rest of its mechanisms.

Dynamic Imports Resolving & PEB Loader Data Structure

Normally, malware authors retrieve the VAs of the APIs by using two Windows APIs which are the **LoadLibrary** and the **GetProcAddress** APIs, in order to avoid detection through the imports normally listed inside the imports table.
However, in this case the author of the loader has decided to go through the **PEB (Process Environment Block) Loader Data Structure - PEB_LDR_DATA**

structure in order to retrieve the necessary information, which is a more stealth way to retrieve the VAs of the necessary APIs.
The pointer to this structure is located at PEB + 0x0C.

Back to where we stopped, immediately after the end of the decryption loop we locate a CALL at address 004085CD and by entering this function we see another CALL at address 004086EF, and inside that function is where the loader of the malware will access the **PEB_LDR_DATA structure**.

```
0040870E    64FF35 3000000      PUSH    DWORD PTR FS:[30]
00408715    58                  POP     EAX
```

In the two instructions above, we notice another obfuscation attempt. In fact instead of pushing the address of PEB onto the stack and then popping that value back to EAX, could just do MOV EAX, DWORD PTR FS:[30].

00408716 8B40 0C MOV EAX, DWORD PTR DS:[EAX+C] ← **move to EAX the pointer to the PEB_LDR_DATA**

00408719 8B48 0C MOV ECX, DWORD PTR DS:[EAX+C] ← **mov to ECX the pointer to the first LDR_MODULE structure of the first module loaded by the windows loader**

0040871C 8B11 MOV EDX, DWORD PTR DS:[ECX] ← **save to EDX the pointer to the LDR_MODULE structure of the next module loaded by the windows loader**

0040871E 8B41 30 MOV EAX, DWORD PTR DS:[ECX+30] ← **mov to EAX the pointer to the name of the first module name loaded by the windows loader.**

Then follows another CALL at address 00408728, to a function dedicated to calculate a magic dword from the name of the currently examined module. If the dword matches the predefined constant, then the loader knows it found the necessary loaded module to continue its mechanisms.

Calculation Algorithm:

00408797 8A10 MOV DL, BYTE PTR DS:[EAX] ← **go through all chars one by one**
00408799 80CA 60 OR DL, 60 ← **start dword calculation**
0040879C 01D3 ADD EBX, EDX
0040879E D1E3 SHL EBX, 1 ← **end dword calculation**
004087A0 0345 10 ADD EAX, DWORD PTR SS:[EBP+10] ← **increase pointer to string name by 2, because it's stored as Unicode**
004087A3 8A08 MOV CL, BYTE PTR DS:[EAX] ← **mov next char value to CL**

004087A5 84C9 TEST CL, CL ← **check if it's zero, which means we reached the end of the string**
004087A7 E0 EE LOOPDNE SHORT 00408797 ← **if it's not jump up to loop for the next char**
004087A9 31C0 XOR EAX, EAX ← **zero out EAX**
004087AB 8B4D 0C MOV ECX, DWORD PTR SS:[EBP+C] ← **move to ECX magic dword**
004087AE 39CB CMP EBX, ECX ← **check if calculated dword = magic dword**
004087B0 74 01 JE SHORT 004087B3 ← **if it is, module located**
004087B2 40 INC EAX
004087B3 5A POP EDX
004087B4 5B POP EBX
004087B5 59 POP ECX
004087B6 89EC MOV ESP, EBP
004087B8 5D POP EBP
004087B9 C2 0C00 RET 0C

The figure that follows demonstrates the condition in which the two values match, when checking the kernel32.dll loaded module.

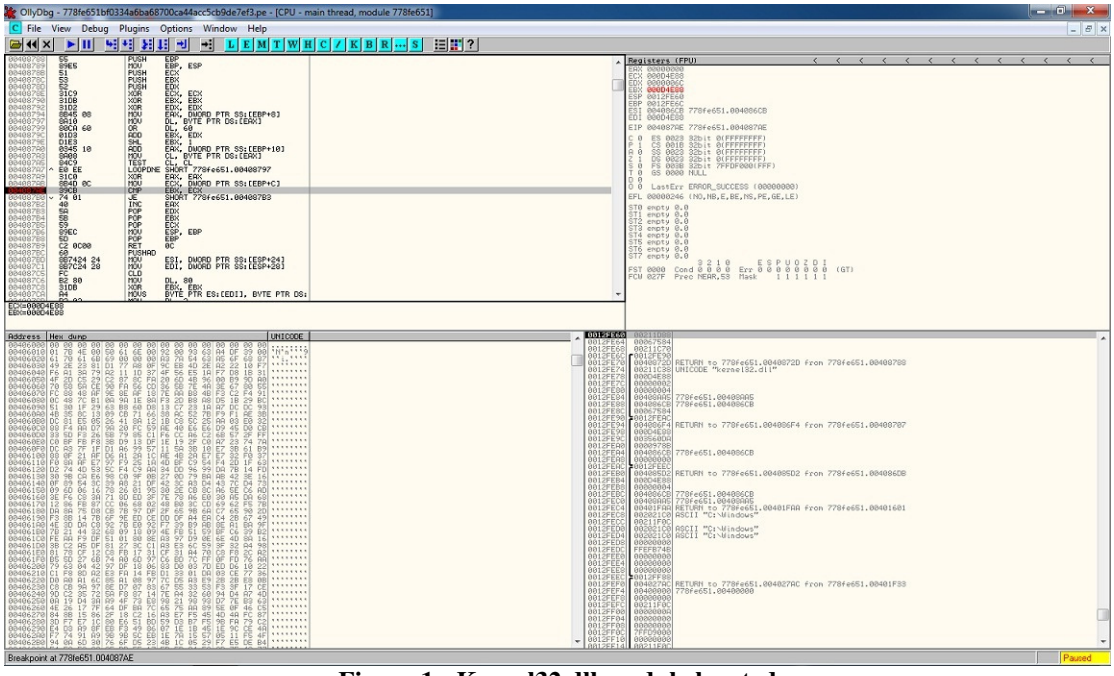

Figure 1 - Kernel32.dll module located

Once the necessary module is located, when exiting from the previous function we will reach the next part of the code that will attempt to find the VAs of specific exported functions from the kernel32.dll.

00408735 8B41 18 MOV EAX, DWORD PTR DS:[ECX+18] ← **get the image base of kernel32.dll from LDR_MODULE structure**
00408738 50 PUSH EAX
00408739 8B58 3C MOV EBX, DWORD PTR DS:[EAX+3C] ← **get the offset of its PE Header**

0040873C 01D8 ADD EAX, EBX
0040873E 8B58 78 MOV EBX, DWORD PTR DS:[EAX+78] ← **get the RVA of its Export Table**

Once the loader of the malware locates the export table of the kernerl32.dll will use it in order to retrieve the VAs of few APIs, four in total, necessary to proceed.

Here it is the table that creates at this stage:

00408AA5 760CBC8B kernel32.LoadLibraryExA
00408AA9 760D05F4 kernel32.VirtualAlloc
00408AAD 760C50AB kernel32.VirtualProtect
00408AB1 760D1837 kernel32.GetProcAddress

Locate and isolate the embedded decrypted executable

Once the VAs of the necessary APIs are stored, we are back to the next instruction after the CALL at address 004085CD that we mentioned earlier.

The piece of code that follows in also of great interest:

004085D2 64A1 30000000 MOV EAX, DWORD PTR FS:[30] ← **get address of PEB**
004085D8 8B40 08 MOV EAX, DWORD PTR DS:[EAX+8] ← **get self image base from PEB (main module)**
004085DB 8983 38153A00 MOV DWORD PTR DS:[EBX+3A1538], **EAX** ← **store self image base**
004085E1 8BBB 38153A00 MOV EDI, DWORD PTR DS:[EBX+3A1538] ← **move self image base to EDI**
004085E7 03BB 60153A00 ADD EDI, DWORD PTR DS:[EBX+3A1560] ← **add EDI a constant (AE000)**

The following five instructions are another example of obfuscation. The final result is always 10000, so it could just do MOV ESI, 10000.
However, this could be a dynamic calculation of the size of the area needed to allocate, based on the characteristics of the file wrapped with this loader.

004085ED BE 61010000 MOV ESI, 161
004085F2 03B3 5C153A00 ADD ESI, DWORD PTR DS:[EBX+3A155C]
004085F8 03B3 6C153A00 ADD ESI, DWORD PTR DS:[EBX+3A156C]
004085FE 81C6 00000100 ADD ESI, 10000
00408604 81E6 0000FFFF AND ESI, FFFF0000

Then uses VirtualAlloc API to allocate some extra memory with **PAGE_EXECUTE_READWRITE** access rights.

```
0040860A   6A 40          PUSH   40
0040860C   68 00300000    PUSH   3000
00408611   56            PUSH   ESI
00408612   6A 00          PUSH   0
00408614   FF93 25153A00  CALL   NEAR DWORD PTR DS:[EBX+3A1525]
```

← **points to Imports Table above, at the address of the VirtualAlloc API.**

Once the new memory area is allocated, it will start writing there some code.

The first code transfer takes place a few instructions later.

```
00408631   F3A4     REP    MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
```

← **ESI points to 00408993, EDI is whatever address was returned by the VirtualAlloc API, and ECX which is the counter is 161.**

Next code transfer to the allocated memory area.

```
0040865C   F3A4     REP    MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
```

← **ESI points to 00402488, EDI is whatever address was returned by the VirtualAlloc API + 4349, and ECX this time is BE.**

Next code transfer.

```
0040866A   F3A4     REP    MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
```

← **ESI points to 00403140, EDI is whatever address was returned by the VirtualAlloc API + 4407, and ECX this time is 17B7.**

Next code transfer.

```
00408678   F3A4     REP    MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
```

← **ESI points to 00406028, EDI is whatever address was returned by the VirtualAlloc API + 5BBE, and ECX this time is 255C.**

The next interesting part of loader's code is at address 0040868C where it calls a function which decrypts a portion of the code transferred to the previously allocated memory area.

```
0040896E   89CE          MOV    ESI, ECX
00408970   83E6 03       AND    ESI, 3
00408973   75 12         JNZ    SHORT  00408987
00408975   8B5D 10       MOV    EBX, DWORD PTR SS:[EBP+10]
00408978   6601DA        ADD    DX, BX
```

0040897B 6BD2 03 IMUL EDX, EDX, 3
0040897E 66F7D2 NOT DX
00408981 C1CA 07 ROR EDX, 7
00408984 8955 10 MOV DWORD PTR SS:[EBP+10], EDX
00408987 3010 XOR BYTE PTR DS:[EAX], DL ← **after performing a few calculations, it XORes the byte in the memory location pointed by EAX with the value stored in DL. Starting address is 002D4349.**
00408989 40 INC EAX
0040898A C1CA 08 ROR EDX, 8
0040898D E2 DF LOOPD SHORT 0040896E ← **loop up 3DD1 times**

Once we exit this function we meet another code transfer.

004087CA A4 MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI] ← **ESI points to the memory area on which the previous function was performing the decryption and EDI , EDI is whatever address was returned by the VirtualAlloc API + 161. ECX which is the counter is equal to 3DD1.**

Before continuing, let's take a look at how the decrypted code that is going to be transferred looks like. I am only showing a small code block from the beginning of it.

002D4349 M8Z€8 g┐┘●┼●ÿ ┼@Ñ→ xá.ã€.º|.´.Í!¸ `L.This●
prog3amÃc£n §tóbe×
002D4389 çu¯Ìi– D?OS.mode...V$D┼ PEL └ År-OX¶ àà⌂9
sSC…*┼◄ÊØ&™‘ð<0Vž@↑┐ ,;¯
002D43C9 ,┘¶„ÁR┼
h>@m!.*h?Äx¼1Î0.tXŸ4Fk ðUPXW0íC‘‰┐┴`™àQ1"‘Èð”8(¨@^.rrsRc
002D4409
Ý ÌV◄+:(àÀ3.0P4}!...┐◘²7{ï└IØÎÙB●◄bh6!è€.È×þÿU– ‰å ì┼▪⌂Ç…ðûßÀ

Well, it looks like that the decrypted code is an executable module, but obviously it is not yet well re-constructed in memory.
The code that follows, aims to reconstruct the decrypted executable in memory, and the following code block show the beginning of the executable after this has been done.

002D0161 MZ€. ...┘.┼.ÿÿ..@ .....@.....€...
002D01A1 º.´.Í!¸ LÍ!This program cannot be run in DOS mode...$.....
002D01E1 PE..L └.År-O.....à.⌂
C.@...┼...à..€& ..ð...0 ...@..┼...┐..
002D0221 .....┘ .....@ ..┼.....┐ ....┼...┼.... .....┼.....
002D0261 ¼1 .. ...0 .¼ .....
002D02A1 .....UPX0....
002D02E1 .à...┼.....┐ .....€..àUPX1.....@...ð...8...┐ .....
002D0321 ....@..à.rsrc....┼...0 ..┘ ...:.....@..À3.04.UPX!..┐◘²7{
´BÛš§← ö ò.ïwg│^└@◘ ªp– àÈà–àörÝ● w.┘

It is quite obvious that the loader just decrypted a UPX packed executable module. When this happens, the most common scenarios are other to dump this memory area to a new file and launch a child process, or transfer the execution directly to the entry point of the decrypted executable in memory.
In any case, we are going to need this executable in order to analyse the next stage, so I am going to isolate it from memory in two simple steps.
First, I am going to dump the whole allocated memory area, because I know the executable is there, and then I will cut-off all the prepended code since I don't need it any more, and save the file.

The following figure demonstrates the second step:

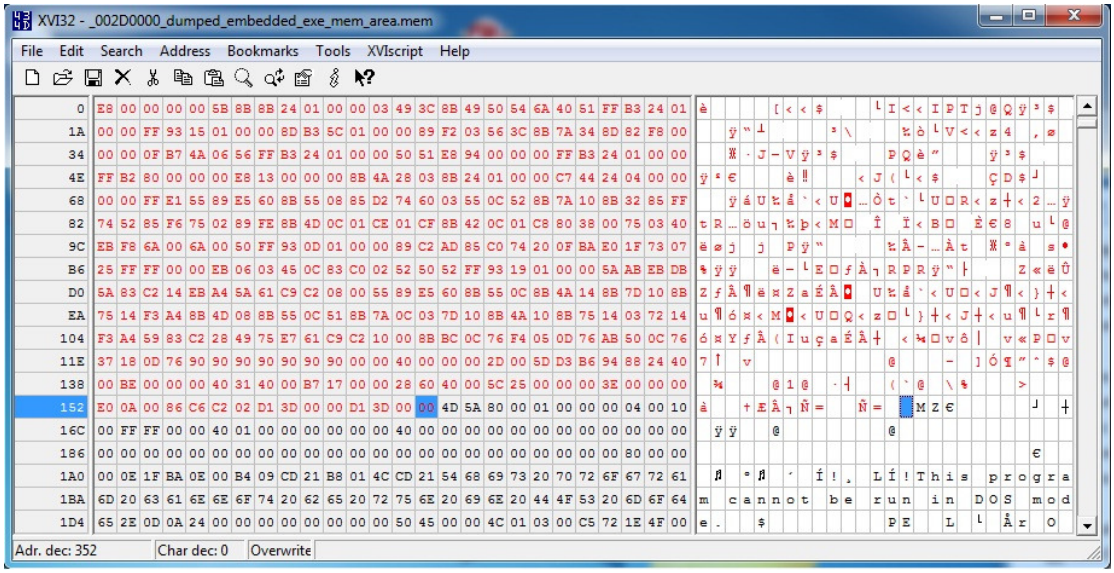

Figure 2 - Selecting prepended bytes to cut-off

At this point we can directly start working on the UPX packed executable we just saved, since anyway the loader is going to jump to its entry point in memory, after writing its code from the allocated memory inside its own's PE image address space.

002D006A FFE1 JMP NEAR ECX ← **jump to entry point of the UPX packed file.**

Going through the third stage of the loader

We can now start working on the UPX packed file we extracted from the loader's memory during the previous part of the analysis.
Manually unpacking a UPX packed file, is quite trivial so I am not going to dedicate any more lines talking about UPX, and I will continue instead with the analysis of the code of the malware's loader.

0040E5D4 55 PUSH EBP ← **OEP**

```
0040E5D5   89E5                  MOV    EBP, ESP
0040E5D7   81EC 34020000         SUB    ESP, 234
0040E5DD   C745 CC 0000000  MOV    DWORD PTR SS:[EBP-34], 0
0040E5E4   C745 D0 0000000   MOV    DWORD PTR SS:[EBP-30], 0
0040E5EB   C745 D4 0000000   MOV    DWORD PTR SS:[EBP-2C], 0
```

A few instructions later, we observe an attempt to detect if the malware is currently running inside a sandbox. I can't tell against which sandbox the following trick was tested by the author, but the here it is how it is implemented.
It actually, pushes on the stack the absolute path of the directory in which it's located and then pushes on the stack the string "sand-box", and finally uses the **strstr function** in order to check if the absolute path contains this substring.

SandBox check:

```
0040E666   51           PUSH   ECX
0040E667   50           PUSH   EAX
0040E668   FF15 AAF84000   CALL  ntdll.strstr
```

Stack View:

```
0006FD4C   0006FD54  s1="c:\users\r.c.e\desktop\matsui\upx_packed_decrypted.pe"
0006FD50   0006FF70   s2 = "sand-box"
```

In case the above check succeeds, the process will terminate.

During this stage, the loader will first copy the imports table from one location to another and then it will attempt to create a child process and inject a thread to it.
If you take a look, a few instructions later you will notice a few calls to the **memcpy function** through which copies the imports table.
Once the imports table is copied, there is a CALL to a function at address 0040E6F8.
This function is dedicated to the creation of the child process and also calls another function dedicated to the injection of the malicious thread to it.
In the next part I am going to demonstrate two ways to keep control of the execution of the injected code on the new thread, which is set to run immediately after creation.

Keep control on injected threads

By entering the function from the CALL mentioned above we can see the piece of code that launches the child process is suspended mode.

```
0040E808   56           PUSH   ESI
0040E809   57           PUSH   EDI
0040E80A   6A 00          PUSH   0
0040E80C   6A 00          PUSH   0
```

```
0040E80E   6A 04          PUSH   4
0040E810   6A 00          PUSH   0
0040E812   6A 00          PUSH   0
0040E814   6A 00          PUSH   0
0040E816   50            PUSH   EAX
0040E817   6A 00          PUSH   0
0040E819   FF15 E4F14000   CALL   kernel32.CreateProcessA
```

Stack View:

```
0006FC6C   00000000  |ModuleFileName = NULL
0006FC70   00403DB9  |CommandLine = "svchost.exe"
0006FC74   00000000  |pProcessSecurity = NULL
0006FC78   00000000  |pThreadSecurity = NULL
0006FC7C   00000000  |InheritHandles = FALSE
0006FC80   00000004  |CreationFlags = CREATE_SUSPENDED
0006FC84   00000000  |pEnvironment = NULL
0006FC88   00000000  |CurrentDir = NULL
0006FC8C   0006FCA0  |pStartupInfo = 0006FCA0
0006FC90   0006FCE4  \pProcessInfo = 0006FCE4
```

As you can see, the author chooses to launch svchost.exe as child process which wouldn't make a user suspect something through the process names from the task manager or any other process enumeration tool.
At this point we need to know the **PID** of the child process that is going to be created, which we can retrieve from the **PROCESS_INFORMATION structure** once the child process has been created. This is because since there are going to be more than one processes with the same name, which are created by Windows, we have to know which one was created by the loader of the malware in order to attach to that one later.

A few lines later at address 0040E828 will CALL the function dedicated to the injection of the malicious thread.
It will first allocate some extra memory on the child process, still in suspended mode.

```
0040E847   C745 F9 0000000   MOV    DWORD PTR SS:[EBP-7], 0
0040E84E   6A 40             PUSH   40
0040E850   68 00301000       PUSH   103000
0040E855   68 D4D50000       PUSH   0D5D4
0040E85A   6A 00             PUSH   0
0040E85C   FF75 08           PUSH   DWORD PTR SS:[EBP+8]
0040E85F   FF15 A8F14000     kernel32.VirtualAllocEx
```

Then it will use the **WriteProcessMemory** API to inject the code the allocated memory area inside the child thread.

```
0040E874   51                 PUSH   ECX
0040E875   68 D4D50000       PUSH   0D5D4
0040E87A   56                 PUSH   ESI
0040E87B   FF75 E8            PUSH   DWORD PTR SS:[EBP-18]
0040E87E   FF75 08            PUSH   DWORD PTR SS:[EBP+8]
0040E881   FF15 B4F14000      CALL   kernel32.WriteProcessMemory
```

Stack View:

```
0006FC54   00000038  |hProcess = 00000038 (window)
0006FC58   7FFA0000  |Address = 7FFA0000
0006FC5C   00401000  |Buffer = UPX_pack.00401000
0006FC60   0000D5D4  |BytesToWrite = D5D4 (54740.)
0006FC64   0006FC68  \pBytesWritten = 0006FC68
```

Method 1 - Injecting an Infinite Loop

As you can see above, the address of the start of the buffer that is going to be copied to the child process is 00401000.
So, at this point since we don't want to miss the execution of the thread, we can go to the buffer and change (in this case) the first 2 bytes from **5589** to **EBFE** which corresponds to a jump instruction that jumps back to itself, creating in this way an **infinite loop**.

Finally, the loader will start the injected thread, but remember..we had set an infinite loop at the beginning of it.

```
0040E88A   50                 PUSH   EAX
0040E88B   6A 00              PUSH   0
0040E88D   6A 00              PUSH   0
0040E88F   FF75 E8            PUSH   DWORD PTR SS:[EBP-18]
0040E892   6A 00              PUSH   0
0040E894   6A 00              PUSH   0
0040E896   FF75 08            PUSH   DWORD PTR SS:[EBP+8]
0040E899   FF15 B8F14000      CALL   kernel32.CreateRemoteThread
```

Once this step is done, we can attach to the child process and analyse the injected thread which keeps looping over the first instruction, until we go there and take control of it, and restore the two original bytes.

Method 2 - Modify EP & Memory Dump

Another trick that we can use in this case, is to wait for the loader to copy the imports table as we saw previously during the explanation of the first method, but instead of letting the loader to copy the code starting from address 00401000 to the child

process, we can set the entry point there, dump and fix the imports, as we normally do during manual unpacking practices.
This technique in this case is safe for the main reason that the code of the injected thread needs to be stand-alone in the context of the process address space in which it runs. In other words, since this piece of code it is injected inside the address space of another process cannot rely on the memory alignment of the other modules, their image base etc.
So it is safe to set the entry point at address 00401000, once the imports are copied and just dump from there and save it as a new executable file.

Conclusion

The behaviour of the loader examined during this article is very similar to the most common loaders used by various types of malwares in nowadays, such as ransomware, fake AVs etc..
Keep in mind that in most of the cases the loader at some point will make use of at least one of the following three APIs, **VirtualAlloc**, **VirtualAllocEx**, and **ZwAllocateVirtualMemory**, so it is good practice to keep an eye on them and at the memory area(s) allocated through them.
Have fun!