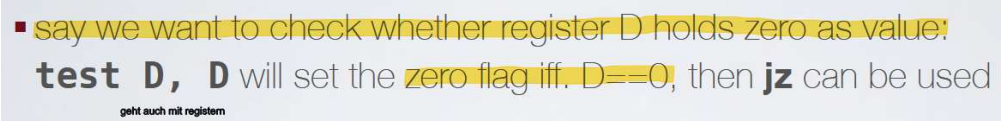
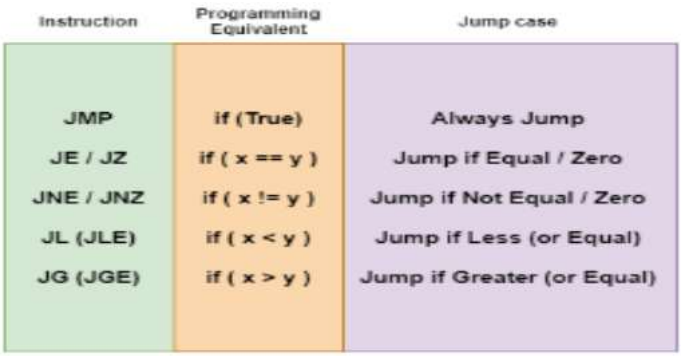
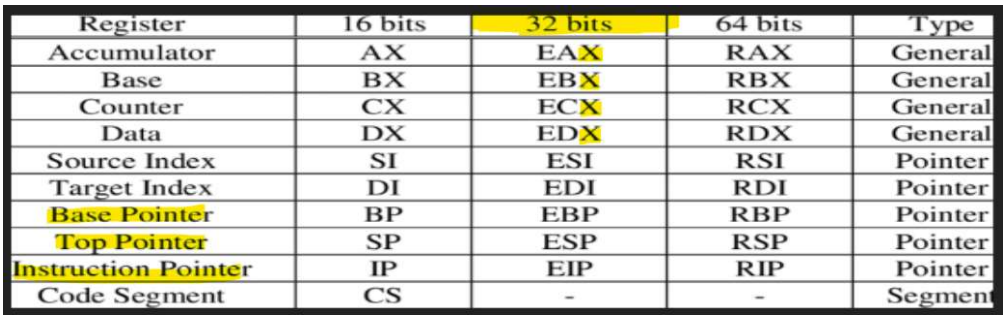
# Altklausuren

- [[Beispielklausur]]
- [[exam20190115examplesolution.pdf]]
- [[exam20190207examplesolution.pdf]]
- [[exam20200116examplesolution.pdf]]
- [[exam20200214examplesolution.pdf]]
- [[exam20200604examplesolution.pdf]]
- [[Altklausur_24.pdf]]

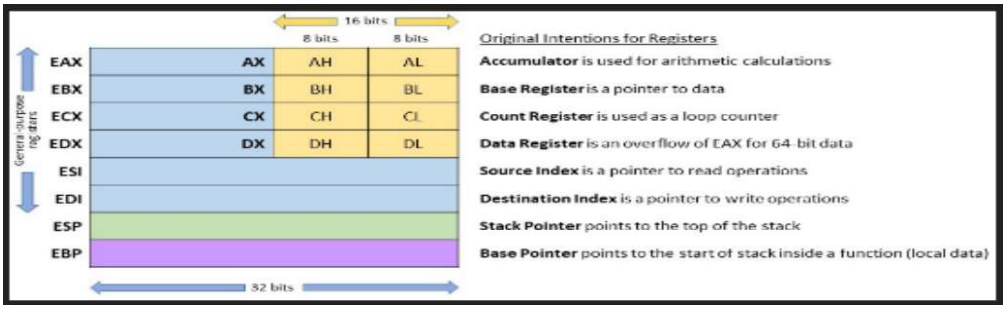
- ASCII
  - https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html
  - https://www.rapidtables.com/convert/number/decimal-to-hex.html

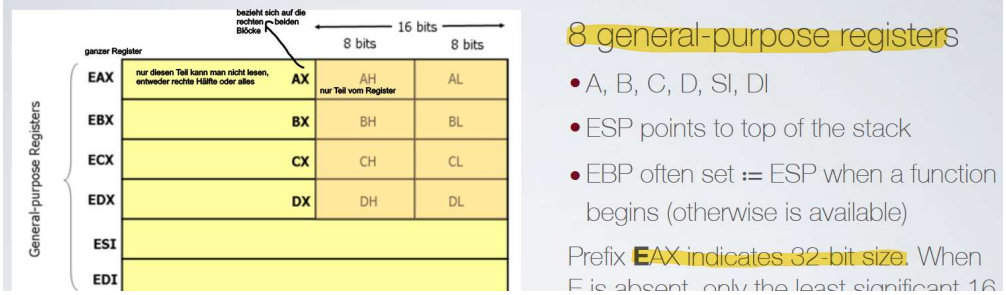
- Assembly
  - 

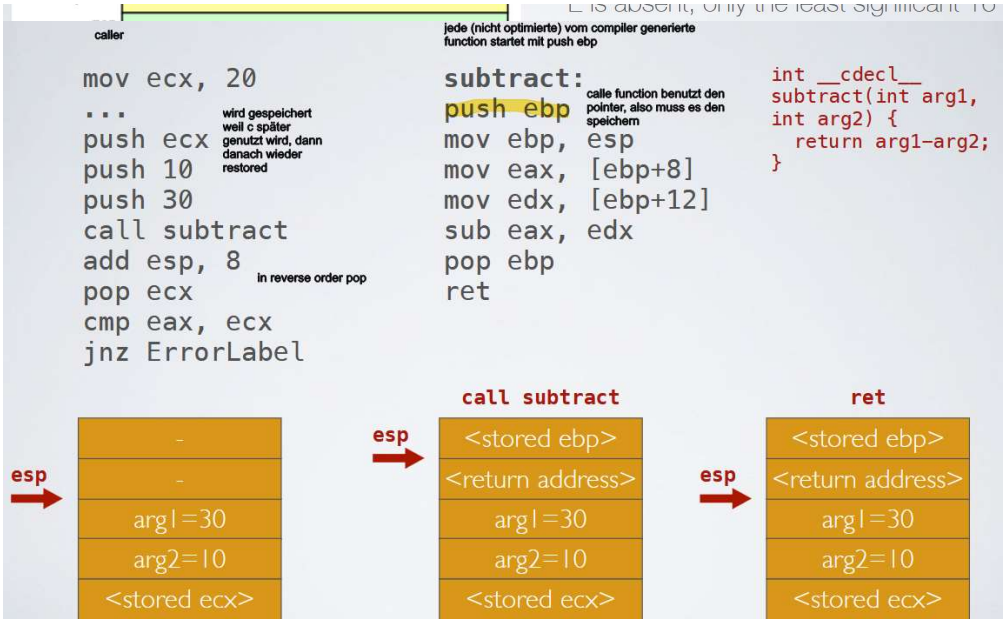
  - 

  - 

| Register | 16 bits | 32 bits | 64 bits | Type |
|---|---|---|---|---|
| Accumulator | AX | EAX | RAX | General |
| Base | BX | EBX | RBX | General |
| Counter | CX | ECX | RCX | General |
| Data | DX | EDX | RDX | General |
| Source Index | SI | ESI | RSI | Pointer |
| Target Index | DI | EDI | RDI | Pointer |
| Base Pointer | BP | EBP | RBP | Pointer |
| Top Pointer | SP | ESP | RSP | Pointer |
| Instruction Pointer | IP | EIP | RIP | Pointer |
| Code Segment | CS | - | - | Segment |

  - 

  - 



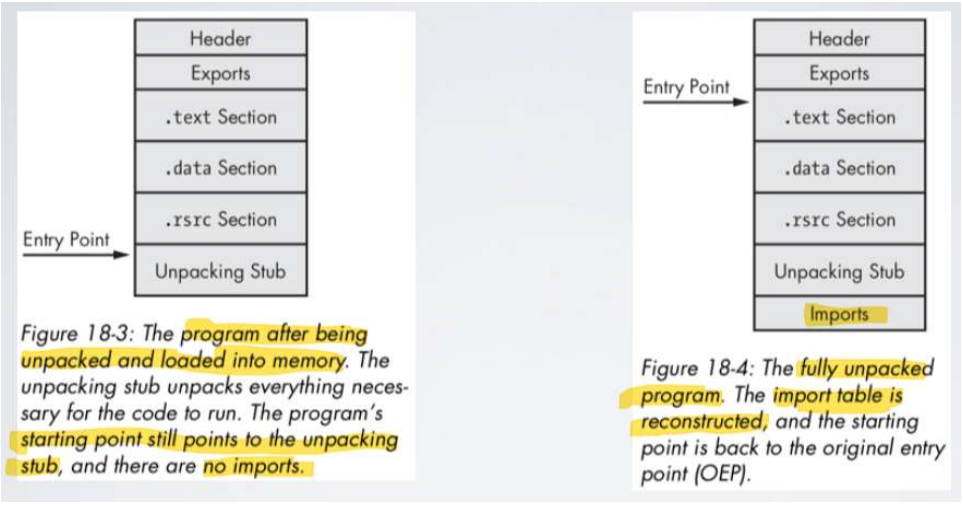

- **Struktur**
- **1** - What does a basic inspection of the PE file (e.g., header, sections, strings, resources) reveal about this sample?
  - **Vorlesung**
    - bei packing sehr wenige libraries und dlls, meistens nur welche die auch für entpacken gebraucht werden
    - *Table 2-3. DLLs and Functions Imported from PackedProgram.exe*

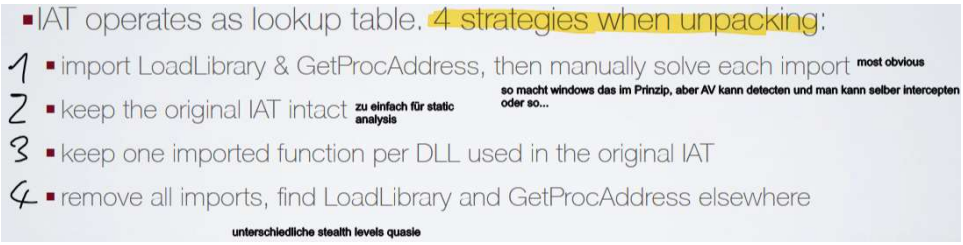
| Kernel32.dll | User32.dll |
|---|---|
| GetModuleHandleA | MessageBoxA |
| LoadLibraryA | |
| GetProcAddress | |
| ExitProcess | |
| VirtualAlloc | |
| VirtualFree | |

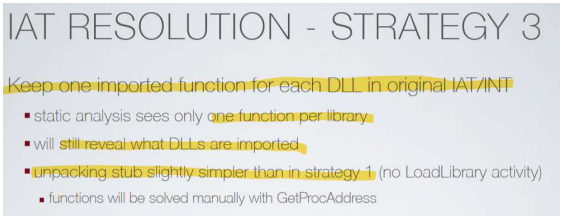
    - *Table 2-6. Section Information for PackedProgram.exe*

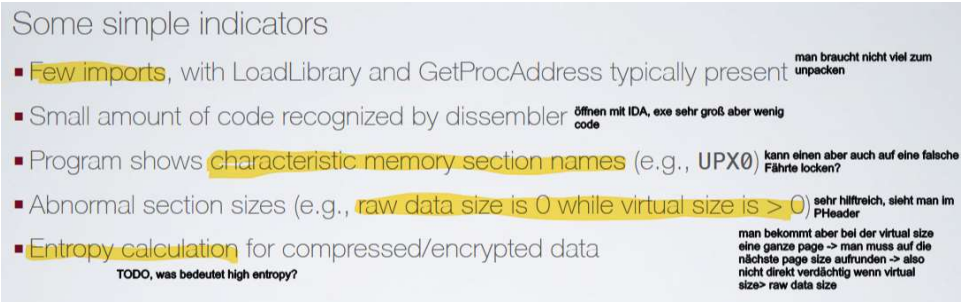
| Name | Virtual size | Size of raw data |
|---|---|---|
| .text | A000 | 0000 |
| .data | 3000 | 0000 |
| .rdata | 4000 | 0000 |
| .rsrc | 19000 | 3400 |
| Dijfpds | 20000 | 0000 |
| .sdfuok | 34000 | 3313F |
| Kijijl | 1000 | 0200 |

    - Encrypted programs have larger entropy (which is an indicator of packing, but not necessarily of malicious behavior




- **Labs**
  - [[Lab03 - 11.10.]]
    - wenige imports (z.B. 8) sind verdächtig und schließen auf packing
    - section mit großer virtual size, aber keiner war-size → wird wahrscheinlich  während runtime da rein geschrieben
    - section ist wirtable und executable → auch verdächtig
  - [[Lab06 - 08.11.]]
    - entry point ist nicht in first section
    - first section ist writable (eigentlich sonst nur readable und executable)
    - raw-size ist 0 während virtual genau 4 pages ist (4*4kb)
    - auch **hohe entropy** in section UPX1 → meistens spricht das für compression mit crypto zeug
    - hat von jeder library schon eine function, damit es sich den load library step im unpacking sparen kann?
  - [[Lab07 - 15.11.]]
    - network libs und so
    - imports
      - URLDownloadToCacheFile
    - libs
      - urlmon.dll

- strings
  - url zu http://www.practicalmalwareanalysis.com/%s/%c.png
    - naheliegend, weil webseite vom buch, hat aber auch placeholder drin → verdächtig
  - | | | | | |
    |---|---|---|---|---|
    | ascii | 29 | - | - | %c%c:%c%c:%c%c:%c%c:%c%c:%c%c |
    | ascii | 5 | - | - | %s-%s |
    - das hier verdächtig, auf jeden fall kein random stuff
    - sieht aus wie eine MAC address (6 Bereiche genau wie MAC address)
    - die substitution strings sind aber nicht zwingend verbunden mit malware
  - | | | | | |
    |---|---|---|---|---|
    | ascii | 64 | - | × | ABCDEFGHIJKLMNOPQRSTUVWXYZabcd |
    - irgendwo wahrscheinlich irgendwas mit encoding

- **Altklausuren**
  - 2019_1
    - A first inspection using **PE Studio** reveals the following:
      - Section names usually indicates UPX (as they're UPX0 and UPX1)
      - For one section (UPX0) *raw-size* is zero, while *virtual-size* is 36864 bytes, this is often an indicator of packer
      - Various strings inside the sample are referring to UPX
      - A string seems to refer to a TXT file, named *\n0r44.txt*
  - 2019_2
    - To perform a basic inspection of the header and the strings we can use the tool **pestudio**. What we can notice different things:
      - In the imports tabs there are only few imports. In addition, one of them (*GetProcAddress*) is usually used to load the address of additional functions at runtime;
      - In the sections tab the most important aspect which stands out is the name of the sections which start with the prefix MPRESS. This is a clear hint about the fact that the malware is packed with some packer;
      - Another hint of the fact that the sample is packed is the fact that the virtual-size required is much greater than the actual raw-size. In addition the tool also highlights that the entry point of the program is outside the first section and that the section MPRESS1 is writable;
      - In the libraries section we can see that there are libraries related to network communication. This means that the sample will make some network activity during its execution
      - In the strings section there are some strings like MPRESS1 MPRESS2 which enforce the belief that the malware is packed. Always in this tab, there is one string (*pad.exe*) which may be associated to an executable and other different strings which are truncated and obfuscated like *GetProc, Create* and so on;
      - Always in the string tab, it is possible to see strings related to functions which use the HTTP protocol like WinHttpConnect, WinHttpReadData which may be loaded at runtime and let us understand that there will be some communication with a possible C2.
  - 2020_1
    - A inspection using PE studio reveals:
      - the name of the three sections of the sample are UPX-like (presence of a UPX packer)
      - for one section (UPX 0) the raw-size is 0 bytes while virtual-size is 49152 bytes (indicator of a packer!)
      - various strings in the sample refer to UPX
      - various string and imports refer to windows cryptography apis
      - the entropy for the second section is big (indicator of a possible packer!)
      - one string refer to a possible executable named cmd.exe
  - 2020_2

- In order to perform a basic inspection of the header and the strings we can use the tool PEstudio. Running it we can notice the following:

  The sample has two sections, named MPRESS1 and MPRESS2. The MPRESS1 has raw-bytes = 5632 and a virtual size = 327680, which is much larger from the raw-bytes, and both sections are writable and executable, sign that the sample is going to fill it during the execution. Also the entropy of sections MPRESS1 and MPRESS2 are high (respectively 7.5 and 6.6). So we can claim that the sample is probably packed using MPRESS packer (answer 2).

  There are only 11 imports for 10 different library and we can spot GetModuleHandle and GetProcAddress; this is another indicator of a packed program because both are in general called during an unpacking procedure to load new functionality.

  From the strings we can spot multiple names of functions that may be called during execution, loading at run time. Also the string ".MPRESS1", so again probably this is the packer.

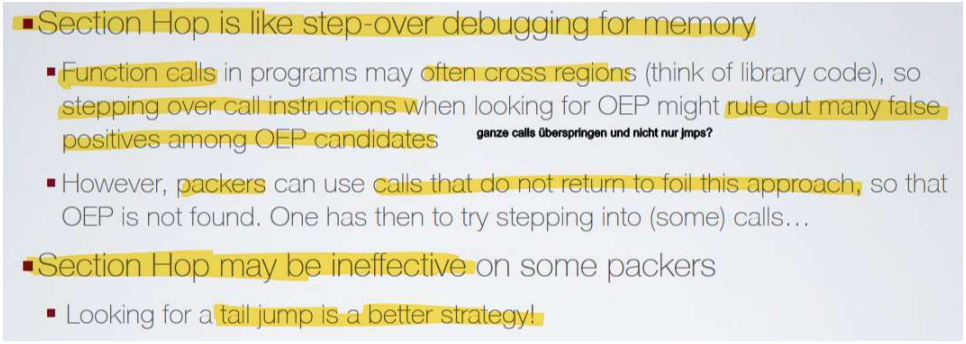
  There are also some resources that may be used by the sample to load code. In particular there are 2 images JPEG (with name 110, 112) and a PUH file with name 113.

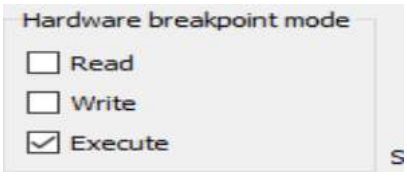
- 2020_3
  - I've used tool PeStudio in order to see some basic information of the PE header.

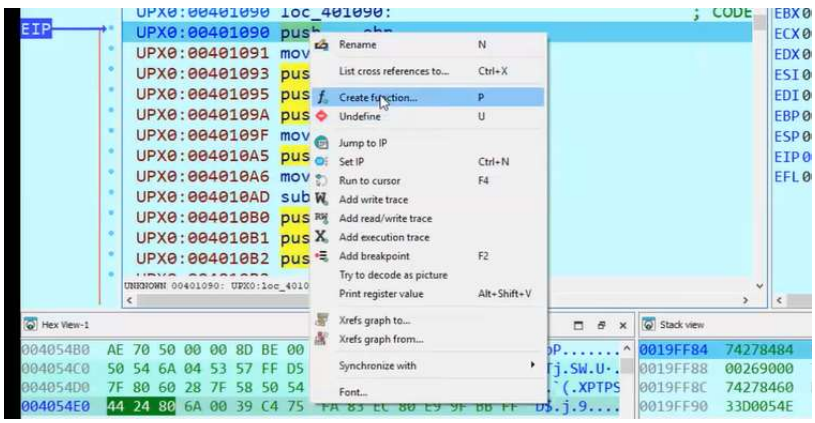
    - sections -> we see that there are 3 sections and their names (MPRESS1, MPRESS2) suggest that probably the sample is packed with MPRESS. Other indicators that suggest this hypothesis is that the entry point is out of the first section and the sections are all writable and the first 2 executable. Then we see also that the virtual size is much bigger than the raw size (especially for the first section) and the entropy of sections are high.

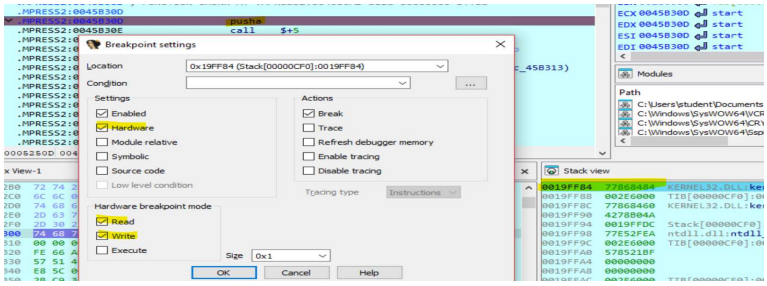
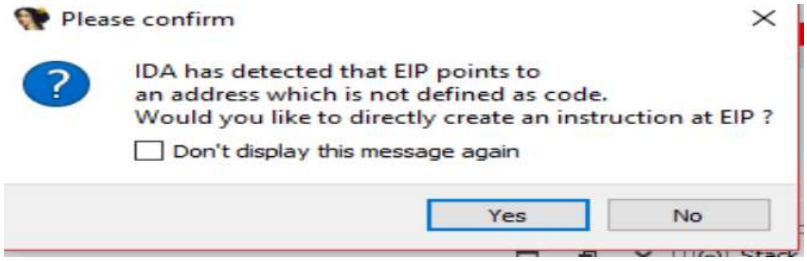
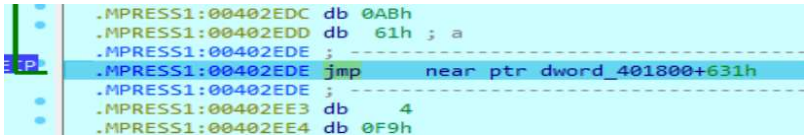
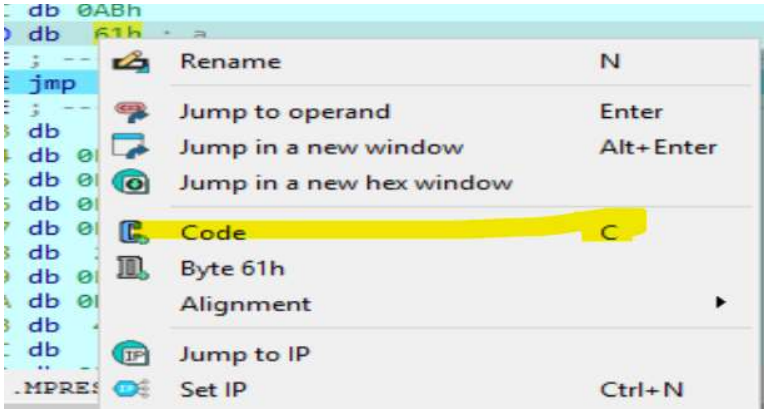
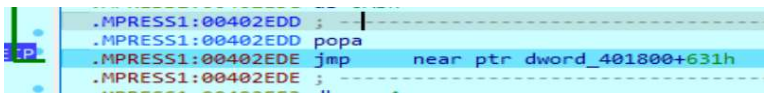
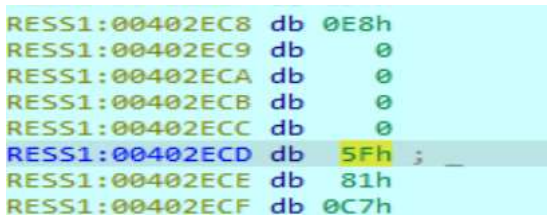
    - imports -> there are many imports:
      - GetModuleHandleA, GetProcAddress that are usually used in packed samples to resolve import addresses at runtime.
      - MessageBox that in general is used to display some messages to the user
      - a function from library ws2_32.dll, that can be used for network communications
      - GetAdaptersAddresses that may be used to take some info about the host machine
      - some functions from library "api-ms-win-crt"

    - strings -> there are 22 strings blacklisted directly by PeStudio, from which we can find the name of the libraries described above, and other ones such as "EFH'.jpg" (that may be a file used by the malware). Then we find also the names of the sections and the number 2.19 that may be the version of the MPRESS used. Then we find also other strings that may be combined together to form a significant instruction, for example "rtualFre", "IsWow6b4", "loseHand".

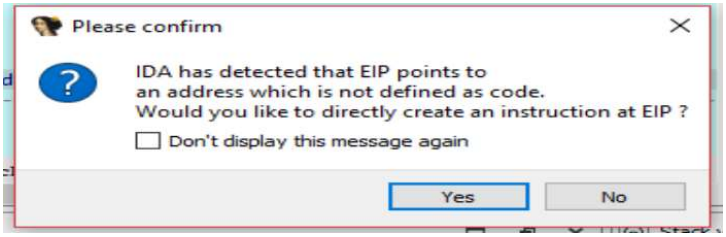
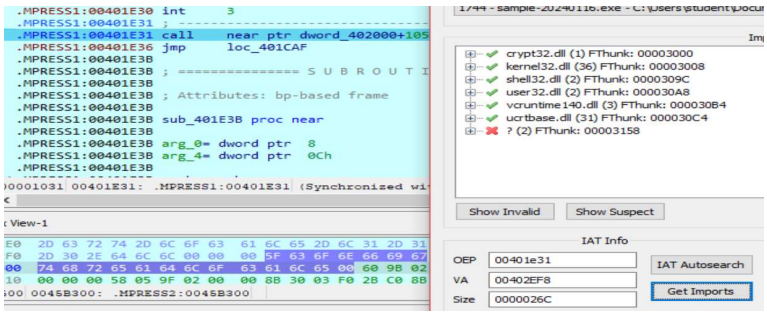
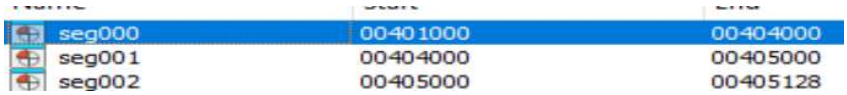
- **2** - Which packer was used to pack this sample? Provide the original entry point (OEP) and the address of the tail jump instruction, detailing how you identified them.
  - **Vorlesung**
    - 
      
    - 
      

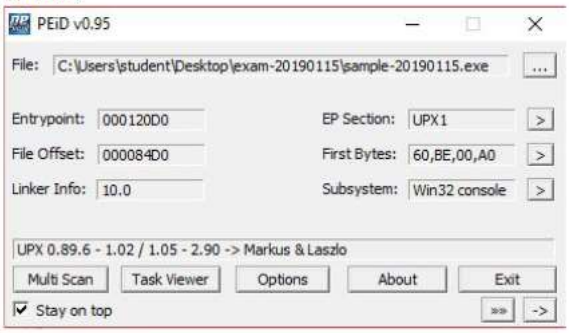
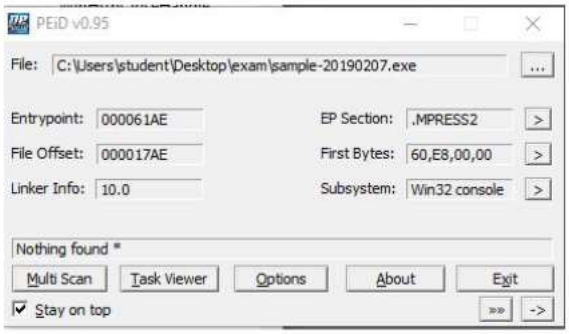
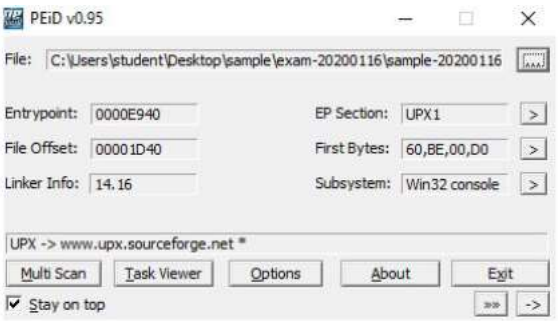

- **Labs**
  - [[Lab03 - 11.10.]]
    - [[Detect it Easy]] um gewissheit zu haben
  - [[Lab06 - 08.11.]]
    - auch wieder Detect it Easy
    - dann in [[IDA]]
      - View → Open subviews → segments (hier gucken in welchen Adressbereich man jumpen will)
      - nach text suchen
        - erst unconditional **jmp**
        - dann wenn man nicht findet conditional jumps **j**
          - **conditional jump**
            - hier kann es sein, dass man bei erreichen des breakpoints und dann step over nicht direkt dahin kommt (ist ja auch conditional), also folgendes wiederholen bis man irgendwann da ist
            - *gerade im breakpoint gelandet*
              - step over
              - *wenn nicht im unpacked code gelandet*
                - run program *und wiederholen*
            - Hardware Breakpoint geht hier **nicht?**
              - **doch** → auf das Ziel
                - wenn man den Breakpoint auf **Execute** setzt
                - 

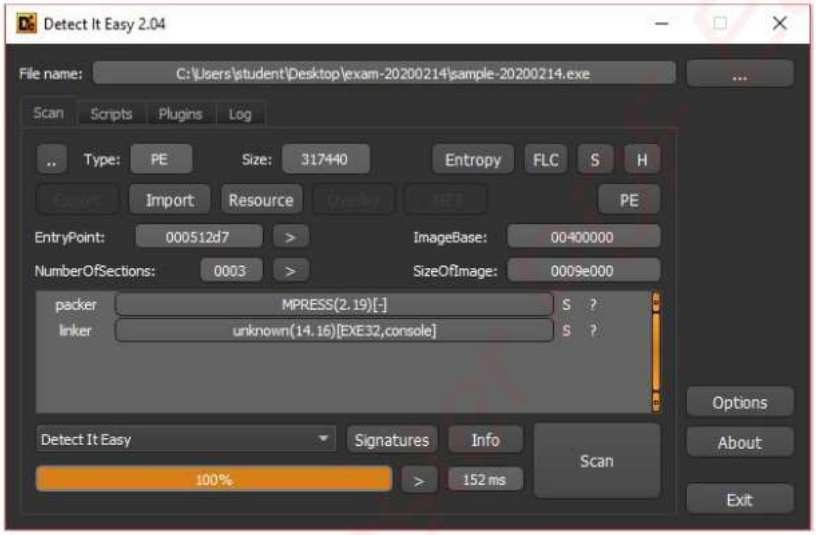
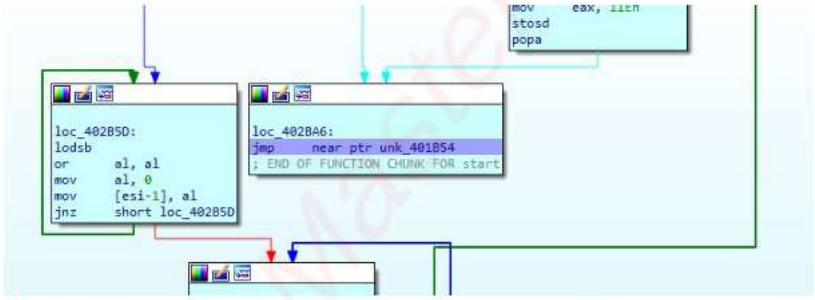
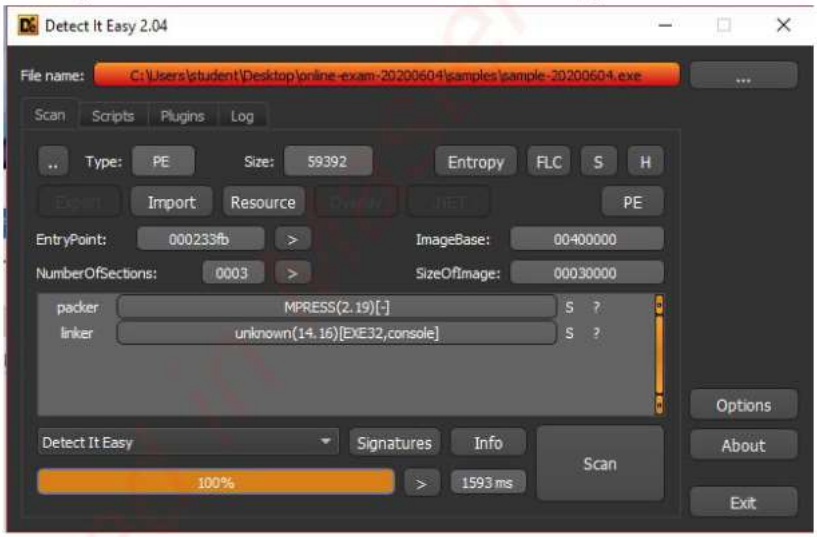
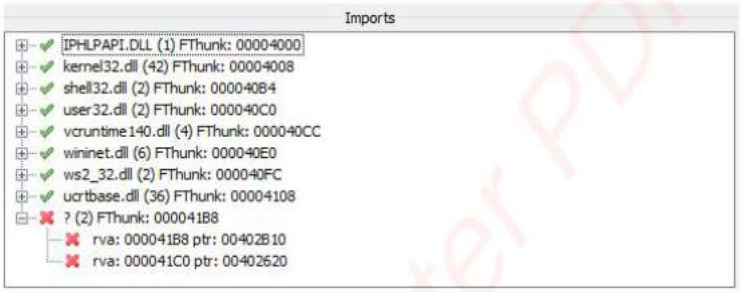
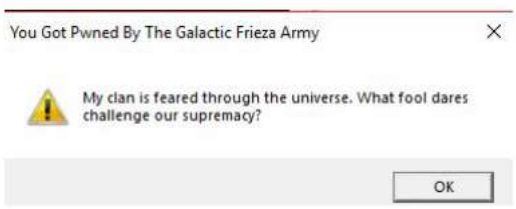
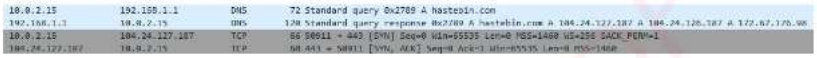
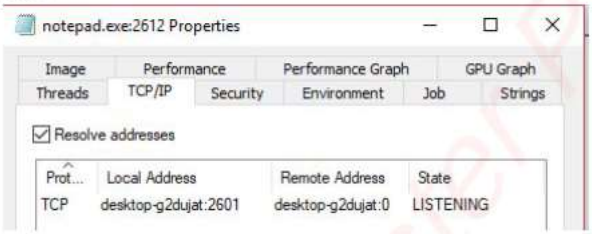
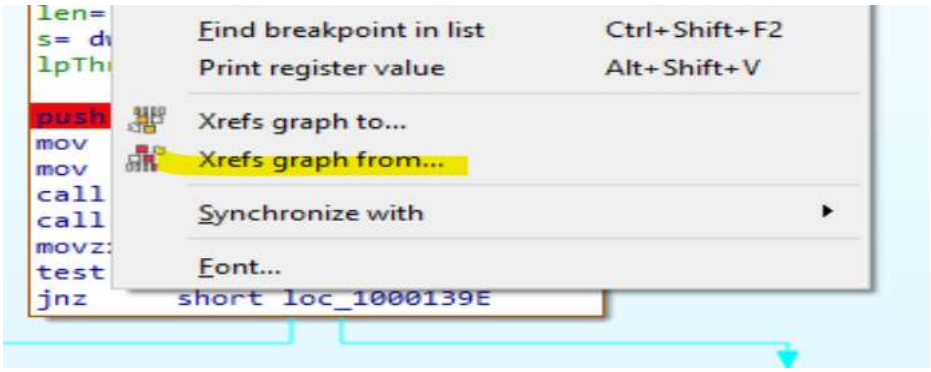
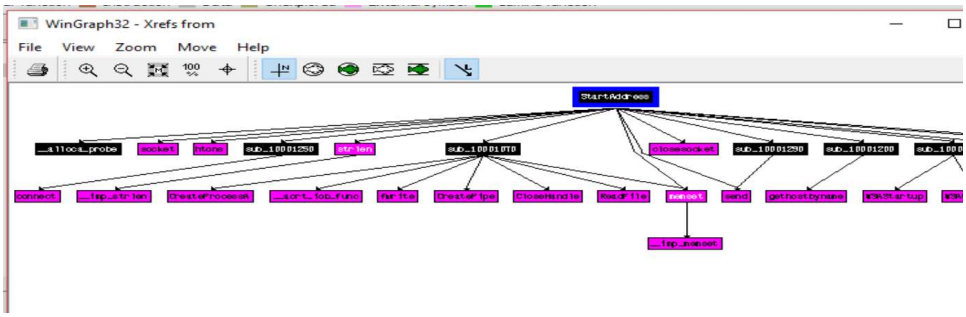
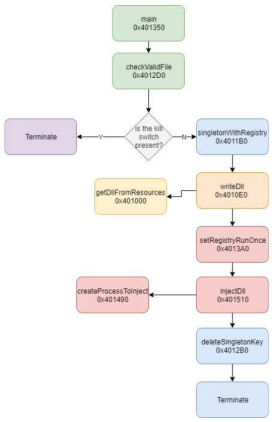
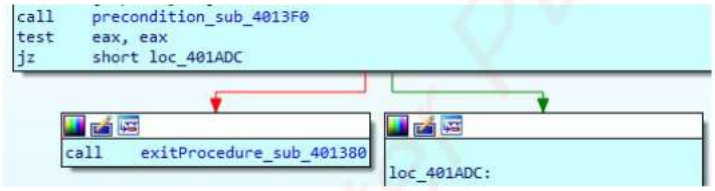
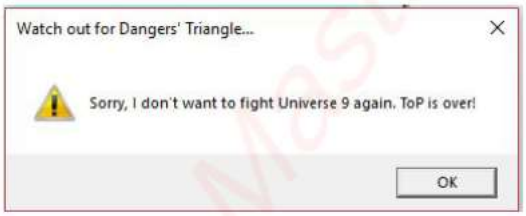
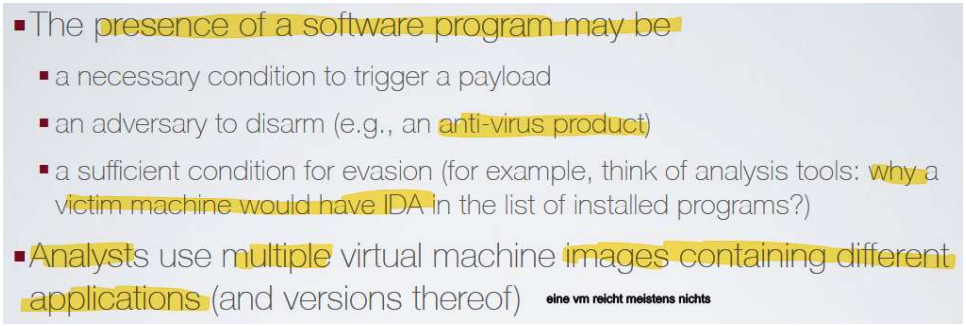
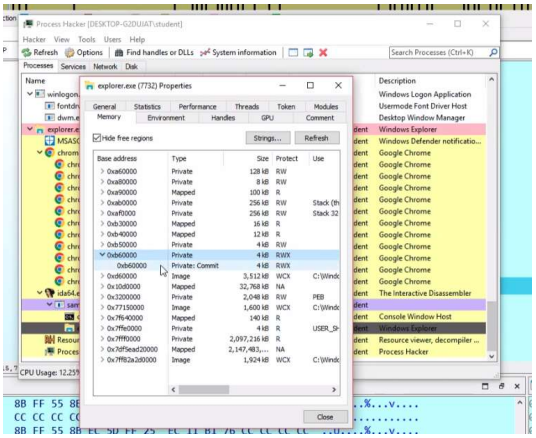
        - man kann auch auf einen jump klicken und dann ALT + UP / DOWN
      - Jump → Mark position
        - dann Jump → jump to marked positions
      - Jumps die sehr weit sind, meistens in die Section wohin unpacked wird
      - wenn man gefunden hat aus dem Code evtl. eine **Funktion erstellen**, wenn nicht automatisch gefunden wird
        - 

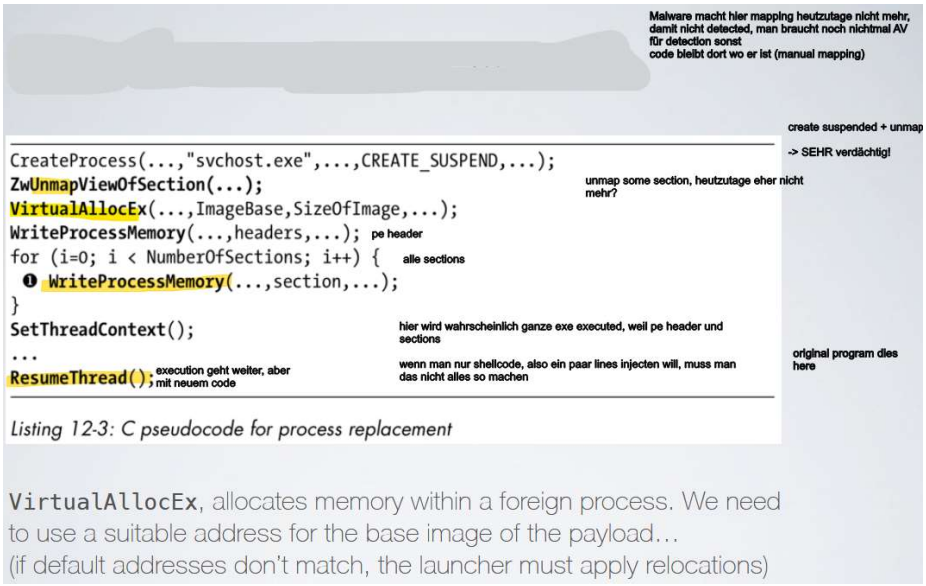
- **alternativ mit memory breakpoints**
  - wenn z.B. MPRESS genutzt wird und kein sprung in eine andere section erfolgt, sondern nur in der section überschrieben wird → sehr schwer tail jump zu finden, weil der jump nicht sehr weit sein muss und auch evtl. statisch noch nicht als code angesehen wird → memory breakpoint **pusha**, **popa**
    - **pusha** im code suchen → breakpoint setzen
    - im Debugger dann im **Stack** stelle markieren, geht evtl. erst ein step nach pusha (dann hittet der breakpoint)
      - 
    - laufen lassen und man sollte nah an den tail jump kommen
    - 
      
    - 
      
    - erkennt hier noch nicht einmal **popa** → rechtsklick → code
      - 
        
      - 
        
      - wenn man so nicht findet, nach vielen 0en suchen und danach dann schauen ob code
        - 
          
        - → hat noch mehr zusammengehörigen code gefunden

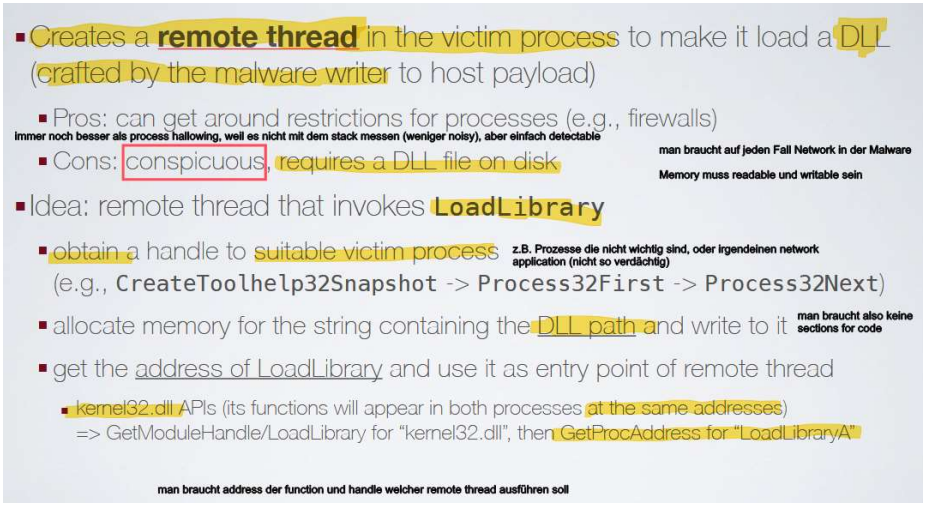
```
RESS1:00402ECC db    0
RESS1:00402ECD ; ---------------------------------------------------------
RESS1:00402ECD pop     edi
RESS1:00402ECE add     edi, 0FFFFFEEEh
RESS1:00402ED4 mov     al, 0E9h
RESS1:00402ED6 stosb
RESS1:00402ED7 mov     eax, 11Eh
RESS1:00402EDC stosd
RESS1:00402EDD popa
RESS1:00402EDE jmp     near ptr dword_401800+631h
RESS1:00402EDE ; ---------------------------------------------------------
RESS1:00402EE3 db    4
```

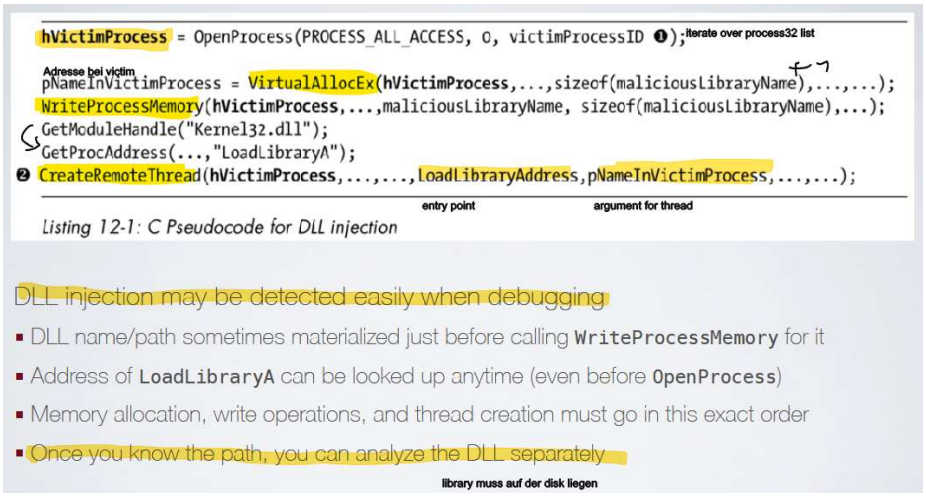
- jetzt haben wir den tail jump, aber wir müssen hier entweder ausrechnen oder dahin springen
  - Hex value:
    401800 + 631 = **401E31**
  - step into
    - 
      
    - ```
      .MPRESS1:00401E30 int     3
      .MPRESS1:00401E31 ; ---------------------------------------------------------
      .MPRESS1:00401E31 call    near ptr dword_402000+105h
      .MPRESS1:00401E36 jmp     loc_401CAF
      .MPRESS1:00401E3B
      .MPRESS1:00401E3B ; =============== S U B R O U T I N E ==
      .MPRESS1:00401E3B
      .MPRESS1:00401E3B ; Attributes: bp-based frame
      .MPRESS1:00401E3B
      ```
    - 
      
- **feststellen ob tail jump richtig ist**
- *meine Variante* (geht bestimmt nicht immer, wenn man z.B. nicht in eine neue Section geht oder section nicht leer)
  - schauen wie die Adressen für die sections sind
    - 
      
  - dann dahin scrollen, wo man denkt, dass der tail jump hinführt
    - 
      
  - section ganz kurz weil noch nicht initialisiert und es führt wie am Pfeil zu sehen nur ein jump hin
  - **x** drücken um dahin zu kommen, von wo aufgerufen → Tail jump gefunden
- *manchmal denkt man auch tail jump, aber dann jumped man erst in andere section und da kommt dann noch ein jump in die selbe section → nach popa suchen*

- **Altklausuren**
  - 2019_1
      ```

- Due to findings in (1), I executed **PEiD** to check if the sample is packed with UPX. The "mode" for PEiD should be set to "Deep mode" in order to catch the UPX signature (which was detected by PEiD).




- 2019_2




Fortunately, searching on the Internet, it is possible to see that MPRESS is a well known executable packer.

- 2020_1




So we can confirm that the packer is UPX. A test with the tool upx reveals that the exe file contains a PE sample:

```
C:\Users\student\Desktop\sample\exam-20200116>upx -l sample-20200116.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2010
UPX 3.05w       Markus Oberhumer, Laszlo Molnar & John Reiser   Apr 27th 2010

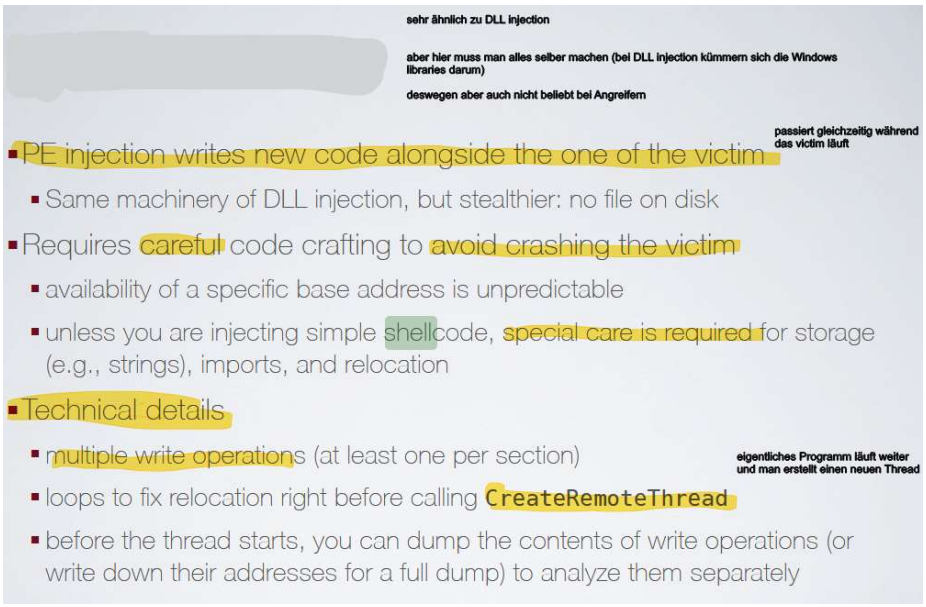
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     14848 ->      9728   65.52%    win32/pe     sample-20200116.exe
```

- The instruction that perform the tail jump is at the address 0x0040EACC. As UPX usually does, the jump is unconditional. The jump is located just above a wide area padded with zeros.

```
UPX1:0040EAC3                push    0
UPX1:0040EAC5                cmp     esp, eax
UPX1:0040EAC7                jnz     short loc_40EAC3
UPX1:0040EAC9                sub     esp, 0FFFFFF80h
UPX1:0040EACC                jmp     near ptr dword_402284
UPX1:0040EACC ; ---------------------------------------------------------------------------
UPX1:0040EAD1                align 4
UPX1:0040EAD4 __load_config_used dd 0A0h             ; Size
UPX1:0040EAD8                dd 0                    ; Time stamp
UPX1:0040EADC                dw 2 dup(0)             ; Version: 0.0
UPX1:0040EAE0                dd 0                    ; GlobalFlagsClear
UPX1:0040EAE4                dd 0                    ; GlobalFlagsSet
UPX1:0040EAE8                dd 0                    ; CriticalSectionDefaultTimeout
UPX1:0040EAEC                dd 0                    ; DeCommitFreeBlockThreshold
UPX1:0040EAF0                dd 0                    ; DeCommitTotalFreeThreshold
UPX1:0040EAF4                dd 0                    ; LockPrefixTable
UPX1:0040EAF8                dd 0                    ; MaximumAllocationSize
UPX1:0040EAFC                dd 0                    ; VirtualMemoryThreshold
UPX1:0040EB00                dd 0                    ; ProcessAffinityMask
UPX1:0040EB04                dd 0                    ; ProcessHeapFlags
UPX1:0040EB08                dw 0                    ; CSDVersion
UPX1:0040EB0A                dw 0                    ; Reserved1
UPX1:0040EB0C                dd 0                    ; EditList
```

  This instruction jumps to a new empty section (UPX0), remember that the section UPX0 have raw size equal to 0 bytes on disk.

- 2020_2
  - Due to findings in answer 1, let's check if we can find any well known packer in the sample. An analysis with Detect it Easy reveals that the packer is MPRESS 2.19.




  At the beginning the tail jump in this case is probably located at address 0x00451577. It's a simple unconditional jump that is executed after the unpacking procedure.

  This is a good candidate for the following reasons:
  - this jump points to another section which wasn't declared (or uninitialized address) as code (long jump)
  - it's the last jump executed after a cycle, which probably unpack the code writing it to the pointed section

- Then during the debugging the malware jump on this section that isn't the original entry point yet. So inspecting the new code I notice another jump to another section (with the same characteristics of the previous one). I put a breakpoint there and after running the code jump to a location that seems to be a prologue for a program.




```
.MPRESS1:00401B53 int     3                                     ; Trap to Debugger
.MPRESS1:00401B54 ; ---------------------------------------------------------------
.MPRESS1:00401B54
.MPRESS1:00401B54 loc_401B54:                                   ; CODE XREF: start:lo
.MPRESS1:00401B54 call    sub_401E28
.MPRESS1:00401B59 jmp     loc_4019D2
.MPRESS1:00401B5E
.MPRESS1:00401B5E ; =============== S U B R O U T I N E =======================
.MPRESS1:00401B5E
.MPRESS1:00401B5E ; Attributes: bp-based frame
.MPRESS1:00401B5E
.MPRESS1:00401B5E sub_401B5E proc near                          ; CODE XREF: sub_401C
```

Another technique consists in putting a hardware breakpoint in the stack after the execution of pusha instruction. Then maybe the stub that creates the code of the real executable should finishes and empty the stack that it uses.

- 2020_3
  - From precedent answer we can infer that the packer used is MPRESS version 2.19. Then we can use tool Detect It Easy to confirm this hypothesis. This tool shows that it has been effectively used MPRESS version 2.19 and it shows also the entropy that is at 98%.




To locate the tail jump, I've searched an instruction that does a jump to an address that is very far from the address of the current one. I've run it in IDA and we can see that the entry point (that points to the unpacking stub) is in the section MPRESS2, so I've looked at a jump that goes to the section MPRESS1.
So the tail jump is the instruction at address 0x0042369B "jmp near ptr byte_4038D5", that is an unconditional jump and goes at address 0x004038D5, that may be the original entry

- point of the unpacking sample. The precedent instruction may be a good candidate for a tail jump also because it is above a big area containing garbage bytes:

```
.MPRESS2:0042369B                jmp     near ptr byte_4038D5
.MPRESS2:0042369B ; END OF FUNCTION CHUNK FOR start
.MPRESS2:0042369B ; ---------------------------------------------------------------
.MPRESS2:004236A0 dword_4236A0   dd 0FFFDD960h           ; DATA XREF: start+C1r
.MPRESS2:004236A4                align 8
.MPRESS2:004236A8                dd 734Dh, 55h dup(0)
.MPRESS2:00423800                dd 200h dup(?)
.MPRESS2:00423800 _MPRESS2       ends
```

- But if in x32dbg we do a "step into", we see that the unpacked code starts earlier than the address 0x004038d5, precisaly at address 0x004027B4, that is the real entry point of the unpacking sample.
  Moreover, at address 0x004027B4,we find a jump:

```
00402784    E8 CF020000          call sample-20200604.402A88
```

  so we do a “step into” to the address 0x00402a88 that seems to be a good prologue for the unpacked sample.

- **3 -** Provide details about the IAT reconstruction process that you carried out to unpack the code. *HINTS: the answer should cover methodological aspects and facts on your output; also, validate it! (e.g., check API calls, compare with sample-20240116-unpacked.exe).*
  - **Labs**
    - [[Lab06 - 08.11.]]
      - wenn man mit dem Debugger von [[IDA]] dann am OEP point ist in [[Scylla]] den Prozess hinzufügen
      - → OEP eingeben → IAT Autosearch → Get Imports → Fump → **Fix Dump**
    - zum **Vergleichen**
      - in [[PeStudio]] imports vergleichen
      - in [[IDA]] schauen ob functions richtig gecallt werden und so
  - **Altklausuren**
    - 2019_1
      - UPX, unconditional jump to new section
      - The tail jump address is 0x00412253. As UPX usually does, the jump is unconditional. The jump is located just above a wide area padded with zero.

```
0041224E    75 FA          jne sample-20190115.41224A
00412250    83EC 80        sub esp,FFFFFF80
00412253    E9 F2F5FEFF    jmp sample-20190115.40184A
00412258    48             dec eax
00412259    0000           add byte ptr ds:[eax],al
0041225B    0000           add byte ptr ds:[eax],al
0041225D    0000           add byte ptr ds:[eax],al
```

      - I loaded the sample in **x32dbg**, and set a breakpoint at the tail jump identified above in (3). Then, I used the “step into” function of the debugger to made the program jumps to the un-packed executable and stop.

        Then, I opened **Scylla** (which was already set to the OEP 0x0040184A) and clicked on “Dump” to dump the unpacked executable on disk. Right after the dump, I made **Scylla**
    - 2019_2
      - At some point, at address *0x0040644E* there is a jump which goes outside the memory region in which *cip* is. By *right-click → follow in dump → 402460* it is possible to see that the memory section to which the instruction jumps is the following:

| Address | Hex | ASCII |
|---|---|---|
| 00402460 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 00402470 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 00402480 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 00402490 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024A0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024B0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024C0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024D0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024E0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 004024F0 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 00402500 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
| 00402510 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |

        The fact that the jump instruction is located outside of the code, it goes to an address far from the current section and it is followed by instructions interpreted as add instructions, these are hits of the fact that this is a good candidate for a tail jump.
        To confirm that this is the real tail jump, it is possible to place a software breakpoint at the address of the jump and see that the previous memory dump changed.
      - Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Lc

| Address | Hex | ASCII |
|---|---|---|
| 00402460 | 8B 78 04 0B FF 50 74 35 8B 50 08 8B 30 03 F0 2B | ‹x..ÿPt5.P..0.ð+ |
| 00402470 | F2 8B DE 8B 48 10 2B CB 74 23 8B 50 0C 03 F2 03 | ò.Þ.H.+Ët#.P..ò. |
| 00402480 | FE 2B C0 2B D2 0B D0 AC C1 E2 07 D0 E8 72 F6 0B | þ+A+Ò.Ð¬ÁâÐèrö. |
| 00402490 | D0 0B D2 74 08 03 DA 29 0B 3B F7 72 E4 E8 00 00 | Ð.Òt..Ú).;÷räè.. |
| 004024A0 | 00 00 58 05 E6 00 00 00 8B 30 03 F0 01 40 06 01 | ..X.æ....0.ð.@.. |
| 004024B0 | 40 0C 58 03 00 0A C0 75 6F 2D 00 10 00 00 80 38 | @.X...Àuo-.....8 |
| 004024C0 | 4D 75 65 8B 78 3C 03 F8 2B C0 66 8B 47 14 03 F8 | Mue.x<.ø+Àf.G..ø |
| 004024D0 | 83 C7 3F E8 09 00 00 00 4B 45 52 4E 45 4C 33 32 | .Ç?è....KERNEL32 |
| 004024E0 | 00 E8 A6 00 00 00 0B C0 74 3E E8 0F 00 00 00 56 | .è¦.....À t>è....V |
| 004024F0 | 69 72 74 75 61 6C 50 72 6F 74 65 63 74 00 50 E8 | irtualProtect.Pè |
| 00402500 | 8E 00 00 00 0B C0 74 20 50 54 6A 04 6A 78 57 8B | ....À t PTj.jxW. |
| 00402510 | D8 FF D0 0B C0 74 10 B0 60 88 07 88 47 28 58 50 | Øÿð.Àt.°`...G(XP |

        By clicking on *Step into* it is possible to reach the first instruction of the unpacked sample.
        However, just trying to reconstruct the IAT starting at this address, **Scylla** is not able to reconstruct it.
        After having stepped into and arriving at instruction *0x00402460*, in the end it is possible to see the instruction at address *0x00402583* which is a *jmp* which brings to a *call* instruction at address *0x00401A98*. In turn, this call instruction brings to address *0x00401E1B* which seems to be a good prologue for a program. Indeed, starting from the address of the call instruction *0x00401A98*, we can reconstruct the IAT using Scylla.

1. Pressing the button *IAT Autosearch* to obtain the IAT information starting from the OEP (*0x00401A98*). At this point the plugin retrieves its virtual address and the size;
2. Then, press the *Get Imports* button to retrieve a list of imported functions. There is an invalid entry (denoted with a red cross) in the table: just *right-click* on it and select *Delete tree node*;
3. At this point, click on the button *Dump* to dump the memory of the process (a file with the suffix *_dump* is created);
4. Lastly, click the button *Fix Dump* loading the file created at step 3. so that a new file (with the suffix *_SCY*) is created and it is the dump of the process with the reconstructed IAT.

By opening with **pestudio** the extracted sample and the one provided already extracted, it is possible to see that the imports are the same.

- 2020_1
  - Putting a breakpoint on the instruction that perform the tail jump we can see that the section UPX0 is populated with the unpacked program. In fact it starts with a call to a subroutine the contains the prologue of the unpacked program.

```
UPX0:00402284
UPX0:00402284 loc_402284:                             ; CODE XREF: UPX1:0040EACC↓j
UPX0:00402284 call    sub_402558
UPX0:00402289 jmp     near ptr dword_401000+1102h
UPX0:00402289 ; ---------------------------------------------------------------------------
```

```
0000000000402558
0000000000402558
0000000000402558
0000000000402558 sub_402558 proc near
0000000000402558 mov     ecx, ___security_cookie
000000000040255E push    esi
000000000040255F push    edi
0000000000402560 mov     edi, 0BB40E64Eh
0000000000402565 mov     esi, 0FFFF0000h
000000000040256A cmp     ecx, edi
000000000040256C jz      short loc_402572
```

  So the original entry point of the sample is at the address 0x00402284.
  In fact Scylla can reconstruct the IAT from the OPE 0x00402284:
- even though 2 imports are missing, it has the same number of imports as unpacked
- 2020_2
  - We opened the sample in IDA, putting a breakpoint in the location of the tail jump. Then we ran it using the debugger and jump to the location of the OEP.

    Afterwards, we opened Scylla, attaching it to the sample's process and setting the OEP to 0x00401B54 as we said before. We click on IAT autosearch, letting Scylla to use its heuristics, and get imports in order to get informations on the IAT and its size.

    Scylla found the IAT at address 0x00402BC0 with size 584, with an invalid tree that we can safely remove. So, we dump the sample and fix it to obtain the final unpacked sample.
- 2020_3

To do the IAT reconstruction, I've used the tool Scylla, attached to the sample running on x32dbg. I've done these steps:
1. Insert in OEP the address 0x004027B4, that is the entry point of the original unpacked program
2. Click on "IAT Autosearch", so Scylla finds an import address table at address 00403A10 and of size 000007B4.
3. Click on "Get Imports" to load the functions in the IAT found; the result is this:




so Scylla has done correctly the import of all the APIs except 2; we can remove them by clicking on "Delete tree node".
4. Click on "Dump" to dump the memory of the process in the file "sample-20200604_dump"
5. Click on "Fix Dump" on the file "sample-20200604_dump" to generate the file "sample-20200604_dump_SCY"

If we do a comparison with the number and type of APIs imported by "sample-20200604-unpacked" (again through PeStudio), we see that all 95 APIs have been reconstructed correctly in the file that we have extracted, so we don't need the 2 invalid APIs, because we just have all the APIs needed to run the malware. In fact with PeStudio on "sample-20200604-unpacked" we see that:

- *unpacked*
- **4** - Provide a brief, high-level description of the functionalities implemented by the sample (what it does, when, how). Try to keep it short (like 10 lines). Reference answers to other questions wherever you see fit.
  - **Altklausuren**
    - 2019-1
      - The sample:
        - Checks for the presence of a kill-switch (if not exists, exits) (answer 10)
        - Use a registry key as singleton (answer 8)
        - Retrieve the DLL from the resource part, decrypt it and write it to a file in user's document directory (answer 11)
        - Activate a persistence mechanism where the DLL is executed on the next login (answer 9)
        - Starts and inject *notepad.txt* with the malicious DLL (answer 11)
        - Remove the singleton registry key (answer 8)

      The injected DLL does:
      1. Creates a thread which waits for 3 seconds
      2. Then shows a message box and launch a network subroutine
      3. The network subroutine contacts the C2 using an HTTP request to *kettle667.biz* if exists, otherwise it uses the IP address 192.168.50.26 (see answer 13)
      4. A command received from the C2 is executed (see answer 14)
    - 2019_2
      - The malware checks the current time: if it is greater or equal than 17:00 or less than 12 it continues, otherwise it exits. Then, it retrieves the IP address of the local machine establishing a
      - connection with the server located at 104.18.49.20. The information sent to the server are a crafted user-agent string and information related to the port from which the sample was connecting. After that, the malware performs shellcode injection on the victim process svchost injecting some payload.
    - 2020_1

- The malware initially create a persistence file inside the download directory contains the sample (using the CopyFileA function) and set the attribute of the file to a hidden file. The name of the file is C0dD410.exe
  Subsequently, checks for all the active/running processes in the system and check if exists a process who is running the sample. If the process exists, load dynamically the function ExitProcess (from a base64 string, see answer 12) and call it to exit.
  Subsequently, open the registry key HKEY_CURRENT_USER\Software\Microsoft\WIndows\CurrentVersion\Run and create a value "spaceship" containing the following path C:\User\student\DOwnloads\C0ldD410.exe in order to execute the sample at each login.
  Subsequently, create a process cmd.exe using the process the execute the sample, load dynamically a lot of functions (see answer 12) and overwrite the memory of the process injecting some shellcode.
  The shellcode shows a popup:




  At the end, the malware performs some network activity connecting to the C2 server at tweetplug.dis.uniroma1.it (ip: 151.100.101.140, port 80) sending some encoded (base64) beaconing data (adapter IP of the local machine and the local time) to receive some commands from the C2:
  - command Q: exit
  - command S: sleep for 4600 seconds
  - command G: perform a directory listing of the desktop directory and send it to the C2 server

- 2020_2
  - The sample does the following:
    - Check for the presents of some programs running, if yes it will exits and change the desktop image
    - Copy the executable in multiple folder
  - - Make the injection, creating the notepad process and injecting a dll extracted from the resource
    - Change the desktop image

    The Dll injected does the following:
    - Make a query to the registry
    - Make a DNS request to 8.8.8.8
    - Send and receive data
- 2020_3
  - The malware does the following principal operations:
    - creates mutex object to guarantee that only one instance of the malware at a time can be executed
    - does a precondition check to see if there is a process module name equal to "basil.exe", "bergamo.exe" or "lavender.exe"; if yes, it does an exit procedure in which create an image and uses it as desktop background
    - then creates a copy of itself in the file "\sniper.exe"
    - creates process "notepad.exe" and injects the shellcode in it
    - retrieve many information about the host and sends them through HTTP POST request to server hastebin.com", at port 443

- **5 -** List the processes, registry keys, files, and network connections created/manipulated by the sample and its byproducts (e.g., injected payloads, second-stage executables), if any, during their functioning. Detail the methodology you used to acquire this list. (Come back to this question to complete it as you acquire further details during the test)
  - **Altklausuren**
    - 2019_1

| Type | Tool | Description |
|---|---|---|
| Network connection | Wireshark<br>Fakenet<br>apateDNS<br>ProcMon | A DNS beacon is sent to *.kettle667.biz (the actual name depends on the machine and current time/date), if failed, an HTTP GET / is made to 192.168.50.26 in order to contact C2 |
| Registry key open/write/close | ProcMon | The un-packed exe creates a new key (answer 8) as singleton mechanism, and a key for DLL run-once at boot (answer 9) |

| | | |
|---|---|---|
| Process creation with injection | ProcMon, ProcessExplorer | The un-packed EXE launches *notepad.exe* and inject DLL in to it. |
| File creation | ProcMon | A DLL named *konrAd.dll* is created in the user's document folder |

- 2019_2
  - - Using IDA, it is possible to find the following information:
      - It is possible to see that the malware tries to dynamically load the function *CopyFileA* passing a path and appending the file name *\sullivan.exe* which may possibly means that it creates that file copyting something into it;
      - By a call to *WSAStartup* it is clear that the malware initializes the library for network connection;
    - By opening **procmon** and running the malware, the interesting things are:
      - for the registry:
      - FOR THE FILESYSTEM:
        - The sample tries to open some files which should be located into the asme folder in which the malware is. The names are like *MSVCR100.dll, WINHTTP.dll* etc.
        - The sample creates a file named sullivan.exe and copies itself into it.
      - for processes:
        - it is possible to see that the malware spawns the process *conhost.exe*
        - is possible to see that it spawns the process svchost.exe
    - Running the program and looking at network activities through **Wireshark**:
      - It is possible to see that it does a dns query searching for the IP of api.paste.ee and it is successful;
      - It receives also some ack by the same server which makes possible for the malware to continue its execution.
- 2020_1
  - Network connection
    - tools used: Wireshark, Fakenet, apateDNS
    - A DNS request (type A) is sent to tweetplug.dis.uniroma1.it to resolve the IP address.
    - A connection to 151.100.101.140 (port 80) is performed to sent the encoded (base64) beaconing data and receive some commands
    - In the case of the command "G": another connection is performed to 151.100.101.140 (port 80) to send a directory listing of the desktop directory

    Registry key open/write/close:
    - tools used: ProcMon
    - The un-packed exe creates a new value inside HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run called spaceship containing the following path C:\User\student\DOwnloads\C0ldD410.exe to execute the sample at each login

    Process creation with injection:
    - tools used: ProcMon
    - The un-packed exe launches cmd.exe using the process that execute the sample and inject a shellcode (shows a popup) into it.

    File creation:
    - tools used: ProcMon
    - A executable named C0ldD410.exe is created in the download directory containing the sample.

  - Registry key involved:
    - HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\spaceship for persistence mechanism

    Files involved:
    - C:\Users\student\DOwnloads\C0ldD410.exe containing the sample for the persistence mechanism
- 2020_2

- The sample when it exits create a file used a desktop image:
  C:\Users\student\Pictures\pride_troopers.jpg.
  We can see file during the debugging in IDA and inspecting the Desktop of the pc

```
push    27h ; csidl
push    0 ; hwnd
call    ds:SHGetFolderPathA
mov     ecx, [ebp+arg_8]
push    ecx
lea     edx, [ebp+pszPath]
push    edx
push    offset aSS_1 ; "%s\\%s"
push    104h
lea     eax, [ebp+pszPath] ; C:\Users\student\Pictures\pride_troopers.jpg
push    eax
call    sprintf
add     esp, 14h
lea     ecx, [ebp+pszPath]
push    ecx ; lpFileName
mov     edx, [ebp+lpType]
push    edx ; lpType
movzx   eax, [ebp+arg_0]
push    eax ; __int16
call    sub_4017E0
add     esp, 0Ch
lea     ecx, [ebp+pszPath]
push    ecx ; Src
call    ds:_strdup
add     esp, 4
```

  - Various copy of the sample in different location with name cell_jr.exe (answer 7). We detect them using ProcMon and using IDA.
  - The Dll created at run time in the folder C:\Users\student\mirai.dll. We see it during debugging in IDA.

```
.data:00404040 ; CHAR aCUsersStudentM[264]
.data:00404040 aCUsersStudentM db 'C:\Users\student\mirai.dll',0
.data:00404040                                         ; DATA XREF: sub_4012B0+3To
.data:00404040                                         ; sub_4012B0+1B7o ...
```

  - In subroutine sub_401230 there is the creation of the process notepad.exe in which the sample make the injection. We detect it using IDA and ProcMon. Here the sample make also a check if the current process is running in 32 bit mode.

- 2020_3
  - We can use the tools ProcMon, Process Explorer, Process Hacker and Wireshark to see practically what the malware does.
  - - process activity -> creates “conhost.exe” (that is the shell) and “notepad.exe” (in which it does a shellcode injection”; then copies itself inside \sniper.exe


    - registry activity -> no relevant operations
    - file activity -> in case the preconditon subroutine fails, it creates “C:\Users\student\Pictures\trio_of_danger.jpg”
    - network activity -> the sample does an HTTP connection to server “hastebin.com”, at port 443.In particular, first it does a DNS query to retrieve the IP address (104.24.127.187), and then sends an HTTP message:



    Then also the shellcode performs a network activity, in particular it’s listening on TCP port 2601:




- **6 -** List the subroutines used by the sample and its byproducts (e.g., injected payloads, second-stage executables), if any, to implement its main functionalities and provide a sketch of the execution transfers among them (e.g sketch a tree/graph).
  - At top **Xrefs graph from...**

- 

- 


- **Altklausuren**
  - 2019_1
    - ...
    - 

    - ...
  - 2019_2
    - sub_401540 (calls WSAStartup to initialize network libraries)
    - sub_4014D0 (calls GetLocalTime and does some check to understand whether to proceed or not. It checks whther ecx >= 17 or if edx <= 12, if neither one of them is right, it exits)
    - sub_401400 ()
      - sub_4017D0 (retrieves the path till the sample)
      - alloactes 260 bytes on the heap
      - retrieves the path of the startup folder C:\\ … \Startup
      - concatenates the two strings until getting C:\\ … \Startup\sullivan.exe
  - 2020_1

- Subroutines in the sample are:
  - the "**main**" subroutine at 0x00401710
  - **createPersistenceFIle** at 0x00401740
    - get the path to download folder
    - get the path to the sample
    - concatenate the path to download folder and the string C0ldD410.exe
    - copy the sample inside the file C0ldD410.exe (using the function CopyFileA) in the download folder
    - set the file attribute to hidden file
  - **checkProcesses** at 0x00401BB0
    - iterate over the processes using the function K32EnumProcesses and check if exist the process of the sample
    - if exist or the call to K32EnumProcesses fail set eax to 1, else set to 0 (some flag)
    - if eax is 1 call **exitProcess** at 0x00401680
      - decrypt two base64 strings: one containing kernel32.dll and one containing ExitProcess using the function CryptBinaryToStringA
      - load dynamically the function ExitProcess function using GetModuleHandleA and GetProcAddress
      - call ExitProcess to exit
  - **registryRunPersistence** at 0x004017E0
    - open the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run and create a value spaceship with the value: C:\Users\student\DOwnloads\C0ldD410.exe to execute the sample at each login
  - **shellcodeInjection** at 0x00401520
    - create the process cmd.exe using the process that execute the sample (in the subroutine at 0x004014A1)
    - load dynamically using GetModuleHandleA and GetProcAddress and store in memory a lot of functions:
      - CreateRemoteThread
      - NtUnmapViewOfSection
      - SetThreadContext
      - SuspendThread
      - ResumeThread
      - VirtualAllocEx
      - WriteProcessMemory
    - use VirtualAllocEx to allocate memory on the process cmd.exe and use WriteProcessMemory to write the payload (shellcode injection) into the process. The shellcode starts at offset 0x00403208.
    - call CreateRemoteThread to execute the injected shellcode.
  - **networkActivity** at 0x00401070
    - call WSAStartup to initialize the networking library
    - call **getBeaconingInfomration** at 0x00401AE0
      - iterate over the adapter IPs and get the IP of the local machine
      - get the local time using GetLocalTime function
    - format the beaconing information in a unique string using sprintf. Encode the string with base64 using CryptBinaryToStringA
    - create the socket
    - resolve the host name tweetuplug.dis.uniroma1.it using a DNS query (type A) and connect to it
    - send the encoded (base64) beaconing data (local time and adapter IP) to receive some commands:
      - command Q: exit
      - command S: sleep for 3600 seconds
      - command G: perform a directory listing of the desktop directory and send it to tweetplug.dis.uniroma1.it (C2 server) on port 80

- 2020_2
  - sub_4015C0: main
    - sub_401180: check processes
      - sub_401000: check process name
        - sub_401210: find substring
    - sub_401120: exitprocess
      - sub_401860: copy image in pictures
        - sub_4017E0: create file image
          - sub_401750: get resource
    - sub_4016A0: copy file sample in multiple folders
      - sub_401650: get folder
      - sub_401610: copy file
    - sub_401300: make injection
      - sub_401230: create process
      - sub_4012B0: copy dll in file

- sub_4017E0: create file
  - sub_401750: get resource
- sub_401860: same as before in sub_401120

After it, there is the execution of the dll injected in a thread created by itself:

- sub_10001280: startAddress of the thread
  - sub_10001210: startup connection
  - sub_10001560: make a query to the registry

2020_3

The function main is the subroutine_401AC0 and does the following operations:

- sub_402450 -> mutex; does decryption by doing a loop in which there are some operations done bit a bit, for example by doing "add eax 17h". Then load dynamically and calls the function CreateMutexA to create a mutex object with name "blasy". Then through GetLastError check if there is an error when trying to open th emutex object and in particular if we have the error ERROR_ALREADY_EXISTS, that means that the mutex is already in use by an instance of the malware. If this is true, it writes "ExitProcess" on the stack, loads this function dynamically and executes it, so the malware exits. Otherwise, continues the execution.

- sub_4013F0 -> loads dynamically and calls EnumProcesses to list all the processes; if it fails => push 1 in eax. Otherwise, does a loop in which calls:

  - sub_401240 -> dynamically loads and calls function OpenProcess (if fails, put 0 in eax and comes back to the caller function), EnumProcessModules and GetModuleBaseName to retrieve the names of the modules of each process of the list given by EnumProcessess. So it check if the the modules names contain a substring of one of these processes: "basil.exe", "bergamo.exe", "lavender.exe":
    - if yes => put 1 in eax
    - otherwise => put 0 in eax

  Then if sub_401240 returns 1 => put 1 in eax

- sub_401380 -> exitProcedure; this is called only if in eax there is a value different from 0 (in particular if the precedent routine has put 1 in eax); this means that it is called only in case there is a process module with name "basil.exe", "bergamo.exe" or "lavender.exe":




This subroutine shows this message to the user:




Then calls:

- sub_4023E0 -> take the folder "C:\Users\student\Pictures" through SHGetFolderPath and calls:
  - sub_401C80 to concatenate it with the string "\trio_of_danger.jpg"
  - sub_402360 -> calls:
    - sub_4022D0 -> finds resource with name 113, loads it, does a LockResource to take its size, then allocates memory and write the resource inside this new memory; then frees the resource

  Then with CreateFile and WriteFile, creates file image "C:\Users\student\Pictures\trio_of_danger.jpg" with the resource 113 extracted from the sample.

Then it calls the function __acrt_iob_func and SystemParametersInfoA, passing as parameter the image created before ("C:\Users\student\Pictures\trio_of_danger.jpg")

and the action 14h, i.e. SPI_SETDESKWALLPAPER; so the image is used as Desktop background. Finally, it exits.

- sub_401BA0 -> this is called if the sub_4013F0 returns 0, so the preconditions are satisfied and the malware can continue its execution. It calls:
  - sub_ -> calls:
    - sub_4019A0 -> calls VerSetConditionMask, that sets the bits of a 64-bit value to indicate the comparison operator to use for a specified operating system version attribute; it's used to build the *dwlConditionMask* parameter of the VerifyVersionInfo function, called immediately after. So VerifyVersionInfo checks if the currently running operating system satisfies the specified requirements, i.e. the return value must be a nonzero value:
      - if the return value is 0 => put 0 in eax
      - otherwise (requirements satisfied) => put 1 in eax

  Then calls many other subroutines to copy the sample itself inside "\sniper.exe"

- sub_401A80 ->
- sub_4018D0 -> shellcodeInjection; calls:
  - sub_401630 -> does a loop bit a bit to decrypt by doing a XOR with key 2Bh, so it encrypt "kernell32.dll"; then does a second loop bit a bit to decrypt another string written on the stack, by doing a XOR again with key 2Bh.
    So before this second encryption we have:

```
0019FEA0  E0 D4 66 00 B4 FE 19 00  7D 42 59 5F 5E 4A 47 6A  àÔf.´þ..}BY_^JGj
0019FEB0  47 47 44 48 6E 53 2B 00  AB F0 7F 77 D4 FE 19 00  GGDHnS+.«ð.wÔþ..
0019FEC0  20 6F 66 00 B0 F0 5A 77  6B 65 72 6E 65 6C 33 32  ·of.°ðZwkernel32
0019FED0  2E 64 6C 6C 00 00 00 00  00 00 3E 74 59 CC A1 77  .dll......>tYÌ¡w
```

    and after:

```
0019FE90  A4 FE 19 00 57 72 69 74  65 50 72 6F 63 65 73 73  ¤þ..WriteProcess
0019FEA0  4D 65 6D 6F 72 79 00 00  56 69 72 74 75 61 6C 41  Memory..VirtualA
0019FEB0  6C 6C 6F 63 45 78 00 00  AB F0 7F 77 D4 FE 19 00  llocEx..«ð.wÔþ..
0019FEC0  20 6F 66 00 B0 F0 5A 77  6B 65 72 6E 65 6C 33 32  ·of.°ðZwkernel32
0019FED0  2E 64 6C 6C 00 00 00 00  00 00 3E 74 59 CC A1 77  .dll......>tYÌ¡w
```

    Then we have again other 2 loops in which it does a decryption again with key 2Bh to decrypt other 2 functions, so in the end we obtain we obtain:

```
0019FE80  43 72 65 61 74 65 52 65  6D 6F 74 65 54 68 72 65  CreateRemoteThre
0019FE90  61 64 00 00 57 72 69 74  65 50 72 6F 63 65 73 73  ad..WriteProcess
0019FEA0  4D 65 6D 6F 72 79 00 00  56 69 72 74 75 61 6C 41  Memory..VirtualA
0019FEB0  6C 6C 6F 63 45 78 00 00  4C 6F 61 64 4C 69 62 72  llocEx..LoadLibr
0019FEC0  61 72 79 41 00 F0 5A 77  6B 65 72 6E 65 6C 33 32  aryA.ðZwkernel32
0019FED0  2E 64 6C 6C 00 00 00 00  00 00 3E 74 0D 00 00 00  .dll......>t....
```

    - sub_4015C0 ->
    - sub_401530 -> dependent on the ckeck of the system information (done again by sub_4019A0, it creates a different process; in this case creates "C:\Windows\SysWow64\notepad.exe".

Then it calls VirtualAllocEx to allocate memory in the created process, then calls WriteProcessMemory to write the malicious payload in the process and then calls CreateRemoteThread to start a new thread that will execute this payload.

- sub_401000 -> connection; calls InternetOpenW that initializes the use of the WinINet functions, then calls InternetConnectW to do an HTTP connection to server "hastebin.com", at port 443; then calls:
  - sub_402160 -> calls:
    - sub_401EE0 -> calls GetLocalTime and constructs the message "Time: %02d:%02d\n"
    - sub_401D10 -> calls GetAdaptersAddresses and constructs the string "IP: %s\n"
    - sub_401F20 -> calls GetComputerName and constructs the string "Computer name: %s\n"
    - sub_401F80 -> calls SHGetKnownFolderPath to retrieve the directory listing of the host machine

  So in the end we have:

```
00406170  54 69 6D 65 3A 20 31 37  3A 33 39 0A 49 50 3A 20  Time:·17:39.IP:·
00406180  31 30 2E 30 2E 32 2E 31  35 0A 43 6F 6D 70 75 74  10.0.2.15.Comput
00406190  65 72 20 6E 61 6D 65 3A  20 44 45 53 4B 54 4F 50  er·name:·DESKTOP
004061A0  2D 47 32 44 55 4A 41 54  0A 0A 48 6F 6D 65 20 64  -G2DUJAT..Home·d
00406180  69 72 65 63 74 6F 72 79  20 63 6F 6E 74 65 6E 74  irectory·content
004061C0  3A 0A 3C 64 3E 43 3A 5C  55 73 65 72 73 5C 73 74  :.<d>C:\Users\st
004061D0  75 64 65 6E 74 5C 2E 65  6E 76 69 0A 3C 64 3E 43  udent\.envi.<d>C
004061E0  3A 5C 55 73 65 72 73 5C  73 74 75 64 65 6E 74 5C  :\Users\student\
004061F0  2E 67 68 69 64 72 61 0A  3C 64 3E 43 3A 5C 55 73  .ghidra.<d>C:\Us
00406200  65 72 73 5C 73 74 75 64  65 6E 74 5C 2E 76 73 63  ers\student\.vsc
00406210  6F 64 65 0A 3C 64 3E 43  3A 5C 55 73 65 72 73 5C  ode.<d>C:\Users\
00406220  73 74 75 64 65 6E 74 5C  33 44 20 4F 62 6A 65 63  student\3D·Objec
00406230  74 73 0A 3C 64 3E 43 3A  5C 55 73 65 72 73 5C 73  ts.<d>C:\Users\s
00406240  74 75 64 65 6E 74 5C 41  70 70 44 61 74 61 0A 3C  tudent\AppData.<
00406250  64 3E 43 3A 5C 55 73 65  72 73 5C 73 74 75 64 65  d>C:\Users\stude
00406260  6E 74 5C 41 70 70 6C 69  63 61 74 69 6F 6E 20 44  nt\Application·D
```

After retrieving the information from the host, it constructs the message to send through HttpSendRequestW:
POST /documents
At: %d\n%s\n, followed by all the information retrieved before.

- *technical*
- **7** - Does the sample make queries about the surrounding environment before unveiling its activities? If yes, describe them and pinpoint specific instructions/functions in the code.
  - **Vorlesung**
    - 

  - **Altklausuren**
    - 2019_1
      - The malware checks (in "checkValidFile" routine at 0x4012D0) for the presence of a file named *n0r44.txt* in the system root directory: if it's present and with the Archive attribute, or it's a directory, the malware continues, otherwise it exits immediately.
    - 2019_2
      - The malware won't run if the checks done at the beginning on ecx and edx are not done. In particular if neither one of the following condition is met:
        - ecx >= 17 (decimal)
        - or edx <= 12 (decimal)

        It won't start. Note that both of them are set to be equal to 16 (decimal)
    - 2020_1
      - Check of Malware schon läuft

- the malware in the subroutine checkProcesses at 0x00401BB0 iterate over the processes and checks for the presence of the process the execute the sample. If the process exist or the call to K32EnumProcesses fail set eax to 1, else set eax to 0 (as a flag). If eax is set to 1 call the subroutine exitProcess at 0x00401680 that load dynamically the function ExitProcess (after decoding two base64 strings containing kernel32.dll and ExitPorcess) and call it to exit.

- 2020_2
  - Before that the real content of the malware appears, the sample checks for a condition in the subroutine 0xsub_401180. Here the malware call K32EnumProcesses, in order to retrieve a list of all the processes in the system. Here the sample iterates over all the processes calling the subroutine sub_401000. In the latter the malware checks the name of the process through K32EnumProcessModules and K32GetModuleBaseNameA, if it contains a substring of one of the processes that we found in plaintext it will return 1.
  - If the return value of the function it's 0, it means that there is one of the program, so the program exits in the main function, in particular in subroutine sub_401120. Before exit, the malware will take the resource 112, copying it in the Pictures folder and it will call SystemParametersInfoA with parameter 14h, in this way it changes the desktop image.

    So in order to allow the sample to continues we need to avoid to run those programs.
  - das sind alles Antivirus Programme
- 2020_3
  - Yes, in sub_402450 it creates a mutex, so checks that there is only one instance of the malware running, otherwise it exits.
    Then, in sub_4013F0 it checks if there is a process module with name "basil.exe", "bergamo.exe" or "lavender.exe":
    - if yes => put 1 in eax
    - otherwise => put 0 in eax
  - So in case in eax there is 1 => it calls sub_401380 that does an exit procedure:


- **8** - Does the sample include any persistence mechanisms? If yes, describe its details and reference specific instructions/functions in the code.
  - **Altklausuren**
    - 2019_1
      - The persistence mechanism is located in subroutine *setRegistryRunOnce* (0x4013A0) which creates a new key at *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce* with the name "*wjsvtbhhjp*" and value "*rundll32 c:\Users\student\Documents\konrAd.dll,exam*", effectively running the DLL export "exam" once at the next login.
    - 2019_2
      - Yes, the sample includes persistence mechanisms by copying itself into the Startup folder under the name sullivan.exe. The subroutine in which it is done is sub_401400.
    - 2020_1
      - Yes, the sample creates an hidden file in the download folder named C0ldD410.exe that contains the sample code. This operation is in the subroutine createPersistenceFIle at 0x00401740
      - Next, in the subroutine registryRunPersistence at 0x004017E0 the malware creates the value spaceship inside the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run that contains the path to the file C0ldD410.exe (inside the download folder) to execute the sample at each login.
    - 2020_2

- The sample in the subroutine sub_4016A0 firstly copy itself in various folders. Basically it use an array called csidl which contains all the codes to the folders in which it copy itself. The folders are the following:

```
F db    0
0 aCUsersStudentD db 'C:\Users\student\Documents\cell_jr.exe',0
0                                       ; DATA XREF: sub_4016A0+66↑o
7 db    0
8 db    0
```

```
.data:00404363 db    0
.data:00404364 aCUsersStudentA db 'C:\Users\student\AppData\Local\cell_jr.exe',0
.data:0040438F db    0
.data:00404390 db    0
.data:00404391 db    0
```

```
.data:00404467 db    0
.data:00404468 aCUsersStudentF db 'C:\Users\student\Favorites\cell_jr.exe',0
.data:0040448F db    0
.data:00404490 db    0
.data:00404491 db    0
```

```
.data:0040456A db    0
.data:0040456B db    0
.data:0040456C aCUsersStudentP db 'C:\Users\student\Pictures\cell_jr.exe',0
.data:00404592 db    0
.data:00404593 db    0
.data:00404594 db    0
.data:00404595 db    0
```

```
.data:0040466E db    0
.data:0040466F db    0
.data:00404670 aCUsersStudentA_0 db 'C:\Users\student\AppData\Roaming\Microsoft\Windows\Start Menu\Pro'
.data:00404670 db 'grams\Startup\cell_jr.exe',0
.data:004046CB db    0
.data:004046CC db    0
.data:004046CD db    0
```

```
.data:00404773 db    0
.data:00404774 aCUsersStudentA_1 db 'C:\Users\student\AppData\Roaming\Microsoft\Windows\Start Menu\cel'
.data:00404774 db 'l_jr.exe',0
.data:004047BE db    0
.data:004047BF db    0
.data:004047C0 db    0
```

```
.data:00404877 db    0
.data:00404878 aCUsersStudentV db 'C:\Users\student\Videos\cell_jr.exe',0
.data:0040489C db    0
.data:0040489D db    0
.data:0040489E db    0
```

  The one that we are interesting in is the 5-th, in which we find the path to the Startup folder, where every file in this folder should run at startup.

- 2020_3
  - In sub_401BA0, it makes a copy of itself in the file "\sniper.exe" in a place depending on the result of sub_4019A0 -> calls VerSetConditionMask, that sets the bits of a 64-bit value to indicate the comparison operator to use for a specified operating system version attribute; it's used to build the *dwlConditionMask* parameter of the VerifyVersionInfo function, called immediately after. So VerifyVersionInfo checks if the currently running operating system satisfies the specified requirements, i.e. the return value must be a nonzero value:
    - if the return value is 0 => put 0 in eax
    - otherwise (requirements satisfied) => put 1 in eax

    So malware can survive at each reboot.

- **9** - Does the sample perform any code injection activities? Which kind of injection pattern do you recognize? Describe the characteristics and behavior of the injected payload, stating also where it is originally stored within the sample.
  - TODO müssen wir die payload selber dumpen können?
    - 
    - if thread still suspended look here for RWX
  - **Vorlesung**

- Vast universe of techniques
  - process hollowing
  - DLL injection
  - PE injection
  - thread execution hijacking
  - reflective DLL injection
  - hook injection
  - APC injection
  - ...

- **Process Hollowing**
  - Disguises malware as a legitimate process
  - Typically robust (w.r.t. crashes)

    man kann auch process übernehmen, aber einfacher oder besser ist selber zu erstellen
  - Idea: start a process in a suspended state, then replace its code
    - when you load a process as suspended, only an external action can start it
    - before that action occurs, the launcher will:
      - deallocate currently mapped sections of the process

        wir wollen hier dann unseren eigenen code hinpacken
      - allocate memory to accommodate for the malicious payload
      - write the payload contents to the address space of the victim

        wirkt erstmal insecure, ist aber windows feature
      - update the context of the suspended main thread (EIP ⇐ entry point for payload)

        wenn process startet vorher auf den richtigen entry point setzen
    - finally, resume the suspended main thread

  - ```
    00401535        push    edi             ; lpProcessInformation
    00401536        push    ecx             ; lpStartupInfo
    00401537        push    ebx             ; lpCurrentDirectory
    00401538        push    ebx             ; lpEnvironment
    00401539        push    CREATE_SUSPENDED ; dwCreationFlags
    0040153B        push    ebx             ; bInheritHandles
    0040153C        push    ebx             ; lpThreadAttributes
    0040153D        lea     edx, [esp+94h+CommandLine]
    00401541        push    ebx             ; lpProcessAttributes
    00401542        push    edx             ; lpCommandLine
    00401543        push    ebx             ; lpApplicationName
    00401544        mov     [esp+0A0h+StartupInfo.dwFlags], 101h
    0040154F        mov     [esp+0A0h+StartupInfo.wShowWindow], bx
    00401557        call    ds:CreateProcessA
    ```

    *Listing 12-2: Assembly code showing process replacement*

    hollowing sieht immer so aus, aber es gibt auch legitime Operationen die so anfangen...

    es müssen noch weitere sachen passieren, damit verdächtig, bzw. bestätigt process hollowing

    Creation flag `CREATE_SUSPENDED` (0x00000004) for `CreateProcess`
    ```

- 

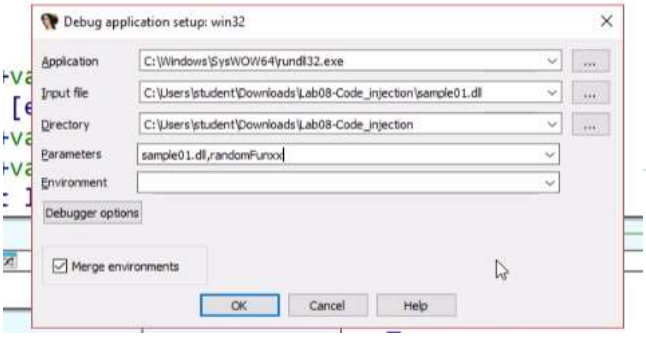

- **DLL Injection**
  - VirtualAllocEx
  - WriteProcessMemory
  - (GetProcAddress)
  - CreateRemoteThread
  - 

  - 


- **PE Injection** - Shellcode?

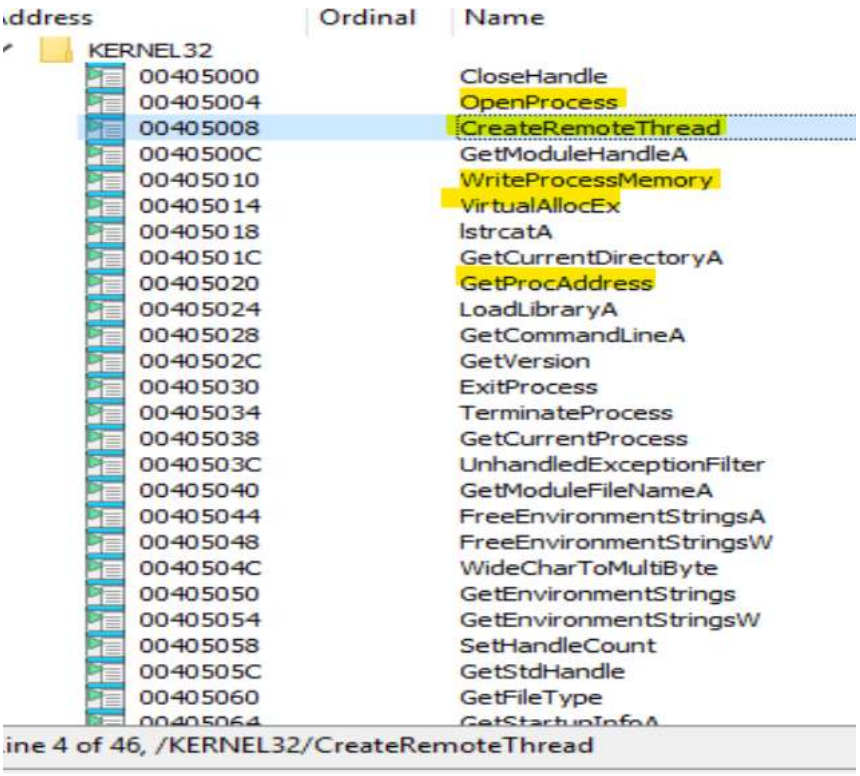
- 


- **Labs**
  - beim debuggen der injected payload
    - [[BlobRunner]] bei shellcode
    - [[rundll32]] bei dll
      - C:WindowsSysWOW64rundll32.exe
      - esp

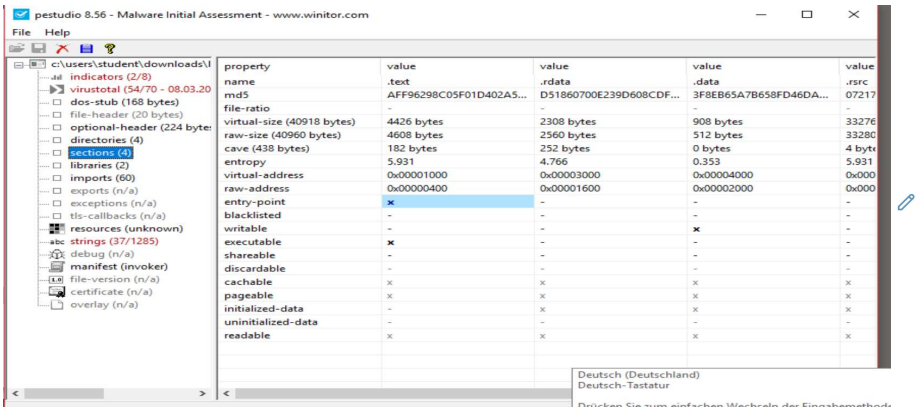



irgendeinen export nehmen, den es nicht gibt (hier random)

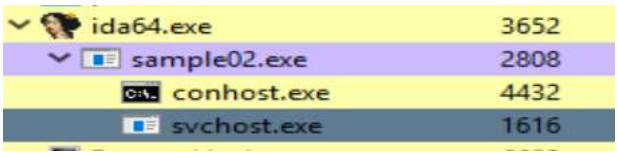
  - erstmal verdächtige imports schauen
    - in der Klausur sind aber evtl. viel mehr um einen in die Irre zu leiten




- .rscr hat normalerweise nur manifest, manchmal aber sehr viel




- → [[Ressource Hacker]] → save as bin → man hat schon dll die injected werden soll
- **Debugger** in IDA zusammen mit [[Process Hacker]] oder [[Process Explorer]] laufen lassen z.B. bei **Hollowing**, weil es gut Threads im suspended state anzeigt




- **Altklausuren**
  - 2019_1

> The malware injects a DLL in *notepad.exe*. The *notepad.exe* process is started (in "*createProcessToInject*" at 0x401490), then the path of the DLL is copied in the remote process memory (after a remote allocation) and then a remote thread is injected in "*dllInjection*" at 0x401510 with the address of "*LoadLibrary*" routine.
>
> This has the effect of starting a thread in the remote process that loads a DLL. As DLLs contain a "main" method named "DllMain", this triggers the other part of the malware which is described following answers (and in answer 7, second part).

  - 2019_2

> Yes, the sample does shellcode injection on the victim process svchost. It can be understood by the fact that the sample uses VirtualAlloc to allocate memory on the victim process and uses WriteProcessMemory to write the payload into the process. At the end, it uses CreateRemoteThread to execute what has been injected.
> It can be possible to use blobrunner and x32dbg to debug the shellcode or simply to execute it.
> By running the shellcode using blobrunner, and using procexp to see what it is happening, it is possible to see that there is an svchost process

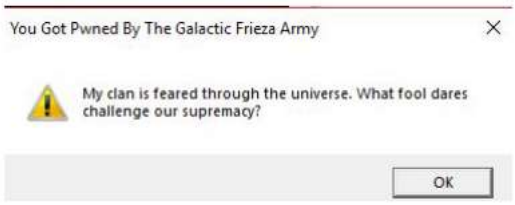
- ...

- 2020_1
  - Yes, the sample does a shellcode injection on the victim process cmd.exe created using the process that execute the sample. It can be understood by the fact that the sample use VirtualAllocEx to allocate memory on the process cmd.exe and use WriteProcessMemory to write the payload into the process. These operations are in the subroutine **shellcodeInjection** at 0x00401520. The shellcode starts at offset 0x00403208.

    At the end, it uses CreateRemoteThread to execute the injected code.

    It can be possible to use blobrunner to debug the shellcode or execute it.

    By running the shellcode (using blobrunner) it's possible to see the the shellcode open a popup:

    The WriteProcessMemory in the subroutine at 0x00401520:

    ```
    push    offset unk_403208
    mov     ecx, [ebp+var_C]
    push    ecx
    mov     edx, [ebp+var_8]
    push    edx
    mov     eax, 4
    imul    ecx, eax, 6
    mov     edx, [ebp+ecx+var_30]
    call    edx ; WriteProcessMemory
    ```

- 2020_2
  - Yes, the sample performs a Dll injection to the process notepad.exe. Basically the injection is composed of two following phases.

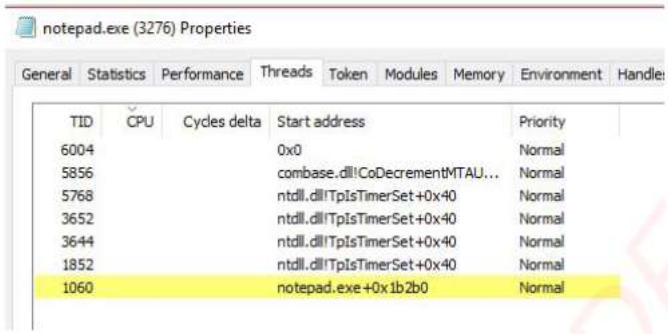
    Initially the sample will create an instance of the notepad.exe executable in the subroutine sub_401230. Then it extracts the malicious code from the resource 113 in the subroutine sub_401750, copying the content of the resource in the file C:\Users\student\mirai.dll in subroutine sub_4012B0.

    The real injection begins after the last passages. Here the malware call GetProcAddress multiple times in order to get all the functions needed for the injection. Then the sample call VirtualAllocEx allocating memory in the victim process and writes (calling WriteProcessMemory) the path to mirai.dll created before. Finally, the sample will call CreateRemoteThread, passing the LoadLibrary function and the memory containing the path to mirai.dll, so that it will load the malicious dll.

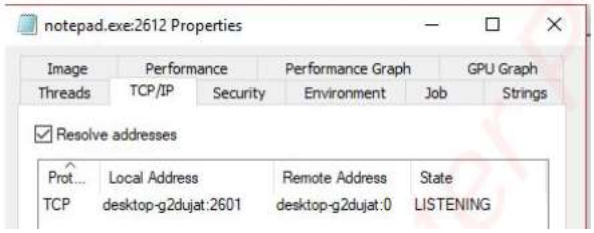
- 2020_3
  - Yes, it does a shellcode injection in sub_4018D0; it calls sub_401530 -> dependent on the check of the system information (done again by sub_4019A0), it creates a different process; in this case creates "C:\Windows\SysWow64\notepad.exe".

    Then it calls VirtualAllocEx to allocate memory in the created process, then calls WriteProcessMemory to write the malicious payload in the process and then calls CreateRemoteThread to start a new thread that will execute this payload.

    To see the shellcode behaviour, we can check the behaviour of the process notepad.exe. Through Process Hacker we can see that it has many threads:



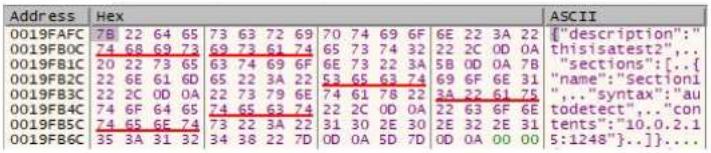

Through Process Explorer, we can see if it has opened TCP/IP connections:




- **10** - Does the sample beacon an external C2? Which kind of beaconing does the malware use? Which information is sent with the beacon? Does the sample implement any communication protocol with the C2? If so, describe the functionalities implemented by the protocol.
  - **Labs**
    - [[Lab07 - 15.11.]]
      - ## How is the beacon crafted?
        - gute Idee ist, falls schwer zu finden wo über den import address table in [[IDA]] gehen
  - **Altklausuren**
    - 2019_1
      - The C2 used in the malware (in the DLL sample) is contacted using both DNS requests for beaconing, where the *Computer name* and *System time* info are encoded (routines 0x10001540 for getting info, 0x10001000 for encoding) and used as subdomains of *kettle667.biz* and by connecting using HTTP at 192.168.50.26 to get instructions.

```
10.0.2.15          192.168.50.26      HTTP      126 GET / HTTP/1.1
192.168.50.26      10.0.2.15          TCP        60 80 → 51479 [ACK] Seq=1 Ack=73
192.168.50.26      10.0.2.15          TCP        71 80 → 51479 [PSH, ACK] Seq=1 A
192.168.50.26      10.0.2.15          TCP       156 80 → 51479 [PSH, ACK] Seq=18
192.168.50.26      10.0.2.15          HTTP       60 HTTP/1.0 200 OK  (text/html)
```

        The encoded DNS request appears to be encoded in a Base64-like algorithm.
      - The protocol is based on an HTTP request type GET, where the sample reads the body of the HTTP response and uses it as command.

        The functionalities are:
        - Remove a file in home folder, named "*very_important_doc.rtf*" (command 44h)
        - Shows a popup (command 4Ah)
        - Sleep for 300 seconds (command 53h)
    - 2019_2
      - The beaconing is done at the server api.paste.ee. The content of what it sends contains the ip address of the local machine and the port 1248 from which the machine will connect to the C2. The destination port is 443. The IP address of the server is 104.18.49.20. The sent data is the following:
      - 

      - After the malwate writes the data described above, it waits until the response of the server using the call WinHttpReceiveResponseData. After the malware receives the data (which is not something particular), it pops up with a message box. The same thing is done even if there is no data to be read but the connection needs to be established.
    - 2020_1

The sample perform a connection (using a socket) to the C2 server at tweetplug.dis.uniroma1.it (ip: 151.100.101.140, port 80) sending the beaconing informations (see answer 10) in a base64 encoded way to receive some instruction. It receive a response from the C2 server with a possible command:

- command Q: exit
- command S: sleep for 3600 seconds
- command G: perform a directory listing of the desktop directory and send it to the C2 server

2020_2

Yes the sample in the dll inject call the function GetCurrentProcessId.

It make a DNS request, as we can see from fakenet

```
    Diverter]   pid:  6736 name: chrome.exe
    Diverter]   pid:  2384 name: rundll32.exe
DNS Server] Received A request for domain ''.
DNS Server] Responding with '192.0.2.123'
    Diverter]   pid:  7996 name: Code.exe
    Diverter]   pid:  7996 name: Code.exe
```

The it send some data calling the function sendto. It will send data to 8.8.8.8

```
call    ds:htons
mov     word ptr [ebp+to.sa_data], ax
push    offset cp       ; "8.8.8.8"
call    ds:inet_addr
mov     dword ptr [ebp+to.sa_data+2], eax
lea     edx, [ebp+buf]
mov     [ebp+Dst], edx
push    1               ; hostshort
call    ds:htons
```

2020_3

In sub_401000 it does a connection with an external command&controller; it calls InternetOpenW that initializes the use of the WinINet functions, then calls InternetConnectW to do an HTTP connection to server “hastebin.com”, at port 443; then calls:

- sub_402160 -> calls:
  - sub_401EE0 -> calls GetLocalTime and constructs the message “Time: %02d:%02d\n”
  - sub_401D10 -> calls GetAdaptersAddresses and constructs the string "IP: %s\n"
  - sub_401F20 -> calls GetComputerName and constructs the string “Computer name: %s\n"
  - sub_401F80 -> calls SHGetKnownFolderPath to retrieve the directory listing of the host machine

So in the end we have:

```
00406170  54 69 6D 65 3A 20 31 37  3A 33 39 0A 49 50 3A 20  Time:·17:39.IP:·
00406180  31 30 2E 30 2E 32 2E 31  35 0A 43 6F 6D 70 75 74  10.0.2.15.Comput
00406190  65 72 20 6E 61 6D 65 3A  20 44 45 53 4B 54 4F 50  er·name:·DESKTOP
004061A0  2D 47 32 44 55 4A 41 54  0A 0A 48 6F 6D 65 20 64  -G2DUJAT..Home·d
004061B0  69 72 65 63 74 6F 72 79  20 63 6F 6E 74 65 6E 74  irectory·content
004061C0  3A 0A 3C 64 3E 43 3A 5C  55 73 65 72 73 5C 73 74  :.<d>C:\Users\st
004061D0  75 64 65 6E 74 5C 2E 65  6E 76 69 0A 3C 64 3E 43  udent\.envi.<d>C
004061E0  3A 5C 55 73 65 72 73 5C  73 74 75 64 65 6E 74 5C  :\Users\student\
004061F0  2E 67 68 69 64 72 61 0A  3C 64 3E 43 3A 5C 55 73  .ghidra.<d>C:\Us
00406200  65 72 73 5C 73 74 75 64  65 6E 74 5C 2E 76 73 63  ers\student\.vsc
00406210  6F 64 65 0A 3C 64 3E 43  3A 5C 55 73 65 72 73 5C  ode.<d>C:\Users\
00406220  73 74 75 64 65 6E 74 5C  33 44 20 4F 62 6A 65 63  student\3D·Objec
00406230  74 73 0A 3C 64 3E 43 3A  5C 55 73 65 72 73 5C 73  ts.<d>C:\Users\s
00406240  74 75 64 65 6E 74 5C 41  70 70 44 61 74 61 0A 3C  tudent\AppData.<
00406250  64 3E 43 3A 5C 55 73 65  72 73 5C 73 74 75 64 65  d>C:\Users\stude
00406260  6E 74 5C 41 70 70 6C 69  63 61 74 69 6F 6E 20 44  nt\Application·D
```

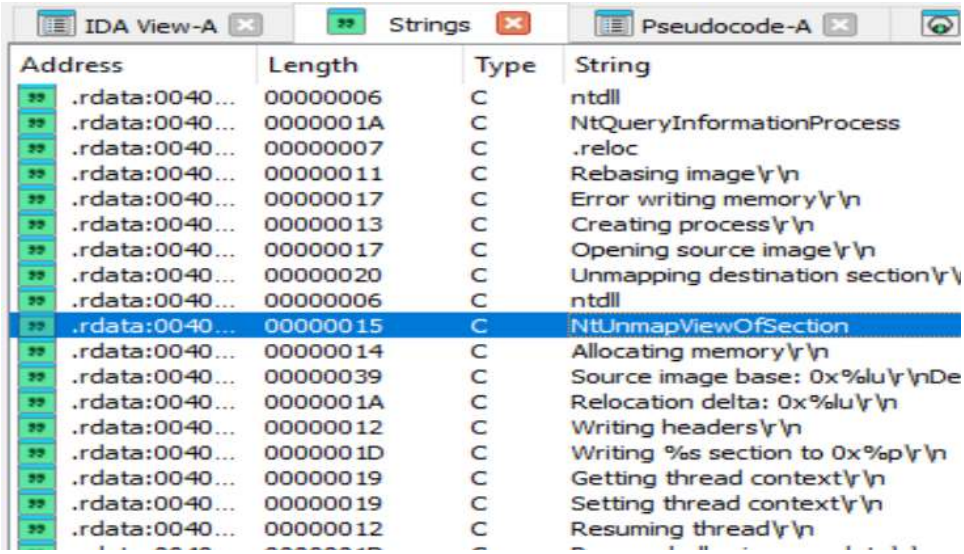
After retrieving the information from the host, it constructs the message to send to the server through HttpSendRequestW:
POST /documents
At: %d\n%s\n, followed by all the information retrieved before.
First it does a DNS query to retrieve the IP address (answer 4).

**11** - List the obfuscation actions (if any) performed by the sample to hide its activities from a plain static analysis. Pinpoint and describe specific code snippets.

**Labs**

[[Lab07 - 15.11.]]

mit [[CyberChef]] habe ich das hier gefunden




- für process hollowing wird manuell importiert, damit versteckter

- **Altklausuren**
  - 2019_1
    - The injected DLL is encrypted with XOR using a two bytes key 0xAB1C (the DLL is decrypted before the write part "getDllFromResources" at 0x401000)
    - The DLL itself is in a resource (the resource is extracted in "getDllFromResources" at 0x401000)
    - The "rundll32" part of the RunOnce registry key for persistence is moved char-by-char in memory, and some "mov" are after a number of instructions (in "setRegistryRunOnce" at 0x4013A0) see image in answer 9
    - The IP address for C2 is obfuscated by a number of overlapping "mov"s (in DLL, in "httpCall" subroutine at 0x100012A0)
  - 2019_2
    - The way in which sullivan.exe is written into the stack. indeed, Ida is not able to understand that it is an array.:

```
mov     [ebp+var_8], eax
mov     [ebp+Source], '\'
mov     [ebp+var_17], 's'
mov     [ebp+var_16], 'u'
mov     [ebp+var_15], 'l'
mov     [ebp+var_14], 'l'
mov     [ebp+var_13], 'i'
mov     [ebp+var_12], 'v'
mov     [ebp+var_11], 'a'
mov     [ebp+var_10], 'n'
mov     [ebp+var_F], '.'
mov     [ebp+var_E], 'e'
mov     [ebp+var_D], 'x'
mov     [ebp+var_C], 'e'
mov     [ebp+var_B], 0
lea     edx, [ebp+Source]
```

    - The injected payload;
    - The way in which the name svchost is built. In particular at the beginning it was scozhost and then, suing a memcpy, the first letters are replaced with svch.

```
call    ds:MessageBoxA
mov     [ebp+Dst], 's'
mov     [ebp+var_8], 'c'
mov     [ebp+var_A], 'o'
mov     [ebp+var_9], 'z'
mov     [ebp+var_8], 'o'
mov     [ebp+var_7], 's'
mov     [ebp+var_6], 't'
mov     [ebp+var_5], 0
push    4               ; Size
push    offset aSvch    ; "svch"
lea     ecx, [ebp+Dst]
push    ecx             ; Dst
call    memcpy
add     esp, 0Ch
```

- When it imports at runtime a bunch of function calls. It is also important that only few of them are effectively used:

```
push    offset aVirtualallocex ; "VirtualAllocEx"
push    offset aKernel32Dll_2 ; "kernel32.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+virtual_alloc], eax
push    offset aNtunmapviewofs ; "NtUnmapViewOfSection'
push    offset aNtdllDll ; "ntdll.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_7C], eax
push    offset aSetthreadconte ; "SetThreadContext"
push    offset aKernel32Dll_3 ; "kernel32.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_5C], eax
push    offset aResumethread ; "ResumeThread"
push    offset aKernel32Dll_4 ; "kernel32.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_78], eax
push    offset aWriteprocessme ; "WriteProcessMemory"
push    offset aKernel32Dll_5 ; "kernel32.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+WriteProcessMemory], eax
```

2020_1

- the function ExitProcess is loaded dynamically using GetModuleHandle and GetProcAddress from two base64 strings (the contains kernel32.dll and ExitProcess)

```
mov     [ebp+pszString], offset aA2vybmvsmziuzg ; "a2VybmVsMzIuZGxs"
mov     [ebp+var_C], offset aRxhpdfbyb2nlc3 ; "RXhpdFByb2Nlc3M="
mov     [ebp+pcbBinary], 20h
push    0 ; pdwFlags
push    0 ; pdwSkip
lea     eax, [ebp+pcbBinary]
push    eax ; pcbBinary
lea     ecx, [ebp+pbBinary]
push    ecx ; pbBinary
push    1 ; dwFlags
push    0 ; cchString
mov     edx, [ebp+pszString]
push    edx ; pszString
call    ds:CryptStringToBinaryA
mov     eax, [ebp+pcbBinary]
mov     [ebp+eax+pbBinary], 0
mov     [ebp+pcbBinary], 20h
push    0 ; pdwFlags
push    0 ; pdwSkip
lea     ecx, [ebp+pcbBinary]
push    ecx ; pcbBinary
lea     edx, [ebp+var_30]
push    edx ; pbBinary
push    1 ; dwFlags
push    0 ; cchString
mov     eax, [ebp+var_C]
push    eax ; pszString
call    ds:CryptStringToBinaryA
```

- the malware loads dynamically a lot of function in the shellcodeInjection subroutine at save it into a memory portion using a index but not use all the function (obfuscation technique!). Only use VirtualAllocEx, WriteProcessMemory and CreateRemoteThread.

2020_2

- The creation of multiple copy of the file exe (answer 8)
- The sample load multiple functions using GetProcAddress and put it in an array, so it difficult to understand which function is called

2020_3

- There are many obfuscations, in particular we find some "suspicious" strings that are written in the stack and so aren't identified as real strings by a static analysis; they are also encrypted, so for exsample in sub_401630, there are 4 loops to decrypt these strings with XOR with key 2Bh; so in the end we obtain we obtain:

```
0019FE80  43 72 65 61 74 65 52 65  6D 6F 74 65 54 68 72 65  CreateRemoteThre
0019FE90  61 64 00 00 57 72 69 74  65 50 72 6F 63 65 73 73  ad..WriteProcess
0019FEA0  4D 65 6D 6F 72 79 00 00  56 69 72 74 75 61 6C 41  Memory..VirtualA
0019FEB0  6C 6C 6F 63 45 78 00 00  4C 6F 61 64 4C 69 62 72  llocEx..LoadLibr
0019FEC0  61 72 79 41 00 F0 5A 77  6B 65 72 6E 65 6C 33 32  aryA.ðZwkernel32
0019FED0  2E 64 6C 6C 00 00 00 00  00 00 3E 74 0D 00 00 00  .dll......>t....
```

  Then there are a lot of strings computed dynamically through memcpy and completed with sprintf.

  Then there are many calls that point to an address in the stack, instead of calling the functions with their name. So they are first loaded in the stack through the functions "GetModuleHandleA" and "GetProcAddress" and then we have a call to the address in which they have been stored. In this way IDA doesn't recognize the "push" that are precedent to the call as parameters of the function and the static analysis becomes less intutive. As an example:

```
call    [ebp+var_10] ; calls CreateMutexA
```

- The function ExitProcess is loaded dynamically using GetModuleHandle and GetProcAddress from two base64 strings (that contain kernel32.dll and ExitProcess)
  The shellcode is written in memory
  The beaconing information are sent to the C2 server in a base64 encoded way
  In sub_401300, the name LoadLibrary is encoded using a add 30h in the variable LoadLibrary; also the name of the Kernel32.dll is encoded.