Hancitor

Alföldi Mátyás

June 12, 2020

Contents

Introduction

File: sv.exe
VT: https://www.virustotal.com/gui/file/995cbbb422634d497d65e12454cd5832cf1b4422189d9ec06efa88ed56891cda
All things point to this being a Hancitor sample.

Static Analysis

0.1 Unpacking/IAT building

At the start one can already see something very interesting, which is the following:

- It sets ecx to 33A418CE
- It then calls lstrcatA with both 0 parameters, which will cause it to fail.
- It then stores ecx in a variable and checks if it is equal to 33A418CE, if it is we exit.

The reason why it does it is, because ecx will be modified in a subcall inside lstrcatA.
If one goes into this subcall we can see that the value stored at 74E10140 is has to do with it. (This value gets set during the start of the application during the loading of it. Also the value stored here will be xor-ed with the calling process-es stack frame. (TODO:what is this value?))
Maybe it is some check if the machine is already infected, or it is just a stack cookie(?) The next part of the code is the loop that calculates function pointers, the following code outputs what offsets are called:

```
int start = 0x48;
int i = 0x1c;
int funcptr = 0;
int some_counter = 0xffe4d8c3;
while (i < 0x225561e)
{
    ++start; // = start - subtr;  - -1
    if (i == 0x2dcab2)//(0x2dcab1 < i && i < 0x2dcab3)
        funcptr = some_counter - 0x2255615;
    if (i % 1000000 == 0)
    {
        ++start;
        some_counter += i;
    }
    if (0x2255614 < i)
        std::cout << std::hex << funcptr + i << std::endl;
    ++i;
}
```

This gives us the following:

- 406643-40664b

Lets see the first offset:
We arrive at a jmp that goes to 4067B9, interestingly x32dbg shows the correct instruction only when going to the exact address.
In Ghidra one needs to disassemble this part aswell otherwise it seems like it thinks of it as data.
Following this is an interesting pattern, which is the following:
One instruction and afterwards a jmp/jcc to the next location.
What comes out of this is a VirtualProtect call, by calculating the address of it into eax.
This VirtualProtect call extends from the start till the .rsrc section,
giving it PAGE_EXECUTE_READWRITE.
A creative way to hide things from AVs.Although they are not very good at hiding what function gets called, since all they do is set a value into eax, and just add another value to it, which will result in the address of the function ptr. They did the same with the values that get passed to the function.
The next part seems to be a xor decryption with the key 07(B5701A07 is stored in eax but only al is used in the xor) starting from 402EA6. It seems the next 0xC7F bytes are decoded this way.
Afterwards the starting address of this area (402EA6) gets popped into eax and we jump there(the decoded area).
In here the first few interesting things that are sticking out like a sore thumb are the sub esp,208 and the pushad/popad combination in between which a PEB referencing can be seen.
What basically happens is that it creates a hash for all the loaded module names, and compares it with a certain value, which is the value for kernel32.dll, when it is found we exit the loop.
We then store what seems to be the base address of the loaded kernel32.dll, which is strange, since we add 0x10 to the InMemoryOrderModuleList which should be InInitializationOrderLinks in the LDR_DATA_TABLE_ENTRY instead of the DLLBase which is located at 0x18 based on this:
https://www.geoffchappell.com/studies/windows/win32/ntdll/structs/ldr_data_table_entry.htm.
It then gets the export directory RVA from the PE header of the kernel32.dll,
and stores the following:

- ImageBase
- Export Directory RVA
- EDir.Base
- ImageBase+EDir.AddressOfNames
- ImageBase+EDir.AddressOfFunctions
- ImageBase+EDir.AddressOfNameOrdinals

Now comes the IAT rebuilding, where another interesting pattern can be seen, which is the following:

- eax is pushed into the stack
- a call happens to the next instruction, of course this pushes the address of the next instruction to the stack
- pop eax and jump to another location
- Finally we add 3 to eax, which when interpreted as data is the name of a function.

The first time GetProcAddress is the function name hidden this way, and this will be resolved with the help of the gathered info from kernel32.dll-s Export Directory. The rest will be simply resolved through GetProcAddress calls, and also some dll-s are loaded. Once this is done the PE header gets modified and we jump to a location which seems like the unpacked PE file.

0.2 Information gathering/Communication

After some heap allocation/xor-ing GetAdaptersAddresses is called.
Gets the windows directory. Calls GetVolumeInformationA, GetVersion, GetComputerNameA, EnumProcesses. (For data exfiltration.) It then tries to open the enumerated processes, and compares it with explorer.exe. (Possibly hopes that explorer.exe is the one which it could open). It will go on and find the process id of explorer.exe.
It then calls OpenProcess,OpenProcessToken, and GetTokenInformation on it. Once this is done it allocates a heap, and calls LookupAccountSidA for account/domain retreival.
After a little this and that, it calls InternetCrackUrlA on http://api.ipify.org If this succeeds, it calls InternetOpenA, InternetConnectA, HttpOpenRequestA, InternetCloseHandle, InternetQueryOptionA.
It basically gets the external IP. Another connection will be made afterwards.
In the end a format string in the form of
GUID=...&BUILD=...&INFO=...&IP=...&TYPE=...&WIN=...
gets created with the gathered info.
It then calls GetModuleHandleA on kernel32.dll and passes the return value to GetProcAddress, which returns GetNativeSystemInfo.
The following info gets placed afterwards onto a newly allocated heap:
2901_67231(few 0 bytes)
http://twereptale.com/4/forum.php—http://charovalso.ru/4/forum.php—http://verectert.ru/4/forum.php
Through parsing we get the first link into another heap allocated place. It then tries to do the connection to it, as also seen from the .pcap file. And will do the same to the other ones.