QUICK START ON THE TECHNIQUES OF HIDING, EMBEDDING AND PROTECTING IN WINDOWS

INTRODUCTION

It should be remembered that none of the defense or attack techniques in question is absolute.

For each there are countermeasures.

Some techniques are used despite obsolescence, because:

- guarantee the launch on the old OS, of which there will always be many;

- Nothing invented instead.

In parallel with this manual, it is recommended to read the series of articles "Malware development" in the blog https://0xpat.github.io

I. HIDING AND EMBEDDING TECHNIQUES

Windows allows you to manipulate processes within a very wide range,

due to the presence of a set of system calls CreateRemoteThread(), ReadProcessMemory(), WriteProcessMemory().

The purpose of these calls is to ensure the operation of debuggers (it logically follows that such

mechanics are available in all operating systems, allowing you to debug a third-party process.

Linux has them too: see ptrace,

https://habr.com/post/430302/

https://github.com/gaffe23/linux-inject

https://www.evilsocket.net/2015/05/01/dynamically-inject-a-shared-library-into-a-running-process-on-androidarm/ ).

Interesting injection technique without ptrace: https://github.com/DavidBuchanan314/dlinject

Simple injector for Linux: https://stackoverflow.com/questions/24355344/inject-shared-library-into-a-process

Therefore, the introduction into other processes and the manipulation of them (within the limits of access rights) is a matter of technique.

1. INJECTION INTO THE PROCESS

Injection into the process is needed to execute your code in someone else's process of the same user, in a situation where there is already access to the system.

The classic goal is browsers (interception and replacement of traffic to implement the MITM attack), instant messengers, email clients, etc.

The sequence of calls goes something like this:

OpenProcess - open someone else's process

VirtualAlloc - we allocate memory in it

VirtualProtect - allow this memory to run

WriteProcessMemory - write your shell code into this memory

CreateRemoteThread - run the shell code as a new stream

If the shell code is short and written with relative addressing, this is usually enough.

But this is a rare case, such code is usually written in assembler.

More often, you need to execute large chunks of logic that are formatted as a dll.

To run a dll in someone else's process, this dll must be copied to someone else's memory,

and configure it - register all imports, configure F-y addresses, etc.

In ordinary life, this is done by the LoadLibrary function.

But this is too conspicuous a function that also accesses the disk.

Two libraries implement a completely diskless injection from memory:

https://github.com/stephenfewer/ReflectiveDLLInjection

https://github.com/dismantl/ImprovedReflectiveDLLInjection

The second library is just an improvement on the first:

- работает кросс-разрядная инъекция (32->64, 64->32, 32->32, 64->64)

It is fundamental that the implementation has an injectable loader that uses only relative addressing.

The injection is as follows:

- The bootloader is written in one piece of memory

- dll is written to another piece of memory

- control is transferred to the bootloader

- the loader searches the import table for loadLibrary and GetProcAddress functions by hashes of their names

- the loader searches in the memory of the dll and configures it

- after configuration, the bootloader transfers control to DllMain

This technique can be improved by converting the injected bootloader into an external one:

- it will be executed from the attacking process, not from within the attacked

- It needs to replace all memory functions with WriteProcessMemory/ReadProcessMemory.

The sources of the finished implementation have not yet been found on the network, but in nature there is such an implementation.

Reflexive loading of dll into its own process: https://github.com/fancycode/MemoryModule

Reflective loading of the 80th level: https://github.com/bats3c/DarkLoadLibrary/blob/master/DarkLoadLibrary/src/ldrutils.c

https://www.mdsec.co.uk/2021/06/bypassing-image-load-kernel-callbacks/

A bypass of nuclear triggers for loading an image into memory is claimed, although how this differs from reflexive boot is not very clear

(UPD: report that the module record does not fall into the PEB)

Additionally, execute arbitrary shell code in the context of the current process:

https://github.com/DimopoulosElias/SimpleShellcodeInjector/blob/master/SimpleShellcodeInjector.c

2. INTERCEPTION OF FUNCTIONS (HOOKS)

Known for more than a dozen years technique:

- in the prologue of most system functions there are several nops (up to 5) provided

especially for this purpose

- the function prologue is replaced by jmp with our handler

- the handler calls the original function to offset the prologue after nop (or does not call - depends on the logic)

In order for everything to work without errors, the signature of our handler function must completely match the data types,

return and call agreement.

If there is no special prologue, the hook can still be put - not always. You need to look at the contents of the prologue.

Just the grouted jmp instructions need to be copied and transferred to the body of the new handler - but there is a creative approach,

and there are no universal recipes.

The Microsoft compiler has a switch: Project Properties -> C/C++ -> Code Generation -> Create An Image with Update

(patchable image) specifically for the purpose of generating empty function prologues.

Such a hook is also called a springboard, detour or inline hook.

In the article "Trampolines In x64" https://www.ragestorm.net/blogs/?p=107 you can see variations of trampolines.

This article https://www.malwaretech.com/2013/10/ring3-ring0-rootkit-hook-detection-22.html

расписано больше видов хуков (в т.ч. в ring0) - IAT hooks, inline hooks, SSDT hooks, SYSENTER\_EIP Hooks, IRP Major Function Hook

3. PROCESS HOLLOWING

Process Hollowing (aka RunPE) is a Windows-based implementation of the Unix equivalent of the fork()/exec() sis pair. That is, starting the process,

and replacing its text with another executable module, adjusting it in the process memory, and then starting it from the entry point.

In Windows, instead of fork()/exec(), CreateProcess() is used, which does both fork() and exec() at once.

That is, there is a substitution of the text of the process, which we cannot control.

Therefore, the implementation of text replacement, import settings and relocations must be done yourself, which is quite difficult

and essentially duplicates the Windows process loader.

The advantages of this trick are that:

- our code looks like another process

- another process may be a trusted OS process (explorer, svchost),

added to the exceptions of UAC, firewall and antivirus.

A detailed implementation is available at

https://github.com/m0n0ph1/Process-Hollowing

Antiviruses recognize process hollowing by comparing the text of the process in memory and on disk.

Implementation in VBScript, for use in macros-droppers:

https://github.com/itm4n/VBA-RunPE

4. PROCESS DOPPELGäNGING

The purpose of the technique is the same - to hide the name of the real startup process.

The essence of the performance is different.

- an NTFS transaction is opened to write to some executable file. As usual, it will be

someone from the trusted processes - svchost, explorer, etc.

- the body of the .exe is replaced by our text

- the process starts

- The transaction is rolled back. Physical writing to the body of the file does not occur.

For all other processes, it looks as if the original file is being launched.

But the thread that opened the transaction sees the new text in the file being launched.

The technique has become obsolete, because starting with some version of Windows 10, NTFS transactions seem to have been canceled altogether.

The implementation is available at

https://github.com/hasherezade/process\_doppelganging/blob/master/main.cpp

Description in English:

https://hshrzd.wordpress.com/2017/12/18/process-doppelganging-a-new-way-to-impersonate-a-process/

There are combos of two techniques:

Process Doppelgänging combo Process Hollowing

https://blog.malwarebytes.com/threat-analysis/2018/08/process-doppelganging-meets-process-hollowing\_osiris/

Transacted hollowing

https://github.com/hasherezade/transacted\_hollowing

4.1. PROCESS HERPADERPING

https://jxy-s.github.io/herpaderping/

https://github.com/jxy-s/herpaderping.git

The technique is similar to Process doppelganging:

- the real text of the program is written to a file and run CreateProcess

- before the actual stream starts, the text is erased with noise (for example, host code)

- uses the race condition in Windows Defender to slip it the left text of the program

5. REPLACEMENT PEB

Lightweight technique to hide the process after the fact.

PEB is a Process Environment Block, a structure with basic information about the process present in every process.

In particular, it has the name of the process and its command line. They can be replaced both from within the process and from the outside.

(suspending process threads using SuspendThread(), replacing writeProcessMemory() data, and resuming threads

using ResumeThread()). The modified process name and command line are then displayed in the list of processes.

Another interesting trick is to overwrite the export table .dll after loading into memory.

After you have obtained the addresses of all the necessary function addresses, you can overwrite the entire export and thus

complicate the identification of the process by known/unique function names.

Optionally, you can wipe the resource partition if they are not in use (or after they have been loaded).

In short, variations with substitution and mashing of service tables/areas are limited only by imagination.

6. SHELL CODE

The name goes back to epic times, when the injected code was designed to get a remote shell on the attacked machine.

Therefore, the code had to meet the requirements:

- be as brief as possible (to erase with less in the attacked process)

- be positionally independent (relative addressing only)

Today, shell code denotes any insertion of native code, regardless of the context of application.

However, the typical context is still an injection into a process, such as roughly into a piece of executable thread (to bypass DEP).

So the above requirements are valid.

A typical shell code for Windows processes does bootstrapping (including to bypass ASLR):

- identifies important milestones in memory - pointers to TEB and PEB

- from PEB gets the address of the import table

- gets the kernel32 address from the import table.dll

- from kernel32.dll gets loadLibrary, GetProcAddress addresses (usually by the hash of the name, not the name itself)

- When you have LoadLibrary and GetProcAddress, you can do whatever you want.

https://habr.com/ru/post/522966/

http://www.hick.org/code/skape/papers/win32-shellcode.pdf

(This dude http://www.hick.org/~mmiller/ by the way has a bunch of interesting, albeit outdated)

https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/

The same code can be written in C without using assembly.

Example - ReflectiveLoader() function https://github.com/dismantl/ImprovedReflectiveDLLInjection/blob/master/dll/src/ReflectiveLoader.c

Hell's Gate: parsing the search for system calls in position-independent C code

https://vxug.fakedoma.in/papers/VXUG/Exclusive/HellsGate.pdf

Конвертер PE->shell

https://github.com/hasherezade/pe\_to\_shellcode

7. INJECT ON/INFECT ON

To embed the load in an existing PE file:

https://github.com/secrary/InfectPE/

https://github.com/JonDoNym/peinjector

The decompression code (loader) and load are added to the file, the original entry point to the bootloader is forwarded.

So you can cling to legal software to cling to your loads.

Digital signatures of course fly off.

II. PROTECTIVE TECHNIQUES

HOW ANTIVIRUSES WORK

AVs receive information about what is happening in the system in one of two ways:

- Subscribing to kernel events using Event Trace for Windows (ETW) https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-

- connecting your driver-minifilter in kernel mode (ring0), and thus subscribing to file system activity and similar events (Windows Defender, Eset, WebRoot, McAfee)

- putting their hooks in user mode (ring3) directly into executable processes (Avast, BitDefender, Symantec, TrendMicro) on functions from ntdll.dll, user32.dll, kernel32.dll etc.

In the first case, nothing can be done, in the second case, the hooks can be removed.

WINDOWS DEFENDER DEVICE

Windows Defender uses a minifilter called WdFilter to subscribe to events on the system.

Caught:

- creating and starting a process

- Loading the process image

- Start streams

- manipulation of a non-native process - writing to memory and starting a remote stream

- downloadable drivers are verified

- Registry operations are intercepted.

Of all the useful things that we managed to find in the analysis - a list of conditions under which you can avoid such attention:

- whitelist of processes (including svchost, werfault and processes of WinDefender itself)

- list of hardened registry keys

https://n4r1b.com/posts/2020/01/dissecting-the-windows-defender-driver-wdfilter-part-1/

https://n4r1b.com/posts/2020/02/dissecting-the-windows-defender-driver-wdfilter-part-2/

https://n4r1b.com/posts/2020/03/dissecting-the-windows-defender-driver-wdfilter-part-3/

https://n4r1b.com/posts/2020/04/dissecting-the-windows-defender-driver-wdfilter-part-4/

EMULATOR DEVICE

The AB emulator is designed for additional analysis of the possible behavior of the program during static analysis of the binary:

https://findpatent.ru/patent/251/2514141.html

1) The emulator does not make all system calls, making a call to "stubs". Especially necessary for the functioning of the program can perform (such as memory allocation),

but in general, no;

2) the emulator is more interested in the sequences of system calls than their result;

3) in some AB emulator performs all branches in the code regardless of the conditions (according to some evidence of VBA32).

To look into all the branches of the code.

4) the emulator is limited by the time of operation (on the order of tens of seconds, or by the number of system calls made).

In some AB, the emulator starts with hooks embedded in your code.

Avast Emulator Detection:

Countering Avast CyberCapture.

BYTE\* A = (BYTE\*)GET\_API(SendMessageA); Any feature that Avast traps

if (A[0] == 0xe9) // Если первая команда - JMP

{

In the sandbox, Avast sets traps with a transition to the address where the sequence of FF25 00000000 bytes (another JMP) is written

i.e. the characteristic feature of the Avast emulator is two consecutive JMP, E9 and FF 25 in the prologue of your function

In normal mode, Awast also puts hooks, but there are fewer of them, and they lead to the addresses of the forcibly embedded aswhook.dll

SIZE\_T W = (SIZE\_T)(A + 5) + (SIZE\_T)(\*(INT32\*)(A + 1));

if (\*(WORD\*)W == 0x25FF && \*(DWORD\*)(W + 2) == 0)

{

debug\_printfA(ORANGE, "Avast CyberCapture (sandbox) detected\n");

GET\_API(ExitProcess)(-1);

}

}

The bitDefender shell detection part device is described here https://habr.com/ru/company/skillfactory/blog/527512/

Emulator sources are available https://github.com/bitdefender/bddisasm

Emulator triggers:

- In-memory shell code definition:

- access to the EIP/RIP instruction pointer register

- access to TEB/PEB structures

- search for system functions such as GetProcAddress

- Self-modification (decoding on the spot)

- deciphering the lines on the stack

- Stack execution

- Using SYSCALL instead of entry points in library .dll

- in kernel mode

- SPECIFIC SWAPGS type instructions

- access to the MSR register

- Access to KPCR (Core Processor Control Area)

ANTI-VIRUS SOFTWARE

There are two types of antivirus detections (hereinafter referred to as AV):

- static (by signature)

- Dynamic (by behavior).

It is easy to fight with the first, it is difficult to fight with the second.

In addition, AVs use fuzzy hashing (ssdeep, imphash), neural networks, and Bayesian filters to detect previously unknown malware.

by the degree of similarity of the text or behavior to already known samples:

https://xakep.ru/2020/09/28/clear-hash/

AB necessarily put detections on known lines and import functions.

Necessary (but insufficient) measures to protect against AB:

- Obfuscation of strings

- import obfuscation (i.e. imported from third-party .dll functions)

- making noise in the code

- Encryption of the code with decryption before execution.

Additional measures to counter AB:

- disabling AB (with sufficient rights)

- Removal of hooks (only for AB running in userland)

The KHOBE (Kernel HOoks Bypassing Engine) technique is known. There is no code in the public, only general information is known.

You need to use a race condition (whose?), presenting a safe code for analysis at the moment and quickly switch the context immediately after analyzing AB.

For more information, see the section "Cleaning from anti-virus detections".

SEARCH FOR SIGNATURES IN KNOWN DATABASES

Windows Defender Rule Decoder:

https://github.com/hfiref0x/WDExtract

The easiest way is to look for already known signatures in the YARA signature database.

Yara is a free tool for malari signature analysis

https://virustotal.github.io/yara/

But the rules themselves from it are not laid out centrally.

There are commercial rulebooks, such as

https://www.nextron-systems.com/2018/12/21/yara-rule-sets-and-rule-feed/

There are free low quality https://github.com/Neo23x0/signature-base/

A large collection of databases, among which there is both slag and relevant: https://github.com/InQuest/awesome-yara

Rule syntax and features: https://habr.com/ru/company/varonis/blog/584618/

You can convert the ClamAV database to the human-friendly Yara format and then search for the necessary detections:

https://resources.infosecinstitute.com/topic/yara-simple-effective-way-dissecting-malware/

> YARA with ClamAV Rules

>

> YARA can be integrated with ClamAv rule database. Perform the below steps to integrate ClamAv rules with YARA:

>

> Download the ClamAV to YARA Python script here: https://code.google.com/p/malwarecookbook/source/browse/trunk/3/3/clamav\_to\_yara.py

> Download and unpack the ClamAV db: http://database.clamav.net/main.cvd

> Run the ClamAV to YARA Python script:

> python clamav\_to\_yara.py –f main.cvd –o testing\_clamav.yara

> Now test the converted rules with YARA like below:

> yara –r testing\_clamav.yara /directory/to/check"

ОТКЛЮЧЕНИЕ WINDOWS DEFENDER

Sometimes AB can simply be disabled, in particular you can add yourself to Windows Defender exceptions (not always) here

HKLM| HKCU\MACHINE\SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths

HKLM| HKCU\MACHINE\SOFTWARE\Policies\Microsoft\Windows Defender\Exclusions\Paths

And in about the same place, completely turn off the AB.

Note that this is done through policies!

OBFUSCATION OF SYSTEM CALLS

The simplest technique is to get the address of a function using GetProcAddress(LoadLibrary(decode("lib.dll"), decode("funcname"))

However, the GetProcAddress and LoadLibrary calls shine here.

The old but still effective GetApi technique was used in the Carberp Trojan:

https://github.com/hzeroo/Carberp/blob/master/source%20-%20absource/pro/all%20source/RemoteCtl/DrClient/GetApi.h

Its essence is to search for functions in the import table by hash of their name. This knocks down signature detections, although it is available to automated analysis.

in disassemblers. The hashing method can be changed periodically.

Another approach - inline system calls - put all the boilerplate code to prepare the system in assembly

https://github.com/JustasMasiulis/inline\_syscall

A library of substitutes for some WinAPI calls:

https://github.com/vxunderground/WinAPI-Tricks

Ultimately, system calls are still caught by AVs operating in kernel mode through call interception subscriptions.

CODE OBFUSCATION

Typically, the code in the form of .dll is compressed, encrypted and packed into an array (see below Hiding data in a code segment),

at run time they allocate memory extract and decompress code then configure the .dll in memory

(relocations, imports and all that). See below about cryptos and packers - that's what it is.

This trick is known to AB and does not save from proactive defense.

So the thought moved on:

1. Unpack each function just before execution.

In each function, a prologue and an epilogue are defined that deciphers and encrypts the body of the function, respectively.

Signatures are used to find the boundary of the function.

An external encoder is required to initially encrypt functions in a file.

Other on-the-fly decoding options are possible, but they are all difficult to implement.

2. own JIT interpreters.

For example, https://github.com/jacob-baines/jit\_obfuscation\_poc

The idea is clear from the name - we translate one code into another code that is unknown to antiviruses and inaccessible

automatic reverse and analysis.

VMProtect-2 - obfuscating virtual machine

https://back.engineering/tags/vmprotect-2/

There is an interesting project of an obfuscating compiler based on LLVM:

https://github.com/obfuscator-llvm/obfuscator

and explanations of its algorithms

https://0xpat.github.io/Malware\_development\_part\_6/

Another commercial:

http://www.star-force.ru/products/starforce-obfuscator/

An interesting approach to decrypting the code is to get the decryption key from the server.

This has some effect against AB emulators.

OBFUSCATION OF STRINGS

Before the advent of constexpr, the obfuscation of lines was done by a third-party utility and a two-pass assembly:

- rows in the code were marked with special markers, usually in the global table of rows

- rows were accessed through the decryption function, by its index in the table

- The utility passed through the finished .exe and replaced them with a ciphertext.

This approach made debugging very difficult and required additional build steps.

making the code unreadable.

constexpr allows you to encrypt the string at the build stage, solving all the listed problems.

However, this works in Visual Studio 2015 or later, requiring the C++ standard 14.

Ready-made encryption library Andrivet ADVObfuscator:

https://github.com/andrivet/ADVobfuscator

See also

https://github.com/Snowapril/String-Obfuscator-In-Compile-Time (based on Andrivet)

https://github.com/adamyaxley/Obfuscate

https://github.com/fritzone/obfy

https://github.com/revsic/cpp-obfuscator

Obfuscation works on the maximum optimization settings in Visual Studio:

- C/C++/ Оптимизация: Полная оптимизация

- C/C++/ Optimization / Optimization of the whole program: On

- C/C++/ Optimization / Deploy inline functions: disable

- C/C++/ Code Generation / Enable String Concatenation: Yes

however this parameter must be tried, affects when how, differently on x64/x86

- Other optimization settings also need to be tried, they can influence.

Developer comment that obfuscator does not work on full optimization for x86 compilation:

"... why the string obfuscater doesn't always work.

The point is in the decrypt method, it turns out when optimization is enabled to deploy the substituted functions,

the compiler inserts its body where this method is called, and since this body uses expressions that it can read at compile time,

then and decrypts the string it gets at compile time.

It turns out that it encrypted and decrypted the string at compile time.

Fixed by disabling the optimization to deploy substituted functions."

One of the known disadvantages of inline obfuscators is the restriction on the length of lines.

Each additional character in a string is an additional recursion of the compiler in compile-time calculations.

The stack for MSVC2015 ends at lengths of approximately 100 characters.

There is also a simple trick used in the absence of C++ (in pure C)

char str[] = { 'H', 'E', 'L', 'L', 'O' };

in this initialization, the string is entered into an array on the stack by mov's at run time, and as the string does not make it to the .data segment

(i.e. it will turn into a set of assembler instructions in a .text of the following form:

mov edx, 8EB5h

push edx

mov edx, 6C6CD488h

push edx

etc

by the way, the decoding of the line in place looks something like this:

mov edx, 8EB5h

xor edx, 8EDAh

push edx

mov edx, 6C6CD488h

xor edx, 0B1C0h

push edx

mov you are, esp

sub esp, 8pm

mov edx, 0F478h

xor edx, 0F459h

push edx

mov edx, 74690CD7h

xor edx, 2CB2h

push edx

The clean line looks like this in the disassembler:

.code:004010A5 aTest001 db 'Test001',0

.code:004010AD aLoremIpsumDolo db 'Lorem ipsum dolor sit amet',0

)

char str[] = "HELLO"; will be populated at compile time as a string and will fall into .data.

OBFUSCATION OF THE ENTRY POINT

This measure is used to counter AB emulators and manual analysis.

The actual entry point is different from the one declared in the PE/ELF headers.

For example, in .dll there is a fake harmless export (some DllMain, DoTheWork, etc.), which performs some actions to avert the eyes.

To start the real load, you need to pull an unexported function at an address known only to the launching context.

Another option is to use a DOS stub. Changing the MZ signature to any other in the PE binary will start the binary in DOS mode.

As a consequence, ab will ignore the true entry point.

In 16-bit mode, emulators do not work; through the 4B DOS interrupt function, you can run a third-party binari.

This can be used in the "breaking the chain" technique.

HIDE DATA IN A CODE SEGMENT

AVs are sensitive to an unusually large section of data (.data, .rdata) - this is a sign of hiding encrypted load code in it.

You can hide data in a section of text. The Microsoft C++ compiler allows you to do this with this trick:

#pragma code\_seg(push, ".text")

#pragma code\_seg(pop)

unsigned char \_\_declspec(allocate(".text")) hiddencode[]={ ... };

According to a similar principle, you can add the load to other sections, pragms / declspec data\_seg, const\_seg.

True, the linker can breed sections with the same names and different access rights, so there is also such an option:

#pragma comment(linker,"/SECTION:.data,EW")

unsigned char PayloadName0[]={}

#pragma comment(linker,"/SECTION:.rdata,R")

unsigned char PayloadName2[]={}

Ab's counteraction to this measure is the frequency analysis of a section of code. The code section has low entropy,

because the number of opcodes is limited, and the statistical distribution of symbols in the code has a clearly defined structure.

Therefore, hiding encrypted and/or packed arrays is quite clearly monitored.

This, in turn, can be countered by weakly changing entropy encryption techniques - for example,

xor 1 byte (of course, if the code is hidden in such an array in the form of .dll. if there is other data, this will not help).

About entropy and in general what your PE file looks like for AV:

https://0xpat.github.io/Malware\_development\_part\_4/

Померять энтропию можно DIE (Detect It Easy).

CRYPTORS AND PACKERS

Reliable implementations of packers have been known since at least the mid-90s.

The idea is simple - one .exe is packed inside another .exe, and when performed performed performs the trick of Baron Munchausen

to pull out and start the load from yourself.

Of course, this is a great tool for hiding code.

Encryption is also added to the packaging.

Packaging can be multi-layered, to complicate the analysis.

The peak of the popularity of packers fell on the 0s.

To determine the type of packer, the PEId program was used (discontinued in 2011).

Now much smarter cryptors are used.

Cryptor in addition takes on the functions of bypassing the emulation of AB, the detection of sandboxes,

sometimes even bypassing UAC and elevating privileges (due to the nature of the startup of the load these functions

it is appropriate to place it on the cryptor).

Also, in addition to the banal pulling of the load from the encrypted array inside itself,

a good cryptor generates plausible import tables, plausible code that confuses AB,

dilutes the entropy of the load, distributes the load randomly in different sections,

generates real resources (lines in localization), in a word it pretends that it is a real program.

A similar approach is described in the https://xss.is/threads/39006/

In short, it is a protective shell that hides your code.

Of course, cryptors are not omnipotent, and they will not remove behavioral detections.

An interesting approach to crypto construction: the load decryption key is located separately from the crypt and is transmitted through the command line (or something else):

https://habr.com/ru/company/solarsecurity/blog/519994/

Using combos from public code in a packer from scratch

https://iwantmore.pizza/posts/PEzor.html

including using

Donut packer https://github.com/TheWover/donut

морфер Shikata Ga Nai https://github.com/EgeBalci/sgn

Open Source Cryptor:

https://github.com/oddcod3/Phantom-Evasion

One more:

https://github.com/ximerus/Kryptonite

Tread with "nanomites":

https://www.apriorit.com/white-papers/293-nanomite-technology

Blocking the debugging of the process by its own debugger;

Replace ALL navigation instructions in the child stream with INT 3 (debug interrupt) codes.

and generating a transition address in the debugger process.

REMOVAL OF HOOKS

To remove other people's hooks, you can use the comparison of the function prologue in the process memory

with a prologue in the file of the corresponding .dll. If they differ, this is a sign that someone else's hook is put on the function.

Fight appropriate: read body functions from a file and replace body functions in memory.

The first 10 bytes are sufficient.

Overview and comparison of different techniques:

https://www.first.org/resources/papers/telaviv2019/Ensilo-Omri-Misgav-Udi-Yavo-Analyzing-Malware-Evasion-Trend-Bypassing-User-Mode-Hooks.pdf

Demo: https://github.com/apriorit/antirootkit-anti-splicer

More: https://github.com/CylanceVulnResearch/ReflectiveDLLRefresher

Hook detection: https://github.com/asaurusrex/Probatorum-EDR-Userland-Hook-Checker

Comparison of userland hooks of different EDRs; Direct work with problems: https://github.com/Mr-Un1k0d3r/EDRs

Keep in mind that the hooks on your process can be repaired after you have removed them.

INTERCEPTION OF OTHER PEOPLE'S FLOWS (INJECTION PROTECTION)

You can intercept someone else's injection into the process by putting your handler on the BaseThreadInitThunk() function.

The creation of a new thread begins with it (including an externally initiated process).

In this handler, you can decide whether to allow or block the flow from starting, according to certain characteristics.

The simplest approach is to start all your threads at once and then block everything else.

If this is not acceptable, you can look at the address and properties of the memory page from which the code is run.

In the injected stream, this is usually a heap. In the flow of a healthy person, this is a section of text (.text).

In particular, this is how injection protection is implemented in the Mozilla Firefox browser.

This technique can be successfully counteracted - from outside the process you can remove the hook on BaseThreadInitThunk

according to the method of removing hooks described above, after which the injection is possible.

Another way is to unscrew all the mitigations to the maximum immediately after the start of the process (see below) in particular by DEP.

and code signing.

https://ethicalchaos.dev/2020/06/14/lets-create-an-edr-and-bypass-it-part-2/

Here is how to protect the process by:

- "innocent" code development

- Hook removal

- Work with direct calls

- Mitigation

- SharpBlock is another technique that uses capturing the start of a child stream using debug events, and patching its entry point to obfuscate the EDR.

PROTECT A PROCESS FROM COMPLETION

1. Deny access through a discretionary access control list (DACL). DACL is empty => process can only be killed by an admin:

https://stackoverflow.com/questions/6185975/prevent-user-process-from-being-killed-with-end-process-from-process-explorer

2. Mark the process as critical (RtlSetProcessIsCritical, NtSetInformationProcess).

Any attempt to stop such a process will result in a BSOD;

if you try to kill such a process through task manager you will receive a warning that the process is critical

and removing it could lead to the collapse of the system.

Requires administrator privileges and SeDebugPrivelege privileges:

RtlSetProcessIsCritical:

https://www.codeproject.com/Articles/43405/Protecting-Your-Process-with-RtlSetProcessIsCriti

NtSetInformationProcess with ProcessInformationClass = BreakOnTermination:

http://www.rohitab.com/discuss/topic/40275-set-a-process-as-critical-process-using-ntsetinformationprocess-function/

Using these calls can lead to drops.

It is empirically established that NtSetInformationProcess with the BreakOnTermination parameter works stably on 32-bit operating systems,

and RtlSetProcessIsCritical on 64-bit.

3. If there is a private key from the digital signature of the Microsoft code (lol))), then starting with Windows Vista

you can make any process protected from any changes from the outside.

also a protected parent process can spawn a protected child process

using a call to the CreateProcess function with the CREATE\_PROTECTED\_PROCESS flag.

This mechanism has been improved in Windows 8.1, but it is not perfect and does not exclude the possibility of making any process protected

or remove protection from digitally signed system processes.

An example of creating a secure child process is available in the description of the UpdateProcThreadAttribute function on MSDN:

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-updateprocthreadattribute

Article about protected processes from Alex Ionescu:

https://www.crowdstrike.com/blog/evolution-protected-processes-part-1-pass-hash-mitigations-windows-81

https://www.crowdstrike.com/blog/evolution-protected-processes-part-2-exploitjailbreak-mitigations-unkillable-processes-and

Presentation by Alex Ionescu:

http://www.nosuchcon.org/talks/2014/D3\_05\_Alex\_ionescu\_Breaking\_protected\_processes.pdf

An example of an exploit for a vulnerability in the Capcom driver that allows you to make any process protected:

https://www.unknowncheats.me/forum/anti-cheat-bypass/271789-pplib-ppl-processes.html

https://github.com/notscimmy/pplib

Article with examples of how to make a process secure and raise its privileges by patching the memory of the process:

https://www.blackhat.com/docs/asia-17/materials/asia-17-Braeken-Hack-Microsoft-Using-Microsoft-Signed-Binaries-wp.pdf

Source drivers that remove digital signature protection:

https://github.com/Mattiwatti/PPLKiller

https://github.com/katlogic/WindowsD

4. Other not the most effective ways to protect processes are described here:

https://security.stackexchange.com/questions/30985/create-a-unterminable-process-in-windows

PROCESS SHUTDOWN PROTECTION

1. You can install the console event handler by calling SetConsoleCtrlHandler,

in which to return 0 for CTRL\_LOGOFF\_EVENT events and CTRL\_SHUTDOWN\_EVENT.

- Works for console programs that do not run other console event handlers.

- Starting with Windows 7 event handling CTRL\_LOGOFF\_EVENT and CTRL\_SHUTDOWN\_EVENT not working

for programs that use the functions of the user32.dll and gdi32.dll libraries.

Example on MSDN:

https://docs.microsoft.com/en-us/windows/console/registering-a-control-handler-function

2. You can call the AbortSystemShutdown function in an infinite loop.

- requires administrator rights and SeShutdownPrivilege privileges

- does not have time to work out if you run the shutdown command in the console with the /t switch with a value of 0 (timeout 0 seconds)

- does not save from executing the shutdown command in the console with the /f switch

- Windows 10 doesn't seem to work.

3. You can create an invisible window and return 0 in the window event handler for WM\_QUERYENDSESSION and WM\_ENDSESSION events.

- Starting with Windows Vista, you need to call the ShutdownBlockReasonCreate function on an event WM\_QUERYENDSESSION,

or hide the window by calling the ShowWindow function with the second parameter set to FALSE (although the window is already created invisible).

- This does not save you from pressing the forced shutdown button.

- does not save from executing the shutdown command in the console with the /f switch

- does not work for console programs, in particular it is useless to use this technique inside a dll rundll32

Learn more on MSDN:

https://docs.microsoft.com/en-us/windows/win32/shutdown/shutdown-changes-for-windows-vista

https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms700677(v=vs.85)

MITIGATIONS

see SetProcessMitigationPolicy()/UpdateProcThreadAttribute(PROC\_THREAD\_ATTRIBUTE\_MITIGATION\_POLICY)

Allows you to enable dep, ASLR, to disable dynamic code generation,

additional checks on signature checks, validity of handles, SEHOP exceptions, and much more.

This is used by browsers and AV, twisting the mitigations on maximals in order to complicate injections into them or constrain the process.

On Windows 10, this is really effective.

Good article http://www.sekoia.fr/blog/microsoft-edge-binary-injection-mitigation-overview/

and the code to it

https://github.com/SekoiaLab/BinaryInjectionMitigation/

demonstrate the protection of the code by mitigation by checking the code signature.

This article https://habr.com/ru/post/494000/ gives an overview of mitigation policies, including the shadow stack, quote:

"Code Integrity Guard (CIG) requires that you sign downloadable binaries.

Arbitrary Code Guard (ACG) ensures that signed pages are immutable,

and dynamic code cannot be generated, which guarantees the integrity of the downloaded binaries.

With the introduction of CIG and ACG, attackers are increasingly resorting to takeover through indirect calls and returns.

известных как Call/Jump Oriented Programming (COP/JOP) и Return Oriented Programming (ROP)."

NETWORK ASYMMETRY

The results of the antivirus depend on the country.

Most of the manufacturers of AB solutions are countries of the collective West.

The effectiveness of AB depends mainly on the verification of neural networks in the "cloud".

With the beginning of the cyber confrontation between the Russian Federation and the United States, the latter prioritized all traffic from their country, to the detriment of other countries (including Western Europe),

in order to strengthen their own security. Apparently, the test capacity is not dimensionless.

Therefore, it has become a normal situation when the same load does not work in the US and works in other countries.

CLEANING FROM ANTI-VIRUS DETECTIONS

Before cleaning, you must first make sure that the antivirus does not drain samples:

- virustotal ALWAYS leaks samples live

- Dyncheck drains samples in dynamic behavior checks. With static checks - it seems not

- Windows Defender needs to disable the "Send samples" option

- The rest of the AVs need to find the option to send samples and cloud protection and disable them.

The general cleaning technique is as follows:

1. We find specific lines of code on which the detection is cocked;

2. Replace it.

Point 1 is long and tedious, it is done as follows:

- disable by comment or ifdef ALL the code of the program, collect

- AB plugged in

- uncomment half of the code

- AB Is Silent

- another half half

- AB Is Silent

- Half Half Half

- AB yells -> found the site!

Next, by the same dichotomy, we reach specific lines:

- Uncomment the line - yells, comment - silent.

Consider the optimizing compiler: the optimizer can throw out a huge block of code,

If you don't see the effect of this code on overall behavior.

For example, if you put return in the middle of the function under test as part of the detection search,

the optimizer can throw out of the final binary and the tail of the f-ii, and its beginning,

because it will consider that in the remainder of the f-iya nothing useful and affecting the overall implementation occurs.

By the same principle, the optimizer throws out unexpected sections of code, which is a great confusion of maps.

As a rule, anti-virus detections are exposed to:

- the name of the binary file (since the exposed name will give detections)

- Microsoft Visual C++ adds a string with the name of the project to the binary: Project Properties -> General -> Target Object Name (there is $(ProjectName) by default)

you need to randomize it, or just erase it with zeros/spaces in the build post-event right in the binary

- separate system calls (CreateRemoteThread, VirtualProtect, CreateFile, CreateProcess, OpenProcess, working with the registry, etc.)

- sequences of system calls (silent on individual calls, yells on sequence)

- open lines

- characteristic algorithms (random number generators, (de)encryption, (de)compression)

- high entropy of the binary file (encrypted/archived arrays, including in the code section)

Additionally, the detection on a blank file can be set:

- When downloading it from a site with a low reputation (to which there were already reports about the presence of suspicious files)

- Alternatively, when downloading a file from another country (affects both the client's address and the site address)

- If there is no hash of the file in the AB database (there are a limited number of launched files in the world)

Under Linux, do not forget to remove characters and extra lines with the strip utility (or even better sstrip).

System calls are either obfuscated by GetApi.h (can be taken directly from carberp/GetApi.h

https://github.com/hzeroo/Carberp/blob/master/source%20-%20absource/pro/all%20source/RemoteCtl/DrClient/GetApi.h)

or, if the desired call is missing in GetApi, in the following sequence:

HANDLE h = LoadLibrary(\_OBFUSCATED("dll.dll"));

void\* f = GetProcAddress(h, \_OBFUSCATED("funcname"));

here both lines are obfuscated.

Sequences of sis are obfuscated in the same way, or sisvys are creatively replaced by analogues.

For how to obfuscate strings, see above.

We dilute the characteristic algorithms with noise code:

- add/remove volatile to local variables

- change the place of definition of the local variable in the code (it was in the prologue of the f-i - put closer to the place of use, and vice versa)

- add noise code between lines (increment of garbage volatile variable, addition, subtraction, other operations)

- such a noise code can be issued in inline-f-u or macro: in the debug assembly, the body of the f-i is disabled, in the combat - constexpr's

random code patterns are created.

- remove the entropy by squeezing the pieces of the array into different sections and supplementing the arrays with unused constant bytes,

assembling the array before using it from chunks.

Another tactic for removing detections is the randomization of function addresses in the final .exe file.

This can be achieved by simply shuffling the list of object files on the linker command line:

link.exe /out:file.exe foo.obj bar.obj --> детект

link.exe /out:file.exe bar.obj foo.obj --> нет детекта

There is a utility to search for signatures in PE files:

https://github.com/vxlabinfo/SignFinder

based on articles

https://vxlab.info/%d1%87%d0%b8%d1%81%d1%82%d0%ba%d0%b0-pe32-%d1%87%d0%b0%d1%81%d1%82%d1%8c-1/

https://vxlab.info/%d1%87%d0%b8%d1%81%d1%82%d0%ba%d0%b0-pe32-%d1%87%d0%b0%d1%81%d1%82%d1%8c-2/

the site is already rotten, but copies remain, for example here

https://ru-sfera.org/threads/chistka-ot-signaturnogo-detekta-antivirusov.2870/

as well as google cleaning-pe32-part-1 cleaning-pe32-part-2

FILE REPUTATION

A file with a new unknown hash will be blocked by AB just in case.

This is called a "lack of reputation."

Therefore, usually the file is wound up with a reputation on controlled machines: they run, and when the AV is blocked,

unlock it manually, add it to the exception and make it work.

In the same way, the OEM SmartScreen works, and the CHROME AV screen.

Description of the reputation engine in Mozilla Firefox:

https://wiki.mozilla.org/Security/Features/Application\_Reputation\_Design\_Doc

BREAKING THE CHAIN

When bypassing AB in the case of a multi-stage load start, almost the only way to avoid detection is to break the startup chain.

So the parent process at the subsequent stages is not the previous stage, but the legitimate OS file.

For example, we want to download and run the second stage of the load from the loader.

If we do this directly, AB will detect a connection between the loader and the load.

If we add an intermediate link to the chain (for example, starting a load with the AT command), the chain will be broken.

One kind soul made a catalog of such system utilities and criteria for their use https://lolbas-project.github.io/

https://github.com/api0cradle/LOLBAS

PARENT PROCESS SPOOFING

A type of chain break that breaks the connection with the generated process.

lpStartupInfo is passed to CreateProcessA, whose lpAttributesList specifies the handle of the desired parent process.

By the way, in this way you can elevate privileges by inheriting the security context of the process.

Details in https://blog.f-secure.com/detecting-parent-pid-spoofing/

ZERG RUSH

Run 100500 different crypt hashes of the same load, thereby overloading the AV.

If you're lucky, AB will only cut 10499:

https://habr.com/ru/company/solarsecurity/blog/519994/

AMSI BYPASS

AMSI is an Antimalware Scan Interface, an antivirus module for analyzing the code of Windows scripting languages.

Он обрабатывает код на PowerShell, C#, VBScript, JavaScript, Windows Script Host (wscript.exe and cscript.exe),

макросов Office VBA, и UAC.

Brief essence: we analyze the sources (if necessary, decompile), only in static, put detections on lines - variable names, strings,

similar patterns, the use of bridges for WinAPI.

Clean sootvetstvenno.

A dude named S3cur3Th1sSh1t has done a great job of organizing all the AMSI workarounds.

Two articles describe how AMSI works and how to work around it:

https://s3cur3th1ssh1t.github.io/Bypass\_AMSI\_by\_manual\_modification/

https://s3cur3th1ssh1t.github.io/Bypass-AMSI-by-manual-modification-part-II/

TL;DR: AMSI puts detections on lines, so we actively rename identifiers in scripts,

glue strings on the fly, use left encodings to store scripts.

Detection scanner (finds the lines on which the detection is exposed):

https://github.com/RythmStick/AMSITrigger

Anti-AMSI Obfuscator:

https://amsi.fail/

Crawling techniques are collected here:

https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell

PowerShell Script Obfuscator:

https://reconshell.com/chimera-powershell-obfuscation-script-for-bypass-amsi-and-antivirus/

Documents are cleaned as follows:

- First of all, we change the wiring (hash of pictures, arrangement of elements, texts, etc.)

- obfuscate VBA code

- set timeouts before unpacking the dropper and before launching the file itself

FAT BINARY

Undeveloped, but long-known and promising technique associated with the need to deliver a universal 32/64-bit load:

https://en.wikipedia.org/wiki/Fat\_binary

https://habr.com/ru/company/macloud/blog/545278/

Due to machinations with executable headers, the starting prologue is common,

which then selects the desired load with the desired entry point.

III. DISCOVERY OF SANDBOXES AND DEBUGGERS

SANDBOX DETECTION

Sandbox detection is necessary in order not to be performed in it. Do not load and do not shine the main load.

Sandboxes are mainly made of virtual machines, but this criterion in itself is insufficient, because a legitimate terminal server may well be spinning on a VM.

Many methods are systematized here:

Al-Khaser: https://github.com/LordNoteworthy/al-khaser

PaFish: https://github.com/a0rtega/pafish

Good article on emulator and sandbox detection: https://0xpat.github.io/Malware\_development\_part\_2/

Below is a very brief and incomplete squeeze of strategies:

1. By machine name (https://www.blackhat.com/docs/us-17/thursday/us-17-Kotler-The-Adventures-Of-Av-And-The-Leaky-Sandbox.pdf):

\*CASE: REYNAPC, MALVAPC, ELEANOREPC, WRIGHTPC, BRIAPC, JORIPC, GABBIPC, HELSAPC, MAMEPC, SHARAIPC, ARACHONPC, FLORIANPC, EDITHPC

\*Various: WIN7-PC, ROGER-PC, DAVID-PC, ADMIN-PC, APIARY7-PC, ANTONY-PC, LUSER-PC, PERRY-PC, KLONE\_X64-PC, 0M9P60J5W-PC, MIKI-PC

\*Avira: C02TT22, C02TT26, C02TT36, C02TT18, C06TT43

\*Comodo: spurtive, circumstellar

\*Others: ZGXTIQTG8837952 (Comodo), ABC-WIN7, PC, WIN-U9ELNVPPAD0, PC-4A095E27CB, WIN-LRG949QLD21

2. By serial numbers and hardware name - MAC addresses of the network card, volume name of the hard disk

(vbox, qemu, vmware, virtual hd)

3. By running in the virtual machine.

4. By CPUID Instruction Execution Time

4.1. By the difference between GetTickCount() before and after Sleep();

5. By lack of activity in the interactive session (mouse, keyboard)

6. NotPetya killswitch

A few examples of sandbox detection implementations:

- https://habr.com/ru/company/solarsecurity/blog/473086/

- Review of Deshman methods from Positive Technologies: https://habr.com/ru/company/pt/blog/507912/

- Combine with multi-stage sandbox detection:

https://blog.talosintelligence.com/2020/05/astaroth-analysis.html

читать с раздела "Anti-analysis/Anti-sandbox mechanisms"

BYPASSING EMULATORS

The emulator is usually part of an antivirus, and it needs to determine very quickly whether to allow this code to work or not.

Because of this, validation in an emulator usually doesn't take long.

This is the main strategy for bypassing emulators - execution delay.

Simple Sleep() has not worked for a long time, because it is intercepted by the emulator, there is no real delay.

Therefore, as a rule, a calculation cycle is used instead of a delay (for example, calculating the pi number with great accuracy).

Many interesting and simple techniques for bypassing emulators:

https://wikileaks.org/ciav7p1/cms/files/BypassAVDynamics.pdf

An approach based on the imperfection of WinAPI emulation - analysis of ECX EDX registers after returning from the call:

https://winternl.com/fuzzing-the-windows-api-for-av-evasion/

https://github.com/jackullrich/Windows-API-Fuzzer

Earlier work

https://github.com/SPTHvx/SPTH/blob/master/articles/files/dynamic\_anti\_emulation.txt

Masking the true sequence of system calls in a large number of noise calls:

https://habr.com/ru/company/pt/blog/551954/

DEBUGGER DETECTION, DEBUG PROTECTION

1. IsDebuggerPresent() - unreliable, the patch function and it returns "we are not debazhat"

2. Search for processes by name (windbg, idapro, etc.)

3. Serifs of the time of passage of characteristic pieces of code

Interesting methods of protection against debugging in the OnionCrypter creeper: https://decoded.avast.io/jakubkaloc/onion-crypter/

- Use signatures from well-known packers (UPX) to loop auto analysis and confuse simpler reversers.

Of course, the load is not covered with the packer whose traces are left.

- an exception is thrown after the debugger is detected

- Three different features for memory allocation - HeapAlloc GlobalAlloc VirtualAlloc. A lot of false allocations make manual analysis difficult,

makes a breakpoint and hook on these features useless.

- starting the load through the callback of the system function. That is, we do not "transfer control to such and such an address",

a "call EnumWhateverA and pass the load entry point to this function as a callback".

IV. FASTENING TECHNIQUES

On Windows, the classic pinning methods are as follows:

- автозагрузка [HKLM| HKCU]\Software\Microsoft\Windows\CurrentVersion\Run

- CoCreateInstance scheduled task(CLSID\_TaskScheduler, ...)

Of the advantages - no admin rights are required, of the disadvantages - so obvious that they give an immediate detection of behavior.

- installing yourself as a service (you can't do without rights)

- BITS - not as commonly used, but doesn't bother AB

- ... (many technicians)

There https://habr.com/ru/post/425177/ a good overview of the techniques.

There http://www.hexacorn.com/blog/2017/01/28/beyond-good-ol-run-key-all-parts/ a huge number of standard

and non-standard expansion points in Windows. In particular, the ideas of launching by non-standard triggers - hardware events - are played out,

expansion points of popular programs etc.

See also "Breaking the Chain" above.

V. RETURN CHANNEL AND COMMUNICATION

C&C servers are always hidden: either behind gaskets (reverse proxies) or behind a torus domain.

Gaskets are lined up in cascades, a lot of them are done, so that the failure of one does not lead to the collapse of the entire network.

When implementing a communication channel with C&C, it should be remembered that the OS can configure its own system proxy.

For initial C&C searches, you can use the Domain Generation Algorithm (DGA)

http://www.marc-blanchard.com/BotInvaders/index.php

Its meaning is to generate pseudo-random domain names

- which are not too many (up to 10k) to sort through in a reasonable time;

- There are a lot of them that they cannot be squawked or otherwise banned/spoofed;

- the list of domains for one month is different from the list of domains for another month;

- It's hard to make a regular to cut them out on dns servers.

Typically, HTTPS is used to communicate with C&C, but the correct port or protocol is not always open.

If the DPI filter cuts HTTP(s) traffic, other protocols are used:

- you can make specific DNS queries to \_the necessary\_ (your) DNS server, and hide the information in domain names

- it is possible to make specific ICMP-parcels \*\*\*

- SMTP/IMAP/POP3 mail protocols are used

https://habr.com/ru/company/kaspersky/blog/522128/

In short, you can use different options for modulating the useful signal on top of the carrier,

which is guaranteed to pass through the firewall.

Traffic is intercepted and analyzed by systems like Suricata https://suricata-ids.org/

detecting anomalies and looking for patterns in traffic.

PyWhat library for automatic traffic parsing

https://habr.com/ru/company/dcmiran/news/t/563206/

https://github.com/bee-san/pyWhat

There are black lists of domains, addresses, SSL certificates, traffic profiles, for example:

https://sslbl.abuse.ch

https://urlhaus.abuse.ch

https://feodotracker.abuse.ch

Techniques such as JA3 client fingerprinting/JA3S server fingerprinting/JARM are used:

https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967

https://habr.com/ru/company/acribia/blog/560168/

Their essence is that the TLS handshake is predictable for the client + server bundle, because a large number of combinations of ciphers appear in the handshake,

taking into account their mutual location. Shaking hands with the same client with the same server is always the same.

With this handshakes remove the fingerprint, glue the TLS version, accepted ciphers, list extensions, elliptic curves and elliptic curve formats, and cover MD5.

Fingerprint tool:

https://github.com/salesforce/ja3

https://ja3er.com/form

The means to combat this is the randomization of the TLS stack (Cipher-Stunting) on both the client and the server (randomization of sslCipherSuite configuration and the like):

https://www.bc-security.org/post/ja3-s-signatures-and-how-to-avoid-them

https://www.vectra.ai/blogpost/c2-evasion-techniques

TL;DR: The Enable-TlsCipherSuite cmdlet allows you to change the combination of client ciphers - but - this is a system-wide configuration.

BCryptAddContextFunction

https://docs.microsoft.com/en-us/windows/win32/secauthn/prioritizing-schannel-cipher-suites

It is necessary to provide for the constant relevance of the address of the C&C server in the body of the program.

Therefore, instead of using regular domain names and DNS infrastructure,

you can use a public infrastructure that cannot be revoked:

- emercoin domains and DNS (can be made as specific queries using the Emercoin protocol,

and regular DNS queries to OpenNIC servers)

- records in cryptocurrency blockchains (modulation of information in amounts, addresses or service records)

- TOR domains (not everywhere TOR is open, and requires specific client code to work on top of TOR)

- Twitter and other public social networks (less often, because accounting can be revoked)

for example https://safe.cnews.ru/news/top/2020-09-08\_hakerynaemniki\_shest\_let

Communications Technology Overview courtesy of Positive Technologies: https://habr.com/ru/company/pt/blog/497608/

DNS-туннелирование: https://habr.com/ru/company/varonis/blog/513160/

To combat detections on network exchange, approaches with traffic masking are used.

To work on an industrial scale, constructors like the C2 Malleable Profile for Cobalt Strike are used

https://www.cobaltstrike.com/help-malleable-c2

https://github.com/threatexpress/random\_c2\_profile

or the C3 https://github.com/FSecureLABS/C3 Gateway Designer

The principle is to take the transport level of the software out of itself, to make it flexibly customizable and maskable,

make it possible to quickly expand and respond flexibly to detections due to modularity (the main functionality of the software is the main module,

network functionality - remote module-plugin).

If all channels inside the network are completely closed, then information can still be sent.

One option is mail (SMTP):

- Look for an email client on the local machine and put the letter in Outbox, not forgetting to clean up after yourself

- looking for an SMTP server in the locale (although they now always require authentication)

- Webmail (e.g. OWA = Outlook Web Access) can also be available on the local machine and the Internet can be closed.

Another option is the so-called "rocket":

https://www.blackhat.com/docs/us-17/thursday/us-17-Kotler-The-Adventures-Of-Av-And-The-Leaky-Sandbox.pdf

consisting, as the name suggests, of two stages.

The first stage should be Fully Undetectable.

It collects information from the system, generates a message, and flashes it into the second stage.

The second stage is a binary with the following properties:

- it annoys AV (there are detections by signature)

- he knows how to contact the C&C server and send him a package stitched by the carrier.

Further events develop as follows:

- Corporate AB sends the second stage to the "cloud" for execution in the sandbox

- The second stage hits the sandbox and starts

- The sandbox is located in the "cloud", outside the perimeter being defended, and from there there is a connection to the Internet

- The task of the sandbox is to investigate the behavior of the sample, so its activity is not jammed, although it is recorded

- The second stage sends data to C&C, and then it doesn't care

- PROFIT

VI. ELEVATION OF PRIVILEGES

BYPASSING UAC

That's the first thing you have to do: https://github.com/hfiref0x/UACME

The methods until about the 20th are outdated and do not work; to the 40th through one.

In a nutshell, the idea of circumvention is as follows:

1. Disguise the current process as legitimate, for which Windows never asks about the need for elevation (by replacing the process name in PEB)

2. pull another process that has auto-elevation set so that it starts the .exe we need (similar to suid root in Unix)

For the second point, there is a tueva hucha of ways.

In no case do not run on a personal machine! YOU CAN MAKE THE OS UNUSABLE!

ELEVATION OF PRIVILEGES (LPE)

For elevation of privileges, an overview of general strategies is here: https://habr.com/ru/post/428602/

Specific exploits quickly become obsolete, so we do not give here.

The general idea of LPE in Windows is to get and use someone else's security token https://habr.com/ru/company/pt/blog/563436/

Theory and practice LPE: https://habr.com/ru/company/otus/blog/530596/

Common techniques for exploiting vulnerabilities in the kernel: https://habr.com/ru/company/pt/blog/566698/

(HalDispatchTable patching, token theft)

VII. RESEARCH

How do you know what's going on inside a black box, whether it's a packer-covered .exe, or an unknown system at all?

INTERCEPTING SYSTEM CALLS

Windows has the Monitor API (bundled with rohitab) and Linux has strace.

We eavesdrop on the system calls of the executable file we are interested in, filter by the necessary criteria, understand the picture of what is happening.

TRAFFIC INTERCEPTION

Wireshark, you can find out addresses, ports, protocols. If you're lucky, you can even look at the details.

Because of the ubiquitous SSL, it has become difficult, but you can slip your root certificate into the system, raise a proxy using it,

and catch traffic already on the proxy.

https://mitmproxy.org/ proxy for HTTPS is also useful for catching problems at the junctions of subsystems.

PATCH DIFFING / BINARY DIFFING

A way to search for a vulnerability by fixing it.

Take the old executable file, take the new executable file (with the vulnerability patch), see the difference,

we calculate the details of operation by the difference.

https://habr.com/ru/company/dsec/blog/479972/

https://wumb0.in/extracting-and-diffing-ms-patches-in-2020.html

FUZZING

Feed the system at the entrance a fierce delirium generated by chance (but according to the rules), and see when it breaks (catch crashes).

Further, correlating the data at the input and crashes, we find a stable picture.

Further, it is already possible to make up a shell code for operation.

There are a lot of tools for this, no one does it manually, everything is automated.

Description of white box fuzzing (with known sources).

https://habr.com/ru/company/dsec/blog/517596/

It's more for autotesters, quality control, and test generation on the fly rather than research/reverse.

And yet the article gives general ideas about the methods and tools.

Popular AFL (American Fuzzy Lop) phaser based on genetic algorithms (morphing of the correct input sample)

https://github.com/google/AFL

he is also under the windu

https://github.com/googleprojectzero/winafl

Substantiation of approaches to fusing with the analysis of the theory:

https://habr.com/ru/company/bizone/blog/570312/

https://habr.com/ru/company/bizone/blog/570534/

https://wcventure.github.io/FuzzingPaper/

It's just a black box that's hopeless. Have to

- reverse/analyze code

- search for input checks

- Stuff the phaser with them so that it can correctly mutate the input and break through for checks

VIII. AUXILIARY SERVICE TECHNICIANS

EXCEPTIONS AND THE POSTHUMOUS STACK

Catching bugs and debugging on an industrial scale is a rather banal task, but the solutions for it are not known to everyone and are not always trivial.

The main way to find out what went wrong is to clear the posthumous call stack and send it telemetry to the server.

Exception Handling Theory and Practice in Windows: https://habr.com/ru/post/536990/

To do this, you need to remove this posthumous stack.

To do this, you need to catch the emergency flight of the program (a.k.a. crash) and remove the necessary information, before letting it die.

There are two main ways to do this:

\* VEH - Vectored Exception Handling

\* SEH - Structured Exception Handling

Linux/\*nix has signals (SIGBUS, SIGSTOP, SIGILL, etc.), man signal

And again:

- VEH is an AddVectoredExceptionHandler() and an article https://docs.microsoft.com/ru-ru/windows/win32/debug/using-a-vectored-exception-handler

- SEH is a \_\_try... \_\_except and https://docs.microsoft.com/en-us/cpp/cpp/try-except-statement?view=vs-2019 article

If SEH is used, then you need to wrap all the main threads in try/except.

If VEH is used, then it is enough to install one common handler in the program prologue.

Из минусов SEH - Ошибка C2712 Cannot use \_\_try in functions that require object unwinding, и решение описано тут

https://stackoverflow.com/questions/51701426/cannot-use-try-in-functions-that-require-object-unwinding-fix

Properties / C/C++ / Code Generation / Enable C++ Exceptions: No

The main drawback of either approach is that with process hollowed processes, no method will give line numbers and function names.

because character loading won't work. There will only be bare addresses.

In the exception handler, we need to clear the stack (code below).

If we want line numbers of code, we need characters (.pdb), and the project needs to be built with options

- C/C++ / General / Debug Information Format: Program Database (/Zi)

- Linker / Debug / Create Debug Information: Optimize for Debugging (/DEBUG)

- Linker / Debug / Create Full Program Database File: Yes

and .pdb must lie NEXT to the dying .exe or .dll.

For combat builds, this is not suitable, but for debugging on internal resources, you can do this. For combat builds, there will be just addresses in the stack, which is also a lot.

The stack removal code is quite small, we will give it here:

#include <windows.h>

#include <Psapi.h>

// Some versions of imagehlp.dll lack the proper packing directives themselves

// so we need to do it.

#pragma pack( push, before\_imagehlp, 8 )

#include <imagehlp.h>

#pragma pack( pop, before\_imagehlp )

#pragma comment(lib, "psapi.lib")

#pragma comment(lib, "dbghelp.lib")

\_\_declspec(noinline) DWORD DumpStackTrace() {

unsigned int i;

void \* stack[100];

unsigned short frames;

SYMBOL\_INFO \* symbol;

HANDLE process;

debug\_printf("PROGRAM CRASHED, STACK TRACE FOLLOWS:\r\n");

process = GetCurrentProcess();

if (! SymInitialize(process, NULL, TRUE))

return 0;

DWORD symOptions = SymGetOptions();

symOptions |= SYMOPT\_LOAD\_LINES | SYMOPT\_UNDNAME | SYMOPT\_LOAD\_ANYTHING | SYMOPT\_CASE\_INSENSITIVE;

SymSetOptions(symOptions);

frames = CaptureStackBackTrace(0, 100, stack, NULL);

symbol = (SYMBOL\_INFO \*)calloc(sizeof(SYMBOL\_INFO) + 256 \* sizeof(char), 1);

symbol->MaxNameLen = 255;

symbol->SizeOfStruct = sizeof(SYMBOL\_INFO);

DWORD offset\_from\_symbol = 0;

#ifdef \_WIN64

IMAGEHLP\_LINE64\* line = (IMAGEHLP\_LINE64\*)calloc(sizeof(IMAGEHLP\_LINE64), 1);

line->SizeOfStruct = sizeof(IMAGEHLP\_LINE64);

#else

IMAGEHLP\_LINE\* line = (IMAGEHLP\_LINE\*)calloc(sizeof(IMAGEHLP\_LINE), 1);

line->SizeOfStruct = sizeof(IMAGEHLP\_LINE);

#endif

for (i = 0; i < frames; i++)

{

SymFromAddr(process, (DWORD64)(stack[i]), 0, symbol);

SymGetLineFromAddr(process, (DWORD64)(stack[i]), &offset\_from\_symbol, line);

debug\_printf( "%i: %s (%s:%i) - 0x%0X\n", frames - i - 1, symbol->Name,

line->FileName, line->LineNumber, symbol->Address);

symbol->Name[0] = 0;

symbol->Address = 0;

if(line->FileName)

line->FileName[0] = 0;

line->LineNumber = 0;

}

free(symbol);

free(line);

return 1;

}

MAP FILES (. MAP)

Included in the linker:

Visual Studio / Project Properties / Linker / Debug / Create Mapping File: YES

If the crashed program does not have symbols in the .pdb, but the address of the crash is known, then the map can find the address of the function, as described here:

https://www.codeproject.com/articles/3472/finding-crash-information-using-the-map-file

REDUCE CODE SIZE

ОТКАЗ ОТ CRT (C RUNTIME LIBRARY)

An example of a program that compiles into a 3k .exe:

hello.cpp:

#include <windows.h>

const char \*str="Message";

int MyMain()

{

MessageBoxA(NULL,str,str,MB\_OK);

ExitProcess(0);

return 0;

}

build.bat:

set PATH=c:\LLVM9\bin

clang++.exe -DUNICODE -c -D\_UNICODE -m32 -std=c++14 -Wall -Os -mno-sse -fms-extensions -fms-compatibility -fno-exceptions -fno-rtti -fomit-frame-pointer -ffunction-sections -fdata-sections -Wno-c++11-narrowing -Wc++11-compat-deprecated-writable-strings \*.cpp -I"c:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\include"

lld-link.exe /subsystem:windows /nodefaultlib /entry:MyMain /libpath:"c:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\lib" /libpath:"c:\Program Files (x86)\Microsoft SDKs\Windows\v7.1A\Lib" \*.o kernel32.lib user32.lib

The default entry point is WinMainCRTStartup in the CRT. When your entry point, the CRT is not needed.

In addition, the /nodefaultlib switch that disables the CRT is specified here.

But the strcpy functions will have to be written by yourself, and exceptions will not be used.

But strcpy is already in shlwapi.lib, so we do it in the code

#include <Shlwapi.h>

#pragma comment(lib,"Shlwapi.lib")

TURN OFF SECURITY CHECKS

The Microsoft compiler puts a lot of additional protective code into the resulting code - canary stack, checking for out of arrays,

nullifying variables at the input to the function. All this gives extra kilobytes and is not needed in the battle code.

- Properties / C/C++ / Code Generation / Basic Runtime Checks (/RTC) - default (it's not very clear what to put)

- Properties / C/ C++ / Code Generation / Control Flow Protection - No

- Properties / C / C ++ / Code creation / Create an image with an update - No (only if you will not put hooks on your own functions)

- Properties / C / C ++ / Language / Remove code and data that is not referenced - Yes /Zc: inline)

- Properties / C / C ++ / Language / Include information about runtime types - No

OPTIMIZATION

- Disable stack frames (Omit frame pointers) - by default, when entering the function, the current top of the stack in the BSP register.

In this way, you can demarcate the personal stack of the current function from the stack of the upstream functions.

if you disable stack frame retention another register is released and the number of assembly instructions is reduced

both by not writing to it, and by the fact that more variables can be stored and passed through registers.

MICROSOFT RICH HEADER

http://ntcore.com/files/richsign.htm

http://bytepointer.com/articles/the\_microsoft\_rich\_header.htm

An abnormal section in the PE header that has been inserted by Microsoft linkers since 1998 and Microsoft Visual Studio 6.0.

It records statistics about the toolchain that collected this binary, such as the number of object files C,

number of C++ object files, number of ASM object files, linker version, resource compiler version,

the number of functions in the import, and all that.

This was most likely done for debugging purposes (to debug the assembly toolchain).

However, you need to understand that this title can be used for forensics as an imprint.

CROSS-BIT CODE

It is possible to execute both 32-bit code in 64-bit mode and vice versa.

The Windows kernel provides gateways such as Heaven's Gate to make system calls from 32-bit mode:

https://medium.com/@fsx30/hooking-heavens-gate-a-wow64-hooking-technique-5235e1aeed73

Another example:

http://blog.rewolf.pl/blog/?p=102

https://github.com/rwfpl/rewolf-wow64ext

GENERATION OF PSEUDORANDOM NUMBERS

The code and parsing of simple PRNG algorithms https://habr.com/ru/post/499490/ here

Theoretical Review of Non-Crypt-Resistant and Crypto-Resistant PRNG: https://habr.com/ru/post/531750/

It should be remembered that the quality of the RNG (namely random, without the letter P in the abbreviation) is the most important link in cryptography.

A good cryptographic algorithm is nullified by the use of a bad RNG in it (for example, to generate a gamma, an IV vector, etc.).

LITERATURE

1. M. Russinovich, D. Solomon - Internals of Microsoft Windows, 6th edition (Part 1) [2013, PDF, RUS] - https://rutracker.org/forum/viewtopic.php?t=4469765

2. M. Russinovich, D. Solomon, A. Ionescu - Internal device microsoft Windows. The main subsystems of the OS. 6th Edition (Part 2) [2014, PDF, RUS] - https://rutracker.org/forum/viewtopic.php?t=4727796

3. Developer Library - Love R. - Linux Kernel: A Description of the Development Process, 3rd ed. [2013, PDF, RUS] - https://rutracker.org/forum/viewtopic.php?t=5169029

4. Programmer's Library - Alger J. - C++ [2008, PDF, RUS] - https://rutracker.org/forum/viewtopic.php?t=694260

5. Chris Kaspersky. Collection of 507 articles - 2017.03.01 [PDF, DOC] - https://rutracker.org/forum/viewtopic.php?t=5375505

6. Understanding Windows Shellcode - mmiller@hick.org - http://www.hick.org/code/skape/papers/win32-shellcode.pdf

7. https://0xpat.github.io/ 0xPat blog Red/purple teamer - цикл статей "Malware development"