Pikabot | ThreatLabz

By Brett Stone-Gross, Nikolaos Pantazopoulos

Published: 2023-05-24 · Archived: 2026-04-05 15:51:24 UTC

Technical Analysis

In the following sections, we focus on Pikabot's core module and its injector since the downloader does not contain any functionality/features worth mentioning.

Core Module Injector

Pikabot uses an injector to run a series of anti-analysis tests and then decrypt and inject the core module payload. If any of these tests fail, Pikabot will terminate execution. ThreatLabz has identified the following anti-analysis methods implemented by the injector:

- Exception Handlers by using *int 2d* and *int 3* instructions to raise them.
- Reading the *BeingDebugged* flag of the process environment block (PEB).
- Use of the Windows API function *Beep* to delay the execution.
- Attempt to load junk and incorrect libraries in order to detect sandboxes.
- Use of the Windows API functions *CheckRemoteDebuggerPresent* and *IsDebuggerPresent* for debugger detection.
- The value of the NtGlobalFlag in the PEB that indicates a debugger is present.
- Use of the Windows API function *NtQueryInformationProcess* with the classes *ProcessDebugPort* and *ProcessDebugFlags*.
- Use of the *GetWriteWatch* API. The implementation seems to have been copied from here: https://github.com/BaumFX/cpp-anti-debug/blob/master/anti_debug.cpp#L260
- Use of the *OutputDebugString* function in order to detect a debugger. The implementation has been copied from here: https://github.com/BaumFX/cpp-anti-debug/blob/master/anti_debug.cpp#L456
- Check the number of processors, which should be greater than or equal to 2.
- Use of the *rdtsc* instruction to check for single stepping during debugging.
- The system random access memory (RAM) must be greater than 2GB.
- Detection of hardware breakpoints.
- Detection by checking the trap flag via *__readeflags*.

ANALYST NOTE: It should be noted that the use of exceptions is used in many parts of the code, for example, during the decryption of the core payload.

The injector decrypts the core module as follows:

1. Loads a set of PNG images, which are stored in the resources section (*RCDATA*), and decrypts them using a bitwise XOR operation with a hardcoded 32-byte key. Note that each PNG image holds an encrypted

chunk of the core module.
2. Decrypt the XOR decrypted data using AES (CBC mode) with the same 32-byte key and use the first 16 bytes of the encrypted data as an initialization vector (IV).

Once the core payload has been decrypted, the Pikabot injector creates a process with a specified file path (e.g. *WerFault*) and injects the core module into it. Finally, the Pikabot injector sets the *PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON* flag in order to protect the injected process from non-signed Microsoft binaries.

Core Module

In the following sections, the core module is analyzed with samples compiled in May 2023.

Anti-Analysis

Similar to the injector, the Pikabot core module performs additional anti-analysis checks. One notable technique is a “sleep” function, which Pikabot uses to delay execution. Instead of using common Windows API functions, Pikabot uses the *NtContinue* API function in order to set a timer. The technique is not new and it is similar to other proof-of-concepts implementations.

In addition to the tests above, Pikabot stops execution if the system's language is any of the following:

- Georgian (Georgia)
- Kazakh (Kazakhstan)
- Uzbek (Cyrillic)
- Tajik (Tajikistan)
- Russian (Russia)
- Ukrainian (Ukraine)
- Belarusian (Belarus)
- Slovenian (Slovenia)

This check is common for many threat actors that originate from countries in the Commonwealth of Independent States (CIS) to reduce the chances of criminal prosecution.

Persistence

Pikabot uses two methods to add persistence on a host:

1. Upon execution, Pikabot retrieves its current execution folder and checks if it is located in the *AppData* folder under a hardcoded folder name (might differ from sample to sample).

   If Pikabot is not run from this specific file path, then it will add persistence on the compromised host by creating a new value with its file path in the *Run* registry key (the key name is hardcoded in the binary). On top of that, Pikabot corrupts the current executable file by replacing it with its PE header (512 bytes length) followed by null bytes (3,584 bytes in length).

2. Pikabot downloads a PowerShell script from the command-and-control server and stores it in *HKEY_CURRENT_USER\Software\predefined_name*, where *predefined_name* is a hardcoded string in the binary file. Additionally, it stores the encrypted command-and-control servers in the same registry path.

   Lastly, Pikabot sets a value in the *Run* registry key to execute a command line that invokes this PowerShell script (e.g., *cmd /q /c start /min " powershell "$cimeter = Get-ItemProperty -Path HKCU:\Software\cimeter; powershell -encodedcommand $cimeter.unbevelledHamuli"*)

   ANALYST NOTE: Pikabot also has the option to directly execute a downloaded file instead of using this persistence mechanism (Even though this is not currently being used).

Command-and-Control Configuration

Pikabot does not store the command-and-control information in a single block (e.g. as Qakbot does). Instead, each component (e.g. URIs) is encrypted using ADVobfuscator and the command-and-control server IP addresses and ports are further decrypted during runtime using the following algorithm. Firstly, Pikabot decrypts a string that includes a set of Base64 encoded strings. Then it parses the string using the delimiter '&' and decrypts the contents by following the below steps:

1. Read the first 32 bytes of the string and use them as an AES key.
2. Decode the rest of the string using Base64.
3. Read the first 16 bytes of the decoded string and use them as an IV.
4. Read the rest of the decoded data and decrypt it using AES (CBC mode).
5. The decrypted output is a Base64 string, which results in the command-and-control server IP address and corresponding port. Note that many of the Pikabot command-and-control servers listen on ports that are identical to the ports used by Qakbot's proxy module such as 1194, 2078, and 2222.

There have been some minor changes over the last few months in the way that Pikabot command-and-control servers have been stored. For example, in previous versions, the command-and-control servers were only encoded using Base64 and no further encryption or parsing was required.

Pikabot also appears to contain a campaign ID and binary version in each sample. These can be observed during the network communication, where the JSON data has the keys "*version*" and "*stream*". The latter appears to be a campaign ID. An example request (before encryption) containing these values is shown below:

{"uuid": "F37670100000074E33652510483", "stream": "BB1@T@2e88e610b66b4205853b211f21873208", "os_version": "Win 10.0 19050", "product_number": 161, "username": "test", "pc_name": "DESKTOP-TEST", "cpu_name": "11th Gen Intel(R) Core(TM)", "pc_uptime": 29884462, "gpu_name": "GPU_NAME", "ram_amount": 8096, "screen_resolution": "1560x1440", "version": "0.5.3", "domain_name": "", "domain_controller_name": "unknown", "domain_controller_address": "unknown", "knock_timeout": 254, "is_elevated": 0}

The campaign ID values observed by Threatlabz are particularly interesting because of the prefixes **BB1** and **eu_bb_0.** These resemble some of the campaign IDs that have been observed in Qakbot binaries, which frequently

contain the prefix **BB** followed by an integer.

Network communication

Pikabot starts by registering the compromised host with the command-and-control servers. The registration process involves collecting system information and reporting it to the command-and-control server with an HTTPS POST request. A variety of data is collected such as the following:

1. Network information by executing the command *ipconfig.exe /all*
2. User/groups information by executing *whoami.exe /all*
3. Windows build information.
4. Generic host information (e.g. available RAM, screen resolution)
5. Domain controllers information.

Similar to other botnets, Pikabot generates a unique bot identifier for the compromised host. The algorithm, which Pikabot uses to generate the bot identifier can be replicated in Python using the code here.

Once the registration procedure has been completed and persistence to the compromised host has been established, Pikabot starts requesting tasks from the server. Pikabot supports the following command types:

- task - a command to execute
- knock - a keep-alive message

The tasks that Pikabot currently supports are described in Table 1.

Task Name	Description
cmd	Executes a shell command via cmd.exe.
destroy	Exits the current process.
shellcode	Injects and executes downloaded shellcode.
dll	Injects a downloaded DLL file.
exe	Injects a downloaded EXE file.
additional	Collects additional host information by executing one of the commands in Table 2.
knock_timeout	Updates the timer value, which indicates how often Pikabot should send a *knock* request.

Table 1 - Pikabot tasks description

Pikabot also supports the “additional” commands shown in Table 2, which are focused on collecting further system information.

Command	Description

screenshoot	Not implemented.
whoami	Executes the shell command *whoami /all*.
ipconfig	Executes the shell command *ipconfig /all*.
processes	Collects process information.

Table 2 - Additional Pikabot commands description

It is worth noting that depending on the network request, Pikabot uses a different URI (which may differ among samples). For example, for reporting a command output, the URI may be *Duenna/ZuGAYDS3Y2BeS2vW7vm?AnacrusisCrotalinae=zH4Tfz*.

The network data encryption procedure is similar to the configuration's decryption process. Pikabot encrypts a network request by following the steps below:

1. The request data is encoded using Base64.
2. Pikabot generates a random 32-byte key and encodes the data again using Base64.
3. Pikabot reads the first 16-bytes of a function prologue and uses these bytes as an IV.
4. The data is encrypted using AES (CBC mode) and encoded with Base64.
5. The 32-byte key is prepended to the encoded data.
6. The hardcoded URI key (e.g. mMG50=) is prepended to the final output.

Explore more Zscaler blogs
