Pikabot Updates | ThreatLabz

By Nikolaos Pantazopoulos

Published: 2024-02-12 · Archived: 2026-04-05 13:36:50 UTC

Technical Analysis

As covered in our previous technical analysis of Pikabot, the malware consists of two components: a loader and a core module. The core module is responsible for executing commands and injecting payloads from a command-and-control server. The malware uses a code injector to decrypt and inject the core module. It employs various anti-analysis techniques and string obfuscation. Pikabot uses similar distribution methods, campaigns, and behaviors as Qakbot. The malware acts as a backdoor, allowing the attacker to control the infected system and distribute other malicious payloads such as Cobalt Strike.

In the following sections, we will describe the latest Pikabot variant, including its capabilities and notable changes compared to previous versions. The analysis was performed on Pikabot binaries with version 1.8.32.

Anti-analysis techniques

As with previous versions of Pikabot, this variant employs a series of different anti-analysis techniques to make the analysis more time-consuming. It should be noted that none of the methods below presents any significant advanced capabilities. Furthermore, Pikabot used a series of more advanced detection features in its loader component in previous versions of the malware.

Strings encryption

The most notable change is the string obfuscation. In previous versions of Pikabot, each string was obfuscated by combining the RC4 algorithm with AES-CBC. This method was highly effective in preventing analysis, particularly when it came to automated configuration extraction. To successfully analyze Pikabot, an analyst would need to detect not only the encrypted string but also its unique RC4 key. Additionally, they would need to extract the AES key and initialization vector, which are unique to each Pikabot payload.It should be noted that the approach the Pikabot malware developers followed is similar to the ADVobfuscator.

In the latest version of Pikabot, the majority of the strings are either constructed by retrieving each character and pushing it onto the stack (Figure 1) or, in some rare cases, a few strings are still encrypted using the RC4 algorithm only.

```
42  memset(winhttp_dll_name, 0, 0xCu);
43  _w_ = ret_char('W');
44  v1 = L"bn-IN";
45  v2 = 7718;
46  winhttp_dll_name[0] = _w_;
47  while...
48  dword_410B04 = dword_410CB0 + 1776749091;
49  winhttp_dll_name[2] = ret_char('N');
50  winhttp_dll_name[7] = ret_char('.');
51  winhttp_dll_name[11] = ret_char('\0');
52  dword_410B04 = dword_410CB0 + 2023342520;
53  winhttp_dll_name[10] = ret_char('L');
54  junk_data_37(L"module", dword_410B04, dword_410B04);
55  v37 = dword_410CB0 ^ 0xE18B9AFA;
56  dword_410CB0 = (dword_410CB0 ^ 0xE18B9AFA) - 1105648297;
57  dword_410B04 = v37;
58  junk_data_37(L"sr-Latn-CS", v37, dword_410CB0);
59  winhttp_dll_name[4] = ret_char(84);
60  junk_data_37(L"module", dword_410CB0, dword_410CB0);
61  dword_410B04 -= 225733620;
62  dword_410CB0 = (dword_410B04 | 0x50C92EDC) - 2137932383;
63  winhttp_dll_name[8] = ret_char('D');
64  winhttp_dll_name[1] = ret_char('I');
65  winhttp_dll_name[9] = ret_char('L');
66  winhttp_dll_name[6] = ret_char('P');
67  dword_410CB0 = dword_410B04 + 2012698725;
68  dword_410B04 -= 719268259;
69  winhttp_dll_name[3] = ret_char('H');
70  v4 = ret_char('T');
71  v5 = global_struct;
72  winhttp_dll_name[5] = v4;
73  dll = load_dll(winhttp_dll_name);
```


Figure 1. String stack construction

Junk instructions

This anti-analysis technique was also implemented in previous versions of Pikabot. Pikabot inserts junk code between valid instructions. The junk code is either inlined in the function or a call is made to a function, which contains the junk code (Figure 2).

```
77  if ( dll )
78  {
79    while ( 1 )
80    {
81      char_junk_string = *junk_string++;
82      if ( !char_junk_string )
83        break;
84      condition_int += char_junk_string;
85      if ( condition_int > 0x2290 )
86      {
87        unused_global_addr = unused_global_addr_2 ^ 0x98526BEA;
88        break;
89      }
90    }
91    unused_global_addr_2 = unused_global_addr | 0x1168933A;
92    memset_func(winhttp_dll_name, 0, 12);
```

Figure 2. Junk code

Anti-debug methods

Pikabot uses two methods to detect a debugging session. They are:

- Reading the *BeingDebugged* flag from the PEB (Process Environment Block).
- Calling the Microsoft Windows API function *CheckRemoteDebuggerPresent*.

Pikabot constantly performs the debugging checks above in certain parts of its code. For example, when it (en/de)codes network data or when it makes a request to receive a network command.

Anti-sandbox evasion

In addition to the anti-debugging checks above, Pikabot uses the following methods to evade security products and sandboxes:

- Pikabot utilizes native Windows API calls.
- Pikabot delays code execution at different stages of its code. The timer is randomly generated each time.
- Pikabot dynamically resolves all required Windows API functions via API hashing.

A Python representation of the algorithm is available below.

```
 api_name = b""
checksum = 0x113B
for c in api_name:
   if c > 0x60:
       c -= 0x20
   checksum = (c + (0x21 * checksum)) & 0xffffffff
print(hex(checksum))
```

Language detection

Identical to previous versions, Pikabot stops execution if the operating system's language is any of the following:

- Russian (Russia)
- Ukrainian (Ukraine)

This is likely an indication that the threat actors behind Pikabot are Russian-speaking and may reside in Ukraine and/or Russia. The language check reduces the chance of law enforcement action and potential criminal prosecution in those regions.

Bot initialization phase

Unlike previous versions, this version of Pikabot stores all settings and information in a single structure at a global address (similar to Qakbot). The analyzed structure is shown below. For brevity, we redacted non-important items

of the structure (such as Windows API names).

```
 struct bot_structure
{
  void *host_info;
  WINHTTPAPI winhttp_session_handle;
  bool bot_error_init_flag;
  FARPROC LdrLoadDll;
  FARPROC LdrGetProcedureAddress;
  FARPROC RtlAllocateHeap;
  FARPROC RtlReAllocateHeap;
  FARPROC RtlFreeHeap;
  FARPROC RtlDecompressBuffer;
  FARPROC RtlGetVersion;
  FARPROC RtlRandomEx;
  ---redacted-
  wchar_t* bot_id;
  bool registered_flag;
  int process_pid;
  int process_thread_id;
  int* unknown_unused_1;
  unsigned short os_arch;
  unsigned short dlls_apis_loaded_flag;
  int unknown_unused_2;
  unsigned char* host_rc4_key;
  int number_of_swap_rounds;
  int beacon_time_ms;
  int delay_time_ms; // Used only during the initialization phase of Pikabot.
  int delay_seed_mul;
  wchar_t* bot_version;
  wchar_t* campaign_tag;
  wchar_t* unknown_registry_key_name;
  cncs_info* active_cnc_info;
  cncs_info* cncs_list;
  int num_of_cncs;
  int unknown_unused_3;
  int max_cnc_attempts;
  wchar_t* user_agent;
  void* uris_array;
  void* request_headers_array;
  TEB* thread_environment_block;
};

struct cncs_info
{
  wchar_t* cnc;
  int cnc_port;
```

```
  int http_connection_settings; // If set to 1 then server's certificate validation is ignored and sets the flag
  int connection_attempts;
  bool is_cnc_unavailable;
  cncs_info* next_cnc_ptr;
};
```

Bot configuration

The latest version of Pikabot stores its entire configuration in plaintext in one address. This is a significant drawback since in previous versions, Pikabot decrypted each required element at runtime and only when required. In addition, many of the configuration elements (e.g. command-and-control URIs) were randomized.

ANALYST NOTE: Despite their randomization, all configuration elements were valid on the server-side. If a bot sent incorrect information, then it would get rejected/banned by the command-and-control server.

The configuration structure is the following:

```
 struct configuration
{
  int number_of_swap_rounds_number_of_bytes_to_read_from_end; // During the bot initialization process, this mem
  size_t len_remaining_structure; // Size of the remaining structure's data minus the last element
  wchar_t* bot_minor_version; // E.g. 32-beta. In some samples, this member contains both the major and minor ve

  size_t len_campaign_name;
  wchar_t* campaign_name;
  size_t len_unknown_registry_key_name;
  wchar_t* unknown_registry_key_name; // Used only in the network command 0x246F.
  size_t len_user_agent;
  wchar_t* user_agent;
  size_t number_of_http_headers;
  wchar_string request_headers[number_of_http_headers];
  int number_of_cnc_uris;
  wchar_string cnc_uris[number_of_cnc_uris];
  int number_of_cncs;
  cnc cns[number_of_cns];
  int beacon_time_ms;
  int delay_time_ms;
  int delay_seed_mul; // Multiplies this value with the calculated value of the operation - delay_seed_mul * 100
  int maximum_cnc_connection_attempts;
  size_t len_bot_version // major version + minor version
  wchar_t* major_version; // 1.8.
  int len_remaining_bytes_to_read; // Added to the first member and shows how many more bytes to read right afte
};

struct wchar_string
{
```

```
  size_t length;
  wchar_t* wstring;
};

struct cnc
{
  size_t len_cnc;
  wchar_t* cnc;
  int cnc_port;
  int connection_attempts;
  bool http_connection_settings;
};
```

Once Pikabot parses the plaintext configuration, it erases it by setting all bytes to zero. We assess that this is an anti-dumping method to avoid automating the extraction of the configuration.

Lastly, Pikabot loads any remaining required Windows API functions and generates a bot identifier for the compromised host. The algorithm is similar to previous versions and can be reproduced with the following Python code.

```
 def checksum(input: int) -> int:
    return (0x10E1 * input + 0x1538) & 0xffffffff

def generate_bot_id_set_1(host_info: bytes, volume_serial_number: int) -> int:
    for current_character in host_info.lower():
        volume_serial_number *= 5
        volume_serial_number += current_character
    bot_id_part_1 = checksum(volume_serial_number & 0xffffffff)
    return bot_id_part_1

def generate_bot_id_set_2(volume_serial_number: int) -> int:
    bot_id_part_2 = checksum(volume_serial_number)
    bot_id_part_2 = checksum(bot_id_part_2)
    return bot_id_part_2

def generate_bot_id_set_3(bot_id_part_2: int) -> int:
    out = []
    for i in range(8):
        bot_id_part_2 = checksum(bot_id_part_2)
        out.append(bot_id_part_2 & 0xff)
    out = bytes(out[-4:])
    return int.from_bytes(out, byteorder='little')

host_info = b"username|hostname"
volume_serial_number = int("",16)
bot_id_part_1 = generate_bot_id_set_1(host_info, volume_serial_number)
```

```
bot_id_part_2 = generate_bot_id_set_2(volume_serial_number)
bot_id_part_3 = generate_bot_id_set_3(bot_id_part_2)
bot_id = f"{bot_id_part_1:07X}{bot_id_part_2 & 0xffff:09X}{bot_id_part_3}"
```

ANALYST NOTE: In some samples, Pikabot does not read the volume serial number due to a bug in their code that causes a failure when calling GetVolumeInformationW.

Network communications

Pikabot contacts the command-and-control server to request and receive network commands. In this version, the network protocol has considerably changed. Pikabot starts by registering the compromised host to its server.

First, Pikabot collects information from the compromised host, such as:

- Monitor's display settings
- Windows version
- Hostname/username and operating system's memory size
- Beacon and delay settings
- Process information such as the process ID, parent process ID and number of threads (see the description of network command 0x985 for a comprehensive list).
- Bot's version and campaign name
- Name of the domain controller

Then Pikabot appends the following information to the registration packet:

- 32-bytes network RC4 key (unique per host), which remains the same for the session. In previous versions, Pikabot was using AES-CBC with a random key/IV per request.
- Unknown registry key name. We observed it used only in the network command with ID 0x246F.
- Number of swap rounds used for encoding the data. This remains the same for the rest of the session.

Next, Pikabot encrypts the data using the RC4 algorithm, encodes the encrypted output, picks a random URI from its list, and sends the data with a POST request to the command-and-control server.

The encoding involves bytes swapping for N times, where N is a randomly generated number in the range 0-25.

ANALYST NOTE: Despite the fact that a round number is set in the configuration (see the configuration structure), this value is ignored and Pikabot replaces it with a random value. Moreover, Pikabot has completely removed the JSON format in its network packets and inserts everything in a raw format.

If the bot registration is successful, Pikabot starts an infinite loop to request and execute commands.

Each incoming network command (with the exception of network command with ID 0x164) has a task ID that is placed at the start of the (decrypted) packet as a QWORD value. In Table 1 below, we list the identified network commands along with a description of their functionality.

Command ID	Description
0x164	Requests command from command-and-control server. The packet includes the command ID, size of bot ID, and the bot ID. The server replies with the same command ID if there is no network command for the bot to execute.
0x555	Reports the output of the executed network command to the command-and-control server.
0x1291	Registers the bot. An unknown integer value (0x1687) is appended in the packet at offset 8.
0x1FED	Updates beacon time.
0x1A5A	Terminates/kills the bot.
0x2672	Not implemented
0x246F	Writes a file to disk and adds registry data using the value name specified in the configuration (*unknown_registry_key_name*).
0xACB	Executes the system command and sends back the output. Includes the error code 0x1B3 if there is no output.
0x36C	Injects the code of a downloaded PE file. The target process information is specified in the network packet.
0x792	Injects the code of a downloaded shellcode. The target process information is specified in the network packet.
0x359	Executes system command and sends back the output. **Note:** Same as 0xACB but does not send the error code.
0x3A6	Executes system command and sends back the output. **Note:** Same as 0xACB but does not send the error code.
0x240	Executes system command and sends back the output. **Note:** Same as 0xACB but does not send the error code.
0x985	Collects processes' information. These are: • Executable's filename • Process ID • Boolean flag, which indicates if it is a Pikabot process.

Command ID	Description
	• Boolean flag, which indicates if Pikabot can access the process with all possible access rights. • Number of threads • Base priority of threads • Process architecture • Parent process ID
0x982	Not implemented

Table 1. Pikabot Network Commands

Explore more Zscaler blogs
