YiBackdoor: Linked to IcedID and Latrodectus | ThreatLabz

By ThreatLabz

Published: 2025-09-23 · Archived: 2026-04-05 21:42:05 UTC

Technical Analysis

In this section, the features and capabilities of YiBackdoor are described along with the code similarities with IcedID and Latrodectus.

ANALYST NOTE: YiBackdoor generates and uses pseudo-random values at different stages (e.g. for generating the registry persistence value name). The malware implements custom algorithms for deriving random values, which are primarily based on the bot ID (used as a seed) combined with an implementation of Microsoft's Linear Congruential Generator (LCG). Since not all pseudo-random values are generated using a single method, ThreatLabz reversed each function and ported them to Python individually. To ensure consistency and clarity throughout this blog, the random values that are referenced can be derived using the Python script available in the ThreatLabz GitHub repository.

Anti-analysis

YiBackdoor includes a limited set of anti-analysis techniques with most of them targeting virtualized environments, and by extension, malware sandboxes. The malware employs the following anti-analysis methods:

- Dynamically loads Windows API functions by walking the loaded modules list, computing an ROR-based hash for each function name, and comparing the results with expected values to identify specific Windows API functions.
- YiBackdoor utilizes the `CPUID` instruction with the parameter `0x40000000` to retrieve hypervisor information. The result is then compared to values that match known hypervisors, including the following:
  - VMWare
  - Xen
  - KVM
  - Virtual Box
  - Microsoft Hyper-V
  - Parallels
- Decrypts strings at runtime by pushing an encrypted string onto the stack, which is then decrypted by performing an XOR operation with a 4-byte key (that is unique for each encrypted string).
- Measures the execution time of a code block to determine if the host is running on a hypervisor. Specifically, YiBackdoor begins by calling the Windows API function `SwitchToThread` followed by a call to the instruction `rdtsc`. Next, YiBackdoor calls the `CPUID` instruction, which triggers a VM exit, and then calls `rdtsc` again to calculate the time taken to execute the `CPUID` instruction. Once the time has been calculated, YiBackdoor calls the `rdtsc` instruction two more times and calculates the execution time

again. This process is repeated 16 times and the final calculated value must be greater than 20 to bypass the detection. This behavior can be reproduced using the following code example.

```
 [[nodiscard]] bool isHyperVisor()
{
   uint64_t timer1 = 0;
   uint64_t timer2 = 0;
   int loop_counter = 16;
   int cpuInfo[4] = { 0 };
   while (loop_counter)
   {
       SwitchToThread();
       uint64_t first_rdtsc_timer_value = __rdtsc();
       __cpuid(cpuInfo, 1);
       timer1 += __rdtsc() - first_rdtsc_timer_value;
       SwitchToThread();
       uint64_t second_rdtsc = __rdtsc();
       uint64_t third_rdtsc = __rdtsc();
       timer2 += ((third_rdtsc
```

It is worth noting that YiBackdoor stores the aforementioned information internally, but does not use the information or transmit it to the C2 server. As a result, the detection methods outlined above currently have no impact on the code's behavior.

Initialization stage

There are several actions that YiBackdoor performs during the initialization phase including injecting code into a remote process and establishing persistence.

YiBackdoor first checks for existing instances of itself by attempting to create a mutex with a host-based name. If the mutex already exists, indicating another instance is active, YiBackdoor will terminate execution.

Code injection

Before proceeding to the core functionality, YiBackdoor ensures that it is running within an injected process. YiBackdoor determines this by checking whether its current memory address falls within the memory range of any loaded DLLs. If it does, YiBackdoor creates a new svchost.exe process and injects its code into it.

The injection begins with YiBackdoor allocating memory in the remote svchost.exe target process and copying its code into that new region. YiBackdoor patches the Windows API function `RtlExitUserProcess` with assembly code that pushes YiBackdoor's entry point on the stack, which is then followed by a return instruction. Thus, when the `RtlExitUserProcess` function is called, the process execution flow will be redirected to the YiBackdoor's entry point. Interestingly, the svchost.exe target process is created without any special flags (e.g., in a suspended state). However, YiBackdoor does have enough time to inject its code between the process creation and

termination. Since the `RtlExitUserProcess` function is hooked, the malware's code executes just as the target process is about to terminate. This injection technique may allow YiBackdoor to evade detection by some security products.

Persistence

After completing the code injection phase, YiBackdoor proceeds to establish persistence on the compromised host using the Windows Run registry key. YiBackdoor first copies itself (the malware DLL) into a newly created directory under a random name. Next, YiBackdoor adds *regsvr32.exe malicious_path* in the registry value name (derived using a pseudo-random algorithm) and self-deletes to hinder forensic analysis.

Backdoor configuration

YiBackdoor contains an embedded configuration stored in an encrypted state. The configuration blob is decrypted and initialized at runtime. The decryption algorithm uses a 64-byte string as the key, as shown in the decryption routine below.

```
def decrypt(data: bytes, key: bytes) -> bytearray:
  decrypted_config = bytearray()
  for i in range(len(data)):
      x = i % len(key)
      y = (i + 1) % len(key)
      cipher = key[x] + key[y]
      cipher = (cipher ^ data[i]) & 0xFF
      decrypted_config.append(cipher)
      rotation_x = ror(n=key[x] >> (key[y] & 7), bits=key[x] > ( rotation_x & 7), bits=key[y]
```

The decrypted configuration data includes the following information:

- A list of C2 servers (separated using a space delimiter) where each C2 server has a boolean flag to indicate if the requests should be in HTTP (false) or HTTPS (true). For instance, the entry `127.0.0.1:0` instructs YiBackdoor to communicate using HTTP to the C2 address 127.0.0.1.
- Three strings that are used for deriving the TripleDES encryption/decryption keys and the initialization vector (IV) during the network communication process.
- Two integer values that YiBackdoor converts to numerical strings, which are used to construct the C2 URI.
- An unknown string identifier, which could represent a campaign or botnet ID. In the sample analyzed by ThreatLabz, this value is set to the string *test*.

The configuration's structure is provided below.

```
 #pragma pack(push, 1)
struct configuration
{
 char C2s[300];
 char response_triple_des_key_table[192];
```

```
 char request_triple_des_key_table[192];
 char triple_des_iv[128];
 uint32_t uri1;
 uint32_t pad;
 uint32_t uri2;
 char botnet_id[64];
};
#pragma pack(pop)
```

ANALYST NOTE: Before decrypting the configuration data, YiBackdoor ensures that the encrypted configuration does not start with the hardcoded string "YYYYYYYYYY". If a match is found, the embedded configuration data is considered corrupted and the execution stops. ThreatLabz has not been able to confirm the reason for this check yet. Moreover, two of the three configuration C2s are local IP addresses, which further supports the argument that YiBackdoor is still in a development or testing phase.

Network communication

Before initializing a network session with the C2, YiBackdoor derives the C2 URL by reading the following values from the decrypted configuration blob.

- C2 domain or IP address.
- Two hardcoded strings that are used as part of the C2 URI.
- Generated bot ID (calculated at runtime).

Thus, the C2 URL is structured as `http(s)://C2/bot_id/uri1/uri2`.

Next, YiBackdoor creates a JSON packet that contains the host's system time (UTC format) and username. The JSON packet is then encrypted using the TripleDES encryption algorithm. The creation of encryption/decryption keys along with the IV is quite unique. The configuration blob includes three strings with each one of them used for deriving the encryption key, decryption key, and IV. However, YiBackdoor does not use their entire values. Instead, it uses the current day of the week as an offset to calculate the starting address of the target value. Using this approach, YiBackdoor manages to have dynamic (and different) encryption keys per day and as a result makes the network traffic more resilient against static-based signatures. This algorithm is shown in the figure below:

```
des_key_info *__fastcall get_tripledes_key_iv(des_key_info *key_info)
{
  void (__stdcall *GetSystemTime)(LPSYSTEMTIME); // rax
  decrypted_config *decrypted_config_ptr; // r8
  unsigned int iv_offset; // r9d
  __int64 parsed_day_of_week; // rdx
  char *iv_string_table; // rcx
  des_key_info *result; // rax
  SYSTEMTIME sys_time; // [rsp+20h] [rbp-18h] BYREF

  GetSystemTime = get_api(0LL, 0, 0x270118E2, 172);
  GetSystemTime(&sys_time);
  decrypted_config_ptr = ::decrypted_config_ptr;
  iv_offset = 0x10 * sys_time.wDayOfWeek;
  parsed_day_of_week = 0x18 * sys_time.wDayOfWeek;
  iv_string_table = ::decrypted_config_ptr->triple_des_iv;
  key_info->response_key = &::decrypted_config_ptr->response_triple_des_key_table[parsed_day_of_week];
  key_info->request_key = &decrypted_config_ptr->request_triple_des_key_table[parsed_day_of_week];
  result = key_info;
  key_info->iv = &iv_string_table[iv_offset];
  return result;
}
```

zscaler | ThreatLabz

Figure 1: Network dynamic key derivation function for YiBackdoor.

The encrypted output is then Base64-encoded and appended to the HTTP header *X-tag*, and sent in an HTTP GET request.

The C2 response decryption process is similar. YiBackdoor verifies the presence of the HTTP header X-tag and decrypts it. The decrypted header contains the same information that was included in the HTTP request. YiBackdoor then decrypts and parses the HTTP body data, which contains incoming commands, which are in a JSON format.

Network commands

YiBackdoor supports the commands described in the table below.

Command Name	**Command Parameters**	**Description**

Systeminfo	None	Collects the following system information:
		• Windows version.
		• List of process names.
		• Network and miscellaneous system information by executing the system commands provided below.
		  ◦ chcp 65001
		  ◦ whoami /all

Command Name	Command Parameters	Description
		◦ arp -a ◦ ipconfig /all ◦ net view /all ◦ nltest /domain_trusts /all_trusts ◦ net share ◦ net localgroup ◦ wmic product get name
screen	None	Takes a screenshot of the compromised host's desktop.
CMD	• Base64-encoded command line to execute. • Timeout value.	Executes a system shell command using cmd.exe.
PWS	• Base64-encoded command line to execute. • Timeout value.	Executes a system shell command using PowerShell.
plugin	• Plugin name. • Command data for the plugin to execute.	Passes a command to an existing plugin to execute based on its name and reports the result to the C2 server.
task	• Base64-encoded and encrypted plugin data.	Initializes and executes a new plugin. If the plugin already exists, then reload the plugin using the data that was received.

Table 1: YiBackdoor network commands.

Note that the command names above use inconsistent casing (e.g., camel case, lowercase, and uppercase).

The structures (in C format) that YiBackdoor uses to parse both tasks received and network commands are shown below.

```
 enum Command
{
 system_info = 0x3,
 screenshot = 0x4,
 execute_new_plugin = 0x5,
 execute_loaded_plugin = 0x8,
 execute_cmd = 0x9,
 execute_powershell = 0xA,
};
struct custom_string
{
 char *string;
 size_t size;
 size_t capacity;
};
#pragma pack(push, 1)
struct task_info
{
  uint32_t task_id;
  Command cmd_id;
  uint32_t unknown_ID;
  custom_string command_parameter;
  custom_string plugin_name;
  uint32_t  timeout_time;
};
#pragma pack(pop)
```

Command status

YiBackdoor reports the output of each command to the C2 by sending an HTTP POST request. Each command status packet is in a JSON format and includes the following information:

- Task ID.
- A boolean value that represents the execution status of the command.
- The output of the command.

The reported output is summarized in the table below.

Network Command	**Reported Information**
Systeminfo	• Collected system information.

Network Command	Reported Information
	• A list of loaded plugins that include the ID and name of each plugin in the format `plugin_name-ID.bin`.
screen	• Screenshot encoded in Base64 format.
task	• A list of loaded plugins that include the ID and name of each plugin in the format `plugin_name-ID.bin`.
plugin	• Output data resulting from executing a command within the specified plugin.
CMD/PWS	• Output data resulting from executing a system shell command formatted in Base64.

Table 2: YiBackdoor command status messages.

ANALYST NOTE: The task status for the network command 'task' is always set to true (success) regardless of the plugin's loading status.

Plugins

YiBackdoor stores each plugin that is received locally in the Windows temporary folder using a random filename with the file extension *.bin*. The malware identifies a target plugin by validating the filename against its own filename generation algorithm. The plugins are reloaded each time YiBackdoor is executed.

Each plugin is stored in an encrypted format. The following Python code snippet represents the encryption/decryption algorithm.

```
def fix_key(key: bytearray, x: int, y: int) -> bytearray:
   temp_val = key[y:y + 4]
   temp_val = int.from_bytes(temp_val, byteorder="little")
   rot_val = (temp_val & 7) & 0xFF
   temp_val = key[x:x + 4]
   temp_val = int.from_bytes(temp_val, byteorder="little")
   temp_val = ror(temp_val, rot_val) & 0xFFFFFFFF
   temp_val += 1
   temp_val &= 0xFFFFFFFF
```

```
        temp_val_x = temp_val.to_bytes(4, byteorder="little")
        rot_val = (temp_val & 7) & 0xFF
        temp_val = key[y:y + 4]
        temp_val = int.from_bytes(temp_val, byteorder="little")
        temp_val = ror(temp_val, rot_val) & 0xFFFFFFFF
        temp_val += 1
        temp_val &= 0xFFFFFFFF
        temp_val_y = temp_val.to_bytes(4, byteorder="little")
        temp_key = key[:x] + temp_val_x + key[x + 4:]
        temp_key = temp_key[:y] + temp_val_y + temp_key[y + 4:]
        return temp_key
    def crypt_plugin(data: bytes, key: int) -> bytes:
        decrypted_plugin = []
        for i in range(len(data)):
            x = (i & 3)
            y = ((i + 1) & 3)
            c = key[y * 4] + key[x * 4]
            c = (c ^ data[i]) & 0xFF
            decrypted_plugin.append(c.to_bytes(1, byteorder="little"))
            key = fix_key(key, x * 4, y * 4)
        return b''.join(decrypted_plugin)
```

YiBackdoor manages and parses any plugins by using the structures provided below.

```
 #pragma pack(push, 1)
struct struct_plugin_execution_info
{
 uint32_t unknown_field;
 uint32_t plugin_id;
 uint8_t do_start_plugin;
 char plugin_disk_name[16];
 IMAGE_DOS_HEADER* plugin_memory_data;
};
#pragma pack(pop)
struct plugin
{
 custom_string plugin_name;
 void *plugin_entry_address;
 void *plugin_data;
 void *sizeof_plugin_data;
 struct_plugin_execution_info *plugin_execution_info;
 void *mapped_plugin_memory_address;
};
struct plugin_manager
{
 plugin *plugins[1];
```

```
 uint64_t number_of_plugins;
 uint64_t max_allowed_plugins;
};
```

Code similarities

ThreatLabz observed notable code overlaps between YiBackdoor, IcedID, and Latrodectus. IcedID is a malware family that consists of several different components such as a downloader (which has gone through various updates in the past), a main module backdoor, and a main module loader. These similarities are present in both critical and non-critical parts of YiBackdoor's code.

The code similarities between YiBackdoor, IcedID, and Latrodectus are the following:

- The use of identical alphabet charsets to derive bot-specific randomized strings. The identified charsets are `aeiou` and `abcedfikmnopsutw`.
- The format (Base64) and length (64-bytes) of YiBackdoor's configuration decryption key matches the RC4 keys used by Latrodectus to encrypt its network traffic.
- YiBackdoor hooks the Windows API function `RtlExitUserProcess` as part of the remote code injection process. This code injection technique is quite uncommon and resembles IcedID's extensive use of this Windows API.
- Although YiBackdoor uses a different approach to calculate the bot ID, part of the process involves the Fowler–Noll–Vo (FVN) hashing algorithm, which is also present in the codebase of IcedID and Latrodectus.
- YiBackdoor includes a Windows GUID list that is not used during execution. The exact same array of GUIDs is present and utilized in both IcedID and Latrodectus. Hence, the GUIDs in YiBackdoor may be code remnants from the latter two malware families.
- The most significant code similarity is the decryption routines for the configuration blob and the plugins. The plugins' decryption routine is identical to the algorithm previously used by IcedID to decrypt the core payload and configuration data. The figure below shows the algorithm, comparing the decryption routine from a (GZIP) IcedID downloader sample and the plugins' decryption routine found in YiBackdoor. Furthermore, the algorithm used to decrypt YiBackdoor's embedded configuration blob is similar to the aforementioned decryption routine found in IcedID samples.

```
index = 0LL;
expected_checksum = *&data[data_size] ^ key[0];
if ( data_size )
{
  while ( 1 )
  {
    x = index & 3;
    y = (index + 1) & 3;
    decrypted_data[index] = data[index] ^ (LOBYTE(key[x]) + LOBYTE(key[y]));
    ++index;
    next_val = __ROR4__(key[x], key[y] & 7);
    key[x] = next_val + 1;
    key[y] = __ROR4__(key[y], (next_val + 1) & 7) + 1;
    if ( index >= data_size_without_key )
      break;
    data = input_data_pointer;
  }
}
calculated_checksum = 0LL;
for ( i = 0LL;
      i < data_size;
      calculated_checksum = (8 * (current_checksum + calculated_checksum)) | ((current_checksum + calculated_checksum) >> 29) )
{
  current_checksum = decrypted_data[i++];
}
if ( expected_checksum != calculated_checksum )
{
  if ( is_memory_allocated )
    virtualfree_wrapper(decrypted_data);
  goto failed_exit;
}
return decrypted_data;
}
```

Plugins' decryption routine in YiBackdoor.

```
index = 0LL;
for ( expected_checksum = LODWORD(data[0]) ^ *&key[data_size_without_key];
      index < data_size_without_key;
      *(data + y) = __ROR4__(*(data + y), next_val & 7) + 1 )
{
  x = index & 3;
  y = (index + 1) & 3;
  y_value = *(data + y);
  decrypted_data[index] = (*(data + 4 * x) + *(data + 4 * y)) ^ key[index];
  ++index;
  next_val = __ROR4__(*(data + x), y_value & 7) + 1;
  *(data + x) = next_val;
}
calculated_checksum = 0;
for ( i = 0LL; i < data_size_without_key; calculated_checksum = __ROL4__(current_checksum + calculated_checksum, 3) )
  current_checksum = decrypted_data[i++];
if ( expected_checksum != calculated_checksum )
{
  if ( is_memory_allocated )
  {
    if ( decrypted_data )
    {
      heap_handle = GetProcessHeap();
      HeapFree(heap_handle, 0, decrypted_data);
    }
  }
  return FALSE;
}
*sizeof_decrypted_data = data_size_without_key;
return TRUE;
```

Configuration decryption routine found in IcedID GZIP downloader.

zscaler | ThreatLabz

Figure 2: Comparison of YiBackdoor and IcedID GZIP decryption routines.
