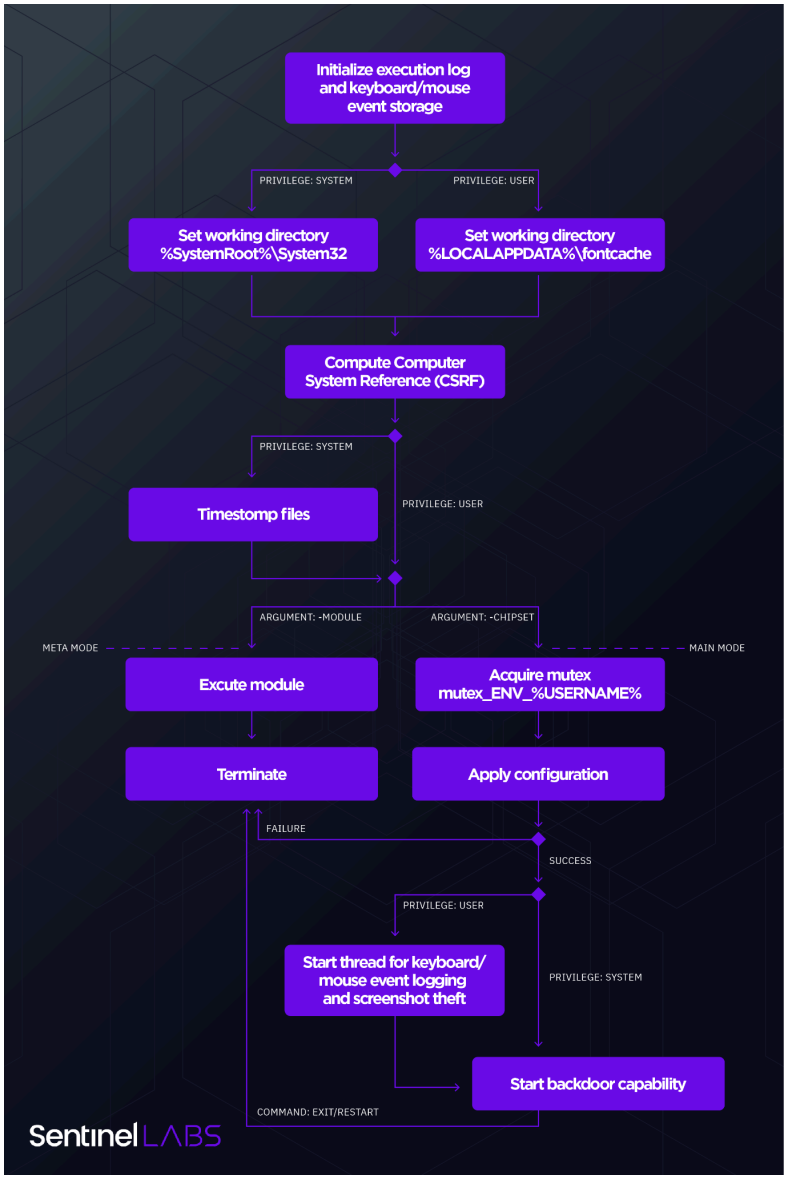
SentinelLABS

# Metador Technical Appendix

An extensive technical analysis of the toolset used by a sophisticated threat actor as described in our detailed report "The Mystery of Metador | An Unattributed Threat Hiding in Telcos, ISPs, and Universities"

## metaMain




*A high-level overview of the operation of metaMain*


*A high-level overview of the backdoor capability of metaMain*

## start_methods

We refer to metaMain as a multi-mode implant because it runs in either of two modes , 'meta' or 'main'. The implant is acutely aware of its own execution context and acts accordingly. The full features of metaMain are described further below. First, we'll detail the different loading schemes supported by metaMain: (1) execution via CDB (in 'main' instead of the 'meta' mode discussed in the main report), (2) HKCMD sideloading, and (3) KL_INJECTED.

## 1. CDB_DEBUGGER

In a metaMain, 'Main'-mode CDB execution flow, metaMain will be loaded as a full-fledged implant. The execution scheme for this flow does not include the Mafalda implant but is very similar to the meta-mode execution previously described.

In this scheme, cdb.exe will be executed with the following command line:

```
c:\windows\system32\cdb.exe -cf c:\windows\system32\cdb.ini
c:\windows\system32\resmon.exe -chipset
```

In this flow, metaMain enters its main function and operates as a standalone implant Notice that in this case, the operators replaced 'defrag.exe' with 'resmon.exe' as the debugged process.

## 2. HKCMD Sideloading

metaMain supports an HKCMD sideloading start method, most likely involving hkcmd.exe DLL search order hijacking, which was not observed in the wild. Nevertheless, analysis of metaMain sheds some light on the files involved. Some of those files are listed in metaMain's built-in timestomping functionality, and include:

```
c:\windows\system32\hkcmd.exe
c:\windows\system32\hccutils.dll
c:\windows\system32\hkcmd.db
```

## 3. KL_INJECTED

metaMain supports a third, "virtual" start method. In this method, the metaMain instance has been started by another metaMain instance that has injected the metaMain's reflective DLL Loader, *Speech02.db*, into a process. The metaMain instance that conducts process

injection executes in the *CDB_DEBUGGER* start mode, with SYSTEM user privileges. The newly injected instance, if running with regular user privileges, will perform keyboard and mouse events logging as well as take screenshots).

To remind, a metaMain instance in the KL_INJECTED start method has been started by another metaMain instance that has injected the metaMain's reflective DLL Loader, *Speech02.db*, into a process. The metaMain instance that conducts process injection executes in the *CDB_DEBUGGER* start mode, with SYSTEM user privileges:

- The metaMain instance creates a thread internally referred to as *k_inj_thread*.

```
[...]
if ( isSYSTEM_byte_1CF8857FD20 )          // if metaMain executes with SYSTEM user privileges
  {
    if ( start_method_dword_1CF88581C50 == 2 )// if the metaMain start mode is CDB_DEBUGGER
    {
      std::wstring::assign(&Block, L"create k_inj_thread ");
      write_in_logbuffer_sub_1CF8853A108(
        (__int64)&Block,
        (__int64)&config_INJECT_KEYLOGGER_PROCESS_qword_1CF8857FCA0);
      if ( v15 >= 8 )
        j_free(Block);
      CreateThread(                       // metaMain creates a thread internally referred to as k_inj_thread
        0i64,
        0i64,
        (LPTHREAD_START_ROUTINE)thread_inject_speech02_in_process_sub_1CF88533770,
        0i64,
        0,
        0i64);
      [...]
    }
```

A metaMain instances creates the *k_inj_thread* thread (IDA Pro pseudocode, trimmed for brevity)

- The *k_inj_thread* thread enumerates the processes that run on the platform where metaMain executes to locate a target process for injection, with a name that equals INJECT_KEYLOGGER_PROCESS – a configuration variable of metaMain.

- After *k_inj_thread* has located a target process, the thread writes into the registry value *HKEY_LOCAL_MACHINE\SOFTWARE\DDE\tpid* the process ID of the target process and injects *Speech02.db* into the process.

To determine that it executes in the KL_INJECTED mode, metaMain evaluates whether the ID of the process in whose context the implant runs is the same as the registry value *HKEY_LOCAL_MACHINE\SOFTWARE\DDE\tpid*.

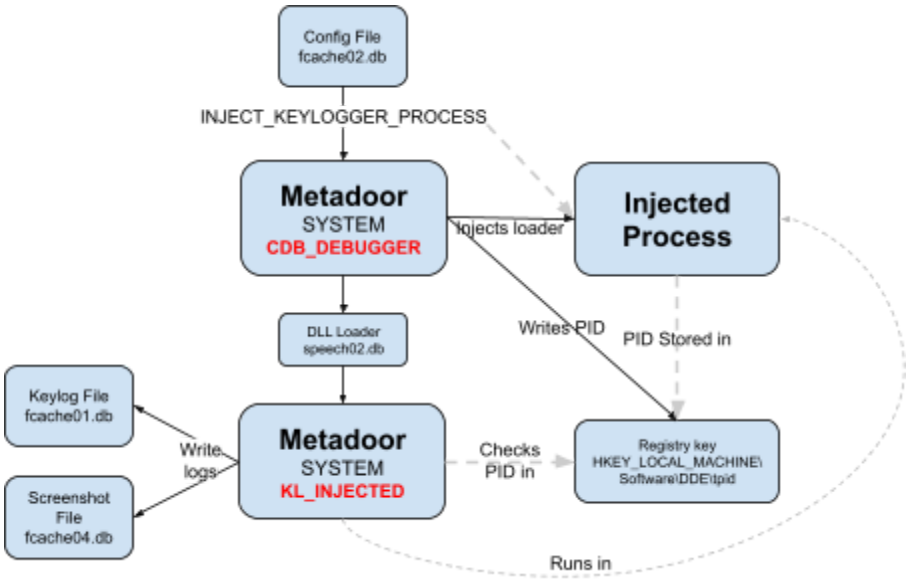
```
[...]

strcopy_sub_1CF88522A98(str_1, L"tpid");
strcopy_sub_1CF88522A98(str, L"SOFTWARE\\DDE");
tpid_value = sub_1CF88537550(str, str_1, v36);

[...]

if ( tpid_value == (unsigned int)kernel32_GetCurrentProcessId() )
{
  strcopy_sub_1CF88522A98(str_method, L"troj_start_method := KL_INJECTED");
  [...]
  start_method_dword_1CF88581C50 = 3;
}
```

A metaMain instance determines that it executes in the KL_INJECTED mode (IDA Pro pseudocode, trimmed for brevity, *tpid_value* holds the registry value *HKEY_LOCAL_MACHINE\SOFTWARE\DDE\tpid*)



## Meta and Main Execution Modes

metaMain maintains an execution log that records events that pertain to the implant's operations, which represents a rich source of information for analysts. metaMain stores this log in both an internal memory region and on the filesystem.

```
2021/05/19 14:41:16 I am < SYSTEM >
2021/05/19 14:41:16 Config dir  C:\Windows\system32\
2021/05/19 14:41:16 computer name:  [...]
2021/05/19 14:41:16 network id:  SFE979EE2
[...]
2021/05/19 14:41:16 my file name: < defrag.exe >
2021/05/19 14:41:16 cdb.exe or kl inj, deleting mem at 0x29CE6830000
2021/05/19 14:41:16 getRegistryInt key SOFTWARE\DDE GetLastError(): 0 The operation completed successfully.
2021/05/19 14:41:16 regInjPid: 0
2021/05/19 14:41:16 troj_start_method := CDB_DEBUGGER
2021/05/19 14:41:16 setting harmless file times
[...]
2021/05/19 14:41:16 cmdline_arg1: -module
2021/05/19 14:41:16 I am meta
2021/05/19 14:41:16 shutdown, ExitProcess
```

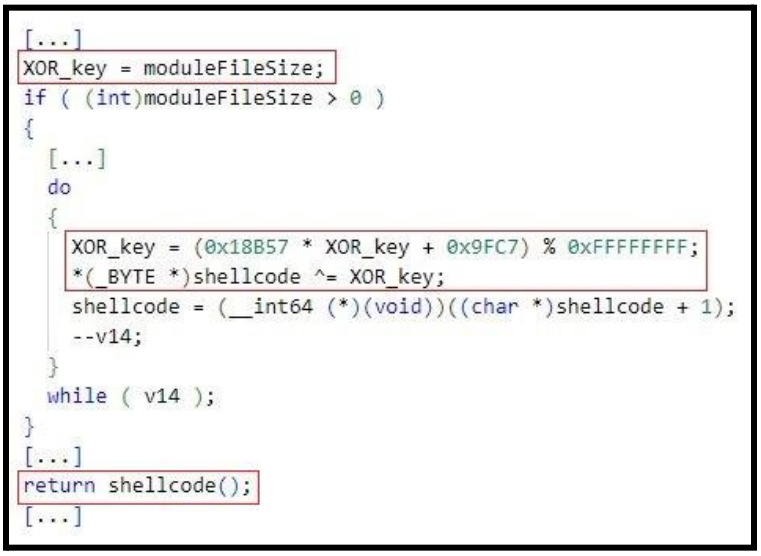
*Entries from a metaMain execution log*

metaMain creates a unique identification number for each infected system, which the implant internally refers to as "network ID" or "CSRF" (possibly an acronym for 'computer system reference'). The network ID is a derivative of the infected platform's computer name and the user privileges with which metaMain executes –whether SYSTEM or regular user privileges. metaMain uses the network ID as a reference for the infected system when communicating with the C2 server and records it in the execution log.

### Meta mode

In the meta execution mode, the implant acts as a loader of operator-provided shellcode stored as a file on the victim's system. metaMain internally refers to this file as the '*metatsploit* file', which is probably where the name of the meta execution mode comes from. Note the extra 't' deviating from the popular metasploit pentesting tool.

To execute in meta mode, the operator starts metaMain by specifying the -module <path> command line parameter. If the <path> doesn't point to a module file, metaMain attempts to execute a default module from the metaMain working directory– `fcache03.db`. The content of a metaMain module file is encrypted via XOR, where the XOR key is a derivative of the size of the module file.



```
[...]
XOR_key = moduleFileSize;
if ( (int)moduleFileSize > 0 )
{
  [...]
  do
  {
    XOR_key = (0x18B57 * XOR_key + 0x9FC7) % 0xFFFFFFFF;
    *(_BYTE *)shellcode ^= XOR_key;
    shellcode = (__int64 (*)(void))((char *)shellcode + 1);
    --v14;
  }
  while ( v14 );
}
[...]
return shellcode();
[...]
```

*metaMain decrypts the content of a module file*

metaMain first decrypts the content of the specified module file, then executes the decrypted file content (i.e., the shellcode), and terminates its own execution.

## Main mode

In the Main execution mode, metaMain acts as a backdoor and executes commands based on operator input. In addition, if metaMain executes with regular user privileges, metaMain logs keyboard and mouse events, and takes screenshots. To execute in the Main execution mode, the operator starts metaMain by specifying the '-chipset' command line parameter.

In main mode, metaMain takes the following actions:

- Acquires a mutex named `mutex_env_%USERNAME%`, where %USERNAME% is the username in whose context metaMain executes.
- Reads, decrypts, and applies the configuration data in the RC4-encrypted `fcache02.db` file from its working directory. The parameters from the configuration file are detailed in the technical appendix. It primarily sets the parameters for metaMain's C2 communications.
- If metaMain executes with regular user privileges, it starts a thread to log keyboard and mouse events, and to take screenshots. metaMain captures low level keyboard and mouse events (WH_KEYBOARD_LL and WH_MOUSE_LL) using a message queue. The implant stores the logged events in an internal memory region and on the file system (in the fcache01.db file that resides in metaMain's working directory). Every 5 seconds, metaMain:
    - Flushes logged keyboard and mouse events from the internal memory region to the fcache01.db file.
    - Takes a screenshot of the entire screen every SCREENSHOT_DELAY_SECS seconds and stores it in the fcache04.db file. SCREENSHOT_DELAY_SECS is a metaMain configuration variable. fcache04.db resides in metaMain's working directory.
- metaMain starts its backdoor capability, which includes executing commands based on operator input and uploading the following data to the C2:
    - Screenshots stored in the fcache04.db file. metaMain deletes fcache04.db after uploading its content.
    - The execution log of metaMain as well as recorded keyboard and mouse events. These are stored in the fcache00.db and fcache01.db files that reside in metaMain's working directory.

Every 24 hours, metaMain flushes the fcache00.db and fcache01.db and then appends them with data from the execution log and captured events stored in metaMain's memory regions. The contents of the caches are then uploaded to the C2 and the local files deleted.

The execution log and logged keyboard and mouse events are encrypted before being written to fcache00.db and fcache01.db. metaMain uses XOR-based encryption such that the XOR key for encrypting each character in the file is a derivative of the character's file position.

```
[...]
file_position_1 = file_position;
if ( data_to_encrypt[1] != *data_to_encrypt )
{
  counter = 0i64;
  v5 = -file_position;
  do
  {
    v6 = 0x11 * file_position_1;
    v7 = (file_position_1 * file_position_1) >> (file_position_1 % 0xB);
    ++file_position_1;
    *(_BYTE *)(counter + v2) ^= v6 - 0xF + (_BYTE)v7;
    v2 = *data_to_encrypt;
    counter = v5 + file_position_1;
    dataSize = data_to_encrypt[1] - *data_to_encrypt;
  }
  while ( counter < dataSize );
}
[...]
```

metaMain encrypts the content of fcache00/01.db

## Working Directory and File System Artifacts

metaMain uses a working directory from which the implant reads files, or where it stores files, during operation. When started, metaMain sets its working directory based on the user privileges with which metaMain executes:

- *%SystemRoot%\System32* if metaMain executes with *SYSTEM* user privileges (*System*).
- *%LOCALAPPDATA%\fontcache* if metaMain executes with regular user privileges (*User*).

The files that reside in metaMain's working directory follow the *fcacheNN.db* naming convention, where *N* is a digit. Some of the files that reside in metaMain's working directory and that metaMain may interact with are:

- *fcache00.db*: This file stores the execution log of metaMain.
- *fcache01.db*: This file stores the keyboard and mouse events that metaMain has logged.
- *fcache02.db*: This file stores metaMain configuration information.
- *fcache03.db*: This file stores shellcode (the implementation of a metaMain module) that metaMain executes if the operator has not specified a file system path to a module file when instructing metaMain to execute a module. metaMain has the capability to execute operator-provided modules implemented in files that the operator has placed on the victim's filesystem.
- *fcache04.db*: This file stores screenshots that metaMain has taken.

In addition, if metaMain executes with *SYSTEM* user privileges, metaMain changes the *CreationTime*, *LastAccessTime*, and *LastWriteTime* file time attributes of metaMain-related files that may reside in the metaMain's working directory (*%SystemRoot%\System32*) to the values of these file attributes of the *%SystemRoot%\System32\ansi.sys or %SystemRoot%\System32\chkdsk.exe* file. This is to reduce the chance of defenders detecting the metaMain-related files in the *%SystemRoot%\System32* directory based on an anomaly in file time attributes. metaMain changes changes the *CreationTime*, *LastAccessTime*, and *LastWriteTime* file time attributes of the following files: *cdb.exe*, *cdb.ini*, *hkcmd.exe*, *hccutils.dll*, *hkcmd.db*, *fcache02.db*, *fcache03.db*, *fcache07.db*, *fcache08.db*, *Speech02.db*, and *Speech03.db*.

## C2 Server Interaction

When it operates in *Main* mode, metaMain downloads data from the C2 server in a continuous loop, which, if available, instructs it to execute backdoor commands.

metaMain supports three different ways of connecting to, and exchanging data with, the C2 server, which we refer to as *C2 communication methods*:

- *TCP KNOCK*: metaMain establishes an indirect and raw TCP socket-based connection to the C2 server. metaMain establishes a direct and raw TCP socket-based connection to another implant, which metaMain internally refers to as *Cryshell*. The role of Cryshell is to act as an intermediary between metaMain and the C2 server, forwarding to the C2 server any data that originates from metaMain and is designated for the C2 server. metaMain authenticates itself to Cryshell through a port knocking and handshaking procedure. Same as metaMain, the Mafalda implant can also communicate to the C2 server through Cryshell. We discuss the port knocking procedure that the Mafalda uses to authenticate itself to Cryshell in greater detail in the Additional Implants section. The port knocking procedure with which metaMain authenticates itself to Cryshell is similar to the one that Mafalda conducts.
- *HTTP*: metaMain establishes an HTTP (Hypertext Transfer Protocol)-based connection to the C2 server. The IP address of the C2 server is stored in the *SERVER_ADDR* configuration variable of metaMain. To send data to the C2 server, metaMain issues the HTTP POST method to the *weathertoday1* resource at the C2 server. To receive data from the C2 server, Mafalda issues the HTTP GET method to request the same resource from the C2 server.
- *NAMED PIPE*: metaMain establishes a named pipe-based connection to the C2 server, where the named pipe is named \\.*pipe**DATABASE01*. To send data to the C2 server, metaMain executes the WriteFile function. To receive data from the C2 server, metaMain executes the ReadFile function.

The configured C2 communication method of metaMain is stored in the *COMMUNICATION_TYPE* configuration variable of metaMain.

metaMain encrypts, or decrypts, the data that it sends to, or receives from, the C2 server using the RC4 encryption algorithm.

## Configuration File

When metaMain executes in 'Main Mode', it attempts to read a configuration file stored under 'fcache02.db'. In absence of that config file, the implant will terminate. The configuration file dictates metaMain's internal operations. The fields expected in the configuration file are as follows:

| Parameter | Details |
|---|---|
| SERVER_ADDR | HTTP C2 Server Address |
| PIPECLIENT_SERVER | Pipe C2 Server Adress |
| PIPECLIENT_USERNAME | Pipe C2 Server Username |
| PIPECLIENT_PASSWORD | Pipe C2 Server Password |
| COMMUNICATION_TYPE | C2 Communication scheme |
| SLEEP_INTERVAL_MINS | Sleep interval in minutes |
| RETRY_DELAY_SECS | Delay between C2 communication retries |
| SCREENSHOT_DELAY_SECS | Delay between screenshots |
| KNOCK_HOP1_IP | TCP knock relay IP |
| KNOCK_FINALIP | TCP knock final destination |
| KNOCK_HOP1_KNOCKBASE | TCP knock first port of sequence |
| KNOCK_HOP1_LISTENPORT | TCP knock opened up port |

| | |
|---|---|
| KNOCK_FINALPORT | TCP known final port |
| INJECT_KEYLOGGER_PROCESS | Process to Inject KL component to |

## metaMain Commands

| | |
|---|---|
| CONFIG_UPDATE *[configuration]* | Reconfigures metaMain based on operator-provided configuration data (*[configuration]*). Stores the configuration data in the *fcache02.db* file, in metaMain's working directory.<br><br>The *[configuration]* parameter is mandatory. |
| PS | Enumerates the processes that run on the platform where metaMain operates and uploads a list of the enumerated processes to the C2 server. |
| META *[path]* | Starts a new metaMain instance in *Meta* mode such that *[path]* is an operator-provided file system path to a module file on the platform where metaMain operates.<br><br>The *[path]* parameter is optional. |
| CRASH | Terminates the execution of metaMain ungracefully. |
| RESTART | Restarts the execution of metaMain. |
| EXIT | Terminates the execution of metaMain gracefully. |
| FLUSH | Updates *fcache00.db* and *fcache01.db* by flushing and appending to the files any execution log as well as |

| | |
|---|---|
| | captured keyboard and mouse events from the metaMain's memory regions where the implant stores these. Enumerates the processes that run on the platform where metaMain operates. Uploads the content of *fcache01.db* and *fcache00.db*, and a list of the enumerated processes to the C2 server. Deletes *fcache00.db* and *fcache01.db*. |
| WINEXEC *[command]* | Executes an operator-provided Windows command (*[command]*) by using the WinExec function.<br><br>The *[command]* parameter is mandatory. |
| DIR *[path] [timestamp]* | Recursively enumerates the files in an operator-provided directory, such that *[path]* is a file system path to a directory on the platform where metaMain operates. Uploads the enumeration results to the C2 server.<br><br>The *[path]* parameter is mandatory.<br><br>If the operator has specified a timestamp in the *[timestamp]* parameter (in *year/month/date* format), enumerates only the files that have been modified later than, or at, the timestamp.<br><br>The *[timestamp]* parameter is optional. |
| NETUSE *[resource] [username] [password]* | Connects to an operator-provided network resource (*[resource]*) using an operator-provided username |

| | |
|---|---|
| | ([*username*]) and password ([*password*]). Uses the WNetAddConnection2W function to connect to the resource.<br><br>The *[resource]*, *[username]*, and *[password]* parameters are mandatory. |
| NETUSEDEL *[resource]* | Disconnects from an operator-provided network resource ([*resource*]). Uses the WNetCancelConnection2W function to connect to the resource.<br><br>The *[resource]* parameter is mandatory. |
| UP *[path]* | Uploads to the C2 server a file such that *[path]* is the operator-provided file system path to the file on the platform where metaMain operates.<br><br>The *[path]* parameter is mandatory. |
| DOWN *[resource] [path]* | Downloads an operator-provided resource ([*resource*]) from the C2 server and stores the resource at an operator-provided file system path ([*path*]) on the platform where metaMain operates.<br><br>The *[resource]* and *[path]* parameters are mandatory. |

# Mafalda


## Mafalda Clear Build 144

As with metaMain, Mafalda keeps an execution log that records events that pertain to the implant's operation. Mafalda variants store the execution log in an internal memory region.

```
[-] VirtualFree 1 failed Error: 487 Attempt to access invalid address.
[-] VirtualFree 2 failed Error: 87 The parameter is incorrect.
Obfuscated thread creation disabled (todo!)
added vectored exception handler
Obfuscated thread creation disabled (todo!)
load bin file C:\Windows\system32\ntdll.dll
ntdll hash: E004197EB89EEAA0B097C9F4DA2F9A9E
load bin file C:\Windows\system32\msv1_0.dll
msv1_0.dll hash: ED2D037C307D2BE5C526E8DF48299FCD
http_connect_to_c2( 5.2.77.52 )
[-] HttpSendRequest: Error: 12029
GET failed, retrying <- Failed C2 connection
```

*Execution log showing an attempt to connect to a C2 at 5.2.77.52*

Mafalda uses obfuscated strings for various purposes, such as to dynamically resolve library function addresses or to store content in the execution log. The older variant obfuscates strings by splitting the strings into multiple portions, with a maximum portion length of 9 characters, and encoding each portion.

To restore an obfuscated string into a valid string, Mafalda first decodes each of the string's portions, and then concatenates the string portions together. A portion of an obfuscated string is encoded using the bitmask 0x7F.

```
[...]
for ( i = 0; i < 9; ++i )
{
  v4[i] = a2 & 0x7F;
  a2 >>= 7;
}
[...]
```

Mafalda string decoding loop

When started, Mafalda creates a thread designated for keyboard and mouse event logging, capturing low level keyboard and mouse events (WH_KEYBOARD_LL and WH_MOUSE_LL) with a message queue.

Mafalda stores captured keyboard and mouse events in an internal memory region. In our analysis environment, we typed the words "testbackdoor" and "keylogger" to test how the keylogging function worked.

```
0:006> du 0x291234a2530
00000291`234a2530  "....@@@@[ 08/30/2022 12:28:30 C:"
00000291`234a2570  "\Windows\System32\rundll32.exe C"
00000291`234a25b0  ":\Users\<user>\Desktop\Metadoor\m"
00000291`234a25f0  "u.dll, #1 - WinDbg 1.2202.7001.0"
00000291`234a2630  "]..[M].t.e.s.t.b.a.c.k.d.o.o.r.["
00000291`234a2670  "M].k.e.y.l.o.g.g.e.r.[M]."
```

*Keyboard and mouse events [M]) captured by Mafalda*

```
[...]
InitializeCriticalSection(&CriticalSection);
if ( (unsigned int)get_processSessionID_sub_1801125F0() )
{
  set_keylog_mouse_hooks_sub_1800B3DF0();
  while ( GetMessageW(&Msg, 0i64, 0, 0) > 0 )
  {
    TranslateMessage(&Msg);
    DispatchMessageW(&Msg);
  }
  return 0i64;
}
else
{
[...]
```

## Mafalda Command and Control

### *Data Management*

Mafalda stores the data designated for the C2 server in a data structure, which we refer to as a *packet*. The packets may contain any data that Mafalda may send to the C2 server, such as:

- The execution log.
- Captured keyboard and mouse events.
- Output of any backdoor commands that Mafalda has executed.

Mafalda manages outgoing packets as a queue, which we refer to as the *outgoing packet pipeline*. Mafalda initializes the outgoing packet pipeline after creating the thread for keyboard and mouse event logging.

```
[...]
while ( 1 )
 {
   std::list<SteamItemDetails_t>::clear((std::list<SteamItemDetails_t> *)packet_pipeline_qword_180232290);
[...]
```

Mafalda initializes the outgoing packet pipeline (IDA Pro pseudocode, trimmed for brevity, packet_list_qword_180232290 labels the packet pipeline)

Mafalda iterates the outgoing packet pipeline and sends all packets in the pipeline to the C2 server:

- After establishing a connection to the C2 server – Mafalda sends logged keyboard and mouse events, and the execution log, which the implant has previously stored in the outgoing packet pipeline.
- After executing a backdoor command – Mafalda sends logged keyboard and mouse events, the execution log, and the output of the command.

Outgoing packets may have names that are descriptive of the packets' content. For example, the packet with the name *keylog\keylog_%time%.txt* is the name of the packet that stores the logged keyboard and mouse events (*%time%* is the local system time when Mafalda sends the packet).

Mafalda also receives packets from the C2 server, which Mafalda processes as they arrive.

*C2 Server Interaction*

Mafalda supports four different ways of connecting to, and exchanging data with, the C2 server, which we refer to as *C2 communication methods*:

- *TCP RAW*: Mafalda establishes a direct and raw TCP (Transmission Control Protocol) socket-based connection to the IP address and the TCP port of the C2 server. The IP address and the TCP port are part of Mafalda's configuration space. Mafalda authenticates itself to the C2 server through a handshake procedure. Mafalda encrypts, or decrypts, the data that it sends to, or receives from, the C2 server using the RC4 encryption algorithm.
- *TCP KNOCK*: Mafalda establishes an indirect and raw TCP socket-based connection to the C2 server. Same as metaMain, Mafalda establishes a direct and raw TCP socket-based connection to another implant, which Mafalda internally refers to as *Cryshell*. The role of Cryshell is to act as an intermediary between Mafalda and the C2 server, forwarding to the C2 server any data that originates from Mafalda and is designated for the C2 server. Mafalda authenticates itself to Cryshell through a port knocking and handshaking procedure. We discuss the port knocking procedure that Mafalda uses to authenticate itself to Cryshell in greater detail in the Additional Implants section. Mafalda encrypts, or decrypts, the data that it sends to, or receives from, the C2 server using the RC4 encryption algorithm.

- *HTTP*: Mafalda establishes an HTTP (Hypertext Transfer Protocol)-based connection to the C2 server. The IP address and the HTTP port of the C2 server are part of Mafalda's configuration space. Mafalda issues the HTTP GET method to request the *ticker* resource from the C2 server. If the server responds with *ok*, Mafalda considers the connection to the C2 server established. If the server responds with *nada*, Mafalda considers the connection to the C2 server failed. To send data to the C2 server, Mafalda encodes the data to Base64 format and issues the HTTP POST method to the *cdn* resource at the C2 server. To receive data from the C2 server, Mafalda issues the HTTP GET method to request the *cnd* resource from the C2 server and decodes the received data from the Base64 format.

- *NAMED PIPE*: Mafalda creates a named pipe to listen for incoming commands from a named pipe-based C2 server. Mafalda attempts to create a named pipe up to 10 times if creation attempts fail, with named pipe names of \\.*pipe**DOMAIN%ID*, where *%ID* is an integer value between 0 and 9. To send data to the C2 server, Mafalda executes the WriteFile function. To receive data from the C2 server, Mafalda executes the ReadFile function.

When connecting to the C2 server using the TCP RAW, or the TCP KNOCK, communication method, Mafalda authenticates itself to the C2 server, or Cryshell, through a handshake procedure:

1. Mafalda generates a random 16 byte value, sends the value to the C2 server, or Cryshell, and receives another 16 byte value back. Mafalda and the C2 server, or Cryshell, use these values to initialize RC4 contexts for exchanging RC4-encrypted data.
2. Mafalda sends RC4-encrypted data to the C2 server, or Cryshell, and receives RC4-encrypted data back. If the data that Mafalda has sent is equal to the data that Mafalda has received, the handshake procedure is considered complete.

If Mafalda fails to establish a connection to the C2 server, the implant sleeps for an amount of time, initializes the outgoing packet pipeline, and attempts to reconnect. Mafalda increases double the amount of time during which the implant sleeps between reconnection attempts until it exceeds 3 hours. Mafalda then terminates if the name of the process in whose context the implant runs is *resmon.exe* or *defrag.exe*.

In our investigations, we observed multiple instances of Mafalda that the Metador threat actor has deployed in the victims' environments, with different C2 communication methods configured:

| File name | C2 Communication Method |
|---|---|
| fcache11.db | TCP RAW |
| fcache13.db | HTTP |
| fcache14.db | NAMED PIPE |

Mafalda instances

If Mafalda successfully establishes a connection to the C2 server, Mafalda builds and sends a packet to the C2 server, which we refer to as the *initial packet*. Among other things, the initial packet contains:

- Information about the platform and execution environment where Mafalda runs, for example, *DESKTOP-FA41KBG\testuser.DESKTOP-FA41KBG.rundll32.exe.4508.UAC:Medium.OS: Windows 10 build 17763.Ses:1.HTTP*, where *DESKTOP-FA41KBG* is the name of the computer where Mafalda runs, *testuser* is the user in whose context Mafalda runs, and *rundll32.exe* and *4508* is the name and the ID of the process that hosts Mafalda.
- A JSON-formatted string that documents in detail the backdoor commands that Mafalda supports and the commands' parameters. Therefore, this string represents a very rich information source to malware analysts.
  The older Mafalda variant supports 54 backdoor commands. Each command that Mafalda supports is associated with a command identification number and a command name. In the Appendix, in the Mafalda Commands section, we list the command names and descriptions that we extracted from Mafalda, that is, from the JSON-formatted string.
  The instructional nature of the backdoor command descriptions stored in the JSON-formatted string and the presence of the word *operator* in these descriptions is an indication that Mafalda is developed by an actor separate from the Mafalda operator (the Metador threat actor).

```
05{
    "name": "download",
    "help": "download file from target computer onto operator machine",
    "params": [
        {
            "name": "filename",
            "help": "filename to download (windows wildcards like *.doc are accepted, but there is no recursive downloading yet)",
            "mandatory": "true",
            "type": "string"
        },
        {
            "name": "--ignore_filesize",
            "help": "ignore the file size limit",
            "mandatory": "false",
            "type": "justflag"
        }
    ]
}
06{
    "name": "upload",
    "help": "upload file from operator to target machine",
    "params": [
        {
            "name": "localfile",
            "help": "local file to upload",
            "mandatory": "true",
            "type": "filefromclient"
        },
        {
            "name": "remotefilename",
            "help": "filename on remote machine",
            "mandatory": "false",
            "type": "string"
        }
    ]
}
```


JSON-formatted string that documents backdoor commands (trimmed for brevity)

*Packet Processing*

After sending the initial packet, Mafalda continues to execute in a loop of sending outgoing packets to, and receiving packets from, the C2 server.

Each packet that Mafalda receives is of a given type and subtype. The packet type and subtype are uniquely identified by identification numbers, which the authors of Mafalda refer to as *outer OPC* and *inner OPC*, respectively:

- The packet of type 0x71 has no impact on the operation of Mafalda.

- The packet of type 0x72 instructs Mafalda to exit the loop of sending outgoing packets to, and receiving packets from, the C2 server, and reconnect to the C2 server after a sleep period.
- The packet of type 0x73 instructs Mafalda that the packet has a subtype:
    - The packet of subtype 0x81 or 0x82 instructs Mafalda to execute the backdoor command with the command identification number stored in the packet.
    - The packet of any other subtype instructs Mafalda to exit the loop of sending outgoing packets to, and receiving packets from, the C2 server, and reconnect to the C2 server after a sleep period.

The functionalities of the backdoor commands that Mafalda implements have a very broad scope and include data and information theft (such as downloading files and taking screenshots), command execution, system registry and file system manipulation, credential theft, and Mafalda reconfiguration.

For example, the *change_c2* command changes the configuration of Mafalda that relates to the C2 server – the Mafalda operator can change, for instance, the server's IP address and TCP port number, as well as the configured C2 communication method. In addition, the *connect_named_pipe* command connects through a named pipe a Mafalda instance, which operates using the *TCP RAW* C2 communication method, to another Mafalda instance that operates using the *NAMED PIPE* C2 communication method. This turns the former Mafalda instance into an entity that forwards communication with the C2 server for the latter Mafalda instance. The *disconnect_named_pipe* command shuts down the named pipe connection between the two Mafalda instances. We depict below the descriptions of the *change_c2*, *connect_named_pipe*, and *disconnect_named_pipe* commands as displayed by *Mafalda simulator*. *Mafalda simulator* is a tool that we developed to pretty-print the documentation on the backdoor commands that Mafalda supports and the commands' parameters that we extracted from Mafalda.

```
MAFALDA_SIMULATOR > change_c2 --help
Changes C2 server (this will NOT be saved anywhere but just affect this implant in-memory)
            --type                          tcp (straight TCP connection), knock (bouncing tcp connection over cryshell implant),
                                            http (HTTP )
            --tcp_ip                        IP Address (or DNS name) of new C2 server when using TCP
            --http_domain                   IP Address (or DNS name) of new C2 server when using HTTP
            --http_port                     Port for HTTP server
            --tcp_port TCP port
            --knock_hop1_ip                 IP Address (or DNS name) of cryshell implant
            --knock_hop1_knockbase          first port of knock sequence
            --knock_hop1_listenport         tcp port that gets opened up by cryshell implant
            --knock_finalip                 IPv4 address of C2 server
            --knock_finalport               TCP port of C2 server
```

The *change_c2* command (Mafalda simulator view)

```
MAFALDA_SIMULATOR > connect_named_pipe --help
Connects to another Mafalda instance that is listening on a named pipe. For now this works only if you are connected via TCP.
HINT: maybe connect_share first!
            remote_address      e.g. 192.168.0.12 or DOMAINCONTROLLER1

MAFALDA_SIMULATOR > disconnect_named_pipe --help
Disconnects named pipe forwarding (no matter if you are the forwarder or the forwardee)
```

The *connect_named_pipe* command (Mafalda simulator view)

## Mafalda Obfuscated Build

The newer variant of Mafalda is an extension of the older variant, with two major differences:

- The implementation and certain operational aspects of the newer Mafalda variant are obfuscated to make analysis challenging.
- The newer Mafalda variant provides additional backdoor commands to implant operators.

This section discusses the differences between the older and the newer Mafalda variant in greater detail.

### Obfuscations Overview

In this section, we provide an overview of the major obfuscation measures that the developers of the newer Mafalda variant have implemented.

*Execution Flow Obfuscation*

The newer Mafalda variant is obfuscated at implementation-level such that the compiled code of the implant consists mainly of:

- Obfuscated code segments.
- Non-obfuscated code segments, the majority of which are functions that implement Mafalda functionalities.

In most cases, the obfuscated code segments start with thunk functions – functions that implement only a single *JMP* instruction that directs execution to a destination location. The thunk functions direct the execution of Mafalda to obfuscated code segments. Such a segment ultimately returns execution to a destination location in the memory mapped to Mafalda that is in the relative vicinity of the thunk function that has directed execution to the segment. This destination location is a non-obfuscated code segment — often the prologue of a function that implements Mafalda functionalities. In summary, the obfuscated code segments effectively obfuscate the invocation of non-obfuscated functions, which makes static analysis a challenging process.

The figure below depicts an instance of execution flow obfuscation through thunk functions that the newer Mafalda variant features. The thunk function *entryRoutine* directs execution to the location *entryRoutine_0*. *entryRoutine_0* marks the beginning of an obfuscated code segment. This code segment ultimately returns the execution to a non-obfuscated code segment – the prologue of the function *sub_17808D17767*.

```
debug019:0000017808D17758                               ; Attributes: thunk
debug019:0000017808D17758
debug019:0000017808D17758                               ; void entryRoutine()
debug019:0000017808D17758                               public entryRoutine
debug019:0000017808D17758                               entryRoutine proc near
debug019:0000017808D17758 000 E9 DE 5E 10 00            jmp     entryRoutine_0
debug019:0000017808D17758                               entryRoutine endp
debug019:0000017808D17758
debug019:0000017808D1775D                               ; ---------------------------------------------------------------------------
debug019:0000017808D1775D 9B                            wait
debug019:0000017808D1775E 85 54 17 D3                   test    [rdi+rdx-2Dh], edx
debug019:0000017808D17762 39 D6                         cmp     esi, edx
debug019:0000017808D17764 04 63                         add     al, 63h ; 'c'
debug019:0000017808D17764                               ; ---------------------------------------------------------------------------
debug019:0000017808D17766 53                            db  53h ; S
debug019:0000017808D17767
debug019:0000017808D17767                               ; =============== S U B R O U T I N E =======================================
debug019:0000017808D17767
debug019:0000017808D17767
debug019:0000017808D17767                               ; __int64 __fastcall sub_17808D17767(__int64, __int64, __int64)
debug019:0000017808D17767                               sub_17808D17767 proc near
debug019:0000017808D17767
debug019:0000017808D17767                               var_3A8= qword ptr -3A8h
debug019:0000017808D17767                               var_398= qword ptr -398h
debug019:0000017808D17767                               var_390= qword ptr -390h
debug019:0000017808D17767                               var_388= qword ptr -388h
debug019:0000017808D17767                               var_378= qword ptr -378h
debug019:0000017808D17767                               var_370= qword ptr -370h
debug019:0000017808D17767                               var_368= qword ptr -368h
debug019:0000017808D17767                               var_358= qword ptr -358h
debug019:0000017808D17767                               var_350= qword ptr -350h
debug019:0000017808D17767                               var_28= byte ptr -28h
debug019:0000017808D17767
debug019:0000017808D17767 000 55                        push    rbp
debug019:0000017808D17768 008 41 54                     push    r12
debug019:0000017808D1776A 010 41 55                     push    r13
debug019:0000017808D1776C 018 41 56                     push    r14
debug019:0000017808D1776E 020 41 57                     push    r15
debug019:0000017808D17770 028 48 8D A8 38 FD FF FF      lea     rbp, [rax-2C8h]
debug019:0000017808D17777 028 48 81 EC A0 03 00 00      sub     rsp, 3A0h
                                                        [...]
```

Execution flow obfuscation through a thunk function (IDA Pro disassembly view, trimmed for brevity)

Some of the obfuscation techniques that the developers of Mafalda have applied to the obfuscated code segments include:

- Opaque predicates, for example, comparing the value in a register against itself, which always evaluates to *TRUE*.
- Use of a single or multiple instructions that have no impact on Mafalda's execution or data, such as left or right rotation of a register value for 0 bytes, exchanging twice the values in two registers, or first pushing and then immediately removing from the stack a given value.
- Use of multiple, instead of one, JMP instructions (trampolines) to direct execution to a destination location.

- Conditional execution based on a flag value in the *RFLAGS* register, for example, the zero flag (*ZF*) or the parity flag (*PF*), such that any of the possible flag values (0 or 1) result in the execution of the code at a given location in the memory mapped to Mafalda.

```
[...]
debug025:0000017808E1D63B 66 93                   xchg    ax, bx
debug025:0000017808E1D63D 48 C1 C6 00             rol     rsi, 0
debug025:0000017808E1D641 66 93                   xchg    ax, bx
debug025:0000017808E1D643 48 8B C4                mov     rax, rsp
debug025:0000017808E1D646 74 0C                   jz      short loc_17808E1D654
debug025:0000017808E1D648 7A 03                   jp      short loc_17808E1D64D
debug025:0000017808E1D64A 7B 01                   jnp     short loc_17808E1D64D
debug025:0000017808E1D64C 59                      pop     rcx
debug025:0000017808E1D64D
debug025:0000017808E1D64D                         loc_17808E1D64D:
debug025:0000017808E1D64D
debug025:0000017808E1D64D EB 02                   jmp     short loc_17808E1D651
debug025:0000017808E1D64D                         ; ---------------------------
debug025:0000017808E1D64F 9D                      db 9Dh
debug025:0000017808E1D650 E3                      db 0E3h
debug025:0000017808E1D651                         ; ---------------------------
debug025:0000017808E1D651
debug025:0000017808E1D651                         loc_17808E1D651:
debug025:0000017808E1D651 75 03                   jnz     short loc_17808E1D656
debug025:0000017808E1D653 95                      xchg    eax, ebp
debug025:0000017808E1D654
debug025:0000017808E1D654                         loc_17808E1D654:
debug025:0000017808E1D654 F3 90                   pause
debug025:0000017808E1D656
debug025:0000017808E1D656                         loc_17808E1D656:
[...]
```

Obfuscated code segment (IDA Pro disassembly view, trimmed for brevity)

*String and Function Parameter Obfuscation*

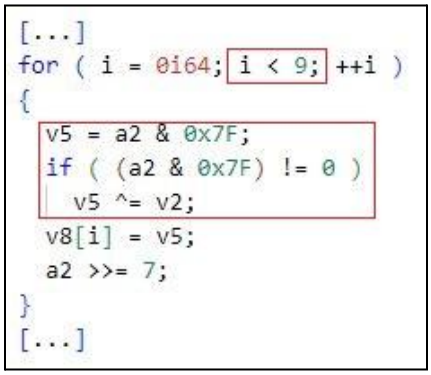
Same as the older Mafalda variant, the newer Mafalda variant often uses obfuscated strings for different purposes, for example, to dynamically resolve library function addresses through library and library export names, or to store content in the execution log. The newer Mafalda variant obfuscates strings by:

- Splitting the strings into multiple portions, with a maximum portion length of 9 characters.

- Encrypting and encoding each string portion.

Therefore, to restore an obfuscated string into a valid string, Mafalda first decodes and decrypts each of the string's portions, and then concatenates the string portions together.

A portion of an obfuscated string is encoded using the bitmask *0x7F* and XOR-encrypted using a portion-specific XOR key of one byte. The figure below depicts a snippet of the function that Mafalda executes to decoding and decrypt a portion of an obfuscated string.



```
[...]
for ( i = 0i64; i < 9; ++i )
{
  v5 = a2 & 0x7F;
  if ( (a2 & 0x7F) != 0 )
    v5 ^= v2;
  v8[i] = v5;
  a2 >>= 7;
}
[...]
```

Mafalda decodes and decrypts a string (newer Mafalda variant, IDA Pro pseudocode, trimmed for brevity, a2 is a portion of an obfuscated string, v2 is the XOR key for the string portion)

To further make analysis challenging, Mafalda often obfuscates numerical function parameters by calculating parameter values prior to function execution using arithmetics and bitwise operations.

Mafalda may also first compute a value using arithmetics and bitwise operations, and if the computed value does or does not match a predefined value, Mafalda assigns the correct values to the obfuscated parameters. The alternative branch assigns wrong values to the obfuscated parameters and exists to confuse analysis tools. Mafalda applies this obfuscation approach when it executes the function that the implant uses to decode and

decrypt obfuscated string portions (labeled *j_str_resolve_sub_18014FE4D* in the figure below).

```
[...]
if ( ((HIDWORD(qword_1802DE238) + dword_1802DE150) ^ dword_1802DE0D4 ^ dword_1802DE1E0)
   + (dword_1802DE134 ^ dword_1802DE224)
   + dword_1802DE154
   + dword_1802DE244 == 0x1BC8A )
{
  v51 = v47 | 0x80;
  v52 = v3 + 752;
  v53 = 0x2DF9B1CF79AEA2F2i64;
}
else
{
  v51 = v47 | 0x40;
  v52 = v3 + 784;
  v53 = 0i64;
}
v54 = j_str_resolve_sub_18014FE4D(v52, v53);
[...]
```

Function parameter obfuscation (IDA Pro pseudocode, trimmed for brevity, v53 is a portion of an obfuscated string)

*String Encryption*

In addition to the string obfuscation approaches that we discussed in the previous section, the newer Mafalda variant works with encrypted versions of string resources that may represent an information source to malware analysts. Such string resources include segments of Mafalda's execution log, debugger messages, and the JSON-formatted string that documents the backdoor commands that Mafalda supports.

For example, in contrast to the execution log of the older Mafalda variant, the execution log of the newer Mafalda variant is encrypted. Given that Mafalda's execution log provides extensive information about the operation of the implant, encrypting the execution log of Mafalda is an effective way to obfuscate the information that the log provides to analysts.

```
0:008> db 00000261`ad72fbb0 L0x300
00000261`ad72fbb0  01 00 76 00 68 00 68 00-6f 00 30 00 47 00 7e 00  ..v.h.h.o.0.G.~.
00000261`ad72fbc0  49 00 63 00 55 00 64 00-6b 00 51 00 58 00 44 00  I.c.U.d.k.Q.X.D.
00000261`ad72fbd0  24 00 4d 00 2d 00 7d 00-43 00 34 00 34 00 21 00  $.M.-.}.C.4.4.!.
00000261`ad72fbe0  73 00 45 00 24 00 7b 00-25 00 6b 00 28 00 48 00  s.E.$.{.%.k.(.H.
00000261`ad72fbf0  77 00 76 00 4c 00 52 00-38 00 2b 00 21 00 48 00  w.v.L.R.8.+.!.H.
00000261`ad72fc00  52 00 77 00 76 00 4c 00-58 00 69 00 3e 00 34 00  R.w.v.L.X.i.>.4.
00000261`ad72fc10  31 00 6d 00 7e 00 54 00-24 00 6f 00 42 00 32 00  1.m.~.T.$.o.B.2.
00000261`ad72fc20  3b 00 53 00 6e 00 7b 00-2c 00 31 00 36 00 3a 00  ;.S.n.{.,.1.6.:.
```

*Encrypted execution log (trimmed for brevity)*

## Mafalda Clear Build 144 Commands

| Command Name | Command Description |
|---|---|
| revert_to_self | Finishes all impersonations and become who you were when this thread started |
| pwd | print working directory |
| cd | change working directory |
| ls | list directory contents |
| download | download file from target computer onto operator machine |
| upload | upload file from operator to target machine |
| screenshot | take screenshot of target machine and save to file |
| shell | execute shell command with cmd.exe |
| exit_implant | terminates implant by killing own process |
| ps | list processes |
| kill | kill process |

| | |
|---|---|
| rm | delete file |
| execute | executes command (does not get output) |
| timestomp | give old file time to file |
| get_privs | use AdjustTokenPrivileges() to give your process additional privileges |
| powershell | execute a powershell command |
| inject_shellcode | Inject shellcode into target process |
| spawn_shellcode | Spawn a new process and inject shellcode into it |
| make_token | Creates a token for a different user |
| get_system | (Metasploit & Cobalt Strike) try to elevate to SYSTEM |
| list_tokens | (Metasploit Incognito) Lists all tokens on machine |
| change_c2 | Changes C2 server (this will NOT be saved anywhere but just affect this implant in-memory) |
| impersonate_token | (Metasploit Incognito) Impersonate a token owned by any process we can access. E.g a fileserver might be a nice place to find tokens :-) |
| connect_share | we connect to a windows share |
| disconnect_share | we disconnect from a windows share |
| uac_info | Gives information about our UAC integrity level and status |
| uac_stealtoken | (Cobalt Strike) Tries to get a UAC-elevated token (so you then can spawn elevated processes) |
| reg_list | Lists a Registry key with all subkeys and values |

| | |
|---|---|
| reg_put_string | Writes a string as REG_SZ to a registry Value |
| reg_put_dword | Writes a value as REG_DWORD to a registry Value |
| hashdump | Dumps password hashes from LSASS.exe. |
| mimi | Run Mimikatz command |
| hidden_mimi_initialize | loads mimi dll into memory |
| psexec | Creates a remote service, lets it run once, then deletes it again |
| reg_del_key | Deletes a Key, all subkeys and all values |
| hidden_set_syscall_nr | sets syscall number (so we can use syscalls instead of NtFunctions) |
| set | set options |
| version | shows build version and build date |
| portfwd_connect | establishes ssh connection from implant to a server (usually where tcpserver.py runs). This does NOT forward any ports yet! |
| portfwd_disconnect | kills the ssh conenction that is used for port forwarding (this also kills ALL port forwards!) |
| portfwd_add | adds a port forward from SSH server into target network |
| portfwd_remove | removes a port forward from SSH server into target network |
| portfwd_status | Shows status of port forwarding |
| hidden_portfwd_initialize | loads dll into memory |
| credential_logger | Patches LSASS.EXE so Windows credentials will be logged to c:\windows\system32\prefetch.dat (this cannot be stopped and will go on until reboot) |

| | |
|---|---|
| cat | types ASCII text file |
| hidden_credential_logger_setparams | Sets parameters for credential logger |
| portscan | UNFINISHED! |
| clear_event_log | Clears the windows event log |
| sleep | disconnect and sleep, the reconnect, e.g. 'sleep 3 h' will sleep for 3 hours |
| run_dotnet | Runs a dotnet library DLL in memory. Requirements: string as input, string as output. |
| connect_named_pipe | Connects to another Mafalda instance that is listening on a named pipe. For now this works only if you are connected via TCP. HINT: maybe connect_share first! |
| disconnect_named_pipe | Disconnects named pipe forwarding (no matter if you are the forwarder or the forwardee) |
| check_edr | Displays information about Antivirus (which this week is called Endpoint Detection and Response (EDR) Systems) |
| list_drives | Lists all HDDs |

## Enumerated Software

| Software | Artifacts |
|---|---|
| Avira | Avkmgr.sys, avipbb.sys, avusbflt.sys, avnetflt.sys, avgntflt.sys, Avira.ServiceHost.exe, Avira.Systray.exe, Avira.OptimizerHost.exe, Avira.VpnService.exe, Avira.SoftwareUpdater.ServiceHost.exe, Avira.Spotlight.Service.exe, avguard.exe, avshadow.exe, protectedservice.exe |

| | |
|---|---|
| FireEye | FeKern.sys, WFP_MRT.sys |
| Raytheon Cyber Solutions | eaw.sys |
| CJSC Returnil Software | rvsavd.sys |
| Verdasys Inc. | dgdmk.sys |
| Altiris (Symantec) | atrsdfw.sys |
| Malwarebytes | mwac.sys, MbamChameleon.sys, farflt.sys, mbamwatchdog.sys, MBAMService.exe, mbamtray.exe, mbam.exe |
| ESET | edevmon.sys, ehdrv.sys |
| SentinelOne | SentinelMonitor.sys |
| BitDefender | edrsensor.sys, hbflt.sys, bdsvm.sys, gzflt.sys, bddevflt.sys, AVCKF.SYS, Atc.sys, AVC3.SYS, TRUFOS.SYS, BDSandBox.sys, bdredline.exe, vsserv.exe, vsservppl.exe, updatesrv.exe, bdagent.exe |
| Hexis Cyber Solutions | HexisFSMonitor.sys |
| Cylance Inc. | CyOptics.sys, CyProtectDrv32.sys, CyProtectDrv64.sys |
| Avast | aswSP.sys, avastsvc.exe, avastui.exe |
| McAfee | mfeaskm.sys, mfencfilter.sys, epdrv.sys, mfencoas.sys, mfehidk.sys, swin.sys, hdlpflt.sys, mfprom.sys, MfeEEFF.sys |
| Dell Secureworks | groundling32.sys, groundling64.sys |
| AVG Technologies | avgtpx86.sys, avgtpx64.sys |
| Symantec | Pgpwdefs.sys, GEProtection.sys, diflt.sys, sysMon.sys, ssrfsf.sys, emxdrv2.sys, reghook.sys, spbbcdrv.sys, bhdrvx86.sys, bhdrvx64.sys, SISIPSFileFilter.sys, symevent.sys, vxfsrep.sys, VirtFile.sys, SymAFR.sys, symefasi.sys, symefa.sys, symefa64.sys, SymHsm.sys, evmf.sys, GEFCMP.sys, VFSEnc.sys, pgpfs.sys, fencry.sys, symrg.sys, NortonSecurity.exe, nsWscSvc.exe, nsWscSvc.exe, ccsvchst.exe, sysidsservice.exe, sysipsservice.exe, sisipsutil.exe, SAFE-Cyberdefense, SAFE-Agent.sys |

| | |
|---|---|
| CyberArk Software | CybKernelTracker.sys |
| Kaspersky | klifks.sys, klifaa.sys, Klifsm.sys |
| Sophos | SAVOnAccess.sys, savonaccess.sys, sld.sys |
| Webroot Software, Inc. | ssfmonm.sys, WRCore.x64.sys, WRkrn.sys, WRSA.exe, WRSkyClient.x64.exe, WRCoreService.x64.exe |
| Carbon Black | CarbonBlackK.sys, carbonblackk.sys |
| Cybereason | CRExecPrev.sys |
| CrowdStrike | Im.sys, CSAgent.sys, CSBoot.sys, CSDeviceControl.sys, cspcm2.sys, |
| Comodo Security Solutions | cfrmd.sys, cmdccav.sys, cmdguard.sys, CmdMnEfs.sys, MyDLPMF.sys |
| Panda Security | PSINPROC.SYS, PSINFILE.SYS, amfsm.sys, amm8660.sys, amm6460.sys |
| F-Secure | fsgk.sys, fsatp.sys, fshs.sys, nif2s64.sys, fsulgk.sys, fshoster32.exe, fsorsp64.exe, fshoster64.exe, fsulprothoster.exe |
| Endgame | esensor.sys |
| Cisco | csacentr.sys, csaenh.sys, csareg.sys, csascr.sys, csaav.sys, csaam.sys |
| Trend Micro Inc | TMUMS.sys, hfileflt.sys, TMUMH.sys, AcDriver.sys, SakFile.sys, SakMFile.sys, fileflt.sys, TmEsFlt.sys, tmevtmgr.sys, TmFileEncDmk.sys, coreFrameworkHost.exe, uiWatchDog.exe, uiWinMgr.exe, TmsaIntance64.exe, AMSPTelemetryService.exe |
| Enigmasoft Spyhunter | shmonitor.exe |
| Check Point Software Technologies | epregflt.sys, medlpflt.sys, dsfa.sys, cposfw.sys |
| Absolute | psepfilter.sys, cve.sys |
| Bromium | brfilter.sys, BrCow_x_x_x_x.sys |
| LogRhythm | LRAgentMF.sys |

| | |
|---|---|
| OPSWAT Inc | libwamf.sys |
| Sysinternals | PROCMON24.SYS, Autoruns.exe, Autoruns64.exe, Dbgview.exe, procexp.exe, procexp64.exe, Procmon.exe, tcpview.exe, sysmon.exe |
| Wireshark | wireshark.exe |
| x64dbg | x64dbg.exe, x32dbg.exe |
| Olly Debugger | ollydbg.exe |
| IDA Pro (WTF?) | ida.exe, ida64.exe |
| Binary Ninja (WTF?) | binaryninja.exe |
| Microsoft WinDbg | windbg.exe |
| VMWare | vmtoolsd.exe, vmhgfs.sys, vmmemctl.sys, vmrawdsk.sys, vmusbmouse.sys, vm3dmp.sys, vmmouse.sys |
| Fellow Hackers | msbuild.exe |