Attackers are abusing MSBuild to evade defenses and implant Cobalt Strike beacons

morphuslabs.com/attackers-are-abusing-msbuild-to-evade-defenses-and-implant-cobalt-strike-beacons-edac4ab84f42

Renato Marinho December 27, 2021

Renato Marinho
Follow
Dec 27, 2021

.

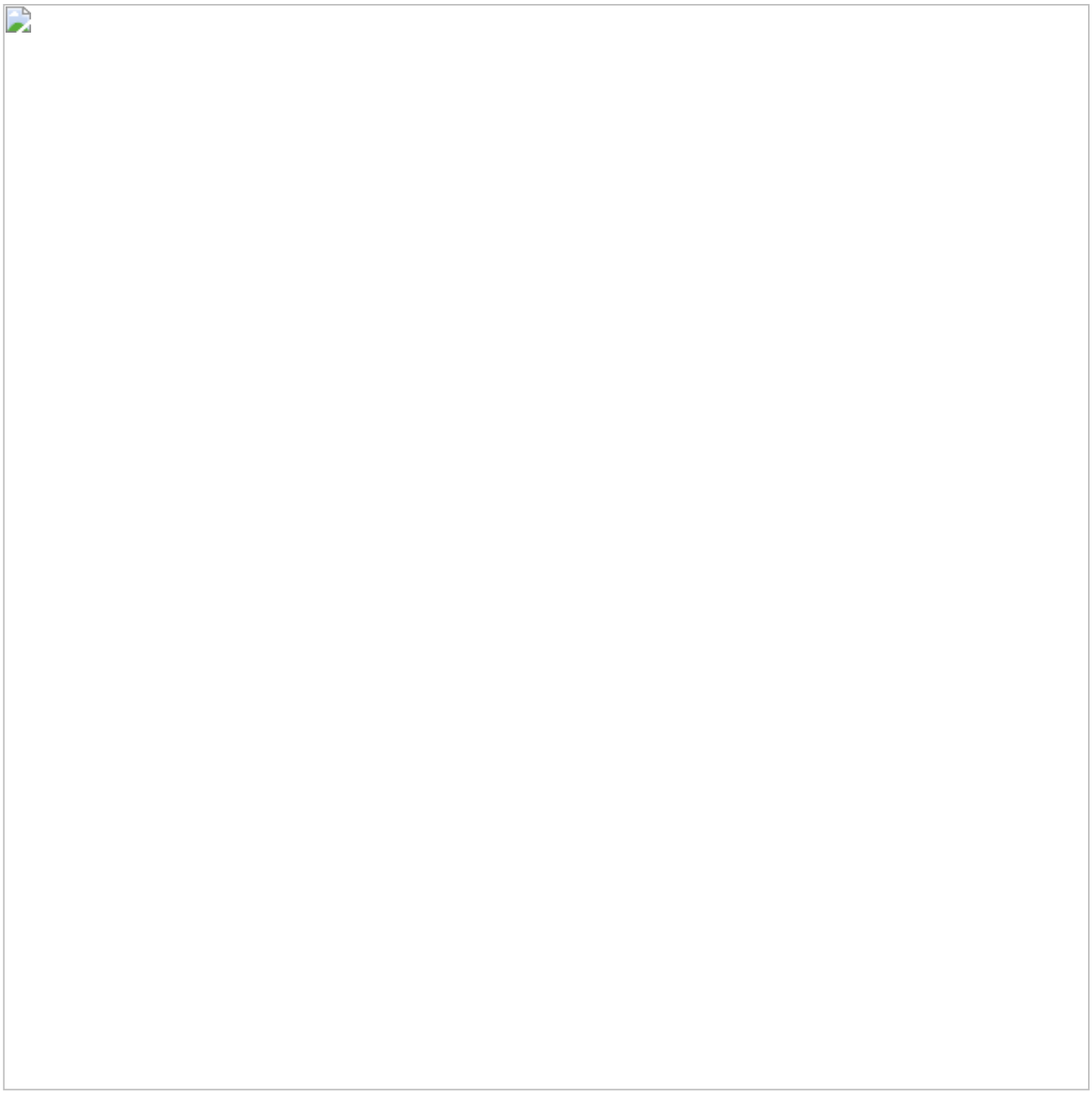
7 min read

Microsoft Build Engine is the platform for building applications on Windows, mainly used in environments where Visual Studio is not installed. Also known as **MSBuild**, the engine provides an XML schema for a project file that controls how the build platform processes and builds software [1]. The project file element named 'Tasks' designates independent **executable components** to run during the project building. Tasks are meant to perform build operations but are **being abused by attackers to run malicious code under the MSBuild disguise**. The technique is mapped on Mitre ATT&CK as "Trusted Developer Utilities Proxy Execution" — T1127.001.

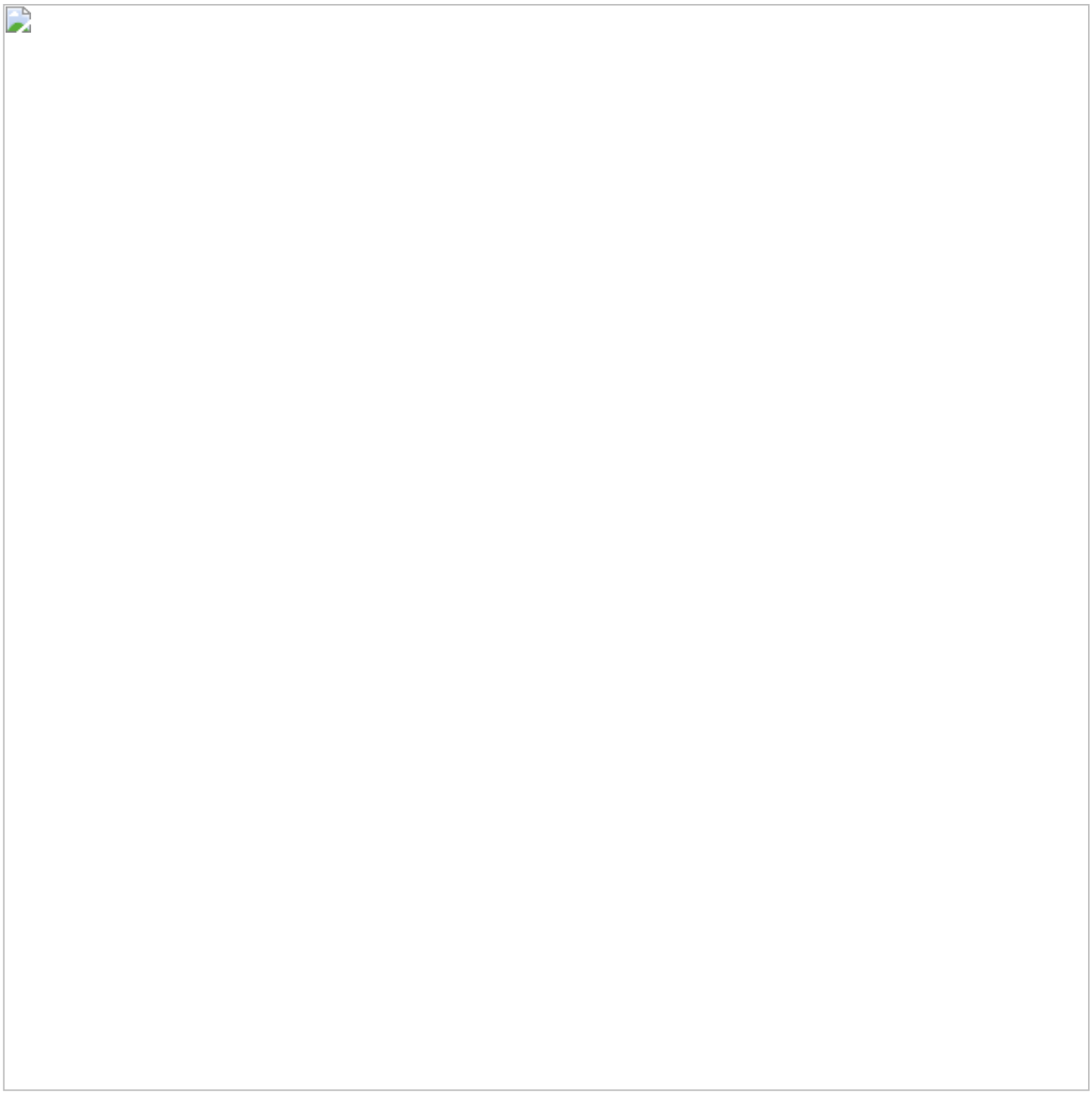
This is the second malicious campaign I got using MSBuild in less than a week. Usually, it starts with an RDP access using a valid account, spreads over the network via remote Windows Services (SCM), and pushes Cobalt Strike beacon to corporate hosts abusing the MSBuild task feature as described in today's diary.

Abusing MSBuild

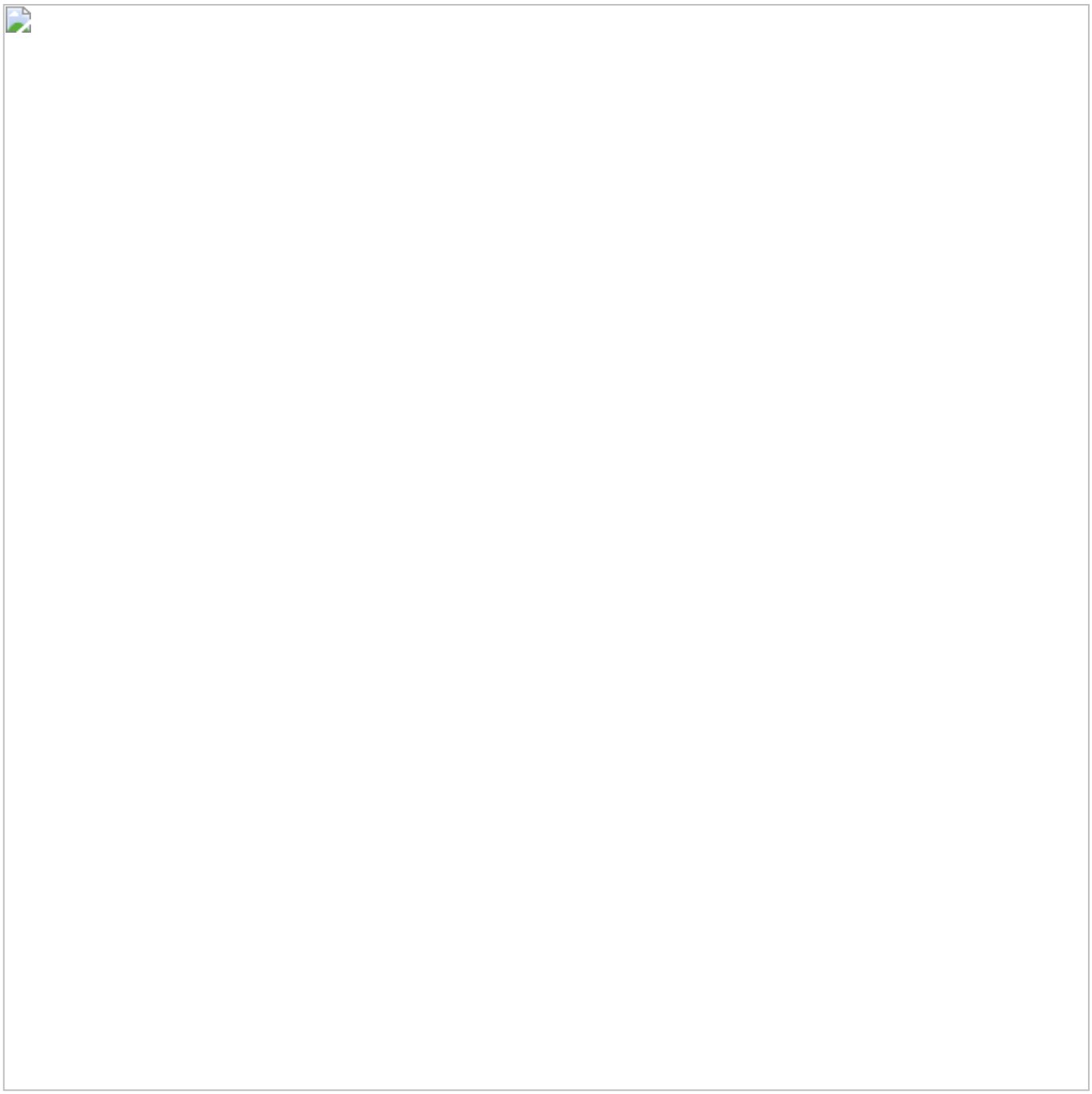
To make it easier to understand how attackers are abusing MSBuild, look at Figure 1. It shows the XML file for a simple project (HelloWorld.csproj) with a task named HelloWorld prepared to compile and execute the custom C# code during project building.

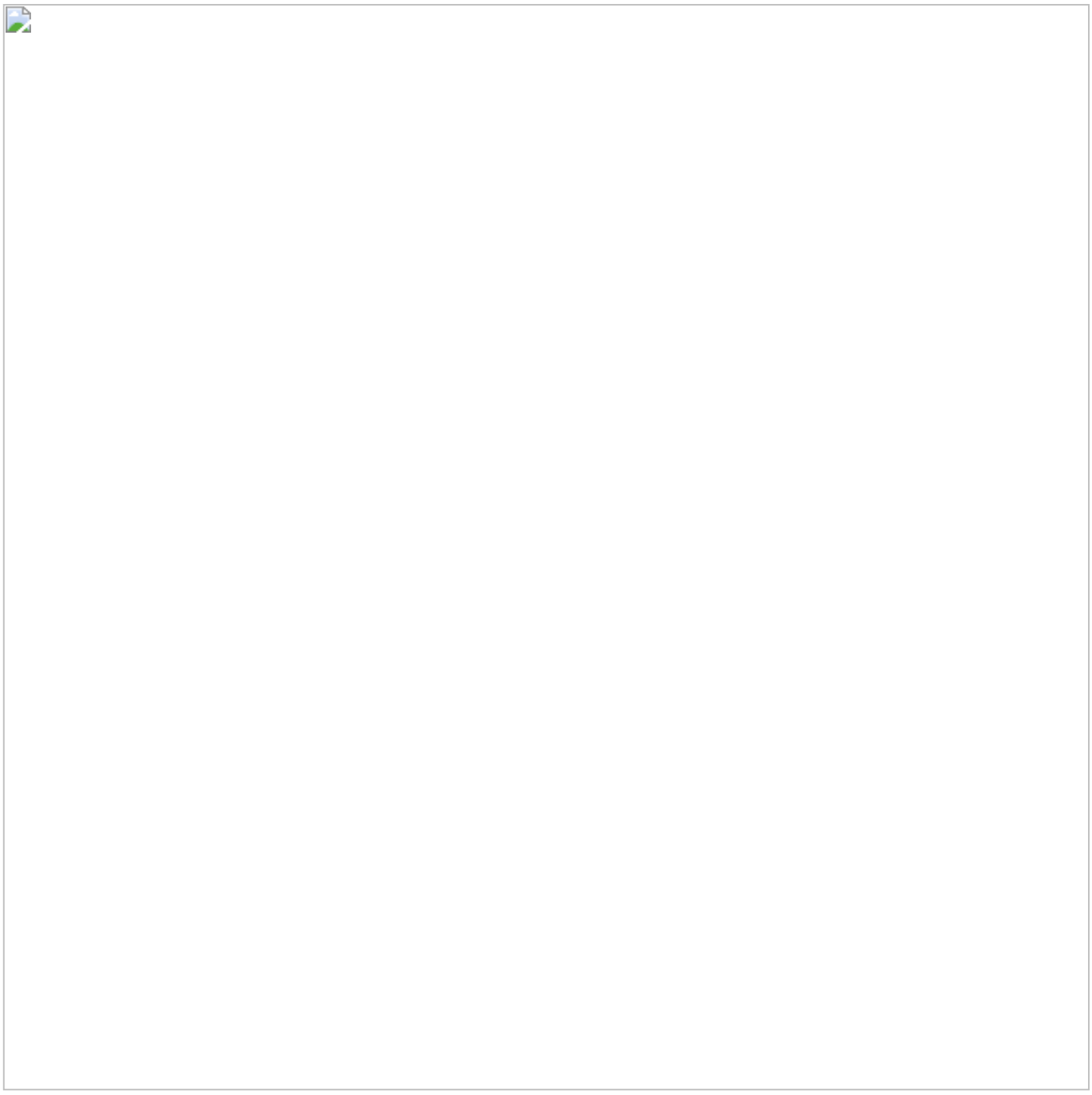


When building the project using MSBuild, as seen in Figure 2, the task HelloWorld will be executed, which in turn will call the ‘Execute()’ and ‘Start()’ methods, which will finally print the “Hello World” message on the console. The ‘Execute()’ method comes from the interface ‘ITask’ implemented by the ‘HelloWorld’ class [2].

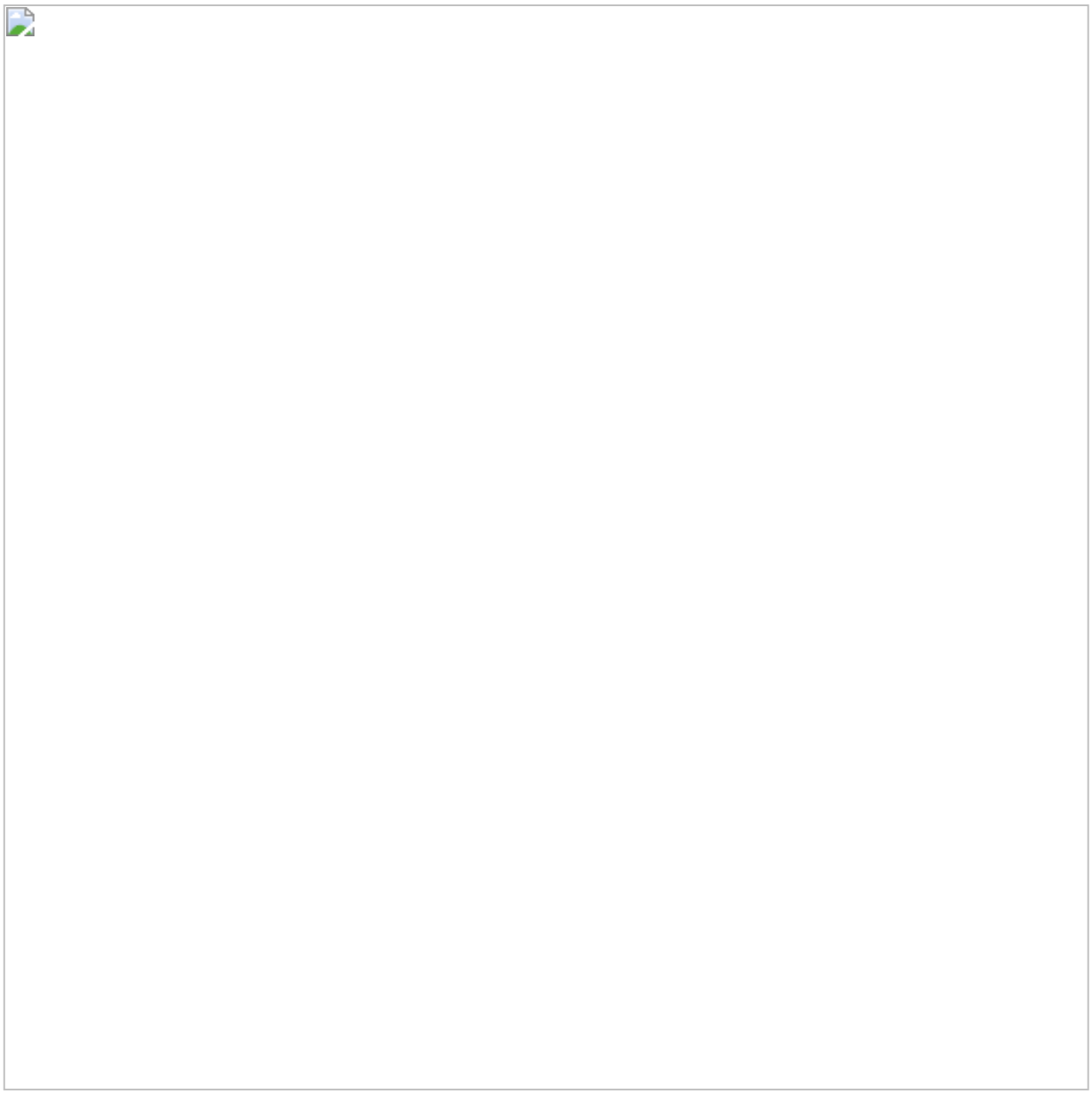


Now, let's look at the malicious MSBuild project file in Figure 3. Using the same principle, when called by MSBuild, it will compile and execute the custom C#, decode and execute the Cobalt Strike beacon on the victim's machine.



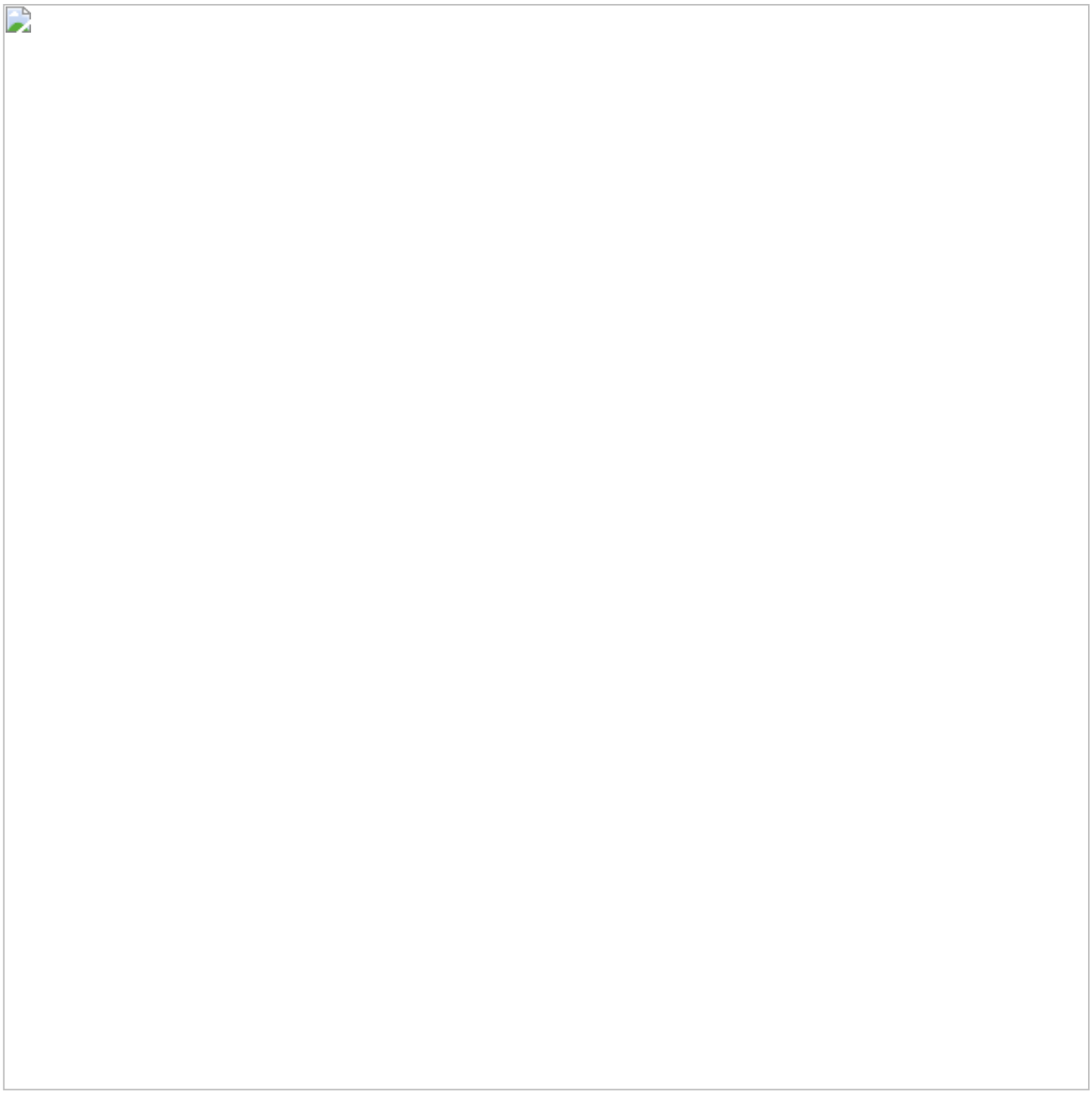


In Figure 5, it’s possible to see the beacon connected to the C2 server (23.227.178.115).

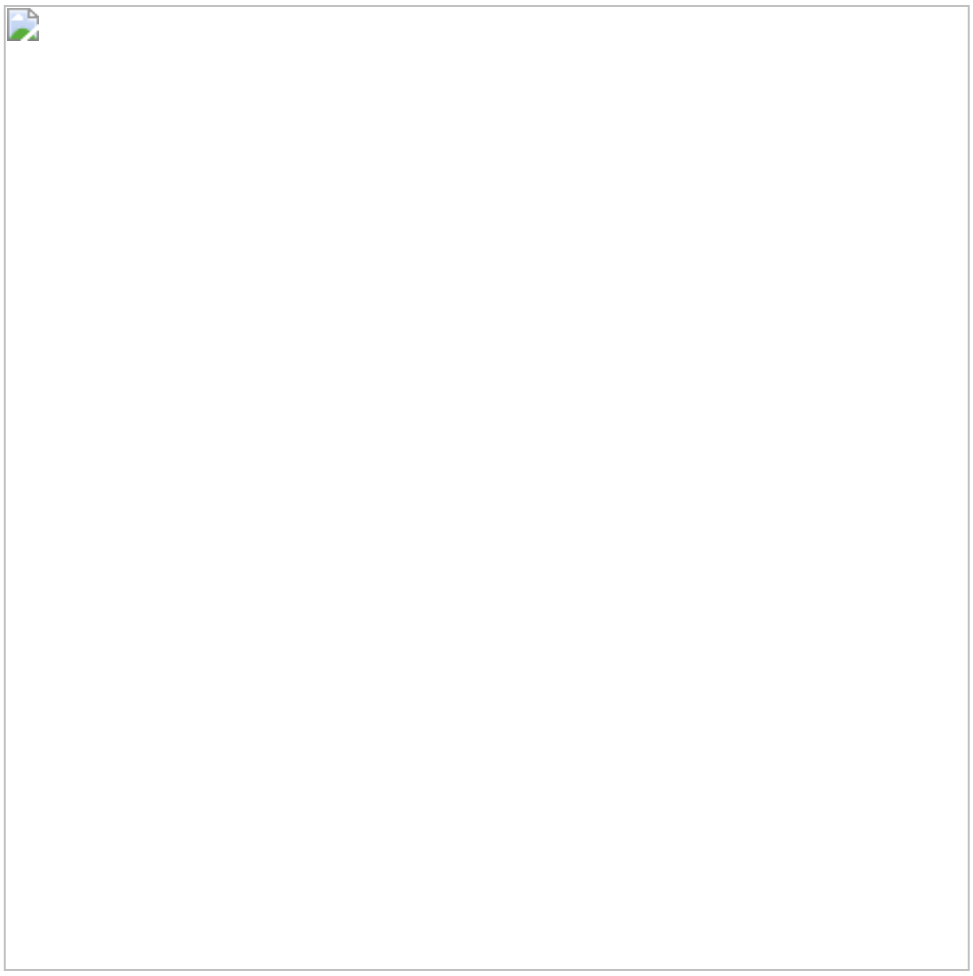


Analyzing the Cobalt Strike beacon

To analyze the code executed by the malicious MSBuild project, first, it's necessary to decrypt the variable 'buff' (refer to Figure 3). The variable is decoded during MSBuild execution by the "for" loop marked in Figure 6. It runs an XOR function between each byte of the 'buff' and key_code arrays. By the end, the 'buff' byte array will store the decrypted malicious content. The rest of the code will allocate memory and execute the payload using *Marshal.GetDelegateForFunctionPointer*.

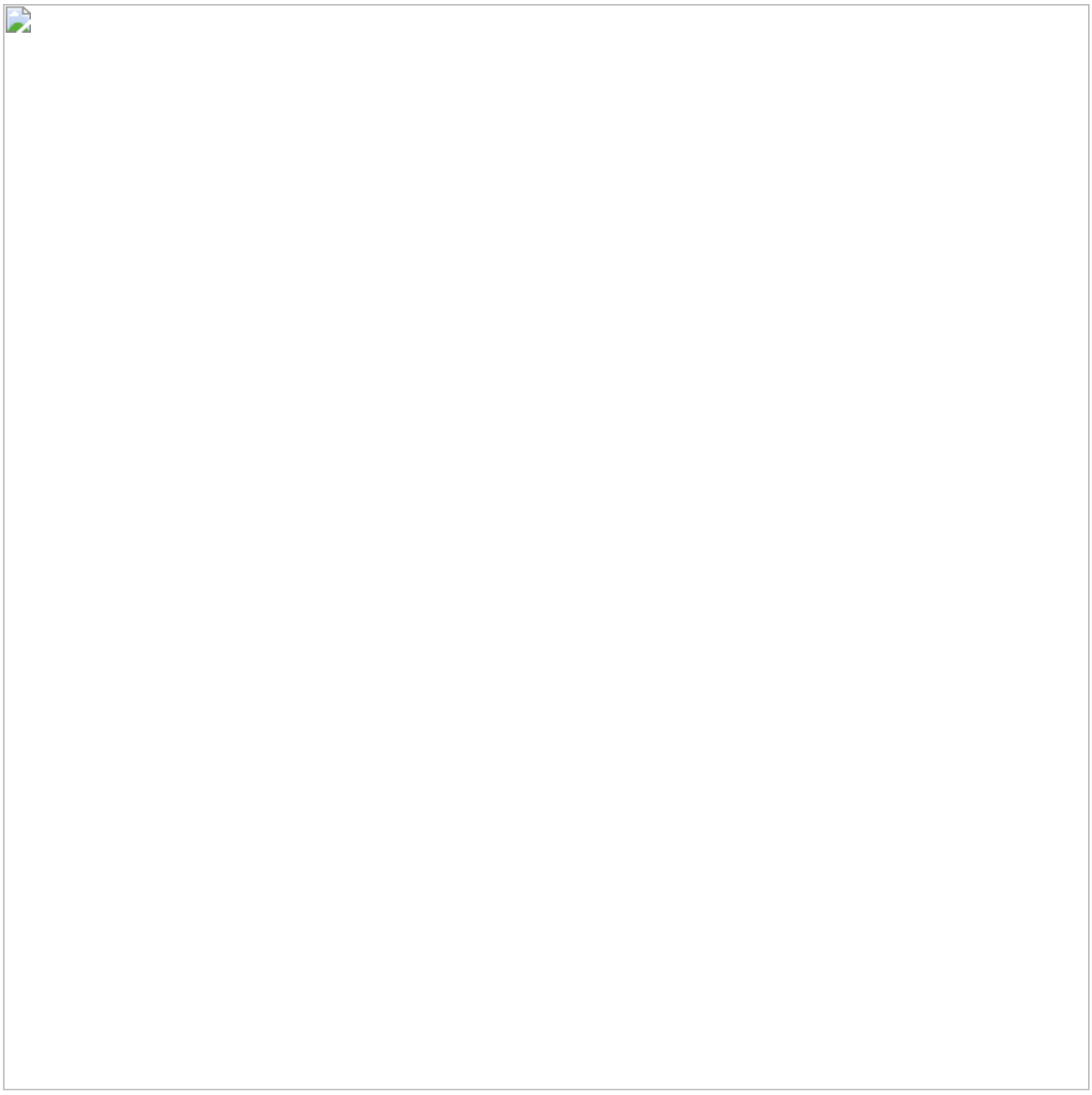


I implemented the same decryption function in Python to decrypt the code, as seen in Figure 7. Before the decryption loop, the script reads the content of buff and key_code from the MSBuild project file and copy to the correspondent variables in the Python script. The script code is available here.



To profile the resulting binary, I started looking for its hash on VirusTotal, which returned no matches. Continuing the low hanging fruit approach, I did a 'strings' and found interesting strings that I have already seen in other Cobalt Strike beacons like "*could not create remote thread in %d: %d*" and "*IEX (New-Object Net.Webclient). DownloadString('http://127.0.0.1:%u/'); %s"*.

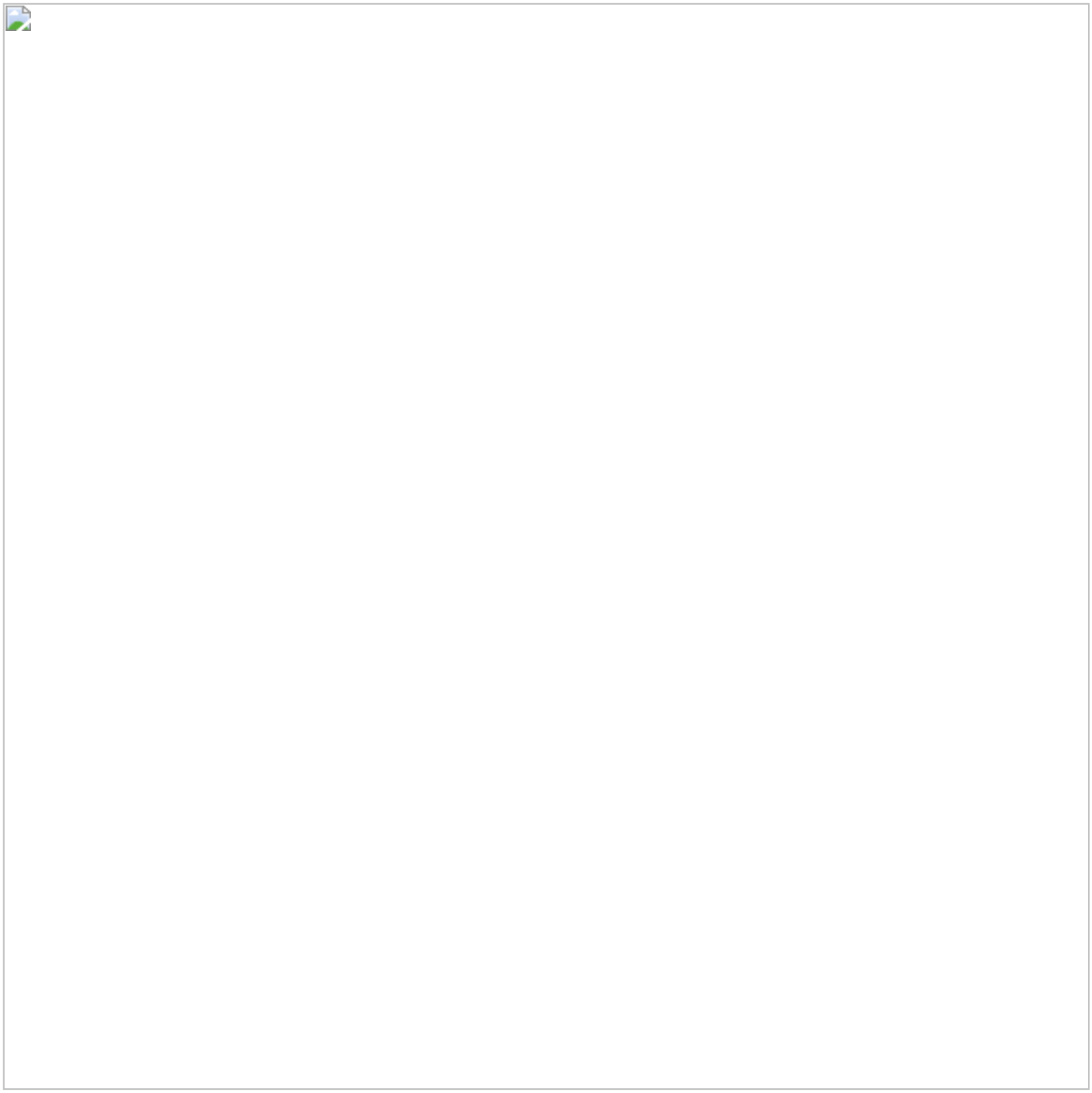
To confirm, I used the tool CobaltStrikeParser from Sentinel-One [3]. This tool parses the Cobalt Strike beacon and extracts its configuration, as seen in Figure 8.



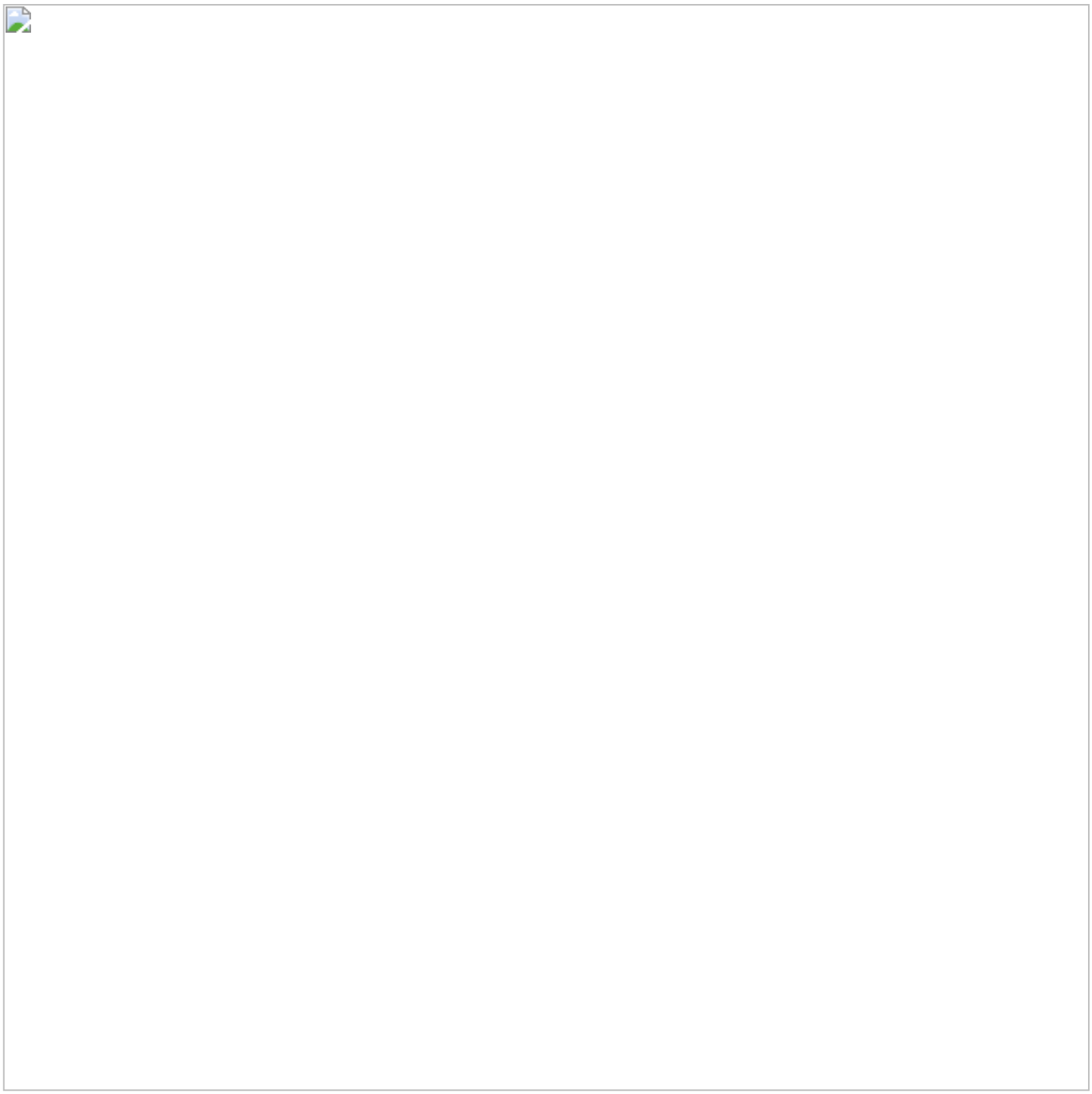
The configuration says that the C2 server (23.227.178.115) will be contacted via HTTPS (encrypted traffic) on port TCP/8888. It also informs the endpoint /dpixel for GET requests and /submit.php for POST requests and that the spawn processes are rundll32.exe — this is the process used to run commands requested by C2 on the victim's machine.

One usual next step when I have a C2 up and running is to analyze the traffic and try to discover more about the campaign. To do so in this case, I verified if the private key for the Cobalt Strike beacon is known using a project by Didier Steven's 1768 [4]. This project not only parses the Cobalt Strike's beacon configuration but also indicates if the corresponding private key was found by Didier on malicious Cobalt Strike servers on the internet. Read more about this project at [5].

So, after running the 1768 tool, I could find that the private key is known for the Cobalt Strike beacon analyzed, as seen in Figure 9.



But, before using the private key to decrypt the Cobalt Strike traffic, remember that the communication with the C2 is SSL encrypted. In Figure 10, there is a sample captured traffic.



One way to decrypt the SSL traffic is to use a man-in-the-middle approach. To this end, I used the project mitmproxy. The communication schema when using a tool like this is to make the client, the Cobalt Strike beacon, talk to the SSL proxy and make the SSL proxy to talk with the C2 server. In the middle (proxy), we will have the traffic unencrypted.

See below command I used to run the mitmproxy:

> $ SSLKEYLOGFILE=”~/.mitmproxy/sslkeylogfile.txt” mitmproxy -k — mode transparent

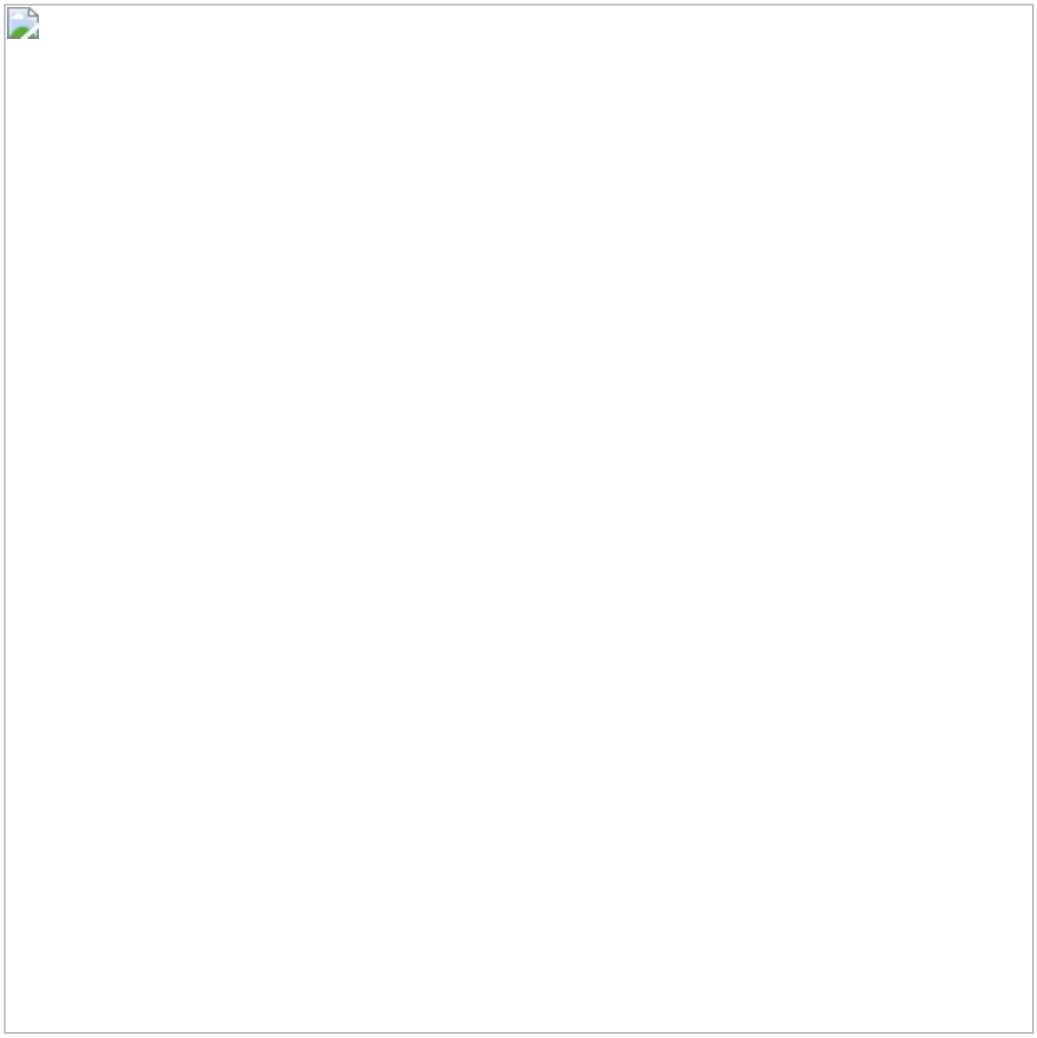
The SSLKEYLOGFILE variable will indicate mitmproxy to store SSL/TLS master keys on the file sslkeylogfile.txt. This file can be used to decrypt the traffic in external tools like Wireshark [https://docs.mitmproxy.org/stable/howto-wireshark-tls/]. The ‘-k’ says to mitmproxy to do not

verify upstream server SSL/TLS certificates and the transparent mode is used when the client does not know or is configured to use a proxy.

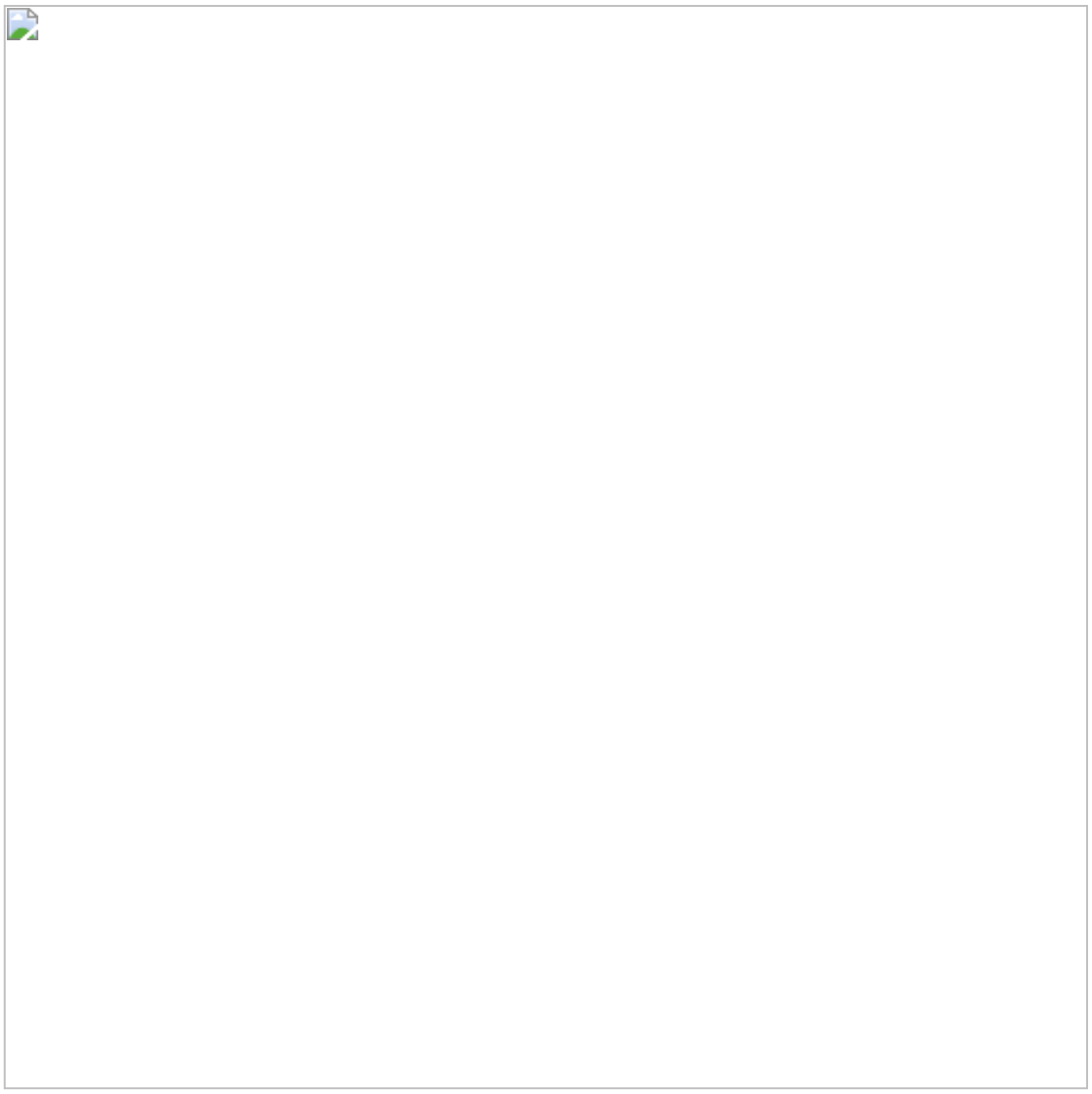
Before running the mitmproxy, remember to enable 'IP forwarding' and create the necessary NAT rules to redirect the SSL traffic from the client machine (where the Cobalt Strike beacon is running) mitmproxy port. For my case, the commands were:

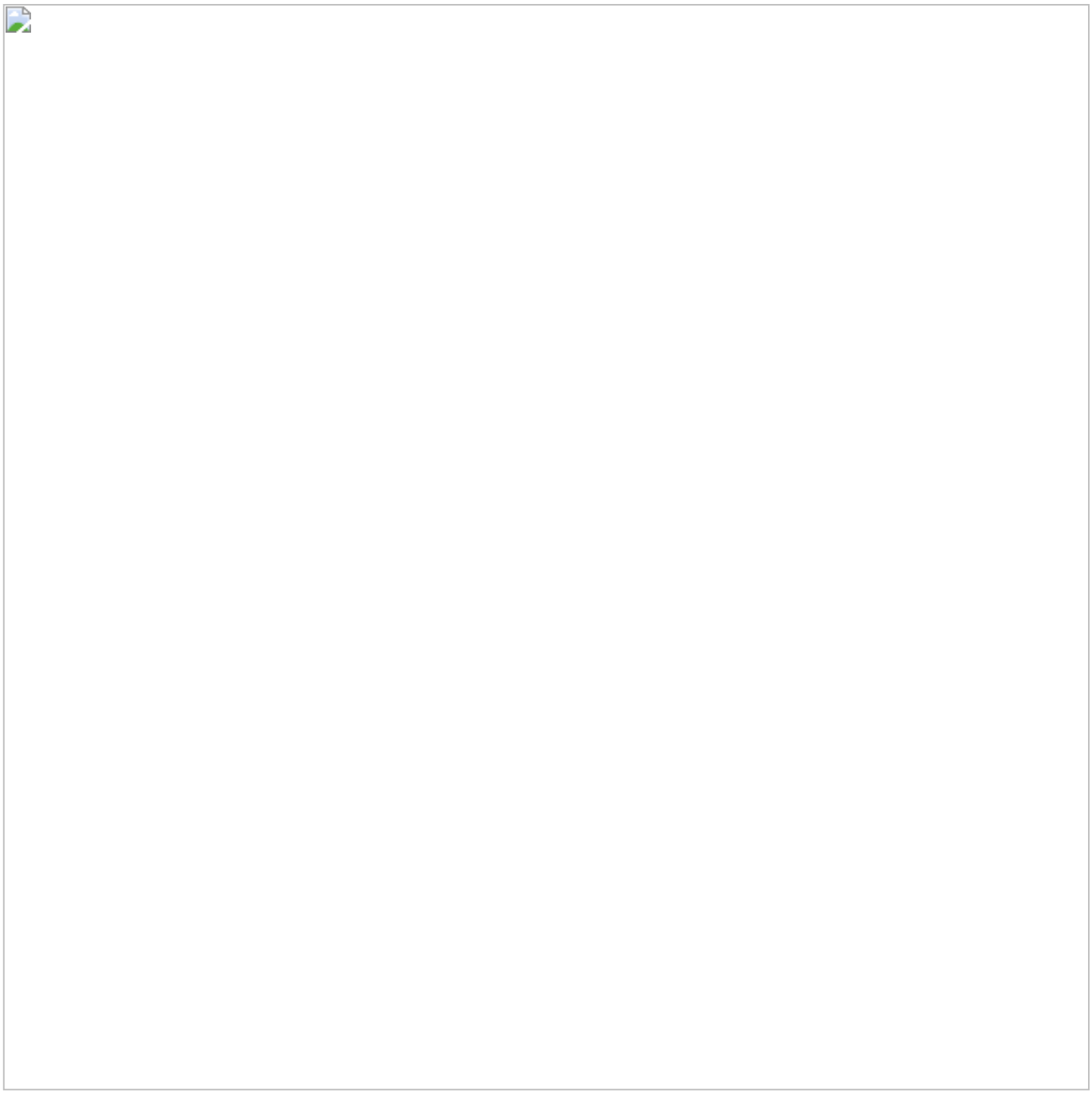
> $ sudo sysctl -w net.ipv4.ip_forward=1
> $ sudo iptables -t nat -A PREROUTING -i <interface> -p tcp — dport 8888 -j REDIRECT — to-port 8080

Another important thing is installing the mitmproxy certificate on the Windows machine running the beacon. The default certificate is located into ~/.mitmproxy/mitmproxy-ca-cert.cer file. Copy it to Windows and install the certificate on Trusted Root Certification Authorities and Trusted Publishers, as seen in Figure 11.



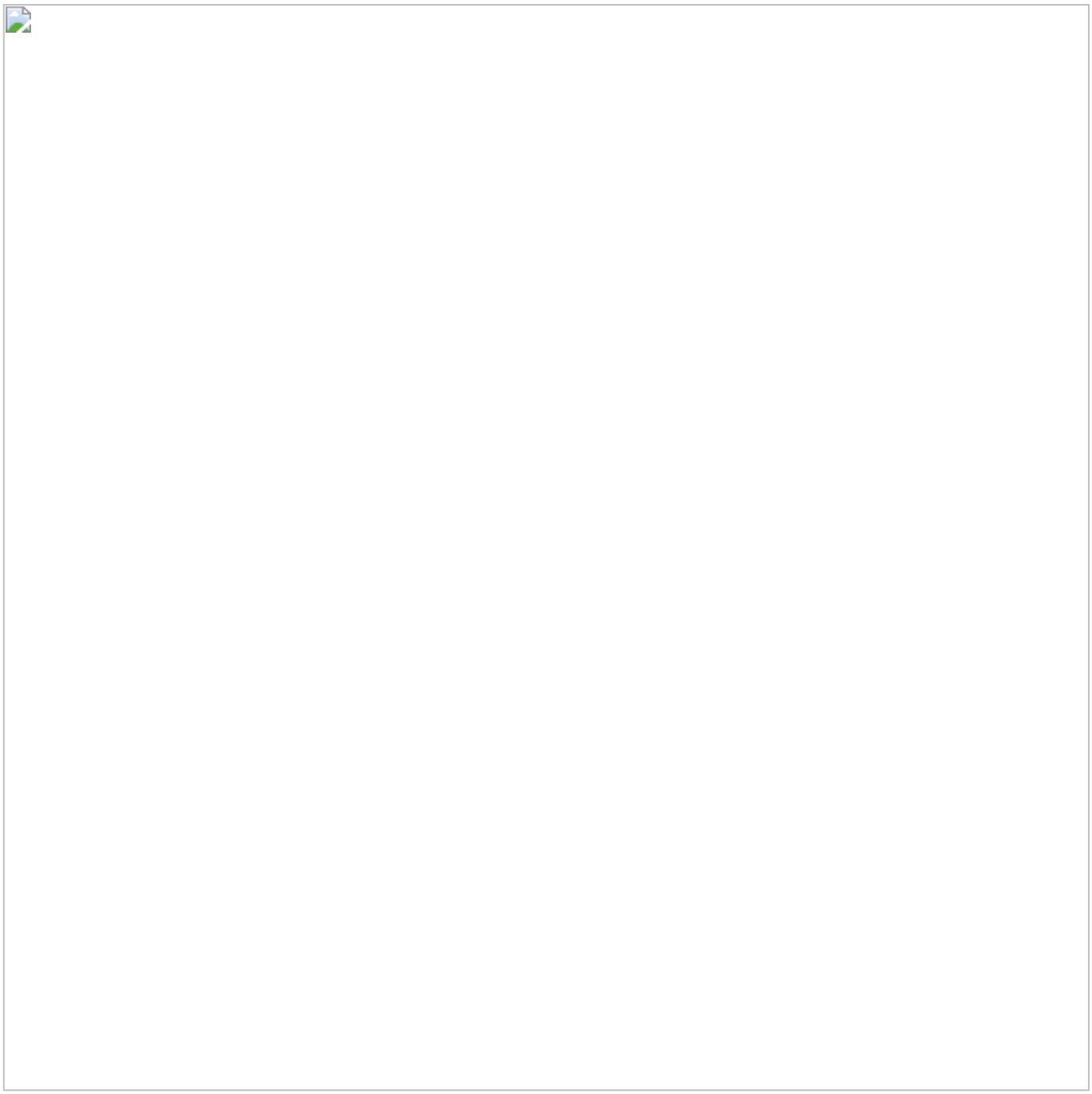
Once the pre-requisites are met, running the mitmproxy could have the SSL unencrypted traffic collected, as seen in Figure 12 and Figure 13.



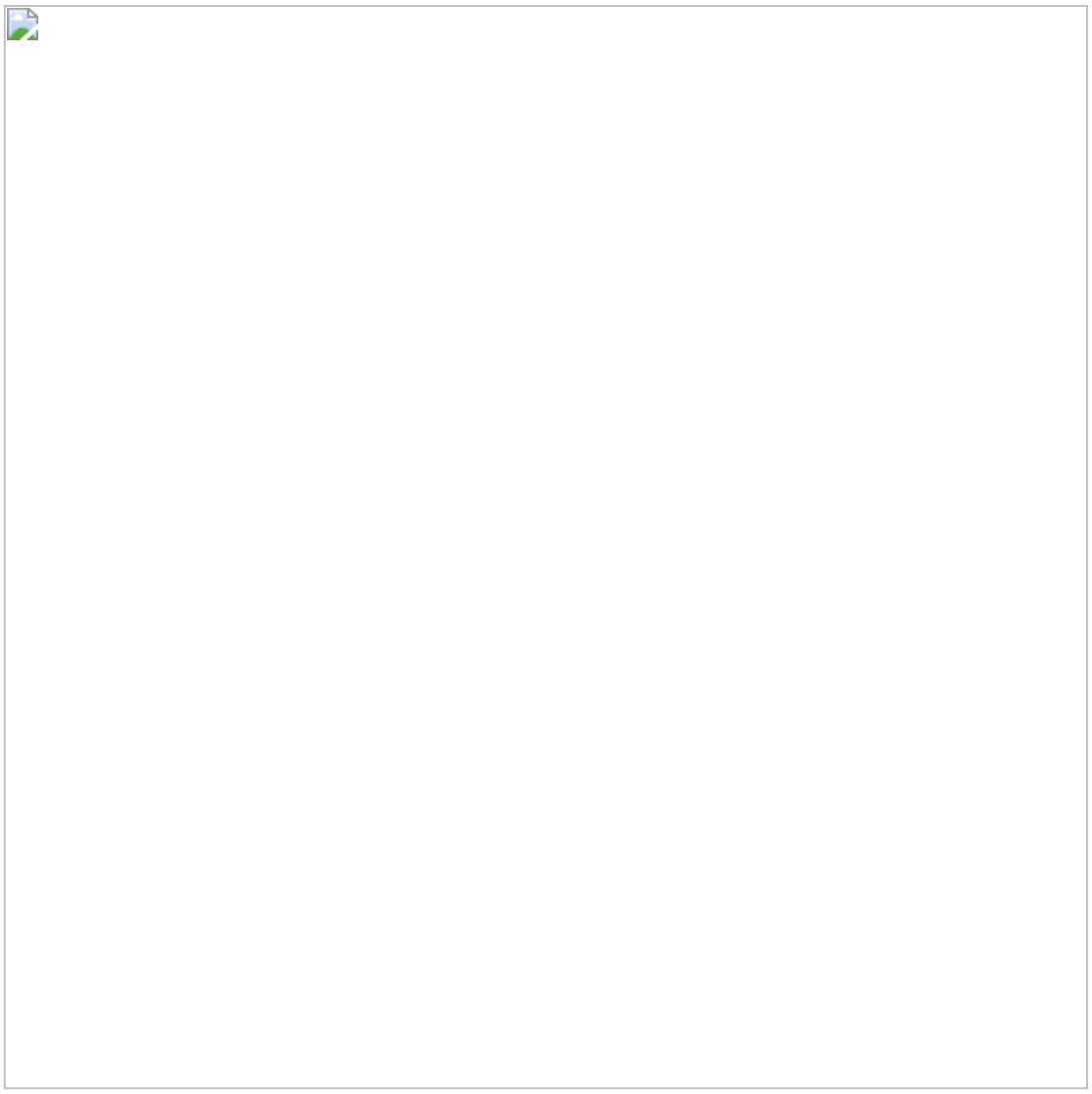


Remember from the Cobalt Strike beacon configuration (Figure 8) that the HTTP get metadata is stored into the ‘Cookie’ header encoded with base64. So, the content marked in red in Figure 13 is the content to be decrypted using the Cobalt Strike private key.

To decrypt the content, I used another excellent tool from Didier Stevens called cs-decrypt-metadata [6] as seen in Figure 14.



Finally, if you want to have the traffic unencrypted on Wireshark, you can use the SSL/TSL keys stored into “~/.mitmproxy/sslkeylogfile.txt”. Import the file using Wireshark menu Edit->Preferences->Protocols->TLS->(Pre)-Master-Secret log file name. After that, it’s possible to see the traffic unencrypted like in Figure 15.



Recommendations

MSBuild composes the list of applications signed by Microsoft that can allow the execution of other codes. According to Microsoft's recommendations [7], these applications should be blocked by the Windows Defender Application Control (WDAC) policy.

There is a note for MSBuild.exe, though, that if the system is used in a development context to build managed applications, the recommendation is to allow msbuild.exe in the code integrity policies.

References

[1] https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022
[2] https://docs.microsoft.com/en-us/visualstudio/msbuild/task-writing?view=vs-2022
[3] https://github.com/Sentinel-One/CobaltStrikeParser
[4] https://blog.didierstevens.com/2021/11/21/update-1768-py-version-0-0-10/
[5] https://blog.didierstevens.com/2021/10/21/public-private-cobalt-strike-keys/

[6] blog.didierstevens.com/2021/11/12/update-cs-decrypt-metadata-py-version-0–0–2/
[7] https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/microsoft-recommended-block-rules