Python and Malware: Developing Stealth and Evasive Malware without Obfuscation

Vasilios Koutsokostas[1] and Constantinos Patsakis[1,2]
[1]*Institute of Problem Solving, Department of Informatics, University of Piraeus, Piraeus, Greece*
[2]*Information Management Systems Institute, Athena Research Center, Artemidos 6, Marousi 15125, Greece*

Keywords: Malware, Antivirus, Python, Evasion, Sandbox.

Abstract: With the continuous rise of malicious campaigns and the exploitation of new attack vectors, it is necessary to assess the efficacy of the defensive mechanisms used to detect them. To this end, the contribution of our work is twofold. First, it introduces a new method for obfuscating malicious code to bypass all static checks of multi-engine scanners, such as VirusTotal. Interestingly, our approach to generating the malicious executables is not based on introducing a new packer but on the augmentation of the capabilities of an existing and widely used tool for packaging Python, PyInstaller but can be used for all similar packaging tools. As we prove, the problem is deeper and inherent in almost all antivirus engines and not PyInstaller specific. Second, our work exposes significant issues of well-known sandboxes that allow malware to evade their checks. As a result, we show that stealth and evasive malware can be efficiently developed, bypassing with ease state of the art malware detection tools without raising any alert.

1 INTRODUCTION

Adversaries are continually trying to attack systems, to gain access to information and other resources. This leads to a continuous arms race that has significantly augmented the sophistication of the methods used to penetrate systems and, to a lesser extent, those deployed to protect them. Therefore, novel security mechanisms are developed using advanced methods to detect malicious patterns exploiting all possible features, using machine learning and artificial intelligence methods in the past few years. However, the adversaries are crafting complex new attacks, exploiting the human factor, and often resort to encryption and other obfuscation methods to hide their malicious traffic and actions.

Malware, a piece of software that is crafted to perform a malicious task in a computing system, is a problem which has plagued computing systems for decades. Furthermore, the relatively recent introduction of cryptocurrencies has significantly changed the cybercrime ecosystem as it has provided a simple monetisation method with some privacy guarantees. Indeed, as reported by several sources, cybercrime has become a multi-billion underground economy with such economic impact ((IC3), 2019; Thomas, 2020) that the World Economic Forum considers it the second most-concerning threat to global commerce over the next decade (Forum, 2020).

The main pillars for detecting and analysing malware are static and dynamic analysis (Gandotra et al., 2014). In the former, there is no execution of the file under inspection. Therefore, we try to correlate patterns in every aspect of the file that can be collected without executing it, including but not limited to imported libraries, file segments, API calls, strings, file structure, entropy, etc. On the contrary, in dynamic analysis, we open and/or execute the file in a controlled testing environment (sandbox) to identify what this piece of software does. In this regard, we keep track of every possible network interaction, filesystem change, memory dump, processes etc. (Or-Meir et al., 2019), replicating a real-world host environment. Moreover, one may debug the binary to execute it, possibly line by line, to understand what it does and how, and even manipulate its behaviour.

The above methods are well-known to malware authors who try to bypass them by introducing obfuscation and other anti-analysis methods (Branco et al., 2012). Modern malware frequently uses packers and encryption to obfuscate their contents and bypass static analysis checks by generating new binaries with different static properties. Similarly, they are often armoured with evasion methods to bypass dy-

Koutsokostas, V. and Patsakis, C.
Python and Malware: Developing Stealth and Evasive Malware without Obfuscation.
DOI: 10.5220/0010541501250136
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 125-136
ISBN: 978-989-758-524-1

namic analysis. Thus, they perform specific checks in the system to determine whether they are being executed in a virtual environment and if known protection mechanisms typical of sandboxes are running, and can assess their execution mode to tell whether they are being debugged (Issa, 2012). If any of these checks is positive, the malware typically changes its behaviour to harden and slow down its analysis. All the above can come under one umbrella to facilitate malware evasion by simultaneously packing the binary and armouring it with a myriad of evasion methods (Liţă et al., 2018).

The main goal of this work is to assess the effort and methods needed to create *stealth malware*. We define this stealth concept in an objective and repeatable way. More precisely, we consider that a malware sample is stealth if (i) it achieves a "clean sheet" after inspection by multi-engine scanners, such as VirusTotal (VT) and (ii) malware sandbox environments do not consider it malicious per se. VT and other similar services used *statically* examine the file with several dozens of antiviruses (AVs). Therefore, even if an AV may detect the malware on execution, VT's verdict might classify it as benign. Note that a clean sheet verdict from VT, which has around 70 AVs clearly shows the trend of the market, meaning that the rest of the AVs, which are minor share of the market are not expected to have different behaviour. Practically, our work starts from understanding why some AVs are erroneously flagging some executables as malicious and uncovers an inherent problem of AV engines when handling Python files. This can be easily escalated to develop undetectable malware. What is even more alarming is the fact that while one may argue that there are several tricks to bypass static AV tests by hiding the payload, we illustrate that a threat actor does not need to cover the payload. Widely used payloads can be simply embedded in Python and escape the detection.

Nevertheless, it is clear that once the user is lured to execute malware, it might be too late to block its actions. Moreover, we consider the malware as stealth if it escapes detection from the state of the art malware sandboxes. To this end, we experimented with the most well-known sandboxes publicly available on the Internet. Our analysis and experiments have uncovered significant issues in these sandbox environments that allow malware to bypass them. Based on the above, our work illustrates critical issues in detecting malware that affects the whole ecosystem, spanning from how AVs statically recognise malware, to the evasion from sandboxed environments. Practically, using our methods, one may efficiently develop malware or armour an existing one so that that it is not detected by a wide range of state of the art tools used for detecting malware.

In what follows, we provide a brief overview of the related work. Then, we discuss the conceptual approach for the development of stealth malware. In Section 5, we analyse our experiments and the extracted results. Then, in Section 6, we discuss our findings and their impact. The article concludes summarising the contributions of our work and streamlining future work.

Ethical Compliance. Our work complies with the standards for conducting offensive security in an ethical way. To this end, we have responsibly disclosed our findings to each sandbox provider individually prior to submitting this work. Moreover, we have not published nor communicated our methods to prevent them from being used in the wild.

2 RELATED WORK

Similar to the use of sandboxes for cats, a malware sandbox is a controlled virtualised environment in which a potentially dangerous file is submitted for inspection, so that it does not "litter" the rest of the system. This environment will automatically execute/open the file and analyse its behaviour, such as filesystem interaction, network connections, registry changes and access, API calls, memory access, etc. The virtualised and isolated nature of the environment prevents the malware from causing any harm to the system performing the analysis. Another approach would be to actually debug the suspicious file and examine in detail command by command and even alter its behaviour.

Clearly, the above is not the ideal for the adversary, so almost all modern malware come equipped with an evasion method leveraging, for instance, sandbox and debugger detection methods. For the sandbox evasion, the malware performs a broad range of checks to assess the environment they are being executed. In essence, the malware will look for *environmental artifacts* (Bulazel and Yener, 2017) which include but are not limited to hardware identifiers, presence of user interaction, sensor readings, uptime, usernames, timing discrepancies, registry values, and hardware specifications (Martignoni et al., 2009; Shi et al., 2014), see Figure 1. Therefore, such a malware would resolve to i) calls to the registry, check the process list and filesystem to perform pattern matching against a predefined set of strings ii) time measurements to determine whether the elapsed time is aligned with the expected processing time and iii) detect possible deviations from the outcome of specific commands. The above indicates that minor details, for instance, the MAC address of the network may easily reveal the virtualised environment as well as the list of running processes or inconsistencies in CPU/GPU specifications. Some malware may also use logical bombs to deliver their payload. For instance, the execution can be delayed based on time constraints or enabled only after proper packet receipt from a specific domain. In fact the time that a honeypot devotes for execution of a sample introduces many differences on what data is collected. As recently reported by Küchler et al. (Küchler et al., 2021), the bulk most of the malware behavior is observed during the first two minutes of execution, while further actions may take up to ten minutes.

It must be highlighted at this point that due to the monetisation model (discussed later on), a sandbox will not execute and inspect a binary for an arbitrary amount of time. Additionally, to analyse as many samples as possible, it cannot provide all the available system resources. Therefore, by delaying the execution, allocating a lot of space and memory, a malware may evade detection. Thus, the sharing of the processing resources may easily expose the virtualised environment as the VM could report the host's processor with a fragment of the available cores. Recently Huang et al. (Huang et al., 2020) introduced PiDicators which do not use API calls but pure assembly code and far fewer checks to determine whether a binary is being executed in a VM triggering far fewer alerts. It has to be noted that the wide adoption of virtualised environments in, e.g. cloud computing, some malware is even more targeted, trying to detect sandboxed environments and not simply virtualised (Yokoyama et al., 2016). For more on evasion methods the interested reader may refer to (Chen et al., 2008; Issa, 2012; Petsas et al., 2014; Uitto et al., 2017; Veerappan et al., 2018; Afianian et al., 2019; Checkpoint Research, 2020; Apostolopoulos et al., 2021).

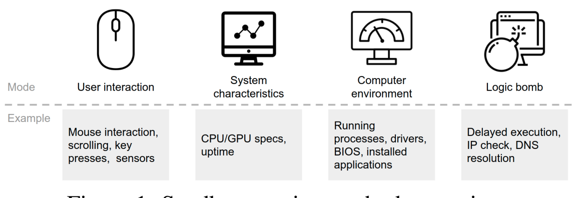

Mode
User interaction
System characteristics
Computer environment
Logic bomb
Example
Mouse interaction, scrolling, key presses, sensors
CPU/GPU specs, uptime
Running processes, drivers, BIOS, installed applications
Delayed execution, IP check, DNS resolution

Figure 1: Sandbox evasion methods overview.

These countermeasures from the malware have resulted in the introduction of anti-evasion methods. For instance, MALGENE (Kirat and Vigna, 2015) performs data flow analysis and data mining on the system calls to determine whether the inspected binary actions could be a result of an evasion method.

VM Cloak (Shi et al., 2017) checks the environment for misconfigurations and differences in execution environments that could reveal that the execution is done in a VM, while Leguesse et al. (Leguesse et al., 2017) harden Android sandboxes which have more sensors to cover. A widely used project for hiding Windows VMs is A. Ortega's pafish[1] which focuses on the checks that are performed by malware.

Recently, D'Elia et al. (Cono D'Elia et al., 2020) introduced a dynamic binary instrumentation based method, called BluePill which allows analysts to instrument the binaries they are dissecting evasive malware in a stealth way so that they cannot determine that they are being debugged. Nevertheless, this is another part of the continuous battle, bringing, for instance, anti-anti evasion methods in this fight (D'Elia et al., 2019).

Finally, it should be noted that bare-metal malware execution environments, so the execution is performed in an actual and not virtualised environment, so there is no VM nor sandbox stain to cover, are also considered in the literature (Kirat et al., 2011; Guan et al., 2017; Kirat et al., 2014; Mutti et al., 2015; Deng and Mirkovic, 2018), nevertheless, they cannot be considered a practical solution for assessing malware samples at the desired rate as they cannot scale efficiently.

3 PYTHON & PYINSTALLER

Python is an interpreted programming language with continuous increasing popularity. Despite its readability and simplicity, it has accumulated several features over the years, making it very attractive for scripting and Rapid Application Development. Currently, it is widely used for server-side web development, machine learning, system scripting and secure software-related engineering, especially offensive.

The fact that Python can be used in all major platforms, as well as the fact that it is easy to write and many exploits and offensive security tools, have been written in Python has pushed a lot of malware authors to write their malware in this programming language[2]. However, we argue that there is another more

[1]https://github.com/a0rtega/pafish

[2]https://unit42.paloaltonetworks.com/unit-42-technical-analysis-seaduke/,https://blog.talosintelligence.com/2020/04/poetrat-covid-19-lures.html,https://blog.netlab.360.com/not-really-new-pyhton-ddos-bot-n3cr0m0rph-necromorph/,https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/

important issue with Python that makes it more attractive for malware authors. AVs have not properly integrated this attack vector in their scope, as we will show in the next paragraphs.

While Python is preinstalled by default in most Unix-like operating systems, it is not the case of Windows. Moreover, Python, as an interpreted language, does not compile to create an executable. To create an executable from a Python script, there are several options, with the most popular one being PyInstaller. PyInstaller takes as input a Python and tries to discover all its module and library dependencies that are needed to properly execute it. To do this, PyInstaller is recursively looking for imports of the necessary files, until it reaches native Python modules and libraries. Once the dependencies are identified, instead of keeping the Python scripts, PyInstaller keeps the compiled Python scripts (.pyc files), usually referred to as Python bytecode. These files, along with an active Python interpreter and environment in the form of what is called the bootloader, are copied in a folder. Thus, PyInstaller allows the packaging of applications in folders and unique executable files without the need to have Python preinstalled.

The bootloader is the core component of PyInstaller as it prepares the environment for executing the Python code and actually executes it. The bootloader is different for each architecture and highly customizable. Once someone launches a bundled Python application, the bootloader is initiated and spawns another child process of itself. The parent bootloader process handles the signals for the two processes and uncompresses all the .pyc files in a folder named _MEIxxxxxx in the temp folder of the host, where xxxxxx is a random number. The child process loads the temporary Python environment with all the needed modules and libraries for the script can be imported and executes the script. Once the child process terminates, the parent process will cleanup and terminate as well.

To compress the files and create a single executable, PyInstaller uses two compression methods, ZlibArchives for Python compiled files (executable Python zip archives) and CArchive for all other files. In this work, we delibrately study PyInstaller as beyond being the most widely used solutions for creating executables from Python, many other installers are based on it. Therefore, the issues reported in this case can be escalated to other installers.

4 CONCEPTUAL APPROACH

Our work's conceptual approach is to progressively determine what triggers detection of a malicious binary in static and dynamic analysis and create patches to remove it. We argue that if VirusTotal and other similar engines consider a binary as benign and the dynamic analysis from a sandbox does not trigger an alert, the binary is deemed benign, even by security savvies. In this regard, a *suspicious* indication of sandbox would be considered simply suspicious. Therefore, it will fall below the detection radars and would be executed by a typical user. While we understand that an anti-malware mechanism may detect it upon execution, this is clearly too late in most cases.

Two individual streams emerged from this basic concept, targeting towards evading each analysis. Once we developed the measures that bypassed each one of them individually, we merged them into a unique binary. Therefore, we will present the approach and experiments individually. As we will detail in the next section, for the static analysis, we uploaded our samples to VirusTotal and used the detection output and classification of each antivirus, the reported YARA rules, as well as the community comments to determine which static properties are the ones that lead to the detection of the malware. To further validate our results, we submitted our results to two more similar engines. For the dynamic analysis with sandboxes, we initially submitted some binaries that collected data from each sandbox environment and then used this as an input to armour our binary with evasion measures. Notably, as discussed later in the article, we identified several important issues for many of the sandboxes that were responsibly communicated to them.

4.1 Bypassing Static Analysis

The methodology behind the technique to bypass the static analysis stems from observations on PyInstaller (https://www.pyinstaller.org/) 4.0 binaries. To generate an executable, PyInstaller adds a lot of "noise" to the generated binaries, from, e.g. the libraries that are appended, and even if the code is not malicious, many AVs falsely treat the executable as malware. In fact, as reported by the community, in numerous occasions even simple "Hello world" Python scripts are flagged as malicious by several AVs as they consider binaries generated by PyInstaller as malicious by default.

The latter exhibits an erroneous policy applied by almost all AVs; at least the ones used in VT, when handling binaries produced by PyInstaller. In practice, none of them understands its output; probably because of its overblown added libraries. Therefore, on the one hand, we have most antivirus for which PyInstaller acts like an efficient *packer*, so one can hide arbitrary code in them. On the other hand, other AVs have understood this capacity and immediately flag the binaries as malicious.

```
powershell -NoP -NonI -W Hidden -Exec
    Bypass -Command New-Object System.
    Net.Sockets.TCPClient("10.0.0.1"
    ,4242);$stream = $client.GetStream
    ();[byte[]]$bytes = 0..65535|%{0};
    while(($i = $stream.Read($bytes, 0,
     $bytes.Length)) -ne 0){;$data = (
    New-Object -TypeName System.Text.
    ASCIIEncoding).GetString($bytes,0,
    $i);$sendback = (iex $data 2>&1 |
    Out-String );$sendback2  =
    $sendback + "PS " + (pwd).Path + ">
     ";$sendbyte = ([text.encoding]::
    ASCII).GetBytes($sendback2);$stream
    .Write($sendbyte,0,$sendbyte.Length
    );$stream.Flush()};$client.Close()
```

Listing 1: A typical Powershell reverse shell.

In what follows, we dig a bit deeper on the problem with PyInstaller to understand the nature of the noise that makes it act like a packer. We start with a simple reverse shell with a PowerShell script which is typically flagged by AVs. The one-line script is provided in Listing 1. Note that similar backdoor mechanisms; e.g. malicious PowerShell execution, are widely used by malware in the wild. Two scripts, one in JavaScript and one in Python were written appending the exact same PowerShell code snippet to their body; therefore, no obfuscation is applied. While both of them are plain ASCII files, with minimal differences in their contents and the malicious string in plain sight, there are significant deviations on their detection from AVs, see Figure 2, which are rather alarming. More precisely, one may observe that the JavaScript file is flagged as malicious by four times more AVs than its Python peer. Notably, none of them were identified correctly, the JavaScript is considered as *text* and the Python as *Java*. While the inconsistency in the detection rate of AVs for almost the same plaintext file cannot be easily understood, the compiled Python file (`pyc`), and Python bytecode in general, illustrates a more catastrophic result. None of the AVs is able to recognise it as malicious; therefore, it shows that none of the AVs understands what is inside a `pyc` file as the conversion to the Python compiled file efficiently obfuscates the contents of the script to bypass the static analysis.

The above illustrates a clear strategy to bypass static analysis for an executable. One has to write a Python script which does all the "dirty job" and compile it using PyInstaller to hide its malicious content. Then, if we masquerade the PyInstaller enough so that it is not considered as such, we may pass any executable without any detection from the AVs.

Based on the above, our strategy is to exploit these inefficiencies in handling binaries generated by PyInstaller. Thus, the plan is to use PyInstaller to create the binaries out of malicious scripts, but then remove all the possible static features that it appends from the binary. The general outline of the method is illustrated in Algorithm 1.

4.2 Bypassing Dynamic Analysis

The dynamic analysis bypass is solely targeted towards bypassing the checks performed by executing the binary in a set of well-known and widely used sandboxes. To this end, we first created a set reconnaissance of executables that were simply collecting environmental data from each sandbox and performing some checks with a standard tool for assessing the sandboxes' quality for malware analysis, pafish. Once collected, the input was then sent to a server that we controlled to gather and analyse it.

Beyond the output of pafish, which identified several misconfigurations and our own findings, one has to consider some particular inherent issues that such services have. The environmental findings have to be considered in the scope of a service offered in a virtualised environment, for a limited amount of time and with the minimum amount of resources to allow for scaling. As a result, a VM cannot always meet a typical computer's specifications in terms of, e.g. memory, disk, etc.

Finally, one has also to consider that most samples in such a sandbox originate from users without paid plans, so these are tested in VMs that are more limited. Based on the market model (see Section 2), if a file is considered benign by the static analysis, and the sandboxes have not identified it as malicious, the chances of the file being rescanned in a "better" VM drop dramatically.

5 EXPERIMENTAL RESULTS

5.1 Static Malware Analysis

Following our findings for the handling of Python bytecode, the main goal of the experiments is to alter the executable in a way that it does not look generated by PyInstaller. In our experiments, we opted to use some standard malicious payloads as a codebase that were executed through Python, create an executable

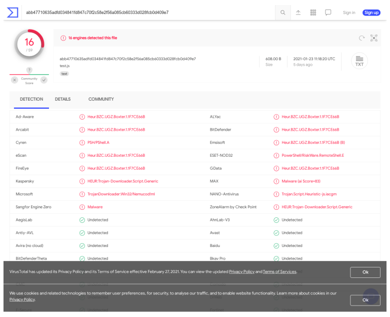


abb47710635adfd034841fd847c70f2c58e2f56a085cb60333d028fcb0d409e7

16 / 59

16 engines detected this file

abb47710635adfd034841fd847c70f2c58e2f56a085cb60333d028fcb0d409e7
test.js
text

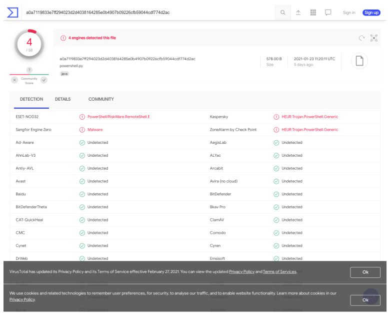
608.00 B Size | 2021-01-23 11:18:20 UTC 5 days ago

Ad-Aware	Heur.BZC.UGZ.Boxter.1.1F7CE66B	ALYac	Heur.BZC.UGZ.Boxter.1.1F7CE66B
Arcabit	Heur.BZC.UGZ.Boxter.1.1F7CE66B	BitDefender	Heur.BZC.UGZ.Boxter.1.1F7CE66B
Cyren	PSH/PShell.A	Emsisoft	Heur.BZC.UGZ.Boxter.1.1F7CE66B (B)
eScan	Heur.BZC.UGZ.Boxter.1.1F7CE66B	ESET-NOD32	PowerShell/RiskWare.RemoteShell.E
FireEye	Heur.BZC.UGZ.Boxter.1.1F7CE66B	GData	Heur.BZC.UGZ.Boxter.1.1F7CE66B
Kaspersky	HEUR:Trojan-Downloader.Script.Generic	MAX	Malware (ai Score=83)
Microsoft	TrojanDownloader:Win32/Nemucod!ml	NANO-Antivirus	Trojan.Script.Heuristic-js.iacgm
Sangfor Engine Zero	Malware	ZoneAlarm by Check Point	HEUR:Trojan-Downloader.Script.Generic
AegisLab	Undetected	AhnLab-V3	Undetected
Antiy-AVL	Undetected	Avast	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefenderTheta	Undetected	Bkav Pro	Undetected

VirusTotal has updated its Privacy Policy and its Terms of Service effective February 27, 2021. You can view the updated Privacy Policy and Terms of Services.

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(a) JavaScript using PowerShell reverse shell.



a0a7119833e7ff294023d2d4038164285e0b4907b09226cfb59044cdf774d2ac

4 / 58

4 engines detected this file

a0a7119833e7ff294023d2d4038164285e0b4907b09226cfb59044cdf774d2ac
powershell.py
java

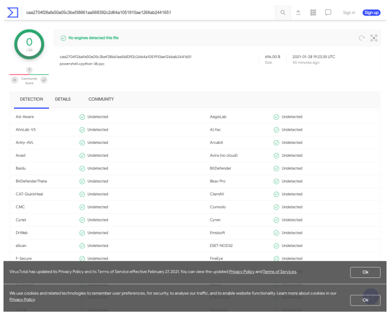
578.00 B Size | 2021-01-23 11:20:11 UTC 5 days ago

ESET-NOD32	PowerShell/RiskWare.RemoteShell.E	Kaspersky	HEUR:Trojan.PowerShell.Generic
Sangfor Engine Zero	Malware	ZoneAlarm by Check Point	HEUR:Trojan.PowerShell.Generic
Ad-Aware	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	ALYac	Undetected
Antiy-AVL	Undetected	Arcabit	Undetected
Avast	Undetected	Avira (no cloud)	Undetected
Baidu	Undetected	BitDefender	Undetected
BitDefenderTheta	Undetected	Bkav Pro	Undetected
CAT-QuickHeal	Undetected	ClamAV	Undetected
CMC	Undetected	Comodo	Undetected
Cynet	Undetected	Cyren	Undetected
DrWeb	Undetected	Emsisoft	Undetected

VirusTotal has updated its Privacy Policy and its Terms of Service effective February 27, 2021. You can view the updated Privacy Policy and Terms of Services.

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(b) Python using PowerShell reverse shell.



caa2704f26afa50a05c3bef38661aa568392c2d64a1051910ae1266ab2441651

0 / 59

No engines detected this file

caa2704f26afa50a05c3bef38661aa568392c2d64a1051910ae1266ab2441651
powershell.cpython-38.pyc

696.00 B Size | 2021-01-28 19:23:35 UTC 45 minutes ago

Ad-Aware	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	ALYac	Undetected
Antiy-AVL	Undetected	Arcabit	Undetected
Avast	Undetected	Avira (no cloud)	Undetected
Baidu	Undetected	BitDefender	Undetected
BitDefenderTheta	Undetected	Bkav Pro	Undetected
CAT-QuickHeal	Undetected	ClamAV	Undetected
CMC	Undetected	Comodo	Undetected
Cynet	Undetected	Cyren	Undetected
DrWeb	Undetected	Emsisoft	Undetected
eScan	Undetected	ESET-NOD32	Undetected
F-Secure	Undetected	FireEye	Undetected

VirusTotal has updated its Privacy Policy and its Terms of Service effective February 27, 2021. You can view the updated Privacy Policy and Terms of Services.

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(c) Compiled Python scipt (`pyc`) of the Python using PowerShell reverse shell script.

Figure 2: Scan results for reverse shell scripts using Javascript and Python.

Algorithm 1: Bypassing static analysis.

```
 1: procedure OBFUSCATE_PAYLOAD(x)
 2:     Select proper payload;
 3:     Parametrise the payload;
 4:     XOR the payload with a random key;
 5:     Convert the XORed payload to base64;
 6: procedure PATCH_BOOTLOADER(exe)
 7:     Rename PyInstaller references to a random
    string
 8:     Rename files and their calls with pyi_ prefix
    to a random prefix.
 9:     Replace default icons
10:     Update linker's flags in WScript
11: procedure PATCH_BINARY(exe)
12:     Add version to the binary
13:     Remove rich header
14:     Rename _RTDATA header to .bss
15:     Recalculate PE32 checksum.
16: Select payload P
17: P'=Obfuscate_payload(P)
18: Use a Python S script to call P';
19: Build a PE32 executable from S to a single file to
    generate the bootlader B
20: B'=Patch_Bootloader(B)
21: Build the PE32 executable PE32 from S to a
    single file with bootlader B'
22: PE32'=Patch_Binary(PE32)
```

with the corresponding bootloader of PyInstaller, and then make the necessary changes to the bootloader and the executable to prevent AVs from detecting it.

Initially, we wrote a script with a known malicious shellcode payload from `msfvenom` and a Powershell command that downloads the EICAR anti-malware testfile and XORed that Powershell command with a random hard-coded string and converted it to base64. The reason for these choices is that both of them are well known to trigger AVs; therefore, if any of them is identified by an AV or a sandbox, it will immediately flag the file as malicious in both static and dynamic analysis. We compiled this script with PyInstaller and submitted the executable to VT. As shown in Figure 4a, multiple AV engines reported our executable as malicious. Moreover, we scanned a simple "hello world" Python script compiled with PyInstaller in VT, and it was also reported as malicious by the same antivirus engines (Figure 4b), verifying again the issues described in the previous section. To further validate our results, we created some binaries with the exact same functionality using C++, Rust, and Go and submitted them for analysis to VT, see Figures 4c, 4d and 4e respectively. It is important to highlight in the latter figures that, contrary to the ones for Python, the AVs have correctly identified the presence of shellcode and Meterpreter, as shown by the names that they attribute to our binaries. The difference is rather important since the shellcode is not encoded in any of the implementations showing that PyInstaller has efficiently hidden it from the AVs once again.

Based on the above, it is apparent that by altering the PyInstaller fingerprint on the executable, we may evade the static analyses of many AVs. Thus, to bypass PyInstaller identification by AVs, we initially made some clear "static" changes. These changes were i) substitution of strings and files from "*pyi_*" to a random short string, ii) rename of "*PyInstaller*" strings to another random short string, iii) replacement of the default icons, and iv) addition of flags to the linker in WScript, see Table 1. After these changes, we built the new bootloader. We then compiled the malicious script with the modified PyInstaller bootloader, managing to reduce the AVs that reported our executable as malicious to four (Figure 5a). Note that the aforementioned actions are bypassing several checks with YARA rules that some AVs might perform, see Figure 3.

Since our binary did not have any version information, we added one and recompiled it. While a trivial action, after scanning this executable on VT, the AVs are reporting our binary as malicious was further reduced to two (Figure 5b). Finally, we opened the last built of our executable with PEtools (https://github.com/petoolse/petools), cleared the rich header and renamed the `_RDATA` header to `.bss` and recalculated the checksum. The removal of the rich header was made to prevent the detection of the binary through the signature of this header (Webster et al., 2017). This final executable achieved zero detections from VT, see Figure 5c. The result was also cross-validated with other custom and multi-engine scanners, e.g. Kaspersky Threat intelligence portal (https://opentip.kaspersky.com/), Gatewatcher (https://intelligence.gatewatcher.com/), MetaDefender (https://metadefender.opswat.com/), see Figure 5f, 5d and 5e, respectively.

Table 1: Linker flags for PyInstaller.

Flag	Description
/BASE:0x00400000	Set base to default Windows PE image base
/DYNAMICBASE:NO	Disable dynamic base
/VERSION:5.2	Set image version
/RELEASE	Set the checksum of the file

```
import "pe"
import "hash"
rule PyInstaller
{
meta:
    description = "Identifies executable converted using PyInstaller."
    author = "@bartblaze"
    date = "2020-01"
    tlp = "White"

strings:
    $ = "pyi-windows-manifest-filename" ascii wide
    $ = "pyi-runtime-tmpdir" ascii wide
    $ = "PyInstaller: " ascii wide

condition:
    uint16(0) == 0x5a4d and any of them or
    (
   for any i in (0..pe.number_of_resources - 1):
     (pe.resources[i].type == pe.RESOURCE_TYPE_ICON and
      hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
      "20d36c0a435caad0ae75d3e5f474650c") //Default PyInstaller icon
    )
}
```

Figure 3: A common YARA rule for detecting PyInstaller. Source: https://github.com/bartblaze/Yara-rules/blob/5f4961049d0d510b11250d5628383398889fc881/rules/generic/PyInstaller.yar.

5.2 Dynamic Analysis with Sandboxes

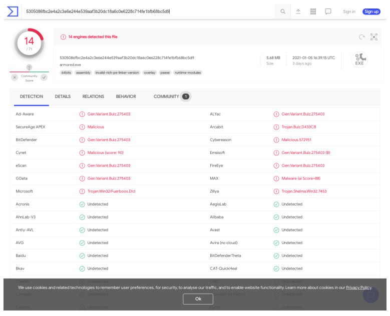
To assess the sandboxes and create a proper evasion method, we first need to establish a ground truth baseline for the environment that the sandboxes use. Therefore, the strategy is to initially create a binary that collects intelligence and then aggregate it to make a binary that exploits it to bypass the detection.

To this end, we first created some reconnaissance binaries that were submitted to Intezer, Any.run, Triage, Hybrid Analysis, the public Cuckoo installation of the Estonian CERT (https://cuckoo.cert.ee/), Cape, and Threat Grid sandboxes. However, not all of them allowed Internet connections to the binaries. Therefore, we used a machine with a public IP to collect the input from the reconnaissance binaries when the Internet connection was available. When this was not the case, we manually inspected the logs that were generated from the sandboxes as we wrote the corresponding logs to the disk and registry.

To bypass the execution of our malicious code in a sandbox environment, we analysed the collected data to identify common deficiencies. The most significant misconfiguration in almost all sandboxes was the CPU specifications. More precisely, there were obvious contradictions regarding the threads and cores of the reported CPU. For instance, a sandbox was reporting an AMD EPYC 7371 16-Core Processor, but in the meantime, it was also reporting two cores and two threads. Therefore, we collected all available CPU specifications from Intel and AMD and added them as dictionaries in our the evasive final malware. An aggregated table of the issues that we identified in each sandbox is reported in Table 2 and will be further discussed in the following paragraphs.

Despite the identified deficiencies, bypassing all of them in a binary is not straight forward. The reason is that continuous calls to read registry values, or WMI is triggering alerts in the sandboxes. Thus, one needs to unify these checks and prioritise them according to the "noise" they introduce to the sandbox. Therefore, in our malicious binary, we introduced several conditions before executing the payload.

Firstly, we check whether any known sandbox or VM process is running in the background. Afterwards, we check whether the threads of the system are more than four and if the available RAM is more than 1 GB which is the bare minimum for most of the 64bit modern computers. Then, we check whether the system is powered on more than a threshold, e.g. 2-3 minutes. Next, we examine the foreground applications and the parent of the process of our binary. The reason for this check is the execution process of a sandbox. In most cases, there is a dropper script which opens the file and exits. However, in a real-world execution environment, one would expect that the user would have some other open programs, whether this is the Explorer, Word, or a terminal that would initiate the execution of the binary. Clearly,


5305086fbc2e4a2c3e6e244e539aaf3b20dc18a6c0e6228c714fe1bfb68bc5d9

14 / 71

14 engines detected this file

5305086fbc2e4a2c3e6e244e539aaf3b20dc18a6c0e6228c714fe1bfb68bc5d9
armored.exe
64bits assembly invalid-rich-pe-linker-version overlay peexe runtime-modules

5.68 MB Size

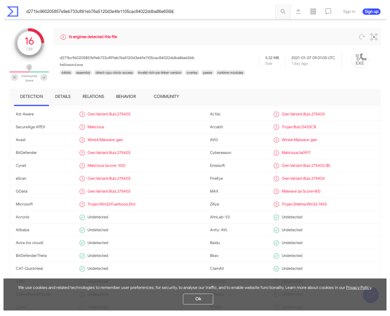
2021-01-05 16:39:15 UTC 3 days ago

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY 1

Ad-Aware	Gen:Variant.Bulz.275403	ALYac	Gen:Variant.Bulz.275403
SecureAge APEX	Malicious	Arcabit	Trojan.Bulz.D433CB
BitDefender	Gen:Variant.Bulz.275403	Cybereason	Malicious.572951
Cynet	Malicious (score: 90)	Emsisoft	Gen:Variant.Bulz.275403 (B)
eScan	Gen:Variant.Bulz.275403	FireEye	Gen:Variant.Bulz.275403
GData	Gen:Variant.Bulz.275403	MAX	Malware (ai Score=88)
Microsoft	Trojan:Win32/Fuerboos.D!cl	Zillya	Trojan.Shelma.Win32.7453
Acronis	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
Antiy-AVL	Undetected	Avast	Undetected
AVG	Undetected	Avira (no cloud)	Undetected
Baidu	Undetected	BitDefenderTheta	Undetected
Bkav	Undetected	CAT-QuickHeal	Undetected

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(a) Original binary with PyInstaller


d271bc960205857e9eb733c891eb76a5120d3e4fe1105cac84022ddba86e656b

16 / 69

16 engines detected this file

d271bc960205857e9eb733c891eb76a5120d3e4fe1105cac84022ddba86e656b
helloword.exe
64bits assembly direct-cpu-clock-access invalid-rich-pe-linker-version overlay peexe runtime-modules

5.32 MB Size

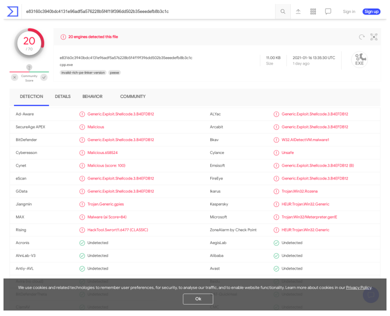
2021-01-07 09:01:05 UTC 1 day ago

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY

Ad-Aware	Gen:Variant.Bulz.275403	ALYac	Gen:Variant.Bulz.275403
SecureAge APEX	Malicious	Arcabit	Trojan.Bulz.D433CB
Avast	Win64:Malware-gen	AVG	Win64:Malware-gen
BitDefender	Gen:Variant.Bulz.275403	Cybereason	Malicious.1a0917
Cynet	Malicious (score: 100)	Emsisoft	Gen:Variant.Bulz.275403 (B)
eScan	Gen:Variant.Bulz.275403	FireEye	Gen:Variant.Bulz.275403
GData	Gen:Variant.Bulz.275403	MAX	Malware (ai Score=83)
Microsoft	Trojan:Win32/Fuerboos.D!cl	Zillya	Trojan.Shelma.Win32.7453
Acronis	Undetected	AhnLab-V3	Undetected
Alibaba	Undetected	Antiy-AVL	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefenderTheta	Undetected	Bkav	Undetected
CAT-QuickHeal	Undetected	ClamAV	Undetected

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(b) Hello world binary with PyInstaller.


e83160c3940bdc4131e96adf5a576228b5f4f19f396dd502b35eeedefb8b3c1c

20 / 70

20 engines detected this file

e83160c3940bdc4131e96adf5a576228b5f4f19f396dd502b35eeedefb8b3c1c
cpp.exe
invalid-rich-pe-linker-version peexe

11.00 KB Size

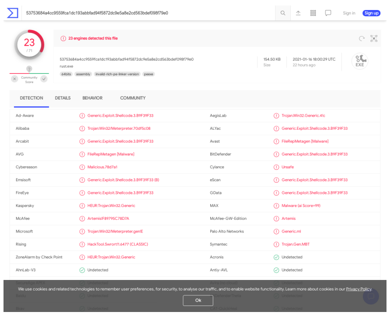
2021-01-16 13:35:30 UTC 1 day ago

DETECTION DETAILS BEHAVIOR COMMUNITY

Ad-Aware	Generic.Exploit.Shellcode.3.B4EFDB12	ALYac	Generic.Exploit.Shellcode.3.B4EFDB12
SecureAge APEX	Malicious	Arcabit	Generic.Exploit.Shellcode.3.B4EFDB12
BitDefender	Generic.Exploit.Shellcode.3.B4EFDB12	Bkav	W32.AIDetectVM.malware1
Cybereason	Malicious.658524	Cylance	Unsafe
Cynet	Malicious (score: 100)	Emsisoft	Generic.Exploit.Shellcode.3.B4EFDB12 (B)
eScan	Generic.Exploit.Shellcode.3.B4EFDB12	FireEye	Generic.Exploit.Shellcode.3.B4EFDB12
GData	Generic.Exploit.Shellcode.3.B4EFDB12	Ikarus	Trojan.Win32.Rozena
Jiangmin	Trojan.Generic.gpies	Kaspersky	HEUR:Trojan.Win32.Generic
MAX	Malware (ai Score=84)	Microsoft	Trojan:Win32/Meterpreter.gen!E
Rising	HackTool.Swrort!1.6477 (CLASSIC)	ZoneAlarm by Check Point	HEUR:Trojan.Win32.Generic
Acronis	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
Antiy-AVL	Undetected	Avast	Undetected

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(c) C++ compiled executable with the malicious payload.


53753684a4cc9559fca1dc193abbfad94f5872dc9e5a8e2cd563bdef098f79e0

23 / 71

23 engines detected this file

53753684a4cc9559fca1dc193abbfad94f5872dc9e5a8e2cd563bdef098f79e0
rust.exe
64bits assembly invalid-rich-pe-linker-version peexe

154.50 KB Size

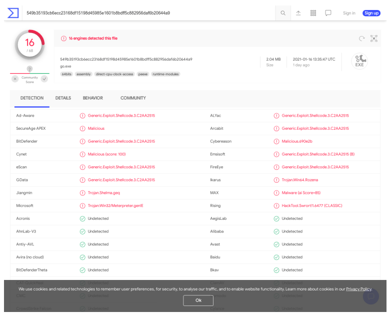
2021-01-16 18:00:29 UTC 22 hours ago

DETECTION DETAILS BEHAVIOR COMMUNITY

Ad-Aware	Generic.Exploit.Shellcode.3.B9F39F33	AegisLab	Trojan.Win32.Generic.4!c
Alibaba	Trojan:Win32/Meterpreter.70df5c08	ALYac	Generic.Exploit.Shellcode.3.B9F39F33
Arcabit	Generic.Exploit.Shellcode.3.B9F39F33	Avast	FileRepMetagen [Malware]
AVG	FileRepMetagen [Malware]	BitDefender	Generic.Exploit.Shellcode.3.B9F39F33
Cybereason	Malicious.78d7a1	Cylance	Unsafe
Emsisoft	Generic.Exploit.Shellcode.3.B9F39F33 (B)	eScan	Generic.Exploit.Shellcode.3.B9F39F33
FireEye	Generic.Exploit.Shellcode.3.B9F39F33	GData	Generic.Exploit.Shellcode.3.B9F39F33
Kaspersky	HEUR:Trojan.Win32.Generic	MAX	Malware (ai Score=99)
McAfee	Artemis!F89795C78D7A	McAfee-GW-Edition	Artemis
Microsoft	Trojan:Win32/Meterpreter.gen!E	Palo Alto Networks	Generic.ml
Rising	HackTool.Swrort!1.6477 (CLASSIC)	Symantec	Trojan.Gen.MBT
ZoneAlarm by Check Point	HEUR:Trojan.Win32.Generic	Acronis	Undetected
AhnLab-V3	Undetected	Antiy-AVL	Undetected

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(d) Rust compiled executable with the malicious payload.


549b35193cb6ecc23168df15198d45985e1601b8bdff5c882956daf6b20644a9

16 / 68

16 engines detected this file

549b35193cb6ecc23168df15198d45985e1601b8bdff5c882956daf6b20644a9
go.exe
64bits assembly direct-cpu-clock-access peexe runtime-modules

2.04 MB Size

2021-01-16 13:35:47 UTC 1 day ago

DETECTION DETAILS BEHAVIOR COMMUNITY

Ad-Aware	Generic.Exploit.Shellcode.3.C2AA2515	ALYac	Generic.Exploit.Shellcode.3.C2AA2515
SecureAge APEX	Malicious	Arcabit	Generic.Exploit.Shellcode.3.C2AA2515
BitDefender	Generic.Exploit.Shellcode.3.C2AA2515	Cybereason	Malicious.690e2b
Cynet	Malicious (score: 100)	Emsisoft	Generic.Exploit.Shellcode.3.C2AA2515 (B)
eScan	Generic.Exploit.Shellcode.3.C2AA2515	FireEye	Generic.Exploit.Shellcode.3.C2AA2515
GData	Generic.Exploit.Shellcode.3.C2AA2515	Ikarus	Trojan.Win64.Rozena
Jiangmin	Trojan.Shelma.geq	MAX	Malware (ai Score=85)
Microsoft	Trojan:Win32/Meterpreter.gen!E	Rising	HackTool.Swrort!1.6477 (CLASSIC)
Acronis	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
Antiy-AVL	Undetected	Avast	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefenderTheta	Undetected	Bkav	Undetected

We use cookies and related technologies to remember user preferences, for security, to analyse our traffic, and to enable website functionality. Learn more about cookies in our Privacy Policy.

(e) Go compiled executable with the malicious payload.

Figure 4: VT detection results for binaries from various languages.

if this is not the case, then some automated script opened the binary for inspection.

Notably, up to this point, no flag is triggered to the sandbox as the checks do not perform any black-listed operation and are considered benign by most of them. If all these checks are passed, then we start the registry checks via WMIC for the CPU model name. We validate with our dictionary the existence of the model and the consistency of the reported threads and cores with the manufacturer's actual ones. Usually, this query to the registry is logged by the sandbox, but without any significant alert. Lastly, we query the registry, again via WMIC, to access system information and find known VM strings in the system model or system manufacturer. Clearly, this is also logged by the sandboxes, without though any high score to issue a malicious verdict. Moreover, not all sandboxes managed to reach this point of execution, so in many instances, these logs were not complete in all of the reports.

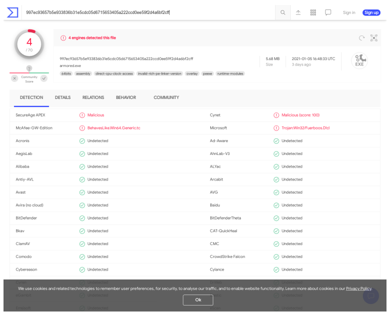
If any of these checks fail, we perform a graceful exit, perform some arbitrary computations beforehand, and add some noise in the analysis. However, after the successful pass of the aforementioned checks, the malicious binary is executed. Quite alarmingly, in all tested sandboxes, our evasion methods succeeded, achieving low scores in both the dynamic analysis, as well as the static analysis offered by the sandboxes. In fact, all of them considered the samples suspicious for spawning another process of itself which can be considered a false-positive indication, but the malicious payload was not delivered as the binary understood that it was executed in a sandbox.

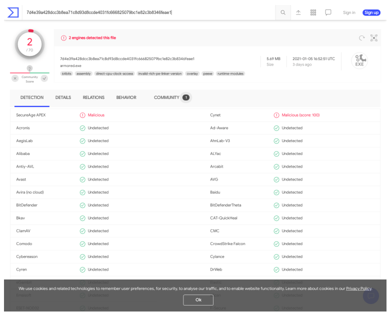
6 DISCUSSION

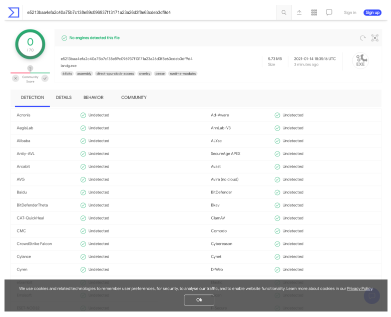
Given the inherent static analysis restrictions, low detection rate from AVs in VT can be considered up to a point expected as our approach is unique and creates an unknown pattern. Nevertheless, the fact that our samples do not simply have few detections, but actually zero is very alarming. It becomes even more worrying because PyInstaller is a widely used tool that is poorly handled. Even the slight changes introduced by us significantly reduced the AVs' detection rate. Notably, these methods can be applied to other languages' packaging, e.g. for Go which is increasingly being used by malware in the past few years[3].

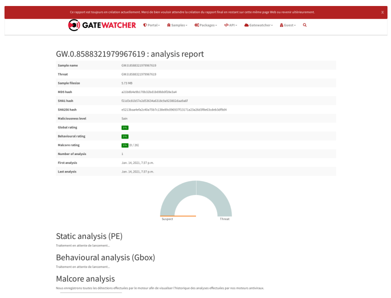
It is worth noticing that the above results indicate

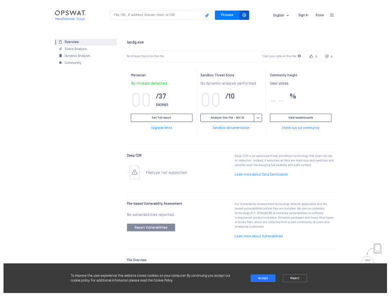
[3]https://unit42.paloaltonetworks.com/the-gopher-in-the-room-analysis-of-golang-malware-in-the-wild/


(a) Patched binary with PyInstaller strings removed.


(b) Patched binary with version fix.


(c) Final binary with all patches.


(d) GATEWATCHER scan results.


(e) OPSWAT scan results.

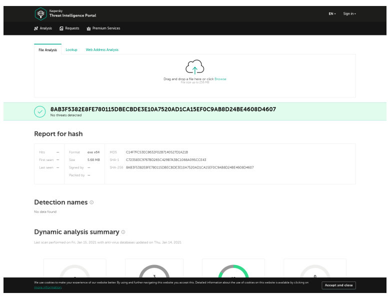

(f) Kaspersky static and dynamic analysis results.

Figure 5: Screenshots of the results of our patched samples from multi-engine scanners.

that AVs do not efficiently handle large executables. For instance, using the UPX feature of PyInstaller to shrink the executable resulted in further detections of the binary. Nevertheless, this can be attributed to the UPX signature. However, the same behaviour was noticed with, e.g. Nuitka (https://nuitka.net/) which created far larger executables.

The results of the dynamic sandbox analysis can be considered in many cases, catastrophic. The reason is that our analysis showcases significant issues in the configuration of the sandboxes that allow the malware to fall below their radar. For instance, the vast majority of sandboxes expose inconsistent CPU specifications (processor name vs cores/CPU) while we also noticed the use of non-existing CPU names in one of them. Similar issues were also detected for GPUs.

Differences between CPU timestamp counters may be more challenging to patch; therefore, they were encountered in most sandboxes. Quite interestingly, the listing of well-known VM processes and obvious VM related strings in Bios and system manufacturer (e.g. QEMU, KVM), small uptime, MAC address vendor and low RAM, trivially exposed the virtualisation environment indicating a poor configuration of the sandbox environment. Moreover, we argue that using a limited set of product Windows IDs that we noticed can also be used to fingerprint sandboxes and bypass them. Therefore, the further randomisation of these IDs is necessary as the purchase of more licences does not solve the problem completely.

Finally, we should also stress the complete absence of foreground processes in all sandboxes. In all occasions, the binary started without any other window opened, clearly showing that a dropper initiated the execution. While one may argue that malware may consider this as part of its persistence, e.g. via registry autorun, it would be relatively easy for the malware to verify the claim and correlate it with the uptime. Therefore, sandboxes must open a couple of windows, e.g. Explorer, to denote some user-initiated action for the binary execution and hide the dropper's existence.

7 CONCLUSIONS

Many issues arise from misclassifications and it is essential to understand which features are the ones that resulted to, e.g. a false positive. Based on this problematic, we studied the case of PyInstaller, a widely used packaging tool for Python scripts. The generated executables are erroneously flagged as malicious regardless of their content, as repeatedly reported online by developers. While many malware authors have

Table 2: Identified misconfigurations and issues of sandbox environments.

Rules	Hybrid Analysis	Any.run	Intezer	Tria.ge	Cuckoo	Thread Grid	Cape
Non existing CPU name				✗			
Bad CPU specifications	✗		✗	✗	✗	✗	✗
Product Key reuse		✗		✗	✗		
Bad GPU name	✗	✗	✗	✗		✗	
Low Available Memory	✗						
Running known VM processes		✗					
Small uptime			✗				
Difference between CPU timestamp counters (rdtsc)	✗			✗	✗	✗	✗
Bad MAC Address	✗						
Hypervisor bit in CPUId					✗		
Known VM brand string in Bios					✗		
Known VM brand string in System Manufacturer					✗		

recently switched to the use of PyInstaller to write their malware, this does not justify why every executable of PyInstaller should be treated as malicious. On the contrary, it implies that AVs do not understand the content of these files and treat them as malicious. Based on this problematic, we have shown that the problem is inherent as AVs cannot efficiently process Python bytecode, which are included in PyInstaller. As a result, we may develop malware which escapes static analysis of all AVs by simply changing some characteristics of PyInstaller binaries. Clearly, Python bytecode decompilation is essential to prevent similar attacks in the near future.

Based on our analysis, it is evident that apart from clear misconfigurations, resource-wise limitations in the sandboxes impose significant constraints that enable their identification. More precisely, to address the numerous requests for scanning binaries, many of the sandboxes resort to using a limited set of resources (CPU/RAM) which especially for the CPU is not properly handled. As illustrated, many of them report contradictory configurations which can be easily detected and bypassed without issuing any significant alert. The analysis of a binary in a virtualised environment which resembles a traditional, modern PC system is very costly, let alone bear metal analysis. Nevertheless, with the continuous increase of samples that have to be checked, the balance is going to be significantly tipped at the dispense of sandboxes. The latter denotes a definite need to improve our existing sandboxes' capabilities to, e.g. enable them to report more realistic configurations without exposing them. Moreover, we should further explore the analysis using symbolic execution of the binary to offer a cost-efficient alternative. Finally, despite the recent advances in malware analysis and the numerous academic works and products touting almost absolute detection rates, we illustrate that undetectable malware might even be in plain sight and evade detection in real-world experiments.

We argue that one can deploy even stealthier malware by minimising the filesystem footprint. To this end, in future work we plan to rewrite the bootloader to extract all the necessary files in memory or use PyOxidizer (https://github.com/indygreg/PyOxidizer), randomising file names in each compilation, further reducing the pattern that one could use to trace it. Fileless approaches (Kumar et al., 2020) in which all the content is loaded in memory through the use of, e.g. Living Off The Land Binaries And Scripts (LOLBins and LOLScripts https://github.com/LOLBAS-Project/LOLBAS) can further decrease the detectability. In parallel, we plan to investigate other packaging and distribution tools for other languages beyond Python to assess their obfuscation abilities.

ACKNOWLEDGEMENTS

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the projects CyberSec4Europe (https://www.cybersec4europe.eu) (Grant Agreement no. 830929), *LOCARD* (https://locard.eu) (Grant Agreement no. 832735).

The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

REFERENCES

Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. (2019). Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28.

Apostolopoulos, T., Katos, V., Choo, K. R., and Patsakis, C. (2021). Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems*, 116:393–405.

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *Blackhat USA*.

Bulazel, A. and Yener, B. (2017). A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, page 2, New York, NY, USA. ACM, ACM.

Checkpoint Research (2020). Evasion techniques. https://evasions.checkpoint.com/.

Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186. IEEE, IEEE.

Cono D'Elia, D., Coppa, E., Palmaro, F., and Cavallaro, L. (2020). On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security*, 15:2750–2765.

D'Elia, D. C., Coppa, E., Nicchi, S., Palmaro, F., and Cavallaro, L. (2019). Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 15–27.

Deng, X. and Mirkovic, J. (2018). Malware analysis through high-level behavior. In *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*, Baltimore, MD. USENIX Association.

Forum, W. E. (2020). Wild wide web consequences of digital fragmentation. https://reports.weforum.org/global-risks-report-2020/wild-wide-web/.

Gandotra, E., Bansal, D., and Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 2014.

Guan, L., Jia, S., Chen, B., Zhang, F., Luo, B., Lin, J., Liu, P., Xing, X., and Xia, L. (2017). Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 339–349, New York, NY, USA. ACM, ACM.

Huang, Q., Li, H., He, Y., Tai, J., and Jia, X. (2020). Pidicators: An efficient artifact to detect various vms. In *International Conference on Information and Communications Security*, pages 259–275. Springer.

(IC3), I. C. C. C. (2019). 2019 internet crime report. https://pdf.ic3.gov/2019_IC3Report.pdf.

Issa, A. (2012). Anti-virtual machines and emulations. *Journal in Computer Virology*, 8(4):141–149.

Kirat, D. and Vigna, G. (2015). Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780, New York, NY, USA. ACM, ACM.

Kirat, D., Vigna, G., and Kruegel, C. (2011). Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412, New York, NY, USA. ACM, ACM.

Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium*, pages 287–301, Berkeley, CA, USA. USENIX Association.

Küchler, A., Mantovani, A., Han, Y., Bilge, L., and Balzarotti, D. (2021). Does every second count? time-based evolution of malware behavior in sandboxes. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society.

Kumar, S. et al. (2020). An emerging threat fileless malware: a survey and research challenges. *Cybersecurity*, 3(1):1–12.

Leguesse, Y., Vella, M., and Ellul, J. (2017). Androneo: Hardening android malware sandboxes by predicting evasion heuristics. In *IFIP International Conference on Information Security Theory and Practice*, pages 140–152, Cham. Springer, Springer International Publishing.

Liţă, C. V., Cosovan, D., and Gavriluţ, D. (2018). Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in upa packers. *Journal of Computer Virology and Hacking Techniques*, 14(2):107–126.

Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 261–272, New York, NY, USA. ACM.

Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., and Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80, New York, NY, USA. ACM, ACM.

Or-Meir, O., Nissim, N., Elovici, Y., and Rokach, L. (2019). Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. (2014). Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA. ACM.

Shi, H., Alwabel, A., and Mirkovic, J. (2014). Cardinal pill testing of system virtual machines. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 271–285, San Diego, CA.

Shi, H., Mirkovic, J., and Alwabel, A. (2017). Handling anti-virtual machine techniques in malicious software. *ACM Transactions on Privacy and Security (TOPS)*, 21(1):2:1–2:31.

Thomas, D. S. (2020). Cybercrime losses: An examination of us manufacturing and the total economy.

Uitto, J., Rauti, S., Laurén, S., and Leppänen, V. (2017). A survey on anti-honeypot and anti-introspection methods. In *World Conference on Information Systems and Technologies*, pages 125–134. Springer.

Veerappan, C. S., Keong, P. L. K., Tang, Z., and Tan, F. (2018). Taxonomy on malware evasion countermeasures techniques. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 558–563. IEEE.

Webster, G. D., Kolosnjaji, B., von Pentz, C., Kirsch, J., Hanif, Z. D., Zarras, A., and Eckert, C. (2017). Finding the needle: A study of the pe32 rich header and respective malware triage. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 119–138. Springer.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., et al. (2016). Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187, Cham. Springer, Springer International Publishing.