CSCI 43200 Final Project Report

By
Chip Simmerman & Wayne Pham
4/20/2022

Introduction:

The intention behind choosing this project was to be able to extend and apply concepts relevant to the information learned in the course. This project was also intended to allow us to indulge the perspective of an attacker while in a controlled environment. As it will be explained later in this report, our project was to create an advanced keylogger and simple antivirus to apply and demonstrate some of the ideas that were covered in class. We will continue this discussion by first covering the high-level background details of the project before going into the technical implementation, testing, and finally a summary of our findings. As a quick side-note, our perception and goals for this project evolved as we became more familiar with our capabilities, so our final product consists mainly of applications rather than research. Therefore, we intend to present our software to demonstrate our abilities and supplement written material in this report for parts that we found to be outside of the scope of the project. Screenshots and demonstration(s) can be found at the end of this document before our listed references.

Background:

We initially knew that the objective of this project was to allow us to explore ideas covered in class more specifically. Since most topics taught in this course seemed to lean towards the perspective of defenders, we decided that it would be interesting to put emphasis on the perspective of an attacker in addition to considering the perspective of a defender. Additionally, we wanted to work on something interactive that would allow us to demonstrate and represent the information learned throughout the course of the project. We originally wanted to attempt a “cat and mouse” game where one of us took on the challenge of constructing a “harmless” but armored piece of malware while the other constructed an anti-malware program to defend. Chip took on the work of creating the keylogger while Wayne explored creating an antivirus program.

While we will cover this in a later section, we eventually found that there is an imbalance in difficulty and documentation between the two parts of our project, and therefore our idea for having the project act as a competition became a bit unrealistic. Nonetheless, we have still accomplished the overall goal of the project and explored this area of security in further detail than covered in the course. Interestingly enough, quite a few parts of the work that Chip completed on the keylogger were coincidentally covered in later sections of the course, which solidified our understanding and faith that our project added relevant value to our learning. In the next section, we will continue by explaining how we implemented our malware and anti-malware programs while also covering some further details we learned along the way.

For the antivirus part of the project, the objective of this “simple antivirus” was to make something that can detect the custom keylogger. On the surface, it seems like a straightforward process. However, once the project went into more detail, the more

Wayne realized that building an antivirus, even a simple albeit stupid one, is much more difficult than previously anticipated.

The first thing that comes to mind is to look for online resources as a guide on how to start making an antivirus. While there are quite a few videos on it. The contents of those videos were completely useless. These videos consist of a Windows Batch script to make it look like that a virus has been detected and display it on the command prompt. Although it looks and feels cool to the average viewer, it is a waste of time for me since Wayne wanted something more tangible than a façade. With that in mind, I realized that the only way to make an antivirus was to read up on how an antivirus truly works.

Other online resources were somewhat useful like security blogs and knowledge bases from the major software security firms like Kaspersky and Sophos. But they were mainly guidelines on how to approach an antivirus solution. Wayne also looked back at the class slides from previous lectures on malware as well. From the slides it describes that Signature scans are essentially “fingerprints” based on samples. The program should detect these fingerprints and then clean or quarantine on request. But the most important aspect for signature-based scanning is to keep the signature database up to date. Since new threats emerge every second, having a database up to date offers the most protection.

Implementation:

First, it’s worth noting that the type of attack we intended to emulate would be in the reconnaissance phase of a larger goal in which the program is either ran manually on a victim’s machine without them knowing, or a victim is curious enough to attempt to run the program on their own. Our keylogger does not tamper with anything on the victim’s machine and has the sole purpose of grabbing as much information as possible and shipping it off remotely for us to see.

The keylogger that Chip built was made using an assortment of libraries in Python. We chose this language because we found there was a lot of documentation and resources online ranging from simple, one-function programs to more advanced keyloggers. Even then, we did not know just how much work this project would take, so we started by writing programs that used libraries like pyhook and pywin32 to capture keystrokes and write them to a text file. We quickly realized that writing a barebones keylogger in Python was as simple as writing any other Python script with a few I/O operations. This led to us expanding upon the keylogger to add more functionality, which in turn enabled us to incorporate new ideas such as file encryption, code obfuscation, and digital signatures. All of the work for our keylogger was completed with the help of Python libraries and online tutorials, of which there is a multitude.

Chip found that there are an astounding number of libraries available in Python that can accomplish just about anything an adversary would want spyware to do. For a couple examples, the keylogger uses pynput to record keystrokes, scipy.io.wavfile to write audio, cryptography.fernet to encrypt files, and SMTP/MIME to send files via email. By adding these libraries and functions to our keylogger, we were able to utilize and experience many different concepts which we saw in class such as SMTP/MIME, AES/SHA256 symmetric encryption, and code obfuscation.

The keylogger was built function by function for it to be modular and easy to work with. First, we have 2 functions which detect when a key is pressed or released. When a key is pressed, we add the key to a list and pass it to another function which writes it to a text file. Before doing so, we filter certain keys like space, backspace, and escape so that we don't have nonsense appended to the file when a special key is pressed. After every so often, we also append a newline to the file in an attempt to keep our output organized. When the key is released, it checks how long the program has been running. After 10 seconds, the program will wrap up all of the files it has collected and send them off via email to a Gmail account we set up for the purpose of this project. Before time has run out, the keylogger will also carry out four more functions to grab information from the victim's machine.

While still grabbing input from the user's keyboard, the keylogger will also grab the system's information including public IP address, private IP address, host name, processor, operating system and version, and machine manufacturer/model. This information, while relatively simple, could be useful for an attacker to gain a better understanding of the machine they're dealing with. After gathering system information, the program will then attempt to grab anything that is copied to the user's clipboard so long as it is text-based. This realistically could grab just about anything that is sensitive, such as usernames and passwords copied from a password manager, recently-used URLs, etc.

Moving on, the keylogger will take a screenshot of what is currently on the user's screen at the time the program is ran, which could obviously contain any sort of useful information both visual and textual. Lastly, the program will access the microphone on the device to record audio before writing it to a .wav file. These functions carry on for up to 10 seconds, which was the time set to make debugging easier. When the timer has expired, the program will take each of the output files and replace them with their encrypted and renamed counterparts. These are the files that then get sent to us via email. In the span of 10 seconds, we can gather all of this information into 5 files, encrypt them using an AES symmetric algorithm, and send them via email. Once we have received the transmitted files, we can download them from our inbox and decrypt them with our private key.

The program is designed to run off of a USB device so nothing will ever be stored on the victim's machine. One might wonder why we go through the hassle of encrypting the files if we're the ones that are trying to receive them. The main reason is to hide our intentions from the user and anyone monitoring network traffic. In a real-world scenario it may seem suspicious or noisy to be consistently emailing a certain number of files to an unknown domain throughout the day, so encrypting the files after gathering the information would help prevent defenders from seeing the information we're gathering. In addition, Chip made the decision to leave the encrypted files on the USB in the case that the target machine is unable to send the encrypted files and needs to be recovered manually. This would not likely be the case, but it is a decision that would need to be considered in a real scenario. One small detail we'd like to include is that the encrypted files were then given names that may seem like legitimate system files to the untrained eye. For example, the executable that the keylogger is packaged with was given the name "WatchDogAntivirus.exe" and we even went so far as to change the .ico icon to that of a blacked-out guard dog. Subsequently, each file that was encrypted matched

the WatchDogAntivirus aesthetic. WatchDogAntivirus.txt holds the keyboard input, WatchDogAntivirusDrivers.txt holds the system information, WatchDogAntivirusSignatureDatabase holds the clipboard information, WatchDogAntivirusAlert.wav holds the microphone recording, and WatchDogAntivirusLogo.png holds the screenshot.

In a further attempt to hide our intentions, we added code obfuscation through pyarmor which uses an algorithm to garble up the code so that it is no longer human (or even machine) readable. Pyarmor first obfuscates your Python code and then uses pyinstaller to package the code and its dependencies into one .exe file. This obfuscation means that when the executable is ran it is much more difficult for anti-malware software to flag it solely based on the instructions the program uses, since the antivirus should only know what instructions the program is executing after it has already executed them. As a final measure to reduce the likelihood of our keylogger being quarantined, Chip created a self-signing certificate which he used to digitally sign the executable. As we have learned in class, this will likely still result in a warning for programs that see the certificate is not on a trusted list, but it should still pass shallow tests.

In the implementation of signature-based scanning, the first thing to sort out is how we are going to replicate a virus “fingerprint”. A fingerprint in a traditional sense is uniquely identifiable. So, the implementation needs to have something that will guarantee uniqueness. The first thing that comes to mind was to use SHA signatures since those can be unique within a list of thousands of known signatures. MD5, and SHA256 would also work too, and we can incorporate that later once the algorithm is figured out. For the algorithm, we will need a scan function that takes in a file via absolute file path. After the scan function receives the absolute file path, it will open the file and read it in bytes. The function will take the file in byte format and convert it into a readable hash. In this case, it’s readable MD5, SHA1, and SHA256. Then, the program takes the readable hash and compares it to an extensive list of known signatures. If any of the MD5, SHA1, or SHA256 signatures in the list matches with the readable hash that was generated previously, quarantine the file, and then remove it. There’s also plans on implementing heuristic scans as well but with how much time there’s left for the project, the prospects of implementing this particular type of detection is slim.

Testing and Results:

Testing the keylogger was actually more simple than we expected, although there were a few hiccups along the way. As Chip built the keylogger, he also took the initiative to test it as well. Because the keylogger is technically malware, we originally planned to run it on and older device one of us had laying around. As development progressed, we realized that it wasn’t absolutely necessary since the functionality of the keylogger doesn't involve changing anything on the target device. Therefore, Chip tested the program on three separate devices running Windows: An older Acer laptop, his current ASUS laptop, and his custom-built desktop PC. The program worked flawlessly for the way it was written on all three devices. We didn’t test it on any devices running Linux or MacOS because some of the libraries we used are specific to Windows like win32gui and win32clipboard. Nonetheless, there are still workarounds for these functions,

although it would have taken more time to complete and wouldn't be necessary to demonstrate the functionality we desired.

One key item to note is that we originally planned to be able to automatically offload and run the program from a USB device as soon as it was plugged into a machine. Unfortunately, this capability is not as easy to pull off as it would seem and typically requires us to have one of a few specific USB models that have a vulnerability in their microcontroller. After looking into how these "bad USBs" are created, Chip found that these exploitable USBs are re-formatted using specific tools to rewrite the instructions that the USB microcontroller can execute so that it emulates a Human Interface Device (HID) instead of a USB. When done correctly, a machine will recognize the bad USB as a HID virtual keyboard, which allows the microcontroller to execute shell commands to run whatever program it needs to. Unfortunately, we do not possess an exploitable USB or the tools to exploit one and the most notable company Hak4 that sells them tends to charge upwards of $100 depending on the features you require.

Consequently, we decided to add a "autorun.inf" configuration file to the USB which tells a device with autorun already enabled to run the executable as soon as the USB is inserted. Of course, the target has to be pre-configured to have autorun enabled but it is still good enough to demonstrate proof-of-concept for us. With autorun enabled on one of our devices, we can then insert the USB with the keylogger on it. Our device will recognize the autorun.inf file, execute the rules set within, and run the keylogger. The keylogger will briefly open a command window before closing it a split-second later to show it ran and then it begins the functions outlined in the implementation section. Therefore, we were still able to show how our ideal attack would be carried out assuming we had access to the resources required.

Another hiccup we encountered during testing was while Chip was actually writing the code for the Python script. As we noted in previous updates, our Windows Defender antivirus program constantly flags the .py file as a trojan and quarantines it whenever we open it in VSCode to edit it. This has been obviously frustrating to deal with and is an example of how over-tuned security mechanisms can be too aggressive to where they actually impact the performance of trustworthy items. Eventually we were able to overcome this through obfuscation and adding a digital signature to the packaged executable, and now Windows Defender will no longer quarantine it.

Speaking of antivirus interactions, Chip actually uploaded the keylogger executable to VirusTotal, which we were later introduced to in class. He first uploaded the unobfuscated and unsigned executable. Surprisingly, it was only detected by 3/66 security vendors, although it was flagged specifically as a keylogger. Next, he uploaded an unsigned version after it was obfuscated and packaged with pyarmor. The results were that only 2/68 security vendors flagged it, and no sandboxes flagged it. Interestingly enough, neither vendor could identify it, so they labeled it as simply malicious/generic trojan. Finally, Chip uploaded the finished product with a digital signature. This resulted in 1/67 security vendors flagging it as a generic trojan. When looking into it, this was triggered specifically because the executable was packaged using pyinstaller. Pyinstaller is one of the most popular tools used to package Python scripts, and so certain antivirus software include rules that flag a program as malicious if it was packaged with pyinstaller. Of course, this is another example of an over-aggressive security feature that can potentially block trustworthy activity from

getting through and seems to still be an issue in general. Additionally, VirusTotal did state that a certificate chain was processed, but it terminated in a root certificate that is not trusted by the trust provider (as expected). It actually included that the signer was “WatchDog Antivirus” and the name of the X.509 certificate was “WatchDog Antivirus”. All of these VirusTotal results were screenshotted and we’ve put them at the end of this report.

On the simple antivirus side, the algorithm surprisingly works and it was able to pick up the unprotected file once the signature was added to the database. As seen in the first demo picture, the entire process starts with scanning the SHA1 signatures to see if any of the signatures in the database matches with the readable hash generated from the given file. If there is a match, the program alerts the user with a message and proceeds to quarantine the file. If a match was not found, the program will then look at a different set of signatures. In this case, the next set of signatures would be MD5 signatures. The process then repeats like the SHA1 scanning function before it. If the program finds a matching signature, it alerts the end user and then proceeds to quarantine the file. If there’s no match to any known MD5 signatures, the program moves to the last set of signatures to scan, which is SHA256 signatures. Likewise, if there’s a match, then it alerts the user and quarantines the file. However, If there is not a match this time, then the program will stop and say that the given file is safe. This presents a flaw in the detection process since malware writers can easily get around this by signing a different signature than the one that is currently located in the database. This is shown in the demo video with the more robust keylogger “WatchDogAntivirus.exe” as it was able to escape detection since it has a different SHA256 signature than the “WatchDogAntivirusUnprotected.exe” keylogger. Computers are not smart enough to recognize that the protected keylogger was in fact the same as the unprotected version since the SHA256 signatures do not match with any in the current state of the database. This illustrates the importance of keeping the signature database up to date since new threats can emerge quickly.

It is also important to note that the implementation of the simple antivirus uses 2 different files to store signatures: A text file and a JSON file. The signatures that are present were found online and were associated with their own scanning functions in the program. This could be much more improved by using a signature API instead of a local database given more time. This will effectively decouple the responsibility of keeping the signature database up to date to a third party service and make scanning more effective since the program would be working with up to date data. As previously shown by Chip, the more robust “WatchDogAntivirus.exe” was a bit harder to detect but it is not impossible. With more signature data to work with, there’s a higher chance that the program would have detected and quarantined the obfuscated keylogger.

Summary and Future Work:

The result of this project was an advanced keylogger completed by Chip and a simplistic antivirus completed by Wayne. The keylogger utilizes many of the ideas that we were introduced to in class, including symmetric key encryption, code obfuscation, certificates and digital signatures, and the use of application-layer protocols. It was very interesting to see just how simple it was to create a python script that can grab a large

amount of data about a target in the span of just a few seconds. Chip found that the real difficulty resided in creating a program that generated as little noise as possible. As we were taught in lecture, an attacker must be 100% perfect after they have infiltrated our defenses. This led to the addition of measures to make the spyware seem as legitimate as possible, and we are pleased to report that very few anti-malware systems have been able to both flag the keylogger as malicious and correctly identify what it is. The keylogger in its final form is about as protected as we believe it can be given our current resources. If we were to continue with this project in the future, it would be very exciting to be able to recreate our simulated attack with the use of an exploited USB designed to run a payload automatically. As explained before, this usually requires a decent amount of effort to create one from scratch, or a decent financial investment into one that has been created by a third party. As it stands right now, we have at the very least been able to simulate how it would work if we had access to a bad USB through the use of enabling autorun and including an autorun.inf configuration file on the drive.

Originally, Wayne wanted to implement heuristic scans as well, but he ran into some problems regarding implementation. A proper heuristic scan would have to read a stream of network traffic and system traffic to detect typical virus behavior, such as writing to an executable or executing multiple system calls involving deleting or encrypting files. This requires the use of a low-level programming and a system of weights for certain actions. But, since the amount of time for the project is getting more limited, heuristic scanning would be an example of work that can be added in the future to the antivirus. If we were given more time for this project, Wayne would have overhauled the Python program to a C++ program to take advantage of C++ capabilities as a systems programming language and its efficiency. Then the program could run as a service under the operating system to periodically scan and verify system traffic to spot typical virus behavior. There would also be a series of weights for these typical activities since there's a chance for a false positive if the system was too aggressive in its actions to flag and quarantine the malicious actor.

Overall both of us learned a lot about how computer security systems work in practice and how it's sometimes really easy to implement an attack vector. While at the same time, Wayne learned how difficult it is to defend a system against it.

Screenshots and Demos

Keylogger Demonstration:

https://youtu.be/LRMnsBsze2A

Files:

https://drive.google.com/drive/folders/1l2tV9aWn0VWG-35oADh9YkKcIR3P6YZE?usp=sharing

Windows Defender Flagging as “Low priority”

Program:Win32/Beareuws.A!ml

Alert level: Low
Status: Active
Date: 4/12/2022 11:44 AM
Category: Potentially Unwanted Software
Details: This program has potentially unwanted behavior.

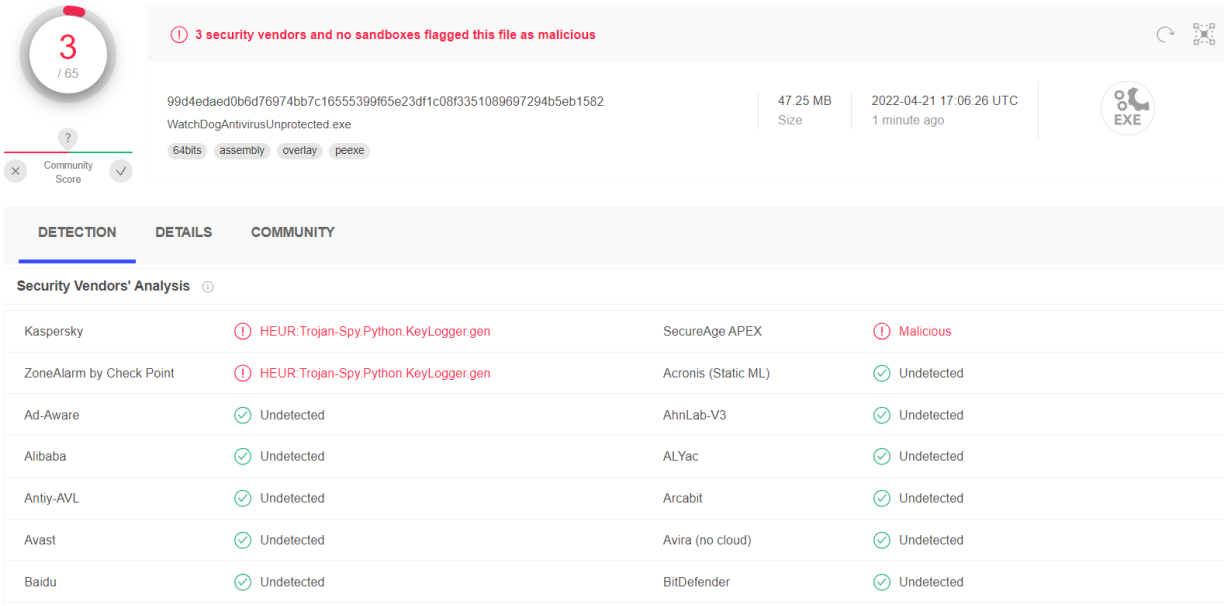
Learn more

Affected items:
file: C:\432SecurityProject\432-Security-Final-Project\ForOffloading\WatchDogAntivirus.exe

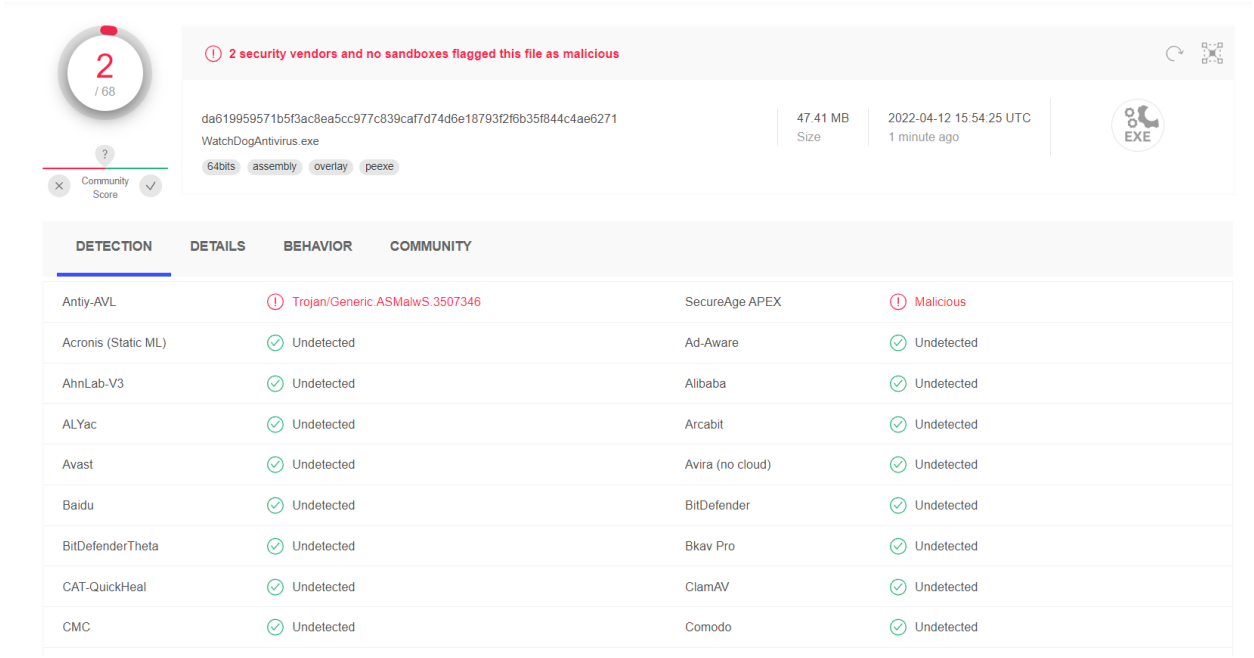
file: D:\WatchDogAntivirus.exe

OK

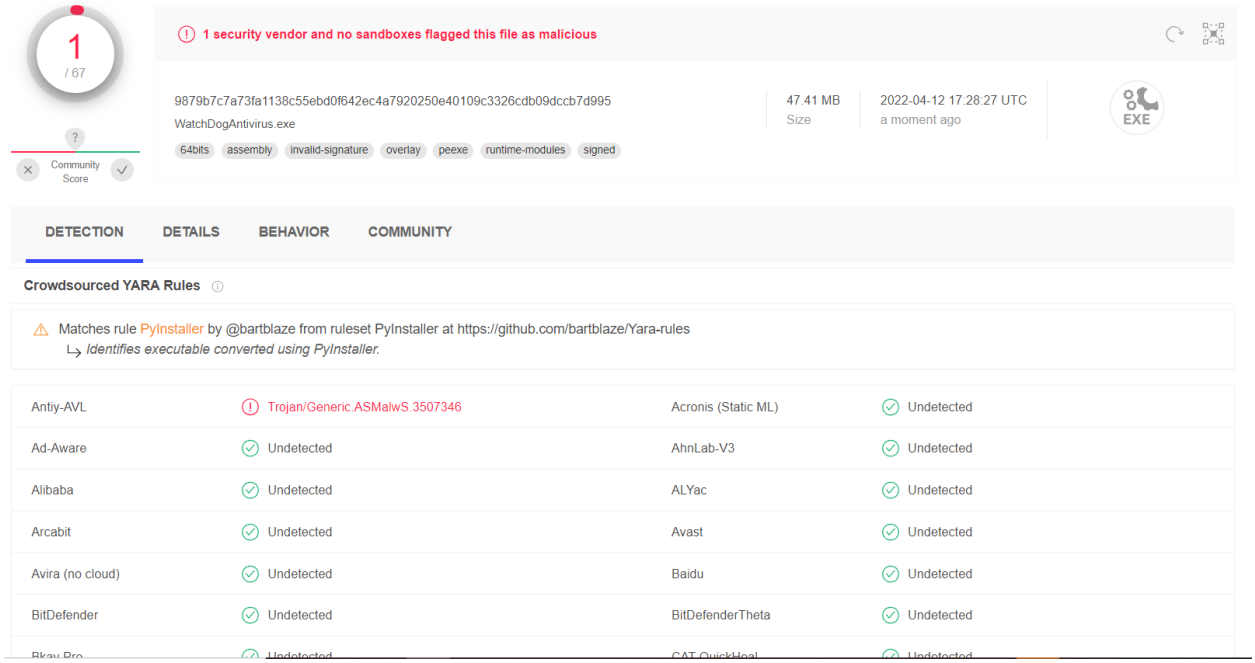
VirusTotal Unsigned/Unobfuscated Executable Results



VirusTotal Unsigned Executable Results



VirusTotal Signed Executable Results



Signature Info

Signature Verification

⚠ A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.

File Version Information

Signers

+ WatchDog Antivirus

X509 Certificates

+ WatchDog Antivirus

VirusTotal Pyinstaller Rule

Ruleset: PyInstaller

https://github.com/bartblaze/Yara-rules

Go to matched rule Copy ruleset

```
import "hash"
import "pe"

rule PyInstaller
{
    meta:
        id = "6Pyq57uDDAEHbltmbp7xRT"
        fingerprint = "ae849936b19be3eb491d658026b252c2f72dcb3c07c6bddecb7f72ad74903e
        version = "1.0"
        creation_date = "2020-01-01"
        first_imported = "2021-12-30"
        last_modified = "2021-12-30"
        status = "RELEASED"
        sharing = "TLP:WHITE"
        source = "BARTBLAZE"
        author = "@bartblaze"
        description = "Identifies executable converted using PyInstaller."
        category = "MALWARE"

    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_resource
}
```

Simple Antivirus Demo

https://drive.google.com/file/d/1F8odD0dXNd_QKrELCbSVdL2TTEtlx3VO/view?usp=sharing

Main program

```
runLoop = True

while runLoop:
    menu()
    userInput = input('Select an option above: ')

    if userInput == '1':
        file = input('Enter an absolute file path to begin scanning: ')
        scan(file)
        runLoop = True

    elif userInput == '2':
        print('Exiting program')
        runLoop = False

    else:
        print('Not a valid input.')
        runLoop = True
```

SHA256 Scanning Function:

```
def scanSHA256(file):
    virusDetected = False

    with open(file, "rb") as file:
        bytes = file.read()
        readableHash = hashlib.sha256(bytes).hexdigest()
        print("SHA256 Signature for this file is: " + readableHash)

    with open("SHA256.txt", 'r') as signatures:
        lines = [line.rstrip() for line in signatures]

        for line in lines:
            if str(readableHash) == str(line.split(";")[0]):
                virusDetected = True

        signatures.close()

    if virusDetected == False:
        print("No known threat is found in this file.")
    else:
        print("A known threat has been detected! File has been quarentined.")
        os.remove(file)
```

References

Awasthi, Dhanishtha. “Antivirus Evasion: Bypass Techniques.” Medium, Medium, 5 Aug. 2020, offs3cg33k.medium.com/antivirus-evasion-bypass-techniques-b547cc51c371.

“Build Software Better, Together.” GitHub, github.com/topics/bypass-antivirus.

Collins, Grant. Create an Advanced Keylogger in Python - Youtube. 28 Mar. 2020, www.youtube.com/watch?v=25um032xgrw.

“Design a Keylogger in Python.” GeeksforGeeks, 10 Aug. 2021, www.geeksforgeeks.org/design-a-keylogger-in-python/.

Hackcat, Written by. “Homemade Keylogger. Writing an Undetectable Keylogger in C#.” HackMag, hackmag.com/coding/diy-keylogger/.

Nick. “How I Made an Advanced Python Keylogger That Sends Emails.” Cybr, 30 Nov. 2021, cybr.com/ethical-hacking-archives/how-i-made-a-python-keylogger-that-sends-emails/.

Shankle, Dexter. “Common Antivirus Bypass Techniques.” LMG Security, 17 Mar. 2021, www.lmgsecurity.com/common-antivirus-bypass-techniques/.

Wells, Roger. “How to Make a Windows Keylogger by Yourself - Dzone Devops.” Dzone.com, DZone, 10 Nov. 2020, dzone.com/articles/how-to-make-a-windows-keylogger-by-yourself.

xp4xbox, and Instructables. “Simple Keylogger.” Instructables, Instructables, 25 July 2020, www.instructables.com/Simple-Keylogger-Python/.

Norman, Alexandre. “pyClamd : Clamav with python.” *pyClamd*, xael.org/pages/pyclamd-en.html. Accessed 22 Apr. 2022.

SophosLabs. “Dangers Of Virus Signature Checksum.” *Naked Security*, 22 Oct. 2010, nakedsecurity.sophos.com/2010/01/17/dangers-virus-signature-checksum.