Davis Garrett (dmg@colostate.edu) & Evan Tone (evan.tone@colostate.edu)

CS 370-001

Shrideep Pallickara

7 May 2025

# TP-D3 Report

How do you justify using a pi? This was the driving question behind our collective project for the semester, a question which held far more nuance than we could have ever expected, and throughout the latter half of the semester, our team has put our minds into answering this. The problem stems from one singular issue: why would your program benefit from running externally, rather than on a normal desktop or virtual machine? The raspberry pi is a neat little unit, but it doesn't have spectacular processing power, nor does it inherently feature hardware foreign to what any consumer PC on the market offers. It may be portable, but so is a laptop, and while it may allow you to run an OS different from your computer, so too does a VM. Ultimately, our initial project started with two prospects to solve this question.

One: The creator wants to benefit from additional computation power. While the raspberry pi is certainly a tiny machine, it is completely separate from other hardware, meaning it can be used to the fullest extent without interfering with any processes on another computer. This very reasoning, however, is why we didn't elect to explore this option. While a machine dedicated to running a program is intriguing, there are certainly uses for such operations, to justify it we argue the program would have to run nigh constantly to explain the use of a pi. Otherwise, there would be little point to the unit as a user could use any external hardware not at max load to run said application, keeping in mind the limited resources of a raspberry pi.

Two: The creator doesn't want whatever the pi is doing on their primary hardware. In this scenario, we instead look at the unit as something more akin to a bare metal sandbox. An environment in which software can run in a controlled setting, can be modified with ease and, importantly, not put large volumes of data, possibly even hardware, at risk. As the raspberry pi is fairly cheap, and easy to both wipe and backup, any operation one might see as risky on more sensitive devices makes far more sense to perform on the pi. As Kaustav Bag argues in his comparison between bare metal, virtual machines and containers, "Dedicated hardware and resources improve security by isolating the environment from other tenants. Being single tenant, It's not affected by side channel attacks where one tenant can steal secrets of its neighbours (Other tenants)." (Bag).

This reasoning is what lead us to deciding our final project. Electing to use two desktops and a single raspberry pi, we designed a system in which the pi can be used as a bare metal sandbox to both mediate transfers and prevent malicious files from infecting a system. The proposal is this: You are an inherently distrustful person and want to make sure your PC never comes into contact with any malware. When someone needs to give you a file, you direct them to a program which (unbeknownst to them) sends the file to the pi unit. When the pi receives a file, it will check it for malicious content via an API, sorting and logging the file based on the response and making additional judgments where need be. If the file is flagged as malware, it will delete the file in question and instead return a log showing why. If it's clean, the program will verify the file, and you can now download it with ease of mind. This level of paranoia may seem extreme, but as malware grows ever more complex their developers have begun to plan for scenarios where the virus is virtual boxed. In a 2017 study published in the Symposium on Security and Privacy, they were able to design simple malware which could "determine that a system is an artificial environment and not a real user device with an accuracy of 92.86%" (Miramirkhani). It may not

be the most glamorous use of the unit, but we believe it's one of the most realistic and practical projects we could approach.

Lets first discuss the program which analyses the file. This code is designed to facilitate three specific functions. Firstly, it must work in tandem with the server running on the pi, awaiting file inputs to work with. Second, using these files, it will make a request via API to an external server, in this case Virustotal, to have the file checked for malware. Third, based on the response received, it must sort the file into its allocated directory, logging where necessary. By and large, this should have been a relatively straightforward program, especially as we worked in python which has myriad libraries to facilitate such. This turned out to be the furthest thing from it.

The ultimate problem with this setup was the API itself. Despite what it may seem, what we even expected, these calls aren't a simple yes / no response. Every service has a different way in which they both expect data to be sent and how they send their own, and Virustotal is no different. This means our code had to format requests in a very specific manner, and likewise receive files under certain constraints, thanks part in partial to us using the free API they offer. To start, sending files is a pain due to the services own outdated documentation. For instance, the string they state is used for grabbing a files "threat severity" is threat_severity, easy right? Well as of this project, this function has been changed to a dictionary with their API v3 (Files). This correlates with wide reaching changes across other functions which renders their own object table effectively useless. To get around this issue, we dance a fine line.

While API calls do exist that pull all the info we need at once, they are either depreciated or locked behind paid API. Instead, we systematically make a series of requests to pull specific information we need to build a proper portfolio. In our code, we make three different API requests. To start, we begin with the initial file upload, sending the file and await confirmation

the server has received the file. We do this to one, upload the file, but more so get the file scan id. After waiting thirty seconds (which we will explain later), we make a second call using the id to get the actual report. This includes a very basic overview of what antiviruses flagged the file as malicious or suspicious. If this response doesn't give a conclusive response (either too many or no flags), we then grab the SHA-256 from the response to make one last API call. This takes the SHA-256 and generates the behavior summary, which is collectively the reasoning behind antivirus flags and how the file reacted to a sandbox environment. We then scan this response for threat severity, and then move the file based on if it breaches our acceptable tolerance (high or critical responses). Everything here could be, according to their documentation, done in one request, be it grabbing the response after an upload or just grabbing tolerance alone, but without any updates, these are the hoops we jump through.

But this is just the limitation of what we can request. Infinitely more frustrating has been dealing with variables beyond our control. In particular, what do you do if the server fails to respond? Virustotal is great, but their free API is immensely unreliable when it comes to giving responses. This is in combination with limitations to file size and request frequency. The API we use has a max limit of four requests a minute, and four hundred a day. If, for any reason, this threshold is broken the system will lock the account for an hour, often a day for new accounts (we burned at least seven accounts while making a system to avoid this). A request is counted as anything that pings the server via API, so built in functions to check a file request status and the like are effectively blacklisted for our purposes. To best optimize our code, we stall at predetermined sections to avoid spamming, and account for events such as if the file has never been analyzed, meaning it must go through the full cycle. Our biggest solution to this however was file logging. When the server finishes, it gives a predictable response which we can log into a file. Rather than starting over, we can instead check this file for key variables we expect and loop until we get

them or quit when it goes for too long. This approach single-handedly solved our biggest issue with the API, in which the malicious file check response gave a null response with zero antivirus checks, what we began to refer to as a 0-0-0 error. Instead of risking a ban sending checks to see if the connection was live, we recursively check our response log to see if the malicious, suspicious, and undetected fields totaled more than zero, confirming we got a full response. If not, we try a new attempt three times at fifteen second intervals, and stop if everything fails, assuming we've hit yet another server error.

While the main problems came from implementing the API functionality, this doesn't mean anything without a client-server system supporting it and allowing users to interact with the product. The core problem to solve with this setup is finding a way to connect both the upload and download users to the pi, allowing for files to pass through it (and thus be sent through the virus checker API).

To accomplish this, we decided to implement a simple TCP socket protocol in python using the "socket" library. The pi runs a server that can be connected to the internet and allows potential clients to connect to it. This server binds to the IP of the pi on the internet, as well as a specified port that would be dedicated to this service. Clients would need to know this IP and port to connect to the server over the socket connection. Clients only stay connected to the server for long enough to complete their specified operation, then are promptly disconnected. This is done as a security measure, as if someone is paranoid enough to set up such a system to scan incoming files, they probably don't want someone to be connected any longer than they need to be to upload the file.

However, being paranoid doesn't mean you don't care about user experience, so the server utilized Python's "threading" library to allocate a new thread to each client that connects.

This is done to facilitate multiple simultaneous connections from client to server. This may seem unnecessary, since clients are only connected for as long as it takes to complete their process, this allows multiple clients to upload at the same time, or for the admin to download a file while someone uploads a different file. Each thread is created as a daemon, meaning they will automatically close if the main server program closes. This allows for efficient thread management and cleanup in the case of a server shutdown. During the development process, we explored a few different options to achieve this goal before settling on threading, such as basic socket blocking and the select library. However, we chose to use threading in our final project because it allows for effective and simple connection of multiple clients to the server. One of the main downsides of threading is scalability, since each thread takes up memory, and the server could theoretically start enough concurrent threads to use up all its memory. However, we determined that our use case is a small, personal server with few connections, so this limitation isn't a concern.

We also implemented a one second timeout for the server. This is necessary to allow the server to be killed with a KeyboardInterrupt from the terminal at any time. If we don't do this, the server.accept() line will indefinitely block until a client request is received, at which point the server will finally process the interrupt and close.

Another roadblock we ran into was letting the receiver know when the file transfer has ended when they are downloading a file. Obviously, the receiver should stop the transfer if the most recent packet is empty, but what if the data ends partway through a packet and another one isn't sent? Initially, we decided on adding a "EOF" flag in bytecode to the end of the data transfer as the solution. When the receiver sees a data packet that ends in "EOF" bytecode, it knows that the data has finished transferring and it can stop expecting to receive more data.

However, after more use and research, we decided that a simpler and more robust solution to this would be to utilize TCP socket half-close. By simply shutting down only the write end of the socket connection, we can easily let the receiving end know that there is no more data to send, while still leaving the connection partially open so the sender can receive an acknowledgement of their transfer from the server. Usually, this is a tough feature to utilize, since it means you can only send 1 file per socket connection. However, for our specific use case where we are only planning to send 1 file at a time, this is a very easy to implement and elegant solution (GeeksforGeeks).

Lastly, we needed to make sure that only trusted users can download from the server. This is also in the spirit of giving untrusted users as little access to server functions as possible to increase security. There is no need for an untrusted user to do anything besides upload a file to the server, so doing any other actions (such as listing the files on the server or downloading them) requires a password set on server startup. To increase the security of this password, we run it through a SHA-256 hash before storing it and compare this to the hash of any client password attempts.

In total, the project does sufficiently achieve our original goal of acting as a “miner’s canary” for malicious files. In the case of such a file being uploaded, the only hardware directly at risk is the pi itself, protecting the admin’s main hardware. All the functionality is in place to allow clients to upload files, have them be checked with the Virustotal API, then be available for download by the admin if they are clean. However, there are certainly many ways in which this project could be developed further or made more secure in order to more fully realize our original intention. Perhaps the most glaring flaw in our implementation is the need for clients to know the correct IP and port for the pi running the server, as well as needing to use the client

program to upload their file. As it stands, this only allows for clients to connect to the pi and upload/download files if they are on the same local network. With more hardware setup from the admin user, they could implement port forwarding from their home router into the pi to allow for connection to the wider internet, but this is a further development than we were able to achieve. The Virustotal API also leaves a lot to be desired, and if we were to do this project again, we might opt for a different way of scanning the files for potential malware. However, the project was an overall success. The pi is an integral part of the process, acting as a bare metal sandbox where potentially dangerous files can be scanned and dealt with, and 2 desktops can simultaneously connect to the pi to upload or download files from it.

Works Cited:

Bag, Kaustav. “Bare Metal vs Virtual Machines vs Containers: Choosing the Right Infrastructure for Your Workloads.” *Medium*, 7 July 2024, medium.com/@kaustav-bag/bare-metal-vs-virtual-machines-vs-containers-choosing-the-right-infrastructure-for-your-workloads-ee63fcd06b6e.

“Files: Information about files” *VirusTotal*, docs.virustotal.com/reference/files.

Miramirkhani, Najmeh, et al. “Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts.” Ieee, 2017, pp. 1009–1024.

GeeksforGeeks. (2024, October 3). *TCP connection termination*.

https://www.geeksforgeeks.org/tcp-connection-termination/