Project 2: Antivirus for Linux

Team Name: BatEngine

Members: Shuangpeng Chen, Weifeng Lin, Darren Ling, Michael Mathew

Introduction

In this project, we developed a basic antivirus scanner. Using the signatures of malicious programs stored in a database, certain files can be scanned and declared as having a virus inside them. This project was for understanding how certain antivirus software applications might function.

Our program is capable of scanning any file or directory and checking it against our whitelist database and our signatures database. If we detect that a virus is present, we can quarantine the file preventing the user from accidentally accessing it. We can also easily update our whitelist and signatures database from the internet.

Members Work

Shuangpeng Chen

Implementing all the modules together, and working on On-Access scanning and managing kernel aspects of the code. Also, worked on the database integration with curl to update our database files. Managed all aspects of updating the our files.

Lin Weifeng

Implemented the scanning of the virus. Accepting a file and checking the file for any virus signatures by parsing along the binary of the specified file and checking if the signature exists. If there is a virus found, we move to the quarantine code. If there was no virus found, then we can simply execute or open the code as necessary. This includes the code to scan an entire directory and subdirectories for potentially malicious files.

Darren Ling

Handled whitelist functionality. Being able to parse through the whitelist and communicate with the rest of the program about if the specified file is on the whitelist or not. This handles all the conversion needed to test the files with their hash and also tests if the files exist or not. Also in charge of the documentation and slide presentation.

Michael Mathew

Implemented the quarantine aspect for removing permissions of infected files. Created function for executing chmod. Using chmod 000, all the permissions such as reading and editing a file would be removed. Infected files would also have ".infected" appended to their names with another function.

Setup Instructions for Testing

Machine setup

Our project is meant to be run on **Ubuntu 16.04.5 LTS 32 bit**(ISO in appendix 3). Our implementation of kernel hooking sometimes doesn't work on other versions of linux so do be careful when running this on a separate virtual machine. Since our program comes pre-compiled, there is no other setup that should be necessary to run our code. We are using the *curl.h* library for our database update implementation (documentation in appendix 1).

Building the Project

If for whatever reason you need to compile the project again, there will be a couple steps to follow. This step can and should be skipped, unless something is wrong (i.e. skip to the "Scanning" section of the document). Since we are using the *curl* library, we need to make sure *curl* is installed on the machine. To install *curl.h* on our linux machine simply use the following command.

sudo apt-get install libcurl4-openssl-dev

After installing *curl.h* we can then compile our program as "anti", with the following line. This should be done from the *./antv* directory

make all

Now we have successfully built the project and can start execution.

Scanning

For our project there are two primary methods of testing, On-Demand scanning and On-Access scanning. For On-Demand scanning, you can scan a file or directory from the command line with the following command.

./anti -scan ***[filename/foldername]***

This will scan the file or directory. If successful, the program will inform the user from command line that the file is safe. otherwise it will display a message to inform the user that the file is infected and has been quarantined. Following that, any infected files will be renamed and the permissions to execute/open the file will be removed.

For On-Access scanning, we must first load the kernel module before On-Access scanning can start. To load the kernel module execute the following command:

./anti -load

To unload the kernel module, simply use

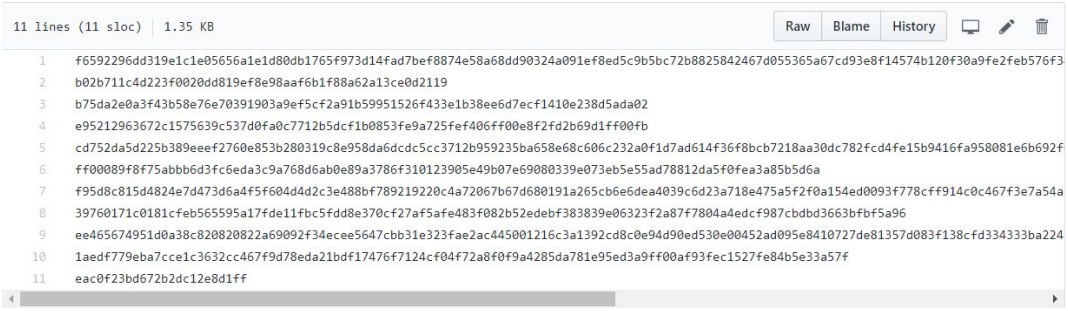
./anti -unload

After this is done, On-Access scanning will scan any files that are opened or executed (e.g. open, execl, execlp, execle, execv, execvp, execvpe) by the user. If there is a virus signature detected, then the program will again inform the user that there was a virus detected, and the file will be quarantined in the same way as On-Demand Scanning.

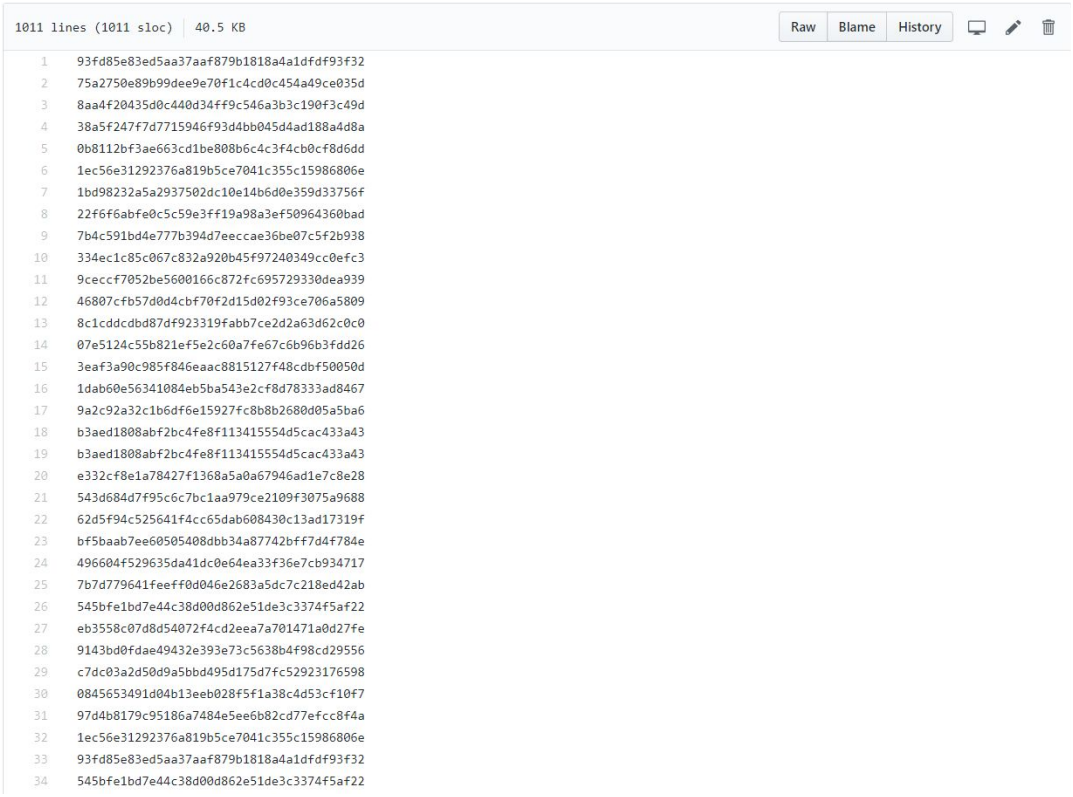
Database Files

Below are examples for how our signatures and whitelist files are formatted.

Infected Signature Database Sample: (signature.out)



Whitelist Database Sample: (whitelist.out)



Updating Databases

To update both the virus signatures database and the whitelist database, simply run the following command:

./anti -update

This will download fresh copies of *signature.out* and *whitelist.out* in the correct directories for the program to function. Then code can be executed normally. The server we are using is a personal server from one of our users and the IP can be changed accordingly if we need to access a different server. (Current IP: 35.231.146.204)

Code Implementation

In this part of the document we will be going over some commands to see exactly how our code works, and what it needs to do, starting with the *Update* command.

./anti -update

Following the execution of the update command, we will be using curl to get our files from our database. After we get the data with curl, we will write the data to local files, “signature.out” and “whitelist.out”. Now we can start using our base functions.

Next is the kernel module. This was a fairly challenging part to implement and we can start by loading the kernel module with the following command:

./anti -load

The way we load the kernel is based on finding the system call table and modify that. First we get the system call table address so that we can modify the table. Then we need to gain write access to the system call table. Once that is done we can hook our kernel into the system calls that we need so that it executes our code instead of the standard execution. Since we will be checking files in our code, we give our own user space code access to normal system call functionality when necessary.

After we have the kernel loaded, we can run or open any file of our choice. From this point on, if we have the kernel loaded, it will move the process to detecting the virus. By accessing out user-space code, it would be as if it was executing:

./anti -scan ***[filename/directory]***

However we got to this part, the code is the same. First we will check against the *whitelist.out* by simply running a SHA1 hash on the file and comparing it with our whitelist database. The current sample whitelist contains a series of files from */bin*, */usr/bin*, and */opt/local/bin*. These files have all been preprocessed with a SHA1 hash. If the file is on our whitelist we will break out of the code and proceed as normal. If we accessed it with On-Access scanning, we will try to open or execute the file as per the user's original instruction. If the file is not on the whitelist, then we will continue to scan for a virus.

The scanning of the virus is simply comparing all the signatures in our database to every part of the binary of the file. By shifting the bytes along the binary of the file, we can check the entire file for our potential virus. The signature can be of any length, and our code will handle it fine.

If the virus is not found, we can execute or open the program as normal, or with On-Demand scanning simply inform the user that the file is safe. If there is a virus found, we need to move onto the quarantine step. In the quarantine step, the file in question is stripped of its ability to be executed, written, or read by the use of chmod. Following that the file will need to be renamed, with an appended ".infected" to the end of the file name. This way, the malicious file is easily spotted by the user.

This concludes our overview on how our antivirus works having run through all the commands and a high level overview of our implementation of the code.

Appendix

1. Included the curl.h header which is used for interacting with the server to update our database. Documentation found here:https://curl.haxx.se/
2. The two following links were provided by the professor for reference on how to hook into the linux kernel.
   http://se7so.blogspot.com/2012/07/hijacking-linux-system-calls-rootkit.html
   https://github.com/ex0dus-0x/hijack
3. ISO for Ubuntu version 16.04.5 32bit LTS
   http://tw.archive.ubuntu.com/ubuntu-cd/16.04/
4. For Character Device reference
   http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/