Report on Secure Local Computing Supporting Shared Multi-User File System Raspberry Pi Project

CS 370

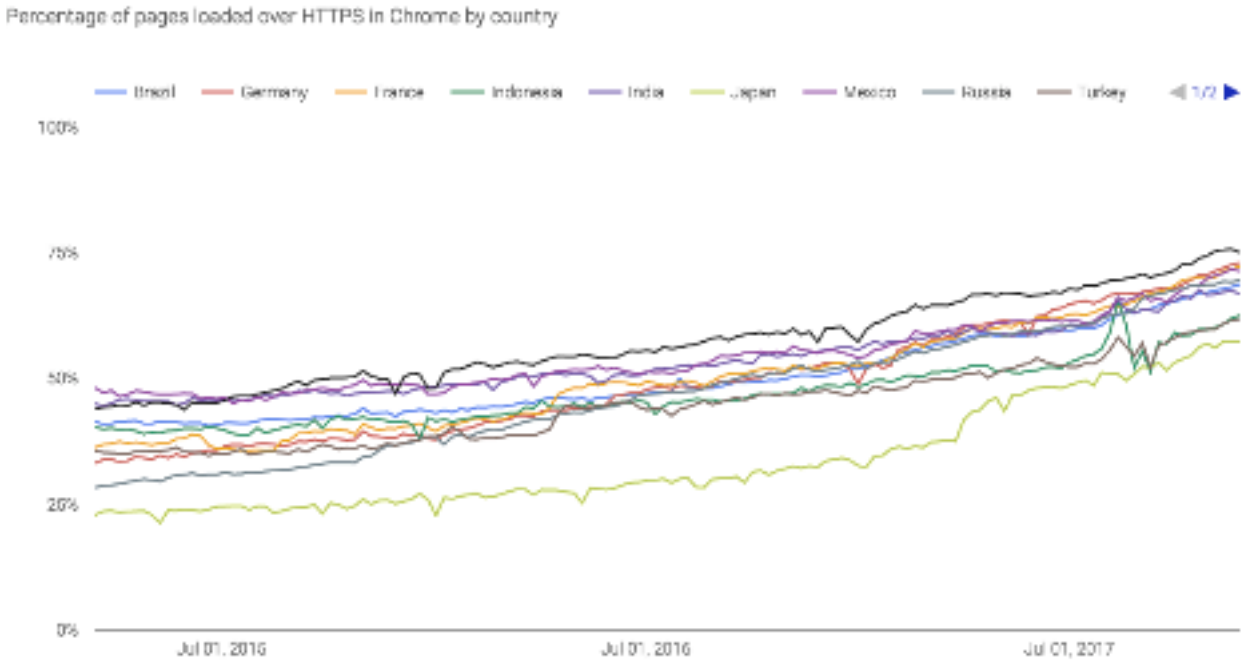
Term Project Option B

By:
Devin Dennis
Tanner Nickels
Jacob Tancher

Introduction

As computer systems continuously evolve, so does the need for more secure methods of file-sharing. With increasing demand for efficient, cheap, and secure computing systems, as well as an ever-growing accessibility to information, it is imperative for users to be able to store and share files without having to worry about the safety of their information.

Our implementation of the secure public computer which supports a multi-user file system was modeled after the Secure Socket Layer (SSL) which according to google is used to secure roughly 90% of their traffic. The SSL/TLS is used to encrypt web connections in HTTPS and ensures that data being sent between two systems is private. SSL encrypted web connections protect against various spoofing attacks and makes email and online shopping secure. SSL/TLS has also been adopted by voice over IP, file transfer, and instant messaging frameworks. Countries worldwide are moving toward the trend of increasing encryption across all web services.



https://transparencyreport.google.com/https/overview

The shared multi-user file system creates a secure environment for text files which ensures each user their information will be private and protected from the other users. This could be rewarding to developing countries whose school systems do not have enough money to buy multiple computers but can afford to buy their students rfid cards and a secure multi-user file system like our own. The students could access the internet and complete homework on the public machine and the students would not be able to view other students work so it could help teachers prevent cheating as well as a place for the teacher to grade everyone's work. The

teachers' administration cards would decrypt every file so he or she can go over each students work and provide feedback.

Problem Characterization

The problem that the group is trying to fix is the issue of security, and efficiency, in public computing. The idea that although certain security measures are present on public computers, that it may be insufficient in its current state. Majority of public computers only come with the basic password protection for an individual user and their files. This is relatively safe, however it provides a sort of rigidness on how you store and share files. The only current option is to either save on an external storage device and share it with whomever you are working on a project with, or send an email. Although email is a relatively safe and reasonable answer, it can get cluttered with an excess of miscellaneous items. As Computer Science Majors, we personally know of things such as a repository. However the general populous does not know such a thing exists. As for saving it on a external device, majority of people realize that the cloud is a safe and relatively quick way of transferring data between users. However cloud storage does cost money, so if cost is an issue that is not an option. Our project aims to make a local repository with an extra level of security that allows it to be efficient, safe, and cheap.

Proposed Solution and Implementation Strategy

RFID

To implement this task we added the raspberry pi RFID module. RFID or Radio-Frequency IDentification, acts like a barcode or magnetic strip on the back of a credit card. Each of these have a unique identifier attached to the pattern or the magnetic strip that, once scanned, can retrieve this data. Each RFID is made up of several elements: the transponder, the antenna, the reader, and the computer. The transponders contain the unique identifiers, most commonly a card or a fab (in our case the raspberry pi uses a card transponder). The antenna is the item that reads the transponder, which has its own benefit and downfalls. Something that sets RFID apart from a bar code or a magnetic strip is that it can be read at a distance and at any distance around the antenna. Some issues that come with a system like this include the possibility to corrupt the transponder or multiple transponders being within proximity of the antenna, creating interference. The reader component simply decodes the transponder, generates a unique code that identifies the individual transponder, and sends the information to the computer. In our case our transponder is a card with a 32-bit hexadecimal identifier associated with it. Because of this, one id card can have one of over 2 billion different unique identifiers, which allows for such variation that it would be difficult to pinpoint which identifier belonged to the system. This is the first level of security that is present in our system. Each of these identifiers can be translated

into a simpler number or variable in which programmers use to do code on, adding another layer of abstraction. In our case, with the two transponders we used to demonstrate our project, we turned the two 32-bit hexadecimals into more simplified UID values: 1000 and 1001.

To achieve this we had to research the raspberry pi's GPIO and the libraries that came with it. Furthermore, we had to research how to get the RFID to scan within our code. Luckily, there were some guidelines on the manufacturer's website that illustrated how to do so. The libraries that were needed in order to achieve this include the following: binascii, socket, signal, and Adafruit_PN532 (The RFID that was used).

Binascii is a library that allows the program to switch from binary to ascii value. This is how we are able to get from the hexadecimal identifier to the simple user ID. Socket and signal libraries allow for communication between the raspberry pi and the RFID module. Lastly the Adafruit_PN532 library just allowed us to configure the GPIO of the raspberry pi to function with the RFID. The program we implemented kept with these guidelines and varied them to be able to communicate with the other portions of our assignment. There are several benefits to reading and converting the identifiers, however, due to this simplification, it is possible to link the simple number to the original identifier within the transponder, leading to a potential security breach.

Permissions

When working with any practical system in which multiple users will have access to files, it's crucial for there to be some system of user permissions. On Linux machines, this is accomplished by a r/w/x file-access method. Every file has user, group, and other permissions that can be set to read, write and execute, identified by corresponding r, w, and x characters. Permissions can be modified in the terminal by the `chmod` command.

In a multi-user system where each user has their own login but can access shared files, this method works excellently. However, in a shared-file system where each individual user may not have their own login (i.e. all users essentially operate as a single user), it becomes rather pointless for there to be user/group/other permissions. Our implementation of permissions checking is much different, essentially relying on a simple boolean value (a user either does or does not have permission) and does not use group or other specifications. Alternatively, we have included several methods in our permissions checking (unused in this project), that could be used for permissions checking on a multi-user Linux system like the CS machines. This includes the use of the `os.access()` paired with `os.R_OK, os.W_OK, os.X_OK,` to determine user permissions. If necessary, there are additional functions that will add permissions to a dictionary, check if a file has specific permissions, or return a readable string defining the full permissions of a file.

The approach we took involves reading a 'permissions file' that includes information about which files belong to each user. When the RFID card is scanned, it runs the driver.py script (described in detail later) which reads the permissions file into a dictionary to be used in permissions checking. Each key in the dictionary corresponds to an RFID and the corresponding value is a list that user has permissions to. In our particular case this isn't the most secure way to check permissions. However, by improving the way permissions are added and removed from the file, and adjusting the permissions of the file itself in the system, it would work fine.

Directory Tree Traversal

In order to encrypt/decrypt files belonging to a user, the program must traverse the entire directory tree. Once the program reads the permissions into a dictionary and receives the RFID from the scanner, it implements the `os.walk()` system call to traverse the entire directory tree. The method will only traverse the tree of the directory given to it which provides a useful amount of functionality. Combined with the `os.path.dirname()` system call, the script will traverse from the directory in which it resides, so the program could be moved without needing to change code. Additionally, this provides a level of security and efficiency since only the files below it will be affected, so any system files that need to remain untouched for security reasons need only be placed in a directory outside of the tree.

From here, all files in the tree can be accessed and encrypted/decrypted. It's important to note that the traversal only gives filenames without the path and if they need to be edited, the program won't be able to find them since it operates within the parent directory. To resolve this, the `path` variable appends the filename to the full directory path so that files can be accessed directly. Another way to approach this would be to change directories as you go with `os.chdir()` but this seemed more complicated and dangerous, so we left it out.

Encryption/Decryption

The cryptographic system our secure environment was modeled after was the Secure Socket Layer (SSL) which is a network protocol used to secure web traffic. Web-browsers uses an asymmetric-key algorithm, RSA, and the host uses an symmetric-key algorithm, AES, to encrypt the data being sent across the network. RSA is used to encrypt the symmetric-key with the public key. The web-browser can then use the private RSA key to decrypt the data using the recovered symmetric-key. In the case of a shared multi-user system where the data is never transferred over a network, the admin acts as both the receiver and the sender.

Driver.py

The Driver.py script will be run automatically when an RFID card is scanned. It checks the 32-bit hexadecimal code associated with an RFID card and assigns it to a 4-bit UID to be used

during encryption/decryption. It proceeds to run Traverse.py with the '-d' flag and the UID (using `subprocess.call()`)to decrypt user files, then waits for the RFID to be scanned again. When the same RFID is re-scanned the script calls Traverse.py with the '-e' encryption flag and exits.

Traverse.py

Traverse.py reads the permissions file (in our case named 'userFiles') into a dictionary and grabs the list of files associated with the UID. We only use two cards so the possible UID numbers for this project are 1000 and 1001. It then uses the `traverse_encrypt()` or `traverse_decrypt` function as appropriate to traverse the directory tree as described above. If a file is in the list of accessible files, it will be encrypted/decrypted. Otherwise, it is skipped.

RW_Permissions.py

This file contains only functions and does not run on its own. It is included simply for the purpose of reading, writing, and editing the permissions file if necessary. The only function in it that we use directly is `permissions_to_dict()` to read the permissions file into a defaultdict and is imported into Traverse.py. It can also create a new permissions file from a list of filenames, add permissions to an existing file, and generate random permissions. An interesting function to mention is `generate_random_permissions()` which creates a given number of files in a new directory, fills them with random content, and assigns random permissions to them in a new permissions file. We mainly designed it for testing purposes.

SSL.py

This file contains all the necessary functions for file encryption and decryption. When a user is logging off the `traverse_encrypt()`method from Traverse.py feeds the `decrypt()` method each of the user's files one by one. Each file generates a unique random password with 36 ascii characters and digits which was accomplished using the random python package. This random password is used as the symmetric-key in the AES algorithm for file encryption and decryption. The file is encrypted using `aes_encrypt()`which appends the filename with ".aes" and removes the plain-text file from the file system. We then used RSA, a asymmetric-key algorithm, to encrypt each files symmetric-key. This means the symmetric-key is encoded in utf-8 and then encrypted using the rsa public key. For example, if the symmetric-key is "foo" we can encode it with utf-8 and encrypt it with a 512-bit key which will output the encrypted key as:

"b"\x97i-\xad\x05_\x90\xf5\x017\xae\xa0\xa0q\xe9
\xfb\xff'B\x10=J\x99\xbb\xc5*\xcb\xc7\xaa\x9bI\xc7\xef\xa6\x000\xa7\xf56\xfb\xdcY\x8d\xbf\x02[\
xf1\xd3\x10y\xf5%\xab\x06a\xdf^\xfbjzs\xc4`""

This provides a layer of security on top of AES. After the symmetric-key has been encrypted it is stored within the ".pwd" file which holds the encrypted passwords for each file in the system.

When a user is logging on the encrypted password is extracted from ".pwd" and then decrypted using `decrypt_symmetric_key()` which relies on the private rsa key. Now that the symmetric-key has been recovered it is used to decrypt the file using `aes_decrypt()` which removes the cipher-text file from the system and writes the plain-text to the file `.` SSL relies upon encoding strings with utf-8 for encryption and decryption since utf-8 is backwards compatible with ascii characters. The pyAesCrypt package was used to accomplish the encoding and decoding. The rsa package was used to generate large prime numbers for the public and private key pair as well as encrypt and decrypt the symmetric key.

Conclusion

Over the course of this project we came to learn some of the intricacies of file encryption/decryption as well as the practicality and functionality of an RFID scanner. We learned what does work and what doesn't work, as well as how practical our implementation is and how it could be improved for commercial use.

Obviously, within the scope of this project, our implementation of a shared multi-user file system is not pragmatic for commercial use. However, we believe that it is a step in a larger direction. With the ease-of-access and sophistication of systems in the United States it wouldn't make any sense. However, we feel that our system, or a similar one, could be effective in developing countries with less accessibility for cheap, secure file storage. In a localized network with multiple users but limited resources for the storage of files and encryption keys, multiple users could use an RFID card to access their personal files without being able to read or manipulate others.

This comes with a caveat. With all files being available on the same system, further measures than our own would have to be taken to prevent any malicious tampering. While files in our system are safe from theft, they are not necessarily safe from manipulation. A potential solution to this would be to set user permissions as files are being encrypted/decrypted so that all inaccessible files are "locked". Furthermore, the storage of encryption keys without the use of a cloud storage method means they must be stored on disk. All keys are encrypted themselves, but in theory, with enough technical knowledge, a user could reverse engineer the encryption process and the files wouldn't truly be secure.

Ultimately, it all boils down to one question: "How secure do you want your files to be?" Better security means more resources are required to make it happen, which isn't always possible. In a trusted environment, perhaps a workplace, this system will work as intended. In a public setting like an internet cafe, we'd suggest storing your files someplace else. However, we believe that this implementation of a shared-file implements effective strategies in cryptography, direct

system interaction, and programming of an external I/O device. It certainly isn't perfect, but it's a good start.

Bibliography

"What Is RFID?" *EPC-RFID INFO*, Bar Code Graphics, Inc., www.epc-rfid.info/rfid.
"Transparency Report." Google, Google, 2017, transparencyreport.google.com/https/overview.
"What is Secure Sockets Layer (SSL)? - Definition from WhatIs.Com." SearchSecurity, Margaret Rouse, Nov. 2016, searchsecurity.techtarget.com/definition/Secure-Sockets-Layer-SSL.