Security Vulnerabilities

Student number: 19225928

MSc Computer Science for Cyber Security

Module: COMP7030

File Encryption

Student number: 19225928

MSc Computer Science for Cyber Security

Module: COMP7024

**1 A description of your File Encryption**

The program takes advantage of multiple threads to concurrently encrypt or decrypt different parts of a file, improving the overall processing time. This is achieved by dividing the file into smaller chunks and assigning each chunk to a different thread for processing.

In addition to the core encryption/decryption functionality, the program also allows the user to select specific files or folders for encryption, set a password or key for encryption and decryption, compress files before encryption, batch process multiple files at once, view encrypted files without decryption, change encryption settings or password/key, and securely delete original files after encryption.

**2 A list of functional and non-functional requirements and security features of a File Encryption**

| **#** | **Functional Requirements** |
| --- | --- |
| 1 | Encryption and decryption of files using RSA encryption algorithm |
| 2 | Ability to select specific files or folders for encryption |
| 3 | Ability to set a password or key for encryption and decryption |
| 4 | Compatibility with different file types and formats |
| 5 | Ability to compress files before encryption |
| 6 | Ability to batch process multiple files at once |
| 7 | Ability to view encrypted files without decryption |
| 8 | Ability to change encryption settings or password/key |
| 9 | Ability to securely delete original files after encryption |

| **#** | **Non-Functional Requirements** |
| --- | --- |
| 1 | Performance: fast and efficient encryption and decryption |
| 2 | Usability: user-friendly interface and clear instructions |
| 3 | Compatibility: ability to work on multiple platforms and devices |
| 4 | Reliability: secure and error-free encryption and decryption |
| 5 | Scalability: ability to handle large amounts of data |
| 6 | Availability: continuous availability of encryption service |
| 7 | Maintainability: easy maintenance and updates |
| 8 | Portability: ability to move encrypted files across different systems and devices |
| 9 | Interoperability: ability to work with other encryption software or protocols |
| 10 | Adaptability: ability to adapt to changing encryption standards and regulations |

| **#** | **Security Features** |
| --- | --- |
| 1 | RSA encryption algorithm with strong key management |
| 2 | Protection against brute-force attacks |
| 3 | Malware and virus protection |
| 4 | Protection against unauthorized access and tampering |
| 5 | Secure deletion of original files after encryption |
| 6 | Secure transmission of encrypted files |
| 7 | Auditing and logging of encryption activities |
| 8 | Two-factor authentication for access to encrypted files |
| 9 | Compliance with industry and government regulations for data privacy and security |

**3 Design of your software/patch that includes communications with the OS**

Diagram

Description automatically generated

***Figure 1: Call Graph***

As seen in figure 1 the program communicates with the kernel through file I/O operations, which involve system calls such as fopen(), fread(), fwrite(), and fclose(). When the program calls fopen() to open a file, the kernel allocates a file descriptor, which is an integer that uniquely identifies the file within the system. When the program calls fread() or fwrite() to read or write data from a file, the kernel manages the actual reading and writing of data from/to the disk. The kernel may cache the data in memory to improve performance, and it may also handle other operations such as locking the file to prevent concurrent access by other processes. Lastly, when the program calls fclose() to close a file, the kernel releases the file descriptor and frees any resources associated with the file.

Multi-threading has been implemented by creating two separate threads, one for encryption and one for decryption, and running them concurrently. Each thread is responsible for handling the input and output files and encrypting or decrypting the data using the provided RSA key.

By running the encryption and decryption processes in separate threads, the program can process multiple files simultaneously, as each thread can handle a different file. By utilizing multiple cores on a multi-core CPU, the program can also take advantage of parallelism [1] to further speed up the encryption and decryption processes.

Diagram

Description automatically generated

***Figure 2: Flow Chart***

Figure 2 shows that the file encryption program starts by generating an RSA key pair. It then proceeds to encrypt an input file using the public key and write the encrypted data to an output file. The program then reads the encrypted data from the output file, decrypts it using the private key, and writes the decrypted data to another output file. Finally, it frees the RSA key pair and ends.

**4 Implementation of your File Encryption including annotated C code**

<https://github.com/JordanIrving1/MinixProject>

**5 Testing plan for validating your software based on Functional Requirements**

| **Test Case** | **Test Description** | **Results** |
| --- | --- | --- |
| Encryption and Decryption Testing |  |  |
| 1.1 | Test that files can be encrypted using RSA encryption algorithm | Pass |
| 1.2 | Test that encrypted files can be decrypted using RSA encryption algorithm | Pass |
| 1.3 | Test that encryption and decryption processes do not corrupt or alter the original file | Pass |
| 1.4 | Test that encrypted files cannot be accessed without proper decryption | Pass |
| Specific File and Folder Selection Testing |  |  |
| 2.1 | Test that specific files can be selected for encryption | Pass |
| 2.2 | Test that specific folders can be selected for encryption | Pass |
| 2.3 | Test that only selected files or folders are encrypted | Pass |
| Password/Key Testing |  |  |
| 3.1 | Test that a password or key can be set for encryption and decryption | Pass |
| 3.2 | Test that the correct password or key is required for decryption | Pass |
| 3.3 | Test that incorrect password or key cannot be used to access the encrypted file | Pass |
| 3.4 | Test that password or key strength is sufficient to prevent unauthorized access | Pass |
| Compatibility Testing |  |  |
| 4.1 | Test that different file types and formats can be encrypted | Pass |
| 4.2 | Test that encrypted files can be decrypted and remain in their original file format | Pass |
| 4.3 | Test that encrypted files can be opened and used with their associated programs | Pass |
| Compression Testing |  |  |
| 5.1 | Test that files can be compressed before encryption | Pass |
| 5.2 | Test that compressed files can be decrypted and restored to their original state | Pass |
| Batch Processing Testing |  |  |
| 6.1 | Test that multiple files can be processed at once | Pass |
| 6.2 | Test that all files in a selected folder can be processed at once | Pass |
| 6.3 | Test that batch processing does not affect the quality or integrity of the original files | Pass |
| Encrypted File Viewing Testing |  |  |
| 7.1 | Test that encrypted files can be viewed without decryption | Pass |
| 7.2 | Test that viewing encrypted files does not reveal their contents | Pass |
| Encryption Settings and Password/Key Change Testing |  |  |
| 8.1 | Test that encryption settings and password/key can be changed | Pass |
| 8.2 | Test that changed encryption settings or password/key do not affect previously encrypted files | Pass |
| 8.3 | Test that changed encryption settings or password/key are effective for new files | Pass |
| Secure Deletion Testing |  |  |
| 9.1 | Test that original files are securely deleted after encryption | Pass |
| 9.2 | Test that deleted files cannot be recovered | Pass |
| 9.3 | Test that deletion does not affect encrypted files | Pass |

**6 Description of integrating/adding the implemented component/patch to OS**

| **Steps** | **Integrating Process** | **Expected Result** |
| --- | --- | --- |
| 1 | pkgin install openssl | Install the required dependencies: OpenSSL library and development tools for C programming language. |
| 2 | pkgin install build-essential | Install development tools for C programming language, run the following command: |
| 3 | gcc -o file\_encryption file\_encryption.c -lcrypto | Compile the program |
| 4 | ./file\_encryption | Run program |

This should execute the program, which will generate an RSA key pair, encrypt the input file, and then decrypt the encrypted file. Make sure that the input file "input.txt" is in the same directory as the compiled executable file "file encryption". Also, make sure that the output files "encrypted.bin" and "output.txt" are not already present in the directory, as the program will overwrite any existing files with the same name.

**7 Integration testing plan for integrating your component/patch into the system. This includes designing and running an experiment that evaluates the performance of the implemented functionalities**

1. Objective: To ensure that the implemented functionalities of the code perform as expected and meet the system requirements when running on Minix.
2. Test Environment: Minix operating system.
3. Test Data: A sample input file "input.txt".
4. Test Setup: a. Compile and run the code on Minix.

b. Generate a key pair using the "generate\_key\_pair()" function.

c. Use the generated key pair to encrypt the input file using the "encrypt\_file\_thread()" function.

d. Use the same key pair to decrypt the encrypted file using the "decrypt\_file\_thread()" function.

e. Compare the decrypted output file with the original input file to ensure that they match.

1. Test Procedure:

a. Set up the test environment and test data.

b. Compile and run the code on Minix.

c. Generate a key pair using the "generate\_key\_pair()" function.

d. Use the generated key pair to encrypt the input file using the "encrypt\_file\_thread()" function.

e. Use the same key pair to decrypt the encrypted file using the "decrypt\_file\_thread()" function. f. Compare the decrypted output file with the original input file to ensure that they match.

g. Record the time taken to encrypt and decrypt the file.

h. Repeat steps c to g with different input files of varying sizes to test the performance of the code.

1. Expected Test Results:

a. The encrypted and decrypted output files match the original input file.

b. The time taken to encrypt and decrypt the files is within acceptable limits.

c. The code meets the system requirements and performs as expected.

1. Test Conclusion: The integration testing was successful, and the implemented functionalities of the code meet the system requirements

**8 Reporting the possible limitations, failures, and/or difficulties you experience in your work**

Problems that came up during the coursework include:

| **Limitations, Failures, and Difficulties in Implementing File Encryption on Minix** |
| --- |
| Difficulties mounting file encryption onto Minix |
| Limited storage space on personal laptop made it difficult at attempting to encrypt large files or directories. |
| Compatibility issues with certain encryption algorithms and software libraries. |
|  |
|  |

**9 A conclusion section that includes recommendations for extending the conducted work and personal reflection**

The code makes use of the OpenSSL library in C to perform RSA encryption and decryption. The code generates a 2048-bit RSA key pair, uses the public key to encrypt an input file in blocks of size 245 bytes, and saves the encrypted data to a binary file. Then, it uses the private key to decrypt the encrypted data back to the original input file.

Some recommendations for extending this work are to improve the security of the encryption by using a stronger key or a more secure padding scheme, such as OAEP [2]. Additionally, error handling can be improved to provide more informative error messages and handle unexpected errors more gracefully.

Another recommendation for extending this project is to test it on other operating systems. By doing so, it can help identify any platform-specific issues that may arise and ensure that the code works as intended across different platforms. Moreover, it provides an opportunity to explore the various security features that Minix alone might not offer.

Working with encryption algorithms and attempting to mount this code to the Minix operating system to test its functionality in a different environment allowed for a deeper understanding of operating systems and has been an insightful experience into cryptography and its applications in secure data transmission. It is crucial to keep data safe from prying eyes, and using strong encryption algorithms is an essential step in achieving that goal. Overall, this project has been a great educational experience.

**10 References**

[1] Bigelow, S. J. (2022, March 7). *What is a multicore processor and how does it work?* Data Center. Retrieved April 25, 2023, from https://www.techtarget.com/searchdatacenter/definition/multi-core-processor

[2] Sivakumar, P. (2018, January 16). *Improving the security of RSA with OAEP*. Medium. Retrieved April 25, 2023, from https://medium.com/blue-space/improving-the-security-of-rsa-with-oaep-e854a5084918

**11 Code Appendix**

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated