COMP.SEC.300

Exercise work – Turms

Sipi Ylä-Nojonen
15.5.2022

Sisällys

COMP.SEC.300 Exercise work – Turms

1. General Description

“Turms” application is Tornado web -based client + server software for transferring files between users using HTTP(S). From usage perspective Turms is optimally used in situations where one user wants to share certain files to another single user. System supports several connections at a time, but scalability concerns are not a priority in the context of this student exercise work. Main purpose of this work was to try to implement such application secure by design by following common design principles, mainly OWASP’s list of top 10 web application security risks.

From practical perspective it must be noted that in most cases normal users are behind some sort of firewall or router and thus port forwarding, or similar actions would be needed to use application outside local area networks. Since primary concern of this exercise work is in the process of creating a http connection utilizing application that follows secure programming principles this document doesn’t concern this kind of practical limitations further other than from security perspective. These concerns are explored more in the 3rd chapter of this document.

2. Structure of the Program

Environment

Exercise project is created with Python version 3.10.4 and is tested on Windows 10.

Used python packages

Project utilizes several widely used packages for different purposes within application. This chapter is mostly concerned with the libraries that interact with security features of the application.

asyncio

Asyncio provides python with possibility of asynchronous execution. In the context of this exercise this removes the need for threading in code written for exercise and thus reduces the risk of race conditions.

cryptography

Cryptography library offers several cryptographic recipes and primitives. It is developed alongside Python to create a standard library for cryptography in Python.

tornado

Tornado is a web application framework originally developed by FriendFeed and later acquired and made open source by Facebook. This exercise utilizes both web application wrapped http server framework and http client implementation provided by Tornado. Framework also supports asynchronous implementation at both ends.

ssl

Python ssl provides TLS/SSL wrapping for socket objects. In this exercise it is used mainly for loading ssl contexts for use in server configuration, but tornado http-server and client use it for the ssl handshake when creating https connections.

logging

Python’s logging library is utilized in this exercise to log events on both ends of http connection such as user requests to the server.

pathvalidate

Pathvalidate offers patterns for validating strings to match eg. file path or a filename. In this exercise it is used mainly to validate user supplied filenames to not contain directory paths and to validate directory paths server side to prevent errors.

Internal structure

Application fits loosely to MVC model so that application utilized controller class that delegates other actions based on user input and a View class that updates GUI. However, Controller is responsible also for delegating view calls and model part consists of more loosely defined components.

Server side of the application consists of Tornado web application -based application that handles encryptor creation for file encryption for responses that need it. It also regulates the actual HTTP(S) server that handles connections clients. For each client request an instance of RequestHandler class is created to handle the response for each request. RequestHandler utilizes encryptors that are created by its host web application as well as static ServerFileHandler for sanitation of request paths and creating file objects for response.

Client side is handled by ConnectionHandler that creates connection to server and delegates Downloader object for saving downloaded files and updates GUI through Controller callbacks.

Even though GUI layout is created as if user were making connection to server all actions are executed as separate HTTP requests as the protocol requires. File transfers are sent chunked from server side and decrypted and written on client side on the go from the streamed response body chunked as they arrive.

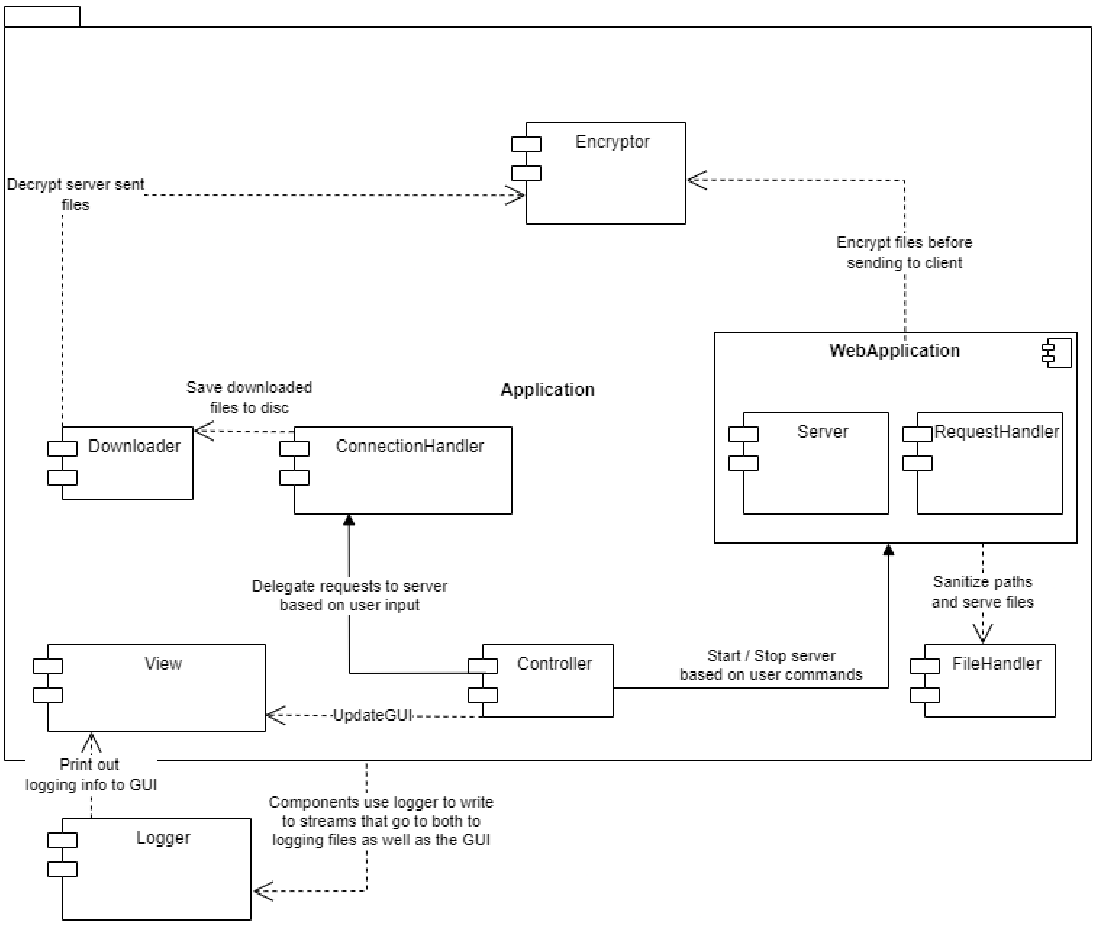
Encryptor
Decrypt server sent files
Encrypt files before sending to client
WebApplication
Application
Save downloaded files to disc
Downloader
ConnectionHandler
Server
RequestHandler
Delegate requests to server based on user input
Sanitize paths and serve files
View
Controller
Start / Stop server based on user commands
FileHandler
UpdateGUI
Print out logging info to GUI
Components use logger to write to streams that go to both to logging files as well as the GUI
Logger

Diagram 1 – Structure of system

3. Secure Programming Solutions

Both server and client-side communication of application utilized Tornado web. It is quite well documented framework that also supports several security functionalities by design, such as usage XSRF-tokens and TLS wrapping HTTP connections. Additionally, Turms server by default only accepts GET and HEAD requests to minimize risk of server modification. This is concerned with “injection” from OWASP top 10 list [3].

Preventing DNS attack as described in “Server-side Request Forgery” [4] is implemented through tornados application framework, which supports host matching thus allowing requests only to ip-address matching predefined host pattern.

Since the HTTPS implementation doesn’t authenticate users and anyone connecting to the server can download files, possibility to separately encrypt the files with key derived from server hosting party defined password which must be known by the downloading party to decrypt downloaded files. This encryption is initialized separately for each client request with new cryptographically safe values for salt that is used to derive the key and for initialization vector for the actual encryption cipher. Encryption is implemented with python cryptography details of which have been explained in chapter 2.

Key derivation is done with cryptography library’s **PBKDF2HMAC()** function, with 1200000 SHA256 iterations and 32-byte salt. The actual encryption is done with AES using derived key and 16-byte initialization vector. Both salt and iv are generated with **os.urandom()** which, according to python documentation is unpredictable enough for cryptographic use. This configuration attempts to follow the guidelines set for OWASP top 10 list 2nd most common problem, “cryptographic failures” [2]. A minimum for SHA256 iterations considered by cryptography is set on amount of 390000 used by Django, a popular Python web application framework used in production [5, 6]. However, since Turms is expected to serve much less connections, it uses three times the amount, which from testing seems to not cause any major performance drops for a single connection. Operation mode for AES was originally with CBC but was along later into the exercise changed to CBF **since CBC is known to be vulnerable to padding oracle attacks** [7] as pointed out by a fellow student.

One main concern in the server is possibility of path traversal when downloading files from the server, which falls under both the most common problem of “broken access control” listed in OWASP top 10 security risks [1] since it is concerned with accessing unauthorized areas with url tampering as well as the 3rd one: “injection” regarding user input sanitation. To prevent this all files on server are served from single directory which makes it possible to check the user input parameter file name to be a valid file name instead of file path. (Same could also be achieved with directories by checking that absolute path of user requested path is within the root downloadable content directory.)

All requests arriving to server are logged containing the method and path of request as well as the Ip-address of the client who created the request.

4. Testing

Testing in this exercise work is limited to manual testing of both use and misuse scenarios to ensure that intended functionality of the implemented application works and that misuse of system wouldn’t leak sensitive data or allow access to unintended parts of the system. Both static testing by examining the code and dynamic testing with different scenarios. Below in table 1 are collected evaluations and test results of certain use cases based on testing the security features explained in chapter 3. Possible vulnerabilities are emphasised with **bold lettering.**

Table 1 - Test cases

Feature / risk	Scenario	Result / current state	Fixed problems

Common error handling	User writes wrong or malformed ip-address, connection to the server fails, configuration file is missing or similar situation that should have application to handle errors gracefully	Errors are logged and error message is shown to user in case of user error. Server handles situations where request is erroneous or missing parameters.	In early stages malformed requests caused tornado server to crash with exceptions and in certain situations even **leak exception messages to the actual response.**
Not allowed methods	User creates request with unsupported methods (not HEAD or GET as hard coded default with Turms)	Server responds with code 405, Method not allowed	-
XSRF-tokens and cross-site request forgery prevention	Server creates xsrf-token and adds it response headers. User makes a POST/DELETE/PUT request and sets request headers to include this token. If token is not present or doesn't match the server token request is denied.	Server responds to POST/PUT/DELETE requests missing an xsrf-token or having incorrect xsrf-token with code 403, Forbidden (this required the respective methods to be allowed for testing unlike hardcoded to be refused by default)	-
Path traversal	User creates request to download file, but the file path included in URL includes relative path to directories other than directory containing server downloadable data.	Server refuses request with code 400, Bad request, if requested path to file in URl "host:port/download/<path to file>" is anything other than filename. (Also, paths directing to child directories of content directory will result in this error.)	-
File fetching	User attempts to request to download a file that is a valid file name but doesn't exist in content directory.	Server responds with code 404, Page not found.	Fixed errors in parsing path, which caused server to also deny downloads residing in content directory.
File encryption	Before sending user requested files are encrypted using AES symmetric encryption. Key for AES is derived with salted SHA256 based on user given password.	New salt for deriving key from password as well as the initialization vector for encrypting are generated each time file is encrypted using cryptographically safe algorithm. Salt or iv values are not saved beyond sending	Changed encryption mode from CBC to CFB as suggested by a fellow student, to mitigate **vulnerability padding oracle**

		them to client alongside the file for decryption. 1200000 iterations of SHA256 are used in the derivation process.	**attacks.** This also removed the need for padding in encryption.
File encryption configuration	Use of separate encryption for sent files can be configured by server host. User will be prompted for password upon server creation accordingly.	Password configuration is checked, and user prompted for password accordingly. Missing password will result in error on server start up when configuration requires one. **Strength of password is not checked beyond it not being empty one.**	Early errors in configuration caused **erroneous interpretation of Boolean values** as they were read as strings leading to *"False" == True*
File checksum	SHA256 hash of file sent is sent along it to ensure that file is undamaged after encryption transfer and decryption.	SHA256 hash results are unique enough to ensure there are no collisions between file results. **Checksum is sent unencrypted and calculated without salt so technically it would be possible for malicious user to use to gain knowledge of sent file. However, actually using eg. rainbow table attack for binary file sized several gigabytes would not be a reasonable threat. This might be achieved for small text files on the other hand similar to attacks on passphrases.**	-
TLS	Server creates key pair and certificate signed with this key to encrypt connection. User connecting must use HTTPS protocol for encrypted communication.	Certificate is properly created and connection to server is encrypted with TLS wrapping. **However, server certificate is self-signed, and its authenticity can't be checked for the possibility man-in-the-middle attack.** **Users are also not authenticated with certificate or other means meaning anyone can still connect to server and download any content.**	-

5. Future

5.1 Known vulnerabilities

This chapter considers known vulnerabilities that exist in Turms application. These considerations are based on testing in chapter 4.

Turms application employs TLS encryption in connections to server but since server certificate is self-signed, users must themselves decide to trust server. There is also no easy way to show user the public key to manually check that it matches server given key, since tornado http client class that is used for the connection handles certificate validation internally with no way to retrieve the certificate information. This makes connection vulnerable to man-in-the-middle attacks. However, since connections are primarily made directly to Ip-addresses, domain spoofing shouldn't be common problem.

On the other hand, user authentication is not checked, which means anyone connecting to server can access and download files that the server offers. As is the TLS implementation around the application connections is mostly for exercise and offers little to no benefit without the separate file encryption.

As for the separate encryption, key for encrypting is derived from server host defined password, and usage of encryption is not enforced by the application making it possible to serve files unencrypted or using a weak password.

5.2 Improvement

One improvement would be some implementation for using actually backed up certificate for the TLS to mitigate the risk of man-in-the-middle attack and ensure safety of connection.

One possibility could be also adding more flexibility into configuring the application such as using user defined server content location instead of hardcoded one without introducing additional vulnerabilities in the process.

6. Sources

[1] OWASP top 10, Broken Access Control, https://owasp.org/Top10/A01_2021-Broken_Access_Control/, checked 7th of May 2022

[2] OWASP top 10, Cryptographic Failures, https://owasp.org/Top10/A02_2021-Cryptographic_Failures/, checked 7th of May 2022

[3] OWASP top 10, Injection, https://owasp.org/Top10/A03_2021-Injection/, checked 7th of May 2022

[4] OWASP top 10, SSRF, https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/, checked 7th of May 2022

[5] Cryptography, using password with fernet, https://cryptography.io/en/latest/fernet/#using-passwords-with-fernet, checked 14th of May 2022

[6] Django password hasher, https://github.com/django/django/blob/main/django/contrib/auth/hashers.py

[7] Microsoft, Timing vulnerabilities with CBC-mode symmetric decryption using padding, https://docs.microsoft.com/en-us/dotnet/standard/security/vulnerabilities-cbc-mode