BAHIR DAR UNIVERSITY

Bahir Dar Institute of Technology (BiT)

Faculty of Computing

Department of Software Engineering

Fundamental of Software Security

Title: Secure Programming Techniques

<- ________________________________ ->

Documented by: -

1. **Abrham Desalegn..............1307050**
2. **Abiy Shiferaw....................1306078**
3. **Jemal Workie.....................1307712**
4. **Smachew Gedefaw............1308736**
5. **Solomon Muhye.................1309375**
6. **Yordanos Ayenew..............1311662**

Submitted to: Instructor Kelemework

Submit date: January 2022

Contents

Question #1 (conceptual)

In Chapter 9, the password manager stored passwords in a file. What would be some of the trade-offs involved in storing the passwords in a relational database instead of in a file? What types of additional input validation might need to be done on usernames and passwords if they are to be stored in a database?

1.1 Trade-offs of Storing Passwords in a Relational Database

Password managers are software applications that help users securely store and manage their passwords for various online accounts. One common way for password managers to store passwords is in a file, which can be encrypted to protect against unauthorized access. However, there are also trade-offs to consider when storing passwords in a file, and an alternative approach is to use a relational database to store the passwords. In this essay, we will explore the pros and cons of each approach and discuss the additional input validation that may be necessary when storing passwords in a database.

Storing passwords in a file has several benefits. One advantage is that it is relatively simple to implement, as the password manager just needs to create a file and write the encrypted password data to it. This can be done using standard file operations available in most programming languages. Additionally, storing passwords in a file allows for easy backup and recovery, as the file can be copied and stored in a secure location in case the original file is lost or corrupted.

However, there are also some potential drawbacks to storing passwords in a file. One concern is the risk of the file being accessed by unauthorized parties, either through physical access to the device on which the file is stored, or through digital means such as hacking. Encrypting the file can mitigate this risk, but it is still possible for the encryption to be broken or for the decryption key to be compromised. Another issue is that storing passwords in a file can be less scalable than using a database, as the file may become large and cumbersome to manage as the number of passwords grows.

An alternative approach to storing passwords is to use a relational database, which is a type of database that organizes data into tables and allows for relationships to be established between the data. One benefit of using a database to store passwords is that it can be more secure than storing them in a file, as the database can be configured with additional security measures such as user authentication and access controls. Additionally, a database is typically more scalable than a file, as it can handle a larger volume of data and can be easily queried and searched to retrieve specific password records.

However, there are also some trade-offs to consider when using a database to store passwords. One disadvantage is that it may require more resources to set up and maintain a database, as it typically requires specialized software and hardware. Additionally, a database may be more complex to implement than a file-based solution, as it requires a structured schema and may involve more advanced programming concepts such as SQL (Structured Query Language).

Here are some additional details on the trade-offs involved in storing passwords in a relational database versus a file:

- Security: One of the main trade-offs between using a database and a file to store passwords is security. A database can be more secure than a file, as it can be configured with additional security measures such as user authentication, access controls, and data encryption. However, setting up and maintaining these security measures can be more complex and resource-intensive than simply encrypting a file. Additionally, a database may be more vulnerable to certain types of attacks, such as SQL injection, that are not a concern with a file-based solution.
- Scalability: Another trade-off to consider is scalability, or the ability of the system to handle a growing volume of data. A database is generally more scalable than a file, as it can handle a larger volume of data and can be easily queried and searched to retrieve specific password records. However, a database may require more resources to set up and maintain, such as specialized hardware and software, which can be a cost consideration.
- Complexity: Using a database to store passwords can be more complex than using a file, as it requires a structured schema and may involve more advanced programming concepts such as SQL. This can be a disadvantage if the password manager is being developed by a small team with limited resources or expertise. On the other hand, storing passwords in a file is generally simpler to implement, as it can be done using standard file operations available in most programming languages.
- Backup and recovery: Storing passwords in a file allows for easy backup and recovery, as the file can be copied and stored in a secure location in case the original file is lost or corrupted. This can be a significant advantage in the event of a disaster or system failure. However, a database can also be backed

up and recovered, although it may require more specialized tools and processes to do so.

Overall, the trade-offs between using a database and a file to store passwords will depend on the specific needs and resources of the password manager. Factors such as security, scalability, complexity, and backup and recovery should all be considered when deciding which approach to take.

1.2 Additional Input Validation for Usernames and Passwords in a Database

If passwords are to be stored in a database, it is important to implement proper input validation to ensure the security and integrity of the data. This may include checking the length and complexity of passwords to ensure they meet certain criteria, as well as validating the format of usernames to ensure they are properly formatted and not easily guessable. Other considerations might include enforcing unique passwords and preventing the reuse of old passwords, as well as implementing rate limiting to prevent brute force attacks.

If usernames and passwords are to be stored in a database, it is important to implement proper input validation to ensure the security and integrity of the data. Here are some examples of additional input validation that might be necessary:

- Password length and complexity: To help prevent against brute force attacks, it is often recommended to implement password length and complexity requirements. This might include enforcing a minimum password length, requiring the use of a combination of upper and lower case letters, numbers, and special characters, and prohibiting the use of common words or easily guessable patterns.
- Password history: To prevent the reuse of old passwords and help protect against compromise, it may be necessary to maintain a history of previously used passwords and prevent users from reusing them. This can be implemented by storing a hash of each password in the database and checking the hash of the new password against the stored hashes to ensure it has not been used before.
- Unique usernames: To help prevent account confusion and protect against impersonation, it may be necessary to enforce unique usernames for each user. This can be done by checking the database to ensure that the requested username is not already in use.

- Rate limiting: To help prevent brute force attacks, it may be necessary to implement rate limiting on login attempts. This can be done by tracking the number of failed login attempts from a given IP address and temporarily blocking further attempts if the threshold is exceeded.
- Data sanitization: To help prevent against SQL injection attacks, it is important to properly sanitize user input before storing it in the database. This may involve using prepared statements or parameterized queries to ensure that user input is properly escaped and does not contain any malicious code.

Overall, the specific input validation requirements will depend on the specific needs and security requirements of the password manager. It is important to carefully consider the potential risks and vulnerabilities of storing passwords in a database and implement appropriate measures to mitigate them.

In conclusion, both storing passwords in a file and using a database have their own benefits and trade-offs. Storing passwords in a file is simple to implement and allows for easy backup and recovery, but it may be less secure and scalable than using a database. On the other hand, a database offers greater security and scalability, but it may be more complex to set up and maintain. Regardless of the approach taken, it is important to implement proper input validation to ensure the security and integrity of the password data.

Question #2 (conceptual)

Write an HTML filter that, given an arbitrary HTML document, produces an HTML document that will not result in the execution of script if loaded into a user's browser, but leaves "basic markup" (fonts, formatting, etc.) intact. Consider the possibility that the input document is not well-formed HTML, and also consider browser-specific features.

2.1 Question explanation

The goal this question is to create an HTML filter that takes in any valid or invalid HTML document as input and produces an output that is safe for loading into any browser. The output should preserve all "basic markup" (fonts, formatting, etc.) while preventing execution of potentially dangerous script or contextually preventing all script and html events.

2.2 HTML filter

What is HTML filter?

HTML filtering is the process of removing malicious HTML code from a web page or email in order to prevent malicious code from being executed by the browser. This helps protect websites and users from becoming victims of malicious attacks such as cross-site scripting and SQL injection. It can be written with some different languages like PHP, CSS, Python, VBScript, Perl, ASP classic, … etc., but JavaScript is usually the go-to programming language for creating HTML filters. Tools such as jQuery, AngularJS, React, and Vue can also be used to build HTML filters. Examples: -

* *Email Spam Filter*: This HTML filter examines emails for certain keywords or content which are deemed as suspicious and then removes them from the email before it is sent to its intended recipient.

* *Pornography Filter*: This HTML filter is used to strip out explicit images and videos found on websites, preventing users from accessing them.

* *Malware Filter*: This type of HTML filter uses heuristics analysis of websites' code in order to detect and block malicious software or threats.

Pros and Cons of using HTML filter

Pros: -

- ✓ HTML filters help improve the security of websites and online applications by preventing users from submitting certain data that could harm the system.
- ✓ They can also reduce the amount of spam messages and other inappropriate content submitted.
- ✓ HTML filters can help protect website owners from malicious attackers.
- ✓ They can be used to help ensure copyright compliance, making sure that only authorized content appears on a website or application.
- ✓ HTML filters allow for flexibility in terms of what types of content are allowed or disallowed on a website.

Cons: -

- × HTML filtering rules can sometimes be too restrictive, blocking out legitimate content which could have been beneficial to the site or application.
- × Unfamiliarity with the specific filter settings may lead to the mistaken inclusion or exclusion of important elements during content screening.
- × False positives (attribute blocks which should have been allowed through but were blocked) may occur during filtering, resulting in unnecessary frustration on part of both webmasters and users attempting to access such websites or applications.

How does HTML filter actually work?

There are two main strategies for achieving this process of filtering HTML document: ***blocking certain elements*** or ***validating tag structure***.

One strategy for creating this sort of filter is to simply block certain types of elements from the input document that could be potentially dangerous. Tags such as `<script>` (in our case only this one) and `<iframe>` should be removed from the output altogether if present in the input as they represent code that can be executed once loaded into the user's browser. Additionally, attributes on existing tags such as `src="script.js"` should also be removed from tags or replaced with safer versions. Similarly, techniques used by modern browsers like inline JavaScript event handlers should be blocked due to their potential danger when executed in a user's browser environment. Allowing these sorts of features could leave end-users vulnerable to cross-site scripting attacks and malicious code running on their computers. By completely blocking these elements and attributes, we can ensure safety when our filtered output is loaded by a user's browser.

An alternative approach would be validating tag structure rather than just blocking all potentially dangerous tags outright, ensuring that our filtered output remains properly formatted and functioning as expected despite being stripped free of dangerous elements/attributes while preserving basic formatting options like font size/style/color etc... A validation step may check for nested unpaired tags, illegal content nesting rules like JavaScript inside html tag (<html> … here … </html>), unbalanced quotes in parameter values or even other edge cases like duplicate attributes applied too frequently within elements etc., before allowing it through our filter system; such errors often lead to unexpected behaviors when rendered by various web browsers so proper validation here will prove beneficial especially when dealing with arbitrarily formed HTML documents which might not adhere strictly to common best practices at times.

Using either one or both of these methods, we can easily create a filter that takes in an arbitrary HTML document as input and produces a safely usable (and correctly structured) version meant exclusively for loading into any kind of browser while preserving "*basic markup*" options intact. By taking advantage of powerful tools such as W3C's Html5Lib library (which provides Python based parsing & serialization) we can implement these strategies in an efficient manner allowing us to efficiently create powerful filters with minimal effort required from the developer's side All this adds up towards providing users safe & consistent browsing experiences no matter where they access content from online regardless if it's through native web applications or mobile browsers expecting topnotch user privacy & security guarantees in all scenarios going forward without having them worry about potentially malicious pieces.

2.3 The required HTML filter

Some explanation about the code

The following HTML filter will strip all script from a given HTML document and leave the basic markup intact[1]. The first step is to look at the HTML document's DOCTYPE. If it contains "XHTML", we can safely assume that the document is well-

[1] To illustrate this one, go to the html file that we put it inside the rar file, and then by making the filter code comment you can see the difference of having filter in the file and not.

formed, and we can be sure that our filter will not break the page. If it does not contain "XHTML", then we must examine the tag names in the document and remove any that are used for scripting or styling (such as <script> or <style>). Finally, if there are any tags left over — for instance, if there were some tags in the DOCTYPE that were not in use—we should remove them as well. This is a filter that is meant to be used in a web application that allows users to upload HTML documents. The goal of this filter is to make sure that the document will not execute any script, but leave all of the basic markup intact. This filter will take an HTML document and turn it into a safe version of itself, which means: - All scripts have been removed - All <script> tags have been removed - All <style> tags have been removed - All <iframe> tags have been removed. The following HTML filter will remove all script tags and everything between them, while leaving basic markup intact.

HTML filter code

```
      .
      .
      .
<script>
      (function () {
        let createdElement = document.createElement("script");
        createdElement.type = "text/javascript";
        createdElement.async = true;
        createdElement.src = "https://example.com/some-js-file.js";
        let scriptValue = document.getElementsByTagName("script")[0];
        scriptValue.parentNode.insertBefore(createdElement, scriptValue);
      })();
    </script>
    <!-- If there are script tags in the attributes of other tags, strip out
only the tags themselves and leave the rest of the attribute's value intact -
->
    <!-- Make sure to leave basic markup (fonts, formatting, etc.) intact
since they are not the enemy/target at least for now -->
```

```
    <div>
      <p style="”color: " aqua; font-wieght: 600px; text-align: center;>
        This is some basic texthose style should be remain unchanged or
        unaffected by the HTML filter.
      </p>
    </div>
    <!--This is a basic HTML filter that will remove all script tags from an
HTML document. It will also remove any event attributes (onclick,
onmouseover, onmousemove, onmousedown, onload etc.) from tags to prevent any
script from being executed.-->
    <script>
      // remove all script tags
      $("script").remove();
      // remove all event attributes
      $("*").removeAttr("onclick onmouseover onload");
    </script>
      .
      .
      .
```

Question #3 (programming)

Implement HTTP digest authorization. Use a password file with salts, and reuse the BasicAuthWebServer from Chapter 9. Implement a program that allows you to add and delete passwords to and from the password file.

3.1 HTTP Digest Authorizations

HTTP Digest Authorization allows users that have a username and password on the server to access protected resources without sending their password over the network in plaintext. When using HTTP Digest Authorization, the server sends a challenge containing a random string (nonce) to the client, and the client must reply with an encrypted message that contains both the username and the valid nonce.

The encrypted message includes hashes (MD5 or SHA-1 algorithms) of several values such as the username, the computer's IP address, a secret token and other related information, which provide security when trying to validate that it is actually an authorized user. The system verifies this response and allows access when correct.

3.2 Using a password file with salts

A file using salts with passwords is a type of file that stores user passwords as well as randomly generated strings called salts. The purpose of the salt is to make it difficult for someone looking to obtain user passwords by gaining access to the password file from an external source, such as a hacker. The salt basically creates an extra layer of protection since it cannot be easily guessed, making it more difficult for malicious actors to gain access to a account protected by the file. Salted passwords are typically much more secure than simple unhashed or hashed passwords, as they greatly reduce the effectiveness of brute-force attempts at cracking them.

3.3 Using MiniPassword Manager

A mini password manager is a type of application used to store digital passwords securely. The software usually stores user-generated passwords locally, rather than on a cloud server, to help protect against online threats and potential data breaches. Some mini password managers also offer additional features such as two-factor authentication and other security protocols.

3.4 Java code for mini password manager [2]

```java
import java.util.*;
import java.io.*;
import java.security.*;
import java.util.Base64;
public class MiniPasswordManager {
    /** dUserMap is a Hashtable keyed by username, and has
     HashedPasswordTuples as its values */
    private static Hashtable dUserMap;
    /** location of the password file on disk */
    private static String dPwdFile;
    /** chooses a salt for the user, computes the salted hash
     of the user's password, and adds a new entry into the
     userMap hashtable for the user. */
    public static void add(String username, String password) throws Exception {
        int salt = chooseNewSalt();
        HashedPasswordTuple ur =
                new HashedPasswordTuple(getSaltedHash(password, salt), salt);
        dUserMap.put(username,ur);
    }
    /** computes a new, random 12-bit salt */
    public static int chooseNewSalt() throws NoSuchAlgorithmException {
        return getSecureRandom((int)Math.pow(2,12));
    }
    /** returns a cryptographically random number in the range [0,max) */
    private static int getSecureRandom(int max) throws NoSuchAlgorithmException {
        SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
        return Math.abs(sr.nextInt());
    }
    /** returns a salted, SHA hash of the password */
    public static String getSaltedHash(String pwd, int salt) throws Exception {
        return computeSHA(pwd + "|" + salt);
    }
    /** returns the SHA-256 hash of the provided preimage as a String */
    private static String computeSHA(String preimage) throws Exception {
        MessageDigest md = null;
        md = MessageDigest.getInstance("SHA-256");
        md.update(preimage.getBytes("UTF-8"));
        byte raw[] = md.digest();
        return new String((Base64.getEncoder().encode(raw)));
```

[2] To see the effect of this code, just run the compiled jave file that we have put in rar file.

```
     }
     /** returns true if the username and password combo is in the database */
     public static boolean checkPassword(String username, String password) {
          try {
               HashedPasswordTuple t = (HashedPasswordTuple)dUserMap.get(username);
               return (t == null) ? false :
                       t.getHashedPassword().equals(getSaltedHash(password,
                               t.getSalt()));
          } catch (Exception e) {

          }
          return false;
     }
     /** Password file management operations follow **/
     public static void init(String pwdFile) throws Exception {
          dUserMap = HashedSaltedPasswordFile.load(pwdFile);
          dPwdFile = pwdFile;
     }
     /** forces a write of the password file to disk */
     public static void flush() throws Exception {
          HashedSaltedPasswordFile.store (dPwdFile, dUserMap);
     }
     /** adds a new username/password combination to the database, or
      replaces an existing one. */
     public static void main(String argv[]) {
          String pwdFile = null;
          String userName = null;
          try {
               pwdFile = argv[0];
               userName = argv[1];
               init(pwdFile);
               System.out.print("Enter new password for " + userName + ": ");
               BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
               String password = br.readLine();
               add(userName, password);
               flush();
          } catch (Exception e) {
               if ((pwdFile != null) && (userName != null)) {
                    System.err.println("Error: Could not read or write " + pwdFile);
               } else {
                    System.err.println("Usage: java " +
                            "com.learnsecurity.MiniPasswordManager" +
                            " <pwdfile> <username>");
               }
          }
     }
}
```

```
/** This class is a simple container that stores a salt, and a
 salted, hashed password */
class HashedPasswordTuple {
     private String dHpwd;
     private int dSalt;
     public HashedPasswordTuple(String p, int s) {
          dHpwd = p; dSalt = s;
     }
     /** Constructs a HashedPasswordTuple pair from a line in
      the password file. */
     public HashedPasswordTuple(String line) throws Exception {
          StringTokenizer st =
                  new StringTokenizer(line, HashedSaltedPasswordFile.DELIMITER_STR);
          dHpwd = st.nextToken(); // hashed + salted password
          dSalt = Integer.parseInt(st.nextToken()); // salt
     }
     public String getHashedPassword() {
          return dHpwd;
     }
     public int getSalt() {
          return dSalt;
     }
     /** returns a HashedPasswordTuple in string format so that it
      can be written to the password file. */
     public String toString () {
          return (dHpwd + HashedSaltedPasswordFile.DELIMITER_STR + (""+dSalt));
     }
}
/** This class extends a HashedPasswordFile to support salted, hashed passwords. */
class HashedSaltedPasswordFile extends HashedPasswordFile {
     /* The load method overrides its parent's, as a salt also needs to be
     read from each line in the password file. */
     public static Hashtable load(String pwdFile) {
          Hashtable userMap = new Hashtable();
          try {
               FileReader fr = new FileReader(pwdFile);
               BufferedReader br = new BufferedReader(fr);
               String line;
               while ((line = br.readLine()) != null) {
                    int delim = line.indexOf(DELIMITER_STR);
                    String username = line.substring(0,delim);
                    HashedPasswordTuple ur =
                            new HashedPasswordTuple(line.substring(delim+1));
                    userMap.put(username, ur);
               }
          } catch (Exception e) {

               System.err.println ("Warning: Could not load password file.");
```

```java
          }
          return userMap;
     }
}
/** This class supports a password file that stores hashed (but not salted)
 passwords. */
class HashedPasswordFile {
     /* the delimiter used to separate fields in the password file */
     public static final char DELIMITER = ':';
     public static final String DELIMITER_STR = "" + DELIMITER;
     /* We assume that DELIMITER does not appear in username and other fields. */
     public static Hashtable load(String pwdFile) {
          Hashtable userMap = new Hashtable();
          try {
               FileReader fr = new FileReader(pwdFile);
               BufferedReader br = new BufferedReader(fr);
               String line;
               while ((line = br.readLine()) != null) {
                    int delim = line.indexOf(DELIMITER_STR);
                    String username = line.substring(0,delim);
                    String hpwd = line.substring(delim+1);
                    userMap.put(username, hpwd);
               }
          } catch (Exception e) {
               System.err.println ("Warning: Could not load password file.");
          }
          return userMap;
     }
     public static void store(String pwdFile, Hashtable userMap) throws Exception {
          try {
               FileWriter fw = new FileWriter(pwdFile);
               Enumeration e = userMap.keys();
               while (e.hasMoreElements()) {
                    String uname = (String)e.nextElement();
                    fw.write(uname + DELIMITER_STR +
                            userMap.get(uname).toString() + "");
               }
               fw.close();
          } catch (Exception e) {
               e.printStackTrace();
          }
     }
}
```

3.5 java code for BasicAuthWebServer

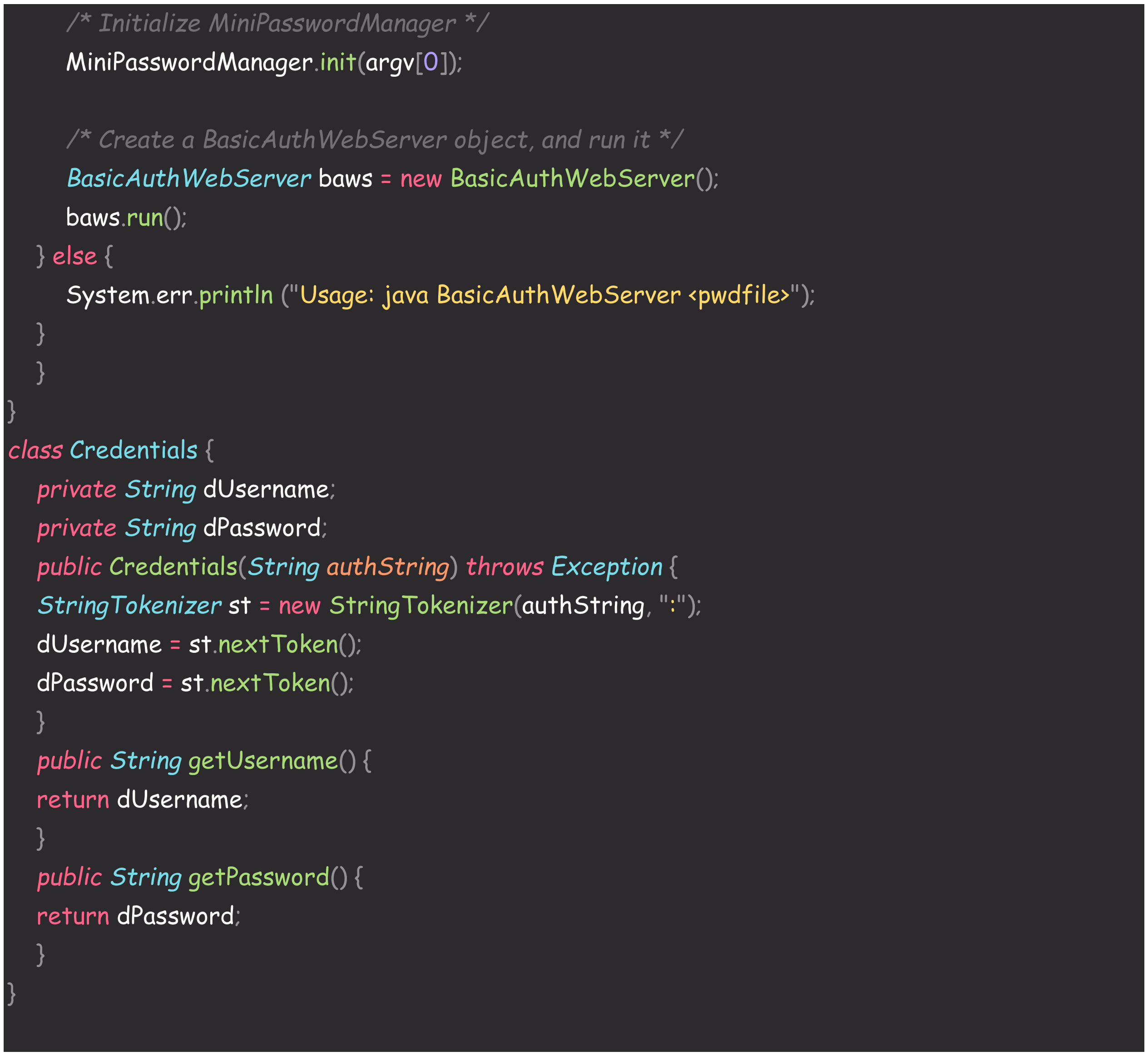
```
// SimpleWebServer.java
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.Base64;
import java.util.Base64.Decoder;
import java.util.Base64.Encoder;
/*This will be used to pause between failed requests*/
import java.util.concurrent.TimeUnit;

public class BasicAuthWebServer {
   /* Run the HTTP server on this TCP port. */
   private static final int PORT = 8080;
   /*This is a static number of max allowed failed requests i set it to 5 for testing purposes*/
   private static final int MAXREQUESTS = 6;
   /*The two arraylists below are to hold the ip number and number of failed requests in parllel*/
   private ArrayList<String> reqIP = new ArrayList<>();
   private ArrayList<Integer> numReq = new ArrayList<>();
   /* The socket used to process incoming connections from web clients */
   private static ServerSocket dServerSocket;

   public BasicAuthWebServer () throws Exception
   {
   dServerSocket = new ServerSocket (PORT);
   }

   public void run() throws Exception
   {
      while (true)
      {
         /* wait for a connection from a client */
         Socket s = dServerSocket.accept();
         /* then process the client's request */
         processRequest(s);
      }
   }

   private String checkPath (String pathname) throws Exception
```

```
{
    File target = new File (pathname);
    File cwd = new File (System.getProperty("user.dir"));
    String s1 = target.getCanonicalPath();
    String s2 = cwd.getCanonicalPath();

    if (!s1.startsWith(s2))
        throw new Exception();
    else
        return s1;
}

/* Reads the HTTP request from the client, and
   responds with the file the user requested or
   a HTTP error code. */
public void processRequest(Socket s) throws Exception
{
/* used to read data from the client */
BufferedReader br = new BufferedReader (new InputStreamReader (s.getInputStream()));

/* used to write data to the client */
OutputStreamWriter osw = new OutputStreamWriter (s.getOutputStream());

/* read the HTTP request from the client */
String request = br.readLine();

String command = null;
String pathname = null;

try
{
    /* parse the HTTP request */
    StringTokenizer st =
    new StringTokenizer (request, " ");
    command = st.nextToken();
    pathname = st.nextToken();
} catch (Exception e)
{
    osw.write ("HTTP/1.0 400 Bad Request\n\n");
    osw.close();
```

```
        return;
    }
    /*This will register the client's ip address*/
    String ipAddress = s.getRemoteSocketAddress().toString().substring(0,
s.getRemoteSocketAddress().toString().length()-6);

    /*initializing a variable of numRequests*/
    int numRequests = 0;
    /*if the given ipaddress has already had failed requests the value of numRequests will be upated
accordingly*/
    if(reqIP.contains(ipAddress))
    {
        numRequests = numReq.get(reqIP.indexOf(ipAddress));
    }
    logEntry("FileRequestslog.txt", command + " " + pathname + " " + ipAddress + "\n");
    if (command.equals("GET"))
    {
        Credentials c = getAuthorization(br);
        /*added the parameter to ensure the requests will not be met if client has exceeded number
of allowable failed requests*/
        if ((MAXREQUESTS >= numRequests) && (c != null) &&
(MiniPasswordManager.checkPassword(c.getUsername(), c.getPassword())))
        {
            String lastLogin = "First Time loging in";
            try
            {
                /*sets lastLogin to last entry on login log for each client if there is no log the default is:
"First time loging in"*/
                BufferedReader bufr = new BufferedReader(new FileReader(c.getUsername() +
"login.txt"));
                String temp = "";
                while(temp != null)
                {
                    lastLogin = temp;
                    temp = bufr.readLine();
                }
            }
            catch(Exception e)
            {
            }
            /*This lets the client know the last time they logged in*/
            osw.write("last login: " + lastLogin + "\n");
            serveFile(osw, pathname);

            if(reqIP.contains(ipAddress))
            {
                int idx = reqIP.indexOf(ipAddress);
                reqIP.remove(idx);
                numReq.remove(idx);
            }
            logEntry(c.getUsername() + "login.txt", c.getUsername() + "\n");
        }
        else if(MAXREQUESTS < numRequests)
        {
            /*If the client exceeds the number of allowable failed requests they will no longer be
prompted to login*/
            osw.write("You have exceeded the number of failed logins. Please contact the
Administrator.");
        }
        else
        {
            logEntry("failedLogins.txt", ipAddress + " " + "\n"); //loging ip address of failed login client
            osw.write ("HTTP/1.0 401 Unauthorized\n");
            int x = 0;

            if(reqIP.contains(ipAddress))
            {
                x =(int)Math.pow(2,numReq.get(reqIP.indexOf(ipAddress)));
            }
            System.out.println("waiting..." + x + " seconds \nYou have had " + numRequests + " failed login
attempts \nYour ip will be locked after " + MAXREQUESTS + " failed login attempts");
            TimeUnit.SECONDS.sleep(x);

            if(reqIP.contains(ipAddress))
            {
                int idx = reqIP.indexOf(ipAddress);
                numReq.set(idx, numReq.get(idx) + 1);
            }
            else
            {
```

```
            reqIP.add(ipAddress);
            numReq.add(1);
          }
          osw.write ("WWW-Authenticate: Basic realm=\"BasicAuthWebServer\"\n\n");
        }
      }
      else if(command.equals("PUT"))
      {
        //if request is PUT use storeFile method
        Credentials c = getAuthorization(br);
        if ((MAXREQUESTS >= numRequests) && (c != null) &&
(MiniPasswordManager.checkPassword(c.getUsername(), c.getPassword())))
        {
          storeFile(br, osw, pathname);
          if(reqIP.contains(ipAddress))
          {
            int idx = reqIP.indexOf(ipAddress);
            reqIP.remove(idx);
            numReq.remove(idx);
          }
        }
        else if(MAXREQUESTS < numRequests)
        {
          osw.write("You have exceeded the number of failed login attempts. \nPlease contact the
Administrator.");
        }
        else
        {
          logEntry("failedLogins.txt", ipAddress + " " + "\n"); //loging ip address of failed login client
          osw.write ("HTTP/1.0 401 Unauthorized\n");
          int x = 0;
          if(reqIP.contains(ipAddress))
          {
            x = (int)Math.pow(2,numReq.get(reqIP.indexOf(ipAddress)));
          }
          System.out.println("waiting..." + x + " seconds \nYou have had " + numRequests + " failed login
attempts \nYour ip will be locked after " + MAXREQUESTS + " failed login attempts");
          TimeUnit.SECONDS.sleep(x);
          if(reqIP.contains(ipAddress))
          {
            int idx = reqIP.indexOf(ipAddress);
            numReq.set(idx, numReq.get(idx) + 1);
        }
        else
        {
            reqIP.add(ipAddress);
            numReq.add(1);
        }
        osw.write ("WWW-Authenticate: Basic realm=\"BasicAuthWebServer\"\n\n");
    }
}
else
{
    /* if the request is a NOT a GET,
    return an error saying this server
    does not implement the requested command */
    osw.write ("HTTP/1.0 501 Not Implemented\n\n");
}
/* close the connection to the client */
osw.close();
}

private Credentials getAuthorization (BufferedReader br) {
try {
    String header = null;
    while (!(header = br.readLine()).equals("")) {
    System.err.println (header);
    if (header.startsWith("Authorization:")) {
        StringTokenizer st = new StringTokenizer(header, " ");
        st.nextToken(); // skip "Authorization"
        st.nextToken(); // skip "Basic"
        return new Credentials(st.nextToken());
    }
    }
} catch (Exception e) {
}
return null;
}

public void serveFile (OutputStreamWriter osw,
                         String pathname) throws Exception {
    FileReader fr=null;
    int c=-1;
    StringBuffer sb = new StringBuffer();

    /* remove the initial slash at the beginning
       of the pathname in the request */
    if (pathname.charAt(0)=='/')
        pathname=pathname.substring(1);

    /* if there was no filename specified by the
       client, serve the "index.html" file */
    if (pathname.equals(""))
        pathname="index.html";

    /* try to open file specified by pathname */
    try {
        fr = new FileReader (checkPath(pathname));
        c = fr.read();
    }
    catch (Exception e) {
        /* if the file is not found,return the
           appropriate HTTP response code  */
        osw.write ("HTTP/1.0 404 Not Found\n\n");
        return;
    }

    /* if the requested file can be successfully opened
       and read, then return an OK response code and
       send the contents of the file */
    osw.write ("HTTP/1.0 200 OK\n\n");
    while (c != -1) {
            sb.append((char)c);
        c = fr.read();
    }
    osw.write (sb.toString());
    }
    public void storeFile(BufferedReader br, OutputStreamWriter osw, String pathname) throws
Exception
    {
```

```
    FileWriter fw = null;
    try
    {
        fw = new FileWriter(checkPath(pathname));
        String s = br.readLine();
        while(s != null)
        {
            fw.write(s + "\n");
            s = br.readLine();
        }
        fw.close();
        osw.write("HTTP/1.0 201 Created\n");
    }
    catch(Exception e)
    {
        osw.write("HTTP/1.0 500 Internal Server Error");
    }
}
public void logEntry(String filename, String record)
{
    try
    {
        FileWriter fw = new FileWriter(filename, true);
        fw.write(getTimestamp() + " " + record);
        fw.close();
    }
    catch(Exception e)
    {
        return;
    }
}
public String getTimestamp()
{
    return(new Date()).toString();
}

/* This method is called when the program is run from
   the command line. */
public static void main (String argv[]) throws Exception {
if (argv.length == 1) {
```



```
        /* Initialize MiniPasswordManager */
        MiniPasswordManager.init(argv[0]);

        /* Create a BasicAuthWebServer object, and run it */
        BasicAuthWebServer baws = new BasicAuthWebServer();
        baws.run();
    } else {
        System.err.println ("Usage: java BasicAuthWebServer <pwdfile>");
    }
    }
}
class Credentials {
    private String dUsername;
    private String dPassword;
    public Credentials(String authString) throws Exception {
    StringTokenizer st = new StringTokenizer(authString, ":");
    dUsername = st.nextToken();
    dPassword = st.nextToken();
    }
    public String getUsername() {
    return dUsername;
    }
    public String getPassword() {
    return dPassword;
    }
}
```

References

- https://okapiframework.org/wiki/index.php/HTML_Filter
- https://www.geeksforgeeks.org/store-password-database/
- https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/
- https://www.youtube.com/watch?v=r2JxqyeVTh4
- https://www.linkedin.com/pulse/how-store-passwords-database-arvind-kumar?trk=pulse-article
- http://www.learnsecurity.com/ntk