BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MULTI-FACTOR AUTHENTICATION IN WEB APPLICATIONS USING PAM

VIAC-FAKTOROVÁ AUTENTIZÁCIA VO WEBOVÝCH APLIKÁCIACH POMOCOU PAM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — MARIÁN KAPIŠINSKÝ
AUTOR PRÁCE

SUPERVISOR — RNDr. MAREK RYCHLÝ, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Bachelor's Thesis Specification

22370

Student:	**Kapišinský Marián**
Programme:	Information Technology
Title:	**Multi-Factor Authentication in Web Applications Using PAM**
Category:	Security

Assignment:

1. Study Pluggable Authentication Modules (PAM), focus on multi-factor authentication setups. Configure the multi-factor authentication for a common service (e.g., sshd) and analyse results. Study HTTP, focus on its state-less nature.
2. Investigate the possibility of using the full PAM stack in web applications, including multi-step conversations.
3. After agreement with the supervisor, develop a solution which would allow the use of the PAM conversation over the web. Create a prototype web application/setup to demonstrate the usage of the solution using FreeOTP.
4. Provide the documentation of the project, evaluate the results and discuss future work.

Recommended literature:

- Andrew G. Morgan, Thorsten Kukuk. The Linux-PAM System Administrators' Guide [online]. Version 1.1.2, 2010. [http://linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html]
- Jan Humpolík. Webová aplikace využívající vícefaktorovou autentizaci [online]. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. 2013. [http://hdl.handle.net/11012/20728]
- Liliana F. B. Soares, Diogo A. B. Fernandes, Mário M. Freire, Pedro R. M. Inácio. Secure user authentication in cloud computing management interfaces. IEEE 32nd International Performance Computing and Communications Conference (IPCCC), San Diego, CA, 2013. [https://doi.org/10.1109/PCCC.2013.6742763]

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 28, 2020
Approval date:	October 16, 2019

Abstract

The aim of this thesis is to implement multi-factor authentication using PAM for web applications. The thesis describes authentication and its modern trends, the related technologies and their incompatibility, as well as the state of authentication in web applications using PAM before the solution, the solution itself, and its integration to an example application. The thesis also provides relevant examples and guides.

Abstrakt

Cieľom tejto práce je implementácia viacfaktorovej autentizácie použitím PAM-u pre webové aplikácie. Práca popisuje autentizáciu a jej moderné trendy, súvisiace technológie a ich nekompatibilitu, ako aj stav autentizácie vo webových aplikáciách použitím PAM-u pred riešením, samotné riešenie a jeho integráciu do vzorovej aplikácie. Práca poskytuje aj príslušné príklady a príručky.

Keywords

web, application, security, multi-factor authentication, HTTP, WebSocket, HTML form, JavaScript, Node.js, N-API, addon, PAM

Kľúčové slová

web, aplikácia, bezpečnosť, viacfaktorová autentizácia, HTTP, WebSocket, HTML formulár, JavaScript, Node.js, N-API, addon, PAM

Reference

KAPIŠINSKÝ, Marián. *Multi-Factor Authentication in Web Applications Using PAM.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšírený abstrakt

[[TODO]]

Multi-Factor Authentication in Web Applications Using PAM

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý, Ph.D. The supplementary information was provided by Jan Pazdziora, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Marián Kapišinský
May 21, 2020

Acknowledgements

I would like to thank my supervision RNDr. Marek Rychlý, Ph.D. for support, feedback and guidance mainly in the formal aspect throughout the writing of my thesis. I would also like to thank my consultant Jan Pazdziora, Ph.D. for support, feedback and guidance mainly in the technical and language aspect throughout the writing of my thesis.

Contents

Introduction **3**

1 Pluggable Authentication Modules **4**
1.1 PAM Framework . . . 4
1.2 Configuring Multi-factor Authentication for SSHD . . . 6
1.3 PAM API - Essential Structures and Functions . . . 7
1.4 PAM-Aware Application . . . 8
1.5 PAM Authentication Module . . . 9
1.6 Advantages of Using PAM . . . 11

2 Authentication in Web Applications **13**
2.1 Authentication . . . 13
2.2 Third-Party Applications . . . 14
2.3 Password Strength . . . 14

3 Authentication using only HTTP **16**
3.1 Hypertext Transfer Protocol . . . 16
3.1.1 HTTP Message . . . 16
3.1.2 Session Management and Cookies . . . 17
3.2 Basic Authentication . . . 18
3.2.1 Disadvantages . . . 19
3.2.2 Setup on the Apache Web Server . . . 20
3.3 Form-based Authentication . . . 21

4 Current State of Authentication in Web Applications Using PAM **24**
4.1 Existing Solutions . . . 24
4.2 Configuration Using Apache Modules . . . 25
4.3 PAM Authentication Using WebSockets . . . 26
4.3.1 Node.js . . . 27
4.3.2 Example Using node-linux-pam . . . 27
4.4 Adding More Factors . . . 29

5 Multi-Factor Authentication in Web Applications Using PAM **31**
5.1 HTTP and PAM Incompatibility . . . 31
5.2 The Basis of the Solution . . . 32
5.3 node-auth-pam . . . 32
5.3.1 Example Usage . . . 36
5.4 The WebSocket Server . . . 37

5.5 The WebSocket Client 39
5.6 Integration to a Web Application 42

6 Conclusion 45
6.1 Future Work 45

Bibliography 47

A How to setup SSSD 50

B How to set up Google Authenticator 51

C CD Content 52

Introduction

With the increasingly advanced development of web technologies, there is also an increasing amount of threats on the Internet. Therefore, the demand for security in web applications is also rising. Single-factor authentication has rendered outdated, so new authentication mechanisms had to be developed. The main mechanism that is now becoming more and more popular, because of its higher security factor is multi-factor authentication. There already is a good number of its implementations, but there is yet no implementation using the Pluggable Authentication Modules for web applications. However, PAM and HTTP and inherently not compatible, so the use of newer technologies is necessary, namely WebSockets.

The chapter 1 describes Pluggable Authentication Modules framework, what is required from an application to be able to communicate with PAM, shows a simple PAM module implementation, describes how authentication using PAM works and demonstrates a multi-factor setup for Secure Shell Daemon.

The chapter 2 describes authentication in web applications, authentication factors, authentication issues with the third-party applications and discusses password strength.

The chapter 3 describes the relevant basics of the Hypertext Transfer Protocol, basic authentication and its disadvantages, and provides an example setup using the Apache web server, and finally the form-based authentication.

The chapter 4 describes several already existing solutions for authentication in web applications using PAM, provides relevant example configurations, describes the WebSocket protocol and Node.js, and provides an example setup, and finally describes the incompatibility issue of the existing solutions and multi-factor authentication.

The chapter 5 describes the HTTP and PAM incompatibility, the basis of the solution, the node-auth-pam addon, the server-side of the solution, the client-side of the solution, and the integration to a web application.

The appendix A provides a guide for configuring the SSSD service.

The appendix B provides a guide for configuring Google Authenticator.

The appendix C describes the content of the attached CD.

This thesis uses Fedora 31 for all examples, the implementation and its demonstration.

Chapter 1

Pluggable Authentication Modules

This chapter takes a look at the Pluggable Authentication Modules framework and its configuration in section 1.1, demonstrates a multi-factor authentication setup for SSHD in section 1.2, describes the requirements for applications that use PAM and authentication process in section 1.4, shows a simple authentication module implementation in section 1.5, and the advantages of using PAM in section 1.6.

1.1 PAM Framework

Pluggable Authentication Modules (PAM) is a common framework, that allows choosing how applications authenticate users. It is a suite of shared libraries, located in `/lib/security` or `/lib64/security`, called PAM modules, written in C. It can be configured to perform single-factor authentication or custom, more complex, authentication schemes (multi-factor authentication) for a wide variety of applications. These applications (also known as the "PAM-aware" applications) are written to be compatible with PAM, in C/C++ . It is typically used by many UNIX/UNIX-like operating systems (e.g. Linux, FreeBSD and Solaris) for user authentication (OS-level security) [4]. The framework is depicted in Figure 1.1.

PAM Configuration

The PAM configuration is located in a single central file at `/etc/pam.conf` or in multiple smaller files named after the service they relate to in `/etc/pam.d/`. It is a stack (also known as the "PAM stack") of required actions that must be completed to be authenticated. Each action is defined on a single line in a single configuration file for an application, which contains exactly one of the following: module type, control flag, module path and module arguments.

Module types

- auth - an action related to user authentication
- account - an action related to user account management

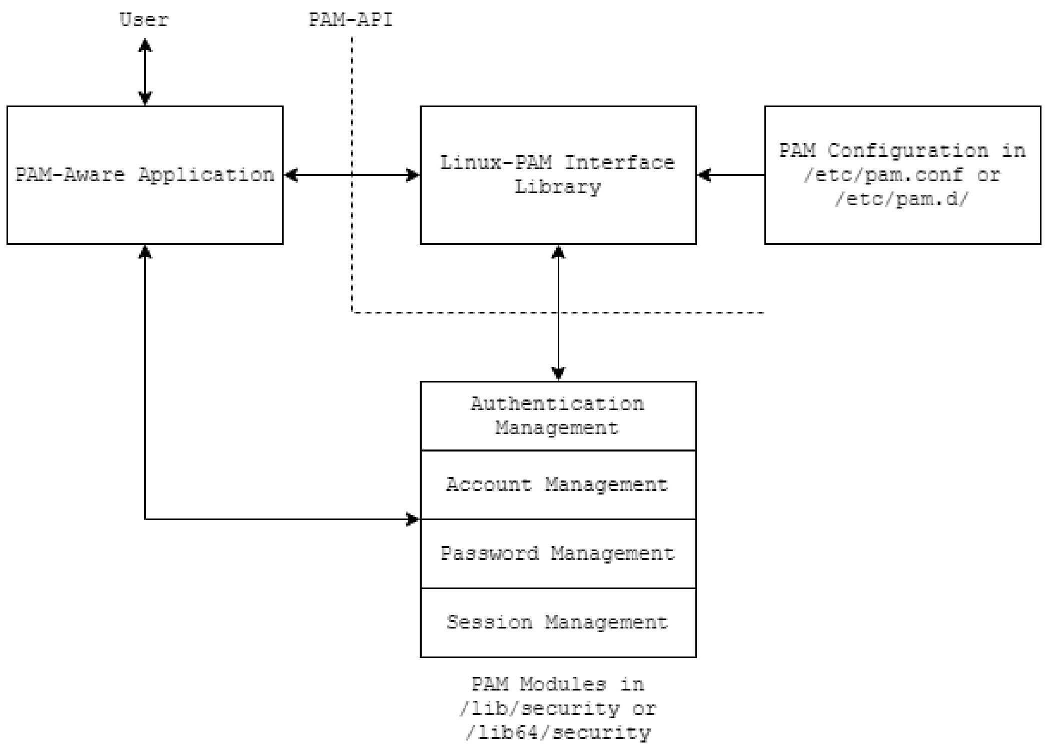

User
PAM-API
PAM-Aware Application
Linux-PAM Interface Library
PAM Configuration in /etc/pam.conf or /etc/pam.d/
Authentication Management
Account Management
Password Management
Session Management
PAM Modules in /lib/security or /lib64/security

Figure 1.1: PAM Framework

- session - an action related to connection or session management
- password - an action related to password management

Control flags

- requisite - the action must be completed successfully to continue processing actions, if not, no more actions are processed
- required - the action must be completed successfully, if not, the rest of the actions are processed, but the request will fail anyway
- sufficient - the action alone is enough to be accepted unless an earlier required actions has failed. If accepted, no further actions will be processed
- optional - if no other actions were successful, the success of this action will result into successful authentication

The module path provides PAM with either the name of the module or a relative path from the default module location (`pam_unix`, `pam_sss`, `pam_deny`, . . .).

Module arguments are used to pass information to a module that can modify the module's behavior.

This thesis only focuses on the *auth required* module configuration. This section was written according to The Linux-PAM System Administrators' Guide [1].

1.2 Configuring Multi-factor Authentication for SSHD

Configuring multi-factor authentication using PAM means including two or more *auth* type modules in the PAM stack in the configuration file for the service. For the demonstration, we have chosen the SSHD service, which supports the *keyboard-interactive* authentication that allows us to configure multi-factor authentication using PAM:

> *"Keyboard-interactive user authentication is intended primarily to accomodate PAM authentication on the server side. It provides for a multiple challenge-response dialog with the user in which the server sends a text query to the user, the user types in a response, and this process can repeat any number of times. So for example, you might configure PAM for SSHD with a module which performs authentication using an RSA security token, or a one-time password scheme."* [5]

Firstly, we need to configure SSHD (`/etc/ssh/sshd_config`) to use only the keyboard-interactive authentication. Find and comment out all lines with the **ChallangeResponseAuthentication** keyword and add a new line with the **AuthenticationMethods** keyword followed by the *keyboard-interactive* value (by doing this, we prevented that only the keyboard-interactive authentication will perform). Also, make sure that **UsePAM** is enabled. The configuration file should look like this:

```
#ChallangeResponseAuthentication yes
#ChallangeResponseAuthentication no
AuthenticationMethods keyboard-interactive
UsePAM yes
```

Now, we need to edit the PAM stack for SSHD at `/etc/pam.d/sshd` (backup the original file). We use `pam_sss`, `pam_reversed_login` and `pam_google_authenticator` modules in our example. The `pam_sss` module authenticates users against the System Security Services Daemon (SSSD). The advantage of using SSSD is that it has access to root privileges, which comes useful later in the chapter 4. The configuration file sssd.conf is attached in appendix A. It authenticates local users against /etc/shadow, but can be configured for other authentication providers, e.g. Active Directory or LDAP. The `pam_reversed_login` is an example module described in section 1.5. It requires the user to enter his reversed username (login). The `pam_google_authenticator` module supports HOTP[1] and TOTP[2] algorithms and can be easily used with the Google Authenticator mobile application. The installation and configuration guide is attached in appendix B.

Example configuration:

```
auth        required        pam_sss.so
auth        required        pam_reversed_login.so
auth        required        pam_google_authenticator.so
account     required        pam_sss.so
session     include         postlogin
```

[1]learn more at RFC 4226 – HOTP: An HMAC-Based One-Time Password Algorithm
[2]learn more at RFC 6238 – TOTP: Time-Based One-Time Password Algorithm

The last line of the configuration file has to be included for SSHD to work correctly. After we restart the SSH service (`systemctl restart sshd`) and run the `ssh` command to connect to a local user account with the first verbosity level (`ssh -v bob@localhost`), we can see how the authentication process proceeds:

```
debug1: Authentications that can continue: keyboard-interactive
debug1: Next authentication method: keyboard-interactive
Password:
Reversed login:
Verification code:
debug1: Authentication succeeded (keyboard-interactive).
```

The first thing we can see is that only keyboard-interactive authentication is enabled and can be performed, which is because we removed any other. Then, we can see that it asks us step by step for a password, our reversed login and an OTP token. This order depends on the order the *auth* type modules are in the configuration file. If we swapped the `pam_reversed_login` module and the `pam_google_authenticator` module, we would be asked for the OTP token before the reversed login.

1.3 PAM API - Essential Structures and Functions

For application and module development, the *pam-devel* package must be installed.

```
$ dnf install pam-devel -y
```

It contains all necessary header files (mainly, `<security/pam_appl.h>` for the application[3] development and `<security/pam_modules.h>` for the module[4] development), from which these structures and functions are essential for further reading:

- Structures
  - `struct pam_message { int msg_style; const char *msg; }`
  - `struct pam_response { char *resp; int resp_retcode; }`

  -
    ```
    struct pam_conv { int (*conv)( int num_msg,
                                   const struct pam_message **msg,
                                   struct pam_response **resp,
                                   void *appdata_ptr );
                      void *appdata_ptr; }
    ```
- Functions
  - `int pam_get_item( pamh, item_type, item)`

[3]learn more in The Linux-PAM Application Developers' Guide
[4]learn more in The Linux-PAM Module Writers' Guide

    - `int pam_get_user( pamh, user, prompt)`
- Application Development Functions
    - `int pam_start( service_name, user, pam_conversation, pamh )`
    - `int pam_authenticate( pamh, flags )`
    - `int pam_end( pamh, pam_status )`
- Module Development Functions
    - `PAM_EXTERN int pam_sm_authenticate( pamh, flags, argc, argv )`
    - `PAM_EXTERN int pam_sm_setcred( pamh, flags, argc, argv )`

1.4 PAM-Aware Application

The application is required to implement a conversation function, which is a callback that allows direct communication between a module and the application. This function is passed to a module in the `pam_conv` structure along with `void *appdata_ptr` that can pass any data defined by the application between the application and a module. The application creates the structure and passes it to the PAM framework as the `pam_conversation` argument of the `pam_start()` function.

The `pam_start()` function is called when the application requires user authentication. It initiates the PAM transaction with passed service name, username (if defined) and the `pam_conv` structure, loads the PAM configuration file for the service line by line in the order, they are specified and returns the PAM handle (`pam_handle_t *pamh`), which contains the loaded information (PAM context). If the first step succeeds, the calling application calls the `pam_authenticate()` function, which serves as an interface to the authentication mechanisms defined in the loaded modules. It calls every mechanism (`pam_sm_authenticate()` function) from each module in the order they were loaded from the configuration file. These modules pass their prompt(s) to the application and obtain user's response(s) using the passed conversation function. Each module either succeeds or fails, and the final result of the authentication process depends on the set control flags. The only exception, where not every module is called, is when a module with the requisite flag fails, then the authentication fails immediately. Lastly, when the authentication process finishes, the application calls the `pam_end()` function to terminate the transaction, and the handle and the context are no longer valid. The return value of the `pam_authenticate()` function call (or generally, the return value of the last PAM API call) is passed as the `pam_status` function argument of the `pam_end()` function, which in case of error informs PAM to perform an appropriate cleanup. The whole process is depicted in Figure 1.2. An example application can be found in [2].

Also, an important thing to mention explicitly is that with the start of a new transaction, a new process in the OS process table is created. While the transaction lives, the PAM handle structure is contained within this process. According to [2], it is also possible for an application to have multiple transactions in parallel.

1.5 PAM Authentication Module

The example module described in this section is a module used for testing and demonstration purposes in this thesis. It prompts the user for his reversed username, so it is called *pam_reversed_login*.

Firstly, to be correctly initialized, *PAM_SM_AUTH* must be **#define**'d before including the `<security/pam_modules.h>` header file, which contains the `pam_sm_authenticate()` and `pam_sm_setcred()` function prototypes that must be defined in the modules' source code.

```
#define PAM_SM_AUTH
#include <security/pam_modules.h>
```

Listing 1.1: Necessary Includes and Defines

Next, there are three helper functions: `setMessages()`, `doPamConv()` and `authenticate()`. The first function sets all four styles of messages in a given array of `pam_message` structures. The second function validates a received response from the user against his reversed username and returns either *AUTH_SUCCESS* or *AUTH_FAIL* on error. The third function uses the conversation function `conv()` from the `pam_conv` structure to send pre-configured messages in the `pam_message` structure to the PAM-Aware application and returns the received response through the `pam_response` structure double pointer. To obtain the conversation function, it calls the `pam_get_item()` function.

```
int doPamConv( pam_handle_t *pamh, int num_msg,
               const struct pam_message **msg,
               struct pam_response **resp ) {

   struct pam_conv *conv;

   int retval = pam_get_item(pamh, PAM_CONV, (void *)&conv);
   if (retval != PAM_SUCCESS) {
      return retval;
   }

   return conv->conv(num_msg, msg, resp, conv->appdata_ptr);
}
```

Listing 1.2: Function for Conversation with the PAM-Aware Application

Finally, there are the essential functions of the module. The `pam_sm_authenticate()` function is the module's implementation of the `pam_authenticate()` interface, which performs the authentication of the user. First, it gets his username using the `pam_get_user()` function (also defined in the PAM API). If the username was specified at the beginning of the transaction (`pam_start()`), it reads it from the PAM handle (`pamh->user`), otherwise it prompts the user using the conversation function. Next, it creates array of four `pam_message` structures and calls the `setMessages()` function to prepare the messages and assigns them to the `pam_message` structure double pointer. The four message styles are:

- `PAM_PROMPT_ECHO_OFF` - do not print text while obtaining the user's response
- `PAM_PROMPT_ECHO_ON` - print text while obtaining the user's response
- `PAM_ERROR_MSG` - display error message, no response is obtained
- `PAM_TEXT_INFO` - display some text, no response is obtained

It also prepares a pointer to the `pam_response` structure, where user's responses will be stored. Next, it obtains the responses by calling the `do_pam_conv()` and validates them with the `authenticate()` function. If any step of the validation fails the *PAM_AUTH_ERR* is returned, otherwise the module finishes with the *PAM_SUCCESS* return value.

```
PAM_EXTERN int pam_sm_authenticate( pam_handle_t *pamh,
                                    int flags,
                                    int argc,
                                    const char **argv ) {

    const char *login = NULL;
    char *reversed_login = NULL;

    if ( ( pam_get_user(pamh, &login, "Login: ") ) != PAM_SUCCESS )
        fprintf(stderr, "Can't get login\n");

    struct pam_message msg[4];
    const struct pam_message **msgp = NULL;
    struct pam_response *resp = NULL;

    setMessages(msg);

    ...

    int retval = doPamConv(pamh, 4, msgp, &resp);

    int status;
    for ( int i = 0; i < 2; i++ ) {

        if ( retval != PAM_SUCCESS || resp == NULL || resp->resp == NULL ) {

            fprintf(stderr, "Didn't get reversed login\n");
            return PAM_SYSTEM_ERR;
        }
        else {

            reversed_login = resp->resp;
        }

        status = authenticate(login, reversed_login);
        if (status == AUTH_FAIL)
            return PAM_AUTH_ERR;

        resp++;
    }

    free(msgp);
    return PAM_SUCCESS;
}
```

Listing 1.3: The Module's Authentication Function

The `pam_sm_setcred()` function is used to alter the credentials of a user. This function is not important for this thesis and always returns *PAM_SUCCESS*.

Installation of the module is done by execution of the following commands (root privileges are required):

```
$ gcc -fPIC -c pam_reversed_login.c
$ gcc -shared -o pam_reversed_login.so pam_reversed_login.o -lpam
$ cp pam_reversed_login.so /lib64/security/
```

The module was written according to the The Linux-PAM: Module Writers' Guide [3]. The entire source code with the installation script is attached in the appendix C.

1.6 Advantages of Using PAM

PAM allows application developers to implement PAM authentication to many different applications without creating or modifying PAM stacks. They can use the same stack for wide variety of applications if it suites their security needs. If not, PAM allows for high flexibility and control over the authentication. It is very easy to modify a PAM stack by adding, removing or editing one or several lines in the configuration file. They can either use already existing modules or develop a new one to suites their needs. [6]

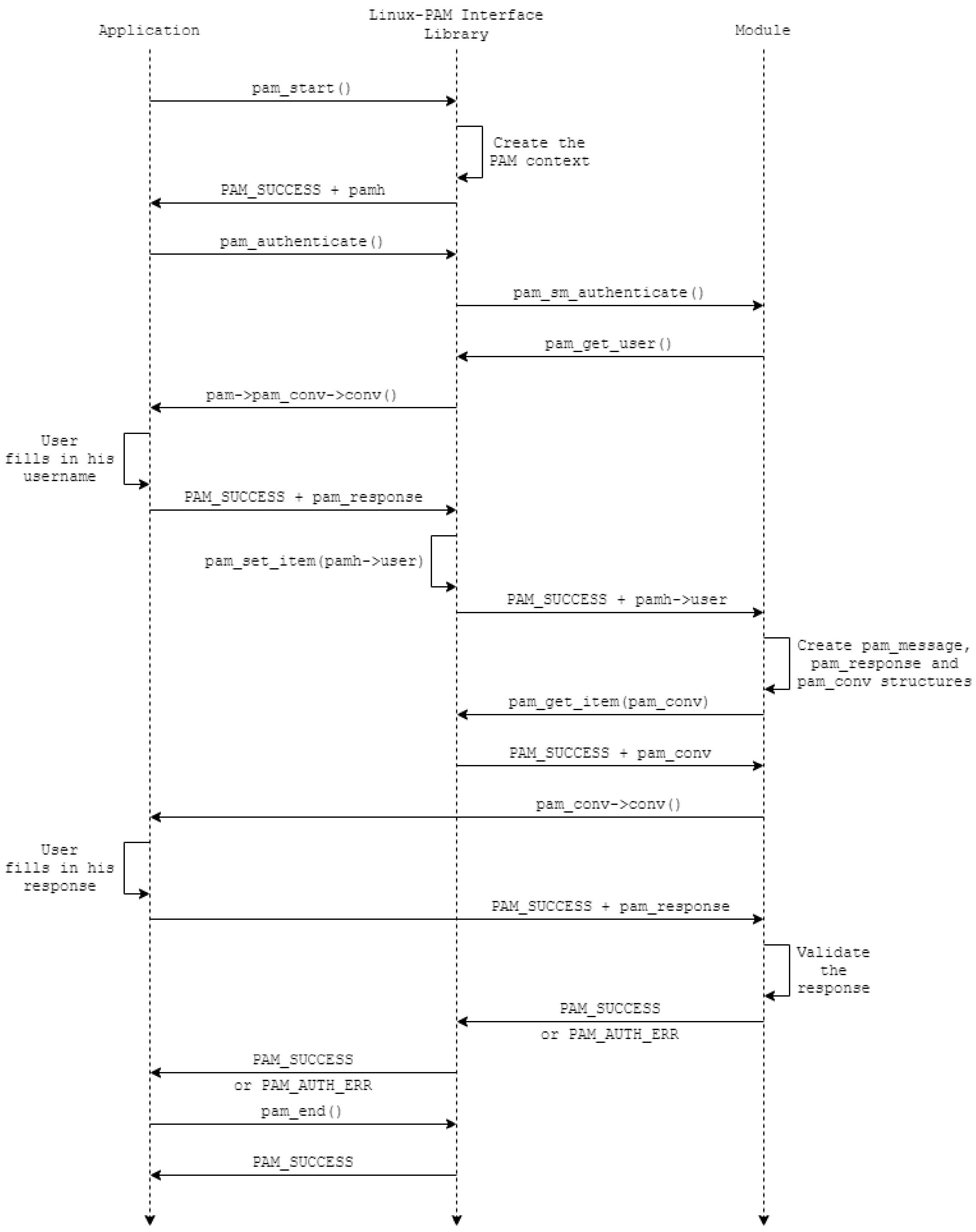


Figure 1.2: PAM Authentication Process

Chapter 2

Authentication in Web Applications

This chapter describes authentication in web applications and authentication factors in section 2.1, authentication issues with the third-party applications in section 2.2, and discusses password strength in section 2.3.

2.1 Authentication

Authentication is the process of verifying a user's identity using the required authentication factors. In order to get authenticated, the user must provide a factor he was asked for (challenge-response dialog). The factor is then verified by an authentication mechanism. On success, the user is authenticated and can use all the features of the application that he has access to. On failure, an error message or a page is displayed. The most common mechanisms are the password-based mechanisms (username and password). First, these mechanisms were used in single-factor authentication, where the only factor required is a password. Now, with the increasing number of threats on the Internet and with the degradation of the security of the single-factor authentication, it was necessary to develop new authentication mechanisms, namely multi-factor authentication mechanisms.

These mechanisms mostly appear in the form of two-factor authentication, where the second factor is required after standard password authentication. The most common mechanisms used as the second (or further) factor are one-time passwords (OTPs, also called OTP tokens or just tokens) that are either HMAC-based (HOTPs) or time-based (TOTPs). The password is generated by one of the algorithms and then delivered to a user via one of several technologies, such as quick response (QR) codes, short message service (SMS), trusted platform (TPM) or near field communication (NFC), [7]. The most common technologies are mobile applications (FreeOTP[1], Google Authenticator[2]) or hardware devices (YubiKey[3]) that generate OTPs locally.

[1]download at FreeOTP's GitHub

[2]download at Google Authenticator's Play store page

[3]buy at Yubico store page

Authentication Factors

As already said, the most common combination of factors for two-factor authentication is a password and an OTP. That is a combination of something we know and something we have. There are four different types of factors - *Something You Know* (passwords, PINs or security questions), *Something You Have* (OTPs, certificates, SMS, ...), *Something You Are* (face recognition, fingerprints, ...) and *Location* (source IP ranges, geolocation) [8]. We can make various combinations of the factors, but using only one type is not considered as multi-factor authentication.

2.2 Third-Party Applications

A problem with authentication comes with third-party applications, where an application (on desktop/mobile, other web application, ...) wants to connect to a web application. If we allowed the application to store our username and password, we would also provide it with more attack possibilities. For this reason, some new authentication protocols were developed, namely Open Authorization (OAuth), OpenID or the Universal Authentication Framework (UAF) protocol and the Universal Second Factor (U2F) protocol by The Fast Identity Online (FIDO) Alliance, [9]. The single sign-on (SSO) is also a trend in authentication in web applications, which allows users to use their identity in multiple web applications without the need for providing any authentication information (password, OTP, ...). The identity is validated and provided to applications by an Identity provider, e.g. Auth0, Google or OpenAthens, [10].

2.3 Password Strength

Password strength also must be mentioned in connection with authentication. Passwords should be at least 8 to 64 characters in length. All printable ASCII and Unicode characters including the space character should by acceptable by the applications. Dictionary words, repetitive or sequential characters (e.g. 'password', 'aaaaa', '1234abcd'), and context-specific words, such as the name of the service or the username should not be allowed. Randomly chosen secrets (e.g. PINs or OTPs) should by at least 6 characters long, [11]. Many applications also require the use of mix of upper-case and lower-case letters, numbers and symbols.

Many modern browsers (Google Chrome, Firefox, ...) have the option to fill in a randomly generated password when a user is choosing one, [12], [13]. They also offer the advantage of storing it to the user account if the user is logged into the browser or locally, so he does not need to remember it.

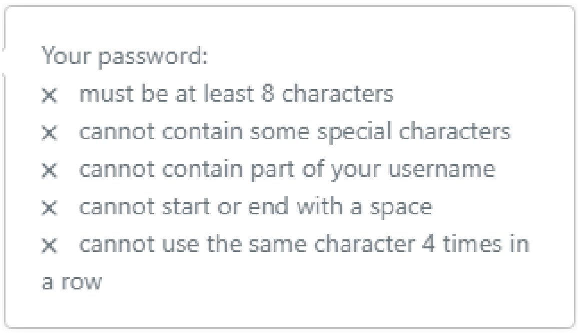
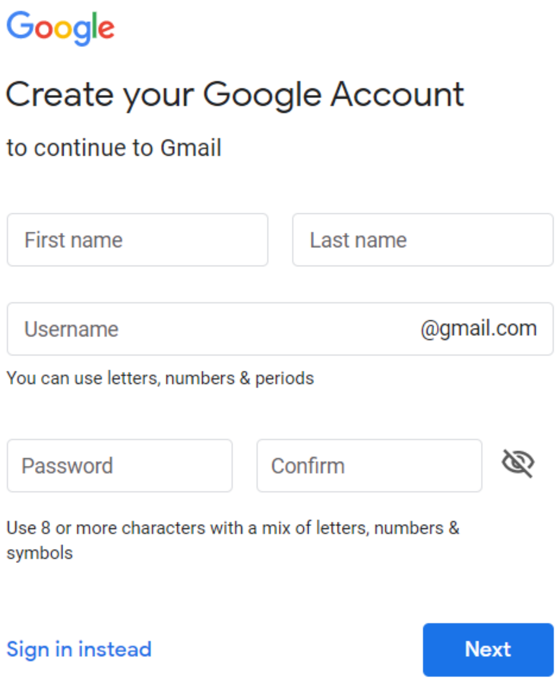

Your password:
× must be at least 8 characters
× cannot contain some special characters
× cannot contain part of your username
× cannot start or end with a space
× cannot use the same character 4 times in a row

Figure 2.1: Password Requirements of Alberta Student Aid



Create your Google Account

to continue to Gmail

First name | Last name

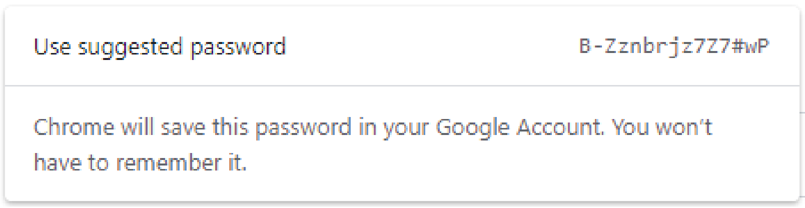
Username @gmail.com

You can use letters, numbers & periods

Password | Confirm

Use 8 or more characters with a mix of letters, numbers & symbols

Figure 2.2: Password Requirements of Google


Use suggested password B-Zznbrjz7Z7#wP

Chrome will save this password in your Google Account. You won't have to remember it.

Figure 2.3: Google Chrome's Password Suggestion

Chapter 3

Authentication using only HTTP

This chapter describes the relevant basics of the Hypertext Transfer Protocol in section 3.1, basic authentication and its disadvantages in section 3.2, and provides an example setup using the Apache web server in section 3.2.2, and finally the form-based authentication 3.3. For further studies on the Hypertext Transfer Protocol, visit the RFC 2616 standards document.

3.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a generic stateless application-level request/response protocol, which allows transfer of resources accessible by an URL (Uniform Resource Locator[1]) over the Internet, such as HTML documents. It is mainly used for web pages and applications, e.g. e-shops and internet banking. The communication via HTTP is client-server based. Client sends a HTTP Request message to the HTTP server, which parses the message, performs requested action and sends a response back to the client.

3.1.1 HTTP Message

As already mentioned, HTTP is a request/response protocol and it uses two types of HTTP messages: requests and responses. Both message types use the format of ARPA Internet Text Messages for transferring entities[2]. Each message consists of a start-line, zero or more header fields (headers), an empty line (blank line terminated with a CRLF) indicating the end of the header fields and a message body (if any). Each line is terminated with a CRLF.

Message Headers

The message header fields contain its name, followed by a colon (":") and its value. There are four types of header fields:

[1]learn more at RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax
[2]learn more at Standard for ARPA Internet Text Messages

- General - Connection, Date, Transfer-Encoding, etc.
- Request - Accept, Authorization, Host, User-Agent, etc.
- Response - Accept-Ranges, Server, WWW-Authenticate, etc.
- Entity - Allow, Content-Length, Content-Type, etc.

The focus of this thesis is authentication, so the only relevant headers are Request - *Authorization* and Response - *WWW-Authentication* headers, because they carry the authentication information.

Message Body

The message body contains the entity-body of a request or response. Not every message does include a message body. For authentication purposes, it serves no use, since all authentication information is transmitted in the headers.

HTTP Request

HTTP Request is a message sent from client to server. The first line of the message specifies the method to be performed on the target (GET, POST, etc.), the target (absolute path of an URL) and protocol version separated with spaces. The network location of the URL is transmitted in a Host header.

```
GET /example HTTP/1.1\r\n
```

HTTP Response

HTTP Response is a reaction to a HTTP Request. It informs the client about the result of his request. The first line of the message consists of the protocol version, a status code and its textural phrase (200 OK, 401 Unauthorized, etc.) separated with spaces.

```
HTTP/1.1 200 OK\r\n
```

3.1.2 Session Management and Cookies

Stateless behavior of HTTP means that every request is treated as a new one, independent of any previous requests from the communication partner. Neither the server nor the client retain any information about each other.

To make HTTP behave as a stateful protocol, a state management mechanism had to be developed. The RFC 6265 standard [14] implements Set-Cookie and Cookie headers. The server creates cookies and sends them in the Set-Cookie header to the client in a HTTP Response, the client stores them and sends them back to the server in the Cookie header in his further HTTP Requests.

Cookies must be stored locally on the server so they are accessible for all server processes, and not only for the process, that created it. If there were more servers on which the application runs, the server which created the cookies must share them with other servers. Otherwise, each server would create its cookies for the same user, and that is inefficient. All cookies are valid until they are deleted or until they expire (defined with the `Expires=<date>` field).

For example, for authentication purposes, after the user is authenticated, the server sends the cookie named SID (session identifier) with the value 31d4d96e407aad42 to the client, that uses it in his further requests, so he does not need to authenticate every time.

Set-Cookie (server to client):

```
Set-Cookie: SID=31d4d96e407aad42
```

Cookie (client to server):

```
Cookie: SID=31d4d96e407aad42
```

Set-Cookie with an expiration date (server to client):

```
Set-Cookie: SID=31d4d96e407aad42; Expires=Mon, 01 Feb 2021 12:34:58 GMT
```

To remove the cookie, the server can send a Set-Cookie header with the expiration date in the past:

```
Set-Cookie: SID=31d4d96e407aad42; Expires=Mon, 01 Feb 2020 12:34:58 GMT
```

The server can also instruct the client to return the cookie to every path and subdomain:

Set-Cookie (server to client):

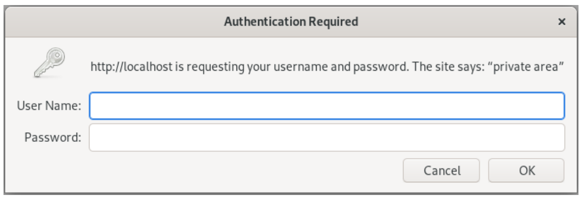
```
Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=myexampleapp.com
```

3.2 Basic Authentication

When a user accesses a web page or application that requires authentication, the browser creates a pop-up window (Figure 3.1). User fills in his username and password and hits the login button. The browser sends the credentials to the server, where they are validated. On success, the server responds with the page the user wanted to see and the client stores the credentials for future requests until the user closes the browser. Otherwise, the server responds with an error status code and an error page. What happens on the HTTP level is described in the following example (depicted in Figure 3.2):

1. Client sends an HTTP Request for the specified location:

   ```
   GET /basic-auth HTTP/1.1\r\n
   Host: localhost\r\n
   \r\n
   ```


Authentication Required

http://localhost is requesting your username and password. The site says: "private area"

User Name:

Password:

Figure 3.1: Pop-Up Window Displayed by Browser

2. Server responds with "401 Unauthorized" status code, what means authentication in required:

   ```
   HTTP/1.1 401 Unauthorized\r\n
   WWW-Authenticate: Basic realm="private area"\r\n
   \r\n
   ```

3. Client asks the user for a username and a password and sends the credentials in the Authorization header back to the server in another HTTP Request for verification:

   ```
   GET /basic-auth HTTP/1.1\r\n
   Host: localhost\r\n
   Authorization: Basic bWFyaWFuOnBhc3N3ZA==\r\n
   \r\n
   ```

   Credentials is *base64* encoded username:password, e.g. "marian:passwd" is encoded as `bWFyaWFuOnBhc3N3ZA==`.

4. Server validates the credentials and responds with either success or failure:

   ```
   HTTP/1.1 200 OK\r\n
   \r\n
   ```

   or

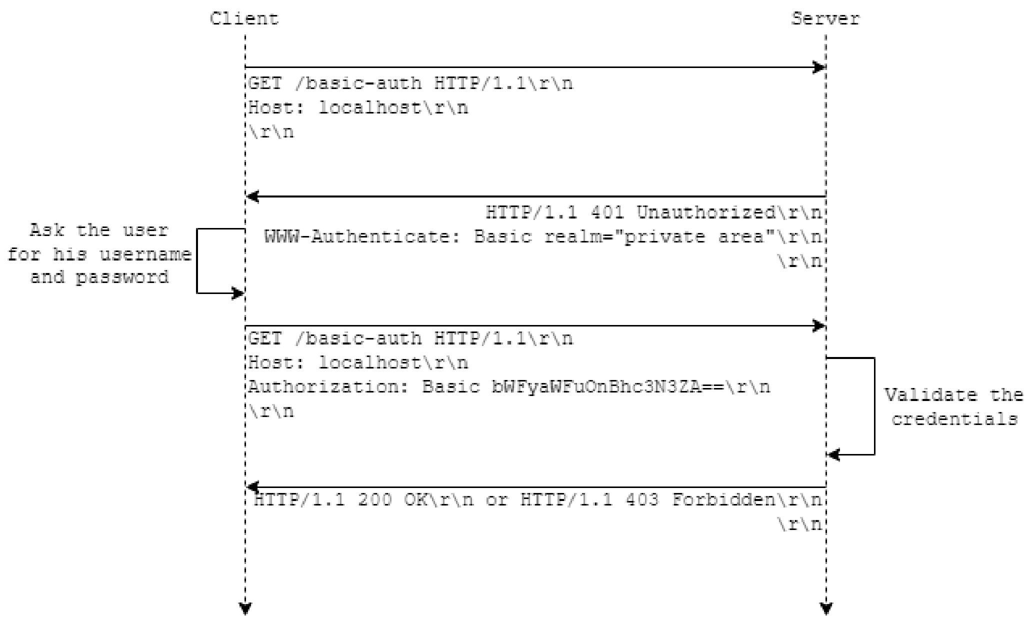
   ```
   HTTP/1.1 403 Forbidden\r\n
   \r\n
   ```

3.2.1 Disadvantages

Firstly, if a user requested the same page over and over, his credentials in the Authorization header would be validated with every request, which is inefficient. This problem is solved by implementing session management 3.1.2.

Secondly, it does not support account creation, so a user can not create a new account and it needs to be created on the server by its administrator. All usernames and passwords are stored locally on the server in a text file. It also does not support the logout option.

The next disadvantage is that basic authentication is only a single-factor. Additionally to that, credentials are only base64 encoded and not encrypted. It is also possible to use



```
GET /basic-auth HTTP/1.1\r\n
Host: localhost\r\n
\r\n
```

```
HTTP/1.1 401 Unauthorized\r\n
WWW-Authenticate: Basic realm="private area"\r\n
\r\n
```

```
GET /basic-auth HTTP/1.1\r\n
Host: localhost\r\n
Authorization: Basic bWFyaWFuOnBhc3N3ZA==\r\n
\r\n
```

```
HTTP/1.1 200 OK\r\n or HTTP/1.1 403 Forbidden\r\n
\r\n
```

Figure 3.2: HTTP-Level Communication for Basic Authentication

digest authentication instead. It uses hashes, which are stronger, but still vulnerable. Using HTTPS (HTTP over TLS/SSL) provides for the best security, as the credentials are being sent over an encrypted connection, but that's not the subject of this thesis.

Furthermore, using basic authentication is not visually modern. Because the pop-up window and native error page are not customizable, they are no longer used. The pop-up window was replaced by a login page with a form, that in case of error shows a custom-made error message with more user friendly information than the basic authentication's error pages contains.

3.2.2 Setup on the Apache Web Server

Firstly, we need to create a file, that stores usernames and passwords. That can be done by using `htpasswd` utility. To create the file you need to specify its location and a username (`htpasswd -c File username`), for example:

```
$ htpasswd -c /etc/basic-auth/passwd marian
```

It will ask us for a password. To add another user user the same command just without the `-c` option:

```
$ htpasswd /etc/basic-auth/passwd anotheruser
```

Then, we need to configure, which Location or Directory we wish to protect in Apache's configuration file. We can either edit the /etc/httpd/httpd.conf file or create a separate file in /etc/httpd/conf.d/filename.conf. Example configuration [15]:

```
<Location /basic-auth>
    AuthType basic
    AuthName "private area"
    AuthBasicProvider file
    AuthUserFile "/etc/basic-auth/passwd"
    Require valid-user
</Location>
```

Now, we have to restart Apache (`systemctl restart httpd`) and we can access the location via your preferred browser at localhost/private/. We can try to log in with correct and incorrect username or password and for more information we can look at httpd access and error logs. The access log (/var/log/httpd/access_log): contains information about all requests done to the server, for example:

```
::1 - marian [30/May/2019:00:22:33 +0200]
"GET /basic-auth HTTP/1.1" 200 36 "-" "Mozilla/5.0
(X11; Fedora; Linux x86\_64; rv:66.0) Gecko/20100101 Firefox/66.0"
```

The error log (/var/log/httpd/error_log): contains information about all errors, that occured on the server, for example:

```
[Thu May 30 00:40:44.041772 2019] [auth_basic:error]
[pid 5356:tid 140020740441856] [client ::1:52312]
AH01617: user marian: authentication failure for "/basic-auth":
Password Mismatch
```

3.3 Form-based Authentication

As already indicated in subsection 3.2.1, basic authentication is usually no longer used in modern web applications. It was replaced by the form-based authentication. Typically [16], when a user accesses an application's URL, the browser sends a GET request to the server, that hands the request to the application. If the application does not find a valid session cookie, the application redirects the browser to a login page with a login form created by the application. The user fills in his username and password and hits the submit button. The browser submits the form (sends a POST request with user's credentials), the server hands the credentials to the application, which typically calls an external application for their validation. On success, the application creates a session and return session cookies. The browser requests the desired URL again, but now the application sees a valid session cookie and returns the desired page. On failure, the application returns the login form and an error message.

POST Request Example with user's username and password:

```
POST /example/login HTTP/1.1
Host: localhost
login=marian&password=passwd
```

Unlike the basic authentication, the form-based authentication can support account creation, does not have to store usernames and passwords in local text files, and supports the logout options and custom-made page design. Everything depends on the application developer and can be configured to needs. However, in terms of security, it also does not use encryption, and it is the developer's responsibility to implement a safe solution, e.g. HTTPS.

```
<form method="POST">
    <dl>
        <dt><label for="login">Login:</label></dt>
            <dd><input type="text" name="login" />
        <dt><label for="password">Password:</label></dt>
            <dd><input type="password" name="password" />
        <dt><input type="submit" name="submit" value="Log in" /></dt>
    </dl>
</form>
```

Listing 3.1: Example HTML Code for a Login Form

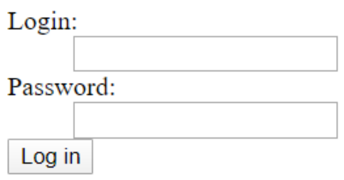


Figure 3.3: Login Form Displayed in a Browser

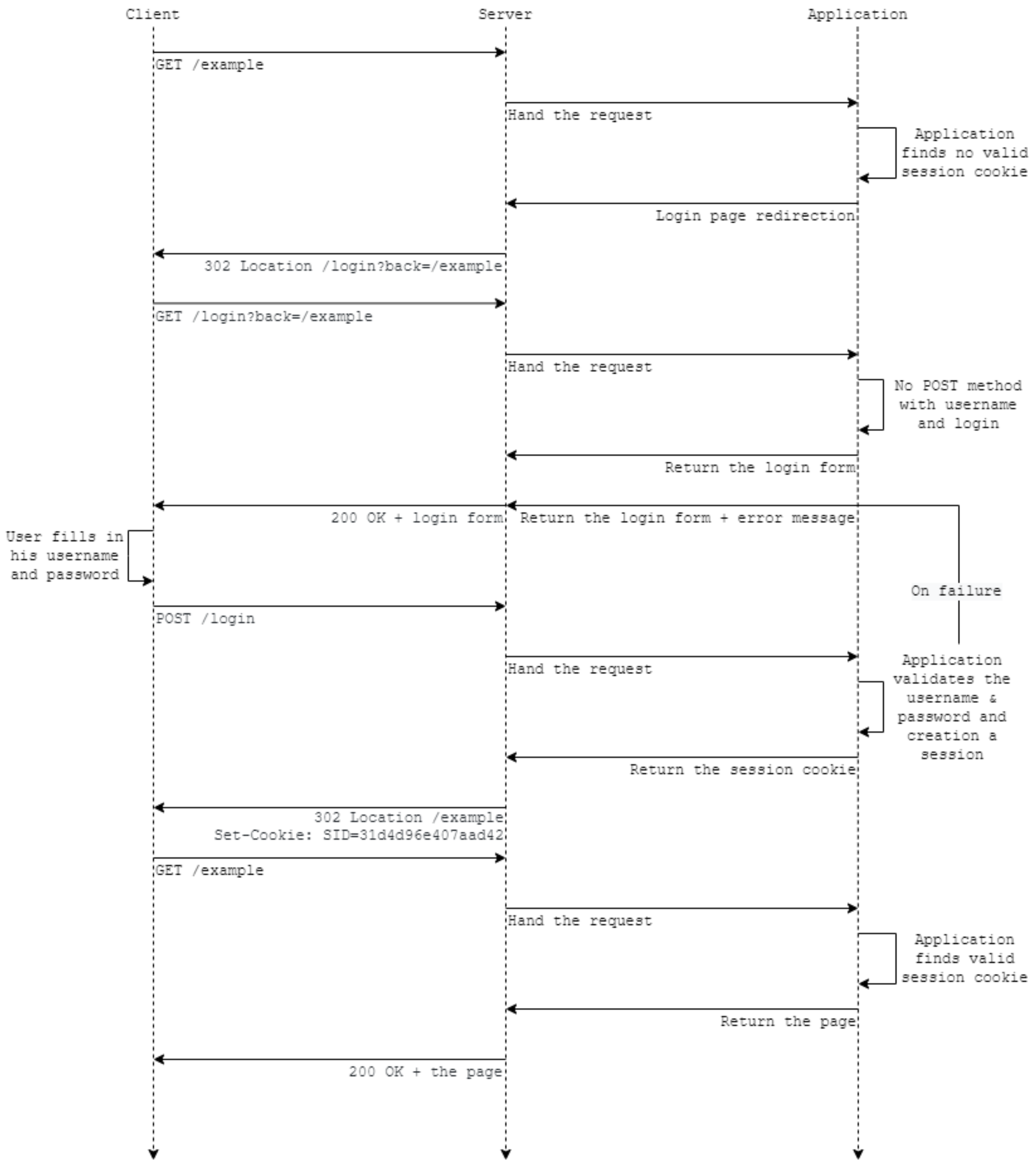

Client
Server
Application
GET /example
Hand the request
Application finds no valid session cookie
Login page redirection
302 Location /login?back=/example
GET /login?back=/example
Hand the request
No POST method with username and login
Return the login form
200 OK + login form
Return the login form + error message
User fills in his username and password
On failure
POST /login
Hand the request
Application validates the username & password and creation a session
Return the session cookie
302 Location /example Set-Cookie: SID=31d4d96e407aad42
GET /example
Hand the request
Application finds valid session cookie
Return the page
200 OK + the page

Figure 3.4: HTTP-Level Communication for Form-Based Authentication [16]

Chapter 4

Current State of Authentication in Web Applications Using PAM

This chapter describes several already existing solutions for authentication in web applications using PAM in section 4.1, the example configuration using mod_authnz_pam and mod_intercept_form_submit Apache modules in section 4.2. Next, it provides the introduction to the WebSocket protocol and Node.js, and an example setup using the Node.js WebSocket library and node-linux-pam addon in section 4.3. Finally, it describes the incompatibility issue of the existing solutions and multi-factor authentication in section 4.4.

4.1 Existing Solutions

There are already several solutions that bring PAM authentication to web applications. The first one to look at is the *mod_authnz_pam* Apache module that makes HTTP basic authentication work with PAM by obtaining username and password from the Authorization header of an HTTP request and running them through a PAM stack. The password is passed to a module in the `*appdata_ptr` member of the `pam_conv` structure. The module sets either the *REMOTE_USER* environment variable on successful authentication, or the *EXTERNAL_AUTH_ERROR* variable in case of an error. So basically, this module serves as an interface between the Apache web server and the PAM library. It can also supplement authentication done by other modules. For PAM, the mod_authnz_pam is a PAM-aware application. [17]

The next solution to look at is the *mod_intercept_form_submit* Apache module that intercepts submission of the application's login form, retrieves the username and password from the POST HTTP request, and calls the mod_authnz_pam module with those credentials. The application is expected to trust the *REMOTE_USER* value if it is set and skip its own authentication. [18]

The final solution to look at is the *node-linux-pam* addon for Node.js. With the use of the *WebSocket* protocol, it is possible to send collected username and password from the application's login form to a Node.js WebSocket server, run them through a PAM stack using the addon and send back the appropriate response using the opened WebSocket connection.

Since all three mentioned solutions pass the handling of authentication to another independent service (PAM), we refer to it as the *external authentication*.

4.2 Configuration Using Apache Modules

This section provides guides for configuring authentication using *mod_authnz_pam* and *mod_intercept_form_submit*. Seeing these configurations work and understanding principles of related modules, described in the previous section, is the first step to understanding the incompatibility of HTTP and PAM technologies. Just to remind, the Apache web server calls the PAM authentication directly using the mod_authnz_pam module.

mod_authnz_pam – example configuration

1. Install the module:

   ```
   $ dnf install mod_authnz_pam -y
   ```

2. Enable SELinux boolean *httpd_mod_auth_pam*:

   ```
   $ setsebool -P httpd_mod_auth_pam 1
   ```

3. Configure Apache in `/etc/httpd/conf.d/mod_authnz.conf`:

   ```
   LoadModule authnz_pam_module modules/mod_authnz_pam.so

   <Location /private>
     AuthType Basic
     AuthName "private area"
     AuthBasicProvider PAM
     AuthPAMService webapp
     Require valid-user
   </Location>
   ```

4. Create PAM stack for the webapp service in `/etc/pam.d/webapp`:

   ```
   auth        required        pam_sss.so
   account     required        pam_sss.so
   ```

   Now, the advantage of the SSSD service mentioned in the section 1.2 comes useful, because Apache does not run with the root privileges, so it could not access the `/etc/shadow` file if the `pam_unix`[1] module was used instead.

5. Restart Apache and access `http://localhost/private` location from a browser

[1] learn more in the pam_unix module guide

mod_intercept_form_submit – example configuration

1. Install the module and perl-CGI:

   ```
   $ dnf install mod_intercept_form_submit perl-CGI -y
   ```

2. Set up the example app:

   ```
   $ curl -Lo /var/www/app.cgi 'http://fedorapeople.org/cgit/\
     adelton/public_git/CGI-sessions.git/plain/app.cgi\
     ?id=intercept-form-submit'
   $ chmod a+x /var/www/app.cgi
   $ dnf install /usr/sbin/semanage -y
   $ semanage fcontext -a -t httpd_sys_script_exec_t \
     '/var/www/app\.cgi'
   $ restorecon -rvv /var/www/app.cgi
   ```

3. Configure Apache in /etc/httpd/conf.d/webapp_intercept.conf:

   ```
   LoadModule intercept_form_submit_module modules/\
   mod_intercept_form_submit.so

   ScriptAlias /app /var/www/app.cgi

   <Location /app/login>
     InterceptFormPAMService webapp
     InterceptFormLogin login
     InterceptFormPassword password
   </Location>
   ```

4. Restart Apache and access the app from a browser at `http://localhost/app`

4.3 PAM Authentication Using WebSockets

With the development of new web technologies, there is also a lot more new possibilities for the web application development. The *WebSocket* protocol [19] provides for bidirectional, full-duplex communication between client and server. That means the client and the server have an open connection and can send messages back and forth. So, it is possible to collect username and password from a login form on a login page using the client-side *JavaScript*, send them to the server via the opened connection, validate them using PAM and send back the appropriate response. For better security, using WebSockets over TLS/SSL is recommended. The example in subsection 4.3.2 uses the WebSocket library[2] and the node-linux-pam[3] addon for *Node.js*.

[2]learn more in the Node.js WebSocket library repository
[3]learn more in the node-linux-pam addon repository

4.3.1 Node.js

Node.js is a free open source server-side JavaScript development and runtime environment that uses asynchronous, event-driven, single-threaded, and non-blocking programming designed for highly scalable network applications. While it supports the development of any server executing any application-level protocol running over TCP/UDP, it found its biggest use case in the web application development. It is used by many modern web applications, such as PayPal, LinkedIn, or eBay. [20]

According to [21], Node.js is highly advisable for building modern web applications that use dynamic page content. It can handle a much larger number of concurrent connections than the Apache web server and is more memory efficient and better in utilizing all available processing power than PHP. However, it lacks in serving static files using its built-in HTTP server.

Install the latest version at the time and all needed dependencies by executing the following commands:

```
$ dnf install gcc-c++ make
$ curl -sL https://rpm.nodesource.com/setup\_14.x | sudo -E bash -
$ dnf install nodejs
```

Node.js Addons

Addons for Node.js are dynamically-linked shared objects written in C/C++. There are three options for implementing addons: *N-API* (or node-addon-api, which is C++ wrapper for N-API), *nan* or direct use of the internal V8 JavaScript engine, libuv, and Node.js libraries. The recommended option is using N-API as it newer and easier to use. Other options should be used only in need for functionality that is not provided by N-API. [22]

4.3.2 Example Using node-linux-pam

Firstly, we need to create the WebSocket server using the Node.js WebSocket library. The server listens on a specified port and waits for a client to connect. When a client connects and sends a message, the server parses it to obtain username and password and hands them to the node-linux-pam addon in the `pamAuthenticate()` function argument (an object containing all necessary data). The addon runs them through the specified PAM stack. The password is passed to a module in the `*appdata_ptr` member of the `pam_conv` structure. We use the webapp PAM stack from the previous section. When the authentication process finishes, the callback function of the `pamAuthenticate()` function is called. Using the WebSocket library, the appropriate response is sent back to the client via the opened connection. The message is expected to be in the “username:password” format.

```
// Load the WebSocket library and the node-linux-pam addon
const WebSocketServer = require('ws').Server;
const { pamAuthenticate, pamErrors } = require('node-linux-pam');
```

```
// Prepare the object for the authentication data
var options = { username: '', password: '', serviceName: 'webapp'};

// Create the WebSocket server
const wss = new WebSocketServer({ port: '1234' });

// Callback function for "on connection" event
wss.on('connection', function(ws) {

  // Callback function for "on message" event
  ws.on('message', function(message) {

    //Parse the data from the client's message
    var cred = message.split(':');
    options.username = cred[0];
    options.password = cred[1];

    // Call the addon and send the appropriate response
    pamAuthenticate(options, function(err, code) {

      if(err) {
        ws.send(JSON.stringify({"message": err}));
      } else {
        ws.send(JSON.stringify({"message": "OK"}));
      }
    });
  });
});
```

Listing 4.1: Simple WebSocket Authentication Server Using node-linux-pam in Node.js

Next, we need to create the WebSocket client using the client-side JavaScript. It initiates a connection with the server, collects username and password from a login form, puts them to the required format and sends them to the server. When the response is received, the client parses it and displays it to the user.

```
// Connect to the WebSocket server
var ws = new WebSocket('ws://localhost:1234');

// Callback function that parses the server's response,
// displays it to the user and closes the connection
ws.onmessage = function(e) {
    var status = JSON.parse(e.data);
    $("#status").text(status.message);
    ws.close();
}

// Collect username and password, and send them to the server
```

```
    function sendUserInput() {
        var cred = $('#login').val() + ':' + $('#passwd').val();
        ws.send(cred);
    }
```

Listing 4.2: Simple WebSocket Client in JavaScript

Finally, for the demonstration, we need to create a simple HTML login page with a form.

```
<!DOCTYPE html>
<html>
  <head>
    <title>PAM Authentication</title>
    <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
    <script src="login.js"></script>
  </head>
  <body>
    <h1>Log In</h1>
    <form onsubmit="sendUserInput(); return false;">
      <input id="login" type="text" />
      <input id="passwd" type="text" />
      <button type="button" onclick="sendUserInput();">Send</button>
    </form>
    <h2 id="status"></h2>
  </body>
</html>
```

Listing 4.3: Simple HTML Page with a Login Form and the WebSocket Client Script

1. Run the server script using the `node` command:

   ```
   $ node main.js
   ```

2. Put the login script and the web page inside `/var/www/html/` directory to make it accessible from a browser using Apache and restart it

3. Access the page from a browser, fill in username and password, hit the Send button and a response message from the server should appear bellow the form

4.4 Adding More Factors

So far, each one of the solutions used the PAM stack configured only for single-factor authentication using the `pam_sss` module. What all these solutions have in common, is that they pass the password to a module in the `*appdata_ptr` member of the `pam_conv` structure. Therefore, they do **not** support multi-factor authentication, because only the first module in a multi-factor stack would get the password, and other modules would produce an error message. The reason is that the conversation functions of both mod_authnz_pam and node-linux-pam cannot send any message to the user nor receive any response. For example, adding the `pam_reversed_login` module to the stack and trying to login again would in case of mod_authnz_pam or mod_intercept_form_submit configuration cause the "Didn't get reversed login" error message to appear in the Apache error log. In case of the WebSocket solution, the "Authentication failure" error message would appear bellow the form.

For a better understanding of how the password is passed to a module, Listings 4.4 and 4.5 show the relevant part of mod_authnz_pam source code[4] and Listings 4.6, and 4.7 show the relevant part of node-linux-pam source code[5]. It is the same in principle; only node-linux-pam uses its `auth_context` data type, which carries the authentication data of the `PamWorker` class instance.

```
    struct pam_conv pam_conversation = { &pam_authenticate_conv,
                                         (void *) password };
```

Listing 4.4: pam_authenticate_with_login_password()

```
    response[i].resp = strdup(appdata_ptr);
```

Listing 4.5: pam_authenticate_conv()

```
    const struct pam_conv local_conversation = {function_conversation,
                              reinterpret_cast<void *>(authContext)};
```

Listing 4.6: PamWorker::Execute()

```
    auth_context *data = static_cast<auth_context *>(appdata_ptr);
    reply->resp = strdup(data->password.c_str());
```

Listing 4.7: PamWorker::function_conversation()

Due to the inherent incompatibility of HTTP and PAM, it is not possible to extend the Apache modules to support multi-factor authentication. That is why the WebSocket protocol was introduced in this chapter. The explanation of the incompatibility problem is provided at the start of the next chapter.

[4]the mod_authnz_pam source code
[5]the node-lixus-pam source code

Chapter 5

Multi-Factor Authentication in Web Applications Using PAM

This chapter describes the HTTP and PAM incompatibility in section 5.1, the basis of the solution in section 5.2, the node-auth-pam addon in section 5.3, the server-side of the solution in section 5.4, the client-side of the solution in section 5.5, and the integration to a web application in section 5.6.

5.1 HTTP and PAM Incompatibility

In section 1.2, the SSHD service was used as an example for multi-factor authentication using PAM. It is an SSH server running as a background process. Unlike HTTP, SSH protocol supports bidirectional full-duplex connection, so the client and the server have an opened connection through which they can send data back and forth. When the connection is established, the client has to authenticate itself to the server [23]. Assuming the SSHD configuration from the section 1.2, the server starts a new thread for authentication against PAM. When a PAM module requires communication with the client for obtaining necessary information from the user, the conversation function uses the opened connection for both sending messages and obtaining responses (if any is expected). When the communication is done, the conversation function returns all responses (if any) to the calling module [24]. Another difference between SSHD and HTTP is that both client and server randomly generate a session ID, which they keep for themselves and use it to identify a session uniquely. In HTTP, the session ID is sent in each request [23].

So, both SSHD and a PAM transaction are running processes, and SSHD uses an open connection for transmitting all necessary data. However, in HTTP, there is no open connection between a client and a server because the protocol is request/response-based. So, it would be necessary to send the current message to the client, store the transaction state, load it back when the client sends a response to the message (in an HTTP request), and proceed with authentication. According to [2], the PAM handle contains the state entirely, however it is not absolutely true, because the `pam_conv` structure contains `void *appdata_ptr`, which is a pointer to any application-defined data. Therefore, it is not possible to serialize the PAM handle structure, store it and load it back.

However, it is possible to implement a PAM authentication *addon* for Node.js using N-API and with the use of the *WebSocket* protocol to synchronize the state of a PAM transaction with the content of a login page by transmitting all necessary data using the opened WebSocket connection between the client and the server and dynamically adjust the page content using JavaScript.

There are also other environments that support the WebSocket protocol, but after the agreement with the consultant, Node.js will be used for the implementation of the solution due to its advantages, popularity, and high accessibility.

5.2 The Basis of the Solution

The basis of the solution is to create an authentication thread for each client connected to the WebSocket server. The thread starts the PAM transaction, calls for authentication, and finally ends the transaction. When a module calls the conversation function:

1. The conversation function passes the current message to the WebSocket client and waits for a response (if any is expected),
2. the client displays the message on the login page to the user, collects and sends his response back to the conversation function (if any is expected),
3. if the module has more than one message defined, the conversation function sets the next message as the current message and repeats the process from 1.,
4. otherwise it returns all responses (if any) to the calling module.

After the authentication is done, the server sends a message with the return value to the client. The client displays status information based on the return value. If the authentication was successful, a session cookie is also sent along with the return value. The client sets the cookie and issues a redirect to the configured page. The cookie contains a session ID (SID) and Expires date (the current date + one day). The session ID is a randomly generated *base64* encoded 16-byte string. The session ID is also stored in a file along with the corresponding username in the "SID::username" format. Each session (a file line) has a timeout set to one day. After the timeout, the session is deleted. If the authentication failed, the client allows the user to try to authenticate again.

The solution consists of three parts, namely the PAM authentication addon – *node-auth-pam*, the WebSocket server, and the WebSocket client. The Figure 5.1 describes the solution using a finite-state machine.

5.3 node-auth-pam

The *node-auth-pam* addon accommodates PAM authentication in Node.js. It is written in C using the N-API library for the Node.js addon creation. It provides the `nodepamCtx` structure (referred to as "context" for the rest of this chapter) wrapped to a JavaScript object and necessary getters for its members that need to be accessible from Node.js and

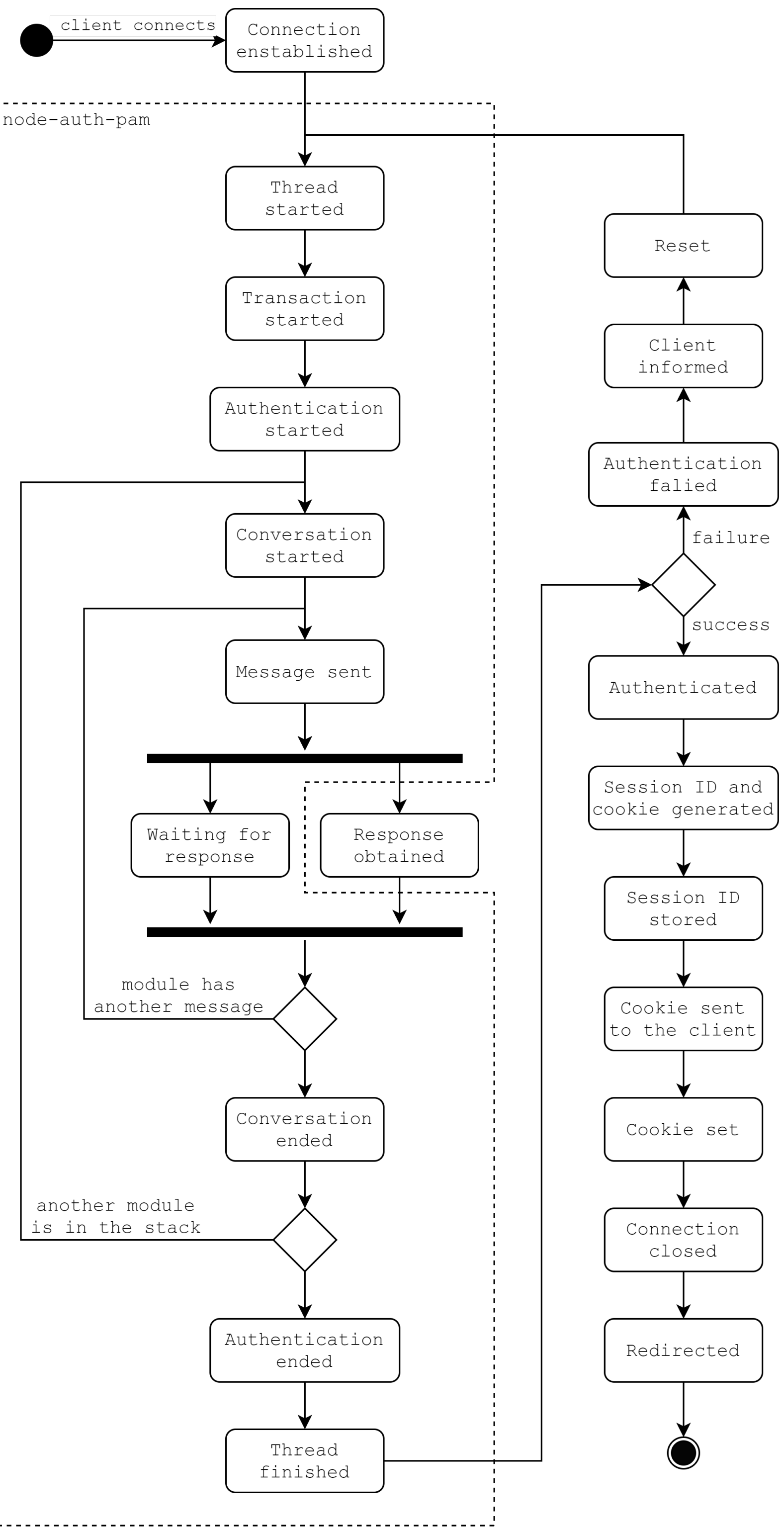


Figure 5.1: The Solution Described by FSM

several functions (also called bindings) that can be called from Node.js. It can be used by any Node.js application that desires authentication against PAM.

The context is a structure that contains all necessary data needed for authentication:

- service - name of the service as defined in `/etc/pam.d/`
- username - name of the user
- message - the current message
- msgStyle - the style of the message
- response - the user's response
- respFlag - true, if the user's response is set
- retval - the return value of PAM authentication, also used for addon's constants – *NODE_PAM_JS_CONV* and *NODE_PAM_ERR*
- thread - the authentication thread
- mutex - the mutex protecting response and respFlag
- tsfn - the thread-safe function

The bindings provided by node-auth-pam:

- `authenticate(service, username, callback(nodepamCtx))`
- `setResponse(nodepamCtx, response)`
- `terminate(nodepamCtx)`
- `cleanUp()`

The `callback` argument of the `authenticate()` binding is a callback function that provides a way for the addon's conversation function to pass the handling of authentication and the context to Node.js using the N-API *thread-safe function*. The thread-safe function [25] is an asynchronous call of a given JavaScript function from additional threads of an addon. The `call_js_cb` argument of `napi_create_threadsafe_function()` allows for more control over the actual call of the JavaScript function. It is a callback function invoked on the addon's main thread every time the thread-safe function is called from a thread. This callback function allows for wrapping the context structure to a JavaScript object and passing it to Node.js. All necessary members of the context can be then accessed by using getter functions defined by the addon:

- user - returns the username
- msg - returns the current message
- msgStyle - returns the style of the message

- retval - returns the return value or *NODE_PAM_JS_CONV* (there is no use case for *NODE_PAM_ERR* as it is only used internally)

The callback function of the `authenticate()` binding must be defined and can be as simple as show in the example 5.3, or can implement more complex logic depending on the state of the context. It is invoked one or multiple times (depends on number of modules in the PAM stack and number of module's messsages) and finally after the transaction is finished.

When a Node.js application requires authentication using the node-auth-pam addon, it calls the `authenticate()` binding. It creates a new context with the service name and the username, creates a thread-safe function of the binding's callback, and starts the authentication thread with the context set as its attribute. Both thread-safe function and the thread are also stored in the context. The thread creates the `pam_conv` structure with the addon's conversation function and passes the context using `*appdata_ptr`, starts the PAM transaction and calls the `pam_authenticate()` function.

When a module calls the conversation function (`nodepamConv()`), it sets `message` to the current message of the module, `msgStyle` to its style, and `retval` to *NODE_PAM_JS_CONV* in the context and uses the thread-safe function from the context to invoke the callback, passing the context to Node.js and waits for a response (the waiting mechanism is shown in the Listing 5.1). The `retval` indicates that the conversation function is waiting. The callback function of the `authenticate()` binding displays the message to the user and obtains a response (`setResponse()` called inside the callback). It is also possible to store the context to a variable and call the `setResponse()` binding outside the callback.

```
while(true) {
    pthread_mutex_lock(&(ctx->mutex));
    if (!ctx->respFlag) {
        pthread_mutex_unlock(&(ctx->mutex));
        continue;
    } else {
        response[i].resp = strdup(ctx->response);
        response[i].resp_retcode = 0;
        pthread_mutex_unlock(&(ctx->mutex));
        break;
    }
}
```

Listing 5.1: Waiting Mechanism of nodepamConv()

The waiting mechanism of the conversation function is not very effective due to the `while()` cycle as it unnecessarily consumes the CPU. It continuously checks `respFlag` until it is set to true, and sets the obtained response to the `pam_response` structure. It would be much more effective if the thread would go to sleep and then be awakened by a *SIGCONT* signal. However, after many attempts it did not work due to undiscovered reason.

The `setResponse()` binding protects the setting of the response (shown in the Listing 5.2) with a `mutex`, so `response` and `respFlag` cannot be accessed by the conversation function until they are both set. In case of *PAM_ERROR_MSG* or *PAM_TEXT_INFO* message styles, the `setResponse()` binding must be called with an empty string due to synchronization issues. Otherwise, it sets the obtained response to `response` and `respFlag` to *true*.

```
pthread_mutex_lock(&(ctx->mutex));

...

if (ctx->msgStyle == PAM_PROMPT_ECHO_OFF ||
      ctx->msgStyle == PAM_PROMPT_ECHO_ON )
    ctx->response = strdup(response);

ctx->respFlag = true;
pthread_mutex_unlock(&(ctx->mutex));
```

Listing 5.2: Setting of the Response to the Context

When `retval` in the context is set to *PAM_SUCCESS*, the authentication was successful and the application can implement a post authentication mechanism, e.g. a session management, or pass the handling to another service. Any other return value is an error that the application can handle according to it needs.

When the authentication finishes, the authentication thread ends the PAM transaction, sets the return value of `pam_authenticate()` to `retval` in the context, calls the thread-safe function for the last time to invoke the callback and pass the final return value to Node.js, and releases the thread-safe function (`napi_release_threadsafe_function()`). The release invokes a finalize callback, which can provided upon the creation of the thread-safe function. It is invoked on the addon's main thread after the thread-safe function is released and provides an opportunity for cleaning up after the thread(s). The addon implements the `ThreadFinished` finalize callback, which terminates (`pthread_join()`) or kills the thread, and frees the context.

The addon also provides `terminate()` and `cleanUp()` bindings. The `terminate()` binding can be called from Node.js to kill the authentication thread, if an error occurs during the authentication process on the Node.js side. The `cleanUp()` binding should be called when the Node.js application is about to finish to prevent some memory leaks.

The addon was written according to the thread-safe function round-trip example provided by one of the Node.js developers Gabriel Schulhof [26].The entire source code is attached in the appendix C.

5.3.1 Example Usage

This section provides an example usage of the *node-auth-pam* addon. The example application prompts the user for his username and runs the authentication. When the callback function of the `authenticate()` binding is invoked, it firstly checks if `retval` is *NODE_PAM_JS_CONV*. If it is, it checks if the `msgStyle` is set to *PAM_ERROR_MSG* or *PAM_TEXT_INFO*, prints the message and calls `setResponse()` with an empty string.

Otherwise, it prompts the user for a response according to `message` and sets the response. If `retval` is set to *PAM_SUCCESS*, it gets the username from the context and prints that the user was authenticated. Otherwise, it prints an error message.

```
const pam = require('bindings')('auth_pam'); // load the addon
const readline = require('readline-sync');

const PAM_SUCCESS = 0;
const PAM_ERROR_MSG = 3;
const PAM_TEXT_INFO = 4;
const NODE_PAM_JS_CONV = 50;

var username = readline.question('Username: ');

pam.authenticate('nodeapp', username, (nodepamCtx) => {
    if (nodepamCtx.retval === NODE_PAM_JS_CONV) {
        if (nodepamCtx.msgStyle === PAM_ERROR_MSG ||
                nodepamCtx.msgStyle === PAM_TEXT_INFO) {
            console.log(nodepamCtx.msg);
            pam.setResponse(nodepamCtx, '');
        } else {
            var response = readline.question(nodepamCtx.msg);
            pam.setResponse(nodepamCtx, response);
        }
    } else if (nodepamCtx.retval === PAM_SUCCESS) {
        // Authentication succeeded, do something
        console.log('User ' + nodepamCtx.user + ' authenticated');
    } else {
        // Authentication failed, do something
        console.log('Authentication failed');
    }
});
```

Listing 5.3: Example of node-auth-pam Usage

The test PAM stack *nodeapp* uses `pam_sss` and `pam_reversed_login` modules.

```
auth        required        pam_sss.so
auth        required        pam_reversed_login.so
account     required        pam_sss.so
```

5.4 The WebSocket Server

The WebSocket server serves as the authentication *daemon* and uses the node-auth-pam addon to authenticate users against PAM. It listens on a given port and waits for clients to connect. When a client connects, the server declares a variable `ctx` for storing the context.

When the client send his first message, the server expects it to be a username. Since it is the clients first message, no context yet exists, so the server calls the `authenticate()` binding that starts the authentication thread. When the callback of the `authenticate()` binding is invoked, the server sends the current message from the context to the client using the opened connection and stores the to declared variable `ctx`. If the style of the message is either *PAM_ERROR_MSG* or *PAM_TEXT_INFO* it calls the `setResponse()` binding with an empty string. Now, the server waits for another message from the client. Since the client now has its context, all other messages received from now on are expected to be responses. So every time the server receives a message from this client, it calls the `setResponse()` binding to set the response to the context, so the waiting conversation function access it. While `retval` in the context is set to *NODE_PAM_JS_CONV* the process of sending messages and setting responses to them continues until all modules satisfy their needs.

When the `retval` changes, the authentication finished, the server sends the actual return value to the client and the `ctx` is cleared. If the authentication succeeded, the server generates a session cookie and sends it to the client along with the return value. It also appends the generated session ID to a file (a sessions file) named after the service in the "SID::username" format and sets a one-day timeout, after which the session is deleted from the file. The file contains session IDs and corresponding usernames of all authenticated users. It is located in the `sessions/` directory, which is located in the root of the package. The web application is then expected to trust this file and validate session IDs against it. If the authentication failed, no session ID and cookie are generated, and another message from the client is assumed as the first message, so the user can try again to authenticate. Example session cookie:

```
SID=WS7ec7twsOptU5aQ6zVEcQ==; Expires=Fri, 29 May 2020 19:20:01 GMT
```

If the connection between the client and the server closes due to any reason, the server calls the `terminate()` binding to kill the running authentication thread. Finally, when the server is about to shutdown (due to an interrupt signal), it calls the `cleanUp()` binding.

Since, Node.js the WebSocket library allows for multiple concurrent connections and it is also possible to have multiple PAM transactions in parallel, the server provide authentication for multiple clients simultaneously. Each connected client has exactly one thread and exactly one context.

The Figure 5.2 shows the sequence diagram of the server-side of the solution. The entire source code of the WebSocket server is attached in the appendix C.

```
wss.on('connection', (ws) => {

  var ctx;

  ws.on('message', (message) => {

    if (!ctx) {
      pam.authenticate(service, message, (nodepamCtx) => {
        if (nodepamCtx.retval === NODE_PAM_JS_CONV) {
          ws.send(JSON.stringify({'msg': nodepamCtx.msg,
                                  'msgStyle': nodepamCtx.msgStyle}));
          ctx = nodepamCtx;
          if (nodepamCtx.msgStyle === msgStyle.PAM_ERROR_MSG ||
                nodepamCtx.msgStyle === msgStyle.PAM_TEXT_INFO)
            pam.setResponse(nodepamCtx, '');
        } else if (nodepamCtx.retval === PAM_SUCCESS) {
          var cookie = generateCookie(cookieName, nodepamCtx.user);
          ws.send(JSON.stringify({'msg': nodepamCtx.retval,
                                  'cookie': cookie}));
          ctx = undefined;
        } else {
          ws.send(JSON.stringify({'msg': nodepamCtx.retval}));
          ctx = undefined;
        }
      });
    } else {
      pam.setResponse(ctx, message);
    }
  });
});
```

Listing 5.4: The WebSocket Server Core

5.5 The WebSocket Client

The WebSocket client is a client-side JavaScript that runs in the browser when a user accesses the login page of the web application. It handles the client-side of the solution, which means it collects the user's input, and modifies the login page according to messages received from the server. If no session cookie for the application is set in the browser, it contacts the server to establish a connection. When the connection is open, it sets the first/initial prompt to "Username:" and displays the form. If a session cookie already exists, the client only displays the "Already authenticated" status in the `#status` element, and issues a redirect to the specified location.

```
    ws.onopen = function (e) {
        $("#promptLabel").text('Username:');
        $("#promptForm").show();
    };
```

Listing 5.5: "Open" Event of the WebSocket Client

When the user fills in his username, the client sends it to the server, which starts the authentication. Now, there are three types of messages (not PAM messages, but *JSON* strings) expected from the server distinguished by the message content:

- `msg` (string), `msgStyle` (integer)

- `msg` is *PAM_SUCCESS* (integer), `cookie` (string)
- `msg` (integer)

In the first case, the message contains `msg` and `msgStyle` fields. It means this message contains a message from a PAM module and its style. The client uses a `switch-case` statement to decide how to display the message, and if it is a "prompt" how to set up the input field. If the style is *PAM_PROMPT_ECHO_OFF*, the client sets the `type` property of the input field to *password*, so the field's content (user's response) is hidden. If the style is *PAM_PROMPT_ECHO_ON*, the client sets the `type` property of the input field to *text*, so the field's content is visible. In case of *PAM_ERROR_MSG* or *PAM_TEXT_INFO*, the client appends the message to a `div` HTML element with the `#messages` ID.

```
    switch (message.msgStyle) {
        case msgStyle.PAM_PROMPT_ECHO_OFF:
            $("#promptLabel").text(message.msg);
            $('#prompt').prop('type', 'password');
            startTimer();
            break;
        case msgStyle.PAM_PROMPT_ECHO_ON:
            $("#promptLabel").text(message.msg);
            $('#prompt').prop('type', 'text');
            startTimer();
            break;
        case msgStyle.PAM_ERROR_MSG:
        case msgStyle.PAM_TEXT_INFO:
            if (message.msgStyle === msgStyle.PAM_ERROR_MSG) {
                $("#messages").append('<p style="color:red">' +
                    message.msg + '</p>');
            } else {
                $("#messages").append('<p>' + message.msg + '</p>');
            }
            break;
        default:
            break;
    }
```

Listing 5.6: Client-side Handling of the Conversation

In the second case, the `msg` field contains *PAM_SUCCESS* and `cookie` fields. It means that authentication finished successfully and the server has created a session and stored it to the sessions file. The client closes the connection with the server, hides the form, display the "Autheticated" status, sets the cookie (session cookie) using the `cookie` property of `document` and issues a redirect to the configured page.

```
    ws.close();
    $("#promptForm").hide();
    $("#status").text('Authenticated');

    document.cookie = message.cookie;
    setTimeout( () => {
        window.location.href = '/';
    }, 3000);
```

Listing 5.7: Clients Behavior on Successful Authentication

In the final case, the `msg` field contains an error return value, so it is possible to display the corresponding error message in the `#status` element. However, it is only useful for debugging of the PAM stack, because it is meaningless to display every error to the user. The administrator/developer of the web application should verify that the configured PAM stack is functional. For that reason, the client displays only the "Wrong username or password, please try again" message, as that is the only error that PAM returns, when everything is configured correctly. That is also why the client in this case deletes the stored username and sets the prompt to the initial prompt, so the user can try to authenticate again.

```
    user = undefined;
    $("#status").text('Wrong username or password, please try again');
    $('#prompt').prop('type', 'text');
    $("#promptLabel").text('Username:');
```

Listing 5.8: Clients Behavior on Authentication Error

The Listing 5.9 shows the necessary HTML code for a login page. The Figure 5.3 shows the sequence diagram of the client-side of the solution. The entire source code of the WebSocket client is attached in the appendix C.

```
    <script type="text/javascript" src="login.js"></script>
    <form hidden id="promptForm" onsubmit="sendUserInput(); return false;">
        <label id="promptLabel" for="prompt"></label>
        <input id="prompt" type="text" />
        <button type="button" onclick="sendUserInput();">Next</button>
    </form>

    <h2 id="status"></h2>

    <div id="messages">
    </div
```

Listing 5.9: Necessary HTML Code For a Login Page

5.6 Integration to a Web Application

The integration of multi-factor authentication to a web application using the solution provided in this chapter is fairly easy. It requires the content of appendix C. It can also be downloaded from the node-auth-pam[1] repository. The `integration/` directory contains all necessary files needed for the integration of the solution to a web application. The `login.html` file contains the necessary HTML code for a login page. It can be edited to needs and taste, but included scripts, the form, and `#status` and `#messages` elements are mandatory and should not be deleted. It can also be slip to several parts, if the application uses a templating language. The `login.js` file contains the client-side JavaScript code for the login page. Only the `window.location.href` path and the WebSocket server address should be edited. An example application is provided in the `node-auth-example/` directory. It is written in Node.js using the *Express* web framework[2] and *EJS* templating language[3]. Use the following commands to run the application:

```
$ npm install
$ node server.js
```

For the server, run the following command at the root of the package:

```
$ npm install
```

If running as root, the addon must be installed using:

```
$ npm run build
```

The server (main.js) supports two command line arguments:

- port – the port to run the server on (default: 1234)
- service – the service name as defined in `/etc/pam.d/` (default: login)

Run the server on desired port using the desired PAM stack. For example:

```
$ node main.js -s webapp
```

The *webapp* stack:

```
auth        required        pam_sss.so
auth        required        pam_google_authenticator.so
account     required        pam_sss.so
```

Access the example application at `localhost:8080` and try to login with a user account.

[1]download here: node-auth-pam
[2]learn more at the Express web page
[3]learn more at the EJS web page

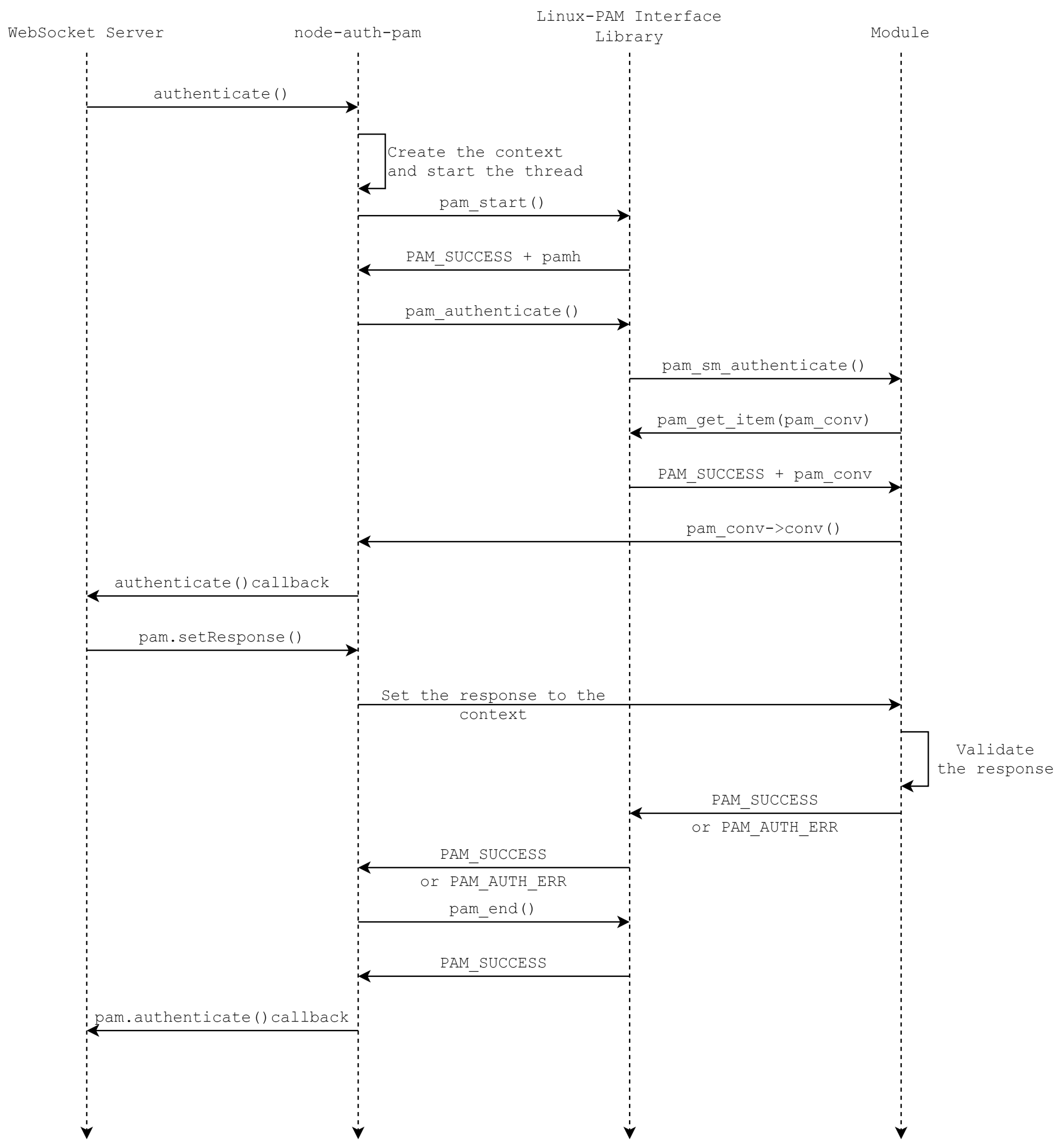


Figure 5.2: Server-side of the Solution

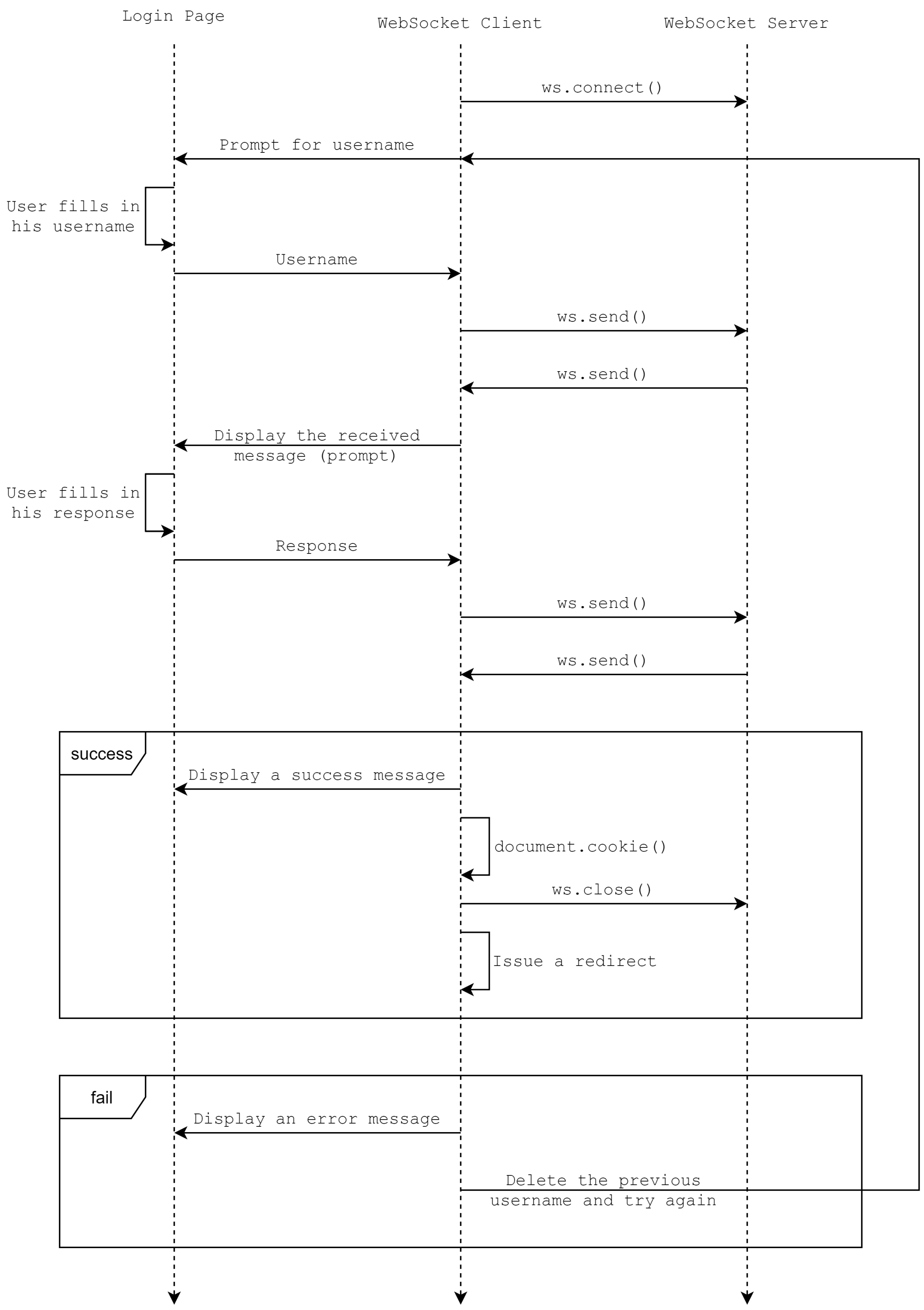


Figure 5.3: Client-side of The Solution

Chapter 6

Conclusion

This thesis introduced the Pluggable Authentication Modules framework for those who were not familiar with this technology. Application developers should be encouraged to use PAM for its ease of use and high flexibility. It further described authentication in web applications and its modern trends. Then, it introduced standard methods of authentication using HTTP and described the current state of integration of PAM and HTTP. After some demonstration, examples that every interested person should try themselves to see them work and break into the PAM and HTTP inherent incompatibility. Finally, the contribution of this thesis is a functional implementation of multi-factor authentication for Node.js using the node-auth-pam addon. It can be used by any Node.js application, but this thesis implemented an authentication server (daemon) with the use of the WebSocket protocol. It also provided a client-side JavaScript that and the necessary HTML code for simple integration of the solution to a web application.

For example, a potentially interested user could be a company or a school that runs its company/school information system that runs a server, and each person has their user account with which they can log in. Another possibility is to use the SSSD service configured to authenticate against an Active Directory or a FreeIPA server that is configured for multi-factor authentication.

The third point of the assignment also required a demonstration using FreeOTP. However, the thesis used Google Authenticator instead because it provides for much easier setup. While FreeOTP requires a running FreeIPA server, Google Authenticator provides the `pam_reversed_login` module that authenticates local users. Running a Node.js application that uses the node-auth-pam as root allows access to all user's configuration files so that it can validate OTP tokens.

6.1 Future Work

The aim of this thesis was to implement multi-factor authentication in web applications using PAM. However, there are three other module types whose support is not yet supported by the node-auth-pam addon. It should be possible to provide support to all of them. Another possible improvement is the implementation of a better waiting mechanism of the conversation function of the node-auth-pam addon.

Bibliography

[1] Morgan, A. G. and Kukuk, T. *The Linux-PAM System Administrators' Guide* [online]. www.linux-pam.org, 2010 [cit. 2020-05-28]. Available at: http://linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html.

[2] Morgan, A. G. and Kukuk, T. *The Linux-PAM Application Developers' Guide* [online]. www.linux-pam.org, 2010 [cit. 2020-05-28]. Available at: http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_ADG.html.

[3] Morgan, A. G. and Kukuk, T. *The Linux-PAM Module Writers' Guide* [online]. www.linux-pam.org, 2010 [cit. 2020-05-28]. Available at: http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_MWG.html.

[4] Geisshirt, K. *Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers.* 1st ed. Packt Publishing Ltd., 2007. ISBN 978-1-904811-32-9.

[5] Barrett, D. J., Silverman, R. E. and Byrnes, R. G. *What the heck is „keyboard interactive" authentication* [online]. www.snailbook.com, 2017 [cit. 2020-05-28]. Available at: http://www.snailbook.com/faq/keyboard-interactive.auto.html.

[6] *Chapter 10. Using Pluggable Authentication Modules* [online]. Red Hat, Inc., 2014 [cit. 2020-05-28]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system-level_authentication_guide/pluggable_authentication_modules.

[7] Soares, L. F. B., Fernandes, D. A. B., Freire, M. M. and Inácio., P. R. M. *Secure user authentication in cloud computing management interfaces* [online]. 2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC), San Diego, CA, 2013 [cit. 2020-05-28]. Available at: https://ieeexplore.ieee.org/document/6742763.

[8] *Multifactor Authentication Cheat Sheet* [online]. OWASP Foundation, 2019 [cit. 2020-05-28]. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html.

[9] *Authentication Cheat Sheet* [online]. OWASP Foundation, 2019 [cit. 2020-05-28]. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html.

[10] *Single Sign-On* [online]. Auth0, Inc., 2018 [cit. 2020-05-28]. Available at: https://auth0.com/docs/sso/current.

[11] Grassi, P. A., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Buff, W. E. et al. *Digital identity guidelines: Authentication and lifecycle management* [online]. NIST Special Publication 800-63B, june 2017 [cit. 2020-05-28]. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf.

[12] Gavin, B. *How to Use Google Chrome to Generate Secure Passwords* [online]. www.howtogeek.com, 2019 [cit. 2020-05-28]. Available at: https://www.howtogeek.com/427007/how-to-use-google-chrome-to-generate-secure-passwords/.

[13] *New password security features come to Firefox with Lockwise* [online]. Mozilla Corporation, 2019 [cit. 2020-05-28]. Available at: https://blog.mozilla.org/firefox/password-security-features/.

[14] Barth, A. and Berkeley, U. *HTTP State Management Mechanism* [online]. Internet Engineering Task Force (IETF), april 2011 [cit. 2020-05-28]. Available at: https://tools.ietf.org/html/rfc6265.

[15] *Authentication and Authorization* [online]. The Apache Software Foundation, 2018 [cit. 2020-05-28]. Available at: https://httpd.apache.org/docs/2.4/howto/auth.html.

[16] Pazdziora, J. *Typical Form-based Authentication* [online]. Adelton, 2013 [cit. 2020-05-28]. Available at: https://github.com/adelton/mod_intercept_form_submit/blob/master/docs/typical_form_based_authentication.txt.

[17] Pazdziora, J. *Apache module mod_authnz_pam* [online]. Adelton, 2013 [cit. 2020-05-28]. Available at: https://www.adelton.com/apache/mod_authnz_pam/.

[18] Pazdziora, J. *Mod_intercept_form_submit* [online]. Adelton, 2013 [cit. 2020-05-28]. Available at: https://www.adelton.com/apache/mod_intercept_form_submit/.

[19] Fette, I. and Melnikov, A. *The WebSocket Protocol* [online]. Internet Engineering Task Force (IETF), december 2011 [cit. 2020-05-28]. Available at: https://tools.ietf.org/html/rfc6455.

[20] Bojinov, V., Herron, D. and Resende, D. *Node.js Complete Reference Guide*. 1st ed. Packt Publishing Limited, 2018. ISBN 9781789952117.

[21] Chaniotis, I. K., Kyriakou, K.-I. D. and Tselikas, N. D. Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing*. october 2015, vol. 97, no. 10, p. 1023–1044. Available at: https://doi.org/10.1007/s00607-014-0394-9.

[22] *Node.js v14.2.0 Documentation* [online]. OpenJS Foundation, 2020 [cit. 2020-05-28]. Available at: https://nodejs.org/dist/latest-v14.x/docs/api/addons.html.

[23] Barrett, D. J. and Silverman, R. E. *SSH, The Secure Shell: The Definitive Guide*. 1st ed. O'Reilly, 2001. ISBN 0-596-00011-1.

[24] Miller, D. and Tucker, D. *auth-pam.c* [online]. OpenSSH, 2003 [cit. 2020-05-28]. Available at: https://github.com/openssh/openssh-portable/blob/master/auth-pam.c.

[25] *Asynchronous Thread-safe Function Calls* [online]. OpenJS Foundation, 2020 [cit. 2020-05-28]. Available at: https://nodejs.org/api/n-api.html#n_api_asynchronous_thread_safe_function_calls.

[26] SCHULHOF, G. and MISSINE, A. *round_trip.c* [online]. gabrielschulhof, 2018 [cit. 2020-05-28]. Available at: https://github.com/gabrielschulhof/abi-stable-node-addon-examples/blob/tsfn_round_trip/thread_safe_function_round_trip/node-api/round_trip.c.

Appendix A

How to setup SSSD

1. Install the sssd service:

   ```
   $ dnf install sssd -y
   ```

2. Create the sssd.conf config file in the /etc/sssd directory with following contents:

   ```
   [sssd]
    domains = PROXY_PROXY
    services = nss,pam

   [domain/PROXY_PROXY]
    id_provider = proxy
    proxy_lib_name = files
    proxy_pam_target = sssd-shadowutils
    pwfield = x
   ```

   The `pwfield = x` is a bug in the sssd-2.2.3-13.fc31.x86_64 package.

3. Restart the sssd service:

   ```
   $ systemctl restart sssd
   ```

Appendix B

How to set up Google Authenticator

1. Install the pam_google_authenticator module:

   ```
   $ dnf install pam_google_authenticator -y
   ```

2. Run the google-authenticator command and follow the configuration guide:

   ```
   $ google-authenticator
   Do you want authentication tokens to be time-based (y/n) y
   ```

   – Scan the generated QR code with the Google Authenticator mobile application and enter the code it generates

   ```
   Do you want me to update your
   "/home/mariankapisinsky/.google\_authenticator" file? (y/n) y
   ```

   – Other configuration settings are optional

3. Use OTP tokens generated by the mobile application for future authentication

Appendix C

CD Content

- `integration/` - contains necessary files for a simple integration to a web application
- `node-auth-example/` - contains an example web application for node-auth-pam
- `pam_reversed_login/` - contains an example PAM module for demonstration/test purposes
- `src/` - contains node-auth-pam addon source files
- binding.gyp - the binding file that describes the configuration to build the node-auth-pam addon
- main.js - the WebSocket server for PAM authentication using node-auth-pam
- package.json
- LICENSE
- README.md