CHALMERS
UNIVERSITY OF TECHNOLOGY

Chalmers University of Technology

Exploit Vulnerabilities in UNIX-based Authentication Systems

TDA602: Language-based Security

Author: Pedro Gonçalo Mendes

Group 32

Chalmers University of Technology

Gothenburg, Sweden, 2018

1. Table of Contents

2. Introduction

In the technological world that we face nowadays, the security of computer systems and networks represents a high priority concern. It has a critical importance that a system can correctly recognize a user that tries to log in into it in order to secure the entire system and also the critical resources (data).

This report analyses the authentication process in systems based in UNIX by studying certain vulnerabilities and threats that such systems present nowadays and policies and measures that can be followed to mitigate the impact of these vulnerabilities in order to increase the security level and thereby ensures a safer system for all users.

Authentication is one of the most fundamental processes in every system and it's the primary line of defense. It ensures the identity and authenticity of a person/user. By using adequate authentication protocols, the system's designer and the system's administrator can create a policy to ensure that a user only has access to his files and contents.

However authentication is an important and fundamental mechanism, it also has some vulnerabilities. These vulnerabilities can be exploited by malicious users and attackers to break authentication in order to attack the system.

Therefore, when implementing an authentication system a detailed study of such threat and vulnerabilities is needed and fundamental in order to find different ways to fight against these to create a secure system for users.

This project is divided into five different parts. Firstly, it will focus on authentication process and secondly, in authentication in UNIX-based system where it will be done a deep and detailed study about this subject. Then, this report will approach the vulnerabilities and the impact of these in such systems and after that, it will study measures against these vulnerabilities. In the end, a UNIX-based authentication system will be implemented in C programming language where it will be tested and implemented all the knowledge gained with this project. After that, it will be possible to discuss all the results and draw conclusions.

3. Authentication

Authentication is the process of verifying a claim that a system entity or resource has a certain attribute [1]. In other words, authentication is the act of confirming the identity claimed by or for a system entity.

The authentication process consists in two steps:

- Identification step: Presenting the claimed attribute value (identifier) to the authentication system (e.g. username);
- Verification step: Presenting or generating authentication information (e.g., password) that corroborates the binding between the entity and the identifier [2].

Normally, computer's systems need this security level, where the user has to be identified by the system and then he has to give any type of attribute that is recognized by the system. Identification is the means by which a user provides a claimed identity to the system (the identifier tells "who the user is"). Authentication is the means of establishing the validity of the claim (the authenticator verifies "that is true") [2].

The authentication procedure consists in four stages. Firstly, the user must be identified and after that, he must provide some kind of authentication information, which is secret and unforgeable. All this authentication data has to be transmitted to the system through a secure channel and finally, this data has to be validated with respect to some reference information (proof of correctness).

The authentication information can be of four different generic types, based on factors of authentication that are unique to the user [3]. These authentication factors are:

- Something the user knows (e.g., a password, a personal identification number (PIN), or answers to a prearranged set of questions).
- Something the user has (possesses) (e.g., electronic keycards, smart cards, keys). This type is referred to as a token.
- Something the user is (e.g., recognition by fingerprint, face and retina, or by DNA).

- Something the user does (biometrical methods, something characteristic about the user) (e.g., recognition by voice pattern, handwriting characteristics).

These types of authentication information can be combined in order to increase the level of security in a system.

In this report, I will focus on password-based authentication (something the user knows), which is used in computer systems and networks and also in UNIX systems.

4. Authentication in UNIX-based Systems

In a UNIX system, a user is authenticated through the username (login name) followed by a password. Then, the system looks up the username in its password file. The password file is usually called /etc/passwd. The password file is used every time a user logs in into the system. This file is composed by seven different fields:

- Username;
- Encrypted password;
- User ID (UID);
- Group ID (GID);
- Comment field;
- Home directory;
- Shell program.

However, all the modern systems have moved the encrypted passwords to another file, called shadow password file.

The UNIX file system is a hierarchical arrangement of directories and file. Everything starts in the directory called root [4]. Usually, there is an entry with the username root, where the UID value is 0. The root is the super-user used for system administration with special and unique privileges that any other user has, capable of doing unrestricted changes in the system. Every time the system is initialized, the first user is always the root.

The User ID (UID) is a unique number that identifies a user's account. It's used by the system to determine which systems resources and files can be accessed by a user.

In Figure 1, it's represented the /etc/passwd file, where it is possible to identify the different accounts (and the root) and the different fields that composed each entry.

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Figure 1: /etc/passwd file

The older versions of UNIX store the encrypted password in this file. In the modern systems, the encrypted password field contains a single character as a placeholder. This different implementation occurs due to security reasons (it is a security failure store encrypted password in a file that is readable by all the users) [4] [5]. If the password field is empty that means that the user doesn't have a password.

Once the authentication process finishes with a valid authentication, a user logs in into the system. After that, a shell program starts running. The name of this executable program that is used as the login shell for the user is stored in shell field in the password file.

In order to increase the level of security, the passwords are encrypted with a hash function. This hash function is a "one-way "encryption algorithm, i.e., the algorithm encrypts the password, but the encrypted password cannot be decrypted to the original (it's impossible to revert the encryption and return to the plaintext password). So, if a malicious attack has access to the password file, it doesn't know the original passwords. In figure 2, it's described how a new password is encrypted and stored in the password file.

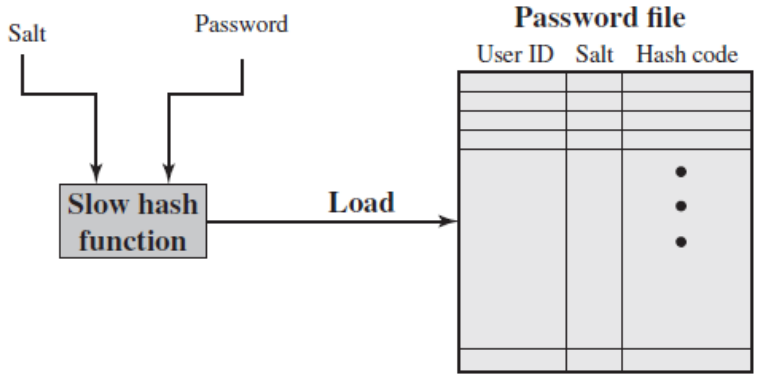

Figure 2: Loading a new password (source: Computer Security: Principles and Practice [2])

As we saw before, to increase the level of security and to make it more difficult to obtain the file with passwords, newest systems store the encrypted password in the shadow password file. This file contains the username and the encrypted password and other some relevant information and shouldn't be readable by all the users. Only the root user and some specific programs (that can run in privileged mode) need and have access to this shadow file.

In summary, when the system starts, it logs in as the root user. Then, the user writes the username and the password. The root looks up the username in /etc/passwd (or in shadow password file in modern UNIX system). If the username was found that means that the user is valid and the root now has to compare the written password by the user with the password stored in the file. For that, it encrypts the written password with a hash function. Then compares the result of this encryption with the encrypted password stored in the file. If these two encrypted passwords are the same, the password written by the user is correct and the user can log in into the system. The value of the UID change from 0 (root) to the user ID (using the function setuid()) and then it runs a shell program. In Figure 3, it is described this process of verification of a password.

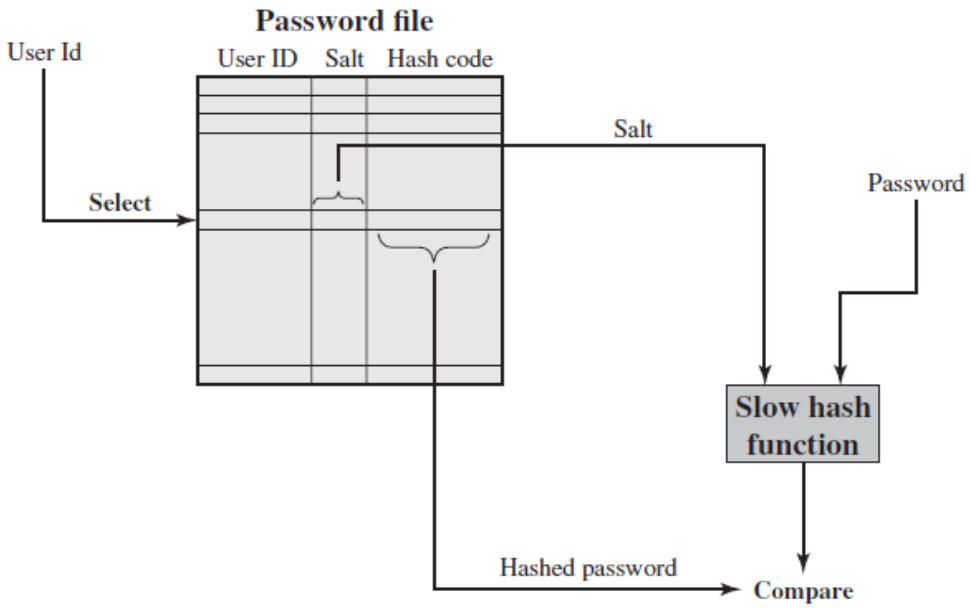

Figure 3: Process of verification of a password (source: Computer Security: Principles and Practice [2])

5. Vulnerabilities in Authentication in UNIX-based Systems

Authentication and session management includes a wide array of aspects of handling user authentication and managing sessions [5]. As we saw before, authentication is a critical aspect of this process. However all authentication mechanisms can be broken and/or undermined, i.e., all these mechanisms present vulnerabilities that can be exploited by malicious people to attack the system. Any account and session management flaw can result in the compromise of user or system administration accounts [5]. All applications, environments and systems that implement authentication are susceptible to have broken authentication and sessions management issues.

The biggest problem in authentication in UNIX-based systems is broken authentication. This occurs when an attacker that shouldn't have access into the system gains access to only one or few user's accounts or to the administrator account in order to compromise the system (e.g. delete, steal, encrypt data stored in the system).

Broken authentication becomes easier when users have weak passwords. An attacker in possession of a valid username and the respective password would have complete access to any resources available to that user and would be significantly closer to being able to access other accounts and perhaps also obtain root/admin level access on the system. The most frequent vulnerabilities related to passwords are:

- Users accounts that have weak or non-existent passwords;
- Users accounts with widely known or openly displayed passwords;
- Weak encryption algorithms;
- Passwords stored in a file with a low level of security and readable to anyone;

The best defense against these vulnerabilities is the implementation of a strong authentication policy in any system. In the next session, we will discuss measures that can be taken to protect the system against these vulnerabilities in order to increase the security level.

A user should always have a password (the system should forbid accounts with no passwords). If there is a user without password any attacker can enter directly into the system. Once logs in into the system, the attacker has more chance to attack the entire system, including the other user accounts and the administrator.

Another problem with passwords is users normally tend to choose words or phrases, like names or something that can easily be related with the user or special patterns (e.g. qwerty, 123456), or numbers that also can be related to the user (e.g. date of birth, phone number).

Attackers have access to a huge number of valid usernames and passwords combinations for credential stuffing, default administrative account list, automated brute force and dictionary attack tools [6].

Given enough hardware resources and time, any password can be cracked using brute force attack. A brute force attack consists of an attacker trying many passwords or passphrases with the hope of eventually guessing the correct password. Also, an attacker can combine this with a dictionary attack. Here, instead of trying all the possible combinations, the attacker tries many passwords and passphrases that are more likely to be chosen such as words in a dictionary. Therefore, a password that in any way resemble a word or some different words (or a phrase) is highly susceptible to a dictionary attack, and on that way is easier for an attacker break the authentication of a system.

A system is more vulnerable if it permits automated attacks such as credential stuffing, permits brute force and related attacks, permits default, weak and well-known passwords, uses weak or ineffective credential recovery and forgot passwords processes and uses plaintext or weak encryption algorithms.

Another important threat against authentication process is exploiting buffer overflows vulnerabilities. Despite operating systems (including UNIX systems) are protected against buffer overflow and so it's possible to identify when it occurs, an attacker can exploit some weakness of the system, related with buffer overflows, in order to crash the entire system and create a Denial of Service (DoS).

Storing the usernames and the respective passwords in one file make this file the main target for an attacker.

Other threat against UNIX systems is related with the fact that a user can gain access on a machine using a guest account. If an attacker logs in using a guest account then, for example, he can run a password guessing program (password cracker) on that machine and on that way he should be able to check a huge number of different possible and probable passwords.

6. Measures against Vulnerabilities in Authentication in UNIX-based Systems

The authentication process entails a high number of issues and different aspect which should be considered in the system's implementation in order to reduce significantly the likelihood of a problem in this process.

The best defense against the previous vulnerabilities in order to have a secure and a robust authentication and session management mechanism is a strong authentication policy.

This policy should ensure that passwords are consistently strong. Some of these critical areas include:

- Password Strength

The password should have some restrictions that require a minimum size and complexity for the password. The password must contain alphanumeric and special characters to increase the complexity.

- Password Use

The number of login attempts per unit of time should be well defined as well as the number of repeated failed login attempts in order to protect against brute force attacks. Passwords provided during failed attempts shouldn't be recorded, as this may expose a user's password. The account must be disabled for a determined period of time sufficient to discourage and prevent brute force attacks but not so long as to allow for a Denial of service.

The system shouldn't indicate whether it was the username or the password that was wrong. If the user request information about the last successful login, he should be informed of the date and time of that and also the number of failed login attempts to his account since that time in order to verify that his account and password are safe.

- Password Aging

The password aging is technique related with the lifetime for the user passwords and it's used by the system administrators to defend against bad passwords. After a specific

period, the user has to change the old password. If a password has a short lifetime it is reused and if a password has a long lifetime the valid passwords can be exposed to others. Passwords should have some minimum and maximum age.

- Password change controls

The system should implement a single mechanism to allow users change a password. However, it's required to provide both their old and new password when changing a password. If the password has been forgotten other authentication methods (instead of username and password) should be implemented, e.g. using a token.

- Password Storage

All passwords must be encrypted by a hash algorithm to avoid exposure.

The password should preferably be stored in a shadow file and only the root and specific programs with root privileges can access to this file

In summary, detailed instructions to create a password should be implemented in a system with specific rules discuss above to ensure that a password created is not weak. The system should have mechanisms to assure that these instructions and rules are followed by the users.

Also, the system administrator should do regularly a proactive checking of password integrity. Users whose password are cracked, so it is not secure anymore, should be notified confidentially. The user has to be obliged to change the password and if the user does not respond to this notification, the system should handle the situation automatically in order to ensure that the other users' accounts are not compromised as well as the security of the system.

With the development of computer and new technologies, other types of authentication can be implemented in UNIX-based systems, like using fingerprints or smart cards. Despite this increase the level of security of a system, using these technologies causes also an increase on the price of the system. So it must be a trade-off between the security level that a certain system needs (related, for example, how sensitive and/or personal is the information stored in the system) and the cost of that system.

Normally, most of the functionalities in UNIX systems are implemented in C language. Operating systems have to be protected against buffer overflows. However, even if a system/application is protected against buffer overflows, there is still a probability of causing a Denial of Service exploiting a buffer overflow vulnerability. For example, when the system is reading the username and/or the password, it doesn't know the dimension of these strings. Due to the computer and memory resources are limited, the maximum size of a string has to be defined. Here, the programmer has to be very careful about how and which function(s) should use to read a string from the standard input (normally the keyboard). In C programming language there are some functions where buffer overflow vulnerabilities can be exploited to cause a denial of service. The authentication methods have to be protected against this threats.

The system should also ignore signals that can cause the termination of the program (i.e. it should ignore ctrl+C and ctrl+Z signals).

7. Implementation of Authentication System based in UNIX

After this study about authentication process, authentication in UNIX system, vulnerabilities in such system and some measures against these vulnerabilities, I implemented a system based in UNIX authentication systems. This system was implemented in C language.

7.1. Description of the system

All the information related with the username, password and account is stored in a file called passwd. Each entry of this file contains information about the user. The first parameter is the username, then encrypted password, UID, Salt, number of consecutive failed login attempts, password age (in number of successful logins), group UID (GUID), comment field (stores the username), home directory and the shell program for each user.

```
user1:FPmadrB54AAfg:1000:FP:0:0:1000:USERname1:/home/user1:/bin/sh
user2:.LbICO3SUn3i2:1000:.L:0:0:1000:USERname2:/home/user2:/bin/sh
user3:A2C6BBhL6Cb0I:1000:A2:0:0:1000:USERname3:/home/user3:/bin/sh
user4:oPoArWLCFCBzA:1000:oP:0:0:1000:USERname4:/home/user4:/bin/sh
user5:34eVd5mCTWIA6:1000:34:0:0:1000:USERname5:/home/user5:/bin/sh
```

Figure 4: passwd file

When the system is initialized, first it requests the username and then the password. Firstly, the username is verified, i.e., the username written is compared to each entry in the file. If username exists, the password written by the user is encrypted (with an hash function and using the salt (in this implementation the hash algorithm used was the C function *crypt,* and depending salt's value it selects which algorithm is used for encryption, one based on MD5 or other on DES algorithms)) and compared with that encrypted pass. If the encrypted passwords are equal, the user logs in into the system.

If one of these two fields are incorrect, the user receives a message saying "Login incorrect: The username and/or the password are incorrect. Try again". The system does not indicate if it was the username or the password that was wrong.

If the username is correct but the password is incorrect, the number of failed attempts is incremented. The user has three attempts to write the correct password. If after this three first attempts the password written is wrong again, the user account is blocked during a specific period of time. During this period of time, the account is blocked and the user cannot access his account. This implementation has the objective to prevent brute force and dictionary attacks.

After this period of time, the account is unblocked and the user can try again. He has three more attempts. If after this three attempts the user does not write the correct password the account is blocked forever and can only be unblocked by the system's administrator. All this information related to the state of an account is also stored in the passwd file. This file is always updated when a user with a correct username tries to login into the system.

Always that a user logins into the system, the password age filed is incremented. When it reaches ten successful logins, a new password has to be created by the user. Also if the first three password attempts are incorrect, then when the user successfully logins into the system, it requires the creation of a new password.

7.2. Concerns and Results

During the implementation of the system, several problems and concerns emerged related with the vulnerabilities and the respective measures against these vulnerabilities studied in section 6.

Regarding with password strength, when the password has to be exchanged, due to age or several incorrect password attempts, the system has some restrictions on the characters that can be used in the password. The user is obligated to have a password. The system only allows alphanumeric and some special characters (characters with the decimal ASCII value between 33 and 126). Also, the new password must have at least 5 characters.

The number of consecutive failed login attempts is also stored in the passwd file. The system only permits three consecutive failed attempts and then it blocks for a period of

time. After that time, the user has three more attempts until the account is blocked forever. This mechanism prevents brute force attacks.

Regarding password aging, the password age, measured in the number of successful logins, is also stored in the passwd file. This prevents the system against old passwords that can be stolen and in that way increases the security level of the system.

To prevent against buffer overflow attacks that could create a denial of service when the system reads an input value (username) into a buffer, the system uses the C function *fgets*. This function guarantees that the system only reads the string of the specific size, ignoring the rest if the buffer does not have enough allocated space. There are other small details that help to prevent against buffer overflows attacks that are described and comment in the code present in the appendix.

```
//handle with bufferoverflows
if (fgets(user_check, LENGTH, stdin) == NULL){
  //error message
  printf("An error occurred in the system. Please initialize the system again\n");
  return -1;
}

sscanf(user_check, "%s", user);
```

Figure 5: Handle with buffer overflow when the user writes the username

The system ignores the signals that can cause the termination of the program. In this implementation, it ignores the SIGINT (signal interruption (ctrl+C)) and SIGTSPT (signal stop (ctrl+Z)) signals. So, it is impossible to kill the execution creating signals, because the system ignores these.

8. Discussion

Authentication is fundamental in almost every systems that exist today. It's the way that a person/user can be recognized and authenticated by a system. However, authentication is a complex process that presents some vulnerabilities that can be exploited by malicious users/attackers to broken authentication in a system and login without having the necessary permission to do it. Every system has vulnerabilities and so also UNIX systems have vulnerabilities in authentication process.

However, there are some ways to mitigate the system's vulnerabilities. A strong authentication and restricted policy should be implemented to avoid the weakness of these systems.

There must be a trade-off between the information that a user should access and received and the information that should keep unknown for a user. A user should receive some feedback about the state of his account, for example, if the account is blocked. However, through these feedback messages, the attacker gains information about the system and about valid usernames.

The system only should permits "strong" passwords (with alphanumeric and special characters and without words and sentences). However, the user will have more difficulty to remember the password. In some cases, the user can write the password in other file or in some note to be easy to remember it, but this increases the probability and facilitates theft of passwords.

When a system is being implemented, the system's designers should pay attention to these details. Sometimes increasing the level of security and the system's complexity with some mechanisms causes the decrease of security in other mechanisms.

9. Conclusion

This report had the main purpose to study measures against vulnerabilities in UNIX-based authentication systems. After doing all this analysis and study, it's possible to come up with some conclusions regarding this main subject.

Firstly, it's possible to realize how important and fundamental the authentication's mechanisms are in every system. However these mechanisms have some vulnerabilities that can be exploited by malicious users to attack the system in different ways. There are some ways to attenuate these threats, for example creating a strong authentication policy that requires "strong" passwords. However, there must always be a relation and a trade-off between the security policy and the information that a user can have about that.

The system's designers and administrators are responsible for all this mechanism to ensure that a system is secure and so they should first study all the problems and concerns related with this and only after that implement the system according to their study.

In summary, an authentication system that responds to all possible vulnerabilities and guarantees a secure system on that point is fundamental and it's the primary line of defense against attackers. Therefore it should be a high priority concern in any system's implementation that requires such authentication mechanisms.

10. References

[1] Shirey, R., “Internet Security Glossary, Version 2,” Network Working Group, Request for Comments: 4949 (RFC4949), August 2007.

[2] Stallings, W., Brown L., Computer Security: Principles and Practice, Pearson, Third Edition, 2015.

[3] “https://en.wikipedia.org/wiki/Authentication,” [April 2018].

[4] Stevens, W. R., Rago, S. A., Advanced Programming in the UNIX Environment, Pearson, Third Edition, 2013.

[5] “https://www.owasp.org/index.php/Broken_Authentication_and_Session_Management,” [April 2018].

[6] “https://www.owasp.org/index.php/Top_10-2017_A2-Broken_Authentication,” [April 2018].

[7] “https://en.wikipedia.org/wiki/Brute-force_attack,” [April 2018].

[8] “http://www.softpanorama.org/Commercial_linuxes/Security/top_vulnerabilities.shtml,” [April 2018].

APPENDIX

```
 READ ME
---- INSTRUCTION TO SUCCESSFULLY COMPILE AND RUN THE PROGRAM ----

In order to run the program first, you need to confirm the UID and GID of your
UNIX system
For that open a bash and run the command id.

example:
run:      id
(output)  uid=1000(pedro) gid=1000(pedro) groups=1000(pedro),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)

In this example, the UID and the GID are 1000.
Now open the passwd file, and in the third and seventh field write the uid and gid
that you found (running the command id)

Each entry of the passwd file contains the following fields:

username:encrypted_password:UID:SALT:No_failed_attemps:Password_age:GID:
comments:home_directory:shellcode

example:
user1:FPmadrB54AAfg:1000:FP:0:0:1000:USERname1:/home/user1:/bin/sh


The valid usernames and the respective correct passwords are
  -username   :   password  (with the dot)
  - user1     :   user1.
  - user2     :   user2.
  - user3     :   user3.
  - user4     :   user4.
  - user5     :   user5.


To compile the program open the terminal in the same directory where the file are
save and run the command make.
After that run the command ./project and the execution will start.


The system will start and then you can write the username.
After that you can entry the password.
If the username and the password are correct a new shell will start.

--------------------------------------------------------------------------------
```

```
/*****************************************************************************
*
*                       Language-based Security
*
*   Project: Exploit Vulnerabilities in in UNIX-based Authentication Systems
*
*   Author: Pedro Gonçalo Bravo Mendes - pedrogo@student.chalmers.se
*           Group 32
*
*
*****************************************************************************/

#ifndef PROJECT_H
#define PROJECT_H

//INCLUDES
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <stdio_ext.h>
#include <string.h>
#include <signal.h>
#include <pwd.h>
#include <sys/types.h>
#include <crypt.h>
#include <pthread.h>

//DEFINES
#define LENGTH 64

#define LINE_BUFFER_SIZE 1000
#define FILENAME "passwd"
#define FILENAME_AUX "passwdaux"

#define TEMP_BLOCKED 3
#define BLOCK_FOREVER 7

//password struct
typedef struct _passwdInfo{
        char *username;         // Username
  char *password;         // Password
        int uid;                //User id
        char *PasswordSalt;     // Make dictionary attack harder
        int PasswordFailed;     // Number of failed attempts
        int PasswordAge;        // Age of password in number of logins
        int guid;                                                       // Group
uid
        char *UIDinfo;                          // UID info (eg: full name)
        char *home_dir;                         // User home directory
        char *login_shell;              // Login shell
}PasswdInfo;

//functions
int InitializeSystem(PasswdInfo **password_data);
char *RequestPassword(PasswdInfo *passwd);
int HandlePassword(PasswdInfo *passwd ,char* encrypt_pass);
PasswdInfo *VerifyUsername (char *username);
int UpdatePassInfo(char *username, PasswdInfo *passUp);
int BlockAccount(PasswdInfo *passwd);
void *UserBlock(void *name);
void*VerifyBlocked ();
char *NewPassword(PasswdInfo *passwd);

#endif
```

```
/*****************************************************************************
*
*                        Language-based Security
*
*   Project: Exploit Vulnerabilities in in UNIX-based Authentication Systems
*
*   Author: Pedro Gonçalo Bravo Mendes - pedrogo@student.chalmers.se
*           Group 32
*
*
*****************************************************************************/

//INCLUDES
#include "project.h"

//ALARMS
void sighandler() {
        //sighandler to ignore keyboard interruptions
        signal(SIGINT, SIG_IGN);                //ignore crtl+C
        signal(SIGTSTP, SIG_IGN);               //ignore crtl+Z
}


//main
int main(int argc, char const *argv[]) {

// Variables
  PasswdInfo *password_data =  NULL;

  int valid = 0, err;
  char *passwd;

  sighandler();

  //Initializing the system
  printf("The system is initializing\n");

  while (1) {

    valid = InitializeSystem(&password_data);

    if (valid == 0){
      /*username is incorrect. however the user doesn't be warm that the username
      is incorrect.
      user should write the username and then the password and only after that
      the system must print an error alerting that username and/or password are
      incorrect*/

      passwd = RequestPassword(password_data);
      printf("Login incorrect: The username and/or the password are incorrect\nTry
again\n");


    }else if (valid == 1){
      //username exists in the system

      if(password_data->PasswordFailed == BLOCK_FOREVER){
        /*the user account is blocked forever, because he wrotes the wrong
        password to many times*/
        printf("Your account is blocked. You have to contact the administrator.
\n");

      }else if(password_data->PasswordFailed == TEMP_BLOCKED){
        /*the user wrote the wrong password 3 times, so the account is tempo-
        rarily blocked. The user has to wait until be possible access again
        to his account*/
```

```
        printf("Your account is temporarily blocked. You have to wait\n");


      }else{
        passwd = RequestPassword(password_data);
        err =  HandlePassword(password_data , passwd);
        if (err == -1){
           printf("ALERT: ERROR!!!\nUpdating the file (UpdatePassInfo)\n");
        }

      }

    }

  }
  return 0;
}
```

```
/*****************************************************************************
*
*                        Language-based Security
*
*   Project: Exploit Vulnerabilities in in UNIX-based Authentication Systems
*
*   Author: Pedro Gonçalo Bravo Mendes - pedrogo@student.chalmers.se
*           Group 32
*
*   PASSWORD.C file
*
*   The functions are explained below
*****************************************************************************/


//INCLUDES
#include "project.h"

//GLOBAL VARIABLES
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;


/*****************************************************************************

      InitializeSystem(PasswdInfo **password_data)

      goal: function where the user writes the username and the system verify
      if the username is valid or not

      arguments: struct password_data

      return value: returns a int value:
                        - 1 in case of success (username exists)
                        - 0 in case of insuccess (username doesn't exist)
                        - -1 in case of error

*****************************************************************************/

int InitializeSystem(PasswdInfo **password_data){

  char user[LENGTH];
  char user_check[LENGTH];

  printf("\n\tLogin:\t");

  //handle with bufferoverflows
  if (fgets(user_check, LENGTH, stdin) == NULL){
    //error message
    printf("An error occurred in the system. Please initialize the system again
\n");
    return -1;
  }

  sscanf(user_check, "%s", user);

  *password_data = VerifyUsername(user);
  if(*password_data == NULL){
      //username doesn't exist
      return 0;
  }

  return 1;
}

/*****************************************************************************/
```

```
/*************************************************************************

      *RequestPassword(PasswdInfo *passwd)

      goal: function where the user writes the password and it's encrypted with
      an hash function (using a salt)

      arguments: struct passwd

      return value:
                  - the encrypted password (written by the user)
                  - NULL in case of error

*************************************************************************/

char *RequestPassword(PasswdInfo *passwd){

  char string_bash[] = "\n\tpassword: ";
  char salt[2];
  char *encrypt_pass;
  char *password;


  /*this function waits for the input (user writes the password)
  the charecters are hidden in the terminal */
  password = getpass(string_bash);

  /*            encryption of the password
    the encryption is done using a hash function (crypt) for security reasons
  so it's possible do encrypt the password then to be compared with the
  password in the file, but is not possible to decrypt a password

    in crypt function a salt is used (salt is a two-character string chosen
  from the set [a-zA-Z0-9./].  This string is used to perturb the algorithm
  in  one of 4096 different ways.

    the strncpy is used to copy the salt from the file and to prevent a buffer
  overflow attack, that could exploit this weakness

  */
  if (passwd != NULL){
    strncpy(salt, passwd->PasswordSalt, 2);
    encrypt_pass = crypt(password, salt);
    return encrypt_pass;
  }
  return NULL;
}

/*************************************************************************/



/*************************************************************************

      *NewPassword(PasswdInfo *passwd)

      goal: function where the user writes the new password.
      the function verifies if the all the caracters are acceptable and
      if is not a empty string (no password).
      If the password does not fulfill the requeriments the user has to write
      a new one.
```

```
      arguments: struct passwd

      return value:
            - the new encrypted password
            - NULL in case of error

*************************************************************************/

char *NewPassword(PasswdInfo *passwd){


  char string_bash[] = "\n\tpassword: ";
  char salt[2];
  char *encrypt_pass;
  char *password;
  int valid = 0, i;

  password = getpass(string_bash);

  while(valid != 1){

    if(strlen(password) < 6){
        valid = -1;
    }

    for(i=0; i<strlen(password); i++){
      if(password[i] < 33 || password[i] > 126)
        valid = -1;
    }

    if(strcmp(password, "") == 0 )
      valid = -1;

    if(valid != -1){
      valid = 1;
    }else{
      password = "";
      valid = 0;
      printf("Password not permited.\n");
      password = getpass(string_bash);
    }
  }


  if (passwd != NULL){
    strncpy(salt, passwd->PasswordSalt, 2);
    encrypt_pass = crypt(password, salt);

    return encrypt_pass;
  }
  return NULL;
}

/*************************************************************************/



/*************************************************************************

       HandlePassword(PasswdInfo *passwd ,char* encrypt_pass)

       goal: verifies if the password id correct.
       if yes, a new shell is initialize
       if no, handles the account depending of specific aspects
```

```
       arguments: struct passwd and the encrypted password written by the user
       (in the moment of login)

       return value:
              - if the password is correct does not return anything and starts
          a new shell.
              - 0 if the password is wrong
              - -1 in case of error

*********************************************************************/

int HandlePassword(PasswdInfo *passwd ,char* encrypt_pass){

  int err=0;

  if (strcmp(encrypt_pass, passwd->password) == 0){
    //right password

    passwd->PasswordAge++;

    if(passwd->PasswordAge > 10){
      //password is too old
      printf("You have to change the password (your password is too old).\n");
                        printf("Entry the new password:\n");

      encrypt_pass = NewPassword(passwd);

      while( (strcmp(encrypt_pass, passwd->password) == 0)) {
        printf("The new password has to be different from the old one.\nEntry the
new password:\n");
        encrypt_pass = NewPassword(passwd);
      }

      passwd->password = encrypt_pass;
      passwd->PasswordAge = 1;

    }else if ( passwd->PasswordFailed > TEMP_BLOCKED ) {
      //user failed the first 3 tries so when it login again has to change password
      printf("For security reasons you have to change your password.\nYour account
probabily was been compromised\n");
      printf("Entry the new password:\n");

      encrypt_pass = NewPassword(passwd);
      while(strcmp(encrypt_pass, passwd->password) == 0 ){
        printf("The new password has to be different from the old one.\nEntry the
new password:\n");
        encrypt_pass = NewPassword(passwd);
      }
      passwd->password = encrypt_pass;
      passwd->PasswordAge = 1;
    }
    passwd->PasswordFailed = 0;

    //updates the passwd file
   err = UpdatePassInfo(passwd->username, passwd);
    if(err == -1){
      printf("ALERT: ERROR!!!\nUpdating the file (UpdatePassInfo)\n");
      return -1;
    }

    // check UID
    if(setuid(passwd->uid) != 0){
      printf("ALERT: ERROR!!!\nThe root cannot login into your acount (setuid)\n");
      return -1;
    }else{
      //login into the account - all requeriments are OK
```

```
      printf("Welcome to your system user %s!\n", passwd->username);

      VerifyBlocked();

      //starts a new shell
      char *new[] = {NULL};
      new[0] = passwd->login_shell;
      new[1] = NULL;

      if( execve(passwd->login_shell, new, NULL) == -1){
        printf("ALERT: ERROR!\nOpening the new shell (execve)\n");
        return -1;
      }
    }

  }else{
    //wrong password
    passwd->PasswordFailed++;

    if (passwd->PasswordFailed == BLOCK_FOREVER){
      /*the user account is blocked forever, because he wrotes the wrong
      password to many times*/
      printf("Login incorrect:\nYour account is blocked. You have to contact the
administrator.\n");

    }else if (passwd->PasswordFailed == TEMP_BLOCKED){
      /*the user wrote the wrong password 3 times, so the account is tempo-
      rarily blocked. The user has to wait until be possible access again
      to his account*/
      printf("Login incorrect:\nYour account is temporarily blocked. You have to
wait\n");

      err = BlockAccount(passwd);
      if (err == -1){
        return -1;
      }

    }else if ( (passwd->PasswordFailed > TEMP_BLOCKED) || (passwd->PasswordFailed
< TEMP_BLOCKED) ){
      /*user has the last chances to write the correct password until it blocks
      the account or   user wrote an incorrect password. he has to try again*/
      printf("Login incorrect:The username and/or the password are incorrect\nTry
again\n");

    }

    err = UpdatePassInfo(passwd->username, passwd);
    if(err == -1){
      printf("ALERT: ERROR!!!\nUpdating the file (UpdatePassInfo)\n");
      return -1;
    }

  }
  return 0;
}

/*************************************************************************/



/*************************************************************************

      BlockAccount(PasswdInfo *passwd)

      goal: creates a threat to "set an alarm" and block the account during
```

```
      a specific time

      arguments: struct passwd

      return value:
            - 0 if successed
            - -1 in case of error

*************************************************************************/

int BlockAccount(PasswdInfo *passwd){

  pthread_t t_block;

  int err = pthread_create(&t_block, NULL, UserBlock, (void*)(passwd->username) );
  if(err != 0) {
    printf("ALERT: ERROR!!!\nError in thread creation\n");
    return -1;
  }
  return 0;
}

/*************************************************************************/



/*************************************************************************

      *UserBlock(void * name)

      goal: blocks the account during a specific time

      arguments: username

*************************************************************************/

void *UserBlock(void * name){

  PasswdInfo *passwd;
  int err = 0;
  char user[LENGTH];
  strcpy(user, ( (char*)name) );

  /*Here in this case an account is blocked during 20 seconds, only because
  to demonstrate the system. In a real system, an account has to be block
  during more time. Depends of each implementation.
  It cannot block during a short period to do not suffer of a brute force
  attack. and it cannot block during a big period to doesn't cause a denial
  of service attack*/

  sleep(20);

  passwd = VerifyUsername(user);
  if(passwd == NULL){
      printf("ALERT: ERROR!!!\nThreads(UpdatePassInfo)\n");

  }else{

    passwd->PasswordFailed = 4;   //UNBLOCKED

    pthread_mutex_lock( &lock );   // RESTRICTED

    err = UpdatePassInfo(passwd->username, passwd);
    if (err == -1){
      printf("ALERT: ERROR!!!\nUpdating the file (UpdatePassInfo)\n");
```

```
    }
    pthread_mutex_unlock( &lock );   // END RESTRICTED

  }
  pthread_exit(NULL);

}

/*****************************************************************************/
```

```
/*****************************************************************************
*
*                        Language-based Security
*
*   Project: Exploit Vulnerabilities in in UNIX-based Authentication Systems
*
*   Author: Pedro Gonçalo Bravo Mendes - pedrogo@student.chalmers.se
*           Group 32
*
*   PassDataBase.C file
*
*   The functions are explained below
*****************************************************************************/

//INCLUDES
#include "project.h"


/*****************************************************************************

    VerifyUsername(char *username)

    goal: verifies if a username is valid

    arguments: username specified by the user

    return value: pointer to the struct that has the information about
    the specified user (in case of username does not exist return NULL pointer)

*****************************************************************************/

PasswdInfo *VerifyUsername (char *username) {

  FILE *fp;
  char buffer[LINE_BUFFER_SIZE];

  static char name[LINE_BUFFER_SIZE], password[LINE_BUFFER_SIZE], pass_salt
[LINE_BUFFER_SIZE], uid_info[LINE_BUFFER_SIZE], home[LINE_BUFFER_SIZE], log_shell
[LINE_BUFFER_SIZE];
  /*the file passwd cointais the username, the uid, the encrtpted password,
  the salt, the number of failed attempts and the pass's age*/
  static PasswdInfo value = {name, password, 0, pass_salt, 0, 0, 0, uid_info,
home, log_shell};

  //open the file
  fp = fopen(FILENAME, "rb");
  if(fp == NULL){
    printf("ALERT: ERROR: File not found\n");
    return NULL;
  }

  //reads each line of the file passwd, looking fot the correct user
  while( fgets(buffer, sizeof(buffer), fp) != NULL ){

    if( sscanf(buffer, "%[^:]:%[^:]:%d:%[^:]:%d:%d:%d:%[^:]:%[^:]:%s",
value.username, value.password, &value.uid, value.PasswordSalt,
&value.PasswordFailed, &value.PasswordAge, &value.guid, value.UIDinfo,
value.home_dir, value.login_shell ) != 10 ){
      fclose(fp);
      return NULL;
    }

    if(strcmp( name, username) == 0){
      //there is a match of username (so the username exists in the system)
      fclose(fp);
      return &value;
```

```
    }
  }

  fclose(fp);
  return NULL;

}

/*************************************************************************/



/*************************************************************************

    UpdatePassInfo(char *username, PasswdInfo *passUp)

    goal: update the passwd file

    arguments: username (that the information has to be updated) and
    passUp (strucut type PasswdInfo that contains all the data about the user
    and the respective password)

    return value: in case of success return 1
                  in case of an error return -1 (for example if the user-
            name cannot be found)


*************************************************************************/

int UpdatePassInfo(char *username, PasswdInfo *passUp) {

  FILE *fp;
  FILE *aux_fp;
  int err = 0;

  char line[LINE_BUFFER_SIZE];
  char name[LINE_BUFFER_SIZE];

  //open the file
  fp = fopen(FILENAME, "r");
  if(fp == NULL){
    return -1;
  }
  aux_fp = fopen(FILENAME_AUX, "w");
  if(aux_fp == NULL){
    fclose(fp);
    return -1;
  }

  //reads each line of the file passwd, looking fot the correct user
  while( fgets(line, sizeof(line), fp) != NULL ){
    if(sscanf(line, "%[^:]", name) != 1){
      //error
      fclose(fp);
      fclose(aux_fp);
      unlink(FILENAME_AUX);
      return -1;
    }

    if( strcmp(name, username) == 0){
      // the username is valid - uddate the information
      if( snprintf(line, sizeof(line), "%s:%s:%d:%s:%d:%d:%d:%s:%s:%s\n",passUp-
>username, passUp->password, passUp->uid, passUp->PasswordSalt, passUp-
>PasswordFailed, passUp->PasswordAge, passUp->guid, passUp->UIDinfo, passUp-
>home_dir, passUp->login_shell) >= sizeof(line) ){
```

```
        //error
        fclose(fp);
        fclose(aux_fp);
        unlink(FILENAME_AUX);
          return -1;
      }
    }

    //writes a auxiliar file with the updated information
    err = fprintf(aux_fp, "%s", line);
    if(err < 0){
      //error
      fclose(fp);
      fclose(aux_fp);
      unlink(FILENAME_AUX);
      return -1;
    }
  }

  fclose (fp);
  fclose(aux_fp);
  rename(FILENAME_AUX, FILENAME);
  return 0;
}

/************************************************************************/



/************************************************************************

    PasswdInfo *VerifyBlocked ()

    goal: verify the blocked accounts

************************************************************************/

void *VerifyBlocked () {

  FILE *fp;
  char buffer[LINE_BUFFER_SIZE];

  static char name[LINE_BUFFER_SIZE], password[LINE_BUFFER_SIZE], pass_salt
[LINE_BUFFER_SIZE], uid_info[LINE_BUFFER_SIZE], home[LINE_BUFFER_SIZE], log_shell
[LINE_BUFFER_SIZE];
  /*the file passwd cointais the username, the uid, the encrtpted password,
  the salt, the number of failed attempts and the pass's age*/
  static PasswdInfo value = {name, password, 0, pass_salt, 0, 0, 0, uid_info,
home, log_shell};

  //open the file
  fp = fopen(FILENAME, "rb");
  if(fp == NULL){
    printf("ALERT: ERROR: File not found\n");
    return NULL;
  }

  //reads each line of the file passwd, looking fot the correct user
  while( fgets(buffer, sizeof(buffer), fp) != NULL ){

    if( sscanf(buffer, "%[^:]:%[^:]:%d:%[^:]:%d:%d:%d:%[^:]:%[^:]:%s",
value.username, value.password, &value.uid, value.PasswordSalt,
&value.PasswordFailed, &value.PasswordAge, &value.guid, value.UIDinfo,
value.home_dir, value.login_shell ) != 10 ){
      fclose(fp);
```

```
      return NULL;
    }

    if(value.PasswordFailed == TEMP_BLOCKED ){
      //there is a match of username (so the username exists in the system)
      value.PasswordFailed = 4;
      UpdatePassInfo(value.username,&value);

    }
  }
  fclose(fp);
  return NULL;

}

/*************************************************************************/
```

```
all: project

project: project.h project.c password.c PassDataBase.c
	gcc -g -Wall project.c password.c PassDataBase.c  -
lcrypt -pthread -o project


clean:
	rm -f *.o project
```

```
user1:FPmadrB54AAfg:1000:FP:0:4:1000:USERname1:/home/user1:/bin/sh
user2:.LbICO3SUn3i2:1000:.L:0:2:1000:USERname2:/home/user2:/bin/sh
user3:A2C6BBhL6Cb0I:1000:A2:0:4:1000:USERname3:/home/user3:/bin/sh
user4:oPoArWLCFCBzA:1000:oP:1:1:1000:USERname4:/home/user4:/bin/sh
user5:34eVd5mCTWIA6:1000:34:0:1:1000:USERname5:/home/user5:/bin/sh
```