Universidade do Minho
Escola de Engenharia

Computer Systems Security

Mestrado Integrado em Engenharia Informática

FUSE File Systems

[Authorization of Operations]

A85227 João Azevedo

A85729 Paulo Araújo

Braga,
Janeiro 2021

Resumo

Este documento tem como objetivo apresentar uma implementação de um sistema de ficheiros *FUSE* num sistema operativo *Linux* onde, adicionalmente, se apresenta uma forma de validar o acesso a ficheiros através de um mecanismo *OTP - "One-Time Password"* através da introdução de códigos de verificação enviados aos contactos de utilizadores autorizados a aceder ao mesmo.

Detalhe da solução

Ao longo das próximas secções iremos detalhar todas as opções tomadas nas diferentes definições da camada do sistema global que compõem toda a verificação de um simples *open()* no sistema de ficheiros desenvolvido. Realça-se o facto de que foram tomadas decisões, ao longo do desenvolvimento, relativas à segurança, robustez e desempenho das diferentes partes, questões essas que determinamos como fulcrais em qualquer sistema a desenvolver.

Contact Storage

Achamos por bem começar por falar deste componente no sentido em que, sem a existência do mesmo, ou com uma pobre implementação, a segurança do sistema de ficheiros a desenvolver pode ser toda comprometida.

Este ficheiro tem uma estrutura simples, semelhante a uma tabela de *hashing*, onde, em cada linha, um *username* é mapeado para um endereço de *e-mail*, que servirá como forma de contacto para o mesmo, em caso de necessidade de validação de uma operação. Eis um exemplo simples da composição de cada uma das suas linhas: "`user1::user1@email.com`".

Acessos indevidos a este ficheiro podem levar a autorizações de operações por parte de *e-mails* falsos, o que compromete toda a integridade do conteúdo lá apresentado. Não consideramos, por outro lado, que seja confidencial a informação lá apresentada, pelo que estabelecemos como fator comparativo o ficheiro em *Unix /etc/passwd*, que associa utilizadores a informação acerca das suas contas com permissões de leitura para *other*.

Existem, portanto, um conjunto de pressupostos que foram adotados para garantir um correto controlo de acesso ao ficheiro de dados (`storage.db`):

1. Criação de um utilizador *superuser* para controlo de acesso ao ficheiro;

   Consideramos importante este ponto no sentido em que o acesso ao ficheiro de dados deve ser controlado por alguém que faça uma gestão administrativa ao mesmo e, para isso, adicionámos um utilizador, tornando-o dono do ficheiro:

   ```
   grupo10@ssi:~$ sudo useradd storage-root
   grupo10@ssi:~$ sudo chown storage-root storage.db
   grupo10@ssi:~$ sudo chgrp storage-root storage.db
   ```

2. Atribuição de correta de permissões ao ficheiro de dados:

   Este ficheiro deve, por um lado, ser acedido pelo nosso programa que gere o sistema de ficheiros para ver os contactos dos donos de sub-árvores onde se incluem os ficheiros a autorizar, como também pensar em termos de um programa *setuid* para escrever/atualizar o mesmo. A atribuição correta de permissões começa por decidir que apenas o dono *storage-root* pode escrever no ficheiro, os outros *bits*, relativos ao grupo e aos outros, apenas podem ler o ficheiro:

   ```
   grupo10@ssi:~$ sudo chmod 644 storage.db #-rw-r--r--
   ```

   Como referido, anteriormente, esta decisão é equiparada ao que se encontra definido em */etc/passwd*.

3. Defininição de um programa *setuid*[5] para controlo de acesso ao ficheiro:

   Foi necessário também criar um programa que permita, a um qualquer utilizador, com acesso ao sistema de ficheiros, escrever ou atualizar o seu contacto e, com esse fim, criámos o programa *updct*:

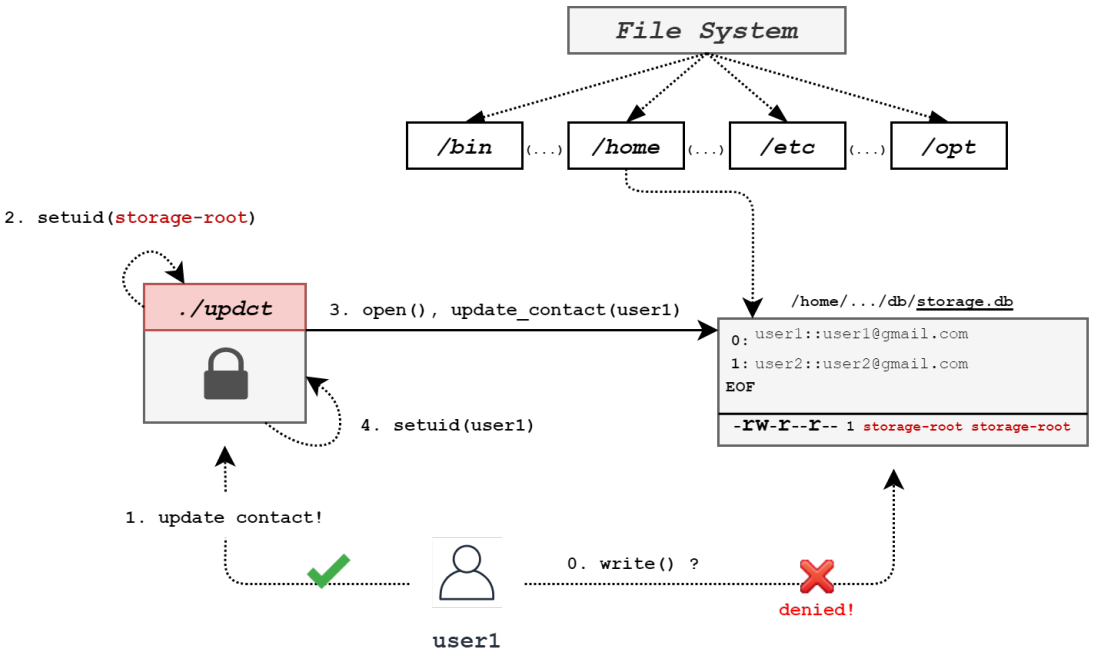

Fig. 1: Programa que atualiza os contactos no ficheiro de dados.

O funcionamento deste programa é semelhante ao */bin/passwd*, onde antes de proceder à escrita no ficheiro é mudado o *effective user id* para o identificador do dono do programa que é também dono do ficheiro de dados, pelo que o *bit* de *setuid* deve estar definido, após compilado[1]:

```
grupo10@ssi:~$ sudo chmod u+s updct
```

File System in User Space (libfuse)

A biblioteca *libfuse* apresenta uma API dinâmica que permite a um qualquer utilizador criar o seu sistema de ficheiros sem ser necessário implementar código no *kernel* diretamente.

Começamos então por experimentar algumas das implementações[2] presentes na diretoria *examples*, desde o *hello.c* ao *passthrough.c*, pelo que escolhemos seguir o segundo exemplo, onde todo o sistema de ficheiro é replicado, através de *symbolic links*, para a raiz do sistema de ficheiros onde é montado.

A primeira operação que é realizada quando executamos este gestor do sistema de ficheiros é o carregamento da base de dados *storage.db*, incluindo também um *timestamp* de quando foi alterada pela última vez, algo que nos irá ajudar, mais tarde, quando tentamos aceder a um contacto de um utilizador.

```
//load contact storage database
load_contact_database(storagedb, STORAGE_PATH);
```

A escolha de funções a implementar para gerir o nosso sistema de ficheiros é vasta e muitas das opções apresentadas não apresentam o foco do sistema a desenvolver. Seguem-se alguns exemplos das funções "implementadas":

```
static const struct fuse_operations xmp_oper = {
    .getattr    = xmp_getattr , .access     = xmp_access ,
    .open       = xmp_open    , .create     = xmp_create ,
    .read       = xmp_read    , .write      = xmp_write  ,
    .statfs     = xmp_statfs  , .chown      = xmp_chown  ,
    //...
};
```

[1] A compilação dos diferentes componentes será explicada mais à frente, recorrendo a ficheiros *Makefile*.

Entre estas todas, destacamos a função *open()*, que requere mais atenção pois é esta a *system call* que será chamada quando um utilizador tenta abrir um ficheiro, seja para escrita ou leitura. Procedemos então à explicação do raciocínio associado a esta função `xmp_open(path, ...)`:

(a) Utilizar o contexto do *fuse* para perceber qual é o utilizador a executar e informações do ficheiro */etc/passwd* e a system call stat para saber quem é o dono de um ficheiro:

```
//who's the user requesting?
username = getpwuid(fuse_get_context() -> uid) -> name
//who's the owner ?
owner    = getpwuid_r(stat(path, ...) -> uid, ...)
```

(b) Bloquear ou permitir acesso mediante o algoritmo seguinte:

```
if (username == owner)
    //allow access
else
    //any changes in db file? -> use stat(path, ...)
    //-> update if needed (modify date > last modification date)

    //owner has contact registred?
    if (has_contact)
        codeX = generate_code()
        send_email(owner, codeX, ...)
        while(time < 30 seconds && not confirmation)
            sleep(...)
            //check API for codeX
        //authorize/deny upon owner decision or block if time expires
    else
        //block access
```

De forma geral, se o utilizador que quer aceder a um ficheiro é dono do mesmo então pode fazê-lo, senão terá de se contactar o dono, caso este tenha um contacto no ficheiro de dados. Em caso afirmativo, foi desenvolvido um módulo de envio de *e-mails* em *C*, onde um exemplo mais prático será apresentado mais à frente e o dono tem, assim, um *link* + código que deverá inserir, assim como a sua decisão, através de uma página *web* desenvolvida para o efeito.

O processo de geração de um código foi também um outro aspeto que tivemos em consideração pois achamos que deve ser impossível, em tempo útil, decifrar o mesmo e, para isso, consideramos o seguinte:

- O código deve ser gerado com, pelo menos, 10 caracteres e o *charset* usado deve garantir uma vasta quantidade de caracteres;
  `charset = {abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ123456789}`

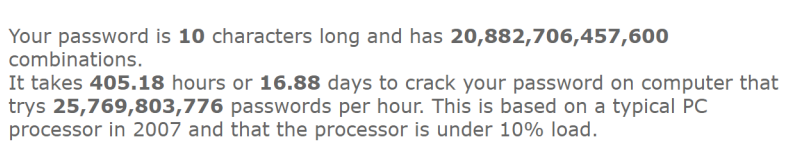

Fig. 2: *Brute force calculator* para um código exemplo "A4Ab3cdseB".

- A API, apresentada de seguida, deve apresentar limitações no que toca à inserção bruta de códigos, para que possa reduzir a possibilidade de os decifrar, em tempo útil;

Com estas premissas e o resultado obtido pelo *Brute force calculator*[6], temos que é praticamente impossível, em menos de 30/40 segundos, que o código seja decifrado.

Relativamente ao contacto ao dono, este é feito através de um *e-mail* enviado utilizando, por exemplo, o servidor *smtp* da Google para o serviço *Gmail*, onde achamos por bem, por segurança, ativar uma comunicação TLS/SSL[3] entre as partes, através de uma verificação de certificados[4] válidos obtidos para ter a certeza que o destino é quem diz que é. Por exemplo, se ativarmos o *verbose output* da *libcurl* temos o seguinte *output*:

```
< 220 2.0.0 Ready to start TLS
* successfully set certificate verify locations:
*  CAfile: /home/devzizu/Desktop/Computer-Systems-Security/tp3
          /filesystem-auth-operations/mail-cert/certificate.pem
*  CApath: /etc/ssl/certs
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* Server certificate:
*  subject: C=US; ST=California; L=Mountain View; O=Google LLC; CN=smtp.gmail.com
*  start date: Jan  5 12:11:24 2021 GMT
*  expire date: Mar 30 12:11:23 2021 GMT
*  subjectAltName: host "smtp.gmail.com" matched cert's "smtp.gmail.com"
*  issuer: C=US; O=Google Trust Services; CN=GTS CA 1O1
*  SSL certificate verify ok.
```

Existem inúmeras fraquezas associadas a *software*, enumeradas no *CWE*[1], relacionadas com a troca de mensagens entre participantes sem verificar a autenticação das partes e a encriptação da comunicação, pelo que achamos importante estabelecer esta propriedade na nossa aplicação.

Code Authorization API

Para a comunicação do código, que desbloqueie/bloqueie a operação sobre um ficheiro, entre o utilizador dono do ficheiro e a aplicação é utilizada uma *API* em *python* que recorre à *framwork Flask*.

Esta componente armazena um conjunto de códigos e a decisão tomada pelo utilizador relativamente ao mesmo. No exemplo seguinte temos a estrutura armazenada, contendo apenas, neste caso, um código:

```
pending : {
    "AAsdas4Asc1": {
        "decision":"authorized",
        "timestamp":"Fri, 22 Jan 2021 16:29:06 GMT"
    }
}
```

Com isto a *API* desenvolvida apenas disponibiliza dois métodos de requisição para o mesmo recurso (URI), enunciados em seguida:

1. ***GET: "/auth-fs/api/codes"***, sem parâmetros apresenta a lista total de códigos (para *debug*), ou fornecendo o campo id=X, obtemos o *status* do código X.
2. ***POST: "/auth-fs/api/codes"***, para inserir um novo código na API;

Para uma maior segurança e robustez desta aplicação tiveram ainda de ser inseridos alguns mecanismos para conter ainda mais as acções que o utilizador poderia realizar, de forma a evitar que este explorasse vulnerabilidades no nosso sistema, nomeadamente:

- **Limpeza de códigos**: a cada segundo é verificada a validade dos códigos existentes e se os mesmos não se encontrarem válidos são removidos. Por válidos consideramos códigos com um tempo de existência inferior a 40 segundos, para tal a importância da existência da *timestamp* referida anteriormente para cada código. Por outro lado, a própria *syntax* do mesmo e tamanho devem seguir uma certa expressão regular, pelo que serão rejeitados os que não a cumprirem, com um *http error code 400*.
- **Limitação de *requests***: foi limitada a realização de *POSTs* a 6 por minuto, para evitar as tentativas de ataque do tipo *denial of service*, através do envio de um grande número de pedidos congestionando a *API*, ou *brute-force* de geração de códigos.
- **Validação de input**: O código e a decisão recebida, na inserção de um recurso, através de POST, são verificados, isto é, se seguem o padrão esperado, impedindo a inserção de "*lixo*" na mesma;

Mais uma vez, nesta componente tivemos em atenção a problemas comuns no desenvolvimento de *software*, relacionados principalmente com controlo de *Input* na nossa API, *flushing* de informação antiga para evitar que sejam utilizados códigos expirados, também o controlo de pedidos simultâneos para evitar negação de serviço nesta componente que deve estar sempre "*on*" que tem como vantagem extra impedir que atacantes tentem, por *brute-force*, gerar autorizações para códigos gerados nos últimos 30 segundos.

Putting everything together

De modo a percebermos melhor o fluxo de mensagens de entre todas as componentes do sistema desenvolvido e em como aplicar o exemplo prático que também iremos mostrar, apresentamos, de seguida, um diagrama geral:

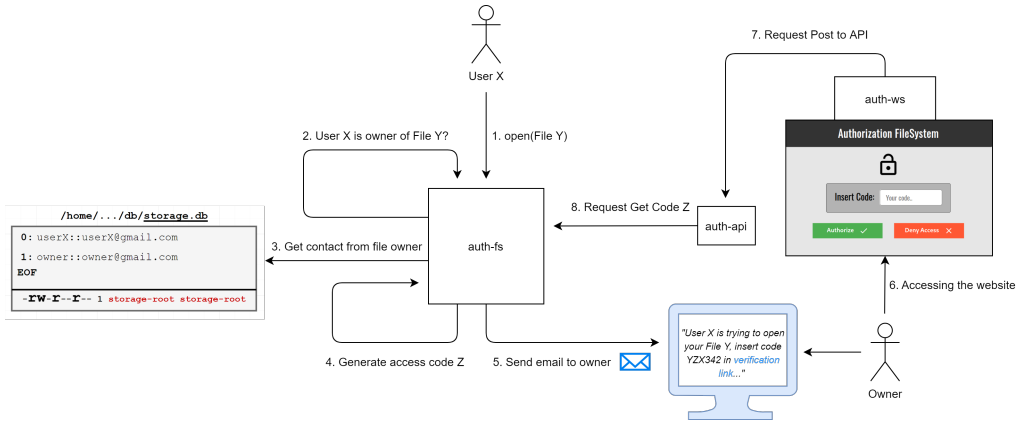

Fig. 3: Diagrama que representa o sistema global.

Decidimos dedicar uma secção deste trabalho a apresentar um caso prático de aplicação e execução do nosso sistema, pelo que a seguinte enumeração servirá como guia:

1. Exemplo prático de aplicação de um caso em que um utilizador *user1* tenta aceder a um ficheiro do utilizador *devzizu* e o acesso é negado pelo mesmo;

   O sistema de ficheiros encontra-se montado em /tmp/auth-fs, pelo que o *user1* procede à visualização do conteúdo do ficheiro *file1* do utilizador *devzizu*, sendo o código gerado de seguida:

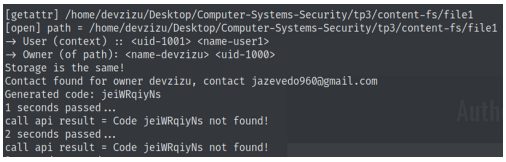

Fig. 4: *Output* do sistema de ficheiros indicando o código gerado.

Entretanto, podemos ver que a aplicação se encontra bloqueada e inclusivé já foram pedidos dois *GETs* à API para o código com resultado "*Not Found*":

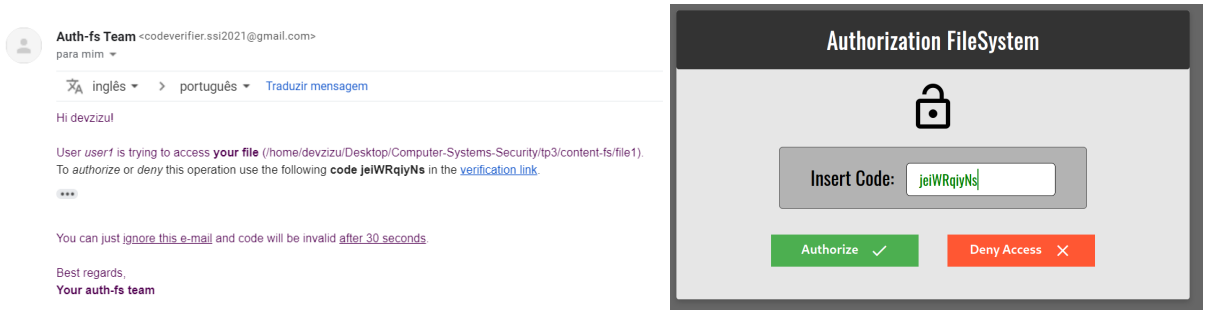

Fig. 5: E-mail recebido pelo dono e inserção do código no servidor *web*

.

A operação será, por exemplo, rejeitada, clicando no botão "Deny Access" e resultará em que o sistema de ficheiros deixa de estar bloqueado, visto que eventualmente lê esse código da API, dando lugar a um erro retornado pelo *open()* ($EACCESS = 13$), que resulta na visualização de um código de erro para "*Permission Denied*":

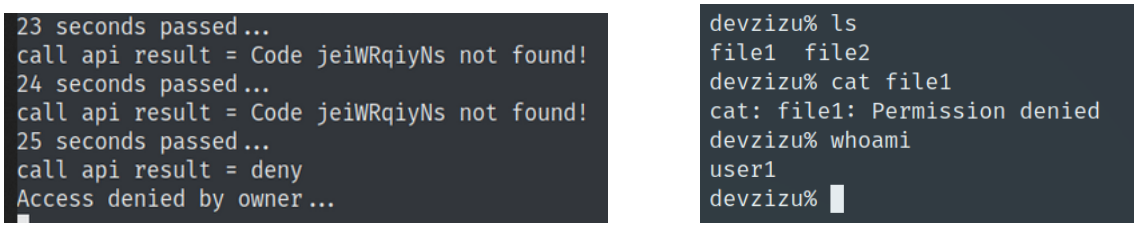

Fig. 6: Operação negada pelo dono (na esquerda *output* do sistema de ficheiros e na direita *output* do comando *cat*).

O desenvolvimento da interface foi algo que não demos demasiada relevância neste documento até então, mas destacamos a forma como a mesma está organizada: temos uma página *web* responsiva que controla o *input* enviado, não deixando fazer pedidos à API se o tamanho do código não for o certo e outras situações, como a sintaxe do código, apresentando *pop-ups* de erros para prevenir *spam* por clicar em botões, todas estas situações apresentadas como possíveis causas de negação de serviço, validação de *input*, controlo de pedidos apresentados na *CWE*.

Project tree - Folder structure

Este tópico foi criado com o intuito de aumentar a ambientação dos leitores deste documento com a estrutura do nosso projeto. Assim, temos um projeto divido em quatro programas independentes:

- **auth-fs**: Lógica do *File System* implementado recorrendo à *libfuse* (*auth-fs.c e fs-tools.c*), a lógica relativa ao envia de mails (*mail.c*), lógica do acesso à *API* por parte da nossa aplicação (*auth-api.c*) e ainda a lógica de todo o acesso e armazenamento no ficheiro *storage.db* (*storage.c*). Este módulo contém uma *Makefile* que faz a compilação de todos estes sub-módulos e execução/montagem do sistema de ficheiros:

```
# Variables that can be changed in Makefile
# FS=auth-fs #program name
# MOUNT_PLACE=/tmp/$(FS) #mount place (default = /tmp/auth-fs)

grupo10@ssi:~$ apt install libjson-c-dev libglib2.0-dev libcurl4-openssl-dev #deps
grupo10@ssi:~$ cd auth-fs && make
```

- **auth-api**: API, em *python*, desenvolvida:

```
grupo10@ssi:~$ apt install python3 python3-pip && pip3 install Flask Flask-Limiter #deps
grupo10@ssi:~$ cd auth-api && python3 auth-api.py
```

- **auth-ws**: Contém os ficheiros necessários para a implementação da *web page*, nomeadamente, o *css, javascript* e *hml*:

```
grupo10@ssi:~$ apt install nodejs npm && sudo npm install -g http-server #deps
grupo10@ssi:~$ cd auth-ws && http-server .
```

- **updct**: este módulo é aquele que contém o programa que permite a um utilizador alterar de forma dinâmica apenas e só o seu contacto do ficheiro *storage.db*:

```
# Note:
# 1. The user storage-root can alter everyones contact, and each user,
#       besides storage-root, can only alter its contact.
# 2. This program should have permissions set as stated in the first section,
#       and make sets the setuid bit with chmod

grupo10@ssi:~$ cd updct && make #make compiles code, only storage-root can do it
grupo10@ssi:~$ cd updct && ./updct
```

Este programa tem extremo cuidado e está desenvolvido tendo em atenção problemas/fraquezas comuns de *software*, por exemplo:

1. O controlo de acesso e inserção de um novo contacto só é feito para o utilizador que está a executar o comando, exceto para o nosso *superuser storage-root*, onde estas permissões se encontram desligadas;
2. Todo o *input* é controlado em termos da sua validação, como por exemplo, verificar se é inserido um *e-mail* válido, assim como o tamanho do mesmo utilizando a função *fgets*, sendo esta mais segura e prevenindo *buffer-overflow attacks*.
3. As permissões só são alteradas durante a escrita do ficheiro e revertidas no fim do processo;

Em termos da atualização do conteúdo do ficheiro o algoritmo é simples, pois é apenas inserida uma nova linha do ficheiro com o novo contacto mapeado para o utilizador que correu o programa. Deste modo, quando o programa do sistema de ficheiros pretende validar uma operação, compara as datas de modificação do ficheiro através da *system call stat* e vê que é necessário recarregar a estrutura para memória principal e, como para as estruturas usamos *hashtables* (da *glib*), um *insert* repetido vai invalidar/substituir um contacto antigo.

- **db**: Diretoria onde se encontra armazenado o ficheiro *storage.db*, cujos caminhos e permissões não devem ser alteradas.

Conclusion

Para solucionar o problema de validação de operações foi necessário percorrer diversas áreas um pouco desconhecidas,sendo essa a essência do problema, fornecendo-nos ferramentas muito importantes para o futuro que podem ser utilizadas não só num contexto académico, mas também na aplicação de um problema de contexto real.

Um dos objetivos, não listados no enunciado, mas que gostaríamos de ter implementado e ao qual chegamos a fazer testes mas sem grande sucesso, é relativo à gestão *multi-threaded* de pedidos de acesso ao sistema de ficheiros, algo que achamos importante num sistema partilhado, mas que traria algum tipo de problemas de concorrência.

Ao longo deste trabalho foi também extremamente importante ter atenção ao conjunto de fraquezas comuns verificadas na *Common Weakness Enumeration* relativas ao desenvolvimento de *software* de modo a evitar problemas comuns que fomos apresentados e que podem causar negação de serviço, preocupação com *inputs*, ataques de *brute-force* na geração de códigos, *interfaces web* cuidadas com proteção contra *spam* e envio de *e-mails* tendo por base certificados validados aquando envio para autenticar e constituir uma comunicação segura entre os participantes.

References

[1] CWE - CWE-699: Software Development (4.3). (n.d.). CWE. https://cwe.mitre.org/data/definitions/699.html

[2] libfuse/libfuse. (n.d.). GitHub. https://github.com/libfuse/libfuse

[3] curl - SSL CA Certificates. (n.d.). CURL. https://curl.se/docs/sslcerts.html

[4] curl - Extract CA Certs from Mozilla. (n.d.). Mozzila. https://curl.haxx.se/docs/caextract.html

[5] Setuid Program Example (The GNU C Library). (n.d.). GNU. https://www.gnu.org/software/libc/manual/html_node/Setuid-Program-Example.html

[6] Brute Force Calculator. (n.d.). Brute Force Calculator. https://tmedweb.tulane.edu/content_open/bfcalc.php