



Mestrado Integrado em Engenharia Informática
Segurança de Sistemas Informáticos

TP3

Grupo 8:

- ✓ Paulo Jorge Pereira Martins (Nº PG17918)
- ✓ João Nuno Alves Lopes (Nº A80397)

ÍNDICE:

Introdução	1
Arquitetura	2
Requerimentos	2
Base de dados	3
Funcionalidades	3

Introdução

Neste trabalho foi-nos proposto o desafio de criar um sistema de ficheiros Linux com base no *libfuse*, incorporando um sistema de autenticação. O objetivo é implementar mecanismos de autenticação por dois passos, com código de verificação temporário, necessário se o utilizador desejar usar funções “open” no sistema de ficheiros caso contrário não conseguiria abrir.

Optamos por realizar em primeira instância uma autenticação por via de login tradicional e numa segunda um sistema de autenticação por código temporário enviado via e-mail, estendendo a segurança com uma autenticação por dois fatores. Na implementação da aplicação optamos por criar uma *GUI* e implementar maioritariamente em *Python*. Nos dias de hoje, onde já é frequente a utilização de dados biométricos como reconhecimento facial ou de impressão digital, consideramos esta tarefa de autenticação por dois fatores interessante de implementar, visto que é cada vez mais um requisito de segurança comum e imprescindível.

Arquitetura

Requerimentos

A aplicação foi programada essencialmente em Python3, com interface gráfica baseada em webviews (pywebview) que comunica com um servidor local (Flask), que comunica de forma assíncrona por via de chamadas Ajax. O servidor é uma thread que expõe páginas web (HTML, CSS, JS) na GUI da webview. A persistência dos dados é em SQLite, implementada por via da biblioteca pyDAL (biblioteca Python que implementa uma Database Abstraction Layer / ORM). A arquitetura implementada é uma variante do modelo MVC (Model, View, Controller), sendo esses os 3 packages principais da aplicação:

- Package “/core”: backend Python e lógica central da aplicação. Pode ser dividido essencialmente nos módulos relativos à autenticação (“authentication.py” que fará a gestão dos logins, registos e códigos de verificação, criando a classe do User, baseada na classe abstrata “user.py” usada pelo “client.py” e o pelo “admin.py”). O módulo “libfuse.py” incorpora todos os mecanismos relativos ao libfuse. Por último o módulo “tools.py” agrega funções simples úteis a diversos mecanismos.
- Package “/views”: implementação da GUI, o módulo “gui.py” cria uma janela “webview” e lança um servidor em “Flask” como thread, que irá expôr páginas na “webview” e comunicar com os restantes packages. O ficheiro “routes.py” expande o servidor com endpoints e interliga as páginas web com o backend Python. A pasta “templates” agega os temples “HTML” e a pasta “static” os ficheiros estáticos, como p.e. CSS, mas também funções em javascript que tratam das chamadas assíncronas.
- Package “models”: persistência. Para além da base de dados (“/models/sqlite”) e ORM (“db.py”), será criado nesta pasta a o “mountpoint” do libfuse (quando a aplicação é lançada esta apaga e/ou cria uma pasta “mountpoint” dentro do “models” temporariamente). O mesmo sucesso com a pasta “/models/storage”, nela serão guardados todos os ficheiros base dos utilizadores e logs.

De forma a correr a aplicação é necessário um ambiente Linux com Python3 instalado, para além das seguintes bibliotecas *Python*:

- Python3
- E as seguintes bibliotecas *Python*:
- fusepy
 - flask
 - pywebview
 - pydal

Quanto ao “libfuse” utilizamos a biblioteca Python “fusepy” e adaptamos no ficheiro “libfuse.py” no *package* “core” os mecanismos correspondentes a cada operação. É de sublinhar que incluímos uma opção de inicializar o libfuse por via da linha de comandos (com o comando:

python3 libfuse.py mountpoint/path storage/path). Inicialmente planeávamos suportar em versão GUI e versão por linha de comandos, no entanto a partir do ponto que implementamos sistema de login e de autenticação deixamos de o suportar devido a limites de tempo e esforço redobrado. No entanto, embora a aplicação neste momento só funcione no formato completo em GUI, os mecanismos base para a versão em linha de comandos continuam presentes.

No ficheiro “requirements.txt” incluímos instruções sobre como instalar as bibliotecas necessárias. O ficheiro “main.py” é o ponto de entrada da aplicação, como Python é uma linguagem interpretada não necessita de compilação, tendo as tecnologias necessárias instaladas bastará executar esse ficheiro para que a aplicação seja lançada. É de sublinhar que devido ao facto de invocarmos comandos do terminal Linux na execução da aplicação, e como certos diretórios (relativos vs absolutos) mudam dependendo se a aplicação é lançada via execução normal ou via terminal, aconselhamos que a aplicação seja executada pela via normal (duplo clique no ficheiro “main.py” e seleccionar executar), uma vez que não testamos métodos alternativos (p.e. é possível lançar pelo terminal com o comando “python3 main.py”, no entanto assumirá diretórios relativos ao terminal em vez de absolutos por esta via, mas não tivemos tempo de contemplar todos os cenários nem testar por esta via de inicialização).

A aplicação apenas foi testada em *Ubuntu 18*.

Base de dados

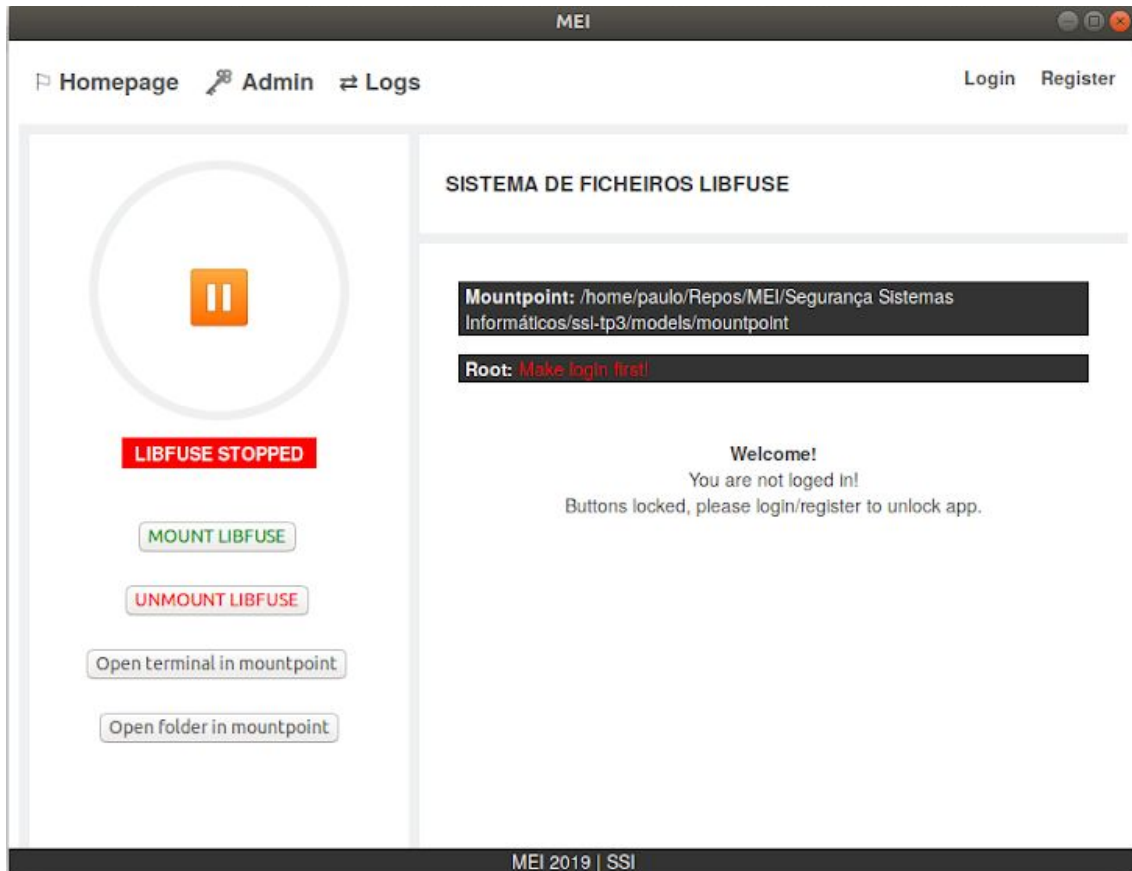
A persistência de dados da aplicação é uma base de dados SQLite que engloba uma tabela “users” com a seguinte estrutura de dados:

```
def tableUsers(self):
    try:
        #Note: id is created automatically if omitted
        self.db.define_table('users',
            Field('username', type='string'),
            Field('email', type='string'),
            Field('password', type='string'),
            Field('uid', type='string'),
            Field('userRole', type='string', default='user'), #user, admin
            Field('secureKey', type='string', default='0', writable=False, readable=False), #secret key token
            Field('dateRegistration', type='datetime', writable=False, readable=False)
        )
    except:
        print('models db.DB() tableUsers Error')
```

Na pasta “models” em “storage” guardamos e geramos “logs” relativos a cada utilizador e de todas as operações registadas pelo “libfuse”.

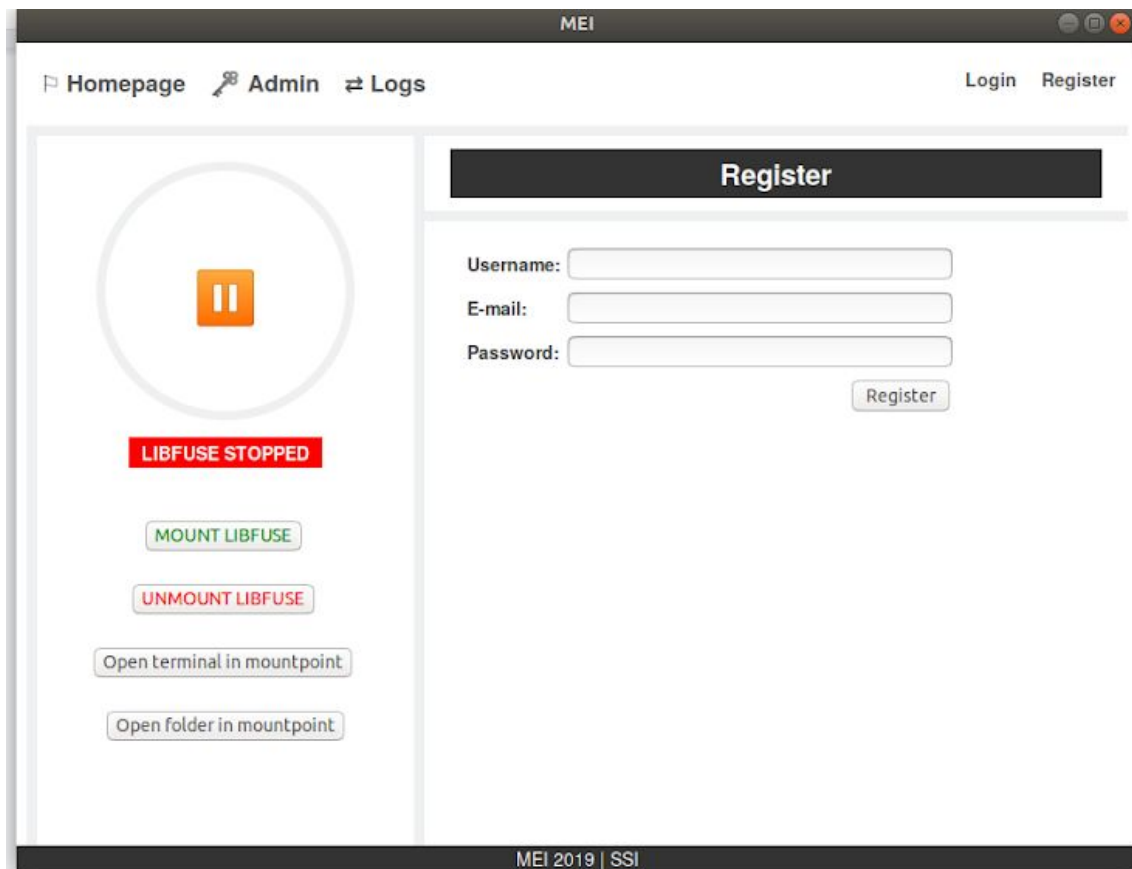
Funcionalidades

Ecrã inicial da aplicação:



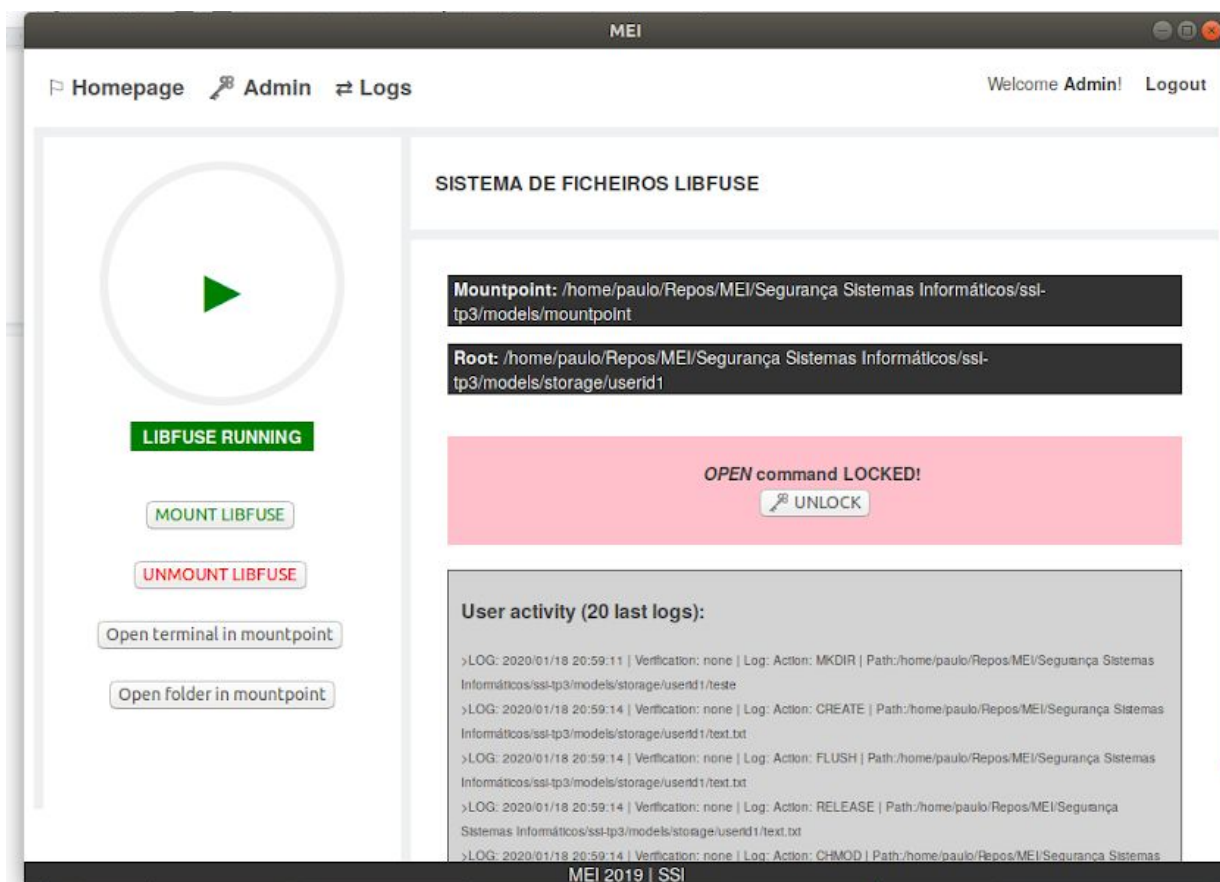
Ao entrar o utilizador apenas poderá efetuar registo ou login na aplicação, os restantes botões estarão bloqueados e não permitem avançar enquanto o login não for efetuado com sucesso, exibindo uma mensagem a pedir login caso o utilizador carregue neles. Podemos também ver o diretório "default" on será criado o mountpoint (nota: pode facilmente ser alterado nas configurações de lançamento). O diretório root será o diretório real onde se encontram os ficheiros do utilizador. No lado esquerdo estão os botões para interação com o libfuse.

Ecrã de registo:



O registo é armazenado numa base de dados SQLite no *package* "models/sqlite". Idealmente, num cenário de segurança real, esta base de dados deveria ser mais protegida (por exemplo, numa arquitetura distribuída cliente-servidor ela, e certos microserviços, poderia estar protegida num servidor externo isolado (de preferência utilizando um engine mais robusto como PostgreSQL p.e.), que apenas exporia dados JSON específicos mediante autenticação. No formato atual, qualquer pessoa com ferramentas simples de visualização de SQLite, poderá abrir e ver os dados e passwords de todos os utilizadores registados. Por uma questão de falta de recursos (não temos acesso a um servidor externo) e por limites de tempo e não se tratar de um ambiente de produção, optamos por SQLite por ser mais prático, apenas como protótipo. É também de sublinhar que todos os dados inseridos nos formulários na aplicação deveriam ser sanitizados, ou seja, "escaped" e tratados para assegurar que utilizadores mal intencionados não explorarem a segurança da aplicação. O mesmo se passa com "cross site scripting" (XSS), idealmente deveriam ser implementados *tokens* de segurança e medidas de proteção para assegurar que não haveria exploração destes problemas de segurança. Tivemos o cuidado apenas com aspetos básicos desse contexto, por se tratar de um protótipo, em ambiente de desenvolvimento e não de produção, e por questões de limite de tempo e recursos, não focamos muito esses aspetos, mas sublinhamos que seriam facilmente tratados com consciência em ambiente de produção real.

Opções após login e opção "MOUNT LIBFUSE" ativa:



Podemos aqui observar o contexto central da aplicação, com todas as opções em exibição depois de login com sucesso. Também lançamos o “libfuse” através da opção “MOUNT LIBFUSE”, que o inicializou numa *thread* isolada, efetuado “mount” do diretório “root”. Este diretório “root” é criado, caso ainda não exista, na função de login, e é sempre armazenado no diretório “models/storage” da aplicação, segundo a seguinte lógica: para cada utilizador é criada uma pasta com o nome “userid” + o “id” associado ao registo do utilizador (p.e. “Userid1”, “userid2”, etc.).

É de sublinhar que na altura do registo do utilizador, é guardado o seu “uid” atual na base de dados. Quando o diretório para os seus ficheiros é criado, efetuamos um comando “chown” por via da biblioteca “os” do *Python*, aplicando o “uid” do utilizador registado como utilizador dono do diretório, desta forma outros utilizadores da máquina não terão permissões de acesso.

É de notar que idealmente o nosso objetivo era proteger os ficheiros do utilizador na pasta real de, que serão projetados no “mountpoint”, numa máquina externa, como por exemplo uma pasta de rede, ou, em alternativa, encriptando e desencriptando a pasta de cada utilizador em cada login, desta forma os ficheiros nunca poderiam ser acedidos por outros utilizadores da mesma máquina. Por falta de recursos (máquinas externas) e por falta de tempo optamos por efetuar tudo localmente, estando as pastas do utilizador expostas na mesma máquina de acesso de forma desprotegida, apenas com permissões mínimas associadas ao “uid”. Tentamos criar as pastas com permissões “700” (apenas leitura, escrita e execução pelo

utilizador dono) mas tal criou problemas na execução da aplicação, uma vez que era executada sem direitos “root”.

Parte destes processos podem ser observados no ficheiro “tools.py” na pasta “core” da aplicação, por exemplo nas linhas:

```
8
9 #mode='0700'
10 def createDir(path, mode='700'):
11     #makedirs honors unmask, so in some systems permissions are ignored
12     #So we first ignore unmask
13     try:
14         original_umask = os.umask(0)
15         if not os.path.exists(path):
16             #os.makedirs(path, int(mode))
17             os.makedirs(path)
18
19     except:
20         print("Error - not possible to create directory")
21     finally:
22         os.umask(original_umask)
23
24
25
26 def dirCHOWN(path, uid):
27     #os.setuid(int(uid))
28     #shutil.chown(path, user=uid)
29     os.chown(path, int(uid), int(uid))
30
```

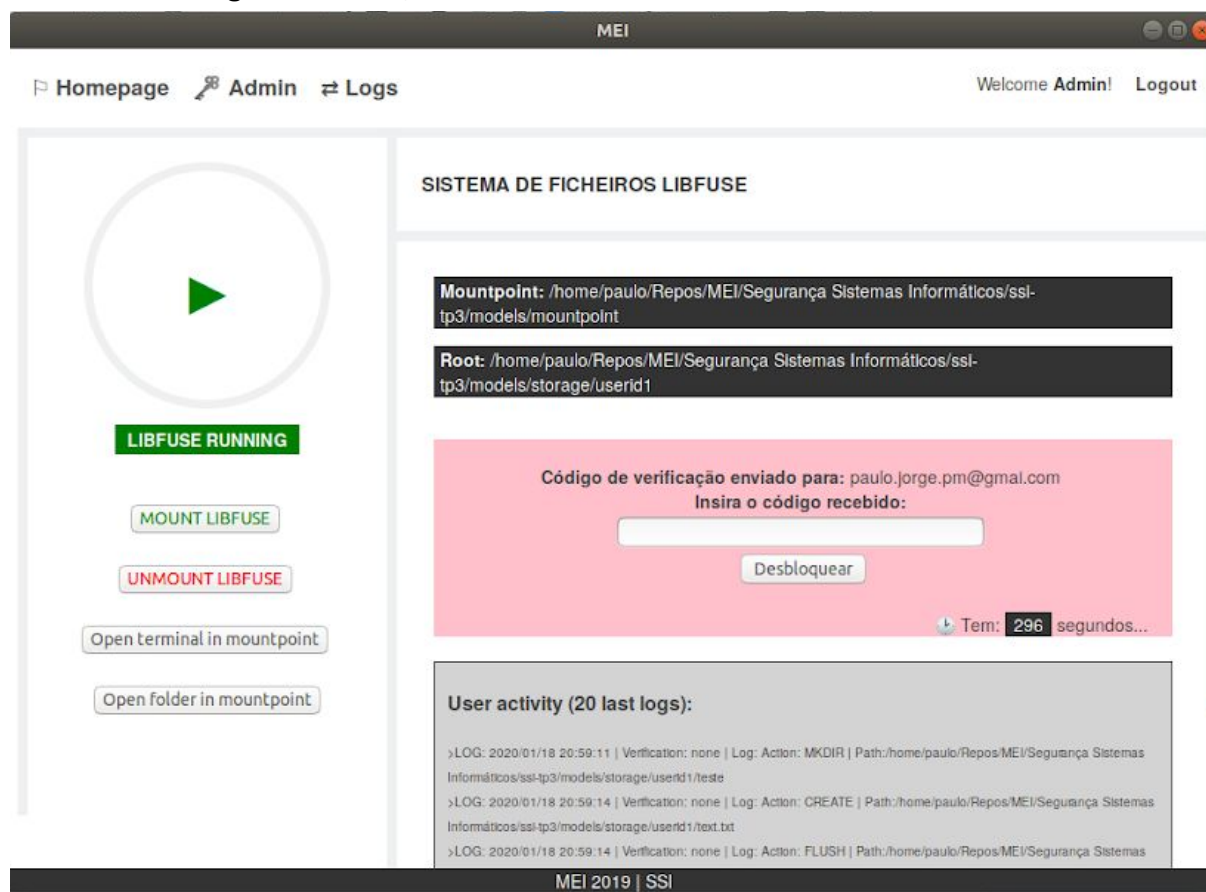
Outro ponto de interesse é o facto de implementamos um sistema de “logs” que exhibe e documenta cada ação dos diferentes utilizadores, como se pode observar no bloco “User Activity”. Os administradores têm acesso aos logs completos armazenados.

É também de sublinhar que antes de desligar a aplicação o utilizador deverá “unmount” o libfuse, através do botão “UNMOUNT LIBFUSE”, caso contrário o “mountpoint” continuará na diretoria, sendo necessário desligá-lo manualmente (efetuamos o “unmount” através do comando “fusermount -u path”).

As opções “open in terminal mountpoint” e “open folder in mountpoint” são apenas atalhos de conveniência, que abrem o sistema de ficheiros ou o terminal já na diretoria do “mountpoint”. É de sublinhar que o “open terminal” poderá não funcionar em todos os sistemas Linux, uma vez que utilizamos o comando “os.system('gnome-terminal --working-directory=path)” para lançar o terminal, no entanto tal é baseado apenas em sistemas com o “gnome”. Já o comando para lançar o diretório de ficheiros é mais transversal, pois usa o comando “os.system('xdg-open path)”, que se demonstrou mais consistente entre os diferentes “Distros” de Linux que testamos.

O objetivo deste trabalho era limitar o comando “Open” por via do “libfuse” por via de um código de verificação temporário, que o utilizador terá de inserir de forma a obter acesso. Todos os comandos no sistema de ficheiros funcionam normalmente no “passtrough” implementado no ficheiro “libfuse.py” exceto o “open”, esse passa por um sistema de verificação. Para iniciar a verificação o utilizador deverá carregar no botão “Unlock” exibido no painel superior, no bloco “Open command locked”.

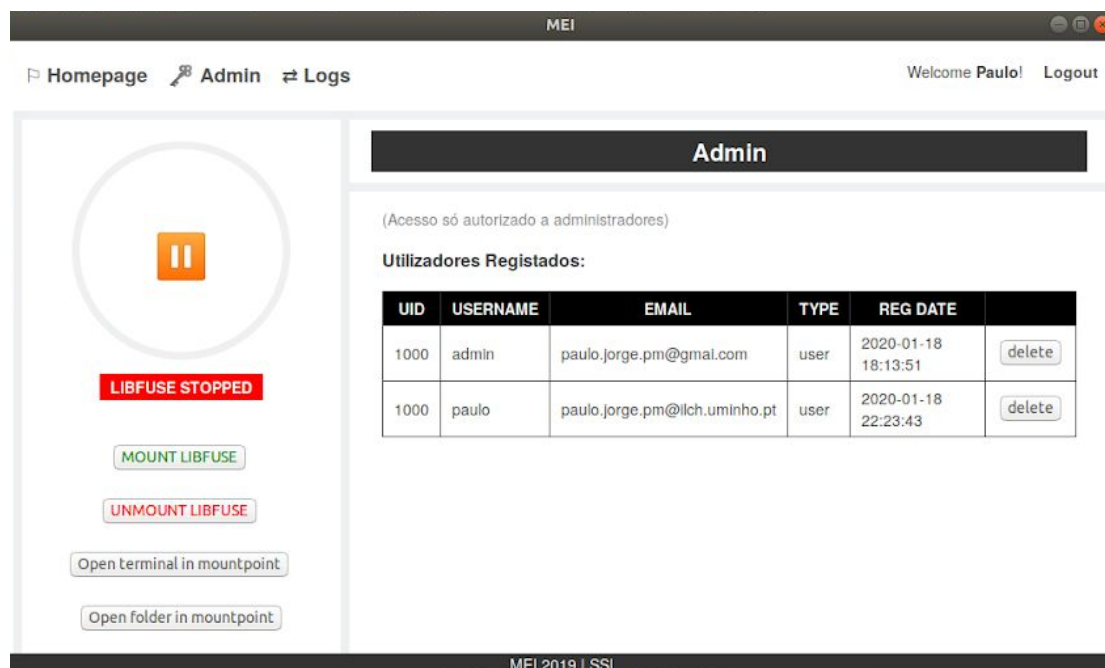
“Unlock” e código de verificação:



Quando um utilizador carrega no botão “Unlock” (print screen anterior) inicia o processo de autenticação por dois factores. Optamos por enviar um código por email, o qual terá de ser inserido no formulário no prazo máximo de 300 segundos (5 minutos). Depois desse prazo o processo será cancelado. O temporizador é programado em AJAX/JS, que chama uma função no backend *Python* que efetua reset ao processo caso o tempo seja ultrapassado. Caso o código esteja correto aparece uma mensagem a indicar que o desbloqueio decorreu com sucesso.

Geramos o código que servirá de autenticação de forma aleatória. É um código alfanumérico com 43 caracteres “url safe” (opcionalmente poderíamos ter optado por um link no e-mail para ativação direta no servidor por parâmetros “GET”). O código é gerado através da biblioteca “secrets” do Python com o comando “secrets.token_urlsafe()” (no ficheiro “routes.py”).

Painel com dados de administração:




Por último é de referir que criamos uma página para os administradores efetuarem gestão dos utilizadores. No entanto por questões de facilidade de prototipagem e para poder ser aberto por terceiros mais facilmente, abrimos o acesso a esta página, neste momento todos os utilizadores podem visualizar através do “link” superior “Admin” (seria facilmente bloqueada no “Flask”, uma vez que na base de dados, na tabela “users” se o utilizador é do tipo normal ou “admin”, os mecanismos já estão implementados apenas abrimos acesso para facilitar acesso.

E por último o “link” “Logs” na barra superior de navegação é um caso idêntico, exibe os “logs” de atividade de todos os utilizadores, mas apenas aos administradores. Para facilitar o acesso desligamos o bloqueio aos “users” do tipo “admin”:

MEI

HomepageAdminLogs

Welcome Paulo!Logout



LIBFUSE STOPPED

MOUNT LIBFUSE

UNMOUNT LIBFUSE

Open terminal in mountpoint

Open folder in mountpoint

LOGS:

Registo dos últimos 100 logs de todos os utilizadores:

>LOG: 2020/01/18 20:59:11 | Verification: none | Log: Action: MKDIR | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/teste

>LOG: 2020/01/18 20:59:14 | Verification: none | Log: Action: CREATE | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/text.txt

>LOG: 2020/01/18 20:59:14 | Verification: none | Log: Action: FLUSH | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/text.txt

>LOG: 2020/01/18 20:59:14 | Verification: none | Log: Action: RELEASE | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/text.txt

>LOG: 2020/01/18 20:59:14 | Verification: none | Log: Action: CHMOD | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/text.txt

>LOG: 2020/01/18 20:59:14 | Verification: none | Log: Action: ULTIMENS | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/text.txt

>LOG: 2020/01/18 20:59:23 | Verification: none | Log: Action: MKDIR | Path:/home/paulo/Repos/MEI/Segurança Sistemas Informáticos/ssi-tp3/models/storage/userid1/teste/teste2

MEI 2019 | SSI