Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Evandro Fernandes Giovanini

Um modelo de segurança para proteger arquivos pessoais

São Paulo
Dezembro de 2015

Um modelo de segurança para proteger arquivos pessoais

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Marco Dimas Gubitoso

São Paulo
Dezembro de 2015

Resumo

O modelo de segurança de sistemas operacionais como o Linux foi criado para separar e isolar os vários usuários entre si, e manter os arquivos do sistema protegidos deles. Em computadores modernos, onde geralmente há um ou poucos usuários, e eles rodam aplicativos de fontes diversas e desconhecidas, este modelo não é o ideal. Um único aplicativo mal intencionado ou que simplesmente tenha uma falha de segurança pode permitir que todos os arquivos pessoais sejam lidos ou alterados indevidamente. Neste trabalho foi desenvolvido um sistema que permite a execução de aplicativos de forma isolada, de modo que cada aplicativo não tenha acesso a todos os arquivos do usuário.

Palavras-chave: linux, segurança, arquivos, pessoais.

Abstract

The security model of operating systems like Linux was created to separate and isolate users, and keep system files protected from them. In modern computers, where there's one or few users, and they run applications from multiple and unknown sources, this model is not ideal. A single application with bad intentions or a simple security flaw could allow for all of a user's personal files to be read or changed without their knowledge or authorization. In this work we developed a system that allows applications to be run in isolation, in a way that a single application doesn't have full access to a user's personal files.

Keywords: linux, security, files, personal.

Sumário

1 Introdução **1**

1.1 Contexto 1

1.2 Motivação e proposta 1

1.3 Organização do texto 2

2 O ambiente de execução **3**

2.1 Criando o ambiente de execução 4

2.2 Criando os diretórios necessários 5

2.3 Montando os diretórios no ambiente 5

2.4 Iniciando aplicativos 6

3 Concessão de acesso **9**

3.1 Comunicação interprocessos com D-Bus 10

3.2 secappd 11

3.3 Fazendo a junção entre diretórios 13

4 Requisitando acesso **15**

4.1 Permissão pela linha de comando: secapp-reqperm 15

4.2 Gerenciador do Secapp 16

4.3 Editor de Texto 17

5 Conclusões **19**

Referências Bibliográficas **21**

Capítulo 1

Introdução

1.1 Contexto

A segurança dos sistemas operacionais modernos foi criada pensando na realidade de décadas atrás, onde era comum que dezenas de usuários usassem um mesmo computador, por exemplo em universidades. Nessa realidade o principal objetivo de segurança era proteger o sistema dos usuários, e os usuários uns dos outros.

Nos dias de hoje é comum computadores serem pessoais, desde notebooks a telefones celulares, e ao invés de aplicativos serem instalados por um administrador de redes o próprio usuário baixa e instala inúmeros aplicativos da internet, nem sempre de fontes totalmente confiáveis. Ao ser executado, cada aplicativo tem acesso total aos dados do usuário e aos diversos recursos do sistema, sendo possível que um aplicativo mal comportado ou mal intencionado cause danos reais. Um único aplicativo pode apagar todos os arquivos do usuário, varrer sua pasta e enviar informações pela internet, capturar informações em segundo plano, etc.

O modelo de segurança descrito e implementado por sistemas modernos foi concebido há décadas atrás e é incompleto e inadequado para o cenário atual.

1.2 Motivação e proposta

O diretório pessoal do usuário em sistemas Linux é usado por todos os aplicativos para armazenamento de arquivos. Por exemplo:

- aplicativo de bate-papo armazena o histórico de conversas e listas de contatos.
- aplicativo de e-mail armazena todos os e-mails recebidos pelo usuário.
- navegador web armazena o histórico de visitas e senhas da internet.

Além disso, o próprio usuário cria documentos importantes, importa fotos, vídeos ou outros tipos de arquivo, e os armazena em seu diretório pessoal.

É de extrema importância então proteger os arquivos pessoais, evitando que seus dados sejam copiados e lidos sem autorização, ou apagados por algum atacante ou agente mal intencionado. Tais agentes podem explorar falhas de algum aplicativo para conseguir acesso a todas as informações do diretório pessoal; de fato, isso já acontece sempre que falhas de segurança são encontradas. Navegadores web são vetores comuns para esse tipo de ataque[1].

A principal motivação deste trabalho foi encontrar uma forma de amenizar ou eliminar este problema, e isolar os aplicativos, de modo que uma falha de segurança ou até mesmo um comportamento intencionalmente ruim em um aplicativo não tenha efeitos negativos em todos os arquivos pessoais do usuário. O projeto resultante chama-se Secapp.

No desenvolvimento do Secapp foram exploradas tecnologias em diversos níveis do sistema operacional, como funcionalidades providas pelo kernel Linux, o funcionamento de comunicação entre processos diferentes definindo uma interface para isso, e a implementação de programas de alto nível que interagem diretamente com o usuário. A grande verticalidade desse problema permitiu explorar diferentes e diversas áreas do sistema como um todo, e isto em si também foi uma motivação para o desenvolvimento deste trabalho.

O Secapp é software livre e seu código fonte pode ser obtido em seu repositório Git[2].

1.3 Organização do texto

Para evitar os problemas descritos quando qualquer aplicativo executado pelo usuário tem acesso total aos seus arquivos, criamos ao longo desse trabalho o sistema Secapp. O Secapp cria um ambiente restrito para executar aplicativos, onde eles não tem acesso total aos arquivos do usuário. A criação deste ambiente será descrita no Capítulo 2. O Secapp também permite que aplicativos tenham acesso liberado a todos os arquivos, em condições onde isso é necessário; como é feita essa concessão será visto no Capítulo 3, e o modo como aplicativos requerem a concessão será estudado no Capítulo 4. As conclusões obtidas ao longo do desenvolvimento serão vistas no Capítulo 5.

[1] Falha de segurança do Firefox: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4495
[2] Repositório com o código fonte do Secapp: https://github.com/evandrofg/secapp

Capítulo 2

O ambiente de execução

Nosso objetivo com o Secapp é impedir que um aplicativo tenha acesso de leitura e escrita a todos os arquivos do usuário. Para entender melhor quais arquivos devem ser vistos normalmente, e quais terão seu acesso limitado, podemos pensar na estrutura de diretórios do Linux em duas partes:

- diretórios de sistema, como usr, lib, bin, dev, etc, proc e var;
- diretórios pessoais de usuários, em home.

Os diretórios de sistema contém arquivos executáveis, bibliotecas e arquivos de dados e configurações usados por programas em sua execução. Eles são necessários para o funcionamento de aplicativos, e são instalados tanto pelo próprio sistema como por aplicativos adicionais que tenham sido instalados.

O diretório pessoal do usuário é compartilhado entre todos os aplicativos. Por exemplo, arquivos de configuração de um navegador web e cliente de e-mail ficam em um subdiretório *.config*, e tanto um editor de texto como uma planilha de cálculo podem salvar arquivos em um diretório *Documentos*. Por isso é necessário que aplicativos enxerguem e tenham acesso ao diretório pessoal. Mas como fazer isso evitando os problemas possíveis, onde um aplicativo mal intencionado poderia causar danos a arquivos criados por outros aplicativos?

Nossa solução é criar um diretório específico para cada aplicativo e fazer com que o aplicativo rodando com Secapp enxergue esse diretório como se fosse o diretório pessoal. Assim, o funcionamento do Secapp é totalmente transparente ao programa, e garante o isolamento que desejamos.

O funcionamento do Secapp consiste em:

- criar um novo diretório de trabalho específico para cada aplicativo;
- replicar os diretórios de sistema (usr, lib, entre outros) exatamente como eles são;
- criar um diretório de arquivos pessoais, que será visto por este aplicativo.

Dentro do diretório pessoal usuário, o Secapp usa o diretório .local/share/secapp para trabalhar. Dentro deste diretório, cada aplicativo terá o seu diretório específico, e dentro deste serão tomados os passos acima.

Na próxima seção iremos analisar o código do SecApp que cria a estrutura de diretórios que acabamos de descrever, muda o ambiente de execução para enxergar essa estrutura ao invés da estrutura real, e então inicia um aplicativo. O código responsável por isso é escrito em C e faz chamadas de sistema ao kernel Linux para executar essas funções.

2.1 Criando o ambiente de execução

Para executar um aplicativo com o Secapp é disponibilizado o comando secapp-run, que recebe como argumento o caminho do executável do aplicativo. Por exemplo:

```
secapp-run /usr/bin/editor
```

Para abrir um aplicativo com o Secapp, o secapp-run faz uma chamada para a função secapp_open, que está implementada em secapp-helper.c. Seu protótipo:

```
int secapp_open (int argc, char **argv, char **envp, int pid);
```

Os argumentos *argc*, *argv* e *envp* refletem o ambiente que chamou o Secapp, e serão passados ao abrir um novo aplicativo, para que este enxergue as mesmas variáveis de ambiente. A variável *pid* guardará o identificador do processo que será aberto para o aplicativo, e será utilizado pela função que chamou secapp_open, como veremos adiante.

O seguinte trecho de código de secapp_open prepara o ambiente e então inicia o aplicativo. Em vários trechos do código são feitas chamadas de sistema ao kernel Linux, que estão documentadas no *Linux Programmer's Guide* (Burkett *et al.*, 1996) e nas páginas de manual do sistema.

```
if ((child_pid = fork()) == 0) {
    if ( mkdir_all (argv[1]) != 0)
            printf("Error: mkdir_all.\n");
    if ( bind_mount_all (sbroota) != 0)
            printf("Error: bind_mount.\n");
    if ( change_root (argc, argv, envp) != 0)
            printf("Error: change_root\n");
    exit(0);
}
```

A primeira chamada, mkdir_all, cria os diretórios. Esta função está implementada em secapp-helper-mkdir.c.

2.2 Criando os diretórios necessários

As funções que criam os diretórios necessários para o ambiente de execução do Secapp estão implementadas em secapp-helper-mkdir.c. A função mkdir_all recebe como argumento o caminho do executável, e chama a função mkdir_one para cada um dos diretórios definidos. Ao final, todos os diretórios necessários terão sido criados no caminho correto, no diretório do aplicativo dentro de .local/share/secapp.

```
int mkdir_one (char *path, char *dir);
```

mkdir_one recebe como argumentos o caminho do executável, e o nome de um diretório a criar. Assim, por exemplo, para criar .local/share/secapp/_usr_bin_editor/lib, receberá _usr_bin_editor e lib como argumentos. Para criar um diretório, a função faz a chamada de sistema:

```
mkdir (tmp_path, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

O primeiro argumento é o caminho completo do diretório a ser criado, e o segundo argumento são as opções, ou *flags*, usadas na criação do diretório.

O resto do código da função simplesmente manipula strings em C para que os caminhos corretos sejam formados para o primeiro argumento da chamada.

Criados os diretórios, a chamada da função secapp_open passa para a função bind_mount_all.

2.3 Montando os diretórios no ambiente

A função bind_mount_all é que de fato prepara os diretórios que acabamos de criar com os arquivos necessários, fazendo uma montagem vinculada de diretórios. A montagem vinculada nada mais é do que apontar o conteúdo de um diretório para outro; por exemplo, fazer com que .local/share/secapp/_usr_bin_editor/lib dentro do diretório pessoal do usuário exiba exatamente o mesmo conteúdo que /lib, um diretório de sistema da raiz do sistema de arquivos.

A função bind_mount_all, assim como mkdir_all, recebe como argumento o caminho do executável que está sendo rodado, que é usado para extrair o diretório dentro de .local/share/-secapp a ser usado. Por exemplo, para o executável /usr/bin/editor, os dados específicos desse aplicativo serão armazenados em .local/share/secapp/_usr_bin_editor. A partir daí, a função bind_mount é chamada:

```
int bind_mount (char *path, char *dir);
```

Ela recebe cada um dos diretórios de sistema que devem ser montados no segundo argumento, e o caminho do executável do aplicativo no primeiro argumento.

A função bind_mount faz a seguinte chamada de sistema:

```
mount (dir, tmp_path, NULL, MS_MGC_VAL|MS_BIND, NULL) != 0)
```

O primeiro argumento é a fonte do comando mount, o diretório a ser apontado pela montagem vinculada. O segundo argumento é o destino e diretório que de fato será montado dentro do diretório do Secapp. O terceiro e quinto parâmetros indicam respectivamente sistema de arquivos e dados, e nós definimos como NULL, enquanto o quarto parâmetro são as opções usadas no mount. A opção MS_BIND indica que o ponto de montagem é do tipo vinculado, enquanto MS_MGC_VAL é passado para o mount não assumir que o argumento é nulo.

bind_mount_all irá montar todos os diretórios de sistema com montagem vinculada. É importante frisar que o diretório /home não é montado desta forma; por exemplo, enquanto .local/share/secapp/_usr_bin_editor/lib está exibindo o conteúdo de /lib, o diretório .local/share/secapp/_usr_bin_editor/home está exibindo o conteúdo real deste diretório, e aí ficam armazenados os arquivos do usuário. Cada aplicativo iniciado com o Secapp terá um diretório pessoal de arquivos único.

Quando o bind_mount_all termina a montagem vinculada dos diretórios do sistema, o controle do programa volta para a função secapp_open, que agora irá chamar a função change_root.

2.4 Iniciando aplicativos

Após criados e preparados os diretórios do ambiente especial de execução de aplicativo com o Secapp, a função change_root é chamada para mudar a raiz de trabalho para esse ambiente, e então executar o aplicativo. O código para isso faz quatro chamadas de sistema ao kernel:

```
if (    (m="chdir" ,rc=chdir(saroot)) == 0
     && (m="chroot",rc=chroot(saroot)) == 0
     && (m="setuid",rc=setuid(getuid())) == 0
   )
   m="execve", execve(argv[1],argv+2,envp);
```

O trecho de código acima altera o diretório de trabalho para o diretório do ambiente especial, faz a chamada ao *chroot* para este diretório e define o id do usuário corretamente. Feito isto o programa finalmente é executado, fazendo a chamada de sistema *execve*.

O *chroot* é uma função que altera a raiz de diretórios vista por um processo. Como alteramos a raiz antes de iniciar a execução do aplicativo, de fato o que será enxergado por ele como raiz serão os arquivos do ambiente especial de execução.

A troca da raiz de diretórios é fundamental para o funcionamento do Secapp. Os aplicativos irão enxergar apenas os arquivos disponibilizados dentro da nova raiz, que é o nosso ambiente de execução. Enquanto os diretórios do sistema são replicados, o diretório pessoal do usuário é único para cada aplicativo executado.

Com a chamada para a função *execve* o aplicativo desejado será executado em um novo processo. O processo do Secapp que iniciou o aplicativo continua seu fluxo de execução. Ele avisa o secappd que um aplicativo foi iniciado, algo que será explicado em mais detalhes na próxima seção, e aguarda o processo criado encerrar para fazer a limpeza da execução, desmontando os diretórios que foram montados de forma vinculada. O trecho de código que faz isso:

```
    wpid =  wait(&status);
        overlay_umount(argv[1]);
        snd_finishedProcess_msg(*pid);
        bind_umount_all (saroot);
```

Além de desmontar os diretórios vinculados, é desmontando o diretório de junção do *overlayfs* e o secappd é avisado que o processo terminou. A junção de diretórios e o funcionamento do secappd serão explicados no próximo capítulo.

Capítulo 3

Concessão de acesso

O Secapp isola cada aplicativo em seu ambiente de execução, com seu próprio diretório de dados. Mas e quando o usuário deseja, por exemplo, enviar pelo aplicativo de e-mail um documento de texto criado por outro aplicativo? De alguma forma é preciso permitir que programas acessem arquivos criados por outros programas em situações especiais. Para esses casos o Secapp permite que aplicativos peçam permissão de leitura aos arquivos pessoais do usuário. Nesta seção, iremos descrever como isso acontece e como foi implementado.

Com o Secapp, em toda sessão do usuário é executado um *daemon*, chamado secappd, com o qual aplicativos podem se comunicar via D-Bus. O secappd é executado no ambiente normal do usuário, e portanto com acesso a todos os seus arquivos pessoais. Ele é responsável por atender requisições de aplicativos rodando com Secapp.

Quando o secappd recebe uma requisição de concessão de acesso de um aplicativo ele exibe um diálogo para o usuário. Caso o usuário aceite liberar o acesso a seus arquivos pessoais, o secappd faz uma junção entre o diretório pessoal real e o diretório pessoal visto por aquele aplicativo. Desta forma, o aplicativo continua sua execução normalmente, mas agora passa a enxergar a estrutura real e completa do diretório pessoal.

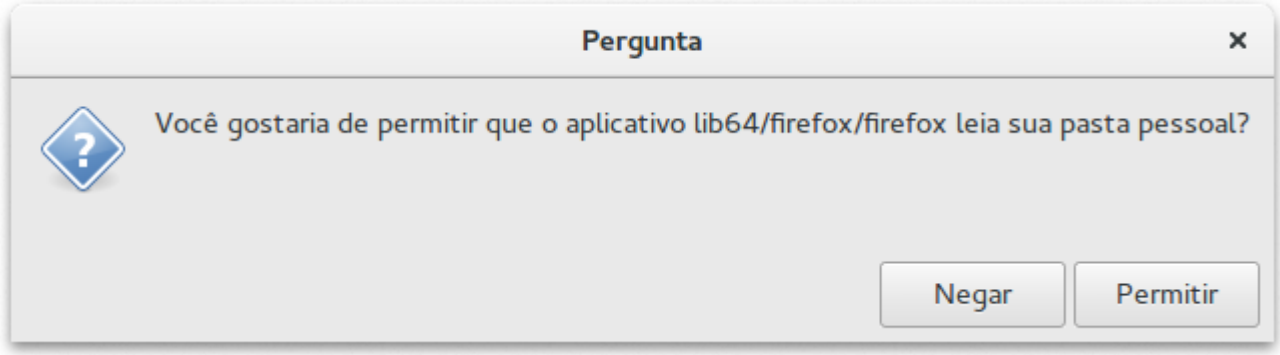

Figura 3.1: *Diálogo de autorização de acesso*

O processo onde um aplicativo pede permissão de acesso ao secappd, e este interage com o usuário, está ilustrado abaixo. Os arquivos são disponibilizados imediatamente e de forma transparente ao aplicativo em execução.

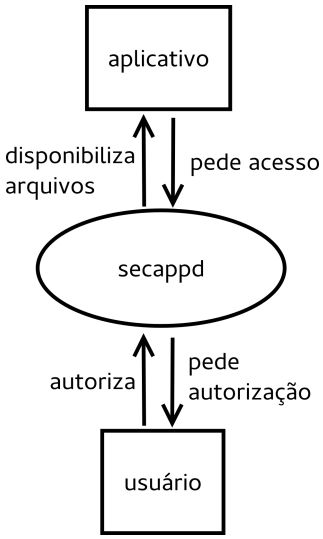

Figura 3.2: *Funcionamento do secappd. Um aplicativo pede permissão ao secappd, que então interage com o usuário. Caso o usuário tenha permitido, o secappd disponibiliza os arquivos para o aplicativo.*

O funcionamento e implementação do secappd serão detalhados a seguir. Antes, vejamos como funciona o principal framework utilizado nesse processo, o D-Bus.

3.1 Comunicação interprocessos com D-Bus

D-Bus é o framework de comunicação entre processos padrão do Linux. O Secapp utiliza o D-Bus para permitir que aplicativos rodando em um ambiente isolado possam, através do envio de mensagens, pedir permissão de acesso aos arquivos reais do usuário quando necessário. Mais informações sobre a arquitetura e funcionamento do projeto D-Bus podem ser vistas no *D-Bus Tutorial* (Pennington *et al.*, 2006).

O D-Bus roda em dois níveis diferentes: o *daemon* de sistema e o de sessão. O *daemon* de sistema permite que eventos do sistema sejam recebidos por todos os usuários, como por exemplo a detecção de uma nova impressora que foi encontrada. O *daemon* de sessão serve para a comunicação entre aplicativos iniciados por um usuário. O SecApp utiliza apenas o *daemon* de sessão.

Aplicativos podem definir o que o D-Bus chama de caminho de objeto, que é um nome a ser referenciado por outros programas que desejam se comunicar com ele. No secappd, o caminho de objeto usado é */org/efg/secapp*.

Cada objeto definido no D-Bus pode ter métodos e sinais. Os métodos são operações que podem ser invocadas em um objeto; por exemplo, o secappd define um método para conceder permissão de acesso a um aplicativo, recebendo seu nome como parâmetro. Quando um programa chama um método do D-Bus o programa que definiu o objeto daquele método faz a ação correspondente.

O D-Bus também permite que os objetos definam sinais, também definidos por quem criou aquele objeto. Programas podem se registrar para escutar um sinal, e quando ele é ativado, uma

função no programa que está escutando é chamada. O secappd usa sinais do D-Bus para se comunicar com o aplicativo que será visto mais tarde, o gerenciador do Secapp. O gerenciador irá escutar por um sinal, e sempre que o secappd fizer certas ações, irá avisar o gerenciador através desse sinal.

O secappd é um programa escrito na linguagem de programação Vala que cria um objeto no D-Bus conectado ao *daemon* de sessão, registra métodos e sinais e define as funções que atendem essas requisições. Na próxima seção iremos detalhar os métodos e sinais definidos, e sua implementação.

A linguagem de programação *Vala* (Lethalman, 2015) tem sintaxe similar a linguagens como Java e C#, mas o seu compilador gera programas em C, que então são compilados pelo compilador C disponível. Vala foi criada especialmente para o uso de aplicativos gráficos em Linux, e por isso se adapta bem a recursos nativos deste sistema operacional como D-Bus, o sistema de objetos GObject e a biblioteca gráfica GTK+. As funções de bibliotecas disponíveis em Vala são geradas automaticamente a partir de suas implementações nativas em C, garantindo que a linguagem se mantém atualizada e compatível com versões recentes.

3.2 secappd

O secappd disponibiliza métodos e sinais para comunicação com outros aplicativos. Para criar um objeto no D-Bus que atenderá requisições de outros processos, usando Vala, basta criar uma classe com o seguinte formato:

```
[DBus (name = "org.efg.Secapp")]
public class SecappServer : Object {


}
```

Os métodos dessa classe automaticamente serão disponibilizados como métodos no D-Bus, ou seja, eles podem ser chamados por outros processos. A seguir vamos ver quais métodos estão disponibilizados pelo secappd.

```
    public int startedProcess (int pid) {
        string path = "";
        get_path_from_pid (pid, path);
        table.insert(pid.to_string(), (string) path);
        updatedProcessTable();
        return 0;
    }
```

O método startedProcess é enviado por outro processo para indicar que um novo aplicativo está sendo executado com o Secapp. O secappd mantém uma lista de processos em execução

para melhor comunicar-se com o gerenciador do Secapp, que será descrito no próximo capítulo.

A primeira linha do método declara uma string, que será usada para guardar o caminho do processo iniciado. A função get_path_from_pid é escrita em C, sendo aqui chamada de forma transparente em Vala, e obtém o caminho do executável a partir do pid, ou seja, o identificador do processo. A função faz isso consultando as informações do processo pelo seu pid, que o kernel Linux disponibiliza em /proc. Com o path e o pid conhecidos, a função os insere na tabela de processos em execução, e chama o updatedProcessTable, que nada mais é do que o sinal que indica uma atualização na tabela. Este sinal estará sendo escutado pelo gerenciador do Secapp. A implementação em Vala de sinais na classe do objeto D-Bus é também muito simples, com a linha de código:

```
    public signal void updatedProcessTable ();
```

Basta fazer a declaração. Quando um outro método da classe chamar este sinal, a comunicação pelo D-Bus será feita a todos os processo que estejam escutando por este sinal.

Além do método startedProcess, existe o finishedProcess, que comunica que um processo foi finalizado.

```
    public int finishedProcess (int pid) {
      string spid;
      spid = pid.to_string();
      table.remove(spid);
      updatedProcessTable();
      return 0;
    }
```

O método recebe o pid do processo que terminou e o remove da lista. Depois ativa o sinal para comunicar que a tabela foi atualizada.

O principal método do secappd atende requisições feitas por programas que desejam autorização para acessar os arquivos do usuário. Vejamos como isso funciona:

```
int requestPermissionByPath (string command){
    int answer = ask_user_by_path(command);
    if (answer){
        secapp_ovl(command);
    }
    return answer;
}
```

O método recebe como argumento o comando que deseja permissão. A função ask_user_by_path cria um diálogo no ambiente gráfico perguntando ao usuário se deseja conceder permissão de

acesso; caso a resposta seja afirmativa, o secappd chama a função secapp_ovl, que faz uma junção entre o diretório pessoal real e o diretório específico do aplicativo. O funcionamento dessa junção será descrito na próxima seção.

Além de receber o pedido de permissão pelo caminho do executável, o secappd também pode recebê-lo pelo pid do processo em execução:

```
int RequestPermissionByPid(int pid){
    int answer = ask_user_by_pid(pid);
    if (answer == 0){
      secapp_ovl_by_pid (pid);
    }
    return answer;
}
```

O funcionamento é análogo. A função descobre o caminho do executável pelo sistema /proc do kernel Linux e pede autorização ao usuário.

Além desses métodos o secappd também pode oferecer a tabela de processos que estão sendo executados com o Secapp através do método abaixo, que é chamado pelo gerenciador do Secapp:

```
    public HashTable<string, string> requestTable (){
        return table;
    }
```

Além da classe com métodos e sinais definidos acima, o secappd tem a seguinte função main, que inicia a execução para escutar requisições do D-Bus:

```
void main (string[] args) {
    Bus.own_name (BusType.SESSION, "org.efg.Secapp", BusNameOwnerFlags.
       NONE,
                  on_bus_aquired,
                  () => {},
                  () => stderr.printf ("Could not aquire name\n"));

    new MainLoop ().run ();
}
```

O programa simplesmente adquire o domínio de D-Bus que definimos e entra no laço de execução, pronto para atender requisições de outros processos.

3.3 Fazendo a junção entre diretórios

Quando um aplicativo pede permissão de acesso ao diretório pessoal real, e o usuário concede a permissão, cabe ao secappd disponibilizar os arquivos. Isso é feito através do overlayfs

do kernel Linux.

O overlayfs permite que dois diretórios diferentes do sistema de arquivos sejam montados em um mesmo diretório. A ideia é colocar um diretório em cima do outro, em níveis. Caso arquivos com mesmo nome existam, o arquivo exibido será o que está no nível superior. A escrita também será sempre no nível superior.

Com o Secapp, o diretório real /home será o nível inferior da junção. Assim, os arquivos reais serão disponibilizados para leitura. Qualquer escrita, mesmo após a junção, será feita no diretório pessoal do aplicativo, por exemplo .local/share/secapp/_usr_bin_editor/home/alice.

As chamadas que fazem a junção de diretórios estão implementadas em secapp-helper-overlay.c. A função secapp_ovl recebe como argumento o nome do executável que está rodando no Secapp e faz a seguinte chamada de sistema para criar a junção:

```
mount("overlay", ovl_path, "overlay", MS_MGC_VAL, ovl_arg);
```

O primeiro, terceiro e quarto argumentos especificam que o tipo do sistema de arquivos a montar é um overlay.

O ovl_path é o diretório onde será montada a junção, ou seja, .local/share/secapp/Aplicativo/home. O ovl_arg é o argumento de montagem que especifica os níveis da junção: o diretório de baixo, o de cima, e um diretório de trabalho necessário para o overlayfs. No nosso caso, o diretório de cima é o mesmo diretório onde a junção será montada, que é o diretório pessoal específico do aplicativo, e o diretório do nível de baixo é o diretório /home real no sistema.

Quando o secappd fizer a junção, o aplicativo terá acesso imediato aos arquivos do diretório pessoal. Os arquivos estarão disponíveis até o fim de sua execução.

Capítulo 4

Requisitando acesso

Aplicativos pedem permissão de acesso aos arquivos reais do usuário através do envio de uma mensagem ao secappd, via D-Bus. Porém o Secapp está sendo desenvolvido agora, e os aplicativos não oferecem suporte a esse tipo de mensagem. Para contornar esse problema foram desenvolvidos dois aplicativos que fazem parte do Secapp: o secapp-reqperm, e o gerenciador de processos do Secapp, o secapp-manager.

4.1 Permissão pela linha de comando: secapp-reqperm

O secapp-reqperm é um programa da linha de comando que recebe como argumento o identificador de um processo, o seu pid, ou o seu caminho na linha de comando, e envia a mensagem ao secappd de requisição de acesso de arquivos. O secappd então trata a mensagem normalmente, perguntando ao usuário se deve conceder acesso e, em caso afirmativo, disponibilizando os arquivos.

A sintaxe do secapp-reqperm é simples:

```
secapp-reqperm /usr/bin/editor
```

Ao invés do caminho do executável também pode ser usado o pid do processo:

```
secapp-reqperm 955
```

O secapp-reqperm foi escrito na linguagem Vala, e seu código fonte está implementado no arquivo secapp-reqperm.vala. O código é simples, e está apenas na função main:

```
void main (string [] args) {
    Secapp sapp = null;
    int arg;

    if (args.length == 1)
        Process.exit(0);

    try {
```

```
        sapp = Bus.get_proxy_sync (BusType.SESSION, "org.efg.Secapp",
                                                    "/org/efg/secapp");
        arg = int.parse(args[1]);
        if (arg == 0)
            sapp.requestPermissionFromPath (args[1]);
        else
            sapp.requestPermissionFromPid (arg);

    } catch (IOError e) {
        stderr.printf ("%s\n", e.message);
    }
}
```

Os métodos chamados, requestPermissionFromPath e requestPermissionFromPid, são métodos de um objeto D-Bus. Isso deve ser declarado no início do arquivo de código fonte:

```
[DBus (name = "org.efg.Secapp")]
interface Secapp : Object {
    public abstract int requestPermissionFromPath (string path) throws
       IOError;
    public abstract int requestPermissionFromPid (int pid) throws IOError;
}
```

Com apenas este código, o programa envia as mensagens definidas para o secappd e torna possível conceder permissões de acesso a qualquer aplicativo.

4.2 Gerenciador do Secapp

Além do secapp-reqperm, foi desenvolvido neste trabalho o Gerenciador do Secapp. O gerenciador é um aplicativo gráfico que lista todos os processos em execução com o Secapp e oferece ao usuário a opção de autorizar acesso aos arquivos pessoais.

Tanto o secapp-reqperm quanto o secapp-manager simplesmente enviam uma mensagem via D-Bus ao secappd. Assim, o usuário ainda precisará confirmar a requisição. Por questões de segurança, não existe um método que permita a concessão de acesso sem interação do usuário, visto que isso poderia levar a abusos por aplicativos para ganhar acesso sem permissão.

O secapp-manager foi escrito em Vala, usando a biblioteca de desenvolvimento de aplicativos gráficos *GTK+* (Mattis *et al.*, 2015).

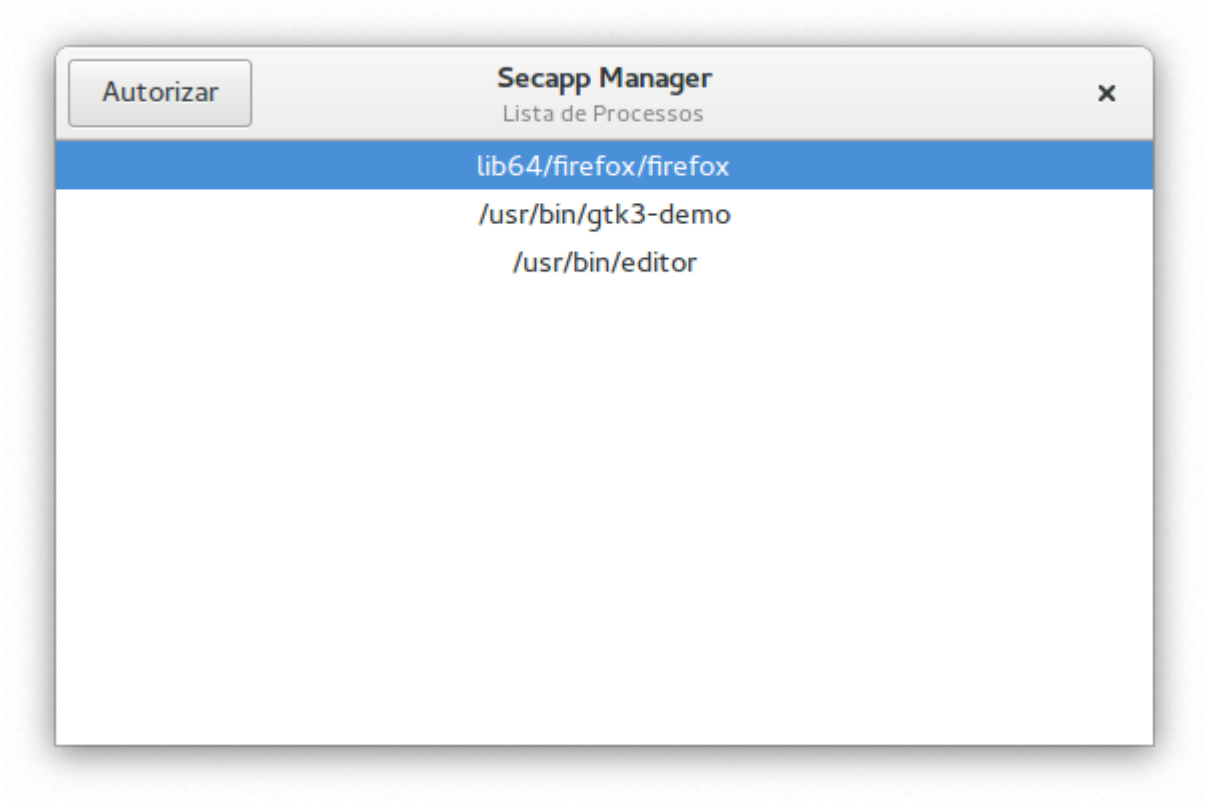

Figura 4.1: *Gerenciador do Secapp*

4.3 Editor de Texto

Para demonstrar como um aplicativo normal poderia fazer a requisição de acesso aos arquivos do Secapp, foi desenvolvido um editor de textos que pode pedir acesso ao Secapp.

O código do editor de textos em si foi retirado de um dos exemplos da *documentação do ambiente gráfico* Gnome (2015). As modificações feitas consistiram em adicionar uma entrada no menu da aplicação para chamar o Secapp, e fazer a implementação do método no código fonte.

A opção no menu é adicionada com o seguinte trecho na definição da interface do Editor, no arquivo editor.ui:

```
    <section>
      <item>
        <attribute name="label">Secapp</attribute>
        <attribute name="action">win.secapp</attribute>
      </item>
    </section>
```

A chamada de função definida no menu é implementada no código, em editor.vala, com o seguinte método:

```
int secapp_work(int pid){
    Secapp sapp = null;
```

```
    try {
        sapp = Bus.get_proxy_sync (BusType.SESSION, "org.efg.Secapp",
                                                    "/org/efg/secapp");

        sapp.requestPermissionFromPid (pid);

    } catch (IOError e) {
        stderr.printf ("%s\n", e.message);
    }
  return 0;
}
```

Com apenas estas simples inclusões, o editor de textos passa a incluir uma opção em seu menu para ter autorização de acesso aos arquivos.

Capítulo 5

Conclusões

O Secapp foi desenvolvido com sucesso, e já está disponível e funcionando para instalação em sistemas Linux. Os seguintes componentes integram o pacote:

- secappd, iniciado em cada sessão do usuário.
- secapp-run, comando usado para iniciar aplicativos.
- secapp-reqperm, comando usado para conceder permissão de acesso.
- secapp-manager, gerenciador que lista aplicativos rodando com o Secapp.
- editor, um simples editor de textos integrado com o Secapp.

Os objetivos foram totalmente alcançados. Com o Secapp, é possível rodar aplicativos de modo que cada um tem seu próprio diretório para armazenar arquivos, e acesso restrito a apenas este diretório. Uma falha de segurança em um aplicativo tem efeitos muito reduzidos quando este é usado em conjunto com o Secapp.

Ainda assim, o sistema completo oferece alguns desafios a serem solucionados. Como o funcionamento do Secapp é transparente a aplicativos, também não fica claro ao usuário que o aplicativo está rodando com acesso restrito. Assim, rodar todos os aplicativos por padrão com o Secapp pode não ser uma boa ideia, ao invés disso os próprios usuários devem ser educados sobre sua existência e optar por utilizá-lo. Essa questão pode ser contornada com a distribuição e divulgação do Secapp, a medida que integrá-lo a um aplicativo é muito simples, como foi demonstrado no caso de uso incluído no pacote.

Como perspectivas para o futuro, além da melhor integração com aplicativos, o sistema de concessão e requisição de permissões do Secapp também poderia ser estendido para permitir o isolamento de outros tipos de serviço, não apenas arquivos do usuário. Por exemplo, poderíamos restringir o acesso a internet e o acesso a dispositivos físicos como microfone e câmera, uma vez que também podem ser explorados por um atacante para violar a privacidade e integridade dos dados do usuário.

Sempre existirão desafios para proteger a privacidade e integridade de usuários de computadores, e esperamos que o Secapp seja um bom passo nessa direção.

Referências Bibliográficas

Burkett *et al.*(1996) B. Scott Burkett, Sven Goldt, John D. Harper, Sven van der Meer e Matt Welsh. Linux programmer's guide. http://www.tldp.org/guides.html, 1996. Citado na pág. 4

Gnome(2015) Projeto Gnome. Exemplo de editor de textos escrito em vala. https://developer.gnome.org/gnome-devel-demos/unstable/textview.vala.html.en, 2015. Citado na pág. 17

Lethalman(2015) Luca Bruno Lethalman. Vala reference manual. https://wiki.gnome.org/Projects/Vala/Manual, 2015. Citado na pág. 11

Mattis *et al.*(2015) Peter Mattis, Spencer Kimball e Josh MacDonald. Gtk+ 3 reference manual. https://developer.gnome.org/gtk3/stable/, 2015. Citado na pág. 16

Pennington *et al.*(2006) Havoc Pennington, David Wheeler, John Palmieri e Colin Walters. D-bus tutorial. http://dbus.freedesktop.org/doc/dbus-tutorial.html, 2006. Citado na pág. 10