source: https://flinnthebin.github.io/ddlogger/

ddlogger

Introduction

dlogger is my Something Awesome Project, focusing on Linux systems-level security.

Background

Using Linux as my daily driver means I have many fun challenges to solve if I want to actually use my computer for any "normal" tasks. My speakers work about half the time, depending on whether I can get PulseAudio working. My AirPods work 100% of the time, as long as I call the script in the right state.

AirPods

Nice. See how, right at the end of the video, my AirPods script in the Bluetooth controller (bluetoothctl) leaks the output:

```
[NEW] Transport /org/bluez/hci0/dev_B0_3F_64_21_7E_D7/sep1/fd7
```

This is a file descriptor, which is just an integer handle to some filepath that allows for the reading/writing of information via the protocol for that particular hardware. For bluetooth in this case, it is allocated to the audio transport stream between my airpods and my computer, allowing me to pass further commands from bluetoothctl to my airpods by passing the file descriptor as a parameter to a system call, along with further protocol parameters.

All I know about the bluetooth protocol is that the Generic Access Profile is usually aggressive on connections because users are not typically engineers and so the protocol uses randomized MAC addresses to obscure unauthenticated connections from non-paired devices. The standard is awful to parse.

Fan

I don't have access to the Alienware Command Center, so I created my own fan controller and manager. This was also achieved using file descriptors and deciphering the protocol (along with the parameters) that would enable me to change settings of the fan through the i8k kernel module. Essentially, to change the fan speed controller we just use the ioctl syscall, with the parameters (i8k_file_descriptor, fan_set_fan_offset, args_list)

Implementation Details

Similarly, ddlogger relies on finding the correct file descriptor for the systems keyboard input, then simply executes a non-binding grab of that stream of data. This captures all the input that the computer receives from the keyboard without any noticeable change in behaviour of the keyboard to the user.

Overview

I like to think of this program as a videogame, or sort of like an operating system. The main function spins up on the main thread, which will just spin forever unless it receieves a signal interrupt. There are 3 singleton entities that exist in our game loop, each with distinct behaviours. Singletons were selected for their thread-safety and compatibility with the design principle of 'locality of behaviour'. Locality of behaviour basically just means each entity does one thing and only one thing. This just makes debugging easier and keeps the code more modular. It could all be done in one monolith class but I don't like to hold all that context in my head.

Procloader

The procloader entity only has two jobs, then falls out of scope and is destroyed. The first job of the procloader is to create a usergroup called 'input', which has permission to read from any /dev/input/ directories. This creates persistence in the system and should mean that any subsequent use of ddlogger does not require root privileges. The second job of the procloader is to create a cronjob that will run itself on boot, so that there is no requirement to call ddlogger on future system startups.

Logger

The logger entity initializes by using cURL to download the keymap.json file that maps to my keyboard from the github repo of this project. This design decision was made so that ddlogger could be compiled in full but then have the keyboard codes dynamically allocated based on the requirements of the application, without the need to recompile the binary. The entity is spun up and then starts reading 24 byte structs from /dev/input/eventX. If suitable structs are found that map to linux keypress event structs, the interpreted data is then converted into a JSON-serializable struct and added to a threadsafe queue (which is a dependency of the logger during construction).

Sender

Like logger, sender is constructed using a dependency injection of the threadsafe queue. Because singletons are threadsafe by design in C++ and the logger/sender use dependency injection during construction, I never ran into any issues with

entities operating on different threads accessing a shared data structure. This was really fortunate. The sender waits until an event struct is in the queue, pops it off the queue, serializes the data into JSON and sends the JSON to a remote server (in this case, 127.0.0.1:8080).

Messages

Messages is a way of logging output to the terminal that facilitates debugging. I tend to agree with Casey Muratori when it comes to projects that require live user input - test-driven development doesn't work. Fuzzing randomized user input and determining controlled, deliberate user input in order to unit test a project is extremely hard. For this reason, we just set a global variable messages to a level (none, info, error, warning, debug) and at compilation time the output will be as desired.

Critical Analysis

Weaknesses

- main.cpp: 32 - this while loop could persist indefinitely, lazy design
- logger.cpp - there is no resilience built in for the aware user. My computer alternates between event9 and event13 on one USB port, If a user unplugged their keyboard logger wouldn't time out and find the new keyboard.
- sender.cpp - if the queue is empty for a long period of time, sender does nothing to ensure that there isn't an issue.
- main.cpp - because all the work is done on the logger and sender work threads, main just functions as a keepalive. this is lazy, main should be managing the activity of the entities or a manager entity should be built that does more to make the program robust and resilient against system change

Strengths

- Entity design - designing the 3 main entities around threadsafe locality of behaviour paid off. the program is relatively stable and even in the face of the weaknesses listed, performs pretty well.
- Whitelist/blacklist - the mousetrap is a very fun thing to have in my program and required only little modification to the keymap.json and some logic to make it work. the mousetrap inspired the program flow of having a whitelist->unlisted/ignore blacklist search of the /dev/input drive for viable eventX candidate devices.
- Good oop - I think I've used the singleton pattern well in a context where thread-safety would be important, but I would appreciate any feedback on that, too.

Conclusion

This assignment was really fun, because we get to pick our own scope. While it is quite a different project than my original proposal, I quickly realised that chaining bash tools togther by piping output from dd into a file out.swp, then sending that file through mutt to an email address was risky. Any single failure of one of those utilities destroyed the keylogger. With a compiled binary, that issue will likely only occur on startup. I think the biggest strength of this project is the robustness of the design. Because it is modular and adheres to the principle of locality of behaviour, debugging is very straightforward. There isn't really any complex logic or data structures that I had to dream up, all of that is handled in the linux header files linux/input.h & linux/input-event-codes.h

ddlogger

The next stage would be to write a decoder, that could interpret the JSON into a formatted text document.