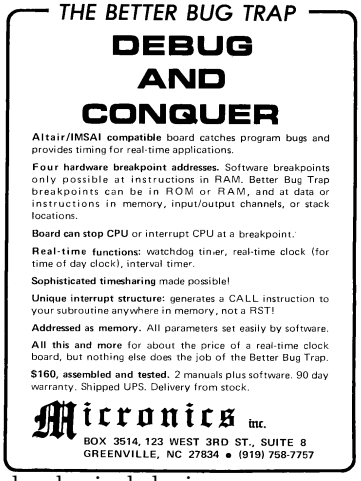
5 A Quick Introduction to the New Facedancer Framework

by gil

Recently, I rewrote the Facedancer software stack with the goal of making it easier to write new emulators for both well-behaved and poorly-behaved devices. In this post I'm going to give an introduction to doing both. I assume you've got a Facedancer board, python3, the pyserial library, and a current revision of the code. I'll start with a very brief overview of the USB protocol itself, then show how to modify the existing USB keyboard emulator code to emulate a different (yet still well-behaved) device, and finally show how to take a well-behaved device and make it misbehave in specific ways.



5.1 USB

The USB protocol defines a bunch of abstractions: Devices, Configurations, Interfaces, and Endpoints. Some of these terms are a bit counterintuitive, understanding of which is not at all aided by how they're referred to by users.

A Device is a physical thing that gets plugged into a USB port. A single physical device may present itself to the operating system as multiple logical devices (think a keyboard with built-in trackpad or one of those annoying USB sticks that pretends it's both a USB mass storage device and a USB CD-ROM so it can install adware). In USB parlance, each of the logical devices is not a Device, but rather an Interface. I'll get to those in a couple paragraphs.

When a device is connected to a host, the host begins the enumeration process, in which it requests and the device responds with a bunch of descriptors that describe how the device can and/or wants to behave. The device presents to the host a set of "configurations"; the host chooses exactly one of these and the device, er, configures itself accordingly. But what's a configuration? It's a set of interfaces!

An Interface is a single logical device as mentioned above: a keyboard XOR a trackpad XOR an external hard drive XOR an external CD-ROM XOR... From the perspective of writing software emulators for these things, this architecture is actually kinda helpful: we can write a single interface implementing a keyboard and then include it in various device implementations. Code reuse FTW.

Each interface contains multiple "endpoints," which are the actual communication channels to and from the host. Only one endpoint is required: endpoint 0 (EP0) is the bidirectional "control" endpoint, used for exchange of descriptors on connection and optionally for asynchronous communication thereafter. (The various ways a device and host can communicate are beyond the scope of this post and, considering the tendency of device manufacturers to fabricate their own protocols to run over USB, probably intractable to cover in any single document. Your best bet to gain understanding are either fuzz it or read the device driver code.)

Endpoints other than EP0 are unidirectional so, in the case of something like an external hard drive that needs to both send and receive large amounts of data, the interface will define two endpoints: one for host-to-device ("OUT") transfers and another for device-to-host ("IN") transfers.

Lastly, the USB protocol (up to and including USB 2.0) is "speak when spoken to": all device communication is initiated by the host, which means even more state machines and callbacks than you might have been expecting.

With that, let's go to the code.

5.2 A Simple Device

All of the source files are in the "client" subdirectory of the SVN tree. You can tell the new stuff from the old:

1. The old libraries are named `GoodFET*`.
2. The old programs are named `goodfet.*`.
3. The new libraries are named `USB*` (plus `MAXUSBApp.py`, `Facedancer.py`, and `util.py`.)
4. The new programs are named `facedancer-*`.

Start by looking at `facedancer-keyboard.py`. It's pretty simple: we import some stuff, open a connection to the serial port, say we want to talk to a `Facedancer` on the serial port, then we want to talk to the `MAXUSBApp` on the `Facedancer`, and we hand this to an instance of the `USBKeyboardDevice` class, which connects the emulated device to the victim and we're off to the races. Easy enough.

The good news here is that you shouldn't have to ever worry about what goes on in the `Facedancer` and `MAXUSBApp` classes; the entirety of the logic specific to any given USB device is contained with the `USBDevice` class, of which (in this case) `USBKeyboardDevice` is a subclass. To create your own device, just create a new class that inherits from `USBDevice` and customize it as you see fit. As an example, look at `USBKeyboardDevice.py` for the implementation of the `USBKeyboardDevice` class.

Way at the bottom of `USBKeyboardDevice.py`, you'll find the definition for the `USBKeyboardDevice` class. It's fairly short: we define a single configuration (notice the configurations are numbered from 1) that contains a single interface, then we send that configuration on to the superclass initializer along with a bunch of magic numbers. These magic numbers are primarily used by the host operating system to figure out which driver to use with the attached device. From the Facedancer side, however, the keyboard functionality is implemented in the `USBKeyboardInterface` class, which takes up most of the file. Scroll back up to the top and look at that now.

The `hid_descriptor` and `report_descriptor` are hard-coded as opaque binary data specific to HID devices (I may abstract away their details at some point, but it's not a particularly high priority). In `__init__`, there's a dictionary mapping descriptor ID numbers to the actual descriptor data, which is sent to the superclass initializer (I'll get into more detail on this in the section on misbehaving devices). Also in `__init__`, a single `USBEndpoint` is instantiated, which includes a callback (`self.handle_buffer_available`).

Remember that the device never initiates a data transfer: the host will ask the device if it has any data ready; if it doesn't, the device (in our case, the MAX3420 USB chip on the Facedancer board itself) will respond with a NAK; if it *does* have data ready, the device will send the data on up. Thus whenever the host asks for data for this particular endpoint, the callback will be invoked. ("Whenever" is a bit misleading because the host will likely send polls faster than we can deal with them, but it's close enough for the time being.)

The `handle_buffer_available` method calls `type_letter`, which sends the keypress over the endpoint. (This abstraction as it stands right now is messy and is high on my list to fix—the `USBEndpoint` class should have "send" and "receive" methods, rather than having to climb up through the abstraction layers to the `send_on_endpoint` call currently in `type_letter`.)

To make a very long story short, writing an emulator for a new device should be straightforward:

1. Subclass `USBInterface` (eg, as `MyNewInterface`), define your set of endpoints and pass them to the superclass initializer, and define endpoint handler functions.
2. Subclass `USBDevice` (eg, as `MyNewDevice`), define a configuration containing `MyNewInterface`, and pass it along to the superclass initializer.

5.3 A Misbehaving Device

If you subclass `USBDevice` and `USBInterface` as described above, the rest of the class hierarchy should do the Right Thing (TM) with regards to the USB protocol itself and talking to the Facedancer to perform it: appropriate descriptors will be sent when requested by the host, correct callback functions will be called when endpoints are polled by the host, etc. But if you want to test how systems react in the face of devices that don't perform exactly as expected, you're going to have to dig in a bit.

The pattern I've tried to follow (though there are certainly deviations, which I intend to deal with—patches appreciated!) is for the `USBDevice` class to handle control messages over endpoint 0 and dispatch them to the appropriate instance of (subclasses of) `USBConfiguration`, `USBInterface`, or `USBEndpoint`. For example, if the host sends a GET_DESCRIPTOR request for the configuration, the request is dispatched to `USBConfiguration.get_descriptor`, which returns the data to be sent in response.

This logic is contained in the `USBDevice.handle_request` method; if you want your custom misbehaving device to do weird stuff for every incoming request, this is the method to override. If, on the other hand, you're looking to mess with just descriptors for a specific abstraction, you're better off overriding the `get_descriptor` method of the `USB*` classes. If you want to send non-standard responses to any of the other control messages (eg, CLEAR_FEATURE, GET_STATUS, etc), you should override the associated `handle_*_request` method of `USBDevice`. (Note that `USBDevice.handle_request` is the method that dispatched to the handle_*_request methods.)

Each of the top-level `USB*` classes (`USBDevice`, `USBConfiguration`, `USBInterface`, and `USBEndpoint`) has a `self.descriptors` member that maps from descriptor number to a descriptor or a function that returns a descriptor. Thus you are not constrained to hard-coding values, you can instead provide a function that creates whatever descriptor you want sent.

To make a somewhat less-long story short, modifying an emulated device to misbehave should be similarly straightforward.

1. Subclass whichever of `USBDevice`, `USBConfiguration`, `USBInterface`, or `USBEndpoint` contains the behavior you want to modify.
2. Override the `descriptor` dictionary in your subclass to change what descriptors get sent in response to requests.
3. Override the `handle_*_request` methods in your subclass of `USBDevice` to change how your device responds to individual requests.
4. Over the `USBDevice.handle_request` method to change how your device responds to *all* requests.

Happy fuzzing!