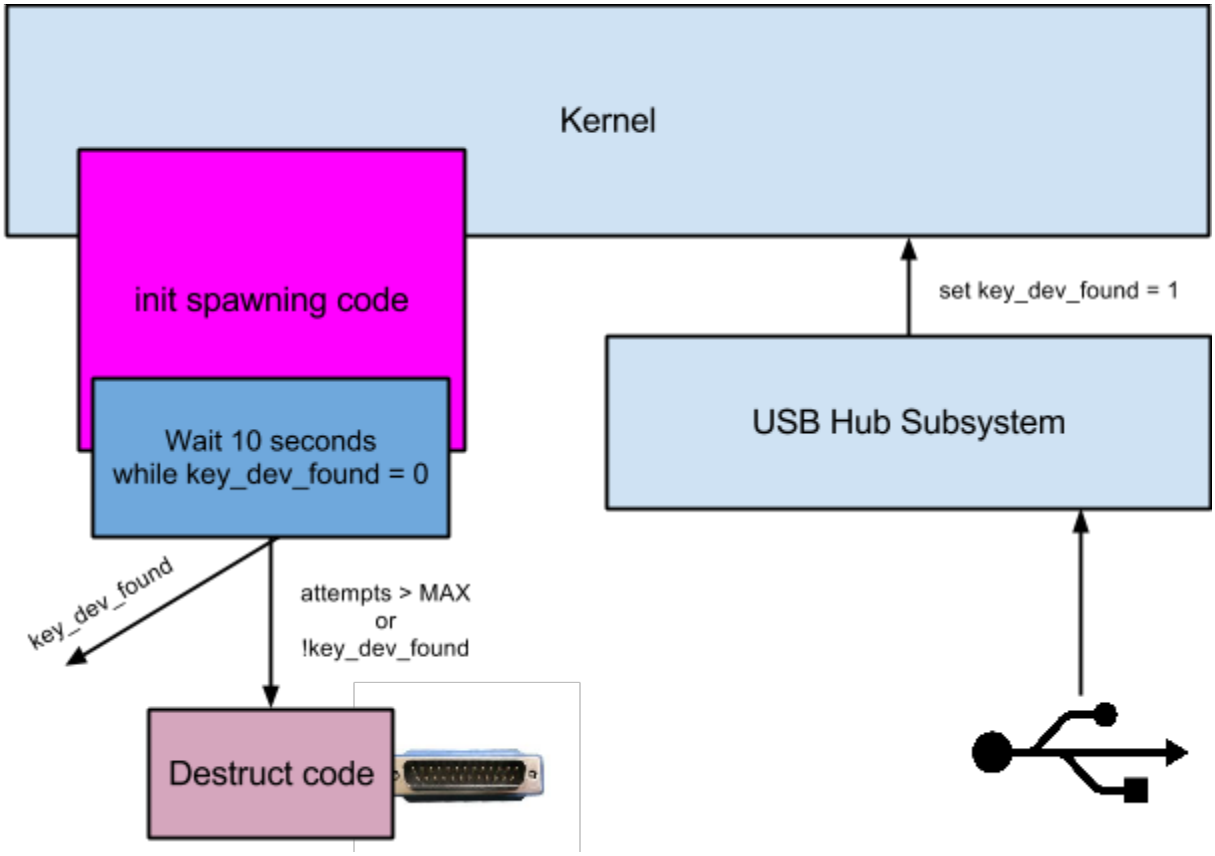
Self-destructing Hard Drive

Ben Buzbee, Michael Backherms, Martin Brown

Project Description

This project is an extension of Sean Easton and Bobby Roy's USB Boot Authentication project. A uniquely keyed USB device needs to be inserted into the computer in order for Linux to boot. The serial number of this USB device is hardcoded into the Linux kernel. At boot time, the system waits in a loop for the matching USB device to be inserted.



Our extension of this project was inspired by Mike Mitchell, a former Teaching Assistant for this class and a member of the OS group at Florida State University. It includes a countdown timer for entering the USB device. If this timer expires, or If the wrong device is inserted 3 times, we initiate the "self-destruct" sequence which starts another countdown timer. To abort the self-destruct sequence, we employ an additional device connected via the parallel port. This device can send signals to the port using 1) a big red button 2) a mailbox key on the left side and 3) another on the right side of the big red button. These three signals need to be in the on position in order to abort the self-destruct sequence before the self-destruct timer expires. Once

the timer expires, a self-destruct signal is sent to the device via the parallel port. We only send the signal, to simulate the destruction which is the responsibility of Mike MItchell.

Our project has evolved over the weeks. Initially, we were seeking to initiate the self-destruct sequence after counting x number of failed login attempts by the user. We discuss this further in the *Challenges* section of this paper.

Motivation

Hollywood is not short of movies with a phrase along the lines of "... will self-destruct in 3, 2, 1." In reality however, physically destroying a hard drive in an attempt to render it unreadable is not a common feature in hard drives from yesterday and today. We were also looking forward to the challenge of experimenting with what we can do in the Linux kernel before boot, when all of the subsystems have not yet been loaded. Moreover, a project like this could potentially be featured on hackaday.com.

Kernel Subsystems

Process management and device management work to some extent in these early stages of the boot process. We weren't able to launch our own processes however. And although we could see the arduino device listed in /dev/ as /dev/ttyACM0, we were not able to read from or write to the device.

Related Work

In addition to the work of Easton and Roy, we're aware of several other self-destructing drives. Toshiba released a version of their self-encrypting hard drives in 2000 which invalidates the drives security key when its power supply is turned off, rendering it indecipherable. Toshiba argues that erasing the data takes hours, and that destroying the device requires physical effort as well as time.

RunCore, a not-so-well-known maker of hard drives released a solid state drive which has two physical buttons. A red button destroys the drive physically by applying a strong current to the NAND flash memory. A green button overwrites the entire disk with random data, making the data unrecoverable.

Research in academia is more concerned with securely erasing the data from a drive. TrueErase by Diesburg et. al. try to mitigate several challenges that arise when one tries to erase a drive. They take into account that not all file systems are the same. TrueErase can erase individual files, as opposed to physical destruction techniques that try to destroy all or nothing.

Contributions

Our contributions of this project are as follows. We share our insight into

1. **User space versus kernel space authentication**

Authentication should be implemented in user-space, not kernel space. We discuss this further in the *Challenges* section.

2. **The Linux Pluggable Authentication Module (Linux PAM)**
   Although we didn't employ PAM, we learned how to use it during the course of this project. Linux already includes PAM, so instead of building and installing PAM from scratch, you should look into the PAM that is already in use by Linux. Installing a new one can corrupt authentication on your machine.
3. **Limitations of reading and writing to and from USB devices before they have been mounted**
   It's not possible to read from or write to a USB device in the early stages of the boot process. Although you may see the device listed, calling *sys_read()* or *filp_open()* on the device returns *errno ENOENT* (no such file or directory). You may find several articles showing you how to make this work, but to our knowledge this is impossible with USB devices at this stage of the boot process.

Challenges

Throughout the development lifecycle, we faced various challenges, both from a decision making standpoint and a physical implementation standpoint.

1. **Kernel space authentication**
   We researched into various methods of authentication, most of which, such as PAM, operated solely in user space. Within the context of the class, these were not adequate solutions. We then began looking into various methods of kernel space authentication, including encrypting the hard drive with dm-crypt, but ended up deciding that extending the USB Authentication project from the previous class was the most efficient solution.

   The USB authentication gave us the ability to feature a "something you have" solution to authentication, as a USB device's serial number becomes instantly readable by the USB hub subsystem upon insertion. While the device was not actually mounted yet, we could authenticate the user with the unique identifier, while also preventing a malicious user from running a programmable USB brute force operation. By exporting shared variables between our main boot process, we could keep track of all USB activity for our system, ensuring that our process operated as intended.

   In the end, however, we came to the conclusion that this process of "authentication, or else" would be better served in user space, as the PAM module could help support multiple users on a computer, while also allowing USB cryptographic methods to be added, such as device public/private key pairs, which would allow for safer, and possibly more complex authentication methods (adding in a "what you know" to the security procedures).
2. **Communication with the device**
   As briefly touched on in the previous section, at the point we stopped the boot process,

we had very limited access to the file system as a whole. When we first attempted to connect the self-destruct prevention device, we utilized an arduino board which communicated with the system via USB. As USB devices were not actually mounted yet, we could not actually communicate with the machine via standard kernel system calls. Our solution was to move from communication with the USB port to the parallel port instead. We decided that a device that operated on the signals directly from the parallel pins would be desirable, as there would be no need to actually mount the device, we could just send different signals to the port and let the device deal with the flipping of switches based on the signal received. This ended up being the correct model for device communication at the specific point we stopped the boot order.

3. **The physical device**

   This ended up becoming our biggest problem, actually dealing with the device. As we were not very adept at dealing with switchboards or physically transporting signals to device parts, we were completely reliant on the devices Mike Mitchell prepared for us. As mentioned in the previous challenge portion, we ended up changing from an Arduino based device to a switchboard that communicated with the parallel port due to our inability to access the USB mount point at our current progress through the boot process.

   The switchboard based device ended up proving to be difficult from a more physical aspect, as we could not send signals straight to the specific components of the device, due to the small amounts of power being sent through each pin. With the combination of a battery and transistor, we were able to send stronger signals to an LED light that was representing our “destruction device”, signifying that we had succeeded in starting the destruction process. When we attempted to create a more stable device (soldering the wires in place for stability, seal the device to conceal wires), the device began sending erratic signals and we were unable to recover a working device from the components. We still have a programmatic solution that works, should the device be repaired to work as before.