CATCHING THE QUEUE HOPPER: REAL-TIME ANOMALY DETECTION BY CODE TRACING

Tao Wang

Adviser: Professor Peter C. Johnson

A Thesis
Presented to the Faculty of the Computer Science Department of Middlebury College in Partial Fulfillment of the Requirements for the Degree of Bachelor of Arts

May 2018
ABSTRACT

Tracking kernel data flow is an important method to detect programs’ abnormal behaviors and catch malware in the act. Doing so is often difficult because of modern kernel design patterns, where, to optimize system performance, tasks are often enqueued and later dequeued by different processes to be handled asynchronously at different times. In this paper, I explore a new way to detect USB activities with potentially malicious intent. I used the FreeBSD kernel and the built-in DTrace software combined to retrieve the pattern of function invocation that the kernel undergoes when a specific USB request is made. I traced the functions that were called during the lifetime of basic USB CRUD operations [11] (create, read, update, delete), and formulated corresponding chains of expected invocations. I was able to discover requests that have abnormal behaviors by comparing the functions invoked by new requests and the expected invocation chain, a rather lightweight approach to monitor processes that create behavior anomalies, an approach that is extendable to other components of the kernel beyond USB.
ACKNOWLEDGEMENTS

I would like to thank Professor Johnson for guiding me through this project and providing valuable advice on conducting experiments and writing.

I would like to thank Professor Scharstein for his help throughout the semester on keeping on schedule.

I would also like to thank my fellow classmates for their feedback, support and cooperation.
# TABLE OF CONTENTS

1 Introduction ................................................. 1  
   1.1 Background ............................................. 1  
       1.1.1 The Problem ....................................... 2  
       1.1.2 Proposed Approaches ............................... 4  
   1.2 My Solutions ............................................ 6  
       1.2.1 FreeBSD & DTrace ................................ 7  
       1.2.2 Code Tracing ..................................... 7  

2 Hacking the Kernel ....................................... 10  
   2.1 Existing Infrastructure ................................. 11  
   2.2 Customized DTrace Probes ............................... 13  
   2.3 Coordinating between Job Queues ....................... 14  
   2.4 Identifying Original Devices ........................... 16  
   2.5 Tracing the Probes ..................................... 17  

3 Data Analysis ............................................. 21  
   3.1 Obtaining Raw Output ................................... 21  
   3.2 Organizing Output ..................................... 22  
   3.3 Further Processing .................................... 23  
   3.4 Visualizing Result ..................................... 26  

4 Future Work ............................................... 30  
   4.1 Train Classifier on Invocation Patterns ............... 30  
       4.1.1 Longer Activities ................................. 30  
       4.1.2 Larger Files ...................................... 31  
       4.1.3 Different Types of Devices ....................... 31  
   4.2 Extend to Other Components of the System .......... 31  

5 Conclusion ................................................. 33  

Appendices .................................................. 35  
   A Python Scripts ........................................... 36  
      A.1 add_probe.py .................................... 36  
      A.2 fbt_chain.py ..................................... 43  
      A.3 copy.sh ........................................... 47  
      A.4 graph.py ......................................... 48  

Bibliography .................................................. 50
LIST OF FIGURES

1.1 Kernel queue model ......................................................... 4
1.2 Sample DTrace output ..................................................... 8

2.1 Sample DTrace probe ....................................................... 11
2.2 Sample output from built-in FBT ......................................... 12
2.3 Example of removing syntactic sugar .................................... 14
2.4 Sample DTrace entry probe ................................................. 14
2.5 Sample DTrace return probe ............................................... 14
2.6 Customized probe for USB transfer function .......................... 15
2.7 Final version of probes for USB transfer ............................... 17
2.8 Sample D script ............................................................. 18
2.9 Customized script to capture inserted probe firing .................. 19
2.10 Sample output from customized probes ............................... 19

3.1 Excerpt of captured output ................................................. 22
3.2 Organized output after processing by FBT ............................ 23
3.3 Process switching with device ID ......................................... 24
3.4 Final organized output ..................................................... 25
3.5 Separated output for endpoint 2-2 ....................................... 26
3.6 Output from each device endpoint ....................................... 27
3.7 First 1000 lines of two endpoints ...................................... 28
CHAPTER 1

INTRODUCTION

Opportunities for malicious software lie in almost every area of the computer, security engineers have never ceased their attempts to look for vulnerabilities and to prevent malicious programs from utilizing these vulnerabilities. Among these different areas of an operating system, the USB component has not received sufficient attention in terms of security research. In this paper, I use USB transactions as the attack surface for the study to find an effective way to prevent malicious software from using the kernel for unexpected purposes.

In Chapter 1, I discuss the historical background on security research about operating systems security and specifically USB attacks and why effective solutions in this subject matter are hard to come by. I introduce two approaches by previous researchers—TaintBochs and KLEE—and their respective weaknesses in terms of performance and comprehensiveness. I then present my solution which uses DTrace and FreeBSD to trace the function calls made during the lifetime of USB requests and formulates expected invocation patterns to detect anomalies.

1.1 Background

Ample research has been done on improving security of operating systems, specifically the kernel which executes almost all actions requested by programs. There are two primary components to the kernel attack surface that are often exploited by attackers—system calls and I/O attacks. I/O attacks includes a wide range of types of attacks that can be carried out. A common I/O attack exploits vulnerabilities during the parsing of meta information request data, usually supplying crafted extra data to make the kernel execute arbitrary commands. Therefore, it is important to track kernels jobs throughout their lifetime and check for abnormal behaviors.
Nevertheless, the modern design pattern of the kernel, where jobs are frequently enqueued and dequeued and handled by different processes to improve performance, creates problems for security researchers who wish to track the activities that take place during the lifetime of a single request, in that we are uncertain which process will pick up the task and when the task will be picked up. If we track the activities on a specific process and the process puts its task into the job queue, we essentially lost track of the task. Without the ability to track a single task within the kernel, it is difficult for researchers to detect any abnormal behaviors of programs. If the request attempts to access some subcomponents of the kernel that are potentially dangerous after switching processes, we will have no way to know. Approaches proposed by earlier researchers focused on different aspects of the problem; in this section I will present both the details of the problem and the proposed approaches by previous researchers and how they can be improved with my own solutions.

1.1.1 The Problem

Having been in use for almost 20 years [7], USB technology is deeply intertwined with many aspects of our lives including transferring data between end devices, connecting accessories to computers and so on. As prominent as USB technology is, there has been a noticeably insufficient amount of research done on USB security, leaving the field an open playground for hackers.

In fact, USB devices are a fantastic collection of targets for attackers due to their wide applications including USB mass storage devices, USB accessories such as keyboards, mice and so on. It is also notable that many people are willing to plug in any strange USB device that they find [12], which gives attackers direct access to their computers.

USB devices are perfect for attackers technically as well for many reasons, ranging
from the widely applicable buffer overflow attacks to more USB specific exploits such as format string bugs and logic errors [10]. In general, when a USB device wants to communicate with the kernel to execute certain tasks, it should construct a USB packet much similar to that of an IP packet, wrapping around the actual data sent with metadata in the front as a header. To the kernel, there is virtually no difference between data sent from a USB mass storage device and a keyboard connected through a USB port, or even worse, from a device specifically manipulated to supply malformatted input to exploit the kernel. Similar to attacks that target the Internet layer in computer networking, USB attacks can exploit the kernel through common techniques such as ill-formed header, as discussed in the previous section.

The severity of a potential USB attack is not trivial either. As a matter of fact, a USB mass storage device is able to reach many subsystems of the kernel as USB is designed to support many heavyweight classes. For example, when playing an audio file from a USB drive, the program will have to involve the audio processing component, or when an operation requires access to a human interface device. A malicious USB request could potentially gain access to more important components of the kernel. Thus, it behooves us to be able to track what code in the kernel is executed as a result of a particular USB activity.

It then becomes critical that security researchers have the ability to track the entire lifetime of a USB request and to monitor each step that it carries out and every function that is called. Tracking code activities is difficult because of multiple reasons including the high number of processes running at the same time and the randomness of the location where the data is stored, etc. The biggest difficulty of all comes from the asynchronous nature of code execution, thanks to the design of modern kernels, where tasks are frequently enqueued and dequeued by different processes to increase kernel efficiency. For example, theoretically, when a request to copy a file from a USB mass
storage device to local disk is made on process 1, the kernel first copies the file into memory then attempts to write the file to destination. If there are other similar tasks going on at the same time, the process will likely put the task on a job queue and when the kernel frees up, process 2 will pull the task from the job queue and finished the writing. As shown in Figure 1.1, I/O operations are scheduled by the CPU to enqueue and dequeue for execution.

![Kernel queue model](image)

**Figure 1.1: Kernel queue model**

USB is far from the only field that should receive attention when it comes to discovering opportunities for attacks, however, its structure is representative of many other subsystems. Using USB as a target, I can easily extend the methods employed in this project to detect program anomalies in other components of the kernel.

### 1.1.2 Proposed Approaches

I used a code tracing tool, namely DTrace, a toolkit built into many UNIX based operating systems, with the FreeBSD kernel to track the activities of USB requests.
DTrace provides probes that are inserted at specific locations in the kernel source code. A probe will fire and return arbitrary information of the programmer's choice when the program executes to the line where it is inserted. By studying the probes that are fired during the execution of a program or a single request, I am able to determine what functions are invoked and in what order are they invoked. In this study specifically, due to the lack of comprehensiveness in the default probes that were already inserted, I inserted DTrace probes into every location relevant to USB requests in the FreeBSD source code. These probes were fired when a specific function was entered or exited from, thus forming a chain of invocation information to indicate the order and layer by which the functions are called during the handling of a specific USB operation. To automate the process of inserting DTrace probes into kernel source code, I wrote a Python script that analyzes the syntax of the kernel source code and inserts customized DTrace probes. I also wrote a tracing script in DTrace’s own D Language which returns the necessary information when the probes are fired. After cleaning up and organizing the output from the tracing script into a more recognizable style, with the organized activation information of a standard operation, I then compared the expected activation record with any incoming operations. If a function is called where it is not supposed to, I could be certain that there is something abnormal taking place inside the program, and from this point determine whether to take any precautious actions such as terminating the program that is sending out the request being investigated.

In the past, many researchers have proposed to solve this problem through different means, too. Chow et al. [9] focused on limiting the active lifetime of sensitive data throughout the entire period when the program is running by using TaintBochs. TaintBochs creates a complete mirrored image of the current memory state and in the mirrored memory “taints”, or annotates sensitive data by setting specific bits in the mirrored memory. TaintBochs checks for inappropriate handling of sensitive data, such as
leaving data sitting in a memory location for too long, not erasing the memory location after dereferencing the data and so on. This method gives security researchers a holistic view on how programs handle sensitive information and thus a better chance of spotting dangerous operations. TaintBochs helps developers find vulnerabilities in their programs and therefore to improve information security. However, “tainting” the data, including copying and tracking the same data in the mirrored memory requires huge amount of memory space and computing power. It is virtually impossible to implement TaintBochs and use it for a large scale project, let alone the whole system, making TaintBochs perfect for research purposes but not a production environment.

Instead of trying to find insecure operations among programs, Cadar et al. [8], authors of KLEE, a symbolic execution tool which automatically generates tests for programs, tried to implement as many tests as possible for the programs. KLEE uses symbolic execution to find all possible paths that a program can take when supplied with different inputs. KLEE then attempts to write tests for all the possible scenarios and account for as many opportunities for attacks as possible. By improving the coverage on tests, Cadar et al. hoped to reduce the chances where programs are shipped with defects and can potentially harm the computer if their vulnerabilities are exploited by attackers. However, this method still requires a considerable amount of manual labor and that the developers of the software actively use KLEE in the development process. More importantly, this method does not help the case where the program was written with the purpose of hacking in mind.

1.2 My Solutions

As the first step to monitor and track code activities in the kernel, I focused on a rather small part of the kernel which handles the USB requests. The default Function Boundary Testing (FBT) probes do not satisfy the requirement of my study as it only
reports a portion of the functions called in the process. Using FreeBSD and the built-in DTrace tools, I modified the FreeBSD kernel source code by inserting my own DTrace probes at each entrance and return location of every function and thus constructed my own Function Boundary Testing suite. I wrote a Python script to insert the probes into the kernel source code and another Python script to clean up and organize the output into a proper activation chain.

1.2.1 FreeBSD & DTrace

FreeBSD [3], an open-source, UNIX based operating system provides a built-in DTrace toolkit and minimum extra infrastructure which makes study results less influenced by other components of the kernel. Being UNIX based, FreeBSD also makes studies relatively easier to be ported to other platforms.

DTrace [2], widely available across many operating systems including FreeBSD, many distributions of Linux and OS X, provides the ability for developers to track code activities by inserting probes into appropriate locations in the kernel source code. The probes fire and print out customized output whenever the program executes to that specific line where the probe is inserted. I used DTrace primarily to indicate when a function is called and when the same function finishes, i.e., returns.

1.2.2 Code Tracing

In order to find a lightweight but comprehensive approach to detecting anomalies in code execution, I experimented with the built-in FBT, Function Boundary Testing, functionality which comes with the DTrace toolkit. This, however, yielded unsatisfying results as it merely tracks a small amount of functions. To achieve a comprehensive understanding of the calling of the functions, I modified the FreeBSD 11.1 [4] kernel
source code by inserting probes at appropriate locations to find out the execution chain of different USB operations. The probes are added at the entrance and every possible return location in each function in order to accurately monitor the life cycle of all the USB related functions.

With the probes in place, I used the D Programming Language[1] specific to DTrace to trace the probes’ firing events. The script produces output similar to the excerpt in Figure 1.2:

```
dtrace: script 'rw.d' matched 5 probes
CPU ID  FUNCTION:NAME
  0 59126  read:entry     1
  0 59126  read:entry     2
  0 59128  write:entry    3
  0 59128  write:entry    4
  0 59126  read:entry     5
  0 59126  read:entry     6
  0 59128  write:entry    7
  0 59126  read:entry     8
  0 59128  write:entry    9
  0 59128  write:entry   10
...
```

Figure 1.2: Sample DTrace output

To further utilize the output and organize it into useful information, I wrote a Python script to organize the probe firing events into layered chains which sometimes range across different processes when enqueued or dequeued from the job queues.

In Chapter 2, I will discuss the existing infrastructure that comes with FreeBSD; the shortcomings in Section 2.1, and the customized DTrace probes and how I inserted them into the kernel source code in Section 2.2. I will discuss how I traced the USB tasks across different processes despite the fact that tasks are frequently enqueued and dequeued by the kernel at different job queues in Section 2.3, how I modified the probes to return information on original devices in Section 2.4, and how I organized the output
into usable information in Section 2.5. In Chapter 3, I will present the results obtained from the experiments, how I cleaned up and organized the results into different formats for interpretation and further analysis as well as the process I undertook to visualize the invocation pattern at the end. In Chapter 4, I discuss the applications and the possibilities of extending this approach of detecting anomalies through code tracing to other subsystems of the kernel.
CHAPTER 2
HACKING THE KERNEL

As discussed in the previous chapter, I used Function Boundary Testing (FBT) to track the invocation chain of the functions called during the lifetime of a single USB transaction. To be able to monitor the calling of every individual function, I needed DTrace probes at the entry and return locations of every single function. Given that the existing probes that came with the default DTrace toolkit only covered a portion of the USB functions and were insufficient for the purpose of this study, I needed to manually insert more probes into the kernel to monitor the invocation of the functions. I wrote a Python script to modify the kernel which added customized DTrace probes to the FreeBSD kernel source code. I inserted the probes at every line of entry and return location inside each function related to USB operations—in this case, every function in files that are under the `/usr/src/sys/dev/usb` directory. The probes triggered whenever a program executes to the line where the probes were inserted. I wrote a script in the D language provided by DTrace to collect the returned information such as process ID, probe name and function name, etc. The information returned were used in post analysis to find an expected invocation pattern of different USB operations.

This chapter describes the process I undertook to study and modify the FreeBSD kernel source code to insert customized DTrace probes to monitor the USB request behaviors. In Section 2.1 I describe the existing DTrace infrastructure and the need to customize what is inserted into the code. In Section 2.2 I describe the process where I wrote a Python script to insert DTrace probes into the kernel source code and in Section 2.3 and 2.4 I talk about the extra modification I did manually to enable the probes to track the requests between processes and threads and from different devices. Lastly, in Section 2.5 I talk about writing D scripts to trace the DTrace probe firing events.
2.1 Existing Infrastructure

The out-of-box FreeBSD kernel comes with 60306 usable probes already in place. The probes, however, are very sparse and spread out in many components. Thus, as a result, specifically in the context of USB operations, the existing probes are not enough to cover all the USB related functions under the USB directory. The default probes only cover part of the functions that are related to USB operations. To have a comprehensive analysis of the invocation chain without missing any of the functions, the probes need to include all functions under the USB directory in the source code for this study. To accomplish that, I had to insert my own probes into the kernel source code and recompile the kernel to utilize the probes inserted. The DTrace toolkit allows for creation and insertion of customized probes into kernel source code. Figure 2.1 is an example of typical DTrace probe insertions at the entry point and before return statements in a function. The probes will fire when the function is first invoked and right before the end of the function. The probes have zero extra arguments to return.

```c
usb_frlength_t
usbd_xfer_frame_len(struct usb_xfer *xfer, usb_frcount_t frindex)
{
    SDT_PROBE0(tpw, kernel, usb_transfer_usbd_xfer_frame_len, entry);
    KASSERT(frindex < xfer->max_frame_count, ("frame\nindex\noverflow"));

    SDT_PROBE0(tpw, kernel, usb_transfer_usbd_xfer_frame_len, return);
    return (xfer->frlengths[frindex]);
}
```

Figure 2.1: Sample DTrace probe

In this specific example, after recompiling and reinstalling the kernel, I would use a tracing script to collection information on including when the function `usbd_xfer_frame_len` is called from the first inserted probe, on which thread is the function called and when the function returns from the second inserted probe. Inserting probes
at such locations for all functions under the `/usr/src/sys/dev/usb` directory enabled me to obtain such information for all USB related functions.

After acquiring the invocation information, the next step was to analyze the calling of the methods and determine the invocation order by looking at when the program enters and exits the methods, i.e., Function Boundary Testing (FBT). The built-in DTrace toolkit supports a very crude version of FBT with the “flowindent” option. However, the option adds indentation to all functions without regard to their calling process or device information, making it difficult to distinguish between actions performed by different USB devices at the same time. For example, Figure 2.2 shows an output obtained using the built-in FBT functionality, which is far from enough for any constructive analysis. The output shows the nested order of function calls when the USB component is in the idle state, but when two USB devices issue requests at the same time, the two separate requests will be intertwined and impossible to distinguish.

![](image)

Figure 2.2: Sample output from built-in FBT

Moreover, the built-in FBT toolkit cannot track individual tasks across different processes, for instance, when a request is enqueued onto the job queue and later dequeued by another process for handling. As a result, the built-in FBT function will have no way to track this event, losing track of code activities across threads, eventually making the output from Function Boundary Testing inaccurate. The lack of cross-process and cross-device support calls for a more robust FBT tool that can track individual requests.
issued by different devices and across different processes.

2.2 Customized DTrace Probes

As a result, the existing probes shipped with the built-in DTrace toolkit do not exist at all locations necessary for my purpose of information collection, requiring me to manually insert more probes into the kernel source code.

The first step I took was to look for the functions that were associated with USB operations. After searching through all directories in the kernel source code and using GNU grep \[5\] to find USB related file names, I found that the FreeBSD kernel code that handles the USB functionalities almost exclusively resided in the /usr/src/sys/dev/usb directory. To trace all the USB code activities, I needed to add probes to every function under this directory, similar to the fashion shown in Figure 2.1. To accomplish this goal, I needed to systematically insert lines into the kernel source code; nonetheless, the source code was full of syntactic sugar which makes it difficult for a simple script to determine the exact workflow of the function. For example, the source code sometimes divides a long line of code into multiple lines to fulfill the 80 character requirement which creates confusion for a script to determine the end of an actual line of code.

I created a script (Appendix A.1) which syntactically analyzes the C source files in the directory to resolve this problem. The script takes a first pass over all the C source files to expand any potentially problematic syntactic sugar that could cause confusion for probe insertion in the second step. For example, as shown in Figure 2.3a and Figure 2.3b, the first pass added the curly brackets back into the if statement so that the script can later determine the boundaries of an if statement by looking at the curly brackets. The first pass also removes any in-line style comments to ensure that every last character of each line is meaningful and usable by the script later in the second step.
if (condition) {  
    statement;  
}

(a) Original statement  
(b) Statement modified by script

Figure 2.3: Example of removing syntactic sugar

In the second pass, after cleaning up the source code, the script inserts an entry probe with no arguments (Figure 2.4) for every single function under the directory and return probes as shown in Figure 2.5 before every return statement and every end of function.

```c
const char *usb_get_serial(struct usb_device *udev) {
    SDT_PROBE0(tpw, kernel, usb_device_usb_get_serial, entry);
    SDT_PROBE0(tpw, kernel, usb_device_usb_get_serial, return);
    return (udev->serial ? udev->serial : "");
}
```

Figure 2.4: Sample DTrace entry probe

Figure 2.5: Sample DTrace return probe

There exists only one entry probe for every function but sometimes multiple return probes due to return statements inside conditional statements. After recompiling and reinstalling the kernel with the extra probes added in, I had a total of 868 probes specifically designated for USB functions available for use.

2.3 Coordinating between Job Queues

After adding the generic entry and return probes, I was able to retrieve invocation information on all the functions in chronological order. Nonetheless, it was still not
enough for me to obtain just the order of all the functions called while a single request is being executed. This is because a task may be enqueued and dequeued during its handling, essentially paused for a period of time. Within this period of time, the function calls that are collected may not be strictly related to the specific task under study. In fact, I needed to obtain information on the enqueuing and dequeuing of the tasks, specifically when and where the job queue related events take place. To accomplish this goal, I needed to modify the probes that were inserted into the queue related functions to return more information than just the fact that it was invoked.

To do so, I searched through the USB folder to look for any job queue related functions, by searching through the content of all C source files under the `/usr/src/sys/dev/usb` directory again using GNU grep. As a result, there were two functions that specialized in handling the interactions between the job queues and USB requests, both in the `usb_transfer.c` file. The two functions, namely `usbd_transfer_enqueue` and `usbd_transfer_dequeue`, are the only two functions within the USB component of the kernel that made them add and remove the USB tasks to and from the job queues, which was rather convenient to modify. I manually added the probes inside these two functions by editing the source file to return extra pieces of information: the address of the job queue and the address of the payload, i.e., the task that is added to the queue. At the end, I also returned the device address and endpoint number as part of the result when these probes fired, as shown in Figure 2.6.

```c
SDT_PROBE4(tpw, kernel, usb_transfer_usbd_transfer_enqueue, entry, xfer, pq, xfer->address, xfer->endpointno);
```

Figure 2.6: Customized probe for USB transfer function

When these two functions fire, DTrace returns the addresses of the queue and the tasks, and the device ID of the sender device of the request, giving me information to determine the identity of the tasks added and thus enabling me to track individual
tasks chronologically across different processes even when they are transferred from one kernel thread to another.

### 2.4 Identifying Original Devices

Having mere process IDs was helpful in tracking tasks and across different processes, but not enough when the situation involves two or more USB devices issuing requests at the same time. To identify the devices that issue each request, I needed to find a unique identifier, whether a number or a string that corresponds solely to the device itself. I started looking for the definition of the payload struct, namely `usb_xfer` in the `usbd_transfer_enqueue` and `usbd_transfer_dequeue` functions. After searching through the header files under the USB directory, I found the definition of the `usb_xfer` struct to be in the `usb_core.h` file where it had two fields that were related to its physical identity, namely `address` and `endpointno`, the device address and the endpoint number. In short, the `address` field points to the physical device and the `endpointno` field points to specific endpoint on the device, such as Control Endpoint and Data-Transfer Endpoint. The `address` and `endpointno` combined serve as an identification for the original device from where the task is issued. After identifying the physical device, I added both the address and endpoint number as extra fields to return to the probes inserted into the `usbd_transfer_enqueue` and `usbd_transfer_dequeue` functions. The final version of the inserted probes in the two queue related functions are shown in Figure 2.7 using the `usbd_transfer_enqueue` function as an example.
void
usbd_transfer_enqueue(struct usb_xfer_queue *pq,
                      struct usb_xfer *xfer)
{
    SDT_PROBE3(tpw, kernel, usb_transfer_usbd_transfer_enqueue,
                entry, xfer, pq, xfer->endpointno, xfer->address);
    if (xfer->wait_queue == NULL) {
        xfer->wait_queue = pq;
        TAILQ_INSERT_TAIL(&pq->head, xfer, wait_entry);
    }
    SDT_PROBE3(tpw, kernel, usb_transfer_usbd_transfer_enqueue,
                return, xfer, pq, xfer->endpointno, xfer-address);
}

Figure 2.7: Final version of probes for USB transfer

After running the Python script that inserted all the probes into the kernel source
code, I recompiled and reinstalled the kernel to enable the probes. The output from
“dtrace -l | grep tpw”, which lists all the DTrace probes installed on the system that has
the string “tpw” in the descriptions, in this case, the provider name showed 868 probes
in total which included every single function under the USB directory in the FreeBSD
kernel source code.

2.5 Tracing the Probes

DTrace provides an infrastructure where probes, inserted at specific locations in the
kernel source code, are fired whenever the lines where they reside are executed. The
probes are monitored and activated by scripts written in the D Language, as mentioned in
previous chapters. The D Language is robust in that it is highly customizable, allowing
users to track specific probes with arbitrary criteria. Figure 2.8 is an example script in
this language, which fires the probes whenever an IO operation (read, write) is invoked
on the process with id 2219:
After the customized DTrace probes were inserted into the kernel source code, to trace the probes in the most concise fashion without collecting extra information, I wrote my own D script to intercept all the firing events of only the probes that were added by me, i.e., specified by the provider name “tpw”. I ran the script in one process, and at the same time fired one of the USB CRUD operations and collected the output of the script after the USB operation was finished, as shown in Figure 2.9. The CRUD operations I ran included adding a file to a USB drive, removing a file from a USB drive, copying from a USB device and modifying a file on a USB drive.
The script prints out the execution name, process id and the probe function name when detecting the firing of any probe, and specifically for the four entry and return probes for the `usbd_transfer_enqueue` and `usbd_transfer_dequeue` functions, the script also prints out the first two arguments returned by the probe, in this case, the address of the job queue and the task payload, as shown in Figure 2.10, the last two arguments returned are the address of the job queue and task payload.

```
0 56919 usb_transfer_usbd_transfer_enqueue:entry intr 12
  usb_transfer_usbd_transfer_enqueue
  ffffe00015c14d8 ffffe00015c1060
```

Figure 2.10: Sample output from customized probes

To summarize, I found out that the existing DTrace probes that came with DTrace and FreeBSD were not enough for the purpose of this project and wrote my own scripts
to insert customized DTrace probes into the kernel source code. I also manually modified probes in two specific functions that handle specifically the enqueue and dequeue operations for the USB requests. Lastly, I wrote a script in the D Language to trace the firing events of the probes to collect raw data.
CHAPTER 3

DATA ANALYSIS

I obtained raw output from a standard USB mass storage device with the help of the DTrace probes I inserted into the USB functions and a customized D Language script that collected the firing events of the probes. The raw output contained every piece of information returned by the probes and formed a basis for analysis. I cleaned up the raw output and organized it into a more comprehensible form using another Python script that I wrote. The organized output provided extra information such as switching processes and queue operations. Being able to track when each task is enqueued and dequeued from the job queues allows me to gain a holistic view of the entire lifetime of an operation and to have a complete record of invocation for each request and device.

3.1 Obtaining Raw Output

To collect raw data for my experiment, I used a standard 32G SSK USB mass storage device for the single-device scenarios and a 32G PNY USB mass storage device when experimenting with two USB devices. I connected the devices to my FreeBSD instance running as a guest on VirtualBox. VirtualBox, by default, does not support porting USB devices that are inserted into the actual machine to the VirtualBox guest machine. I downloaded the VirtualBox VM Extension Pack [6] from the VirtualBox official website to enable USB passthrough. To precisely control the amount of probe firing information collected by the script shown in Figure 2.9, I wrote a shell script (Appendix A.3) to manage the raw data collection. The shell script first ran the D script used to collect DTrace output in the background and then executed one of the five different USB operations: add, delete, read, mount and unmount. Once the USB operation finished executing, the shell script would be notified to kill the D script in the background and redirect its output to a text file. For the add, delete and read operations involving an
actual file, I used a plain text file which contained only one line of text that read “Hello World”. Figure 3.1 is an excerpt of the output captured and unmodified when I added the test file to the USB device:

```
0 57347 usb_...:entry intr 12 usb_..._start
0 56958 usb_...:entry intr 12 usb_..._defer
0 56919 usb_...:entry intr 12 usb_..._enqueue ff...d8 ff...60 2 2
0 56919 usb_...:entry intr 12 usb_..._enqueue
0 57420 usb_...:return intr 12 usb_..._enqueue ff...d8 ff...60 2 2
0 57420 usb_...:return intr 12 usb_..._enqueue
```

Figure 3.1: Excerpt of captured output

The raw output contains unfiltered information returned from the general probes that I inserted into the kernel source code, as shown in Figure 3.1; besides the miscellaneous information in the first two columns, the output contains the name of the probe fired, the type of the probe fired, the process on which the probes were fired, etc. For the probes in the two enqueue and dequeue functions, the output contains three more pieces of information: two addresses, respectively of task payload and job queue, and the device number, as shown in the third and fifth line in Figure 3.1.

### 3.2 Organizing Output

The unmodified output contained all raw information I needed, but was very difficult to interpret for both humans and machines. I wrote another Python script (Appendix A.2) to organize the output into more recognizable fashion. The script reads in the raw output of the tracing D script and keeps a call stack of the function calls. Every time the script reads in a line describing an entry probe (i.e., the function is being called as the first function ever or before the previous function ever returns), it adds one level of indentation. When the script reads in a line describing a return probe, it pops the returned function off the call stack and removes one level of indentation correspondingly.
The script will then print the indented output with only an arrow pointing either left or right indicating an entry or return event, the probe name which consists of both file name and function name, and the process ID. The script also identifies and keeps track of different addresses of task payloads and job queues so that I could trace different payloads across different processes.

```plaintext
1  -> usb_transfer_usbd_transfer_start 12
2  -> usb_transfer_usbd_callback_ss_done_defer 12
3  **********Adding Payload 0 at Queue 0**********
4  -> usb_transfer_usbd_transfer_enqueue 12
5  <- usb_transfer_usbd_transfer_enqueue 12
6  -> usb_process_usb_proc_msignal 12
7  <- usb_process_usb_proc_msignal 12
8  <- usb_transfer_usbd_callback_ss_done_defer 12
9  ===========Switched to process 14==========
10 -> usb_transfer_usb_callback_proc 14
11 -> usb_transfer_usb_command_wrapper 14
```

Figure 3.2: Organized output after processing by FBT

Figure 3.2 is an excerpt of the organized output. The excerpt depicts the beginning lines of a typical workflow for an operation where a file is added to a USB mass storage device mounted on the computer. The excerpt shows the list of functions called in order during the operation and how the functions are nested within each other. The excerpt also includes incidents such as switching process (ninth line) and enqueueing task to job queue (third line) which are highlighted with special lines.

### 3.3 Further Processing

The output could now serve as a clear record of function invocations for any USB operation that it captures. However, in real-life scenarios, there sometimes can be multiple USB operations taking place at the same time and the current organization of output would combine the function calls from different devices altogether, making it difficult
to distinguish the actions of a single device among many. I further modified the Python script (Appendix A.2) to include the device number in the output for lines that indicate the enqueue (add) and dequeue (pop) events for the task payloads, as shown in Figure 3.3.

**********Adding Payload 0 from Device 2-2 at Queue 0**********
**********Popping Payload 0 from Device 2-2 at Queue 3**********

Figure 3.3: Process switching with device ID

After the modification, looking at the indented output produced by the Python script, I was able to tell which device sent the task payload that was being enqueued or dequeued. The information helped to determine what device to which each function was related based on the assumption where once a task payload is dequeued from the job queue, the methods following the dequeue event are related to that task. With this assumption, I was able to append a device ID to most of the invocation records in the organized output to distinguish between methods invoked by different devices.

To utilize this functionality, I conducted another experiment using both USB devices mentioned above. I mounted both USB devices before running a different shell script which simultaneously copies a text file onto both USB devices and records the function invocation in the background, similar to the script mentioned in Section 3.1. Figure 3.4 shows an excerpt of the organized output from this experiment.
As shown in the first line in Figure 3.4, I appended -1 to the output when no task payload had been dequeued from the job queue as it is unclear which device issued the specific function call. Once a task payload was dequeued from the job queue (in this case, the payload numbered 0 which was issued by the USB device identified as 2-2 was dequeued from the job queue numbered 3), the functions following this dequeue event all the way until the next dequeue event were considered to have been issued by the endpoint numbered 2 on the USB device at address 2. I further separated the output by device number into individual files, one for each endpoint, containing only the function invocations but not process switching or enqueue/dequeue events as they were not helpful anymore since all and only the function calls related to this device were in one place. As expected, there were four distinct endpoints in total in the output, two for each USB device. An example of the separated output for endpoint 2-2 in the aforementioned experiment is shown in Figure 3.5.
3.4 Visualizing Result

Another problem arose when I attempted to understand the invocation pattern produced in Section 3.3. Even the separated output contained way more lines than were possible for meaningful analysis: respectively, the output for the two USB devices contained 12367 and 15314 lines, combining two endpoints, in total. To understand the output from a higher level and abstract out the pattern from the invocation information, I wrote a Python script (Appendix A.4) to extract the indentation levels to a list and plotted the list as a bar plot. The plots for the four endpoints of the two devices are shown in Figure 3.6.
Figure 3.6: Output from each device endpoint

The four graphs depict the level of nested function calls during the life cycle of each USB “copy” operation for each endpoint of the two USB devices. To see the pattern more clearly, I took the first 1000 lines from the output of one endpoint of each device and plotted new graphs from them, the results were much clearer to see, as shown in Figure 3.7a and Figure 3.7b:
Comparing the patterns in these figures, it is easy to see that there are many similar repeated peaks in both figures and I can assume that this pattern is typical for add operations for USB devices.

Once I conduct more experiments on different types of USB operations, and I have integrated information on both nested level and specific functions called, I will be able to extract more general and representative patterns for different operations and these patterns will form the basis of the anomaly detection for USB requests. For example, once the system has an expected pattern of function invocation recorded, and it detects a USB request that does not follow the expected chain of information or it has invoked certain functions that were never called by any of the USB requests, it can confidently say that this request is not doing what it is advertised to do. Based on the observation, the system can decide to either kill or report the request with anomalies.

In conclusion, using a shell script, I automated the tracing process and collected raw data using the DTrace script. With the function invocation information, I wrote a Python script to organize the raw data into a chained format to better perform Function Boundary Testing, and to mark the events where the tasks were enqueued and dequeued from the job queue and where each task came from. After being able to separate all the dis-
distinct endpoints for different devices, I wrote another Python script to plot the indentation level information into graphs and demonstrated the similarities between their patterns. Table 3.1 shows the three scripts I wrote to conduct the experiments and organize the results.

<table>
<thead>
<tr>
<th>Script</th>
<th>Appendix</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fbt_chain.py</td>
<td>A.2</td>
<td>Organize raw output</td>
</tr>
<tr>
<td>graph.py</td>
<td>A.4</td>
<td>Turn organized results into graph</td>
</tr>
<tr>
<td>copy.sh</td>
<td>A.3</td>
<td>Run experiment automatically</td>
</tr>
</tbody>
</table>

Table 3.1: All scripts used
CHAPTER 4
FUTURE WORK

4.1 Train Classifier on Invocation Patterns

I was able to conjecture a general pattern with the two images produced from the indentation levels of the first 1000 lines through empirical observation. Nevertheless, it becomes much more difficult to extract patterns from the entirety of all tens of thousands of lines without additional analysis using some machine learning algorithm. Due to the real-time nature of the anomaly detection system described in this paper, where an ideal algorithm has records of expected function invocation information and runs classification on function invocations that come in as a running stream, it behooves us, in the future, to develop an online classification algorithm to look for abnormal behaviors of USB devices. Working towards this goal, I could conduct more experiments with the following modifications in order to gain a comprehensive collection of function invocation data to train the online machine learning algorithm:

4.1.1 Longer Activities

Currently, all USB operations conducted in the experiments were of short periods of time with simple CRUD operations. I will gain a better variety of function invocation records if the invocations are done with more operations in series and take longer time, which will be helpful for the machine learning algorithm to base its generated patterns on.
4.1.2 Larger Files

As described in Section 3.1, the test file used in the experiments to be copied and deleted was merely a text file containing one line of “Hello World”, which is not large enough to trigger potential edge conditions that might have something to do with caching the file copied, etc. Using larger and different types of files could also trigger different USB component behaviors to help the machine learning algorithm train.

4.1.3 Different Types of Devices

As of now, the experiments described only used USB mass storage devices as test subjects. However, as mentioned, there are many more types of USB devices than mere pen drives. I could run different experiments with other types of USB devices such as USB keyboard, USB mice, printer and so on. Different USB behaviors will invoke different functions inside the USB component in the kernel source code and will help the algorithm to eliminate false negatives when it sees behaviors that are not normal to a USB mass storage device but could be from a USB accessory, and thus increase recall stat, defined as the ratio of true positives over true positives plus false negatives, of the machine learning algorithm.

4.2 Extend to Other Components of the System

USB is far from the only component in the kernel that needs attention from security engineers, and the same approach described in this paper can be applied to other subcomponents of the system too. For any confined subcomponent in the kernel, I can follow the steps similar to those I did to the USB component in this paper and insert DTrace probes into all the functions related to that specific subcomponent. Next, I can start running experiments where I exhaust the possible operations for this subcomponent.
to obtain function invocation patterns and organize the output into recognizable fashion.
Lastly, I could train a separate machine learning algorithm specialized in detecting ab-
normal behaviors for tasks interacting with this subcomponent.
CHAPTER 5

CONCLUSION

Using an out-of-box version of the FreeBSD kernel and the DTrace toolkit that comes with the operating system, I implemented Function Boundary Testing on the USB component of the kernel. DTrace provides a mechanism where we can trace the firing events of probes at different locations of the kernel source code to track the executions of different pieces of code and invocations of different functions in the kernel. As the default probes that are built into the DTrace toolkit are not enough to track every single function that are related to USB operations, I wrote a script to automatically insert 861 DTrace probes into every function that is related to USB operation in some manner. The probes are inserted to the beginnings of every function as well as every location where the function exits, whether it’s a return statement or the last line of the function. Besides the standard probes for ordinary USB functions, I also modified the probes in the `usbd_transfer_enqueue` and `usbd_transfer_dequeue` functions to return addresses of the job queue and task payload, as well as the address and endpoint number for the physical USB devices.

After adding the probes and recompiling the kernel, I used a customized D script to collect only the firing events of the probes that I added specifically, which tells me information about when each function is called in the life cycle of a USB operation and more specifically in what order they are invoked. I wrote another script to organize the output into a more recognizable form.

From the organized output I can formulate an expected invocation pattern of different USB operations, e.g., add file, delete file, etc. With the expected invocation chain in place, I can now run the same tracing process for each USB operation and compare the invocation pattern with the expected one to see if there exists any anomalies and halt the process immediately if it is determined to be accessing components of the kernel that it...
is not supposed to by invoking unnecessary functions.

This anomaly detection method involving Function Boundary Testing, modified accordingly, can be widely applied to other subsystems of the kernel as well. As long as an operation is known to invoke a certain set of functions in a general order, we can monitor the firing of events with minimum overhead and halt the operation of the potentially malicious program in real-time.
Appendices
A.1 add_probe.py

This script analyzes the syntax of the kernel source code and automatically inserts entry
and return probes at all appropriate locations.

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import sys

types = ['void', 'int', '_t', '*']

def main():
    if len(sys.argv) < 3:
        print('Usage: ./add_probe.py header.h path/to/directory ...')
        print('Usage: ./add_probe.py --revert path/to/directory')
        exit(1)

    path = sys.argv[2]

    # Revert all changes made, back to a no-probe world
    if sys.argv[1] == '--revert':
        all_files = os.listdir(path)
        for source_file in all_files:
if source_file.endswith('.orig'):  
    os.rename(os.path.join(path, source_file), \ 
              os.path.join(path, source_file)[:-5])

return

# Find all the C source files
all_source_files = os.listdir(sys.argv[2])
source_files = []
for original_source_file in all_source_files:
    if original_source_file.endswith('.c'):
        source_files.append(original_source_file)

defined_probes = set()
declared_probes = set()

# For every C source file
for source_file in source_files:
    new_cfile = []
    old_cfile = []  # backup of the original file
    wait_one_line = False
    with open(os.path.join(path, source_file), 'r') as cfile:
        inter_cfile = []
        for row in cfile:
            if '/**' in row and row.strip().endswith('*/'):
                row = row[:row.index('*/')] + '\n'
inter_cfile.append(row)

if wait_one_line:
    if row.strip().endswith('{'):
        wait_one_line = False
    elif row.strip().endswith(')'):
        inter_cfile.append('{
')
        wait_one_line = False
    elif ';' in row.strip():
        inter_cfile.append('}'}
        wait_one_line = False
    if (row.strip().startswith('if ') or \
        row.strip().startswith('else ') or \
        row.strip().startswith('} else ')):
        if row.strip().endswith('{'):
            continue
        if row.strip().endswith('}')):
            inter_cfile.append('}'}
            wait_one_line = True

func_name = None
in_function = False
func_begin = True
name_on_next_line = False
levels_down = 0
return_void = False
line_num = 0
for row in inter_cfile:
    line_num += 1
    old_cfile.append(row)

if '#endif' in row and '#include' in old_cfile[-2]:
    new_cfile.append('#include' + 
    '<dev/usb/usb_tpw_probe_declare.h>

if row.strip().endswith('}):
    levels_down += 1
if row.strip().startswith('}'):  
    levels_down -= 1
if levels_down == 0:
    if return_void:
        probe_use = '	SDT_PROBE0(tpw, kernel, ' + 
        '%s, return);
' % (source_file[:-2] + '_' + func_name)
        probe_declare = '	SDT_PROBE_DECLARE(tpw, kernel, ' + 
        '%s, return);
' % (source_file[:-2] + '_' + func_name)
        probe_defined = '	SDT_PROBE_DEFINE0(tpw, kernel, ' + 
        '%s, return);
' % (source_file[:-2] + '_' + func_name)
        new_cfile.append(probe_use)
        declared_probes.add(probe_declare)
        defined_probes.add(probe_defined)
in_function = False
return_void = False  # Stop tracking return type
new_cfile.append(row)
    continue

# Return probes
if in_function and row.strip().startswith('return'):
    probe_use = '\tSDT_PROBE0(tpw, kernel, ' + \
       '%s, return);\n' % (source_file[:-2] + '_' + func_name)
    probe_declare = '\tSDT_PROBE_DECLARE(tpw, kernel, ' + \
       '%s, return);\n' % (source_file[:-2] + '_' + func_name)
    probe_defined = '\tSDT_PROBE_DEFINE0(tpw, kernel, ' + \
       '%s, return);\n' % (source_file[:-2] + '_' + func_name)
    new_cfile.append(probe_use)
declared_probes.add(probe_declare)
defined_probes.add(probe_defined)

# Return probe has to be added before return statement
new_cfile.append(row)

if name_on_next_line:
    func_name = row.split('(')[0]
name_on_next_line = False
# Entry probes

if func_begin and row == '{\n':
    probe_use = '\tsDT_PROBE0(tpw, kernel,’ + \n           ’%s, entry);\n’ \n    % (source_file[:-2] + ’’ + func_name)
    probeDeclare = '\tsDT_PROBEG_DECLARE(tpw, kernel,’ + \n                  ’%s, entry);\n’ \n    % (source_file[:-2] + ’’ + func_name)
    probeDefined = '\tsDT_PROBE_DEFINE0(tpw, kernel,’ + \n                   ’%s, entry);\n’ \n    % (source_file[:-2] + ’’ + func_name)
    new_cfile.append(probe_use)
    declared_probes.add(probeDeclare)
    defined_probes.add(probeDefined)

    func_begin = False

if not in_function:
    for t in types:
        if row.endswith(t + ’\n’):
            if t == ’int’ and len(row.split(’ ’)[-1]) > 4:
                continue
            if t == ’*’ and (len(row.strip()) == 1 or \n                    row.strip().startswith(’*’)):
                continue

41
if t == 'void':
    return_void = True
    func_begin = True
    in_function = True
    name_on_next_line = True
    break

new_cfile = ''.join(new_cfile)
os.rename(os.path.join(path, source_file), \
    os.path.join(path, source_file + '.orig'))
with open(os.path.join(path, source_file), 'w') as out_cfile:
    out_cfile.write(new_cfile)

with open(os.path.join(path, 'usb_tpw_probe_declare.h'), 'w') as declare_file:
    declare_file.write('#include <sys/sdt.h>
')
    declare_file.write('SDT_PROVIDER_DECLARE(tpw);
')
    declare_file.write(''.join(list(declared_probes)))

with open(os.path.join(path, 'usb_tpw_probe.h'), 'w') as define_file:
    define_file.write('#include <sys/sdt.h>
')
    define_file.write('#ifndef USB_TPW_PROBE
')
    define_file.write('SDT_PROVIDER_DEFINE(tpw);
')
    define_file.write(''.join(defined_probes))
define_file.write('#define USB_TPW_PROBE\n')
define_file.write('#endif\n')

if __name__ == '__main__':
    main()

A.2 fbt_chain.py

This script organizes the raw output from DTrace into indented recognizable shape.

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys

def main():
    if len(sys.argv) < 2:
        print('Usage: ./chain_fbt.py dtrace_output.txt')
        exit(1)

    filename = sys.argv[1]
    lines = []
    with open(filename, 'r') as dtrace_out:
        for row in dtrace_out:
            if len(row.strip()) != 0:
                lines.append(row.strip())
lines = lines[2:-1]

with open(filename[:-4] + '.chain', 'w') as outputfile:
    current_process = -1
    current_device = -1

    current_FUNCS = {}
    indent_levels = {}

    queue_dict = {}
    payload_dict = {}

    for entry in lines:
        splitted = entry.split()

        funcname = splitted[5]
        pid = int(splitted[4])
        provider = splitted[3]
        functype = splitted[2].split(':')[1]

        if functype == 'entry':
            if current_process != pid:
                for i in range(indent_levels.get(pid, 0)):
                    outputfile.write('	')
                outputfile.write('==========Switched to process' + 
                   ' %s==========
' % (pid))
current_process = pid

for i in range(indent_levels.get(pid, 0)):
    outputfile.write('	')

if ('enqueue' in funcname or 'dequeue' in funcname) \
and len(splitted) > 7:
    payload_id = payload_dict.get(splitted[-3], None)
    if payload_id == None:
        payload_id = len(payload_dict)
        payload_dict[splitted[-4]] = payload_id
    queue_id = queue_dict.get(splitted[-3], None)
    if queue_id == None:
        queue_id = len(queue_dict)
        queue_dict[splitted[-3]] = queue_id
    device_id = int(splitted[-2]) % 128
    device_addr = int(splitted[-1]) % 128
    outputfile.write('**********%s Payload %s from Device' + \
                     ' %s at Queue %s**********
' % ('Adding' if 'enqueue' in funcname else 'Popping', \
                payload_id, str(device_addr) + '-' + str(device_id), \
                queue_id))

if 'dequeue' in funcname:
    current_device = str(device_addr) + '-' + str(device_id)
    continue

outputfile.write('-> %s	%s	%s
' \

% (funcname, pid, current_device))
indent_levels[pd] = indent_levels.get(pid, 0) + 1
current_funcs.setdefault(pid, []).append(funcname)
else:
    if ('enqueue' in funcname or 'dequeue' in funcname) 
        and len(splitted) > 7:
        continue

if len(current_funcs[pd]) <= 0:
    continue

if current_funcs[pd][-1] == funcname:
    if current_process != pid:
        for i in range(indent_levels.get(pid, 0)):
            outputfile.write('	')
        outputfile.write('========Switched to process' + 
            ' %s========
' % (pid))
        current_process = pid
    indent_levels[pd] = indent_levels.get(pid) - 1
for i in range(indent_levels.get(pid)):
    outputfile.write('	')

outputfile.write('<= %s\t%s\t%s
' 
    % (funcname, pid, current_device))
current_funcs[pd] = current_funcs[pd][:-1]
if __name__ == '__main__':
    main()

A.3 copy.sh

Shell script that copies a file to two USB devices simultaneously.

#!/usr/local/bin/bash

# Copies a file into both usb at the same time

file=$1
usb1=$2
usb2=$3
script=$4

dtrace -s $script -b 100m > copy.txt &
sleep 2

cp $file $usb1 &
cp $file $usb2 &
sync
sleep 5

kill %1
sleep 1

pkill -f dtrace
A.4 graph.py

This script turns organized FBT output into graphs for visualization.

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'''
Graphs the chains to histograms
'''

import sys
import numpy as np
import matplotlib.pyplot as plt

def main():
    if len(sys.argv) < 2:
        print('Usage: ./graph.py chain.?')
        exit(1)

    filename = sys.argv[1]
    device_num = filename.split('.')[-1]

    data = []
    with open(filename, 'r') as chain_output:
        for row in chain_output:
            level = 0
            while row[0] == ' ':
                row = row[48:]
            while row[0] == '	':
                level += 1
                row = row[5:]
level += 1
row = row[1:]
data.append(level)

# histo(data)
bar(data, device_num)

def bar(data, device_num):
    plt.bar(range(len(data)), data, width=1)
    plt.ylabel('Nested Level')
    plt.title('Device %s' % device_num)
    plt.show()

def histo(data):
    plt.hist(data, normed=False, bins=10)
    plt.ylabel('Nested Level')
    plt.title('Histogram for Device 2')
    plt.show()

if __name__ == '__main__':
    main()
BIBLIOGRAPHY


