Perimeter-Crossing Buses: a New Attack Surface for Embedded Systems

Sergey Bratus
Dept. of Computer Science
Dartmouth College
Hanover, NH USA
sergey@cs.dartmouth.edu

Travis Goodspeed
Straw Hat
travis@radiantmachines.com

Peter C. Johnson
Dept. of Computer Science
Dartmouth College
Hanover, NH USA
pete@cs.dartmouth.edu

Sean W. Smith
Dept. of Computer Science
Dartmouth College
Hanover, NH USA
sws@cs.dartmouth.edu

Ryan Speers
River Loop Security
rmspeers@gmail.com

ABSTRACT

Any channel crossing the perimeter of a system provides an attack surface to the adversary. Standard network interfaces, such as TCP/IP stacks, constitute one such channel, and security researchers and exploit developers have invested much effort into exploring the attack surfaces and defenses there. However, channels such as USB have been overlooked, even though such code is at least as complexly layered as a network stack, and handles even more complex structures; drivers are notorious as a breeding ground of bugs copy-pasted from boilerplate sample code.

This paper maps out the bus-facing attack surface of a modern operating system, and demonstrates that effective and efficient injection of traffic into the buses is real and easily affordable. Further, it presents a simple and inexpensive hardware tool for the job, outlining the architectural and computation-theoretic challenges to creating a defensive OS/driver architecture comparable to that which has been achieved for network stacks.

1. INTRODUCTION

Securing a system requires preventing attackers from exploiting the inevitable vulnerabilities in real-world systems, particularly commodity embedded devices.

Whether a vulnerability can be exploited depends on the attacker's ability to deliver the exploit's crafted data to the vulnerable code or data location. Accordingly, practical defense must be based on understanding, in the targeted systems, the possible data flows that can deliver crafted inputs to this vulnerability locus.

For exploits delivered over TCP/IP networks, there exists a solid understanding of how data flows through TCP/IP networks and stacks, which informs the methodologies of network defense. However, we know of *no such understanding or methodology for data delivered over perimeter-crossing buses.* In this paper, we take initial steps to fill this gap in understanding.

The Network Channel. In particular, attacks on network stacks depend on the attacker's ability to craft and inject packets into a network medium. This fundamental dependency is the reason why network security is characterized by first-class engineering effort invested in packet crafting libraries and tools, such as *libnet* [3], *libdnet* [2], *Scapy* [6], *LORCON* [4], and many others.

Furthermore, practical defense of networked systems is based on understanding the paths a packet can take through a network and on through the target stack, and providing the defender with the ability to filter packets and their payloads *at multiple locations throughout their paths*, with the filter receiving the fullest possible context at each location.

Linux's *Netfilter* serves as a sterling example of this approach, and is arguably the most successful deployed OS security architecture. Netfilter provides a combination of hooks at critical junctures in a packet's path through the kernel and programmable environments to evaluate per-packet predicates and more complex history-backed assertions on a packet's protocol field values (e.g., *filter* table rules and *conntrack*-based rules respectively). BSD's *Pf* provides similar functionality.

Most importantly, Netfilter reflects the layered structure of TCP/IP stacks and can evaluate its rules in many contexts corresponding to its layers, at the exact spots where a protocol's reassembled payload is handed to the next layer for processing. Netfilter thus avoids a NIDS dilemma of either having to judge "raw" packets lacking sufficient context cues to their meaning, or having to rely on a mock-up "lite" stack for reconstructing these contexts/streams/sessions that does not fully match the target's stack and can thus be cheated

into mis-reconstructing the relevant streams.

The Bus Channel. By contrast, popular bus-facing kernel subsystems have so far received very little attention, despite being as crucial to a system's data exchange functionality as network stacks, and despite exhibiting a stacked structure that "routes" data to a multitude of potentially vulnerable kernel modules in the system just like a network stack routes its payloads to protocol handlers.

Several engineering factors make the bus-facing stacks, and especially device driver code, attractive to attackers. Firstly, the closer a driver is to the bus, the more likely it is that its code is based on a template provided by the hardware vendor in addition to the chipset or bus controller datasheet. These templates are treated as "sample code", and are typically written by hardware engineers with little knowledge of or anticipation for potential security issues.

Secondly, bus-facing device-related code is typically debugged against real devices, and the traffic/message patterns generated by these devices are implicitly assumed to be the only possible ones on the bus, i.e., at the receiving boundary of the software communicating with the bus. Thus, debugging driver code is biased towards eliminating accidental errors and device or host requests inappropriate for their receiving state. As a result of this bias, the community can overlook the more general problem of safely handling arbitrary binary input strings.

We observe that although a number of vulnerabilities have been found with USB fuzzing (see Sec. 8), and device drivers are commonly known to harbor clusters of vulnerabilities[1], we know of no systematic exploration of the USB-reachable interfaces on the same scale as network interface-facing software has been explored. We attribute this to the lack of tools for easily scripted USB fuzzing; a lack of well-known architectural models like the OSI model that would help identify classes of reachable targets may also play a role.

This Paper. Thus, the purpose of this paper is present the results of our effort to map out the bus-facing attack surface of a modern operating system, to demonstrate that effective and efficient injection of traffic into the buses is real and easily affordable, and to outline the architectural and computation-theoretic challenges that a defensive OS/driver architecture will face to bring about an improvement comparable to that of Netfilter for network stacks.

The collective experience of security practitioners is well encapsulated by the maxim "Security will not get better until tools for practical exploration of the attack surface are made available" (formulated by Joshua Wright). In this paper we provide both tools and models for practical exploration of the USB-facing attack surface. We apply the lessons learned in attack surface exploration of *networks* (such as importance of scanning, mapping of routed data flows, emphasis on well-engineered packet crafting, and availability of injection tools) to *buses*, and show their relevance.

[1]The *Month of Kernel Bugs* serves as a great example.

To summarize, we make these initial contributions:

- We present a top-level data flow analysis for the attack surface within the FreeBSD kernel reachable through its USB-facing subsystems
- We present a hardware tool and software framework for efficient crafting and injection of USB packets.

Section 2 provides a technical overview of the common USB perimeter-crossing bus. Section 3 compares networking to USB, as attack channels. Section 4 discusses how to reproduce the "scanning" step of network security in the context of USB. Section 5 presents our custom kernel probes to explore where decisions are taken for USB I/O data traversing the kernel. Section 6 presents some preliminary attack surface analysis using these tools. Section 7 presents *Facedancer*, our custom hardware board to probe and test this surface. Sections 8 and 9 review prior work and conclude.

2. OVERVIEW OF USB PROTOCOLS AND ABSTRACTIONS

This section presents an overview of the USB protocols and related kernel abstractions for the reader familiar with network security-related design issues but unfamiliar with the workings of USB. Our goal is to show that techniques which have become elementary for security analysis of network stacks are just as relevant for USB stacks. We also argue that, despite their differences, these stacks have considerable conceptual similarity in security challenges of safely handling untrusted inputs.

We realize that comparing USB protocols and layers with the OSI networking model entities may be criticized as "apples to oranges." We argue, however, that challenges of *safe input handling across layers of a layered system* apply to all system and components that accept such input, no matter what the transport. We counter the criticism with a programming maxim attributed to Kernighan and Pike: if one has found an instance of a bug, it's critical that one consider what other parts of the system may have the same bug.

2.1 USB layers and subsystems

USB uses differential signaling on a pair of wires, referred to as *D+* and *D-*; two others, *VBUS* and *GND*, provide power. USB peripheral controller chips, such as MAX3420, interpret signals into packets, and allow (host) processors or (embedded device) microcontrollers to send and receive packets by writing or reading the controller's registers over a simpler protocol, such as SPI. As with network protocols, an implementor of a higher USB layer can be entirely isolated from the details of physical signaling.

Unlike most OSI protocols, USB 1.x and 2.0 are strictly unidirectional: USB nodes are either *hosts* or *devices*; devices never speak until spoken to, and must be polled by the host (master) to send any data to the host. (This scheme changes for USB 3.0 due to performance limitations of polling; however, we defer USB 3.0 to follow-on work.) Due to the wide variation between computational powers of devices and therefore time needed to produce a response to a

host request, device-mode controllers implement automatic NAK-ing of host polling requests while a host command is being processed by the device; when its results are ready, an ACK is sent, followed by the appropriate response packets.

Hosts start their packets with the 1-byte *address* of the device (or 0 for "broadcasts") and the 1-byte *endpoint* on the device that the packet is intended for. All other devices are assumed to ignore packets for non-zero addresses that are not theirs; an address is assigned to an attached device, via a broadcast packet, after the host senses and queries the device for various *device descriptor structures*; a well-behaved device sets its address to the assigned one, and stops responding to broadcasts.

A USB packet is the basic unit of USB communication, their closest analogue in the OSI networking world being a TCP/IP packet *fragment*. USB packet sizes are typically subject to stronger physical limitations on size than in other media; there is also more variability between devices. As a result, protocols do not attempt to align their data structures with packet boundaries, but rather assemble them from a sequence of packets. The last packet in a sequence carries a flag to indicate the end of the sequence.

Multiple packets combine to form a single unit of transfer at the higher layer of abstraction: *transactions* and *transfers*. These abstractions are implemented by supporting data structures and code at the endpoints of a USB connection, and must be maintained in sync.

USB's unidirectional nature has a particularly important consequence: the context of any packet sent by a device is presumed to be set by a previous request from the host, and thus does not contain any information identifying this context. This is different from OSI networking, where packets, as a rule, carry data identifying the higher level abstraction (such as a TCP session's ports and acknowledgment numbers or UDP-based transaction sequence numbers). This identifying information affects the routing of packet payloads through the kernel to the appropriate consumer module; in USB, the routing information resides in the data structures of the host context and may never be sent on the wire.

For example, in FreeBSD, I/O operations originate in system calls, and are routed through its Common Access Method (CAM) layer, in the form of *CAM Control Blocks (CCBs)*. These CCBs are eventually translated by the `umass` module into USB transfer structs. CCBs do not need to cross the wire because *any* response arriving from the USB bus is expected to be the response to the outstanding I/O request. This response will be parsed and routed based on the *function code* contained in the CCB and the information previously supplied by the device in the various *device descriptor* structures it sent to the host when it was attached.

Despite these differences, the USB subsystem exhibits the same layered design as OSI stacks: packets are received by a peripheral controller, which hands them to the CPU (or device microcontroller) via an interrupt handler and/or a DMA transaction, and then the payload is passed through a series of callbacks registered by drivers and subsystems for

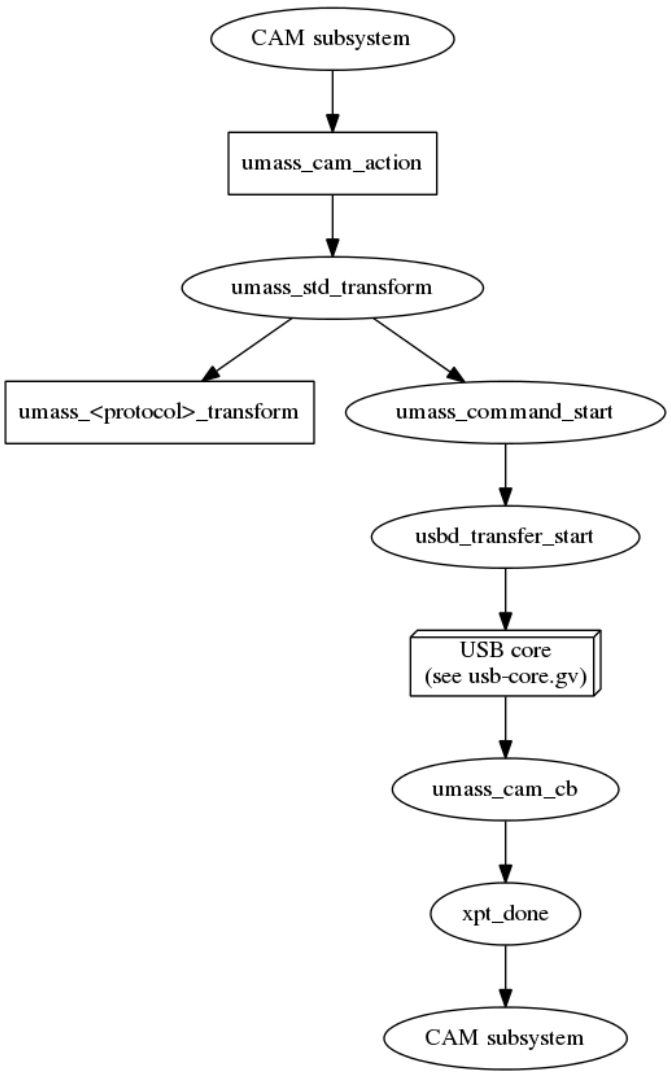

Figure 1: FreeBSD 9 USB Common Access Method (CAM) control flow.

further processing.

The callback mechanism design resembles those for registering protocol handlers in TCP/IP stacks, and is subject to similar assumptions on the part of the implementors, e.g., with respect to validity of data passed in to lower layers.

Figures 1 and 2 show the flow graph of the FreeBSD 9 USB stack functions involved in sending a request to a USB mass storage device and receiving its response. The FreeBSD stack unifies handling of USB mass storage device communications via the CAM layer, which represents headers of requests sent or pending in a transaction.

3. USB AND NETWORKING COMPARED

In any practical assessment of a network's ability to handle inputs safely, the physical abilities of the attacker to *capture* traffic, *replay* captured traffic, and *inject* crafted traffic

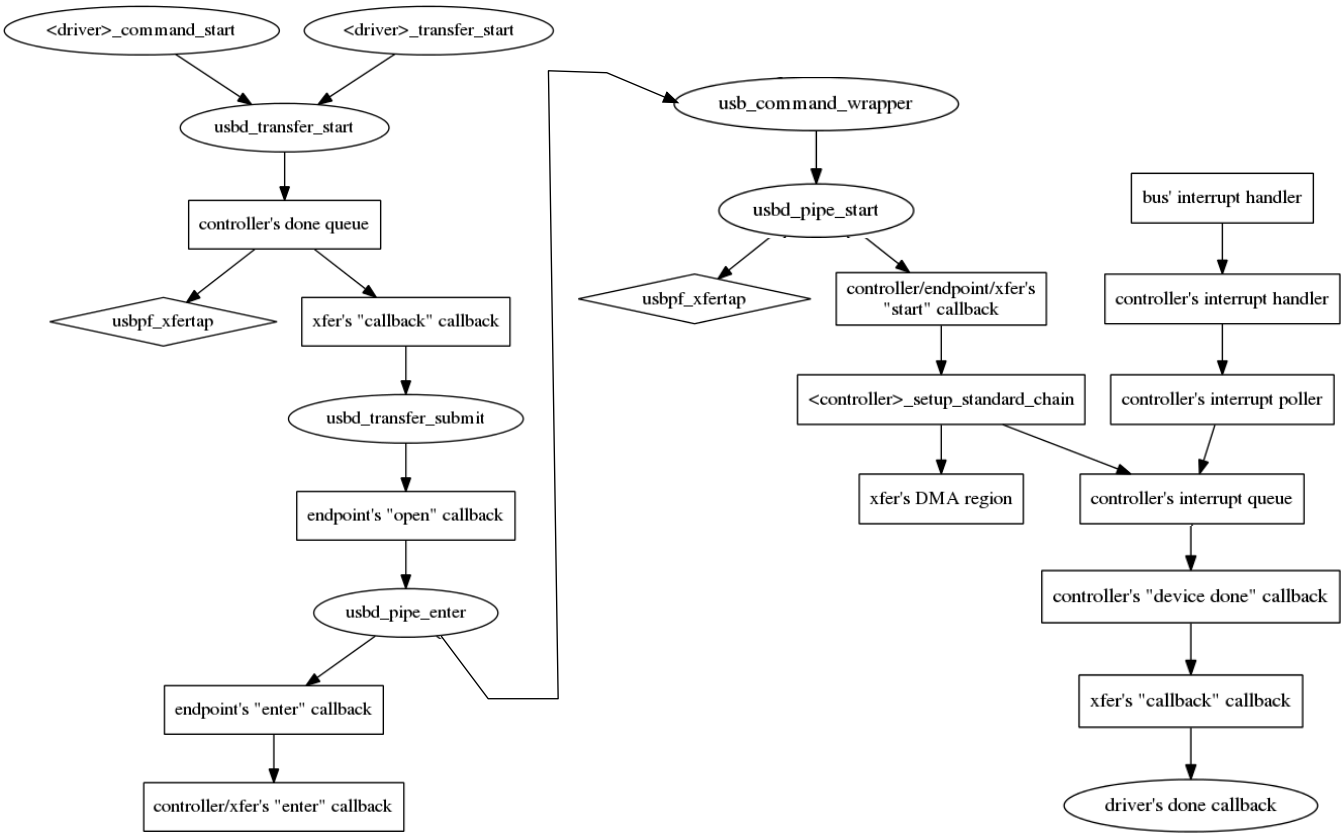

Figure 2: FreeBSD 9 USB Core control flow.

(with or without the ability to spoof its origin) are the fundamental building blocks of attack surface exploration (see, e.g., [12]). These capabilities can be regarded as primitive building blocks of attacks. Although the path from any of these capabilities to a full-fledged attack is rarely trivial, removing capabilities also eliminated entire sections of the possible attacks tree. By the same token, the presence of these capabilities and relative ease with which they can be leveraged determine the shape of the tree.

In Table 1 we draw the analogies between standard OSI networking concepts and those of USB, and illustrate how experience with assessing network environments for incorrect assumptions and thus potential vulnerabilities transfers directly to assessing USB for same. This table shows the non-obvious application of network analysis methods to USB.

For instance, a USB *transfer* is analogous to the request-response pattern widely used in the traditional networking model. The assumption made by implementors of higher-level protocols is that the device will respond to the transfer request with a relevant transfer response. If, however, an attacker is able to subvert the USB controller, she could hijack the request-response session and surreptitiously modify state on either the host or the device, with neither being any wiser.

In the same vein, a USB *packet* is equivalent in spirit to an IP *packet fragment* in that a single USB message (request or response) may be split into multple USB packets. Like IP fragments, all USB packets for a given transaction must be received before they can be reassembled and the transaction can be passed on to a higher processing layer. Also like IP fragments, the controller and higher layers often assume that the data actually received conforms to the parameters claimed in previous metadata. Violating this assumption could enable an attacker to extract or corrupt data in memory.

Table 1 highlights these and other similarities between networking abstractions and USB abstractions, including how an attacker could potentially take advantage of them, and how she could employ them.

4. "PORT SCANNING" FOR USB: IDENTIFYING VULNERABLE ENDPOINTS

While measuring the attack surface is most thoroughly performed locally using static analysis techniques or probing, a real-world attacker would need to perform her own scan of supported drivers in order to choose an exploit appropriate to the victim's operating system and drivers. In this section, we discuss methods for performing quick scans of this surface over the bus, roughly equivalent to what the Nmap tool does over a traditional network.

Determining the presence or absence of a driver is almost trivial, just as port scanning is trivial in TCP until perfor-

USB Abstraction	Ethernet Analogy	Assumption	Violation	Attacker Use
Transfer	One round-trip, potentially NAKed.	The intended device will reply to the transfer.	Non-compliant controller.	Hijack session, change state under the feet of the host.
Transaction	One set of transfers, all but the last NAKed.	The host controller complies with the USB specification on transactions.	Hijack session on disconnect.	Use of trusted session context.
Packet	Packet Fragment	Implicit length of concatenated frames will match explicit length of transaction.	non-compliant device	Memory corruption, info leak.
Controller	Ethernet Card	—	—	—
Bus	D+/D- Pair	Electrically legal signals, but in reality those widely outside of spec are accepted.	non-compliant controller	Damage frames for session hijack, jamming.

Table 1: Mapping between USB and networking concepts and attacks.

mance enhancements are added. Our hardware tool (Section 7) performs this by enumerating itself as a suspected VID and PID, then waiting to see whether the device sends any packets beyond enumeration.

As memory-corruption exploits are nearly always specific to a given operating system, it becomes necessary to fingerprint the target environment in order to know which vulnerability to exploit and which payload to use. A scanner has a number of options at its disposal to deduce the identity of an operating system and its revision. Network tools such as Nmap must rely upon tricky features such as the predictability of TCP sequence numbers and behavioral responses to TCP Explicit Congestion Notifications; in contrast, USB host fingerprinting is considerably easier, thanks to the multitude of accessible drivers presently available.

For example, Linux and BSD include support for each driver by every known VID:PID pair, while Windows and Mac drivers often support just those identifiers which match the vendor's hardware. Linux can be distinguished from BSD by support for rarer hardware, such as particularly poorly documented wireless cards. Windows can be distinguished from Mac OS X by the enumeration sequence.

Once the operating system is known, the version can be deduced by the same method, but this time looking for VID:PID pairs which have been recently added. A quick diff of any two kernel revisions will provide plenty of new and removed pairs within a single module, and the presence of one combined with the absence of another will uniquely identify the kernel revision.

5. INSTRUMENTATION AND MODES OF ANALYSIS

To explore the attack surface exposed via USB, we designed and implemented an instrumentation framework to probe the "routing" points of the FreeBSD USB stack, and tools for interpreting and filtering the traces (which is tricky, due to the asynchrony and interrupts involved). In other words, this is a beginning of a proper USB firewall, probing wherever decisions are taken for USB I/O data traversing the kernel.

To do this, we used the dtrace subsystem in FreeBSD, but found the standard probes insufficient and in some cases unreliable. Therefore, we implemented our own probes to monitor function entry and return events, basic block boundaries, and, most importantly, control flow decision points where multiplexing of payloads occurs. Table 2 describes the static probes we inserted; Figure 3 gives examples of the probe primitives we defined.

probe	qty	event
`FUNC_ENTER(f)`	204	upon function entry
`FUNC_RETURN(f)`	356	upon function exit
`BB_START(f, n)`	1235	upon starting basic block n in a given function
`BB_FINISH(f, n)`	1235	upon finishing basic block n
`MUX`	30	immediately prior to invocation of a callback

Table 2: Summary of static probes in our instrumentation framework.

```
char *foo(int bar, char *baz)
{
FUNC_ENTER(foo);
BB_START(foo, 0);
BB_FINISH(foo, 0);
    if(bar < 10) {
BB_START(foo, 1);
        printf("Error! bar is less than 10.\n");
BB_FINISH(foo, 1);
FUNC_RETURN(foo);
        return NULL;
    }
BB_START(foo, 2);
    toupper(*baz);
BB_FINISH(foo, 2);
FUNC_RETURN(foo);
    return baz;
}
```

Figure 3: Example of a function instrumented with our static probes.

Our rationale behind focusing on multiplexing points is that,

at these points, "routing" decisions occur with respect to payloads arriving on the bus. Knowing the provenance of data that affects these decisions and causes a particular branch to be taken allows us to construct custom payloads to take the path we desire for payload delivery. The data affecting these decisions can come in the packet or, most frequently, is derived from previous transactions with the device, such as responses to device descriptor requests. Since we control all phases of device communication, we can shape future routing decisions. Thus, our instrumentation gives us a systematic method of constructing and delivering payloads through the layers and subsystems of the USB-facing kernel code.

Paths less traveled. Our route tracing capability allows us to collect data on which paths are frequently traveled, which are invoked less frequently, and which do not seem to be taken at all. Using data provenance knowledge at mux points, we can cover the paths less traveled and therefore less debugged or assumed impossible by the developer. We note that exposing such developer assumptions is the bread and butter of vulnerability development, where a violated assumption—formally speaking, violation of a contract of a particular module or API—are known to frequently yield exploitable vulnerabilities (c.f. compare the use of contraint solvers for vulnerability finding [14, 8]).

Tom and Jerry Attacks. In Tom and Jerry cartoons, Jerry frequently entices Tom with a tasty treat, only to swap it out at the last second for unpalatable or dangerous objects, with Tom never expecting the change. In the same way, a USB host does not expect a USB device to change its state unless specifically requested. This can lead to vulnerable assumptions of impossibility of certain payloads that a well-behaving device would not send based on its supposed current state. Our instrumentation, combined with other tools currently in development, will help expose such assumptions and play them out using the Facedancer capability to emulate any device.

Note that this class of assumptions is broader than the classic TOCTOU because it need not involve any explicit checks, just the assumptions on the correct behavior of a device. As far as we know, such bait-and-switch attacks, when data structures being worked with by the kernel spontaneously change, have not been comprehensively studied for kernel I/O other than network communications—where extensive literature on protocol analysis exists. Again, we hope to fill that gap.

6. MEASURING THE EXPOSED USB SURFACE

Using our instrumentation, we gathered execution traces during operations with a USB thumbdrive on our FreeBSD test system. We want to sketch out the amount of code within the FreeBSD kernel reachable from the USB subsystem. These numbers are preliminary, as the tracing tool discussed above will deliver more accurate results on both code coverage and reachability in various scenarios. However, even top-level statistics are useful because rough estimates of bugs per line of this particular code is known. Thus, we could expect to estimate the size of the attack surface, even from the number of lines of count involved. Moreover, our reachability results with valid devices allow us to estimate code that is not commonly exercised and is therefore more likely to be buggy.

category	qty
drivers	63
vendors	667
products	1979

Table 3: Quantity of device drivers, supported vendors and products in FreeBSD USB stack.

For our case study of devices we use FreeBSD because we expect a smaller attack surface.

FreeBSD does not aim for broad device support and takes a conservative approach on including drivers they need. Popular Linux distributions, such as Ubuntu, take the opposite approach by including support for as many USB devices and device classes as possible. Note that support for USB devices in both systems comes in two flavors: standard classes of devices (e.g., HID, mass storage) are supported by drivers based on their class, whereas devices not belonging to standard classes are supported based on vendor and device id. FreeBSD uses an additional mechanism called "quirks" to diverge handling of standard class devices known to be exceptional with special case drivers or handlers. Quirks, in particular the 217 quirk definitions, affect control flow in drivers by setting a kernel global variable indicating the function to use for testing them, thus creating additional routes through the kernel that our Facedancer framework can leverage.

We note that Linux, the other natural free software choice for a case study, takes a much more aggressive approach to including device drivers. To wit, FreeBSD's USB implementation is 100k lines of code, whereas Linux 3.4.6 takes over 250k. Table 3 presents a summary of USB devices supported by FreeBSD; the drivers contain an average of 850 lines of code each. According to a 2011 study by Coverity [10], the Linux kernel exhibits 0.62 defects per thousand lines of code. As the same report notes, however, the industry average is 1.0 defects per KLOC, and we hypothesize that some driver code is closer to this mark.

A typical path of an I/O bus request is as follows: the filesystem creates a *CAM Control Block* (CCB) describing the request in a generic SCSI-like format and passes it to the *Common Access Method* (CAM) layer. The CAM layer searches the list of attached devices to find the physical device that can fulfill the request, and passes it on to a callback registered by the device when it was connected. In the case of a USB mass storage device, the request is routed to the `umass` driver, which then translates the CCB to the command protocol supported by the drive (e.g., ATAPI, SCSI, UFI). The `umass` driver wraps this translated command in a structure describing a USB transfer and hands it off to the controller, to be notified by hardware interrupt when it is complete. Figure 4 shows such a path traversal.

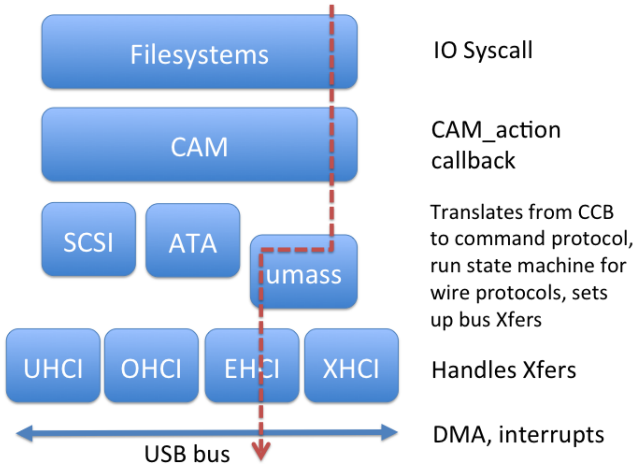

Figure 4: Path of a bus IO request through USB stack layers.

We explored this path in great detail using our instrumentation, and found a total of 30 points at which a routing decision was made. Additionally, we measured the length of this path at the granularity of both basic blocks and lines of code.

Table 4 compares the number of lines of code executed during our trace relative to the total number of lines of code in each module of the FreeBSD USB subsystem. The extensive amount of unexercised code as shown in this table provides a potentially fertile ground for attackers in search of vulnerabilities to exploit, as this code likely receives less attention from developers and testers.

Table 5 presents, for each function called within each module, how many basic blocks were executed against how many basic blocks exist. That is, the "total basic blocks" does not include functions that were not called during our tests. While these percentages are predictably higher than those for lines of code because so many functions are not reflected, a non-trivial number of unexercised basic blocks remain.

Figure 5 illustrates the fraction of lines of code executed within each function called during our testing. We find it interesting to note that the larger functions typically have a smaller percentage of their code exercised, whereas the smaller functions are usually exercised more fully.

7. HARDWARE-SUPPORTED USB INJECTION AND FORWARDING

Our software instrumentation described in Section 5 analyzed the execution paths inside the target of attack. To complement this framework, we built a custom hardware board and supporting software to generate any USB packet any attacking device might generate.

Using a standard PC as an attack tool was not feasible; because of the physical-layer properties of USB, timing is extremely important. Computers often provide host-only USB abilities, and other electronics (for example, cameras) typically act only as devices. In order to interact with computer as a USB device, an attacker must control the firmware on a product with a USB-capable chipset.

module	executed	total	percentage
ehci	552	3352	16%
umass	385	2681	14%
usb_busdma	278	949	29%
usb_compat_linux	14	1124	1%
usb_controller	45	431	10%
usb_dev	19	1618	1%
usb_device	734	2322	32%
usb_dynamic	3	93	3%
usb_hub	170	1789	10%
usb_lookup	11	198	6%
usb_msctest	35	679	5%
usb_parse	51	225	23%
usb_pf	15	319	5%
usb_process	44	298	15%
usb_quirk	7	727	1%
usb_request	380	1690	22%
usb_transfer	803	2559	31%
usb_util	45	207	22%

Table 4: Lines of code executed relative to total lines of code, per module.

7.1 Prior Work

Previously-published USB fuzzing work has used either virtualized hardware or standalone boards. While our tool isn't intended to replace either of these techniques in a researcher's toolbox, we believe it to be a better solution for use in security research.

Simulation tools, such as those added to VMWare and Qemu, can allow for pure-software USB devices to be attached to the host operating system. This technique has been used to to implement a working USB exploit [15], but it is less than desirable in two ways. First, the resulting exploit runs only in simulation; it must be manually ported to run in hardware. Second, the technique is restricted to operating systems which are amenable to running in a virtualized environment, such as the Playstation 3 bootloader exploit distributed as PSGroove [5].

Standalone products such as the Teensy [7] are intended as USB development boards and onto which attacker-controlled firmware can be loaded to fuzz or exploit a host. The Teensy is connected to the programming laptop, reset, and then flashed with the firmware. It is then unplugged and connected instead to the victim host, where it is enumerated and runs based on the on-board firmware. The time required for this round trip is acceptable for developing USB devices, but it is unacceptably long for fuzz-testing and system exploration.

7.2 Our Solution: Facedancer

Here we present the Facedancer board, which provides a more rapid and flexible approach to prototyping USB devices, specifically with the purpose of fuzzing USB hosts. The Facedancer, shown in Figure 6, consists of the following major components, logically connected in the following

module	executed	total	percentage
ehci	109	227	48%
umass	74	151	49%
usb_busdma	81	107	76%
usb_compat_linux	2	3	67%
usb_controller	5	15	33%
usb_dev	4	6	67%
usb_device	130	205	63%
usb_dynamic	1	2	50%
usb_hub	33	45	73%
usb_lookup	5	8	62%
usb_msctest	11	24	46%
usb_parse	13	21	62%
usb_pf	2	11	18%
usb_process	7	12	58%
usb_quirk	5	10	50%
usb_request	73	127	57%
usb_transfer	159	244	65%
usb_util	12	17	71%

Table 5: Basic blocks executed relative to total number of basic blocks in functions used during a USB mass storage device session, per module.

order:

- USB mini-B port for connecting to a USB A port on the attacker's host machine.
- USB-to-serial (FTDI) chip which translates the host's USB to serial.
- MSP430 microcontroller running firmware from the GoodFET [1] project, including our `maxusb` application.
- USB Peripheral Controller IC with an SPI interface (MAX3420).
- USB mini-B port for connecting to a USB A port on the victim machine (system-under-test).

Our `maxusb` application firmware is purposefully kept very basic, merely passing commands between the host and the MAX3420 chip. This chip provides the digital logic and analog circuitry necessary to implement a full-speed USB 2.0-compliant peripheral. All advanced logic and the device emulator itself are written entirely in host-side Python which provides rapid-development, excellent real-time logging, and eliminates the need to re-flash the firmware on the interface device (i.e., the Facedancer).

Our device emulators themselves are written as extensions of the GoodFETMaxUSB class, exposing nearly all of the underlying functionality of the chip. We believe that our use of very few abstractions in the calling convention will help researchers to discover new tricks on the bus, rather than complying with the abstraction layers imposed by a cleaner stack.

Facedancer is a forwarder, router, and rewriter. It can be fully standalone, but it is easier to drive from another laptop as it is not self-powered and programming custom payloads to inject is easier in Python,

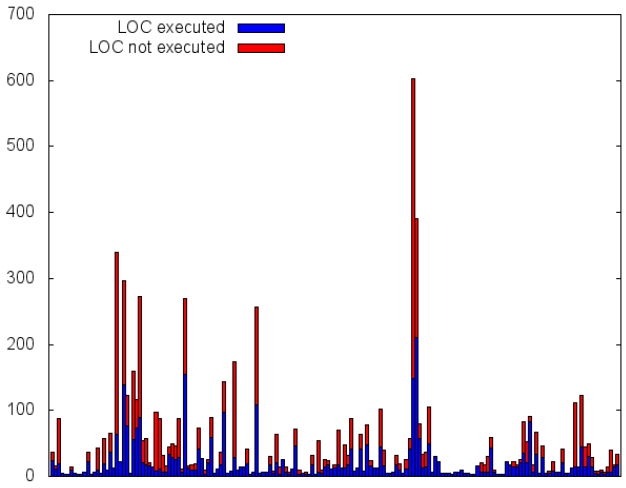

Figure 5: For each function, the number of lines of code executed while processing an ordinary I/O request (blue) overlayed on the total number of lines of code (red). Each point on the x-axis represents a distinct function within FreeBSD's USB stack.

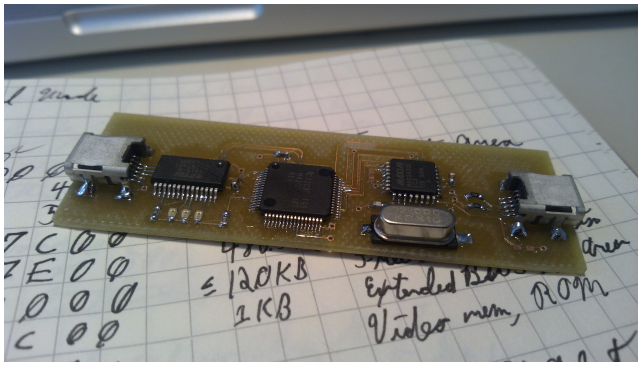

Figure 6: Facedancer, a USB device emulator.

8. RELATED WORK

The USB attack surface has been previously explored and exploited in papers and conference presentations, and in fact has been compared to TCP [9].

8.1 Attacks Against USB Hosts

Jodeit and Johns [13] implemented a software mutation-based man-in-the-middle USB fuzzer using the QEMU machine emulator and virtualizer. Packet modification was performed via Python user-space binding. They presented results of basic tests which produced four Windows XP kernel crashes and one application crash.

Davis [11] used a $1,400 hardware USB analyzer and the Python SendKeys library, in combination with the Frisbee fuzzing library, to fuzz USB on commercial operating systems. Although a pieced-together solution, the fuzzing uncovered bugs including multiple HID-class memory corruptions in Windows 7, a hub-class kernel stack overflow in Solaris 11 Express, a printer-class kernel stack overflow in Solaris 11 Express, multiple image-class integer overflows in

the Microsoft Xbox 360, and an interface descriptor memory corruption in OS X.

CVE-2009-4067 [15] describes a buffer-overflow vulnerability in a device driver, designed to allow the Auerswald PBX/System telephone to communicate with Linux systems via USB. The vulnerability arose due to improper processing of String Descriptors, resulting in exploitable stack overflows in the kernel. This proof-of-concept was conducted via QEMU virtualization.

8.2 Attacks Against USB Devices

Although compromising USB hosts may appear to provide the best value for an attacker and thus the natural focus of defensive efforts, exploiting USB devices enables truly automated "sneaker net" propagation attacks. In such an attack, a compromised device passes both the payload and a (multi-)device infector to the host, the host in turn infects devices it encounters, which pass the infection to other hosts, and so on. Thus device exploitation must not be ignored, even though a device's firmware would seem to be a poor value-for-effort achievement; indeed, it can enable autonomous "sneaker net" worms.

The following publicly-demonstrated exploits against kernel/bootloaders on USB devices are known to date:

PSGroove: This famous exploit used a crafted USB hub-class configuration descriptor take to advantage of a heap overflow and thus "jailbreak" the Sony Playstation 3[2]. Prior to delivering the payload, the exploit manipulates the heap by "plugging" and "unplugging" fake USB devices with large device descriptors until the device on port 4 misreports its size, which allows the payload to overwrite one of `malloc`'s boundary tags. Related exploit code is publicly available [5].

iPod Touch exploit: Fuzzing all possible USB control messages of the iPod touch 2G's DFU mode caused a reboot. Further investigation revealed a heap overflow tickled only when lengths of a specific USB control message exceed 0x100 bytes [3].

iPhone and iPod Touch: A null pointer dereference exploit was found by fuzzing from the host, and uses the `"0x21,2"` USB control message[4].

9. CONCLUSIONS

Security researchers and exploit developers have invested a great deal of effort in exploring the attack surfaces and data flows in code receiving input from network interfaces, such as TCP/IP stacks. As a result of this effort we have powerful and flexible defensive architectures such as Netfilter, which provide the defender with the capabilit and necessary context to filter traffic flow throughout the kernel. We show that similar analysis is necessary and in fact critical for code that receives input data over buses, such as USB. Such code is just as complexly layered as a network stack and handles more complex structures; drivers are a notorious breeding ground for bugs copy-pasted from boilerplate sample code. Yet a broad view of data flows and attack surface for bus-facing code is still missing.

Our paper maps out the bus-facing attack surface of a modern OS, and demonstrates that effective and efficient injection of arbitrary traffic into the buses is real and easily affordable, and we present a simple and inexpensive hardware tool for the job. Our hardware and software tools together will help researchers fuzz intelligently, so that fuzzed payloads don't fail before the code block/function where they need to get through kernel routing. The time for assuming only something special, expensive, and rare can so inject into USB are over; we provide a tool that makes this exploration within reach of anyone with Python.

[2]See `http://ps3wiki.lan.st/index.php?title=PSJailbreak_Exploit_Reverse_Engineering` for more information.

[3]`http://theiphonewiki.com/wiki/index.php?title=Usb_control_msg(0xA1,_1)_Exploit`

[4]`http://theiphonewiki.com/wiki/index.php?title=Usb_control_msg(0x21,_2)_Exploit`

10. ACKNOWLEDGMENTS

This research was supported by the Department of Energy under Award No. DE-OE0000097. The views and opinions in this paper are those of the authors and do not necessarily reflect those of the United States Government or any agency thereof.

The authors would like to thank the REcon reverse engineering conference and its organizer, Hugo Fortier, for providing a unique and valuable forum for discussion of the ideas presented in this paper. Additionally, Brandon Wilson's examples of USB device emulation on Texas Instruments graphing calculators provided valuable inspiration in developing the Facedancer board.

11. REFERENCES

[1] GoodFET. `http://goodfet.sourceforget.net`.
[2] libdnet. `http://libdnet.sourceforget.net`.
[3] libnet. `http://libnet.sourceforget.net`.
[4] lorcon. `http://802.11ninja.net/`.
[5] PSGroove. `https://github.com/psgroove/psgroove`.
[6] scapy. `http://www.secdev.org/projects/scapy/`.
[7] Teensy. `http://www.pjrc.com/teensy`.
[8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *NDSS*, 2011.
[9] D. Barrall and D. Dewey. "plug and root," the USB key to the kingdom. In *Black Hat Briefings*, 2005.
[10] Coverity. 2011 Open Source Integrity Report. `http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf`, 2011.
[11] A. Davis. USB - undermining security barriers. *Black Hat Briefings*, August 2011.
[12] T. Goodspeed, S. Bratus, R. Melgares, R. Speers, and S. W. Smith. Api-do: Tools for exploring the wireless attack surface in smart meters. In *45th Hawaii International Conference on System Science (HICSS)*, pages 2133–2140, 2012.
[13] M. Jodeit and M. Johns. Usb device drivers: A stepping stone into your kernel. In *2010 European Conference on Computer Network Defense* , 2010.

[14] J. Vanegue, S. Heelan, and R. Rolles. SMT Solvers in Software Security. In *6th USENIX Workshop of Offensive Technologies*. USENIX, July 2012.

[15] R. D. Vega. Linux USB device driver - buffer overflow. *MWRI Security Advisory*, October 2009.