How USB Does (and Doesn't) Work

A Security Perspective

PETER C. JOHNSON

Peter C. Johnson received his bachelor's degree from the University of California, San Diego and worked for a couple of computer systems companies in the Bay Area before escaping back to academia. He is currently polishing up his PhD dissertation at Dartmouth (not coincidentally related to security of USB stacks) and will begin work as a visiting assistant professor of computer science at Middlebury College in fall 2014. pete@cs.dartmouth.edu

USB devices are easy to take for granted: They're innocuous by their nature (who's afraid of a keyboard?) and by their ubiquity. However, the architecture of the Universal Serial Bus ecosystem is surprisingly complex and deeply embedded in modern operating systems. Furthermore, having risen to awareness on the backs of traditionally "dumb" devices like keyboards and mice, the features of the USB protocol that very much resemble wide-area networking protocols can be underappreciated. This combination of complexity, embeddedness, and underappreciation is the unholy trinity of security "features." In the following paragraphs, I hope to sprinkle some holy water on this situation, so come along while I first share some fire and brimstone, then give reason for hope. To the Batmobile!

"It can't be that bad," you're saying to yourself, "a USB attack requires physical access." While absolutely true, this misses a crucial technicality: An attack over USB must indeed be delivered physically, *but the attacker herself need not be physically present.* How many people, upon finding a USB thumb drive lying on the ground, are able to resist the temptation to plug it into the first computer they find? Sufficient anecdotal evidence exists to suggest the number is "few enough for us to worry" (though I wish you the best of luck in getting IRB approval to verify this experimentally). I don't mean to imply that the physical nature of USB is impotent as a defense, but that it is not dependable.

Speaking of Stuxnet, once the USB drive prepared by [REDACTED] was plugged into a machine beyond the defensive air gap, the "vulnerability" it initially exploited was that Windows was configured to execute autorun.inf on any inserted devices. The realization that such critical systems were thus (mis)configured no doubt makes the sysadmin- and security-minded out there a bit light-headed, and the same people might be tempted to breathe a sigh of relief that the initial infection vector could be so easily shut off. Completely setting aside the raft of zero-day exploits also employed by Stuxnet, indulging in the aforementioned sigh of relief is a bit premature.

How Bad Is It Really?

In March 2013, Microsoft patched three similar vulnerabilities (CVE-2013-1285, CVE-2013-1286, CVE-2013-1287) in all extant versions of Windows that allowed "escalation of privilege" [4]. NIST's National Vulnerability Database puts it a bit more starkly [7–9]:

- Access Complexity: Low
- Authentication Required: None
- Confidentiality Impact: Complete
- Integrity Impact: Complete
- Availability Impact: Complete

These were *not* system configuration issues, like failing to disable execution of autorun.inf; these were bugs in the kernel's USB stack, ring-0 code that is run automatically *every* time a USB device is plugged in to the system. Running such code with such privileges is a

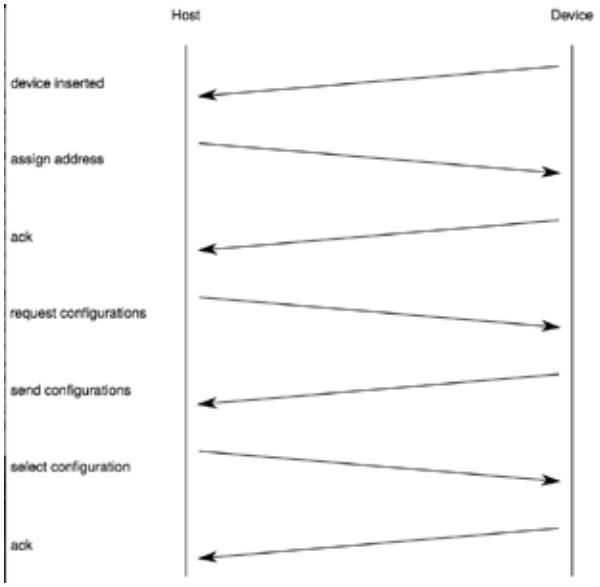

Figure 1: USB bus enumeration process as a ladder diagram

somewhat natural consequence of the "U" in USB: To support a broad array of devices, the kernel must get the device to identify itself so that the kernel can load the appropriate driver. This process is called "bus enumeration" (because the host is taking roll of devices on the bus) and looks something like this:

Kernel: Stop! What is your name?

Device: It is Arthur, King of the Britons.

Kernel: What is your quest?

Device: To seek the Holy Grail.

Kernel: What is the airspeed of an unladen swallow?

The device's response at this point is key. If the answer is "I don't know," the device finds itself tossed from the Bridge of Death, never to return; if the answer is "What do you mean? An African or European swallow?" the kernel loses its mind and the Bridge of Death is no longer guarded. This example is surprisingly illustrative and not just the injection of a predictable computer nerd trope: A device can respond according to the USB protocol with an identification the kernel accepts, it can respond according to the protocol with an identification the kernel rejects ("I don't know"), or it can deviate from the protocol entirely ("African or European?").

If you squint only a bit, the bus enumeration process caricatured in Figure 1 bears more than a passing resemblance to the three-step TCP handshake or the request-response nature of an SMTP transaction. Figure 1 shows the enumeration process in the form of the ladder diagram we all know and love from the networking world. Indeed, the USB protocol sports a great number of features reminiscent of traditional network protocols: addressing, packetized data, sequence numbers, acknowledgments, and so on. (I'll return to this comparison later on, I promise.)

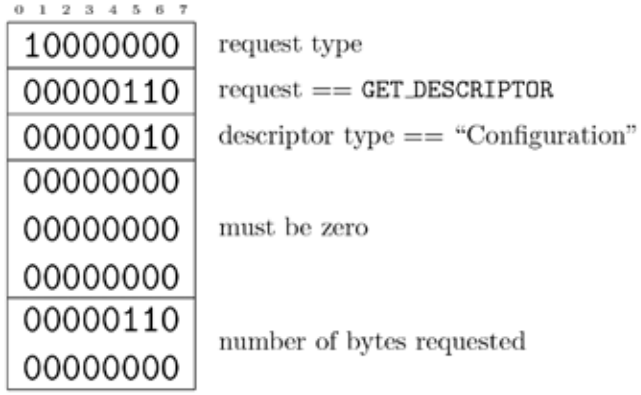

Figure 2: Bit-level description of message sent during bus enumeration from host to device requesting configuration descriptor

The bugs Microsoft patched in 2013 were failures to correctly handle protocol deviations that allowed complete system compromise. Unfortunately, precise details on the patched vulnerabilities are difficult to come by, though we have good reason to believe the problems arose when parsing descriptors during enumeration. Parsing is one of those oft-underappreciated aspects of protocol implementation that should be relatively straightforward to get right, yet can lead to rather catastrophic failures. In the case of bus enumeration, the complexity of the messages involved can't have helped. Figures 2 and 3 show a couple of packets sent during enumeration, specifically the host-to-device message that requests a configuration and the device-to-host response. (The specific semantics aren't important, so don't worry if "configurations," "interfaces," and "endpoints" mean nothing to you.)

The configuration request shown in Figure 2 is fairly simple, but the response (Figure 3) is anything but. After the standard

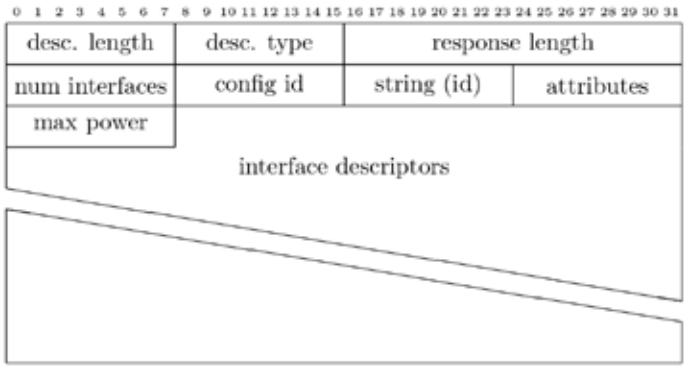

Figure 3: Bit-level description of message sent during bus enumeration from device to host containing configuration descriptors

header stuff (the details of which—barring the length fields—are secondary for this discussion), the device sends the descriptors of all interfaces contained within that particular configuration. Again, what a USB interface is doesn't really matter right now; the important point is that it's another message format that needs to be parsed. Not just that, but interfaces contain endpoints, the descriptors for which are also embedded in this single response. The result is a (relative) ton of data with all manner of length fields littered throughout, whose correct interpretation is vital to the correct interpretation of the message as a whole, and which are mutually dependent—that is, if the length field of one descriptor is messed up, the rest of the parse necessarily goes off the deep end.

This complexity makes implementing both host- and device-side logic dealing with descriptors a delicate task, but it gets better!

The Device *Is* the Application

Let's now move up the stack a bit and look at the application layer. In the world of networking, applications are, practically speaking, presented with their choice of either streams (TCP) or datagrams (UDP). Beyond that, they're more or less on their own, although standards like SMTP and DNS have been created to enforce some consistency (and, hopefully, quality). Proprietary protocols, on the other hand, are a completely different story: Vendors can and do design and implement protocols however they darn well please, which frequently results in the less-good kind of media attention. (Diebold, anyone?)

Fortunately for us innocent bystanders, some of the local effects of poorly designed or implemented application protocols can be mitigated by running server processes as an unprivileged user or in a chroot jail. Ideally, then, if a vulnerability in a protocol design or implementation is discovered, only resources owned by the unprivileged user or those within the chroot jail are susceptible to compromise. Other methods, including virtual machines and Linux containers, provide isolation sufficient to protect against whole-system compromise, although it isn't immediately clear how any of these map to the USB realm.

The USB protocol allows similar encapsulation (indeed, the USB Mass Storage Specification calls for stuffing raw SCSI commands inside USB packets much like iSCSI stuffs them inside IP packets). This freedom brings with it the same double-edged sword as in the networking world: Although developers can define their own protocols to create exciting new applications, they also run the risk of introducing vulnerabilities as they increase systems' attack surfaces. But wait! USB doesn't deal with applications, it deals with *devices*!

The implications of this are numerous and not altogether encouraging. First, it means that, once shipped, devices are often stuck with a specific version of a protocol implementation—one that might be buggy (i.e., vulnerable) and difficult to upgrade. Second, the "server" implementation of the protocol frequently exists in the device driver—*which usually runs as kernel code*—so if the protocol is vulnerable, an exploit necessarily results in total system compromise. Third, although standards such as USB Mass Storage and USB Human Interface Device exist to bring some sanity to the land, many devices ship binary drivers. That's right: Devices can ship black-box code, implementing black-box protocols, that implicitly runs inside the kernel's address space.

That's okay, at least USB doesn't let the device pick which driver to talk to—nothing like inetd for networking services. Hmm? What's that you say? I already described how a device identifies itself to the kernel? And there's nothing to stop a device from identifying itself as a device with a known-vulnerable driver? And the kernel will happily load said driver and let the device talk to it, no questions asked? Well, that's potentially worrisome.

Unfortunately, it's true: In addition to the potentially unreliable nature of USB device driver protocols and implementations, a newly plugged device is in charge of choosing precisely which device driver to communicate with. To make matters worse, modern operating systems ship with support for a huge number of devices, many drivers for which haven't seen maintenance in years. To think there aren't exploitable vulnerabilities lurking among that crufty code would be naïve.

Okay, I'm Scared. Help?

In the preceding paragraphs, I've painted a pretty bleak picture. The (sort of) good news is that we don't know of any vulnerabilities in extant USB stacks. Of course, that doesn't mean there aren't any, nor does it mean that *other* people don't know about them or, if they do, that they aren't actively exploiting them. I said at the beginning that I'd give reason for hope, and here's where that comes in. I also said I'd return to the similarities between the USB architecture and the networking systems we all know and love. Two birds, one stone.

It's true that USB is an underappreciated attack vector; in contrast, networks are not. Because the two are so similar, we can take advantage of decades of tools, techniques, research, and lore in defending networks and apply it to the task of defending USB.

First and foremost, we know there's a problem and, as G.I. Joe would say, "Knowing is half the battle." My colleagues and I at Dartmouth have published work [3] that explores the attack surface presented by FreeBSD's USB stack, and the tools to mount such an attack. Andy Davis wrote an extensive whitepaper on USB driver vulnerabilities in 2013 [5]. As I mentioned earlier, Microsoft found a vulnerability, fixed it, and shipped the fix in its monthly Patch Tuesday event as opposed to waiting for a larger Service Pack update, evidence that Microsoft is convinced this area is worth defending as well.

Figure 4: The Facedancer board

Additionally, beyond the venerable microkernel model, a number of research projects have explored techniques to isolate device drivers in the name of system stability [2, 6, 10]. Microsoft has also started moving USB drivers to userspace. Though these measures won't eliminate vulnerabilities, they will help contain side effects of potentially buggy drivers.

We've also developed tools to help find vulnerabilities in USB implementations. Travis Goodspeed created the open source Facedancer (Figure 4) board [1] to facilitate exploration of both host- and device-side USB stacks, and I wrote the Python-based software stack to drive it.

The Facedancer board hosts an MSP430 microcontroller connected via SPI to a MAX3420 or MAX3421 USB controller. When connected to both a host machine and a target machine, the host can run Python code that causes the Facedancer to appear to the target as any USB device it wants. The Python framework handles as much or as little of the device enumeration process as you want it to: It allows you to write *in software* any USB device you can imagine, well-behaved or not. The latter is key: We want to emulate USB devices that deliberately misbehave so that we can probe the robustness of existing USB stacks that are not suitable for static analysis (either because they are too complex or because they are closed source).

We currently have code that emulates a USB keyboard, a USB thumb drive, and a USB FTDI serial connection. All of these have been successfully tested against real operating systems' USB stacks. The next step is to modify them to misbehave and see how the operating systems respond. If you're interested in playing around with a Facedancer but you'd prefer not to dig out your soldering iron, you can buy pre-assembled (and pre-flashed!) boards from http://int3.cc.

Mr. Samwise Gamgee holds that "it's the job that's never started as takes longest to finish." We've started. It is my hope that this article raises awareness among the operating system community that there may be exploitable vulnerabilities in this area of the code, and thus spur efforts to address them soon.

References

[1] GoodFET: http://goodfet.sourceforge.net.

[2] Silas Boyd-Wickizer and Nickolai Zeldovich, "Tolerating Malicious Device Drivers in Linux," *Proceedings of the USENIX Annual Technical Conference*, 2010.

[3] Sergey Bratus, Travis Goodspeed, Peter C. Johnson, Sean W. Smith, and Ryan Speers, "Perimeter-Crossing Buses: A New Attack Surface for Embedded Systems," *Proceedings of the 7th Workshop on Embedded Systems Security (WESS 2012),* 2012.

[4] Microsoft Corporation, "Vulnerabilities in Kernel-Mode Drivers Could Allow Elevation of Privilege," Microsoft Security Bulletin MS13-027: https://technet.microsoft.com/library/security/ms13-027, 2013.

[5] Andy Davis, "Lessons Learned from 50 Bugs: Common USB Driver Vulnerabilities," technical report, NCC Group, 2013.

[6] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, "Fault Isolation for Device Drivers," *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09),* 2009.

[7] NIST, Vulnerability Summary for CVE-2013-1285: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1285, 2013.

[8] NIST, Vulnerability Summary for CVE-2013-1286: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1286, 2013.

[9] NIST, Vulnerability Summary for CVE-2013-1287: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1287, 2013.

[10] Michael M. Swift, Brian N. Bershad, and Henry M. Levy, "Improving the Reliability of Commodity Operating Systems," *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP '03),* 2003.